CORSO DI LAUREA IN INGEGNERIA INFORMATICA

FONDAMENTI DI INFORMATICA II: RACCOLTA DI ESERCIZI

GEPPINO PUCCI

A.A. 2001/2002

Chapter 1

Formal Specification of Computational Problems

Problem

A *computational problem* Π is a *relation* between a set $\mathcal{I}$ (the set of *instances*) and a set $\mathcal{S}$ (the set of *solutions*). Algebraically, we have

$$\Pi \subseteq \mathcal{I} \times \mathcal{S}.$$

As an additional constraint, we require that for any instance $i \in \mathcal{I}$ there is *at least* one solution $s \in \mathcal{S}$ such that $(i, s) \in \Pi$. We say that s is a *solution to instance* i of problem Π.

Note: For a single $i \in \mathcal{I}$ there could be two *distinct* solutions $s_1, s_2 \in \mathcal{S}$ such that $(i, s_1) \in \Pi$ and $(i, s_2) \in \Pi$. In general, there can be several solutions to the same problem instance.

Examples

Integer Sum Let Z denote the set of all integers. Then

$$\begin{aligned}
\mathcal{I} &= Z \times Z; \\
\mathcal{S} &= Z; \\
\Pi &\subseteq \mathcal{I} \times \mathcal{S} = (Z \times Z) \times Z \\
&= \{((a, b), c) : a, b, c \in Z \text{ and } c = a + b\}.
\end{aligned}$$

Graph Reachability A *directed graph* is a pair $G = (V, E)$, with $V \subseteq Z^+$ and $E \subseteq V \times V$. V is the set of *nodes* of G, while E is the set of *edges*. A *path* in G is a sequence $\pi = \langle v_1, v_2, \ldots, v_k \rangle$, $k \geq 1$, with $v_i \in V$, $1 \leq i \leq k$ and $(v_j, v_{j+1}) \in E$, $1 \leq j < k$. Our problem can be defined as follows:

$$\begin{aligned}
\mathcal{I} &= \Big\{\langle G = (V, E), u, v\rangle : V \subseteq Z^+, E \subseteq V \times V \text{ and } u, v \in V\Big\}\,; \\
\mathcal{S} &= \Big\{\langle v_1, v_2, \ldots, v_k\rangle : k \geq 1, v_i \in Z^+, 1 \leq i \leq k\Big\} \cup \{\epsilon\}; \\
\Pi &= \{(\langle G, u, v\rangle, \pi) : \pi = \langle v_1, v_2, \ldots, v_k\rangle \text{ is a path in } G \text{ with } v_1 = u \text{ and } v_k = v\} \\
&\quad \cup \{(\langle G, u, v\rangle, \epsilon) : \text{ there is no path } \pi \text{ in } G \text{ from } u \text{ to } v\}\,.
\end{aligned}$$

Size of a problem instance

The *size* of an instance is a *reasonable* measure of "how large" the instance is. The concept of size can not be made completely formal and depends on the particular problem being studied. For example, for *Integer Sum*, we can use the number of bits of the binary representation of the two integers as the size of the instance; for *Graph Reachability*, the most natural measure for the size of an instance is $|V| + |E|$; for *Sorting* it is natural to take the number of items to be sorted as the size of the instance.

Algorithm

An *algorithm* is a well-defined, *deterministic* computational procedure that transforms a given *input* into a *unique output* through a finite sequence of *basic steps*. Therefore, an algorithm computes a *function* whose domain is the set of inputs and whose values are the ouputs. An algorithm can be specified once we agree on a *computational model*, that is, an abstraction of a computing device characterized by a rigorously defined set of basic steps. A popular model of computation is the *Random Access Machine* (RAM), an abstraction of a traditional, sequential computer and its instruction set.

Each basic step of the computational model can be associated with a *cost*. The *running time* of an algorithm on a particular input is the global cost of the basic steps executed by the algorithm on that input. To ease the analysis of the running time of an algorithm for a particular problem, it is customary to identify a subset of "crucial" basic steps, which are given unit cost, while the remaining basic steps are neglected by assigning them zero cost. Particular care must be exercised in selecting the "crucial" steps, in particular, selection must encompass all those steps whose number is roughly equal to the total number of steps executed. When in doubt, it is advisable to assign unit cost to *all* steps. As an example, the running time of a sorting algorithm is often evaluated by assigning unit cost uniquely to comparison steps.

We say that an algorithm $\mathcal{A}$ *solves* a problem $\Pi \subseteq \mathcal{I} \times \mathcal{S}$ if and only if $\mathcal{A}$ computes a function $f_{\mathcal{A}}$ satisfying the following properties:

(a) $domain(f_{\mathcal{A}}) \supseteq \mathcal{I}$;

(b) $\forall i \in \mathcal{I} : (i, f_{\mathcal{A}}(i)) \in \Pi$.

Note: An algorithm associates a *single* solution to any problem instance, even when multiple solutions exist.

Example

Consider the following "toy" problem:

$$\begin{aligned} \mathcal{I} &= \{1, 2, 3\}; \\ \mathcal{S} &= \{a, b, c, d\}; \\ \Pi &= \{(1, a), (1, b), (2, c), (3, d)\}. \end{aligned}$$

The following is an algorithm for Π.

```
$\mathcal{A}_\Pi(x)$
  if $x = 1$ then return $a$
  if $x = 2$ then return $c$
  if $x = 3$ then return $d$
              else return $f$
```

Algorithm $\mathcal{A}_\Pi$ satisfies both Properties (a) and (b), therefore $\mathcal{A}_\Pi$ solves Π. Note that $\mathcal{A}_\Pi$ does something more, since it returns f for any value of x different from 1, 2, or 3. Therefore $\mathcal{A}_\Pi$ also solves $\Pi' = \mathcal{I}' \times \mathcal{S}'$, with $\mathcal{I}' = \{i : i \geq 4\}$ and $\mathcal{S}' = \{f\}$. This shows that a single algorithm may solve more than one problem. In contrast, there may exist many algorithms solving the same problem.

Exercise 1.1 We say that two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ are *functionally distinct* if the functions $f_{\mathcal{A}_1}$ and $f_{\mathcal{A}_2}$, respectively computed by the two algorithms, differ on at least one input $x \in \mathcal{I}$.

(a) How many functionally distinct algorithms may exist for the *Integer Sum* problem seen in class?

(b) How many for the *Graph Reachability* problem seen in class?

Please, justify your answers.

Answer:

(a) No two functionally distinct algorithms may exist for *Integer Sum*, since there is a unique solution for any instance.

(b) For *Graph Reachability*, there are infinitely many mutually distinct algorithms, since there are infinitely many instances that admit more than one solution.

$\square$

Exercise 1.2 Let U be a finite set. Given two arbitrary subsets of U, $A, B \subseteq U$, consider the problem of returning an element $u \in A \cap B$, if $A \cap B \neq \emptyset$, or returning ϵ if $A \cap B = \emptyset$. Cast this as a *computational problem* by specifying

(a) the set of instances $\mathcal{I}$;

(b) the set of solutions $\mathcal{S}$;

(c) the appropriate relation Π.

Answer:

(a) Let $\mathcal{F}(U)$ be the family of all subsets of U. Then $\mathcal{I} = \mathcal{F}(U) \times \mathcal{F}(U)$.

(b) $\mathcal{S} = U \cup \{\epsilon\}$ (note that we are assuming that $\epsilon \notin U$.)

(c) Given $(A, B) \in \mathcal{I}$ and $y \in \mathcal{S}$ we have:

$$(A, B)\ \Pi\ y \Longleftrightarrow (y \in A \cap B) \text{ or } [(A \cap B = \emptyset) \text{ and } (y = \epsilon)]\,.$$

$\square$

Exercise 1.3 Intuitively, the merging problem consists in combining two sorted sequences $(x_1, x_2, \ldots, x_m)$ and $(x_{m+1}, x_{m+2}, \ldots, x_n)$ into one sorted sequence $(y_1, y_2, \ldots, y_n)$.

(a) Formally specify the sets $\mathcal{I}$ and $\mathcal{S}$ and the relation $\Pi \subseteq \mathcal{I} \times \mathcal{S}$ for the merging problem.

(b) Give a reasonable measure for the size of an instance $i \in \mathcal{I}$.

Answer:

(a) Let U be a totally ordered universe set, and let SS be the set of *Sorted Sequences* of elements of U, i.e.

$$SS = \{(a_1, a_2, \dots, a_k) \in U^k : a_1 \leq a_2 \leq \dots \leq a_k, k \in Z^+\}.$$

Then, $\mathcal{I} = SS \times SS$ and $\mathcal{S} = SS$. Problem $\Pi \in \mathcal{I} \times \mathcal{S}$ is specified as:

$$\left(\left((x_1, x_2, \dots, x_m), (x_{m+1}, x_{m+2}, \dots, x_n)\right), (y_1, y_2, \dots, y_n)\right) \in \Pi$$

iff the two *multisets* (i.e., sets with possibly repeated elements)

$$\{x_1, x_2, \dots, x_m, x_{m+1}, x_{m+2}, \dots, x_n\} \text{ and } \{y_1, y_2, \dots, y_n\}$$

are equal.

(b) Given $((x_1, x_2, \dots, x_m), (x_{m+1}, x_{m+2}, \dots, x_n)) \in \mathcal{I}$ as input, n is the most natural measure of the input size.

$\square$

Exercise 1.4 Give a formal characterization of the problem of sorting an arbitrary sequence of integers.

Exercise 1.5 Give a formal characterization of the problem of mergingtwo ordered sequences of integers.

Exercise 1.6 Give a formal characterization of the following problem. Given a sequence of integers $(x_1, x_2, \dots, x_n)$, determine whether there exist indices i, j, with $1 \leq i \neq j \leq n$, such that $x_i = x_j$.

Chapter 2

Recurrence Relations and Divide-and-Conquer Algorithms

Consider the following *recurrence*:

$$\begin{cases} T(n) = s(n)T(f(n)) + w(n), & \text{for } n > n_0, \qquad (2.1.a) \\ T(n) = T_0, & \text{for } n \leq n_0. \qquad (2.1.b) \end{cases}$$

In (2.1), n is a nonnegative integer variable, and n_0 and T_0 are nonnegative integer constants. Functions $s(\cdot), f(\cdot)$ and $w(\cdot)$ are nondecreasing, nonnegative integer functions of n (as a consequence, $T(\cdot)$ is also a nondecreasing and nonnegative integer function). Finally, $f(n) < n$ for any $n > n_0$. Regarding the integrality constraint, sometimes we will make use of a slightly improper but simpler notation (e.g., $f(n) = \sqrt{n}$ instead of $f(n) = \lceil \sqrt{n} \rceil$.)

Equation (2.1) is often useful in the analysis of *divide-and-conquer* algorithms, where a *problem instance* of size at most n_0 is solved directly, while an instance of size $n > n_0$ is solved by

(i) decomposing the instance into $s(n)$ instances of the same problem of size *at most* $f(n) < n$ each;

(ii) recursively, solving the $s(n)$ smaller instances;

(iii) combining the solutions to the $s(n)$ instances of size $f(n)$ into a solution to the instance of size n.

Here, $w(n)$ is an *upper bound* to the overall running time of the decomposition and the combination procedures. Also, T_0 is an *upper bound* to the running time of the algorithm on instances of size $n \leq n_0$. With the given interpretation of $n_0, T_0, s(\cdot), f(\cdot)$, and $w(\cdot)$, Equation (2.1) *uniquely* defines a function $T(n)$, which represents an *upper bound* to the running time complexity of the given algorithm for any problem instance of size n.

The following notation is useful to formulate the general solution of Equation (2.1). We let $f^{(0)}(n) = n$, and for $i > 0$, $f^{(i+1)}(n) = f(f^{(i)}(n))$. We also denote by $f^{\star}(n, n_0)$ the largest k such that $f^{(k)}(n) > n_0$. (note that, if $n \leq n_0$, $f^{\star}(n, n_0)$ is not defined.)

With the above notation, $f^{(\ell)}(n)$ is the size of a single problem instance at the ℓ-th level of recursion, $\ell = 0$ corresponding to the initial call. Level $\ell = f^{\star}(n, n_0)$ is the last for which $f^{(\ell)}(n) > n_0$ and hence it is the last for which Equation (1.a) applies. At level $f^{\star}(n, n_0) + 1$, Equation (1.b) applies instead.

Thus, for $0 \leq \ell \leq f^{\star}(n, n_0)$, the work spent on a single problem instance at level ℓ is $w(f^{(\ell)}(n))$. For $\ell = f^{\star}(n, n_0) + 1$, the work per problem instance is T_0.

The instance at level 0 generates $s(n)$ instances at level 1, each of which generates $s(f(n))$ instances at level 2, each of which generates $s(f^{(2)}(n))$ instances at level 3, ..., each of which generates $s(f^{(\ell-1)}(n))$ instances at level ℓ. Therefore, *the total* number of instances at level ℓ is

$$s(n) \cdot s(f(n)) \cdot s(f^{(2)}(n)) \cdot \ldots \cdot s(f^{(\ell-1)}(n)) = \prod_{j=0}^{\ell-1} s(f^{(j)}(n)), \tag{2.2}$$

where if $\ell - 1 < 0$ the value of (2.2) is assumed to be 1.

By combining the considerations of the last three paragraphs, we obtain the following expression for the general solution of Equation (2.1):

$$\begin{cases} T(n) = \sum_{\ell=0}^{f^{\star}(n,n_0)} \left(\left[\prod_{j=0}^{\ell-1} s(f^{(j)}(n)) \right] w(f^{(\ell)}(n)) \right) + \\ \qquad + \left[\prod_{j=0}^{f^{\star}(n,n_0)} s(f^{(j)}(n)) \right] T_0, & \text{for } n > n_0, \quad (2.3.a) \\ \\ T(n) = T_0, & \text{for } n \leq n_0. \quad (2.3.b) \end{cases}$$

Exercise 2.1 Study Subsection 2-6 *Iterated functions* at p.40 of Cormen, Leiserson and Rivest's book and solve Cases (b) ($f(n) = n - 1$, $n_0 = 0$), (c) ($f(n) = n/2$, $n_0 = 1$), (e) ($f(n) = n^{1/2}$, $n_0 = 2$) and (f) ($f(n) = n^{1/2}$, $n_0 = 1$) of the problem proposed there at the end of the subsection. Observe the following correspondence between the notation $f^*_{n0}(n)$ adopted by Cormen, Leiserson and Rivest's book and the notation $f^*(n, n_0)$ introduced above:

$$f^*_{n0}(n) = f^*(n, n_0) + 1.$$

Answer: For the sake of generality, we will consider our functions to be real-valued. Moreover, the domain of f is $\Re$ in cases b, c, and $[0, \infty) \subseteq \Re$ in cases e, f.

(b) Note that $f^{(1)}(n) = n - 1$, $f^{(2)}(n) = f^{(1)}(n) - 1 = n - 2$, and, in general, $f^{(i)}(n) = f^{(i-1)}(n) - 1 = n - i$. If $n \leq 0$, then $f^*_0(n) = 0$. Otherwise, letting

$$f^{(i)}(n) = n - i \leq 0,$$

we get $i \geq n$. Thus, the least $i \geq 0$ such that $f^{(i)}(n) = n - i \leq 0$ is $\lceil n \rceil$. Hence,

$$f^*_0(n) = \begin{cases} \lceil n \rceil & \text{if } n > 0, \\ 0 & \text{otherwise.} \end{cases}$$

(c) Note that $f^{(1)}(n) = n/2$, $f^{(2)}(n) = f^{(1)}(n)/2 = n/2^2$, and, in general, $f^{(i)}(n) = f^{(i-1)}(n)/2 = n/2^i$. If $n \leq 1$, then $f^*_1(n) = 0$. Otherwise, letting

$$f^{(i)}(n) = n/2^i \leq 1,$$

we get $n \leq 2^i$, whence $i \geq \log_2 n$. Thus, the least $i \geq 0$ such that $f^{(i)}(n) = n/2^i \leq 1$ is $\lceil \log_2 n \rceil$. Hence,

$$f^*_1(n) = \begin{cases} \lceil \log_2 n \rceil & \text{if } n > 1, \\ 0 & \text{otherwise.} \end{cases}$$

(e) Note that $f^{(1)}(n) = n^{1/2}$, $f^{(2)}(n) = (f^{(1)}(n))^{1/2} = n^{1/4}$, and, in general, $f^{(i)}(n) = (f^{(i-1)}(n))^{1/2} = n^{1/2^i}$. If $n \leq 2$, then $f^*_2(n) = 0$. Otherwise, letting

$$f^{(i)}(n) = n^{2^{-i}} \leq 2,$$

we get $\log_2 n^{1/2^i} = \log_2 n/2^i \leq \log_2 2 = 1$. Therefore, we have $2^i \geq \log_2 n$, whence $i \geq \log_2 \log_2 n$. Thus, the least $i \geq 0$ such that $f^{(i)}(n) = n^{2^{-i}} \leq 2$ is $\lceil \log_2 \log_2 n \rceil$.

Hence,

$$f_2^*(n) = \begin{cases} \lceil \log_2 \log_2 n \rceil & \text{if } n > 2, \\ 0 & \text{otherwise.} \end{cases}$$

(f) If $n \leq 1$, then $f_1^*(n) = 0$. Otherwise,

$$f^{(i)}(n) = n^{1/2^i} > 1,$$

for all i. Thus, $f_1^*(n)$ is undefined. Hence,

$$f_1^*(n) = \begin{cases} \text{undefined} & \text{if } n > 1, \\ 0 & \text{otherwise.} \end{cases}$$

$\square$

Exercise 2.2 Consider the recurrence $T(n) = 2T(\frac{n}{2}) + w(n)$, with $T(1) = T_0$, an arbitrary constant. Write the general solution. Specialize your formula in the following cases:

(a) $w(n) = a$ (a constant); **(c)** $w(n) = a \log_2^2 n$; **(e)** $w(n) = an^2$;

(b) $w(n) = a\log_2 n$; **(d)** $w(n) = an$; **(f)** $w(n) = n/\log_2 n$.

Answer:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + w(n) \\ &= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i w(n/2^i) \qquad k \geq 1 \\ &= 2^{\log_2 n} T(1) + \sum_{i=0}^{\log_2 n - 1} 2^i w(n/2^i) \\ &= nT_0 + \sum_{i=0}^{\log_2 n - 1} 2^i w(n/2^i). \end{aligned}$$

(a) $w(n) = a$ (a constant).

$$\begin{aligned} \sum_{i=0}^{\log_2 n - 1} 2^i w(n/2^i) &= a \sum_{i=0}^{\log_2 n - 1} 2^i \\ &= a \left(2^{\log_2 n} - 1\right) \\ &= a(n-1) \end{aligned}$$

Thus,

$$T(n) = nT_0 + a(n-1) = (a+T_0)n - a.$$

(b) $w(n) = a\log_2 n$.

$$\begin{aligned}
\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= a\sum_{i=0}^{\log_2 n-1} 2^i \log_2 \frac{n}{2^i} \\
&= a[\log_2 n + 2(\log_2 n - 1) + 2^2(\log_2 n - 2) + \ldots \\
&\quad + 2^{\log_2 n-2}\cdot 2 + 2^{\log_2 n-1}\cdot 1] \\
&= a\left[\underbrace{1+1+\cdots+1}_{\log_2 n} + \underbrace{2+2+\cdots+2}_{\log_2 n-1} + \underbrace{2^2+2^2+\cdots+2^2}_{\log_2 n-2} + \right. \\
&= \left. \quad \cdots + \underbrace{2^{\log_2 n-2} + 2^{\log_2 n-2}}_{2} + \underbrace{2^{\log_2 n-1}}_{1}\right] \\
&= a\left[\sum_{i=0}^{\log_2 n-1} 2^i + \sum_{i=0}^{\log_2 n-2} 2^i + \sum_{i=0}^{\log_2 n-3} 2^i + \ldots + \sum_{i=0}^{1} 2^i + \sum_{i=0}^{0} 2^i\right] \\
&= a[(2^{\log_2 n} - 1) + (2^{\log_2 n-1} - 1) + \cdots + (2^2 - 1) + (2-1)] \\
&= a\left[\sum_{i=1}^{\log_2 n} 2^i - \log_2 n\right] \\
&= a\left[2^{\log_2 n+1} - 2 - \log_2 n\right] \\
&= a[2n - \log_2 n - 2].
\end{aligned}$$

Thus,

$$T(n) = nT_0 + a(2n - \log_2 n - 2) = (T_0 + 2a)n - a\log_2 n - 2a.$$

(c) $w(n) = a\log_2^2 n$.

$$\begin{aligned}
\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= a\sum_{i=0}^{\log_2 n-1} 2^i \log_2^2 \frac{n}{2^i} \\
&= a[\log_2^2 n + 2^1(\log_2 n - 1)^2 + 2^2(\log_2 n - 2)^2 + \ldots + 2^{\log_2 n-1}\cdot 1^2] \\
&= a\left[\underbrace{1+1+\cdots+1}_{\log_2^2 n} + \underbrace{2+2+\cdots+2}_{(\log_2 n-1)^2} + \underbrace{2^2+2^2+\cdots+2^2}_{(\log_2 n-2)^2} + \right. \\
&= \left. \quad \cdots + \underbrace{2^{\log_2 n-2} + 2^{\log_2 n-2}}_{2^2} + \underbrace{2^{\log_2 n-1}}_{1}\right]
\end{aligned}$$

Observing that

$$(\log_2 n - i)^2 = \sum_{k=1}^{\log_2 n - i} (2k-1),$$

the above expression can be written as

$$\begin{aligned}
\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= a\left[1\cdot \sum_{i=0}^{\log_2 n-1} 2^i + 3\cdot \sum_{i=0}^{\log_2 n-2} 2^i + 5\cdot \sum_{i=0}^{\log_2 n-3} 2^i + \cdots \right.\\
&\quad \left. + (2\log_2 n - 3)\cdot \sum_{i=0}^{1} 2^i + (2\log_2 n - 1)\cdot \sum_{i=0}^{0} 2^i \right]\\
&= a[1(2^{\log_2 n} - 1) + 3(2^{\log_2 n-1} - 1) + 5(2^{\log_2 n-2} - 1) + \cdots\\
&\quad + (2\log_2 n - 3)(2^2 - 1) + (2\log_2 n - 1)(2-1)]\\
&= a[1(2^{\log_2 n}) + 3(2^{\log_2 n-1}) + 5(2^{\log_2 n-2}) + \cdots\\
&\quad + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2)\\
&\quad - (1 + 3 + 5 + \cdots + (2\log_2 n - 3) + (2\log_2 n - 1))]\\
&= a[1(2^{\log_2 n}) + 3(2^{\log_2 n-1}) + 5(2^{\log_2 n-2}) + \cdots\\
&\quad + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2)] + \log_2^2 n. \qquad (2.4)
\end{aligned}$$

Now,

$$\begin{aligned}
1(2^{\log_2 n}) &+ 3(2^{\log_2 n-1}) + 5(2^{\log_2 n-2}) + \cdots + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2\\
&= 2(2^{\log_2 n}) + 4(2^{\log_2 n-1}) + 6(2^{\log_2 n-2}) + \cdots + (2\log_2 n - 2)2^2 + (2\log_2 n)2\\
&\quad - [(2^{\log_2 n}) + (2^{\log_2 n-1}) + (2^{\log_2 n-2}) + \cdots + 2^2 + 2]\\
&= 2^2(2^{\log_2 n-1} + 2\cdot 2^{\log_2 n-2} + 3\cdot 2^{\log_2 n-3} + \cdots + \log_2 n\cdot 1) - 2(2^{\log_2 n} - 1)\\
&= 2^2(2n - \log_2 n - 2) - 2(n-1)\\
&\quad \text{(see Point (b))}\\
&= 6n - 4\log_2 n - 6. \qquad (2.5)
\end{aligned}$$

Substituting (1.11) into (1.10), we get

$$\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) = a(6n - 4\log_2 n - 6 - \log_2^2 n).$$

Therefore,

$$T(n) = nT_0 + a(6n - 4\log_2 n - 6 - \log_2^2 n)$$

$$= \quad (T_0 + 6a)n - 4a\log_2 n - a\log_2^2 n - 6a.$$

Comment We have encountered $\sum_{i=1}^m ix^i, \sum_{i=1}^m i^2x^i$ in Parts (b) and (c) above. The following alternative approach can be used to evaluate series of the form $S(k) = \sum_{i=1}^m i^k x^i$, for $k = 0, 1, 2, \ldots$.

$$S(0) = \sum_{i=1}^m x^i = \frac{x^{m+1} - x)}{x-1}. \tag{2.6}$$

Note that

$$\frac{d}{dx}S(0) = \frac{d}{dx}\left(\sum_{i=1}^m x^i\right) = \sum_{i=1}^m ix^{i-1},$$

therefore,

$$S(1) = x\frac{d}{dx}S(0).$$

In general,

$$S(k+1) = x\frac{d}{dx}S(k) \quad k \geq 0.$$

Hence, starting with (1.12), one can successively evaluate $S(1), S(2), \ldots$, using this equation.

(d) $w(n) = an$.

$$\begin{aligned}\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= a\sum_{i=0}^{\log_2 n-1} 2^i(\frac{n}{2^i}) \\ &= an\log_2 n.\end{aligned}$$

Thus,

$$T(n) = an\log_2 n + nT_0.$$

(e) $w(n) = an^2$.

$$\begin{aligned}\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= a\sum_{i=0}^{\log_2 n-1} 2^i(\frac{n^2}{2^{2i}}) \\ &= an^2\sum_{i=0}^{\log_2 n-1}\frac{1}{2^i} \\ &= an^2\cdot 2(1-\frac{1}{n}) \\ &= 2an(n-1).\end{aligned}$$

Thus,

$$T(n) = nT_0 + 2an^2 - 2an = 2an^2 + (T_0 - 2a)n.$$

(f) $w(n) = n/\log_2 n$.

$$\begin{aligned}
\sum_{i=0}^{\log_2 n-1} 2^i w(n/2^i) &= \sum_{i=0}^{\log_2 n-1} 2^i (\frac{n}{2^i}) \frac{1}{\log_2(\frac{n}{2^i})} \\
&= n \sum_{i=0}^{\log_2 n-1} \frac{1}{\log_2(\frac{n}{2^i})} \\
&= n \sum_{i=1}^{\log_2 n} \left(\frac{1}{i}\right) \\
&= n \ln \log_2 n + O(n),
\end{aligned}$$

since

$$\log_e(\log_2 n + 1) = \int_1^{\log_2 n+1} \frac{1}{x}dx \le \sum_{i=1}^{\log_2 n} \frac{1}{i} \le 1 + \int_1^{\log_2 n} \frac{1}{x}dx = 1 + \log_e \log_2 n.$$

Thus,

$$T(n) = nT_0 + n \ln \log_2 n + O(n) = n \ln \log_2 n + O(n).$$

□

Exercise 2.3 Solve the following recurrence when the parameter n is an integral power of 3:

$$\begin{cases} T(n) = 6T(\frac{n}{3}) + n(n-1), & n > 1, \\ T(1) = 4. \end{cases}$$

Answer: The following table summarizes all the relevant information obtained from the recursion tree:

level	**size**	**work**	**# problems**
0	n	$n^2 - n$	1
1	$\frac{n}{3}$	$\frac{n^2}{9} - \frac{n}{3}$	6
2	$\frac{n}{9}$	$\frac{n^2}{81} - \frac{n}{9}$	6^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ℓ	$\frac{n}{3^\ell}$	$\left(\frac{n}{3^\ell}\right)^2 - \frac{n}{3^\ell}$	6^ℓ
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_3 n - 1$	$\frac{n}{3^{\log_3 n - 1}}$	$\left(\frac{n}{3^{\log_3 n-1}}\right)^2 - \frac{n}{3^{\log_3 n-1}}$	$6^{\log_3 n - 1}$
$\log_3 n$	$\frac{n}{3^{\log_3 n}}$	4	$6^{\log_3 n}$

Using the information in the above table we can write:

$$\begin{aligned}
T(n) &= 4 \cdot 6^{\log_3 n} + \sum_{\ell=0}^{\log_3 n - 1} 6^\ell \left[\left(\frac{n}{3^\ell}\right)^2 - \frac{n}{3^\ell}\right] \\
&= 4 \cdot n^{1+\log_3 2} + n^2 \sum_{\ell=0}^{\log_3 n-1} \left(\frac{6}{9}\right)^\ell - n \sum_{\ell=0}^{\log_3 n - 1} 2^\ell \\
&= 4 \cdot n^{1+\log_3 2} + 3n^2 \left(1 - \left(\frac{2}{3}\right)^{\log_3 n}\right) - n\left(2^{\log_3 n} - 1\right) \\
&= 4 \cdot n^{1+\log_3 2} + 3n^2 - 3n^{2+\log_3 2 - 1} - n^{1+\log_3 2} + n \\
&= 3n^2 + n.
\end{aligned}$$

$\square$

Exercise 2.4 Solve the following recurrence when the parameter n is a power of two:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + \frac{3}{4}n^2 + 2\log n - 1, \ n > 1, \\ T(1) &= 1. \end{aligned}$$

Answer: Let $f(n) = \frac{3}{4}n^2 + 2\log n - 1$. Then, for $n > 1$,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + f(n) \\ &= T\left(\frac{n}{4}\right) + f\left(\frac{n}{2}\right) + f(n) \\ &\vdots \\ &= T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} f\left(\frac{n}{2^j}\right), \text{ for } 1 \leq i \leq \log n. \end{aligned}$$

For $i = \log n$, we get

$$T(n) = T(1) + \sum_{j=0}^{\log n - 1} f\left(\frac{n}{2^j}\right).$$

We have:

$$\begin{aligned} \sum_{j=0}^{\log n-1} f\left(\frac{n}{2^j}\right) &= \frac{3}{4}n^2 \sum_{j=0}^{\log n-1} 4^{-j} + 2\sum_{j=0}^{\log n-1} (\log n - j) - \sum_{j=0}^{\log n-1} 1 \\ &= n^2\left(1 - \frac{1}{n^2}\right) + (\log n)(\log n + 1) - \log n \\ &= n^2 + \log^2 n - 1. \end{aligned}$$

Therefore we have:

$$T(n) = n^2 + \log^2 n. \tag{2.7}$$

Since $1^2 + \log^2 1 = 1$, (1.13) holds for any value of $n \geq 1$. □

Exercise 2.5 **(a)** Solve the following recurrence when the parameter n is a power of two and c and d are positive constants:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \left(\sqrt{2} - 1\right) c\sqrt{n}, \ n > n_0, \\ T(n_0) &= dn_0\sqrt{n_0}. \end{aligned}$$

(b) Determine the value of n_0 which minimizes the solution.

Answer:

(a) For $n > n_0$ we have:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \left(\sqrt{2}-1\right)c\sqrt{n} \\
&= 2^2T\left(\frac{n}{2^2}\right) + \left(1+\sqrt{2}\right)\left(\sqrt{2}-1\right)c\sqrt{n} \\
&= 2^3T\left(\frac{n}{2^3}\right) + \left(1+\sqrt{2}+\left(\sqrt{2}\right)^2\right)\left(\sqrt{2}-1\right)c\sqrt{n} \\
&\vdots \\
&= 2^iT\left(\frac{n}{2^i}\right) + \left(\sqrt{2}-1\right)c\sqrt{n}\sum_{k=0}^{i-1}\left(\sqrt{2}\right)^k.
\end{aligned}$$

For $i = \log n/n_0$ we get

$$\begin{aligned}
T(n) &= dn\sqrt{n_0} + \left(\sqrt{2}-1\right)c\sqrt{n}\sum_{k=0}^{\log(n/n_0)-1}\left(\sqrt{2}\right)^k \\
&= dn\sqrt{n_0} + c\left(\sqrt{\frac{n}{n_0}}-1\right)\sqrt{n} \\
&= \left(d\sqrt{n_0} + c\frac{1}{\sqrt{n_0}}\right)n - c\sqrt{n} \\
&= n\,\mathit{coeff}(n_0) - c\sqrt{n}.
\end{aligned}$$

(b) By taking the partial derivative of $T(n)$ with respect to $\sqrt{n_0}$ we obtain

$$\frac{\delta T(n)}{\delta\sqrt{n_0}} = \frac{\delta\,\mathit{coeff}(n_0)}{\delta\sqrt{n_0}} = d - \frac{c}{n_0},$$

whence

$$\frac{\delta T(n)}{\delta\sqrt{n_0}} \geq 0 \text{ iff } n_0 \geq \frac{c}{d},$$

for any value of the parameter n. Since n_0 has to be an integral power of two, the solution is minimized for either

$$n_0' = \max\left\{1, 2^{\lfloor\log c/d\rfloor}\right\} \quad \text{or} \quad n_0'' = \max\left\{1, 2^{\lceil\log c/d\rceil}\right\},$$

depending on whether or not $\mathit{coeff}(n_0') \leq \mathit{coeff}(n_0'')$. □

Exercise 2.6 Solve the following recurrence when the parameter n is a power of four:

$$\begin{aligned} T(n) &= 16\,T\left(\frac{n}{4}\right) + 2\,n^2, \quad n > 1, \\ T(1) &= 0. \end{aligned}$$

Answer: The above recurrence can be solved in a number of standard ways. Here, we choose to illustrate a trick (also applicable in other cases) that simplifies the recurrence. Letting $T(n) = n^2 Q(n)$, we obtain $T(n/4) = (n^2/16)Q(n/4)$, and $T(1) = Q(1) = 0$. Therefore, the recurrence for $T(n)$ can be rewritten for $Q(n)$ as follows:

$$\begin{aligned} Q(n) &= Q\left(\frac{n}{4}\right) + 2, \quad n > 1, \\ Q(1) &= 0. \end{aligned}$$

To solve for Q, unfold the relation $k-1$ times to obtain:

$$Q(n) = Q\left(\frac{n}{4^k}\right) + 2k.$$

Letting $k = (1/2)\log_2 n$, we have $Q(n/4^k) = Q(1) = 0$, whence $Q(n) = 2k = \log_2 n$. Finally,

$$T(n) = n^2 \log_2 n.$$

□

Exercise 2.7 Solve the following recurrence when the parameter n is a power of two:

$$\begin{aligned} T(n) &= (\log n)T\left(\frac{n}{2}\right) + 1, \ n > 1, \\ T(1) &= 1. \end{aligned}$$

Answer:

$$\begin{aligned} T(n) &= (\log n)T\,(n/2) + 1 \\ &= (\log n)(\log(n/2)T(n/4) + 1) + 1 \\ &= (\log n)(\log n - 1)T(n/4) + 1 + \log n \\ &= (\log n)(\log n - 1)(\log n - 2)T(n/8) + 1 + \log n + (\log n)(\log n - 1) \\ &\vdots \\ &= \left(\prod_{j=0}^{i-1}(\log n - j)\right) T(n/2^i) + 1 + \sum_{j=0}^{i-2}\left(\prod_{k=0}^{j}(\log n - k)\right), \end{aligned}$$

for $2 \le i \le \log n$. For $i = \log n$, $T(n/2^i) = T(1) = 1$, and we get

$$\begin{aligned} T(n) &= \prod_{j=0}^{\log n - 1} (\log n - j) + 1 + \sum_{j=0}^{\log n - 2} \left(\prod_{k=0}^{j} (\log n - k) \right) \\ &= (\log n)! + 1 + \sum_{j=0}^{\log n - 2} \frac{(\log n)!}{(\log n - j - 1)!} \\ &= \frac{(\log n)!}{0!} + \frac{(\log n)!}{(\log n)!} + \sum_{k=1}^{\log n - 1} \frac{(\log n)!}{k!} \\ &= (\log n)! \left(\sum_{k=0}^{\log n} \frac{1}{k!} \right) \end{aligned}$$

Since $\sum_{k=0}^{\log n} 1/k! < \sum_{k=0}^{\infty} 1/k! = e$ we have $T(n) = O((\log n)!)$. □

Exercise 2.8

(a) Solve the following recurrence when the parameter n is a double power of two (i.e., $n = 2^{2^i}$, for some $i \ge 0$).

$$\begin{cases} T(n) = \sqrt{n}T\left(\sqrt{n}\right) + \sqrt{n} - 1, \quad n > 2 \\ T(2) = 1. \end{cases}$$

(b) Design a divide-and-conquer algorithm for the problem of finding the maximum of a set of $n = 2^{2^i}$ numbers that performs a number of comparisons obeying to the above recurrence.

Answer:

(a) Let us compute $T(n)$ for small values of $n = 2^{2^i}$, e.g., $n = 2, 4\left(= 2^{2^1}\right), 16\left(= 2^{2^2}\right)$, $256\left(= 2^{2^3}\right)$.

$$\begin{aligned} T(2) &= 1 \\ T(4) &= 2 \cdot T(2) + 2 - 1 = 2 \cdot 1 + 2 - 1 = 3 \\ T(16) &= 4 \cdot T(4) + 4 - 1 = 4 \cdot 3 + 4 - 1 = 15 \\ T(256) &= 16 \cdot T(16) + 16 - 1 = 16 \cdot 15 + 16 - 1 = 255 \end{aligned}$$

Based on the above values, we guess that $T(n) = n - 1$. Let us prove our guess by induction on $i \ge 0$, where $n = 2^{2^i}$. The base of the induction holds, since $T\left(2^{2^0}\right) = T(2) = 1 = 2 - 1$.

Let us now assume that $T\left(2^{2^k}\right) = 2^{2^k} - 1$ for all values $k < i$. For $k = i$, we have:

$$\begin{aligned}
T\left(2^{2^i}\right) &= \left(2^{2^i}\right)^{1/2} \cdot T\left(\left(2^{2^i}\right)^{1/2}\right) + \left(2^{2^i}\right)^{1/2} - 1 \\
&= 2^{2^{i-1}} \cdot T\left(2^{2^{i-1}}\right) + 2^{2^{i-1}} - 1 \\
&= 2^{2^{i-1}} \cdot \left(2^{2^{i-1}} - 1\right) + 2^{2^{i-1}} - 1 \\
& \text{(inductive hypothesis)} \\
&= 2^{2^{i-1}} \cdot 2^{2^{i-1}} - 1 \\
&= 2^{2^i} - 1.
\end{aligned}$$

The inductive thesis follows.

(b) Let $A[1..n]$ be an array of $n = 2^{2^i}$ numbers. A recursive algorithm SQRT_MAX performing a number of comparisons obeying to the above recurrence is the following:

```
SQRT_MAX($A$)
$n \leftarrow$ length($A$)
**if** $n = 2$
  **then if** $A[1] \geq A[2]$
          **then return** $A[1]$
          **else return** $A[2]$
**for** $i \leftarrow 1$ **to** $\sqrt{n}$
    **do** $TMP[i] \leftarrow$ SQRT_MAX ($A\,[\,(i-1) * \sqrt{n} + 1 \,..\, i * \sqrt{n}\,]$)
$max \leftarrow TMP[1]$
**for** $i \leftarrow 2$ **to** $\sqrt{n}$
    **do if** $max < TMP[i]$
         **then** $max \leftarrow TMP[i]$
**return** $max$
```

For $n > 2$, the above algorithm recursively determines the maxima for the sub-arrays

$$A\left[\,(i-1) * \sqrt{n} + 1 \,..\, i * \sqrt{n}\,\right], \quad 1 \leq i \leq \sqrt{n},$$

and then determines the overall maximum by performing $\sqrt{n} - 1$ comparisons among these maxima. The correctness of the algorithm follows from the fact that the above sub-arrays induce a partition of the n indices of the original array. Since $i*\sqrt{n} - ((i-1) * \sqrt{n} + 1) + 1 = \sqrt{n}$, for any $1 \leq i \leq \sqrt{n}$, the number $T(n)$ of comparisons performed by SQRT_MAX(A) when length(A) $= n$ is clearly given by the recurrence solved in Part (a). Therefore $T(n) = n - 1$. □

Exercise 2.9 Solve the following recurrence when the parameter n is 2^{3^i}, for some $i \geq 0$:

$$\begin{aligned} T(n) &= n^{2/3}T\left(n^{1/3}\right) + n^{2/3} - 1,\ n > 2, \\ T(2) &= 1. \end{aligned}$$

Answer: We have:

$$\begin{aligned} T(8) &= 4 \cdot T(2) + 4 - 1 = 4 \cdot (1+1) - 1 = 7, \\ T(512) &= 64 \cdot T(8) + 64 - 1 = 64 \cdot (7+1) - 1 = 511. \end{aligned}$$

As in the previous exercise, we guess that $T\left(2^{3^i}\right) = 2^{3^i} - 1$, for any $i \geq 0$ and prove our guess by induction on i. The basis is clearly true, since $T\left(2^{3^0}\right) = 1 = 2^{3^0} - 1$. For $i > 0$ we have:

$$\begin{aligned} T\left(2^{3^i}\right) &= \left(2^{3^i}\right)^{2/3} \cdot T\left(\left(2^{3^i}\right)^{1/3}\right) + \left(2^{3^i}\right)^{2/3} - 1 \\ &= 2^{2 \cdot 3^{i-1}} T\left(2^{3^{i-1}}\right) + 2^{2 \cdot 3^{i-1}} - 1 \\ &= 2^{2 \cdot 3^{i-1}} \left(2^{3^{i-1}} - 1\right) + 2^{2 \cdot 3^{i-1}} - 1 \\ &= 2^{2 \cdot 3^{i-1}} \cdot \left(2^{3^{i-1}} - 1 + 1\right) - 1 \\ &= 2^{2 \cdot 3^{i-1}} \cdot 2^{3^{i-1}} - 1 \\ &= \left(2^{(2+1) \cdot 3^{i-1}}\right) - 1 \\ &= 2^{3^i} - 1, \end{aligned}$$

which completes our proof. □

Exercise 2.10 Solve the following recurrence when the parameter n is a power of two:

$$\begin{aligned} T(n) &= \frac{T^2\left(n/2\right)}{n} + n,\ n > 1, \\ T(1) &= 2. \end{aligned}$$

Answer: Let us compute $T(n)$, for small values of the parameter n, by "manually" unfolding the recursion, so to get an idea of the form of the solution.

$$\begin{aligned} n = 1: &\quad T(1) = 2; \\ n = 2: &\quad T(2) = T^2(1)/2 + 2 = 4; \\ n = 4: &\quad T(4) = T^2(2)/2 + 4 = 8; \end{aligned}$$

$$n = 8: \quad T(8) = \quad T^2(4)/2 + 8 = 16.$$

The above values suggest that

$$T(n) = 2n \tag{2.8}$$

is a plausible guess. Let us now try to confirm our guess by using induction. Since $T(1) = 2 = 2 \cdot 1$, Relation (1.14) holds for the base. Assume now that $T(n') = 2n'$, for any power of two $n' < n$. We have:

$$\begin{aligned} T(n) &= \frac{T^2\,(n/2)}{2} + n \\ &= \frac{(2 \cdot n/2)^2}{n} + n \\ &= \frac{n^2}{n} + n = 2n, \end{aligned}$$

therefore (1.14) holds for any power of two. □

Exercise 2.11 Develop a divide-and-conquer algorithm to compute the maximum and the minimum of a sequence $(a_1, a_2, \ldots, a_n)$. Analyze the number of comparisons. (To be interesting, the algorithm should perform fewer than $2(n-1)$ comparisons, which could be achieved by simply computing maximum and minimum seperately.) Show a diagram of the comparisons performed by your algorithm on input (7,4,5,2,1,6,3,8).

Answer: We divide $S = (a_1, a_2, \ldots, a_n)$ into two sequences, $S_1 = (a_1, a_2, \ldots, a_{n/2})$ and $S_2 = (a_{n/2+1}, a_{n/2+2}, \ldots, a_n)$ Then $\max\{S\} = \max\{\max\{S_1\}, \max\{S_2\}\}$, $\min\{S\} = \min\{\min\{S_1\}, \min\{S_2\}\}$. The algorithm is the following:

```
MAXMIN($S$)
Let $S = \{a_1, a_2, \ldots, a_n\}$
   **if** $n = 2$
      **then if** $a_1 \geq a_2$
               **then return** $(a_1, a_2)$
               **else return** $(a_2, a_1)$
   $S_1 \leftarrow \{a_1, a_2, \ldots, a_{n/2}\}$
   $S_2 \leftarrow \{a_{n/2+1}, a_{n/2+2}, \ldots, a_n\}$
   $(max_1, min_1) \leftarrow$MAXMIN$(S_1)$
   $(max_2, min_2) \leftarrow$MAXMIN$(S_2)$
   **return** (MAX$(max_1, max_2)$, MIN$(min_1, min_2)$)
```

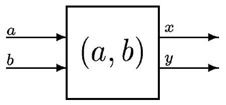
In the above algorithm, when the sequence S has two elements, say $S = (a_1, a_2)$, we simply compare a_1 and a_2 to obtain $\max\{S\}$ and $\min\{S\}$, thus requiring only one comparison. If

$|S| > 2$, the number of comparisons required to yield $\max\{S\}$ and $\min\{S\}$, given $\max\{S_1\}$, $\min\{S_1\}$, $\max\{S_2\}$ and $\min\{S_2\}$ is 2 (one to compute $\text{MAX}(max_1, max_2)$ and one to compute $\text{MIN}(min_1, min_2)$). Hence,

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 2, & \text{if } n > 2, \\ 1, & \text{if } n = 2. \end{cases}$$

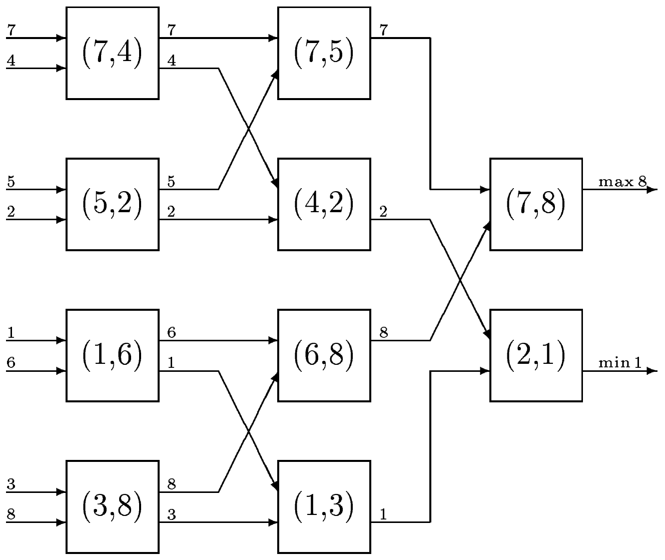
$$\begin{aligned} T(n) &= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=1}^{k} 2^i, \quad k \geq 1 \\ &= \frac{n}{2} T(2) + 2(2^{\log_2 n - 1} - 1) \\ &= \frac{n}{2} + 2(\frac{n}{2} - 1) \\ &= \frac{3n}{2} - 2. \end{aligned}$$

Diagram:



This diagram depicts a comparison between values a and b. The outputs x and y denote the maximum and minimum value, respectively.

The following diagram shows the comparisons performed by the algorithm on input $S = (7, 4, 5, 2, 1, 6, 3, 8)$.



Observe that $T(8) = \frac{3 \cdot 8}{2} - 2 = 10$. After $n/2$ comparisons, there are $n/2 - 1$ comparisons organized in a tree of minimum computations, and $n/2 - 1$ comparisons organized in a tree of maximum computations. $\square$

Exercise 2.12 On input two $n \times n$ matrices, Strassen's multiplication algorithm leads to a recurrence of the form

$$\begin{cases} T(N) &= 7T\left(\frac{N}{4}\right) + aN/4, \qquad N > 1, \\ T(1) &= 1, \qquad\qquad\qquad\quad N = 1, \end{cases}$$

where $N = n^2$ is the number of entries of the matrices.

(a) Show that the exact solution is

$$T(N) = \left(\frac{a}{3} + 1\right) N^{\frac{\log_2 7}{2}} - \left(\frac{a}{3}\right) N.$$

(b) Find (an approximation to) N_0 such that $T(N_0) = T_{DEF}(N_0)$ when $a = 15$. Recall that $T_{DEF}(N) = 2N^{3/2} - N$.

Answer:

(a) To verify the given solution, we can simply plug it into the recurrence equations:

$$\begin{aligned} T(1) &= 1^{\frac{1}{2}\log 7}\left(\frac{a}{3} + 1\right) - 1\left(\frac{a}{3}\right) \\ &= \frac{a}{3} + 1 - \frac{a}{3} = 1. \\ T\left(\frac{N}{4}\right) &= \left(\frac{a}{3} + 1\right)\left(\frac{N}{4}\right)^{\frac{1}{2}\log 7} - \frac{a}{3}\frac{N}{4} \\ &= \left(\frac{a}{3} + 1\right)\frac{N^{\frac{1}{2}\log 7}}{4^{\frac{1}{2}\log 7}} - \frac{aN}{12} = \left(\frac{a}{3} + 1\right)\frac{N^{\frac{1}{2}\log 7}}{7} - \frac{aN}{12}. \\ T(N) &= 7T\left(\frac{N}{4}\right) + \frac{aN}{4} = 7\left(\left(\frac{a}{3} + 1\right)\frac{N^{\frac{1}{2}\log 7}}{7} - \frac{aN}{12}\right) + \frac{aN}{4} \\ &= \left(\frac{a}{3} + 1\right)N^{\frac{1}{2}\log 7} - \frac{7aN}{12} + \frac{3aN}{12} = \left(\frac{a}{3} + 1\right)N^{\frac{1}{2}\log 7} - \frac{aN}{3}. \end{aligned}$$

However, we can also derive this solution from the recurrence solution given in class:

$$T(N) = \sum_{l=0}^{f^*(N,N_0)} \left(\prod_{j=0}^{\ell-1} s(f^{(j)}(N))\right) w(f^{(\ell)}(N)) + \prod_{\ell=0}^{f^*(N,N_0)} s(f^{(\ell)}(N))T_0.$$

For this problem, $s(N) = 7, f(N) = \frac{N}{4}, T_0 = 1, w(N) = \frac{aN}{4}$, and $N_0 = 1$. Thus, $f^*(N, N_0) = f^*(N, 1) = \log_4 N - 1 = \frac{1}{2}\log N - 1$. Plugging these into the formula, we get:

$$\begin{aligned}
T(N) &= \sum_{\ell=0}^{\frac{1}{2}\log N - 1} \left(\prod_{j=0}^{\ell-1} 7 \right) \frac{a}{4}\frac{N}{4^\ell} + \prod_{\ell=0}^{\frac{1}{2}\log N - 1} 7 \\
&= \frac{aN}{4} \sum_{\ell=0}^{\frac{1}{2}\log N - 1} \left(\frac{7}{4}\right)^\ell + 7^{\frac{1}{2}\log N} \\
&= \frac{aN}{4}\frac{4}{3}\left(\frac{7^{\frac{1}{2}\log N}}{N} - 1 \right) + 7^{\frac{1}{2}\log N} \\
&= 7^{\frac{1}{2}\log N}\left(\frac{a}{3} + 1\right) - \frac{a}{3}N \\
&= N^{\frac{1}{2}\log 7}\left(\frac{a}{3} + 1\right) - \frac{a}{3}N.
\end{aligned}$$

(b) Let $T_{DEF}(N_0) = T(N_0)$. We have:

$$\begin{aligned}
2N_0^{3/2} - N_0 = 6N_0^{\frac{\log_2 7}{2}} - 5N_0 \\
2N_0^{3/2} - 6N_0^{\frac{\log_2 7}{2}} + 4N_0 = 0 \\
N_0^{1/2} - 3N_0^{\frac{\log_2 7 - 2}{2}} + 2 = 0.
\end{aligned}$$

By trial and error, we obtain $83616 < N_0 < 83617$. □

Exercise 2.13 For n a power of 2, we say that an $n \times n$ matrix M is *repetitive* if either $n = 1$ or, when $n > 1$, M has the form

$$M = \begin{bmatrix} A & A \\ B & A \end{bmatrix},$$

where A and B are in turn $(n/2) \times (n/2)$ repetitive matrices.

(a) Design an efficient algorithm to multiply two repetitive matrices.

(b) Write the recurrence relation for the number $T(n)$ of arithmetic operations that your algorithm performs on $n \times n$ repetitive matrices.

(c) Solve the recurrence relation obtained at the previous point.

Answer:

(a) Let M_1 and M_2 be the two $n \times n$ repetitive matrices to be multiplied, and let

$$M_1 = \begin{bmatrix} A & A \\ B & A \end{bmatrix} \text{ and } M_2 = \begin{bmatrix} C & C \\ D & C \end{bmatrix}.$$

We have

$$M_1 \times M_2 = \begin{bmatrix} A \times C + A \times D & 2 \cdot A \times C \\ A \times D + B \times C & A \times C + B \times C \end{bmatrix}, \tag{2.9}$$

where $\times$ and $+$ denote, respectively, row-by-column multiplication and matrix sum, and $\cdot$ denotes multiplication by a scalar.

Since A, B, C, D are in turn repetitive matrices of size $n/2 \times n/2$, Equation (1.15) implies that three (recursive) row-by-column products of $n/2 \times n/2$ repetitive matrices, three sums of two $n/2 \times n/2$ (general) matrices and one multiplication of an $n/2 \times n/2$ (general) matrix by a scalar suffice to compute a row-by-column product of two $n \times n$ repetitive matrices. Let SUM(X, Y) be an algorithm that returns the sum of two $n \times n$ matrices X and Y, and let SC_PROD(c, X) be an algorithm that returns the scalar product $c \cdot X$. Clearly, we have $T_{\text{SUM}}(n) = T_{\text{SC\_PROD}}(n) = n^2$ The algorithm for multiplying two repetitive matrices is the following:

```
REP_MAT_MULT($M_1, M_2$)
    $n \leftarrow$ rows($M_1$)
    **if** $n = 1$ **then return** $M_1[1,1] \cdot M_2[1,1]$
    $A \leftarrow M_1[1..n/2, 1..n/2]$
    $B \leftarrow M_1[(n/2+1)..n, 1..n/2]$
    $C \leftarrow M_2[1..n/2, 1..n/2]$
    $D \leftarrow M_2[(n/2+1)..n, 1..n/2]$
    $T1 \leftarrow$REP_MAT_MULT($A, C$)
    $T2 \leftarrow$REP_MAT_MULT($A, D$)
    $T3 \leftarrow$REP_MAT_MULT($B, C$)
    $M[1..n/2, 1..n/2] \leftarrow$ SUM($T_1, T_2$)
    $M[1..n/2, (n/2+1)..n] \leftarrow$ SC_PROD($2, T_1$)
    $M[(n/2+1)..n, 1..n/2] \leftarrow$ SUM($T_2, T_3$)
    $M[(n/2+1)..n, (n/2+1)..n] \leftarrow$ SUM($T_1, T_3$)
**return** M
```

(b) To multiply two repetitive $n \times n$ matrices, we perform three recursive calls on $n/2 \times n/2$ matrices and then combine the results of the three calls using three sums of $n/2 \times n/2$

matrices and one scalar product. Therefore, the total work is $w(n) = 4 \cdot (n^2/4) = n^2$, and we obtain the following recurrence:

$$\begin{cases} T(n) &= 3T(n/2) + 3T_{\text{SUM}}(n/2) + T_{\text{SC\_PROD}}(n/2) \\ &= 3T(n/2) + n^2, \qquad n > 1, \\ T(1) &= 1. \end{cases} \tag{2.10}$$

(c) In Recurrence (1.16), we have $s(n) = 3$, $f^{(i)}(n) = n/2^i$, $w(n) = n^2$, $T_0 = 1$ and $N_0 = 1$. Therefore,

$$\begin{aligned} T(n) &= \sum_{\ell=0}^{\log_2 n - 1} 3^\ell n^2/4^\ell + 3^{\log_2 n} \\ &= n^2 \sum_{\ell=0}^{\log_2 n - 1} (3/4)^\ell + n^{\log_2 3} \\ &= 4n^2 \left(1 - (3/4)^{\log_2 n}\right) + n^{\log_2 3} \\ &= 4n^2 \left(1 - n^{\log_2 3 - 2}\right) + n^{\log_2 3} \\ &= 4n^2 - 3n^{\log_2 3}, \end{aligned}$$

whence $T(n) = \Theta(n^2)$. Let us prove that the above formula is correct. We have $T(1) = 1 = 4 \cdot 1 - 3 \cdot 1^{\log_2 3}$. If the formula holds for values of the parameter less than n, then

$$\begin{aligned} T(n) &= 3T(n/2) + n^2 \\ &= 3\left(4(n/2)^2 - 3(n/2)^{\log_2 3}\right) + n^2 \\ &= 3n^2 - 9n^{\log_2 3}/3 + n^2 \\ &= 4n^2 - 3n^{\log_2 3}. \end{aligned}$$

Observe that the sum of two repetitive matrices is a repetitive matrix and therefore contains many repeated entries, which need to be computed only once. Indeed, to sum two $n \times n$ repetitive matrices, we only have to compute two sums of $n/2 \times n/2$ repetitive matrices, with no extra arithmetic operations required for combining the results (we just have to make repeated copies of matrix entries). Hence, when summing two repetitive matrices,

$$\begin{cases} T_{\text{SUM}}(n) = 2T_{\text{SUM}}(n/2), \quad n > 1, \\ T_{\text{SUM}}(1) = 1. \end{cases}$$

(An identical recurrence, with the same motivation, holds for $T_{\text{SC\_PROD}}(n)$). By unfolding the above recurrence for $n = 2^k$, we have

$$T_{\mathrm{SUM}}(n)\,(\text{resp., } T_{\mathrm{SC\_PROD}}(n)) = 2^k \cdot T_{\mathrm{SUM}}(1)\,(\text{resp., } T_{\mathrm{SC\_PROD}}(1)) = n.$$

However, we cannot substitute these new running times for $T_{\mathrm{SUM}}(n)$ and $T_{\mathrm{SC\_PROD}}(n)$ in Recurrence (1.16), since we really need to sum (or take scalar products of) *general* matrices in the combining phase of REP_MAT_MULT. This is due to the fact that *the product of two repetitive matrices is not necessarily repetitive,* therefore, matrix $A \times C$ and matrix $A \times D$, for instance, may contain an arbitrary number of distinct elements. Summing $A \times C$ and $A \times D$ may entail as many as $(n/2)^2$ distinct scalar sums. □

Exercise 2.14 Let n be an even power of two, and let Π be the problem of merging $\sqrt{n}$ sorted sequences, each containing $\sqrt{n}$ elements, into a single sorted sequence of n elements.

(a) Design and analyze an algorithm for Π that performs at most $(n/2)\log n$ comparisons.

(b) Use the algorithm for Π developed in Part **(a)** to sort a sequence of n elements in at most $n \log n$ comparisons.

(c) Knowing that a lower bound for sorting is $n \log n - \gamma n$ comparisons, for a fixed constant $\gamma > 0$, determine a lower bound on the number of comparisons needed to solve Π.

Answer:

(a) Consider an input vector $A[1..n]$ containing the concatenation of the $\sqrt{n}$ sorted sequences. We can use the standard MERGESORT algorithm, halting the recursion when the subproblem size is $s = \sqrt{n}$. Algorithm SQRT_SORT below assumes a global knowledge of vector A:

```
SQRT_SORT($i, j$)
if $j - i + 1 = \sqrt{\text{length}(A)}$
  then return
$middle \leftarrow \lfloor (i+j)/2 \rfloor$
SQRT_SORT($i$, $middle$)
SQRT_SORT($middle + 1, j$)
MERGE($A, i, middle, j$)
return
```

The outer call is clearly SQRT_SORT$(1, \text{length}(A))$. Recall that MERGE$(i, middle, j)$ performs at most $j - i + 1$ comparisons (one for each element of the resulting sorted sub-array). Let $n = \text{length}(A)$ and $s = \sqrt{n}$. Then, the recurrence on the number of comparisons $T_{\text{SS}}(n)$ performed by SQRT_SORT(1, length(A)) is:

$$\begin{aligned} T_{\text{SS}}(s) &= 0 \\ T_{\text{SS}}(n) &= 2 \cdot T_{\text{SS}}\left(\frac{n}{2}\right) + n, \quad n > s. \end{aligned} \tag{2.11}$$

Unfolding $k-1$ times we have:

$$T_{\text{SS}}(n) = 2^k T_{\text{SS}}\left(\frac{n}{2^k}\right) + kn,$$

whence, by setting $k = \log(n/s) = \log\sqrt{n} = (1/2)\log n$,

$$\begin{aligned} T_{\text{SS}}(n) &= (n/s)T_{\text{SS}}(s) + (\log(n/s)) \cdot n \\ &= (n/2)\log n. \end{aligned}$$

since $T_{\text{SS}}(s) = T_{\text{SS}}(\sqrt{n}) = 0$. Therefore, our algorithm meets the required bound on the number of comparisons.

(b) Given n elements, we first group them into $\sqrt{n}$ sets of size $\sqrt{n}$ and sort each group using MERGESORT. Then we use SQRT_SORT to obtain the sorted sequence.

```
SORT($A$)
$n \leftarrow$ length($A$)
**for** $i \leftarrow 1$ **to** $\sqrt{n}$ **do**
    MERGESORT($A, (i-1) \cdot \sqrt{n} + 1, i \cdot \sqrt{n}$)
SQRT_SORT($1, n$)
**return**
```

By setting $s = 1$ in Recurrence (1.17), we observe that MERGESORT requires at most $T_{\text{MS}}(m) = m \log m$ comparisons to sort a sequence of m numbers. Therefore the overall number of comparisons performed by SORT(A) is

$$\begin{aligned} T_{\text{S}}(n) &= \sqrt{n}T_{\text{MS}}\left(\sqrt{n}\right) + T_{\text{SS}}(n) \\ &= \sqrt{n} \cdot \sqrt{n}\log\sqrt{n} + (n/2)\log n \\ &= (n/2)\log n + (n/2)\log n \\ &= n\log n. \end{aligned}$$

Essentially, SORT(A) coincides with MERGESORT, with the only difference that the activity performed by MERGESORT is viewed as the cascade of two phases.

(c) In Part **(b)**, we showed how to sort n elements by first sorting $\sqrt{n}$ sequences of size $\sqrt{n}$, and then solving Π. Let $T_{A_\Pi}(n)$ be the running time of any algorithm solving Π. Since we can separately sort the $\sqrt{n}$ sequences in $(n/2)\log n$ comparisons (calling MERGESORT $\sqrt{n}$ times) it must be

$$(n/2)\log n + T_{A_\Pi}(n) \geq n \log n - \gamma,$$

or we would obtain an algorithm for sorting which beats the lower bound. Therefore

$$T_{A_\Pi}(n) \geq (n/2)\log n - \gamma.$$

□

Exercise 2.15

(a) Design an optimal divide-and-conquer algorithm which takes as input a vector of $N = 2^n - 1$ elements and outputs a heap containing the same elements.

(b) Write the recurrence associated with the algorithm of Part (a).

(c) Solve the recurrence of Part (b).

Answer:

(a) Let H be the input vector containing $N = 2^n - 1$ elements. Our algorithm works directly on H as follows. We first (recursively) create two subheaps H_1 and H_2 (note that $|H_1| = |H_2| = 2^{n-1} - 1$) rooted at the two children of the root. These heaps are stored according to the typical heap indexing scheme (i.e., if a node is stored at index i, its children are found at indices $2i$ and $2i + 1$) in the locations of H starting from index 2 for H_1, and index 3 for H_2. Then we "meld" H_1 and H_2 into a single heap by calling HEAPIFY($H, 1$) (see Cormen, Leiserson and Rivest's book, page 143) to extend the heap property to the whole vector.

Algorithm R_BUILD_HEAP(i) implements the above strategy when the root of the (sub)-heap is stored in $H[i]$. Clearly, the entire heap is built by calling R_BUILD_HEAP(1).

```
R_BUILD_HEAP($i$)
if $2i \geq (N+1)/2$ then HEAPIFY($H, i$)
{ take care of last two levels }
return
R_BUILD_HEAP($2i$)
R_BUILD_HEAP($2i+1$)
HEAPIFY($H, i$)
return
```

The correctness of the above algorithm is easily shown by induction.

(b) Let $S(i)$ be the set of array indices storing the subheap rooted at index i, for any $i \geq 0$. We have $S(i) = \{i\} \cup S(2i) \cup S(2i+1)$, and $S(2i) \cap S(2i+1) = \emptyset$. Moreover $|S(2i)| = |S(2i+1)|$, whence $|S(2i)| = |S(2i+1)| = (|S(i)| - 1)/2$. As a consequence, the recursive calls will work on a set of indices whose size is less than half the size of the one associated with the outer call. This property is essential to guarantee that the algorithm is efficient. Also, note the size of any subheap H built by our recursive algorithm is a power of two minus one, therfore we can write the recurrence using the exponent as the parameter. Recalling that HEAPIFY takes time ck on a heap with $2^k - 1$ nodes, we have:

$$\begin{aligned} T(k) &= 2T(k-1) + ck, \ k > 1, \\ T(1) &= 1. \end{aligned}$$

(c) We have:

$$\begin{aligned} T(k) &= 2T(k-1) + ck \\ &= 2(2T(k-2) + c(k-1)) + ck \\ &\vdots \\ &= 2^i T(k-i) + c\sum_{j=0}^{i-1} 2^j (k-j) \\ &\vdots \\ &= 2^{k-1} + c\sum_{j=0}^{k-2} 2^j (k-j) \end{aligned}$$

We have:

$$\sum_{j=0}^{k-2} 2^j (k-j) = 3 \cdot 2^{k-1} - k - 2,$$

therefore, for any $k \geq 1$,

$$T(k) = (3c+1)2^{k-1} - c(k+2).$$

Note that $T(k) = O(2^k)$, therefore the algorithm is linear in the number of elements of the heap. □

Exercise 2.16 Consider an array storing the preorder sequence of a binary search tree of distinct elements.

(a) Describe an efficient recursive algorithm that determines if a key x belongs to the array.

(b) Evaluate the running time of the algorithm as a function of the number of nodes n and the depth d of the tree.

Answer:

(a) We devise an algorithm that "simulates" binary search on a binary search tree:

```
TREE_SEARCH($T, x$)
**if** $T =$ **nil then return** "not found"
**if** root($T$) $= x$ **then return** "found"
**if** $x <$ root($T$)
    **then return** TREE_SEARCH(left($T$), $x$)
    **else return** TREE_SEARCH(right($T$), $x$)
```

However, we are only given the preorder sequence $p(T)$ of T, stored in a vector A, therefore we have to find a way of identifying left and right subtrees as subsequences of $p(T)$.

It can be easily shown that for any tree T, $p(T)$ has the following recursive structure:

$$p(T) = \text{root}(T) \cdot p(\text{left}(T)) \cdot p(\text{right}(T)),$$

(note that $p(\text{left}(T))$ (resp., $p(\text{right}(T))$) is the empty string if left(T) (resp., right(T)) is empty). Therefore the preorder sequence of a non empty left subtree starts soon after the root, which is the first element of the sequence. In contrast, the preorder sequence of a non empty right subtree terminates the sequence. In order to completely identify the two sequences, we only have to determine the *starting point* in the array of $p(\text{right}(T))$. This can be accomplished efficiently by performing a binary search on the vector, using the root as the key:

```
FIND_RIGHT($h, k$, root)
{ $h$ and $k$ are the lower and upper bounds of the
subvector where the search takes place}
**if** $h > k$ **then return** $h$
middle $\leftarrow (h + k)$**div**2
**if** root $< A$[middle]
    **then return** FIND_RIGHT($h$, middle $- 1$, root)
    **else return** FIND_RIGHT(middle $+ 1, k$, root)
```

The correctness of the above procedure is due to the fact that all the elements in $p(\text{left}(T))$ (resp., $p(\text{right}(T))$) are smaller (resp., larger) than root(T).

We are now ready to give the required search procedure

```
ARRAY_SEARCH($i, j, x$)
**if** $i > j$ **then return** "not found" { empty subtree}
**if** $A[i] = x$ **then return** "found"
right $\leftarrow$ FIND_RIGHT($i + 1, j, A[i]$)
**if** $x < A[i]$
    **then return** ARRAY_SEARCH($i + 1$, right $- 1, x$)
    **else return** ARRAY_SEARCH(right, $j, x$)
```

(b) If d is the depth of the tree, ARRAY_SEARCH will perform at most d calls of procedure FIND_RIGHT. Each of such calls takes time $O(\log |h-k|) = O(\log n)$. Therefore the overall running time is $O(d \log n)$. Such algorithm is more efficient than a trivial scan of the array for depths as large as $O\left(\frac{n}{\log N}\right)$. □

Exercise 2.17 Assuming that n is a power of 4, solve the following recurrence:

$$\begin{cases} T(1) = 0 \\ T(2) = 2 \\ T(n) = T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + 3\frac{n}{2} \qquad (n \geq 4). \end{cases}$$

Answer: The above recurrence could stem from a divide-and-conquer algorithm that, on input an instance of size n, splits the instance into three subinstances of size $n/2$, $n/4$, and $n/4$, respectively, and recombines the results performing $3n/2$ work. We note that the sum of the sizes of the subinstances is n, and that the work is linear in such sum. This scenario is reminiscent of recurrences where the input instance is partitioned into a constant number k of subinstances of equal size n/k and the split/combination work is

$\Theta(n)$, i.e.,

$$T'(n) = kT'\left(\frac{n}{k}\right) + \Theta(n). \tag{2.12}$$

By applying the Master Theorem, we know that the above recurrence yields $T'(n) = \Theta(n \log n)$. Pursuing the analogy between our recurrence and Recurrence (1.18), let us try to prove by induction that there exists a constant $a > 0$ such that $T(n) \leq an \log n$. We have two base cases to cover, namely,

$$\begin{aligned} T(1) &= 0 \\ &\leq a \cdot 1 \cdot \log 1 = 0, \end{aligned}$$

which is true for any value of a, and

$$\begin{aligned} T(2) &= 2 \\ &\leq a \cdot 2 \cdot \log 2 = 2a, \end{aligned}$$

which is true if and only if $a \geq 1$. Assume that our statement holds for powers of four which are less than n. We then have:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + \frac{3}{2}n \\ &\leq a\frac{n}{2}\log\frac{n}{2} + 2a\frac{n}{4}\log\frac{n}{4} + \frac{3}{2}n \\ &= a\frac{n}{2}(\log n - 1) + a\frac{n}{2}(\log n - 2) + \frac{3}{2}n \\ &= an\log n - \frac{3}{2}an + \frac{3}{2}n. \end{aligned}$$

The thesis will follow if we make sure that

$$-\frac{3}{2}an + \frac{3}{2}n \leq 0,$$

for which again it suffices to choose $a \geq 1$. In conclusion, by picking $a = 1$, we have proved that

$$T(n) \leq n \log n. \tag{2.13}$$

By conducting a completely symmetric reasoning to the above, we can show that $T(n) \geq bn \log n$ for any constant $b \leq 1$. By picking $b = 1$ we then have:

$$T(n) \geq n \log n. \tag{2.14}$$

By combining (1.19) and (1.20) we can then conclude that

$$T(n) = n \log n,$$

for any value of $n \geq 1$. Note that we obtained the *exact* solution to our recurrence by providing perfectly matching lower and upper bounds. Clearly, this is not always the case, and for other recurrences in this class the best we can do is to determine the order of magnitude of their solution. □

Exercise 2.18

(a) Let M be a square matrix with $N = 4^k$ elements. Prove that, if A, B and C are $\frac{\sqrt{N}}{2} \times \frac{\sqrt{N}}{2}$, the following relation holds between M and its inverse M^{-1}:

$$M = \begin{bmatrix} A & B \\ 0 & C \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} A^{-1} & -A^{-1}BC^{-1} \\ 0 & C^{-1} \end{bmatrix}.$$

(b) Develop a divide-and-conquer algorithm for the inversion of an upper triangular matrix (all the elements below the main diagonal are null).

(c) Write the recurrence relation for $T(N)$, defined as the number of scalar multiplications performed by the algorithm of Part (b). Assume that the multiplication of square matrices is done according to the definition.

(d) Solve the recurrence of Part (c).

Exercise 2.19 Consider an algorithm A_1 solving a computational problem Π in time $T_1(n) = n \log n$. Suppose we can devise for the same problem a divide-and-conquer strategy that, for $n > 1$, generates two instances of Π of size $n/2$ with work $w(n) = a$, where a is a constant greater than 4. Let n be a power of 2.

(a) Evaluate the running time of an algorithm A_2 based on the divide and conquer strategy, under the assumption that $T_2(1) = 0$.

(b) Consider an algorithm A using the divide-and-conquer strategy for $n > n_0$ and calling instead A_1 for $n \leq n_0$. Determine the running time $T(n)$ of A (note that $T(n)$ will be a function of n_0).

(c) Find the value of n_0 that minimizes $T(n)$ (note that n_0 will be a function of a).

Exercise 2.20 Let $f(n) = \log n$, and define, for any $n \geq 2$, $\log^\star n = f^\star(n, 1)$. Show that $\log^\star n = o(\log\log n)$.

Exercise 2.21 Solve the following recurrence when the parameter n is a double power of two (i.e., $n = 2^{2^i}$, for some $i \geq 0$).

$$\begin{cases} T(n) = nT\left(\sqrt{n}\right) & n > 2 \\ T(2) = 4. \end{cases}$$

Exercise 2.22 Consider the following recurrence:

$$T(n) = \begin{cases} T_0 & n \leq n_0, \\ \sum_{i=1}^{k} a_i T\left(\lfloor c_i n \rfloor\right) + cn & n > n_0, \end{cases}$$

where T_0, n_0, k, $a_1, a_2, \ldots a_k$ are positive integer constants, and $c_1, c_2, \ldots, c_k$ are positive real constants less than one. Prove that if $\sum_{i=1}^{k} a_i c_i < 1$ then $T(n) = O(n)$.

Chapter 3

Convolutions and the Discrete Fourier Transform

3.1 Linear and Cyclic Convolution

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two arbitrary vectors of n components. The *linear convolution* of $\boldsymbol{a}$ and $\boldsymbol{b}$, denoted $\boldsymbol{w} = \boldsymbol{a} \star \boldsymbol{b}$, is a vector of $2n-1$ components such that, for $0 \leq i \leq 2n-1$,

$$w_i = \sum_{j=0}^{i} a_j b_{i-j}. \tag{3.1}$$

Recall that in the above definition we are implicitly assuming that $a_k, b_k = 0$ for all indices k such that $k > n-1$. If we want to be rid of such assumption, then we can write, for $0 \leq i \leq 2n-1$,

$$w_i = \sum_{j=\max\{0,i-n+1\}}^{\min\{i,n-1\}} a_j b_{i-j}, \tag{3.2}$$

where the bounds in the summation are chosen in such a way that the indices j and $i-j$ always range between 0 and $n-1$. Recall that w_i is the i-th coefficient of the polynomial of degree bound $2n$ which is obtained by multiplying the two polynomials whose coefficient representations are $\boldsymbol{a}$ and $\boldsymbol{b}$. By evaluating the polynomials on the $2n$ $2n$-th roots of unity, pointwise-multiplying the values and interpolating from the resulting vector, we obtain the following relation:

$$\boldsymbol{w} = DFT_{2n}^{-1}\left(DFT_{2n}(\boldsymbol{a}|\boldsymbol{0}) \cdot DFT_{2n}(\boldsymbol{b}|\boldsymbol{0})\right), \tag{3.3}$$

where $\cdot$ denotes component-wise product, and $(\boldsymbol{a}|\boldsymbol{0})$, $(\boldsymbol{b}|\boldsymbol{0})$ are obtained by padding $\boldsymbol{a}$ and $\boldsymbol{b}$ with n zeroes. Equation (2.3) gives us a way to compute $\boldsymbol{w}$ in $\Theta(n \log n)$ time by applying the FFT algorithm.

Let us now introduce a new vector operator. For $\boldsymbol{a}$ and $\boldsymbol{b}$, vectors of n components, we define the *cyclic convolution* (also called *wrapped convolution*) of $\boldsymbol{a}$ and $\boldsymbol{b}$, denoted $\boldsymbol{a} \circledast \boldsymbol{b}$, as a vector $\boldsymbol{z}$ of n components such that, for $0 \leq i \leq n-1$,

$$z_i = \sum_{j=0}^{n-1} a_j b_{(i-j) \bmod n}. \tag{3.4}$$

Note the similarity between (2.1) and (2.4). However, recall that if $\boldsymbol{a}$ and $\boldsymbol{b}$ have n components, then $\boldsymbol{a} \star \boldsymbol{b}$ has $2n-1$ components, while $\boldsymbol{a} \circledast \boldsymbol{b}$ has only n components.

Cyclic convolution can be thought of as a "wrapped" version of linear convolution. To see this, note that, for $0 \leq i \leq n-1$,

$$z_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}. \tag{3.5}$$

Therefore, to compute z_i, we start by multiplying, a_0 by b_i, then a_1 by b_{i-1} ... until we multiply a_i by b_0. (This is exactly what we do when computing the linear convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$.) Then, we proceed by "wrapping around" vector $\boldsymbol{b}$ and multiplying a_{i+1} by b_{n-1}, a_{i+2} by b_{n-2} and so forth.

Observe the multiplication pattern of the a_i's and the b_j's in cyclic convolution:

$$\begin{array}{ccccccccccc}
z_0 & = & a_0b_0 & + & a_1b_{n-1} & + & \ldots & + & a_{n-2}b_2 & + & a_{n-1}b_1 \\
z_1 & = & a_0b_1 & + & a_1b_0 & + & \ldots & + & a_{n-2}b_3 & + & a_{n-1}b_2 \\
\vdots & & & & & & & & & & \\
z_i & = & a_0b_i & + & a_1b_{i-1} & + & \ldots & + & a_{n-2}b_{i+2} & + & a_{n-1}b_{i+1} \\
\vdots & & & & & & & & & & \\
z_{n-1} & = & a_0b_{n-1} & + & a_1b_{n-2} & + & \ldots & + & a_{n-2}b_1 & + & a_{n-1}b_0
\end{array}$$

We can write the above system as $\boldsymbol{w} = C(\boldsymbol{b}) \times \boldsymbol{a}$, where

$$C(\boldsymbol{b}) = \begin{bmatrix}
b_0 & b_{n-1} & \ldots & b_2 & b_1 \\
b_1 & b_0 & \ldots & b_3 & b_2 \\
\vdots & \vdots & \ldots & \vdots & \vdots \\
b_i & b_{i-1} & \ldots & b_{i+2} & b_{i+1} \\
\vdots & \vdots & \ldots & \vdots & \vdots \\
b_{n-1} & b_{n-2} & \ldots & b_1 & b_0
\end{bmatrix}$$

Note that the columns of $C(\boldsymbol{b})$ are obtained as consecutive *cyclic right shifts* of $\boldsymbol{b}$. Matrix $C(\boldsymbol{b})$ is called a *circulant* matrix with first column $\boldsymbol{b}$. A circulant matrix admits a very compact representation, since the matrix is uniquely specified by its first column.

Note that computing $\boldsymbol{w} = \boldsymbol{a} \circledast \boldsymbol{b}$ according to the definition takes $\Theta(n^2)$ time. However, it turns out that we can use the FFT to compute cyclic convolutions in $\Theta(n \log n)$ time. We have:

Theorem 3.1 (Cyclic Convolution Theorem) *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two arbitrary vectors of n components, and let $\boldsymbol{z} = \boldsymbol{a} \circledast \boldsymbol{b}$. Then*

$$\boldsymbol{z} = DFT_n^{-1}\left(DFT_n(\boldsymbol{a}) \cdot DFT_n(\boldsymbol{b})\right), \tag{3.6}$$

where $\cdot$ denotes component-wise product.

Therefore, simply computing the DFT's of $\boldsymbol{a}$ and $\boldsymbol{b}$ *with no padding*, multiplying their components and then taking the inverse DFT gives us the cyclic convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$.

Proof: It suffices to show that $DFT_n(\boldsymbol{z}) = DFT_n(\boldsymbol{a}) \cdot DFT_n(\boldsymbol{b})$. We have:

$$(DFT_n(\boldsymbol{a}))_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \quad \text{and} \quad (DFT_n(\boldsymbol{b}))_i = \sum_{k=0}^{n-1} b_k \omega_n^{ik},$$

therefore

$$(DFT_n(\boldsymbol{a}))_i \cdot (DFT_n(\boldsymbol{b}))_i = \left(\sum_{j=0}^{n-1} a_j \omega_n^{ij}\right)\left(\sum_{k=0}^{n-1} b_k \omega_n^{ik}\right) = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_j b_k \omega_n^{i(j+k)}. \tag{3.7}$$

On the other hand, from Equation (2.5) we have:

$$\begin{aligned}(DFT_n(\boldsymbol{z}))_i &= \sum_{p=0}^{n-1}\left(\sum_{s=0}^{p} a_s b_{p-s} + \sum_{s=p+1}^{n-1} a_s b_{n+p-s}\right)\omega_n^{ip} \\ &= \sum_{p=0}^{n-1}\sum_{s=0}^{p} a_s b_{p-s}\omega_n^{ip} + \sum_{p=0}^{n-1}\sum_{s=p+1}^{n-1} a_s b_{n+p-s}\omega_n^{ip}. \end{aligned} \tag{3.8}$$

Let us now show that, for any $0 \le i \le n-1$, Formulae (2.7) and (2.8) coincide. For this purpose, it suffices to note that each term $a_\ell b_m$, with $0 \le \ell, m \le n-1$, appears only once in (2.7), multiplied by $\omega_n^{i(\ell+m)}$. In (2.8), if $\ell + m < n$, then $a_\ell b_m$ appears only once in the first double summation, for $p = \ell + m$ and $s = \ell$, and multiplied by $\omega_n^{ip} = \omega_n^{i(\ell+m)}$. If $\ell + m \ge n$, then $a_\ell b_m$ appears only once in the second double summation, for $p = \ell + m - n$ and $s = \ell$, and multiplied by $\omega_n^{ip} = \omega_n^{i(\ell+m-n)} = \omega_n^{i(\ell+m)}\omega_n^{-in} = \omega_n^{i(\ell+m)}$. The theorem follows. $\square$

The following is an immediate corollary of Equation (2.3) and Theorem 2.1.

Corollary 3.1 *Let* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *be two arbitrary vectors of* n *components. Then*

$$\boldsymbol{a} \star \boldsymbol{b} = (\boldsymbol{a}|\mathbf{0}) \circledast (\boldsymbol{b}|\mathbf{0}),$$

where $(\boldsymbol{a}|\mathbf{0})$ *and* $(\boldsymbol{b}|\mathbf{0})$ *are vectors of* $2n$ *components.*

Corollary 2.1 gives us a way to compute $\boldsymbol{w} = \boldsymbol{a} \star \boldsymbol{b}$ through a cyclic convolution. In what follows, we investigate the inverse relation: in particular, we will write $\boldsymbol{z} = \boldsymbol{a} \circledast \boldsymbol{b}$ as a function of $\boldsymbol{w}$.

Recall that

$$\begin{aligned} z_i &= \sum_{j=0}^{i} x_j y_{i-j} + \sum_{j=i+1}^{n-1} x_j y_{n+i-j} \\ &= z_i^1 + z_i^2, \end{aligned} \tag{3.9}$$

for $0 \leq i \leq n-1$. Note that $z_{n-1}^2 = 0$. By making the bounds in the summation explicit, we can rewrite (2.2) as

$$w_i = \begin{cases} \sum_{j=0}^{i} x_j y_{i-j}, & \text{for } 0 \leq i \leq n-1, \\ \\ \sum_{j=i-n+1}^{n-1} x_j y_{i-j}, & \text{for } n \leq i \leq 2n-2. \end{cases} \tag{3.10}$$

From (2.9) and (2.10) it immediately follows that $z_{n-1}^1 = w_{n-1}$, therefore $z_{n-1} = w_{n-1}$. For $0 \leq i \leq n-2$ we have:

$$\begin{aligned} z_i^1 &= w_i \\ \\ z_i^2 &= \sum_{j=i+1}^{n-1} x_j y_{n+i-j} \\ &= \sum_{\substack{j\ =\ (i+n)- \\ -n+1}}^{n-1} x_j y_{(i+n)-j} \\ &= w_{i+n}. \end{aligned}$$

Therefore, for $0 \leq i \leq n-2$, $z_i = w_i + w_{i+n}$.

Exercise 3.1 The complex number $\omega = e^{i\pi/4} = \frac{\sqrt{2}}{2} + \frac{i\sqrt{2}}{2}$ is an 8-th principal root of unity in the complex field.

(a) Compute ω^i for $i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

(b) Write the Fourier transform matrix $F_8 = [\omega^{ij}]$ for $i, j = 0, 1, \ldots, 7$.

(c) Write $F_8^{-1} = (1/8)[\omega^{-ij}]$ for $i, j = 0, 1, \cdots, 7$.

(d) Compute $\boldsymbol{X} = F_8\boldsymbol{x}$, for $\boldsymbol{x} = (1, 0, 1, -1, 0, 0, -1, 1)$.

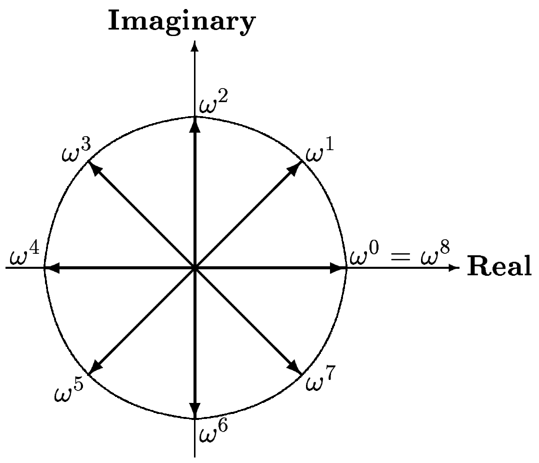
(e) Let $\boldsymbol{x} = (0, 0, 1, 0, 0, 0, 0, 0)$ and $\boldsymbol{y} = (0, 0, 0, 0, 1, 0, 0, 0)$. Compute the cyclic convolution $\boldsymbol{z} = \boldsymbol{x} \circledast \boldsymbol{y}$ using the definition.

(f) Repeat (e) using the cyclic convolution theorem.

Answer:

(a) Observe that, since $\omega^0 = \omega^8 = 1$, $\omega^r = \omega^{r \bmod 8}$. Let $d = \frac{\sqrt{2}}{2}$. We have:

$\omega^0 = 1 \qquad \omega^1 = (d + di)$
$\omega^2 = (d + di)^2 = d^2 + 2\frac{1}{2}i - d^2 = i$
$\omega^3 = \omega^2(d + di) = di + di^2 = -d + di$
$\omega^4 = \omega^3(d + di) = -d^2 + (di)^2 = -1$
$\omega^5 = \omega^4(d + di) = -d - di$
$\omega^6 = \omega^5(d + di) = -d^2 - 2d^2 i -$
$\qquad -(di)^2 = -i$
$\omega^7 = \omega^6(d + di) = -di - di^2 = d - di$
$\omega^8 = \omega^7(d + di) = d^2 - (di)^2 = 1$
$\omega^9 = \omega^8(d + di) = d + di = \omega^1$
$\ldots$



(b)

$$F_8 = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} & \omega^{12} & \omega^{14} \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} & \omega^{18} & \omega^{21} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} & \omega^{24} & \omega^{28} \\ \omega^0 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25} & \omega^{30} & \omega^{35} \\ \omega^0 & \omega^6 & \omega^{12} & \omega^{18} & \omega^{24} & \omega^{30} & \omega^{36} & \omega^{42} \\ \omega^0 & \omega^7 & \omega^{14} & \omega^{21} & \omega^{28} & \omega^{35} & \omega^{42} & \omega^{49} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & (d+di) & i & (-d+di) & -1 & (-d-di) & -i & (d-di) \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & (-d+di) & -i & (d+di) & -1 & (d-di) & i & (-d-di) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -(d-di) & i & (d-di) & -1 & (d+di) & -i & (-d+di) \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & (d-di) & -i & (-d-di) & -1 & (-d+di) & i & (d+di) \end{bmatrix}$$

(c)

$$8F_8^{-1} = \begin{bmatrix} \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} & \omega^{-0} \\ \omega^{-0} & \omega^{-1} & \omega^{-2} & \omega^{-3} & \omega^{-4} & \omega^{-5} & \omega^{-6} & \omega^{-7} \\ \omega^{-0} & \omega^{-2} & \omega^{-4} & \omega^{-6} & \omega^{-8} & \omega^{-10} & \omega^{-12} & \omega^{-14} \\ \omega^{-0} & \omega^{-3} & \omega^{-6} & \omega^{-9} & \omega^{-12} & \omega^{-15} & \omega^{-18} & \omega^{-21} \\ \omega^{-0} & \omega^{-4} & \omega^{-8} & \omega^{-12} & \omega^{-16} & \omega^{-20} & \omega^{-24} & \omega^{-28} \\ \omega^{-0} & \omega^{-5} & \omega^{-10} & \omega^{-15} & \omega^{-20} & \omega^{-25} & \omega^{-30} & \omega^{-35} \\ \omega^{-0} & \omega^{-6} & \omega^{-12} & \omega^{-18} & \omega^{-24} & \omega^{-30} & \omega^{-36} & \omega^{-42} \\ \omega^{-0} & \omega^{-7} & \omega^{-14} & \omega^{-21} & \omega^{-28} & \omega^{-35} & \omega^{-42} & \omega^{-49} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & (d-di) & -i & (-d-di) & -1 & (-d+di) & i & (d+di) \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & (-d-di) & i & (d-di) & -1 & (d+di) & -i & (-d+di) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -(d-di) & -i & (d+di) & -1 & (d-di) & i & (-d-di) \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & (d+di) & i & (-d+di) & -1 & (-d-di) & -i & (d-di) \end{bmatrix}$$

(d)

$$\boldsymbol{X} = F_8 \cdot \boldsymbol{x} = F_8 \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1+0+1-1+0+0-1+1 \\ 1+0+i-(-d+di)+0+0-(-i)+(d-di) \\ 1+0+(-1)-(-i)+0+0-(-1)+(-i) \\ 1+0+(-i)-(d+di)+0+0-i+(-d-di) \\ 1+0+1-(-1)+0+0-1+(-1) \\ 1+0+i-(d-di)+0+0-(-i)+(-d+di) \\ 1+0+(-1)-i+0+0-(-1)+i \\ 1+0+(-i)-(-d-di)+0+0-i+(d+di) \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ (1+2d)+(2-2d)i \\ 1 \\ (1-2d)+(-2-2d)i \\ 1 \\ (1-2d)+(2+2d)i \\ 1 \\ (1+2d)+(-2+2d)i \end{bmatrix}$$

(e) Let $n = 8$. Since only x_2 and y_4 are nonzero, the only way $x_l y_{(i-l) \bmod n}$ can be nonzero is when $l = 2$ and $i = 6$ (we get the value of l directly; for i, from $(i - l) \bmod n = 4$, we get $(i - 2) \bmod n = 4$, which implies $i = 6$). Thus, $z_6 = x_2 y_4 = 1$, while all the other z_i's are zero.

(f) First, we note that, since $\boldsymbol{x}$ and $\boldsymbol{y}$ have only one nonzero element each, and that the element is a 1, $F_8\boldsymbol{x}$ is column 2 of F_8 and $F_8\boldsymbol{y}$ is column 4 of F_8. Now, from the definition of F_8, we know that $[F_8]_{ij} = \omega^{i \cdot j}$; thus, the i-th element of $F_8\boldsymbol{x} \cdot F_8\boldsymbol{y}$ is $\omega^{i \cdot 2 + i \cdot 4} = \omega^{i \cdot 6}$. But this is exactly column 6 of F_8. Therefore, multiplying F_8^{-1} by $F_8\boldsymbol{x} \cdot F_8\boldsymbol{y}$ we obtain the column 6 of the identity matrix, namely $(0\,0\,0\,0\,0\,0\,1\,0)$, exactly the answer in Part (e). □

Exercise 3.2 Given a vector $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$, the *circulant* matrix $C(\boldsymbol{x})$ is an $n \times n$ matrix whose first column is $\boldsymbol{x}$, while the remaining columns are obtained as consecutive cyclic right shifts of $\boldsymbol{x}$. Design and analyze efficient algorithms for the following problems:

(a) Determining the product of two circulant matrices.

(b) Determining a solution (if any) to the linear system $C(\boldsymbol{x})\boldsymbol{y} = \boldsymbol{b}$.

(c) Determining whether $C(\boldsymbol{x})$ is invertible and, if so, computing its inverse.

Answer: Note that

$$C(\boldsymbol{x}) = \begin{bmatrix} x_0 & x_{n-1} & \cdots & x_2 & x_1 \\ x_1 & x_0 & \cdots & x_3 & x_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n-2} & x_{n-3} & \cdots & x_0 & x_{n-1} \\ x_{n-1} & x_{n-2} & \cdots & x_1 & x_0 \end{bmatrix},$$

therefore $[C(\boldsymbol{x})]_{ij} = x_{(i-j) \bmod n}$, for $0 \le i, j \le n-1$.

(a) Let $C(\boldsymbol{x})$ and $C(\boldsymbol{y})$ be two circulant matrices. By the definition of row-by-column product we have, for $0 \leq i, j \leq n-1$,

$$\begin{aligned}
[C(\boldsymbol{x}) \times C(\boldsymbol{y})]_{ij} &= \sum_{s=0}^{n-1} [C(\boldsymbol{x})]_{is} [C(\boldsymbol{y})]_{sj} \\
&= \sum_{s=0}^{n-1} x_{(i-s) \bmod n} y_{(s-j) \bmod n}.
\end{aligned}$$

Let $k = (i-s) \bmod n$. Note that when s varies between 0 and $n-1$, so does k. Moreover, $s = (i-k) \bmod n$. By substituting s with k in the above summation we obtain:

$$\begin{aligned}
[C(\boldsymbol{x}) \times C(\boldsymbol{y})]_{ij} &= \sum_{k=0}^{n-1} x_k y_{((i-k) \bmod n - j) \bmod n} \\
&= \sum_{k=0}^{n-1} x_k y_{((i-j) \bmod n - k) \bmod n} \\
&= (\boldsymbol{x} \circledast \boldsymbol{y})_{(i-j) \bmod n}.
\end{aligned}$$

This suffices to show that the product of $C(\boldsymbol{x})$ and $C(\boldsymbol{y})$ yields a circulant matrix $C(\boldsymbol{z})$, with $\boldsymbol{z} = \boldsymbol{x} \circledast \boldsymbol{y}$. If circulant matrices are represented by storing only their first column, then the representation of their product can be computed in $O(n \log n)$ time using the FFT algorithm.

(b) Consider the system

$$C(\boldsymbol{x}) \times \boldsymbol{y} = \boldsymbol{b}, \tag{3.11}$$

where $\boldsymbol{x}$ and $\boldsymbol{b}$ are arbitrary complex vectors and $\boldsymbol{y}$ is the vector of the unknowns. Observe that

$$\begin{aligned}
[C(\boldsymbol{x}) \times \boldsymbol{y}]_i &= \sum_{k=0}^{n-1} x_{(i-k) \bmod n} y_k \\
&= (\boldsymbol{y} \circledast \boldsymbol{x})_i.
\end{aligned}$$

Let F_n be the Fourier matrix of order n, and let $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{B}$ denote, respectively, $F_n\boldsymbol{x}$, $F_n\boldsymbol{y}$ and $F_n\boldsymbol{b}$. By the cyclic convolution theorem, System 2.11 is equivalent to the following system

$$\boldsymbol{X} \cdot \boldsymbol{Y} = \boldsymbol{B},$$

where $\cdot$ denotes component-wise product. Note that such system consists of n equations, one for each component of the (unknown) vector $\boldsymbol{Y}$. For $0 \le i \le n-1$, the i-th equation is

$$X_i Y_i = B_i, \tag{3.12}$$

hence it contains the single unknown Y_i. Therefore it immediately follows that

1. The system has one and only solution iff $X_i \neq 0$, for $0 \le i \le n-1$.
2. The system has no solution iff there exists an index i such that $X_i = 0$ and $B_i \neq 0$.
3. The system has infinite solutions iff for each index i such that $X_i = 0$, then $B_i = 0$, and at least one such index exists.

By the equivalence of Systems 2.12 and 2.11, the above considerations also apply to our original system. Note that $\boldsymbol{X}$ and $\boldsymbol{B}$ can be computed in $O(n \log n)$ time using the FFT algorithm, and that the subsequent test (as specified in Points 1..3 above) can be performed in additional $O(n)$ time.

Consider the case when at least one solution exists and define $\overline{\boldsymbol{Y}}$ as

$$\overline{Y}_i = \begin{cases} B_i/X_i & \text{if } X_i \neq 0 \\ 0 & \text{otherwise,} \end{cases}$$

for $0 \le i \le n-1$. Then, a solution to System 2.11 can be computed in $O(n \log n)$ time as $\overline{\boldsymbol{y}} = F_n^{-1}\overline{\boldsymbol{Y}}$.

(c) By a well known theorem in linear algebra, $C(\boldsymbol{x})$ is invertible if and only if the linear system

$$C(\boldsymbol{x})\boldsymbol{y} = \boldsymbol{b}$$

has one and only solution $\boldsymbol{y} \in C^n$, for any given vector $\boldsymbol{b} \in C^n$. By the results in Part (b), we can therefore conclude that $C(\boldsymbol{x})$ is invertible if and only if $X_i = (F_n\boldsymbol{x})_i \neq 0$ for $0 \le i \le n-1$. Such condition can clearly be tested in $O(n \log n)$ time using the FFT algorithm to compute $\boldsymbol{X}$ and a linear scan to test that $X_i \neq 0$, for $0 \le i \le n-1$.

Let us now prove that $[C(\boldsymbol{x})]^{-1}$ is itself a circulant matrix. To see this, consider the system

$$[C(\boldsymbol{x})]\boldsymbol{y} = (1, 0, \ldots, 0). \tag{3.13}$$

Since $C(\boldsymbol{x})$ is invertible, System (2.13) has one and only solution $\overline{\boldsymbol{y}}$ such that

$$[C(\boldsymbol{x})]\overline{\boldsymbol{y}} = \overline{\boldsymbol{y}} \circledast \boldsymbol{x} = (1, 0, \ldots, 0).$$

Consider now the circulant matrix $C(\overline{\boldsymbol{y}})$. We have

$$C(\boldsymbol{x}) \times C(\overline{\boldsymbol{y}}) = C\left(\boldsymbol{x} \circledast \overline{\boldsymbol{y}}\right) = C(\ (1, 0, \ldots, 0)\ ) = I_n,$$

where I_n is the $n \times n$ identity matrix. Therefore

$$[C(\boldsymbol{x})]^{-1} = C(\overline{\boldsymbol{y}})$$

by the uniqueness of the inverse matrix. As shown in Part (b), System (2.13) can be solved in time $O(n \log n)$, therefore the inverse of $C(\boldsymbol{x})$ can be computed within the same time bound. □

Exercise 3.3 Given the sequence

$$\boldsymbol{a} = \left(a_{-(n-1)}, a_{-(n-2)}, \ldots, a_{-1}, a_0, a_1, \ldots, a_{n-2}, a_{n-1}\right),$$

the *Toeplitz* matrix $M = \mathrm{T}(\boldsymbol{a})$ is an $n \times n$ matrix of elements $m_{ij} = a_{i-j}$, for $i, j = 0, 1, \ldots, n-1$.

(a) Let $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$. Describe an algorithm which computes $M\boldsymbol{x}$ in $O(n \log n)$ time.

(b) Based on Part (a), give an upper bound on the cost of multiplying two Toeplitz matrices.

Answer:

(a) We have

$$M = \mathrm{T}(\boldsymbol{a}) = \begin{bmatrix} a_0 & a_{-1} & \cdots & a_{-(n-2)} & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \cdots & a_{-(n-2)} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n-2} & \cdots & \ddots & \ddots & a_{-1} \\ a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{bmatrix}.$$

Observe that all the elements on a fixed diagonal are the same. By the definition of matrix-vector product we obtain:

$$(M\boldsymbol{x})_i = \sum_{k=0}^{n-1} m_{ik} x_k = \sum_{k=0}^{n-1} a_{i-k} x_k, \text{ for } i = 0, 1, \ldots, n-1.$$

Note that the i-th element of the product is obtained by summing all the products $a_r x_s$ with $r + s = i$. This fact immediately reminds us of convolution between $\boldsymbol{a}$ and $\boldsymbol{x}$. To simplify the computation of indices, let us rename $\boldsymbol{a}$ as $\overline{\boldsymbol{a}} = (\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{2n-2})$. Note that $\bar{a}_j = a_{-(n-1)+j}$, for $0 \le j \le 2n-2$. The linear convolution $\boldsymbol{c} = \overline{\boldsymbol{a}} \star \boldsymbol{x}$ will have $3n-2$ components, namely

$$c_j = \sum_{k=\max\{0,j-2n+2\}}^{\min\{j,n-1\}} \bar{a}_{j-k} x_k, \text{ for } j = 0, 1, \ldots, 3n-3.$$

For $n - 1 \le j \le 2n-2$, we have

$$c_j = \sum_{k=0}^{n-1} \bar{a}_{j-k} x_k = \sum_{k=0}^{n-1} a_{-(n-1)+j-k} x_k.$$

Let $i = -(n-1) + j$ in the above formula. Note that when j varies between $n-1$ and $2n-2$, i varies between 0 and $n-1$. We have

$$c_{i+(n-1)} = \sum_{k=0}^{n-1} a_{i-k} x_k = (M\boldsymbol{x})_i. \tag{3.14}$$

Equation (2.14) establishes the required relation between Toeplitz matrix - vector multiplication and linear convolution. Since $\overline{\boldsymbol{a}} \star \boldsymbol{x}$ is an $O(n \log n)$ operation, we obtain a multiplication algorithm whose running time is within the required bound.

(b) The product $C = A \times B$ of two $n \times n$ matrices can be seen as n matrix-vector multiplications between A and each column of B, with any such product providing a distinct column of C. By employing the algorithm developed in Part (a), we can perform matrix-matrix multiplication in $O(n^2 \log n)$ time, provided that A is a Toeplitz matrix (note that no restriction on B is necessary). $\square$

Exercise 3.4 Design and analyze an efficient algorithm to compute the k-th power of a polynomial $p(x)$ of degree-bound n.

Answer: Let $\odot$ be any associative operation defined on a semiring R. For any $x \in R$, the problem of computing

$$x^{(k)} = \overbrace{x \odot x \odot \cdots \odot x}^{k \text{ times}}$$

can be approached as follows. Define $d = \lfloor \log k \rfloor + 1$ (d is the number of binary digits needed to represent k), and let $k_0, k_1, \ldots, k_{d-1}$ be such that

$$k = \sum_{i=0}^{d-1} k_i 2^i, \text{ with } k_i \in \{0,1\}, k_{d-1} = 1.$$

Then

$$x^{(k)} = x^{(k_0 2^0)} \odot x^{(k_1 2^1)} \odot x^{(k_2 2^2)} \odot \cdots \odot x^{(k_{d-1} 2^{d-1})}.$$

These considerations suggest the following algorithm to compute $x^{(k)}$:

```
EXPONENTIATE($x, k$)
$d \leftarrow \lfloor \log k \rfloor + 1$
$h \leftarrow k$
$z \leftarrow e_\odot$ {$e_\odot$ is the identity element w.r.t. $\odot$ in $R$}
$y \leftarrow x$
for $i \leftarrow 0$ to $d-1$
    do $ki \leftarrow h$ mod 2 {during the $i$-th iteration, $ki = k_i$}
    $h \leftarrow h$ div 2 {now, $h$ has binary representation $k_{d-1} \ldots k_{i+1}$}
    if $ki = 1$ then $z \leftarrow z \odot y$
    $y \leftarrow y \odot y$ {$y = x^{(2^{i+1})}$}
return $z$
```

Let $\boldsymbol{a}$ be the coefficient representation of $p(x)$. The above algorithm applies to our case when $x = \boldsymbol{a}$ and $\odot = \star$, that is, $\odot$ represents linear convolution between two vectors of size n_1 and n_2, for any $n_1, n_2 > 0$ (to see that linear convolution is an associative operation, think of it as performing polynomial multiplication). In order to analyze the time complexity of EXPONENTIATE in this case, we have to take into account the growth in size of the vectors resulting from applying the operator $\star$. We have the following facts:

1. Since $x^{(2^i)} = x^{(2^{i-1})} \star x^{(2^{i-1})}$, we have that $\text{size}\left(x^{(2^i)}\right) < 2 \cdot \text{size}\left(x^{(2^{i-1})}\right)$, for $i \geq 1$, while $\text{size}\left(x^{(2^0)}\right) = \text{size}(x) = n$. By unfolding the recurrence, we have that at the beginning of the i-th iteration, $i \geq 0$,

$$\text{size}(y) = \text{size}\left(x^{(2^i)}\right) \leq 2^i n.$$

2. Note that $e_\star$ is the scalar 1, therefore, at the beginning of the 0-th iteration, $\text{size}(z) = 1$. For $i \geq 1$, at the beginning of the i-th iteration we have that

$$\text{size}(z) \leq \sum_{j=0}^{i-1} \text{size}\left(x^{(2^j)}\right) \leq n \sum_{j=0}^{i-1} 2^j < 2^i n.$$

Therefore, the bulk of the work done during the i-th iteration of the algorithm consists of computing two convolutions on vectors of size $O(2^i n)$. Since linear convolution can be computed in time $O(m \log m)$ on inputs of size m, it follows that the i-th iteration of EXPONENTIATE takes time

$$\begin{aligned} T_i &= O\left(n2^i \log(n2^i)\right) \\ &= O\left((n \log n)2^i + ni2^i\right). \end{aligned}$$

Recalling that $d = \lfloor \log k \rfloor + 1$, the overall time complexity is

$$\begin{aligned} T &= \sum_{i=0}^{d-1} T_i \\ &= O\left(n2^d \log n + n2^d d\right) \\ &= O\left(nk \log(nk)\right). \end{aligned}$$

□

Exercise 3.5 (The Cooley-Tukey FFT Algorithm) Let $n = pq$, with $p, q > 1$. Given a vector $\boldsymbol{x}$ of size n, the *Cooley-Tukey FFT algorithm* computes $DFT_n(\boldsymbol{x}) = F_n\boldsymbol{x}$ by performing the following five steps:

1. Arrange $\boldsymbol{x}$ into a $p \times q$ matrix, in row major order;

2. For $0 \leq j \leq q-1$, substitute column X^j of the resulting matrix with $DFT_p(X^j) = F_pX^j$;

3. For $0 \leq i \leq p-1$, $0 \leq j \leq q-1$, multiply the (i,j)-th entry of the matrix by ω_{pq}^{ij};

4. For $0 \leq i \leq p-1$ substitute row X_i of the resulting matrix with $DFT_q(X_i) = F_qX_i$;

5. Read out $DFT_n(\boldsymbol{x})$ by enumerating the entries of the resulting matrix in colum-major order.

Let $n = 12$, $p = 3$, $q = 4$, and let $\boldsymbol{x} = (0,0,1,1,1,0,1,0,0,0,1,1)$. Compute $DFT_{12}(\boldsymbol{x})$ by applying the Cooley-Tukey algorithm and showing the array at the end of each step.

Answer: Let $\omega = \omega_{12} = e^{\pi i/6} = \sqrt{3}/2 + i/2$. On input $\boldsymbol{x} = (0,0,1,1,1,0,1,0,0,0,1,1)$, the Cooley-Tukey algorithm executes as follows.

Step 1: Arrange $\boldsymbol{x}$ into a 3×4 matrix, in row major order. We obtain:

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

Step 2: Transform columns. We must replace each column X^j with F_3X^j, where F_3 is based on the third root $\omega_3 = \omega^4 = -1/2 + (\sqrt{3}/2)i$. The result is:

$$X = \begin{bmatrix} 1 & 0 & 3 & 2 \\ -\frac{1}{2} + \frac{\sqrt{3}}{2}i & 0 & 0 & \frac{1}{2} - \frac{\sqrt{3}}{2}i \\ -\frac{1}{2} - \frac{\sqrt{3}}{2}i & 0 & 0 & \frac{1}{2} + \frac{\sqrt{3}}{2}i \end{bmatrix}.$$

Step 3: Multiply entry (i,j) by ω^{ij}. Note that the first column and the first row will be left unchanged (multiplied by $\omega^0 = 1$), and there are several 0-entries in the matrix. In fact, we only need to compute $\omega^3 = e^{\pi i/2} = i$ and $\omega^6 = e^{\pi i} = -1$. The array has now become:

$$X = \begin{bmatrix} 1 & 0 & 3 & 2 \\ -\frac{1}{2} + \frac{\sqrt{3}}{2}i & 0 & 0 & \frac{\sqrt{3}}{2} + \frac{1}{2}i \\ -\frac{1}{2} - \frac{\sqrt{3}}{2}i & 0 & 0 & -\frac{1}{2} - \frac{\sqrt{3}}{2}i \end{bmatrix}.$$

Step 4: Transform rows. Similarly to Step 2, we replace each row X_i with F_4X_i, where F_4 is based on the fourth root $\omega_4 = \omega^3 = i$. The result is:

$$D = \begin{bmatrix} 6 & -2-2i & 2 & -2+2i \\ \frac{\sqrt{3}-1}{2} + \frac{1+\sqrt{3}}{2}i & 0 & -\frac{1+\sqrt{3}}{2} + \frac{\sqrt{3}-1}{2}i & -1+\sqrt{3}i \\ -1-\sqrt{3}i & -\frac{1+\sqrt{3}}{2} + \frac{1-\sqrt{3}}{2}i & 0 & \frac{\sqrt{3}-1}{2} - \frac{1+\sqrt{3}}{2}i \end{bmatrix}.$$

Step 5: Read out the transform in column major order. We finally obtain:

$$\begin{aligned} &DFT_{12}(\boldsymbol{x}) = \\ &\quad \left(6, \frac{\sqrt{3}-1}{2} + \frac{1+\sqrt{3}}{2}i, -1-\sqrt{3}i, -2-2i, 0, -\frac{1+\sqrt{3}}{2} + \frac{1-\sqrt{3}}{2}i,\right. \\ &\qquad \left. 2, -\frac{1+\sqrt{3}}{2} + \frac{\sqrt{3}-1}{2}i, 0, -2+2i, -1+\sqrt{3}i, \frac{\sqrt{3}-1}{2} - \frac{1+\sqrt{3}}{2}i\right). \end{aligned}$$

$\square$

Exercise 3.6 (Bluestein's technique) Let $n \geq 1$ be an arbitrary integer (not necessarily a power of two). Given a complex vector $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$, let $\boldsymbol{X} = DFT_n(\boldsymbol{x})$, that is,

$$X_i = \sum_{k=0}^{n-1} x_k \omega_n^{ik}, \quad \text{for } 0 \leq i \leq n-1.$$

If $\beta = \sqrt{\omega_n}$, then $\omega_n^{ik} = \beta^{2ik} = \beta^{-(i-k)^2} \beta^{i^2} \beta^{k^2}$, hence

$$X_i \beta^{-i^2} = \sum_{k=0}^{n-1} \left(x_k \beta^{k^2} \right) \beta^{-(i-k)^2}.$$

(a) Show that, if $a_k = x_k \beta^{k^2}, b_k = \beta^{-k^2}$, and $c_k = X_k \beta^{-k^2}$, for $0 \leq k \leq n-1$, and if $m \geq 2n-1$, then $c_0, c_1, \ldots, c_{n-1}$ are the first n terms of the cyclic convolution of $(a_0, a_1, \ldots, a_{n-1}, 0^{m-n})$ and $(b_0, b_1, \ldots, b_{n-1}, 0^{m-(2n-1)}, b_{n-1}, \ldots, b_2, b_1)$.

(b) Using the result of Part (a) and the cyclic convolution theorem, argue that $\boldsymbol{X} = DFT_n(\boldsymbol{x})$ can be computed in $O(n \log n)$ time for *any* value of n.

Answer:

(a) Let $\tilde{\boldsymbol{a}} = (a_0, a_1, \ldots, a_{n-1}, 0^{m-n})$, $\tilde{\boldsymbol{b}} = (b_0, b_1, \ldots, b_{n-1}, 0^{m-(2n-1)}, b_{n-1}, \ldots, b_2, b_1)$ and let $\tilde{\boldsymbol{c}} = \tilde{\boldsymbol{a}} \circledast \tilde{\boldsymbol{b}}$. By the definition of cyclic convolution, we have, for $0 \leq i \leq n-1$,

$$\begin{aligned}
\tilde{c}_i &= \sum_{k=0}^{m-1} \tilde{a}_k \tilde{b}_{(i-k) \bmod m} \\
&= \sum_{k=0}^{n-1} a_k \tilde{b}_{(i-k) \bmod m} && (3.15) \\
&= \sum_{k=0}^{i} a_k b_{i-k} + \sum_{k=i+1}^{n-1} a_k b_{k-i} && (3.16) \\
&= \sum_{k=0}^{i} x_k \beta^{k^2} \beta^{-(i-k)^2} + \sum_{k=i+1}^{n-1} x_k \beta^{k^2} \beta^{-(k-i)^2} \\
&= \sum_{k=0}^{n-1} x_k \beta^{k^2} \beta^{-(i-k)^2} && (3.17) \\
&= c_i. && (3.18)
\end{aligned}$$

Equality (2.15) comes from the fact that

$$\tilde{a}_k = \begin{cases} a_k, & \text{if } k \leq n-1, \\ 0 & \text{if } k > n-1. \end{cases}$$

Equality (2.16) comes from the fact that

$$\tilde{b}_{(i-k) \bmod m} = \begin{cases} \tilde{b}_{(i-k)} = b_{i-k}, & \text{if } 0 \leq i-k \leq n-1, \\ \tilde{b}_{m+(i-k)} = b_{k-i}, & \text{if } -(n-1) \leq i-k < 0. \end{cases}$$

Equality (2.17) holds since $\beta^{-(k-i)^2} = \beta^{-(i-k)^2}$, for any $0 \leq i, k \leq n-1$. Finally, Equality (2.18) immediately follows from the definition of the c_i's. Hence, we have shown that $\tilde{c_i} = c_i$ for $0 \leq i \leq n-1$.

(b) We choose m to be the power of two closest to, but greater than $2n-1$. Then, we apply the strategy developed in Part (a) to first compute the vectors $\tilde{\boldsymbol{a}}$ and $\tilde{\boldsymbol{b}}$ in $O(n)$ time and then obtain their cyclic convolution $\tilde{\boldsymbol{c}} = \tilde{\boldsymbol{a}} \circledast \tilde{\boldsymbol{b}}$. Since the size of the vectors is now a power of two, we can employ the FFT algorithm to compute $\tilde{\boldsymbol{c}}$ as

$$\tilde{\boldsymbol{c}} = DFT_m^{-1}\left(DFT_m(\tilde{\boldsymbol{a}}) \cdot DFT_m(\tilde{\boldsymbol{b}})\right)$$

in $O(m \log m)$ time. Subsequently, we obtain $\boldsymbol{c}$ by simply picking the first $n-1$ components of $\tilde{\boldsymbol{c}}$. Once we have $\boldsymbol{c}$, we can finally compute $\boldsymbol{X}$ in additional $O(n)$ time as

$$X_i = c_i \beta^{i^2}, \quad \text{for } 0 \leq i \leq n-1.$$

Let us summarize the time taken. Since $2n \leq m \leq 4n$, we have $O(m \log m) = O(n \log n)$. Therefore, the overall time is: $O(n) + O(n \log n) + O(n) = O(n \log n)$. □

Exercise 3.7 Consider the linear convolution $\boldsymbol{u} \star \boldsymbol{x}$, where both sequences have length n and $\boldsymbol{u} = (1, 1, \ldots, 1)$. Design an algorithm that performs the above operation in time $O(n)$.

Answer: Let $\boldsymbol{w} = \boldsymbol{u} \star \boldsymbol{x}$. Recall that $\boldsymbol{w}$ has $2n-1$ components and that

$$w_i \;=\; \sum_{j=\max\{0,i-n+1\}}^{\min\{n-1,i\}} u_j x_{i-j}$$

$$= \begin{cases} \sum_{j=0}^{i} x_j & \text{if } 0 \le i \le n-1, \\ \\ \sum_{j=i-n+1}^{n-1} x_j & \text{if } n \le i \le 2n-2. \end{cases} \qquad (3.19)$$

From (2.19) we can easily derive the following two recurrences.

$$\begin{cases} w_0 = x_0 \\ w_i = w_{i-1} + x_i, \quad 1 \le i \le n-1. \end{cases} \qquad \begin{cases} w_{2n-2} = x_{n-1} \\ w_{2n-2-i} = w_{2n-2-i+1} + x_{n-1-i}, \quad n-1 \ge i \ge 1. \end{cases}$$

(Note that both recurrences compute w_{n-1}). The algorithm is the following:

```
UNIT_LIN_CONV($\boldsymbol{x}$)
    $n \leftarrow$ length($\boldsymbol{x}$)
    $z_0 \leftarrow x_0$
    $z_{2n-2} \leftarrow x_{n-1}$
    for $i \leftarrow 1$ to $n-1$ do
        $z_i \leftarrow z_{i-1} + x_i$
        $z_{2n-2-i} \leftarrow z_{2n-2-i+1} + x_{n-1-i}$
    return $\boldsymbol{z}$
```

The program performs exactly $2n-2$ additions and therefore runs in linear time. □

Exercise 3.8 Consider the problem of computing the cyclic convolution $\boldsymbol{x} \circledast \boldsymbol{y}$ of the vectors $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$.

(a) Evaluate the running time of an algorithm based on the definition of cyclic convolution when exactly k entries of $\boldsymbol{x}$ and k entries of $\boldsymbol{y}$ are nonzero.

(b) Compare your solution to Point (a) with an FFT-based solution and discuss for which values of k and n the former yields a more efficient algorithm than the latter.

(c) Repeat the argument for (a) and (b) when k divides n and the nonzero elements of $\boldsymbol{x}$ and $\boldsymbol{y}$ have indices which are an integer multiple of n/k.

Answer:

(a) Recall that for $0 \le i \le n-1$,

$$z_i = \left(\boldsymbol{x} \circledast \boldsymbol{y}\right)_i = \sum_{j=0}^{n-1} x_j y_{(i-j) \bmod n}.$$

Fix two indices r, s, with $0 \leq r, s \leq n-1$, such that $x_r, y_s \neq 0$. Since the equation

$$i - r \bmod n = s$$

has a unique solution in $\{0, 1, \ldots, n-1\}$:

$$i = \begin{cases} r+s & r+s \leq n-1 \\ r+s-n & \text{otherwise,} \end{cases} \tag{3.20}$$

each pair of nonzero indices will contribute exactly to a component of $\boldsymbol{z}$. Therefore we can write the following algorithm:

```
DEF_CONV($\boldsymbol{x}, \boldsymbol{y}$)
$r \leftarrow 0$; $s \leftarrow 0$
for $i \leftarrow 1$ to $k$ do
    while $x_r = 0$ do $r \leftarrow r+1$
    while $y_s = 0$ do $s \leftarrow s+1$
    $\overline{x}_i \leftarrow (r, x_r)$; $\overline{y}_i \leftarrow (s, y_s)$
    $r \leftarrow r+1$; $s \leftarrow s+1$
    { $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{y}}$ contain the $k$ pairs (index,value)
    corresponding to nonzero entries of $\boldsymbol{x}$ and $\boldsymbol{y}$ }
for $i \leftarrow 0$ to $n-1$ do $z_i \leftarrow 0$
for $i \leftarrow 1$ to $k$ do
    for $j \leftarrow 1$ to $k$ do
        let $\overline{x}_i = (r, x_r)$ ; let $\overline{y}_j = (s, y_s)$
        if $r + s \leq n-1$
            then $z_{r+s} \leftarrow z_{r+s} + x_r y_s$
            else $z_{r+s-n} \leftarrow z_{r+s-n} + x_r y_s$
        { perform the nonzero products and update
        the corresponding component of $\boldsymbol{z}$ }
return $\boldsymbol{z}$
```

The correctness of the above algorithm follows directly from Equation (2.20). As for its running time, the extraction of the nonzero indices and the initialization of $\boldsymbol{z}$ take $O(n)$ time altogether, while the actual computation of the z_i's requires $O(k^2)$ time, for a total time

$$T_{\mathrm{DEF}}(n, k) = O\left(n + k^2\right).$$

(b) If the k nonzero components of $\boldsymbol{x}$ and $\boldsymbol{y}$ have arbitrary indices, there is no general way of exploiting the sparseness of the vectors when computing their discrete Fourier transforms, therefore the FFT-based convolution algorithm requires $T_{\mathrm{FFT}}(n, k) = O(n \log n)$

time, regardless of k. Since

$$O\left(n+k^2\right) = O(n\log n) \text{ iff } k = O\left(\sqrt{n\log n}\right),$$

the algorithm developed in Part (a) asymptotically outperforms the FFT-based algorithm for $k = o\left(\sqrt{n\log n}\right)$.

(c) Any algorithm based on the definition has to compute at least k^2 products and initialize the n components of $\boldsymbol{z} = \boldsymbol{x} \circledast \boldsymbol{y}$, therefore the complexity stays $\Omega(n+k^2)$ when the nonzero components have indices rn/k, with $0 \le r \le k-1$. In contrast, the FFT-based algorithm can be improved by only operating on the sub-vectors $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{y}}$ of nonzero components. To see that this approach is correct, consider the Cooley-Tukey algorithm (Exercise 2.5) to compute $F_n\boldsymbol{x}$ in terms of transforms of order k and n/k (the argument for $F_n\boldsymbol{y}$ is identical). The steps of the algorithm are the following:

1. *Put $\boldsymbol{x}$ in a $k \times n/k$ array, in row major order.* Note that the first column is $\overline{\boldsymbol{x}}$, while the other columns are all zero.

2. *Transform colums.* We only have to transform $\overline{\boldsymbol{x}}$ (the first column).

3. *Multiply entry (i,j) by $(\omega_n)^{ij}$.* The only nonzero elements are in positions (i,j) with $j=0$. Thus, this step can be skipped.

4. *Transform rows.* The i-th row of the matrix has a single nonzero element as its first component, $(F_k\overline{\boldsymbol{x}})_i$. Therefore its tranform is a vector with all components equal to $(F_k\overline{\boldsymbol{x}})_i$.

5. *Read out the transform in colum major order.* Note that the resulting vector is an n/k-fold repetition of $F_k\overline{\boldsymbol{x}}$.

After computing $F_n\boldsymbol{x}$ and $F_n\boldsymbol{y}$, we perform component-wise product and then compute the inverse transform. By the convolution theorem, the end result is $\boldsymbol{z} = \boldsymbol{x} \circledast \boldsymbol{y}$. Since the component-wise product still yields a periodic vector, its reverse transform $\boldsymbol{z}$ has only k nonzero components at positions rn/k, with $0 \le r \le k-1$. This can be easily seen by applying the Cooley-Tukey algorithm again, this time folding the vector in an $n/k \times k$ array. Then, all rows are identical and transforming the columns leaves only the first row with nonzero elements. Moreover, the r-th nonzero element of $\boldsymbol{z}$ is $\left(F_k^{-1}\left(F_k\overline{\boldsymbol{x}} \cdot F_k\overline{\boldsymbol{y}}\right)\right)_r$.

Note that after extracting the nonzero components of $\boldsymbol{x}$ and $\boldsymbol{y}$, we only need to operate on vectors of size k. Therefore the running time of the algorithm is

$$T_{\mathrm{FFT}}(n,k) = O(n + k\log k).$$

In this case, the FFT-based algorithm always outperforms the algorithm based on the definition. □

Exercise 3.9 Show that both linear and cyclic convolution are commutative and associative operators.

Exercise 3.10 Let $n = pq$. Let $x = (x_0, x_1, \ldots, x_{n-1})$ be a *periodic sequence* of period q, that is, $x_{i+q} = x_i$, for $i = 0, 1, \ldots, n-1-q$. Let $X = (X_0, X_1, \ldots, X_{n-1}) = Fx$. Prove that $X_k = 0$ unless k is a multiple of p. (*Hint:* base your argument on the Cooley-Tukey algorithm introduced in Exercise 2.5).

Exercise 3.11 Consider the following equation system

$$\boldsymbol{x} \circledast \boldsymbol{x} = \boldsymbol{y},$$

where $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ is a vector of complex unknowns and $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$ is a given vector of complex numbers.

(a) How many solutions has the system? In case the number of solutions is a function of $\boldsymbol{y}$, derive this function.

(b) Give an $O(n\log n)$ algorithm that, on input $\boldsymbol{y}$, outputs *one* solution to the system, if one exists.

Exercise 3.12 The complex number $\omega_3 = -1/2 + i\sqrt{3}/2$ is the principal third root of the unity in the complex field.

(a) Evaluate ω_3^i for $i = 0, 1, 2, 3, 4$.

(b) Write the Fourier matrix F_3.

(c) Write F_3^{-1}

(d) Let $\boldsymbol{x} = (0, 1, 2)$. Let $\boldsymbol{y}$ be the cyclic convolution of 8 vectors all equal to $\boldsymbol{x}$. Compute $\boldsymbol{y}$.

Exercise 3.13 Let $(X_0, X_1, \ldots, X_{n-1}) = DFT_n(x_0, x_1, \ldots, x_{n-1})$. Consider now the vector $(Y_0, Y_1, \ldots, Y_{2n-1}) = DFT_{2n}(x_0, 0, x_1, 0, \ldots, x_{n-1}, 0)$. Write the Y_k's as a function of the X_i's.

Exercise 3.14 Let m and n be two integers, with $m \gg n$ a multiple of n. Describe and analyze an algorithm to multiply the two polynomials $p(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_0$ and $q(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_0$ in time $O(m \log n)$.

Exercise 3.15 Let $\boldsymbol{x}$ be a complex vector. For $\alpha \neq 0$ an arbitrary complex number, let A be the symmetric $n \times n$ Vandermonde matrix whose (i, j)-th element is $a_{ij} = \alpha^{ij}$, for $0 \leq i, j \leq n-1$. Using the ideas of Bluestein's technique (Exercise 2.6), show how to apply the results of Exercise 2.3 to compute the product $A\boldsymbol{x}$ in $O(n \log n)$ time. (*Hint:* rewrite $A\boldsymbol{x}$ as $T\boldsymbol{z}$, where T is a Toeplitz matrix and $\boldsymbol{z}$ is a suitable complex vector.

Chapter 4

Dynamic Programming

Exercise 4.1 Design and analyze an algorithm to determine the minimum value achievable by a full parenthesization of the following expression, given in input:

$$A = a_1 O_1 a_2 O_2 \dots a_{n-1} O_{n-1} a_n,$$

where a_i is a positive integer, for $1 \le i \le n$ and $O_j \in \{+, \cdot\}$, for $1 \le j \le n-1$.

Answer: For $1 \le i \le j \le n$, let $A_{i..j} = a_i O_i a_{i+1} \dots O_{j-1} a_j$, and let $m[i,j]$ be the minimum value attained by a full parenthesization of $A_{i..j}$. Since both $+$ and $\cdot$ are minimized when their operands are minimized, such minimizing parenthesization must embed, at the outer level, two minimizing parenthesizations of $A_{i..k}$ and $A_{k+1..j}$, respectively, for some k, $i \le k < j$. Therefore, we can write the following recurrence for $m[i,j]$:

$$m[i,j] = \begin{cases} a_i & i = j \\ \min\{m[i,k]\, O_k\, m[k+1,j] : i \le k < j\} & i < j, \end{cases}$$

The algorithm follows.

```
MINIMIZE($A$)
$n \leftarrow \lceil \text{length}(A)/2 \rceil$
**for** $i \leftarrow 1$ **to** $n$ **do** $m[i,i] \leftarrow a_i$
**for** $\ell \leftarrow 2$ **to** $n$ **do**
    { process subexpressions of increasing length ...}
    for $i \leftarrow 1$ to $n - \ell + 1$ do
        { starting at $a_i$...}
        $j \leftarrow i + \ell - 1$
        { ... and ending at $a_j$ }
        $m[i, j] \leftarrow \infty$
        for $k \leftarrow i$ to $j - 1$ do
            $t \leftarrow m[i, k]\, O_k\, m[k+1, j]$
            if $t < m[i, j]$
              then $m[i, j] \leftarrow t$
return $m[1, n]$
```

The above algorithm consists of three nested loops of $O(n)$ iterations each, for a total of $O(n^3)$ time. □

Exercise 4.2 Write an algorithm to find the maximum value that can be obtained by an appropriate placement of parentheses in the expression

$$x_1/x_2/x_3/\ldots x_{n-1}/x_n,$$

where $x_1, x_2, \ldots, x_n$ are positive rational numbers and "/" denotes division.

Answer: For $1 \leq i \leq j \leq n$, denote by $X_{i\ldots j}$ the subexpression $x_i/x_{i+1}/\ldots/x_j$. Given a full parenthesization $\mathcal{P}_{1\ldots n}$ of $X_{1\ldots n}$, let its cost $c(\mathcal{P}_{1\ldots n})$ be the value obtained by performing the division according to the order dictated by the parentheses. An optimal parenthesization of $X_{1\ldots n}$ is one that maximizes the above cost function.

Any full parenthesization $\mathcal{P}_{1\ldots n}$ of $X_{1\ldots n}$ contains, at the outer level, parenthesizations $\mathcal{P}_{1\ldots k}$ and $\mathcal{P}_{k+1\ldots n}$ of the subsequences $X_{1\ldots k}$ and $X_{k+1\ldots n}$ for a given value k, $1 \leq k \leq n-1$. Moreover, this property holds recursively for $\mathcal{P}_{1\ldots k}$ and $\mathcal{P}_{k+1\ldots n}$. The relation among the costs of the above parenthesizations is the following:

$$c(\mathcal{P}_{1\ldots n}) = \frac{c(\mathcal{P}_{1\ldots k})}{c(\mathcal{P}_{k+1\ldots n})}.$$

The key observation upon which we will base our algorithm is that any maximizing (resp., minimizing) parenthesization $\overline{\mathcal{P}}_{1\ldots n}$ must be formed by a parenthesization $\overline{\mathcal{P}}_{1\ldots k}$ that *maximizes* (resp., *minimizes*) c for the string $X_{1\ldots k}$, and a parenthesization $\overline{\mathcal{P}}_{k+1\ldots n}$ that *minimizes* (resp., *maximizes*) c for $X_{k+1\ldots n}$, for some value k, $1 \leq k \leq n-1$. Indeed, if it were not so, a better $\overline{\mathcal{P}}_{1\ldots k}$ or $\overline{\mathcal{P}}_{k+1\ldots n}$ would immediately yield a better $\overline{\mathcal{P}}_{1\ldots n}$.

Let $M[i,j]$ denote the cost of a maximizing parenthesization of $X_{i\ldots j}$, $1 \le i \le j \le n$, and let $m[i,j]$ denote the cost of a minimizing parenthesization of $X_{i\ldots j}$. Based on the above observations, we can write the following recurrence for $m[i,j]$ and $M[i,j]$:

$$\begin{aligned} M[i,j] &= m[i,j] = x_i & \text{if } i = j \\ M[i,j] &= \max\left\{ \frac{M[i,k]}{m[k+1,j]} : i \le k < j \right\} & \text{if } i < j \\ m[i,j] &= \min\left\{ \frac{m[i,k]}{M[k+1,j]} : i \le k < j \right\} & \text{if } i < j \end{aligned}$$

The algorithm follows immediately from the above recurrence.

```
CHAIN_DIVISION($x_1, x_2, \ldots, x_n$)
for $i \leftarrow 1$ to $n$ do
    $M[i,i] \leftarrow m[i,i] \leftarrow x_i$
for $\ell \leftarrow 2$ to $n$ do {compute the values of $M$ and $m$ for substrings of length $\ell$}
    for $i \leftarrow 1$ to $n - \ell + 1$ do
        $j \leftarrow i + l - 1$
        $M[i,j] \leftarrow 0$
        $m[i,j] \leftarrow \infty$
        for $k \leftarrow i$ to $j - 1$ do
            $t_1 \leftarrow M[i,k]/m[k+1,j]$
            $t_2 \leftarrow m[i,k]/M[k+1,j]$
            {$M[i,k]$, $m[i,k]$, $M[k+1,j]$ and $m[k+1,j]$ already
            available at this point}
            if $M[i,j] < t_1$ then $M[i,j] \leftarrow t_1$
            if $m[i,j] > t_2$ then $m[i,j] \leftarrow t_2$
return $M[1,n]$
```

The above algorithm computes the cost of an optimal parenthesization in $O(n^3)$ time. If we are interested in actually determining the structure of the parenthesization, it is sufficient to compute two additional tables, $s_M[1\ldots n, 1\ldots n]$ and $s_m[1\ldots n, 1\ldots n]$, with $s_M[i,j]$ (resp., $s_m[i,j]$) recording at which index k the maximizing (resp., minimizing) parenthesization of $X_{i\ldots j}$ is split into optimal parenthesizations for $X_{i\ldots k}$ and $X_{k+1\ldots j}$. Note that s_M and s_m can be computed without increasing the running time of the algorithm. □

Exercise 4.3 In Algorithm MATRIX_CHAIN_ORDER (CLR, page 306), determine the *exact* number of times that the following Line is executed:

$$\textbf{do } q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_i$$

Answer: Line 9 is executed once in each iteration of the innermost loop,

for $k \leftarrow i$ **to** $j-1$ **do** ...

This loop is executed once in each iteration of the loop

for $i \leftarrow 1$ **to** $n-\ell+1$ **do** ...,

which is in turn executed once in each iteration of the loop

for $l \leftarrow 2$ **to** n **do** ....

Recall that $j = i + \ell - 1$. Therefore, the total number of times that Line 9 is executed is

$$\begin{aligned}
T_9(n) &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 1 \\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) \\
&= \sum_{l=2}^{n} (l-1)(n-l+1) \\
&= \sum_{l=2}^{n} l(n-l+1) - \sum_{l=2}^{n}(n-l+1) \\
&= (n+1)\sum_{l=2}^{n} l - \sum_{l=2}^{n} l^2 - \sum_{h=1}^{n-1} h \text{ (we set } h = n-l+1 \text{ in the second sum)} \\
&= (n+1)\left[\frac{n(n+1)}{2} - 1\right] - \frac{n(n+1)(2n+1)}{6} + 1 - \frac{(n-1)n}{2} \\
&= \frac{n^3+2n^2+n}{2} - n - 1 - \frac{2n^3+3n^2+n}{6} + 1 - \frac{n^2-n}{2} \\
&= \frac{3n^3-2n^3}{6} + \frac{2n^2-n^2-n^2}{2} + \frac{3n-6n-n+3n}{6} - 1 + 1 \\
&= \frac{n^3-n}{6}.
\end{aligned}$$

Note that $T_9(n) = \Theta(n^3)$. □

Exercise 4.4 Give an algorithm that uses the vector s computed by Algorithm MATRIX_CHAIN_ORDER (CLR, page 306) to print the optimal parenthesization for the matrix chain.

Answer: Let s be the array computed by MATRIX_CHAIN_ORDER. Recall that $s[i,j]$ stores the splitting index k of the optimal parenthesization of the subchain $A_{i..j}$ of matrices $A_i, A_{i+1}, \ldots, A_j$, with $1 \le i \le k < j \le n$. We can write the following recursive algorithm.

```
PRINT_OPTIMAL_PARENS($i, j$)
if $i = j$
   then print('$A_i$')
          return
print('(')
$k \leftarrow s[i, j]$
PRINT_OPTIMAL_PARENS($i, k$)
PRINT_OPTIMAL_PARENS($k + 1, j$)
print(')')
return
```

Let us charge one time unit for any **print** statement, and let $T(n)$ be the running time of PRINT_OPTIMAL_PARENS$(1, n)$. When $n = 1$, the above procedure simply prints A_1. When $n > 1$, the number of **print** statements is the number of **print** statements performed by the two recursive calls plus 2. The size of the subinstances is $s[1, n]$ and $n - s[1, n]$, respectively. We obtain the following recurrence

$$\begin{cases} T(n) = T(s[1,n]) + T(n - s[1,n]) + 2, & n > 1, \\ T(1) = 1. \end{cases}$$

Let us prove by induction that $T(n) = 3n - 2$ (which is exactly the number of symbols of a full parenthesization of $A_1 A_2 \dots A_n$). The base case trivially holds. Assuming that $T(k) = 3k - 2$, for $1 \le k < n$, we obtain

$$\begin{aligned} T(n) &= T(s[1,n]) + T(n - s[1,n]) + 2 \\ &= 3s[1,n] - 2 + 3(n - s[1,n]) - 2 + 2 \\ &= 3n - 2, \end{aligned}$$

and the inductive thesis follows. Therefore, PRINT_OPTIMAL_PARENS runs in linear time. □

Exercise 4.5 Let $x_1 x_2 \dots x_m \in \Sigma^\star$, where Σ is a finite alphabet. Let DEL(i) denote the operation of deleting x_i from the string, and let INS(c, i) denote the operation of inserting a new character $c \in \Sigma$ just before x_{i+1}. Consider now two strings X and Y in $\Sigma^\star$, and define the *edit distance* from X to Y, $ED(X, Y)$, as the *minimum* number of DEL and INS operations needed to transform X into Y. As an example, if $X =$ `man` and $Y =$ `women`, then $ED(X, Y) = 4$, which can be obtained through the following *transformation sequence*:

$$\text{INS}(\texttt{w}, 0) \qquad (\rightarrow \texttt{wman})$$

$$\begin{aligned} \text{INS}(\texttt{o},0) &\quad (\to \texttt{woman}) \\ \text{DEL}(2) &\quad (\to \texttt{womn}) \\ \text{INS}(\texttt{e},2) &\quad (\to \texttt{women}). \end{aligned}$$

(a) Design and analyze a dynamic programming algorithm which, on input two strings $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$, computes $ED(x,y)$.

(b) Design and analyze a recursive algorithm that uses the information computed by the previous algorithm to print a shortest transformation sequence.

Answer:

(a) Let $LCS(X,Y)$ denote the length of a *longest common subsequence* of X and Y (CLR, page 317). We have:

$$ED(X,Y) = m + n - 2 \cdot LCS(X,Y).$$

In order to prove that $ED(X,Y) \leq m + n - 2 \cdot LCS(X,Y)$, let Z be any longest common subsequence of X and Y. Then a possible trasformation sequence first deletes all the characters in X that are not in Z and then inserts all the characters in Y that are not in Z. The length of such transformation sequence is $m+n-2\cdot LCS(X,Y)$. In order to prove that $ED(X,Y) \geq m+n-2\cdot LCS(X,Y)$, consider an arbitrary shortest transformation sequence. Let $X' = x_{j_1}x_{j_2}\ldots x_{j_h}$ be the subsequence of X made of characters upon which DEL operations are not invoked. Similarly, let $Y' = y_{j_1}y_{j_2}\ldots y_{j_k}$ be the subsequence of Y made of characters upon which INSERT operations are not invoked. Clearly, X' and Y' are common subsequences of X and Y, since the x_{j_i}'s must already belong to Y and the y_{j_i}'s must already belong to X. The length of such sequence is $m+n-h-k \geq m+n-2\cdot LCS(X,Y)$.

Let us now develop a dynamic programming algorithm for the problem using the properties of a longest common subsequence of X_i and Y_j (CLR, pages 315–316). For $1 \leq i \leq m$ and $1 \leq j \leq n$, let X_i and Y_j denote the prefixes $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$ of X and Y, respectively. Moreover, let $X_0 = Y_0 = \varepsilon$. For $0 \leq i \leq m$ and $0 \leq j \leq n$ we have:

$$ED(X_i,Y_j) = \begin{cases} j & \text{if } i = 0, \\ i & \text{if } j = 0, \\ ED(X_{i-1},Y_{j-1}) & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \min\{ED(X_{i-1},Y_j), ED(X_i,Y_{j-1})\} + 1 & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

In order to prove the above relationship, consider first the case $i = 0$. Then, the minimum tranformation sequence needed to transform $X_0 = \varepsilon$ into Y_j is clearly a sequence of j INS operations, one for each character of Y_j. Analogously, when $j = 0$, then the minimum tranformation sequence needed to transform X_i into ε consists of i DEL operations. Let now $i, j > 0$. When $x_i = y_j$, we have:

$$\begin{aligned}
ED(X_i, Y_j) &= i + j - 2 \cdot LCS(X_i, Y_j) \\
&= (i-1) + (j-1) + 2 - 2 \cdot (LCS(X_{i-1}, Y_{j-1}) + 1) \\
&= (i-1) + (j-1) - 2 \cdot LCS(X_{i-1}, Y_{j-1}) \\
&= ED(X_{i-1}, Y_{j-1}).
\end{aligned}$$

When $x_i \neq y_j$, we have:

$$\begin{aligned}
ED(X_i, Y_j) &= i + j - 2 \cdot LCS(X_i, Y_j) \\
&= (i + j - 1) + 1 - 2\,(\max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\}) \\
&= \min\{(i+j-1) - 2 \cdot LCS(X_i, Y_{j-1}), (i+j-1) - 2 \cdot LCS(X_{i-1}, Y_j)\} + 1 \\
&= \min\{ED(X_i, Y_{j-1}), ED(X_{i-1}, Y_j)\} + 1.
\end{aligned}$$

The algorithm follows.

```
EDIT_DISTANCE($X, Y$)
$m \leftarrow$ length($X$)
$n \leftarrow$ length($Y$)
$ED[0,0] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $m$ **do**
    $ED[i,0] \leftarrow i$
    $B[i,0] \leftarrow$ '$D$'
**for** $j \leftarrow 1$ **to** $n$ **do**
    $ED[0,j] \leftarrow j$
    $B[0,j] \leftarrow$ '$I$'
**for** $i \leftarrow 1$ **to** $m$ **do**
    **for** $j \leftarrow 1$ **to** $n$ **do**
        **if** $x_i = y_j$
          **then** $ED[i,j] \leftarrow ED[i-1,j-1]$
                $B[i,j] \leftarrow$ '$M$'
          **else if** $ED[i-1,j] \leq ED[i,j-1]$
                **then** $ED[i,j] \leftarrow ED[i-1,j] + 1$
                      $B[i,j] \leftarrow$ '$D$'
                **else** $ED[i,j] \leftarrow ED[i,j-1] + 1$
                      $B[i,j] \leftarrow$ '$I$'
**return** $ED[m,n]$, $B$
```

The running time $T_{\text{ED}}(m,n)$ of the algorithm is determined by the two nested loops needed to compute the entries of Tables ED and B. Since only constant work is performed for each such entry, we have $T_{\mathrm{ED}}(m,n) = \Theta(mn)$. Finally, the space requirement of the algorithm is also $\Theta(mn)$.

(b) Table B returned by EDIT_DISTANCE can be readily used to output a shortest transformation sequence. Algorithm PRINT_SEQUENCE below assumes a global knowledge of such table.

```
PRINT_SEQUENCE($i, j$)
if ($i = 0$) and ($j = 0$)
   then return
if $B[i,j] =$ 'M'
   then PRINT_SEQUENCE($i-1, j-1$)
   else if $B[i,j] =$ 'D'
           then PRINT_SEQUENCE($i-1, j$)
                  print('DEL(', $i$, ')')
           else PRINT_SEQUENCE($i, j-1$)
                 print('INS(', $y_j$, ',', $i$, ')')
return
```

The correctness of the above algorithm follows immediately from the structure of Table B, which contains the relevant information needed to reconstruct an optimal transformation sequence. In particular, $B[i,j]$ records whether $x_i = y_j$ (a match) or which value between $ED(X_{i-1}, Y_j)$ and $ED(X_i, Y_{j-1})$ is minimum. According to such information we can select the appropriate operation to be appended to the transformation sequence. In order to estimate the running time $T_{\mathrm{PS}}(m,n)$ of PRINT_SEQUENCE, observe that at each recursive call the sum of its parameters decreases by at least one. Therefore $T_{\mathrm{PS}}(m,n) = O(m+n)$.
□

Exercise 4.6 Design and analyze a dynamic programming algorithm that returns a Longest Monotonically Increasing Subsequence (LMIS) of a string $A = a_1 a_2 \ldots a_n$ of n integers.

Answer: For $1 \le i \le n$, let $\ell[i]$ be the length of an LMIS with the additional constraint that the first element be a_i. Clearly, the length of any LMIS of $a_1 a_2 \ldots a_n$ is $\max\{\ell[i] : 1 \le i \le n\}$. Also, let $A_i = \{j : i < j \le n \text{ and } a_i < a_j\}$. Then, the following recurrence must hold:

$$\ell[i] = \begin{cases} 1 & \text{if } A_i = \emptyset, \\ 1 + \max\{\ell[j] : j \in A_i\} & \text{otherwise.} \end{cases}$$

To show that the above relation holds, note that if $A_i = \emptyset$, then all elements a_j with $i < j \le n$, if any, are no bigger than a_i, hence the only monotonically increasing subsequence with first element a_i is a_i itself. When $A_i \neq \emptyset$, consider a given LMIS starting at a_i. Such LMIS has clearly length at least 2. Now, if the second element of such subsequence is a_k, then $a_k > a_i$ and $k > i$ (by the definition of monotonically increasing subsequence), therefore $k \in A_i$. Moreover, all the elements of the subsequence, excluding a_i, must form an LMIS starting at a_k. If this were not the case, we could find a monotonically increasing subsequence starting at a_i longer than the LMIS itself, a contradiction.

The above relation immediately yields a dynamic programming algorithm for the LMIS problem. In the algorithm, we maintain a variable *start*, storing the starting point of the longest subsequence found so far, *len*, storing the length of such subsequence, and finally a vector $next[i]$, $1 \leq i \leq n$ containing the following information:

$$next[i] = \begin{cases} 0 & \text{if } A_i = \emptyset, \\ k & \text{if } \ell[k] = \max\{\ell[j] : j \in A_i\}. \end{cases}$$

The algorithm uses vector *next* to output the desired LMIS for $a_1 a_2 \ldots a_n$.

```
L_M_I_S($a_1 a_2 \ldots a_n$)
$len \leftarrow 0$
for $i \leftarrow n$ downto 1 do
    $\ell[i] \leftarrow 1$
    $next[i] \leftarrow 0$
    for $j \leftarrow i+1$ to $n$ do
        if $a_j > a_i$ then
            if $\ell[i] < 1 + \ell[j]$ then
                $\ell[i] \leftarrow 1 + \ell[j]$
                $next[i] \leftarrow j$
    if $len < \ell[i]$ then
        $len \leftarrow \ell[i]$
        $start \leftarrow i$
$curr \leftarrow start$
$LMIS[1] \leftarrow a_{curr}$
for $j \leftarrow 2$ to $len$ do
    $curr \leftarrow next[curr]$
    $LMIS[j] \leftarrow a_{curr}$
return $LMIS$
```

The running time $T(n)$ of the above algorithm is upperbounded by the number of iterations of the two nested loops needed to compute vectors ℓ and *next*. Therefore

$$T(n) = \Theta\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1\right)$$

$$= \Theta\left(\sum_{i=1}^{n-1} i\right)$$
$$= \Theta\left(n^2\right).$$

□

Exercise 4.7 Given the string $A = a_1a_2 \ldots a_n$, we say that $A_{i..j} = a_ia_{i+1} \ldots a_j$ is a *palindrome substring* of A if $a_{i+h} = a_{j-h}$, for $0 \le h \le j - i$. (Intuitively, a palindrome substring is one which is identical to its "mirror" image. For example, if $A = accaba$, then both $A_{1..4} = acca$ and $A_{4..6} = aba$ are palindrome substrings of A.)

(a) Design a dynamic programming algorithm that determines the length of a longest palindrome substring of a string $A = a_1a_2 \ldots a_n$ in $O(n^2)$ time and $O(n^2)$ space.

(b) Modify your algorithm so that it uses only $O(n)$ space, while the running time remains unaffected.

Answer:

(a) It is worth noting that there are no more than $O(n^2)$ *substrings* in a string of length n (while there are exactly 2^n *subsequences*). Therefore, we could scan each substring, check for palindromicity and update the length of the longest palindrome substring discovered so far. Since the palindromicity test takes time linear in the length of the substring, this simple idea yields a $\Theta(n^3)$ algorithm. However, we can use dynamic programming to devise a much better algorithm. For $1 \le i \le j \le n$, define

$$P[i,j] = \begin{cases} \textbf{true} & \text{if } A_{i..j} \text{ is a palindrome substring,} \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Clearly, $P[i,i] = \textbf{true}$, while $P[i,i+1] \Leftrightarrow a_i = a_{i+1}$, for $1 \le i \le n-1$. It is also immediate to see that for $j - i + 1 \ge 3$ (i.e., for strings of length at least 3), we have

$$P[i,j] \Leftrightarrow (P[i+1,j-1] \textbf{ and } a_i = a_j)\,. \qquad (4.1)$$

Note that in order to obtain a well defined recurrence, we need to explicitly initialize *two* distinct diagonals of the boolean array $P[i,j]$, since the recurrence for entry $[i,j]$ uses the value $[i-1,j-1]$, which is two diagonals away from $[i,j]$ (in other words, for a substring of length ℓ, we need to know the status of a substring of length $\ell - 2$).

The following algorithm is immediately obtained from the above considerations.

```
LONGEST_PALINDROME_SUBSTRING($A$)
$n \leftarrow$ length($A$)
$max \leftarrow 1$
for $i \leftarrow 1$ to $n-1$ do
   $P[i,i] \leftarrow$ true
   { note that $P[n,n]$ will be never used below }
   if $a_i = a_{i+1}$
      then $P[i,i+1] \leftarrow$ true;
            $max \leftarrow 2$
      else $P[i,i+1] \leftarrow$ false
for $\ell \leftarrow 3$ to $n$ do
   { check the substrings of length $\ell$ }
      for $i \leftarrow 1$ to $n-\ell+1$ do
            $j \leftarrow i+\ell-1$
            if $(P[i+1,j-1]$ and $a_i = a_j)$
               { $P[i+1,j-1]$ already available at this point }
               then $P[i,j] \leftarrow$ true
                     $max \leftarrow \ell$
               else $P[i,j] \leftarrow$ false
return $max$
```

Since the algorithm performs a constant number of operations for each of the $\Theta(n^2)$ substrings of A, it takes $O(n^2)$ time, while the space needed to store the table $P[i,j]$ is clearly $O(n^2)$.

(b) Note that by the ℓ-th iteration of the outer **for** loop, we only need values $P[i,j]$ with $j-i+1 = \ell-2$ (needed for iteration ℓ), $\ell-1$ (needed for iteration $\ell+1$), or ℓ, (the ones that we are computing). These are the values of P on diagonals $\ell-2$ and $\ell-1$ and ℓ. Therefore, at any time in the algorithm, it is sufficent to store no more than $3n$ entries of P. The algorithm above can be easily modified as follows.

```
LINEAR_SPACE_L_P_S($A$)
$n \leftarrow$ length($A$)
$max \leftarrow 1$
for $i \leftarrow 1$ to $n-1$ do
   $P[i,1] \leftarrow$ true
   { $P$ is an array with only 3 columns }
   if $a_i = a_{i+1}$
      then $P[i,2] \leftarrow$ true
            $max \leftarrow 2$
      else $P[i,2] \leftarrow$ false
```

```
for $\ell \leftarrow 3$ to $n$ do
    { check the substrings of length $\ell$ }
        for $i \leftarrow 1$ to $n - \ell + 1$ do
            if ($P[i+1,1]$ and $a_i = a_{i+\ell-1}$)
              then $P[i,3] \leftarrow$ true
                   $max \leftarrow \ell$
              else $P[i,3] \leftarrow$ false
            $P[i,1] \leftarrow P[i,2]$
            $P[i,2] \leftarrow P[i,3]$
            { shift relevant entries one column left }
return $max$
```

We can further improve the above algorithm so that it uses only two column vectors. In fact, after we check for palindromicity of the substring of length ℓ starting at i, we could first save $P[i,2]$ into $P[i,1]$ (which is not needed anymore) and then store the newly computed value directly in $P[i,2]$, rather than $P[i,3]$. However, the given algorithm is sufficient to achieve linear space, with a running time which is is no more than three times the running time of the algorithm of Part (a), whose space requirement was $\Theta(n^2)$. □

Exercise 4.8 A binary string of length $2k$, $k \geq 0$, is *balanced* if it contains k zeros and k ones. Design and analyze an algorithm to determine the length of the longest balanced substring of a binary string $x_1 x_2 \ldots x_n$.

Answer: Let $\Delta[0] = 0$ and, for $1 \leq i \leq n$, let

$$\Delta[i] = \left|\{h : x_h = 1; 1 \leq h \leq i\}\right| - \left|\{k : x_k = 0; 1 \leq k \leq i\}\right|.$$

In other words, $\Delta[i]$ is the difference between the number of one and zero components of index at most i. It is straightforward to prove that the following recurrence holds:

$$\Delta[i] = \begin{cases} 0 & i = 0, \\ \Delta[i-1] + 1 & i > 0 \text{ and } x_i = 1, \\ \Delta[i-1] - 1 & i > 0 \text{ and } x_i = 0. \end{cases}$$

Observe that $-i \leq \Delta[i] \leq i$. Let now $x_{i\ldots j}$ denote the substring $x_i x_{i+1} \ldots x_j$, for $1 \leq i < j \leq n$. We can prove the following:

$$x_{i\ldots j} \text{ is balanced } \Leftrightarrow \Delta[j] = \Delta[i-1].$$

Indeed, by the above definition,

$$\Delta[j] - \Delta[i-1] = \left|\{h : x_h = 1, i \le h \le j\}\right| - \left|\{k : x_k = 0, i \le k \le j\}\right|,$$

therefore $\Delta[j] - \Delta[i-1] = 0$ (or, equivalently, $\Delta[j] = \Delta[i-1]$) if and only if the number of zero and one components in $x_{i\ldots j}$ is the same. The above discussion immediately suggests a dynamic programming algorithm which first computes vector Δ and then proceeds to evaluate all possible differences $\Delta[j] - \Delta[i-1]$, for $1 \le i < j \le n$, maintaining the maximum value of $j - i + 1$ such that $\Delta[j] - \Delta[i-1] = 0$. Such algorithm would run in $\Theta(n^2)$ time. In fact, we can get a linear-time algorithm by first sorting the $\Delta[i]$'s (carrying along their source index) using the same idea of BUCKET_SORT (CLR, page 181), and then computing the maximum and minimum index within each bucket. The algorithm follows:

```
MAX_BALANCED_SUBSTRING($x_1 x_2 \ldots x_n$)
$\Delta[0] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $n$ **do**
    **if** $x_i = 1$
       **then** $\Delta[i] \leftarrow \Delta[i-1] + 1$
       **else**  $\Delta[i] \leftarrow \Delta[i-1] - 1$
{ compute the $\Delta[i]$'s }
**for** $k \leftarrow -n$ **to** $n$ **do** $B[k] = \emptyset$
**for** $i \leftarrow 0$ **to** $n$ **do** $B[\Delta[i]] \leftarrow B[\Delta[i]] \cup \{i\}$
{ $B$ is an array of $2n+1$ buckets. Bucket $B[k]$
contains all the indices $i$ with $\Delta[i] = k$}
*maxlen* $\leftarrow 0$
**for** $k \leftarrow -n$ **to** $n$ **do**
    **if** $B[k] \ne \emptyset$
       **then** $M \leftarrow$ MAX($B[k]$)
            $m \leftarrow$ MIN($B[k]$)
            *len* $\leftarrow M - m$
            { *len* is the length of balanced substring $x_{m+1\ldots M}$ }
            **if** *maxlen* $<$ *len*
               **then** *maxlen* $\leftarrow$ *len*
**return** *maxlen*
```

In order to prove that the above algorithm is correct, let ℓ be the length of the longest balanced substring, and let ℓ' be the value returned by the algorithm. Now, whenever *maxlen* is assigned a new value, say $M - m$, we have that $x_{m+1\ldots M}$ is balanced, since $\Delta[m]$

and $\Delta[M]$ belong to the same bucket. Hence

$$\ell' \leq \ell.$$

Let now $x_{h\ldots k}$ be a balanced substring of maximum length. Then $h-1$ and k will be in the same bucket (since $\Delta[h-1] = \Delta[k]$). Moreover, $h-1$ (resp., k) must be the minimum (resp., the maximum) value in the bucket, or otherwise there would be a longer balanced substring. As a consequence, *maxlen* will be compared with $k-h+1=\ell$. Hence

$$\ell' \geq \ell,$$

and the correctness of the algorithm follows.

The algorithm performs two loops of length n and $2n+1$, respectively. In the second loop, we perform no more than $2n$ comparisons altogether to compute the minima and the maxima in all the buckets. Therefore the overall running time is $\Theta(n)$. □

Exercise 4.9 Given a language $L \subseteq \{0,1\}^*$, let DECIDE_L be an algorithm that decides whether $X \in L$ in time $T_L(n)$, with $n = |X|$. Based on algorithm DECIDE_L:

(a) Design and analyze an algorithm DECIDE_L^2 that decides whether $X \in L^2$.

(b) Design and analyze an algorithm DECIDE_$L^\star$ that decides whether $X \in L^\star$.

Answer:

(a) Recall that $X \in L^2$ iff there exist strings $U, V \in L$ such that $X = U \cdot V$, where $\cdot$ denotes concatenation between strings. Given a string X of length n, let $X_{i\ldots j}$ be the substring of X (of length $j-i+1$) spanning from the i-th to the j-th character, for $1 \leq i \leq j \leq n$. Algorithm DECIDE_L^2 will then scan all positions j, with $1 \leq j \leq n-1$, to check (using DECIDE_L) whether both $X_{1\ldots j}$ and $X_{j+1\ldots n}$ are in L . Moreover, DECIDE_L^2 must check whether the empty string $\varepsilon \in L$ (in this case, every string in L is also in L^2). The algorithm follows.

```
DECIDE_$L^2(X)$
$n \leftarrow$ length$(X)$
**if** DECIDE_$L(\varepsilon)$
   **then if** DECIDE_$L(X)$
            **then return true**
```

```
for $j \leftarrow 1$ to $n-1$ do
    if DECIDE_$L(X_{1\ldots j})$
        then if DECIDE_$L(X_{j+1\ldots n})$
            then return true
return false
```

At the j-th iteration, DECIDE_L^2 calls DECIDE_L on strings of length j and $n-j$. Therefore its running time is

$$\begin{aligned} T(n) &= O\left(T_L(n) + \sum_{j=1}^{n-1} T_L(j) + \sum_{j=1}^{n-1} T_L(n-j)\right) \\ &= O\left(T_L(n) + 2\sum_{j=1}^{n-1} T_L(j)\right) \\ &= O(nT_L(n)). \end{aligned}$$

(b) Given a string $X \in \{0,1\}^\star$ of length n, for each pair i,j, with $1 \le i \le j \le n$, define the quantity $m_X[i,j]$ as follows:

$$m_X[i,j] = \begin{cases} \textbf{true} & \text{if } X_{i\ldots j} \in L^\star, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Using a dynamic programming approach, algorithm DECIDE_$L^\star$, on input X, computes $m_X[i,j]$ in increasing order of the substring length $j-i+1$ (which varies from 1 to n). Clearly, DECIDE_$L^\star(X)$ will return **true** iff $m_X[1,|X|] = \textbf{true}$. As customary in dynamic programming, the values $m_X[i,j]$ are stored in a look-up table. The algorithm uses the following characterization of $L^\star$:

$$X \in L^\star \text{ iff either } X \in L \text{ or } X = U \cdot V, \text{ with } U, V \in L^\star \text{ and } |U|, |V| < |X|.$$

Therefore, to compute $m_X[i,j]$, we look up $m_X[i,k]$ and $m_X[k+1,j]$ for all k such that $i \le k \le j-1$. If there is a value $\bar{k}$ for which $m_X[i,\bar{k}] = m_X[\bar{k}+1,j] = \textbf{true}$ (i.e., $X_{i\ldots\bar{k}}$, $X_{\bar{k}+1\ldots j} \in L^\star$), then we set $m_X[i,j] = \textbf{true}$. Otherwise, $m_X[i,j]$ is set to **true** only if DECIDE_$L(X_{i\ldots j}) = \textbf{true}$ (i.e., $X_{i\ldots j} \in L$). Here is the code for DECIDE_$L^\star$.

```
DECIDE_$L^\star$($X$)
$n \leftarrow$ length($X$)
for $\ell \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n - \ell + 1$ do
        $j \leftarrow i + \ell - 1$
        $t \leftarrow$ false
        for $k \leftarrow i$ to $j - 1$ do
            $t \leftarrow t$ or ($m_X[i,k]$ and $m_X[k+1,j]$)
        $m_X[i,j] \leftarrow t$ or DECIDE_$L$($X_{i \ldots j}$)
return $m_X[1,n]$
```

Clearly, DECIDE_$L^\star$ decides $L^\star$. Let us now consider its running time. On a string of length n, DECIDE_$L^\star$ computes $n(n+1)/2$ values $m_X[i,j]$, each taking time $O(T_L(n)+n)$. The second term in the running time comes from the time spent to perform the table look-ups, and must be included to account for the case when $T_L(n) = o(n)$ (e.g., $L = \{0,1\}^\star$). Hence, we obtain an overall running time of $O\left(n^2 T_L(n) + n^3\right)$.

We can devise another algorithm, with a better time and space complexity, by observing that another equivalent characterization of $L^\star$ is the following:

$$X \in L^\star \text{ iff } X = U \cdot V, \text{with } U \in L \text{ and } V \in L^\star,\ |V| < |X|.$$

Let $s[i] = $ **true** iff $X_{i \ldots n} \in L^\star$, for $1 \leq i \leq n$. Moreover, let $s[n+1] = $ **true**. The algorithm works as follows:

```
DECIDE_$L^\star$($X$)
$n =$ length($X$)
$s[n+1] \leftarrow$ true
for $i \leftarrow n$ downto 1 do
    $t \leftarrow$ false
    for $j \leftarrow n$ downto $i$ do
        $t \leftarrow t$ or (DECIDE_$L$($X_{i \ldots j}$) and $s[j+1]$)
    $s[i] \leftarrow t$
return $s[1]$
```

Note that $s[i]$ is computed by means of $n - i + 1$ calls to DECIDE_L and $O(n-i)$ other work. Therefore the overall time complexity is $O\left(n^2 T_L(n)\right)$. This improves on the previous algorithm whenever $T_L(n) = o(n)$. Moreover the new algorithm only requires linear rather than quadratic space. □

Exercise 4.10 Design and analyze a dynamic programming algorithm which, on input two nonnegative integers n and k, with $n \geq k$, outputs $\binom{n}{k}$ in $O(nk)$ time.

Exercise 4.11 Let $\odot$ be a binary operator taking integer operands and defined as

$$a \odot b = (a+b)^2.$$

Let a be the minimum value achievable through a full parenthesization of the expression

$$a_1 \odot a_2 \odot \ldots \odot a_n,$$

where a_i is an integer, for $1 \leq i \leq n$. Design and analyze a dynamic programming algorithm which, on input $a_1, a_2, \ldots, a_n$, first prints an optimal parenthesization and then returns a.

Exercise 4.12 Design and analyze a dynamic programming algorithm which, on input $x \in \{0,1\}^\star$ determines the minimum number p of palindrome substrings $y_1, y_2, \ldots, y_p \in \{0,1\}^\star$ such that $x = y_1 \cdot y_2 \cdot \ldots \cdot y_p$.

Chapter 5

NP Completeness

Exercise 5.1 Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

Answer: Suppose without loss of generality that algorithm A consists of a sequence of calls to subroutines $S_1, \ldots, S_m$, with each subroutine called once in that order. Assume that each subroutine S_i has a (polynomial) running time bounded by $p_i(n)$, with $p_i(n) \le p(n) = n^k$. Note that A might call S_1 on its input, then call S_2 on the return value provided by S_1, and so on until S_m is called on the value provided by S_{m-1}. We show by induction that the largest size of the return value and the worst-case running time of the i-th call are both $O(p^i(n))$, with

$$p^i(n) = \overbrace{p(p(\ldots(p(}^{i \text{ times}} n))\ldots) = n^{k^i}.$$

For $i = 1$, the argument of S_1 is of size at most n. Since S_1 has running time $O(p(n))$, its return value has also size $O(p^1(n)) = O(n^k)$. Assume that the proposition holds for any $i < m$, and consider the $(i+1)$-th call. By the inductive hypothesis, the size of the argument of S_{i+1} has size $O(p^i(n))$. Since S_{i+1} has running time $O(p(n))$, the running time of the $(i+1)$-th call and the size of the return value are both $O\left((p^i(n))^k\right) = O(p^{i+1}(n))$. The inductive thesis follows.

After the m-th call, we have taken time

$$O\left(\sum_{i=1}^{m} p^i(n)\right) = O\left(mp^m(n)\right) = O\left(mn^{k^m}\right),$$

which is polynomial for any constant k and m.

On the other hand, suppose that A simply makes n nested calls to a subroutine S, i.e., on input n, A computes

$$S^n(n) = \overbrace{S(S(\ldots(S(}^{n \text{ times}} n)) \ldots).$$

Suppose that S takes linear time and that its return value is twice as long as its input. It follows that the running time and the size of the return value of the i-th call are both $\Theta(n2^i)$. Therefore, the total running time is

$$\Theta\left(n \sum_{i=1}^{n} 2^i\right) = \Theta(n2^n).$$

□

Exercise 5.2 Prove that the class NP of languages is closed under the following operations:

(a) Union of two languages.

(b) Intersection of two languages.

(c) Concatenation of two languages.

(d) Kleene star of a language.

Answer: Observe that we can re-state the definition of $L \in NP$ (CLR, page 927) equivalently as follows:
Definition A language L is in NP iff there exists a verification algorithm A, and polynomials p, q such that:

- $L = L_A$;
- $\forall x \in L, \exists y$ such that $|y| \leq p(|x|)$ and $A(x, y) = 1$;
- A on input (x, y) halts in time $\leq q(|x| + |y|)$.

In what follows we use the notation $(p + q)(n)$ to denote the polynomial whose value on n is $p(n) + q(n)$.

(a) Let $L_1, L_2 \in NP$, with verification algorithms A_1, A_2 (i.e., $L_1 = L_{A_1}$, $L_2 = L_{A_2}$), and polynomial bounds p_1, q_1, and p_2, q_2, respectively.

Define a new verification algorithm A as follows:

```
$A(x, y)$
**if** $A_1(x, y) = 1$
    **then return** 1
    **else return** $A_2(x, y)$
```

Note that $A(x, y) = 1$ iff $A_1(x, y) = 1$ or $A_2(x, y) = 1$. We have:

1. $L_1 \cup L_2 \subseteq L_A$. Let $x \in L_1 \cup L_2$. Then $x \in L_1$ or $x \in L_2$. If $x \in L_1$, then $\exists y$ such that $A_1(x, y) = 1$. Hence, $A(x, y) = 1$. Otherwise, if $x \in L_2$, then $\exists y$ such that $A_2(x, y) = 1$. Hence, $A(x, y) = 1$. Therefore $x \in L_A$.

2. $L_A \subseteq L_1 \cup L_2$. Let $x \in L_A$. Then $\exists y$ such that $A(x, y) = 1$. This implies that either $A_1(x, y) = 1$ or $A_2(x, y) = 1$, that is, $x \in L_{A_1}$ or $x \in L_{A_2}$. Therefore $x \in L_{A_1} \cup L_{A_2} = L_1 \cup L_2$.

3. $\forall x \in L_A, \exists y$ such that $A(x, y) = 1$. If $x \in L_1$, we have $|y| \leq p_1(|x|)$. If $x \in L_2$, we have $|y| \leq p_2(|x|)$. Threfore $|y| \leq p_1(|x|) + p_2(|x|) = (p_1 + p_2)(|x|)$.

4. A on (x, y) takes time $O((q_1 + q_2)(|x| + |y|))$ and is therefore polynomially bounded.

This proves that $L_1 \cup L_2 \in NP$.

(b) Let $L_1, L_2 \in NP$, with verification algorithms A_1, A_2, and polynomial bounds p_1, q_1 and p_2, q_2, respectively. Moreover, let # be a distinguished character not in the alphabet of the certificates. Define a new verification algorithm A as follows:

```
$A(x, y)$
**if** $y \neq y_1 \# y_2$
  **then return** 0
**if** $A_1(x, y_1) = 1$
  **then if** $A_2(x, y_2) = 1$
        **then return** 1
**return** 0
```

Note that $A(x, y) = 1$ iff $y = y_1 \# y_2$ and $A_1(x, y_1) = A_2(x, y_2) = 1$. We have:

1. $L_1 \cap L_2 \subseteq L_A$. Let $x \in L_1 \cap L_2$. Then $x \in L_1$ and $x \in L_2$. Then, $\exists y_1, y_2$ such that $A_1(x, y_1) = 1$ and $A_2(x, y_2) = 1$. This implies that $A(x, y_1 \# y_2) = 1$. Therefore $x \in L_A$.

2. $L_A \subseteq L_1 \cap L_2$. Let $x \in L_A$. Then $\exists y_1\#y_2$ such that $A(x, y_1\#y_2) = 1$. This implies that $A_1(x, y_1) = 1$ and $A_2(x, y_2) = 1$. Hence $x \in L_{A_1}$ and $x \in L_{A_2}$. Therefore, $x \in L_1 \cap L_2$.

3. $\forall x \in L_A, \exists y$ such that $A(x, y) = 1$. Moreover, since $y = y_1\#y_2$, with $|y_1| \leq p_1(|x|)$ and $|y_2| \leq p_2(|x|)$, we have $|y| = |y_1| + |y_2| + 1 \leq (p_1 + p_2)(|x|) + 1$. Therefore $|y|$ is polynomially bounded.

4. A on (x, y) runs in time $O((q_1 + q_2)(|x| + |y|))$.

This proves that $L_1 \cap L_2 \in NP$.

(c) Given a string x, let $x_{i...j}$ denote the substring of x (of length $j - i + 1$) from the ith to the jth character. Define $x_{i...j} = \varepsilon$ if $i > j$. Let $L_1, L_2 \in NP$, with verification algorithms A_1, A_2, and polynomial bounds p_1, q_1 and p_2, q_2, respectively. Moreover, let # be a distinguished character not in the alphabet of the certificates. Define a new verification algorithm A as follows:

```
$A(x, y)$
if $y \neq y_1\#y_2$
  then return 0
for $k \leftarrow 0$ to $|x|$ do
    if $A_1(x_{1...k}, y_1) = 1$ and $A_2(x_{k+1...|x|}, y_2) = 1$
      then return 1
return 0
```

Note that $A(x, y) = 1$ iff $y = y_1\#y_2$ and $\exists 0 \leq k \leq |x|$ such that $A_1(x_{1...k}, y_1) = 1$ and $A_2(x_{k+1...|x|}, y_2) = 1$. We have:

1. $L_1L_2 \subseteq L_A$. Let $x \in L_1L_2$. Then $\exists 0 \leq k \leq |x|$ such that $x_{1...k} \in L_1$ and $x_{k+1...|x|} \in L_2$. Hence, $\exists y_1, y_2$ such that $A_1(x_{1...k}, y_1) = 1$ and $A_2(x_{k+1...|x|}, y_2) = 1$. So, $A(x, y_1\#y_2) = 1$, i.e. $x \in L_A$.

2. $L_A \subseteq L_1L_2$. This is immediate from our definition of A.

3. $\forall x \in L_A$, $\exists y$ such that $A(x, y) = 1$ and $|y| \leq (p_1 + p_2)(|x|) + 1$.

4. When running A on (x, y), there are at most $|x|+1$ executions of A_1, each taking time $\leq q_1(|x| + |y|)$, and at most $|x| + 1$ executions of A_2, each taking time $\leq q_2(|x| + |y|)$. So, A has a polynomial time bound $O(|x|(q_1 + q_2)(|x| + |y|))$.

This proves that $L_1L_2 \in NP$.

(d) We can exploit the advantage of guessing the right certificate by encoding the substring divisions of x in the certificate y. Namely, let $\#$, $\&$ be distinguished characters not in the alphabet of the certificates. A certificate for a string x in $L^\star$ will be of type

$$y = y_1\#y_2\#\ldots\#y_k\#m_1\&m_2\&\ldots\&m_{k-1},$$

where $1 \leq k \leq |x|$, $m_0 = 0 \leq m_1 \leq \ldots m_{k-1} \leq m_k = |x|$, and, for any i, $1 \leq i \leq k$, y_i is a potential certificate for $x_{m_{i-1}+1\ldots m_i}$'s membership in L. Define a new verification algorithm A as follows:

```
$A(x, y)$
for $k \leftarrow 1$ to $|x|$ do
    $m_0 \leftarrow 0, m_k \leftarrow |x|$
    if $y = y_1\#y_2\#\ldots\#y_k\#m_1\&m_2\&\ldots\&m_{k-1}$
      then $t \leftarrow$ true
           for $i \leftarrow 1$ to $k$ do
               do $t \leftarrow t$ and $A_0(x_{m_{i-1}+1\ldots m_i}, y_i)$
           if $t$ then return 1
return 0
```

$A(x, y) = 1$ iff $\exists k, 1 \leq k \leq |x|$, such that $y = y_1\#y_2\#\ldots\#y_k\#m_1\&m_2\&\ldots\&m_{k-1}$ and, for any i, $1 \leq i \leq k$, $A_0(x_{m_{i-1}\ldots m_i}, y_i) = 1$. We have:

1. $L^\star \subseteq L_A$. Let $x \in L^\star$. Then there is a value k, $1 \leq k \leq |x|$, such that x is the concatenation of strings $x_{m_{i-1}+1\ldots m_i} \in L$, for $1 \leq i \leq k$. Then, for each such i there is a y_i such that $A_0(x_{m_{i-1}\ldots m_i}, y_i) = 1$. Thus, if $y = y_1\#y_2\#\ldots\#y_k\#m_1\&m_2\&\cdots\&m_{k-1}$, we have $A(x, y) = 1$. Therefore, $x \in L_A$.

2. $L_A \subseteq L^\star$. Let $x \in L_A$. Then, there is a $y = y_1\#y_2\#\cdots\#y_k\#m_1\&m_2\&\ldots\&m_{k-1}$ such that $A(x, y) = 1$. By our definition of A, this implies that $x_{m_{i-1}\ldots m_i} \in L$ for any i, $1 \leq i \leq k$. Therefore, $x \in L^\star$.

3. Since there are at most $|x|$ y_i's, with $|y_i| \leq p_0(|x|)$, and at most $|x|$ m_i's, with $|m_i| \leq \log|x|$, and at most $2|x|$ extra-characters in y, we have $|y| = O(|x|(p_0(|x|) + \log|x| + 2))$, which is polynomially bounded.

4. A on (x, y) runs A_0 at most $|x|$ times (because $k \leq |x|$), each taking time $\leq q_0(|x| + |y|)$. Thus, A runs in time $O(|x|q_0(|x| + |y|))$, and is therefore polynomially bounded.

This proves that $L^\star \in NP$. $\square$

Exercise 5.3 Prove that $<_P$ is a transitive relation. That is, for $L_1, L_2, L_3 \subseteq \{0,1\}^\star$,

$$(L_1 <_P L_2 \text{ and } L_2 <_P L_3) \Rightarrow L_1 <_P L_3.$$

Answer: Let $f(x), g(x)$ denote the polynomial-time computable functions that reduce L_1 to L_2 and L_2 to L_3, respectively. Let $h(x) = g(f(x))$. For all strings $x \in \{0,1\}^\star$ we have:

$$\begin{aligned} x \in L_1 \quad &\text{iff} \quad f(x) \in L_2 \\ y = f(x) \in L_2 \quad &\text{iff} \quad g(y) = g(f(x)) \in L_3 \end{aligned}$$

Hence

$$x \in L_1 \quad \text{iff} \quad h(x) = g(f(x)) \in L_3.$$

Note that $h(x) = g(f(x))$ is polynomial-time computable, since it is the composition of two polynomial-time computable functions. This proves that $L_1 <_P L_3$. □

Exercise 5.4 We say that a function f is computable in *quasi linear* time $T_f(n)$ if there are nonnegative constants c and k such that $T_f(n) \leq cn(\log n)^k$. Show that reducibility in quasi linear time is a transitive relation.

Answer: Consider three languages L_1, L_2 and L_3 such that L_1 is reducible in quasi linear time to L_2, and L_2 is reducible in quasi linear time to L_3. By the definition of reduction, there exist reduction functions f from L_1 to L_2 computable in quasi linear time $T_f(n) \leq c_f n(\log n)^{k_f}$, and g from L_2 to L_3 computable in quasi linear time $T_g(n) \leq c_g n(\log n)^{k_g}$. In the previous exercise, we have shown that $h(x) = g(f(x))$ is a reduction function from L_1 to L_3. It remains to show that $h(x)$ is computable in quasi linear time.

Let $y = f(x)$ and $h(x) = g(y)$. Let also $|x| = n$. We have $|y| \leq T_f(|x|) \leq c_f n(\log n)^{k_f}$. Therefore, $h(x) = g(y)$ can be computed in time

$$\begin{aligned} T_h(n) &\leq T_f(n) + T_g(T_f(n)) \\ &= c_f n(\log n)^{k_f} + c_g \left(c_f n(\log n)^{k_f}\right) \left(\log \left(c_f n(\log n)^{k_f}\right)\right)^{k_g} \\ &= (c_g c_f) n \left(\log n\right)^{k_f + k_g} (1 + o(1)) \end{aligned}$$

Therefore, there exist constants $c_h > c_g c_f$ and $k_h = k_f + k_g$ such that $T_h(n) \leq c_h n(\log n)^{k_h}$. This shows that L_1 is reducible in quasi linear time to L_3. □

Exercise 5.5 Prove that $L \leq_P L^c$ iff $L^c \leq_P L$.

Answer:

$$\begin{aligned} f \text{ reduces } L \text{ to } L^c &\Leftrightarrow \forall x \in \Sigma^\star : \ x \in L \text{ iff } f(x) \in L^c \\ &\Leftrightarrow \forall x \in \Sigma^\star : \ (x \notin L) \text{ iff } (f(x) \notin L^c) \\ &\Leftrightarrow \forall x \in \Sigma^\star : \ x \in L^c \text{ iff } f(x) \in L \\ &\Leftrightarrow f \text{ reduces } L^c \text{ to } L. \end{aligned}$$

□

Exercise 5.6 Under the assumption that $P \neq NP$, prove or disprove the following statements:

(a) $\{0,1\}^\star \in P$.

(b) There are NP-complete languages that are regular. Recall that a regular language is one which is accepted by a Deterministic Finite-State Automaton (DFSA).

(c) If L contains an NP-complete subset, then L is NP-complete.

(d) All NP-Complete problems can be solved in time $O\left(2^{p(n)}\right)$, for some polynomial $p(n)$.

(e) The *halting problem* is NP-complete.

(f) The *halting problem* is NP-hard.

Answer:

(a) True $\{0,1\}^\star$ is decided by the following constant-time algorithm:

```
$A_{\{0,1\}^\star}(x)$
return 1
```

(b) False Given a regular language L, any DFSA that accepts L yields a linear-time decision algorithm A_L for L. To see this, associate a distinct label to each state and use conditional jumps to "simulate" transitions. On string x, we will perform exactly $|x|$ jumps before either accepting or rejecting, according to whether the last jump leads to a final or a nonfinal state. This proves that for any regular language L, $L \in P$.

(c) False Counterexample: $\{0,1\}^\star \supset L_{\mathrm{SAT}}$, but Point **(a)** proves that $\{0,1\}^\star \in P$.

(d) True For $L \in NP$, let A_L be the polynomial-time algorithm verifying L and running in time $T_A(|x| + |y|) \leq c_1(|x| + |y|)^h$, where $|y| \leq c_2|x|^k$ when $x \in L$. We can write the following decision algorithm for L:

```
DECIDE_L($x$)
for each $y \in \{0,1\}^\star$, $|y| \leq c_2|x|^k$ do
    if $A_L(x,y) = 1$ then return 1
return 0
```

DECIDE_$L(x)$ returns 1 if and only if there exists a "short" certificate for x, which is the case if and only if $x \in L$. Therefore DECIDE_L decides L. The running time of DECIDE_$L(x)$ is $O\left(|x|^{hk}2^{c_2|x|^k}\right) = O\left(2^{c_2|x|^k+|x|}\right) = O\left(2^{p(|x|)}\right)$.

(e) False Recall that the halting problem corresponds to the following language:

$$L_H = \{y \in \{0,1\}^\star : y = \langle M, x\rangle, M \text{ is a Turing machine which terminates on input } x\}.$$

We know that L_H is an undecidable language. On the other hand, since $NPC \subseteq NP$, Point **(d)** proves that any NP-Complete problem is decidable. Therefore the halting problem cannot be NP-Complete.

(f) True Consider an arbitrary language $L \in NP$, and let DECIDE_L be the exponential decision algorithm for L developed in Point **(d)**. Consider the following program, based on DECIDE_L:

```
$A_L(x)$
if DECIDE_$L(x) = 1$
   then return 1
   else while true do
               { loop forever }
```

A_L either returns 1 or goes into an infinite loop. Let M_{A_L} be a Turing Machine encoding algorithm A_L. Define the following function:

$$f(x) = \langle M_{A_L}, x\rangle$$

Clearly, f is computable in polynomial time, since it takes constant time to encode the Turing Machine and linear time to copy the input string. We now prove that f reduces L

to L_H, the language of the halting problem. We have

$$\begin{aligned} x \in L &\Leftrightarrow \text{DECIDE\_}L(x) = 1 \\ &\Leftrightarrow A_L(x) \text{ terminates} \\ &\Leftrightarrow \langle M_{A_L}, x \rangle \in L_H \end{aligned}$$

We have proved that for any language $L \in NP$, $L <_P L_H$. Hence L_H is NP-Hard. □

Exercise 5.7 Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

Answer: Let $\Phi(x_1, ..., x_m)$ be a boolean formula, and let SAT be a (rather unlikely) subroutine deciding satisfiability in polynomial time $O(p(n))$, where $n \geq m$ is the size of formula Φ. We can find a satisfying assignment to Φ (assuming that there is one, which can be ascertained with one call to SAT) by iteratively finding a truth assignment $s(1)$ for x_1, then finding an assignment $s(2)$ for x_2, and so on until we have an assignment for all the variables. Our invariant will be that after the i-th iteration, the formula $\Phi(s(1), \ldots, s(i), x_{i+1}, \ldots, x_m)$ (i.e., the formula where the variables $x_1, \ldots, x_i$ are substituted with the boolean constants $s(1), \ldots, s(i) \in \{\textbf{false}, \textbf{true}\}$) is satisfiable.

The algorithm works as follows: having found assignments $s(1), s(2), \ldots, s(i-1)$ for the first $i-1$ variables, we call SAT on $\Phi(s(1), \ldots, s(i-1), \textbf{false}, x_{i+1}, \ldots, x_m)$. If this formula is satisfiable, then $s(i) = \textbf{false}$. If the formula is not satisfiable, then $s(i) = \textbf{true}$. In the latest case, $\Phi(s(1), \ldots, s(i-1), \textbf{true}, x_{i+1}, \ldots, x_m)$ must be satisfiable, because our loop invariant/induction hypothesis tells us that $\Phi(s(1), \ldots, s(i-1), x_i, \ldots, x_m)$ is satisfiable (and $\Phi(s(1), \ldots, s(i-1), \textbf{false}, x_{i+1}, \ldots, x_m)$ is not). The algorithm follows:

```
FIND_ASSIGNMENT($\Phi(x_1, x_2, \ldots, x_m)$)
if SAT($\Phi(x_1, x_2, \ldots, x_m)$)= "no"
    then return "formula is not satisfiable"
for $i \leftarrow 1$ to $m$
    do $s[i] \leftarrow$ false
    if SAT($\Phi(s[1], \ldots, s[i], x_{i+1}, \ldots, x_m)$) = "no"
        then $s[i] \leftarrow$ true
return $s$
```

At stage i, it takes polynomial time to prepare $\Phi(s(1), \ldots, s(i), \textbf{false}, x_{i+2}, \ldots, x_m)$; then SAT takes time $p(n)$ to decide the satisfiability of this formula. Since there are $m = O(n)$ iterations, the overall running time is polynomial. □

Exercise 5.8 Consider the following decision problem:

BI_SAT (DOUBLE SATISFIABILITY):
INSTANCE: $\langle \Phi(x_1, x_2, \ldots, x_n) \rangle$, Φ is a boolean formula

QUESTION: Are there two *distinct* satisfying assignments for Φ?

Show that BI_SAT is NP-Complete.

Answer: Let us first show that BI_SAT $\in NP$. Consider the following straightforward algorithm.

```
VERIFY_BI_SAT($x, y$)
if $x \neq \langle \Phi(x_1, x_2, \ldots, x_n) \rangle$
   then return 0
if $y \neq \langle (b_1^1, b_2^1, \ldots, b_n^1), (b_1^2, b_2^2, \ldots, b_n^2) \rangle$
   then return 0
{ the $b_i^j$'s are boolean values that form two
  truth assignments for the variables of $\Phi$ }
same ← true
for $i \leftarrow 1$ to $n$ do same ← same and $(b_i^1 = b_i^2)$
if same then return 0
{ truth assignments must be distinct}
if $\Phi(b_1^1, b_2^1, \ldots, b_n^1)$ and $\Phi(b_1^2, b_2^2, \ldots, b_n^2)$
   then return 1
return 0
```

The algorithm performs two evaluations of Φ plus some extra steps whose number is linear in $|\langle \Phi \rangle|$. Since a boolean formula can be evaluated in time polynomial in its length, VERIFY_BI_SAT verifies BI_SAT in polynomial time.

The second step is to show that BI_SAT is NP-Hard. We show that SAT $<_P$ BI_SAT, where SAT is the Boolean Formula Satisfiability problem.

Let $\Phi(x_1, x_2, \ldots, x_n)$ be a formula, and let x_{n+1} be a new variable. We define our reduction function as follows:

$$f(\langle \Phi(x_1, x_2, \ldots, x_n) \rangle = \langle \Phi(x_1, x_2, \ldots, x_n) \wedge (x_{n+1} \vee \neg x_{n+1}) \rangle.$$

Let us show that

$$\langle \Phi(x_1, x_2, \ldots, x_n) \rangle \in \text{SAT} \Leftrightarrow f(\langle \Phi(x_1, x_2, \ldots, x_n) \rangle) \in \text{BI\_SAT}.$$

Suppose $\Phi(x_1, x_2, \ldots, x_n) \in$ SAT. Then there is a truth assignment $(b_1, b_2, \ldots, b_n)$ to variables $(x_1, x_2, \ldots, x_n)$ satisfying $\Phi(x_1, x_2, \ldots, x_n)$. Since $(x_{n+1} \vee \neg x_{n+1})$ is true for both

$x_{n+1} = \textbf{false}$ and $x_{n+1} = \textbf{true}$, we have that $f(\Phi(x_1, x_2, \dots, x_n))$ is satisfied by the two assignments $(b_1, b_2, \dots, b_n, \textbf{false})$ and $(b_1, b_2, \dots, b_n, \textbf{true})$. Conversely, if $f(\Phi(x_1, x_2, \dots, x_n))$ has two satisfiying assignments $(b_1^1, \dots, b_n^1, b_{n+1}^1)$ and $(b_1^2, \dots, b_n^2, b_{n+1}^2)$ then $\Phi(x_1, x_2, \dots, x_n)$ is clearly satisfied by both assignments $(b_1^1, b_2^1, \dots, b_n^1)$ and $(b_1^2, b_2^2, \dots, b_n^2)$, since, in order for $\Phi(x_1, x_2, \dots, x_n) \wedge (x_{n+1} \vee \neg x_{n+1})$ to be true, both operands $\Phi(x_1, x_2, \dots, x_n)$ and $(x_{n+1} \vee \neg x_{n+1})$ must be true.

Finally, note that f creates a new variable x_{n+1} and computes the encoding of the new formula. Such activity can be accomplished in time polynomial in $|\langle\Phi(x_1, x_2, \dots, x_n)\rangle|$. □

Exercise 5.9 Consider the following decision problem:

M_SAT (MAJORITY SATISFIABILITY):

INSTANCE: $\langle\Phi(x_1, x_2, \dots, x_n)\rangle$, Φ is a boolean formula

QUESTION: Is $\Phi(x_1, x_2, \dots, x_n)$ true for *more* than a half of the possible 2^n input assignments?

Show that M_SAT is NP-hard.

Answer: We show that SAT $<_P$ M_SAT. Given a formula $\Phi(x_1, x_2, \dots, x_n)$, define

$$\begin{aligned} f(\langle\Phi(x_1, x_2, \dots, x_n)\rangle) &= \langle\Phi'(x_1, x_2, \dots, x_n, x_{n+1})\rangle, \\ &\text{with} \quad \Phi'(x_1, x_2, \dots, x_n, x_{n+1}) = \Phi(x_1, x_2, \dots, x_n) \vee x_{n+1}. \end{aligned}$$

Note that f is trivially computable in time polynomial in $|\langle\Phi(x_1, x_2, \dots, x_n)\rangle|$.

Let us show that f reduces SAT to M_SAT. First note that Φ' is satisfied by any of the 2^n assignments $(x_1, x_2, \dots, x_n, \textbf{true})$. If $\Phi \in$ SAT, then there exists an assignment $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ such that $\Phi(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n) = \textbf{true}$. Then, Φ' is also satisfied by the assignment $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n, \textbf{false})$, for a total of at least $2^n + 1 = 2^{n+1}/2 + 1$ satisfying assignments, therefore $f(\langle\Phi\rangle) \in$ M_SAT. Vice versa, if Φ is not satisfiable, then the assignments $(x_1, x_2, \dots, x_n, \textbf{true})$ are all and only those satisfying Φ'. Since these are $2^n < 2^{n+1}/2 + 1$, $f(\langle\Phi\rangle) \notin$ M_SAT. □

Exercise 5.10 Consider the following decision problem:

0-1 IP (0-1 INTEGER PROGRAMMING):

INSTANCE: $\langle A, \boldsymbol{b}\rangle$, where A is an integer $m \times n$ matrix and $\boldsymbol{b}$ is an integer m-vector.

QUESTION: Is there an n-vector $\boldsymbol{x}$ with components in $\{0,1\}$ such that $(A\boldsymbol{x})_i \geq b_i$, for $1 \leq i \leq m$?

Prove that 0-1 IP is NP-complete.

Answer: A certificate for an instance $(A, \boldsymbol{b})$ of 0-1 IP is clearly a 0-1 cols(A)-vector x. Here is the verification algorithm:

```
VERIFY_IP($a, y$)
if ($a \neq \langle A, \boldsymbol{b} \rangle$) or ($y \neq \langle \boldsymbol{x} \rangle$)
   then return 0
$m \leftarrow$ rows($A$)
$n \leftarrow$ cols($A$)
if (length($\boldsymbol{b}$) $\neq m$) or (length($\boldsymbol{x}$) $\neq n$)
   then return 0
for $i \leftarrow 1$ to $m$ do
    for $j \leftarrow 1$ to $n$ do
        if ($a_{i,j}$, $b_i$ noninteger) or ($x_j \notin \{0,1\}$)
           then return 0
$\boldsymbol{c} \leftarrow$ MAT_VEC_MULT($A, \boldsymbol{x}$)
for $i \leftarrow 1$ to $m$ do
    if $c_i < b_i$ then return 0
return 1
```

VERIFY_IP(a, y) is a legal verification algorithm for 0-1 IP, since it returns 1 if and only if a is a well-formed encoding $\langle A, \boldsymbol{b} \rangle$ of an instance of IP, y is a well formed encoding of a 0-1 cols(A)-vector $\boldsymbol{x}$, and $A\boldsymbol{x} \geq \boldsymbol{b}$. Moreover, since matrix-vector multiplication can be performed in polynomial time, the algorithm is clearly polynomial.

To show 0-1 IP is NP-hard, we show that 3-CNF-SAT $\leq_P$ 0-1 IP. Let $\Phi(x_1, x_2, \ldots x_n) = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF made of k clauses. Without loss of generality, in what follows we assume than no clause C_j contains both x_i and $\overline{x_i}$, since in this case C_j is a tautology and can be eliminated from Φ without affecting the value of the formula on any of the assignments. We will say that $x_i = 1$ if x_i is assigned the value **true**, and $x_i = 0$ if x_i is assigned the value **false**. If a boolean variable has value $x_j = \alpha$, with $\alpha \in \{0,1\}$, then the value of $\overline{x_j}$ is $(1-\alpha)$. With this convention, a 0-1 n-vector can be seen as a truth assignment to the n boolean variables of Φ.

From our instance Φ of 3-CNF-SAT, we build an instance $(A, \boldsymbol{b})$ of 0-1 IP in the following way:

- A is a $k \times n$ matrix, where row i is built from clause C_i of Φ in the following way: if boolean variable x_j does not appear in C_i, then $a_{i,j} = 0$. If x_j is a literal in C_i, then $a_{i,j} = 1$. If $\overline{x_j}$ is a literal in C_i, then $a_{i,j} = -1$.

- $\boldsymbol{b}$ is a k-vector such that $b_i = 1 - |\{\text{negative literals in } C_i\}|$.

Given the above definition of A and $\boldsymbol{b}$, for $1 \leq i \leq k$, the i-th inequality $(A\boldsymbol{x})_i \geq b_i$ can be rewritten as follows:

$$\sum_{x_j \in C_i} x_j + \sum_{\overline{x_j} \in C_i} (1 - x_j) \geq 1. \tag{5.1}$$

Assume now that Φ is satisfiable. Then there must exist a truth assignment to the n variables such that satisfies all clauses. Let $(t_1, t_2, \ldots, t_n)$ be the 0-1 n-vector corresponding to such assignment. Then, the sum of the values $\alpha_i^1, \alpha_i^2, \alpha_i^3$ of the three literals in each clause C_i dictated by the t_j's is at least 1. Hence, all inequalities are satisfied at the same time by setting $x_j = 1$ if $t_j = \textbf{true}$, and $x_j = 0$ otherwise. Vice versa, any 0-1 n-vector $(x_1, x_2, \ldots, x_n)$ satisfying all the k inequalities yields a satisfying truth assignment for Φ. Hence, f is a reduction from 3-CNF-SAT to IP. □

Exercise 5.11 Consider the following decision problem:

DF (DISTINCT FORMULAE):
INSTANCE: $\langle \Phi(x_1, x_2, \ldots, x_n), \Psi(x_1, x_2, \ldots, x_n) \rangle$, with $\Phi(x_1, x_2, \ldots, x_n)$ and $\Psi(x_1, x_2, \ldots, x_n)$ boolean formulae.
QUESTION: Is there a truth assignment $(b_1, b_2, \ldots, b_n)$ such that $\Phi(b_1, b_2, \ldots, b_n) \neq \Psi(b_1, b_2, \ldots, b_n)$?

Show that DF is NP-complete.

Answer: We first show that DF $\in NP$. A candidate certificate for DF is a truth assignment to the n variables. The verification algorithm VERIFY_DF first checks whether its first input $x = \langle \Phi(x_1, x_2, \ldots, x_n), \Psi(x_1, x_2, \ldots, x_n) \rangle$, that is, x is a well-formed encoding of an instance of DF; then checks that its second input encodes a truth assignment $(b_1, b_2, \ldots, b_n)$. If this is the case, then the algorithm checks whether $\Phi(b_1, b_2, \ldots, b_n) \neq \Psi(b_1, b_2, \ldots, b_n)$. The running time of VERIFY_DF is clearly polynomial in the size of its inputs. For brevity, we omit the code of the algorithm.

In order to show that DF is NP-Hard, we provide a polynomial-time reduction from SAT to DF. Recall that an instance of SAT is $\langle \Phi(x_1, x_2, \ldots, x_n) \rangle$ and the question is whether Φ is satisfiable, that is, whether there is a truth assignment $(b_1, b_2, \ldots, b_n)$ such that $\Phi(b_1, b_2, \ldots, b_n) = \textbf{true}$. Our reduction function is the following:

$$f(\langle \Phi(x_1, x_2, \ldots, x_n) \rangle) = \langle \Phi(x_1, x_2, \ldots, x_n), \Psi(x_1, x_2, \ldots, x_n) = x_1 \wedge \neg x_1 \rangle.$$

Note that the second formula in $f(\langle\Phi(x_1, x_2, \ldots, x_n)\rangle)$ is a contradiction, therefore its evaluation yields **false** on all truth assignments.

Clearly, f is computable in polynomial time. It remains to show that f is indeed a reduction. Assume that $\langle\Phi(x_1, x_2, \ldots, x_n)\rangle \in$ SAT. Then there is a truth assignment $(b_1, b_2, \ldots, b_n)$ such that $\Phi(b_1, b_2, \ldots, b_n) = \mathbf{true}$. On such assignment, we have

$$\mathbf{true} = \Phi(b_1, b_2, \ldots, b_n) \neq \Psi(b_1, b_2, \ldots, b_n) = \mathbf{false},$$

hence $f(\langle\Phi(x_1, x_2, \ldots, x_n)\rangle) \in$ DF. Vice versa, if $\langle\Phi(x_1, x_2, \ldots, x_n)\rangle \notin$ SAT, then

$$\Phi(b_1, b_2, \ldots, b_n) = \Psi(b_1, b_2, \ldots, b_n) = \mathbf{false},$$

on *all* truth assignments $(b_1, b_2, \ldots, b_n)$. Therefore $f(\langle\Phi(x_1, x_2, \ldots, x_n)\rangle) \notin$DF. □

Exercise 5.12 Consider the following problem:

TWO-CLIQUE :
INSTANCE: $\langle G, h, k\rangle$, with G an undirected graph and $h, k > 0$.

QUESTION: Does G contain two *disjoint* cliques of size h and k?

(a) Show that TWO-CLIQUE is in NP.

(b) Show that TWO-CLIQUE is NP-hard.

Answer:

(a) Consider the following verification algorithm A.

```
$A(x, y)$
if $x \neq \langle G = (V, E), h, k\rangle, h, k > 0$
    then return 0
if $y \neq \langle U_1, U_2\rangle, U_1, U_2 \subset V$
    then return 0
if $|U_1| = h$ and $|U_2| = k$ and $U_1 \cap U_2 = \emptyset$
    then if IS_CLIQUE$(G, U_1)$ and IS_CLIQUE$(G, U_2)$
        then return 1
return 0
```

Subroutine IS_CLIQUE(G, U) checks the adjacency list of G to make sure that U is a clique. Clearly, $L_A =$ TWO-CLIQUE. The length of an accepting certificate y is clearly $O(|V|) = O(|x|)$. Finally, IS_CLIQUE(G, U) can clearly be implemented in polynomial time, therefore A is polynomial.

(b) Let us consider the following reduction function f from CLIQUE to TWO-CLIQUE.

$$f(\langle G = (V, E), h\rangle) = \langle G' = (V \cup \{u\}, E), h, 1\rangle,$$

where $\langle G = (V, E), h\rangle$ is a CLIQUE instance and $u \notin V$. Note that u is an isolated node in G'.

Let us first prove that f is indeed a reduction. If $\langle G = (V, E), h\rangle \in$ CLIQUE then there is a subset K of V which forms an h-clique. Now, K is also an h-clique in G', and $\{u\}$ is a 1-clique in G' disjoint from K. Therefore $\langle G', h, 1\rangle = f(\langle G, h\rangle) \in$ TWO-CLIQUE. Consider now the case $f(\langle G, h\rangle) \in$ TWO-CLIQUE. If $h = 1$, then clearly $\langle G, h\rangle \in$ CLIQUE. Let now $h > 1$. Then there is an h-clique K in $(V \cup \{u\}, E)$. Since u is not adjacent to any other vertex in V, u is not contained in the h-clique. Therefore K is also an h-clique in G. So $\langle G, h\rangle \in$ CLIQUE. Finally, f simply copies G and adds an extra node, therefore f is computable in linear time. $\square$

Exercise 5.13 Consider the following decision problems:

OMC (ODD-MAX-CLIQUE):
INSTANCE: $\langle G = (V, E)\rangle$, with G an undirected graph.

QUESTION: Is the maximum clique size odd?

EMC (EVEN-MAX-CLIQUE):
INSTANCE: $\langle G = (V, E)\rangle$, with G an undirected graph.

QUESTION: Is the maximum clique size even?

(a) Show that OMC $<_P$ EMC.

(b) Show that if EMC is NP-complete then OMC is NP-complete.

Answer:

(a) Let $G = (V, E)$ be an undirected graph, and let $G' = (V', E')$ be defined as follows:

$$\begin{aligned} V' &= V \cup \{\alpha\}, \;\; \alpha \notin V; \\ E' &= E \cup \{\{\alpha, v\} : v \in V\}. \end{aligned}$$

Let now $f(\langle G \rangle) = \langle G' \rangle$. Clearly, $f(\langle G \rangle)$ can be computed in time polynomial in $|V|$ and $|E|$. Let $M \subseteq V$ be a max-clique for G, and $M' \subseteq V'$ be a max-clique for G'. Then, the following two claims hold:

1. $\alpha \in M'$.
   If this were not the case, since $\{\alpha, u\} \in E'$ for each $u \in M'$, $M' \cup \{\alpha\}$ would be a clique of size strictly greater than M', a contradiction.

2. $|M'| = |M| + 1$.
   By Claim 1, $\alpha \in M'$ and $M' - \{\alpha\}$ is a clique for G. Hence

$$|M| \geq |M'| - 1.$$

   $M \cup \{\alpha\}$ is a clique for G'. Hence

$$|M'| \geq |M| + 1$$

From Claim 1 and Claim 2 we conclude that G has an odd max clique iff G' has an even max clique. This proves that f reduces OMC to EMC. Therefore OMC $<_P$ EMC.

(b) Suppose that EMC is NP-complete. Then, from Part **(a)** it follows that OMC $\in$ NP. Therefore, it is sufficient to show that EMC $<_P$ OMC, which requires an identical argument to the one used in Part **(a)**, since function f also reduces EMC to OMC. □

Exercise 5.14 Consider the following decision problem:

IS (INDEPENDENT SET):

INSTANCE:	$\langle G = (V, E), k \rangle$, with G an undirected graph, and $k > 0$.
QUESTION:	Is there a subset $S \subseteq V$, $|S| = k$, with $\{u, v\} \notin E$ for each $u, v \in S$?

(a) Show that IS is *NP*-Complete.

(b) Assume that you are given an $O(|V| + |E|)$ algorithm for IS. Show how to use the algorithm to determine the *maximum* size of an independent set in time $O((|V| + |E|) \log |V|)$.

Answer: In order to prove that IS $\in NP$, consider the following verification algorithm A.

```
$A(x, y)$
if $x \neq \langle G = (V, E), k \rangle, k > 0$
    then return 0
if $y \neq \langle U \rangle, U \subseteq V$
    then return 0
if $|U| = k$
   then if IS_INDEPENDENT$(G, U)$
           then return 1
return 0
```

Subroutine IS_INDEPENDENT(G, U) checks the adjacency list of G to make sure that U is an independent set. Clearly, L_A = IS. The length of an accepting certificate y is $O(|V|) = O(|x|)$. Finally, IS_INDEPENDENT(G, U) can clearly be implemented in polynomial time, therefore A is polynomial.

Next we show that CLIQUE $<_P$ IS, hence IS is NP-hard. Consider the following transformation:

$$f(\langle G = (V, E), k \rangle) = \langle G^c = (V, E^c), k \rangle,$$

where $E^c = \{\{u, v\} : u \neq v \in V \text{ and } \{u, v\} \notin E\}$. Then:

1. Since there is an edge (u, v) in G^c if and only if $(u, v) \notin E$, G^c can be determined by checking all the pairs of vertices in $O(|V|^2)$ time. Therefore f is computable in polynomial time.

2. If G contains a clique $U \subseteq V$ of size k, then no pair of vertices in U will be connected by an edge in G^c. Therefore U is an IS of size k for G^c.

3. If G^c has an IS U of size k, then any pair of distinct vertices in U will be connected by an edge in G, therefore U is a clique of size k for G.

(b) Let DECIDE_IS$(\langle G = (V, E), k \rangle)$ be our (unlikely) $O(|V| + |E|)$ algorithm that decides IS. Based on DECIDE_IS, we can write the following recursive algorithm:

```
MAX_SIZE$(\langle G = (V, E) \rangle, i, j)$
if $i = j$ then return $i$
$middle \leftarrow \lceil (i + j)/2 \rceil$
if DECIDE_IS$(\langle G = (V, E), middle \rangle)$
        then return MAX_SIZE$(\langle G = (V, E) \rangle, middle, j)$
        else return MAX_SIZE$(\langle G = (V, E) \rangle, i, middle - 1)$
```

When we call MAX_SIZE($\langle G = (V, E)\rangle, 1, |V|$), we basically perform a binary search on all possible cardinalities of an independent set. The correctness of the algorithm follows from the observation that there is an independent set of size h iff the size of the maximum independent set is $\geq h$. Therefore a binary search approach can be applied, yieding the desired running time of $O((|V| + |E|)\log |V|)$. $\square$

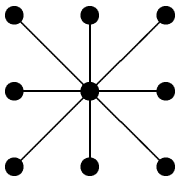
Exercise 5.15 A problem closely related to problem IS, defined in the previous exercise, is the following. Given an undirected graph $G = (V, E)$, a *maximal* independent set is an indepent set S such that, for each $v \in V - S$, $S \cup \{v\}$ is not independent. That is, S cannot be "upgraded" to a larger independent set.

(a) Give an example of a graph where there is a *maximal* independent set of size much smaller than the size of the *maximum* independent set.

(b) Show that the problem of determining a maximal independent set can be solved in polynomial time.

Answer:

(a) Consider the following "star" graph:



Clearly, the node at the center of the star makes a *maximal* independent set by itself, since all other nodes are connected to it. However, the *maximum* independent set contains eight nodes. Note that the above example can be generalized to yield a discrepancy of $\Theta(|V|)$ between the size of a a maximal and a maximum independent set, for any value of $|V|$.

(b) We build our maximal independent set S incrementally as follows. We start from the empty set and perform a linear scan the nodes. We add a new node v to S if $S \cup \{v\}$ is still independent. The algorithm follows.

```
GREEDY_MAXIMAL_INDEPENDENT_SET($G = (V, E)$)
$n \leftarrow |V|$
$S \leftarrow \emptyset$
**for** $i \leftarrow 1$ **to** $n$ **do**
    $indep \leftarrow$ **true**
    **for each** $u \in Adj[v_i]$ **do**
        **if** $u \in S$
            **then** $indep \leftarrow$ **false**
    **if** $indep$
        **then** $S \leftarrow S \cup \{v_i\}$
**return** $S$
```

The set S returned by the above algorithm is an independent set by construction. Let us now prove that S is maximal. Assume, for the sake of contradiction, that S is not maximal. Then, there is a node $v_i \in V - S$ such that $S \cup \{v_i\}$ is an independent set. Note that v_i was not added to S, therefore, at the end of the i-th iteration of the outer loop, variable *indep* was **false**. This means that there was a node $u \in S$ such that $u \in Adj[v_i]$, which contradicts the hypothesis that $S \cup \{v_i\}$ is an independent set.

Note that the outer loop is executed $|V|$ times. During iteration i, we execute the inner loop $|Adj[v_i]|$ times, for a total of $\Theta(|E|)$ iterations altogether. In each iteration, the check $u \in S$ can be performed in $O(\log |S|)$ time (using –say– a binary search tree to store S). Since $|S| \leq |V|$, the running time of the above algorithm is then $O(|V| + |E| \log |V|)$. □

Exercise 5.16 Given undirected graphs $G_1 = (V(G_1), E(G_1))$ and $G_2 = (V(G_2, E(G_2))$, we say that G_1 is isomorphic to G_2 if there is a one-to-one function $\pi : V(G_1) \rightarrow V(G_2)$ such that $\{u, v\} \in E(G_1)$ iff $\{\pi(u), \pi(v)\} \in E(G_2)$. Consider the following decision problem:

SI (SUBGRAPH ISOMORPHISM):

INSTANCE:	$\langle G = (V(G), E(G)), H = (V(H), E(H))\rangle$, with G and H undirected graphs
QUESTION:	Does H contain a subgraph $H' = (V(H'), E(H'))$, with $V(H') \subseteq V(H)$ and $E(H') \subseteq E(H))$ that is isomorphic to G?

Show that SI is NP-complete.

Answer: SI is clearly in NP. Given a string $x = \langle G, H\rangle \in$ SI, a certificate y for SI is $\langle H' = (V(H'), E(H'), \pi\rangle$. Note that π can be represented as a sequence of $|V(G)|$ pairs $(u, \pi(u))$, with $u \in V(G)$, therefore the encoding of y is polynomial in the size of the instance. On input $\langle x, y\rangle$, the verifier first checks that the encodings for the instance and the certificate are well-formed, then checks that H' is indeed a subgraph of H with $|V(G)|$ nodes, and finally checks that for any edge (u, v) in $E(G)$, edge $(\pi(u), \pi(v))$ is in $E(H')$ and viceversa. These checks clearly take time polynomial in the size of $\langle x, y\rangle$. The code of the algorithm is omitted for the sake of brevity.

In order to show that SI is NP-Hard, we show that CLIQUE $<_P$ SI. Recall that an instance of CLIQUE is $\langle G, k\rangle$ and the question is whether G contains a complete subgraph of size k. Let C_k be the graph $(\{1, 2, \ldots, k\}, \{\{u, v\} : 1 \le u \ne v \le k\})$, that is, C_k is the complete graph built on vertices $V(C_k) = \{1, 2, \ldots, k\}$. Our reduction function is

$$f(\langle G, k\rangle) = \langle C_k, G\rangle.$$

Clearly, f is computable in polynomial time. To see that f reduces CLIQUE to SI, note that if G contains a complete subgraph with k nodes, then such subgraph is clearly isomorphic to C_k (all complete graphs with the same number of nodes are isomorphic). Viceversa, if G contains a subgraph isomorphic to C_k, then such subgraph is itself a clique of k nodes (a complete graph can only be isomorphic to another complete graph). This suffices to show that CLIQUE $<_P$ SI, and the claim follows. $\square$

Exercise 5.17 Consider the following decision problem:

HS (HITTING SET):
INSTANCE: $\langle n, m, C_1, C_2, \ldots, C_m, k\rangle$, with $C_i \subseteq \{1, 2, \ldots, n\}$ for $1 \le i \le m$, and $k \le n$.
QUESTION: Is there a subset $S' \subseteq \{1, 2, \ldots, n\}$ with $|S'| = k$ and such that $S' \cap C_i \ne \emptyset$, for $1 \le i \le m$?

Show that HS is NP-complete.

Answer: A candidate certificate for HS is a subset $S' \subseteq \{1, 2, \ldots, n\}$. The verification algorithm first checks whether its first input x is a well-formed encoding $x = \langle n, m, C_1, C_2, \ldots, C_m, k\rangle$ of an instance of HS; then checks that its second input encodes a subset of $\{1, 2, \ldots, n\}$ of cardinality k. If this is the case, the algorithm proceeds to check whether $S' \cap C_i \ne \emptyset$, for $1 \le i \le m$. Each such test can clearly be accomplished in polynomial time. Therefore HS $\in NP$.

In order to show that HS is NP-Hard, we exhibit a reduction from VERTEX_COVER (VC) to HS. Recall that an instance of VC is $\langle G = (V, E), k\rangle$ and the question is whether V contains a subset V' of size k such that each edge in E has at least one of its endpoints in V'.

Let $\pi : V \to \{1, 2, \ldots, |V|\}$ be an arbitrary one-to-one function from V to $\{1, 2, \ldots, |V|\}$. Our reduction function is the following:

$$f(\langle G = (V, E), k\rangle) = \langle |V|, |E|, C_1, C_2, \ldots, C_{|E|}, k\rangle,$$

where $C_i = \{\pi(u), \pi(v)\}$ iff the i-th edge in E is $\{u, v\}$.

Clearly, f is computable in polynomial time. To show that f reduces VC to HS, it is sufficient to observe that, by construction, G contains a vertex cover V' with k nodes if and only if $\pi(V') \subseteq \{1, 2, \ldots, |V|\}$ has nonempty intersection with all the C_i's. The proof follows since $|\pi(V')| = |V'| = k$. □

Exercise 5.18 Show that the HAMILTONIAN CIRCUIT problem for undirected graphs can be polynomially reduced to the following problem:

TSP (TRAVELING SALESMAN PROBLEM):

INSTANCE: $\langle G = (V, E), w, k\rangle$, where G is a *complete*, undirected graph, $w : E \to N$ is a weight function and $k > 0$.

QUESTION: Is there a hamiltonian cycle in G containing all the nodes, whose total weight (i.e., the sum of the weights on the edges of the cycle) is exactly k?

Answer: An instance of HAMILTONIAN CIRCUIT is simply an undirected graph $G = (V, E)$. Consider the following reduction function:

$$\begin{aligned} f(\langle G = (V, E)\rangle) &= < G' = (V, E'), w : E' \to \{1, 2\}, |V| >, \\ E' &= \{\ \{u, v\} : u \neq v, u, v \in V\}, \\ w(e) &= \begin{cases} 1 & \text{if } e \in E' \cap E, \\ 2 & \text{otherwise.} \end{cases} \end{aligned}$$

Clearly, f can be computed in time at most quadratic in $|\langle G\rangle|$. Let us now show that f is indeed a reduction. If $\langle G\rangle \in$ HAMILTONIAN CIRCUIT, then G' contains a circuit whose edges are in E. Under function w, the total weight of such cycle is exactly $|V|$, therefore $f(\langle G\rangle) \in$ TSP. Vice versa, if $\langle G\rangle \notin$ HAMILTONIAN CIRCUIT, no such circuit exists, and any circuit in G' will include edges in $E' - E$, yielding a total weight strictly greater than $|V|$. Therefore $f(\langle G\rangle) \notin$ TSP. □

Exercise 5.19 Given a language $L \in NP$, consider the following three cases.

(a) $L = \Sigma^*$

(b) $L \neq \emptyset, \Sigma^*$ is accepted by a DFSA.

(c) L contains an NP-complete subset.

Under the assumption $P \neq NP$, decide, for each of the above cases, whether 1) L is NP-complete or 2) L is not NP-complete or 3) L might be NP-complete or not. Redo the exercise under the assumption $P = NP$.

Exercise 5.20 Let $L_1, L_2 \in \{0,1\}^*$. Under the assumption that $P \neq NP$, prove or disprove the following propositions:

(a) $L_1 \in P \Rightarrow L_1^c \in NP$.

(b) $L_1 <_P L_2 \Leftrightarrow L_1^c <_P L_2^c$.

(c) $L_1 <_P L_{\text{SAT}} \Rightarrow L_1 \in NPC$.

(d) $L_1 <_P L_{\text{SAT}} \Rightarrow L_1 \in NP$.

(e) $L_1 <_P L_2$ and $L_2 <_P L_1 \Rightarrow L_1, L_2 \in P$.

(f) A reduction function f is a one-to-one correspondence.

(g) If we restricted the input set of CLIQUE to graphs $G = (V, E)$ of degree at most 7, then the resulting subproblem would be in P.

(h) If there is an algorithm for CLIQUE with running time $N^{O(\log N)}$, then every other problem in NP has an algorithm with a running time of the same form.

Exercise 5.21 Consider the following decision problem:

BF_SAT (BALANCED FORMULA SATISFIABILITY):

INSTANCE:	$\langle \Phi(x_1, x_2, \ldots, x_{2n}) \rangle$, Φ is a boolean formula
QUESTION:	Is there a satisfying assignment in which *exactly* n variables have value **false**?

Prove that BF_SAT is NP-Complete.

Exercise 5.22 Consider the following decision problem:

NCBF (NON CONSTANT BOOLEAN FORMULA):
INSTANCE: $\langle \Phi(x_1, x_2, \ldots, x_n) \rangle$, Φ is a boolean formula
QUESTION: Is $\Phi(x_1, x_2, \ldots, x_n)$ a *non constant* function? (i.e., $\Phi \not\equiv$ **false** and $\Phi \not\equiv$ **true**)

Show that NCBF is NP-Complete.

Exercise 5.23 Consider the following decision problem:

CoH (CLIQUE or HAMILTONIAN):
INSTANCE: $\langle G = (V, E), k \rangle$, with G an undirected graph and $k > 0$
QUESTION: Does G contain *either* a clique of size k *or* a hamiltonian circuit?

Show that CoH is NP-Complete.

Exercise 5.24 Consider the following decision problem:

RH (ROOT-HAMILTONIAN):
INSTANCE: $\langle G = (V, E) \rangle$, with G an undirected graph
QUESTION: Does G contain a simple cycle of length at least $\left\lceil \sqrt{|V|} \right\rceil$?

Show that RH is NP-Complete.

Exercise 5.25 Given an undirected graph G, recall that a *hamiltonian path* is a simple path that touches all nodes of G. Consider the following two problems:

HP (HAMILTONIAN PATH):
INSTANCE: $\langle G = (V, E) \rangle$, with G an undirected graph
QUESTION: Does G contain a hamiltonian path?

k-P (k-PATH):
INSTANCE: $\langle G, u, v, k \rangle$, with $G = (V, E)$ an undirected graph, $u \neq v \in V$ and $k > 0$
QUESTION: Does G contain a simple path containing at least k edges from u to v ?

(a) Show that HP is NP-Complete.

(b) Show that k-P is NP-Complete.

(c) Show that HP and k-P are both in P when the graph G is restricted to be acyclic.