Chapter 2

Recurrence Relations and Divide-and-Conquer Algorithms

Consider the following *recurrence*:

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

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 $n > n_0$.

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[1]) $f(n) < n$ each;

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

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

[1]Needless to say, whenever the quantities featured in the recurrence are upper bounds, the resulting solution $T(n)$ will be an upper bound to the running time, while exact values yield the exact running time of the resulting algorithm.

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 $n \leq n_0$, $f^\star(n, n_0)$ would not be defined. Conventionally, we set $f^\star(n, n_0) = -1$ for $n \leq n_0$.

With the above notation, $f^{(\ell)}(n)$ is the size of a single problem instance at the ℓ-th level of recursion, where $\ell = 0$ corresponds 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 level for which Equation (2.1.a) applies. At level $f^\star(n, n_0) + 1$, Equation (2.1.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

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

where if $\ell - 1 < 0$ the value the above product is conventionally taken 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):

$$T(n) = \sum_{\ell=0}^{f^\star(n,n_0)} \left(\left[\prod_{j=0}^{\ell-1} s(f^{(j)}(n)) \right] w(f^{(\ell)}(n)) \right) + \left[\prod_{j=0}^{f^\star(n,n_0)} s(f^{(j)}(n)) \right] T_0,$$

where, for $f^\star(n, n_0) = -1$, the value of the summation in the above expression is conventionally assumed to be 0.

The correctness of the above derivation can be proved by induction on n as follows. Let us start with the base case(s) $n \leq n_0$ and recall that, conventionally, we set $f^\star(n, n_0) = -1$ for $n \leq n_0$. Then, the closed formula correctlt yields T_0, since the summation and the product within evaluate to 0 and 1, respectively.

Assume now that the formula yields the correct value of $T(k)$, for $k < n$ and $n > n_0$.

We have that $T(n) = s(n)T(f(n)) + w(n)$, and, by the inductive hypothesis,

$$T(f(n)) = \sum_{\ell=0}^{f^\star(f(n),n_0)} \left(\left[\prod_{j=0}^{\ell-1} s(f^{(j+1)}(n)) \right] w(f^{(\ell+1)}(n)) \right) + \left[\prod_{j=0}^{f^\star(f(n),n_0)} s(f^{(j+1)}(n)) \right] T_0.$$

Observe that, by the definition of $f^\star$, in case $f(n) \leq n_0$, then $f^\star(f(n), n_0) = -1$, while $f^\star(n, n_0) = 0$. Otherwise, the maximum index k for which $f^{(k)}(f(n)) > n_0$ is clearly one less than the maximum index k for which $f^{(k)}(n) > n_0$, hence, in all cases, $f^\star(f(n), n_0) = f^\star(n, n_0) - 1$. We have:

$$\begin{aligned}
& s(n)T(f(n)) \\
&= s(f^{(0)}(n)) \left\{ \sum_{\ell=0}^{f^\star(n,n_0)-1} \left(\left[\prod_{j=0}^{\ell-1} s(f^{(j+1)}(n)) \right] w(f^{(\ell+1)}(n)) \right) + \left[\prod_{j=0}^{f^\star(n,n_0)-1} s(f^{(j+1)}(n)) \right] T_0 \right\} \\
&= s(f^{(0)}(n)) \left\{ \sum_{\ell=0}^{f^\star(n,n_0)-1} \left(\left[\prod_{j'=1}^{\ell} s(f^{(j')}(n)) \right] w(f^{(\ell+1)}(n)) \right) + \left[\prod_{j'=1}^{f^\star(n,n_0)} s(f^{(j')}(n)) \right] T_0 \right\}
\end{aligned}$$

(by substituting $j' = j + 1$ in the two products)

$$= \sum_{\ell=0}^{f^\star(n,n_0)-1} \left(\left[\prod_{j'=0}^{\ell} s(f^{(j')}(n)) \right] w(f^{(\ell+1)}(n)) \right) + \left[\prod_{j'=0}^{f^\star(n,n_0)} s(f^{(j')}(n)) \right] T_0$$

(by bringing $s(f^{(0)}(n))$ within the two products)

$$= \sum_{\ell'=1}^{f^\star(n,n_0)} \left(\left[\prod_{j'=0}^{\ell'-1} s(f^{(j')}(n)) \right] w(f^{(\ell')}(n)) \right) + \left[\prod_{j'=0}^{f^\star(n,n_0)} s(f^{(j')}(n)) \right] T_0$$

(by substituting $\ell' = \ell + 1$ in the summation.)

Observe now that $w(n)$ can be rewritten as $\left[\prod_{j'=0}^{0-1} s(f^{(j')}(n))\right] w(f^{(0)}(n))$, which is exactly the term of the summation for $\ell' = 0$. Therefore we obtain

$$T(n) = s(n)T(f(n)) + w(n) = \sum_{\ell'=0}^{f^\star(n,n_0)} \left(\left[\prod_{j'=0}^{\ell'-1} s(f^{(j')}(n)) \right] w(f^{(\ell')}(n)) \right) + \left[\prod_{j'=0}^{f^\star(n,n_0)} s(f^{(j')}(n)) \right] T_0$$

and the inductive thesis follows.

Exercise 2.1 Determine $f^*(n, n_0)$ for the following values of $f(n)$ and n_0: **(a)** $(f(n) = n - 1, n_0 = 0)$, **(b)** $(f(n) = n/2, n_0 = 1)$, **(c)** $(f(n) = n^{1/2}, n_0 = 2)$ and **(d)** $(f(n) = n^{1/2}, n_0 = 1)$.

Answer: Observe that function iteration is well defined also for real-valued functions. Therefore, for the sake of generality, we will consider $f(n)$ to be real-valued, with domain $\Re$ in cases **(a)** and **(b)**, and $[0, \infty)$ in cases **(c)** and **(d)**

(a) 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 $f^*(n, n_0) = -1$. Otherwise, letting

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

we get $i < n$. Thus, the largest $i \geq 0$ such that $f^{(i)}(n) = n - i > 0$ is $\lceil n \rceil - 1$. Hence,

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

(b) 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 $f^*(n, 1) = -1$. Otherwise, letting

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

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

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

(c) 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 $f^*(n, 2) = -1$. Otherwise, letting

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

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

Hence,

$$f^*(n,2) = \begin{cases} \lceil \log_2 \log_2 n \rceil - 1 & \text{if } n > 2, \\ -1 & \text{otherwise.} \end{cases}$$

(d) If $n \le 1$, then $f^*(n,2) = -1$. Otherwise,

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

for all i. Thus, $f^*(n,1)$ is undefined, since there infinitely many values of i satisfying $f^{(i)}(n) > 1$. Hence,

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

Observe that the degeneracy in the latter case is due to the fact that we are assuming that $f(n)$ is a real-valued function which may return noninteger values. In fact, no such phenomenon can be observed for integer-valued functions, where the value of $f^*(n, n_0)$ cannot be larger than $n - n_0$, since each iteration of f must decrease the value of its argument by at least one. □

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:

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

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

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

Thus,

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

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

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

Observing that

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

the above expression can be written as

$$
\begin{aligned}
\sum_{i=0}^{\log_2 n - 1} 2^i w(n/2^i) &= a \Bigg[1\cdot \sum_{i=0}^{\log_2 n - 1} 2^i + 3\cdot \sum_{i=0}^{\log_2 n - 2} 2^i + 5\cdot \sum_{i=0}^{\log_2 n - 3} 2^i + \cdots \\
&\quad + (2\log_2 n - 3)\cdot \sum_{i=0}^{1} 2^i + (2\log_2 n - 1)\cdot \sum_{i=0}^{0} 2^i \Bigg] \\
&= a[1(2^{\log_2 n} - 1) + 3(2^{\log_2 n - 1} - 1) + 5(2^{\log_2 n - 2} - 1) + \cdots \\
&\quad + (2\log_2 n - 3)(2^2 - 1) + (2\log_2 n - 1)(2 - 1)] \\
&= a[1(2^{\log_2 n}) + 3(2^{\log_2 n - 1}) + 5(2^{\log_2 n - 2}) + \cdots \\
&\quad + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2) \\
&\quad - (1 + 3 + 5 + \cdots + (2\log_2 n - 3) + (2\log_2 n - 1))] \\
&= a[1(2^{\log_2 n}) + 3(2^{\log_2 n - 1}) + 5(2^{\log_2 n - 2}) + \cdots \\
&\quad + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2)] + \log_2^2 n. \qquad (2.2)
\end{aligned}
$$

Now,

$$
\begin{aligned}
1(2^{\log_2 n}) &+ 3(2^{\log_2 n - 1}) + 5(2^{\log_2 n - 2}) + \cdots + (2\log_2 n - 3)2^2 + (2\log_2 n - 1)2 \\
&= 2(2^{\log_2 n}) + 4(2^{\log_2 n - 1}) + 6(2^{\log_2 n - 2}) + \cdots + (2\log_2 n - 2)2^2 + (2\log_2 n)2 \\
&\quad - [(2^{\log_2 n}) + (2^{\log_2 n - 1}) + (2^{\log_2 n - 2}) + \cdots + 2^2 + 2] \\
&= 2^2(2^{\log_2 n - 1} + 2\cdot 2^{\log_2 n - 2} + 3\cdot 2^{\log_2 n - 3} + \cdots + \log_2 n \cdot 1) - 2(2^{\log_2 n} - 1)
\end{aligned}
$$

$$
\begin{aligned}
&= 2^2(2n - \log_2 n - 2) - 2(n-1) \\
&\quad \text{(see Point (b))} \\
&= 6n - 4\log_2 n - 6. \qquad (2.3)
\end{aligned}
$$

Substituting (2.3) into (2.2), 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,

$$
\begin{aligned}
T(n) &= nT_0 + a(6n - 4\log_2 n - 6 - \log_2^2 n) \\
&= (T_0 + 6a)n - 4a\log_2 n - a\log_2^2 n - 6a.
\end{aligned}
$$

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 $k = 0, 1, 2, \ldots$.

$$S(0) = \sum_{i=1}^{m} x^i = \frac{x^{m+1} - x)}{x-1}. \qquad (2.4)$$

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 (2.4), 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

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

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:

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

□

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:

$$T(n) = n^2 + \log^2 n. \tag{2.5}$$

Since $1^2 + \log^2 1 = 1$, (2.5) 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:

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

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 $i \ge 0$).

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

(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:

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

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 ← length(A)
if n = 2
   then if A[1] ≥ A[2]
            then return A[1]
            else return A[2]
for i ← 1 to √n
     do TMP[i] ← SQRT_MAX (A [ (i − 1) ∗ √n + 1 .. i ∗ √n ])
max ← TMP[1]
for i ← 2 to √n
     do if max < TMP[i]
           then max ← TMP[i]
return max
```

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

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

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:

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

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

$$T(n) = 2n \tag{2.6}$$

is a plausible guess. Let us now try to confirm our guess by using induction. Since $T(1) = 2 = 2 \cdot 1$, Relation (2.6) 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 (2.6) 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))

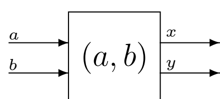
In the above algorithm, when the sequence S has two elements, say $S = (a_1, a_2)$, we simply compare a_1 and a_2 to obtain $\max\{S\}$ and $\min\{S\}$, thus requiring only one comparison. If

$|S| > 2$, 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 MAX(max_1, max_2) and one to compute MIN(min_1, min_2)). Hence,

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

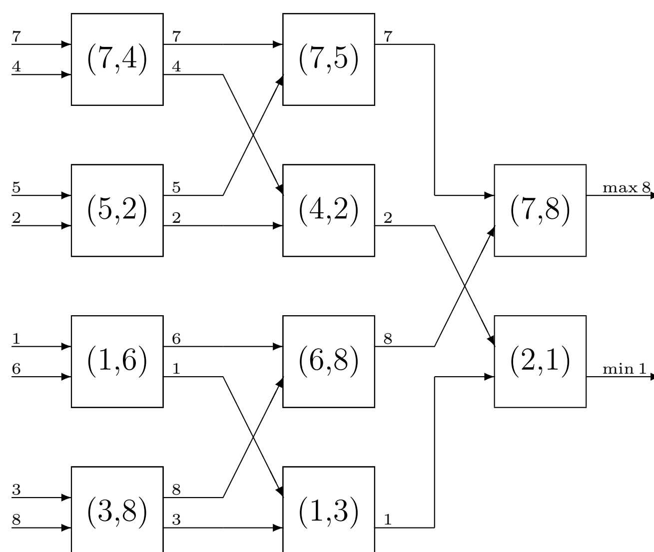
$$\begin{aligned} T(n) &= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=1}^{k} 2^i, \;\; k \geq 1 \\ &= \frac{n}{2}T(2) + 2(2^{\log_2 n - 1} - 1) \\ &= \frac{n}{2} + 2(\frac{n}{2} - 1) \\ &= \frac{3n}{2} - 2. \end{aligned}$$

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. □

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

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

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

$$M_1 \times M_2 = \begin{bmatrix} A \times C + A \times D & 2 \cdot A \times C \\ A \times D + B \times C & A \times C + B \times C \end{bmatrix}, \tag{2.7}$$

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 (2.7) 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 $\text{SUM}(X, Y)$ be an algorithm that returns the sum of two $n \times n$ matrices X and Y, and let $\text{SC_PROD}(c, X)$ 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(M1, M2)
    n ← rows(M1)
    if n = 1 then return M1[1,1] · M2[1,1]
    A ← M1[1..n/2, 1..n/2]
    B ← M1[(n/2+1)..n, 1..n/2]
    C ← M2[1..n/2, 1..n/2]
    D ← M2[(n/2+1)..n, 1..n/2]
    T1 ←REP_MAT_MULT(A, C)
    T2 ←REP_MAT_MULT(A, D)
    T3 ←REP_MAT_MULT(B, C)
    M[1..n/2, 1..n/2] ← SUM(T1, T2)
    M[1..n/2, (n/2+1)..n] ← SC_PROD(2, T1)
    M[(n/2+1)..n, 1..n/2] ← SUM(T2, T3)
    M[(n/2+1)..n, (n/2+1)..n] ← SUM(T1, T3)
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:

$$\begin{cases} T(n) &= 3T(n/2) + 3T_{\mathrm{SUM}}(n/2) + T_{\mathrm{SC_PROD}}(n/2) \\ &= 3T(n/2) + n^2, \qquad n > 1, \\ T(1) &= 1. \end{cases} \tag{2.8}$$

(c) In Recurrence (2.8), 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,

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

(An identical recurrence, with the same motivation, holds for $T_{\mathrm{SC_PROD}}(n)$). By unfolding

the above recurrence for $n = 2^k$, we have

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

However, we cannot substitute these new running times for $T_{\text{SUM}}(n)$ and $T_{\text{SC_PROD}}(n)$ in Recurrence (2.8), 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$(i, \textit{middle})$
SQRT_SORT$(\textit{middle} + 1, j)$
MERGE$(A, i, \textit{middle}, j)$
return

The outer call is clearly SQRT_SORT$(1, \text{length}(A))$. Recall that MERGE$(i, \textit{middle}, j)$ 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:

$$\begin{aligned} T_{\text{SS}}(s) &= 0 \\ T_{\text{SS}}(n) &= 2 \cdot T_{\text{SS}}\left(\frac{n}{2}\right) + n, \quad n > s. \end{aligned} \tag{2.9}$$

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)
$n \leftarrow \text{length}(A)$
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 (2.9), 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 ≥ (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 ← (h + k)div2
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 $\text{root}(T)$.

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 ← 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.,

$$T'(n) = kT'\left(\frac{n}{k}\right) + \Theta(n). \tag{2.10}$$

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 (2.10), 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

$$T(n) \leq n \log n. \tag{2.11}$$

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:

$$T(n) \geq n \log n. \tag{2.12}$$

By combining (2.11) and (2.12) 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 $T(n) = O(n)$.