Divide-and-conquer

1. Describe a *divide-and-conquer* algorithm to find the minimum of a set of \( n \) numbers. Your algorithm should clearly describe how the problem is divided into subproblems. Either give some pseudo-code or describe your algorithm clearly in words. Assume that the set is stored in an array \( A = [A[1], \ldots, A[n]] \).

(Only a divide-and-conquer algorithm will be accepted here. Of course you can find the minimum in time \( O(n) \), but the point is to understand the divide-and-conquer paradigm.)

What is the running time of your algorithm?

**Solution:** I hope that one idea for a solution is pretty clear here. What do you do?

Divide the array \( A \) into two (nearly equal) parts. Find the minimum of each part (recursively). So you’ll have \( \min\left\{ \ell \right\} \) and \( \min\{r\} \) which are the minimums of the “left” and “right” sublists. Then the minimum of the whole list is the smaller of \( \min\left\{ \ell \right\} \) and \( \min\{r\} \).

Suppose, for the sake of convenience that \( n \) is a power of 2, i.e. \( n = 2^k \) for some non-negative integer \( k \).

At level \( i \) (in the recursion) you will have a collection of \( 2^i \) sets, each of size \( n/2^i \). So on the “bottom” level, you’ll end up with \( n/2 \) pairs of numbers. You’ll find the minimum of each pair (using a single comparison) to give you a set of \( n/2 \) numbers as you work your way back up the recursive calls.

Then you’ll perform \( n/4 \) comparisons to end up with \( n/4 \) numbers, and so forth. As you work your way back up the recursive calls, at level \( i \) you will have reduced the list down to a set of \( n/2^{k-i} \) numbers, which you have divided into pairs and you will find the minimum of each pair.

In total, how many comparisons will you perform? As I said, if you use this idea, you’ll perform \( n/2 \) comparisons, then \( n/4 \), then \( n/8, \ldots \), then finally a single comparison to finish finding the minimum. So the total number of comparisons is

\[
\frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{n} = 1 + 2 + 4 + 8 + \cdots + n/2 = 1 + 2^1 + 2^2 + 2^3 + \cdots + 2^{k-1} = 2^k - 1
\]

(using the fact that \( \sum_{i=0}^{k-1} c^i = \frac{c^{k-1} - 1}{c-1} \).)
So the total number of comparison is \(O(2^k) = O(n)\). The time consuming part of the task will be in constructing the sublists, as you do in the MergeSort algorithm. At each level, constructing the smaller lists will take time \(O(n)\). Since you have \(\log n\) levels in the recursion, this will then take time \(O(n \log n)\).

Therefore, overall, this can be done in time \(O(n \log n)\).

(Obviously, this is less efficient than the straightforward method of examining each element in turn, which takes only \(O(n)\) time to find the minimum. As I said, however, the point was to understand the divide-and-conquer idea.)

2. Use the Master Method to find an asymptotic expression for \(T(n)\).

(Assume that \(T(n) = c\) for \(n \leq d\), for some constants \(c > 0\) and \(d \geq 1\).)

**Solution:** Note that in all cases here, you want to consider the function \(n^{\log_b a}\) and compare this to the function \(f(n)\) in a suitable fashion.

(a) \(T(n) = 2T(n/2) + \log n\)

Here we have that \(n^{\log_b a} = n^{\log_2 2} = n\). Recalling that I stated in class that \(\log n = O(n^{1/k})\) for all \(k \geq 1\), we note that \(\log n = O(n^{1/2})\). In other words, there is an \(\varepsilon > 0\) such that \(\log n = O(n^{1/2+\varepsilon})\).

So we’re in Case 1, and \(T(n)\) is \(\Theta(n)\).

(b) \(T(n) = 7T(n/3) + n^2\)

First note that \(\log_3 7 < 2\). Therefore, we can find \(\varepsilon > 0\) such that \(n^2 = \Omega(n^{\log_3 7 + \varepsilon})\). Also, we note that \(7 \cdot (n/3)^2 = (7/9)n^2\), with \(\delta = 7/9\), we see that we’re in Case 3.

Therefore \(T(n)\) is \(\Theta(n^2)\).

(c) \(T(n) = 9T(n/3) + n^3 \log n\)

Since \(\log_3 9 = 2\), we’re comparing \(n^2\) and \(n^3 \log n\). Obviously, we can again find \(\varepsilon > 0\) (like \(\varepsilon = 1\), say) so that \(n^3 \log n = \Omega(n^{2+\varepsilon})\).

So we seem to be in Case 3.

Look at \(a f(n/b)\). This is \(9 \cdot (n/3)^3 \log(n/3) = (1/3)n^3 \log(n/3) \leq (1/3)n^3 \log n\). (Why is this last inequality true?) We can use \(\delta = 1/3\) in our condition for Case 3.

So \(T(n)\) is \(\Theta(n^3 \log n)\).

(d) \(T(n) = 16T(n/2) + (n \log n)^4\)

\(n^{\log_4 16} = n^4\). Since \(f(n) = (n \log n)^4 = n^4 \log^4 n\), we are in Case 2 this time with \(k = 4\) in that Theorem. (That is, \(n^4 \log^4 n\) is obviously \(\Theta(n^4 \log^4 n)\).)
So we conclude that $T(n)$ is $\Theta(n^4 \log^5 n)$. (Note the increase in the exponent on the logarithm that comes from the Master Theorem.)

(e) $T(n) = 4T(n/2) + n^2$
Here we’re comparing $n^{\log_2 4} = n^2$ and $n^2$. Like the previous example right before this one, we’re in Case 2 with $k = 0$ this time.
This means that $T(n)$ is $\Theta(n^2 \log n)$.

(f) $T(n) = 2T(n/2) + 10n$
Since $\log_2 2 = 1$, we’re comparing $n^1 = n$ and $10n$. So we’re again in Case 2 with $k = 0$ in this instance too.
Therefore $T(n)$ is $\Theta(n \log n)$.

(g) $T(n) = 8T(n/2) + 1000n^2$
$\log_2 8 = 3$, so we compare $n^3$ and $1000n^2$. We’re in Case 1 since we can find $0 < \varepsilon < 1$ such that $1000n^2$ is $O(n^{3-\varepsilon})$.
This means that $T(n)$ is $\Theta(n^3)$.

(h) $T(n) = 3T(n/3) + \sqrt{n}$
Again we’re in Case 1 here since we’re comparing $n^{\log_3 3} = n$ and $\sqrt{n}$ (take any $0 < \varepsilon \leq \frac{1}{2}$).
$T(n)$ is $\Theta(n)$.

(i) $T(n) = 2T(n/2) + n \log n$
This satisfies Case 2 with $k = 1$ (why?). So $T(n)$ is $\Theta(n \log^2 n)$.

(j) $T(n) = 3T(n/2) + n^2$
This recurrence satisfies Case 3 with, say, $\delta = \frac{3}{4}$. (Why?)
So $T(n)$ is $\Theta(n^2)$.

(k) $T(n) = T(n/2) + 2^n$
This is also an example of Case 3. Certainly there is $\varepsilon > 0$ such that $2^n$ is $\Omega(n^{(\log_2 2)+\varepsilon}) = \Omega(n^\varepsilon)$. (What is a value of $\delta$ that would work here for the other condition on $f(n) = 2^n$?)
In this case we have that $T(n)$ is $\Theta(2^n)$.

3. Suppose that a queue has $N$ people in it, where each person has a different height from one another. At the front of the queue, you can see $P$ different people, and from the rear of the queue you can see $R$ different people (this is because of the differences in heights, tall people block people shorter than themselves).
How many queues satisfy this property? Let $Q(N, P, R)$ denote the number of queues with this property. You want to find an expression for $Q(N, P, R)$ (given in terms of $N, P,$ and $R$).

**Solution:** First we note a few easy observations.

$Q(N, P, R) = 0$ if $N < 0$.

$Q(1, 1, 1) = 1$ and $Q(N, 1, 1) = 0$ if $N > 1$. (Why?)

$Q(N, N, 1) = 1$ as the people must be lined up in order from shortest to tallest.

$Q(N, P, R) = 0$ if $P + R > N + 1$, and $Q(N, P, R) = 0$ if $N \geq 3$ and $P + R < 3$. (Again, why?)

$Q(N, P, R) = Q(N, R, P)$ since it doesn’t really matter what we call the “front” of the queue and the “back” of the queue.

$Q(N, 2, 1) = (N - 2)!$ as this type of queue must have the second-tallest person at the front of the queue (blocking out all but one other person), and the tallest at the back (since $R = 1$). Since we can arrange the remaining $N - 2$ people in the middle however we like, this gives us $(N - 2)!$ possible queues.

Things start getting more complicated (and interesting) once $P > 2$ (but we’ll still keep $R = 1$). The $P$ people that you can see from the front need not necessarily be the $P$ tallest people. Think about this and see why it’s true. For an example, suppose that there are 10 people, and, from front-to-back of the queue, they are arranged in the order 4; 2; 3; 1; 9; 7; 6; 5; 8; 10. From the front of the queue, you see three different people, since the 4 blocks out the 1; 2; and 3, and the 9 blocks out everyone except for the tallest person on the end.

So for the case when $P > 2$ and $R = 1$, what can we do to simplify the problem, or recast it as a similar problem, but with a smaller number of people? First of all, regardless of the position of the people (with the tallest person at the back, of course, since $R = 1$), note that you’ll always see the second-tallest person somewhere in the queue. That person will block out every person behind him. So select an allowable position for the second-tallest person, choose the appropriate number of people to put behind him in the queue, and arrange those people behind him in some order. Then, ignoring all the of people behind the second-tallest person, you’re left with a smaller problem of a similar type.

In other words, consider the smaller problem where the second-tallest person is now considered to be at the end of the queue. Then you have $N'$ people left over, of which you can see $P - 1$ people from the front and $R = 1$ person from the back of the queue (ignoring all of the
people behind the second-tallest person). There are \( Q(N', P - 1, 1) \)
ways to arrange such a queue of people. The value of \( N' \) depends upon
where the second-tallest person was placed (and so how many people
are left over standing in front of him). Note that the second-tallest
person could have been standing in any position from \( P - 1 \) up to \( N - 1 \)
in the original queue. This means that we can write an expression for
\( Q(N, P, 1) \) in terms of a sum of other \( Q \) values, depending upon where
the second-tallest person was located. This expression can be written
as follows:

\[
Q(N, P, 1) = \sum_{k = P - 1}^{N - 1} Q(k, P - 1, 1) \cdot \binom{N - 2}{k - 1} \cdot (N - k - 1)!
\]

(Computing these numbers will involve a lot of recursive calls. It’s
more efficient to do this via dynamic programming, computing them
from “the bottom up”. But this isn’t the point of this exercise.)

For the most general case when \( P > 2 \) and \( R > 2 \), we use the formu-
las we have above. First of all, we will assume in what follows that
\( P \geq R \), otherwise we will swap \( P \) and \( R \), using our observation that
\( Q(N, P, R) = Q(N, R, P) \). (This will make the expression that comes
out in the end not to involve a “min” or “max” in it.)

In this general case, we have that the tallest person is somewhere “in
the middle” of the queue. This then divides the queue into two parts
that overlap on this common tallest person. What do we do in this
case? The tallest person can be in any position from \( P \) up to position
\( N - R + 1 \) in the queue. (Why?) Then, having determined where the
tallest person is located, we select the remaining people who will go in
front of her in the queue, and the remaining people will go in the back
of the queue. Then these form two subproblems. The people in the
front can be arranged in \( Q(N', P, 1) \) ways and the remaining people in
\( Q(N - N' + 1, 1, R) \) ways. This leads us to the result that when \( P > 2 \)
and \( R > 2 \) (but \( P + R \leq N + 1 \), and assuming that \( P \geq R \)), we have

\[
Q(N, P, R) = \sum_{k = P}^{N - R + 1} Q(k, P, 1) \cdot Q(N - k + 1, R, 1) \cdot \binom{N - 1}{k - 1}.
\]

(As above, computing these numbers could involve a lot of recursive
calls, but can be done more efficiently in a systematic manner.)