Divide and conquer

The idea of an inductive proof: If I know a statement holds for values less than n, then I can use this to prove that it holds also for n and arguing this is probably easier than if I knew nothing about whether the statement holds for values less than n.

Divide and conquer turns this into an algorithmic design principle: If I know how to solve the problem for input sizes less than, then let's take an input of size n, split it into pieces of size less than n, solve the problem on these smaller instances, and then use the obtained solutions to obtain a solution for our input of size n. Hopefully, this is easier than obtaining a solution from scratch, without solutions to smaller instances of the problem.

Example: Mergesort

If $n = 1$, the input array is trivially sorted, so we stop (the base case).

If $n > 1$ (the inductive step), we split the input into two pieces of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Since $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$, we can sort these two pieces by invoking the algorithm recursively on them, and then we're left with the simpler problem of merging two sorted lists.

Another example: Quicksort

If $n=1$, the input is sorted

If $n>1$, we split the input into two non-empty pieces L and R such that all elements in L are no greater than a pivot p and all elements in R are no less than p. The entire input can then be sorted by sorting L and R individually, which we can do by invoking the algorithm recursively on them because $|L|>0$ and $|R|>0$, that is $|L|=n-|R|<n$ and $|R|=n-|L|<n$.

While this type of recursive algorithm is rather elegant, it is rather tricky to analyze. The problem is that "normally" we determine the cost of an algorithm by observing that each operation takes constant time and then bounding the number of times each operation is executed as part of loops in the algorithm. Here we call the entire algorithm recursively.

We can still rather easily write down the running time of such an algorithm by specifying it as a recurrence relation. Let us consider Merge sort. If $T(n)$ denotes its running time, then $T(1) \in O(1)$. For $n>1$, we call the algorithm recursively on two inputs of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. We know exactly how long this takes: $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$ time. Once these two recursive calls return, we spend $O(n)$ time to merge the two

sorted lists they produce. Thus,

$$T(n) = \begin{cases} O(1) & n=1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & n>1 \end{cases}$$

This fully specifies the running time of merge sort but isn't very informative. For example, the three recurrences

$$T(n) = \begin{cases} O(1) & n=1 \\ T(n/2) + O(n) & n>1 \end{cases}$$

$$T(n) = \begin{cases} O(1) & n=1 \\ 2T(n/2) + O(n) & n>1 \end{cases}$$

$$T(n) = \begin{cases} O(1) & n=1 \\ 3T(n/2) + O(n) & n>1 \end{cases}$$

look rather similar but have dramatically different solutions of $T(n) \in O(n)$, $T(n) \in O(n \lg n)$, and $T(n) \in O(n^{\log_2 3}) \subseteq O(n^{1.59})$, respectively. So what we would like to do is to "solve" these recurrences to obtain a closed form for $T(n)$. Before discussing how to do this, let us agree on a convention: Assuming the algorithm terminates for every input, then $T(n) \in O(1)$ for all $n \leq n_0$, where n_0 is some constant. This allows us to write, for example, the merge sort recurrence as

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n),$$

which means

$$T(n) = \begin{cases} O(1) & n \le n_0 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & n > n_0 . \end{cases}$$

Solving recurrences

We discuss three techniques for solving recurrences, that is, turning them into closed forms:

- Induction
- The Masks Theorem
- Recursion trees

Induction is the most general of the three but requires us to make an educated guess what the solution is, which we then verify. The guess can be obtained using recursion trees or experience.

The Masks Theorem is less general but covers many common recurrences and is very convenient to use.

Recursion trees are good for visualizing how a recurrence expands. This helps to obtain a guess of the solution to be proved using induction or, if done rigorously enough, directly gives a solution.

Induction

Back to merge sort. We guess that $T(n) \in O(n \lg n)$, that is, $T(n) \le c n \lg n$ for some $c > 0$, $n_0 > 0$ and all $n \ge n_0$. We prove this by induction:

For $2 \le n \le 3$, we have $T(n) \in O(1)$ and $n \lg n \ge n$. Thus, we can find a constant c such that $T(n) \le c n \lg n$.

For $n \ge 4$, we have $T(n) \le T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + dn$, for some $d > 0$. By the inductive hypothesis, this gives

$$\begin{aligned}
T(n) &\le c\lceil n/2 \rceil \lg \lceil n/2 \rceil + c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + dn \\
&\le n(c \lg \lceil n/2 \rceil + d) \\
&\le n(c \lg (n/2+1) + d) \\
&= n\left(c \lg \left[\frac{n/2+1}{n/2} \cdot \frac{n}{2}\right] + d\right) \\
&\le n\left(c \lg \left(\frac{3}{4}n\right) + d\right) \quad \left(\text{because } \frac{n/2+1}{n/2} \le \frac{3}{2} \text{ for } n \ge 4\right) \\
&\le n(c \lg n - c \lg \tfrac{4}{3} + d) \\
&\le cn \lg n \quad \text{as long as } c \lg \tfrac{4}{3} \ge d, \\
&\qquad c \ge \frac{d}{\lg 4/3}.
\end{aligned}$$

What about the other two recurrences above?

- $T(n) = T(n/2) + O(n)$. We claim that this gives $T(n) \in O(n)$, that is, $T(n) \le cn$.

  For $n \le 2$, we have $T(n) \in O(1)$, so $T(n) \le cn$ for a sufficiently large $c > 0$.

For $n > 2$, we have $T(n) \leq T(n/2) + dn$, which gives

$$T(n) \leq \frac{cn}{2} + dn$$

$$= \left(\frac{c}{2} + d\right) n$$

$$\leq cn \quad \text{as long as } c \geq 2d.$$

- $T(n) = 3T(n/2) + O(n)$. We claim that this gives $T(n) \in O(n^{\log_2 3})$.

  For $1 \leq n \leq 2$, we have $T(n) \in O(1)$ and $n^{\log_2 3} \geq n$, so $T(n) \leq cn^{\log_2 3}$, for sufficiently large $c > 0$.

  For $n > 2$, we get

$$T(n) \leq 3c \cdot \left(\frac{n}{2}\right)^{\log_2 3} + dn$$

$$= \frac{3c\, n^{\log_2 3}}{2^{\log_2 3}} + dn$$

$$= cn^{\log_2 3} + dn,$$

  but that's not what we wanted to prove. The problem is that our claim wasn't strong enough. Let's try again. We claim that $T(n) \leq c_1 n^{\log_2 3} - c_2 n$, for some $c_1 > c_2 > 0$.

  For $1 \leq n \leq 2$, we have $c_1 n^{\log_2 3} - c_2 n \geq (c_1 - c_2) n \geq T(n)$ because $T(n) \in O(1)$, so we can choose c_1 large enough to make this true.

For $n > 2$, we get

$$T(n) \le 3\left[c_1\left(\frac{n}{2}\right)^{\log_2 3} - c_2\frac{n}{2}\right] + dn$$

$$= \frac{3c_1 n^{\log_2 3}}{2^{\log_2 3}} - \left(\frac{3}{2}c_2 - d\right)n$$

$$= c_1 n^{\log_2 3} - c_2 n \text{ as long as}$$

$$\frac{3}{2}c_2 - d \ge c_2, \text{ that is } c_2 \ge 2d.$$

So, why were we able to prove $T(n) \le c_1 n^{\log_2 3} - c_2 n$ when we weren't able to prove the weaker claim $T(n) \le cn^{\log_2 3}$? We're using induction: a stronger claim gives us a stronger inductive hypothesis to work with. This doesn't always work, but it's often the trick to try when we fail to prove a bound we "know" to be true. The way our initial attempt failed also informed us how to choose an appropriate stronger claim: we had a linear term we couldn't get rid of, so let's try to strengthen the claim by subtracting a linear term.

- Final exercise: We know binary search takes $O(\lg n)$ time. Its recurrence is

$$T(n) = T(n/2) + O(1)$$

We claim that $T(n) \le c \lg n$ for some $c > 0$.

For $2 \le n < 4$, we have $T(n) \in O(1)$ and $\lg n \ge 1$, so $T(n) \le c \lg n$, for large enough $c > 0$.

For $n \geq 4$, we have

$$\begin{aligned} T(n) &\leq T(n/2) + d \\ &\leq c \lg\left(\frac{n}{2}\right) + d \\ &= c \lg n - c + d \\ &\leq c \lg n \quad \text{as long as } c \geq d, \end{aligned}$$

The Masks Theorem

Theorem: Let $T(n)$ be given by the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

(i) If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$, for some $\varepsilon > 0$, and $a f(n/b) \leq c f(n)$, for some $c < 1$, then $T(n) \in \Theta(f(n))$.

(ii) If $f(n) \in O(n^{\log_b a - \varepsilon})$, for some $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.

(iii) If $f(n) \in \Theta(n^{\log_b a} \lg^c n)$, for some $c \geq -1$, then $T(n) \in \Theta(f(n) \lg n)$.

Before proving this, let's look at a few examples:

- Merge sort again: Ignoring floors and ceilings (which we can almost always do), we have

$$T(n) = 2T(n/2) + \Theta(n)$$

$n^{\log_2 2} = n$, so $f(n) \in \Theta(n^{\log_2 2})$ and

Case (iii) applies. So $T(n) \in \Theta(n \lg n)$.

- Binary search:

$$T(n) = T(n/2) + \Theta(1)$$

$n^{\log_2 1} = n^0 = 1$, so $f(n) \in \Theta(n^{\log_2 1})$, so we have Case (iii) again, $T(n) \in \Theta(\lg n)$.

- Selection:

$$T(n) = T\left(\frac{7n}{10}\right) + \Theta(n)$$

$n^{\log_{10/7} 1} = 1$, so $f(n) \in \Omega(n^{\log_{10/7} 1 + \varepsilon})$ for $\varepsilon = 1$. Case (i) applies and we have $T(n) \in \Theta(n)$.

- Matrix multiplication:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$\log_2 7 > 2$, so $f(n) \in O(n^{\log_2 7 - \varepsilon})$ for $\varepsilon = \log_2 7 - 2 > 0$, so Case (i) applies and $T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Proof of the Masks Theorem: Let us first spell out the recurrence:

$$T(n) = \begin{cases} \Theta(1) & n \le n_0 \\ aT\left(\frac{n}{b}\right) + f(n) & n > n_0 \end{cases}.$$

Then $$T(n) = \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta(n^{\log_b a}).$$

This is easy to prove by induction:

For $\frac{n_0}{b} < n \le n_0$, we have $\lceil \log_b \frac{n}{n_0} \rceil = 0$, so the sum is empty. Since $cn \le n_0$, we have $n^{\log_b a} \in \Theta(1)$ and $T(n) \in \Theta(1) = \Theta(n^{\log_b a})$.

For $n > n_0$, we have

$$\begin{aligned} T(n) &= a\,T\left(\frac{n}{b}\right) + f(n) \\ &= a\left[\sum_{i=0}^{\lceil \log_b \frac{n/b}{n_0} \rceil - 1} a^i f\left(\frac{n/b}{b^i}\right) + \Theta\left(\left(\frac{n}{b}\right)^{\log_b a}\right)\right] + f(n) \\ &= \sum_{i=0}^{\lceil \log_b \frac{n}{bn_0} \rceil - 2} a^{i+1} f\left(\frac{n}{b^{i+1}}\right) + \Theta\left(a \cdot \frac{n^{\log_b a}}{a}\right) + f(n) \\ &= \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta(n^{\log_b a}) \end{aligned}$$

Now we can consider the three cases:

(i) $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ and $a f(n/b) \le c f(n)$

Then $$\begin{aligned} \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) &\le \sum_{i=0}^{\infty} a^i f\left(\frac{n}{b^i}\right) \\ &\le \sum_{i=0}^{\infty} c^i f(n) \\ &= \frac{1}{1-c} f(n) \in \Theta(f(n)) \end{aligned}$$

Since $f(n) \in \Omega(n^{\log_b a + \varepsilon})$, we thus have

$T(n) \in O(f(n))$. Since $T(n) \geq f(n)$, we also have $T(n) \in \Omega(f(n))$, so $T(n) \in \Theta(f(n))$.

(ii) $f(n) \in O(n^{\log_b a - \varepsilon})$.

Then $\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \varepsilon}$

$$= c n^{\log_b a - \varepsilon} \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} b^{\varepsilon i}$$

$$\leq c n^{\log_b a - \varepsilon} \int_0^{\lceil \log_b \frac{n}{n_0} \rceil} b^{\varepsilon i} \, di$$

$$= c n^{\log_b a - \varepsilon} \left[b^{\varepsilon i} \varepsilon \ln b \right]_0^{\lceil \log_b \frac{n}{n_0} \rceil}$$

$$\leq c n^{\log_b a - \varepsilon} \varepsilon \ln b \, n^{\varepsilon}$$

$$\in O(n^{\log_b a})$$

Thus, $T(n) \in \Theta(n^{\log_b a})$

(iii) $f(n) \in \Theta(n^{\log_b a} \lg^c n)$, for some $c \geq -1$.

Then $\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) \in \Theta\left(\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \lg^c \left(\frac{n}{b^i}\right) \right)$

$$= \Theta\left(n^{\log_b a} \cdot \sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} \lg^c \left(\frac{n}{b^i}\right) \right)$$

Now $\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} \lg^c \left(\frac{n}{b^i}\right) \leq \lg^c n \cdot \left(\log_b \frac{n}{n_0} - 1 \right) \in O(\lg^{c+1} n)$

Also, $\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} \lg^c \left(\frac{n}{b^i}\right) \geq \sum_{i=0}^{\lceil \log_b \sqrt{n} \rceil - 1} \lg^c \left(\frac{n}{b^i}\right) \geq \lceil \log_b \sqrt{n} \rceil \cdot \lg^c (\sqrt{n})$

$$= \left\lceil \frac{1}{2} \log_b n \right\rceil \cdot \frac{1}{2^c} \cdot \lg^c n \in \Omega(\lg^{c+1} n).$$

Thus, $\sum_{i=0}^{\lceil \log_b \frac{n}{n_0} \rceil - 1} a^i f\left(\frac{n}{b^i}\right) \in \Theta(n^{\log_b a} \lg^{c+1} n) = \Theta(f(n) \lg n)$

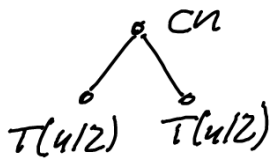
Since $f(n) \lg n \in \Omega(n^{\log_b a})$, this shows that

$T(n) \in \Theta(f(n) \lg n)$. □

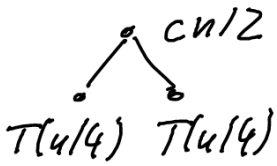
Recursion trees

Using recursion trees, we simply expand the recurrence and sum up the parts of the expansion. Let's look at merge sort again:

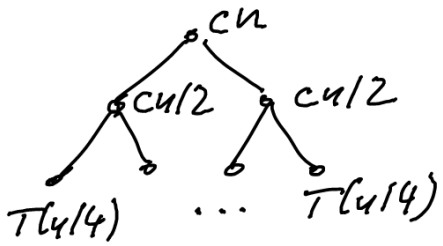
$T(n) \leq 2T(n/2) + cn$, for some $c > 0$. This gives us the tree for $T(n)$:



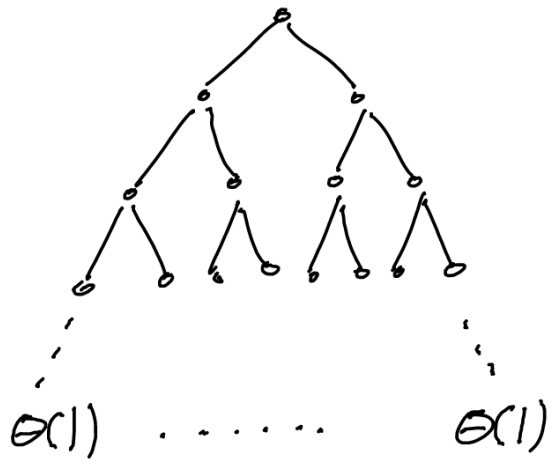
Now we can replace $T(n/2)$ with its own tree



which gives



If we continue this until we hit the base case $T(n) \in \Theta(1)$ for $n \leq n_0$, we obtain:



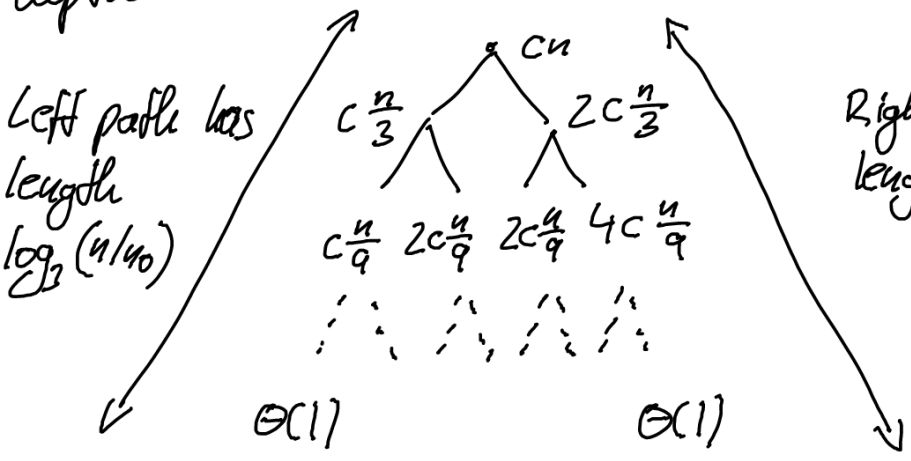
Level	Input size	Cost per level
0	n	cn
1	$n/2$	$2 \cdot c\frac{n}{2} = cn$
2	$n/4$	$4 \cdot c\frac{n}{4} = cn$
3	$n/8$	⋮
⋮	⋮	
$\lg \frac{n}{n_0}$	n_0	$\Theta(n/n_0)$

Total cost is then $\leq cn \lg \frac{n}{n_0} + \Theta(n/n_0) \in O(n \lg n)$.

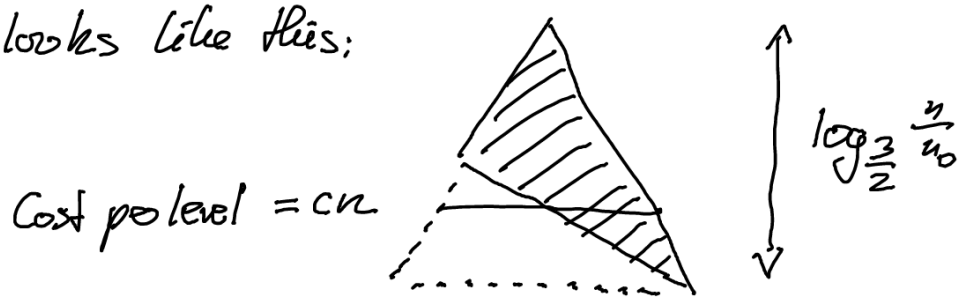
Would things change dramatically if we skewed the split:

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) \; ?$$

No, but we have to apply the recursion tree method more carefully because not all leaves have the same depth:



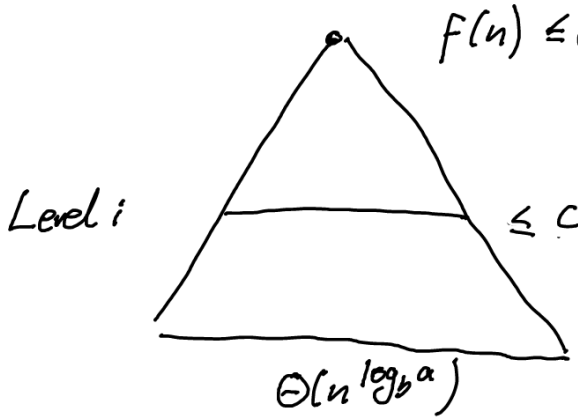
So our tree looks like this:



We still get $T(n) \in O(n \lg n)$.

Let's have a look at the Masters Theorem again:

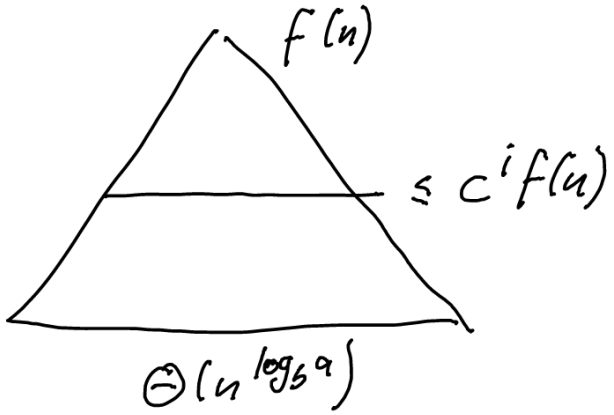
If $f(n) \in O(n^{\log_b a - \varepsilon})$, we get



That is, the cost of the leaves dominates.

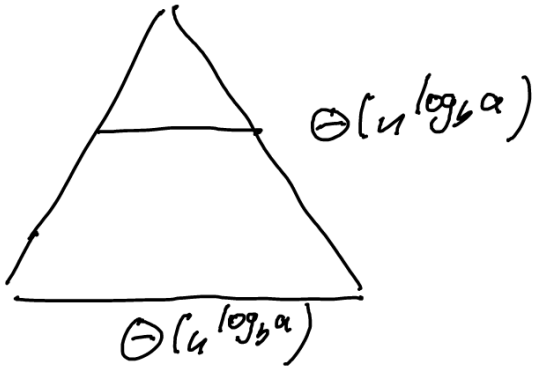
If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ and $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$,

we get



That is, the root dominates the cost of the whole tree.

Finally, if $f(n) \in \Theta(n^{\log_b a})$, we get



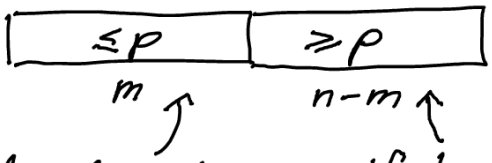
All levels cost the same and we have $\Theta(\lg n)$ of them.

Selection

Given an array of n elements, we want to find the kth smallest one. For $k=1$, we get the minimum, for $k=n$, we get the maximum, for $k=n/2$, we get the median.

The trivial solution is to sort the input and report the kth element. This takes $O(n \log n)$ time. For $k=1$ or $k=n$, we can solve the problem in linear time. Can we achieve the same for arbitrary k?

Assume we sort using Quicksort. Do we really have to sort the whole input to find the kth smallest element?



If $k \leq m$, the element we're looking for is here and is the kth smallest element.

If $k > m$, the element we're looking for is here and is the $(k-m)$th smallest element

There is no need to sort the side we're not interested in. This gives the following selection algorithm:

```
QuickSelect(A, ℓ, r, k)
if r ≤ ℓ then return A[ℓ]
p = FindPivot(A, ℓ, r)
m = Position(A, ℓ, r)
if k ≤ m - ℓ + 1 then return QuickSelect(A, ℓ, m, k)
                 else return QuickSelect(A, m+1, r, k-m)
```

The correctness of this procedure is obvious. What about its running time? We have

$$T(n) \leq O(n) + T(\max(m, n-m))$$

If m is always 1 or m is always $n-1$, we obtain

$$T(n) = cn + c(n-1) + c(n-2) + \ldots = \frac{cn(n+1)}{2} \in \Theta(n^2)$$

If m is always $n/2$, we obtain

$$T(n) = \Theta(n) + T(n/2),$$

which has the solution $T(n) \in \Theta(n)$ by the Masks Theorem. So, what we want to ensure is that the pivot p we find is the median, but that's just selection again. How do we break the cycle?

Option 1: Pick a random element and use it as pivot. This is easy, really fast, and, even though it can fail, it never does in practice.

Option 2: Find a way to select an approximate median by applying selection to only a subset of the input elements.

```
FindPivot(A, ℓ, r)
n' = ⌈(r-ℓ+1)/5⌉ - 1
for i = 0 to n'-1 do
    Sort A[ℓ+5i, ℓ+5i+4] using insertion sort
    B[i] = A[ℓ+5i+2]
B[n'] = A[5n']
return QuickSelect(B, 0, n', ⌈n'/2⌉)
```

Observation: FindPivot(A, ℓ, r) takes $T(\lceil n/5 \rceil) + O(r-\ell+1)$ time, where $T(n)$ is the running time of QuickSelect.

Lemma: Let p be the element returned by FindPivot(A, ℓ, r), and let $n = r-\ell+1$. Then at least $\frac{3n}{10} - 5$ elements are less than p and at least $\frac{3n}{10} - 5$ elements are greater than p, assuming all elements in A are distinct.

The assumption of distinct elements is not a problem because Hoare's partition algorithm divides the elements equal to p evenly among the left and right halves. The lemma is simply easier to formulate and prove with this assumption.

Proof: There are $n' = \lceil n/5 \rceil$ elements in B. There are at least $\lceil n'/2 \rceil - 1$ elements in B less than the pivot p returned, and there are $\lceil n'/2 \rceil - 1$ elements in B greater than p. Let us focus on the elements greater than p. The argument for counting elements less than p is analogous.

of the elements in B greater than p at most one is not the median of a group of 5 elements in A. For each element greater than p that is the median of a group of 5 elements in A, at least 3 elements in this group are greater than p. Thus, A contains at least $3(\lceil n'/2 \rceil - 2) + 1 = 3\lceil n'/2 \rceil - 5$ elements greater than p. Substituting $n' = \lceil n/5 \rceil$, we obtain that at least $3\lceil \lceil n/5 \rceil / 2 \rceil - 5 \geq 3n/10 - 5$ elements in A are greater than p. □

Lemma: QuickSelect takes $O(n)$ time.

Proof: Apart from calling itself recursively on at most $7n/10 + 5$ elements and from calling FindPivot, QuickSelect takes $O(n)$ time. FindPivot takes $O(n) + T(\lceil n/5 \rceil)$ time. This gives the recurrence

$$T(n) \leq cn + T(\lceil n/5 \rceil) + T(7n/10 + 5)$$

for some constant $c > 0$. We claim that $T(n) \leq dn$ for some constant $d > 0$. For $1 \leq n < 120$, we have $T(n) \in \Theta(1)$, so $T(n) \leq dn$ for d large enough.

For $n \geq 120$, we have $1 \leq \lceil n/5 \rceil < n$ and $1 \leq 7n/10 + 5 < n$, so we can apply the inductive hypothesis to $T(\lceil n/5 \rceil)$ and $T(7n/10 + 5)$. This gives

$$\begin{aligned} T(n) &\leq cn + d\lceil n/5 \rceil + d(7n/10 + 5) \\ &\leq cn + d(n/5 + 1) + d(7n/10 + 5) \\ &= (c + d/5 + \tfrac{7d}{10})n + 6d \\ &\leq c + d/5 + \tfrac{7d}{10} + \tfrac{d}{20} n \end{aligned}$$

$$= \left(c + \frac{4+14+1}{20} d\right) n$$
$$= \left(c + \frac{19}{20} d\right) n$$
$$\leq dn \quad \text{provided } d \geq 20c. \qquad \square$$

Matrix Multiplication

The naive algorithm for multiplying two $n \times n$ matrices takes $O(n^3)$ time:

```
Multiply (A, B, n)
for i = 1 to n do
    for j = 1 to n do
        C[i,j] = 0
        for k = 1 to n do
            C[i,j] = C[i,j] + A[i,k] B[k,j]
```

Using divide and conquer does not seem to help at first, but it does in the end. Assume for the sake of simplicity that $n = 2^k$ for some $k \in \mathbb{N}$. (We can always pad A and B with rows and columns of zeroes to ensure this without more than quadrupling their size).

If $k = 0$, then $n = 1$ and multiplying A with B becomes basic scalar multiplication:

$$C_{11} = a_{11} \cdot b_{11}$$

If $k > 0$, we can split A, B and $C = A \times B$ into four submatrices of size $(n/2) \times (n/2)$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

These submatrices satisfy the equalities

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

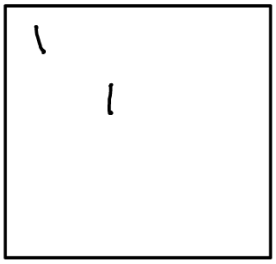
Each of these 8 matrix products takes $T(n/2)$ time. Adding two $(n/2) \times (n/2)$ matrices takes $\Theta(n^2)$ time. Thus, we obtain the recurrence

$$T(n) = 8T(n/2) + \Theta(n^2)$$

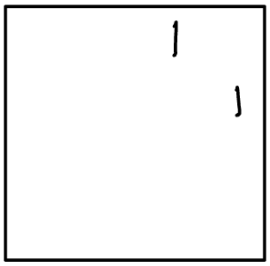
By the Masks Theorem, this gives $T(n) = \Theta(n^3)$. How can we do better? We start by visualizing the computation of C_{11}, C_{12}, C_{21} and C_{22} as a vector-matrix-vector product:

$$C_{ij} = \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix} \cdot \begin{bmatrix} M_{ij} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{21} & B_{12} & B_{22} \end{bmatrix}$$

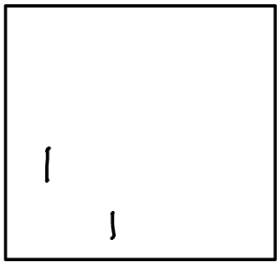
This gives the matrices



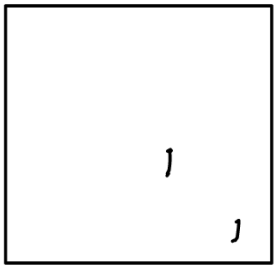
M_{11}



M_{12}



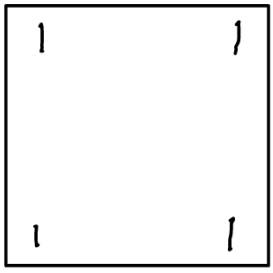
M_{21}



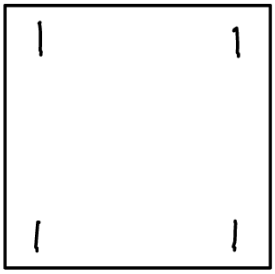
M_{22}

(All other entries are 0.)

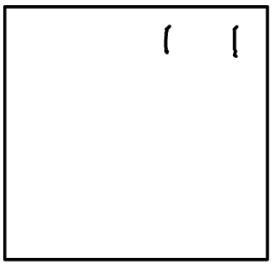
Each of these matrices corresponds to two matrix multiplications over A and B. On the other hand a matrix such as this one



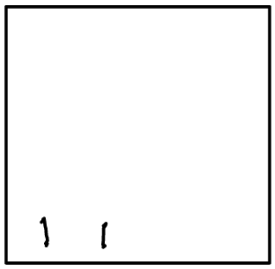
represents a single matrix multiplication over A and B: $(A_{11} + A_{22})(B_{11} + B_{22})$! We now define 7 such matrices such that $M_{11}, M_{12}, M_{21}, M_{22}$ are linear combinations of these matrices



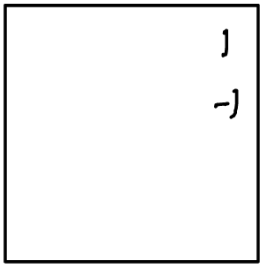
M_1



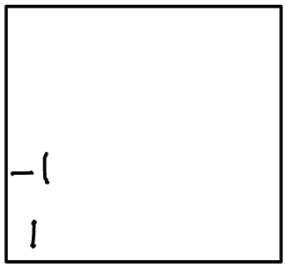
M_2



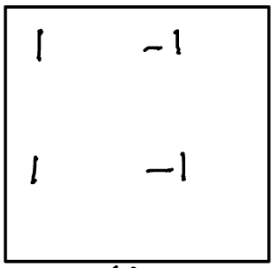
M_3



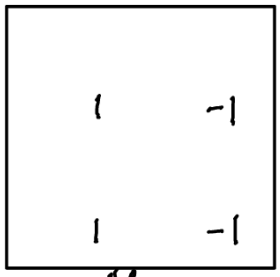
M_4



M_5

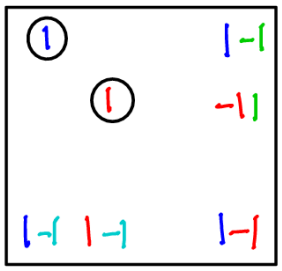


M_6

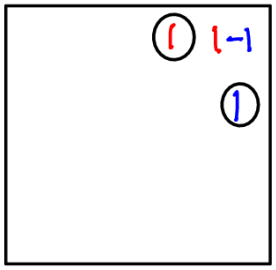


M_7

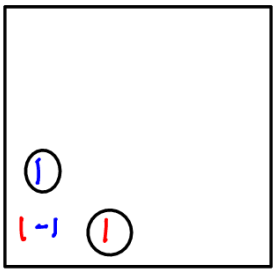
Then $M_{11} = M_1 + M_7 - M_4 - M_3$



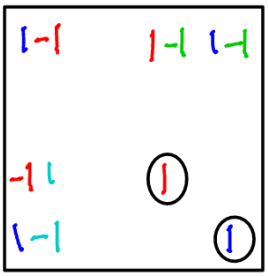
$M_{12} = M_2 - M_4$



$M_{21} = M_3 - M_5$



$M_{22} = M_1 - M_6 - M_2 - M_5$



Now we are essentially done because

$$C_{11} = M_1 + M_7 - M_4 - M_3$$
$$C_{12} = M_2 - M_4$$
$$C_{21} = M_3 - M_5$$
$$C_{22} = M_1 - M_6 - M_2 - M_5$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = A_{11}(B_{12} + B_{22})$$
$$M_3 = A_{22}(B_{11} + B_{21})$$
$$M_4 = (A_{11} - A_{12})B_{22}$$
$$M_5 = (A_{22} - A_{21})B_{11}$$
$$M_6 = (A_{11} + A_{21})(B_{11} - B_{12})$$
$$M_7 = (A_{12} + A_{22})(B_{21} - B_{22})$$

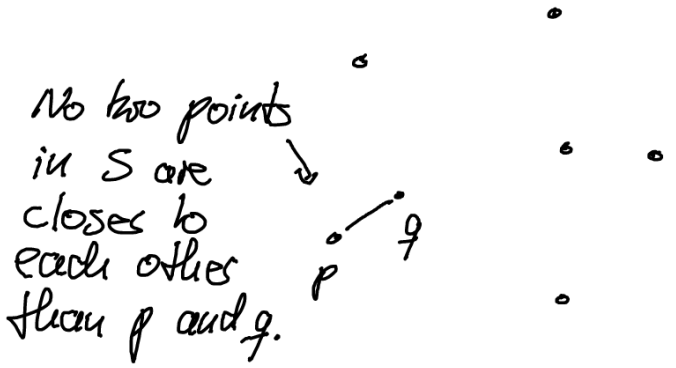
Thus, we can compute $C_{11}, C_{12}, C_{21}, C_{22}$ using 7 $(n/2)\times(n/2)$ matrix multiplications and 18 matrix additions as opposed to 8 matrix multiplications and 4 additions. This gives the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2)$$

which, by the Master Theorem, has the solution $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$. Using similar but more complicated ideas, one can achieve a running time of $\Theta(n^{2.36})$.

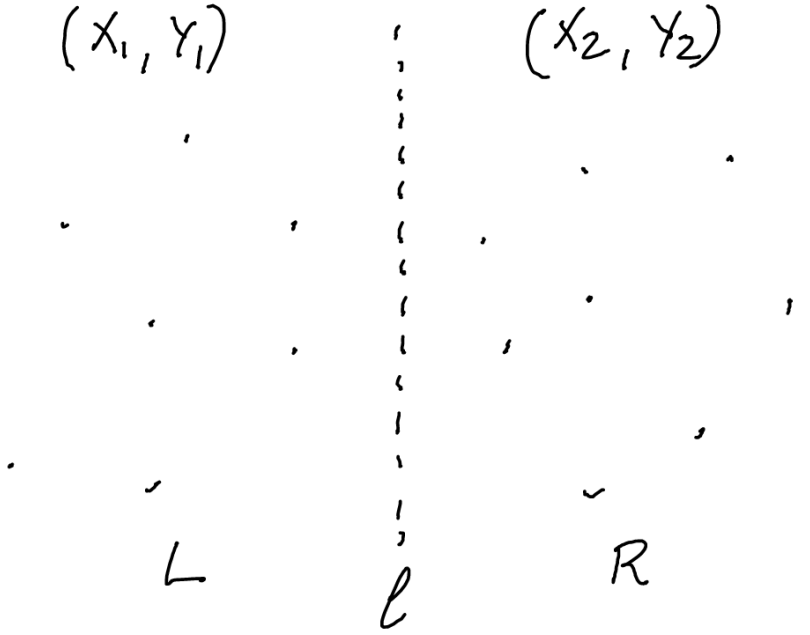
Closest Pair

Let $d(p,q)$ be the Euclidean distance between points p and q. Given a point set S, the closest pair of S is a pair $(p,q) \in S \times S$ such that $d(p,q) = \min \{ d(r,s) \mid r,s \in S \}$.



The naive algorithm tries all pairs of points and remembers the closest pair it finds. This takes $\Theta(n^2)$ time. We can in fact solve this problem in $\Theta(n \lg n)$ time, using divide and conquer.

As a preprocessing step, we make two copies of the point set and sort one by x-coordinates and the other by y-coordinates. We call these two sorted lists X and Y. We can divide X into a left half and a right half. Call them X_1 and X_2. This takes $O(1)$ time. In $O(n)$ time, we can partition Y into corresponding lists Y_1 and Y_2 containing the points of X_1 and X_2 in y-sorted order. (Deciding whether a point in Y should go into Y_1 or Y_2 requires comparing its x-coordinate to the maximum x-coordinate in X_1.)



Now we can divide the point pairs in S into three groups:

- Pairs in L
- Pairs in R
- Pairs with one point in L and one point in R.

Thus, we can find the closest pair in S by first finding the closest pairs in L and R and then checking whether there is an L-R pair that is closer than

both. We will show that the latter can be done in $\Theta(n)$ time. Since we can split X and Y into X_1, X_2 and Y_1, Y_2 in $\Theta(n)$ time and we can find the closest pairs in L and R by recursing on the input (X_1, Y_1) and (X_2, Y_2), this gives the recurrence

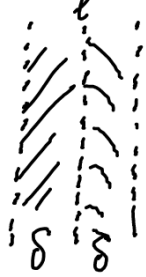
$$T(n) = 2T(n/2) + \Theta(n)$$

with solution $T(n) \in \Theta(n \lg n)$. Adding the sorting cost for producing the lists X and Y for the top-level invocation gives a total running time of $\Theta(n \lg n)$. (Note that it is crucial that we do not sort X and Y at the beginning of each recursive call because this would give

$$T(n) = 2T(n/2) + \Theta(n \lg n)$$

with solution $T(n) \in \Theta(n \lg^2 n)$.)

So let's figure out how we find a closest L-R pair. Let δ_1 be the closest distance in L, and let δ_2 be the closest distance in R. These are returned by the recursive calls on L and R. Let $\delta = \min(\delta_1, \delta_2)$. We are interested in an L-R pair only if its distance is less than δ! Let l be a vertical line separating L from R. All points that can be members of such L-R pairs have distance of most δ from l.



Let Y_1^δ and Y_2^δ be the sublists of points in Y_1 and Y_2 with distance at most δ from ℓ. These lists can be produced in $\Theta(n)$ time.

Now consider a point $p \in Y_1^\delta$. Any point $q \in Y_2^\delta$ with $d(p,q) < \delta$ must satisfy $y_p - \delta \le y_q \le y_p + \delta$. This gives the following algorithm for finding an L-R pair with distance less than δ if such a pair exists:

```
FindLR(Y1, Y2, ℓ, δ)
  Y1^δ = sublist of points in Y1 at distance ≤δ from ℓ
  Y2^δ = sublist of points in Y2 at distance ≤δ from ℓ
  q = first point in Y2^δ
  pair = ∅
  for every point p ∈ Y1^δ do
    while y_q < y_p − δ do
      if q is the last point in Y2^δ then return pair
      else q = the next point in Y2^δ
    r = q
    while r ≠ ∅ and y_r ≤ y_p + δ do
      if d(p, r) < δ then
        δ = d(p, r)
        pair = (p, r)
      if r is not the last point in Y2^δ then
        r = the next point in Y2^δ
      else r = ∅
  return pair
```

The correctness of the algorithm follows immediately from our discussion. We need to prove that it takes

linear time.

Producing Y_1^δ and Y_2^δ takes $\Theta(n)$ time. The for-loop has $|Y_1^\delta| \in O(n)$ iterations. The first while-loop has $|Y_2^\delta| \in O(n)$ iterations in total, summed over all iterations of the for-loop because it scans Y_2^δ once. For the second while-loop, we prove that it has $O(1)$ iterations per iteration of the for-loop. Then, it has $O(n)$ iterations in total and the running time of FindCR is $\Theta(n)$ as claimed.

To prove this claim, we observe that this while-loop inspects only points r in Y_2^δ with $y_p - \delta \le y_r \le y_p + \delta$ plus possibly one extra point. Then, it suffices to prove the following lemma:

Lemma: *There are at most 8 points in Y_2^δ with y-coordinates in the interval $[y_p - \delta, y_p + \delta]$.*

Proof: Let Q be the set of points in Y_2^δ with y-coordinates in $[y_p - \delta, y_p + \delta]$. All these points are contained in a rectangle B with area $2\delta^2$ and their pairwise distances are at least δ because they all belong to R. We divide B into 8 $\delta/2 \times \delta/2$ squares. If Q contained more than 8 points, one such square would have to contain two points in Q. This, however, is impossible because two points in the same square have distance at most $\delta/\sqrt{2} < \delta$ from each other. □