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 argue 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 \(n\), 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 \(\lfloor n/2 \rfloor\) and \(\lceil n/2 \rceil\). Since \(\lfloor n/2 \rfloor < n\), 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 $|L| \geq 0$ 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 $T(n/2)$ and $T(n/2)$. We know exactly how long this takes: $T(n/2)$ and $T(n/2)$ time. Once these two recursive calls return, we spend $O(n)$ time to merge the two
Sorted lists they produce. Thus,

\[
T(n) = \begin{cases} 
0(1) & n=1 \\
T(n/2) + T(n/2) + O(n) & n>1 
\end{cases}
\]

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

\[
T(n) = \begin{cases} 
0(1) & n=1 \\
T(n/2) + O(n) & n>1 
\end{cases}
\]

\[
T(n) = \begin{cases} 
0(1) & n=1 \\
2T(n/2) + O(n) & n>1 
\end{cases}
\]

\[
T(n) = \begin{cases} 
0(1) & n=1 \\
3T(n/2) + O(n) & n>1 
\end{cases}
\]

look rather similar but have dramatically different solutions of \( T(n) \in O(n) \), \( T(n) \in O(n \log n) \), and \( T(n) \in O(n \log^3 n) \in O(n^{1.59}) \), 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

\[
T(n) = T(n/2) + T(n/2) + O(n),
\]

which means
$T(n) = \begin{cases} \Theta(1) & n \leq n_0 \\ T(T(n/2)) + T(L(n/2)) + O(1) & n > n_0 \end{cases}$

**Solving recurrences**

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

- Induction
- The Master Theorem
- Recursion trees

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

The Master 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 $T(n) \in O(n \log n)$, that is, $T(n) \leq cn \log n$ for some $c > 0$, $n_0 > 0$ and all $n > n_0$. We prove this by induction:
For 2 ≤ n ≤ 3, we have T(n) ∈ O(1) and n/log n ≥ n.
Thus, we can find a constant c such that
T(n) ≤ cn/log n.

For n > 4, we have T(n) ≤ T(n/2) + T(n/2) + dn,
for some d > 0. By the inductive hypothesis, this gives

\[ T(n) \leq c \frac{n}{\log \frac{n}{2}} \leq n \left( c \frac{\log \frac{n}{2}}{\log n} + d \right) \]

\[ \leq n \left( c \log \left( \frac{n}{2} \right) + d \right) \quad \text{(because} \quad \frac{n}{2} \leq \frac{3}{4} n \quad \text{for} \quad n > 4) \]

\[ \leq n (c \log n - c \log \frac{4}{3} + d) \]

\[ \leq cn \log n \quad \text{as long as} \quad c \log \frac{4}{3} > d \quad \text{or} \quad c \geq \frac{d}{\log \frac{4}{3}}. \]

What about the other two recurrences above?

- T(n) = T(n/2) + O(1). We claim that this gives T(n) ∈ O(1), that is, T(n) ≤ cn.

For n ≤ 2, we have T(n) ∈ O(1), so T(n) ≤ cn for a sufficiently large c > 0.
For \( n \geq 2 \), we have \( T(n) = T(n/2) + dn \), which gives

\[
T(n) \leq \frac{cn}{2} + dn
\]
\[
= (\frac{c}{2} + d)n
\]
\[
\leq cn \quad \text{as long as} \quad c > 2d.
\]

\( T(n) = 3T(n/2) + O(n) \). We claim that this gives \( T(n) \in O(n \log^3) \).

For \( n \leq 2 \), we have \( T(n) \in O(1) \) and \( n \log^3 \approx n \), so
\( T(n) \leq cn \log^3 \), for sufficiently large \( c > 0 \).

For \( n \geq 2 \), we get

\[
T(n) \leq 3C \cdot \left(\frac{n}{2}\right) \log^3_n + dn
\]
\[
= \frac{3C n \log^3}{2 \log^3} + dn
\]
\[
= cn \log^3 + dn,
\]

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
\( T(n) \leq c_1 n \log^3 - c_2 n \), for some \( c_1 > c_2 > 0 \).

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

\[
T(n) \leq 3 \left[ C_1 \left( \frac{n}{2} \right) \log_2^3 - C_2 \frac{n}{2} \right] + dn
\]

\[
= 3C_1 n \log_2^3 - (\frac{3}{2} C_2 - d) n
\]

\[
= C_1 n \log_2^3 - C_2 n \text{ as long as } \frac{3}{2} C_2 - d > C_2 \text{, which is } C_2 > 2d.
\]

So, why were we able to prove \( T(n) \leq C_1 n \log_2^3 - C_2 n \) when we weren't able to prove the weaker claim \( T(n) \leq C_1 n \log_2^3 \)? 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 feel we need 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.

0. **Final exercise:** We know binary search takes \( O(\log n) \) time. Its recurrence is

\[
T(n) = T(n/2) + O(1)
\]

We claim that \( T(n) \leq C \log n \) for some \( C>0 \).

For \( 2n < 4 \), we have \( T(n) \leq O(1) \) and \( \log n \), so \( T(n) \leq C \log n \), for large enough \( C>0 \).
For \( n \geq 4 \), we have

\[
T(n) \leq T(n/2) + d
\leq c \log(n/2) + d
= c \log n - c + d
\leq c \log n \quad \text{as long as } c > d,
\]

The Master's 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 \( f(n) \in \Theta(n^{\log_b a} \log cn) \), for some \( c > -1 \), then \( T(n) \in \Theta(f(n) \log n) \).

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

\[
T(n) = 2T(n/2) + \Theta(n)
\]

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


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

- **Binary search:**
  
  \[
  T(n) = T(n/2) + \Theta(1)
  \]

  \[
  n^{\log_2 1} = n^0 = 1, \text{ so } f(n) \in \Theta(n^{\log_2 1}) \text{, so we have Case (iii) again}, \ T(n) \in \Theta(\log n).
  \]

- **Selection:**

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

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

- **Matrix multiplication:**

  \[
  T(n) = 7T(n/2) + \Theta(n^3)
  \]

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

**Proof of the Master's Theorem:** Let us first spell out the recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & n \leq n_0 \\
\alpha T\left(\frac{n}{\beta}\right) + f(n) & n > n_0
\end{cases}
\]
Then

\[ T(n) = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta(n^{\log_b \alpha}). \]

This is easy to prove by induction:

For \( \frac{n_0}{b} \leq n \leq n_0 \), we have \( T\left(\frac{n}{b}\right) = 0 \), so the sum is empty. Since \( n \leq n_0 \), we have \( n^{\log_b \alpha} \in \Theta(1) \) and \( T(n) \in \Theta(1) = \Theta(n^{\log_b \alpha}) \).

For \( n > n_0 \), we have

\[
T(n) = a T\left(\frac{n}{b}\right) + f(n)
\]

\[
= a \left[ \sum_{i=0}^{\log_b \frac{n}{n_0} - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta\left(\frac{n}{b^i}\right)^{\log_b \alpha} \right] + f(n)
\]

\[
= \sum_{i=0}^{\log_b \frac{n}{n_0} - 2} a^{i+1} f\left(\frac{n}{b^{i+1}}\right) + \Theta(a \cdot \frac{n^{\log_b \alpha}}{\alpha}) + f(n)
\]

\[
= \sum_{i=0}^{\log_b \frac{n}{n_0} - 1} a^i f\left(\frac{n}{b^i}\right) + \Theta(n^{\log_b \alpha})
\]

Now we can consider the three cases:

(i) \( f(n) \in \Omega(n^{\log_b \alpha + \epsilon}) \) and \( a \cdot \ln(b) < c \cdot f(n) \)

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

Since \( f(n) \in \Omega(n^{\log_b \alpha + \epsilon}) \), we thus have
\[ T(n) \in O(f(n)). \text{ Since } T(n) \geq f(n), \text{ we also have } T(n) \in \Omega(f(n)), \text{ so } T(n) \in \Theta(f(n)). \]

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

\[
\sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} a_i f\left(\frac{n}{b_i}\right) \leq C \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} a_i \left(\frac{n}{b_i}\right)^{\log_b a - \varepsilon}
\]

\[
= Cn^{\log_b a - \varepsilon} \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} b_i^{\varepsilon i}
\]

\[
\leq Cn^{\log_b a - \varepsilon} \int_0^{\lceil \log_{b_{\text{wo}}} n \rceil} b_i^{\varepsilon i} \, di
\]

\[
= Cn^{\log_b a - \varepsilon} \left[ b_i^{\varepsilon i} \right]_0^{\lceil \log_{b_{\text{wo}}} n \rceil}
\]

\[
\leq Cn^{\log_b a - \varepsilon} \cdot \varepsilon \ln b \cdot n^\varepsilon
\]

\[
eq O(n^{\log_b a})
\]

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

(iii) \( f(n) \in \Theta(n^{\log_b a \cdot \lg c \cdot n}) \) for some \( c > 1 \).

\[
\sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} a_i f\left(\frac{n}{b_i}\right) \in \Theta\left( \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} a_i \left(\frac{n}{b_i}\right)^{\log_b a \cdot \lg c\left(\frac{n}{b_i}\right)} \right)
\]

\[
= \Theta\left( n^{\log_b a} \cdot \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} \lg c\left(\frac{n}{b_i}\right) \right)
\]

Now \( \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} \lg c\left(\frac{n}{b_i}\right) \leq \lg c \cdot n \cdot \left( \log_{b_{\text{wo}}} \left( n^\frac{1}{\log_b a} \right) - 1 \right) \leq C \lg c \cdot n \)

Also, \( \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} \lg c\left(\frac{n}{b_i}\right) \geq \sum_{i=0}^{\lceil \log_{b_{\text{wo}}} n \rceil - 1} \lg c\left(\frac{n}{b_i}\right) \geq \Gamma \log_b \frac{n}{\ln^7 1} \cdot \lg c \left(\frac{n}{\ln^7 1}\right)
\]

\[
= \frac{1}{2} \log_b n^7 \cdot \frac{1}{2} \cdot \lg c \cdot n \in \Omega(\lg c^2 \cdot n).
\]
Thus, \( \sum_{i=0}^{\log_5 \frac{n}{\log\, n}} a^i f\left(\frac{n}{5^i}\right) \in \Theta\left(n^{\log_5 a / \log\, n} \cdot g(n)\right) = \Theta(f(n) \lg n) \)

Since \( f(n) \lg n \in \Omega(n^{\log_5 a}) \), 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 paths of the expansion. Let's look at merge sort again:

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \text{ for some } c > 0. \]

This gives us the tree for \( T(n) \):

\[ cn \]
\[ T\left(\frac{n}{2}\right) \quad T\left(\frac{n}{2}\right) \]

Now we can replace \( T\left(\frac{n}{2}\right) \) with its own tree:

\[ cn/2 \]
\[ T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \]

which gives

\[ cn \]
\[ \frac{cn}{2} \quad \frac{cn}{2} \]
\[ T\left(\frac{n}{4}\right) \quad \ldots \quad T\left(\frac{n}{4}\right) \]

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 | $n/2$ | $2 \cdot \frac{cn}{2} = cn$
2 | $n/4$ | $4 \cdot \frac{cn}{4} = cn$
3 | $n/8$ | $\vdots$

$\Theta(1) \ldots \ldots \ldots \Theta(1)$ \quad $\lg \frac{n}{n_0} \quad \Theta(n/n_0)$

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

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:

- Left path has length $\log_3 (n/n_0)$
- Right path has length $\log_3 (n/n_0) / 2$

So our tree looks like this:

Cost per level = $cn$
We still get $T(n) = \Theta(\lg n)$.

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

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

$$f(n) \leq c \cdot n^{\log_b a - \epsilon} \cdot b^i$$

That is, the cost of the leaves dominates.

If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and $af(n) \leq c_i f(n)$ for some $c_i < 1$,

we get

$$f(n) \leq c_i f(n)$$

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 \( k \)th 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 \( k \)th 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 \( k \)th smallest element?

\[
\begin{array}{cc}
\leq p & \geq p \\
\uparrow & \uparrow \\
m & n - m
\end{array}
\]

If \( k \leq m \), the element we're looking for is here.
And is the \( k \)th 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 \(A[l]\)
\(p = \text{FindPivot}(A, l, r)\)
\(m = \text{Partition}(A, l, r)\)
if \(k \leq m - l + 1\) then return \(\text{QuickSelect}(A, l, m, k)\)
else return \(\text{QuickSelect}(A, m+1, r, k-m)\)

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 \(T(n) \in \Theta(n^2)\) by the Master 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 works does in practice.
Option 2: Find a way to select an approximate median by applying selection to only a subset of the input elements.

\[
\text{FindPivot}(A, l, r) \\
\quad n' = \left\lceil \frac{r-l+1}{5} \right\rceil - 1 \\
\quad \text{for } i = 0 \text{ to } n'-1 \text{ do} \\
\quad \quad \text{Sort } A[l+5i, l+5i+4] \text{ using insertion sort} \\
\quad \quad B[i] = A[l+5i+2] \\
\quad \quad B[n'] = A[5n'] \\
\quad \text{return } \text{QuickSelect}(B, 0, n', \lceil \frac{n'}{2} \rceil)
\]

Observation: \text{FindPivot}(A, l, r) \text{ takes } T(\lceil n/57 \rceil) + O(r-l+1) \text{ time, where } T(n) \text{ is the running time of } \text{QuickSelect}.

Lemma: Let \( p \) be the element returned by \text{FindPivot}(A, l, r), and let \( n = r-l+1 \). Then at least \( \frac{12n}{10} - 5 \) elements are less than \( p \) and at least \( \frac{8n}{10} - 5 \) 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 \( n' = \lceil n/57 \rceil \) elements in \( B \). There are at least \( \lceil n'127 \rceil - 1 \) elements in \( B \) less than the pivot \( p \) returned, and there are \( \lceil n'127 \rceil - 1 \) 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 greates than p at most one is 
not the median of a group of 5 elements in A.
For each element greates than p that is the median of
a group of 5 elements in A, at least 3 elements in
this group are greates than p. Thus, A contains at
least \(3(\frac{n}{127}-2)+1=3\frac{n}{127}-5\) elements
greates than p. Substituting \(n'=\frac{n}{157}\), we obtain
that at least \(3\frac{n}{157}/\frac{27}{5} \geq 3\frac{n}{10}-5\) elements
in A are greates than p.

\[\square\]

**Lemma:** QuickSelect takes \(O(n)\) time.

**Proof:** Apart from calling itself recursively on at
most \(\frac{n}{10}+5\) elements and from calling FindPivot,
QuickSelect takes \(O(n)\) time. FindPivot takes \(O(n)\) +
\(T(\frac{n}{157})\) time. This gives the recurrence

\[T(n) \leq cn + T(\frac{n}{157}) + T(\frac{n}{10}+5)\]

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 \(\frac{n}{157} < n\) and \(1 \leq \frac{n}{10}+5 \leq n\),
so we can apply the inductive hypothesis to \(T(\frac{n}{157})\)
and \(T(\frac{n}{10}+5)\). This gives

\[T(n) \leq cn + d\frac{n}{157} + d(\frac{n}{10}+5)\]
\[\leq cn + d(\frac{n}{157}+1) + d(\frac{n}{10}+5)\]
\[= (c + \frac{d}{157} + \frac{7d}{10})n + 6d\]
\[\leq c + \frac{d}{157} + \frac{7d}{10} + \frac{d}{20} n\]


\[
= \left( c + \frac{4 + 4 + 1}{20} \right) n \\
= \left( c + \frac{19}{20} d \right) n \\
\leq d n \text{ provided } d \geq 20c. 
\]

\[\square\]

Matrix Multiplication

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

\[
\text{Multiply } (A, B, n) \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\hspace{1cm} \text{for } j = 1 \text{ to } n \text{ do} \\
\hspace{2cm} C[i, j] = 0 \\
\hspace{1cm} \text{for } k = 1 \text{ to } n \text{ do} \\
\hspace{2cm} C[i, j] = C[i, j] + A[i, k]B[k, j]
\]

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 \((n/2) \times (n/2)\). 

\[
\begin{array}{cc}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{array}
\]

\[
\begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}
\]

\[
\begin{array}{cc}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{array}
\]

These submatrices satisfy the equalities

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

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 Master's Theorem, this gives \( T(n) = \Theta(n^3) \). How can we do better? We start by visualizing the computation of \( C_{11}, C_{12}, C_2 \) and \( C_{22} \) as a vector-matrix-vector product:

\[
\begin{array}{cccc}
B_{11} & B_{12} & B_{21} & B_{22}
\end{array}\cdot
\begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}
\]

\[
C_{ij} = \begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}\cdot
\begin{array}{cc}
M_{ij}
\end{array}
\]

This gives the matrices
\[
\begin{array}{cccc}
M_{11} & M_{12} & M_{21} & M_{22} \\
(\text{All other entries are 0.})
\end{array}
\]

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

\[
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array}
\]

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 \(M_1, M_2, M_4, M_7\) are linear combinations of these matrices.
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 $C_1, C_2, C_2', C_2$ using 7 \((n/12) \times (n/12)\) 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

\[ T(n) = \Theta(n^{\log_{2}7}) = \Theta(n^{2.81}). \]  

Using similar but more complicated ideas, one can achieve a running time of \(\Theta(n^{2.38})\).

**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

\[ d(p,q) = \min \{ d(r,s) \mid r,s \in S \}. \]

No two points in \(S\) are closer to each other than \(p\) and \(q\).

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 \(\Theta(n \log n)\) 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 these \( X_1 \) 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 \( X_1 \).)

\[
\begin{array}{c@{\ldots@{}}c}
(X_1, Y_1) & (X_2, Y_2) \\
& \\
& \\
& \\
& \\
& \\
& \\
& \\
& \\
L & L & R
\end{array}
\]

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

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

with solution $T(n) = \Theta(n \log^2 n)$.)

So let's figure out how we find a closest $L-R$ pair. Let $S_1$ be the closest distance in $L$, and let $S_2$ be the closest distance in $R$. These are returned by the recursive calls on $L$ and $R$. Let $S = \min(S_1, S_2)$. We are interested in an $L-R$ pair only if its distance is less than $S$. Let $l$ be a vertical line separating $L$ from $R$. All points that can be members of such $L-R$ pairs have distance at most $S$ from $l$. 

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

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

```
FindLR(Y_1, Y_2, l, \delta)
    Y_1 = sublist of points in $Y_1$ at distance \leq \delta from $l$
    Y_2 = sublist of points in $Y_2$ at distance \leq \delta from $l$
    q = first point in $Y_2$
    pair = \emptyset
    for every point $p \in Y_1$ do
        while $y_q < y_p - \delta$ do
            if $q$ is the last point in $Y_2$ then return pair
            else $q = the next point in $Y_2$
        end
        $r = q$
        while $r \neq \emptyset$ and $y_r \leq y_p + \delta$ do
            if $d(p, r) < \delta$ then
                $\delta = d(p, r)$
                $pair = (p, r)$
            end
            if $r$ is not the last point in $Y_2$ then
                $r = the next point in $Y_2$
            else $r = \emptyset$
        end
    end
    return pair
```

The correctness of the algorithm follows immediately from our discussion. We need to prove that it takes
Produce $Y_1^\delta$ and $Y_2^\delta$ takes $\Theta(n)$ time. The for-loop has $128^\delta \in O(n)$ ileahons. The first while-loop has $128^\delta \in O(n)$ ileahons in total, summed over all ileahons of the for-loop because it scans $Y_2^\delta$ once. For the second while-loop, we prove that it less $O(1)$ ileahons per iteration of the for-loop. Thus, it has $O(n)$ ileahons in total and the running time of $\text{FindCR}$ is $\Theta(n)$ as claimed.

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

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

**Proof:** Let $Q$ be the set of points in $Y_2^\delta$ with $y$-coordinates in $[y_p - \delta, y_p + \delta]$. All these points are contained in a rectangle with area $2\delta^2$ and their pairwise distances are at least $\delta$ because they all belong to $R$. We divide $B$ into $8 \times 8/12 \times 8/12$ squares. If $Q$ contained more than 8 points, one such square would contain at least two points in $Q$. This, however, is impossible because two points in the same square have distance at most $8/12 < \delta$ from each other. \[\square\]