# COMP3121/9101
# Algorithm Design

## Practice Problem Set 2 – Divide and Conquer

**[K]** – key questions **[H]** – harder questions **[E]** – extended questions **[X]** – beyond the scope of this course

## Contents

## § SECTION ONE: RECURRENCES

**[K] Exercise 1**. Determine the asymptotic growth rate of the solutions to the following recurrences. You may use the Master Theorem if it is applicable.

(a) $T(n) = 2T(n/2) + O(n)$.

(b) $T(n) = 2T(n/2) + \sqrt{n} + \log n$.

(c) $T(n) = 8T(n/2) + n^{\log_2 n}$.

(d) $T(n) = T(n-1) + n$.

*Solution.* (a) Here, we realise that $a = 2$, $b = 2$, and $f(n) = n(2 + \sin n)$. But $\sin n \leq 1$ for all $n$ and so, $f(n) = \Theta(n)$. The critical exponent is

$$c^* = \log_b a = \log_2 2 = 1.$$

Thus, the second case of the Master Theorem applies and we get

$$T(n) = \Theta(n \log n).$$

(b) Again, we repeat the same process. We realise that $a = 2$, $b = 2$, and $f(n) = \sqrt{n} + \log n$. So, the critical exponent is $c^* = 1$. Since $\log n$ *eventually* grows slower than $\sqrt{n}$, we have that

$$f(n) = \Theta(\sqrt{n}). = \Theta\left(n^{1/2}\right).$$

This implies that

$$f(n) = O(n^{0.9}) = O(n^{c^*-0.1}),$$

so the first case of the Master Theorem applies and we obtain $T(n) = \Theta(n)$.

(c) Here, $a = 8$, $b = 2$, and $f(n) = n^{\log_2 n}$. So the critical exponent is

$$c^* = \log_b a = \log_2 8 = 3.$$

On the other hand, for large enough $n$, we have that $\log_2 n \geq 4$. So

$$f(n) = n^{\log_2 n} = \Omega(n^4).$$

Consequently,

$$f(n) = \Omega(n^{c^*+1}).$$

To be able to use the third case of the Master Theorem, we have to show that for some $0 < c < 1$, the following holds for sufficiently large $n$:

$$af\left(\frac{n}{b}\right) < cf(n).$$

In our case, this translates to

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < cn^{\log_2 n}.$$

Now, we have

$$\begin{aligned} 8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} &= 8\left(\frac{n}{2}\right)^{\log_2 n - \log_2 2} \\ &= 8\left(\frac{n}{2}\right)^{\log_2 n - 1} \\ &< 8\left(\frac{n}{2}\right)^{\log_2 n} \\ &= \frac{8n^{\log_2 n}}{2^{\log_2 n}} \\ &= \frac{8}{n}n^{\log_2 n}. \end{aligned}$$

If $n > 16$, then

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < \frac{1}{2}n^{\log_2 n},$$

so the required inequality is satisfied with $c = \frac{1}{2}$ for all $n > 16$. Therefore, by case 3 of the Master Theorem, the solution is

$$T(n) = \Theta(f(n)) = \Theta\left(n^{\log_2 n}\right).$$

(d) Note that the Master Theorem does not apply; however, we can alter the proof of the Master Theorem to obtain the solution to the recurrence. For every $k$, we have $T(k) = T(k-1) + k$. So unrolling the recurrence, we obtain

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= \underbrace{[T(n-2) + (n-1)]}_{T(n-1)} + n \\ &= T(n-2) + (n-1) + n \\ &= [T(n-3) + (n-2)] + (n-1) + n \\ &= \dots \\ &= T(1) + (n - (n-2)) + (n - (n-3)) + \dots + (n-1) + n \\ &= T(1) + (2 + 3 + \dots + n) \\ &= T(1) + \frac{n(n+1)}{2} - 1 \\ &= \Theta(n^2). \end{aligned}$$

□

**[K] Exercise 2**. Explain why we cannot apply the Master Theorem to the following recurrences.

(a) $T(n) = 2^n T(n/2) + n^n$.

(b) $T(n) = T(n/2) - n^2 \log n$.

(c) $T(n) = \frac{1}{3}T(n/3) + n$.

(d) $T(n) = 3T(3n) + n$.

*Solution.* (a) The Master Theorem requires the value of $a$ to be constant. Here, the value of $a = 2^n$ depends on the input size which is non-constant.

(b) The Master Theorem requires $f(n)$ to be non-negative. We can see that, for $n \in \mathbb{N}$, $f(n) \leq 0$. So, the Master Theorem cannot be applied.

(c) The Master Theorem requires $a \geq 1$. Here, $a = 1/3$ and so, the conditions of the Master Theorem are not met.

(d) The Master Theorem requires $b > 1$. However, we see that we can write $T(n)$ as

$$T(n) = 3T\left(\frac{n}{1/3}\right) + n;$$

in this case, we see that $b = 1/3 < 1$. So the conditions of the Master Theorem are not met.

□

**[H] Exercise 3**. Consider the following naive Fibonacci algorithm.

**Algorithm 1** $F(n)$: The naive Fibonacci algorithm

```
Require: $n \geq 1$
  if $n = 1$ or $n = 2$ then
      return 1
  else
      return $F(n-1) + F(n-2)$
  end if
```

When analysing its time complexity, this yields us with the recurrence $T(n) = T(n-1) + T(n-2)$. Show that this yields us with a running time of $\Theta(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$ which is the golden ratio. How do you propose that we improve upon this running time?

This is a standard recurrence relation. Guess $T(n) = a^n$ and solve for $a$.

*Solution.* We solve $T(n) = T(n-1) + T(n-2)$. Using the standard trick of solving recurrence relations, we guess $T(n) = a^n$ for some $a$. Then this produces the recurrence:

$$a^n = a^{n-1} + a^{n-2}.$$

Dividing both sides by $a^{n-2}$, we obtain the quadratic equation:

$$a^2 - a - 1 = 0.$$

Then the quadratic equation yields

$$a = \frac{1 \pm \sqrt{5}}{2}.$$

In other words, we obtain

$$T(n) = A \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + B \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n = \Theta\left(\varphi^n\right),$$

where $\varphi = \frac{1+\sqrt{5}}{2}$.

The inefficiency comes from the fact that we are making unnecessary and repeated calls to problems that we have already seen before. For example, to compute $F(n)$, we compute $F(n-1)$ and $F(n-2)$. To compute $F(n-1)$, we compute $F(n-2)$ again. Because these values never change, a way to improve the overall running time is to *cache* the results that we have computed previously and only make $O(1)$ calls for results that we have already computed. This reduces our complexity dramatically from exponential to $O(n)$. This is an example of *dynamic programming*, something we will see later down the track. $\square$

## § SECTION TWO: DIVIDE AND CONQUER

**[K] Exercise 4.** Suppose you have $n$ sorted lists and suppose that there are $N$ elements across all $n$ lists. Note that each list is not necessarily the same length. Design an $O(N \log n)$ algorithm that merges all $n$ lists into one sorted list.

For example, if we have two lists $A_1 = [1, 2]$ and $A_2 = [1, 4]$, then the final sorted list should return $B = [1, 1, 2, 4]$.

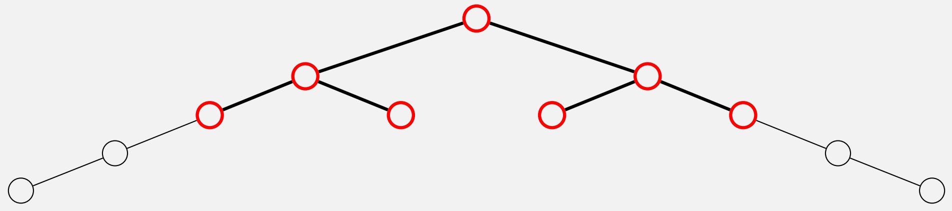
*Solution.* At each recursion, we pair up each of the $n$ lists to form $n/2$ pairs. For each pair of lists, we can merge them into a sorted list in $O(k)$ time, where $k$ is the number of elements in the final sorted list (i.e. the sum of the number of elements in both lists). At each level, we eventually merge all $k$ elements in each pair; therefore, in each level, we perform an $O(N)$ operation merge. Since we are dividing each list into pairs, we have $\log n$ depth and at each depth, we perform an $O(N)$ operation. Therefore, the overall algorithm has running time $O(N \log n)$. □

**[K] Exercise 5.** You are given a string $S$ of length $n$ and an integer $k$. Design an $O(n^2)$ algorithm that finds the longest substring of $S$ such that each unique character in the substring has frequency at least $k$. For example, if $S = abbacccd$ and $k = 2$, then the longest substring is $abbaccc$ since the frequency of $a$ and $b$ is 2 and the frequency of $c$ is $3 \geq k$.

*Solution.* Note that any character which has a frequency smaller than $k$ can't be included in our substring. Therefore, we recurse on that particular character and find the longest substring on both sides of the split recursively. This ensures that we never consider a character that we know will never appear in our substring. In the worst case, finding each character takes $O(n)$ and the maximum depth of our recursion is $O(n)$ since we could split at every index. Therefore, this gives our algorithm a running time of $O(n^2)$. □

**[K] Exercise 6.** Let $T$ be a binary tree with $n$ nodes; we see that a *subtree* of $T$ is any connected subgraph of $T$. A binary tree is *complete* if every internal node has two children, and every leaf node has exactly the same depth. Design an $O(n)$ algorithm that finds the depth of the *largest complete subtree*.

The following diagram illustrates a binary tree with the largest complete tree highlighted.



*Solution.* We outline each of the three steps of a divide and conquer algorithm for this problem.

- **Divide**: We divide our tree into a left and right subtree. In each subtree, we find the depth of the largest complete subtree as well as the depth of the largest complete subtree rooted at the left and right children respectively. Just finding the depth of the largest complete subtree contained in the left and right subtrees isn't enough for our combine stage to properly work. We need information about the depth at the root of the tree to potentially merge our complete subtrees together.
- **Conquer**: We recursively call on the left and right subtrees, with the base case being the leaves with depth 1 and 1 respectively.
- **Combine**: We now have the largest complete subtree on the left and right subtrees at the root. Let the depth of the largest complete left subtree be $LCS(L)$ and the depth of the largest complete right subtree be $LCS(R)$. Since we have the largest complete subtree at the root of the left and right children respectively, the depth at the

root $r$ is given by $LCS(r) = \min(LCS(L), LCS(R)) + 1$. Our final algorithm then returns the maximum between $LCS(r)$, the depth of the largest complete subtree contained entirely in the left subtree, and the depth of the largest complete subtree contained entirely in the right subtree.

To prove that our algorithm is correct, note that, at every recursion, we first find the largest complete tree at the root of the tree. Clearly, the root has depth 1 and we can form a complete subtree by taking the minimum depth among the left and right subtrees that still form a complete subtree. Therefore, $LCS(r)$ is at least $1+\min(LCS(L), LCS(R))$. To show that $LCS(r)$ is at most $1 + \min(LCS(L), LCS(R))$, we see that removing $r$ gives a depth of $LCS(r) - 1$ for the left and right subtree. Among these subtrees, we can find two complete subtrees in the left and right subtrees. But the complete subtree of maximum depth rooted at $r$ must be the minimum depth among the left and right subtrees. Therefore, $\min(LCS(L), LCS(R)) \geq LCS(r)-1$ which implies that $LCS(r) \leq \min(LCS(L), LCS(R))+1$. The largest complete subtree, therefore, must either be contained entirely on the left, entirely on the right, or at the root. This is exactly what our algorithm returns.

The time complexity is $O(n)$, where $n$ is the number of vertices in our binary tree because we are recursing on every vertex. Therefore, each vertex has $O(1)$ work, giving our algorithm the running time $O(n)$. □

**[H] Exercise 7**. You and a friend find yourselves on a TV game show! The game works as follows. There is a **hidden** $n \times n$ table $A$. Each cell $A[i, j]$ of the table contains a single integer and no two cells contain the same value. At any point in time, you may ask the value of a single cell to be revealed. To win the big prize, you need to find the $n$ cells each containing the **maximum** value in its row. Formally, you need to produce an array $M[1..n]$ so that $A[r, M[r]]$ contains the maximum value in row $r$ of $A$, i.e., such that $A[r, M[r]]$ is the largest integer among $A[r, 1], A[r, 2], \ldots, A[r, N]$. In addition, to win, you should ask at most $n\lceil \log N\rceil$ many questions. For example, if the hidden grid looks like this:

| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| Row 1 | **10** | 5 | 8 | 3 |
| Row 2 | 1 | **9** | 7 | 6 |
| Row 3 | -3 | **4** | -1 | 0 |
| Row 4 | -10 | -9 | -8 | **2** |

then the correct output would be $M = [1, 2, 2, 4]$.

Your friend has just had a go, and sadly failed to win the prize because they asked $N^2$ many questions which is too many. However, they whisper to you a hint: they found out that $M$ is **non-decreasing**. Formally, they tell you that $M[1] \leq M[2] \leq \cdots \leq M[n]$ (this is the case in the example above).

Design an algorithm which asks at most $O(n \log n)$ many questions that produces the array $M$ correctly, even in the very worst case.

*Solution.* We first find $M[N/2]$. We can do this in $N$ questions by simply examining each element in row $N/2$, and finding the largest one.

Suppose $M[N/2] = x$. Then, we know that $M[i] \leq x$ for all $i < N/2$ and that $M[j] \geq x$ for all $j > N/2$. Thus, we can recurse to solve the same problem on the sub-grids

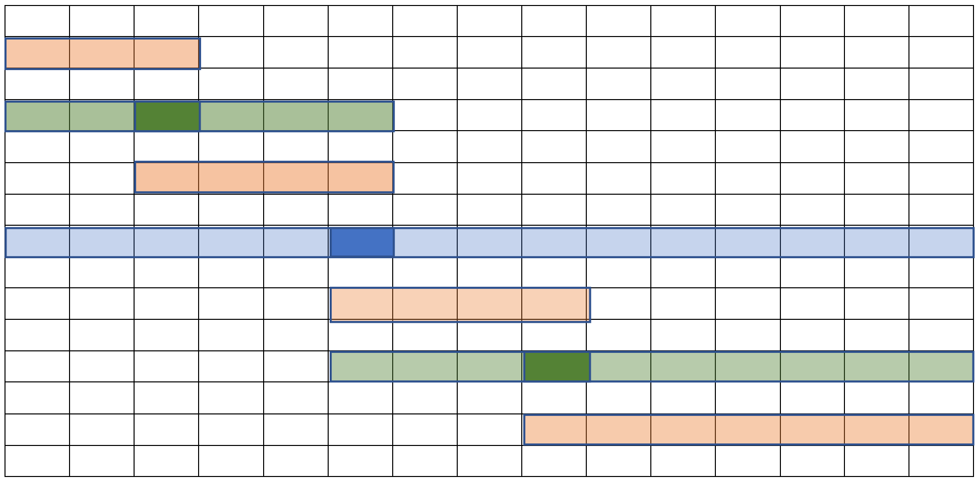
$$A\left[1..\left(\frac{N}{2}-1\right)\right][1..x]$$

and

$$A\left[\left(\frac{N}{2}+1\right)..N\right][x..N].$$

It remains to show that this approach uses at most $2N\lceil \log N\rceil$ questions.

**Claim**. At each stage of recursion, at most two cells of any column are investigated.



*Solution.* We observe the following.

- In the first call of the recursion, only one cell of each column has been investigated. These cells are marked in blue, with the maximum in dark blue.
- In the second call of the recursion, we investigate two cells of one column, and one cell of each of the other columns. These cells are marked in green, with the maxima in dark green.
- In the third call of the recursion, we investigate two cells in each of three columns, and one cell of each of the other columns. These cells are marked in orange.

Since the investigated ranges overlap only at endpoints, the claim holds true. ☐

So in each recursion call, at most $2N$ cells were investigated. The search space decreases by at least half after each call, so there are at most $\lceil \log_2 N \rceil$ many recursion calls. Therefore, the total number of cells investigated is at most $2N \lceil \log_2 N \rceil$.

Additional exercise: using the above reasoning, try to find a sharper bound for the total number of cells investigated. ☐

**[H] Exercise 8**. Define the Fibonacci numbers as

$$F_0 = 0, F_1 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 2.$$

Thus, the Fibonacci sequence is as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots.$$

(a) Show, by induction or otherwise, that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) Hence or otherwise, give an algorithm that finds $F_n$ in $O(\log n)$ time.

*Solution.* (a) We proceed by induction. If $n = 1$, then we have

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

as required.

Let $n \geq 2$. Now, suppose that

$$\begin{pmatrix} F_{(n-1)+1} & F_{n-1} \\ F_{n-1} & F_{(n-1)-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}.$$

Then

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} F_n + F_{n-1} & F_{n-1} + F_{n-2} \\ F_{n-1} + F_{n-2} & F_{n-2} + F_{n-3} \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} && \text{(inductive hypothesis)} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n}. \end{aligned}$$

This finishes the induction proof.

(b) Let $G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. We can compute $G^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2$ in $O(1)$ since each element in the matrix is just a sum and product of small numbers. Likewise, we can also compute powers of $G$ that are powers of two; in other words, we compute the matrices: $G, G^2, G^4, \ldots$.

To determine $F_n$, we need to find which combination of $G^{2^\ell}$ matrices we want to take. Thus, we can write $n$ in its binary representation; that is, write

$$n = a_0 2^0 + a_1 2^1 + a_2 2^2 + \ldots.$$

Each $a_i$ term tells us precisely which matrices of the form $G^{2^i}$ we want to multiply to obtain a final matrix $H$. The top right (or bottom left) of $H$ then tells us precisely what $F_n$. To argue the time complexity, we observe that there are maximally $\lfloor \log_2 n \rfloor$ to consider when writing down the binary representation of $n$. Therefore, we are performing $\log n$ many constant time operations, which give us $O(\log n)$ time complexity.

□

**[E] Exercise 9**. Let $A$ be an array of $n$ integers, not necessarily distinct or positive. Design a $\Theta(n \log n)$ algorithm that returns the maximum sum found in any contiguous subarray of $A$.

*Solution.* Note that brute force takes $\Theta(n^2)$.

The key insight to this problem is the following; once we split the original array in half, the maximum subarray must occur in one of the following cases:

- the maximum subarray sum occurs *entirely* in the left half of the original array.
- the maximum subarray sum occurs *entirely* in the right half of the original array.
- the maximum subarray sum overlaps across both halves.

We can compute the first two cases simply by making recursive calls on the left and right halves of the array. The third case can be computed in linear time. We now fill in the details of the algorithm.

Split the array into the left half and right half at the midpoint $m$.

- We can compute the maximum subarray sum in the left half using recursive calls on the left half of the array $A[1..m]$.
- Similarly, we can compute the maximum subarray sum in the right half using recursive calls on the right half of the array $A[(m+1)..n]$.
- To compute the maximum sum in the case where the subarray overlaps across both halves, we think of such a subarray as containing a portion from the left half and a portion from the right half. Now, we
  - find the maximum sum of any subarray of the form $A[l..m]$ by scanning left from $m$, and
  - similarly maximise the sum $A[(m+1)..r]$ by scanning right from $m+1$.

  Adding these two results gives the maximum sum of a subarray of the form $A[l..r]$ where $l \leq m < r$ as required.
- Our algorithm then returns the maximum of these three cases.
  - One caveat is that, if the maximum sum is negative, then our algorithm should return 0 to indicate that choosing an *empty* subarray is best.

To compute the time complexity of the algorithm, note that we're splitting our problem from size $N$ to two $N/2$ problems and doing an $O(n)$ overhead to combine the solutions together. This gives us the recurrence relation

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

This falls under Case 2 of the Master Theorem, so $T(n) = \Theta(n \log n)$ is the total running time of the algorithm. □

**[E] Exercise 10**. Suppose you are given an array $A$ containing $2n$ numbers. The only operation that you can perform is make a query if element $A[i]$ is equal to element $A[j]$, $1 \leq i, j \leq 2n$. Your task is to determine if there is a number which appears in $A$ at least $n$ times using an algorithm which runs in linear time.

Compare the numbers in pairs; how many potential candidates can exist?

*Solution.* Create two arrays $B$ and $C$, initially both empty. Repeat the following procedure $n$ times:

- Pick any two elements of $A$ and compare them.
  - If the numbers are different, discard both of them.
  - If instead the two numbers are equal, append one of them to $B$ and the other to $C$.

**Claim**. If a value $x$ appears in at least half the entries of $A$, then the same value also appears in at least half the entries of $B$.

*Proof.* Suppose such a value $x$ exists. In a pair of distinct numbers, at most one of them can be $x$, so after discarding both, $x$ still makes up at least half the remaining array elements. Repeating this, $x$ must account for at least half the values appearing in $B$ and $C$ together. But $B$ and $C$ are identical, so at least the entries of $B$ are equal to $x$.

We can now apply the same algorithm to $B$, and continue reducing the instance size until we reach an array of size two. The elements of this array are the only candidates for the original property, i.e., the only values that could have appeared $n$ times in the original array.

There is a special case; $B$ and $C$ could be empty after the first pass. In this case, the only candidates for the property are the values which appear in the last pair to be considered. For each of these, perform a linear scan to find their frequency in the original array, and report the answer accordingly.

Finally we analyse the time complexity. Clearly, each step of the procedure repeated $n$ times above takes constant time, so we can reduce the problem from array $A$ (of length $2n$) to array $B$ (of length $\leq n$) in $\Theta(n)$ time. Letting the instance size be *half* the length of the array (to more neatly fit the Master Theorem), we can express the worst case using the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Now $a\, f(n/b) = f(n/2)$, which is certainly less than say $0.9f(n)$ for large $n$ as $f$ is linear. By case 3 of the Master Theorem, the time complexity is $T(n) = \Theta(f(n)) = \Theta(n)$, as required. □

**[E] Exercise 11**. You are given $n$ points on a plane. The $i$th point is labelled with coordinates $(x_i, y_i)$. Design an $O(n \log n)$ algorithm that finds the pair of points that is closest together, where "closenes" is measured by its Euclidean distance.

*Solution.* Given our set of points, we sort our points by its $x$-coordinate and find the middle point – say $a$ – to divide our set into two parts. During the pre-processing stage, we also sort our points by its $y$-coordinate to ensure that we have a linear-time comparison later on.

This gives all the points whose $x$-coordinate is smaller than the $x$-coordinate of $a$ in one list, and all points whose $x$-coordinate is larger than the $x$-coordinate of $a$ in the other list. From these two smaller lists, we have one of three situations:

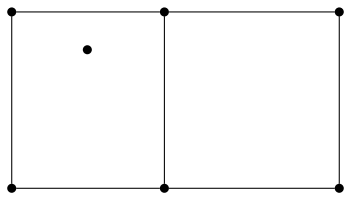
- The closest pair of points are situated *entirely* inside the left list;
- The closest pair of points are situated *entirely* inside the right list;
- The closest pair of points have one point in the left list and one point in the right list.

The first two subproblems can be solved recursively on the list of points that are situated entirely on the left and right sides respectively. We now manage the final situation. From the first two subproblems, let $\delta_L$ be the distance of the closest pair of points on the left subproblem and $\delta_R$ be the distance of the closest pair of points on the right subproblem. Define $\delta = \min\{\delta_L, \delta_R\}$. Clearly, the closest pair of points where one point is on the left and one point is on the right must have distance at most $\delta$; otherwise, the pair of points whose distance is $\delta$ are closer.

Therefore, any pair of points whose distance is at most $\delta$ must be contained within a $2\delta$-width rectangle centred around the dividing line (i.e. the $x$-coordinate of $a$). This reduces our search space to looking at points within a rectangle of width $2\delta$. We now need to narrow down our search space even more since the height of our current rectangle is unbounded.

Consider any point $b$ inside our current strip. To check if there is some point whose distance is smaller than $\delta$, we need to ensure that it is also within a $\delta$-distance vertically. Therefore, we consider any point inside a $\delta \times 2\delta$ rectangular slab containing $b$. We now claim that there are *at most* 6 points inside this rectangle.

To see why this is true, consider the following placement of points.



These are the *extremities* of point placement because any seventh point implies that there are two points in exactly one side whose distance is necessarily smaller than $\delta$. But this is a contradiction because we would have found this earlier in our algorithm. Therefore, for *each* point in our $\delta \times 2\delta$ rectangle, we perform at most 6 comparisons to update our minimum by finding all points that is vertically within $\delta$ distance in the other half of our rectangle. In other words, the merging process of our algorithm has running time $O(n)$.

This gives our recurrence

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

which gives $T(n) = \Theta(n \log n)$ by the Master Theorem. $\square$

## § SECTION THREE: QUICKSELECT

**[K] Exercise 12**. Consider $n$ points on a plane. The $i$th point has coordinates $(x_i, y_i)$. You are also given an integer $k$.

(a) Design an $O(n \log n)$ algorithm that finds the $k$ closest points from the origin, where "closeness" is measured by its Euclidean distance from the origin. In other words, we say that point $i$ is $\sqrt{x_i^2 + y_i^2}$ distance away from the origin. You may choose to return the $k$ closest points in any order.

(b) Design an algorithm with expected running time $O(n)$ that solves the same problem.

There is an easy way to solve part **(a)**; how might you optimise your strategy given that you can return your solution in *any* order for part **(b)**?

*Solution.*

(a) For each point $i$, compute its distance from the origin $x_i^2 + y_i^2$ (why is it enough to look simply at $x_i^2 + y_i^2$?) in constant time and store the distance in an array $A$. We can then sort the array by MergeSort with $O(n \log n)$ running time. The $k$ closest points are then the first $k$ points in $A$ (i.e. $A[1], A[2], \dots, A[k]$ after sorting).

(b) This is a classical problem that can be solved with QuickSelect. Firstly, we can compute each distance in constant time and store the distances in an array $A$. Therefore, we may assume that the array $A$ contains all of the distances for each corresponding point. From this, we can now use QuickSelect to find the $k$ smallest elements in $A$. This has expected running time $O(n)$ due to QuickSelect and every other operation is linear in the worst case. Therefore, the expected running time is $O(n)$.

□

**[K] Exercise 13**. Suppose that we modify the median of medians QuickSelect algorithm to use blocks of 7 rather than 5. Determine a worst-case recurrence, and solve for the asymptotic growth rate.

*Solution.* We first recurse to find the median of the $n/7$ block medians. That median is guaranteed to be greater than four elements in each of $n/14$ blocks, and less than four elements in each of $n/14$ blocks. Therefore the worst case partition is in a ratio of $2 : 5$, and we may have to recursively select from $5n/7$ of the original items. Therefore the recurrence is

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + \Theta(n).$$

Now we can deduce a pattern that arises from the number of items:

- the 2 subproblems at the second level of the recursion have a total of $6n/7$ items,
- the 4 subproblems at the third level of the recursion have a total of $36n/49$ items,
- and so on.

The total size of all subproblems is bounded above by

$$n\left(1 + \frac{6}{7} + \left(\frac{6}{7}\right)^2 + \dots\right) = \frac{n}{1 - \frac{6}{7}} = 7n,$$

so the total time taken (for forming and sorting blocks as well as partitioning) is $\Theta(n)$. □

**[K] Exercise 14**. Given two *sorted* arrays $A$ and $B$, each of length $n$, design a $O(\log n)$ algorithm to find the (lower) median of all $2n$ elements of both arrays. You may assume that $n$ is a power of two and that all $2n$ values are distinct.

*Solution.* If $A[n/2] < B[n/2]$, then $A[n/2]$ is smaller than the median, and $B[n/2+1]$ is larger than the median. We can therefore discard the first half of $A$ and the second half of $B$, and the median of the remaining $n$ elements is our answer.

Similarly, if $A[n/2] > B[n/2]$, we instead discard the second half of $A$ and the first half of $B$. We have transformed an instance of size $n$ to an instance of size $n/2$ using only one comparison, so the recurrence is

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

The critical exponent is $c^* = \log_2 1 = 0$, so by Case 2 of the Master Theorem, the asymptotic solution is $T(n) = \Theta(\log n)$. Note that this is the same recurrence as binary search. □

**[K] Exercise 15**. Given an array $A$ of $n$ distinct integers and an integer $k < n/2$, design an $O(n)$ algorithm which finds the median as well as the $k$ values closest to the median from above and below. You may assume that $n$ is odd.

*Solution.* Apply median of medians quickselect to find the values $k$ ranks below and $k$ ranks above the median (i.e. the $\left(\frac{n+1}{2} - k\right)$th and $\left(\frac{n+1}{2} + k\right)$th smallest values. Then scan through the array, checking whether each element lies between these bounds. □

**[E] Exercise 16**. Suppose that we are given a set $S$ of $n$ elements. Each element has a value and a weight. For an element $x \in S$, let

- $S_{<x}$ be the set of elements whose value is less than the value of $x$,
- $S_{>x}$ be the set of elements whose value is greater than the value of $x$.

Let $R \subseteq S$ be a subset of $S$ and define its weight $w(R)$ to be the sum of the weights of the elements in $R$. The *weighted median* of $S$ is any element $x$ such that $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$.

Design an $O(n)$ algorithm that computes the weighted median of $S$.

*Solution.* This effectively uses the same median of median quickselect algorithm with a slight modification. Once we partition on $x$, we compute the sum of the weights of all elements in $S_{<x}$ (i.e. $w(S_{<x})$) and compute the sum of the weights of all elements in $S_{>x}$ (i.e. $w(S_{>x})$). If both are greater than $1/2$, then $x$ is the weighted median of $S$. Otherwise, one of them is greater than $1/2$ and we recurse on that side. Note that the additional computation takes linear time and so, it doesn't worsen the overall time complexity of our algorithm. □

## § SECTION FOUR: KARATSUBA'S TRICK

**[K] Exercise 17**. Let $P(x) = a_0 + a_1x$ and $Q(x) = b_0 + b_1x$, and define $R(x) = P(x) \cdot Q(x)$. Find the coefficients of $R(x)$ using only three products of pairs of expressions each involving the coefficients $a_i$ and/or $b_j$.

Addition and subtraction of the coefficients do not count towards this total of three products, nor do scalar multiples (i.e. products involving a coefficient and a constant).

*Solution.* We simply use Karatsuba's trick.

Note that

$$\begin{aligned} R(x) &= (a_0 + a_1x)(b_0 + b_1x) \\ &= a_0b_0 + (a_0b_1 + a_1b_0)x + a_1b_1x^2 \\ &= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1] + a_1b_1x^2. \end{aligned}$$

Therefore, we have three multiplications: $a_0b_0$, $a_1b_1$, and $(a_0 + a_1)(b_0 + b_1)$. □

**[K] Exercise 18**. Recall that the product of two complex numbers is given by

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc).$$

(a) Multiply two complex numbers $a+bi$ and $c+di$, where $a, b, c, d$ are real numbers, using only three multiplications.

(b) Find $(a + ib)^2$ using only two multiplications.

(c) Find the product $(a + bi)^2(c + di)^2$ using only five multiplications.

*Solution.*

(a) We see that

$$\begin{aligned} (a + bi)(c + di) &= (ac - bd) + i(ad + bc) \\ &= (ac - bd) + i\left[(a + b)(c + d) - ac - bd\right]. \end{aligned}$$

Therefore, we only need three multiplications: $ac$, $bd$, $(a + b)(c + d)$.

(b) Expanding and simplifying, we have that

$$\begin{aligned} (a + ib)^2 &= a^2 - b^2 + 2abi \\ &= (a - b)(a + b) + (a + a)bi. \end{aligned}$$

We only require two multiplications here: $(a - b)(a + b)$, $(a + a)b$.

(c) We first see that

$$(a + bi)^2(c + di)^2 = \left[(a + bi)(c + di)\right]^2.$$

Using the previous part, computing the square can be done using two multiplications. From part (a), we could compute $(a+bi)(c+di)$ using three multiplications. Therefore, we first compute the three multiplications to find $(a+bi)(c+di)$ and then the remaining two multiplications to compute its square. This gives five multiplications.

□

**[K] Exercise 19**. Let $P(x) = a_0+a_1x+a_2x^2+a_3x^3$ and $Q(x) = b_0+b_1x+b_2x^2+b_3x^3$, and define $R(x) = P(x)\cdot Q(x)$. You are tasked to find the coefficients of $R(x)$ using only twelve products of pairs of expressions each involving the coefficients $a_i$ and/or $b_j$.

Can you do better?

*Solution.* We use Karatsuba's trick. We see that

$$\begin{aligned}
R(x) &= (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3) \\
&= (a_0 + a_1x)(b_0 + b_1x) + (a_0 + a_1x)(b_2 + b_3x)x^2 \\
&+ (a_2 + a_3x)(b_0 + b_1x)x^2 + (a_2 + a_3x)(b_2 + b_3x)x^4 \\
&= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]\,x + a_1b_1x^2 \\
&+ \left(a_0b_2 + [(a_0 + a_1)(b_2 + b_3) - a_0b_2 - a_1b_3]\,x + a_1b_3x^2\right)x^2 \\
&+ \left(a_2b_0 + [(a_2 + a_3)(b_0 + b_1) - a_2b_0 - a_3b_1]\,x + a_3b_1x^2\right)x^2 \\
&+ \left(a_2b_2 + [(a_2 + a_3)(b_2 + b_3) - a_2b_2 - a_3b_3]\,x + a_3b_3x^2\right)x^4.
\end{aligned}$$

Note that each of the last four lines involves only three multiplications. Expanding and regrouping will yield the coefficients of $R(x)$. □