Due Friday 29^{th} of September at 6pm Sydney time (week 3)

In this assignment we review some basic algorithms and data structures, and we apply the divide-and-conquer paradigm. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of "one sympathy mark please" for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the Ed forum. We will maintain a FAQ thread for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on Overleaf, and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. 'General' here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is UNSW Smarthinking, which is the university's official writing support service.

Please review the UNSW policy on plagiarism. Academic misconduct carries severe penalties.

Please read the Frequently Asked Questions document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

Question 1 *Asymptotics*

Read about asymptotic notation in the review material and determine if $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both (i.e., $f(n) = \Theta(g(n))$) or neither of the two, for the following pairs of functions. Justify your answers.

[A] **[5 marks]**

$$f(n) = \sqrt[6]{n^2 - n}; \quad g(n) = \log_2\left(n^5\right)$$

[B] **[5 marks]**

$$f(n) = n; \quad g(n) = (\sin(\pi n) + \cos(\pi n))n$$

[C] **[5 marks]**

$$f(n) = 4^{n^2 \log_2(n)}; \quad g(n) = (n!)^n$$

[D] **[5 marks]**

$$f(n) = n^{2-\cos(\pi n)}; \quad g(n) = n\sqrt{n}$$

You might find L'Hôpital's rule useful: if $f(x), g(x) \to \infty$ as $x \to \infty$ and they are differentiable, then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

Remember that when dealing with asymptotics in computer science, we only care about integer values of n (input sizes are integers).

[A] $g(n) = O(f(n))$.

Firstly, note that for $n \geq 2$ we have $n^2 - n > {}^{n^2}/_{2^6}$ and by log laws $g(n) = 5\log_2 n$, so we can use monotonicity of the sixth root, rearrange, and multiply by $g(n)$ to find

$$\frac{5\log_2 n}{\sqrt[6]{n^2 - n}} < \frac{5\log_2 n}{\sqrt[6]{n^2/2^6}}.$$

We see then that

$$\lim_{n\to\infty} \frac{5\log_2 n}{\sqrt[6]{n^2/2^6}} = 10 \lim_{n\to\infty} \frac{\log_2 n}{n^{1/3}} = 10 \lim_{n\to\infty} \frac{3}{n\ln 2 \cdot n^{-2/3}} = \frac{30}{\ln 2} \lim_{n\to\infty} n^{-1/3} = 0.$$

Since this is an upper bounding limit of ${}^{g(n)}/_{f(n)}$, and terms are positive, this means that $\lim_{n\to\infty} {}^{g(n)}/_{f(n)} = 0$, so that there exists some $\varepsilon > 0$ such that for sufficiently large n we have ${}^{g(n)}/_{f(n)} < \varepsilon$. In other words, there exists some C (specifically ε), such that $g(n) \leq Cf(n)$ for all sufficiently large n. This is the definition of big-Oh notation for $g(n) = O(f(n))$.

To see that this relationship does not go both ways, simply observe that the limit being zero means that ${}^{f(n)}/_{g(n)}$ diverges as $n \to \infty$, so there is no bounding constant.

[B] $f(n) = \Theta(g(n))$.

To approach this question, remember that asymptotics in CS are over the naturals, and comparisons are made with absolute values. With this in mind, when $n \in \mathbb{N}$, we have $\sin(\pi n) = 0$, and $\cos(\pi n) = (-1)^n$, so

$$|g(n)| = |\sin(\pi n) + \cos(\pi n)||n| = |(-1)^n||n| = |n| = |f(n)|.$$

So, clearly $f(n) = \Theta(g(n))$.

[C] $g(n) = O(f(n))$.

We can first simplify $f(n)$ to get

$$f(n) = 4^{n^2 \log_2(n)} = 2^{2n^2 \log(n)} = 2^{\log_2(n^{2n^2})} = n^{2n^2}.$$

Now, since we have $n! < n^n$ for $n > 2$,

$$g(n) = (n!)^n < (n^n)^n = n^{n^2} < n^{2n^2} = f(n).$$

Thus, $g(n) = O(f(n))$.

To see the converse fails, one can use the above to show ${}^{f(n)}/_{g(n)} > {}^{n^{2n^2}}/_{n^{n^2}} = n^{n^2}$, which diverges to infinity as $n \to \infty$, so has no bounding constant.

[D] No relation.

We can intuitively see there is no relation, as $2 - \cos(\pi n)$ oscillates between 1 and 3, so the growth of $f(n)$ will swap between that of n and n^3. We show neither relationship holds by contradiction.

To show $f(n) \neq O(g(n))$, suppose there exists some constant $C > 0$ such that $f(n) \leq Cg(n)$ for sufficiently large n. Consider a sufficiently large and odd n, so that $f(n) = n^3$. Then we have $n^3 \leq Cn\sqrt{n}$, so that $C \geq n\sqrt{n}$. Since C is constant, and independent of n, this is a contradiction. Thus, $f(n) \neq O(g(n))$.

For the other direction, proceed similarly by assuming $g(n) = O(f(n))$ with even n, so that $f(n) = n$. Then $n\sqrt{n} \leq Cn$, so that $C \geq \sqrt{n}$. This is also a contradiction, so we have $g(n) \neq O(f(n))$.

Question 2 *Convenience Store*

Serge owns and runs a convenience store where he purchases items from m different manufacturers and resells them to his customers. Each manufacturer sells items up to n at a time. When buying in bulk from a manufacturer, Serge receives a discount. To find out the cost per item when purchasing i items at a time from manufacturer k, Serge can ask for a quote from the manufacturer, denoted $Q(i, k)$. The manufacturers will give each quote in constant time, but if Serge asks for more than $\lceil \log_2 n \rceil$ quotes from the same manufacturer, he will be blocked for spam and shunned from the business community forever. It is never more expensive to purchase more items from a single manufacturer (that is, $Q(i, k) \geq Q(j, k)$ for all manufacturers k and all quantities of items $i < j$).

Serge also has an array $S[1..m]$, where $S[k]$ is the price he sells the items from manufacturer k for in his shop.

Serge can sell items for a profit whenever the cost he pays per item is cheaper than the price he sells them for in his shop. Serge is interested in determining the smallest number of items he needs to buy from each manufacturer in order to sell the items for a profit. For each manufacturer $1 \leq k \leq m$, we denote this quantity $M[k]$. That is, $M[k]$ is the smallest number of items that Serge must buy from manufacturer k in order to sell the items for a profit. For all manufacturers, $Q(n, k) < S[k]$, so it is always possible for Serge to make a profit.

2.1 [6 marks] Serge is interested in purchasing from a particular manufacturer k. Design an algorithm that determines the smallest number of items Serge needs to purchase from manufacturer k in order to sell them for a profit, without asking for more than $\lceil \log_2 n \rceil$ quotes.

Example: Suppose manufacturer k sells 1 item for \$2, 2 or 3 items for \$1.50 each, and 4 items for \$1.25 each. That is, $Q(1, k) = \$2, Q(2, k) = Q(3, k) = \$1.50$ and $Q(4, k) = \$1.25$. Serge sells items from manufacturer k for \$1.75 each ($S[k] = \$1.75$). The smallest number of items Serge needs to sell to make a profit is 2, so $M[k] = 2$.

$Q(i, k)$ gives the cost per item when i items are purchased from manufacturer k, and $S[k]$ is the price Serge can sell the items for.

Algorithm & Correctness: Given that the price of a single item is non-increasing when you buy more, the underlying $[Q(i, k)]$ price list is non-increasing for a particular manufacturer k. Serge can sell the item from the manufacturer k for a profit if and only if: the buying price per item $Q(t, k)$ (if he buys t items from manufacturer k) is smaller than the fixed selling price per item $S[k]$.

Hence we need to find the smallest value larger than $S[k]$ in a non-increasing list $[Q(i, k)]$.

We can do this by performing a binary search on the underlying list $[Q(i, k)]$.

Complexity: Since the loop of the binary search will run no more than $\lceil \log_2 n \rceil$ turns and in each turn, we will only quote once for $Q(mid, k)$. Hence the total quote number would be no more than $\lceil \log_2 n \rceil$.

2.2 [14 marks] Each of Serge's m manufacturers does not produce items of the same quality. In fact, Serge has numbered them in increasing order of quality, so manufacturer 1 produces the lowest quality items and manufacturer m produces the highest quality items. Serge knows for a fact that when purchasing items from a higher quality manufacturer, he needs to purchase the same or fewer items to sell them for a profit. That is, if $k > l$, then $M[k] \leq M[l]$.

Serge would like to compute $M[k]$ for all m manufacturers. Serge has realised that it is possible to use your method from Q2.1 on each of the m manufacturers for a total time complexity of $O(m \log n)$, but has decided that this is not fast enough.

Assuming $m < n$, design an algorithm that runs in $O(n)$ time to determine the smallest number of items Serge needs to buy from each manufacturer in order to sell the items for a profit while asking each manufacturer for at most $\lceil \log_2 n \rceil$ quotes.

Your algorithm should use the fact that $M[k] \leq M[l]$ for all manufacturers $k > l$.

- Start by looking at exercise 6 in problem set 2, which has similarities to this problem.
- If we know $M[k]$ for some particular k, this gives us information about $M[l]$ for all $l < k$ and all $l > k$. Which value of $M[k]$ gives the most useful information?
- The time complexity analysis is quite difficult for this question. In fact, it's not even obvious what the worst case is! Start by quantifying the amount of work at each level of recursion, then use this to determine what the worst case looks like. You can then develop a recurrence that describes this worst case.

Algorithm & Correctness:
We first find $M[m/2]$. The possible range for this is $[1, n]$ so we can use binary search in this range as the previous part to find out.

Suppose $M[m/2] = x$. We know that $M[i] \geq x$ for all manufacturers $i < m/2$ and that $M[j] \leq x$ for all manufacturers $j > m/2$. Thus, we can recursively solve the same problem on the smaller size for manufacturers $[1, m/2 - 1]$ with a range $[x, n]$ and manufacturers $[m/2 + 1, m]$ with a range $[1, x]$.

For each manufacturer k, $M[k]$ is only calculated once when k is exactly the middle of the recursive range. The range of this binary search is never larger than $[1, n]$ which means each calculation of $M[k]$ will cost no more than $\lceil \log n \rceil$ quotes to the manufacturer k. Hence each manufacturer will not be asked for more than $\lceil \log n \rceil$ times.

Time Complexity:
Let $T(n, m)$ denote the time taken to solve the problem with a maximum of n items and m manufacturers. We have $T(1, m) = O(m)$, and $T(n, 1) = O(\log n)$, and $T(n, m) = T(k, \lfloor \frac{m}{2} \rfloor) + T(n - k + 1, m - \lfloor \frac{m}{2} \rfloor - 1) + O(\log n)$ where k is the minimum item for the manufacturer in position $m/2$ in the table. Making the simplifying assumption that this

recurrence

$$\begin{aligned}
T(n,m) &\approx T(x_1^{(1)}, \left\lfloor \frac{m}{2} \right\rfloor) + T(x_2^{(1)}, \left\lfloor \frac{m}{2} \right\rfloor) + O(\log n) && \text{(with } x_1^{(1)} + x_2^{(1)} = n+1\text{)} \\
&= \cdots \\
&= \sum_{u=0}^{\lceil \log_2 m \rceil} \sum_{v=1}^{2^u} O\left(\log x_v^{(u)}\right) && \text{(with } \sum_{v=1}^{2^u} x_v^{(u)} = n + 2^u - 1\text{)} \\
&= O\left(\sum_{u=0}^{\lceil \log_2 m \rceil} \sum_{v=1}^{2^u} \log x_v^{(u)} \right) \\
&= O\left(\sum_{u=0}^{\lceil \log_2 m \rceil} \log \left(\prod_{v=1}^{2^u} x_v^{(u)} \right) \right) \\
&\leq O\left(\sum_{u=0}^{\lceil \log_2 m \rceil} 2^u \log \left(\frac{\sum_{v=1}^{2^u} x_v^{(u)}}{2^u} \right) \right) && \text{(by the AM/GM inequality)}
\end{aligned}$$

with equality only occurring when $x_{v_1}^{(u)} = x_{v_2}^{(u)}$ for all valid v_1, v_2 and for all u (that is, when $x_1^{(1)} = x_2^{(1)} = n/2$, $x_2^{(2)} = x_2^{(2)} = x_3^{(2)} = x_4^{(2)} = n/4$ and so on). Since $T(n,m)$ is maximised when the equality condition is met in the AM/GM inequality, we must have:

$$\begin{aligned}
T(n,m) &\leq 2T(n/2, m/2) + O(\log n) \\
&\leq 2T(n/2, n/2) + O(\log n) \\
&= O(n)
\end{aligned}$$

where the last line follows from case 1 of the master theorem.

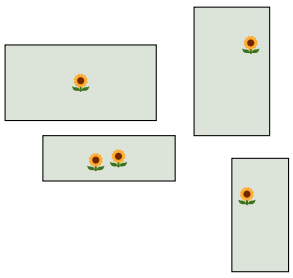
Note for students:
We do not expect you to produce time complexity justification as rigorous as this. Marks were awarded as long as you provided some reasonable explanation as to why the worst case time complexity is equivalent to the recurrence of the form $T(n) = 2T(n/2) + O(\log n)$.

For example, you could show that at each level of recursion, if we are working on subarrays with $n_1, n_2, \ldots, n_k$ elements, we perform a binary search on each subarray. The time complexity at each level is then $\log(n_1) + \log(n_2) + \cdots + \log(n_k)$, which is equivalent to $\log(n_1 n_2 \ldots n_k)$ by applying the fact that $\log(a) + \log(b) = \log(ab)$. The worst case will be the one that maximises $\log(n_1 n_2 \ldots n_k)$ at each level, which is equivalently the case that maximises $n_1 n_2 \ldots n_k$. This is maximised when $n_1 = n_2 = \cdots = n_k$ (this can be shown in various ways, including the AM/GM inequality or reducing to the case with just two variables), which occurs when we split the problem in half at each step, giving the recurrence $T(n) = 2T(n/2) + O(\log n)$.

Question 3 *In the Night Garden*

Alice is planting n_1 flowers $f_1, \ldots, f_{n_1}$ among n_2 rectangular gardens $\mathcal{G}_1, \ldots, \mathcal{G}_{n_2}$. Bob's task is to determine which flowers belong to which gardens. Alice informs Bob that no two gardens overlap; therefore, if a flower belongs to a garden, then the flower belongs to *exactly* one garden and a garden can contain multiple flowers. If a flower f_i does not belong to any garden, then Bob returns an *undefined* garden for f_i.

Each garden $\mathcal{G}_i$ is given by a pair of points $\mathcal{G}_{BL}[i]$ and $\mathcal{G}_{TR}[i]$, representing the bottom left and top right corners of the garden respectively. Each flower is represented by a point $F[i]$ representing its location. Let $n = n_1 + n_2$.



A collection of $n_1 = 5$ flowers and $n_2 = 4$ gardens.

More formally, you are given three arrays:

- $\mathcal{G}_{BL} = [(x_1, y_1), \ldots, (x_{n_2}, y_{n_2})]$, where $\mathcal{G}_{BL}[i] = (x_i, y_i)$ represents the bottom left point of garden $\mathcal{G}_i$,
- $\mathcal{G}_{TR} = [(x_1, y_1), \ldots, (x_{n_2}, y_{n_2})]$, where $\mathcal{G}_{TR}[i] = (x_i, y_i)$ represents the top right point of garden $\mathcal{G}_i$, and
- $F = [(x_1, y_1), \ldots, (x_{n_1}, y_{n_1})]$, where $F[i] = (x_i, y_i)$ represents the location of flower f_i.

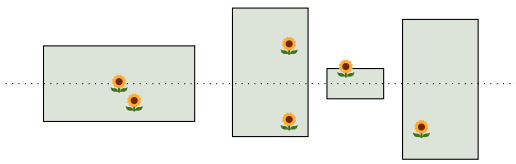
You are not guaranteed that a flower belongs to a garden; it is possible that a flower is planted in none of the gardens. Your goal is to return the garden that a flower f_i belongs to (if any), for *each* f_i.

- For example, in the diagram above, if the flower ordering is sorted by its x-coordinate and the rectangular gardens are sorted by their x-coordinate of $\mathcal{G}_{BL}$, then we return

$$f_1 : \mathcal{G}_1, \quad f_2 : \mathcal{G}_2, \quad f_3 : \mathcal{G}_2, \quad f_4 : \mathcal{G}_4, \quad f_5 : \mathcal{G}_3.$$

  Your output can be in any form (e.g. array, list, set, dictionary), so long as you maintain the correct pair of flowers and gardens.

3.1 [12 marks] We first solve the special case where all of the gardens intersect with a horizontal line. Design an $O(n \log n)$ algorithm to determine which flowers belong to which gardens (if such a garden exists).



A collection of $n_1 = 6$ flowers and $n_2 = 4$ gardens that intersect with a horizontal line.

Hint. What do you know about two adjacent gardens if they have to intersect with a horizontal line?

Following our hint, all the gardens intersecting at a horizontal line implies that for all gardens, we must have their horizontal coordinates form disjointed intervals. From now on any $f_i \in F$, we use $(\bar{x}_i, \bar{y}_i)$ to represent its location. For any $\mathcal{G}_j$, we denote its top right and bottom left coordinate as (x_j^*, y_j^*) and (x'_j, y'_j), respectively.

Algorithm: We start by sorting the sets of (x_j^*, y_j^*) and (x'_j, y'_j) coordinates (i.e., the coordinates in $\mathcal{G}_{TR}$ and $\mathcal{G}_{BL}$) separately in ascending order of the x-coordinate using MERGESORT. Since all the horizontal intervals are disjointed, (x_j^*, y_j^*) and (x'_j, y'_j) for a given $\mathcal{G}_j$ would have the same index in the corresponding sorted set. Then for each flower i we proceed with a modified version of binary search with initial range of $[1, n_2]$ on the set of all gardens and initial pivot $\mathcal{G}_j$ where $j = \lfloor \frac{1+n_2}{2} \rfloor$. We recursively proceed with the algorithm under the following cases:

- [A] If $\bar{x}_i \in (x_j^*, x'_j)$ and $\bar{y}_i \in (y_j^*, y'_j)$ then we have that $f_i : \mathcal{G}_j$,
- [B] on the contrary; if $\bar{x}_i \leq x_j^*$ then we update the upper bound of our search space to $(j-1)$, else if $\bar{x}_i > x'_j$ then the lower bound of our search space changes to $(j+1)$. The middle element of the updated search space would be chosen as the pivot for the next iteration.

In the case where [A] is false until the search space reduces to $\emptyset$, then we return that f_i does not have a corresponding garden.

Correctness: Note that each garden does not overlap and all of them intersect on the horizontal line would imply that for any $j \neq k, (x_j^*, x'_j) \cap (x_k^*, x'_k) = \emptyset$. Hence if $\bar{x}_i \in (x_j^*, x'_j)$ then there can only exist 1 possible solution $\mathcal{G}_j$ as any other $\mathcal{G}_k$ will intersect the horizontal line and overlap $\mathcal{G}_j$ forming a contradiction. Furthermore, as our ranges are sorted, $\bar{x}_i \leq x_j^*$ would imply that no solutions for any garden $\mathcal{G}_k$ with $x_k^* > x_j^*$.

Time complexity: Each MERGESORT takes $O(n_2 \log(n_2))$ time, then for each flower f_i our modification of binary search which reduces the search space by half with our pivot takes $O(1)$ time at each iteration. This process then costs us a run time complexity of $O(n_1 \log(n_2))$. In total, our algorithm will run in

$$O(n_2 \log(n_2)) + O(n_1 \log(n_2)) = O((n_2 + n_1) \log(n_2)) = O(n \log(n))$$

time as required.

3.2 **[8 marks]** We now remove the assumption that every garden intersects with a horizontal line. Design an $O(n(\log n)^2)$ algorithm to determine which flowers belong to which gardens (if such a garden exists).

Hint. Divide and conquer, and use the previous part as a subroutine.

Since n depends on two factors, you need to be careful with where you choose to divide the input. Is it good enough to just split only the gardens or only the flowers?

Consider how ideas from 3.1 can be used to split the input. For any given horizontal line, we can divide the flowers into 2 sets and the gardens into 3 sets. If a flower is in one of these sets, what sets could its garden possibly be in?

There is also an $O(n \log n)$ solution...

We use divide and conquer by considering the median y-coordinate of the gardens *and* flowers.

Algorithm: Sort $\mathcal{G}_{TR}$, $\mathcal{G}_{BL}$ and F, keeping track of the original garden $\mathcal{G}_i$ and flower f_i indices as you sort. Let y_m denote the median among the y-coordinates of $\mathcal{G}_{TR}$ and F.

Consider all of the gardens that are completely above the median line (i.e. the y-coordinate of $G_{BL}[i]$ is above the $y = y_m$ line) and call it G_U. Conversely, consider all of the gardens that are completely below the median line (i.e. the y-coordinate of $G_{TR}[j]$ is below the $y = y_m$ line) and call it G_L. Finally, consider all of the gardens that intersect the median line and call them G_M.

Likewise, consider all of the flowers that are completely above the median line (i.e. the y-coordinate of F is above the line $y = y_m$) and call it F_U. Conversely, consider all of the flowers that are completely below the median line (i.e. the y-coordinate of F is below the line $y = y_m$ and call it F_L.

We now perform our divide-and-conquer algorithm.

- **Divide**: We divide our flower input into F_U and F_L according to the above description, and we divide our garden input into G_U, G_L, and G_M according to the above description.
- **Conquer**: Each half is recursively solved by applying the algorithm on G_U and F_U to obtain the solutions to the top half, and applying the algorithm on G_L and F_L to obtain the solutions to the bottom half, until we arrive at the base case where $n_1 + n_2 \leq 1$ (i.e. there is either one flower or one garden).
- **Combine**: To combine the solutions, we apply 3.1 on G_M to locate all of the flowers that lie in gardens in G_M (i.e. the merge step calls the solution of 3.1 on G_M and F).

Correctness: The algorithm from 3.1 retrieves all of the flowers and returns their respective gardens. To argue the correctness of our algorithm, it suffices to argue that the description of our divide step is correct since the combine step arises from the correctness of 3.1.

- Consider all of the flowers that belong to some garden in G_U. Since these gardens are completely above the median line, such flowers must also be located completely above the median line; in other words, these flowers must belong to F_U.
- Likewise, consider all of the flowers that belong to some garden in G_L. Since these gardens are completely below the median line, such flowers must also be located completely below the median line; in other words, these flowers must belong to F_L.

This shows that our recursive step is correct. The base case is trivial to solve and we can always arrive at the base case. If $n_1 + n_2 > 1$, then either:

- we have at least one garden and one flower, in which case the recursive step will split the input into our base case;
- we have at least two gardens and no flowers, in which case the recursive step will split the gardens according to the median and we eventually arrive at our base case;
- or we have at least two flowers and no gardens, which follows from the previous case.

In all three cases, the recursive step will eventually recurse down to our trivial base case which can be solved in constant time.

N.B. This tells us that it is in fact not enough to just recurse on the flowers or gardens alone since the base case may not necessarily be reached. In fact, our recursive step requires us to look at both the flowers and the gardens.

Time complexity: We finally argue the time complexity bound. Preprocessing the array requires sorting via merge sort which takes $O(n \log n)$ time.

- Let $T(n)$ denote the amount of time required to solve the problem on an input where $n_1 + n_2 = n$. The merging step clearly takes $O(n \log n)$ since it is just calling the subroutine from 3.1. We now argue that $|G_L| + |F_L|$ and $|G_U| + |F_U|$ are both bounded by $n/2$ (ignoring floors and ceilings).
  - This comes directly from the median line chosen since by definition, the median line will split the flowers and gardens approximately into two equal parts.
  - Ignoring floors and ceilings in our analysis, we have that $|G_L| + |F_L| \leq n/2$ and $|G_U| + |F_U| \leq n/2$.
- Since we are recursively solving each of these halves, our recurrence becomes

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

We can then unroll the recurrence to obtain

$$\begin{aligned}
T(n) &\leq 2T(n/2) + cn \log n \\
&\leq 2\left[2T(n/4) + c(n/2)\log(n/2)\right] + cn \log n \\
&\leq 4T(n/4) + 2cn \log n \\
&\leq \dots \\
&\leq 2^k T(n/2^k) + kcn \log n \\
&\leq 2^{\log_2 n} T(1) + cn \log_2 n \log n && (k = \log_2 n) \\
&= n + cn\,(\log n)^2 \\
&= O(n\,(\log n)^2),
\end{aligned}$$

as required.

N.B. It is actually possible to reduce the time complexity even further to $O(n \log n)$. What preprocessing step could you do to improve the running time even further?