EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 3

This homework has 8 questions, for a total of 100 points and 9 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up*.

(0 pts) 0. **Before you start; before you submit.**

(a) Carefully review What to give when you "give an algorithm" before starting this assignment, and apply it to the solutions you submit.

(b) If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

Solution:

(10 pts) 2. **Bodybuilders in the magic room.**

You are at a party with n bodybuilders having distinct weights, and want to identify the kth lightest person, for some particular k. You are allowed to ask questions of the form "Is person A lighter than person B?" An easy solution is to use a sorting algorithm, which takes $\Theta(n \log n)$ comparisons. However, you suspect you can do better because you have access to a "magic room."

The "magic room" works as follows: if you bring any S people into this room, it will identify the person whose weight is the $\lceil S/2 \rceil$th lightest among them. Give an algorithm for the above problem that uses only $O(n)$ comparisons and $O(\log n)$ accesses to the magic room.

Solution: The algorithm resembles binary search. If $n = 1$, then $k = 1$ as well, so just return the single person. Now, suppose $n \geq 2$.

1. Bring all n people into the magic room, which identifies the $\lceil n/2 \rceil$th lightest person x.

2. Compare x to everyone else using $n-1$ comparisons. Let S_{lo} be the set of all people who are lighter than x, and S_{hi} be the set of all people who are heavier than x.

3. Finally, there are three cases. If $k = \lceil n/2 \rceil$, then return x. If $k < \lceil n/2 \rceil$, then recursively solve the problem on S_{lo} with parameter k. If $k > \lceil n/2 \rceil$, then recursively solve the problem on S_{hi} with parameter $k - \lceil n/2 \rceil$.

For correctness, observe that if $k = \lceil n/2 \rceil$ then the person identified by the magic room is exactly the one we seek. If $k < \lceil n/2 \rceil$, then we remove people who are heavier than the one we seek, so we still seek the kth lightest person, which the recursive call will identify. If $k > \lceil n/2 \rceil$, then we remove $\lceil n/2 \rceil$ people who are lighter than the desired one, so we now seek the $(k - \lceil n/2 \rceil)$th lightest person from the remaining ones, which the recursive call will identify.

By inspection, the recurrence for the number of comparisons is $C(n) = C(n/2) + (n-1) = C(n/2) + O(n)$. By the Master Theorem, $C(n) = O(n)$. (Alternatively, we could use substitution to "unroll" the recurrence, to get $C(n) \leq n + n/2 + n/4 + \cdots = 2n = O(n)$.) Similarly, the recurrence for the number of accesses to the magic room is $R(n) = R(n/2)+1$. The Master Theorem (or substitution) gives $R(n) = O(\log n)$, as desired.

3. **Pikachu: I trade you!**

You are a savvy investor who is interested in an especially hot asset that is now on the market: a rare Pikachu Pokémon trading card. You want to purchase and sell this card *just once*, and time these events to maximize your rate of return, i.e., the percentage by which the card's value increases between purchase and sale.

Your top trading-desk researchers have prepared their best estimates of how much the card's value will change on each of the following n days. These estimates are given as nonnegative real *multiplicative* factors, i.e., a 10% gain is represented by 1.1, and a 20% loss is represented by 0.8. Given the array $A[1, \ldots, n]$ of these estimates, you want to find indices $1 \leq i \leq j \leq n$ that *maximize*

$$A[i] \cdot A[i+1] \cdots A[j] \ ,$$

i.e., the aggregate growth in value if the card is purchased at the start of day i and sold at the end of day j. (If the card would lose value every day, this can be indicated by taking $i > j$.)

For instance, if $A[1, \ldots, 6] = [0.8, 1.1, 0.95, 1.5, 0.8, 1.25]$, then the maximum aggregate growth is 1.5675, which is achieved by $i = 2$, $j = 6$. (Note that it is also achieved by $i = 2$, $j = 4$. So, an optimal choice of i and j is not necessarily unique, but the maximum aggregate growth is unique.)

In this question you will design a divide-and-conquer algorithm that solves this problem. You may assume that two real numbers can be multiplied in constant time, and for simplicity, you may assume that n is a power of two (if you wish).

Solutions to Homework 3

(7 pts) (a) Suppose that for some index k of interest, you want to optimize the aggregate growth under the constraint that you own the card on day k and sell it strictly after day k (i.e., you purchase it at the start of some day $i \leq k$ and sell it at the end of some day $j > k$). Give an algorithm that solves this problem in $O(n)$ time. On input $A[1, \ldots, n]$ and k, it should output the maximum aggregate growth, and corresponding purchase and sale days i, j that achieve it.

Solution: The key idea is to compute an optimum subarray $A[i, \ldots, k]$ that ends at index k, and an optimum subarray $A[k+1, \ldots, j]$ that begins with index $k+1$. Then, we return the union $A[i, \ldots, j]$ of these two subarrays, which we show below is an optimum subarray under the constraint that $i \leq k < j$. This algorithm is made precise in the following pseudocode.

```
1: function MaxGrowthIncluding(A[1, ..., n], k)
2:     a ← A[k], a_temp ← a, i ← k
3:     for z = k − 1, ..., 1 do
4:         a_temp ← A[z] · a_temp
5:         if a_temp > a then a ← a_temp, i ← z
6:     b ← A[k + 1], b_temp ← b, j ← k + 1
7:     for z = k + 2, ..., n do
8:         b_temp ← b_temp · A[z]
9:         if b_temp > b then b ← b_temp, j ← z
10:    return (g = ab, i, j)
```

This algorithm takes $O(n)$ time because there are $O(n)$ iterations, each of which takes only $O(1)$ time.

Correctness: By inspecting the code, it can be seen that the aggregate growth a of $A[i, \ldots, k]$ is maximized among all subarrays ending with index k, and the aggregate growth b of $A[k+1, \ldots, j]$ is maximized among all subarrays starting with index $k+1$. We claim that the aggregate growth $g = ab$ of $A[i, \ldots, j]$ is maximized among all subarrays for which $i \leq k < j$. For if not, there is some other subarray $A[i', \ldots, j']$ with $i' \leq k < j'$ that has larger aggregate growth $g' > g$. Splitting this subarray, let a' and b' be the aggregate growths of $A[i', \ldots, k]$ and $A[k+1, \ldots, j']$, respectively. Then $g' = a'b' > g = ab$, so either $a' > a$ or $b' > b$ (or both). But this contradicts the optimality of either $A[i, \ldots, k]$ or $A[k+1, \ldots, j]$ (or both). This completes the proof.

(8 pts) (b) Now, give a divide-and-conquer algorithm that solves the main problem in $O(n \log n)$ time. Given the array $A[1, \ldots, n]$, it should output the optimal aggregate growth.

Solution: The key idea is as follows: we split the input array into two halves. Then, we recursively find an optimum solution within the first half, and similarly for the second half. Then, we use the algorithm from the previous part to find an optimum

solution whose starting point is in the first half and whose ending point is in the second half, by setting the argument k to be the midpoint of the array. Finally, we return the best of the three solutions (or some $i > j$ if none of these options are profitable). Precise pseudocode is as follows. (Although this was not required, this code returns both the optimum aggregate growth *and* indices i, j that achieve it, since these are what an investor actually wants to know.)

```
1: function MaxGrowth(A[1, ..., n]))
2:     if n = 1 then
3:         return (A[1], i = 1, j = 1), or (1, i = 1, j = 0) if A[1] < 1
4:     (g_l, i_l, j_l) ← MaxGrowth(A[1, ..., n/2])
5:     (g_r, i_r, j_r) ← MaxGrowth(A[n/2 + 1, ..., n]) + (0, n/2, n/2)    ▷ shift indices to back half of array
6:     (g_c, i_c, j_c) ← MaxGrowthIncluding(A[1, ..., n], n/2)
7:     return one of the above tuples with largest growth, or (1, 1, 0) if all growths are less than 1
```

Correctness: For the base case $n = 1$, we can either buy the card at the start of day 1 and sell it at the end of day 1, or not. If we buy and sell, the growth is $A[1]$ and $i = j = 1$. Otherwise, the growth is 1 and $i > j$ to indicate that no trading is done. The algorithm outputs an optimum choice between these two options.

For the recursive case, suppose that i^* and j^* are an optimum choice for the entire array, i.e., the product $A[i^*] \cdot A[i^*+1] \cdots A[j^*]$ is maximized among all possible choices. There are only four possible cases: (1) $i^*, j^* \leq n/2$, (2) $i^*, j^* > n/2$, (3) $i^* \leq n/2 < j^*$, (4) it is more profitable not to trade at all.

In the first case, the first recursive call will return an optimum solution for the first half, which is therefore an optimum for the entire array. (It may not return the exact values i^*, j^*, because there could be other equally profitable choices in the first half, but it will return some optimum pair.) In the second case, the second recursive call will similarly return an optimum solution for the entire array, but with the indices reduced by $n/2$; the "shift" fixes this. In the third case, the call to MaxGrowthIncluding will similarly return an optimum solution. In the fourth case, all of the subroutine calls will return growths less than 1. In every case, our algorithm's answer is an optimum for the entire array.

Running Time: Let $T(n)$ be the algorithm's running time on an array of n elements. Lines 4 and 5 each take $T(n/2)$ time, while Line 6 takes $O(n)$ time, as shown in the previous part. The remaining lines take $O(1)$ time. Thus, the running time satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) = O(n \log n)$ by the Master Theorem.

4. **Playing at Pinball Pete's.**

There is a game at Pinball Pete's in which you need *exactly* p points to win. You start at zero points, and can score one, two, or three points per turn. You are interested in the number of different ways a person can win. Different orderings of the same numbers of points count as distinct ways to win (for example, if $p = 3$, then scoring 1 point and then 2 points is different than scoring 2 points and then 1 point). Define $SP(p)$ to be the number of ways to score exactly p points.

(10 pts) (a) Derive, with justification, a recurrence relation for $SP(p)$. Remember to include the base case(s).

Solution: There are three bases cases:

- $SP(0) = 1$: there is exactly one way to win because we have 0 points, and we need 0 points, so we have already won.
- $SP(1) = 1$: there is exactly one way to win, which is by scoring 1 point.
- $SP(2) = 2$: we can win by scoring 1 point twice or by scoring 2 points once, so there are two ways to win.

The recursive case is $p \geq 3$. Since winning requires at least one turn, there are *exactly three* distinct routes to get exactly p points: first get $p - 1$ points and then score 1 point; first get $p - 2$ points and then score 2 points; or first get $p - 3$ points and then score 3 points. So the number of ways to get exactly p points is $SP[p] = SP[p-1] + SP[p-2] + SP[p-3]$.
Putting all this together yields the recurrence

$$SP[p] = \begin{cases} 1 & \text{if } p \leq 1, \\ 2 & \text{if } p = 2, \\ SP[p-1] + SP[p-2] + SP[p-3] & \text{otherwise.} \end{cases}$$

Alternatively, we could define the base cases as $SP(i) = 0$ for all *negative* $i < 0$, because there is no way to get a negative number of points, and $SP(0) = 1$, for the reason given above. Then the same recurrence as above holds for all $p \geq 1$, for the same reasons. This yields the same results as above, but we do not even need to manually determine the number of ways to get 2 or 3 points.

(10 pts) (b) Give pseudocode for a (bottom-up) dynamic-programming algorithm that, on input p, outputs $SP(p)$, the number of ways to get exactly p points in $O(p)$ time. Is the algorithm "efficient," as defined by this course? (For simplicity, here you may assume that integer addition takes constant time.)

Solution: The pseudocode is as follows:

Input: Non-negative integer p, the number of points needed to win

```
1: function NumWaysToScore(p)
2:     allocate SP[0, ..., p]
3:     SP[0] ← 1
4:     SP[1] ← 1
5:     SP[2] ← 2
6:     for i = 3, ..., p do
7:         SP[i] ← SP[i − 1] + SP[i − 2] + SP[i − 3]
8:     return SP[p]
```

Correctness: The algorithm uses the recurrence from the previous part to compute the values of $SP[i]$ in order, from $i = 0, \ldots, p$. Because the expression for $SP[i]$ uses only values of $SP[j]$ for various $j < i$, the needed values have already been computed when they are used. Therefore, the algorithm computes correct values for all $SP[i]$.

Running time analysis: This algorithm runs in $\Theta(p)$ time because there are about p iterations, each of which takes $\Theta(1)$ time. However, the input *size* is $\Theta(\log p)$, because the size of an integer (in digits) is the logarithm of its value. Thus, the running time is *exponential*, not polynomial, in the input size, so the algorithm is *not* efficient according to the definition used in this class.

(3 EC pts) (c) *Optional extra credit.* There is another popular game with some strange scoring rules. In each "turn" it is possible to score two, three, or six points. Moreover, immediately after (and only after) scoring six points, one can attempt to score either one or two "extra" points, but this attempt might fail (resulting in zero extra points).

Define $FP(p)$ to be the number of ways to score exactly p points in this game (again, order matters). As above, derive (with justification and base case(s)) a recurrence relation for $FP(p)$, give pseudocode for a dynamic-programming algorithm that outputs $FP(p)$ given input p, and analyze its running time.

Solution: The key insight is that scoring six points, followed by the results of the extra-point(s) attempt, is equivalent to simply scoring either six, seven, or eight points, using alternative scoring rules with no extra-point procedure. That is, we can uniquely rewrite any sequence of scoring events from the original game as a sequence under the alternative rules, and vice versa. (For this it is important that "score six, then score zero by failing the extra-point(s) attempt, then score two" is considered different from "score six, then score two extra points"; see the clarification on Piazza 329 about this.)

So, the recurrence is

$$FP(p) = FP(p-2) + FP(p-3) + FP(p-6) + FP(p-7) + FP(p-8)\ .$$

Due to the several terms here, it is simplest to define the base cases as $FP(i) = 0$ for all negative $i < 0$, and $FP(0) = 1$, as we did in the alternative answer to part (a).

5. **A pebble game.**

Many two-player strategic games, like the several variants of Nim, can be modeled as follows. There is a directed acyclic graph $G = (V, E)$ presented in topological order: the vertices are labeled as $V = \{1, \ldots, n\}$, and every edge $(u, v) \in E$ has $u < v$. Moreover, every vertex $i < n$ has at least one outgoing edge.

Two players, called maize and blue, take turns in the following game on this graph. They start with a pebble at vertex 1, and maize plays first. On each turn, the acting player must move the pebble from the current vertex u to a new vertex v along some edge $(u, v) \in E$ of the graph, of the acting player's choice. If a player moves the pebble to vertex n (the final one), then that player wins the game.

Give a dynamic-programming algorithm that, given the graph G as input, determines which player can be guaranteed a win by playing perfectly, no matter how the opponent plays.

(10 pts) (a) Define $W(i)$ to indicate whether the acting player can guarantee a win if the pebble is at vertex i. Derive, with justification, a recurrence relation for $W(i)$. Remember to include appropriate base case(s).

Solution: The base case and recurrence for W is:

$$W(i) = \begin{cases} \text{false} & \text{if } i = n, \\ \bigvee_{j:(i,j)\in E} \neg W(j) & \text{otherwise.} \end{cases}$$

The justification for the base case is: if the pebble is at vertex n, then the acting player has lost, so there is no possible way to win, and therefore $W(n) = \text{false}$.

For the recursive case: if the pebble is at vertex $i < n$, then the acting player can guarantee a win exactly when it can make a move to put its opponent in a "losing position," i.e., one for which there is no winning strategy. In other words, i is a winning position exactly when there is some edge $(i, j) \in E$ for which the opposing player *cannot* guarantee a win from j, i.e., $W[j] = \text{false}$. (The above "OR of NOTs" is also equivalent to a "NOT of ANDs," by De Morgan's laws.)

(10 pts) (b) Give pseudocode for a (bottom-up) dynamic programming algorithm that solves the problem, and analyze its running time.

Solution: Using the recurrence we get the following dynamic programming algorithm. Notice that the algorithm fills the table W from "back" to "front," which is valid because every edge $(i, j) \in E$ has $i < j$, so when setting $W[i]$ all the needed values of $W[j]$ have already been correctly set. The running time is $O(n + m)$, where $m = |E|$ is the number of edges in the graph, because the inner loop examines each edge in the graph exactly once.

```
1: allocate table W[1...n] of booleans
2: W[n] = false
3: for i = n − 1 down to 1 do
4:     W[i] ← false
5:     for all edges (i, j) ∈ E (walk the adjacency list of i) do
6:         if W[j] = false then W[i] ← true
```

(5 pts) (c) Extend your algorithm from the previous part to also output a "winning strategy" for whichever player has one. This should be represented as an array specifying, for each vertex u, where the player should move the pebble if it is at vertex u on that player's turn. Also, analyze the algorithm's running time.

Solution: Let Next$[1 \dots n]$ be an array, where Next(i) indicates where a player can move to, from position i, in a winning strategy. (If there is no winning strategy from position i, the entry can be undefined or some failure symbol.)

For each vertex $i < n$ where $W[i] =$ true, there must exist at least one edge (i, j) where $W[j] =$ false, due to the recurrence for $W[i]$ in the previous parts. So, when we set $W[i] =$ true in the algorithm from the previous part, we also set Next$[i] = j$ for some such j. When we set $W[i] =$ false, we leave Next$[i]$ undefined.

This means that whenever the acting player is in a position with a winning strategy, they will place their opponent in a position that does not have a winning strategy. So, no matter what move the opponent makes (if the game has not already ended with a loss for the opponent), it will place the original acting player in a position with a winning strategy again, and this will repeat until the game ends with the original player winning.

This modification preserves the $O(n + m)$ running time of the previous algorithm because it does just $O(1)$ additional work when setting each $W[i]$.

6. **Houston, we have a problem!**

Alina and Kaitlyn decide to make the drive from Ann Arbor to Houston to cheer on the Wolverines at the College Football National Championship Game! They start their drive from Ann Arbor and make their way to Houston. Along their journey, there are n hotels, located at positive distances $a_1 < a_2 < \cdots < a_n$ from Ann Arbor. They can stop only at these specified hotels, but may choose which one(s) to spend their night(s) at, and which ones to skip. They must end their journey at the last hotel, located at distance a_n.

The trip from Ann Arbor to Houston is 1300 miles. They don't want to drive too much or too little in a day, so they are aiming to go about 500 miles per day. However, due to the distance between the hotels, this may not always be possible. If they drive x miles on a given day, they will face a "penalty" of $(500 - x)^2$ for that day.

For example, if $a_1 = 450, a_2 = 670, a_3 = 1300$, then we have the following possible choices of

hotels (by their indices) and the associated penalties:

$$\begin{aligned} 1,2,3 &\to (500-450)^2+(500-220)^2+(500-630)^2 &&= 97800 \\ 2,3 &\to (500-670)^2+(500-630)^2 &&= 45800 \\ 1,3 &\to (500-450)^2+(500-850)^2 &&= 125000 \\ 3 &\to (500-1300)^2 &&= 640000 \end{aligned}$$

Therefore, the minimum possible total penalty is 45,800. Alina and Kaitlyn should drive to the second hotel (at 670 miles) on the first day, then drive the remaining 630 miles to Houston on the second day.

The goal in this problem is to devise an efficient algorithm that, given an array $A[1 \ldots n]$ of the distances $a_1, \ldots, a_n$, determines which hotel(s) to stop at so as to minimize the total penalty incurred on the trip.

Hint: Consider a function $p(j)$ that represents the minimum possible total penalty when starting from Ann Arbor and ending at hotel j.

(10 pts) (a) Give, with justification, a recurrence relation for $p(j)$ that is suitable for a dynamic-programming solution to the problem. Be sure to identify the base case(s).

Solution: As a base case, we have $p(1) = (500 - a_1)^2$ because this is the penalty incurred if we end our (first and only) day at the first hotel.

There are two cases for ending at hotel j: either we started our day from some previous hotel, or we are in the first day and started from Ann Arbor.

- If we started the day from a previous hotel, the total trip penalty ending at hotel j will consist of: (1) the total trip penalty up to but not including the last day's penalty, plus (2) the penalty from the last day of the trip. We want to minimize the total of these two components over all possible prior hotels, which is $\min_{i<j}\{p(i) + (500 - (a_j - a_i))^2\}$.
- If we started the day from Ann Arbor, then the total penalty is just $(500 - a_j)^2$.

The minimum possible penalty is the minimum of these two cases. Combining everything we know, the applicable recurrence relation is:

$$p(j) = \min\{\min_{i<j}\{p(i) + (500 - (a_j - a_i))^2\}, (500 - a_j)^2\} \; .$$

Alternative solution: We can simplify the above by defining $a_0 = 0$, a phantom hotel at 0 miles (in Ann Arbor). The recurrence then simplifies slightly to:

$$p(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{i<j}\{p(i) + (500 - (a_j - a_i))^2\} & \text{otherwise.} \end{cases}$$

The reasoning is the same as above, with starting the day from a_0 equivalent to starting from Ann Arbor.

(10 pts) (b) Give pseudocode for a (bottom-up) dynamic programming algorithm that solves the problem, and analyze its running time.

Solution:

Input: $A[1, \ldots, n]$ that represents distance of hotels from the start point

```
1: function MinPenalty(A[1, ..., n])
2:     allocate p[0, ..., n]
3:     p[0] ← 0
4:     for j = 1, ..., n do
5:         p[j] = ∞
6:         for i = 1, ..., j − 1 do
7:             p[j] ← min(p[j], p[i] + (500 − (A[j] − A[i]))^2)
8:     return p[n]
```

Correctness: by inspection, the code computes the values $p[j]$ according to the above recurrence, from $j = 0, \ldots, n$. Because the expression for $p[j]$ uses only values of $p[i]$ for $i < j$, the needed values have already been computed when they are used. Therefore, the algorithm computes correct values for all $p[j]$.

Running time: This algorithm has a running time of $O(n^2)$. Allocating p takes $O(n)$ time. Filling each entry of p takes $O(n)$ iterations of the inner for loop, which each take constant time. Since there are n iterations of the outer for loop, the overall running time is $O(n^2)$. All other operations take constant time.

Finding an optimum sequence of hotels: The above algorithm just finds the minimum penalty that can be obtained, not an actual sequence of hotels that obtains it. This can be done in the usual way by storing some extra information in a separate table $b[1, \ldots, n]$, and "backtracking." Specifically, whenever we change a $p[j]$ value, we store the corresponding value of i that induced the change in $b[j]$. Then, at the end, starting from $j = n$, we backtrack from $b[n]$ to $b[b[n]]$ etc., taking each value as one of the hotels to stay. This takes just $O(n)$ additional time.

(6 EC pts) 7. **Optional extra credit: flash mob!**

You've accepted a summer internship at a startup that's making a new flash mob app. Here's how it works: Given n people in the plane, the app will periodically convene a flash mob of size k, where k is some constant. The objective is to find a size-k subset of the n people so that the sum of the pairwise distances between those k people is as small as possible. In other words, given a set of n points, we wish to find a subset P of exactly k points that minimizes the sum $\sum_{p,q \in P} d(p, q)$, where $d(\cdot, \cdot)$ denotes the Euclidean distance. Give an $O(n \log n)$-time algorithm for this problem.

Solution: We generalize the divide-and-conquer algorithm for finding the closest pair of points. That's simply the flash mob problem for the special case of $k = 2$. The key idea is

to show that in the "merge" step where we consider solutions that involve points on both sides of the split line, for each such point we only need to consider a constant number of points above it. In the case of $k = 2$, we showed that it suffices to consider 7 points above our current point. In the general case, it will be some number that depends on k. But, because k is some fixed constant, this number of points will be a constant too, and that's all that we need!

First, we'll use the term *flash mob distance* for a set of k points to be the sum of the distances between each pair of those k points. That's what we seek to minimize.

As before, we divide the points into the left and right halves and use recursion to find the closest flash mob distance on the left or right. Call that distance d. Now, to find the best flash mob distance for "crossing" points that are not all on one side of the midpoint line, we consider the "band" of all points within a horizontal distance of d on either side of the line. No point outside that band can be part of a flash mob that crosses the line and has a flash mob distance of less than d.

As before, we scan the list of points sorted from lowest to highest to collect the set of points, lowest to highest, within the band. We wish to show that, when we consider a point p in that band, we need only consider a constant number of points above it, which amounts to a constant amount of work to check for a set with a smaller flash-mob distance. Clearly, we need not consider any point more than distance d above it. So, as before, any potentially relevant points are in a $2d \times d$ box, with our current point on the bottom boundary of the box.

In the case $k = 2$, we divided the $2d \times d$ box into 8 "sub-boxes," and showed that there can be at most one point in each of them. Here, we'll divide it into a different number of sub-boxes. Let's define the size of each sub-box as $\ell \times \ell$, and then determine what ℓ should be. If there were k points in a sub-box, their flash mob distance would be less than $\binom{k}{2} \cdot 2\ell = k(k-1)\ell$, because the maximum distance between any two points in that box is less than 2ℓ. We want to choose ℓ so that this total distance is less than d; this implies that there cannot be k (or more) points in that box. For if there were, it would contradict the assumption that the closest flash mob on the left or right of the dividing line is d.

Since $k(k-1)\ell$ is a strict upper-bound on the flash mob distance of k points in an $\ell \times \ell$ box, it suffices to choose ℓ so that $k(k-1)\ell = d$, i.e., $\ell = d/(k(k-1))$. Next, there are at most $(2d/\ell) \cdot (d/\ell) = 2k^2(k-1)^2$ sub-boxes in any $2d \times d$ box. Each of those sub-boxes contains fewer than k points, so the total number of points that can be in any $2d \times d$ box is at most $2k^2(k-1)^3$, which is a constant that only depends on k.

Therefore, our recurrence relation for this divide-and-conquer algorithm is $T(n) = 2T(n/2) + O(n)$ and $T(1) = O(1)$, which we have seen solves to $O(n \log n)$.