CS 577: Introduction to Algorithms — Homework 2 Solutions

For this assignment, we did not expect student to give formal proofs, unless the question clearly said so. However, in these solutions, we are giving both an informal argument as well as a complete formal proof. As a result, the solutions are longer than necessary.

Part 1: Both sequences a and b are concave.

(a)

To understand intuitively how concave sequences look like we give some examples of concave sequences in Figure 1 and some examples of sequences that are not concave in Figure 2.

Figure 1: Three concave sequences.

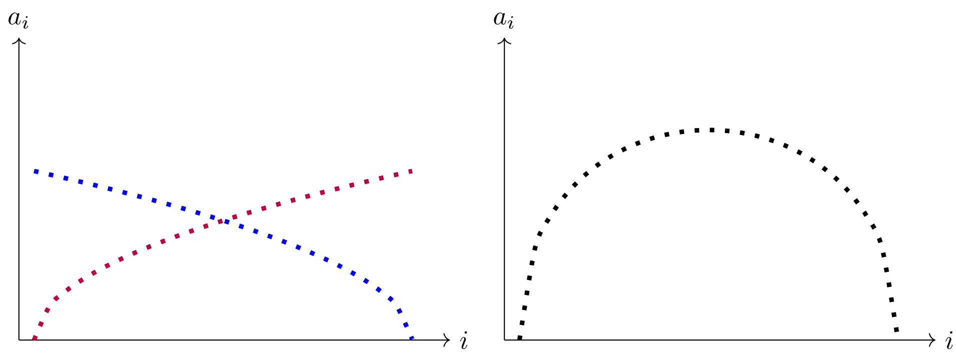
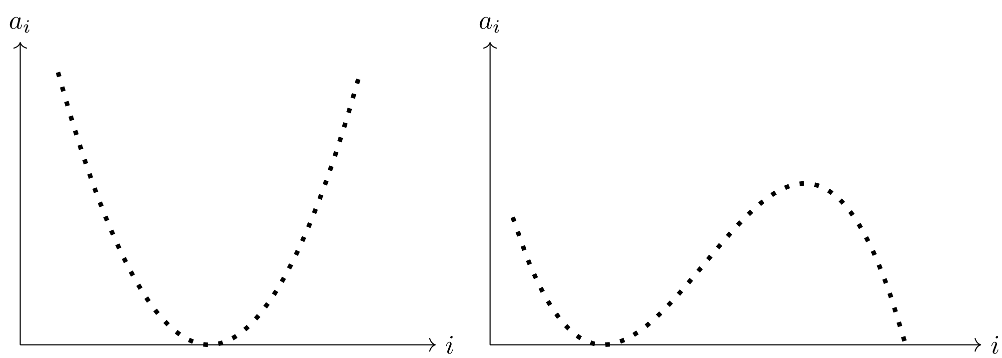


Figure 2: Two sequences that are not concave.



If both a and b are concave, we have

$$a_i - a_{i-1} \geq a_{i+1} - a_i \tag{1}$$
$$b_{k-i} - b_{k-i+1} \geq b_{k-i-1} - b_{k-i} \tag{2}$$

Adding (1) and (2) we get

$$a_i - a_{i-1} + b_{k-i} - b_{k-i+1} \geq a_{i+1} - a_i + b_{k-i-1} - b_{k-i}$$
$$(a_i + b_{k-i}) - (a_{i-1} + b_{k-i+1}) \geq (a_{i+1} + b_{i+1}) - (a_i + k - i - 1)$$
$$y_i - y_{i-1} \geq y_{i+1} - y_i$$

for $1 \leq i < n$. Thus, the sequence y_i is also concave as a function of i.

(b)

1. **High Level Idea.**
   Computing v_k corresponds to finding the maximum of the sequence $y_i = a_i + b_{k-i}$ for $1 \leq i < k$. To find the maximum element of a sequence *in general* one needs to check all its elements (even if you read $n-1$ elements you still need to check whether the last one is the maximum). This lower bound holds when the sequence is arbitrary. Our sequence has *structure* which we can exploit to get an efficient search algorithm. We proved in the previous question that y_i is a concave sequence. How does a concave sequence look like? It can be increasing

   $$0,\ 2,\ 4,\ 8,\ 10,\ 12,\ 14,\ 16,\ 18,\ 20$$

   or decreasing

   $$10,\ 9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1$$

   or first increasing and then decreasing

   $$0,\ 2,\ 4,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0$$

   Can it be first decreasing and then increasing?

   $$7, 4, 2, 3, 5$$

   The answer is no. In this case the minimum element of our sequence has greater elements both on its left and its right. Therefore, the concavity constraint is violated at the minimum.

   Once a concave sequence starts decreasing it can never increase again. Suppose that we check element y_i: if y_{i+1} is smaller than y_i we are then sure that all elements right of y_i will also be smaller and therefore we do not have to check them. It only makes sense to continue our search for the maximum in the elements left of y_i. On the other hand if we find out that y_{i+1} is larger than y_i then we know that we are still in the increasing part of our sequence and therefore we need to check the elements right of y_i. This nice property suggests that we solve this problem recursively (divide and conquer). It remains to find a way to choose the point y_i. Since at each round we will be throwing away either the elements left of y_i or those on its right it makes sense to look at the middle point so that we throw away exactly half of the elements each time (exactly like binary search).

   The algorithm FINDMAX uses these ideas to find the maximum element in a concave sequence. To compute v_k we call FINDMAX$([0, ..., k])$.

Algorithm 1 FINDMAX$(i_\ell, i_h))$

Input: i_ℓ, i_h, a, b
Output: The maximum value of the concave sequence y_i for $i \in \{i_\ell, \dots, i_h\}$
if $i_h - i_\ell \leq 2$ **then**
  Check *all* elements of the list and return the maximum.
$i = (i_h + i_\ell)/2$
if $y_i \geq y_{i-1}$ *and* $y_i \geq y_{i+1}$ **then**
  return y_i
else if $y_{i-1} < y_{i+1}$ **then**
  return FINDMAX$(i+1, i_h)$
else if $y_{i-1} > y_{i+1}$ **then**
  return FINDMAX$(i_\ell, i-1)$

---

2. **Running Time.**
   The running time is: $T(k) = T(k/2) + O(1) = O(\log k)$

3. **Correctness.**

Proof. We have to prove that $\textsc{FindMax}(0, k)$ correctly return v_k.

- Base Case: When $k \leq 2$, $i_h - i_\ell \leq 2$ thus the algorithm run through each value then return the maximum value.
- Inductive Hypothesis: Assume that the algorithm return correctly when $i_h - i_\ell < n$
- Inductive Step: We have to prove that the algorithm also return correctly when $i_{max} - i_{min} = n$.
  - $y_i \geq y_{i-1}$ and $y_i \geq y_{i+1}$: This implies that y_i is a local maxima on the sequence z. However, because the sequence $y_i = a_i + b_{k-i}$ is also concave as a function of i, y_i is also the global maxima of the sequence. Thus, the algorithm return correctly.
  - $y_{i-1} < y_{i+1}$: This implies that the sequence z is increasing at p i or the maximal value is on the right half of p i. Thus, it recursively calls the function for the right half of the remaining p which returns correctly (Inductive Hypothesis).
  - $y_{i-1} > y_{i+1}$: Similar to the prior case, we recursively call the function to the left.

□

Part 2: Only sequence b is concave while a is not.

Let's first argue that computing the value for a single k must take $\Omega(k)$ time. Let's assume that we have an algorithm $\mathcal{A}$ that takes as input the sequences a and b and computes v_k. Now consider an arbitrary sequence a_i. Can we use the algorithm $\mathcal{A}$ to find the maximum element of a_i? Yes, since we can set all elements of the sequence b to be 0. Then the constant sequence b is concave and $b_i + a_{k-i} = a_{k-i}$. Therefore, if we call $\mathcal{A}$ with input lists $b = (0, \ldots, 0)$ and a it will simply return the maximum element of a. We showed that $\mathcal{A}$ solves the problem of finding the maximum element in an arbitrary sequence. Does this imply anything about the running time of $\mathcal{A}$? We already discussed in Part 1 that finding the maximum element in an arbitrary sequence of length k requires $O(k)$ time since we have to check all its elements. Therefore, the running time of any such algorithm $\mathcal{A}$ is $\Omega(k)$.

(a)

We will first introduce some convenient notation for this problem. Let $y_i^k = a_i + b_{k-i}$, that is y_i^k is the *total* value we obtain when when we use i pounds on pixie dust and $k - i$ pounds on dragon scales. Recall that in this problem we defined $i(k)$ to be the *largest* index i that maximizes the sequence y_i^k. We want to prove that the sequence $i(k)$ increases with k.

To prove our claim we first need to understand how the sequence y_i^k behaves as k increases. It makes sense to look at difference $y_i^{k+1} - y_i^k$; we keep the amount of pounds that we use on pixie dust constant $(= i)$ and use $k + 1 - i$ pounds instead of $k - i$ pounds on dragon scales. Since we spend the same amount on pixie dust, using this difference we just check whether spending more on dragon scales increases the total value. Indeed we have

$$y_i^{k+1} - y_i^k = a_i + b_{k+1-i} - a_i - b_{k-i} = b_{k+1-i} - b_{k-i}.$$

Now to prove that $i(k+1)$ is greater than $i(k)$, it suffices to show that in going from k to $k+1$, z increases more at index $i(k)$ than at any index smaller than $i(k)$. Then, y^{k+1} is still larger at $i(k)$ than at any index smaller than $i(k)$. Formally, we will show that for any $j < i(k)$,

$$y_j^{k+1} - y_j^k \leq y_{i(k)}^{k+1} - y_{i(k)}^k. \tag{3}$$

This will imply

$$y_j^{k+1} \leq y_{i(k)}^{k+1} - (y_{i(k)}^k - y_j^k) \leq y_{i(k)}^{k+1},$$

where for the second inequality we used the fact that $y_{i(k)}^k - y_j^k \geq 0$ since y^k is maximum at $i(k)$. We rewrite inequality (3) that we want to prove

$$
\begin{aligned}
y_j^{k+1} - y_j^k &\leq y_{i(k)}^{k+1} - y_{i(k)}^k \\
a_j + b_{k+1-j} - (a_j + b_{k-j}) &\leq a_{i(k)} + b_{k+1-i(k)} - (a_{i(k)} + b_{k-i(k)}) \\
b_{k+1-j} - b_{k-j} &\leq b_{k+1-i(k)} - b_{k-i(k)} \qquad (4)
\end{aligned}
$$

Now this last inequality resembles the definition of a concave sequence. The only problem is that the definition has adjacent indices: $b_{i+1} - b_i \leq b_i - b_{i-1}$. Let $\delta(i) \coloneqq b_{i+1} - b_i$. From the definition of concavity we see that $\delta(i)$ is a decreasing function [1] of i. Therefore if $\ell \geq j$ we have

$$b_{\ell+1} - b_\ell \leq b_i - b_{i-1}. \qquad (5)$$

Now notice that in inequality (4) we have that $i(k) > j$ and therefore $k - i(k) < j - i(k)$. Now by equation (5) we conclude that

$$b_{k+1-j} - b_{k-j} \leq b_{k+1-i(k)} - b_{k-i(k)}.$$

Therefore, inequality (3) is true and the proof is complete.

(b)

1. **High Level Idea.**
   Using the fact that $i(k)$ is non-decreasing as a function of k, we will reduce the size of the problem by dividing it into smaller subproblems. Sometimes one can guess the general idea of an algorithm by its time complexity. It is very useful to be able to roughly guess what an algorithm does just by looking at its time complexity. In this case the required bound on running time suggests that we should try a merge sort type recursive structure. The high level idea is to compute v_k (and therefore also $i(k)$) for some value of $k \in \{0, \ldots, n\}$ and partition the computation of the rest of the v_j's into two subproblems, one that computes $v_1, \ldots, v_{k-1}$ and one that computes $v_{k+1}, \ldots, v_n$. Importantly, we want the inputs to these subproblems to be smaller than the input of the original problem. We now examine which parts of the initial lists a, b are relevant to these subproblems. In what follows we use the subscripts h and ℓ as abbreviations of the words high and low. We start with two subproblems, that we solve recursively:

   - P_h: Compute v_j for $j = k+1, \ldots, n$.
     Using the monotonicity of $i(k)$ we get that to compute these v_j we only need the lists $A_h = [a_{i(k)}, \ldots, a_n]$ and $B_h = [b_0, \ldots, b_{n-i(k)}]$.
   - P_ℓ: Compute v_j for $j = 0, \ldots, k-1$.
     Again from the monotonicity of $i(k)$ we get that to compute these v_j we only need $A_\ell = [a_0, \ldots, a_{i(k)}]$ and $B_\ell = [b_0, \ldots, b_{k-1}]$.

   Observe that to compute v_k each element of list a gets *paired* with exactly one element of list b. Therefore, naively computing any v_j of subproblem P_h requires $O(\text{length}(A_h))$ time, while computing any v_j of subproblem P_ℓ requires $O(\text{length}(A_\ell))$ time. The lengths of the lists B_ℓ, B_h do not affect the running time of the subproblems, and therefore we can ignore them.

   Algorithm FINDALL uses the above ideas to solve this problem. To find all the v_k for $k = 1, \ldots, n$ we call FINDALL$(0, n, 0, n)$. To simplify notation we do not include the lists a, b in the arguments of algorithm FINDALL. Therefore, we can express the running time informally as

   $$T(n) = T(\text{length}(A_\ell)) + T(\text{length}(A_h)) + O(n)$$

   At this point we note that it is not necessarily the case that $\text{length}(A_\ell) = \text{length}(A_h) = n/2$, even if we choose $k = n/2$. However, we have that $\text{length}(A_\ell) + \text{length}(A_h) = n + 1$, which suggests that we are doing linear amount of work at every level of recursion. In order to bound the number of levels of recursion we choose $k = n/2$ (that is we halve the range of v_j to be computed at each round).

[1] The (discrete) derivative of a concave function is a decreasing function.

2. **Correctness.**
As we discussed above, since we decide which parts of the list a to keep for each subproblem using the monotonicity of $i(k)$, the correctness of algorithm FINDALL follows from Question 4.

3. **Running Time.**
To upper bound the running time of Algorithm FINDALL, notice that we have $\log_2(n)$ levels of recursion. At any level, the total amount of work done is $O(1)$ times the sum of the lengths of the sublists of list a in recursive calls at that level. We would like to have that this sum of the lengths is $O(n)$ for all levels of recursion. We already saw that for the first two subproblems we have

$$\text{length}(A_\ell) + \text{length}(A_h) = n + 2.$$

It is not exactly $n + 1$ because $a_{i(k)}$ appears in both A_ℓ and A_h. Observe that at each level the two sublists of a that are created can share *at most one* element [2]. At the i-th level of recursion of Algorithm FINDALL we have 2^i subproblems. Therefore the number of repetitions is upper bounded by $2^i - 1$ (this is the case where all but the "lowest" and "highest" subproblems share one element). Therefore the sum of the lengths of the sublists of all 2^i subproblems is upper bounded by

$$\text{length}(a) + (2^i - 1) \leq 2(n + 1),$$

since the maximum number of subproblems is n (at level $i = \log n$). Since we do $O(n)$ work $\log n$ times we conclude that the overall running time is $O(n \log n)$.

Algorithm 2 FINDALL$(p_\ell, p_h, k_\ell, k_h)$

Input: $p_\ell, p_h, k_\ell, k_h, a, b$
Output: The entire sequence $v_k = \max_{i \in \{p_\ell, \ldots, p_h\}} a_i + b_{k-i}$ for all $k \in \{k_\ell, \ldots, k_h\}$
if $k_h = k_\ell$ **then**
  return $v_k = \max_{i \in \{p_\ell, \ldots, p_h\}} a_i + b_{k-i}$ (Manually search for the maximum value in $p_h - p_\ell$ time).
$k = (k_h + k_\ell)/2$
Manually find the largest index $i(k)$ that maximizes the sum $a_i + b_{k-i}$ for all $i \in \{p_\ell, ..., p_h\}$
return [FINDALL$(p_\ell, i(k), k_\ell, k - 1)$, $a_{i(k)} + b_{k-i(k)}$, FINDALL $(i(k), p_h, k + 1, k_h)$]

---

[2] this is because $i(k)$ is just increasing and not strictly increasing