

CSE 101: Homework 4 Solutions

1. Exercise 2.23

- (a) If we split A into two halves $A_L = A[1 \dots \text{floor}(n/2)]$ and $A_R = [\text{ceil}(n/2) \dots n]$, we know that if a majority element exists then it must appear more than $n/4$ times in either A_L, A_R . We can recurse as follows:

Majority($A[1 \dots n]$):

1. if n is 1, return $A[1]$
2. Let A_L, A_R be the first and second halves of A . $M_L = \text{Majority}(A_L)$, $M_R = \text{Majority}(A_R)$.
3. If neither half has a majority return “no majority”. Otherwise, Check whether M_L or M_R is a majority element of A (linear time operation). If so, return that element, else return “no majority.”

Let’s first analyze the running time. We split the array in half and recurse on both halves and in step 3 we do an $O(n)$ operation twice giving $T(n) = 2T(n/2) + O(n)$. By the master theorem this is $O(n \log n)$.

The correctness of this algorithm simply follows from the fact that if a majority exists in the array, it must exist in either the left or right halves as well. We then collect these majorities and check the entire array to verify that they are in fact majorities.

- (b) Let’s implement the hint into an algorithm and then analyze it.

Discard($A[1 \dots n]$):

1. Let T be the empty list.
2. For $i=1$ to $n/2$: if $A[2i-1] = A[2i]$: add $A[2i]$ to T .
3. return Discard(T)

Majority($A[1 \dots n]$):

1. $x = \text{Discard}(A)$
2. If x is a majority element of A , return x . Otherwise return “no majority”.

Let’s first analyze the running time of this algorithm. Each Discard(A) call removes at least half of the elements so at worst takes

$T(n) = T(n/2) + O(n)$ which by the master's theorem gives $T(n) = O(n)$. Step 2. of majority is an $O(n)$ operation as well giving a total running time of $O(n)$.

Let's now argue that this algorithm is correct. We want to prove the following claim about Discard: If x is a majority element of A , then it is also a majority element of T . There are two types of pairings possible: 1) Two elements are different, 2) Two element are the same. Since x can have at most half its elements falling into type 1), x must have more than half that are added to T from type 2 pairings. We only keep type 2) pairings, so the majority property is kept.

2. We did this problem in discussion. Let $T(j)$ be the value of the contiguous subsequence (CSS) of maximum sum that ends exactly at a_j or 0 if no positive sum CSS ends at a_j . Then we can express $T(j) = \max\{0, a_j + T(j-1)\}$. This gives the following simple algorithm:

CSS($a_1, \dots, a_n$):

- (a) Set $T(0) = 0$.
- (b) For $i = 1$ to n , do $T(i) = \max\{0, a_i + T(i-1)\}$
- (c) return $\max_i T(i)$.

This returns the value of maximum sum CSS. The running time is $O(n)$.

We can prove correctness by induction. The claim is that $T(j)$ as defined by the max expression is exactly the larger of the value of the CSS the ends at a_j or 0. The base case is that $T(0) = 0$, which fits the claim. The induction step is assume that $T(j)$ fits the claim for $j < k$, now we would like to show that $T(k)$ fits the claim. If a maximum sum CSS is going to end at a_k it must include a_k and the largest CSS that went up to a_{k-1} . This is exactly the max expression.

3. Let $x[1..n]$ and $y[1..m]$ be the two strings. Imagine truncating each to $x[1..i]$ and $y[1..j]$, then asking how long of a substring at the suffix of each string can we match? Define $LCS(i,j)$ to be the length of the longest matching suffix between $x[1..i]$ and $y[1..j]$. The value of $LCS(i,j)$ hinges on whether $x[i] == y[j]$, if not then there is no suffix substring that can match exactly so $LCS(i,j) = 0$. If they do match, then $LCS(i,j) = 1 + LCS(i-1, j-1)$. Here is the algorithm:

LCS($x[1..n], y[1..m]$):

- (a) Let $LCS(i,0) = LCS(0,i) = 0$ for all $i = 1, \dots, n$.
- (b) For $i = 1$ to n , For $j = 1$ to m do:
 - i. If $x[i]$ equals $y[j]$, then $LCS(i,j) = 1 + LCS(i-1, j-1)$.
 - ii. Else $LCS(i,j) = 0$.

(c) return $\max_{i,j} LCS(i, j)$.

This algorithm's runtime is dominated by the nested for loop and takes $O(mn)$ time.

The correctness hinges on the fact that $LCS(i, j)$ is in fact the length of the longest common suffix between $x[1...i]$ and $y[1...j]$. It's clear that the longest common suffix should be $1 + LCS(i-1, j-1)$ if the ends, $x[i]$ and $y[j]$ are the same, and 0 if they are not. The longest common substring is one of the suffixes, which is returned in final step.

4. Let the sum of all the elements in the list be called $M = \sum_{i=1}^n a_i$. We are looking for an algorithm that runs in polynomial time in n and M . We will solve this problem by considering a slightly more general version. Let $SS(i, A, B)$ be true if and only if there is a way to partition $a_1, \dots, a_i$ into 3 sets I, J, K where the sum of all elements in I equals A , and the sum of all elements in J equals B , and the sum of all the elements in K equals $M_i - A - B$ where $M_i = \sum_{k=1}^i a_k$. This is going to be the recurrence we solve for, so be sure to read it again and understand it before moving on. Also note that the answer to the problem's question will be contained in the truth value of $SS(n, M/3, M/3)$.

The base cases that we can fill right away are $SS(0, 0, 0) = True$, and $SS(0, A, B) = False$ for all $A = 1, \dots, M$ and $B = 1, \dots, M$.

Given that we know $SS(i-1, A, B)$ for all $A = 1, \dots, M$ and $B = 1, \dots, M$, we can solve for any $SS(i, A, B)$ as follows: $SS(i, A, B) = SS(i-1, A - a_i, B)$ OR $SS(i-1, A, B - a_i)$ OR $SS(i-1, A, B)$. In other words, SS with $i-1$ contains all the different A, B values we can achieve by partitioning $a_1, \dots, a_{i-1}$ into 3 groups, so to calculate SS with i we must take one of those true $i-1$ partitions and add a_i to one of them. This is the correctness argument for the algorithm that follows.

$SSEqual(a_1, \dots, a_n)$:

- (a) Let $M = \sum_{i=1}^n a_i$. If 3 does not divide M return false. Set $SS(0, 0, 0) = True$ and $SS(0, A, B) = False$ for all $A = 1, \dots, M$ and $B = 1, \dots, M$.
- (b) For $i = 1$ to n do: For $j = 1$ to M do: For $k = 1$ to M do: $SS(i, j, k) = SS(i-1, j - a_i, k)$ OR $SS(i-1, j, k - a_i)$ OR $SS(i-1, j, k)$ (Note: that by convention any negative index put into SS will return False.)
- (c) return $SS(n, M/3, M/3)$.

The running time of this algorithm is dominated by the triple nested for loop which takes $O(nM^2)$ time.

5. The max flow is 13, and the corresponding cut is $L = \{S, C, F\}$ vs $R = \{T, A, B, D, E, G\}$. The amount of flow from L to R is $6 + 1 + 2 + 4 = 13$.