CSE 202: Design and Analysis of Algorithms **(Due: 10/05/19)**

Homework #0

Instructor: Ramamohan Paturi *Name:* Shihan Ran, *Netid:* A53313589

Problem 1: Maximum sum among nonadjacent subsequences

Problem Description

Find an efficient algorithm for the following problem:

We are given an array of real numbers $V[1 \dots n]$. We wish to find a subset of array positions, $S \subseteq [1 \dots n]$ that maximizes $\sum_{i \in S} V[i]$ subject to no two consecutive array positions being in S. For example, say $V = [10, 14, 12, 6, 13, 4]$, the best solution is to take elements $1, 3, 5$ to get a total of $10 + 12 + 13 = 35$. If instead, we try to take the 14 in position 2, we must exclude the 10 and 12 in positions 1 and 3, leaving us with the second best choice 2, 5 giving a total of $14 + 13 = 27$.

Solution

(High-level description)

For $S \subseteq [1 \dots n]$ and $i \in S$, what we care is whether we choose $V[i]$ or not. For illustration, let's regard the maximum sum of current position as $dp[i]$.

If we choose to take $V[i]$, then following the rule that *no two consecutive array positions can be chosen*, we can only get a maximum sum $V[i] + dp[i-2]$. If we choose not to take $V[i]$, then the maximum sum is $dp[i-1]$.

Beacuse we always want to get a maximum sum value, thus we can arrive at a equation that:

$$dp[i] = \max(V[i] + dp[i-2], dp[i-1])$$

After going over n element of V, the final $dp[n]$ can be the maximum sum among nonadjacent subsequences.

Now, let's pay a few attention to the equation again. As we only use $dp[i-2]$ and $dp[i-1]$ when we calculate $dp[i]$ each time, actually we do not have to record the whole $dp[1 \dots n]$. Instead, we only need two variables, say a and b, to record the exact value of $dp[i-2]$ and $dp[i-1]$ at current position, and use a temp to record the current value $dp[i]$. That could save lots of space when n is extramely large.

Another thing we need to pay attention to is: the initial value of a and b. As we noticed, when calculating $temp = \max(V[1] + a, b)$ at position 1, a and b should have an initial value of 0.

(Pseudo Code)

Algorithm 1: Maximum sum among nonadjacent subsequences

Input: $V[1 \dots n]$
Output: b as maximum sum

```
if n = 0 then
    return 0;
let a, b = 0;
for i = 1; i ≤ n; i++ do
    const temp = b;
    b = max(a + V[i], b);
    a = temp
return b;
```

(Correctness)

As defined above, our recurrence is $dp[i] = \max(V[i] + dp[i-2], dp[i-1])$.

To prove this is correct, consider the optimal solution for $dp[i]$. There are two cases: either $V[i]$ is chosen in the solution for $dp[i]$, or it is not.

Case 1: If $V[i]$ is not chosen, then the maximum sum among nonadjacent subsequences of length i is just the same as $i - 1$; by definition, this is $dp[i]$.

Case 2: If $V[i]$ is chosen, then the remaining elements selected for $dp[i]$ should be drawn from 1 through $i-2$, that is $dp[i-2]$. Therefore, $dp[i] = V[i] + dp[i-2]$

Finally, since $dp[i]$ is a maximization, the larger of these two cases is $dp[i]$.

As base cases, we define $dp[0]$ and $dp[1] = 0$.

(Time complexity)

The operation in the *For* loop can be completed in $O(1)$ time, thus in $O(n)$ iterations the algorithm results in a maximum sum.

(Data Structure)

Array is enough. We are able to find the $V[i]$ in $O(1)$ time.

Problem 2: Maximum difference in an array

Problem Description

Given an array A of integers of length n, find the maximum value of $A(j) - A(i)$ over all choices of indexes such that $j > i$.

Solution

(High-level description)

At first glance, I thought we could use a sort algorithm and gets $max - min$ as result. But wait, the problem description says *the indexes should be* $j > i$, which means *Order matters.*

What we really need to do is to record the minimum value min and $maxdifference$ we've ever seen so far. When at a new position i, we compare current value $A(i)$ with min, if $A(i) < min$, then swatch their values; if $A(i) > min$, then compare $A(i) - min$ with $maxdifference$.

After going over n element of A, the final $maxdifference$ can be the maximum difference in an array.

(Pseudo Code)

Algorithm 2: Maximum difference in an array

Input: $A[1 \dots n]$
Output: $maxdifference$ as maximum difference

```
1 let min = A[1];
2 let maxdifference = 0;
3 for i = 2; i ≤ n; i++ do
4     if A[i] < min then
5         min = A[i];
6     if maxdifference < A[i] − min then
7         maxdifference = A[i] − min;
8 return maxdifference;
```

(Correctness)

The instructor answered on Pizza that we may assume that $n > 1$.

The base case is $n = 2$ and there are three cases: $A[1] < A[2]$, $A[1] = A[2]$ and $A[1] > A[2]$. As the algorithm goes, we first initialize $min = A[1], maxdifference = 0$.

Case 1: $A[1] < A[2]$. Then we'll skip the first *if* in the *for* loop and then return $maxdifference = A[2] - A[1]$.

Case 2: $A[1] = A[2]$. Then we'll skip the two *if* in the *for* loop and return $maxdifference = 0$.

Case 3: $A[1] > A[2]$. Then we'll update $min = A[2]$ and skip the second *if* in the *for* loop. We return 0 as result.

All three cases are correct for base case.

If the case is correct for n. Then for $n + 1$, there are three cases: $A[n + 1] < min$, $A[n + 1] = min$ and $A[n + 1] > min$.

Case 1: $A[n + 1] < min$. Then we'll update $min = A[n + 1]$. We only have $n + 1$ elements, so there won't be a $s > n + 1$, we get the maximum value of $A(s) - A(n + 1)$. Thus, we return $maxdifference$ of $A[1...n]$.

Case 2: $A[n + 1] = min$. Same as Case 1, we still return $maxdifference$ of $A[1...n]$.

Case 3: $A[n + 1] > min$. If $maxdifference < A[n + 1] - min$, then we update $maxdifference$ and return it. Else, we return $maxdifference$ of $A[1...n]$.

All three cases are correct for n and $n + 1$. By induction, this algorithm is corrrect.

(Time complexity)

The operation in the *For* loop can be completed in $O(1)$ time, thus in $O(n)$ iterations the algorithm results in a maximum difference.

(Data Structure)

Array is enough. We are able to find the $A[i]$ in $O(1)$ time.

Problem 3: Maximum difference in a matrix

Problem Description

Given an $n \times n$ matrix $M[i, j]$ of integers, find the maximum value of $M[c, d] - M[a, b]$ over all choices of indexes such that both $c > a$ and $d > b$.

Solution

(High-level description)

Basically Problem 3 shares the same idea with Problem 2: *to record the minimum value min and* $maxdifference$ *we've ever seen so far*. The differences is that Problem 2 is one dimensional while Problem 3 is two dimensional.

When at a new position (c, d), since the problem description says *the indexes such that both* $c > a$ *and* $d > b$, so we can only choose the minimum value from (a, b) with the restriction that $1 \leq a \leq c-1, 1 \leq b \leq d-1$. Compare $maxdifference$ with $M[c, d] - min$, and update the $maxdifference$ if $M[c, d] - min > maxdifference$.

The trickiest thing is how to find the minimum value from (a, b) efficiently. For (c, d), we have the constriction that $1 \leq a \leq c-1, 1 \leq b \leq d-1$. Let's assume the minimum of $M[1 \dots c-1, 1 \dots d-1]$ is min. Now let's look at $(c, d+1)$, the constriction is $1 \leq a \leq c-1, 1 \leq b \leq d$, which means we should find the minimum of $M[1 \dots c-1, 1 \dots d]$. The interesting part of this problem is: if we already know the minimum of $M[1 \dots c-1, 1 \dots d-1] = min$, to compute $M[1 \dots c-1, 1 \dots d]$, we only need to compare the value of min and the minimum of $M[1 \dots c-1, d]$. The principle of avoid common redundancy will save us lots of time.

We maintain a list $T[1 \dots n]$ to record the minimum of $M[1 \dots c-1, 1 \dots d-1]$. As we iterate by rows, we update it.

(Pseudo Code)

Algorithm 3: Maximum difference in a matrix

Input: $M[1 \dots n, 1 \dots n]$
Output: $maxdifference$ as maximum difference

```
1  initial T[1...n] = ∞ ;
2  let maxdifference = 0;
3  for i = 1; i ≤ n − 1; i++ do
4  |   // Update T ;
5  |   T[1] = min(T[1], M[i, 1]);
6  |   for j = 2; j ≤ n; j++ do
7  |   |   T[j] = min(T[j], T[j − 1], M[i + 1, j]) ;
8  |   // Compare the min with M[c, d] ;
9  |   for j = 2; j ≤ n; j++ do
10 |   |   maxdifference = max(maxdifference, M[i, j] − T[j − 1]) ;
11 return maxdifference;
```

(Correctness)

Loop Invariant:

For the *i-th* outer loop, $T[j]$ is the smallest element in $M[p, q]$ where $1 \leq p \leq i$ and $1 \leq q \leq j$

Induction:

1. Initialization:
    - When $n = 2$, firstly, we update $T[1]$ as the smaller one in $\{\infty\ ,\ M[1, 1]\ \}$, so $T[1]$ satisfies that $T[j]$ is the smallest element in $M[p, q]$ where $1 \leq p \leq 1$ and $1 \leq q \leq 1$ $(i = j = 1)$.

- Then when j increasing from 2, we update $T[j] = \min(T[j], T[j-1], M[i,j])$, where $T[j]$ before updating is ∞, $T[j-1]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq i$ and $1 \leq q \leq j-1$. So after updating, $T[j]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq i$ and $1 \leq q \leq j$.
- Therefore, $T[j]$ is the smallest element in $M[p,q]$ where $p = i = 1$ and $1 \leq q \leq j$

2. Induction:

- Assume that For the *k-th* outer loop, $T[j]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq k$ and $1 \leq q \leq j$.
- So for $k+1$, before updating, $T[1]$ is the smallest element in $M[p,1]$ where $1 \leq p \leq k$. When using $T[1] = \min(T[1], M[i,1])$ in the $k+1$ outer loop, we get $T[1]$ is the smallest in $M[p,1]$ where $1 \leq p \leq k+1$.
- Then, when we update $T[j] = \min(T[j], T[j-1], M[i,j])$, we know $T[j-1]$ is the smallest in $M[p,q]$ where $1 \leq p \leq k+1,\ 1 \leq q \leq j-1$. And $T[j]$ before updating is the smallest in $M[p,q]$ where $1 \leq p \leq k,\ 1 \leq q \leq j$. So after updating with $M[k+1,j]$, it's easy to obtain that $T[j]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq k+1$ and $1 \leq q \leq j$.
- Therefore, $T[j]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq i$ and $1 \leq q \leq j$.

Correctness when terminate

- Because when $n \geq i$, this loop will terminate, and $T[j]$ is the smallest element in $M[p,q]$ where $1 \leq p \leq n$ and $1 \leq q \leq j$.
- And we select the maximal difference from the maximal among $M[i,j] - T[j-1]$ in every loop, so we will return maximal difference between $M[i,j]$ and $M[p,q]$ where $p < i$ and $q < j$. Therefore, our algorithm is correct.

(Time complexity)

$O(n^2)$. Since there are both outer loop of n size and inner loop of n size.

(Data Structure)

Two dimensional Array for M, and one dimensional array for T. We are able to find the element in $O(1)$ time.

Problem 4: Pond sizes

Problem Description

You have an integer matrix representing a plot of land, where the value at a location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of a pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

Solution

(High-level description)

Although the problem description says it's an integer matrix, we can regard it as a connected graph. The connectivity depends on vertically, horizontally and diagonally adjacent.

Our high-level idea is: We start with a position (i, j), if it has a value of zero, then it indicates water. We first set $size = 1$, then we would like to find if the adjacent region is filled with water too, which means we should check the 8 adjacent positions $[(i+1, j), (i-1, j), (i, j+1), (i, j-1), (i-1, j-1), (i-1, j+1), (i+1, j+1), (i+1, j-1)]$. Notice, we should check if the position is at the edge so that index will not have access violation.

If the value of adjacent positions is also zero, we should increase the sizes of pond: $size + 1$. Also, to avoid double counting, after we visited each position, we should use a mark so that we will not visit it again in the future. For this problem, we can simple set $M[i, j] = 1$.

(Pseudo Code)

Algorithm 4: Pond Size

Input: $M[1 \dots m, 1 \dots n]$
Output: A list containing the sizes of all ponds

```
1 initial res as an empty list ;
2 for i = 1; i ≤ m; i++ do
3     for j = 1; j ≤ n; j++ do
4         if M[i, j] is 0 then
5             size = 1 ;
6             checkAdjacentPositions(M, i, j, size) ;
7             add size into res ;
8 return res;
```

(Correctness)

Actually, for each position (i, j), we use DFS to search for all children nodes of it. As depth first search going, we count $size$ and mark the children as visited. This ensures both termination and the completeness.

(Time complexity)

We can easily draw the conclusion that in algorithm *Pond Size*, the outer and inner loop creates $O(mn)$ time complexity, here m is the number of rows and n is the number of columns of M. For operations inside the inner loop, we should notice a fact that there is at most once visit for each position (since we mark $M[i, j] = 1$ for those ponds and will visit it again in the next time).

Overall speaking, the total time complexity for this algorithm is $O(mn)$.

Algorithm 5: *checkAdjacentPositions*

Input: $M[1 \dots m, 1 \dots n], i, j, size$
Output: None but updates size

```
// check access violation;
if i < 1 or j < 1 or i > m or j > n or M[i, j] is not 0 then
    return ;
// M[i, j] is 0;
size = size + 1 ;
M[i, j] = 0;
// check adjacent positions;
// vertically;
checkAdjacentPositions(M, i + 1, j, size) ;
checkAdjacentPositions(M, i − 1, j, size) ;
// horizontally;
checkAdjacentPositions(M, i, j + 1, size) ;
checkAdjacentPositions(M, i, j − 1, size) ;
// diagonally;
checkAdjacentPositions(M, i − 1, j − 1, size) ;
checkAdjacentPositions(M, i − 1, j + 1, size) ;
checkAdjacentPositions(M, i + 1, j + 1, size) ;
checkAdjacentPositions(M, i + 1, j − 1, size) ;
```

Problem 5: Frequent elements

Problem Description

Design an algorithm that, given a list of n elements in an array, finds all the elements that appear more than $n/3$ times in the list. The algorithm should run in linear time. n is a nonnegative integer.

You are expected to use comparisons and achieve linear time. You may not use hashing. Neither can you use excessive space. Linear space is sufficient. Moreover, you are expected to design a deterministic algorithm. You may also not use the standard linear-time deterministic selection algorithm. Your algorithm has to be more efficient (in terms of constant factors) than the standard linear-time deterministic selection algorithm.

Solution

(High-level description)

There can be at most 2 elements that appear more than $n/3$ times. So the size of output will only be $0, 1, 2$. Thus, we maintain two counters to keep record of the most frequent elements. Specifically, we use a dict containing only two elements to realize this goal. When traversing through the array, for element a_i, we have the following situations:

1. if a_i is in the dict and $Count_{a_i} > 0$, then $Count_{a_i} + 1$;
2. if a_i is not in the dict:

   2.1 The dict has less than two elements, add a_i into it, and set $Count_{a_i} = 1$.

   2.2 The dict has two elements x, y, with $Count_x > 0$ and $Count_y > 0$. Then do $Count_x - 1$ and $Count_y - 1$. If after decreasing, we get $Count = 0$. Remove this element from dict.

After traversing through the whole input array, we will get two most frequent elements. Notice! They may not appear more than $n/3$ times. So we need to traverse through the array again and count the appearance of these two elements. Compare the appearance with $n/3$ and then output the result.

(Pseudo Code)

In the next page.

Algorithm 6: Frequent elements

Input: $A[1 \dots n]$
Output: Elements appear more than $n/3$ times in A

initial dict $D = \{\}$;
for $i = 1; i \leq n; i++$ **do**
  if $A[i]$ *in dict* D **then**
    $D[A[i]] = D[A[i]] + 1$ // Update Count;
  else
    if D *has less than two elements* **then**
      Add $A[i]$ in D and set $D[A[i]] = 1$;
    else
      for *element in dict* D **do**
        $D[element] = D[element] - 1$;
        if $D[element]$ *is* 0 **then**
          remove *element* from D;
let x, y is the two elements of D;
let $Count_X, Count_Y = 0$;
for $i = 1; i \leq n; i++$ **do**
  if $A[i]$ *is* x **then**
    $Count_X = Count_X + 1$;
  if $A[i]$ *is* y **then**
    $Count_Y = Count_Y + 1$;
initial *ret* as an empty list ;
if $Count_X > n/3$ **then**
  add x to *ret*;
if $Count_Y > n/3$ **then**
  add y to *ret*;
return *ret*;

(Correctness)

There can be at most 2 elements that appear more than $n/3$ times, otherwise will be over $3 * n/3 = n$ times. So if we want to use voting algorithm, we only need two counters.

We want to prove that the dict stores the most frequent two elements. Say the elements stored in the dict is a, b. The first two different elements will be assigned to a and b, with $Count_a$ and $Count_b$. We increase the $Count$ when we see a and b is reasonable. The reason why we decrease $Count_a$ and $Count_b$ when the current element is neither a nor b is that if you have some frequent elements, even though you decrease the count when you meet different elements, finally the count will still be larger than 0. You can regard this as a offset of different numbers, only the most frequent ones can survive in the end.

But we can not guarantee its appeared over $n/3$ times. That's why we do another counting for frequent number a and b again.

(Time complexity)

The operation in the *For* loop can be completed in $O(1)$ time. Since we traverse through the array, we get a time complexity of $O(n)$.

Also, the space complexity is linear.

(Data Structure)

Dict is more efficient for recording the appearance times since you can asses the value of its element by simply using the key in $O(1)$ time. But, considering this specific problem, we only need to record two elements and its appearance times. Four variable is also enough.