601.433/633 Introduction to Algorithms **Fall 2024**
Homework #5 **Due:** October 15, 2024, 9am

Bella Lu
Partners: Jecia Mao, Luna Liu

---

1 Submatrices (33 points)

Let $A \in \{0,1\}^{n \times m}$ be a matrix with n rows, m columns, and where every entry is either 0 or 1. We will let A_{ij} denote the entry in row i and column j, so for example A_{11} is the top-left entry, A_{n1} is the bottom-left entry, A_{1m} is the top-right entry, and A_{nm} is the bottom-right entry. Suppose that we want to find the largest integer k such that A contains a $k \times k$ contiguous submatrix consisting of all 0's. In other words, we want to find the largest k such there exist values i, j such that $A_{xy} = 0$ for all $i - k < x \leq i$ and $j - k < y \leq j$.

We will design a dynamic programming algorithm that runs in $O(nm)$ time for this problem.

(a) (17 points) For every $i, j \in \mathbb{N}$ with $1 \leq i \leq n$ and $1 \leq j \leq m$, let $S(i, j)$ denote the maximum value of k such that there is a $k \times k$ contiguous submatrix of A consisting of all 0's whose bottom-right corner is at (i, j) (row i, column j). Write a recursive formula for $S(i, j)$, and prove that your formula is correct.

Note: you will need to use this formula in the next part to get an $O(nm)$-time algorithm, so make sure that your formula is not too big/slow.

We can define the following OPT function:

$S(i, j)$ = max square matrix size of 0's such that the bottom right corner is $A_{i,j}$.

$$S(i,j) \begin{cases} 0 & \text{if } A_{ij} = 1, \\ 1 & \text{if } i = 1 \text{ or } j = 1, A_{ij} = 0, \\ min[S(i-1, j-1), S(i, j-1), S(i-1, j)] + 1 & \text{if } i > 1 \text{ and } j > 1, A_{ij} = 0 \end{cases}$$

Induction on indices for correctness

Base case:
A_{1j} for j = 1, ... , m
A_{i1} for i = 1, ... , n

If $A_{1j} = 0 : S(1, j) = 1$
If $A_{i1} = 1 : S(i, 1) = 0$

For the sake of induction, let $1 \leq i' < i$ and $1 \leq j' < j$. We assume that all $S(i', j')$ are correct.

Case 1: If $A_{ij} = 1$, $S(i, j) = 0$.

Case 2: If $A_{ij} = 0$,

1. Prove that $S(i, j) \geq min[S(i-1, j-1), S(i, j-1), S(i-1, j)] + 1$

Without loss of generality, assume that $S(i-1, j-1)$ is the minimum, and equals k. This means that the other two, $S(i, j-1)$ and $S(i-1, j)$, must be at least k. For example, assume that $S(i-1, j-1), S(i, j-1), S(i-1, j)$ all equals k. If A_{ij} has a zero, then there would be a $(k+1)$ by $(k+1)$ matrix such that the bottom right corner is A_{ij}. Thus, $S(i, j)$ must be at least k+1, by geometry. The inequality holds.

2. Prove that $S(i, j) \leq min[S(i-1, j-1), S(i, j-1), S(i-1, j)] + 1$

If we assume that $S(i, j) = k$, then $S(i-1, j-1), S(i, j-1), S(i-1, j)$ must each be at least k-1. This means that the minimum of the three S's plus 1 must be at least k. The inequality holds.

Thus, for $i > 1, j > 1$,
$S(i, j) = 0$ if $A_{ij} = 1$, and
$S(i, j) = min[S(i-1, j-1), S(i, j-1), S(i-1, j)] + 1$ if $A_{ij} = 0$

The OPT function is correct. ■

(b) (16 points) Give a dynamic programming algorithm based on your solution to part (a), and prove that it correctly finds the largest possible value of k and runs in time $O(nm)$.

Algorithm

k = -inf

for i = 1 to n

    for j = 1 to m

        if $A_{ij} = 1$

            $OPT(i, j) = 0$

        else if i = 1 or j = 1

            $OPT(i, j) = 1$

        else

            $OPT(i, j) = min[OPT(i-1, j-1), OPT(i, j-1), OPT(i-1, j)] + 1$

        if $OPT(i, j)$ is greater than k, set k to $OPT(i, j)$

return k

Correctness

This algorithm covers the base case (if i = 1 or j = 1), which is the first row and column. This correctly computes the max square matrix size of 0's at each entry because if it has a value of zero then it is a 1x1 matrix with a zero, then if it has a value of one (covered in first if loop), then there are no matrix of zero there.

Since we have a nested for loop for every element in the matrix, it will cover every element in the table. Excluding the base case, that leaves us with the rest of the table. We have proven in part a that this formula correctly calculates $OPT(i, j)$.

The second if statement keeps track of the largest possible value of k currently, and updates if a larger value comes up. We return k at the end, which is the largest $OPT(i, j)$ calculated. The algorithm is correct. ■

Runtime

Inside the inner for loop, there are two if loops. All if/else statements are constant, and the sections inside the statements are constant. Thus, the running time inside the inner for loop is constant.

There are two for loops, one nested in each other, and each iteration does constant time work, so the run time is $O(nm)$. ■

2 Completing Homeworks (33 points)

Suppose (hypothetically) that you were taking a class, possibly called "Introduction to Algorithms", in which the homeworks were extremely difficult. After enough complaining, the professor decided to make the following changes. There are two homework assignments each week rather than one, an "easy" assignment and a "hard" assignment. The hard assignment is worth more points, but it is in fact so difficult that you can only complete it if you're completely rested and prepared, meaning that you cannot have done either of the assignments the week before.

More formally, let n be the number of weeks in the class, let h_i be the number of points for the hard assignment in week i, and let e_i be the number of points for the easy assignment in week i. Note that h_i does not have to be equal to h_j for $i \neq j$ (although it might be), and similarly with e_i and e_j. Assume that you know all of these values in advance. Then the goal is compute a schedule which in each week tells you whether to do nothing, the easy assignment, or the hard assignment and maximizes the total number of points, subject to the restriction that if you do a hard assignment in week i you cannot have done *any* assignment in week $i - 1$.

(a) (11 points) One obvious approach would be to choose a hard assignment in week i if we get more points than if we completed the easy assignments for weeks i and $i - 1$. This would be the following algorithm:

$i = 1$
while $(i < n)$ {

```
  if (h_{i+1} ≥ e_{i+1} + e_i) {
    choose no assignment in week i,
    choose the hard assignment in week i+1,
    i = i + 2
  }
  else {
    choose the easy assignment in week i,
    i = i + 1
  }
}
```

Give an instance in which this algorithm does not return the optimal solution. Also say what the optimal solution is (and its value) and what the algorithm finds instead.

	Day 1	Day 2	Day 3
Easy	1	2	3
Hard	2	4	9

Optimal solution: $1 + 0 + 9 = 10$
We can do the easy home on day 1, then do the hard homework on day 3, which means we can't do a homework on day 2.

Algorithm solution: $0 + 4 + 3 = 7$
Doing the hard homework on day 2 is greater than doing two easy homework on day 1 and 2. Since we chose the hard homework on day 2, we can only do the easy assignment on day 3. The algorithm solution is 7, so it is not optimal.

(b) (22 points) Give an algorithm with $O(n)$ worst case running time which takes as input the values $e_1, \ldots, e_n$ and $h_1, \ldots, h_n$ and returns the value of the optimal schedule. Prove its correctness and running time.

Total points after doing easy assignment on day i: $e_i + OPT(i-1)$
Total points after doing hard assignment on day i: $h_i + OPT(i-2)$

Thus, we can define the following OPT function:

$$OPT(i) \begin{cases} e_1 & \text{if } i = 1, \\ max(e_2 + e_1, h_2) & \text{if } i = 2, \\ max[e_i + OPT(i-1), h_i + OPT(i-2)] & \text{if } i > 2 \end{cases}$$

Algorithm

Let A(i) = the total points by day i, with some arbitrary schedule of doing the homework

for i = 1 to n

if i = 1

$A(1) = e_1$

else if i = 2

$A(2) = max(e_2 + e_1, h_2)$

else

$A(i) = max[e_i + A(i-1), h_i + A(i-2)]$

return $A(n)$

Correctness

Base Case:

If it is the first day (if i = 1), you can only pick the easy homework because we can't do a hard one.

If it is the second day (if i = 2), pick total easy points for the first two days or hard points for only the second day, whichever is larger.

For the sake of induction, let $1 \leq i' < i$. We assume that all $A(i') = OPT(i')$, which means they are correct.

1. Prove that $A(i) \leq OPT(i)$

By definition of an OPT function, it is the largest total points on day i. Thus, any total points on day i must be at most the optimal solution, which is the max.

2. Prove that $A(i) \geq OPT(i)$

By our induction hypothesis, $A(i-1) = OPT(i-1)$ and $A(i-2) = OPT(i-2)$

Thus, this is the equivalent of $max[e_i + OPT(i-1), h_i + OPT(i-2)] \geq OPT(i)$

Case 1: Max is $e_i + OPT(i-1)$
We assume by contradiction that $e_i + OPT(i-1) < OPT(i)$. This means that, after doing the easy homework on day i, the max points on day i is larger than the optimal at day i-1 plus e_1. If this is the case, then $OPT(i-1)$ wasn't the optimal, and it could have been larger. This can't be true, thus $e_i + OPT(i-1) \geq OPT(i)$.

Case 2: Max is $h_i + OPT(i-2)$
We assume by contradiction that $h_i + OPT(i-2) < OPT(i)$. This means that, after doing the hard homework on day i, the max points on day i is larger than the optimal at day i-2 plus h_1. If this is the case, then $OPT(i-2)$ wasn't the optimal, and it could have been larger. This can't be true, thus $h_i + OPT(i-2) \geq OPT(i)$.

$\therefore$ A(i) = OPT(i) at every point in the algorithmm which means the algorithm is correct. ■

Runtime

There is one for loop, running n iterations. Each if/else statement does constant work because it is simply checking the value of i. Each section inside each if/else statement also does constant work because it is assigning a value to $OPT(i)$ and accessing a constant number of elements. The return statement is constant time because it is simply accessing an element. Thus, the running time of the algorithm is $O(n)$. ■

3 Mobile Business (34 points)

Let's say that you have a great idea for a new food truck, and in order to save money you decide to run it out of your RV so you can live where you work. Each day i there is some demand for your food in Baltimore and some demand in Washington – let's say you would make B_i dollars by being in Baltimore and W_i dollars by being in Washington. However, if you wake up in one city (due to being there the previous day) and want to serve in the other city, it costs you M dollars to drive there.

The goal in this problem is to devise a maximum-profit schedule. A schedule is simply an assignment of locations to days – for each day i, the schedule says whether to serve in Baltimore or Washington. The profit of a schedule is the total profit you make, minus M times the number of times you have to move between cities. For the starting case, you can assume that on day 1 you wake up in Baltimore.

For example, let $M = 10$ and suppose that $B_1 = 1, B_2 = 3, B_3 = 20, B_4 = 30$ and $W_1 = 50, W_2 = 20, W_3 = 2, W_4 = 4$. Then the profit of the schedule ⟨Washington, Washington, Baltimore, Baltimore⟩ would be $W_1 + W_2 + B_3 + B_4 - 2M = 100$, where one of the M's comes from driving from Baltimore to Washington on day 1, and the other comes from driving from Washington to Baltimore on day 3. The profit of the schedule ⟨Washington, Baltimore, Baltimore, Washington⟩ would be $W_1 + B_2 + B_3 + W_4 - 3M = 50 + 3 + 20 + 4 - 30 = 47$.

Given the fixed driving cost M and profits $B_1, \ldots B_n$ and $W_1, \ldots, W_n$, devise an algorithm that runs in $O(n)$ time and computes the profit of an optimal schedule. As always, prove correctness and running time.

We can define the following OPT functions:

$OPT_B(k)$ = max profit from day 1 to k, when we stay in Baltimore on day k

$$OPT_B(k) \begin{cases} B_1 & \text{if } i = 1, \\ max[OPT_B(i-1) + B_i, OPT_W(i-1) + B_i - M] & \text{if } i > 1 \end{cases}$$

$OPT_W(k)$ = max profit from day 1 to k, when we stay in Washington on day k

$$OPT_W(k) \begin{cases} W_1 - M & \text{if } i = 1, \\ max[OPT_W(i-1) + W_i, OPT_B(i-1) + W_i - M] & \text{if } i > 1 \end{cases}$$

Algorithm

Let $A_B(i)$ = the total profit when we are in Baltimore on day i, after some arbitrary schedule of staying/traveling
Let $A_W(i)$ = the total profit when we are in Washington on day i, after some arbitrary schedule of staying/traveling

for i = 1 to n
    if i = 1
        $A_B(1) = B_1$
        $A_W(1) = W_1 - M$
    else
        $A_B(i) = max[A_B(i-1) + B_i, A_W(i-1) + B_i - M]$
        $A_W(i) = max[A_W(i-1) + W_i, A_B(i-1) + W_i - M]$

return $max[A_B(n), A_W(n)]$

Correctness
Base Case:
In the problem statement, it is given that we wake up in in Baltimore on day 1 (if i = 1). So staying in Baltimore on day 1 is $A_B(1) = B_1$. Going to Washington on day 1 is $A_W(1) = W_1 + M$ because we have to travel.

For the sake of induction, let $1 \leq i' < i$. We assume that all $A_B(i) = OPT_B(i)$ and $A_W(i) = OPT_W(i)$, which means they're correct.

For $A_B(i)$:

1. Prove that $A_B(i) \leq OPT_B(i)$
By definition of an OPT function, it is the maximum profit on day i. Thus, any total profit on day i must be at most the optimal solution, which is the max.

2. Prove that $A(i) \geq OPT(i)$
By the induction hypothesis, $A_B(i-1) = OPT_B(i-1)$ and $A_W(i-1) = OPT_W(i-1)$ are correct.
Thus, this is equivalent to $max[OPT_B(i-1) + B_i, OPT_W(i-1) + B_i - M] \geq OPT(i)$
Case 1: Max is $OPT_B(i-1) + B_i$
We can assume by contradiction that $OPT_B(i-1) + B_i < OPT(i)$. This means that, after going to Baltimore on day i, the max profit on day i is larger than the optimal at day i-1 plus B_1. If this is the case, then $OPT_B(i-1)$ wasn't the optimal, and it could have been larger. This can't be true, thus $OPT_B(i-1) + B_i \geq OPT(i)$

Case 2: Max is $OPT_W(i-1) + B_i - M$
We can assume by contradiction that $OPT_W(i-1) + B_i - M < OPT(i)$. This means that, after going to Baltimore on day i by traveling from Washington, the max profit on day i is larger than the optimal at day i-1 plus $B_1 - M$. If this is the case, then $OPT_W(i-1)$ wasn't the

optimal, and it could have been larger. This can't be true, thus $OPT_W(i-1)+B_i-M \geq OPT(i)$

Thus, $A_B(i) = OPT_B(i)$.

For $A_W(i)$:

Follows same logic as $A_B(i)$.

$\therefore A_B(i) = OPT_B(i)$ and $A_W(i) = OPT_W(i)$ at every point in the algorithm, which means the algorithm is correct. ■

Runtime
There is one for loop, running n iterations. Each iteration does constant work because it is assigning values to $OPT_B(i)$ and $OPT_W(i)$ and accessing a constant number of elements. The return statement is also constant time because it's accessing two elements then getting the maximum. Thus, the running time of the algorithm is $O(n)$. ■