Michael Inoue
405171527

Ling
Discussion 1C 2-3:50PM

CS 180

Homework 5

20 November 2019

1 Solution

Consider the closest pair algorithm:

Algorithm 1: Closest pair of points

Input : n points P with coordinates (x, y)
Output: Closest pair of points, $(x*, y*)$

1 Sort points by increasing x coordinate using merge sort and store in array P_x
2 Sort points by increasing y coordinate using merge sort and store in array P_y
3 return Algorithm 2 (n, P, P_x, P_y)

Algorithm 2: Closest pair of points (recursive helper function)

Input : n points P with coordinates (x, y), P_x, P_y
Output: Closest pair of points, $(x*, y*)$

if *number of points* ≤ 3 **then**
  Compute distance between each point
  Let $(d1, d2)$ be the smallest distance between each
  return $(d1, d2)$
end
Let L_x be the left subarray of P_x, (P_x, size $\frac{n}{2}$)
Let L_y be the set of points in L_x sorted in order of increasing y coordinate by passing through P_y
Let R_x be the right subarray of P_x, ($P_x + \frac{n}{2}$, size $\frac{n}{2}$)
Let R_y be the set of points in R_x sorted in order of increasing y coordinate by passing through P_y
Let (l'_1, l'_2) = Algorithm2(n, P, L_x, L_y)
Let (r'_1, r'_2) = Algorithm2(n, P, R_x, R_y)
Let $mindistlr = min(dist(l'_1, l'_2), dist(r'_1, r'_2)$
Let (x_1, x_2) = the points (l'_1, l'_2) or (r'_1, r'_2) with minimum distance
Let x' be the maximum x coordinate in L_x
foreach *point* $p = (x, y) \in P_y$ **do**
  if *dist(x, x′)* $\leq mindistlr$ **then**
    Add p to array of points S_y
  end
end
foreach *point* p *in* S_y **do**
  Compute the distance between p and every other point in S_y
  Let *mindistbetween* be the minimum such distance
  Let (x'_1, x'_2) be the points with such a minimum distance, *mindistbetween*
end
if $mindistlr <$ *mindistbetween* **then**
  return (x_1, x_2)
else
  return (x'_1, x'_2)
end

With the closest pair algorithm recreated above, we first sort the points by increasing x-coordinate and store them in an array, P_x, and then sort the points by increasing y-coordinate and store them in an array, P_y. Note that maintaining these sorted lists of points will allow us to efficiently divide and search these points by increasing x and y coordinate in later recursive calls of our algorithm. In each recursive call of our algorithm, we divide our list of x-coordinates into two arrays, L_x and R_x and our list of y-coordinates into two arrays L_y and R_y. Constructing L_x and R_x is simple: we may simply take the first half of P_x and store it in L_x and take the second half of P_x and store it in R_x. Constructing L_y and R_y is a bit trickier, but still straightforward. We may iterate through every point in P_y and compare its x-coordinate to the rightmost coordinate in L_x. If it is less than or equal to it, then we add the point to L_y. Otherwise (it is greater), we add it to R_y. Our result is a sorted array of points by order of increasing y-coordinate L_y and R_y, containing the points in L_x and L_y, respectively.

Finally, we must account for the closest pair of points between our set L_x and R_x, respectively. We may accomplish by taking every point p such that its x coordinate is within a distance *mindistlr* (see line 12 of Alg. 2) of a point in L_y, and then add p to an array S_y of potential candidates of closest points between L_x and R_x.

We may then consider each point $p \in S_y$ and every other point and find the minimum distance between each point, and afterwards compare this distance to *mindistlr* to determine which pair of points to return. This finishes the algorithm.

Time Complexity:

We first run merge sort twice on our initial list of points in increasing order of their x-coordinates and then in again in increasing order of their y-coordinates. Merge sort runs in $O(n \log n)$, so this process runs in $O(n \log n)$.

We then divide our array of points, P_x, in two subarrays, L_x and R_x. This simply requires pointing L_x and R_x to a certain point in the P_x array, so this runs in $O(1)$. We then construct an array of points in increasing order of y-coordinates from P_y, divided based on whether points are in L_x or R_x. This simply requires a linear scan through P_y, checking each point p's x component with the last point in L_x, and adding p to L_y if its x component is greater or equal to the last point's x component, or to R_y otherwise. This is a linear scan through P_y, so it runs in $O(n)$.

Now, we iterate through every element in S_y and compare it to every other element in S_y. We claim that there are at most 16 elements in S_y. We will prove such a claim by contradiction.

Let us create a vertical line going through the last element of L_x. This will effectively be our boundary line L between L_x and R_x. Now, let us partition the space around L into 16 boxes a distance at most *mindistlr* from L. We observe that there must be at most 1 point in each box. Suppose towards a contradiction, that there are two points in the same box. Then, by geometry, these points are at most a distance $\frac{mindistlr}{\sqrt{2}}$ from each other. But, then that would mean that the *mindistlr* is not the minimum distance between any two points in L_x or R_x, which is a contradiction. Thus, there is at most one point in each box.

Now, suppose, towards a contradiction, that there are some points p and p' that don't fall within our 16 boxes, but dist(p, p') $<$ *mindistlr*. Now, if either p and p' are not in the 16 boxes, then, using geometry, we have a distance of at least $\frac{3}{2}$*mindistlr* between p and p'. But this is a contradiction, as dist$(p, p') <$ *mindistlr*. Thus, every pair of points between L_x and R_x with a distance less than *mindistlr* must fall within the 16 boxes we have created. As we have proven that each box must contain at most one point, this means that we only have 16 points in total, and thus S_y has at most 16 points.

The main point of this proof is to demonstrate that comparing the distance between each pair of points in S_y will run in constant time. We will have no more than 16x15 = 240 comparisons in S_y. Thus, checking the minimum distance in S_y runs in constant time, $O(1)$.

We then compare the distances we computed and return the points with minimum distance, which runs in $O(1)$ time.

We are thus left with the recurrence relation for the time complexity recursive portion of our algorithm (Alg. 2), $T'(n)$:

$$T'(n) = 2T'\left(\frac{n}{2}\right) + O(n) + O(1) \tag{1}$$

$$= 2T'\left(\frac{n}{2}\right) + O(n) \tag{2}$$

In lecture, we proved that this solves as $T'(n) = O(n \log n)$, so the recursive part of our algorithm runs in $O(n \log n)$.

Now, adding up the nonrecursive (Alg. 1) and recursive parts (Alg. 2) of our algorithm, we have our total time complexity, $T(n)$:

$$T(n) = O\left(n \log n\right) + O\left(n \log n\right) \tag{3}$$

$$= O\left(n \log n\right). \tag{4}$$

2 Solution

(a) Consider the ordered graph $G = (V, E)$ with $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4), (v_4, v_5)\}$. The algorithm shown in the problem would output $L = 2$, as it would first choose the edge (v_1, v_2) (as v_2 is the node with the smallest index connected to v_1) and then choose (v_2, v_5), the only remaining node connected to v_2. This is incorrect. The correct output would be $L = 3$, as we could traverse the path (v_1, v_3, v_4, v_5), which has 3 edges and is a longer path (the longest path, specifically).

(b) Solution:

Algorithm 3: Longest path in ordered graph

Input : Ordered graph G with vertices V and edges E
Output: Longest path from v_1 to v_n, L

Initially an array of integers A is empty
Each $A[i]$ for $i = 1, 2, ..., n$ is -1
$A[1]$ is 0
for *i = 1,2..., n* **do**
    Consider each edge coming to v_j from v_i
    A[j] = max(A[i] + 1, A[j])
end
return longest path $A[n]$

We first prove the correctness of our algorithm. Our algorithm utilizes dynamic programming to update the longest path of each vertex v_j for $j = 1, 2, ..., n$. It accomplishes this by storing the value of the longest path in an array, A, where $A[i]$ is the value of the longest path to $A[i]$. Now, each $A[i]$ is defined to be -1, indicating that a path has not been found yet to each v_i. We initialize $A[1]$ to be 0 to indicate that the shortest path from v_1 to itself is 0. Now, we prove by induction that the algorithm is correct.

Our base case is for k=1 and k=2. $A[1] = 0$, which is the shortest path from v_1 to itself, so our base case for k=1 is correct. Now, for k=2, we two vertices, with at least one edge from v_1 to v_2 have that $A[2] = max(A[1] + 1, A[j])$. This will set $A[2] = A[1] + 1$, or 1, and will return the longest path, $A[2]$, which is 1, which is correct.

Now, assume that for some $k = n - 1$ that $A[1], ..., A[n - 1]$ yields the longest path to $v_1, ..., v_n$, respectively. Now, by our inductive hypothesis, there must be some node v_x connected to v_n such that the longest path $A[x]$ is the longest path before reaching v_n, as our graph is ordered. Suppose towards a contradiction that $A[n]$ does not contain the longest path to v_n. Then there is some v_y connected to v_n such that $A[n] = A[y] + 1$, and $A[y] < A[x]$. But this is impossible, as that would violate line (6) our algorithm, as $A[n]$'s value is only updated to the maximum value between $A[y] + 1$ and $A[x] + 1$. Thus, $A[n]$ contains the longest path to v_n, and our algorithm yields correct results.

Time Complexity

Our algorithm first initializes each entry of an array of size n (the number of vertices) to -1. This takes $O(n)$ time.

Next, our algorithm traverses each edge in our graph and makes a constant time update to our array. With m such edges, this runs in $O(m)$ time.

Thus, our total time complexity, $T(n)$, is:

$$T(n) = O(n) + O(m) \tag{5}$$
$$= O(m + n) \tag{6}$$

3 Solution

Algorithm 4: Determine the maximum quality of a string y

Input : String of letters $y = y_1y_2...y_n$
Output: Maximum quality of our string

```
1  Initially each A[i] = -∞ (i = 1, 2, ..., n)
2  A[0] = 0
3  for i = 1,...,n do
4  |  for 0 ≤ j ≤ i-1 do
5  |  |  if A[i] < A[j] + quality(y_{j+1}...y_i) then
6  |  |  |  A[i] = A[j]
7  |  |  end
8  |  end
9  end
10 return A[n]
```

We now prove the correctness of our algorithm. We will prove by induction.

For our base case, we have a string of length $k = 1$. Indeed, our algorithm will first set $A[1] = A[0] + quality(1)$, or simply $quality(1)$, which is optimal, as there is only one character in the string.

Now for our inductive step, suppose that for every substring of length $k \leq n-1$, $A[k]$ returns an optimal solution. Now, we must show that $A[n]$ returns an optimal solution.

We know that the ith character in the string must be included. Thus, we will have some optimal substring which ends at some jth position. Moreover, the optimal solution for the ith solution will be the substring starting at this jth position added to the substring starting at the $(j+1)th$ position and ending at i.

But, we have that our algorithm will account for such a substring, as it will choose among all possible j substrings the one which will yield the maximum quality (seen in the updating component of the inner for loop on line (6)).

Thus, we have that $A[i]$ has the optimal solution for $y_1, ..., y_i$. Hence, our algorithm is optimal for all substrings in y, and thus for $y = y_1y_2, ..., y_n$, our algorithm returning $A[n]$ is optimal.

Time Complexity:

We first set every element except $A[0]$ in an array of size $n+1$ to $-\infty$. This runs in $O(n)$ time.

We have two loops: an inner loop and an outer loop. Our outer loop simply iterates at most n times. Our inner loop simply iterates at most n times and performs constant operations (comparisons, updating an array, and calling $quality$). Hence, we perform roughly $nxn = n^2$ computations, and thus this portion runs in $O(n^2)$

Adding up our total time complexities, we have our total time complexity, $T(n)$:

$$T(n) = O(n) + O(n^2) \tag{7}$$
$$= O(n^2) \tag{8}$$

4 Solution

(a) Consider the following processing power schedule:

A: {2, 2, 1, 50}

B: {1, 2, 4, 1}

The proposed solution would first choose 2, then choose 4, as 2+1 < 4. Then, the last element it could choose is 1. The plan proposed would hence have a total processing value of 2+4+1 = 7. But our optimal plan would be to choose 2, 2, 1, 50, which would yield a total processing value of 2+2+1+50 = 55, so the proposed solution is incorrect.

(b)

Algorithm 5: Find the maximal processing value

Input : Processing values $a_1, a_2, ..., a_n, b_1, b_2, ..., b_n$
Output: Maximal processing value

```
Arrays A, B are empty arrays of size n + 1
A[0] = B[0] = 0
A[1] = a_1
B[1] = b_1
for 2 ≤ i ≤ n do
    A[i] = a_i + max(A[i-1], B[i-2])
    B[i] = b_i + max(B[i-1], A[i-2])
end
return (max(A[n], B[n]))
```

We will first prove the correctness of our algorithm by induction.

Now, for our base case, we have $k = 1$, so we only have two possible processing power values in our schedule, a_1 and b_1. Our algorithm will simply yield the larger value of A[1] and B[1], which is optimal.

Now for our inductive hypothesis, assume that for $k \leq n - 1$, A[k] and B[k] are the optimal processing values of a plan which ends at computers A and B, respectively.

Hence, we have that max(A[k], B[k]) would be the optimal process value of a plan with k time slots (*). We want to show that our algorithm is optimal for the $(k = n)$th time slot, that is, A[n] and B[n] contain the optimal processing values for a plan ending at computers A and B, respectively. Note that this, along with statement (*) yields that we will have the optimal processing value for any schedule and will finish the proof.

Now, consider the nth time interval. Now, if we finish at computer A at time n, then we either were at computer A at time $n - 1$ or computer B at time $n - 2$ and were in the process of switching. By our inductive hypothesis, A[n-1] contains an optimal processing value for a plan finishing at A at time $n-1$ and B[n-2] contains an optimal processing value for a plan finishing at B at time $n - 2$.

Hence, we have that A[n] = a_i + A[n-1] or A[n] = a_i + B[n-2], or equivalently: A[n] = a_i + max(A[n-1], B[n-2]).

Now, if we finish at computer B at time n, then we either were at computer B at time $n-1$ or computer A at time $n-2$ and were in the process of switching. By our inductive hypothesis, A[n-1] contains an optimal processing value for a plan finishing at B at time $n - 1$ and A[n-2] contains an optimal processing value for a plan finishing at A at time $n - 2$.

Hence, we have that B[n] = b_i + B[n-1] or B[n] = b_i + A[n-2], or equivalently: B[n] = b_i + max(B[n-1], A[n-2]).

So A[n] and B[n] have optimal processing values, and we are done: our algorithm is optimal.

Time Complexity: Our algorithm utilizes a for loop which runs at most $n-1$ times and performs constant operations with each iteration (comparisons and updating A[i] and B[i] for each i iteration). Hence, we have our total time complexity, $T(n) = O(n)$.

5 Solution

Algorithm 6: Determine the maximal value of a rod

Input : Rod of length n, array of prices P (P[1] is the first price, P[2] is the second, etc.) for each piece of length $i \leq n$
Output: Maximal value of rod

```
1  Initially array of prices A is empty
2  Set A[i] = −∞ for i = 2, 3, 4, ..., n
3  Set A[0] = 0
4  Set A[1] = P[1]
5  for 2 ≤ i ≤ n do
6  |  for 0 ≤ j ≤ i-1 do
7  |  |  if P[j] + A[i-j] > A[i] then
8  |  |  |  A[i] = P[j] + A[i-j]
9  |  |  end
10 |  end
11 end
12 return A[n]
```

We will first prove the correctness of the algorithm by induction.

For our base case, we have a rod of size $k = 1$. In this case, our algorithm will set A[1] = p_1 and then return A[1], or p_1. Indeed, as p_1 is the only possible value, it must be a maximal value, and hence is optimal.

Now, for our inductive hypothesis, assume that for some $k \leq n - 1$, A[k] contains the maximal value for a rod of length k. We wish to show that A[n] will contain the maximal value for a rod of length n. Now, if a rod is of length n, then either it will be split into smaller rods or it will not.

If it is not, then keeping the rod intact is the maximal value (P[n]). Our algorithm accounts for this by starting at index $j = 0$ (i.e. splitting a rod into a rod of length 0 and a rod of length n, or effectively, not splitting the rod at all). Now if the rod is divided, then it will be divided into at least two components, one of a length j, and another of length $i - j$, which may be divided further if doing so will yield a greater value.

But our algorithm indeed finds the maximal value to all other divided rods of length $i - j$ and j, A[i-j] + P[j], as, by our inductive hypothesis, the value of these rods will have maximal value for their respective division, so the maximal value of these rods should be stored in A[n] by the end of the algorithm.

Hence, A[n] will be the maximal value of a rod of length n by the end of the algorithm, and as A[n] is returned at the end of the algorithm, our algorithm is optimal.

Time Complexity

Our algorithm first sets n values in an array of size $n + 1$ to -∞. This runs in $O(n)$ time.

Next, our algorithm utilizes two for loops, an inner loop and an outer loop. The outer loop simply performs an iteration on a counter variable, i. This iterates at most $n - 1$ times. The inner loop performs an iteration on a counter variable j, which iterates at most n times. The body of the loop runs in constant time, performing a constant number of comparisons and updates in our array A.

In total, we will have nx$n = n^2$ iterations of our inner loop, each of which runs at constant time, so this portion of our algorithm runs in $O(n^2)$ time.

Adding up our time complexities, we have our total time complexity, $T(n)$:

$$T(n) = O(n) + O(n^2) \tag{9}$$
$$= O(n^2) \tag{10}$$

6 Solution

Algorithm 7: Find the maximum definite value of coin game

Input : A list of n coins with values $v_1, ..., v_n$
Output: Optimal value of coins from playing game

```
Initially an array A of values of size n + 1 is empty
for k = 1; k ≤ n; k++ do
    for i = 1, j = k; i ≤ n-k; i++, j++ do
        values optA, optB, optC = 0
        if i+2 ≤ j then
            optA = A[i+2][j]
        end
        if i+1 ≤ j-1 then
            optB = A[i+1][j-1]
        end
        if i ≤ j - 2 then
            optC = A[i][j-2]
        end
        optI = v_i + min(optA, optB)
        optJ = v_j + min(optC, optB)
        A[i][j] = max(optI, optJ)
    end
end
return A[1][n]
```

We will first prove the correctness of our algorithm by induction.

As a preliminary measure, we note the following properties of the coin game and how we will approach solving this problem:

1.) The opponent maximizes the total value of their coins; that is, they are intelligent and make choices that maximize their wealth at the end of the game.

2.) Suppose we have a list of coins, starting at the ith position and ending at the jth position. Then, either choosing the ith or jth coin will be optimal.

case i: Suppose we choose the ith coin. Then our opponent will choose either the jth or $i+1th$ (leaving us with the sequence of coins $(i+1, j-1)$ or $(i+2, j)$, respectively) on the basis of minimizing our total profit: that is, if $opt(i, j)$ is the optimal choice for a sequence of coins beginning at position i and ending at position j, then we have:

$$opt(i, j) = v_i + min(opt(i+1, j-1), opt(i+2, j)) \tag{11}$$

case j: Suppose we choose the jth coin. Then our opponent will choose either the ith or $j+1th$ (leaving us with the sequence of coins $(i+1, j-1)$ or $(i, j-2)$, respectively) on the basis of minimizing our total profit: that is, if $opt(i, j)$ is the optimal choice for a sequence of coins beginning at position i and ending at position j, then we have:

$$opt(i, j) = v_j + min(opt(i+1, j-1), opt(i, j-2)) \tag{12}$$

3.) We will denote the array A[i][j] to contain the optimal possible value we can receive if we go first in

choosing from a sequence of coins beginning at the ith coin and ending at the jth coin.

4.) Due to the nature of traversing 'forward' and 'backwards' with regards to our variables i and j, respectively, we will fill our array along the diagonal (as we use the diagonal entries to compute new entries); thus, we will use a blocking variable k, to accomplish this (line (2) of our algorithm).

Now for our base case, we have that $k = 2$. Then, our algorithm will choose the maximum of $optI = v_i + 0 = v_i$ and $optJ = v_j + 0 = v_j$ to fill our entry $A[1][2]$, which gets returned. Indeed, the maximum value of v_i and v_j is optimal, so our base case holds.

Now, for our inductive step, assume that for a sequence of size $k \leq n-1$, and $i = n-k, j = k$, A[i][j] contains the maximal value of a sequence of coins starting at the ith position and ending at the jth position. Then, for the nth coin we encounter, we may thinking of adding a coin to the left or right side of our current sequence of length $n-1$. Suppose, without loss of generality (as i, j are entirely symmetric), that the coin is added on the right side of our sequence. Now, our algorithm must make a choice: choose the new coin added to our sequence, or don't. If we do, then we have chosen the $j+1th$ coin and our opponent will choose either the jth or ith coin to minimize our profits. Thus, we are left with $v_{j+1} + \min(optC, optB)$ as our total value. But, if choosing the $j+1th$ coin is optimal, then by our inductive hypothesis, the rest of our choices in the sequence prior to the jth coin is optimal, and hence our final value is optimal. Now, if choosing the ith coin instead is optimal, then our opponent will choose either the $i+1th$ or $j+1th$ coin to minimize our profits. Thus, we are left with $v_i + \min(optA, optB)$ as our total value. But, if choosing the ith coin is optimal, then by our inductive hypothesis, the rest of our choices in the sequence after the ith coin is optimal, and hence our final value is optimal.

Now, our algorithm determines whether to choose the ith or $j+1th$ coin by comparing the values of either choice; hence, our final choice should also be optimal.

Thus, our algorithm yields correct results for a sequence of size $k = n$, so A[n-k][n] = A[n-(n-1)][n] = A[1][n] is the optimal value for a sequence of size n. This value is returned, so our algorithm is optimal.

Time Complexity: Our algorithm utilizes two for loops: an inner loop and an outer loop. The outer loop iterates at most n times. Then inner loop iterates at most n times, and performs a constant number of comparisons, computations, and updates to an array A. Thus, we have nxn=n^2 iterations of our inner loop, with a constant complexity in each iteration, so our algorithm runs in $O(n^2)$ time.