CS255 (section 2)
Design and Analysis of Algorithms
San Jose State University
Final Exam, 19/May/2014, duration: 2h15m

Read all questions before starting the midterm. When you're asked to write an algorithm, you should use pseudocode, C, or Java.

(10 points) Question 1.

Give two reasons why it may be preferable to implement a $\Theta(n^2)$ algorithm over an $\Theta(n)$ algorithm that solves the same problem.

Answer

Here are two reasons (there could be others as well).

1. If we are only interested in solving instances where n is not very large, the $\Theta(n^2)$ algorithm can be faster due to larger constant factors on the $\Theta(n)$ algorithm.
2. If the $\Theta(n)$ algorithm is more complicated to implement than the $\Theta(n^2)$ algorithm, then we may opt for the $\Theta(n^2)$ algorithm. Execution time is not the only thing that matters; Implementation and maintenance of software matters too.

Point deduction:

- -3 points for giving an unreasonable reason
- -5 points for giving only 1 reason
- -10 points for saying complete nonsense

(15 points) Question 2.

Suppose you are given an array A with n distinct integers. You are told that the sequence of values $A[1], A[2], ..., A[n]$ is unimodal: For some index p between 1 and n, the values in the array entries increase up to position p and then decrease the remainder of the way until position n.

Write an algorithm to return the maximum value of array A. Your algorithm should run in $\Theta(\lg n)$.

Answer

This is similar to binary search. Divide and conquer solves the problem. Find *mid* position of the array. If $A[mid] < A[mid + 1]$, then the maximum element is in the righthand part of the array, otherwise it's in the lefthand part. In each iteration we cut the size of the array by a half and this will give us $\Theta(\lg n)$ time.

```
MAX-UNIMODAL(A, left, right)
    if left == right
        return A[left]
    else
        mid = ⌊(left + right)/2⌋
        if A[mid] < A[mid + 1]
            return MAX-UNIMODAL(A, mid + 1, right)
        else
            return MAX-UNIMODAL(A, left, mid)
```

Initial call: MAX-UNIMODAL$(A, 1, n)$

Recurrence for running time, $T(n) = T(n/2) + \Theta(1)$
Identical to binary search. Solution is $T(n) = \Theta(\lg n)$

Point deduction:

- -2 points for minor mistake in code
- -5 points for major mistake in code
- -5 points for not putting any code but giving the idea in words
- -15 points for not giving $\Theta(\lg n)$ algorithm

($10 + 5 = 15$ points) Question 3.

Consider a list of cities c_1, c_2, ..., c_n. Assume we have a boolean function $f(c_i, c_j)$ that returns 1 if c_i and c_j are in the same province, and returns 0 otherwise.

(a) Using the UNION-FIND data structure, write pseudocode for an algorithm that puts each city in a set such that c_i and c_j are in the same set if and only if they are in the same province.

Answer

Start by creating n sets, one for each city. Then compare every pair of cities and put them in the same set (via UNION) in case the two cities are in the same province.

Below is the pseudocode, assuming the cities are stored in an array c.

```
CITY-PROVINCE(c, n, f)
    for i = 1 to n
        MAKE-SET(c[i])
    for i = 1 to n − 1
        for j = i + 1 to n
            if f(c[i], c[j]) == 1
                UNION(c[i], c[j])
```

Point deduction:

- -2 points for minor mistake in code
- -3 points for not so minor mistake in code
- -5 points for major mistake in code (but has correct idea)
- -8 points for being almost totally off
- -10 points for being totally off

(b) When the cities are stored in the UNION-FIND data structure, if you are given two cities c_i and c_j, how do you check if they are in the same province?

Answer

Just need to check if FIND-SET($c[i]$) == FIND-SET($c[j]$). If yes, they are in the same province. Otherwise they are not.

Point deduction:

- -2 points if some nonsense is mentioned on top of a correct answer
- -5 points for being totally off

(15 points) Question 4.

The minimum bottleneck spanning tree (MBST) is a spanning tree that seeks to minimize the most expensive edge in the tree. More specifically, for a tree T over a graph G, we say that e is a bottleneck edge of T if it's an edge with maximal cost. The tree T is a minimum bottleneck spanning tree if T is a spanning tree and there is no other spanning tree of G with a cheaper bottleneck edge. (Recall that a spanning tree is a tree that touches all the vertices of a graph.)

Assume you are given a graph $G = (V, E)$ and a weight w. Give a $O(|V| + |E|)$ time algorithm that returns TRUE if G has a MBST with bottleneck edge with cost no more than w, and return FALSE otherwise. (You can answer this question in plain english alluding to algorithms given in class.)

Answer

Remove from G all edges with weight greater than w (this takes $\Theta(E)$). If the resulting graph consists of a single connected component, then there will be a MBST with bottleneck edge with cost no more than w. Otherwise, no such MBST will exist.

To check if the graph consists of a single component, we can run a modified DFS (or BFS). For example, run BFS starting on an arbitrary vertex. The graph is a single connected component if and only if all the vertices of the graph have been visited (i.e., they are all non-white when BFS ends).

Running BFS takes $O(V + E)$. Checking that all vertices have been visited takes $\Theta(V)$. Thus, the overall strategy takes $\Theta(E) + O(V + E) + \Theta(V) = O(V + E)$

Point deduction:

- -2 points for minor mistake
- -5 points for getting the idea but not explaining it properly
- -10 points for not giving an $O(V + E)$ algorithm
- -15 points for being totally off

(10 points) Question 5.

Suppose problem Y is polynomial time reducible to problem X, i.e., $Y \leq_p X$. If Y cannot be solved in polynomial time, is it possible that X be solved in polynomial time? Justify your answer.

Answer

No. Because if we could solve X in polynomial time, then we would be able to solve Y in polynomial time using the reduction, which contradicts the hypothesis.

Point deduction:

- -2 points for minor mistake
- -5 points for wrong justification
- -10 points for being totally off

($10 + 15 + 10 = 35$ points) Question 6.

You are managing the construction of billboards along a stretch of a highway that runs west-east for M miles. You may assume this highway is a straight line to keep things simple. The possible sites for billboards are given by numbers $x_1 < x_2 < \ldots < x_n$, each in the interval $[0, M]$, specifying their position in miles measured from the western end of the road. If you place a billboard at position x_i, you receive a profit of $p_i > 0$.

Your goal is to place billboards at a subset of the sites so as to maximize your total profit, subject to the restriction that no two billboards are within 5 miles of each other.

For example, suppose $M = 20$ and $n = 5$ with the x_i and r_i values as shown below,

i	1	2	3	4	5
x_i	6	7	12	13	14
p_i	5	6	5	3	1

Then the best solution is to place billboards at x_1 and x_3 to achieve a revenue of 10.

The input to this problem is a set of pairs (x_i, p_i), sorted in increasing order by the x_i values.

(a) Your boss suggests a greedy approach to the problem: Put a billboard a location x_1. From then on, put a billboard at the next smallest x_i that is more than 5 miles from your most recently placed billboard.

Give an example of an input on which this approach does not find an optimal solution.

Answer

Suppose $p_3 = 2$ instead of 5 in the example shown. Then the boss' strategy would return x_1 and x_3 with a total profit of $5 + 2 = 7$. The optimal solution however would be x_2 and x_4 with a total profit of $6 + 3 = 9$.

Point deduction:

- -2 points for minor mistake
- -5 points for not so minor mistake
- -10 points for giving a wrong example

(b) Write a recurrence for obtaining the maximum possible profit. Explain your reasoning and justify the correctness of your recurrence.

[Hint: As a first step towards the solution, define $e(j)$ to be the easternmost site that is more than 5 miles away from x_j. In other words, if you place a billboard at x_j, then $x_1, x_2, \ldots, x_{e(j)}$ are also valid places to place billboards (subject to the same restriction about distances). Note that computing the $e(j)$ values takes time $O(n)$.]

Answer

Let $e(j)$ be defined as it is in the hint, and let $\mathrm{OPT}(n)$ be the maximum profit obtainable by placing billboards on a subset of the locations $x_1, \ldots, x_n$.

Consider location x_n. Two things can happen,

1. we place a billboard at location x_n
2. we don't place a billboard at location x_n

$\mathrm{OPT}(n)$ will be the best of these two alternatives.

If alternative (1) is the best one then $\mathrm{OPT}(n) = p_n + \mathrm{OPT}(e(n))$. (The locations $x_{e(n)+1}, \ldots, x_{n-1}$ are all within 5 miles of x_n and cannot be used.) On the other hand, if alternative (2) is the best one then $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$.

As base case we have $\mathrm{OPT}(0) = 0$.

$$\mathrm{OPT}(n) \quad = \quad \begin{cases} 0 & \text{, if } n = 0 \\ \max\big(p_n + \mathrm{OPT}(e(n)), \mathrm{OPT}(n-1)\big) & \text{, if } n > 0 \end{cases}$$

Point deduction:

- -2 points for showing pseudocode instead of recurrence
- -3 points for minor mistake
- -7 points for major mistake, but getting the basic idea
- -10 points for having just a little bit of the answer correct (say, the base case of the recurrence and little more.)

(c) Based on your answer to (b), write a dynamic programming (or a memoized) algorithm to solve the problem. What's the running time of your algorithm? Justify the running time obtained.

Answer

Algorithm takes as input two arrays: $x[1..n]$ with the possible billboard locations, and $p[1..n]$ with the corresponding profits.

```
MAX-PROFIT-BILLBOARDS(x, p, n)
    Let e[1..n] be an array      // to store the e(j) values
    Let opt[0..n] be an array    // to store the optimal values for subproblems
    e = COMPUTE-E-VALUES(x, n)
    opt[0] = 0
    for i = 1 to n
        opt[i] = max(p[i] + opt[e[i]], opt[i − 1])
    return opt[n]
```

```
COMPUTE-E-VALUES(x, n)
    Let e[1..n] be an array
    j = 0
    for i = 1 to n
        while x[i] − x[j + 1] > 5
            j = j + 1
        e[i] = j
    return e
```

The running time of the algorithm is equal to the running time of COMPUTE-E-VALUES, plus the time to compute the n subproblems. Each subproblem takes constant time, so the time to compute the n subproblems is $\Theta(n)$.

The running time for COMPUTE-E-VALUES is $\Theta(n)$. Note that the body of the **while** loop is executed at most n times in all of the **for** loop iterations. (I would accept a more trivial $\Theta(n^2)$ implementation for COMPUTE-E-VALUES)

Point deduction:

- -2 points for wrong running time
- -2 points for minor error in code
- -5 points for major error in code, or no code but explaining the idea, or code based on answer in (b)
- -10 points for being totally off