CSEP521: Algorithms November 11, 2022

Midterm

Anup Rao Due: November 21, 2022

Read the fine print[1]. An algorithm is said to run in polynomial time if it runs in time $O(n^d)$ for some constant d on inputs of size n.

1. (45 points, 5 each) For each of the following problems answer **True** or **False** and BRIEFLY JUSTIFY you answer.

   (a) $n^{3.05} = O(n^3 \log n)$.
   False. $n^{0.05}$ grows faster than $\log n$, as we discussed in class.

   (b) $2^n = \Omega(2^{2n})$.
   False. For every constant c, as soon as $2^n > c$, we have $c \cdot 2^{2n} > 2^n$.

   (c) There is a polynomial time algorithm for deciding whether a graph has an odd length cycle or not.
   True. We can use breadth first search to check whether a graph is bipartite or not, this is the same algorithm.

   (d) If the running time of an algorithm satisfies the recurrence $T(n) \leq 3T(n/2) + n^5$, then $T(n) = O(n^3)$.
   False. The first recursive step itself takes time $O(n^5)$.

   (e) Suppose an input to the MST problem does *not* have distinct edge weights. Suppose there is a cycle in the graph where two of the edges have the heaviest weight. Is it true that every MST must exclude *both* of these heavy edges?
   False. Consider a cycle where every edge has the same weight.

   (f) In an input to the MST problem, if e is the unique lightest edge touching a vertex v, then e must be part of every MST.
   True. This follows from the cut property applied to the vertex v.

   (g) There is an $O(n^{1.1})$ time algorithm for multiplying two n-bit numbers.
   True. This can be done using the FFT.

   (h) If an undirected graph contains a matching of size k, then it is possible that it also has a vertex cover of size $k - 1$
   False. Every edge of the matching requires one vertex in the cover to cover it.

   (i) You are given an input the set cover problem, with m sets over an n element universe. You are promised that $m > n$, and either at most $\sqrt{n}$ of the sets can cover the whole universe, or at least n sets are required. There is a polynomial time algorithm for

[1] No collaboration on this one. The problems have been carefully chosen for their pedagogical value, and hence might be similar to those given in past offerings of this course at UW, or similar to other courses at other schools. Using any pre-existing solutions from these sources, for from the web, constitutes a violation of the academic integrity you are expected to exemplify, and is strictly prohibited. Most of the problems only require one or two key ideas for their solution. It will help you a lot to spell out these main ideas so that you can get most of the credit for a problem even if you err on the finer details. Please justify all answers. Some other guidelines for writing good solutions are here: `http://www.cs.washington.edu/education/courses/cse421/08wi/guidelines.pdf`.

deciding which case is satisfied by the input.
True. Run the algorithm from class, which will either find a set cover of size $O(\sqrt{n}\log n)$ or not.

2. (20 points) You are given an $n \times n$ checkerboard and a checker. You are allowed to move the checker from any square x to a square y on the board as long as square y is directly above x, or y is directly to the left of the square directly above x, or y is directly to the right of the square directly above x. Every time you make a move from square x to square y, you earn $p(x, y)$ points, where $p(x, y)$ is an integer (possibly negative) that is part of the input. Give an algorithm that on input the values $p(x, y)$, outputs the maximum score that can be achieved by placing the checker on some square on the bottom row and moving the checker all the way to some square of the top row. The algorithm should run in polynomial time.

Solution. Given any square x on the board, let S_x denote the best score that can be achieved starting at that square. Then observe that if x is on the top row, $S_x = 0$. For every other square x, the maximum square can be obtained by making a valid move to a square in a row that is higher, and then making the optimal moves from there.

Let us write $\mathsf{valid}(x)$ to denote the set of valid squares that you can move to from x. Then we have that for x not on the top row,

$$S_x = \max_{y \in \mathsf{valid}(x)} \{p(x, y) + S_y\}.$$

The algorithm simply computes these values starting from the top row to the bottom row. In the description of the algorithm, we shall assume that the square (i, j) is in the i'th row and j'th column, and that the top row is the 1st row.

Input: Scores $p(x, y)$ for each move (x, y).
begin
  for $j = 1, \ldots, n$ **do**
    Set $S_{(1,j)} = 0$.
  end
  for $i = 2, \ldots, n$ **do**
    for $j = 1, \ldots, n$ **do**
      Set $S_{(i,j)} = \max_{y \in \mathsf{valid}((i,j))}\{p((i, j), y) + S_y\}$
    end
  end
  return $\max_j S_{(n,j)}$.
end

Algorithm 1: Checkers

Clearly, the algorithm correctly computes S_x for every square on the board. The final output is then the score achievable from the best starting square. The running time of the algorithm is $O(n^2)$.

3. (15 points) Suppose you are choosing between the following three algorithms:

- Algorithm A solves the problem by dividing it into five subproblems of half the size, recursively solves each subproblem, and then combines the solution in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$, and then combines the solution in constant time.
- Algorithm C solves the problem by dividing it into nine subproblems of one third the size, recursively solves each subproblem, and then combines the solutions in quadratic time.

What are the running times of each of these algorithms?

Solution

- The recurrence for Algorithm A is $T(n) = 5T(n/2) + O(n)$. By the Master theorem, it runs in time $O(n^{\log_2 5})$.
- The recurrence for Algorithm B is $T(n) = 2T(n-1) + O(1)$. This algorithm runs in time $O(2^n)$.
- The recurrence for Algorithm C is $T(n) = 9T(n/3) + O(n^2)$. By the Master theorem, it runs in time $O(n^2 \log n)$.

4. (20 points) Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$, we want to calculate the length of the longest common substring, namely the largest k for which there are i, j such that $x_i x_{i+1} \ldots x_{i+k-1} = y_j y_{j+1} \ldots y_{j+k-1}$. Show how to do this in time $O(mn)$.

Solution. Denote $opt(i, j)$ as the longest common substrings that starts at x_i and y_j. Then we have the following rule:

$$opt(i,j) = \begin{cases} 1 + opt(i+1, j+1) & \text{if } x_i = y_j \\ 0 & \text{else.} \end{cases}$$

And the boundary conditions are

$$opt(i,n) = \begin{cases} 1 & \text{if } x_i = y_n \\ 0 & \text{else.} \end{cases}$$

$$opt(m,j) = \begin{cases} 1 & \text{if } x_m = y_j \\ 0 & \text{else.} \end{cases}$$

So we have the following algorithm:

```
Input: strings x_1, ..., x_m and y_1, ..., y_n
Result: length of longest common strings
Initialize table OPT[m][n]
for i=m,m-1,...,1 do
    for j=n,n-1,...,1 do
        if i == m or j == n then
            OPT[i][j] = (x_i == y_j)
        end
        else if x_i == y_j then
            OPT[i][j] = 1 + OPT[i+1][j+1]
        end
        else
            OPT[i][j] = 0
        end
    end
end
return max(OPT[i][j])
```

Runtime: The loop will use time $O(mn)$. In order to find the maximal value in a array of size mn, we only need to scan the array and update the value when we find a larger element. So it will cost $O(mn)$. In total, this will be $O(mn)$.

Correctness: First we prove that the recurrence is correct with induction. For the case when $i = m$ or $j = n$, the recurrence just consider all the possible cases and hence correct. Assume we have already prove for $i > r$ or $j > s$. Now consider the case when $i = r$. For $OPT[r][j]$, if $x_r \neq y_j$, then $OPT[r][j]$ can only be 0. If $x_r == y_j$, since the common strings starts at x_r and y_j, the length must be 1 plus the length of the rest, which will be the common strings starts at x_{r+1} and y_{j+1}. By induction $OPT[r+1][j+1]$ will be the correct value, so $OPT[r][j]$ is also correct. Simialr argument can performed for $j = s$. So by induction $OPT[i][j]$ will be computed correctly as the length of the longest common strings that starts at x_i, y_j. Then the algorithm just returns the maximal value.