CSE421: Design and Analysis of Algorithms November 8, 2020

Homework 5

Anup Rao Due: November 14, 2020

Read the fine print[1]. Each problem is worth 10 points:

1. Given a sequence of integers $x_1, \ldots, x_n$ (possibly including negative integers) and an interval of coordinates $I = [i, j]$, write x_I to denote the sum $\sum_{i \leq k \leq j} x_k$. Give a linear time algorithm to find the interval that maximizes x_I, given the numbers $x_1, \ldots, x_n$ as input.

   Solution. **Idea**: Iterating through the sequence once to find the max element and set interval max interval[i, j] where i = j = index of that element. Then iterate through the sequence again. Set the current sum to the sum of consecutive subsequence where sum is not negative. Update the maxSum and the corresponding interval if needed.

[1] In solving the problem sets, you are allowed to collaborate with fellow students taking the class, but **each submission can have at most one author**. If you do collaborate in any way, you must acknowledge, for each problem, the people you worked with on that problem. 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`.

```
Input: A sequence of integers
Result: Interval[i, j] where 1 ≤ i, j ≤ n such that the sum of all the integers in this
        interval is maximum
Set i, j, curStart = 1;
Set curSum, maxSum = x_1;
for integer ℓ in 1 through n do
    if x_ℓ > maxSum then
        maxSum = x_ℓ;
        i = ℓ;
        j = ℓ;
    end
end
if maxSum ≤ 0 then
    output [i, j];
    end of algorithm;
end
curSum = 0; maxSum = 0;
for integer s in 1 through n do
    curSum += x_s;
    if curSum < 0 then
        curSum = 0;
        curStart = s + 1;
    end
    else
        if curSum > maxSum then
            maxSum = curSum;
            j = s; i = curStart;
        end
    end
end
output [i, j];
```

Runtime: The algorithm goes through the sequence twice. Inside each for loop, all the operation can be done in constant time. So the algorithm has runtime $O(n)$.

Correctness: CurSum keeps track of the largest interval ending at x_s. We look at all positive intervals, and for each one of them we compare it with maxSum. Therefore the interval we output is the max interval among all the possible positive intervals. The first iteration sets up maxSum. It ensure if all elements in the sequence are negative integers, then we will output the single largest elements since the sum only decline when adding another negative integer to the sum.

2. Given a sequence of characters $c_1, \ldots, c_n$, we say that a subsequence is a *palindrome* if it reads the same forwards and backwards. For example, "a,b,a,c,a,b,a" is a palindrome. Give an $O(n^2)$ time algorithm to find the longest palindrome subsequence in the input sequence $c_1, \ldots, c_n$. For example, in the sequence $c, l, m, a, l, f, d, c, a, f, m$, the longest palindrome subsequence is m, a, d, a, m. HINT: For $i < j$, let $p(i, j)$ denote the length of the longest palindrome in $x_i, \ldots, x_j$. Express $p(i, j)$ in terms of $p(i + 1, j), p(i, j - 1), p(i + 1, j - 1)$. Evaluate the values $p(i, j)$ in order of increasing $|i - j|$.

Solution. The pseudocode is given below.

```
Input: A list c[1, ..., n] of characters.
Result: The longest palindrome subsequence of c.
Let P(i, j) be the n × n matrix where the i, jth entry represents the best palindrome found
 thus far between c_i and c_j, inclusive.;
Let P(i, j) be the empty string whenever i > j;
Let P(i, i) be c_i for all i;
for length = 2, ..., n do
    for start = 1, ..., n − length + 1 do
        Let end = start + length − 1;
        if c[start] == c[end]) then
            Let P(start, end) = c_start + P(start + 1, end − 1) + c_end;
        end
        else
            if P(start, end − 1).length > P(start + 1, end) then
                Let P(start, end) = P(start, end − 1);
            end
            else
                Let P(start, end) = P(start + 1, end);
            end
        end
    end
end
return P(1, n);
```

Runtime: The first three assignments, since they are to an array of size $n \times n$ and each assignment occurs only once to a location, each run in $O(n^2)$. Then, inside the double for loop, each statement runs in constant time(it is essentially a lookup). Each loop has up to n values it can take, so the overall double for loop runs in $O(n^2)$. The return is merely a table lookup, so it runs in constant time. Thus, the entire algorithm runs in $O(n^2)$ time.

Proof of correctness: We now are left with the task of proving $|OPT(i, j)| = |P(i, j)|$, where $|OPT(i, j)|$ refers to the longest palindrome subsequence from i to j. It is sufficient to

prove

$$|OPT(i,j)| \geq |P(i,j)| \tag{1}$$
$$|OPT(i,j)| \leq |P(i,j)| \tag{2}$$

To prove equation (1), we use the fact that the palindrome subsequence $|P(i,j)|$ is feasible solution. Given that it is clear that $|OPT(i,j)| \geq |P(i,j)|$.
To prove equation (2), we perform an induction on the length of the sequence, that is an induction on $j - i + 1$.
Base Case: $j = i+1$. The longest palindrome is of length 1 or 2 depending on whether $c_i = c_j$. This implies that $|OPT(i,j)| \leq max\{P(i,j-1), P(i+1,j), 2 + P(i+1,j-1)\} = |P(i,j)|$.
Induction Hypothesis: $|OPT(i,j)| \leq |P(i,j)|$ $\forall i,j$ s.t $j - i + 1 = k$
Consider the solution $OPT(i,j)$ ($\forall i,j$ s.t $j - i + 1 = k + 1$). We have the following cases:

(a) $c_i \in OPT(i,j) \wedge c_j \notin OPT(i,j)$: We can conclude that $OPT(i,j) = OPT(i,j-1)$.

(b) $c_i \notin OPT(i,j) \wedge c_j \in OPT(i,j)$: We can conclude that $OPT(i,j) = OPT(i+1,j)$.

(c) $c_i \in OPT(i,j) \wedge c_j \in OPT(i,j)$: We can conclude that $OPT(i,j) = c_i.OPT(i+1,j-1).c_j$

Now we have, $|OPT(i,j)| \leq max\{|OPT(i,j-1)|, |OPT(i+1,j)|, 2 + |OPT(i+1,j-1)|\}$. Now by induction hypothesis we have $|OPT(i,j)| \leq |P(i,j)|$.

Common errors. Some solutions returned the length of the longest palindromic subsequence instead of the subsequence itself. Some solutions filled the matrix in an incorrect order, trying to call entries the algorithm hadn't filled out yet.

3. You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product i you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the selling price of the product is c_i Assume the a_i, b_i and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that runs in time that is polynomial in X, Y, n and determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none, if desired.

Solution. The crux of this problem is to identify precisely which actions are available to the machine:

- Make a vertical cut
- Make a horizontal cut
- Do nothing (and sell the current item)

Input: Dimensions of cloth X,Y, and a list of item values and dimensions.
Result: Best possible value of the cloth
Let *cut* be an X by Y dimensional array with every entry initialized to 0.
for $x \in [0, X-1]$ **do**
  for $y \in [0, Y-1]$ **do**
    for $x_{cut} \in [1, x-1]$ **do**
      $cut[x,y] = max(cut[x,y], cut[x_{cut},y] + cut[x - x_{cut}, y])$
    end
    for $y_{cut} \in [1, y-1]$ **do**
      $cut[x,y] = max(cut[x,y], cut[x,y_{cut}] + cut[x, y - y_{cut}])$
    end
    for $item \in Items$ **do**
      if $item_{dimensions} == (x,y)$ **then**
        $cut[x,y] = max(cut[x,y], item_{value})$
      end
    end
  end
end
return $cut[X-1, Y-1]$
// Note: This does not actually retrieve the necessary cuts. The cuts could be retrieved by storing which actions are taken along the way, and storing those actions along side their corresponding values in *cut*.

Run time: The outer two loops lead to $O(XY)$ iterations over the inner most piece, which does tries every possible vertical cut, horizontal cut, and item. The overall runtime is $O(XY) \cdot O(X + Y + n) = O(XY(X + Y + n))$.

Proof of correctness: We have to prove that $OPT(x, y) = cut(x, y)$. Here, OPT refers to the optimum solution to the problem and *cut* refers to the solution returned by the above algorithm. It is sufficient to prove

$$OPT(x, y) \geq cut(x, y) \tag{3}$$
$$OPT(x, y) \leq cut(x, y) \tag{4}$$

To prove equation (1), we use the fact that the solution returned by $cut(x, y)$ is a feasible solution and hence $OPT(x, y)$ can only do better, impying $OPT(x, y) \geq cut(x, y)$.
We prove equation 2 by induction on the size of xy.
Base Case: $(x, y) = (1, 1)$. It is clear here that $OPT(1, 1)$ could be 0 or the maximum price given by a product of dimension 1×1. In both cases, $OPT(1, 1) = cut(1, 1)$.
Induction Hypothesis: $OPT(x', y') \leq cut(x', y')\ \forall x' \leq x, y' \leq y$.
To prove: $OPT(x + 1, y) \leq cut(x + 1, y)$. Let us consider the optimum solution. It is true that there exist an i such that the piece given by dimensions $(x + 1) \times y$ is cut horizontally or vertically. This says that $OPT(x + 1, y) = OPT(i, y) + OPT(x + 1 - i, y)$(when cut horizontally) or $OPT(x + 1, y) = OPT(x + 1, i) + OPT(x + 1, y - i)$(when cut vertically). By induction hypothesis $OPT(x', y') \leq cut(x', y')$ for all $x' \leq x$ and $y' \leq y$. This implies $OPT(x+1, y) \leq cut(x+1, y)$. A similar argument would give $OPT(x, y+1) \leq cut(x, y+1)$. This completes the proof.

Common errors. Some solutions used recursion. Some solutions used a mix of recursion and storing values; these solutions got full credit, although a fully dp solution is preferred. Many people did not rotate products. Some people tried cutting various products without trying every possible cut.