CS 180 HW 4

Narek Daduryan

December 1, 2021

1

(a)

Solution. Let $n = 5$, and
Let $x = \{10, 9, 10, 9, 10\}$, and
Let $s = \{8, 1, 1, 1, 1\}$.
Then, the optimal solution is to reboot on day 2 and day 4. This will have a total of $8+8+8 = 24$ terabytes of data, which is more than any other rebooting choices.

□

(b)

Give an efficient algorithm that takes values for $x_1, x_2, ..., x_n$ and $s_1, s_2, ..., s_n$ and returns the total number of terabytes processed by an optimal solution.

Solution. Use a dynamic programming approach. First, let's define OPT(i,j) to be the optimal number of terabytes over $x_i, x_{i+1}, ..., x_n$, using the given sequence s, starting at s_j. (Eg for the given example, OPT(2,0) is the optimal solution of x_2, x_3 where at x_2 we have s_0, and at x_3 we have s_1, and so on.

We are going to use a recursive approach by *pre-pending* elements to our x sequence. Let's do this by recognizing the following decision: given a sequence of work x, we can either
(1) Reboot at x_0 or,
(2) Not reboot at x_0
Now let's define the recurrence relation of OPT(i,j) based on the two choices above:
(1) If we reboot at x_i, then we simply take the optimal solution of the rest of the sequence x, but we reset out sequence s due to the reboot. In other words,

$$\text{OPT}(i, j) = \text{OPT}(i + 1, 0)$$

(2) If we don't reboot at x_i, then we still take the optimal solution of the rest of the sequence x, but we don't reset the sequence s. Additionally, we add on the amount of work done of x_i, which is the value of x_i, but clamped by the value of s_j. In other words,

$$\text{OPT}(i, j) = \text{OPT}(i + 1, j + 1) + \min(x_i, s_j)$$

If we analyze the recurrence relation, notice the value of OPT(i,j) depends only on the value of OPT(i+1, _); meaning, if we visualize the 2D input set of OPT(i,j), where values of i represent column, then the value of one column depends only on the next column. Thus, if we use dynamic programming from right-to-left (increasing values of i down to 0), then we will not recurse at all. We can iterate over values of j (ie the rows) in any order, say from bottom-to-top. When we reach OPT(0,0), this is our final solution.

```
function solve(x, s, n):
    opt = 2d array [n][n] initialized to 0;

    for i from n-1 to 0:
        for j from 0 to n-1:
            opt[i][j] = max(
                // if we reboot at x_i
                opt[i+1][0],
                // if we don't reboot at x_i
                opt[i+1][j+1] + min(x[i], s[j])
            )

    return opt[0][0]
```

Note: this is a simplification of the algorithm. I assume that the "opt" array will return 0 for any index out-of-bounds. In a real algorithm, we would have to do bounds-checking and let opt[i][j] be zero if the index is out of bounds.

Since we have no recursion here due to dynamic programming, and this is a 2-d dynamic programming on n, then the run time of this algorithm is $O(n^2)$. Also note that this could be made slightly more efficient by skipping over some redundant values in the 2d array, such as opt[0][2], which is redundant since we cannot use that value anyways. Regardless, this will not change the runtime complexity.

$\square$

2

Let $G = (V, E)$ be an undirected graph and each edge $e \in E$ is associated with a positive weight l_e. Assume weights are distinct. Prove or disprove:

Solution. Recognize that this is a similar problem to the minimum cost to make one string into another. We can use a 2D dynamic programming.

Let OPT(i,j) be the optimal solution to making the string $s_i, ...s_j$ a palindrome. Thus, our final solution for a string s is OPT(1, n) where n is the length of s.

Then, let's find the recurrence relation of OPT to define it. For a palindrome, we can work "outside-in" to determine if the characters from each end of the string are equal. There are two possibilities for OPT(i,j):
(1) $s_i = s_j$
(2) $s_i \neq s_j$

In case (1), we then move "inwards" to test the string $s_{i+1}, ..., s_{j-1}$. In other words, we return OPT(i+1, j-1). In case (2), we try to do an insertion on either end, and then move inwards. Here are the two recurrence relations:
(1) $\text{OPT}(i, j) = \text{OPT}(i + 1, j - 1)$
(2) $\text{OPT}(i, j) = \min(\text{OPT}(i + 1, j), \text{OPT}(i, j - 1)) + 1$

Notice that if we visualise the 2d array of OPT, then OPT only relies on values to the bottom-right (i+1 or j-1). Thus, we can start our dynamic programming algorithm at the "bottom-right" (i = n, j = n) and move "top-left" so we have every answer cached.

Notice we reduce the sample space by constraining j to start from i.

```

function solve(s, n):
    opt = 2d array [n][n] initialized to infinity;

    for i from n to 1:
        for j from i to n:
            if i == j:
                opt[i][j] = 0
            // left and right chars are equal
            else if s[i] == s[j]:
                opt[i][j] = opt[i+1][j-1]
            else
                opt[i][j] = min(
                    // insert char into left side
                    opt[i+1][j] + 1,
                    // insert char into right side
                    opt[i][j-1] + 1
                )

    return opt[1][n];
```

Since we have a 2d dynamic programming, and we have optimized the path that we iterate so we have every value cached, the runtime of this algorithm is $O(n^2)$ where n is the length of the input string s.

$\square$

3

Solution. Use 2d dynamic programming on $W \times L$

Assume we have a function $\text{OPT}(w, l)$ which gives the optimal (minimum) wasted space on a rectangle of size $w \times l$. Let's define a recurrence relation on this function.

Given a square of size $w \times l$, and the given smaller rectangles $(a_1 \times b_1), ..., (a_k \times b_k)$, we can do the following options:

1. For any smaller rectangle $r = (a_i \times b_i)$ that fits within $(w \times l)$, place r onto the top-left of $(w \times l)$. This will create some remainder that we can use in the following ways:
   (a) split into a larger horizontal piece $(w \times l - b_i)$, and the smaller remainder $(w - a_i \times b_i)$. (See Figure 1)
   (b) split into a larger vertical piece $(w - a_i \times l)$, and the smaller remainder $(a_i \times l - b_i)$. (See Figure 2)
2. If no such rectangle fits, then return $w \times l$ as the wasted space.

Thus, we can have a dynamic programming algorithm as such:

```
function solve(W, L, a, b, k):
    opt = 2d array [W+1][L+1];
        initialize opt[0][x] = 0 and opt[x][0] = 0;

    for x from 1 to W:
      for y from 1 to L:
        // by default, the wasted space is x * y
        opt[x][y] = x * y;

        for i from 1 to k:
          if rectangle i fits in (x * y):
            opt[x][y] = min(opt[x][y],
              min(
                // option (1)
                opt[x][y-b[i]] + opt[x-a[i]][b[i]],
                // option (2)
                opt[x-a[i]][y] + opt[a[i]][y-b[i]]
              )
            )
    return opt[W][L];

doesRectangleFit(a, b, x, y):
    return a <= x && b <= y;
```

The runtime of this will include a $O(WL)$ term since the dynamic programming is over a 2d array of size $W * L$. However, for each element in the 2d dp array, we also do another for loop, iterating over each rectangle (k times). Thus, for each dp entry, we also have $O(k)$ term. Thus, the total runtime of the algorithm is $O(kWL)$

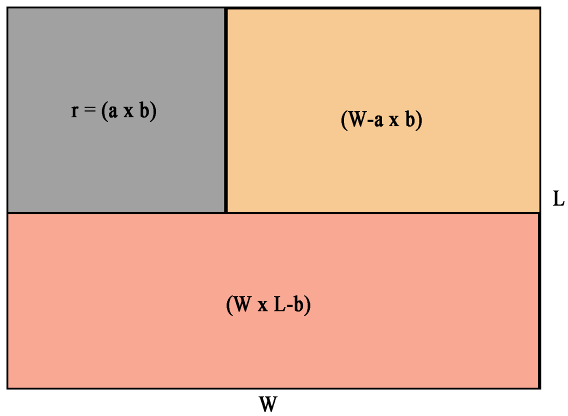

Figure 1: The grey box is rectangle r, and the other two are the remainder

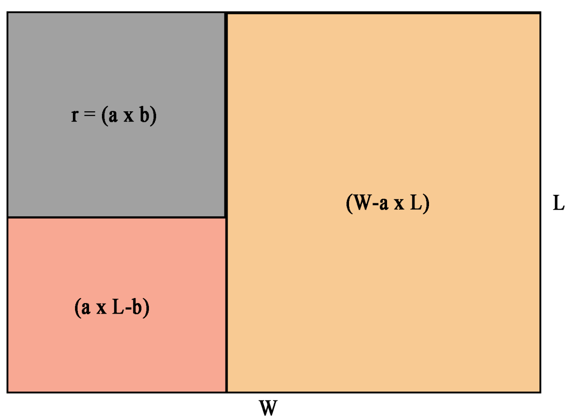

Figure 2: The grey box is rectangle r, and the other two are the remainder

□

4

We now define the Hitting Set Problem as follows. We are given a set $A = \{a_1, ..., a_n\}$, a collection $B = \{B_1, ..., B_m\}$ of subsets of A, and a number K. We are asked: Is there a hitting set $H \subseteq A$ for B such that the size of H is at most k? Prove that the vertex cover problem $\leq_p$ the hitting set problem.

Solution. Recap of the vertex cover problem: We are given a graph $G = (V, E)$, and we want to know the minimum number of nodes $N \subseteq V$ such that $\forall (u, v) \in E$, either $u \in N$ or $v \in N$.

If we can do a polynomial reduction of the vertex cover problem into the Hitting Set problem, then we prove that vertex cover problem $\leq_p$ hitting set problem.

Recap: a polynomial reduction exists when we can use $f(n)$ solvers for Y to solve an instance of X (and we are allowed polynomial-time transformations $g(n)$, and $f(n)$ is polynomial time).

Thus, let's assume we are given an instance of the vertex cover problem. This instance includes the graph $G = (V, E)$. Now, let $A = V$ (A is the set of all vertices of our graph), and let $B = E$ (that is, $\forall (u, v) \in E$, $B_i = \{u, v\}$). Notice now, that a hitting set H of A and B is a set of nodes ($H \subseteq V$) such that H contains at least one element in each B_i, meaning H contains at least one node of each edge in E, meaning that H is a set of nodes such that they are a vertex-cover of G.

Now, we can try for all integers k from 0 to $|E|$, so see if there exists a hitting set on A and B that is $\leq k$. When we find the first one, we return that k (this is the smallest-sized vertex cover).

```
function vertex_cover(V, E):
    for k from 1 to |E|:
        if (hitting_set(V, E, k):
            return k
    return -1
```

Recognize that if V and E are already in the correct formats, we can simply pass them as-is to the "hitting_set" function. If not, we can transform them in polynomial time (loop over each edge in E, and for each edge, add that list to B). Also, note that we have a for-loop that runs for $|E|$ iterations. Thus, we call "hitting_set" polynomial number of times ($O(|E|)$). Thus, since we have a polynomial transformation, and a polynomial number of calls to "hitting_set", we can state that the vertex cover problem is polynomial reducible to the hitting set problem. Thus, by the definition of polynomial reduction,

$$\text{vertex cover problem} \leq_p \text{hitting set problem}$$

$\square$