CS5800 Algorithms

Recitations from Oct 21, 2019 to Oct 25, 2019

Ravi Sundaram

Problem 1 (More Dynamic Programming)

1. An example of where dynamic programming can be used is text editors justifying text in a paragraph so that there's the least amount of unnecessary white space in the whole block.

   Your input is a sequence of n words where word i has length l_i (length is defined through number of characters). You want to print them in a paragraph that has a width of w characters. If a line in the paragraph contains t words with combined length L, then the line has $w - L - (t-1)$ extra spaces. The $t-1$ term corresponds to the spaces between the words. We define the *penalty* of the line as the square of the number of extra spaces. The problem is to justify the words so that the sum of the penalties of the lines is minimized. For example, if $w = 10$ and the words are

   the blue freak

   then the following option

   the blue
   freak

   has penalty $2^2 + 5^2 = 29$.

   (a) Explain with an example why we square the penalty in each line.

   (b) Give a dynamic programming solution that minimizes the sum of the penalties.

2. Let's revisit the All Pairs Shortest Path (APSP) problem. Remember how in Floyd-Warshall algorithm we recursed on the set of 'legal vertices'. At the start, we make some labeling of the vertices $v_1, v_2, ..., v_n$. Our subproblem d_{ij}^k is the shortest path from v_i to v_j such that the path only traverses through vertices $\{v_1, v_2, ..., v_k\}$. When we consider d_{ij}^{k+1}, then we have two cases. Either the shortest path remains unchanged, or there's a shorter path using the vertex v_{k+1} (along with all the old vertices $\{v_1, v_2, ..., v_k\}$). If we are in the second case, then the shorter path can be expressed as $d_{im}^m + d_{mj}^m$. So therefore our recursion (without the base cases) is

   $$d_{ij}^{k+1} = \min(d_{ij}^k, d_{ik}^k + d_{kj}^k)$$

   Now, instead of recursing on the set of 'legal vertices', let's recurse on the set of 'legal edges'. Give an all pairs shortest path dynamic programming solution that recurses on the set of available edges for the path. Make sure to analyze the time complexity.

3. On the weekend Ravi likes to visit the local trampoline park. One of the attractions there is a track to run on with trampolines placed at various spots on the track.

   When you run up to one of trampolines on the track, you have two options. You can either sidestep it and keep running on the track, or you can choose to jump on it. If you choose to jump on it, the trampoline will launch you up in the air and you will land some distance ahead on the track. Fun! Except you might fly over other trampolines.

Ravi, being a professor in computer science, decides he wants to maximize the distance he spends in the air. Help him achieve his goal! Formally, you are given an array $T[1..n]$, where $T[i]$ is the distance of the ith trampoline from the start of the track, and an array $D[1...n]$ where $D[i]$ is the distance traveled in the air if a person jumps on trampoline i.

For clarity, if a person jumps on trampoline i, they travel $D[i]$ distance in the air and they land at $T[i] + D[i]$, where they can resume taking trampolines.

(a) Give a dynamic programming algorithm that returns the maximum distance Ravi can spend in the air. Make sure to analyze the time complexity. If you get $O(n^2)$, good job! Can you think of a way to make it even faster? (Hint: Think preprocessing).

(b) After giving Ravi the optimal jump sequence, he uses it everytime he visits the trampoline park. The Khoury legal team hears about this and decides this is too much of a liability for the college. After intense negotiations, they limit Ravi to k jumps per run. Help Ravi come up with a dynamic programming algorithm that returns the maximum distance he can spend in the air if he jumps on at most k trampolines.

Solution:

1. (a) If the penalty is not squared, then taking a word from the end of one line and placing it at the end of another line doesn't change the penalty. For example,

green goose
green goose

will have the same penalty (suppose the width of each line is 20) as

green
green goose goose

even though the first one is better spaced.

(b)
- English Description of problem: $OPT[i]$ will contain the penalty of the optimal arrangement of the first i words of the the input.
- English Description of logic: When we add the $i + 1$th word into an arrangment, it can go on a line with some of the previous words in the input sequence. So we need to check all possible arrangments of making a new line. The first possible arrangmement is starting with word $i - k$ to word i, where $i - k$ is the smallest index such that words $i-k$, $i-k+1$,..., $i+1$ all fit in the same line. To this penalty we also add $OPT[i-k-1]$ that we already have computed. Then the second possible arrangement is $i - k + 1$, $i - k + 2$,..., $i + 1$, and to the penalty this line creates we add $OPT[i - k]$. We continue in this manner till its just the penalty of word $i + 1$ in a line by itself to which we add $OPT[i]$.

- Recurrence Relation (and Base Case)

$$OPT[i+1] = \begin{cases} 0 & \text{if } i+1=0 \\ \min\limits_{\substack{j \text{ s.t.} \\ l_{j+1}+l_{j+2}+..+l_{i+1}+(i-j)\leq w}} (OPT[j] + P(j+1, i+1)) \end{cases}$$

  Where $P(j+1,i+1)$ is the penalty of the line with words $j+1, j+2, ..., i+1$ formally defined as $(w - (l_{j+1} + l_{j+2} + .. + l_{i+1} - (i-j)))^2$.
- Iterative Algorithm - We have one loop starting at $i = 1$ up to $i = n$. However at each step, we need to loop over all possible arrangements of a new line to find the minimum.
- Number of subproblems -There are $O(n)$ subproblems, one for each word.
- Time per subproblem - We need to find the min over all possible new lines. There are in the worst case a linear number of words per characters in a line, so the number of possible arrangments of a new line is $O(w)$.
- Total Runtime - We get $O(nw)$.
- Final Return Value - $OPT[n]$

2. - English Description of problem: $d[i,j,k]$ will contain the distance of the shortest path between vertices v_i, v_j that uses only edges $e_1, e_2, ..., e_k$.
   - English Description of logic: When we allow an additional edge e_{k+1} for the minimal path between v_i, v_j, the shortest path will either be unchanged or it will contain e_{k+1}. If it's unchanged, then the value remains $d[i,j,k]$. If it is changed, then let $e_{k+1} = (v_a, v_b)$ (directed from v_a to v_b). Then the shortest path is from v_i to v_a, then e_{k+1}, then from v_b to v_j. Note that the shortest path from v_i to v_a using the first k edges will be contained in $d[i,a,k]$, which is something we already have computed.
   - Recurrence Relation (and Base Case)

$$d[i,j,k+1] = \begin{cases} \infty & \text{if } k=0, i\neq j \\ 0 & \text{if } k=0, i=j \\ \min(d[i,j,k], d[i,a,k] + w(e_{k+1}) + d[b,j,k]) & \text{otherwise} \end{cases}$$

     where $w(e_{k+1})$ denotes the weight of edge e_{k+1}.
   - Iterative Algorithm - The outermost loop will be over k, starting from 0 and up to m, where m is the number of edges. The next two loops will be over i, j in any order.
   - Number of subproblems - $m \cdot n \cdot n = mn^2$
   - Time per subproblem - Constant, we just check values precomputed in the memoization array, plus the weight of an edge.
   - Total runtime - $O(mn^2)$
   - Final Return Value - For any pair of vertices i, j, you can return $d[i,j,m]$.

3. (a) - English Description of problem: $Air[l+1]$ will contain the max distance travelled through the air starting from trampoline $l+1$ till the end of the track.
   - English Description of logic: When we are at trampoline l, we can either choose to jump on the trampoline, or we can choose to skip the trampoline and keep running down the track. If we jump on the trampoline, we add $D[l]$ to our total air distance, and then we are at the next trampoline with the minimum j such that $T[j] \geq T[l] + D[l]$. Basically, trampoline j is the earliest trampoline we arrive at after landing back on the track.

- Recurrence Relation (and Base Case)

$$Air[l] = \begin{cases} 0 & \text{if } l > n \\ \max(Air[l+1], D[l] + Air[j]) & \text{where } j \text{ is min index s.t } T[j] \geq D[l] + T[l] \end{cases}$$

- Iterative Algorithm - The loop will be over l, starting from n and down to 0.
- Number of subproblems - n
- Time per subproblem - Naively it's linear to find the min such j. However, we can assume that $T[1...n]$ is sorted (as the trampolines are given to us in order as they come down the track). So when we are at trampoline k, we can compute the sum $T[k]+D[k]$ in constant time, and then we run binary search to find the min such j so that $T[j] \geq T[k] + D[k]$
- Total runtime - $O(n \log n)$
- Final Return Value - $Air[1]$

(b)
- English Description of problem: $Air[l+1, t]$ will contain the max distance travelled through the air using at most t trampolines starting from trampoline $l+1$ and till the end of the track.
- English Description of logic: When we are at trampoline l, we can either choose to jump on the trampoline, or we can choose to skip the trampoline and keep running down the track. If we skip, we are at the next trampoline and we don't do anything to t as we didn't jump. If we jump on the trampoline, we add $D[l]$ to our total air distance, and then we are at the next trampoline with the minimum j such that $T[j] \geq T[k] + D[k]$. We also need to decrement t to note that we used a jump.
- Recurrence Relation (and Base Case)

$$Air[l, t] = \begin{cases} 0 & \text{if } l > n \\ -\infty & \text{if } t < 0 \\ \max(Air[l+1, t], D[l] + Air[j, t-1]) & \text{where } j \text{ is min index s.t } T[j] \geq D[l] + T[l] \end{cases}$$

- Iterative Algorithm - The outer loop is over t, starting at $t = 0$ up to k. The inner loop will be over l, starting from n and down to 0.
- Number of subproblems - nk
- Time per subproblem - We can preprocess the min j such that $T[j] \geq D[l]+T[l]$ for each l before we start the dynamic programming in $n \log n$ time and store it in an auxillary array. Then computing each subproblem will only take constant time.
- Total runtime - $O(n \log n + nk)$
- Final Return Value - $Air[1, k]$

...

■

Problem 2 (Extras...)

1. Alice has to plan her working life for the next T years. There are n different jobs that Alice can do, and every year Alice must do exactly one job. Each job pays different amounts depending on the year. For example, being a sports journalist pays more when there's a FIFA World Cup

happening that year. Formally, during year t a job j pays $P(j, t)$. Give a dynamic programming algorithm for the maximum possible earnings of Alice over the next T years, supposing she can switch jobs at most S times.

Dynamic Programming Solution Guidelines

Dynamic programming can be very tricky and this template will help guide you through solving new problems. Get in the habit of going through the list and filling everything out step by step. We will grade harshly on missing items. And if there's no english description of the two items mentioned below then the solution will get an **AUTOMATIC 0** on homeworks and exams with no exceptions.

1. (**AUTOMATIC 0 IF MISSING**) English description of the
   (a) problem you are actually solving with your recursion.
   (b) logic behind your recursion.
2. The actual recursion. And don't forget the base cases!
3. Brief description of how an iterative algorithm would loop through the memoization array and fill the values (make sure your order makes sense, and that values are already filled when you call them). Pseudocode isn't required, just a couple sentences.
4. The number of subproblems you have to solve.
5. How much time each subproblem takes to solve (usually constant or linear).
6. Final running time.
7. Final value we have to return.