Introduction to Algorithms: 6.006
Massachusetts Institute of Technology April 12, 2011
Professors Erik Demaine, Piotr Indyk, and Manolis Kellis Problem Set 6

Problem Set 6

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Both Part A and Part B questions are due **Friday, April 29** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF format using LaTeX. Your solution to Part B should be a valid Python file which runs from the command line. A template for writing up solutions in LaTeX is available on the course website. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem. See the course website for our full grading policy.

Part A:

Problem 6-1. [10 points] **Placing Parentheses**

You are given an arithmetic expression containing n real numbers and $n-1$ operators, each either $+$ or $\times$. Your goal is to perform the operations in an order that maximizes the value of the expression. That is, insert parentheses into the expression so that its value is maximized.

For example:

- For the expression $6 \times 3 + 2 \times 5$, the optimal ordering is to add the middle numbers first, then perform the multiplications: $(6 \times (3 + 2)) \times 5 = 150$.
- For the expression $0.1 \times 0.1 + 0.1$, the optimal ordering is to perform the multiplication first, then the addition: $(0.1 \times 0.1) + 0.1 = 0.11$.
- For the expression $(-3) \times 3 + 3$, the optimal ordering is $((-3) \times 3) + 3) = -6$.

(a) [3 points] Clearly state the set of subproblems that you will use to solve this problem.

Solution:

This problem is *very* similar to the matrix chain multiplication problem done in lecture.

First, we define some notation. Denote the numbers by $a_1, a_2, \ldots, a_n$, and the operators by $op_1, op_2, \ldots, op_{n-1}$, so the given expression is $a_1\ op_1\ \ldots\ op_{n-1}\ a_n$.

Let $M[i, j]$ be the *maximum* value obtainable from the subexpression beginning at a_i and ending at a_j (i.e., $a_i\ op_i\ \ldots\ op_{j-1}\ a_j$), and let $m[i, j]$ be the *minimum* value obtainable from the subexpression beginning at a_i and ending at a_j.

We must keep track of both the minimum and the maximum because the maximal value of an expression may result from multiplying two negative subexpressions.

(b) [4 points] Write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems.

Solution:

To solve the subexpression $a_a \ldots a_b$, we can split it into two problems at the kth operator, and recursively solve the subexpressions $a_a \ldots a_k$ and $a_{k+1} \ldots a_b$. In doing so, we must consider all combinations of the minimizing and maximizing subproblems. The base cases are $M[i,i] = m[i,i] = a_i$, for all i.

$$\begin{aligned} M[a,b] = \max_{a \le k < b} (\max(&M[a,k] \; op_k \; M[k+1,b], \\ &M[a,k] \; op_k \; m[k+1,b], \\ &m[a,k] \; op_k \; M[k+1,b], \\ &m[a,k] \; op_k \; m[k+1,b])) \end{aligned}$$

$$\begin{aligned} m[a,b] = \min_{a \le k < b} (\min(&M[a,k] \; op_k \; M[k+1,b], \\ &M[a,k] \; op_k \; m[k+1,b], \\ &m[a,k] \; op_k \; M[k+1,b], \\ &m[a,k] \; op_k \; m[k+1,b])) \end{aligned}$$

(c) [3 points] Analyze the running time of your resulting dynamic programming algorithm, including the number of subproblems and the time spent per subproblem.

Solution:

There are $O(n^2)$ subproblems, two for each pair of indices $1 \le a \le b \le n$. The subproblem $M[a,b]$ must consider $O(b-a) = O(n)$ smaller subproblems. Thus the total running time is $O(n^3)$.

Problem 6-2. [15 points] **Text Formatting**

You are given a sequence of words. Each word has a length. Your goal is to split the text up into lines. Each line's ***badness*** is defined as:

$$\left(\left(\sum_{\text{words in line}} \text{length of word} \right) - (\text{target line length}) \right)^2$$

where the target line length is a given value. You want to minimize the total badness of all the lines in your result.

(a) [3 points] Clearly state the set of subproblems that you will use to solve this problem.

Solution: For the position between words i and $i+1$, the minimum total badness that can be achieved for all lines before the line starting at word $i+1$ if a line break is placed between i and $i+1$.

(b) [4 points] Write a recurrence relating the solution of a general subproblem to solutions of smaller subproblems.

Solution: Minimum over $0 \leq j < i$ of the minimal total badness up to that point plus the badness of a line starting at word $j+1$ and going to word i. (This is assuming that word indices start at 1, and the total badness at index 0 in the array is 0.)

(c) [3 points] What's the running time of your resulting dynamic programming algorithm?

Solution: Iterate over n words. Each time, iterate over $O(n)$ subproblems, where calculating the minimal total badness of a subproblem can take $O(n)$ work. So $O(n^3)$ total work.

(d) [5 points] Say you want to minimize the *maximum* badness of any line in the result, instead of minimizing the sum of each line's badness. How would you modify your algorithm?

Solution: The subproblems should be the maximum badness of any line before the line starting at word $i+1$ if there is a line break between word i and word $i+1$. The recurrence iterates over the same words, but for each word it iterates over it takes the maximum of the badness of the new last line, and the previous maximum badness.

Problem 6-3. [10 points] **Dungeons and Dragons and Dynamic Programming**

You are travelling through a dungeon which is represented by an $n \times n$ grid. Each square has a monster to fight or a healing potion. You start in the bottom row, and from any square, you can only move to the following squares:

1. The square immediately above
2. The square that is one up and one to the left (but only if you are not already in the leftmost column)
3. The square that is one up and one to the right (but only if you are not already in the rightmost column)

Each time you move to a square (i, j), your Hit Points change by $\mathrm{HP}(i, j)$. This amount might be positive, if there's a healing potion on that square, or negative, if there's a monster to fight. Your Hit Points start at H and cannot go above this value even if you drink a healing potion. All Hit Point values are integers.

In addition, when you move to a square (i, j), you gain $\mathrm{XP}(i, j)$ Experience Points. Give an algorithm that figures out the sequence of moves from somewhere along the bottom edge to somewhere

along the top edge that maximizes the total number of Experience Points you gain. However, you must do this without ever reaching 0 or fewer Hit Points.

You are free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination. Your Hit Points and Experience Points are affected by your starting square as well as whatever other squares you move to.

Analyze the running time of your algorithm.

Solution: The subproblems are the greatest number of experience points which can be gained while reaching square (i, j) with $1 \leq h \leq H$ hit points remaining.

Start out with $A[0, j, h] = \mathrm{XP}(0, j)$ if $h = \min(H + \mathrm{HP}(0, j), H)$ and $A[0, j, h] = -\infty$ otherwise, for all $0 \leq j < n$. The recurrence is as follows when $1 \leq h < H$ as well as when $h = H$ and $\mathrm{HP}(i, j) \leq 0$:

$$A[i, j, h] = \mathrm{XP}(i, j) + \max(A[i-1, j-1, h'], \text{ if } j > 1$$
$$A[i-1, j, h']$$
$$A[i-1, j+1, h'], \text{ if } j < n)$$

where $h' = h - \mathrm{HP}(i, j)$. (If $h' \leq 0$ then we set $A[i, j, h] = -\infty$.) Since life gain cannot increase Hit Points to more than H, when $h = H$ and $\mathrm{HP}(i, j) > 0$ we need to take the maximum for all values of h' between $H - \mathrm{HP}(i, j)$ and H inclusive.

The running time of this algorithm is $O(n^2 H)$. For each of n^2 positions on the board, we do $O(1)$ work to fill in the maximum experience points for $h < H$, and we do $O(H)$ work to fill in the maximum experience points for $h = H$.

Problem 6-4. [15 points] **Railroads and Gauges**

Professor Martin McFly has been transported back in time to the year 1885, with nothing but his laptop, and needs to make enough money to fix his flying DeLorean and get back to the 80s. He decides to take a job for the railroad company solving a pressing problem that they are having.

There are n towns, each with one of five different gauges (track widths) of railroad tracks. We can call these different gauges A, B, C, D, and E. The railroad company also provides McFly with a map of the proposed railroad connections between the towns, which also contains a number of switching stations where three tracks (possibly of different gauges) get merged together. These switches are the only type available, and you are not allowed to change their setup.

Professor McFly is put in charge of determining what gauge of track to place in each connection (track gauges cannot be changed except at switching stations). The track within each town is in place, and you are therefore unable to change what gauge to use until the first switching station out of the town.

The cost at each switching station can be 0, 1, or 2. If all tracks are the same gauge, the cost is 0. If two are the same and one is different, the cost is 1. If all three gauges are different, the cost is 2. McFlys job is to minimize the cost over all the railroad connections.

One thing that Professor McFly notices is that the map he is given is actually a spanning tree over the graph of all possible connections between the towns. All the vertices in this tree have degree of either 1 (a town) or 3 (a switching station). The vertices that correspond to towns have the gauges set already, and all McFly must do is determine what gauges of track to use for the edges from the switching stations.

You are to help Professor McFly out by giving an algorithm to output the optimal (lowest) cost of the railroad connections, and print the list of what gauge of track each connection is. You must prove your algorithm to be correct, and justify a bound on its running time. Since McFly has no power outlet to plug his laptop into, he needs an algorithm that runs as efficiently as possible.

Solution: To facilitate ordering of the computation, choose an arbitrary root (any town), and represented the entire graph as a binary tree with that root, and towns at the leaves.

With this rooted tree layout, at every switching station, we make a gauge assignment decision for the parent edge based on the current assignments for the children edges. Every edge can be labeled A, B, C, D, or E, and each of these labelings is associated with a cost. At the leaves (towns), if a town has gauge A, the edge leaving from that town has cost 0 for A, and cost infinity for all other labelings, and similar for B, C, D, and E. At every switching station (internal node), we can update the cost of each labeling for the parent edge, based on the previously computed costs for the children edges. This computation can be made locally, saving a pointer to the choice of children that gave us the optimal value for each labeling. Once these local choices are propagated to the root, a global choice can be made, and each edge assignment that was required to reach that optimum can be propagated downward towards the leaves.

The local update is:

- $\text{cost}[\text{parent}, A] = \min_{i,j}(\text{cost}[\text{left}, i] + \text{cost}[\text{right}, j] + \text{diffs}(A, I, j))$
- $\text{pointer}[\text{parent}, A] =$ the (i, j) pair minimizing the above expression

Note:

- This fills in a table for cost[parent, A] by looking up the costs of left and right, and the cost of switching.
- The min is taken over all gauge labels i, j in {A,B,C,D,E} for the left and right edges.
- cost[parent, B] and so on are computed similarly.
- diffs(x,y,z) is the cost function for a switching station (0 if all same, 1 if only one differs, 2 if three different gauge values meet at that switching station.

Time is $25 * n$ for each gauge at each node, so running time is $5 * 25 * n = O(n)$.

Part B:

Problem 6-5. [50 points] **Website Rankings**

In class, you saw that the length of the longest common subsequence can be computed in $O(n^2)$ time for two strings x and y of length n, using dynamic programming. In this problem, you will learn how to compute it in $O(n \log n)$ time for non-repetitive strings.

Theory part (25 points). Definitions for three sequences $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_m)$, and $z = (z_1, \ldots, z_n)$:

- We say that $t = (t_1, \ldots, t_k)$ is a ***subsequence of z*** if there is a sequence $(i_1, \ldots, i_k)$ of k indices such that $i_1 < i_2 < \ldots < i_k$, and for all j, $t_j = z_{i_j}$.
  Example: $(1, 3, 4)$ and $(3, 2, 6, 5)$ are both subsequences of $(1, 3, 2, 6, 4, 5)$.
- We write $\mathrm{LIS}(z)$ to denote the length of the longest *increasing* subsequence of z.
  Example: $\mathrm{LIS}(\,(6, 2, 7, 5, 3, 4)\,) = 3$, and this corresponds to the subsequence $(2, 3, 4)$.
- We write $\mathrm{LCS}(x, y)$ to denote the length of the longest common subsequence of x and y.
  Example: $\mathrm{LCS}(\,(6, 2, 7, 5, 3, 4),\ (5, 6, 1, 7, 3, 4)\,) = 4$, and this corresponds to the subsequence $(6, 7, 3, 4)$.
- We say that z is ***non-repetitive*** if no integer appears twice in it.
  Example: $(5, 6, 1, 7, 3, 4)$ *is* non-repetitive, and $(4, 6, 1, 7, 1, 3)$ is *not*.

Design algorithms for the following two problems:

1. Show how to compute $\mathrm{LIS}(z)$ for a sequence z of n integers in $O(n \log n)$ time.
   Hint 1: Use the following sequence of arrays A_i. Let $A_i[j]$, where $i, j \in \{1, 2, \ldots, n\}$, be the lowest integer that ends an increasing length-j subsequence of $(z_1, \ldots, z_i)$. If $(z_1, \ldots, z_i)$ has no increasing subsequence of length j, then $A_i[j] = \infty$.
   Hint 2: How can A_i be turned into A_{i+1} in $O(\log n)$ time? How can $\mathrm{LIS}(z)$ be extracted from A_n?
2. Show how to compute $\mathrm{LCS}(x, y)$ for two non-repetitive integer sequences x and y of length n in $O(n \log n)$ time.
   Hint 1: Reduce to the previous problem.
   Hint 2: Create a sequence z from y in the following way. Remove all integers from y that do not appear in x, and replace the other ones by their index in x. How is $\mathrm{LIS}(z)$ related to $\mathrm{LCS}(x, y)$?

Coding part (25 points). Consider two web search engines A and B. We send the same query, say "`6.006`", to both search engines, and in reply we get a ranking of the first k pages according to each of them. How can we measure the similarity of these two rankings? Various methods for this have been designed, but here, we'll use the simplest of them: LCS, the length of the longest common subsequence of the rankings.

Your task is to write a function `longest_common_subsequence()` that uses the above $O(n \log n)$ algorithm to compute LCS for two rankings. Your function should take two lists of

URLs as its arguments, and should return their LCS as an integer. We assume that pages are identical only if their URLs are identical. No URL appears twice in a ranking.

Sample Input

```
http://courses.csail.mit.edu/6.006/spring08/
http://courses.csail.mit.edu/6.006/fall07/
http://www.eecs.mit.edu/ug/newcurriculum/6006blurb.pdf
http://alg.csail.mit.edu/
http://courses.csail.mit.edu/6.006/fall09/

http://courses.csail.mit.edu/6.006/fall09/
http://courses.csail.mit.edu/6.006/spring08/
http://mit.worldcat.org/profiles/MITLibraries/lists/899062
http://courses.csail.mit.edu/6.006/fall07/
http://www.eecs.mit.edu/ug/newcurriculum/6006blurb.pdf
```

Sample Output

```
3
```