CSCI-GA.1170-001/002 Fundamental Algorithms November 19, 2015

Solutions to Problem 1 of Homework 8 (6 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 17*

Using dynamic programming, find the optimum printing of the text *"Not all those who wander are lost"*, i.e. $\ell_1 = 3, \ell_2 = 3, \ell_3 = 5, \ell_4 = 3, \ell_5 = 6, \ell_6 = 3, \ell_7 = 4$, with line length $L = 14$ and penalty function $P(x) = x^3$. Will the optimal printing you get be consistent with the strategy "print the word on as long as it fits, and otherwise start a new line"? Once again, you have to actually find the alignment, as opposed to only finding its penalty.

Solution:

Following the algorithm given in the notes, we maintain two arrays m and s such that $m(i)$ denotes the total badness of the text $l_1, \ldots, l_i$. Initialize $m(0) = 0$. As long as $l_1, \ldots, l_i$ fit in one line, set $m(i)$ to be the badness of that line. Loop on i going up to n. For a given i let k range over $i, i-1, \ldots$ as long as $l_k, \ldots, l_i$ fits in one line. For each of those k values calculate $m(k-1)$ plus the badness of the line $l_k, \ldots, l_i$. Pick the k that gives the smallest sum and set $m(i)$ equal to that sum. We can also $s(i) = k$ which means that in the optimal splitting of text $l_1, \ldots, l_i$ the last line starts with l_k

m(1), s(1)

l_1 fits in the line. So $m(1) = (14-3)^3 = 1131$ and $s(1) = 1$

m(2), s(2)

l_1, l_2 fit in the line. So $m(2) = (14-(3+3+1))^3 = 343$ and $s(2) = 1$

m(3), s(3)

l_1, l_2, l_3 fit in the line. So $m(3) = (14-(3+3+5+2))^3 = 1$ and $s(3) = 1$

m(4), s(4)

$$
\begin{aligned}
k = 4 &\implies P = (14-3)^3 = 1131 \\
k = 3 &\implies P = (14-(3+5+1))^3 = 125 \\
k = 2 &\implies P = (14-(3+5+3+2))^3 = 1 \\
m(4) &= Min\{m(3)+1131, m(2)+125, m(1)+1\} \\
m(4) &= m(2)+125 \\
m(4) &= 468 \\
\implies s(4) &= 3
\end{aligned}
$$

m(5), s(5)

$$
\begin{aligned}
k = 5 &\implies P = (14 - (6))^3 = 512 \\
k = 4 &\implies P - (14 - (3 + 6 + 1))^3 = 64 \\
m(5) &= Minm(4) + 512, m(3) + 64 \\
m(5) &= m(3) + 64 \\
m(5) &= 65 \\
\implies s(5) &= 4
\end{aligned}
$$

m(6), s(6)

$$
\begin{aligned}
k = 6 &\implies P = (14 - 3)^3 = 1131 \\
k = 5 &\implies P = (14 - (6 + 3 + 1))^3 = 64 \\
k = 4 &\implies P = (14 - (3 + 6 + 3 + 2))^3 = 0 \\
m(6) &= Min\{m(5) + 1131, m(4) + 64, m(3) + 0\} \\
m(6) &= m(3) + 0 \\
m(6) &= 1 \\
\implies s(6) &= 4
\end{aligned}
$$

m(7), s(7)

$$
\begin{aligned}
k = 7 &\implies P = (14 - 4)^3 = 1000 \\
k = 6 &\implies P = (14 - (3 + 4 + 1))^3 = 216 \\
m(7) &= Min\{m(6) + 1000, m(5) + 216\} \\
m(7) &= m(5) + 216 \\
m(7) &= 281 \\
\implies s(7) &= 6
\end{aligned}
$$

$s(7) = 6 \implies$ in the optimal splitting of the given text, the last line starts with l_6 which is *"are"*. Therefore this line is *"are lost"*

$s(5) = 4 \implies$ in the optimal splitting the line ending with *"wander"* starts with *"who"*. Therefore this line is *"who wander"*

$s(3) = 1 \implies$ in the optimal splitting the line ending with *"those"* starts with *"Not"*. Therefore this line is *"Not all those"*

Therefore in the optimal splitting, the first line contains the text *"Not all those"*. Second line contains the text *"who wander"*. Third line contains the text *"are lost"*. The penalty turns out to be 281. If we used the greedy strategy, the first line contains the text *"Not all those"*. Second line contains the text *"who wander are"*. Third line contains the text *"lost"*. The penalty in this case is 1132

Therefore the optimal printing derived using dynamic programming is not consistent with the greedy strategy □

CSCI-GA.1170-001/002 Fundamental Algorithms November 19, 2015

Solutions to Problem 2 of Homework 8 (12 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 17*

(a) (8 points) You are given n integers $a_1, \ldots, a_n \geq 0$, and a target $T \geq 0$. Design an $O(nT)$ algorithm to determine is there exists a subset of the a_i's that sum to T. For example, if $n = 5$ and $a_1 = 3$, $a_2 = 5$, $a_3 = 2$, $a_4 = 11$, $a_5 = 3$, then the answer is YES for $T = 10$ (e.g., $3 + 5 + 2 = 10$), but NO for $T = 9$.

Solution:

Define a 2d-array S of size nT such that $S[i, T]$ is *true* if there is a subset of $a_k's$ in the set of integers $\{a_1, \ldots, a_i\}$ such that their sum evaluates to T and $S[i, T]$ is false otherwise.

- $S[i, 0] = true,\ \forall i$. We can choose the empty set $\emptyset$ for the sum to evaluate to 0
- $S[0, T] = false,\ \forall T$. The sum of 0 number of elements can never evaluate to T
- There can be a subset of $a_1, \ldots, a_i$ adding up to T if there is a subset of $a_1, \ldots, a_{i-1}$ adding up to T or if there is a subset of $a_1, \ldots, a_{i-1}$ adding up to $T - a_i$

Therefore from the above three cases we have the following recurrence equation

$$S[i, T] = \begin{cases} true & \text{if } T = 0 \\ false & \text{if } i = 0 \\ S[i-1, T] \vee S[i-1, T - a_i] & \text{otherwise} \end{cases}$$

□

The first row and the first column of the matrix are filled using the two base cases and then the entire matrix is constructed iteratively. Time taken to fill each of entry matrix takes only $O(1)$ time as it involves taking *or* between two booleans. Therefore the total time taken to fill S is $O(nT)$ and in the end $S[n, T]$ is returned.

(b) (4 points) Solve part (a) using only T *bits* of extra memory (in addition to the a_i's themselves).

Solution:

Define a boolean array B of size $T + 1$ and initialize all it's entries to *false*.

$B[k]$ is YES if there exists a subset of a_i's whose sum evaluates to k. Therefore $B[0] = 0$, since the empty set evaluates to sum 0.

If $a_i \leq T$ then $B[a_i] = true$ i.e choose the single element a_i as the subset, so the sum evaluates to a_i

Now for all a_i, starting from 0 traverse up to T and if $B[j]$ is true then set $B[j + a[i]]$ to *true* i.e if there is a subset of elements such that their sum evaluates to j then add $a[i]$ to these subset so that the sum then evaluates to $j + a[i]$

```
1  Algorithm: SubsetSum(n, T)
2  B ← NewArray(T + 1)
3  for i ← 0 to T do
4  |  B[i] ← false
5  end
6  for i ← 1 to n do
7  |  if a[i] ≤ T then
8  |  |  B[a[i]] ← true
9  |  end
10 end
11 for i ← 1 to n do
12 |  for j ← 0 to T do
13 |  |  if B[j] ← true then
14 |  |  |  B[j + a[i]] ← true
15 |  |  end
16 |  end
17 end
18 Return B[T]
```

Algorithm 1: Dynamic Programming Algorithm to determine if there exists a subset of a_i's whose sum evaluates to T

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 19, 2015

Solutions to Problem 3 of Homework 8 (8 (+7) Points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 17*

Imagine a unary alphabet with a single letter x. A (valid) *bracketing* B is a string over three symbols $x, (,)$ defined recursively as follows: (1) a single letter x is a bracketing, and (2) for any $k \geq 2$, if $B_1, \ldots, B_k$ are (valid) bracketings, then so is $B = (B_1 B_2 \ldots B_k)$. A bracketing B is called *binary* if rule (2) can only be applied with $k = 2$. Then the length n of B is the number of x's it has (i.e., one ignores the parenthesis).

For example, there are 11 possible bracketings of length $n = 4$: $(xxxx)$, $((xx)xx)$, $((xxx)x)$, $(x(xxx))$, $(x(xx)x)$, $(xx(xx))$, $((xx)(xx))$, $(x(x(xx)))$, $((x(xx))x)$, $(x((xx)x))$, $(((xx)x)x)$, of which *only the last five* are binary.

(a) (4 points) Let $b(n)$ denote the number of binary bracketings of length n. Show that $b(n)$ is given by the following recurrence:

$$b(n) = \sum_{i=1}^{n-1} b(i)b(n-i) \ .$$

Solution:

Divide the given string into two partitions. The length of first partition being i and the second partition being $n - i$. Clearly i can vary from 1 to $n - i$. Therefore the number of binary bracketings in the first partition is $b(i)$ and the second partition is $b(n - i)$

If the length is 1 then there is only one valid bracketing possible i.e x. Therefore $b(1) = 1$

Let α be one of binary bracketing in the first partition out of the $b(i)$ possibilities. We can choose any of the $b(n - i)$ bracketings from the second partition along with α to get a binary bracketing of length n. Therefore using this specific partition, the number of binary bracketings of length n will be $b(i)b(n - i)$. Also i can vary from 1 to $n - 1$. Thus $b(n)$ evaluates to

$$b(n) = \sum_{i=1}^{n-1} b(i)b(n-i)$$

□

(b) (4 points) Use the result from part (a) to give a dynamic programming algorithm to compute $b(n)$ given n as input. What is the running time of your algorithm? Assume that multiplication of two integers takes time $O(1)$.

Solution:

```
1 Algorithm: b(n)
2 b[1] ← 1
3 for i ← 2 to n do
4 |   for j ← 1 to i-1 do
5 |   |   b[i] ← b[j]b[i − j]
6 |   end
7 end
8 Return b[n]
```

Algorithm 2: Dynamic Programming Algorithm to calculate the number of binary bracketings in $O(n^2)$ time

From the above two *for* loops, it is clear that the above algorithm takes $1 + 2 + \ldots + n - 1 = O(n^2)$ time. □

(c) (7 points (**Extra credit**)) Generalize part (a) and (b) by giving a similar recurrence(with proof) as part (a) to find the total number $f(n)$ of bracketings of length n, and then give a dynamic programming algorithm to compute $f(n)$ and analyze its running time.

Solution:

Given a string, $(x \ldots x)$ of length n

- If the length of the string is 1, then there is only one possible bracketing (x). Therefore $f(1) = 1$
- Otherwise split the string into some p number of partitions such that $(x \ldots x) = (\underbrace{(x \ldots x)}_{\text{i}} \underbrace{(x \ldots x)}_{\text{j}} \underbrace{(x \ldots x)}_{\text{k}} \ldots\ldots \underbrace{(x \ldots x)}_{\text{p}})$
  where $(1 \leq i \leq n), (1 \leq j \leq n-i), (1 \leq k \leq n-i-j), \ldots, (1 \leq p \leq n-i-j-\ldots-p-1)$ then the number of bracketings in this case will be

$$f(n) = \sum_{i=1}^{n} f(i) \sum_{j=1}^{n-1} f(j) \sum_{k=1}^{n-i-j} f(k) \ldots\ldots$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 19, 2015

Solutions to Problem 4 of Homework 8 (8 (+3) points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, November 17*

Let $\star : \{1, \ldots, k\} \times \{1, \ldots, k\} \mapsto \{1, \ldots, k\}$ be a binary operation. Below we assume the values of $a \star b$ for $a, b \in \{1, \ldots, k\}$ are stored in some $k \times k$ array M such that $M[a][b] = a \star b$. Consider the problem of examining a string $x = x_1 x_2 \ldots x_n$, where each $x_i \in \{1, \ldots, k\}$, and deciding whether or not it is possible to parenthesize the expression $x_1 \star x_2 \star \ldots \star x_n$ in such a way that the value of the resulting expression is a given target element $t \in \{1, \ldots, k\}$. Notice, the multiplication table is neither commutative or associative, so the order of multiplication matters (and, hence, the result of the expression is not even well defined unless a complete "parenthesization" is specified). For example, consider the following multiplication table and the string $x = 2221$.

Table 1: Multiplication table

	1	2	3
1	1	3	3
2	1	1	2
3	3	3	3

Parenthesizing it $(2 \star 2) \star (2 \star 1)$ gives $t = 1$, but $((2 \star 2) \star 2) \star 1$ gives $t = 3$. On the other hand, no possible parenthesization gives $t = 2$ (you may check this).

(a) (8 points) Assume you are given as input the following: n, k, t, $x[1 \ldots n]$ and M. Give a dynamic programming algorithm that runs in time polynomial in n and k and outputs YES if there exists a paranthesization for x that results in the product equal to t, and NO otherwise. For instance, in the above example with $x = 2221$, the answer is YES if $t = 1$ or $t = 3$, but NO if $t = 2$.

Solution:

Define a 3d-array R and initialize it to 0 such that $R[i, j, k]$ denotes the number of paranthesizations of $x_i \star x_{i+1} \star \ldots \star x_j$ such that the product evaluates to k.

Therefore, the base case is $R[i, i, x_i] = 1$ i.e the number of paranthesizations of x_i such that the product evaluates to x_i is 1

Otherwise, if $i < j$, let $p \in \{1, \ldots, k\}$ and let $S_p = \{(\alpha_y^p, \beta_y^p)\}$ where $1 \leq \alpha_y^p \leq k$, $1 \leq \beta_y^p \leq k$ and $|S_p| \leq k$ be the set of all tuples such that $\alpha_y^p \star \beta_y^p = p$.

Then for any given expression $(x_i \star x_{i+1} \star \ldots \star x_j)$, we can divide it into two partitions $(x_i \star \ldots \star x_t)$ and $(x_{t+1} \star \ldots \star x_j)$ where $i \leq t < j$.

If $x_i \star x_{i+1} \star \ldots \star x_j = p \implies (x_i \star \ldots \star x_t) = \alpha_y^p$ and $(x_{t+1} \star \ldots \star x_j) = \beta_y^p$.

The number of ways in which α_y^p can be obtained from $(x_i \star \ldots \star x_t)$ using different paranthesizations is $R[i, t, \alpha_y^p]$.

Similarly, the number of ways in which β_y^p can be obtained from $(x_{t+1} \star \ldots \star x_j)$ using different paranthesizations is $R[t+1, j, \beta_y^p]$

Note that t ranges from i to $j-1$ and y ranges from 1 to $|S_p|$. Therefore, the total number of ways to obtain p from $(x_i \star x_{i+1} \star \ldots \star x_j)$ using different paranthesizations is

$$R[i, j, p] = \sum_{y=1}^{|S_p|} \sum_{t=i}^{j-1} R[i, t, \alpha_y^p] R[t+1, j, \beta_y^p]$$

Using the above recurrence equation and the base cases fill all the entries of R and at then end the answer is YES if $R[1, n, t] > 0$ and NO otherwise

□

(b) (3 points (**Extra credit**)) Analyze the running time of your algorithm.

Solution:

From the above recurrence relation, it is clear that evaluating $R[i, j, p]$ i.e one entry of the 3d matrix R takes $(j-i)|S_p|$ steps. Now we have k number of 2d arrays of size $n \times n$. Evaluating some p^{th} matrix out of these k number of 2d array takes time $O(n^3)|S_p|$ time where $1 \leq p \leq k$

Therefore total time taken to calculate all the k number of 2d arrays i.e a 3d array R of size $n \times n \times k$ will be $O(n^3)(|S_1| + \ldots + |S_k|)$. Time taken to constructs the sets $S_1, \ldots, S_k$ is $O(k^2)$ as we need to traverse the multiplication table of size k^2 and therefore $|S_1| + \ldots + |S_k| = k^2$.

Therefore the running time of the algorithm is $O(n^3 k^2)$ □