CS 5114
Solutions to Homework Assignment 4

Meghendra Singh

February 24, 2017

[20] 1. CLR Problem 15-4. ***Printing neatly***

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, ..., l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of neatness is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

State the implicit optimization problem formally. Follow the dynamic programming paradigm given in class to solve the optimization problem. In particular:

A. Carefully identify the subproblems that will be solved. How are they parameterized? How many subproblems are there? What information will your algorithm store for each subproblem?

B. What are the base cases? How do you compute the information for the base cases?

C. How do you compute the information for the general (non-base) case?

Once you have answered the questions in the paradigm, write your algorithm in CLR-style pseudocode. Analyze its space and time complexities. Your algorithm need only solve for the optimal value of the objective function; it does not have to "print" the paragraph as is suggested in the text of the problem.

You may be aware that TEX uses dynamic programming to build "optimal" paragraphs and pages.

The implicit optimization problem for ***Printing neatly*** can be formally written as an INSTANCE and a SOLUTION:

INSTANCE: A sequence S of n words with lengths l in terms of characters for the words in S. Length of the i^{th} word is equal to l_i. $\therefore S$ and l can be defined as:
$S = 1, 2, 3, ..., n$ and $l = \{l_1, l_2, l_3, ..., l_n\}$

SOLUTION: Some partitioning of S into p-lines with a maximum of M characters in each line, such that $\sum_{k=1}^{p-1} w_k^3$ is minimized. Here, w_k is the number of extra space characters at the end of the k^{th} line.

If there are j words on a line k, having a total character length of J, there will be 1 space character between any two words, resulting in total number of "useful" characters on line k are $J+j-1$. Since, any line can only contain a maximum of M characters, the number of extra space characters on line k are $M-(J+j-1)$. This can also be generalized for any line containing words i to j as we will see in the subproblems description below. Using the dynamic programming paradigm, first we need to identify the subproblems, the solutions to which will enable us solve the problem described above. For this, lets assume a general line L, containing the contiguous sequence of words from i to j, i.e. $i, i+1, i+2, ..., j$ $\forall 1 \leq i \leq j \leq n$. Let $c[i,j]$ be the cost for printing line L, defined as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } j = n \\ [M - (j - i + \sum_{k=i}^{j} l_k)]^3 & \text{if } j \neq n \ \& \ M \geq (j - i + \sum_{k=i}^{j} l_k) \\ \infty & \text{if } j \neq n \ \& \ M < (j - i + \sum_{k=i}^{j} l_k) \end{cases}$$

The three cases above represent three possible conditions on line L. The first case specifies that j is the last word present in L and is also the last word of the entire input text sequence S. Since, the last word of the input would "always" lie on the last line, this implies that L is the last line, and its cost should not be considered in the optimization problem. Hence, $c[i,j]$ for $j = n$ is specified as 0. The second case basically means that $c[i,j]$ is the cube of the difference between the maximum possible characters in L and the total number of "useful" characters in L, if L is not the last line and all the "useful" characters can be accommodated in line L, i.e the maximum characters per line constraint of M is preserved. The third case is the opposite of the second case, it specifies that if the maximum characters per line constraint of M is violated, the line is extremely expensive (∞ cost).

SUBPROBLEMS: Let the cost of printing a paragraph till the line L (as discussed above) be $p[j]$ (as j is the last word of L). This basically represents the cost of printing the paragraph till the last word of L, which is j. So, we can define our problem using the following recurrence:

$$p[j] = p[i-1] + c[i,j] \tag{1}$$

Here, we assume that line L begins at the word i and ends at the word j. So the cost of printing the paragraph till L could be written as a sum of the cost of printing till line $L-1$, i.e. $p[i-1]$ and the cost of printing line L, i.e. $c[i,j]$. Using (1), we can now formulate the optimization problem in terms of our subproblems as follows:

$$p[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j}(p[i-1] + c[i,j]) & \text{if } j > 0 \end{cases}$$

The base case occurs when $j = 0$, i.e. there is no text to print, in which case the optimal cost is zero. The general case basically optimizes over the possible partitions for the sequence $i, ..., j$. We can think of this as keeping all the words from i to j in the first line (when $i = 1$), then moving the last word (i.e. j) to the second line, while keeping the remaining words (i.e. i to $j - 1$) in the first line, and so on. This will continue until we fill the j^{th} line with the last word (i.e. j) and each line from 1 to j has one word each (this happens when $i = j$, and we will have a total of j partitions). The optimization will then pick the partition that has the minimum cost of printing. Using the general case we can estimate the number of subproblems that will be generated for when $j = n$. For $j = n$, i would take the values $1, 2, 3, ..., n$, and for each value of i, we will have i subproblems. So the total number of subproblems will be the sum: $1+2+3+...+n$. This is the sum of first n natural numbers, which can be computed as: $\frac{n(n-1)}{2} \approx n^2$ subproblems. The pseudocode for the algorithm PRINT-NEATLY is as follows:

PRINT-NEATLY(l, n, M)

```
 1  // initialize the line cost and paragraph cost arrays
 2  let c[1...n, 1...n] and p[0...n] be two empty arrays
 3  // compute the line costs c[i, j]
 4  for i = 1 to n
 5      for j = 1 to n
 6          s = 0
 7          for k = i to j
 8              s = s + l_k
 9          s = s + j − i
10          if j == n
11              c[i, j] = 0
12          else if M ≥ s
13                  c[i, j] = (M − s)^3
14          else if M < s
15                  c[i, j] = ∞
16  // compute the paragraph costs
17  p[0] = 0
18  for j = 1 to n
19      p[j] = p[0] + c[1, j]
20      for i = 2 to j
21          if (p[i − 1] + c[i, j]) < p[j]
22              p[j] = p[i + 1] + c[i, j]
23  // return the optimal cost of the paragraph till the last word
24  return p[n]
```

The time complexity for PRINT-NEATLY can be computed by looking at the number of times each of its statements would execute. We see that in the worst line 8 would execute n^3 times (This happens when $i = 1$ and $j = n$). Similarly, lines 21 and 22 would execute n^2 times in the worst case. This will result in a worst case runtime complexity for PRINT-NEATLY to be $\Theta(n^3)$. Also, PRINT-NEATLY maintains the 2-dimensional array $c[1...n, 1...n]$ and the

1-dimensional array $p[1...n]$. Since, c is the largest data structure maintained, this will lead to a space complexity for PRINT-NEATLY to be $\Theta(n^2)$.

---

[20] 2. CLRS Exercise 16.2-4.
Professor GEKKO has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana.The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations. The professors goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

Let GEKKO be the computational problem described in the exercise. Give a formal statement of the problem. Give the pseudocode of your greedy algorithm to solve it. Analyze the time complexity. Prove the optimality of your algorithm using a substitution argument.

The formal statement for the GEKKO problem, written as an INSTANCE and a SOLUTION is as follows:

INSTANCE: Sequence L of n refilling locations $1, 2, ..., n$ including the starting point and the destination. An array D of length n, with each $D[i]$ containing the distance between locations $i-1$ and i, $D[i]$ is always less than or equal to m miles.

SOLUTION: Sequence S of refiling locations, such that $S \subseteq L$ such that $|S|$ is minimized.
A greedy algorithm for GEKKO would always give a locally optimal solution. Consider that there are i refilling locations between the start location s and $s+m$ miles. A greedy solution G would select the farthest stop that can be travelled without refilling. Therefore, G would select the i^{th} location as the first stop because, i is the farthest point that can be reached from s within the bound of m miles. For each location between s and i G would check if it is possible to not stop at the location and not run out of water. G would not select any of the $s+1$ and $i-1$ as there exists i, which is the farthest location from s, and is still within m miles from s. While checking location i, G would infer that, i is within m miles from s, but $i+1$ is outside this bound, and hence i is the farthest location that should be selected as the new s. The pseudocode of the greedy algorithm form GEKKO problem is as follows:

GEKKO-GREEDY(D, m)

```
1  // n is the total number of refilling locations
2  let S be an empty set
3  n = D.Length
4  d = 0
5  for i = 2 to n − 1
6       if d + D[i] ≤ m and d + D[i + 1] > m
7            S = S ∪ i
8            d = 0
9       else d = d + D[i]
10 return S
```

In GEKKO-GREEDY above, lines 6 through 9 always execute $n-2$ times irrespective of the input. Assuming that all the other lines of GEKKO-GREEDY execute in constant time, the asymptotic time complexity of GEKKO-GREEDY can be written as: $\Theta(n-2) \approx \Theta(n)$.

An Optimal solution O to the GEKKO problem, can be defined as a solution which respects the problem constraint that the maximum distance between any two consecutive water stops is m miles, in such a way that the number of water stop locations selected by O is minimum (i.e. $\nexists S$, such that $|S| < |O|$).

Proof of optimality for GEKKO-GREEDY by substitution argument: Let G, be a greedy solution generated by GEKKO-GREEDY[1], defined as the sequence: $G : l_1, l_2, l_3, ..., l_k$, where each l_i is a water stop (refilling location) present in G, and l_k being the last such refiling location present in G. Also, let O be an optimal solution for the GEKKO problem, defined as the sequence: $O : l_1^{'}, l_2^{'}, l_3^{'}, ..., l_k^{'}$, where each $l_i^{'}$ is a water stop (refilling location) present in O, and $l_k^{'}$ being the last such refiling location present in O. If we substitute the last location in O (i.e. $l_k^{'}$) with the lat location in G (i.e. l_k), we will get a new solution O' that can be written as follows:

$$O' : l_1^{'}, l_2^{'}, l_3^{'}, ..., l_{k-1}^{'}, l_k \tag{2}$$

It can be proved that O' is also an optimal solution, because $|O'| = |O|$, i.e. the number of locations in O' is the same as the number of locations in O, which makes O' an optimal solution by the definition of an optimal solution. If we continue to substitute the last location that differs between O' and G, with the corresponding location in G, we will continue to get new optimal solutions (viz. $O'', O''', ..., O^k$), until we reach one of the following three cases for the substituted solution O^k:

A. $O^k : l_1^{'}, l_2^{'}, l_3^{'}, ..., l_1, l_2, ..., l_k$

B. $O^k : l_1, l_2, l_3, ..., l_k$

C. $O^k : l_i, l_{i+1}, ..., l_k$ $[\forall i \geq 2]$

[1]For the sake of simplicity, I have assumed that the set S returned by GEKKO-GREEDY can be easily converted into the sequence G using the distance array D.

Case A, implies that, $|O| = |O^k| > |G|$, because all the locations present in G were substituted in O^k, but O^k still has some locations in the beginning that were not substituted by locations in G. This basically implies that there are no more locations available in G that can be used to substitute locations in O^k, i.e. G has been entirely "contained" in O^k. This is a contradiction of our base assumption that O and consequently O^k are optimal, as in this case we have $|G| < |O|$. Therefore, **Case A** is not possible. **Case C** implies that $|O| = |O^k| < |G|$ (i.e. there are some locations in the beginning of G, which are left to be substituted into O^k, but there are no more locations available in O^k that can be substituted with locations that remain to be substituted in G). **Case C** also implies that, the locations l_1 to l_{i-1} were not included in O^k, while still satisfying the constraint that professor Gekko, never travelled more than m miles, without taking a stop at a refill location. This implies that the distance between the source and $l_i \leq m$. G by definition would only contain one location between the starting point and m miles from the starting point. **Case C** also suggests that G includes the locations $l_1, ..., l_i$ between the starting point and m miles from the starting point. This violates the definition of G being a greedy solution, and is once again a contradiction to our base assumption that G is a greedy solution. Hence, **Case C** is also not possible. This leaves us with **Case B**, as the only possible case, which also implies that $|O| = |O^k| = |G|$, and hence proves that G is also an optimal solution.

---

[20] 3. CLRS Exercise 16.2-6.
Show how to solve the fractional knapsack problem in $O(n)$ time. Note that you cannot assume anything about the order of the items. Think about the special structure that a greedy solution to a fractional knapsack problem has. Use that to guide the algorithm you design.

We can formally specify FRACTIONAL-KNAPSACK as an INSTANCE and SOLUTION as follows:

> INSTANCE: A set of n items I given by $\{1, 2, 3, ..., n\}$. Integer values corresponding to the n items given by $v_1, v_2, ..., v_n$. Positive weights for the n items given by $w_1, w_2, ..., w_n$. Bound $W \geq 0$ specifies the maximum weight that can be carried in the knapsack.
>
> SOLUTION: Real values $\alpha_1, \alpha_2, ..., \alpha_n$, for the n items, such that $0 \leq \alpha_i \leq 1$ and $\sum_{i=1}^{n} \alpha_i v_i$ is maximized.

In order to obtain $O(n)$ running time, while not making any assumptions about the arrangement of the items (for example, items being sorted on the value of $r_i = \frac{v_i}{w_i}$), we need to think about another approach by which we can ensure that items with with high values of $r_i = \frac{v_i}{w_i}$ are selected, before items with low values of r_i, by the greedy algorithm LINEAR-KNAPSACK, all the while respecting the weight bound of W. If we try to sort the items using r_i values and then apply the FRACTIONAL-KNAPSACK, we will incur a worst case running time of $O(n \lg n)$, which is undesirable for this problem. We can however use the i^{th} order-statistics in linear time, to get the i^{th} largest value of the r_is and use this to somehow break our problem into smaller subproblems. We can specifically use the SELECT with $i = \lfloor \frac{n}{2} \rfloor$, to get the lower-median m of the r_i values in $O(n)$ time and use m as a

criteria to prioritize the selection of high r_i items, thereby maximizing the total value of the knapsack. If we compare m with the r_i values for each of the n items, we can get either of the following three cases:

A. $r_i = m$ i.e. the ratio of value per unit weight for item i is equal to the median.

B. $r_i > m$ i.e. the ratio of value per unit weight for item i is greater than the median.

C. $r_i < m$ i.e. the ratio of value per unit weight for item i is less than the median.

We can use this information to partition our set of items (I) into three subsets: E, H and L, by iterating over items in I once (only requiring $O(n)$ time), these can be defined as follows:

$$i \in E \text{ if } r_i = m$$
$$i \in H \text{ if } r_i > m$$
$$i \in L \text{ if } r_i < m$$

By the above definitions, H would contain approximately half ($\frac{n}{2}$) of the n items, that have the highest value per unit weight (i.e. highest r_i), followed by items in E and items in L. Intuitively, we should select all the items from H before selecting any items in E, to get the maximum value per unit weight. Similarly, we should first select all items in E, before moving to items in L, in the case that we have already selected all the items in H, and the weight bound of W is still satisfied. Prioritising the selection of items in I using this approach will give us the optimal solution (i.e. proportions of the n items that maximize the value of our knapsack) while keeping the runtime at $O(n)$. The pseudocode for LINEAR-KNAPSACK follows:

```
LINEAR-KNAPSACK(v, w, W, I, S)
 1  let E, H, L, R be empty sets
 2  n = |S|
 3  let M be an empty Map (i.e.a Hashtable/Dictionary data structure)

 4  // We need this Map to get the indices for all r_i = v_i/w_i after they are put in R
 5  // We also make the simplifying assumption that r_is are be unique,
 6  // so as to avoid any key collisions in M
 7  for each i ∈ I
 8      R = R ∪ (v[i]/w[i])
 9      M(v[i]/w[i]) = i

10  // Compute median for the set R
11  // We again make a simplifying assumption that SELECT, can work with sets
12  // as well as arrays. We can alternatively, convert the R to an array in
13  // Θ(n) time and then pass this array into SELECT, without affecting
14  // the overall runtime of LINEAR-KNAPSACK
15  m = SELECT(R, ⌊n/2⌋)
16  W_E = W_H = W_L = 0

17  // Now compute the three sets of items, E, H and L using the value of m
18  // and also compute the sum of weights of all items in each of these sets
19  for each r ∈ R
20      if r == m
21          E = E ∪ M(r)
22          W_E = W_E + w[M(r)]
23      else if r > m
24              H = H ∪ M(r)
25              W_H = W_H + w[M(r)]
26          else
27              L = L ∪ M(r)
28              W_L = W_L + w[M(r)]
29  // Remainig pseudocode is on the next page, "Couldn't find a way to fix the linenumbers"
30  // "My apologies for the same"
```

```
1  // Compute proportions of items, and return the optimal solution
2  // first case: if the total weight of set H is greater than W
3  if W_H > W
4       return LINEAR-KNAPSACK(v, w, W, H, S)
5  else if W_H == W
6            for each i ∈ H
7                 S[i] = 1
8            return S

9  // second case: if the total weight of H is less than W
10 // but total weight of H ∪ E ≥ W
11 else if (W_H < W) & ((W_H + W_E) ≥ W)
12      X = 0
13      for each i ∈ H
14           S[i] = 1
15      X = W_H
16      for each j ∈ E
17           if W − X ≥ w[j]
18                S[j] = 1
19                X = X + w[j]
20           else
21                S[j] = (W−X)/w[j]
22                X = X + S[j]w[j]
23      return S

24 // third case: if the total weight of H ∪ E is less than W
25 else if ((W_H + W_E) < W)
26      X = 0
27      for each i ∈ H
28           S[i] = 1
29      X = W_H
30      for each j ∈ E
31           S[j] = 1
32      X = X + W_H
33      for each k ∈ L
34           if W − X ≥ w[k]
35                S[k] = 1
36                X = X + w[k]
37           else
38                S[k] = (W−X)/w[k]
39                X = X + S[k]w[k]
40      return S
```

LINEAR-KNAPSACK takes 4 input parameters, which are $v, w, W, IandS$. Here v and w are the arrays of values and weights respectively for the n items. W is the weight bound as

specified in the problem instance. I is the array of all items as specified in the problem instance, the elements of I uniquely identify each item and also act as indices for accessing their corresponding weight and value in the w and v arrays. The array S represents the final solution that will be returned by LINEAR-KNAPSACK, i.e. the proportion corresponding to each item $i \in I$. Each element s_i of S is initialized to 0, before passing S as a parameter to LINEAR-KNAPSACK for the first time. LINEAR-KNAPSACK changes the values of s_i based and returns the updated S as an optimal solution. The recurrence relation for LINEAR-KNAPSACK can be written as:

$$T(n) = T(\frac{n}{2}) + \Theta(n)$$

$T(n)$ has this form because, every-time LINEAR-KNAPSACK would make a recurrsive call (line 4 of the pseudocode on page 9), it would reduce the input size n to $\frac{n}{2}$ because we essentially split I using median m and recurrse only on the elements of H, which has size $\frac{n}{2}$. The other lines of LINEAR-KNAPSACK would execute in some constant multiple of n, resulting in the second term $\Theta(n)$ of the recurrence. We can use **Case 3** of Master theorem (with $a = 1$, $b = 2$ and $f(n) = \Theta(n)$) and prove that $T(n) \in \Theta(n) \implies T(n) \in \Omega(n)$ and $T(n) \in O(n)$.
