ALGORITHMS - CSE 202 - FALL 1998
Dynamic Programming and Greedy Method Homework 2 Sample Solutions

Problem 3-9 in Skiena: Consider the following data compression technique. We have a table of m text strings, each of length at most k. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains $(a, ba, abab, b)$ and the data string is $bababbaababa$, the best way to encode it is $(b, abab, ba, abab, a)$ - a total of five code words. Give and $O(nmk)$ algorithm to find the length of the best encoding. You may assume that the string has an encoding in terms of the table.

Solution:

The subproblems involve the suffixes(could also do prefixes) of the data string D. When considering the suffix $D[i..n]$, the shortest encoding can be determined by finding which substrings starting at $D[i]$ match any of the encoding strings, and if that encoding string was used what the smallest encoding of the rest of the data would be. $L[i]$ will store the minimal number of encoding strings needed to encode the data string suffix $D[1..n]$. Assume that the array S holds the m encoding strings.

$$L[i] = \min_{1 \le j \le m} \begin{cases} 1 + L[i + Len(S[j])] \\ \text{if } D[i..(i + Len(S[j]) - 1)] = S[j] \text{ and } (i + Len(S[j])) \le n \\ \\ \infty, \text{otherwise} \end{cases}$$

The base case is $L[n + 1] = 0$. In order to output the encoding strings used, the index of the encoding string that results in a minimal $L[i]$ with be stored in $ES[i]$.

Inputs: array of encoding strings S, data string D, number of encoding strings m, largest size of encoding string k, length of data string n

Outputs: the length of the best encoding

FINDENCODINGLENGTH(S,D,m,k,n)

```
1.  for i = 1 to n
2.      L[i] = ∞
2.      ES[i] = nil
3.  // use results from previous suffixes to compute length
4.  for i = n downto 1
5.      for str = 1 to m
6.          if D[i..(i + length(S[str]) − 1)] = S[str] then
7.              if (1 + L[i + Len(S[str])]) < L[i] then
8.                  L[i] = 1 + L[i + length(S[str])]
9.                  ES[i] = str
10. return L[1]
```

This algorithm's worst-case running time is $\Theta(nmk)$. There are n subproblems and to calculate each one it is necessary to compare m strings of length at most k.

To reproduce the list of encoding strings used we can use the following algorithm.

OUTPUTSTRINGS(S,m,ES)

```
1.  for i = 1 to n
2.      if ES[i] ≠ nil then
3.          output S[ES[i]]
```

Problem 3-2 in Skiena Consider the problem of storing n books on shelves in a library. The order of the books is fixed by the cataloging system and so cannot be rearranged. Therefore, we can speak of a book b_i, where $1 \leq i \leq n$, that has a thickness t_i and height h_i. The length of each bookshelf at the library is L. The values of h_i are not necessarily assumed to be all the same. In this case, we have the freedom to adjust the height of each bookshelf to that of the tallest book on the shelf. Thus, the cost of a particular layout is the sum of the heights of the largest book on each shelf.

a) Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.

Solution: Consider a set of three books, each of thickness 1, and with the first book having height 1, and the second and third books having height 2. Suppose that $L = 2$. Clearly, the b_1 must be placed on the first shelf. The greedy algorithm then places b_2 on the first shelf. This now fills the first shelf, and so we then must place b_3 on the second shelf.

Since the largest book on each shelf is of height 2, the cost of this layout is 4. However, we can do better. Instead of this layout, we can place the first book on the first shelf, and the last two books on the second shelf. In this case, the first shelf has only one book of height 1, and the second shelf has books only of height 2, and so the cost of this layout is 3, which is better than what the greedy algorithm produced. Notice that this example is simple, which is why it is the best example. The simplest example is the best example!

You should write a proof of the optimality of the greedy algorithm for the case where all the book heights are constant (problem 3-1), and determine what in that proof goes wrong in the case of non-equal heights.

b) Give an algorithm for this problem, and analyze it's time complexity.

Solution:
We'll do this with dynamic programming, and break the problem up into sub-problems. Think about the way in which the books are placed on the shelf. We can iteratively place the books on the shelf, at each step, we can make a decision to either place the book on the current shelf, or to start a new shelf. Sometimes, of course, we will have no choice: the current shelf may not be able to fit the book, and we would then be forced to start a new shelf.

We'll solve this problem first by going backwards. We'll loop from n down to 1 and determine the cheapest way of placing books i through n if we start a new shelf with book i. We store the best cost in $COST[i]$. Assume that L represents the length of the shelves, and that H is an array containing the heights of the books. We will let $COST[i]$ be the best possible cost of placing book i through n. For each i, m can be determined by satisfying the inequality, $\sum_{q=i}^{m} H[q] \leq L$.

$$cost[i] = \min_{1 \leq k \leq m} \{cost[i+k] + \max\{H[i], ..., H[i+k-1]\}\}$$

The base case is $cost[n] = H[n]$. The algorithm would look something like the following:

Inputs: An array of heights H and an array of thicknesses T, both indexed from 1 to n, containing positive real numbers. The length of each shelf L is assumed to be a global constant. We further assume that $T[i] \leq L$ for each i, $1 \leq i \leq n$ (no fat books that won't fit on any shelf!).

Outputs: An array COST, indexed from 1 to n. COST[i] indicates the total cost of shelving books i through n starting on a new shelf. Also, a list of numbers indicating which books should mark the first book on the shelf.

prodecure PLACEBOOKS (H, T)
var

```
    COST : array(1..n) of real;
    currentshelf : real;
    i, j, start : integer;
begin
    COST[n] := H[n];
    for i := n − 1 downto 1 loop
      currentshelf := T[i];
          Append start to left end list.
      for j := i + 1 to n loop
        currentshelf := currentshelf + T[j];
        if currentshelf ≤ L and
          max{H[i], ..., H[j]} + COST[j + 1] ≤ COST[i] then
            COST[i] = max{H[i], ..., H[j]} + COST[j + 1]
            NextShelf[i] = j + 1
        end if;
      end for loop;
    end for loop;
end PLACEBOOKS;
```

The outer loop goes through the different subproblems in reverse order. For each subproblem, the inner loop iterates through the books to be placed on the shelf.

The if statement on the inside checks two things: First, if there is room to place the next book on the shelf. If there isn't, then there is no point in even attempting to place the book on the shelf. If there is, then we need to check to see if it will be cheaper to place the current book on the shelf we're on, or to start a new shelf. If we want to place the book on the current shelf, then we add to the current size of the shelf we're working on and increment the loop counter. Again, if not, then we have to add to the cost of the current subproblem and start a new shelf.

Performance: The execution of the inside of the innermost for loop is bounded by a constant amount. The number of times the innermost loop is executed is bounded by a n, and the outer loop is executed $n - 1$ times, so the performance will be $O(n^2)$.

Reconstruction: To reconstruct the book layout we store the book that will start the next shelf in $NextShelf[i]$. Then we place book 1 through $NextShelf[1] - 1$ on the first shelf. Then books $NextShelf[1]$ through $NextShelf[NextShelf[1] - 1]$, etc.

Problem 3-8 in Skiena Let us assume that we are given a string $x = x_1 x_2 \ldots x_{n-1} x_n$ of characters from an alphabet $\{a_1, \ldots, a_k\}$, with a multiplication $*$ defined on ordered pairs of the alphabet, with n as the length of a given string, and k as the number of elements in the given alphabet. We are interested in determining whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a, where a is some element in the alphabet.

The algorithm presented below is an iterative algorithm using dynamic programming. We will break the problem down into sub-problems. Given a string, we want to examine all ways to break the string up into two contiguous parts, and for each partition, we can determine whether or not the substrings can be parenthesized to form a left and right terms so that the resulting multiplication gives us the desired product. This algorithm can be expressed using a recursive scheme:

Recursive solution: Function ISP (short for "is string parenthesizable")
Inputs: A string S of alphabet elements indexed from $left$ to $right$, and a target element $target$.
Output: $parenthesizable$, a boolean representing true iff the string is parenthesizable to give a result $target$.

```
function ISP (S, left, right, target)
var
  partition, alpha1, alpha2 : integer;
  parenthesizable : boolean;
begin
  [base case]
  if left = right then
    if S[left] = target then
      return true;
    else
      return false;
    end if;
  end if;
  [recursive case]
  parenthesizable := false;
  for partition := left to right - 1 loop
    for alpha1 := 1 to k loop
      for alpha2 := 1 to k loop
        if alpha1 * alpha2 = target then
          if ISP(partition, S[left..partition], target) and
             ISP(partition, S[partition + 1..right], target) then
            parenthesizable :=true;
          end if;
        end if;
      end for loop;
    end for loop;
  end for loop;
  return parenthesizable;
end ISP;
```

Of course, a much better way to do this, as mentioned, is a dynamic programming approach using iteration instead of recursion. Having considered sub-problems as being the problem of determining whether or not a given substring has a parenthesization giving a particular product, we may simply start with the given string, and consider ALL possible contiguous substrings, and determine whether or not they have parenthesizations giving any particular target value.

We will iterate by first checking all substrings of length 2, then by checking all substrings of length 3, and so on until we check the given string itself. For each substring, we will have to test to see if it has a parenthesization for any possible target value in the alphabet. As an added bonus, if the string is in fact parenthesizable to get the given product, this algorithm will also output such a parenthesization.

Iterative solution: Procedure ISP (short for "is string parenthesizable")

Inputs: A string S of alphabet elements indexed from 1 to n.

Outputs: $parenthesizable$, a triply indexed array of boolean values representing whether or not a particular substring of S is parenthesizable to give a particular result. More specifically, the value $parenthesizable[i][L][R]$ is true iff the substring $S[L..R]$ has a parenthesization giving the target value a_i, which is one of the alphabet elements. The other output is, $order$, another triply indexed array, this one of string values. $order[i][L][R]$ is defined only when $parenthesizable[i][L][R]$ has the value **true**, and in that case will be a string a characters representing a correct parenthesization of $S[L..R]$ to acheive the target value a_i. Notice that both $parenthesizable[i][L][R]$ and $order[i][L][R]$ make sense only if $L \leq R$.

procedure ISP (S, n)
var
$parenthesizable$: array {integer,integer,integer} of boolean := false;
$order$: array {integer,integer,integer} of string;
$size, partition, i, j, left, right$: integer;
begin
[single length strings]
for $i := 1$ **to** n **loop**
for $j := 1$ **to** k **loop**
if $S[i] = a_i$ **then**
$parenthesizable[j][i][i] =$ **true**;
$order[j][i][i] = S[i]$;
end if;
end for loop;
end for loop;
[strings of length greater than 1]
for $size := 2$ **to** n **loop**
for $left := 1$ **to** $n - size + 1$ **loop**
for $partition := left$ **to** $left + size - 2$ **loop**
for $i := 1$ **to** k **loop**
for $leftterm := 1$ **to** k **loop**
for $rightterm := 1$ **to** k **loop**
if $leftterm * rightterm = i$ **and**
$parenthesizable[leftterm][left][partition]$ **and**
$parenthesizable[rightterm][partition + 1][left + size - 1]$ **then**
$parenthesizable[i][left][left + size - 1] :=$ **true**;
$order[i][left][left + size - 1] :=$ "(" **concat**

```
                    order[leftterm][left][partition] concat
                    order[rightterm][partition + 1][left + size − 1] concat
                    ")";
                end if;
              end for loop;
            end for loop;
          end for loop;
        end for loop;
      end for loop;
    end for loop;
end ISP;
```

The **concat** operation is simply string concatenation. This is used to produce a correct parenthesization of a string (if one exists).

Let's look at this algorithm more carefully. The outermost two loop structures loop through the different sub-problems, ordered first by size, and then by where the left end of the sub-problem starts. The third loop loops through the different ways the sub-problem can be broken up. Remember that given a string, it can be broken up into two parts a number of different ways.

The fourth nested loop will loop through the k different alphabet symbols, and for each one, we wish to check to see if a parenthesization exists for that resulting target value. Finally, the innermost two loops will loop through the different values that each of the two terms can take on.

Performance: The first three nested loops are executed no more than n times each, and then on the inside of that, the three innermost nested loops are executed k times each, and the main body of the six nested loops is executed a constant amount of time during each pass. Therefore, the performance of this algorithm should be $O(k^3n^3)$. Of course, we should check to see that the number of times the outer three loops are executed isn't asymptotically any LESS than n^3.

The outermost two loops will give us $\frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}$ loops. (Why?) The inner most loop gives us the number of ways that the sub-problem can be broken into two parts. For a sub-problem of size s, there will be $s - 1$ different way to break up the problem into two parts. The number of sub-problems of size s is $n - s + 1$. (Why?) Thus we can add up the total number of times that the outermost three nested loops will execute, and we get:

$$\sum_{s=2}^{n}(n - s + 1)(s - 1)$$

And this quantity will be a cubic polynomial. (Why?) Hence, in fact, the running time of this algorithm will be $O(k^3n^3)$. Can you do better than this?

PROBLEM 1:

Solution:

Let's say there are k stations numbered $1, 2, ..., k$ between New York City and Reno. Professor Midas should use a greedy method: He should drive past gas stations until he reaches a gas station without whose services he would be stranded (i.e., gasless) in the next inter-station stretch of interstate. Every time Midas stops at a station, he should fill up his tank. If Midas can go all the way to Reno using his current level of gas, he should do so. (A more precise formulation would lead to tedious technicalities.) Let's call this strategy S. We characterize strategies by the gas stations that are selected and the ones that are not. Let $N(S)$ be the number of gas stations where Midas stops, if he follows strategy S. Consider S as an array, where for $1 \leq i \leq k$:

$$\text{Let } S[i] = \begin{cases} 1, & \text{if Midas stops at ("selects") station } i \\ 0, & \text{if Midas does not select station } i \end{cases}$$

Let's use similar notation for other strategies.

Observe that if any two consecutive gas stations are more than n miles apart (or if he starts off from New York City without enough gas to even get to the first station, or if the kth gas station is more than n miles away from Reno), there is no solution to Midas' problem; in other words, there is no strategy that will work at all for these situations. Assume that these situations do not occur.

We want to prove that $N(S) \leq N(U)$ for all possible strategies U. Let's do a proof by induction. Let T be an optimal strategy in the sense that $N(T) \leq N(U)$ for all possible strategies U.

Let $P(i)$ be the property "There exists a strategy T_i such that:
(*) $N(T_i) \leq N(T)$ (i.e., T_i is at least as good a strategy as T), and
(**) $T_i[j] = S[j]$ for $1 \leq j \leq i$ (i.e., T_i makes the same station selection decisions as S up to and including station i)."

Base case: $i = 1$. Let T_1 be the same strategy as T, but set $T_1[1]$ equal to $S[1]$. There are three cases. Case 1: $S[1] = T[1]$: In this case, $T_1[1] = S[1] = T[1]$, so T_1 and T are the same strategy. Thus $N(T_1) = N(T)$ trivially, and T_1 satisfies (*). Also, since $T_1[1] = S[1]$ by construction, T_1 also satisfies (**). Case 2: $S[1] = 1$ and $T[1] = 0$: In this case, strategy S selects station 1, but T does not. However, since S is the strategy that dictates that Midas successively selects the furthest gas station that will keep him from being stranded, that means that the T strategy will leave Midas stranded in the next stretch of interstate, before he can reach station 2 or Reno, whichever comes first. T can not be optimal, so this case can not happen. Case 3: $S[1] = 0$ and $T[1] = 1$: In this case, strategy T selects station 1, but S does not (which means that Midas will not get stranded before reaching station 2 or Reno, whichever comes first). We let $T_1[1] = S[1] = 0$, so T_1 also does not select station 1. We also set $T_1[2]$ equal to 1, to guarantee that T_1 will not leave Midas stranded. (If $k = 1$, Midas can go all the way to Reno.) We know that: (1) T and T_1 make the same selection decisions for the other stations ($j \geq 3$), (2) T selects station 1 but T_1 does not, and (3) T_1 selects station 2 and T may or may not. In any case, $N(T_1) \leq N(T)$. So, T_1 satisfies (*). Also, since $T_1[1] = S[1]$ by construction, T_1 also satisfies (**).

Inductive hypothesis. Assume $P(i)$ is true. Show that $P(i+1)$ is true ($1 \leq i < k$): "There exists a strategy T_{i+1} such that:
(***) $N(T_{i+1}) \leq N(T)$ (i.e., T_{i+1} is at least as good a strategy as T), and
(****) $T_{i+1}[j] = S[j]$ for $1 \leq j \leq i+1$ (i.e., T_{i+1} makes the same station selection decisions as S up to and including station $i+1$)."

Inductive step. By the inductive hypothesis, there exists a strategy T_i that satisfies (*) and (**). Let T_{i+1} be the strategy such that:

$$T_{i+1}[j] = \begin{cases} T_i[j], & 1 \leq j \leq i \\ S[j], & j = i+1 \\ T_i[j], & i+1 < j \leq k \end{cases}$$

In other words, T_{i+1} is the same strategy as T_i, but set $T_{i+1}[i+1]$ equal to $S[i+1]$. Think of T_i as the "old" optimal strategy and T_{i+1} as the "new" optimal strategy that we are developing from the "old" one. There are three cases. Case 1: $S[i+1] = T_i[i+1]$: In this case, $T_{i+1}[i+1] = S[i+1] = T_i[i+1]$, so T_{i+1} and T_i are the same strategy. Thus $N(T_{i+1}) = N(T_i) \leq N(T)$ (by the inductive hypothesis), and T_{i+1} satisfies (***). Also, since $T_{i+1}[i+1] = S[i+1]$ by construction, and $T_{i+1}[j] = T_i[j] = S[j]$ for $1 \leq j \leq i$ by the inductive hypothesis, then T_{i+1} also satisfies (****). Case 2: $S[i+1] = 1$ and $T_i[i+1] = 0$: In this case, strategy S selects station $i+1$, but T_i does not. However, since S is the strategy that dictates that Midas successively selects the furthest gas station that will keep him from being stranded, that means that the T_i strategy will leave Midas stranded in the next stretch of interstate, before he can reach station $i+2$ or Reno, whichever comes first. (Remember that, by the inductive hypothesis, T_i made the same selection decisions as S for the first i stations.) T_i can not be optimal, so this case can not happen. Case 3: $S[i+1] = 0$ and $T_i[i+1] = 1$: In this case, strategy T_i selects station $i+1$, but S does not (which means that Midas will not get stranded before reaching station $i+2$ or Reno, whichever comes first; remember the inductive hypothesis). We let $T_{i+1}[i+1] = S[i+1] = 0$, so T_{i+1} also does not select station $i+1$. We also set $T_{i+1}[i+2]$ equal to 1, to guarantee that T_{i+1} will not leave Midas stranded. (If $k = i+1$, Midas can go all the way to Reno.) We know that: (1) T_i and T_{i+1} make the same selection decisions for the other stations ($j \neq i+1, i+2$), (2) T_i selects station $i+1$ but T_{i+1} does not, and (3) T_{i+1} selects station $i+2$ and T_i may or may not. In any case, $N(T_{i+1}) \leq N(T_i) \leq N(T)$ (by the inductive hypothesis). So, T_{i+1} satisfies (***). Also, since $T_{i+1}[i+1] = S[i+1]$ by construction, and $T_{i+1}[j] = T_i[j] = S[j]$ for $1 \leq j \leq i$ by the inductive hypothesis, then T_{i+1} also satisfies (****).

Conclusion. We may now conclude that $P(k)$ is true. In other words, there exists a strategy T_k such that:
(1) $N(T_k) \leq N(T)$ (i.e., T_k is at least as good a strategy as T), and
(2) $T_k[j] = S[j]$ for $1 \leq j \leq k$ (i.e., T_k makes the same station selection decisions as S up to and including station k; THAT MEANS THAT T_k AND S ARE THE SAME STRATEGY!!!) So, we now know that S (the strategy formerly known as T_k) is at least as good a strategy as our optimal strategy T, in the sense that S selects no more gas stations than T does [while not stranding Midas]. So, S itself is an optimal strategy.