Greedy Algorithms and Dynamic Programming

The Greedy Principle

- **The problem:** We are required to find a feasible solution that either maximizes or minimizes a given objective solution**.**
- It is easy to determine a feasible solution but not necessarily an optimal solution.
- The greedy method solves this problem in stages, at each stage, a decision is made considering inputs in an order determined by the selection procedure which may be based on an optimization measure.
- The greedy algorithm always makes the choice that looks best at the moment.
  - For each decision point in the greedy algorithm, the choice that seems best at the moment is chosen
- It makes a local optimal choice that may lead to a global optimal choice.

Activity Selection Problem

- Scheduling a resource among several competing activities.
- S = {1,2, 3, …, *n*} is the set of *n* proposed activities
- The activities share a resource, which can be used by only one activity at a time -a Tennis Court, a Lecture Hall etc.,
- Each activity *i* has a start time, *si* and a finish time *fi*, where $si \leq fi$.
- When selected, the activity takes place during time (*si, fi*)
- Activities *i* and *j* are compatible if $si \geq fj$ or $sj \geq fi$
- The activity-selection problem selects the maximum-size set of mutually compatible activities
- The input activities are in order by increasing finishing times.
- $f1 \leq f2 \leq f3 \ldots \leq fn$; Can be sorted in O(*n log n*) time

Procedure for activity selection (from CLRS)

Procedure GREEDY_ACTIVITY_SELECTOR(*s, f*)
$n \leftarrow$ length [*S*];
$A \leftarrow \{1\}$;
$j \leftarrow 1$;
for $i \leftarrow 2$ to n
    do if $si \geq fj$
        then $A \leftarrow A \cup \{i\}$;
            $j \leftarrow i$;

i	si	fi
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

- Initially we choose activity 1 as it has the least finish time.
- Then, activities 2 and 3 are not compatible as s2 < f1 and s3 < f1.
- We choose activity 4, s4 > f1, and add activity 4 to the set A.
- A = {1, 4}
- Activities 5, 6, and 7 are incompatible and activity 8 is chosen
- A = {1,4,8}
- Finally activity 10 is incompatible and activity 11 is chosen
- A {1,4,8,11}
- The algorithm can schedule a set of n activities in Θ (n) time.

Huffman codes

Huffman codes are used to compress data. We will study Huffman's greedy algorithm for encoding compressed data.

Data Compression

- **A given file can be considered as a string of characters.**
- **The work involved in compressing and uncompressing should justify the savings in terms of savings in storage area and/or communication costs.**
- **In ASCII all characters are represented by bit strings of size 7.**
- **For example if we had 100000 characters in a file then we need 700000 bits to store the file using ASCII.**

Example

The file consists of only 6 characters as shown in the table below.
Using the fixed-length binary code, the whole file can be encoded in 300,000 bits.
However using the variable-length code , the file can be encoded in 224,000 bits.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

A variable length code gives frequent characters, short code words and infrequent characters long words.
In the above variable-length code, 1-bit string represents the most frequent character a, and a 4-bit string represents the most infrequent character f.

Let us denote the characters by C_1, C_2, …, C_n and denote their frequencies by f_1, f_2, ,,,, f_n. Suppose there is an encoding E in which a bit string S_i of length s_i represents C_i, the length of the file compressed by using encoding E is

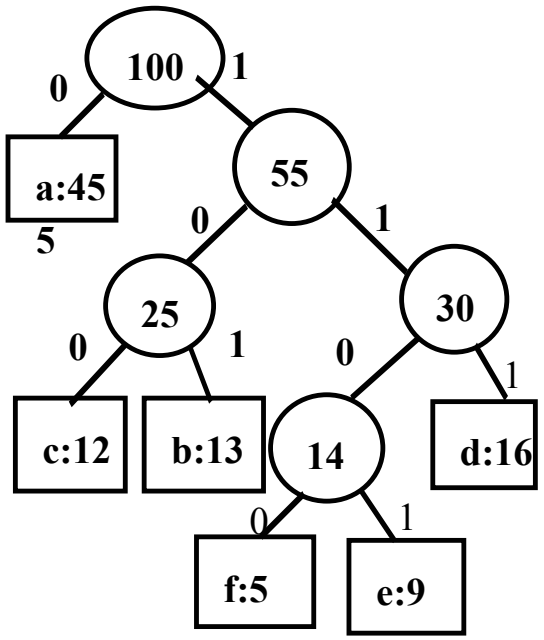
$$L(E,F) = \sum_{i=1}^{n} s_i \cdot f_i$$

Prefix Codes

- The prefixes of an encoding of one character must not be equal to a complete encoding of another character.
  - •1100 and 11001 are not valid codes
  - •because 1100 is a prefix of 11001
- This constraint is called the prefix constraint.
- Codes in which no codeword is also a prefix of some other code word are called prefix codes.
- Shortening the encoding of one character may lengthen the encodings of others.
- To find an encoding E that satisfies the prefix constraint and minimizes L(E,F).

The prefix code for file can be represented by a binary tree in which every non leaf node has two children. Consider the variable-length code of the table above, a tree corresponding to the variable-length code of the table is shown below.

Note that the length of the code for a character is equal to the depth of the character in the tree shown.



Greedy Algorithm for Constructing a Huffman Code

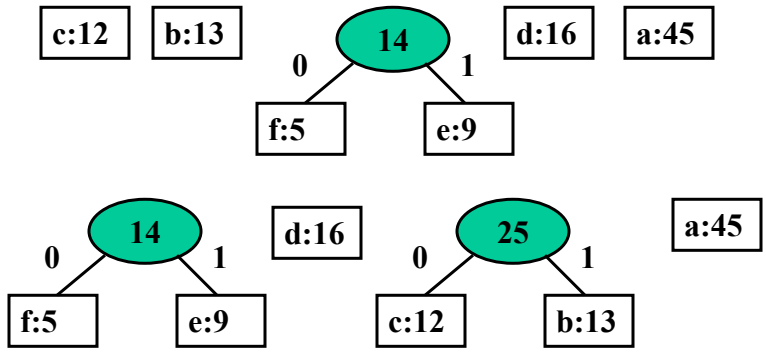
The algorithm builds the tree corresponding to the optimal code in a bottom-up manner.
The algorithm begins with a set of $|C|$ leaves and performs a sequence of 'merging' operations to create the tree.
C is the set of characters in the alphabet.

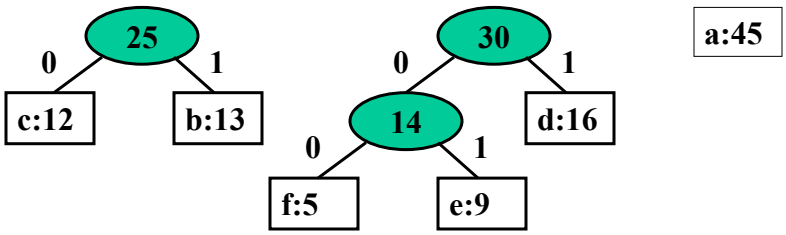
Procedure Huffman_Encoding(S,f);
Input : S (a string of characters) and f (an array of frequencies).
Output : T (the Huffman tree for S)

```
1.    insert all characters into a heap H according to
                                    their frequencies;
2.    while H is not empty do
3.       if  H contains only one character x then
4.           x ← root (T);
5.       else
6.           z ← ALLOCATE_NODE();
7.           x ← left[T,z] ← EXTRACT_MIN(H);
8.           y ← right[T,z] ← EXTRACT_MIN(H);
9.           f_z ← f_x + f_y;
10.          INSERT(H,z);
```

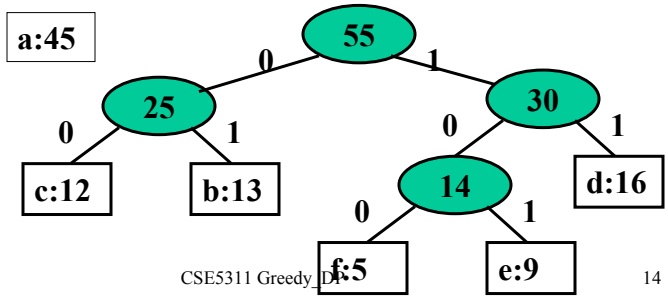
f:5 | e:9 | c:12 | b:13 | d:16 | a:45

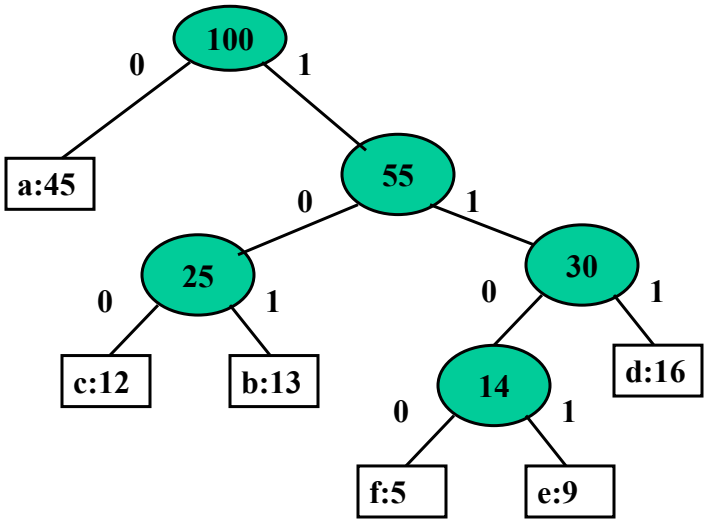
- The algorithm is based on a reduction of a problem with n characters to a problem with n-1 characters.
- A new character replaces two existing ones.





Suppose Ci and Cj are two characters with minimal frequency, there exists a tree that minimizes L (E,F) in which these characters correspond to leaves with the maximal distance from the root.





Complexity of the algorithm

Building a heap in step 1 takes O(n) time
Insertions (steps 7 and 8) and
deletions (step 10) on H
take O (log n) time each
Therefore Steps 2 through 10 take O(n log n) time

Thus the overall complexity of the algorithm is
O(n logn).

Dynamic programming techniques

Topics

- Basics of DP
- Matrix-chain Multiplication
- Longest Common subsequence
- All-pairs Shortest paths

Further Reading

Chapter 16 from Textbook

Dynamic programming

- Solves problems by combining the solutions to subproblems
- DP is applicable when subproblems are not independent
  Subproblems share subsubproblems
  In such cases a simple
  Divide and Conquer strategy solves common subsubproblems.
- In DP every subproblem is solved just once and the solution is saved in a table for future reference (avoids re-computation).
- DP is typically applied to optimization problems
- A given problem may have many solutions, DP chooses the optimal solution.

Four stages of Dynamic Programming

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed results

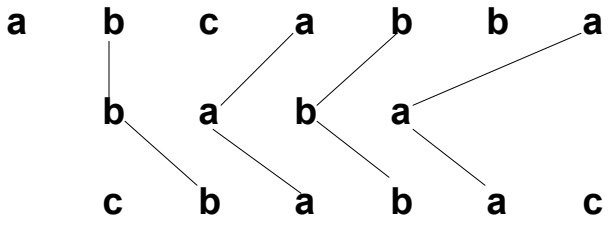
Longest common subsequence

A subsequence is formed from a list by deleting zero or more elements (the remaining elements are in order)

A common subsequence of two lists is a subsequence of both.
The longest common subsequence (LCS) of two lists is the longest among the common subsequences of the two lists.

Example:

abcabba and cbabac are two sequences
baba is a subsequence of both



To find the length of an LCS of lists x and y, we need to find the lengths of the LCSs of all pairs of prefixes.

- a prefix is an initial sublist of a list

If $x = (a_1,a_2,a_3, \ldots, a_m)$ and
$y = (b_1,b_2,b_3, \ldots, b_n)$
$0 \le i \le m$ and $0 \le j \le n$

Consider an LCS of the prefix $(a_1,a_2,a_3, \ldots, a_i)$ from x and of the prefix $(b_1,b_2,b_3, \ldots, b_j)$ from y.

If i or j = 0 then one of the prefixes is ε and the only possible common subsequence between x and y is ε and the length of the LCS is zero.

L(i,j) is the length of the LCS of $(a_1,a_2,a_3, \ldots, a_i)$ and $(b_1,b_2,b_3, \ldots, b_j)$.

BASIS: If i+j = 0, then both i and j are zero and so the LCS is ε.

INDUCTION: Consider i and j, and suppose we have already computed L(g,h) for any g and h such that

$g+h < i+j$.

1. If either *i* or *j* is 0 then L(i,j) = 0.
2. If i>0 and j>0, and $a_i \neq b_j$ then
   L(i,j) = max(L(i,j-1),L(i-1,j)).
3. If i >0 and j> 0, and $a_i = b_j$ then L(i,j) = L(i-1,j-1)+1.

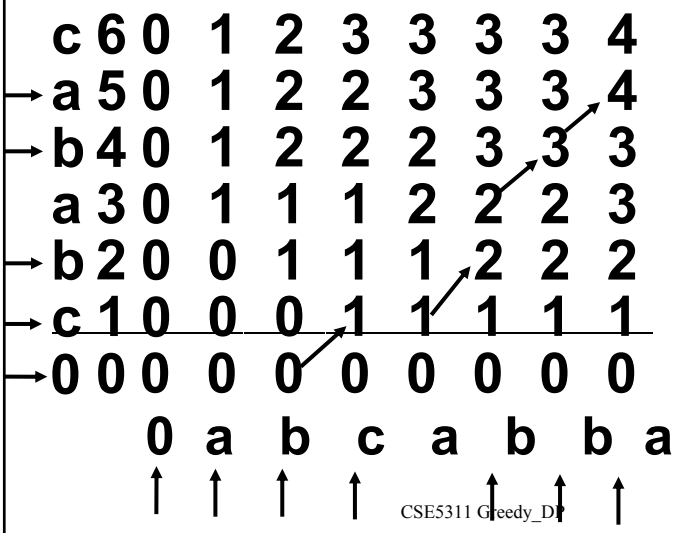
1. If either i or j is 0 then L(i,j) = 0.
2. If i>0 and j>0, and $a_i \neq b_j$ then
   L(i,j) = max(L(i,j-1),L(i-1,j)).
3. If i >0 and j> 0, and $a_i = b_j$ then L(i,j) = L(i-1,j-1)+1.

		b				
		a				
	c	0	1	1	2	2
	a	0	1	1	1	1
	ε	0	0	0	0	0
		ε	a	b	c	a

Procedure LCS(x,y)
Input : The lists x and y
Output : The longest common subsequence and it's length

```
1. for j ← 0 to n do
2.       L[0,j] ← 0;
3. for  i ← 1 to m do
4.       L[i,0] ←0;
5.       for j ← 1 to n do
6.             if a[i] ≠ b[j] then
7.                   L[i,j] ← max {L[i-1,j],L[i,j-1]};
8.             else
9                    L[i,j] ← 1+L[i-1,j-1];
```


Example:
Consider, lists x = abcabba and y = cbabac



Consider another example

abaacbacab and bacabbcaba LCS : bacacab

b	0	1	2	3	4	5	5	5	6	7	7
a	0	1	2	3	4	4	4	5	6	6	6
c	0	1	2	3	4	4	4	5	5	5	6
a	0	1	2	3	4	4	4	4	5	5	6
b	0	1	2	3	3	4	4	4	4	5	5
c	0	1	2	3	3	3	3	4	4	4	4
a	0	1	2	2	3	3	3	3	3	3	4
a	0	1	2	2	2	2	2	2	3	3	3
b	0	1	1	1	1	2	2	2	2	2	2
a	0	0	1	1	1	1	1	1	1	1	1`
0	0	0	0	0	0	0	0	0	0	0	0
	0	b	a	c	a	b	b	c	a	b	a

Matrix-chain Multiplication

Consider the matrix multiplication procedure

```
        MATRIX_MULTIPLY(A,B)
1.      if columns[A] ≠ rows[B]
2.          then error "incompatible dimensions"
3.          else for i ← 1 to rows[A]
4.                   do for j ←1 to columns[B]
5.                      do C[i,j] ←0;
6.                          for k ← 1 to columns [A]
7.                              do C[i,j] ← C[i,j]+A[i,k]*B[k,j];
8.                  return C
```

The time to compute a matrix product is dominated by the number of scalar multiplications in line 7.

If matrix A is of size ($p \times q$) and B is of size ($q \times r$), then the time to compute the product matrix is given by pqr.
Consider three matrices A1, A2, and A3 whose dimensions are respectively
(10×100), (100×5) (5×50).
Now there are two ways to parenthesize these multiplications

I $((A_1 \times A_2) \times A_3)$
II $(A_1 \times (A_2 \times A_3))$

First Parenthesization

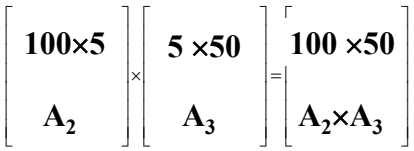
Product $A_1 \times A_2$ requires $10 \times 100 \times 5 = 5000$ scalar multiplications
$A_1 \times A_2$ is a (10×5) matrix
$(A_1 \times A_2) \times A_3$ requires $10 \times 5 \times 50 = 2500$ scalar multiplications.
Total : 7,500 multiplications

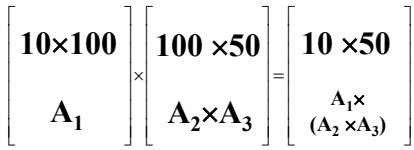
$$\begin{bmatrix} 10\times100 \\ A_1 \end{bmatrix} \times \begin{bmatrix} 100\times5 \\ A_2 \end{bmatrix} = \begin{bmatrix} 10\times5 \\ A_1\times A_2 \end{bmatrix} \qquad A_1$$

$$\begin{bmatrix} 10\times5 \\ A_1\times A_2 \end{bmatrix} \times \begin{bmatrix} 5\times50 \\ A_3 \end{bmatrix} = \begin{bmatrix} 10\times50 \\ (A_1\times A_2)\times A_3 \end{bmatrix}$$

Second Parenthesization

Product $A_2 \times A_3$ requires $100 \times 5 \times 50 = 25{,}000$ scalar multiplications
$A_2 \times A_3$ is a (100×50) matrix
$A_1 \times (A_2 \times A_3)$ requires $10 \times 100 \times 50 = 50{,}000$ scalar multiplications
Total : 75,000 multiplications.





The first parenthesization is 10 times faster than the second one!!

How to pick the best parenthesization ?

The matrix-chain matrix multiplication

Given a chain $(A_1, A_2, \ldots, A_n)$ of n matrices, where for i = 1,2, …, n matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

The order in which these matrices are multiplied together can have a significant effect on the total number of operations required to evaluate the

An optimal solution to an instance of a matrix--chain multiplication problem contains within it optimal solutions to the subproblem instances.

Let, P(n) : The number of alternative parenthesizations of a sequence of n matrices

We can split a sequence of n matrices between kth and (k+1)st matrices for any k = 1, 2, …, n-1 and we can then parenthesize the two resulting subsequences independently,

$$P(n) = \begin{cases} 1 \ \textit{if} \ n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) \ \textit{if} \ n \geq 2 \end{cases}$$

This is an exponential in *n*

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$ or

$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$

or $(A_1 \times (A_2 \times A_3)) \times A_4$

Structure of the Optimal Parenthesization

$A_{i..j} = A_i \times A_{i+1} \times \ldots \times A_j$

An optimal parenthesization splits the product

$A_{i..j} = (A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$

for $1 \leq k < n$

The total cost of computing $A_{i..j}$

= cost of computing $(A_i \times A_{i+1} \times \ldots \times A_k)$

+ cost of computing $(A_{k+1} \times A_{k+2} \times \ldots \times A_j)$

+ cost of multiplying the matrices $A_{i..k}$ and $A_{k+1..j}$.

$A_{i...k}$ must also be optimal if we want $A_{i..j}$ to be optimal. If $A_{i..k}$ is not optimal then $A_{i..j}$ is not optimal. Similarly $A_{k+1..j}$ must also be optimal.

Recursive Solution

We'll define the value of an optimal solution recursively in terms of the optimal solutions to subproblems.

m[i,j] = minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$

m[1,n] = minimum number of scalar multiplications needed to compute the matrix $A_{1..n.}$

If i = j ; the chain consists of just one matrix

$A_{i..i} = A_i$ - no scalar multiplications

m[i,i] = 0 for i = 1, 2, …, n.

m[i,j] = minimum cost of computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying these two matrices

Multiplying $A_{i..k}$ and $A_{k+1....j}$ takes $p_{i-1} p_k p_j$ scalar multiplications

$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ for $i \leq k < j$

The optimal parenthesization must use one of these values for k, we need to check them all to find the best solution.

Therefore,

$$m[i,j] = \begin{cases} 0 \ \ if \ \ i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j]\} + p_{i-1} p_k p_j \end{cases}$$

Let s[i,j] be the value of k at which we can split the product $A_i \times A_{i+1} \times \ldots \times A_j$
to obtain the optimal parenthesization.
s[i,j] equals a value of k such that
$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ for $i \le k < j$

Procedure Matrix_Chain_Order (p)

Input: sequence $(p_0, p_1, \ldots p_n)$
Output : an auxiliary table m[1..n,1..n] with m[i,j] costs and another auxiliary table s[1..n,1..n] with records of index k which achieves optimal cost in computing m[i,j]

```
1.    n←length[p]-1;
2.    for i ← 1 to n
3.        do m[i,i] ←0;
4.    for l ← 2 to n
5.        do for i ← 1 to n-l+1
6.               do j ← i+l-1
7.                  m[i,j]←∞;
8.                  for k ← i to j-1
9.                      do q ←m[i,k]+m[k+1,j]+p_{i-1}p_k p_j;
10.                        if q < m[i,j];
11.                           then m[i,j] ←q;
12.                                s[i,j] ←k;
13.   return m and s
```

Consider Four Matrices

A1 : 10 × 20 **A2 : 20×50**

A3: 50×1 **A4 : 1 ×100**

MIN[(10,000+500),(1000+200)

j↓ i→	1	2	3	4
1	0	--	--	--
2	10,000	0	--	--
3	1200	1000	0	--
4	2200	3000	5000	0

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$

$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$

10×50

and $(A_1 \times (A_2 \times A_3)) \times A_4$

20×1

$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ for $i \leq k < j$

Consider, A1 (30×35)A2 (35×15)A3 (15×5), A4(5×10), A5(10×20), A6(20×25)

m

j↓/i→	1	2	3	4	5	6
1	0	--	--	--	--	--
2	15,750	0	--	--	--	--
3	7,875	2,625	0	--	--	--
4	9,375	4,375	750	0	--	--
5	11,875	7,125	2,500	1,000	0	--
6	15,125	10,500	5,375	3,500	5,000	0

s

j↓/i→	1	2	3	4	5
2	1	-	-	-	-
3	1	2	-	-	-
4	3	3	3	-	-
5	3	3	3	4	-
6	3	3	3	5	5

(A1..A3)× (A4..A6)

(A1 × (A2 × A3)) ×

((A4 × A5) × A6)

Complexity? With and without DP?

- $T(1) \geq 1$
- $T(n) \geq 1 + \sum_{k=1}^{n-1} T(k) + T(n-k) + 1) \; for \;\; n \geq 1$
- $T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n$
- Exponential in n

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of *n* words of length $l_1, l_2, \ldots, l_n$, measured in input 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* and leave exactly one space between words, the number of extra space characters at the end of the line is

$$M - j + i - \sum_{k=i}^{j} l_k$$

We wish to minimize the sum, over all the lines except the last of the 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.

Hints

- Assume that no word is longer than a line
- Determine the cost of a line containing words *i* through *j* (cost is the number of free spaces)
- We want to minimize the sum of line costs over all lines in the paragraph.
- Try to represent the above by a recursive expression

- **A ski rental agency has *m* pairs of skis, where the height of the *i*th pair of skis is s_i. There are *n* skiers who wish to rent skis, where the height of the *i*th skier is h_i. Ideally, each skier should obtain a pair of skis whose height matches with his own height as closely as possible. Design an efficient algorithm to assign skis so that the sum of the absolute differences of the heights of each skier and his/her skis is minimized.**