Introduction to Algorithms and Data Structures

Lesson 16: Advanced Algorithm
Dynamic Programming

Professor Ryuhei Uehara,

School of Information Science, JAIST, Japan.

uehara@jaist.ac.jp

http://www.jaist.ac.jp/~uehara

Developing Algorithms based on Dynamic Programming

Objects: optimization problems
problem of finding an optimal solution among those satisfying given constraints.

Problem solving by dynamic programming

1. Characterize a structure of an optimal solution.
2. Define an optimal solution recursively.
   (construct a solution using solutions to subproblems)
3. Compute a value of an optimal solution in a bottom-up manner (in the way to fill in a table)
4. Construct an optimal solution using information obtained.
   (not only finding a value of an optimal solution but also constructing an optimal solution by following in the table)

Number of combinations

Problem P1: Calculate the number C(n, k) of combinations to choose k items from n different items.

Using the formula,

$$C(n, k) = C(n-1, k-1) + C(n-1, k), \quad \text{if } 0<k<n,$$

$$C(n, 0) = C(n, n) = 1.$$

Therefore, we have the following program.

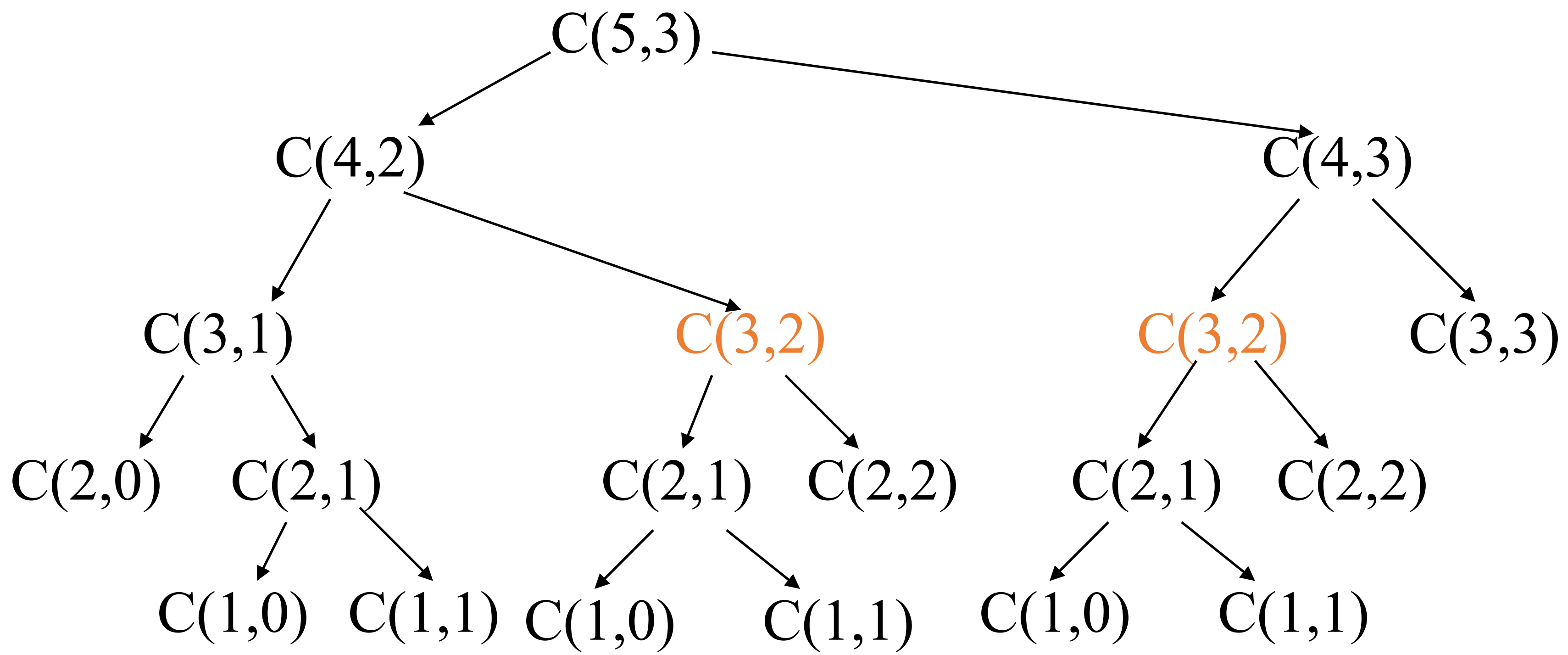
```
int C(int n, int k){
  if(k==0 || k==n) return 1;
  return C(n-1, k-1) + C(n-1, k);
}
```

When you implement the program in practice, you will find that it takes much time. Why does it take time?

Analysis of computation time:

Let T(n,k) be time to compute C(n,k). Then, we have $T(n,k)=T(n-1,k-1)+T(n-1,k)$. Thus, $T(n,k)=C(n,k)$.
This is an exponential function.

Let's analyze behavior of the program!



How is the function called

The function is called many times for the same value.

Example : C(3,2) is called twice. ➔ redundant

If we store the value C(n,k) as the (n,k) element of an array when it is first computed, then the same value is never computed twice. Basically, it suffices to fill in the table.

Fill in the table C(n,k)!

Formula : $C(n,k) = C(n-1,k-1) + C(n-1,k)$

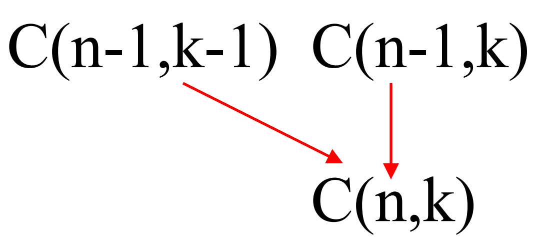
If the values in the (n-1)-st row are available, C(n,k) is easily computed. ➔Thus, we should fill in the table from the 1st row.

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3						
4						
5						

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4						
5						

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5						

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1



Each element of the table can be computed in constant time.

Thus, the total time is $O(n^2)$.

C program is as follows:

```
int C(int n, int k){
  C[0][0]=1;
  for(i=1; i<=n; i++){
    C[i][0]=1; C[[i][i]=1;
    for(j=1; j<i; j++)
      C[i][j] = C[i-1][j-1] + C[i-1][j];
  }
  return C[n][k];
}
```

Exercise E1: Consider an algorithm that computes C(n,k) based on the Naïve idea such that it computes C(n,k) correctly if C(n,k) itself does not overflow.

Naive Algorithm 2:

Using the formula $C(n,k) = \dfrac{n!}{(n-k)!\ k!}$, it can be computed in O(n).

Here, note that this algorithm may suffer from numerical overflow.

Computation of Fibonacci number

Problem P2: Compute the Fibonacci number F(n) defined by

$F(n) = F(n-1) + F(n-2)$, if $n>1$,

$F(0) = F(1) = 1$.

Fibonacci number:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Exercise E2: Have an argument similar to that in Problem 1.

Supplemental information:

Using the golden ration $\phi = (1+\sqrt{5})/2 \fallingdotseq 1.61803$,

the Fibonacci number F(n) can be represented as

$F(n) = O(\phi^n)$.

Longest Common Subsequence

Problem P3: Given two strings A and B of lengths n and m, find the longest substring common to both of them.

Example: For A = G A A T T C A G T T A and B= G G A T C G A, the longest common substring is GATCA.

A= GAATTC AGTTA

B=GGA T CGA

Any substring A' of A is a substring of B if characters of A' appear in the same order in the string B.

➔It can be determined in linear time.

Exercise E3: Write a program to determine whether the first string of two input strings is a substring of the second string in linear time.

Algorithm P3-A0: (Brute-Force Algorithm)
For each substring A' of a string A, determine whether A' is a substring of a string B, and finally output the longest common substring.

Analysis of computation time:

- There are 2^n different substrings of a string of length n.
- If this substring is longer than the string B, obviously it is not a substring of B.
- Otherwise, each test takes O(m) time.
- Thus, the total time is $O(2^n m)$ time.

Is it possible to have faster algorithm?
Is there any polynomial-time algorithm?

Algorithm P3-A1:

$A = a_1 a_2 \ldots a_n$, $B = b_1 b_2 \ldots b_m$

$L[i,j]$ = the length of the longest substring common to $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$

Observation:

(0) if $i=0$ or $j=0$, $L[i,j]=0$.

(1) $a_i = b_j \rightarrow L[i,j] = L[i-1,j-1]+1$

(2) $a_i \neq b_j \rightarrow L[i,j] = \max\{ L[i,j-1], L[i-1, j]\}$

A=GAATTC AGTTA
B= GGATC GA
when $a_i = b_j$

A=GAATTC AGTTA
B=GGATCG A
when $a_i \neq b_j$

Therefore, it suffices to fill in the table L[i,j] in order.

Since the table size is $n \times m$, it takes O(nm) time.

Algorithm P3-A1:

```
for(i=0; i<=n; i++)
   L[i][0]=0;
for(j=0; j<=m; j++)
   L[0][ j] = 0;
for(i=1; i<=n; i++)
   for(j=1; j<=m; j++)
      if( a[i] == b[j]) L[i][j] = L[i-1][j-1]+1;
      else L[i][j] = max{ L[i][j-1], L[i-1][j] };
return L[n][m];
```

Example for the case
A=XYXXZXYZXY and
B=ZXZYYZXXYXXZ

A= X Y XXZXYZXY,
B=ZXZYYZXXYXXZ

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3	3	3	3
4	0	0	1	1	2	2	2	3	4	4	4	4	4
5	0	1	1	2	2	2	3	3	4	4	4	4	5
6	0	1	2	2	2	2	3	4	4	4	5	5	5
7	0	1	2	2	3	3	3	4	4	5	5	5	5
8	0	1	2	3	3	3	4	4	4	5	5	5	6
9	0	1	2	3	3	3	4	4	5	5	6	6	6
10	0	1	2	3	4	4	4	5	5	6	6	6	6

Construction of an optimal solution

The value of an optimal solution is obtained by filling in the table. How can we construct an optimal solution achieving the value?

In the problem of finding the longest common substring, we want to find not only the length (value of an optimal solution) but also the longest such substring (optimal solution) itself.

When we fill in the table, we memorize which table element determined L[i][j].

(1) $a_i = b_j \rightarrow L[i,j] = L[i-1,j-1]+1$

(i-1, j-1) is memorized

(2) $a_i \neq b_j \rightarrow L[i,j] = \max\{ L[i,j-1], L[i-1, j]\}$

if $L[i,j-1] > L[i-1,j]$ then (i, j-1) is memorized, and otherwise, (i-1, j) is memorized.

Concrete program

```
for(i=1; i<n; i++)
  for(j=1; j<m; j++){
    if( A[i] == B[j] ){
      L[i][j] = L[i-1][j-1] + 1;
      B1[i][j] = i-1; B2[i][j] = j-1;
    } else {
       L[i][j] = max2(L[i][j-1], L[i-1][j]);
       if( L[i][j-1] > L[i-1][j] ){
         L[i][j] = L[i][j-1];
         B1[i][j] = B1[i][j-1]; B2[i][j] = B2[i][j-1];
      } else {
         L[i][j] = L[i-1][j];
         B1[i][j] = B1[i-1][j]; B2[i][j] = B2[i-1][j];
      }
    }
  }
```

Exercise E4 : Write a program in practice to see its behavior.

For the case: A=XYXXZXYZXY, B=ZXZYYZXXYXXZ

Table for backtrack

	1	2	3	4	5	6	7	8	9	10	11	12
1	(0,0)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,6)	(0,7)	(0,7)	(0,9)	(0,10)	(0,10)
2	(0,0)	(0,1)	(0,1)	(1,3)	(1,4)	(1,4)	(1,4)	(1,4)	(1,8)	(1,8)	(1,8)	(1,8)
3	(0,0)	(2,1)	(0,1)	(1,3)	(1,4)	(1,4)	(2,6)	(2,7)	(2,7)	(2,9)	(2,10)	(2,10)
4	(0,0)	(3,1)	(0,1)	(1,3)	(1,4)	(1,4)	(3,6)	(3,7)	(3,7)	(3,9)	(3,10)	(3,10)
5	(4,0)	(3,1)	(4,2)	(1,3)	(1,4)	(4,5)	(3,6)	(3,7)	(3,7)	(3,9)	(3,10)	(4,11)
6	(4,0)	(5,1)	(4,2)	(1,3)	(1,4)	(4,5)	(5,6)	(5,7)	(3,7)	(5,9)	(5,10)	(4,11)
7	(4,0)	(5,1)	(4,2)	(6,3)	(6,4)	(4,5)	(5,6)	(5,7)	(6,8)	(5,9)	(5,10)	(4,11)
8	(7,0)	(5,1)	(7,2)	(6,3)	(6,4)	(7,5)	(5,6)	(5,7)	(6,8)	(5,9)	(5,10)	(7,11)
9	(7,0)	(8,1)	(7,2)	(6,3)	(6,4)	(7,5)	(8,6)	(8,7)	(6,8)	(8,9)	(8,10)	(7,11)
10	(7,0)	(8,1)	(7,2)	(9,3)	(9,4)	(7,5)	(8,6)	(8,7)	(9,8)	(8,9)	(8,10)	(7,11)

If we trace the table from L[10][12] in reverse order,

L[10][12] →L[7][11]→L[5][10]→L[3][9]→L[2][7]→L[1][4]→L[0][1]

a[8]b[12] a[6]b[11] a[4]b[10] a[3]b[8] a[2]b[5] a[1]b[2]

Thus, the longest common substring is

123456789A 123456789ABC

XYXXZXYZXY ZXZYYZXXYXXZ XYXXXZ

Problem P4: (Knapsack Problem)
Given n objects o_i (i=1, ... , n) and their weights w_i, prices v_i, and the capacity (or weight limit) C of a knapsack, find an optimal way of packing objects into the knapsack to meet the capacity constraint in such a way that the total price is maximized.

Input: $I = \{w_1, \dots, w_n; v_1, \dots, v_n; C\}$. A solution is represented by a subset S of {1,2,...,n}.
An optimal solution is such a set S satisfying the

Capacity constraint $\sum_{i \in S} w_i \leqq C$

and maximizing

total sum of prices $\sum_{i \in S} v_i$.

Assumption: Assume that weight of any object does not exceed the capacity C because any object with weight exceeding C is never selected.

Example： Consider the case in which $(w_1, ..., w_5)=(2,3,4,5,6)$, $(v_1, ..., v_5)=(4,5,8,9,11)$, $C=10$.

$V[k]$ = value of an optimal solution for objects up to the k-th one.

Then, by the definition

$V[1] \leqq V[2] \leqq \cdots \leqq V[n]$.

In this example, we have

$V[1] = v_1=4, w_1=2 \leqq C,$

$V[2] = v_1+v_2=4+5=9, w_1+w_2=2+3 \leqq C,$

$V[3] = v_1+v_2+v_3=4+5+8=17, w_1+w_2+w_3=2+3+4 \leqq C,$

$V[4] = v_1+v_2+v_4=4+5+9=18, w_1+w_2+w_4=2+3+5 \leqq C,$

$V[5] =v_3+v_5=8+11=19, w_3+w_5=4+6 \leqq C.$

Here, {1,2,3,4} is not a solution since the total weight exceeds the capacity 10.

In this example, an optimal solution to a subproblem may not be included in an optimal solution. Thus, we cannot apply Dynamic Programming to find a solution in the above order.

Then, what about a method to examine all possible ways of choosing objects?

For each object there are two ways, to choose or not to choose.
⇒there are 2^n ways to choose objects.
It takes exponential time if we examine all possible cases.

To apply Dynamic Programming, an optimal solution must be defined recursively so that it includes a solution to a subproblem.

D[i,j] = the largest total price among all possible ways to choose objects from objects 1, ... , i so that the total weight is j.
It is 0 if there is no way to choose them so that the total weight is j.

If an optimal solutions for objects 1,...,i-1 is known, we just consider two cases, to add an object i and not to add it. Thus, we have

$D[i,j] = \max\{D[i-1, j], D[i-1,j-w_i]+v_i\}$

This implies the property of **Optimal Substructure**.

Example： Let $(w_1, \ldots, w_5)=(2,3,4,5,6)$, $(v_1, \ldots, v_5)=(4,5,8,9,11)$, C=10.

i=1 ➔ only two ways to choose object 1 or not choose it:

$D[1,w_1]=D[1,2]=v_1=4$, $D[1,j]=0$, $j \neq 2$,

i=2 ➔ there are four cases: {}, {1}, {2}, {1,2}

$D[2,2]=4$, $D[2,3]=5$, $D[2,5]=9$, $D[2,j]=0$ $j \neq 2,3,5$

k	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0
1	4								
2	4	5		9					
3	4	5	8	9	12	13		17	
4	4	5	8	9	12	13	14	17	18
5	4	5	8	9	12	13	15	17	19

(Blue square) indicates a new solution

$w_1=2, v_1=4$

$w_2=3, v_2=5$

$w_3=4, v_3=8$

$w_4=5, v_4=9$

$w_5=6, v_5=11$

We can ignore a set of objects if their total weight exceeds 10.

Algorithm P4-A0:

```
Input : n objects o_i(i=1, … , n): weight w_i and price v_i,  capacity C.
for(i=1; i<=C; i++)
    D[0,i] = 0;
for(k=1; k<=n; k++)
  for(i=1; i<=C; i++)
    if(i < w_i) D[k,i] = D[k-1,i];
    else {
      if(D[k-1,i-w_i]+v_i > D[k-1,i])
        D[k,i] = D[k-1,i-w_i]+v_i;
      else
        D[k,i] = D[k-1, i];
    }
max=0;
for(i=1; i<=C; i++)
  if(D[n,i]>max) max = D[n,i];
return max;
```

Want to construct an optimal solution with the value of optimal solution.

We maintain not only the table D[i,j] but also the combination to give the value of D[i,j].

$D[i,j] = \max\{D[i, j-1], D[i-w_j,j-1]+v_j\}$
$T[i,j] = j \quad \text{if } D[i,j]=D[i-w_j,j-1]+v_j$
$T[i,j] = 0 \quad \text{if } D[i,j]=D[i,j-1]$

Then, we can construct an optimal solution by tracing back the value of D from D[i,n] giving the optimal solution.

k	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0
1	4/1								
2	4/0	5/2		9/2					
3	4/0	5/0	8/3	9/0	12/3	13/3		17/3	
4	4/0	5/0	8/0	9/0	12/0	13/0	14/4	17/0	18/4
5	4/0	5/0	8/0	9/0	12/0	13/0	15/5	17/0	19/5

values of D[i,j]/T[i,j]

k	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0
1	4/1								
2	4/0	5/2		9/2					
3	4/0	5/0	8/3	9/0	12/3	13/3		17/3	
4	4/0	5/0	8/0	9/0	12/0	13/0	14/4	17/0	18/4
5	4/0	5/0	8/0	9/0	12/0	13/0	15/5	17/0	19/5

Values of D[i,j]/T[i,j]

The value of an optimal solution is given by D[5,10]=19.
D[5,10]=19, T[5,10]=5 ≠ 0, output object 5.
Since w_5=6, its predecessor is D[4,10-6]=D[4,4],
D[4,4]=8, T[4,4]=0, output nothing. The predecessor is D[3,4]=8.
D[3,4]=8, T[3,4]=3 ≠ 0, output object 3.
Since w_3=4, its predecessor is D[2,4-4]=D[2,0].
Now the total weight becomes 0, and thus this is the end.
After all, the set of objects for an optimal solution is {3,5}.

Algorithm P4-A1:

Input：n objects o_i(i=1, … , n): weight w_i and price v_i, capacity C.

```
for(i=0; i<=C; i++)
     D[0,i] = T[0,i]=0;
for(k=1; k<=n; k++)
  for(i=1; i<=C; i++)
     if(i < w_k){ D[k,i] = D[k-1,i]; T[k,i]=0;}
     else {
        if(D[k-1,i-w_k]+v_k > D[k-1,i])
           {D[k,i] = D[k-1,i-w_k]+v_k; T[k,i]=k;}
        else
           {D[k,i] = D[k-1, i]; T[k,i]=0;}
     }
k=0;
for(i=1; i<=C; i++)
  if(D[n,i]>D[n, k]) k = i;
for(i=n; i>0 && k>0; i--)
  if( T[i,k] > 0) {
      Output T[i,k]; k = k - w_i;
  }
```

Analysis of Computation Time

From the structure of the algorithm the computation time is given by

$O(nC)$.

(1) If the capacity C is polynomial in the number n of objects

⇒ this computation time is a polynomial in n.

(2) If C is much larger than n.

The value C itself can be represented by log C bits.

⇒Time is proportional to an exponential function in input size.

It is called **a pseudo-polynomial time algorithm.**

Exercise E5：Algorithm P4-A1 uses two 2-dimensional arrays. Show that one of them cab be replaced by a one-dimensional array.

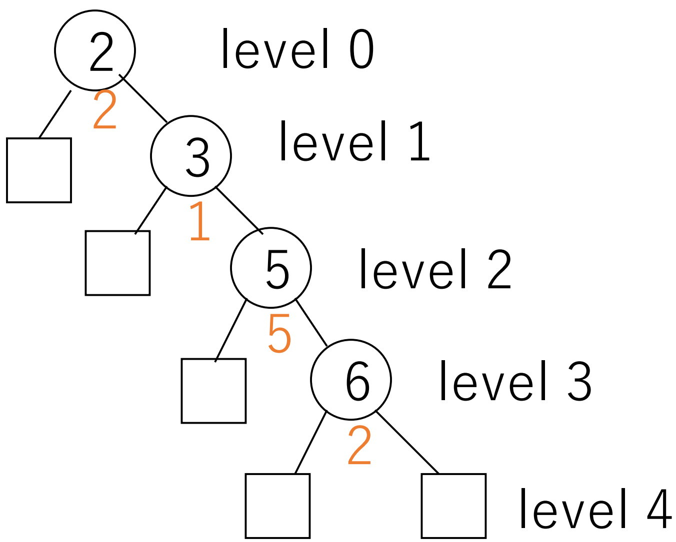
Problem P5: (Construction of an optimal binary search tree)
When probability that each element is asked is given, store n data in a binary search tree so that the expected number of comparisons to locate a query in the tree is minimized.

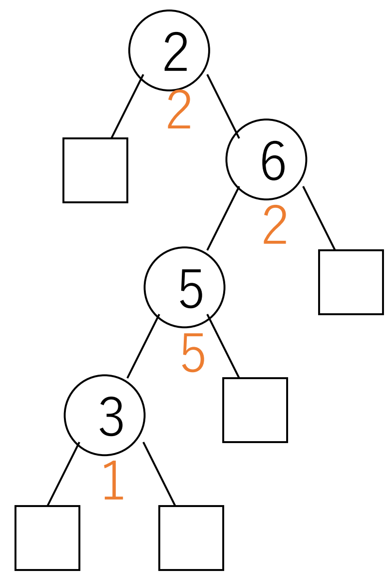
Data to be stored： $S = \{a_1, a_2, \ldots, a_n\}$, $a_1 \leqq a_2 \leqq \cdots \leqq a_n$
A priori knowledge：Assume that only elements of S are retrieved.
probability for Find(a_i, S) is p_i
When S is stored in a binary search tree,
let the level of a node a_i containing an element of S be level(a_i).
the number of comparisons for searching a_i is level(a_i) +1
(assuming the level of the root node is 0)
Therefore, the cost of a search tree (expected number of comparisons) is given by

$$\text{Cost of search tree} = \sum p_i \times [\text{level}(a_i) + 1]$$

Examlpe：

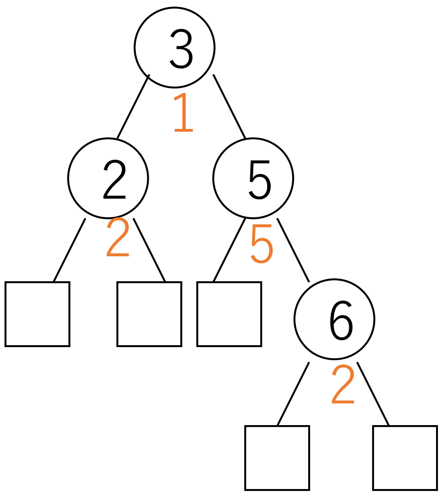
	a[1]	a[2]	a[3]	a[4]
S	2	3	5	6
	2/10	1/10	5/10	2/10
	p_1	p_2	p_3	p_4

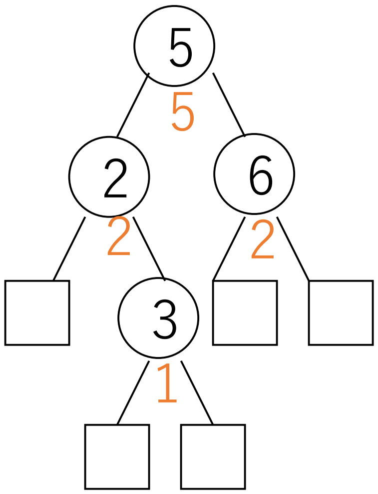




cost=(2*1+1*2+5*3+2*4)/10
=2.7

cost=(2*1+2*2+5*3+1*4)/10
= 2.5





cost=(1*1+(2+5)*2+2*3)/10 =2.1

cost=(5*1+(2+2)*2+1*3)/10 = 1.6

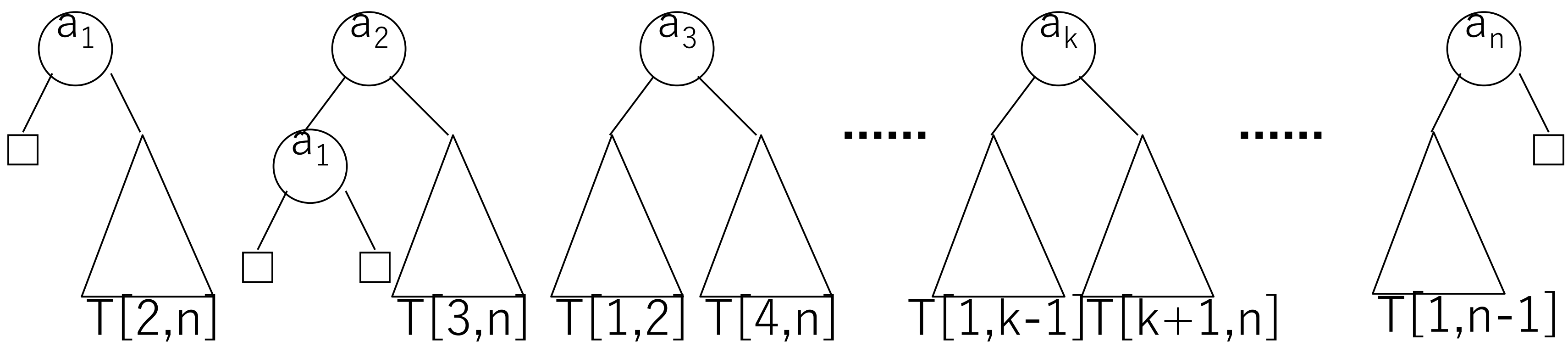
If we enumerate all search trees and compute their costs, then we can find an optimal search tree. But it is not efficient to enumerate all of them.

Construction of an optimal binary search tree

Characterize structure of an optimal solution and define the value of an optimal solution recursively.

$T[i,j]$ = minimum-cost tree for a subset $\{a_i, a_{i+1}, \ldots, a_j\}$
$i=1, \ldots, n, \quad j=i, i+1, \ldots, n$

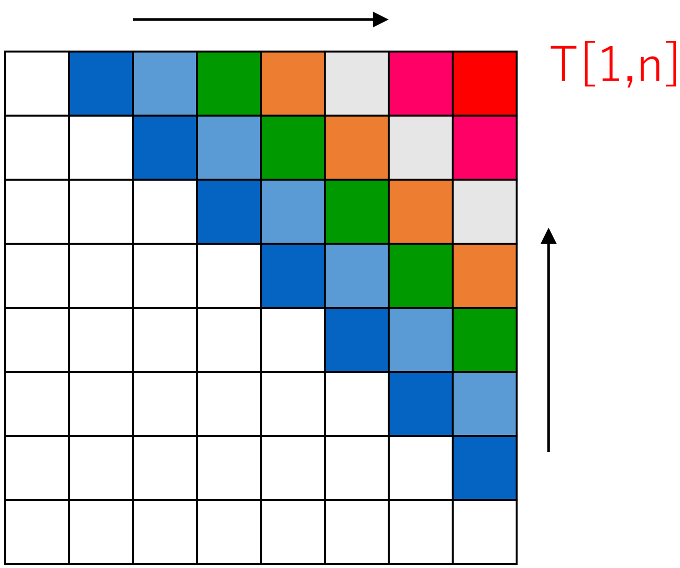
Enumerating all possibilities:



If T[2,n], T[3,n], … , T[1,2], T[4,n], … , T[1,n-1] are all available, costs of those trees can be computed. If we choose the minimum-cost tree, we can determine its root a_k.

Computing the value of optimal solution in a bottom-up fashion

{T[i, i+1], i=1, 2, ... , n-1} is computed・・・・・difference 1
{T[i, i+2], i=1, 2, ... , n-2} is computed・・・・・difference 2
:
{T[i, i+k], i=1, 2, ... , n-k} is computed・・・・・difference1 k
:
Finally, we compute T[1, n], which is the value of optimal solution.



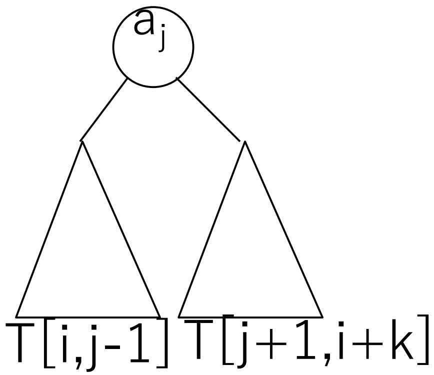
How to compute T[i, i+k]

T[i,i+k] = min-cost tree for a subset $\{a_i, a_{i+1}, \dots, a_{i+k}\}$. Thus, k+1 different roots $a_i, a_{i+1}, \dots, a_{i+k}$ are possible.

If we choose a_j as a root,

an optimal solution has T[i,j-1] as its left subtree and T[j+1,i+k] as right subtree.

Note that one level is increases than when computing the costs for T[i,j-1] and T[j+1,i+k].



$T[i,j-1] = \Sigma\ p_m \times [\text{level}\ (a_m) + 1]$

If we increase the level by one,

$T'[i,j-1] = \Sigma\ p_m \times [\text{level}\ (a_m) + 2] = T[i,j-1] + \Sigma\ p_m$

That is, we have the value one level down by adding $p_i + p_{i+1} + \dots + p_{j-1}$ to T[i,j-1]. Same for T'[j+1,i+k].

Thus, the cost with a_j at the root is given by：

$$p_j + T'[i,j-1] + T'[j+1,i+k]$$
$$= T[i,j-1] + T[j+1,i+k] + p_i + p_{i+1} + \dots + p_{j+k}$$

How to compute T[i, i+k]

C[i,j] = cost of the minimum-cost tree T[i,j] for $\{a_i, a_{i+1}, \dots, a_j\}$
W[i,j] = $p_i + p_{i+1} + \dots + p_j$
Then,

the cost when a_j is the root is given by the following:
C[i,j-1]+C[j+1,i+k]+ W[i, i+k]

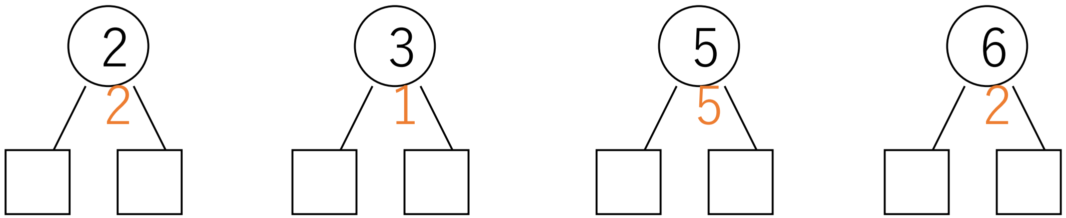
C[i,i+k] is obtained by taking the minimum value while varying j.
Considering the cases where a_i and a_{i+k} are roots, we have

$$C[i,i+k] = \min\{ C[i+1,i+k]+W[i,i+k], \min\{C[i,j-1]+C[j+1,i+k]+W[i,i+k],\ j=i+1, \dots, i+k-1\}, C[i,i+k-1]+W[i,i+k]\}$$

k=1, 2, … , n-i

C[i,j] = cost of minimum-cost tree T[i,j] for $\{a_i, a_{i+1}, \ldots, a_j\}$

W[i,j] = $p_i + p_{i+1} + \ldots + p_j$

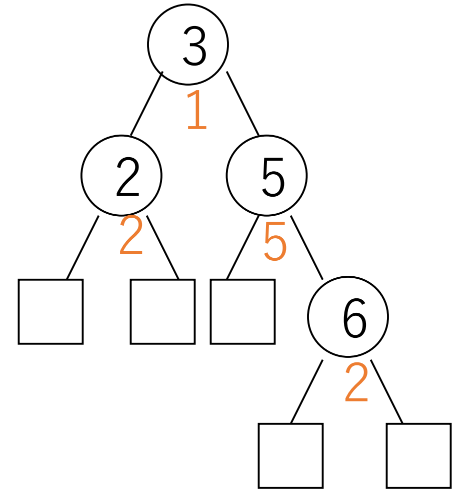


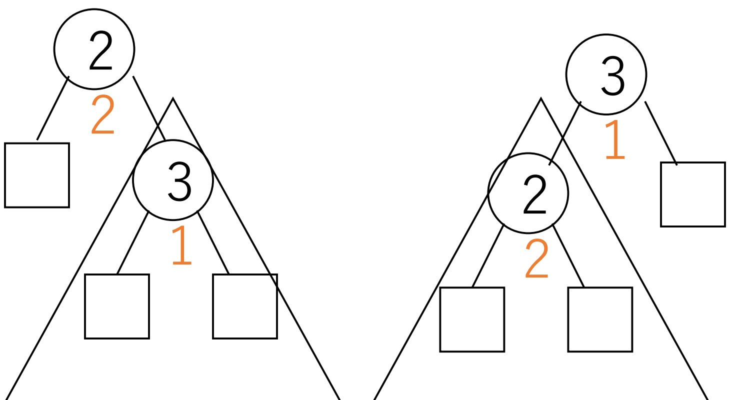
T[1,1] C[1,1]=0.2

T[2,2] C[2,2]=0.1

T[3,3] C[3,3]=0.5

T[4,4] C[4,4]=0.2





T[2,2]

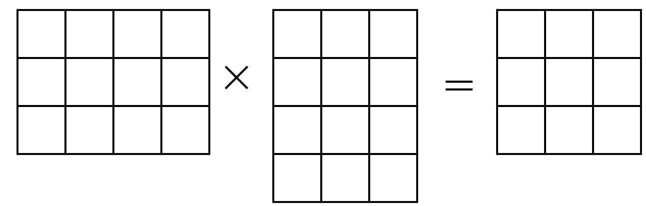
T[1,1]

Cost
=0.2+C[2,2]+W[2,2]
=0.2+0.1+0.1=0.4

Cost
=0.1+C[1,1]+W[1,1]
=0.1+0.2+0.2=0.5

Problem P6: （Chained Matrix Product）
Given a sequence of n matrices <$A_1, A_2, \ldots, A_n$>, find an order of matrix products to minimize the number of operations to compute the matrix product $A_1 \times A_2 \times \ldots \times A_n$.

Product of a $p \times q$ matrix and $q \times r$ matrix is a $p \times r$ matrix using $p \times q \times r$ operations (multiplication and additioN).



3×4 4×3 3×3

Example： $A_1 = 10 \times 20$ matrix, $A_2 = 20 \times 5$ matrix, $A_3 = 5 \times 25$ matrix.
$((A_1 \times A_2) \times A_3)$ require $(10 \times 20 \times 5) + (10 \times 5 \times 25) = 2250$ ops.
$(A_1 \times (A_2 \times A_3))$ requires $(10 \times 20 \times 25) + (20 \times 5 \times 25) = 7500$ ops.
Thus, the former needs less operations.

For product of four matrices, there are many orders for their product.

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

It suffices to obtain the number of operations for all of

Precisely,

$$\frac{1}{n+1}\binom{2n}{n}$$

Exercise E6：Prove that there are $O(4^n/n^{3/2})$ ways for parenthe-sizations. This is known as the Catalan number.
Hint: Suppose there are P(n) ways for parenthesization. In each sequence we can parenthesize it by dividing it between its k-th and (k+1)-st position into subsequences independently. Thus, we have

$P(1) = 1$

$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$

Due to Richard P. Stanley, the Catalan number has 207 representations!
(http://www-math.mit.edu/~rstan/ec/)

Characterize structure of an optimal solution and define the value of an optimal solution recursively.

To compute the product of 4 matrixes

$((A_1 \times (A_2 \times A_3)) \times A_4)$ last is the product of (A_1, A_2, A_3) and A_4

$(((A_1 \times A_2) \times A_3) \times A_4)$ last is the product of (A_1, A_2, A_3) and A_4

$((A_1 \times A_2) \times (A_3 \times A_4))$ last is the product of (A_1, A_2) and (A_3, A_4)

$(A_1 \times ((A_2 \times A_3) \times A_4))$ last is the product of A_1 and (A_2, A_3, A_4)

$(A_1 \times (A_2 \times (A_3 \times A_4)))$ last is the product of A_1 and (A_2, A_3, A_4)

If we know an optimal orders for subsequences, it suffices to check the three ways of partitions.

$((A_1, A_2, A_3), A_4)$, $((A_1, A_2), (A_3, A_4))$, $(A_1, (A_2, A_3, A_4))$

Generally, the problem is the place for the first partition.

$((A_1, \ldots, A_k), (A_{k+1}, \ldots, A_n))$ $k = 1, 2, \ldots, n-1$

If we know an optimal order for computation for each subsequence, then an optimal order for computation is obtained.

Let the size of each matrix be $p_i \times q_i$. Then, only if we have

$q_1=p_2, q_2=p_3, \ldots, q_n=p_{n+1}$

the product of those matrices is defined. Thus, we only specify $p_1, p_2, \ldots, p_n, p_{n+1}$ for input.

If we take the product of matrices from A_i to A_j then the $p_i \times q_j=p_{j+1}$ matrix is obtained.

M[i,j] = the smallest number of computations to calculate the product of matrices from A_i to A_j. For the computation it suffices to evaluate all possible productions of matrices from A_i to A_k and those from A_{k+1} to A_j for each k between i and j.

The product for A_i through A_k is a $p_i \times p_{k+1}$ matrix, and that for A_{k+1} through A_j is a $p_{k+1} \times p_{j+1}$ matrix.

Thus, the number of operations we need to compute them is

$p_i p_{k+1} p_{j+1}$.

Therefore, the recurrence equation for M[i,j] is

$$M[i, j] = \min\{M[i,k]+M[k+1,j]+p_i p_{k+1} p_{j+1}, k=i,i+1, \ldots, j-1\}.$$

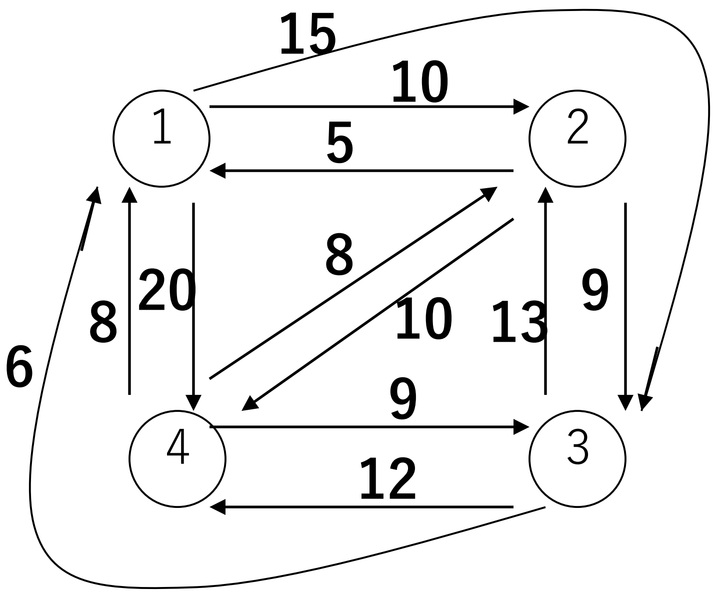
algorithm P6-A0:

```
input : matrix sizes (p_1 rows p_2 columns),(p_2, p_3), ... ,(p_n, p_{n+1}).
for(i=1; i<=n; i++)
    M[i,i] = 0;
for(d=1; d<=n; d++)
  for(i=1; i<=n-d; i++)
    j=i+d;
    msf = M[i,i]+M[i+1,j]+p_i p_{i+1} p_{j+1};
    for(k=i; k<j; k++)
       if( M[i,k]+M[k+1,j]+p_i p_{k+1} p_{j+1} < msf)
          msf =  M[i,k]+M[k+1,j]+p_i p_{k+1} p_{j+1};
     M[i,j] = msf;
  }
return M[1,n];
```

Exercise E7 : The above algorithm only finds the value of an optimal solution. Modify it so that an optimal order of computation is also obtained.

Problem P7: （Travelling-Salesperson Problem）

Given a weighted graph for a road network interconnecting n cities, find a shortest closed tour starting from a city and coming back to it after visiting every city.



$$L = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

L[i,j] is the distance between city i and city j.

tour(1,2,3,4,1) : length=10+9+12+8=39,
tour(1,2,4,3,1) : length=10+10+9+6=35,
tour(1,3,2,4,1) : length =15+13+10+20=58,
tour(1,3,4,2,1) : length =15+12+8+5=40, etc.

How many tours are there in total?

If there is a road between any two cities, then there are (n-1)! tours.
Using the Stirling's formula, we have

$$n! \fallingdotseq \sqrt{(2\pi n)}(n/e)^n.$$

This is roughly an exponential function n^n.

Numbering the cities as 1, 2, … , n, and assume that we start at city 1 and come back to it.
The set of all cities: N={1, 2, … , n}. S is a subset of N.
For a subset S not containing city 1,

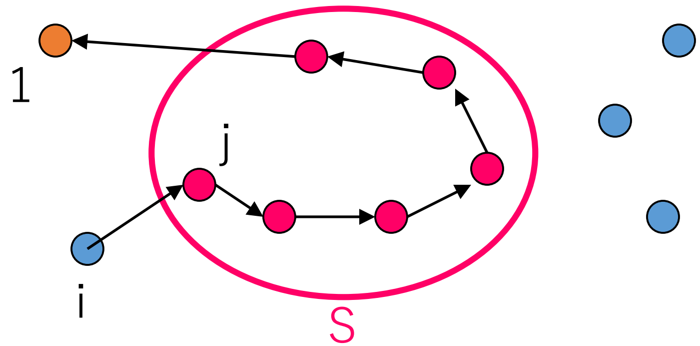
g(i, S) = the length of a shortest path from city i coming back to city 1 through every city of S.

Note that i does not belong to S.

Then, the length of the shortest tour is given as

g(1, {2, 3, … , n}).

Let’s define g(i, S) recursively!



To apply Dynamic Programming, an optimal solution must be defined recursively so that it includes a solution to a subproblem.

If j is a candidate among S for the city after city i, the length of an optimal path to city 1 is given by

$g(j, S-\{j\})$.

Thus, the recurrence equation for g(i, S) becomes

$g(i, S) = \min_{j \in S}\{L[i,j] + g(j, S-\{j\})\}$,

where i is not an element of S.

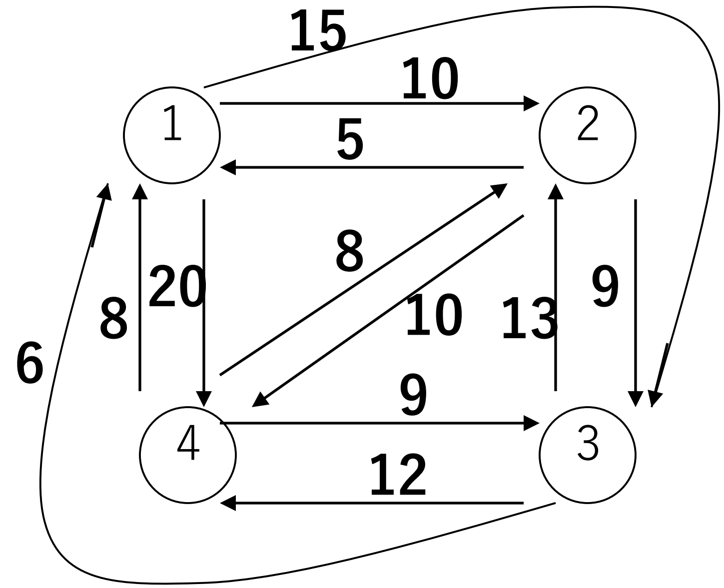
L[i,j] denotes the distance between city i and city j.

|S|=0

g(2, Φ)=L[2,1]=5
g(3, Φ)=L[3,1]=6
g(4, Φ)=L[4,1]=8

|S|=1

g(2, {3})=L[2,3]+g(3, Φ)=15
g(2, {4})=L[2,4]+g(4, Φ)=18
g(3, {2})=L[3,2]+g(2, Φ)=18
g(3, {4})=L[3,4]+g(4, Φ)=20
g(4, {2})=L[4,2]+g(4, Φ)=13
g(4, {3})=L[4,3]+g(3, Φ)=15



|S|=2

g(2, {3,4})=min{L[2,3]+g(3, {4}), L[2,4]+g(4,{3})}=min{29,25}=25
g(3, {2,4})=min{L[3,2]+g(2, {4}), L[3,4]+g(4,{2})}=min{31,25}=25
g(4, {2,3})=min{L[4,2]+g(2, {3}), L[4,3]+g(3,{2})}=min{23,27}=23

|S|=3

g(1, {2,3,4})=min{L[1,2]+g(2, {3,4}), L[1,3]+g(3, {2,4}),L[1,4]+g(4, {2,3})}
= min{35, 40, 43} = 35.

Therefore, the length of an optimal tour is 35.

Algorithm P7-A0:

Input: A graph representing distances among cities.

```
U={1, 2, ... , n}.
for(i=2; i<=n; i++)
  g(i, Φ)=L[1,i];
for(k=1; k<n; k++){
  for each subset S of size k not containing 1 {
    for each i not contained in S
      g(i, S) = min_{j∈S}{L[i,j] + g(j, S-{j})}
}
return g(1, {2, 3, ... , n});
```

Exercise E8： Express the computation time and amount of storage of the above algorithm as functions of n.
(Hint: It is not a polynomial, but it's better than n^n.)