Data Structures and Algorithms
Lecture notes: Algorithm paradigms: Dynamic programming

Lecturer: Michel Toulouse

Hanoi University of Science and Technology
michel.toulouse@soict.hust.edu.vn

22 octobre 2020

Outline

Problems with divide-and-conquer

During the divide part of divide-and-conquer some subproblems could could be generated more than one time.

If subproblems are duplicated, the computation of the solution of the subproblems is also duplicated, solving the same subproblems several times obviously yield very poor running times.

Example : D&C algo for computing the Fibonacci numbers

It is a sequence of integers. After the first two numbers in the sequence, each other number is the sum of the two previous numbers in the sequence :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$$

The following divide-and-conquer algorithm computes the first $n + 1$ values of the Fibonacci sequence :

```
function Fib(n)
  if (n ≤ 1) then return n ;
  else
    return Fib(n − 1) + Fib(n − 2)
```

Computing Fibonacci numbers

function $Fib(n)$
  if $(n \leq 1)$ **then return** n;
  else
    return(Fib$(n-1)$ + Fib$(n-2)$);

First the following divide steps take place :

$$
\begin{aligned}
Fib(7) &= Fib(6) + Fib(5) \\
Fib(6) &= Fib(5) + Fib(4) \\
Fib(5) &= Fib(4) + Fib(3) \\
Fib(4) &= Fib(3) + Fib(2) \\
Fib(3) &= Fib(2) + Fib(1) \\
Fib(2) &= Fib(1) + Fib(0) \\
Fib(1) &= 1 \\
Fib(0) &= 0
\end{aligned}
$$

Computing Fibonacci numbers

```
function Fib(n)
        if (n ≤ 1) then return n ;
        else
          return Fib(n − 1) + Fib(n − 2) ;
```

Next solutions to sub-problems are combined to obtain the solution of the original problem instance :

$$
\begin{array}{lcl}
Fib(0) & = & 0 \\
Fib(1) & = & 1 \\
Fib(2) & = & Fib(1) + Fib(0) = 1 + 0 = 1 \\
Fib(3) & = & Fib(2) + Fib(1) = 1 + 1 = 2 \\
Fib(4) & = & Fib(3) + Fib(2) = 2 + 1 = 3 \\
Fib(5) & = & Fib(4) + Fib(3) = 3 + 2 = 5 \\
Fib(6) & = & Fib(5) + Fib(4) = 5 + 3 = 8 \\
Fib(7) & = & Fib(6) + Fib(5) = 8 + 5 = 13
\end{array}
$$

Time complexity of DC Fibonacci

```
function Fib(n)
        if (n ≤ 1) then return n ;
        else
          return Fib(n − 1) + Fib(n − 2) ;
```

One possible recurrence relation for this algorithm that counts the number of time the function *Fib* is called is the following :

$$\mathrm{T}(n) = \begin{cases} 1 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ \mathrm{T}(n-1) + \mathrm{T}(n-2) + 1 & \text{if } n > 1 \end{cases}$$

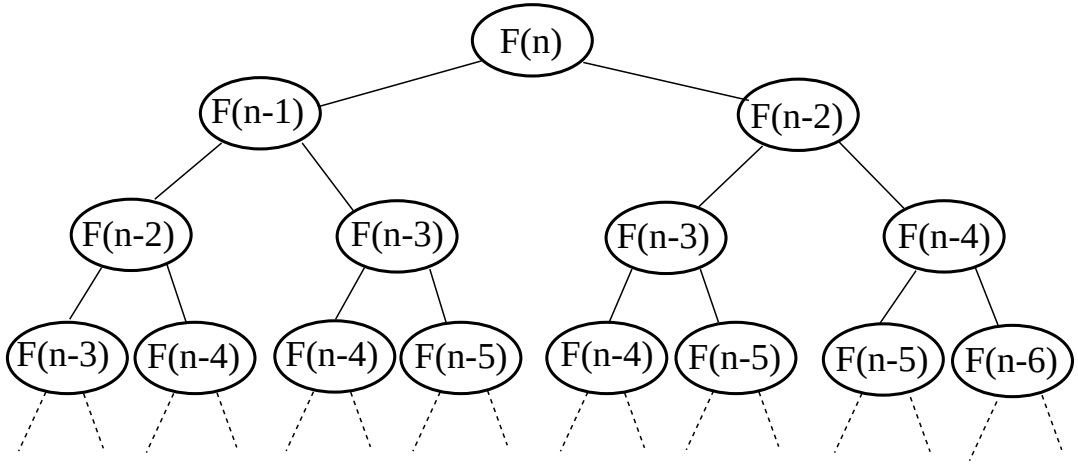
The solution of this recurrence is $O((\frac{1+\sqrt{5}}{2})^n)$, the time complexity is exponential.

Call tree of D&C Fibonacci

function *Fib*(n)
  if ($n \leq 1$) **then return** n ;
  else
    return Fib($n-1$) + Fib($n-2$) ;

The poor time complexity of *Fib* derived from re-solving the same sub-problems several times.

D&C generates the following call tree :



Dynamic Programming

Dynamic programming algorithms avoid recomputing the solution of same subproblems by storing the solution of subproblems the first time they are computed, and referring to the stored solution when needed.

So, instead of solving the same subproblem repeatedly, arrange to solve each subproblem only one time

Save the solution to a subproblem in a table (an array), and refer back to the table whenever we revisit the subproblem

Dynamic programming main ideas

Instead of solving the same subproblem repeatedly, arrange to solve each subproblem only one time

Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem

"Store, don't recompute"

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

- here computing Fib(4) and Fib(5) both require Fib(3), but Fib(3) is computed only once

Can turn an exponential-time solution into a polynomial-time solution

Designing dynamic programming algorithms

There are two dynamic programming algorithm design options :

- Top down : start to compute with the whole problem
- Bottom up : start to compute with the D&C base case

Top-down dynamic programming for Fibonacci

A top-down design is a recursive algorithm where the original problem is decomposed into smaller problems and once the base cases are solved, solutions to subproblems are computed out of results in the table :

```
function DyFib(n)
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (table[n] != 0) return table[n];
  else
    table[n] = DyFib(n − 1) + DyFib(n − 2)
    return table[n];
```

input	0	1	2	3	4	5	6	7	8	9	10	11	12	13
solution	0	1												

Bottom up dynamic programming for Fibonacci

Bottom up design is an iterative algorithm which first compute the base cases and then uses the solutions to the base cases to start computing the solutions to the other larger subproblems :

```
function fib_dyn(n)
        int *table, i ;
        table = malloc((n + 1) * sizeof(int)) ;
        for (i = 0; i ≤ n; i + +)
            if (i ≤ 1)
                table[i] = i ;
            else
                table[i] = table[i − 1] + table[i − 2] ;
        return f [n] ;
```

input	0	1	2	3	4	5	6	7	8	9	10	11	12	13
solution	0	1												

$fib\_dyn \in \Theta(n)$ as opposed to the exponential complexity $O((\frac{1+\sqrt{5}}{2})^n)$ for fib_rec.

When do we need DP

Before writing a dynamic programming algorithm, first do the following :

- Write a divide-and-conquer algorithm to solve the problem
- Next, analyze its running time, if it is exponential then :
  - it is likely that the divide-and-conquer generates a large number of identical subproblems
  - therefore solving many times the same subproblems

If D&C has poor running times, we can consider DP.

But successful application of DP requires that the problem satisfies some conditions, which will be introduced later...

Writing a DP algorithm : the bottom-up approach

Writing a DP often starts by writing a D&C recursive algorithm and use the D&C recursive code to write an iterative bottom up DP algorithm :

- Create a table that will store the solution of the subproblems
- Use the "base case" of recursive D&C to initialize the table
- Devise look-up template using the recursive calls of the D&C algorithm
- Devise for-loops that fill the table using look-up template
- The function containing the for loop returns the last entry that has been filled in the table.

An example : making change

Devise an algorithm for paying back a customer a certain amount using the smallest possible number of coins.

For example, what is the smallest amount of coins needed to pay back \$2.89 (289 cents) using as denominations "one dollars", "quaters", "dimes" and "pennies".

The solution is 10 coins, i.e. 2 one dollars, 3 quaters, 1 dime and 4 pennies.

Making change : a recursive solution

Assume we have an infinite supply of n different denominations of coins.

A coin of denomination i worth d_i units, $1 \leq i \leq n$. We need to return change for N units.

Here is a recursive D&C algorithm for returning change :

function Make_Change(i,j)
  if $(j == 0)$ **then return** 0 ;
  else
    return min(make_change($i-1, j$), make_change($i, j-d_i$) + 1) ;

The function is called initially as Make_Change(n,N).

C code implementing recursive make_change

```
#include <stdio.h>
#define min(a,b)((a<b)? a:b)
int make_change(int d[], int n, int N)
{
  if(N == 0) return 0;
  else if (N < 0 || (N > 0 && n <= 0)) return 1000;
  else{
    return min(make_change(d,n-1,N), make_change(d,n,N-d[n-1]) + 1);
  }
}
int main()
{
    int d[] = {1, 5, 10, 25};
    int N = 13;
    int n = sizeof(d)/sizeof(d[0]);
    int ans = make_change(d, n, N);
    printf("Minimal # of coins = %d\n",ans);
    return 0;
}
```

Making change : DP approach

Assume $n = 3$, $d_1 = 1$, $d_2 = 4$ and $d_3 = 6$. Let $N = 8$.

To solve this problem by dynamic programming we set up a table $t[1..n, 0..N]$, one row for each denomination and one column for each amount from 0 unit to N units.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$									
$d_2 = 4$									
$d_3 = 6$									

The entry $t[i, j]$ indicates the minimum number of coins needed to refund an amount of j units using only coins from denominations 1 to i.

Making change : DP approach

The initialization of the table is obtained from the D&C base case :

if $(j == 0)$ **then return** 0

i.e. $t[i, 0] = 0$, for $i = 1, 2, 3$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0								
$d_2 = 4$	0								
$d_3 = 6$	0								

Making change : DP approach

If $i = 1$, then only denomination $d_1 = 1$ can be used to return change.

Therefore $t[1, j]$ for $j = 1..8$ is $t[i, j] = t[i, j - d_i] + 1$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0								
$d_3 = 6$	0								

For example, the content of entry $t[1, 4] = t[1, 3] + 1$ means that the minimum number of coins to return 4 units using only denomination 1 is the minimum number of coins to return 3 units + 1 = 4 coins.

Making change : DP approach

If the amount of change to return is smaller than domination d_i, then the change needs to be return using denominations smaller than d_i

For those cases, i.e. if $(j < d_i)$ then $t[i, j] = t[i - 1, j]$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3					
$d_3 = 6$	0	1	2	3	1	2			

For all the other entries of the table we write the code of the DP algorithm using the recursive function

Designing DP from the D&C recursive calls

Here we look at the recursive call of the D&C to design the table look-up of the DP algorithm :

```
function  Make_Change(i,j)
  if (j == 0) then return 0;
  else
    return min(make_change(i − 1, j), make_change(i, j − d_i)+ 1);
```

The recursive call

$$min(make\_change(i-1,j), make\_change(i,j-d_i)+1)$$

indicates two options to fill entry $t[i,j]$:

1. Don't use a coin from d_i, then $t[i,j] = t[i-1,j]$
2. Use at least one coin from d_i, then $t[i,j] = t[i,j-d_i]+1$.

We deduce the following table look-ups :

$$t[i,j] = \min(t[i-1,j], t[i,j-d_i]+1)$$

DP bottom-up algorithm

```
coins(n, N)
  int d[1..n] = d[1, 4, 6];
  int t[1..n, 0..N];
  for (i = 1; i ≤ n; i++) t[i, 0] = 0; */base case */
  for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ N; j++)
      if (i == 1) then t[i, j] = t[i, j − d_i] + 1
      else if (j < d[i]) then t[i, j] = t[i − 1, j]
      else t[i, j] = min(t[i − 1, j], t[i, j − d[i]] + 1)
  return t[n, N];
```

The algorithm runs in $\Theta(nN)$.

Making change : DP approach

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

To fill entry $t[i, j], j > 0$, we have two choices :

1. Don't use a coin from d_i, then $t[i, j] = t[i - 1, j]$
2. Use at least one coin from d_i, then $t[i, j] = t[i, j - d_i] + 1$.

Since we seek to minimize the number of coins returned, we have

$$t[i, j] = \min(t[i - 1, j], t[i, j - d_i] + 1)$$

The solution is in entry $t[n, N]$

Making change : getting the coins

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

We can use the information in the table to get the list of coins that should be returned :

- Start at entry $t[n, N]$;
- If $t[i, j] = t[i - 1, j]$ then no coin of denomination i has been used to calculate $t[i, j]$, then move to entry $t[i - 1, j]$;
- If $t[i, j] = t[i, j - d_i] + 1$, then add one coin of denomination i and move to entry $t[i, j - d_i]$.

Exercises 1 and 2

1. Construct the table and solve the making change problem where $n = 3$ with denominations $d_1 = 1, d_2 = 2$ and $d_3 = 3$ where the amount of change to be returned is $N = 7$
2. Construct the table and solve the making change problem where $n = 4$ with denominations $d_1 = 1, d_2 = 3, d_3 = 4$ and $d_4 = 5$ where the amount of change to be returned is $N = 12$

Solutions table

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Each entry in the above table is a solution to a subproblem.

For example, entry $t[1, 3]$ is the solution to the subproblem where an amount of 3 needs to be returned using only coins of denomination 1.

Amount	0	1	2	3
$d_1 = 1$	0	1	2	3

Entry $t[2, 6]$ is the number of coins that have to be returned when an amount of 6 needs to be returned using only coins of denomination 1 or 4.

Amount	0	1	2	3	4	5	6
$d_1 = 1$	0	1	2	3	4	5	6
$d_2 = 4$	0	1	2	3	1	2	3

Optimization problems

DP is often used to solve optimization problems that have the following form

$$\begin{array}{ll} \min & f(x) \textbf{ or} \\ \max & f(x) \\ \text{s.t.} & \textit{some constraints} \end{array} \tag{1}$$

Making change is an optimization problem. The problem consists to minimize the function $f(x)$, i.e. to minimize the number of coins returned

There is only one constraint : the sum of the value of the coins is equal to the amount to be returned

Optimal Substructure

In solving optimization problems with DP, we find the optimal solution of a problem of size n by solving smaller problems of same type

The optimal solution of the original problem is made of optimal solutions from subproblems

Thus the subsolutions within an optimal solution are optimal subsolutions

Solutions to optimization problems that exhibit this property are say to be based on optimal substructures

Optimal Substructure

Make_Change() exhibits the optimal substructure property :

- Each entry $t[i, j]$ in the table is the optimal solution (minimum number of coins) that can be used to return an amount of j units using only denominations d_1 to d_i.
- The optimal solution of problem (i, j) is obtained using optimal solutions (minimum number of coins) of sub-problems $(i-1, j)$ and $(i, j-d_i)$.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

The optimal solution for $t[i, j]$ is obtained by comparing $t[i-1, j]$ and $t[i, j-d_i]+1$, taking the smallest of the two.

Optimal Substructure

To compute the optimal solution, we can compute all optimal subsolutions

Often we start with all optimal subsolutions of size 1, then compute all optimal subsolutions of size 2 combining some subsolutions of size 1. We continue in this fashion until we have the solution for n.

Note, not all optimization problems satisfy the optimal substructure property. When it fails to apply, we cannot use DP.

DP for optimization problems

The basic steps are :

1. Characterize the structure of an optimal solution, i.e. the problem meet the optimal substructure property
2. Give a recursive definition for computing the optimal solution based on optimal solutions of smaller problems.
3. Compute the optimal solutions and/or the value of the optimal solution in a bottom-up fashion.

Integer 0-1 Knapsack Problem

Given n objects with integer weights w_i and values v_i, you are asked to pack a knapsack with no more than W weight (W is integer) such that the load is as valuable as possible (maximize). You cannot take part of an object, you must either take an object or leave it out.

Example : Suppose we are given 4 objects with the following weights and values :

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

Suppose $W = 5$ units of weight in our knapsack.

Seek a load that maximize the value

Problem formulation

Given

- n integer weights $w_1, \ldots, w_n$,
- n values $v_1, \ldots, v_n$, and
- an integer capacity W,

assign either 0 or 1 to each of $x_1, \ldots, x_n$ so that the sum

$$f(x) = \sum_{i=1}^{n} x_i v_i$$

is maximized, s.t.

$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Explanation

$x_i = 1$ represents putting Object i into the knapsack and $x_i = 0$

represents leaving Object i out of the knapsack.

The value of the chosen load is $\sum_{i=1}^{n} x_i v_i$. We want the most valuable load, so we want to maximize this sum.

The weight of the chosen load is $\sum_{i=1}^{n} x_i w_i$. We can't carry more than W units of weight, so this sum must be $\leq W$.

Solving the 0-1 Knapsack

0-1 knapsack is an optimization problem.

Should we apply dynamic programming to solve it ? To answer this question we need to investigate two things :

1. Whether subproblems are solved repeatedly when using a recursive algorithm.
2. An optimal solution contains optimal sub-solutions, the problem exhibits optimal substructure

Optimal Substructure

Does integer 0-1 knapsack exhibits the optimal substructure property ?

Let $\{x_1, x_2, \ldots, x_k\}$ be the objects in an optimal solution x.

The optimal value is $V = v_{x_1} + v_{x_2} + \cdots + v_{x_k}$.

We must also have that $w_{x_1} + w_{x_2} + \cdots + w_{x_k} \leq W$ since x is a feasible solution.

Claim :

If $\{x_1, x_2, \ldots, x_k\}$ is an optimal solution to the knapsack problem with weight W, then $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Optimal Substructure

Proof : Assume $\{x_1, x_2, \ldots, x_{k-1}\}$ is not an optimal solution to the subproblem. Then there are objects $\{y_1, y_2, \ldots, y_l\}$ such that

$$w_{y_1} + w_{y_2} + \cdots + w_{y_l} \leq W',$$

and

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}}.$$

Then

$$v_{y_1} + v_{y_2} + \cdots + v_{y_l} + v_{x_k} > v_{x_1} + v_{x_2} + \cdots + v_{x_{k-1}} + v_{x_k}.$$

However, this implies that the set $\{x_1, x_2, \ldots, x_k\}$ is not an optimal solution to the knapsack problem with weight W.

This contradicts our assumption. Thus $\{x_1, x_2, \ldots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Recursive solution : problem decomposition

Seeking for a recursive algorithm to a problem one must think about the ways the problem can be reduced to subproblems.

Let $K[n, W]$ denote the 0-1 knapsack problem instance to be solved where n is the initial number of objects to be considered and W is the initial capacity of the sack.

There are two ways this problem can be decomposed into smaller problems. If $K[i, j]$ be the recursive function :

- one can add the ith object in the knapsack, thus reducing the initial problem to one with $i - 1$ objects yet to consider and capacity $j - w_i$: $K[i - 1, j - w_i]$
- one can choose to disregard object i (don't put in the sack), thus generating a new subproblem with $i - 1$ objects and capacity j unchanged : $K[i - 1, j]$

Recursive solution : writing the algorithm

The base case will be when one object is left to consider. The solution is

$$K[1,j] = \begin{cases} v_1 & \text{if } w_1 \leq j \\ 0 & \text{if } w_1 > j. \end{cases}$$

Once the value of the base case is computed, the solution to the other subproblems is obtained as followed :

$$K[i,j] = \begin{cases} K[i-1,j] & \text{if } w_i > j \\ max(K[i-1,j], K[i-1,j-w_i] + v_i) & \text{if } w_i \leq j. \end{cases}$$

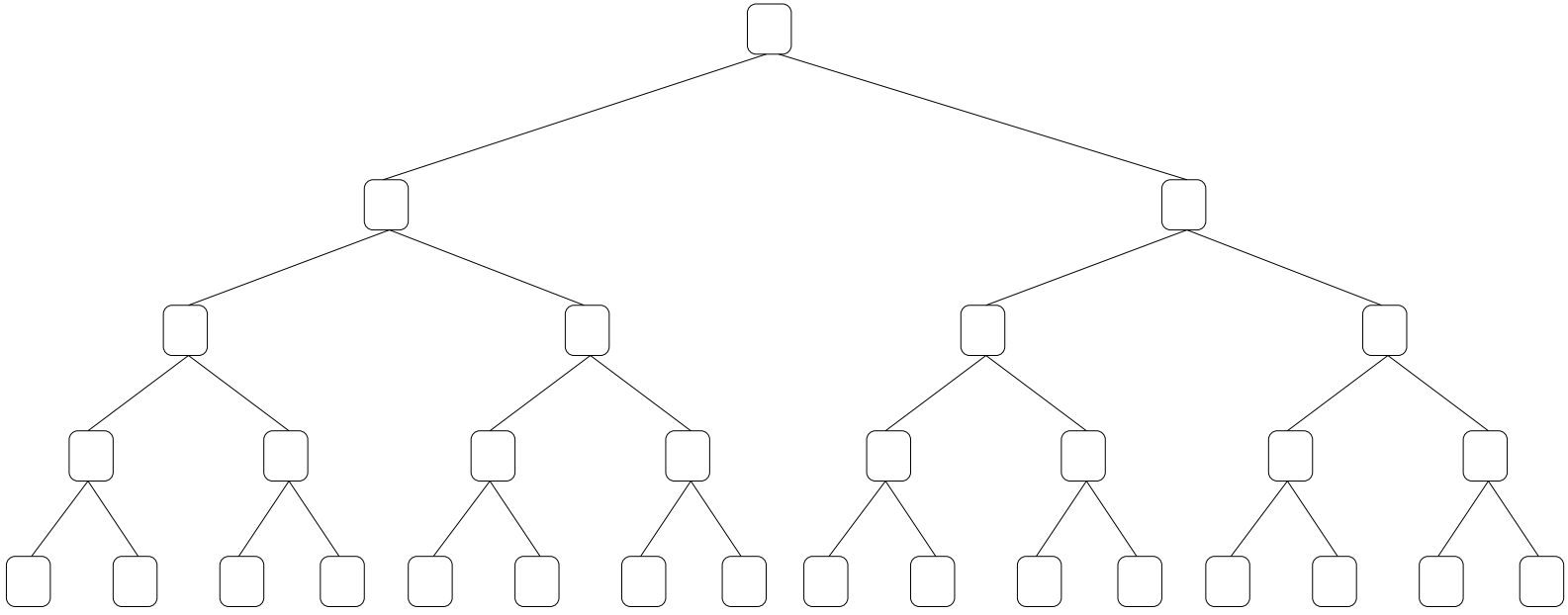
This recursive function is initially called with $K[n,W]$.

Divide & Conquer 0-1 Knapsack

```
int K(i, W)
  if (i == 1) return (W < w[1]) ? 0 : v[1]
  if (W < w[i]) return K(i − 1, W);
  return max(K(i − 1, W), K(i − 1, W − w[i]) + v[i]);
```

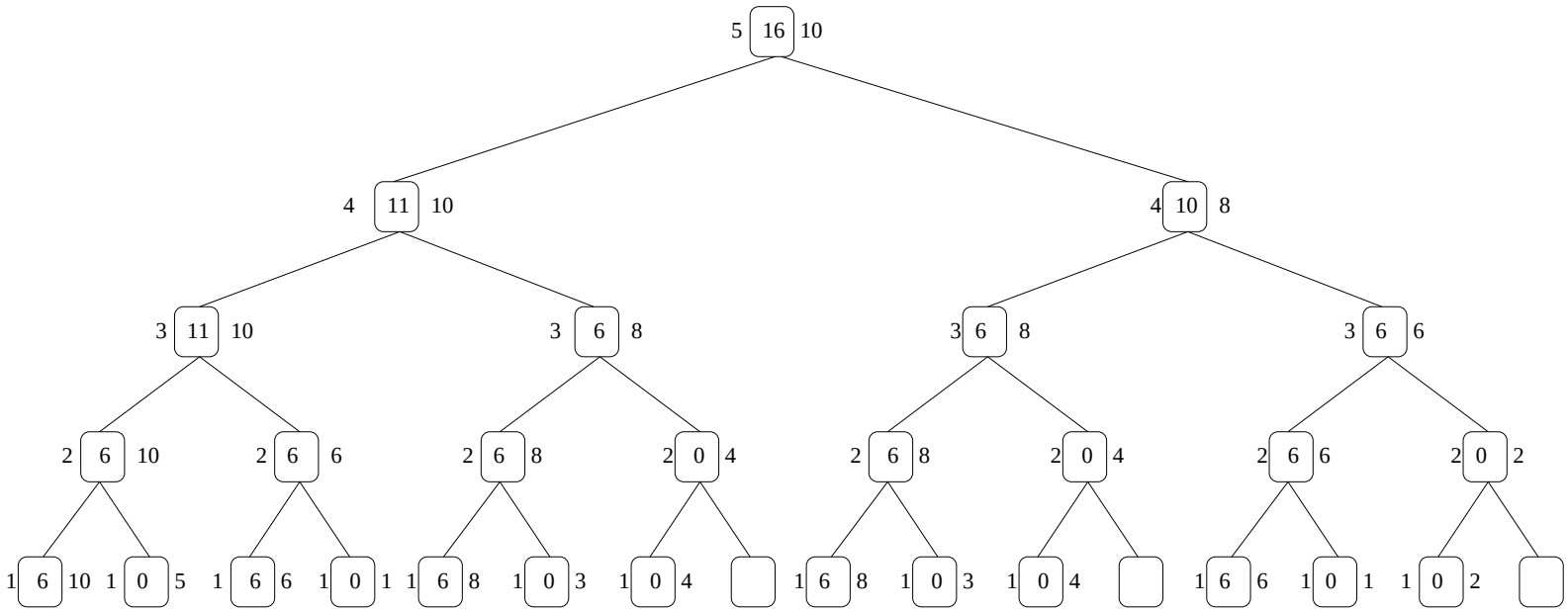
Solve for the following problem instance where $W = 10$:

i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6



i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

```
int K(i, W)
  if (i == 1) return (W < w[1]) ? 0 : v[1];
  if (W < w[i]) return K(i − 1, W);
  return max(K(i − 1, W), K(i − 1, W − w[i]) + v[i]);
```



C code implementing recursive 0-1 knapsack

The initial call to $K(n-1, W)$ because array indexes in C start at 0, so values of object 1 are in val[0] and wt[0], etc.

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int K(int W, int wt[], int val[], int n) {
   // Base Case
   if (n ==0) return (W < wt[0])? 0 : val[0];
   //Knapsack does not have residual capacity for object n
   if (wt[n] > W) return K(W, wt, val, n - 1);
   else
     return max(
         val[n] + K(W - wt[n], wt, val, n - 1),
         K(W, wt, val, n - 1));
}
int main() {
   int val[] = { 6, 3, 5, 4, 6};  int wt[] = { 6, 5, 4, 2, 2 };
   int W = 10;
   int n = sizeof(val) / sizeof(val[0]);
   printf("The solution is %d\n", K(W, wt, val, n-1));
   return 0;
}
```

Analysis of the Recursive Solution

Let $T(n)$ be the worst-case running time on an input with n objects.

If there is only one object, we do a constant amount of work.

$$T(1) = 1.$$

If there is more than one object, this algorithm does a constant amount of work plus two recursive calls involving $n-1$ objects.

$$T(n) = \begin{cases} 1 & \texttt{if } n = 1 \\ 2T(n-1)+1 & \texttt{if } n > 1 \end{cases}$$

The solution to this recurrence is $T(n) \in \Theta(2^n)$

Overlapping Subproblems

We have seen that the maximal value is $K[n, W]$.

But computing $K[n, W]$ recursively cost $2^n - 1$.

While the number of subproblems is only nW.

Thus, if $nW < 2^n$, then the 0-1 knapsack problem will certainly have overlapping subproblems, therefore using dynamic programming is most likely to provide a more efficient algorithm.

0-1 knapsack satisfies the two pre-conditions (optimal substructure and repeated solutions of identical subproblems) justifying the design of an DP algorithm for this problem.

0-1 Knapsack : Bottom up DP algorithm

Declare a table K of size $n \times W + 1$ that stores the optimal solutions of all the possible subproblems. Let $n = 6$, $W = 10$ and

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											
4											
5											
6											

0-1 Knapsack : Bottom up DP algorithm

Initialization of the table :

The value of the knapsack is 0 when the capacity is 0. Therefore, $K[i, 0] = 0$, $i = 1..6$.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0										
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : Bottom up DP algorithm

Initialization of the table using the base case of the recursive function : if (i == 1) return (W < w[1]) ? 0 : v[1]

This said that if the capacity is smaller than the weight of object 1, then the value is 0 (cannot add object 1), otherwise the value is $v[1]$

Since $w[1] = 3$ we have :

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : Bottom up DP algorithm

The DP code for computing the other entries of the table is based on the recursive function for 0-1 knapsack :

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

```
int K(i, W)
  if (i == 1) return (W < w[1]) ? 0 : v[1];
  if (W < w[i]) return K(i − 1, W);
  return max(K(i − 1, W), K(i − 1, W − w[i]) + v[i]);
```

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : Bottom up DP algorithm

The bottom-up dynamic programming algorithm is now (more or less) straightforward.

```
function 0-1-Knapsack(w, v, n, W)
   int K[n, W + 1];
   for(i = 1; i ≤ n; i++)   K[i, 0] = 0;
   for(j = 0; j ≤ W; j++)
      if (w[1] ≤ j) then K[1, j] = v[1];
      else K[1, j] = 0;
   for (i = 2; i ≤ n; i++)
       for (j = 1; j ≤ W; j++)
           if (j ≥ w[i] && K[i − 1, j − w[i]] + v[i] > K[i − 1, j])
              K[i, j] = K[i − 1, j − w[i]] + v[i];
           else
              K[i, j] = K[i − 1, j];
   return K[n, W];
```

0-1 Knapsack Example

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

```
for (i = 2; i ≤ n; i++)
    for (j = 1; j ≤ W; j++)
        if (j ≥ w[i] && K[i − 1, j − w[i]] + v[i] > K[i − 1, j])
            K[i, j] = K[i − 1, j − w[i]] + v[i];
        else
            K[i, j] = K[i − 1, j];
```

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

Finding the Knapsack

How do we compute an optimal knapsack ?

With this problem, we don't have to keep track of anything extra. Let $K[n, k]$ be the maximal value.

If $K[n, k] \neq K[n-1, k]$, then $K[n, k] = K[n-1, k-w_n] + v_n$, and the nth item is in the knapsack.

Otherwise, we know $K[n, k] = K[n-1, k]$, and we assume that the nth item is not in the optimal knapsack.

Finding the Knapsack

In either case, we have an optimal solution to a subproblem.

Thus, we continue the process with either $K[n-1, k]$ or $K[n-1, k-w_n]$, depending on whether n was in the knapsack or not.

When we get to the $K[1, k]$ entry, we take item 1 if $K[1, k] \neq 0$ (equivalently, when $k \geq w[1]$)

Finishing the Example

- Recall we had :

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

- We work backwards through the table

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

- The optimal knapsack contains $\{1, 2, 4, 6\}$

Exercise 3 : 0-1 Knapsack

Solve the following 0-1 knapsack instance with $W = 10$:

```
int K(i, W)
  if (i == 1) return (W < w[1]) ? 0 : v[1];
  if (W < w[i]) return K(i − 1, W);
  return max(K(i − 1, W), K(i − 1, W − w[i]) + v[i]);
```

i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10
1	0										
2	0										
3	0										
4	0										
5	0										

What is the optimal value ? Which objects are part of the optimal solution ?

Exercise 4 : 0-1 Knapsack

Solve the following 0-1 knapsack problem : $W = 10$

i	1	2	3	4	5	6
w_i	4	2	3	1	6	4
v_i	6	4	5	3	9	7

Matrix-Chain Multiplication Problem (MCM)

Given a sequence of matrices $A_1, A_2, \ldots, A_n$, where matrix A_i has size $a \times b$, find a full parenthesization of the product $A_1 A_2 \cdots A_n$ such that the number of scalar multiplications required to compute $A_1 A_2 \cdots A_n$ is minimized.

Recall that to compute the product of an $a \times b$ matrix with a matrix $b \times c$ requires abc scalar multiplications.

MCM Example

We can express the sizes of the sequence of matrices as a sequence $d_0, d_1, \ldots, d_n$ of $n+1$ positive integers, since the number of columns of A_i is the number of rows of A_{i+1} for each i.

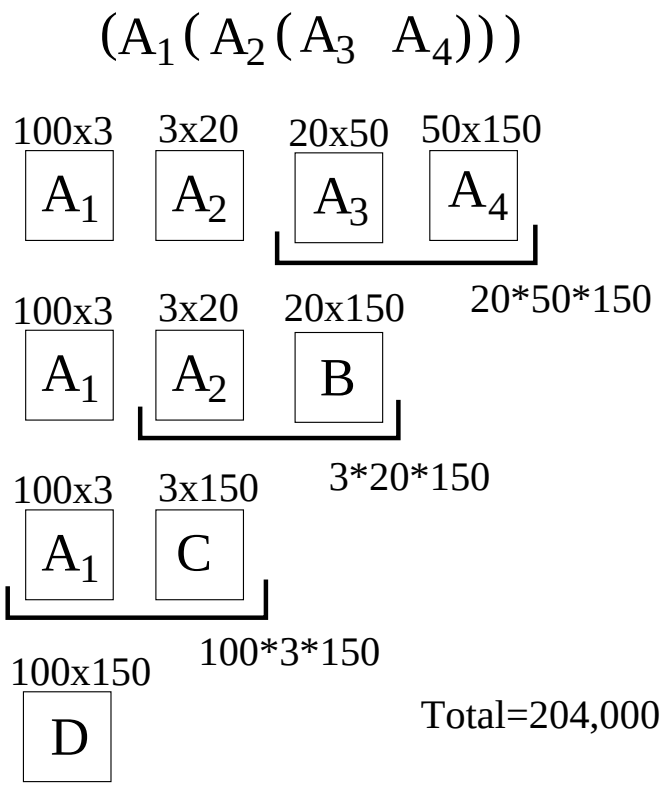
If we are given four matrices, A_1, A_2, A_3, and A_4 with sizes 100×3, 3×20, 20×50, 50×150, they can be represented by the sequence $100, 3, 20, 50, 150$.

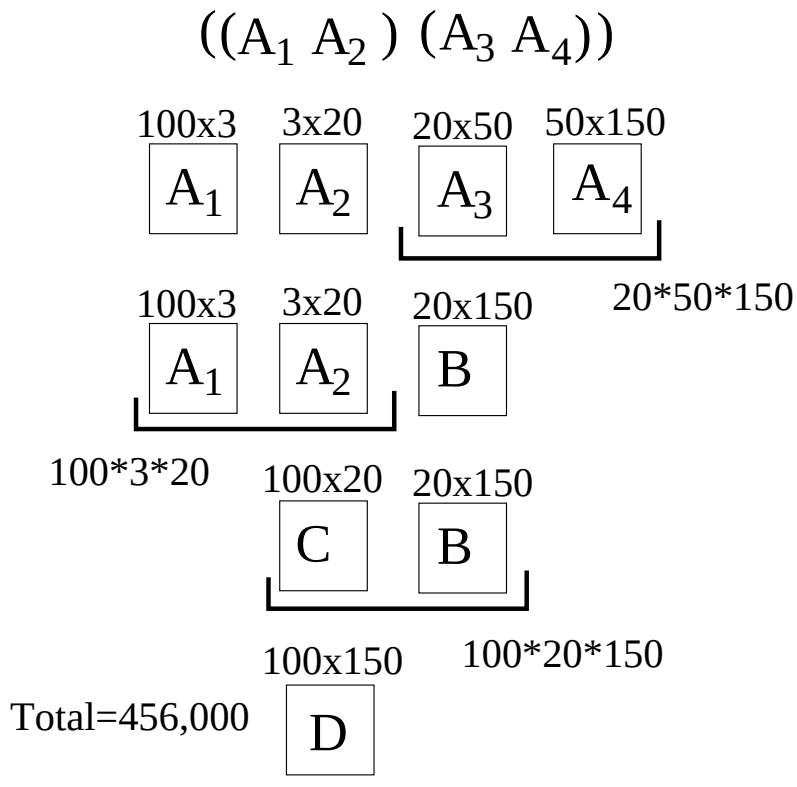
Assume we want to compute the product $A_1 A_2 A_3 A_4$.

Two (of the 5) ways to compute this product are $(A_1(A_2(A_3 A_4)))$ and $((A_1 A_2)(A_3 A_4))$.

MCM Example

Computing the product as $(A_1(A_2(A_3A_4)))$ requires 208,500 operations, and computing it as $((A_1A_2)(A_3A_4))$ requires 456,000 operations.





Solving MCM

Let $M[1, n]$ be the minimum number of scalar multiplications for parenthesizing the sequence of matrices $A_1 A_2 \cdots A_n$

The optimal solution $M[1..n]$ will look like $(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$, where $1 \leq k < n$, and the two halves are parenthesized in some way.

Therefore, a divide-and-conquer approach for the matrix multiplication problem is to divide the problem $A_1 A_2 \cdots A_n$ into two subproblems $A_1 A_2 \cdots A_k$ and $A_{k+1} \cdots A_n$.

Solving MCM

The cost of the optimal solution $M[1..n]$ is the cost of each of the two subsolutions $M[1..k]$ and $M[k+1..n]$ plus the cost of multiplying the final two matrices :

$$M[1..n] = M[1..k] + M[k+1..n] + d_0 \times d_k \times d_n$$

It must be that k partitions the sequence $A_1 A_2 \cdots A_n$ into subsequences such that $M[1..k]$ and $M[k+1..n]$ are two optimal solutions, i.e. $(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$ is the optimal paranthetisation of $A_1 A_2 \cdots A_n$

Thus

$$M[1,n] = \min_{1 \leq k < n} \{M[1,k] + M[k+1,n] + d_0 d_k d_n\}.$$

Solving MCM

Since the split occurs at some k, where $i \leq k < j$, we can compute

$$M[i,j] = \min_{i \leq k < j} \{M[i,k] + M[k+1,j] + d_{i-1} d_k d_j\}.$$

We have two possibilities at this point.

1. Find $M[1,n]$ using a divide-and-conquer algorithm.
2. Compute the entries $M[i,j]$, $1 \leq i,j < n$, from the bottom up.

The first method results in an algorithm with an exponential time complexity

Optimal Substructure

MCM satisfies the requirement that optimal solutions combine solutions of two subproblems that are also optimal

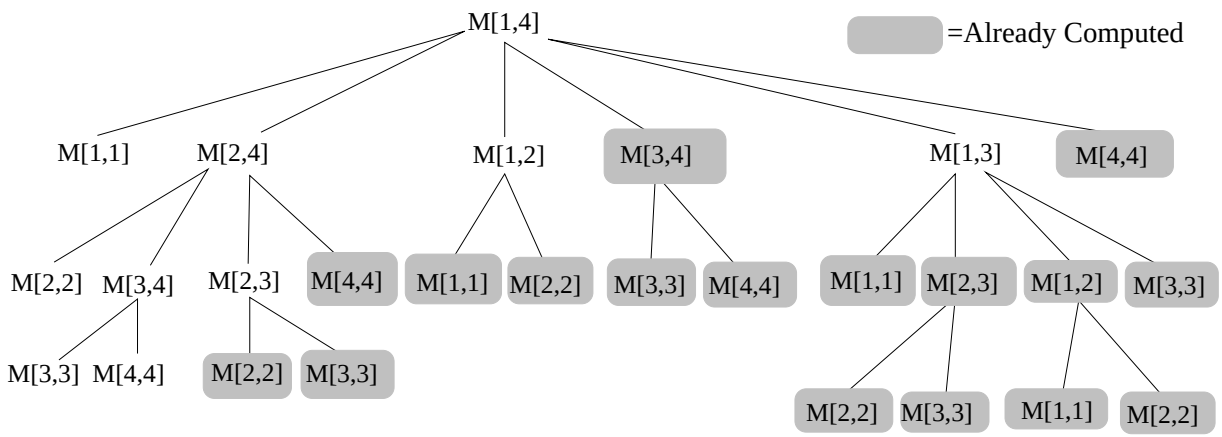
The parenthesization of $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$ must be optimal

Otherwise there exist another parenthetisation P' of one of the two subproblems, for example $A_1 \cdots A_k$, such that $M'[1..k] < M[1..k]$ in which case $M[1..n] = M[1..k] + M[k+1..n] + d_0 \times d_k \times d_n$ is not an optimal solution

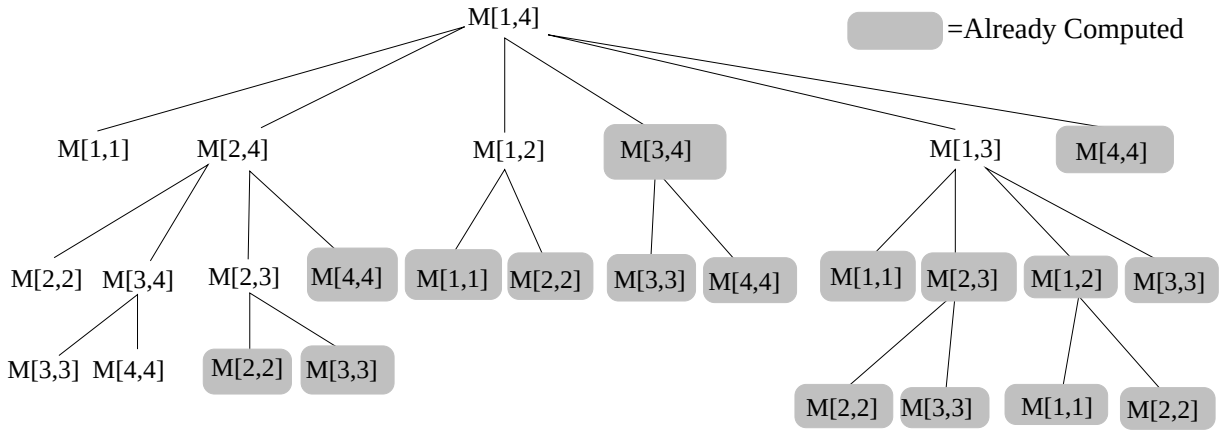
Divide-and-Conquer for MCM

int M(i, j)
  if $i == j$ **then return** (0);
  else
    return $min_{i \leq k < j}(\mathrm{M}(i,k) + \mathrm{M}(k+1,j) + d_{i-1} d_k d_j)$;

The recursion tree for the MCM problem with 4 matrices :



Divide-and-Conquer for MCM



- Notice that many of the calls are repeated (all the shaded boxes).
- The divide-and-conquer algorithm has the following recurrence

$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) + cn$$

with $T(0) = 0$ and $T(1) = 1$.

Analysis of the recurrence for MCM

If $n > 0$,

$$T(n) = 2\sum_{k=1}^{n-1} T(k) + cn$$

Therefore

$$\begin{aligned} T(n) - T(n-1) &= (2\textstyle\sum_{k=1}^{n-1} T(k) + cn) \\ &\quad - (2\textstyle\sum_{k=1}^{n-2} T(k) + c(n-1)) \\ &= 2T(n-1) + c \end{aligned}$$

That is

$$T(n) = 3T(n-1) + c$$

Analysis of the recurrence for MCM

using the iteration method :

$$
\begin{aligned}
T(n) &= 3T(n-1)+c \\
&= 3(3T(n-2)+c)+c \\
&= 9T(n-2)+3c+c \\
&= 9(3T(n-3)+c)+3c+c \\
&= 27T(n-3)+9c+3c+c \\
&= 3^k T(n-k)+c\sum_{l=0}^{k-1} 3^l \\
&= 3^n T(0)+c\sum_{l=0}^{n-1} 3^l \\
&= c3^n + c\frac{3^n-1}{2} \\
&= (c+\tfrac{c}{2})3^n - \tfrac{c}{2} \\
T(n) &\in \Theta(3^n).
\end{aligned}
$$

This recurrence can be solved using the recursion tree method. There are $n-1$ levels, each level i execute $3^i c$ operations, yielding the following summation : $3^0+3^1+3^2+\cdots+3^{n-1}$, the dominant term in this expression is $3^{n-1} \in \Theta(3^n)$

Dynamic programming solution

MCM is a candidate for DP :

1. The recursive algorithm solves some subproblems more than one time
2. It satisfies the optimal substructure condition : an optimal solution to $(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$ is based on parenthesizations of $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$ that are optimal as well.

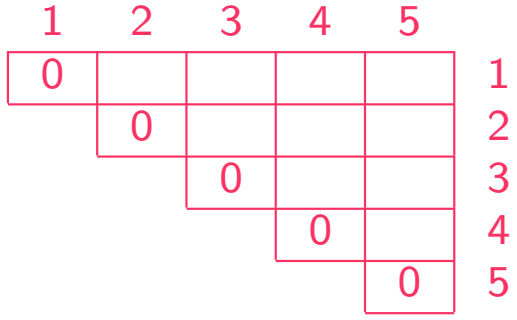
A dynamic programming solution to MCM

int M(i, j)
  if $i == j$ **then return** (0);
  else
    return $min_{i \leq k < j}(\mathrm{M}(i,k) + \mathrm{M}(k+1,j) + d_{i-1} d_k d_j)$;

We define a 2-dimensional array $M[n, n]$ to store solution of each subproblem (a subsequence of matrices to multiply) :

- row i refers to the position of the first matrix in the sequence
- column j refers to the position of the last matrix in the sequence.

Given we first solve the base cases, i.e. sequences with only one matrix (where $i = j$), we first fill the entries $M[i, i]$ of the table :



A dynamic programming solution to MCM

```
int M(i, j)
  if i == j then return (0);
  else
    return min_{i≤k<j}(M(i, k) + M(k + 1, j) + d_{i−1} d_k d_j);
```

The next larger problem sizes to solve involve the product of two matrices.

We multiple each 2 consecutive matrices, i.e. $(1, 2), (2, 3), (n-1, n)$, and store the solution into $M[i, i+1]$

Here k can only take one value, $k = i$. According to the recursive algo $M[i, i+1] = M[i, i] + M[i+1, i+1] + d_{i-1} d_k d_j = d_{i-1} d_k d_j$

1	2	3	4	5	
0	$d_0 d_1 d_2$				1
	0	$d_1 d_2 d_3$			2
		0	$d_2 d_3 d_4$		3
			0	$d_3 d_4 d_5$	4
				0	5

A dynamic programming solution to MCM

```
int M(i, j)
  if i == j then return (0);
  else
    return min_{i≤k<j}(M(i, k) + M(k + 1, j) + d_{i−1} d_k d_j);
```

The next larger problem sizes to solve involve the product of three consecutive matrices.

We multiple each 3 consecutive matrices, i.e. $(1,3),(2,4),(n-2,n)$, and store the solution into $M[i, i+2]$

Here k can only take two values, $k = i$ and $k = i+1$. According to the recursive algo $M[i, i+2] = min_{k=i}^{k<j}(M[i,k] + M[k+1,j] + d_{i-1} d_k d_j)$

1	2	3	4	5	
0	$d_0 d_1 d_2$				1
	0	$d_1 d_2 d_3$			2
		0	$d_2 d_3 d_4$		3
			0	$d_3 d_4 d_5$	4
				0	5

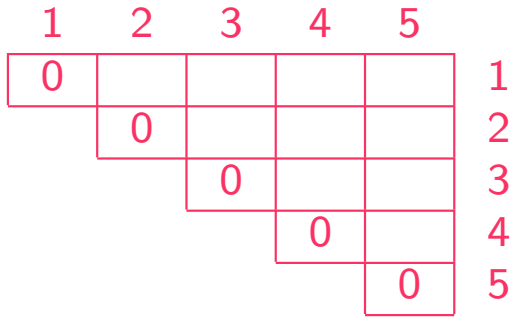
MCM dynamic programming Example

We are given the sequence $(4, 10, 3, 12, 20, 7)$.

The 5 matrices have sizes 4×10, 10×3, 3×12, 12×20, and 20×7.

We need to compute $M[i, j]$, $1 \leq i, j \leq 5$.

We know $M[i, i] = 0$ for all i.



MCM Example

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.
Given the sequence $(4, 10, 3, 12, 20, 7)$:

$M[1,2] = 4 \times 10 \times 3 = 120$,
$M[2,3] = 10 \times 3 \times 12 = 360$,
$M[3,4] = 3 \times 12 \times 20 = 720$,
$M[4,5] = 12 \times 20 \times 7 = 1680$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

MCM Example (continued)

There are two ways one can split a sequence of 3 matrices in two sub-sequences : (1) (2,3) or (1,2) (3). Given 5 matrices there are 3 sequences of 3 matrices. $(4, 10, 3, 12, 20, 7)$

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + d_0 d_2 d_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + d_0 d_1 d_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases}$$

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + d_1 d_3 d_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + d_1 d_2 d_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases}$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + d_2 d_4 d_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + d_2 d_5 d_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example (continued again)

A sequence of 4 matrices can be split in 3 different ways (1) (2,4), or (1,2) (3,4), or (1,3) (4). Given 5 matrices, there are 2 sequences of 4 matrices. $(4, 10, 3, 12, 20, 7)$

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + d_0 d_3 d_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \\ M[1,2] + M[3,4] + d_0 d_2 d_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,1] + M[2,4] + d_0 d_1 d_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \end{cases}$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + d_1 d_4 d_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + d_1 d_3 d_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + d_1 d_2 d_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example (continued *again*)

Now the product of 5 matrices. $(4, 10, 3, 12, 20, 7)$

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + d_0 d_4 d_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1640 \\ M[1,3] + M[4,5] + d_0 d_3 d_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + d_0 d_2 d_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + d_0 d_1 d_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

MCM Example : Optimal Cost and Solution

The optimal cost is $M[1,5] = 1344$.

What is the optimal parenthesization ?

We didn't keep track of enough information to find that out.

The algorithm can be modified slightly to keep enough information to derive the optimal parenthesization.

Each time the optimal value for $M[i,j]$ is found, store also the value of k that was used.

Then we can just work backwards, partitioning the matrices according to the optimal split.

MCM Example : Optimal Solution

If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $(A_1A_2)(A_3A_4A_5)$. The first half is done.

MCM Example : Optimal Solution

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The optimal solution for the second half comes from entry $M[3,5]$.

The value of k here is 4, so now we have $(A_1A_2)((A_3A_4)A_5)$.

Thus the optimal solution is to parenthesize as $(A_1A_2)((A_3A_4)A_5)$.

MCM : Dynamic Programming Algo.

```
int MCM(int *MM, int n) {
  int M[n, n], min;
  for (i = 1; i ≤ n; i++) M[i, i] := 0;
  for (s = 1; s < n; s++)
    for (i = 1; i ≤ n − s; i++)
      min = ∞;
      for (k = i; k < i + s; k++)
        if (M[i, k] + M[k + 1, i + s] + d_{i−1} d_k d_{i+s} < min)
          min = M[i, k] + M[k + 1, i + s] + d_{i−1} d_k d_{i+s};
      M[i, i + s] = min;}
```

This algorithm has 3 nested loops. The summation is $\sum_{s=1}^{n} \sum_{i=1}^{n-1} \sum_{k=i}^{i+s} 1$. The complexity of dynamic programming MCM is $\Theta(n^3)$.

Exercises 6 and 7 on matrix-chain multiplications

6. Given the sequence 2, 3, 5, 2, 4, 3, how many matrices do we have and what is the dimension of each matrix. Using the previous dynamic programming algorithm for matrix-chain multiplication, compute the parenthetization of these matrices that minimize the number of scalar multiplications.
7. Given the sequence 5, 4, 6, 2, 7, how many matrices do we have and what is the dimension of each matrix. Using the previous dynamic programming algorithm for matrix-chain multiplication, compute the parenthetization of these matrices that minimize the number of scalar multiplications.

Longest common subsequence (LCS)

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc" and "ace" are subsequences of "abcdef".

LCS problem : Given 2 sequences, $X = [x_1 \dots x_m]$ and $Y = [y_1 \dots y_n]$, find a subsequence common to both whose length is longest.

For example, "ADH" is a subsequence of length 3 of the input sequences "ABCDGH" and "AEDFHR".

Brute-force algorithm for LCS

Brute-force algorithm : For every subsequence of X, check whether it is a subsequence of Y. Time : $\Theta(n2^m)$

- $2^m - 1$ subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check : scan Y for first letter, from there scan for second, and so on.

Optimal substructure

Consider the input sequences be $X = [x_1 \dots x_m]$ and $Y = [y_1 \dots y_n]$ respectively of length m and n.

Let $lcs(X[1..m], Y[1..n])$ be the length of the LCS of X and Y. The length of the LCS is computed recursively as follow :

- if the last character of both sequences match, i.e. if $X[m] == Y[n]$ then $lcs(X[1..m], Y[1..n]) = 1 + lcs(X[1..m-1], Y[1..n-1])$
- if the last character of both sequences do not match, i.e. if $X[m] \neq Y[n]$ then $lcs(X[1..m], Y[1..n]) = max(lcs(X[1..m-1], Y[1..n]), lcs(X[1..m], Y[1..n-1]))$

Optimal substructure

Given sequences "AGGTAB" and "GXTXAYB", last characters match, so length of LCS can be written as :
$lcs("AGGTAB","GXTXAYB") = 1 + lcs("AGGTA","GXTXAY")$

For sequences "ABCDGH" and "AEDFHR", the last characters do not match, so length of LCS is : $lcs("ABCDGH","AEDFHR") = max(lcs("ABCDG","AEDFHR"), lcs("ABCDGH","AEDFH"))$

So the LCS problem has optimal substructure property as the optimal solution of the main problem can be solved using optimal solutions to subproblems.

A recursive algorithm for LCS

```
int lcs(char *X, char *Y, int i, int j )
  if (i == 0||j == 0) return 0;
  if (X[i] == Y[j])
    return 1 + lcs(X, Y, i − 1, j − 1);
  else
    return max(lcs(X, Y, i, j − 1), lcs(X, Y, i − 1, j));
```

Time complexity of this recursive algorithm is $O(2^n)$ in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Building a partial call tree for sequences "AXYT" and "AYZX", it can be verified that the recursive algorithm solves the same subproblems several times. Soon you will observe that $lcs("AXY","AYZ")$ is being solved twice.

Top Down DP algorithm for LCS

```
int TD − DP − lcs(char *X, char *Y, int i, int j)
  if (i == 0||j == 0)
    table[i, j] = 0; /* re-calculate each time */
    return table[i, j];
  if (X[i] == Y[j])
    if (table[i, j] == −1)
      table[i, j] = 1 + TD − DP − lcs(X, Y, i − 1, j − 1);
      return table[i, j]
    else return table[i, j];
  if (X[i] ≠ Y[j])
    if (table[i, j] == −1)
      table[i, j] = max(TD − DP − lcs(X, Y, i, j − 1),
                        TD − DP − lcs(X, Y, i − 1, j));
      return table[i, j]
    else return table[i, j];
```

Bottom up DP for LCS

LCS-Length(X,Y,m,n)
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$
for $i = 1$ to m $c[i, 0] = 0$
for $j = 0$ to n $c[0, j] = 0$
for $i = 1$ to m
  for $j = 1$ to n
    if $X[i] == Y[j]$
      $c[i, j] = c[i - 1, j - 1] + 1$
      $b[i, j] =$"$\nwarrow$"
    else if $c[i - 1, j] \geq c[i, j - 1]$
      $c[i, j] = c[i - 1, j]$
      $b[i, j] =$"$\uparrow$"
    else $c[i, j] = c[i, j - 1]$
      $b[i, j] =$"$\leftarrow$"
return c and b

Computational complexity is $\Theta(mn)$

Example

LCS-Length(X,Y,m,n)
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$
for $i = 1$ to m $c[i, 0] = 0$
for $j = 0$ to n $c[0, j] = 0$
for $i = 1$ to m
  for $j = 1$ to n
    if $X[i] == Y[j]$
      $c[i, j] = c[i - 1, j - 1] + 1$
      $b[i, j] =$"$\nwarrow$"
    else if $c[i - 1, j] \geq c[i, j - 1]$
      $c[i, j] = c[i - 1, j]$
      $b[i, j] =$"$\uparrow$"
    else $c[i, j] = c[i, j - 1]$
      $b[i, j] =$"$\leftarrow$"
return c and b

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$X = [ABCBDAB]$
$Y = [BDCABA]$
$m = 7$; $n = 6$
LCS is $[BCBA]$

Printing the LCS

```
Print-LCS(b,X,i,j)
if i == 0 or j == 0 return
if b[i,j] == "↖"
   Print-LCS(b,X,i-1,j-1)
   print x_i
else if b[i,j] == "↑"
   Print-LCS(b,X,i-1,j)
else if b[i,j] == "←"
   Print-LCS(b,X,i,j-1)
```

Computational complexity is $\Theta(m+n)$

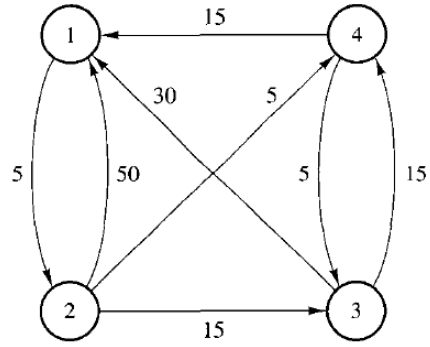
Printing the LCS

- Initial call is Print-LCS(b,X,m,n)
- $b[i,j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.
- When $b[i,j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries with $\nwarrow$ in them

Exercises : longest common subsequences

8. Find the LCS for $X = [ABAZDC]$ and $Y = [BACBAD]$
9. Prove that the LCS for the input sequences $X = [ABCDGH]$ and $Y = [AEDFHR]$ is "ADH" of length 3.

The All-Pairs Shortest-Path Problem



$$L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 15 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

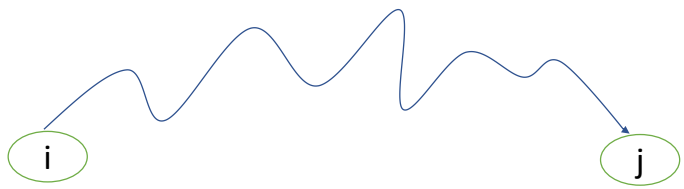
Problem definition :

- $G = \{V, E\}$ is a connected "directed" graph, V is the set of nodes ($|V| = n$) and E is the set of edges.
- Each edge has an associated nonnegative length. A distance matrix $L[i, j]$ gives the length of each edge :
  - $L[i, i] = 0$, $L[i, j] \geq 0$ if $i \neq j$,
  - $L[i, j] = \infty$ if the edge (i, j) does not exist.
- *Find the shortest path distance between each pair of nodes in the graph*

Reasoning to find a D-&-C algorithm

The n nodes in the graph are numbered consecutively from 1 to n

The shortest path between a pair of nodes i, j uses some intermediary nodes in the set $\{1, 2, \ldots, n\} \setminus \{i, j\}$.



One way to reduce the problem size for i, j is to consider a smaller set of nodes $\{1, 2, \ldots, n\} \setminus \{i, j\}$ from which the shortest path can be built, for example $\{1, 2, \ldots, n\} \setminus \{i, j, n\}$

The problem is now to find the shortest distance between i, j using as intermediary set $\{1, 2, \ldots, n-1\}$

A D-&-C algo for all-pairs shortest path

As nodes are numbered consecutively, a sub-problem for finding the shortest path between i, j is *finding the shortest path between* i, j *using only nodes in the set* $\{1, 2, \ldots, k\}$ *where* $k \leq n$.

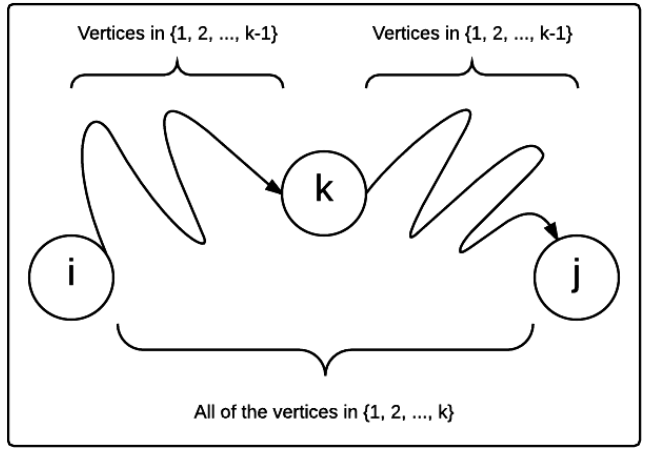
Let denote the function that returns the length of the shortest path between nodes i, j using only intermediary nodes from the set $\{1, 2, \ldots, k\}$ as $S - Path(i, j, k)$

The value returns by $S - Path(i, j, 0)$, the length of the shortest path between i, j with no intermediate node, is the cost of the direct edge from i to j in the graph, i.e. $S - Path(i, j, 0) = L[i, j]$

Shortest path D&C

There are two possible ways one can compute the value of $S - Path(i, j, k)$

1. one may decide not to include node k in the computation of the shortest path between i, j (only uses nodes in the set $\{1, 2, \ldots, k-1\}$, in which case $S - Path(i, j, k) = S - Path(i, j, k-1)$
2. one may decide to select node k, from i to k then from k to j using only intermediate nodes $\{1, 2, \ldots, k-1\}$, then $S - Path(i, j, k) = S - Path(i, k, k-1) + S - Path(k, j, k-1)$



The length of the path from i to k to j is the concatenation of the shortest path from i to k and the shortest path from k to j, each one only using intermediate vertices in $\{1, 2, \ldots, k-1\}$.

Shortest path D&C

Then we can define $S - Path(i, j, k)$ as a recursive function to compute the shortest path between nodes i and j

function $S - Path(i, j, k)$
  if $(k == 0)$ **then return** $L[i, j]$; /* Base case */
  else
    return $\min(S - Path(i, j, k-1),$
      $S - Path(i, k, k-1) + S - Path(k, j, k-1))$

The initial call is $S - Path(i, j, n)$. This function is run for each pair of node i and j

The above recursive algorithm is used to design the Floyd-Warshall dynamic programming algorithm that solves the all pairs of shortest paths problem.

Floyd-Warshall Algorithm

It is a bottom up dynamic programming algorithm. It starts by computing the shortest path for all pairs of nodes for $k = 0$. Then it considers $k = 1$, $k = 2$, until $k = n$.

A matrix D gives the length of the shortest path between each pair of nodes

$D_0 = L$, the direct distances between nodes.

Algorithm Floyd$(L[n, n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i, j] = min(D[i, j], D[i, k] + D[k, j])$
  return D

After each iteration k, D_k contains the length of the shortest paths that only use nodes in $\{1, 2, \ldots, k\}$ as intermediate nodes.

Floyd Algorithm

Algorithm Floyd$(L[n,n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i,j] = min(D[i,j], D[i,k] + D[k,j])$
  return D

At iteration k, the algo checks each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k that is better than the present optimal path passing only through nodes in $\{1, 2, \ldots, k-1\}$.

$$D[i,j] = min(D[i,j], D[i,k] + D[k,j])$$

Execution of Floyd's algorithm

Algorithm Floyd$(L[n,n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i,j] = min(D[i,j], D[i,k] + D[k,j])$
  return D

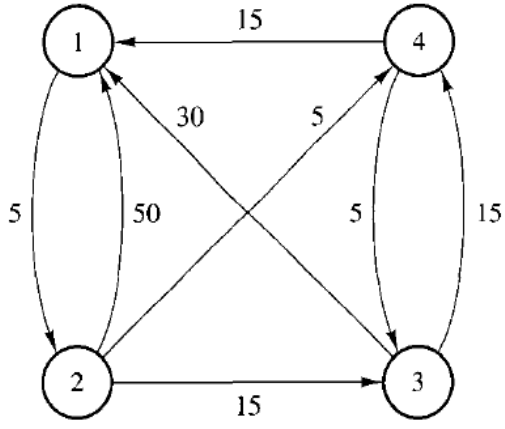
Base case $D_0 = L$, the smallest problem instances

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

Algorithm Floyd$(L[n, n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i, j] = min(D[i, j], D[i, k] + D[k, j])$
  return D

For $k = 1$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through node 1.



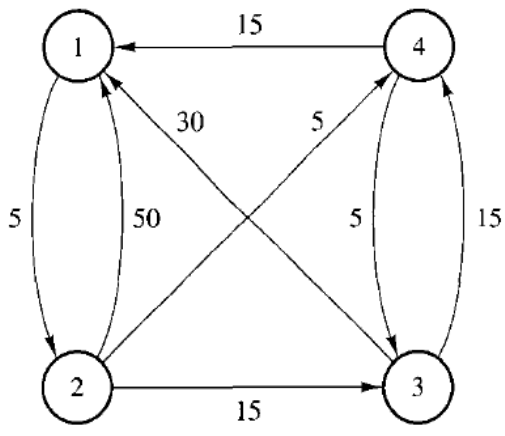
$$\begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

Algorithm Floyd$(L[n, n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i, j] = min(D[i, j], D[i, k] + D[k, j])$
  return D

For $k = 2$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through nodes 1 and 2.



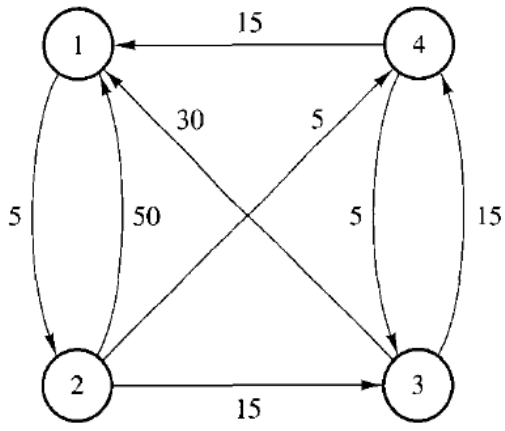
$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

Algorithm Floyd$(L[n, n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i, j] = min(D[i, j], D[i, k] + D[k, j])$
  return D

For $k = 3$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through nodes $\{1, 2, 3\}$.



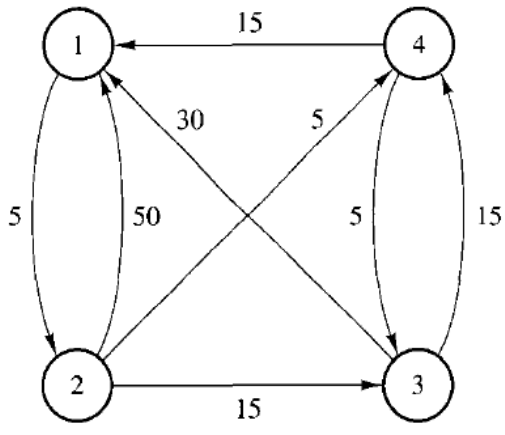
$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

Algorithm Floyd$(L[n, n])$
  $D = L$
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        $D[i, j] = min(D[i, j], D[i, k] + D[k, j])$
  return D

For $k = 4$, solution, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through any nodes.



$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Computing the shortest paths

We want to know the shortest paths, not just their length.

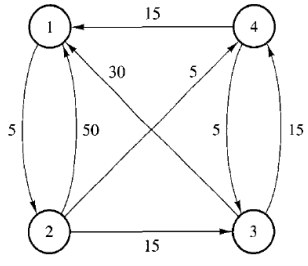
For that we create a new matrix P of size $n \times n$.

Then use the following algorithm in place of the previous one :

Algorithm Floyd$(D[n,n])$
Input : An array D of shortest path lenghts
Output : The shortest path between every pair of nodes
$P[n,n]$ an $n \times n$ array initialized to 0
for $(k = 1; k \leq n; k++)$
  for $(i = 1; i \leq n; i++)$
    for $(j = 1; j \leq n; j++)$
      if $D[i,k] + D[k,j] < D[i,j]$ **then**
        $D[i,j] = D[i,k] + D[k,j]$
        $P[i,j] = k$;

Computing the shortest pathsK $k = 1$

Algorithm Floyd$(D[n, n])$
  Input : An array D of shortest path lenghts
  Output : The shortest path between every pair of nodes
  $P[n, n]$ an $n \times n$ array initialized to 0
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        if $D[i, k] + D[k, j] < D[i, j]$ **then**
          $D[i, j] = D[i, k] + D[k, j]$
          $P[i, j] = k$;



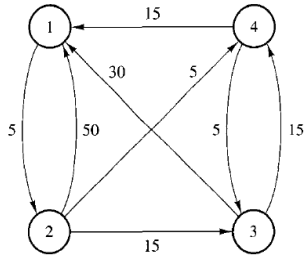
$$\begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

0	0	0	0
0	0	0	0
0	1	0	0
0	1	0	0

Computing the shortest paths : $k = 2$

Algorithm Floyd$(D[n, n])$
  Input : An array D of shortest path lenghts
  Output : The shortest path between every pair of nodes
  $P[n, n]$ an $n \times n$ array initialized to 0
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        if $D[i, k] + D[k, j] < D[i, j]$ **then**
          $D[i, j] = D[i, k] + D[k, j]$
          $P[i, j] = k$;



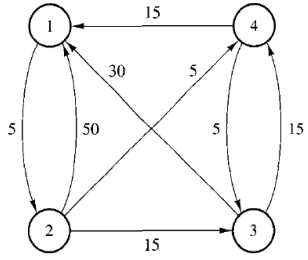
$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

0	0	2	2
0	0	0	0
0	1	0	0
0	1	0	0

Computing the shortest paths : $k = 3$

Algorithm Floyd$(D[n, n])$
  Input : An array D of shortest path lenghts
  Output : The shortest path between every pair of nodes
  $P[n, n]$ an $n \times n$ array initialized to 0
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        if $D[i, k] + D[k, j] < D[i, j]$ **then**
          $D[i, j] = D[i, k] + D[k, j]$
          $P[i, j] = k$;



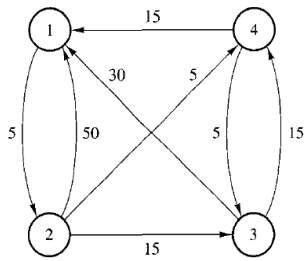
$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

0	0	2	2
3	0	0	0
0	1	0	0
0	1	0	0

Computing the shortest paths : $k = 4$

Algorithm Floyd$(D[n, n])$
  Input : An array D of shortest path lenghts
  Output : The shortest path between every pair of nodes
  $P[n, n]$ an $n \times n$ array initialized to 0
  for $(k = 1; k \leq n; k++)$
    for $(i = 1; i \leq n; i++)$
      for $(j = 1; j \leq n; j++)$
        if $D[i, k] + D[k, j] < D[i, j]$ **then**
          $D[i, j] = D[i, k] + D[k, j]$
          $P[i, j] = k$;



$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

0	0	4	2
4	0	4	0
0	1	0	0
0	1	0	0

Computing the shortest paths

- The matrix P is initialized to 0.
- When the previous algorithm stops, $P[i,j]$ contains the number of the last iteration that caused a change in $D[i,j]$.

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- If $P[i,j] = 0$, then the shortest path between i and j is directly along the edge (i,j).
- If $P[i,j] = k$, the shortest path from i to j goes through k.

Computing the shortest paths

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- Look recursively at $P[i,k]$ and $P[k,j]$ to find other intermediate vertex along the shortest path.
- In the table above, since, $P[1,3] = 4$, the shortest path from 1 to 3 goes through 4. If we look recursively at $P[1,4]$ we find that the path between 1 and 4 goes through 2. Recursively again, if we look at $P[1,2]$ and $P[2,4]$ we find direct edge.
- Similarly if we look recursively to $P[4,3]$ we find a direct edge (because $P[4,3] = 0$). Then the shortest path from 1 to 3 is 1,2,4,3.

Exercises : All pairs shortest paths

10. Compute the all pairs of shortest paths for the following oriented graph.

$$L = \begin{pmatrix} 0 & 5 & 10 & 3 \\ \infty & 0 & 1 & 4 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 0 \end{pmatrix}$$

11. Compute the all pairs of shortest paths for the following oriented graph.

$$L = \begin{pmatrix} 0 & 3 & 8 & \infty & 4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & 5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Cost of dynamic programs

Calculate how many table or list entries you fill in (sometimes you don't use the entire table, just all entries under the diagonal or something like that).

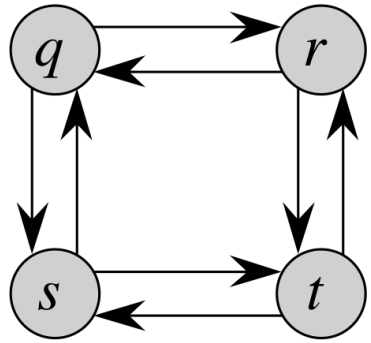
Calculate how much work it takes to compute each entry.

Multiply the two numbers.

Then add the cost of retrieving the answer from the table.

Example : problem fails the optimal substructure test

Unweighted longest simple path : *Find a "simple" path from u to v consisting of the most edges.*
We may assume that the problem of finding an unweighted longest simple path exhibits optimal substructure, i.e. if we decompose the longest simple path from u to v into subpaths $p_1 = u \rightarrow w$ and $p_2 = w \rightarrow v$ then p_1 and p_2 must also be the longest subpaths between $u - w$ and $w - v$. The figure below shows that this is not the case :



Consider the path $q \rightarrow r \rightarrow t$, the longest simple path from q to t. $q \rightarrow r$ is not the longest, rather $q \rightarrow s \rightarrow t \rightarrow r$ is.

Dynamic Programming : Conclusion

As we have seen, dynamic programming is a technique that can be used to find optimal solutions to certain problems.

Dynamic programming works when a problem has the following characteristics :

- **Optimal Substructure :** If an optimal solution contains optimal subsolutions, then a problem exhibits *optimal substructure*.
- **Overlapping subproblems :** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has *overlapping subproblems*.

Dynamic programming takes advantage of these facts to solve problems more efficiently.

It works well only for problems that exhibit *both* properties.

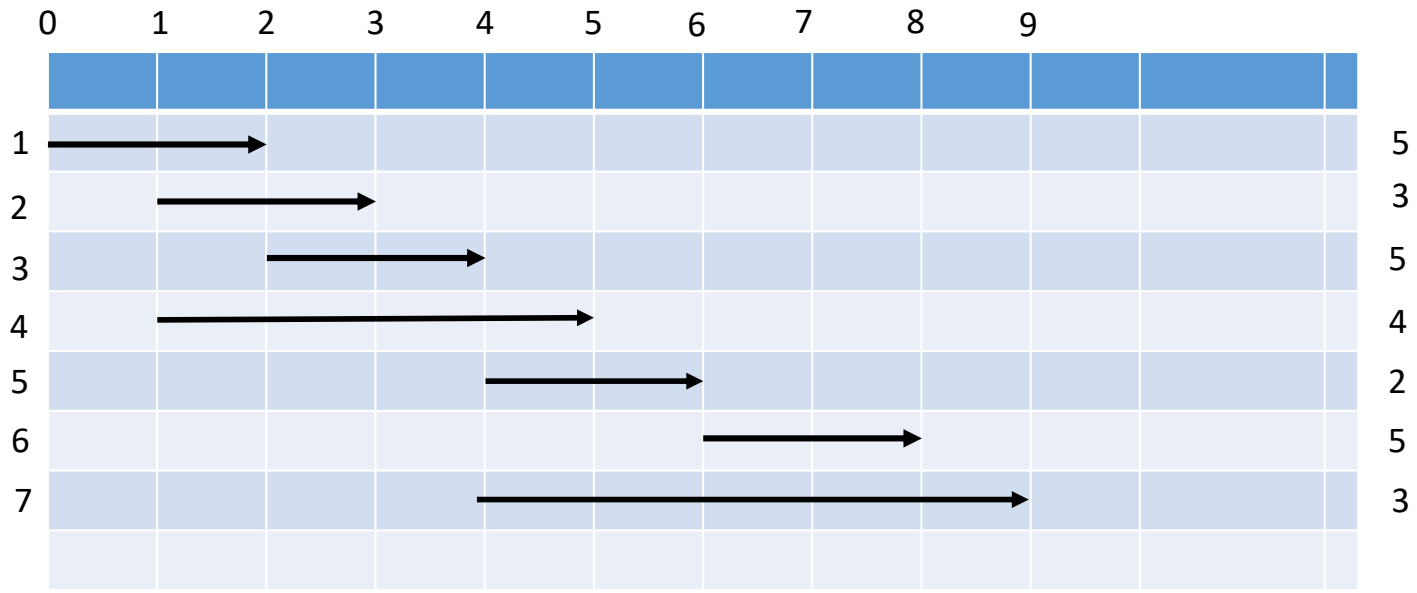
Exercise 12 : write a dynamic programming algorithm

You have a scheduling problem in which the input consists of a list L of requested activities to be executed where each request i consists of the following elements :

- a time interval $[s_i, f_i]$ where s_i is the start time when the activity should begin and f_i is the moment when the activity should finish

- a benefit b_i which is an indicator of the benefit for performing this activity.

Below is a list of 7 requests that need to be scheduled over a period of 9 time units. On the right side of the requests table is the benefit associated to each request. For example, request 3 has to run during the interval $[2, 4]$ for a benefit of $b_3 = 5$.



Given a list L of activity requests, the optimization problem is to schedule the requests in a non-conflicting way such to maximize the total benefit of the activities that are included in the schedule (requests i and j conflict if the time interval $[s_i, f_i]$ intersects with the time interval $[s_j, f_j]$).
Write a dynamic programming algorithm for this scheduling problem. The recursive algorithm is as follow. In the request table above, the list L of requests is ordered in increasing order of the requests finishing time. We use this ordering to decompose the problem into sub-problems. Let

$B_i =$ the maximum benefit that can be achieved using the first i requests in L

So B_i is the value of the optimal solution while considering the subproblem consisting the i first requests. The base case is $B_0 = 0$. We are given a

predecessor function $pred(i)$ which for each request i is the largest index $j < i$ such that request i and j don't conflict. If there is no such index, then $pred(i) = 0$. For example, $pred(3) = 1$ while $pred(2) = 0$.
We can now define a recursive function that we call $HST(L, i)$:

- If the optimal schedule achieving the benefit B_i includes request i, then $B_i = B_{pred(i)} + b_i$
- If the optimal schedule achieving the benefit B_i does not include request i, then $B_i = B_{i-1}$

$HST(L, i)$
  if $(i == 0)$ $B_i = 0$;
  else
    $B_i = max\{HST(L, i-1), HST(L, pred(i)) + b_i\}$

Answer the following questions :

13. Write a bottom up dynamic programming algorithm corresponding to the above HST recursive algorithm

14. For the problem instance described in the request table above, construct the table of all the subproblems considered by your dynamic programming algorithm and fill it with the optimal benefit for each subproblem

Exercise 15 : Longest decreasing subsequence

Write a dynamic programming algorithm to solve the longest decreasing subsequence problem (LDS) : Given a sequence of integers $s_1, \ldots, s_n$ find a subsequence $s_{i_1} > s_{i_2} > \ldots > s_{i_k}$ with $i_1 < \ldots < i_k$ so that k is the largest possible. For example, for the sequence 15, 27, 14, 38, 63, 55, 46, 65, 85, the length of the LDS is 3 and that sequence is 63, 55, 46. In the case of the following sequence 50, 3, 10, 7, 40, 80, the LDS is 50, 10, 7 of length 3.

1. Give a recursive function for computing this problem
2. Using your recurrence, give a dynamic programming algorithm
3. Test your algorithm on this simple sequence [5,0,3,2,1,8]

Dynamic Programming

1. Give a recurrence relation for this problem

   int M(i, j)
   if $j == 1$ **then return** 1 ;
   else
   return 1 + $max_{1 \leq k < j}\{\text{M}(1, k)$ and $s_k > s_j\}$;

   $L[j] = 1 + max\{L[i] : i < j$ and $s_i > s_j\}$ (where max equal 1 if $j == 1$)

2. Using your recurrence relation, give a dynamic programming algorithm

   function LDS(s)
   for j=1 to n do
   L[j] = 1
   P[j] = 0
   for i=1 to j-1 do
   if ($s_i > s_j \& L[i] + 1 > L[j]$) then
   P[j] = i
   L[j] = L[i]+1

3. Test your algorithm on this simple sequence [5,0,3,2,1,8]