# Dynamic Programming

Chapter 6 of Dasgupta

Please READ the whole chapter

# Dynamic Programming

The material being addressed corresponds - *but does not strictly follow* - chapter 6 of the book Algorithms by Dasgupta et al.

You should **read** the **whole** chapter.

Do not worry if there are things that you do not fully understand at first, the tutorial coming Thursday should make it all clear.

# We learned already…

On Lecture 2 we learned about NP, P and NPC.

We also learned how to prove membership to NPC using polynomial reduction.

We know now how to show that a problem is indeed hard.

But are the ‘easy’ problems always ***easy*** to identify as such?

# First impressions may deceive…

Now we consider problems that may seem difficult **until** we approach them efficiently.

This week we use **dynamic programming**.

We will understand that dynamic programming is fundamentally different from recursion, despite the similarities to the eye.

# Topics

The idea behind Dynamic Programming.

The magic of Dynamic Programing.

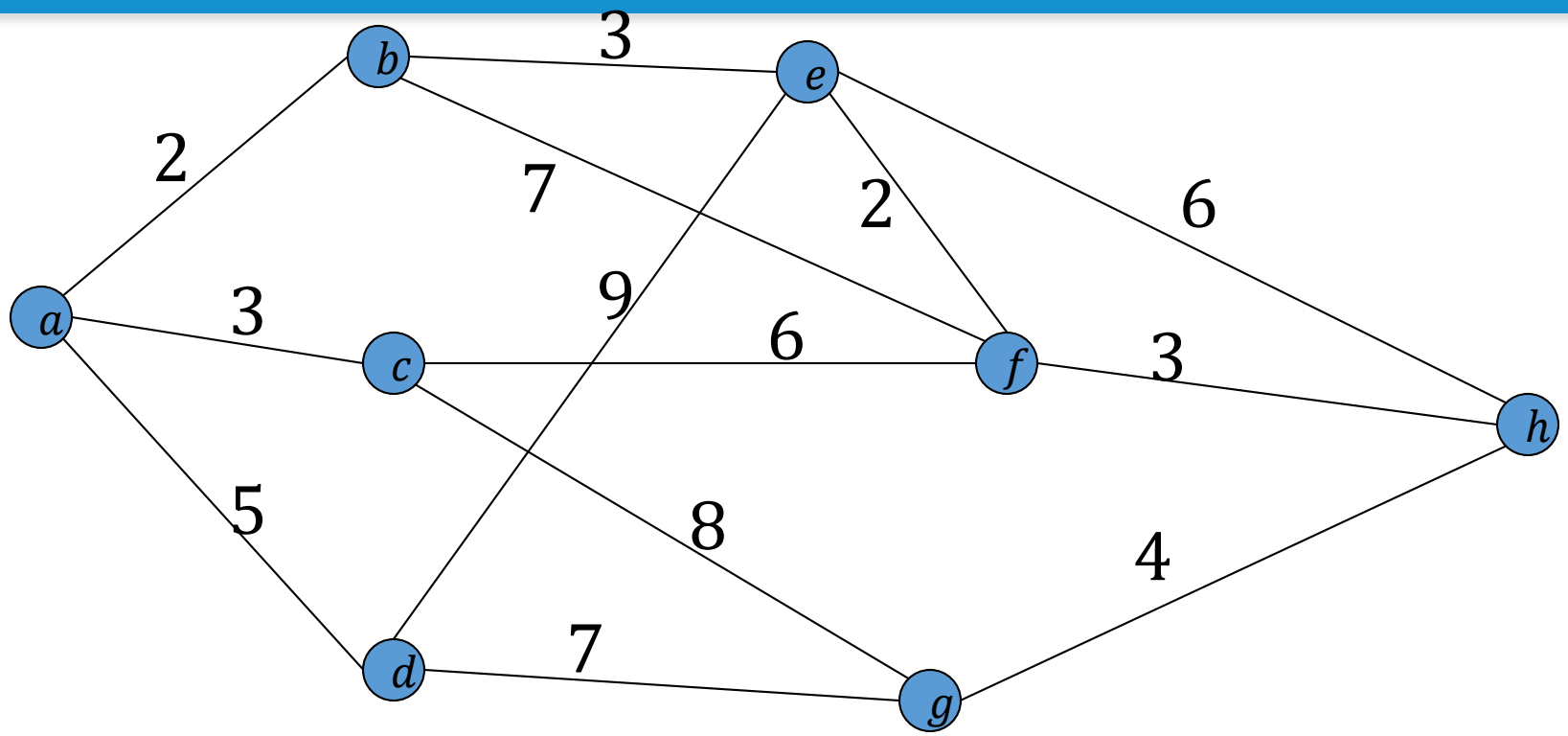
The relation with ‘divide and conquer’.

Pseudo polinomiality.

Three faces of dynamic programming.

The curse of dimensionality.

# Motivation




Suppose you need to travel from $a$ to $h$ and pay the least possible amount of money. You know friends who live in all cities you may travel to. How do you find the cheapest journey?

# Dynamic Programming

DP is a method for solving certain kind of problems. DP can be applied when the solution of a problem can be found from solutions to sub-problems.

We need to find a **recursive formula** for the solution. We can **recursively** solve sub-problems, starting from the trivial case, and save their solutions in memory.

In the end we'll get the solution of the whole problem.

**NOTE**: the principle of optimality (**validity** of the recursive formula) is fundamental!
**NOTE**: a recursive formula satisfying the principle of optimality is called a ***Bellman equation***.
**NOTE**: despite the recursive nature of dynamic programming it is **not** the same as **implementing a recursive function**!

# Properties needed from the problem

- Simpler Sub-problems
  - We should be able to break the original problem to smaller sub-problems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of sub-problem solutions
- Sub-problem Overlap
  - Unlike ‘branch and bound’ - which is based on a partition of the solution space - sub-problems may overlap

# Dynamic Programming

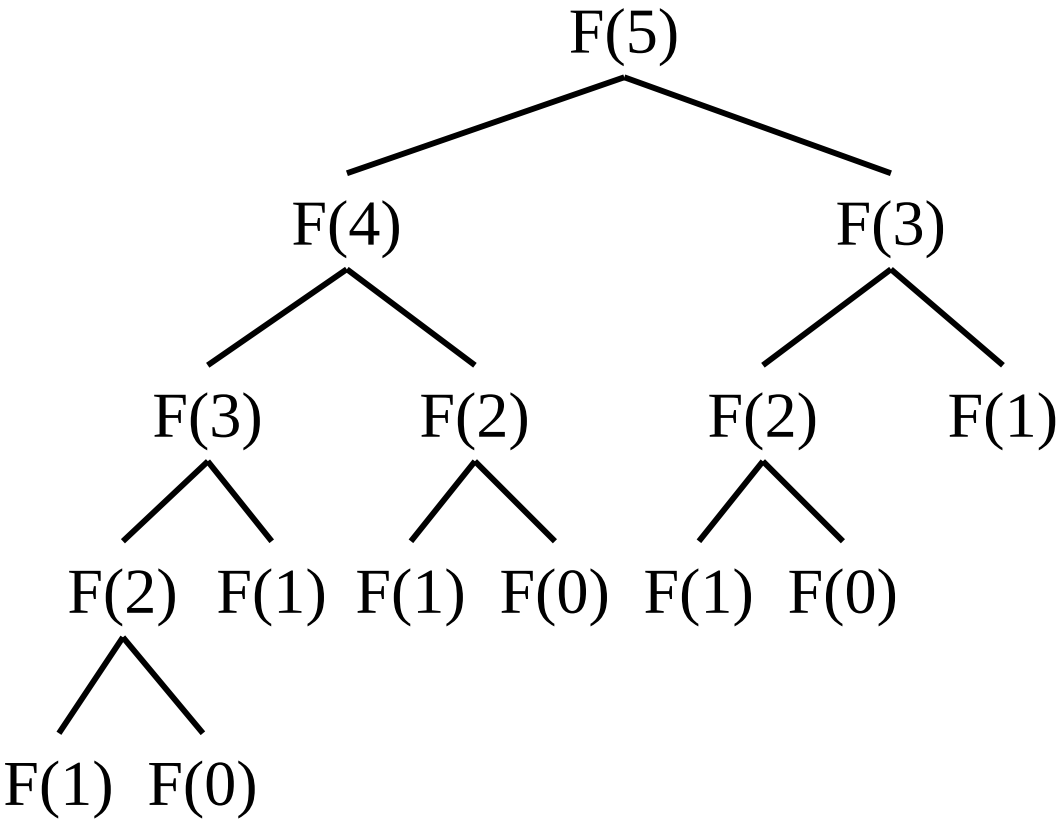
- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Optimal solution to the problem contains optimal solutions to sub-problems
- Compute the value of an optimal solution in a bottom-up fashion
- Use ‘table’ (the state space) to save the solutions to sub-problems
- **Avoid repeated work**

# Fibonacci numbers

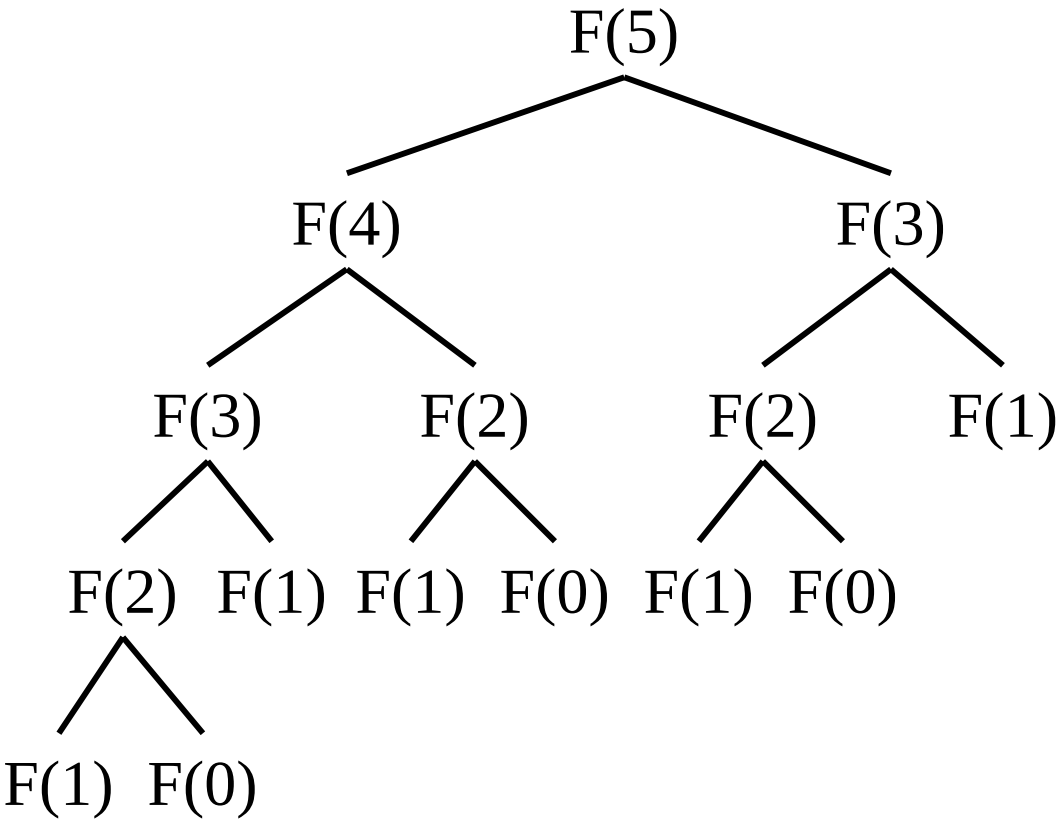
F(1) = F(0) = 1
F(n) = F(n-1) + F(n-2)

```
fib( n:integer )
{
  if ( n==0 or n==1 )
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

# Running Time of Algorithm 1

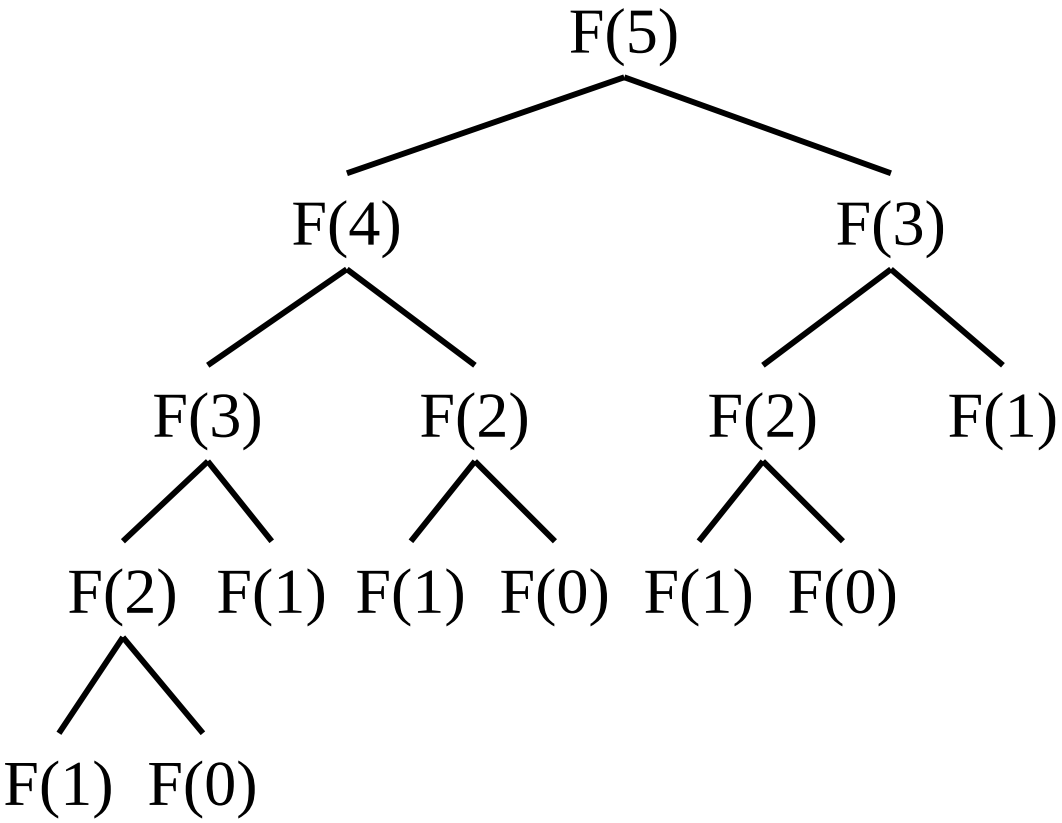



# Running Time of Algorithm 1




$$T(n) = k + T(n-1) + T(n-2) \approx T(n-1) + T(n-2)$$

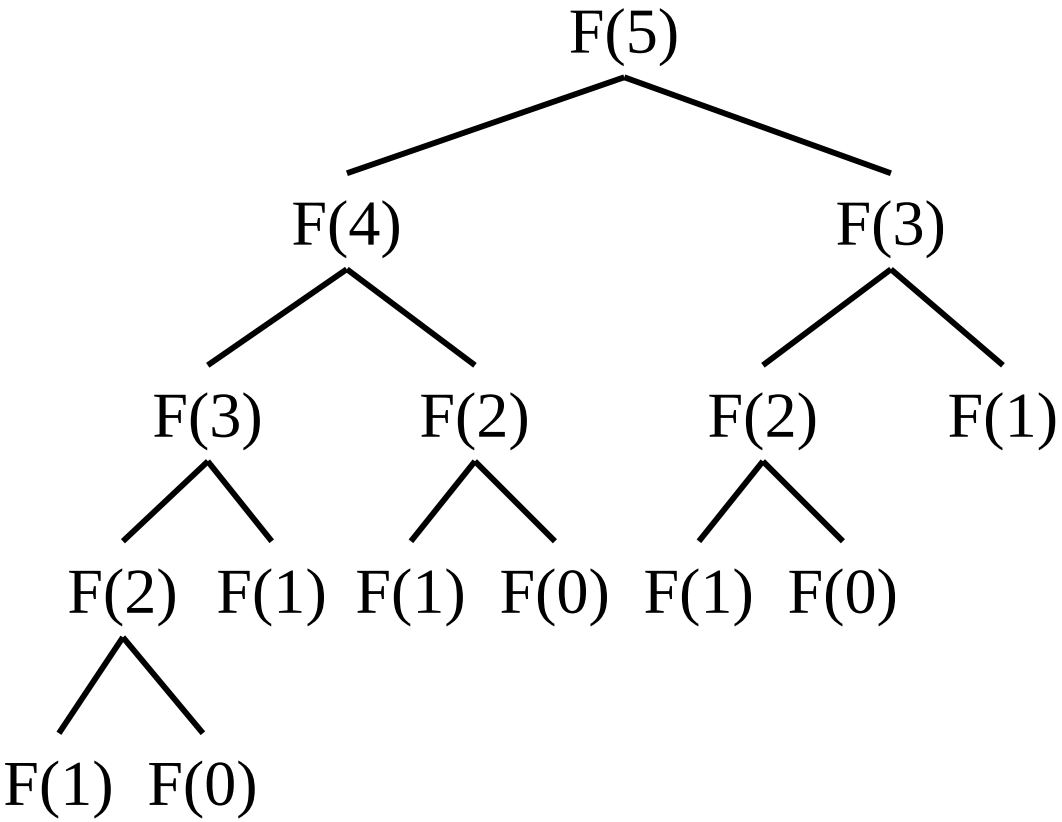
# Running Time of Algorithm 1




Running time approximately equal to the Fibonacci number itself!

See https://en.wikipedia.org/wiki/Fibonacci_number#Closed_form_expression for $F(n) = \text{round}(\frac{\varphi^n}{\sqrt{5}})$ with $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.62$

# Running Time of Algorithm 1




Running Time is $O(\varphi^n) = O(2^n)$

# Algorithm 2

```
fib( n:integer )
{   f[0]=1; f[1]=1;
    for i=2 to n
    {
       f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

Time complexity is $O(n)$.

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| F(n) | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |

# Dynamic Programming is **not** recursion

The difference between algorithm 1 and algorithm 2 to compute Fibonacci numbers illustrates the difference between **recursion** and **dynamic programming**:

Dynamic programming builds new states of computation from previous states without repeating the same computations.

Note: Fibonacci numbers are so special that in fact can be computed even faster…

# Algorithm 3

```
fib( n:integer )
{
 phi = (1+sqrt(5))/2;
 return round(phi^n / sqrt(5));
}
```

Time complexity is $O(1)$.

Strictly speaking we do not need dynamic programming to evaluate the Fibonacci number of order $n$. We did use it to explain the difference between DP and recursion.

# Each Dynamic Programming **assignment** requires the following steps in the answer

(1) The subproblems to solve.

(2) The optimal value, expressed in terms of the subproblems

(3) Initial values.

(4) The recurrence

(5) Analysis of the used space.

(6) Analysis of the running time.

# LCS

Longest common subsequence

- `x = "sariempiolcewe"`
- `y = "westigmupsalrte"`

# LCS

Longest common subsequence

- x = “**s**ar**i**e**mp**io**l**cew**e**”
- y = “we**s**t**i**g**m**u**p**sa**l**rt**e**”

# LCS

Longest common subsequence

- x = "**s**ar**i**e**mp**io**l**cew**e**"
- y = "we**s**t**i**g**m**u**p**sa**l**rt**e**"

$$LCS(X_m, Y_n) = \begin{cases} 1 + LCS(X_{m-1}, Y_{n-1}) & \text{if } X_m = Y_n \\ \max\{LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})\} & \text{otherwise} \end{cases}$$

$$LCS(\emptyset, Y) = LCS(X, \emptyset) = 0$$

Solution: Proceed from the end of the strings and

- If $x_m = y_n$ add 1 to the number of symbols in LCS($x_{m-1}$, $y_{n-1}$)
- If $x_m \neq y_n$
  - Skip last symbol from *x or* last symbol from *y*
  - Decide which symbol to skip by comparing LCS($x_m$, $y_{n-1}$) and LCS($x_{m-1}$, $y_n$)

# Note how the first four steps have been done…

(✓) The subproblems to solve.

(✓) The optimal value, expressed in terms of the subproblems

(✓) Initial values.

(✓) The recurrence

(5) Analysis of the used space.

(6) Analysis of the running time.

# LCS Implementation

```
int lcsRec(int i, int j) {
  if (i==0 || j==0) return 0;
  else if (x[i] == y[j])
    return lcsRec(i-1,j-1) + 1;
  else
    return max(lcsRec(i-1,j),lcsRec(i,j-1));
}
```

Recurrence for time complexity:

$T(2n) = T(2n-2)+k$ if $x[n]=y[n]$

$T(2n) = 2T(2n-1)+k$ if $x[n]!=y[n]$

$T(2n) = k$ if $n=0$

$$
\begin{aligned}
T(2n) &= 2T(2n-1) \\
&= 2(2T(2n-2)) \\
&= 4T(2n-2) \\
&= 4(2T(2n-3)) \\
&= 8T(2n-3) \\
&= \ldots \\
&= 2^{i}T(2n-i) \\
&= 2^{2n} = 4^{n}
\end{aligned}
$$

# Remember the cliffhanger?

Previous implementation computes the **optimal value** of LCS

**Question**: extend it to ***find*** the solution!

**Question**: Is LCS it in NP?

**Question**: Is LCS hard?

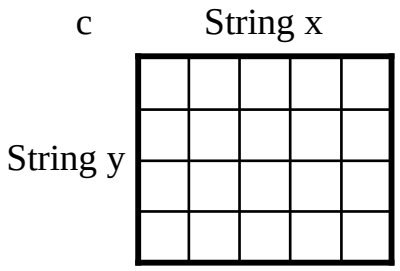
# Check the companion notebook

# LCS Implementation

```
int lcsRec(int i, int j) {
  if (i==0 || j==0) return 0;
  else if (x[i] == y[j])
    return lcsRec(i-1,j-1) + 1;
  else
    return max(lcsRec(i-1,j),lcsRec(i,j-1));
}
```

# LCS alternative implementation

```
int lcsMemo(int i, int j) {
  if (c[i][j] != -1) return c[i][j];
  else if (x[i] == y[j]) {
    c[i][j] = lcsMemo(i-1,j-1) + 1;
    return c[i][j];
  }
  else {
    c[i][j] = max(lcsMemo(i-1,j),lcsMemo(i,j-1));
    return c[i][j]; }
  }
```




$$T(n) = O(n^2)$$

# Runtime Comparison

| N | Recursive | Memo |
|---|---|---|
| 10 | 0.004 | 0.002 |
| 15 | 0.774 | 0.002 |
| 17 | 4.760 | 0.003 |
| 18 | 11.216 | 0.003 |
| 20 | 154.464 | 0.003 |
| 100 | | 0.005 |

# Are we there yet?

The memo version of LCS is a version of DP.

DP can be implemented in a forward or in a backward fashion.

The memo version is the backward fashion.

We now illustrate the forward version. This is DP is a systematic way to fill the table.

Both versions differ in the way you read the same bellman equation…

# Longest Common Subsequence

Instance: two sequences of strings A, and B.
Find a longest common subsequence.
Example:

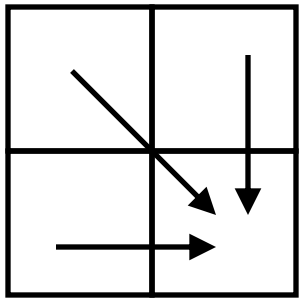
```
AAAATTGA
TAACGATA
Sol: AAATA
```

How about?

```
ATTTAAAAATTCTTTTTGGGGATAT
AATATGGTTTCTTTGATATGGG
```

TACCBT

ATBCBD



| i \ j | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | | A | T | B | C | B | D |
| 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | T | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | C | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| 4 | C | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| 5 | B | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 6 | T | 0 | 1 | 2 | 2 | 2 | 3 | 3 |

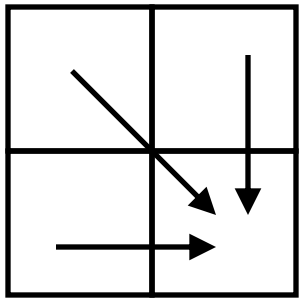
$L(i,j) = 0$ if i=0 or j=0

$L(i,j) = L(i-1,j-1) + 1$ if i,j > 0 and x[i] = y[j]

$L(i,j) = \max\{ L(i-1,j), L(i,j-1) \}$ if i,j > 0 and x[i] != y[j]

ABDDBE

BEBEDE



| i \ j | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | | B | E | B | E | D | E |
| 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | | | | | | |
| 2 | B | 0 | | | | | | |
| 3 | D | 0 | | | | | | |
| 4 | D | 0 | | | | | | |
| 5 | B | 0 | | | | | | |
| 6 | E | 0 | | | | | | |

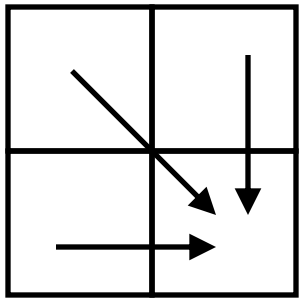
$L(i,j) = 0$ if i=0 or j=0

$L(i,j) = L(i-1,j-1) + 1$ if i,j > 0 and x[i] = y[j]

$L(i,j) = \max\{ L(i-1,j), L(i,j-1) \}$ if i,j > 0 and x[i] != y[j]

ABDDBE

BEBEDE



| i \ j | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | | B | E | B | E | D | E |
| 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | B | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | D | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| 4 | D | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| 5 | B | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| 6 | E | 0 | 1 | 2 | 2 | 3 | 3 | 3 |

$L(i,j) = 0$ if i=0 or j=0

$L(i,j) = L(i-1,j-1) + 1$ if i,j > 0 and x[i] = y[j]

$L(i,j) = \max\{ L(i-1,j), L(i,j-1) \}$ if i,j > 0 and x[i] != y[j]

# Questions about DP for LCS

Previous implementation computes the optimal value.

**Question**: what is the running time of the previous algorithm?

**Question**: what is the minimum space needed by the previous algorithm?

**Question**: extend the previous algorithm to find the optimal solution.

**Question**: what are the running time and space requirements of the new algorithm, that finds the optimal solution?

# The six steps, in more detail.

(1) What is the sub problem to solve? (In other words: What will be in the table?)

(2) What is the optimal value, expressed in terms of the subproblems? (In other words: How do you find the optimal value once the table is filled?)

(3) What are the initial values? (In other words: What values can you fill in right away?)

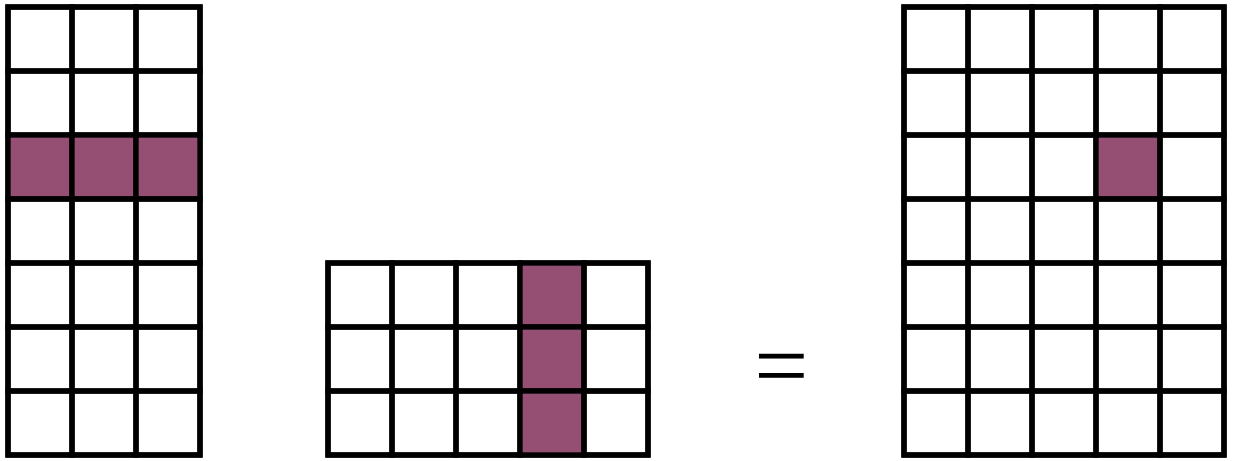
(4) What recurrence is used?. (In other words: Given the initial values, how to compute the rest?)

(5) What is the used space? (In other words: What is the size of the DP table?)

(6) What is the running time? (This is usually the size of the table times the time it takes to compute one value of the table.)

Note: may you need an additional step to find the solution, as in the LCS by backtracking on the table, you should provide an analysis of its time and memory as well.

# Matrix Multiplication




A: $p \times q$

B: $q \times r$

$A \times B$: $p \times r$ requires $p \times q \times r$ scalar multiplications

# Matrix-Chain Multiplication

4 matrices: A B C D

Sizes: 50x10, 10x40, 40x30, 30x5

( ( ( A B ) C ) D ) : 87500 multiplications

( ( A ( B C ) ) D ) : 34500 multiplications

( ( A B ) ( C D ) ) : 36000 multiplications

( A ( ( B C ) D ) ) : 16000 multiplications

( A ( B ( C D ) ) ) : 10500 multiplications

# How many ways are there?

$A_1, A_2, A_3, \ldots, An$

Let $T(n)$ be the **number of ways** to compute the product of *n* matrices.

Suppose the last multiplication is

- $(A_1, A_2, \ldots, Ai\ )\ (A_{i+1}, \ldots, An), \quad 1 \le i \le n-1$

$T(i)\ T(n-i)$ ways for a fixed $i$

$T(n) = \sum_{i \le n\ \_\ 1} T(i)T(n-i)$

$T(1) = 1$

| *n* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| T(*n*) | 1 | 1 | 2 | 5 | 14 | 42 | 132 | 429 | 1430 | 4862 |

See https://en.wikipedia.org/wiki/Catalan_number

# Notation

Let m(L,R) be the optimal **number of multiplications** needed to compute

$A_L A_{L+1} \ldots A_R$ where $1 \leq L$ and $R \leq n$

We want to compute $m(1, n)$

$m(L, R) = 0$ if $L = R$

$m(L, R) = ?$ if $L < R$

$A_i$ is $c_{i-1}$ by $c_i$ matrix

# Optimal ordering of matrix multiplication

Suppose the last multiplication is

- $(A_L, A_{L+1}, \ldots, A_i)\,(A_{i+1}, \ldots, A_R)$
- where $L \leq i \leq R-1$

$m(L,R) = m(L,i) + m(i+1,R) + c_{L-1}\, c_i\, c_R$

We don't know the value of $i$.

We have to try them all

$$m(L,R) = \min_{L \leq i \leq R-1} \{m(L,i) + m(i+1,R) + c_{L-1} c_i c_R\}$$

| L \ R | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | | | | | |
| 2 | - | 0 | | | | |
| 3 | - | - | 0 | | | |
| 4 | - | - | - | 0 | | |
| 5 | - | - | - | - | 0 | |
| 6 | - | - | - | - | - | 0 |

$m(L,R) = 0$ if $L = R$,

$m(L,R) = \min_{L \le i \le R}\{ m(L,i) + m(i+1,R) + c_{L-1} * c_i * c_R \}$ otherwise

$m(1,5) = \min_{1 \le i \le 4}\{ m(1,i) + m(i+1,5) + c_0 * c_i * c_5 \}$

$= \min \{ \; m(1,1) + m(2,5) + c_0 * c_1 * c_5 ,$

$m(1,2) + m(3,5) + c_0 * c_2 * c_5 ,$

$m(1,3) + m(4,5) + c_0 * c_3 * c_5 ,$

$m(1,4) + m(5,5) + c_0 * c_4 * c_5 \; \}$

| L \ R | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 224 | 176 | **218** | 276 | 350 |
| 2 | - | 0 | 64 | **112** | 174 | 250 |
| 3 | - | - | 0 | **24** | 70 | 138 |
| 4 | - | - | - | **0** | 30 | 90 |
| 5 | - | - | - | - | 0 | 90 |
| 6 | - | - | - | - | - | 0 |

| $i$ | $c_i$ |
|---|---|
| 0 | 7 |
| 1 | 8 |
| 2 | 4 |
| 3 | 2 |
| 4 | 3 |
| 5 | 5 |
| 6 | 6 |

$$m(1,4) = \min_{1 \le i \le 3}\{ m(1,i) + m(i+1,4) + c_0 * c_i * c_4 \}$$

$$= \min \{ \; m(1,1) + m(2,4) + c_0 * c_1 * c_4 \,,$$
$$m(1,2) + m(3,4) + c_0 * c_2 * c_4 \,,$$
$$m(1,3) + m(4,4) + c_0 * c_3 * c_4 \; \}$$

$$= \min\{ \quad 0 \quad + \quad 112 \quad + 7 * 8 * 3,$$
$$224 \; + \quad 24 \quad + 7 * 4 * 3,$$
$$176 \; + \quad 0 \quad + 7 * 2 * 3 \quad \}$$

$$= \min\{ 280, 332, 218 \} = 218$$

Running time $O(n^3)$

# Questions about DP for CMM

Previous implementation computes the optimal value of Chained Matrix Multiplication.

**Question**: what is the running time of the previous algorithm?

**Question**: what is the minimum space needed by the previous algorithm?

**Question**: extend it to solve the **find** the optimal solution.

**Question**: what are the running time and space requirements of the new algorithm, that finds the optimal solution?



VU

# 0-1 Knapsack problem

Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items

Each item *i* has some weight $w_i$ and benefit value $b_i$ (all $w_i$, $b_i$ and *W* are integer values)

Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack Problem is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

# Recursive Formula for subproblems

It means, that the best subset of $S_k$ that has total weight $w$ is one of the two:

1) the best subset of $S_{k-1}$ that has total weight $w$, **or**

2) the best subset of $S_{k-1}$ that has total weight $w$-$w_k$ plus the item $k$

- B[k,w] means the best we can get with items 1..k and weight no more than w
- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

# Recursive Formula

The best subset of $S_k$ that has the total weight $w$, either contains item $k$ or not.

First case: $w_k > w$. Item $k$ can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable

Second case: $w_k <= w$. Then the item $k$ can be in the solution, and we choose the case with greater value

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

# 0-1 Knapsack Algorithm

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
      if w_i <= w // item i can be part of the solution
          if b_i + B[i-1,w-w_i] > B[i-1,w]
              B[i,w] = b_i + B[i-1,w- w_i]
          else
              B[i,w] = B[i-1,w]
      else B[i,w] = B[i-1,w]  // w_i > w
```

# Running time

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
      < the rest of the code >
```

$O(W)$

Repeat $n$ times

$O(W)$

What is the running time of this algorithm?

$O(n*W)$

Straightforward algorithm takes $O(2^n)$

# Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

# Example (2)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | | | | |
| 1 | 0 | | | | |
| 2 | 0 | | | | |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

for w = 0 to W
    B[0,w] = 0

# Example (3)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | | |
| 2 | 0 | | | | |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

```
for i = 0 to n
    B[i,0] = 0
```

# Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | |
| 2 | 0 | | | | |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
        B[i,w] = $b_i$ + B[i-1,w- $w_i$]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i$ > w

# Example (5)

Items:

| |
|---|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | |
| 2 | 0 | **3** | | | |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    **$B[i,w] = b_i + B[i-1,w- w_i]$**
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | |
| 2 | 0 | 3 | | | |
| 3 | 0 | **3** | | | |
| 4 | 0 | | | | |
| 5 | 0 | | | | |

i=1

$b_i$=3

$w_i$=2

w=3

w-$w_i$=1

if $w_i$ <= w // item i can be part of the solution

  if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]

    **B[i,w] = $b_i$ + B[i-1,w- $w_i$]**

  else

    B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w]  // $w_i$ > w

# Example (7)

Items:

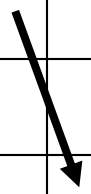
1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | |
| 2 | 0 | 3 | | | |
| 3 | 0 | 3 | | | |
| 4 | 0 | **3** | | | |
| 5 | 0 | | | | |



$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    **$B[i,w] = b_i + B[i-1,w- w_i]$**

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (8)

Items:

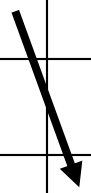
1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | |
| 2 | 0 | 3 | | | |
| 3 | 0 | 3 | | | |
| 4 | 0 | 3 | | | |
| 5 | 0 | **3** | | | |



$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    **$B[i,w] = b_i + B[i-1,w- w_i]$**

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (9)

Items:

| |
|---|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | **0** | | |
| 2 | 0 | 3 | | | |
| 3 | 0 | 3 | | | |
| 4 | 0 | 3 | | | |
| 5 | 0 | 3 | | | |

i=2

$b_i$=4

$w_i$=3

w=1

$w-w_i$=-2

if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
        B[i,w] = $b_i$ + B[i-1,w- $w_i$]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i$ > w

# Example (10)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | | |
| 2 | 0 | 3 | **3** | | |
| 3 | 0 | 3 | | | |
| 4 | 0 | 3 | | | |
| 5 | 0 | 3 | | | |

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

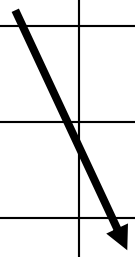
if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
        B[i,w] = $b_i$ + B[i-1,w- $w_i$]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i$ > w

# Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | | |
| 2 | 0 | 3 | 3 | | |
| 3 | 0 | 3 | **4** | | |
| 4 | 0 | 3 | | | |
| 5 | 0 | 3 | | | |



i=2

$b_i$=4

$w_i$=3

w=3

w-$w_i$=0

if $w_i$ <= w // item i can be part of the solution
  if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
    **B[i,w] = $b_i$ + B[i-1,w- $w_i$]**
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // $w_i$ > w

# Example (12)

Items:

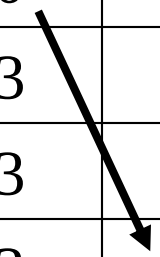
1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | | |
| 2 | 0 | 3 | 3 | | |
| 3 | 0 | 3 | 4 | | |
| 4 | 0 | 3 | **4** | | |
| 5 | 0 | 3 | | | |



$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    **$B[i,w] = b_i + B[i-1,w- w_i]$**

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (13)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | | |
| 2 | 0 | 3 | 3 | | |
| 3 | 0 | 3 | 4 | | |
| 4 | 0 | 3 | 4 | | |
| 5 | 0 | 3 | **7** | | |

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    **$B[i,w] = b_i + B[i-1,w- w_i]$**

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | **0** | |
| 2 | 0 | 3 | 3 | **3** | |
| 3 | 0 | 3 | 4 | **4** | |
| 4 | 0 | 3 | 4 | | |
| 5 | 0 | 3 | 7 | | |

i=3

$b_i$=5

$w_i$=4

w=1..3

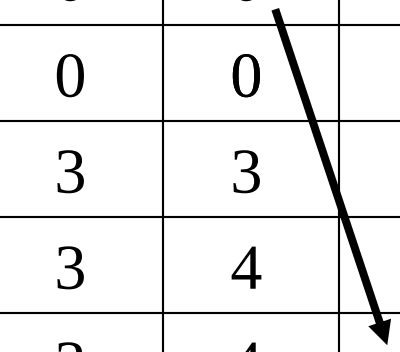
if $w_i$ <= w // item i can be part of the solution
    if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
        B[i,w] = $b_i$ + B[i-1,w- $w_i$]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i$ > w

# Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 3 | 3 | 3 | |
| 3 | 0 | 3 | 4 | 4 | |
| 4 | 0 | 3 | 4 | **5** | |
| 5 | 0 | 3 | 7 | | |



$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    **$B[i,w] = b_i + B[i-1,w- w_i]$**
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | |
| 2 | 0 | 3 | 3 | 3 | |
| 3 | 0 | 3 | 4 | 4 | |
| 4 | 0 | 3 | 4 | 5 | |
| 5 | 0 | 3 | 7 → | **7** | |

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    **B[i,w] = B[i-1,w]**
else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Example (16)

Items:

| |
|---|
| 1: (2,3) |
| 2: (3,4) |
| 3: (4,5) |
| 4: (5,6) |

| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 → | **0** |
| 2 | 0 | 3 | 3 | 3 → | **3** |
| 3 | 0 | 3 | 4 | 4 → | **4** |
| 4 | 0 | 3 | 4 | 5 → | **5** |
| 5 | 0 | 3 | 7 | 7 | |

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

# Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

| w \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 3 | 3 | 3 | 3 |
| 3 | 0 | 3 | 4 | 4 | 4 |
| 4 | 0 | 3 | 4 | 5 | 5 |
| 5 | 0 | 3 | 7 | 7 → | **7** |

$i=3$

$b_i=5$

$w_i=4$

$w=5$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    **B[i,w] = B[i-1,w]**
else $B[i,w] = B[i-1,w]$ // $w_i > w$

# Comments

This algorithm only finds the max possible value that can be carried in the knapsack.

**Question**: what is the running time of the previous algorithm?

**Question**: what is the minimum space needed by the previous algorithm?

**Question**: extend it to find the optimal solution.

**Question**: what are the running time and space requirements of the optimization version?

# Notes about DP for the knapsack

The time and space used depend on the capacity W of the knapsack. This number can be arbitrarily large!

For any W given a priori the algorithm is polynomial.

This is an example of a **pseudo polynomial** algorithm.

The knapsack problem is NPC!

# Polynomial-Time Approximation Schemes

A problem L has a fully polynomial-time approximation scheme (FPTAS) if it has a polynomial-time (in n and $1/\varepsilon$) $(1-\varepsilon)$-approximation algorithm, for any fixed $\varepsilon > 0$.

*Remember that knapsack is a maximization problem*

0/1 Knapsack has a FPTAS, with a running time that is $O(n^3 / \varepsilon)$.

We show this with an example.

Note: this is 9.2.4 in Dasgupta.

Note: next week formal definition of approximation.

# Knapsack Problem: other version of Dynamic Programming

Define OPT(i, p) = min weight subset of items 1, …, i that yields value exactly p.

- Case 1: OPT does not select item i.
  - OPT selects best of 1, …, i-1 that achieves exactly value p
- Case 2: OPT selects item i.
  - consumes weight $w_i$, new value needed = $p - p_i$
  - OPT selects best of 1, …, i-1 that achieves exactly value $p - p_i$

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 0 or v = 0

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | | | | | | | | | | | | | | | |
| 2 | 0 | | | | | | | | | | | | | | | |
| 3 | 0 | | | | | | | | | | | | | | | |
| 4 | 0 | | | | | | | | | | | | | | | |
| 5 | 0 | | | | | | | | | | | | | | | |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 1 , v = …

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | | | | | | | | | | | | | | | |
| 3 | 0 | | | | | | | | | | | | | | | |
| 4 | 0 | | | | | | | | | | | | | | | |
| 5 | 0 | | | | | | | | | | | | | | | |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 2 , v = …

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | 1 | 3 | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 3 | 0 | | | | | | | | | | | | | | | |
| 4 | 0 | | | | | | | | | | | | | | | |
| 5 | 0 | | | | | | | | | | | | | | | |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 3 , v = …

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | 1 | 3 | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 3 | 0 | 1 | 3 | 5 | 6 | 8 | x | x | x | x | x | x | x | x | x | x |
| 4 | 0 | | | | | | | | | | | | | | | |
| 5 | 0 | | | | | | | | | | | | | | | |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 4, v = …

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | 1 | 3 | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 3 | 0 | 1 | 3 | 5 | 6 | 8 | x | x | x | x | x | x | x | x | x | x |
| 4 | 0 | 1 | 3 | 5 | 6 | 7 | 9 | 11 | 12 | 14 | x | x | x | x | x | x |
| 5 | 0 | | | | | | | | | | | | | | | |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

# Knapsack: FPTAS

$$OPT(i,p) = \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0, p > 0 \\ OPT(i-1,p) & \text{if } p_i > p \\ \min\{OPT(i-1,p), w_i + OPT(i-1,p-p_i)\} & \text{otherwise} \end{cases}$$

i = 5, v = 8

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | 1 | 3 | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 3 | 0 | 1 | 3 | 5 | 6 | 8 | x | x | x | x | x | x | x | x | x | x |
| 4 | 0 | 1 | 3 | 5 | 6 | 7 | 9 | 11 | 12 | 14 | x | x | x | x | x | x |
| 5 | 0 | 1 | 3 | 5 | 6 | 7 | 7 | 8 | 10 | 12 | 13 | 14 | 16 | 18 | 19 | 21 |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

S={1, 2, 5}

# Knapsack: FPTAS Tracing Solution

←

i = 5, v = 8

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 1 | 0 | 1 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 2 | 0 | 1 | 3 | x | x | x | x | x | x | x | x | x | x | x | x | x |
| 3 | 0 | 1 | 3 | 5 | 6 | 8 | x | x | x | x | x | x | x | x | x | x |
| 4 | 0 | 1 | 3 | 5 | 6 | 7 | 9 | 11 | 12 | 14 | x | x | x | x | x | x |
| 5 | 0 | 1 | 3 | 5 | 6 | 7 | 7 | 8 | 10 | 12 | 13 | 14 | 16 | 18 | 19 | 21 |

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

W = 11

S={1, 2, 5}

# Knapsack: FPTAS

Intuition for approximation algorithm.

- Round all values to lie in smaller range.
- Run dynamic programming algorithm II on rounded instance.
- Return optimal items in rounded instance.

| Item | Value | Weight |
|---|---|---|
| 1 | 3,934,221 | 1 |
| 2 | 5,956,342 | 2 |
| 3 | 11,810,013 | 5 |
| 4 | 21,217,800 | 6 |
| 5 | 27,343,199 | 7 |

original instance

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 5 |
| 4 | 4 | 6 |
| 5 | 6 | 7 |

rounded instance after dividing by 5,000,000

W = 11

S = { 1, 2, 5 }

# Comparison between DP and Divide-and-Conquer

The solution to the problems is constructed from the solutions to the subproblems.

Divide-and-conquer: merge sort

- Any division of the problem into subproblems leads to the same solution
- Disjoint subproblems
- Top-down computation

Dynamic programming: matrix-chain multiplication

- Different divisions of the problem leads to different solutions. Among them one of them may lead optimal solution.
- Overlapping subproblems
- Bottom-up computation

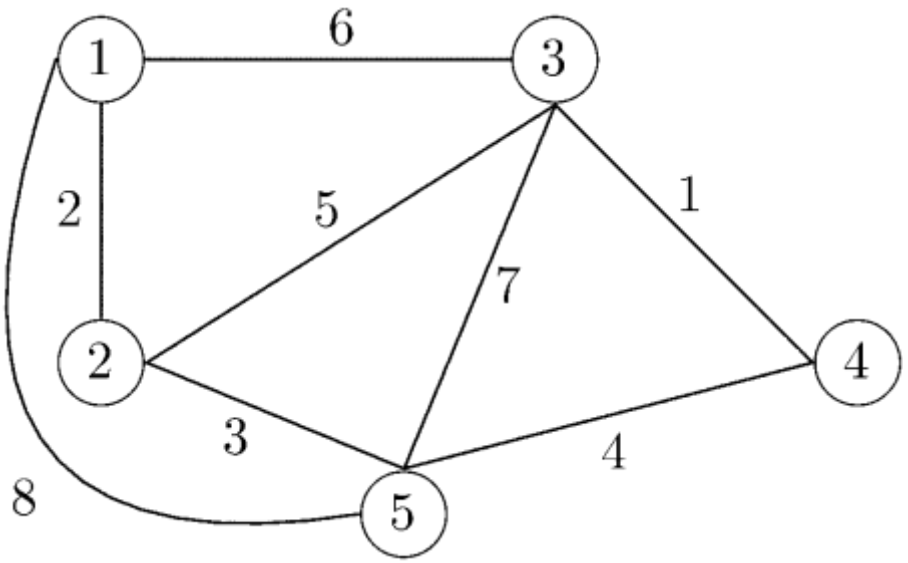
# Three faces of Dynamic Programming

Dynamic programming comes in **at least** three flavors
The differences are in number of operations and memory usage
In other words: the difference are in memory and time complexity

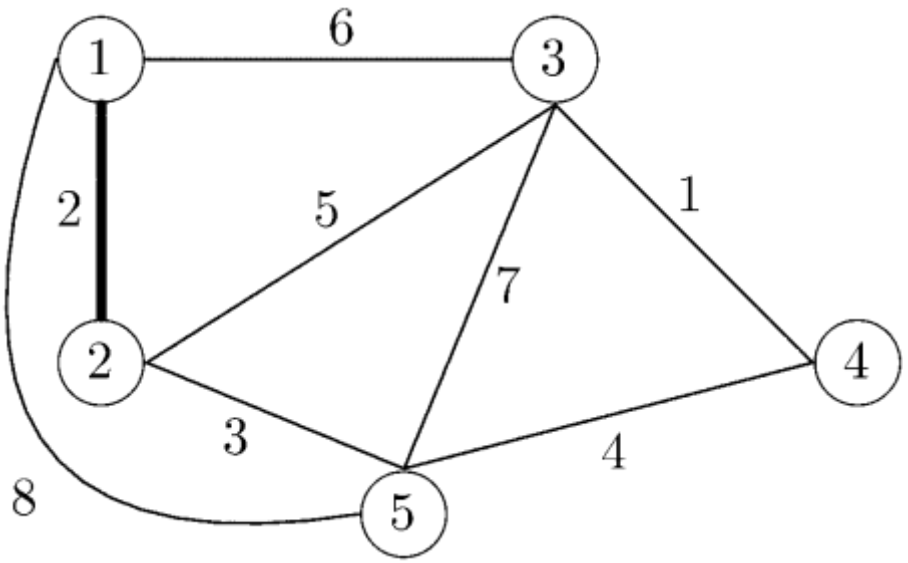
# Type one: minimum spanning tree

$$\begin{cases} f_1(\{v_1\}) & = & 0 \\ f_{k+1}\left(S \cup \{\arg\min_{j \notin S}[d(S,j)]\}\right) & = & f_k(S) + d(S,j) \end{cases}$$

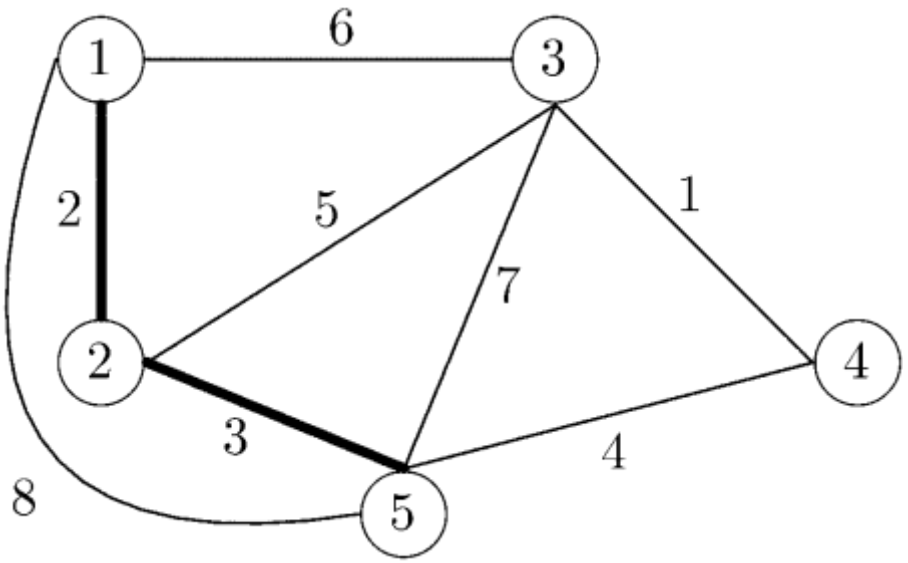
# minimum spanning tree



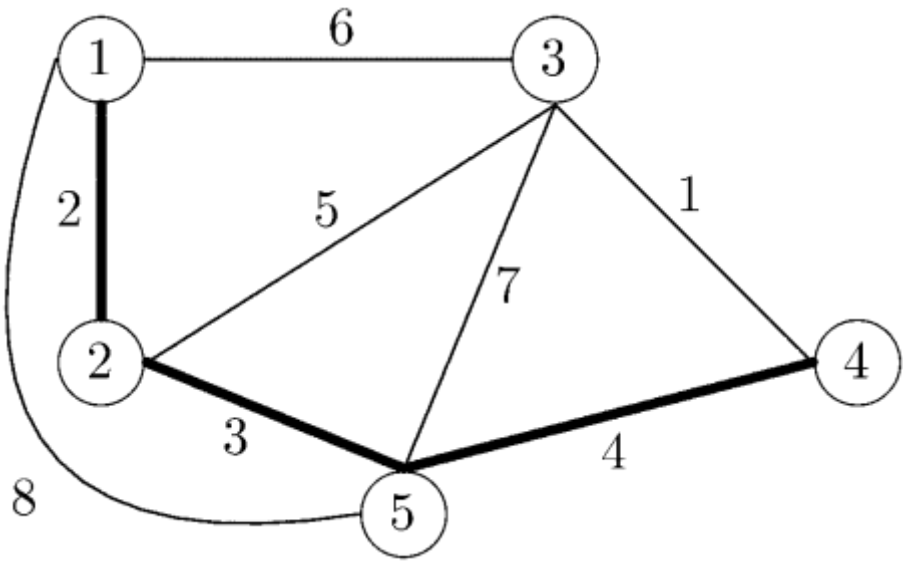
# minimum spanning tree



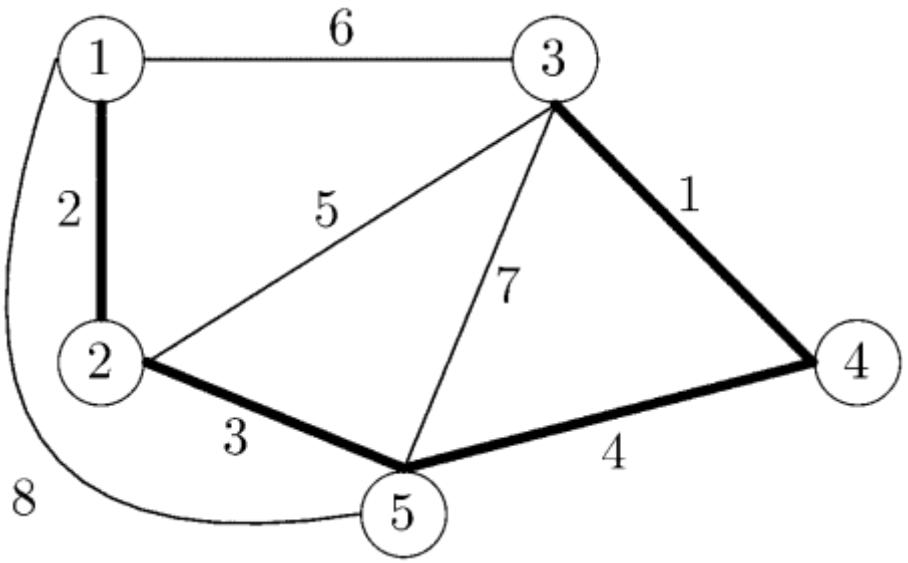
# minimum spanning tree



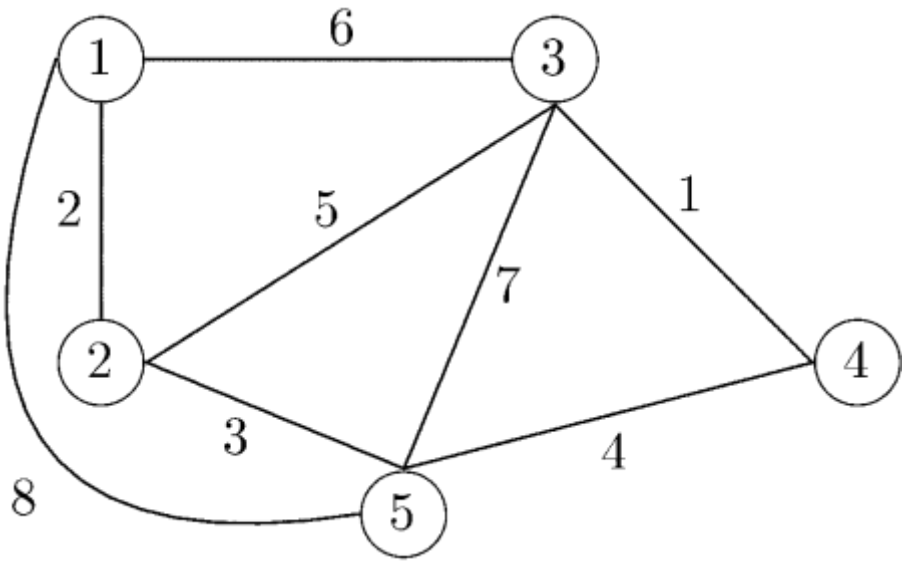
# minimum spanning tree



# minimum spanning tree



# minimum spanning tree



# minimum spanning tree

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

# minimum spanning tree

$$f(\{1\}) = 0$$

# minimum spanning tree

$$f(\{1\}) = 0$$
$$f(\{1,2\}) = 2$$

# minimum spanning tree

$$f(\{1\}) = 0$$
$$f(\{1,2\}) = 2$$
$$f(\{1,2,5\}) = 5$$

# minimum spanning tree

$$
\begin{aligned}
f(\{1\}) &= 0 \\
f(\{1,2\}) &= 2 \\
f(\{1,2,5\}) &= 5 \\
f(\{1,2,5,4\}) &= 9
\end{aligned}
$$

# minimum spanning tree

$$
\begin{aligned}
f(\{1\}) &= 0 \\
f(\{1,2\}) &= 2 \\
f(\{1,2,5\}) &= 5 \\
f(\{1,2,5,4\}) &= 9 \\
f(\{1,2,5,4,3\} = V) &= 10
\end{aligned}
$$

# Type two:
# all pairs shortest path

$$\begin{cases} f_0(j,k) &= d_{jk} \\ f_i(j,k) &= \min\limits_{\ell \in V}\{f_{i-1}(j,k), f_{i-1}(j,\ell) + f_{i-1}(\ell,k)\} \end{cases}$$

# all pairs shortest path

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

# all pairs shortest path

$$\begin{bmatrix} 0 & 2 & 6 & \infty & \underline{5} \\ 2 & 0 & 5 & \infty & 3 \\ 6 & 5 & 0 & 1 & 7 \\ \infty & \infty & 1 & 0 & 4 \\ \underline{5} & 3 & 7 & 4 & 0 \end{bmatrix}$$

# all pairs shortest path

$$\begin{bmatrix} 0 & 2 & 6 & \underline{7} & 5 \\ 2 & 0 & 5 & \underline{6} & 3 \\ 6 & 5 & 0 & 1 & 7 \\ \underline{7} & \underline{6} & 1 & 0 & 4 \\ 5 & 3 & 7 & 4 & 0 \end{bmatrix}$$

# all pairs shortest path

$$\begin{bmatrix} 0 & 2 & 6 & 7 & 5 \\ 2 & 0 & 5 & 6 & 3 \\ 6 & 5 & 0 & 1 & \underline{5} \\ 7 & 6 & 1 & 0 & 4 \\ 5 & 3 & \underline{5} & 4 & 0 \end{bmatrix}$$

# all pairs shortest path

$$\begin{bmatrix} 0 & 2 & 6 & 7 & 5 \\ 2 & 0 & 5 & 6 & 3 \\ 6 & 5 & 0 & 1 & 5 \\ 7 & 6 & 1 & 0 & 4 \\ 5 & 3 & 5 & 4 & 0 \end{bmatrix}$$

# “Nice” state space

$$F_0 = \begin{bmatrix} 0 & 2 & 6 & \infty & 8 \\ 2 & 0 & 5 & \infty & 3 \\ 6 & 5 & 0 & 1 & 7 \\ \infty & \infty & 1 & 0 & 4 \\ 8 & 3 & 7 & 4 & 0 \end{bmatrix} \qquad F_1 = \begin{bmatrix} 0 & 2 & 6 & \infty & \mathbf{5} \\ 2 & 0 & 5 & \infty & 3 \\ 6 & 5 & 0 & 1 & 7 \\ \infty & \infty & 1 & 0 & 4 \\ \mathbf{5} & 3 & 7 & 4 & 0 \end{bmatrix}$$

$$F_2 = \begin{bmatrix} 0 & 2 & 6 & \mathbf{7} & 5 \\ 2 & 0 & 5 & \mathbf{6} & 3 \\ 6 & 5 & 0 & 1 & 7 \\ \mathbf{7} & \mathbf{6} & 1 & 0 & 4 \\ 5 & 3 & 7 & 4 & 0 \end{bmatrix} \qquad F_3 = \begin{bmatrix} 0 & 2 & 6 & 7 & 5 \\ 2 & 0 & 5 & 6 & 3 \\ 6 & 5 & 0 & 1 & \mathbf{5} \\ 7 & 6 & 1 & 0 & 4 \\ 5 & 3 & \mathbf{5} & 4 & 0 \end{bmatrix} \qquad F_4 = \begin{bmatrix} 0 & 2 & 6 & 7 & 5 \\ 2 & 0 & 5 & 6 & 3 \\ 6 & 5 & 0 & 1 & 5 \\ 7 & 6 & 1 & 0 & 4 \\ 5 & 3 & 5 & 4 & 0 \end{bmatrix}$$

# Type three:
# Traveling Salesman Problem

$$\begin{cases} c_1(1, \{k\}, k) & = & c_{1k} \\ c_i(1, S, k) & = & \min\limits_{m \in S \setminus \{k\}} [c_{i-1}(1, S \setminus \{k\}, m) + c_{mk}] \end{cases}$$

# Traveling Salesman Problem

$$\begin{bmatrix} \infty & 2 & 6 & 7 & 5 \\ 2 & \infty & 5 & 6 & 3 \\ 6 & 5 & \infty & 1 & 5 \\ 7 & 6 & 1 & \infty & 4 \\ 5 & 3 & 5 & 4 & \infty \end{bmatrix}$$

# Traveling Salesman Problem

$$c_2(1,\{2,3\},3) = \min\{c_1(1,\{2\},2) + c_{23}\} = c_{12} + c_{23} = 2 + 5 = 7$$
$$c_2(1,\{2,3\},2) = \min\{c_1(1,\{3\},3) + c_{32}\} = c_{13} + c_{32} = 6 + 5 = 11$$
$$c_2(1,\{2,4\},4) = \min\{c_1(1,\{2\},2) + c_{24}\} = c_{12} + c_{24} = 2 + 6 = 8$$
$$c_2(1,\{2,4\},2) = \min\{c_1(1,\{4\},4) + c_{42}\} = c_{14} + c_{42} = 7 + 6 = 13$$
$$c_2(1,\{2,5\},5) = \min\{c_1(1,\{2\},2) + c_{25}\} = c_{12} + c_{25} = 2 + 3 = 5$$
$$c_2(1,\{2,5\},2) = \min\{c_1(1,\{5\},5) + c_{52}\} = c_{15} + c_{52} = 5 + 3 = 8$$
$$c_2(1,\{3,4\},4) = \min\{c_1(1,\{3\},3) + c_{34}\} = c_{13} + c_{34} = 6 + 1 = 7$$
$$c_2(1,\{3,4\},3) = \min\{c_1(1,\{4\},4) + c_{43}\} = c_{14} + c_{43} = 7 + 1 = 8$$
$$c_2(1,\{3,5\},5) = \min\{c_1(1,\{3\},3) + c_{35}\} = c_{13} + c_{35} = 6 + 5 = 11$$
$$c_2(1,\{3,5\},3) = \min\{c_1(1,\{5\},5) + c_{53}\} = c_{15} + c_{53} = 5 + 5 = 10$$
$$c_2(1,\{4,5\},5) = \min\{c_1(1,\{4\},4) + c_{45}\} = c_{14} + c_{45} = 7 + 4 = 11$$
$$c_2(1,\{4,5\},4) = \min\{c_1(1,\{5\},5) + c_{54}\} = c_{15} + c_{54} = 5 + 4 = 9$$

# Traveling Salesman Problem

$$c_3(1,\{2,3,4\},4) = \min\{c_2(1,\{2,3\},3) + c_{34}; c_2(1,\{2,3\},2) + c_{24}\} = \min\{7+1; 11+6\} = 8$$
$$c_3(1,\{2,3,4\},3) = \min\{c_2(1,\{2,4\},4) + c_{43}; c_2(1,\{2,4\},2) + c_{23}\} = \min\{8+1; 13+5\} = 9$$
$$c_3(1,\{2,3,4\},2) = \min\{c_2(1,\{3,4\},4) + c_{42}; c_2(1,\{3,4\},3) + c_{32}\} = \min\{7+6; 8+5\} = 13$$
$$c_3(1,\{2,3,5\},5) = \min\{c_2(1,\{2,3\},3) + c_{35}; c_2(1,\{2,3\},2) + c_{25}\} = \min\{7+5; 11+3\} = 12$$
$$c_3(1,\{2,3,5\},3) = \min\{c_2(1,\{2,5\},5) + c_{53}; c_2(1,\{2,5\},2) + c_{23}\} = \min\{5+5; 8+5\} = 10$$
$$c_3(1,\{2,3,5\},2) = \min\{c_2(1,\{3,5\},5) + c_{52}; c_2(1,\{3,5\},3) + c_{32}\} = \min\{11+3; 10+5\} = 14$$
$$c_3(1,\{2,4,5\},5) = \min\{c_2(1,\{2,4\},4) + c_{45}; c_2(1,\{2,4\},2) + c_{25}\} = \min\{8+4; 13+3\} = 12$$
$$c_3(1,\{2,4,5\},4) = \min\{c_2(1,\{2,5\},5) + c_{54}; c_2(1,\{2,5\},2) + c_{24}\} = \min\{5+4; 8+6\} = 9$$
$$c_3(1,\{2,4,5\},2) = \min\{c_2(1,\{4,5\},5) + c_{52}; c_2(1,\{4,5\},4) + c_{42}\} = \min\{11+3; 9+6\} = 14$$
$$c_3(1,\{3,4,5\},5) = \min\{c_2(1,\{3,4\},4) + c_{45}; c_2(1,\{3,4\},3) + c_{35}\} = \min\{7+4; 8+5\} = 11$$
$$c_3(1,\{3,4,5\},4) = \min\{c_2(1,\{3,5\},5) + c_{54}; c_2(1,\{3,5\},3) + c_{34}\} = \min\{11+4; 10+1\} = 11$$
$$c_3(1,\{3,4,5\},3) = \min\{c_2(1,\{4,5\},5) + c_{53}; c_2(1,\{4,5\},4) + c_{43}\} = \min\{11+5; 9+1\} = 10$$

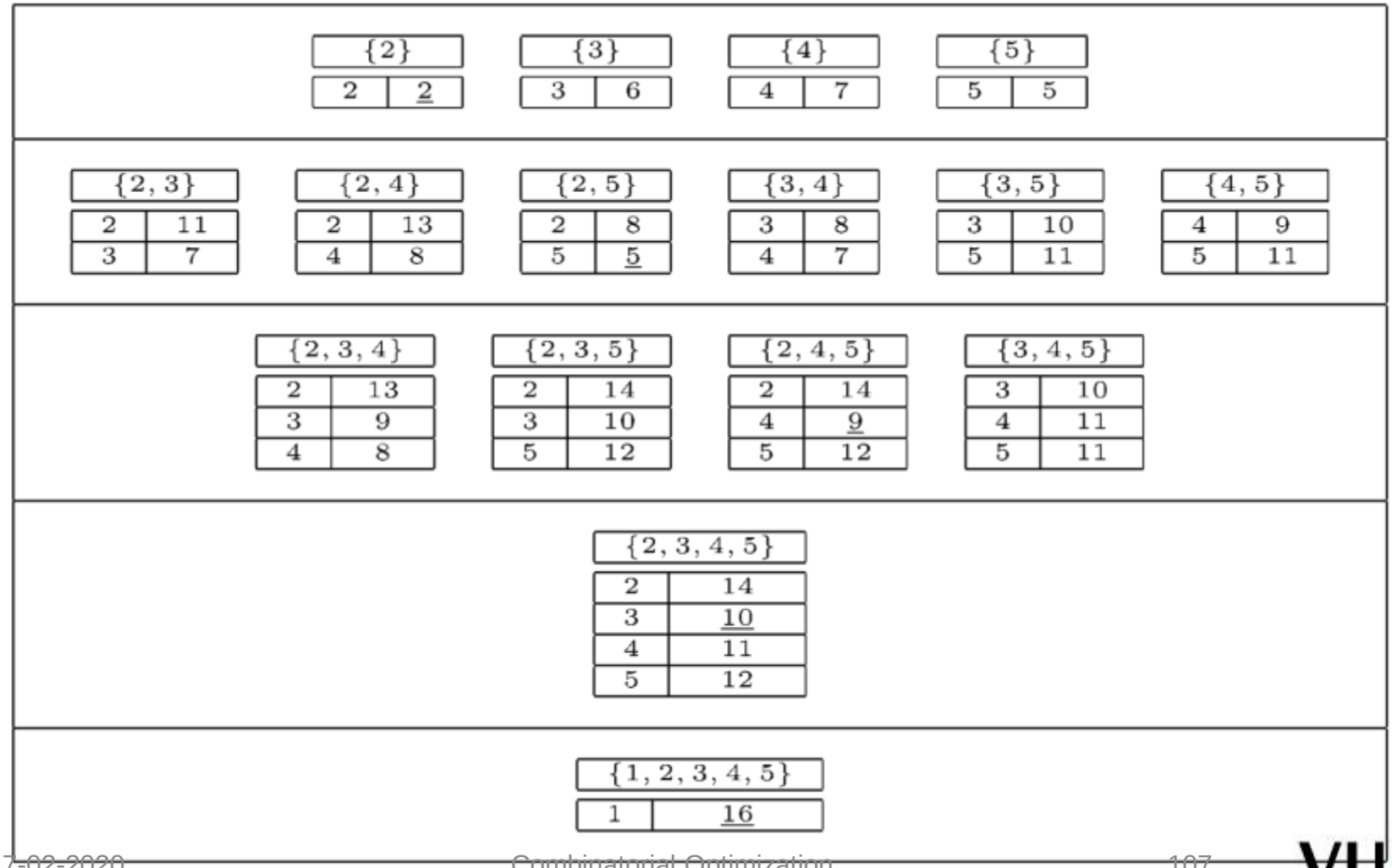
# Traveling Salesman Problem

$$
\begin{aligned}
c_4(1,\{2,3,4,5\},5) &= \min\{c_3(1,\{2,3,4\},4)+c_{45}; c_3(1,\{2,3,4\},3)+c_{35}; c_3(1,\{2,3,4\},2)+c_{25}\} \\
&= \min\{8+4; 9+5; 13+3\} = 12 \\
c_4(1,\{2,3,4,5\},4) &= \min\{c_3(1,\{2,3,5\},5)+c_{54}; c_3(1,\{2,3,5\},3)+c_{34}; c_3(1,\{2,3,5\},2)+c_{24}\} \\
&= \min\{12+4; 10+1; 14+6\} = 11 \\
c_4(1,\{2,3,4,5\},3) &= \min\{c_3(1,\{2,4,5\},5)+c_{53}; c_3(1,\{2,4,5\},4)+c_{43}; c_3(1,\{2,4,5\},2)+c_{23}\} \\
&= \min\{12+5; 9+1; 14+5\} = 10 \\
c_4(1,\{2,3,4,5\},2) &= \min\{c_3(1,\{3,4,5\},5)+c_{52}; c_3(1,\{3,4,5\},4)+c_{42}; c_3(1,\{3,4,5\},3)+c_{32}\} \\
&= \min\{11+3; 11+6; 10+5\} = 15
\end{aligned}
$$

# Traveling Salesman Problem

$$c_5(1,\{1,2,3,4,5\},1) = \min\{c_4(1,\{2,3,4,5\},5) + c_{51}; c_4(1,\{2,3,4,5\},4) + c_{41};$$
$$c_4(1,\{2,3,4,5\},3) + c_{31}; c_4(1,\{2,3,4,5\},2) + c_{21}\}$$
$$= \min\{12+5; 11+7; 10+6; 14+2\} = \mathbf{16}$$

# Traveling Salesman Problem



# The curse of dimensionality

$$\sum_{k=0}^{n} k(k-1) \binom{n}{k} = O(n^2 2^n)$$

Space and time required

# Complexity considerations

TSP is NP-hard
Brute force: $O(n!)$
Lowest known: $O(n^2 2^n)$
Branch & Bound: $O(2^{n^2})$
$n^2 2^n << n! <<< 2^{n^2}$ for all $n > 7$
B&B is the most used framework!
Why?

May you have become interested on DP for NPC problems, consider reading
https://research.vu.nl/ws/portalfiles/portal/42163578
NOTE: this suggestion is extra! Not exam material.