

# **SC2001/ CX2101: Algorithm Design and Analysis**

## **Part 2**

**Huang Shell Ying**

**Office: N4-02b-38**

**Email: [assyhuang@ntu.edu.sg](mailto:assyhuang@ntu.edu.sg)**

# Topics

- Analysis Techniques (3 hours)
- Dynamic Programming (5 hours)
- String Matching (3 hours)
- Introduction to NP Completeness (2 hours)

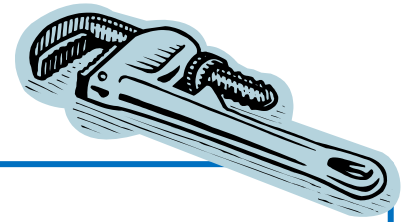
## Lecture Delivery Method

1. Recorded lectures in **Course Media(Media Gallery) /Home**. Videos of one chapter in one PlayList.
2. Weekly review lectures/Q & As in **LT1** on **Mondays 12.30pm – 1.30pm**, Week 8 to Week 13. No lecture on Fridays unless notified otherwise.

# Schedule

<b>Week</b>	<b>Lecture materials to be studied by end of the week</b>	<b>Tutorials</b>	<b>Example classes</b>
<b>7</b>	<b>Analysis techniques (up to slide 38)</b>	<b>Graphs</b>	<b>Project 1</b>
<b>8</b>	<b>Analysis techniques (up to end), DP (up to DP slide 18)</b>	<b>Graphs</b>	<b>Project 2</b>
<b>9</b>	<b>DP (up to DP slide 40)</b>	<b>Analysis techniques</b>	<b>Project 2</b>
<b>10</b>	<b>DP (up to end)</b>	<b>DP</b>	<b>Quiz</b>
<b>11</b>	<b>String matching (up to end)</b>	<b>DP</b>	<b>Quiz</b>
<b>12</b>	<b>NP completeness (up to end)</b>	<b>String matching</b>	<b>Project 3</b>
<b>13</b>		<b>NP completeness</b>	<b>Project 3</b>

Review lectures are from Week 8 to Week 13



# Analysis Techniques

Huang Shell Yíng

**Reference:** Computer Algorithms: Introduction to Design and Analysis, 3<sup>rd</sup> Ed, by Sara Basse and Allen Van Gelder.

# Outline

- Review of the big oh, big omega, big theta
- Solving recurrences (1)
  1. The substitution method
  2. The iteration method
  3. The master method.
- Solving recurrences (2)
  1. Solving linear homogeneous recurrences with constant coefficients

# Review of the big oh, big omega, big theta

## The Big-oh notation:

Definition: Let  $f$  and  $g$  be 2 functions such that

$f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$  and  $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ , if there exists positive constants  $c$  and  $n_0$  such that

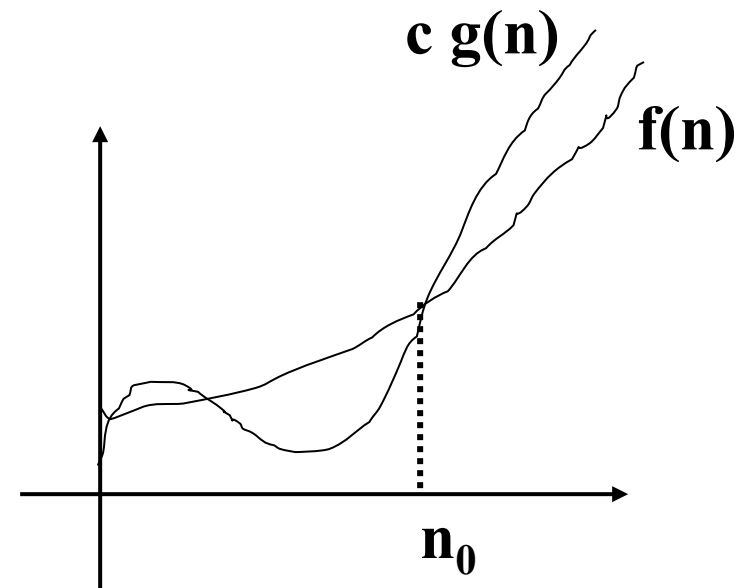
$$f(n) \leq c * g(n) \text{ for all } n > n_0$$

then  $f(n) = O(g(n))$ .

Alternative definition: if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$$

then  $f(n) = O(g(n))$ .



# Review of the big oh, big omega, big theta

Example:  $f(n)=\lg(n)$ ,  $g(n)=n$ ,

Let  $c=1$ ,  $n_0=1$ , then for all  $n>1$

$$\lg(n) \leq n, \text{ i.e., } f(n) \leq g(n)$$

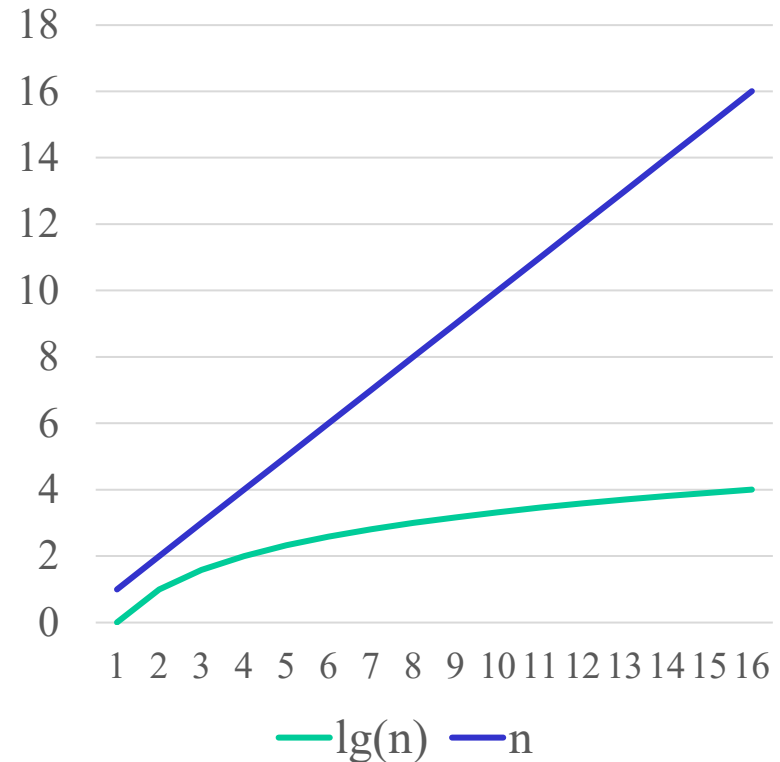
so  $f(n) = O(g(n))$ .

Another way: Since

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\lg(n)}{n} = 0 < \infty$$

so  $f(n) = O(g(n))$ .

$g(n)$  gives the **asymptotic upper bound** for  $f(n)$ .



# Review of the big oh, big omega, big theta

## The big Omega notation

Definition: Let  $f$  and  $g$  be 2 functions such that

$f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$  and  $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ , if there exists positive constants  $c$  and  $n_0$  such that

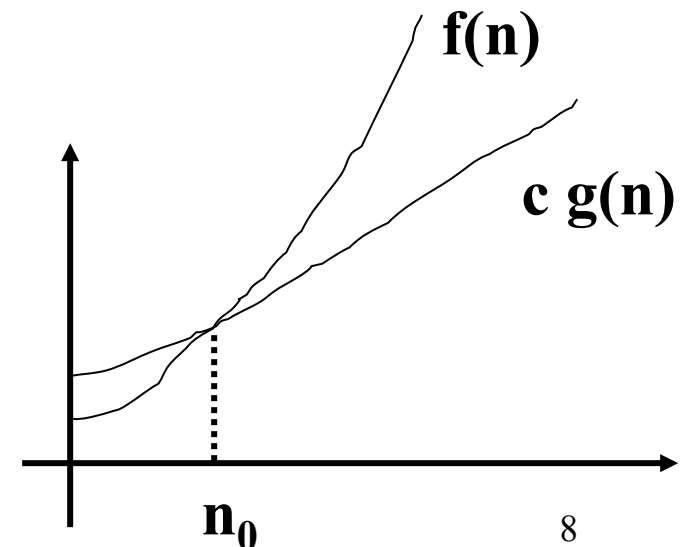
$$f(n) \geq c * g(n) \text{ for all } n > n_0$$

then  $f(n) = \Omega(g(n))$ .

Alternative definition: if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

then  $f(n) = \Omega(g(n))$ .





# Review of the big oh, big omega, big theta

Example:  $f(n) = n^2$ ,  $g(n) = 4n + 3$

Let  $c = 1/4$ ,  $n_0 = 1$ , then for all  $n > 1$

$$n^2 \geq (4n+3)/4 \quad \text{i.e., } f(n) \geq (1/4)g(n)$$

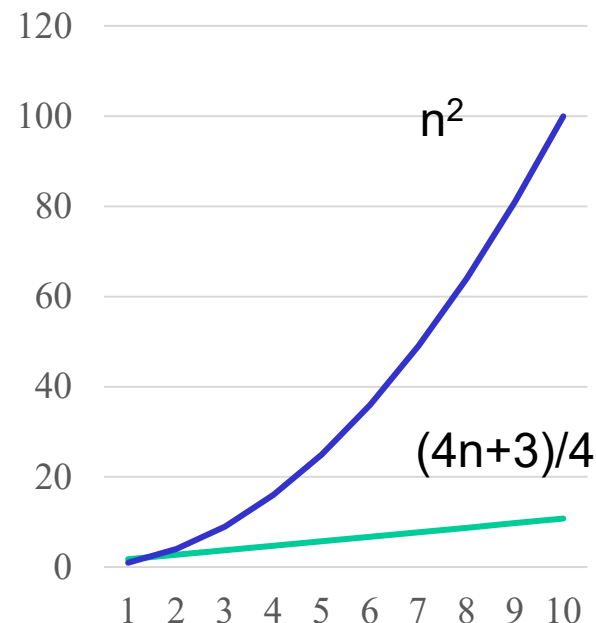
so  $f(n) = \Omega(g(n))$ .

Another way: Since

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n^2}{4n+3} = \lim_{n \rightarrow \infty} \frac{n}{4 + \frac{3}{n}} = \infty > 0$$

so  $f(n) = \Omega(g(n))$ .

$g(n)$  gives the **asymptotic lower bound** for  $f(n)$ .



# Review of the big oh, big omega, big theta

## The big Theta notation

Definition: Let  $f$  and  $g$  be 2 functions such that

$f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$  and  $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ , if there exists positive constants  $c_1$ ,  $c_2$  and  $n_0$  such that

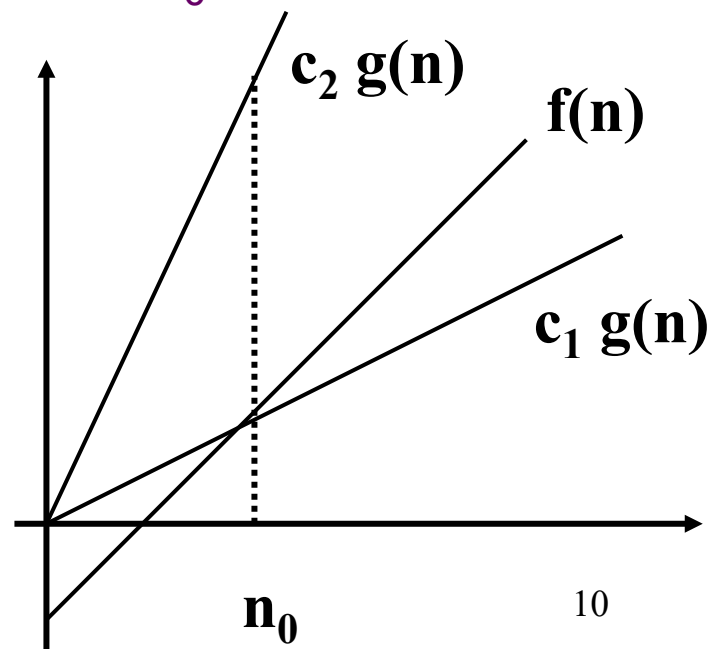
$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n > n_0$$

then  $f(n) = \theta(g(n))$ .

Alternative definition: if

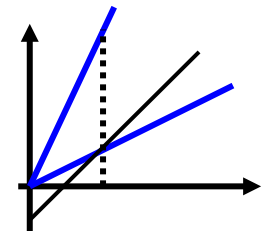
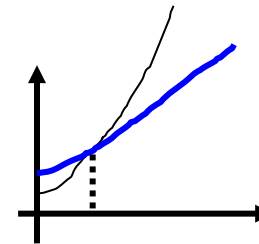
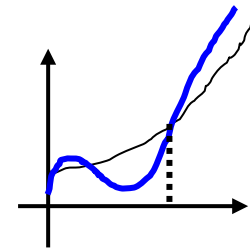
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \quad (0 < c < \infty)$$

then  $f(n) = \theta(g(n))$ .



# Review of the big oh, big omega, big theta

- The idea of the  $O$ ,  $\Omega$  and  $\theta$  definitions is to establish a relative order among functions.
- We compare the relative rates of growth.
- If  $f(n) = O(g(n))$ ,  $g(n)$  gives the asymptotic upper bound
- If  $f(n) = \Omega(g(n))$ ,  $g(n)$  gives the asymptotic lower bound
- If  $f(n) = \theta(g(n))$ ,  $g(n)$  gives the asymptotic tight bound



# Recursive algorithms and Recurrence relations

- Many problems have a recursive solution
- A common way of analysis for such solution algorithms will involve a recurrence relation that needs to be solved
- A recurrence is an equation or inequality that describe a function in terms of its value on smaller inputs, e.g.

$$M(n) = 2M(n-1) + 1$$

## Example 1: Towers of Hanoi

Move all disks from the first pole to the third pole subject to the condition that only one disk can be moved at a time and that no disk is ever placed on top of a smaller one.

```
void TowersOfHanoi(int n, int x, int y, int z)
```

```
{ // Let  $M(n)$  be the total no. of disk moves
```

```
  if (n == 1)
```

```
    cout << "Move disk from " << x << " to " << y << endl;
```

```
    // this has one disk move
```

```
  else {
```

```
    TowersOfHanoi(n-1, x, z, y);
```

```
    // this involves  $M(n-1)$  disk moves
```

```
    cout << "Move disk from " << x << " to " << y << endl;
```

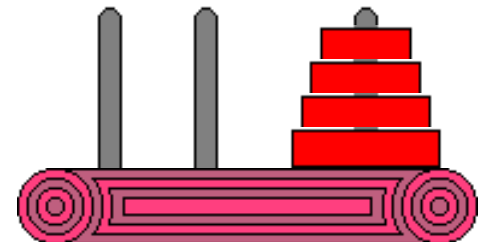
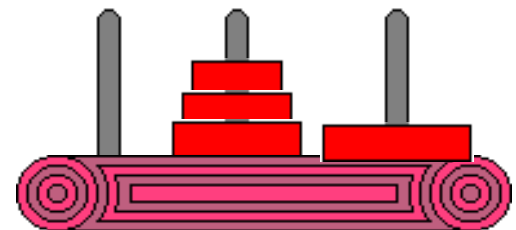
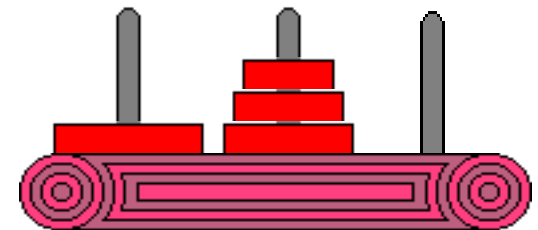
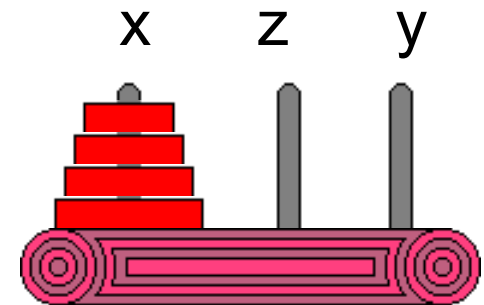
```
    // one disk move
```

```
    TowersOfHanoi(n-1, z, y, x);
```

```
    // another  $M(n-1)$  disk moves
```

```
  }
```

```
}
```



**The number of disk moves:**       $M(1) = 1;$   
    $M(n) = 2M(n-1) + 1$

Example 2: Merge sort

```
void mergesort(int l, int m)
{
    int mid = (l+m)/2;
    if (m-l > 1) {
        mergesort(l, mid);
        mergesort(mid+1, m);
    }
    merge(l, m);
}
```

Let  $M(n)$  be the total no.  
of comparisons between  
array elements.

$M(2) = 1;$   
 $M(n) = 2M(n/2) + n - 1$

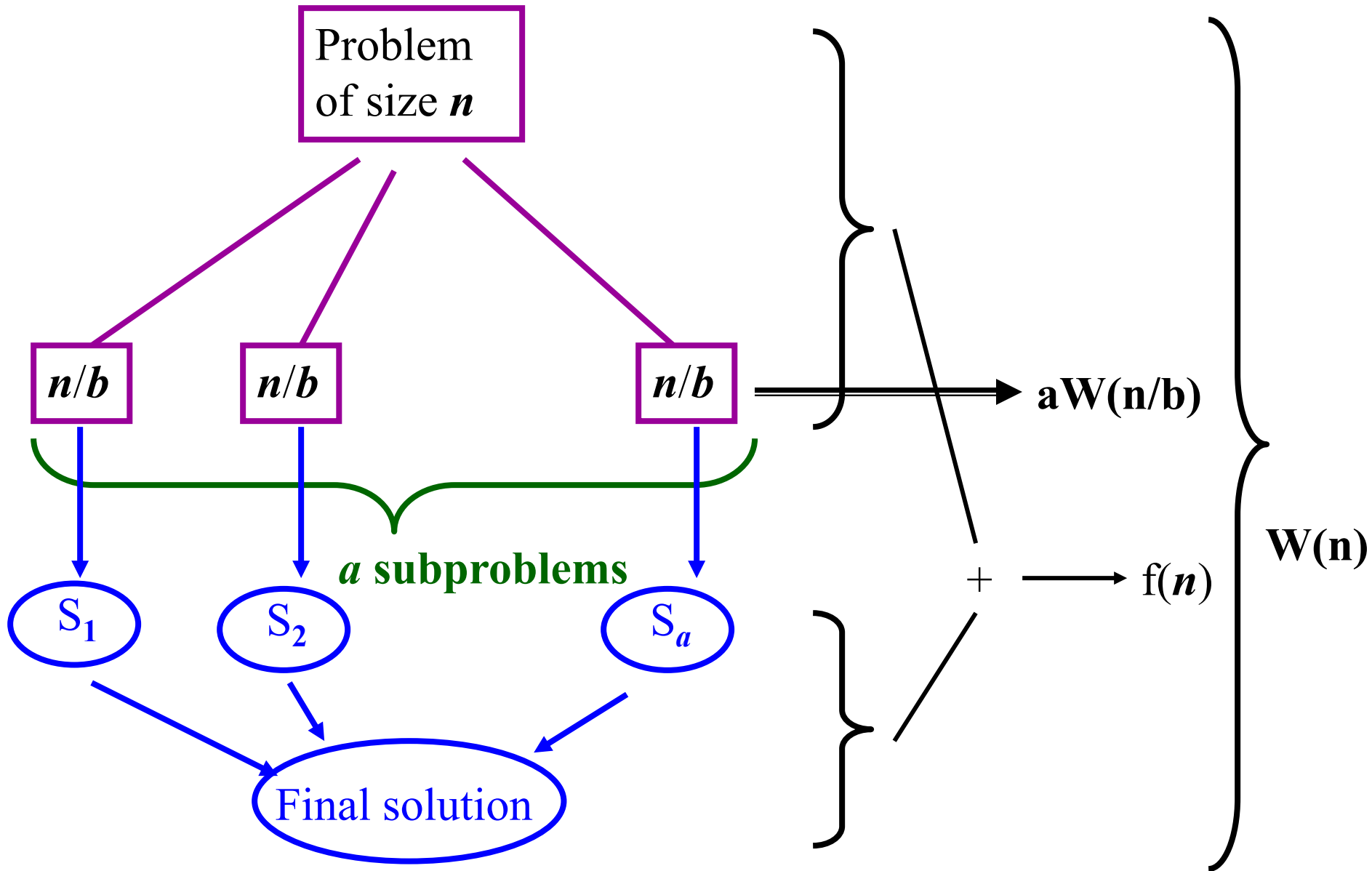
# Solving recurrences (1)

- We want to solve recurrences of the form

$$W(n) = aW(n/b) + f(n)$$

where  $a \geq 1$  and  $b > 1$  are constants,  $f(n)$  is a function of  $n$ .

- The recurrence describes the computational cost of an algorithm that uses the “divide-and-conquer” approach.
- $f(n)$  is the cost of dividing the problem and combining the results of the subproblems.
- Usually the problem of size  $n$  is divided into subproblems of sizes either  $\lceil n/b \rceil$  or  $\lfloor n/b \rfloor$ . However it does not change the asymptotic behaviour of the recurrence.





# Solving recurrences (1)

- Examples

<b><math>W(n) = 2W(n/2) + 2</math></b>	Finding the max and min from a sequence
<b><math>W(n) = W(n/2) + 2</math></b>	Binary search
<b><math>W(n) = 3W(n/2) + cn</math></b>	Multiplying two $2n$ -bits integers
<b><math>W(n) = 2W(n/2) + n - 1</math></b>	Merge sort
<b><math>W(n) = 7W(n/2) + 15n^2/4</math></b>	Multiplying two $n \times n$ matrices

# Solving recurrences (1)

We describe three methods:

- 1) The substitution method
- 2) The iteration method
- 3) The master method.

## 1. The substitution method

- It is a “guess and check” strategy. First guess the form of the solution and then use mathematical induction to prove it.
- A powerful method because often it is easier to prove that a certain bound (in the form of the  $O$  notation) is valid than to compute the bound.

- but the method is only useful when it is easy to guess the form of the solution.
- **Mathematical Induction:** If  $p(a)$  is true and, for some integer  $k \geq a$ ,  $p(k+1)$  is true whenever  $p(k)$  is true, then  $p(n)$  is true for all  $n \geq a$ .
- Example: The worst case for merge sort ( $n = 2^k$ )

$$W(2) = 1$$

$$W(n) = 2 W(n/2) + n - 1$$

Guess  $W(n) = O(f(n))$  then prove it.

Show (i)  $W(2) \leq f(2)$  (ii) for some integer  $k \geq 2$ , assume  $W(n) = O(f(n))$  for  $n \leq 2^k$ , prove  $W(2n) \leq f(2n)$  then  $W(n) = O(f(n))$  for all  $n \geq 2$ .

## First guess: $W(n) = O(n^2)$

Proof by mathematical induction that  $W(n) \leq cn^2$ :

(1) Base case:  $W(2) = 1 \leq 2^2$ ;

(2) Inductive step: assume that  $W(n) = O(n^2)$  for  $n \leq 2^k$ .  
Now consider  $n = 2^{k+1}$

$$\begin{aligned} W(2^{k+1}) &= 2W(2^k) + 2^{k+1} - 1 \\ &\leq 2 * (2^k)^2 + 2^{k+1} - 1 \\ &= 2 * (2^k)^2 + 2 * 2^k - 1 \\ &\leq 4 * (2^k)^2 \\ &= (2^{k+1})^2 \end{aligned}$$

$$\text{i.e. } W(2^{k+1}) \leq (2^{k+1})^2$$

A lot is added  
from step 3 to  
step 4

Thus  $W(n) = O(n^2)$ . But is this the best guess?

**Second guess:**  $W(n) = O(n)$ , i.e.  $W(n) \leq c * n$

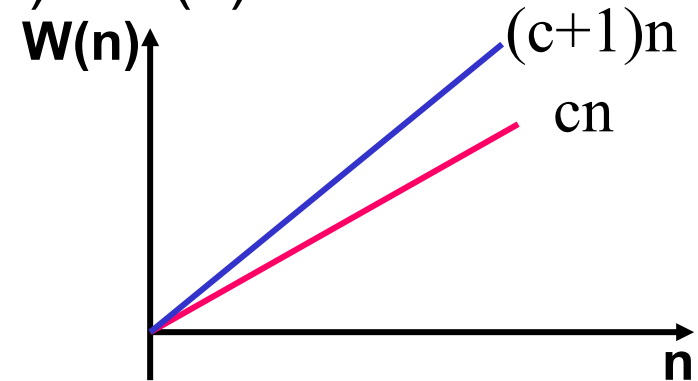
Proof by mathematical induction:

(1) Base case:  $W(2) = 1 \leq 2c$ ;

(2) Inductive step: assume that  $W(n) = O(n)$  for  $n \leq 2^k$ .

Now consider  $n = 2^{k+1}$

$$\begin{aligned} W(2^{k+1}) &= 2W(2^k) + 2^{k+1} - 1 \\ &\leq 2 * c * 2^k + 2^{k+1} - 1 \\ &= c * 2^{k+1} + 2^{k+1} - 1 \end{aligned}$$



Thus  $W(2^{k+1}) \leq (c+1) * 2^{k+1} - 1$  but we cannot say  
 $W(2^{k+1}) \leq c * 2^{k+1}$

Thus  $W(n) \neq O(n)$ .

### Third guess: $W(n) = O(n \lg n)$

Proof by mathematical induction:

- (1) Base case:  $W(2) = 1 \leq 2 \lg 2$ ;
- (2) Inductive step: assume that  $W(n) \leq n \lg n$  for  $n \leq 2^k$ .  
Now consider  $n = 2^{k+1}$

$$\begin{aligned} W(2^{k+1}) &= 2W(2^k) + 2^{k+1} - 1 \\ &\leq 2 * k * 2^k + 2^{k+1} - 1 \\ &= k * 2^{k+1} + 2^{k+1} - 1 \\ &\leq (k + 1) * 2^{k+1} \end{aligned}$$

Thus  $W(n) = O(n \lg n)$  is a very close upper bound.

# What if the base condition does not hold?

Consider the recurrence ( $n = 2^k$ ) :

$$W(1) = 1$$

$$W(n) = 2 W(n/2) + n - 1$$

Prove that  $W(n) = O(n \lg n)$ :

- (1) Base case:  $W(1) = 1 > c \lg 1$ ;
  - (2) Recall the big-O notation: for  $f(n) = O(g(n))$ , we need  $f(n) \leq c * g(n)$  for all  $n > n_0$ .
  - (3) Thus to prove  $W(n) = O(n \lg n)$  , we may use another base case.
    - We have  $W(2) = 3 < c * 2 * \lg 2$  for any  $c > 1$ .
    - We can assume that  $W(n) \leq c n \lg n$  for  $n \leq 2^k$  then prove  $W(2^{k+1}) \leq c * (k + 1) * 2^{k+1}$
- Then  $W(n) = O(n \lg n)$ .

## What can we say about the general case of $n$ ?

The worst case for merge sort :

$$W(2) = 1$$

$$W(n) = W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor) + n - 1$$

Proof <sup>+</sup>:

- (1)  $W(n)$  is a monotonically increasing function. So when  $n$  is not a power of 2, that is,  $2^k < n < 2^{k+1}$ , then  $W(2^k) \leq W(n) \leq W(2^{k+1})$ .
- (2) We have proved that  $W(n) = O(n \lg n)$  for powers of 2, so,  $W(2^{k+1}) \leq c * (k+1) * 2^{k+1}$ .
- (3) For any  $n < 2^{k+1}$  for some  $k$ ,  $W(n) \leq W(2^{k+1})$ . Therefore  $W(n) \leq c * (k+1) * 2^{k+1} < c * \lg(2n) * (2 * n) < 4cn \lg n$

Therefore  $W(n) = O(n \lg n)$ .

<sup>+</sup> See *The design and analysis of Algorithms* by **Anany Levitin** (pp481-483) about **Smoothness Rule**.  
Analysis Techniques SC2001/CX2101 24



## 2. The iteration method

- The idea is to expand (iterate) the recurrence and express it as a summation of terms depending only on  $n$  and the initial condition.
- Techniques for evaluating summations can then be used to provide bounds on the solution.
- Example:

$$W(1) = 1, W(2) = 1, W(3) = 1,$$

$$W(n) = 3W\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n$$

we expand (iterate) it:

$$\begin{aligned} W(n) &= 3W\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n \\ &= 3\left(3W\left(\left\lfloor \frac{n}{4^2} \right\rfloor\right) + \left\lfloor \frac{n}{4} \right\rfloor\right) + n \end{aligned}$$

$$= 3^2 W(\lfloor \frac{n}{4^2} \rfloor) + 3 \lfloor \frac{n}{4} \rfloor + n$$

$$= 3^2 (3W(\lfloor \frac{n}{4^3} \rfloor) + \lfloor \frac{n}{4^2} \rfloor) + 3 \lfloor \frac{n}{4} \rfloor + n$$

$$= 3^3 W(\lfloor \frac{n}{4^3} \rfloor) + 3^2 \lfloor \frac{n}{4^2} \rfloor + 3 \lfloor \frac{n}{4} \rfloor + n$$

we need to iterate until we reach one of the boundary conditions, i.e  $\lfloor \frac{n}{4^i} \rfloor = 1, 2$  or  $3$ .

E.g.  $n=64$ ,  $4^3 \leq 64 < 4^4$  and  $\lfloor \frac{64}{4^3} \rfloor = 1$ ;

$n=255$ ,  $4^3 \leq 255 < 4^4$  and  $\lfloor \frac{255}{4^3} \rfloor = 3$ ;

This means if  $4^i \leq n < 4^{i+1}$  then  $i = \lfloor \log_4 n \rfloor$ . So

$$W(n) = 3^i W(a) + 3^{i-1} \lfloor \frac{n}{4^{i-1}} \rfloor + \dots + 3^2 \lfloor \frac{n}{4^2} \rfloor + 3 \lfloor \frac{n}{4} \rfloor + n$$

$a = 1, 2$  or  $3$

$$W(n) = 3^i W(a) + 3^{i-1} \lfloor \frac{n}{4^{i-1}} \rfloor + \dots + 3^2 \lfloor \frac{n}{4^2} \rfloor + 3 \lfloor \frac{n}{4} \rfloor + n$$

$$\leq 3^{\log_4 n} W(a) + 3^{i-1} \frac{n}{4^{i-1}} + \dots + 3^2 \frac{n}{4^2} + 3 \frac{n}{4} + n$$

Let  $x = 3^{\log_4 n}$  then

$\log_4 x = \log_4 n \log_4 3$  then

$4^{\log_4 x} = 4^{\log_4 n \log_4 3}$  then

$x = n^{\log_4 3}$ , i.e.  $3^{\log_4 n} = n^{\log_4 3}$

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = 4$$

$$W(n) \leq n^{\log_4 3} + n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = O(n)$$

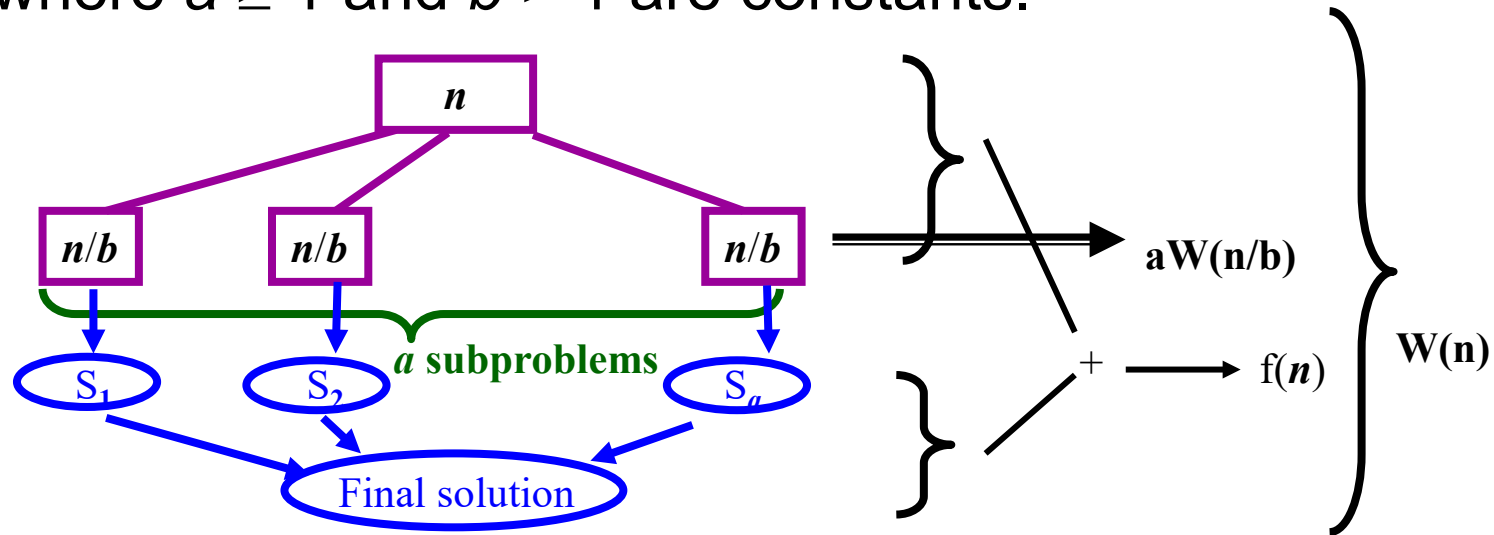
- The iteration method usually leads to lots of algebra.
- We should focus on how many times the recurrence needs to be iterated to reach the boundary condition.

### 3. The master method

- The master method provides a “manual” for solving recurrences of the form

$$W(n) = aW(n/b) + f(n)$$

where  $a \geq 1$  and  $b > 1$  are constants.



- We are able to determine the asymptotic tight bound in the following three cases

## The master theorem

For  $W(n) = aW(n/b) + f(n)$   $a \geq 1$  and  $b > 1$

The manual:

1. If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$ , then  $W(n) = \theta(n^{\log_b a})$ .

2. If  $f(n) = \theta(n^{\log_b a})$ , then  $W(n) = \theta(n^{\log_b a} \log n)$ .

If  $f(n) = \theta(n^{\log_b a} \log^k n)$ ,  $k \geq 0$ ,  
then  $W(n) = \theta(n^{\log_b a} \log^{k+1} n)$

3. If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$ , and if  $a f(n/b) \leq c f(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $W(n) = \theta(f(n))$ .

What is  $n^{\log_b a}$ ?

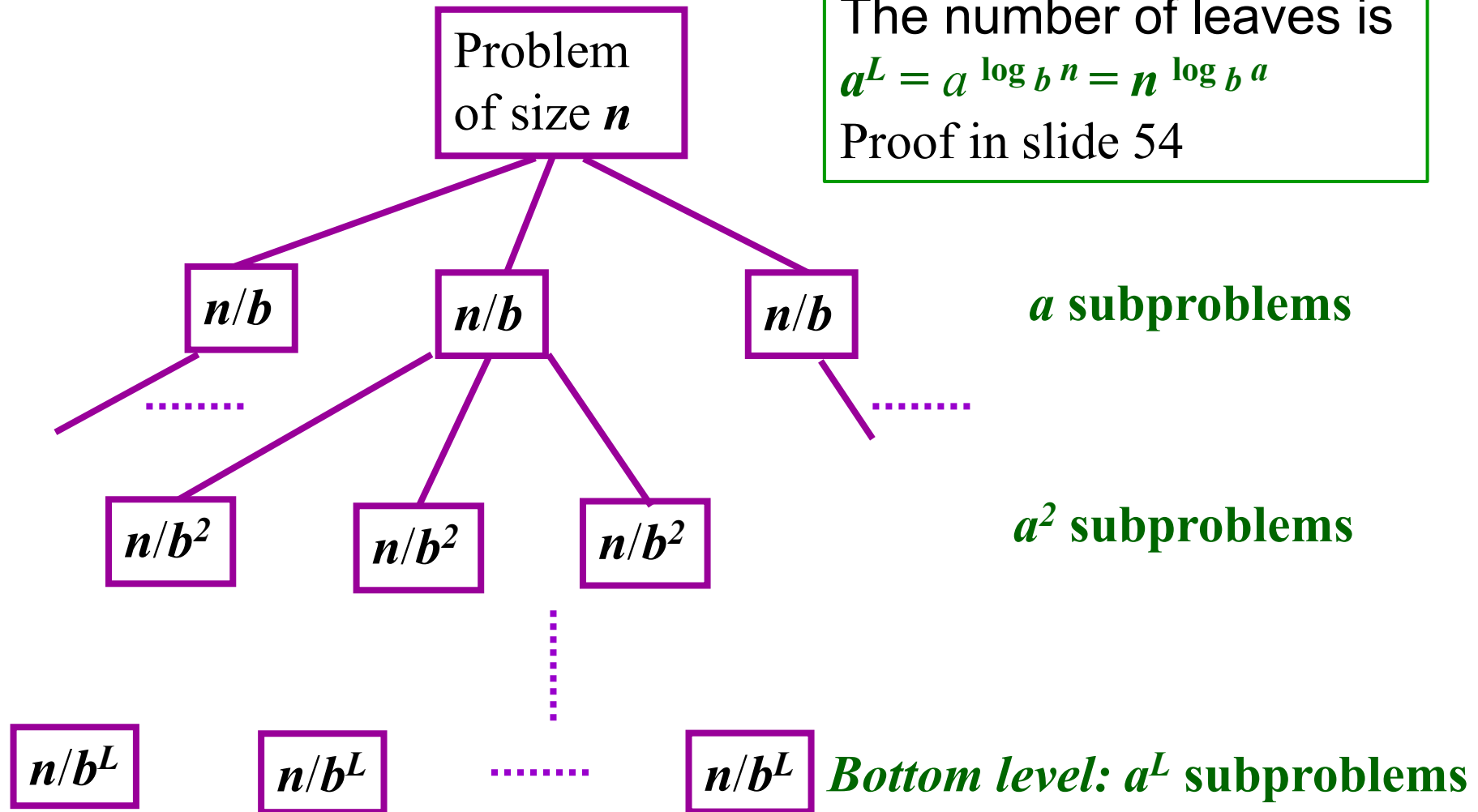
The depth of the tree

$$L = \log_b n$$

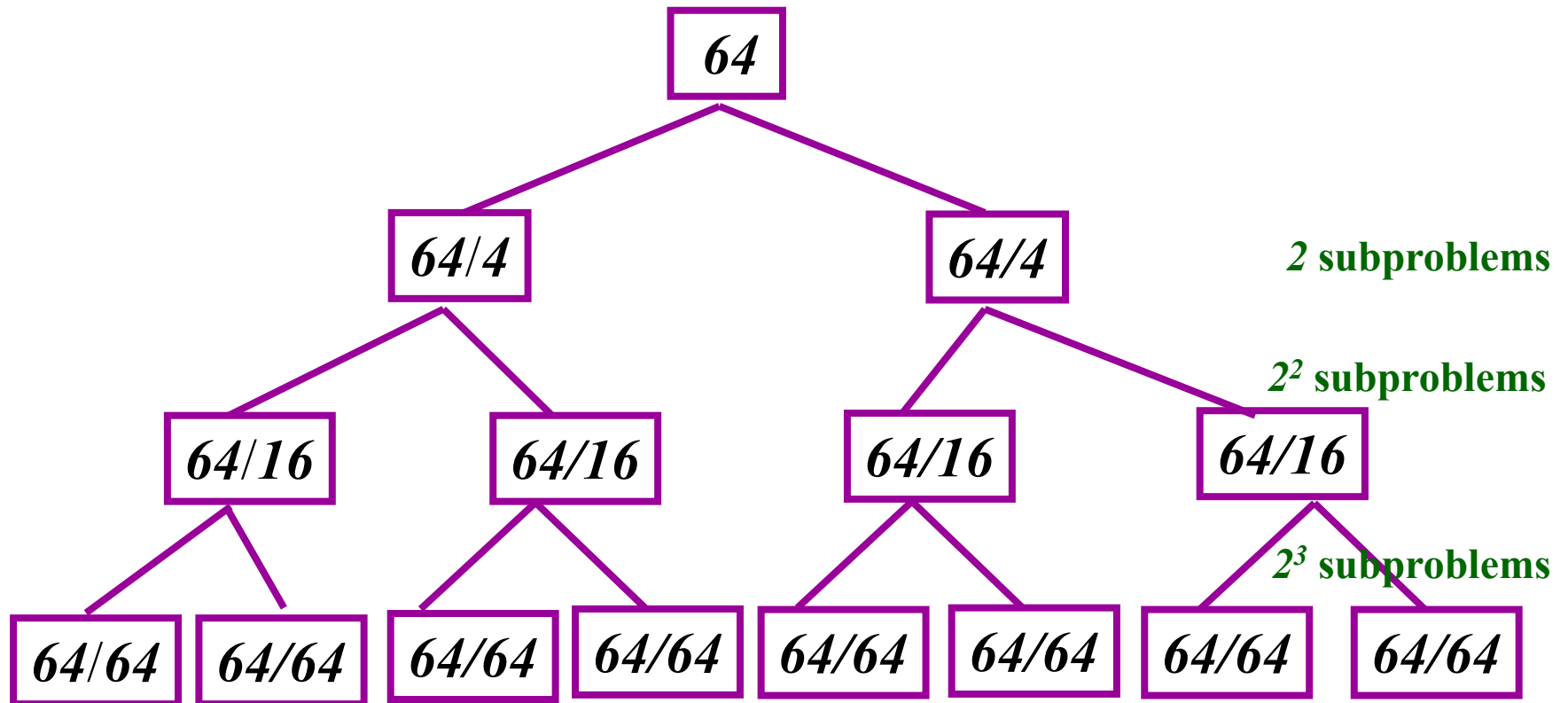
The number of leaves is

$$a^L = a^{\log_b n} = n^{\log_b a}$$

Proof in slide 54



E.g.  $n = 64$ ,  $a = 2$ ,  $b = 4$



Depth of tree  $L = \log_4 64$ , Number of leaves  $= 8 = 2^{\log_4 64} = 64^{\log_4 2}$   
( $a^{\log_b n} = n^{\log_b a}$ )

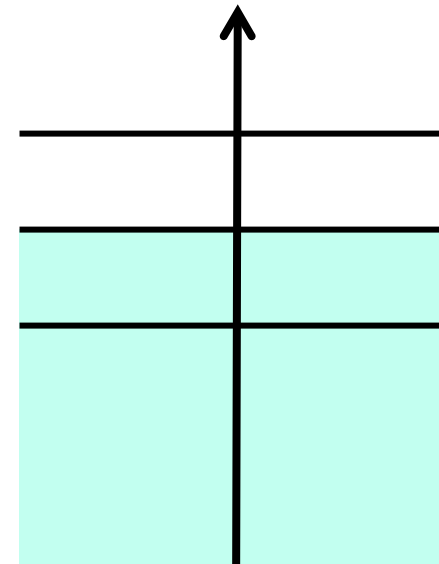
## Examples

1)  $W(n) = 3W(n/3) + 2$ ,  
so  $a = 3$ ,  $b = 3$ ,  
 $n^{\log_b a} = n^1$   
 $f(n) = 2 = \theta(1) = O(n^1)$

$$n^{\log_b a} = n$$

$$n^{1-\varepsilon}$$

$$1$$



Complexity

We may let  $\varepsilon = 0.5$  then we confirm  $2 = O(n^{1-0.5})$ ,

i.e.  $f(n) = O(n^{1-\varepsilon})$

$\Rightarrow f(n) = O(n^{\log_b a - \varepsilon})$  (case 1)

thus  $W(n) = \theta(n^{\log_b a})$

$W(n) = \theta(n)$ .



## Examples

2)  $W(n) = 4W(n/4) + n - 1,$

so  $a = 4, b = 4,$

$$n^{\log_b a} = n^1$$

$$f(n) = n - 1$$

We have

$$f(n) = n - 1$$

$$= \theta(n^1),$$

$$= \theta(n^{\log_b a}),$$

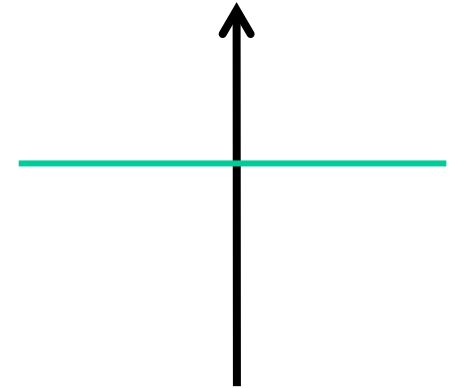
(case 2)

thus

$$W(n) = \theta(n^{\log_b a} \log n)$$

$$= \theta(n \log n)$$

$$n^{\log_b a} = n$$



Complexity

## Examples

3)  $W(n) = 2W(n/2) + n \lg n$ ,

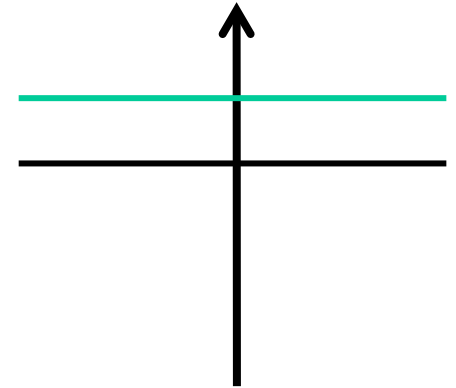
so  $a = 2$ ,  $b = 2$ ,

$$f(n) = n \lg n$$

$$n^{\log_b a} = n^1$$

$$n^{\log_b a} \lg^k n$$

$$n^{\log_b a} = n$$



Complexity

We have

$$f(n) = \theta(n^1 \lg n),$$

$$= \theta(n^{\log_b a} \lg^k n), \quad (\text{case 2: } k = 1)$$

thus

$$W(n) = \theta(n^{\log_b a} \lg^2 n)$$

$$= \theta(n (\lg n)^2)$$

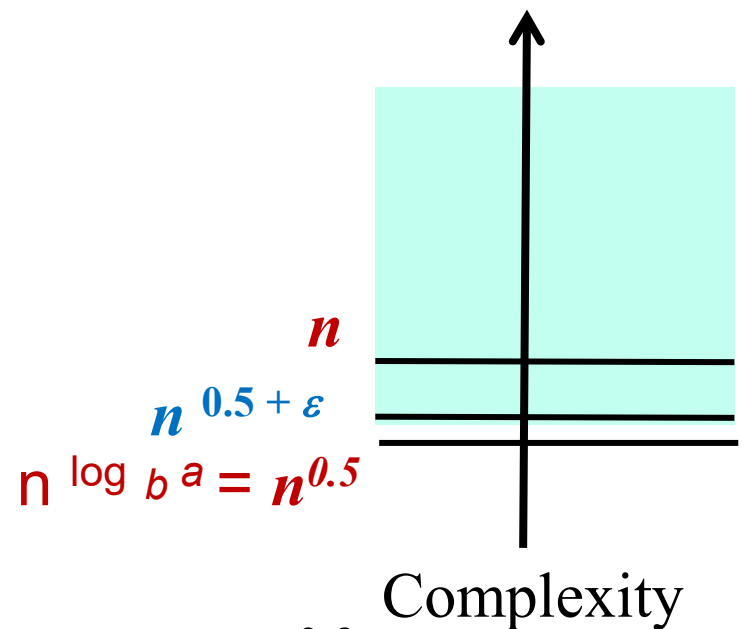
## Examples

4)  $W(n) = 2W(n/4) + n,$

so  $a = 2, b = 4,$

$$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$$

$$f(n) = n = \theta(n)$$



We may let  $\epsilon = 0.1$  then we have  $n = \Omega(n^{0.6})$

i.e.  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , and

for all sufficiently large  $n$ , we can find a value for  $c$ , say,  $c = 3/4$ , to show that  $a f(n/b) \leq c f(n)$ . (case 3)

$$a \cdot f(n/b) = 2 \cdot f(n/4) = n/2 \leq c \cdot n$$

thus  $W(n) = \theta(n)$ .

# Sometimes the master method cannot apply

Example 1:  $W(n) = 3W(n/3) + n/\lg n$ ,  $n^{\log b^a} = n^1$

$f(n) = n/\lg n = O(n^1)$  because  $\lim_{n \rightarrow \infty} \frac{n/\lg n}{n^1} = \lim_{n \rightarrow \infty} \frac{1}{\lg n} = 0$

$f(n) = O(n^{1-\varepsilon})$  ? (L'Hôpital's rule, slide 55)

i.e.  $n/\lg n = O(n^{1-\varepsilon})$  ?

No, because asymptotically,  $n/\lg n > n^{1-\varepsilon}$  for any  $\varepsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n/\lg n}{n^{1-\varepsilon}} = \lim_{n \rightarrow \infty} \frac{n^\varepsilon}{\lg n} = \infty$$

This recurrence falls into the gap between case 2 and case 3.

So the Master Theorem cannot apply.

# Sometimes the master method cannot apply

Example 2:  $W(n) = W(n/3) + f(n)$

$$\text{where } f(n) = \begin{cases} 3n + 2^{3n} & \text{for } n = 2^i \\ 3n & \text{otherwise} \end{cases}$$

so  $a = 1$ ,  $b = 3$  then  $n^{\log_b a} = n^0$

let  $\varepsilon = 1$  then  $f(n) = \Omega(n^{0+1})$ , case 3?

$a f(n/b) \leq c f(n)$  for all sufficiently large  $n$ ?

When  $n = 3 * 2^i$ ,  $a f(n/b) = f(2^i) = n + 2^n$ , but  $cf(n) = c(3n)$

i.e.  $a f(n/b) > c f(n)$ . E.g. for  $n = 6$  or greater

So the Master Theorem cannot apply.

- Notice that when we want to find the order of a recurrence, the initial conditions are not important. This is because the running costs of the terminating conditions are small constants that do not affect the order.

## Solving recurrences (2)

- Definition: A linear homogeneous recurrence relation of degree  $k$  with constant coefficients is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k},$$

where  $c_1, c_2, \dots, c_k$  are real constants and  $c_k \neq 0$ .

The two different notations:  $A(n)$  and  $a_n$

- When using  $A(n)$ , we mean the function value with parameter  $n$
- When using  $a_n$ , we mean the  $n$ th term in a sequence  $a_1, a_2, \dots, a_n$ .
- If we list  $A(1), A(2), \dots, A(n)$  in a sequence, we can write them as  $a_1, a_2, \dots, a_n$ . They are equivalent.

## Solving recurrences (2)

- Definition: A linear homogeneous recurrence relation of degree  $k$  with constant coefficients is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k},$$

where  $c_1, c_2, \dots, c_k$  are real constants and  $c_k \neq 0$ .

- **Linear**:  $a_{n-1}, a_{n-2}, \dots, a_{n-k}$  appear in separate terms and to the first power
- **Homogeneous**: the total degree of each term is the same, e.g. no constant term
- **Constant coefficients**:  $c_1, c_2, \dots, c_k$  are fixed real constants that do not depend on  $n$
- **Degree  $k$** : the expression for  $a_n$  contains the previous  $k$  terms  $a_{n-1}, a_{n-2}, \dots, a_{n-k}$ , ( $c_k \neq 0$ )



- Examples
  - *A linear homogeneous recurrence relation of degree 2:  $a_n = a_{n-1} + a_{n-2}$*
  - *A linear homogeneous recurrence relation of degree 1:  $a_n = 1.04a_{n-1}$*
  - *A linear homogeneous recurrence relation of degree 3 :  $a_n = a_{n-3}$*
- Non-examples
  - $a_n = a_{n-1} + a_{n-2} + 1$ : non-homogeneous
  - $a_n = a_{n-1}a_{n-2}$  : not linear
  - $a_n = na_{n-1}$  : coefficient not constant

- A linear homogeneous recurrence relation of degree  $k$  can be systematically solved, i.e. find the explicit expression for  $a_n$
- The basic approach is to look for solutions of the form  $a_n = t^n$  where  $t$  is a constant
- If  $a_n = t^n$  is a solution for

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

Then

$$t^n = c_1 t^{n-1} + c_2 t^{n-2} + \dots + c_k t^{n-k}$$

$$\Rightarrow t^k = c_1 t^{k-1} + c_2 t^{k-2} + \dots + c_k \quad (\text{divide both side by } t^{n-k})$$

$$\Rightarrow t^k - c_1 t^{k-1} - c_2 t^{k-2} - \dots - c_k = 0$$

- This means if we can solve the equation

$$t^k - c_1 t^{k-1} - c_2 t^{k-2} - \dots - c_k = 0$$

to find  $t$ , then  $a_n = t^n$  is a solution for

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

We call

$$t^k - c_1 t^{k-1} - c_2 t^{k-2} - \dots - c_k = 0$$

the **characteristic equation** of

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

The solutions to the characteristic equation are called the **characteristic roots**

- We consider a linear homogeneous recurrence relation of degree 2

$$a_n = Aa_{n-1} + Ba_{n-2} \text{ for all } n \geq 2$$

where A and B are real constants

- The **characteristic equation**

$$t^2 - At - B = 0$$

may have

- 1) two distinct roots
- 2) a single root

- **Theorem 1 (Distinct Roots Theorem)**

Suppose a sequence  $a_0, a_1, a_2, \dots$  satisfies a recurrence relation

$$a_n = Aa_{n-1} + Ba_{n-2} \text{ for all } n \geq 2$$

where  $A$  and  $B$  are real constants and  $B \neq 0$ . If the characteristic equation

$$t^2 - At - B = 0$$

has two distinct roots  $r$  and  $s$ , then  $a_0, a_1, a_2, \dots$  is given by the explicit formula

$$a_n = Cr^n + Ds^n$$

where  $C$  and  $D$  are determined by the values of  $a_0$  and  $a_1$ .

- Example 1:

$$F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 2, \quad \text{and } F_0 = F_1 = 1$$

The characteristic equation is

$$t^2 - t - 1 = 0$$

The roots are

$$t = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \begin{cases} \frac{1+\sqrt{5}}{2} \\ \frac{1-\sqrt{5}}{2} \end{cases}$$

$$F_n = C \left( \frac{1 + \sqrt{5}}{2} \right)^n + D \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

For  $ax^2 + bx + c = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- To find C and D, we have

$$F_0 = 1 = C \left( \frac{1 + \sqrt{5}}{2} \right)^0 + D \left( \frac{1 - \sqrt{5}}{2} \right)^0 = C \cdot 1 + D \cdot 1 = C + D$$

$$F_1 = 1 = C \left( \frac{1 + \sqrt{5}}{2} \right)^1 + D \left( \frac{1 - \sqrt{5}}{2} \right)^1 = C \left( \frac{1 + \sqrt{5}}{2} \right) + D \left( \frac{1 - \sqrt{5}}{2} \right)$$

To solve this system of 2 equations with 2 unknowns, from

$$C + D = 1$$

$$\Rightarrow \left( \frac{1 + \sqrt{5}}{2} \right) C + \left( \frac{1 - \sqrt{5}}{2} \right) D = \left( \frac{1 + \sqrt{5}}{2} \right)$$

Then

$$D\left(\left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)\right) = \left(\frac{1+\sqrt{5}}{2}\right) - 1$$

$$\Rightarrow D\sqrt{5} = \left(\frac{1+\sqrt{5}}{2}\right) - 1$$

$$\Rightarrow D = \left(\frac{-1+\sqrt{5}}{2\sqrt{5}}\right)$$

$$\text{Then } C = 1 - D = 1 - \left(\frac{-1+\sqrt{5}}{2\sqrt{5}}\right)$$

$$\Rightarrow C = \frac{1+\sqrt{5}}{2\sqrt{5}}$$

We can write

$$D = \left(\frac{-(1-\sqrt{5})}{2\sqrt{5}}\right)$$



- So

$$F_n = \left( \frac{1 + \sqrt{5}}{2\sqrt{5}} \right) \left( \frac{1 + \sqrt{5}}{2} \right)^n + \left( \frac{-(1 - \sqrt{5})}{2\sqrt{5}} \right) \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

After simplifying it, we get

$$F_n = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{n+1} - \frac{1}{\sqrt{5}} \left( \frac{1 - \sqrt{5}}{2} \right)^{n+1}$$

for all  $n \geq 0$ .

- Example 2:

$$a_n = 5a_{n-1} - 6a_{n-2}, a_0 = 9, a_1 = 20$$

The characteristic equation is

$$t^2 - 5t + 6 = 0$$

$$\Rightarrow (t - 2)(t - 3) = 0 \Rightarrow \text{two roots: } t = 2, t = 3$$

$$a_n = C2^n + D3^n \quad \text{for all } n \geq 0.$$

To find  $C$  and  $D$ :

$$9 = C + D, \Rightarrow 18 = 2C + 2D$$

$$20 = 2C + 3D$$

$$\text{Thus } D = 2, C = 7 \quad \text{So } a_n = 7 \cdot 2^n + 2 \cdot 3^n \quad \text{for all } n \geq 0$$

- **Theorem 2 (Single-Root Theorem)**

Suppose a sequence  $a_0, a_1, a_2, \dots$  satisfies a recurrence relation

$$a_n = Aa_{n-1} + Ba_{n-2} \text{ for all } n \geq 2$$

where  $A$  and  $B$  are real constants and  $B \neq 0$ . If the characteristic equation

$$t^2 - At - B = 0$$

has a single (real) root, then  $a_0, a_1, a_2, \dots$  is given by the explicit formula

$$a_n = Cr^n + Dnr^n$$

where  $C$  and  $D$  are determined by the values of  $a_0$  and any other known value of the sequence.

- Example

$$b_n = 4b_{n-1} - 4b_{n-2} \text{ for all } n \geq 2$$

with  $b_0 = 1$ ,  $b_1 = 3$ .

The characteristic equation is

$$t^2 - 4t + 4 = 0$$

$$\Rightarrow (t - 2)^2 = 0 \quad \Rightarrow \text{single root } t = 2$$

The explicit formula is

$$b_n = C2^n + Dn2^n$$

where  $C$  and  $D$  are determined by the values of  $b_0$  and  $b_1$ .

We have  $1 = C$  and  $3 = 2C + 2D$ , so  $D = \frac{1}{2}$  and  $C = 1$ .

- Therefore

$$b_n = 2^n + (1/2)n2^n = (1 + n/2) 2^n$$

Theorem 1 can be generalised to the recurrence relation

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

with characteristic equation

$$t^k - c_1 t^{k-1} - c_2 t^{k-2} - \dots - c_k = 0$$

having  $k$  distinct roots.

Theorem 2 can be generalised to less than  $k$  distinct roots.

# Proof of $a^{\log_b n} = n^{\log_b a}$

- Let  $L = \log_b n$ , i.e.  $b^L = n$ 
  - $\Rightarrow (b^L)^{\log_b a} = n^{\log_b a}$
  - $\Rightarrow (b^{\log_b a})^L = n^{\log_b a}$
  - $\Rightarrow a^L = n^{\log_b a}$
  - $\Rightarrow a^{\log_b n} = n^{\log_b a}$

# L'Hôpital's rule

L'Hôpital's rule states that for functions  $f(x)$  and  $g(x)$ , if:

$$\lim_{n \rightarrow \infty} f(x) = \lim_{n \rightarrow \infty} g(x) = \pm\infty$$

then:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

where the prime (') denotes the derivative.

# Dynamic Programming

Huang Shell Ying

**Reference:** Computer Algorithms: Introduction to Design and Analysis, 3<sup>rd</sup> Ed, by Sara Basse and Allen Van Gelder. Sections *10.1, 10.2 & 10.3*



# Outline

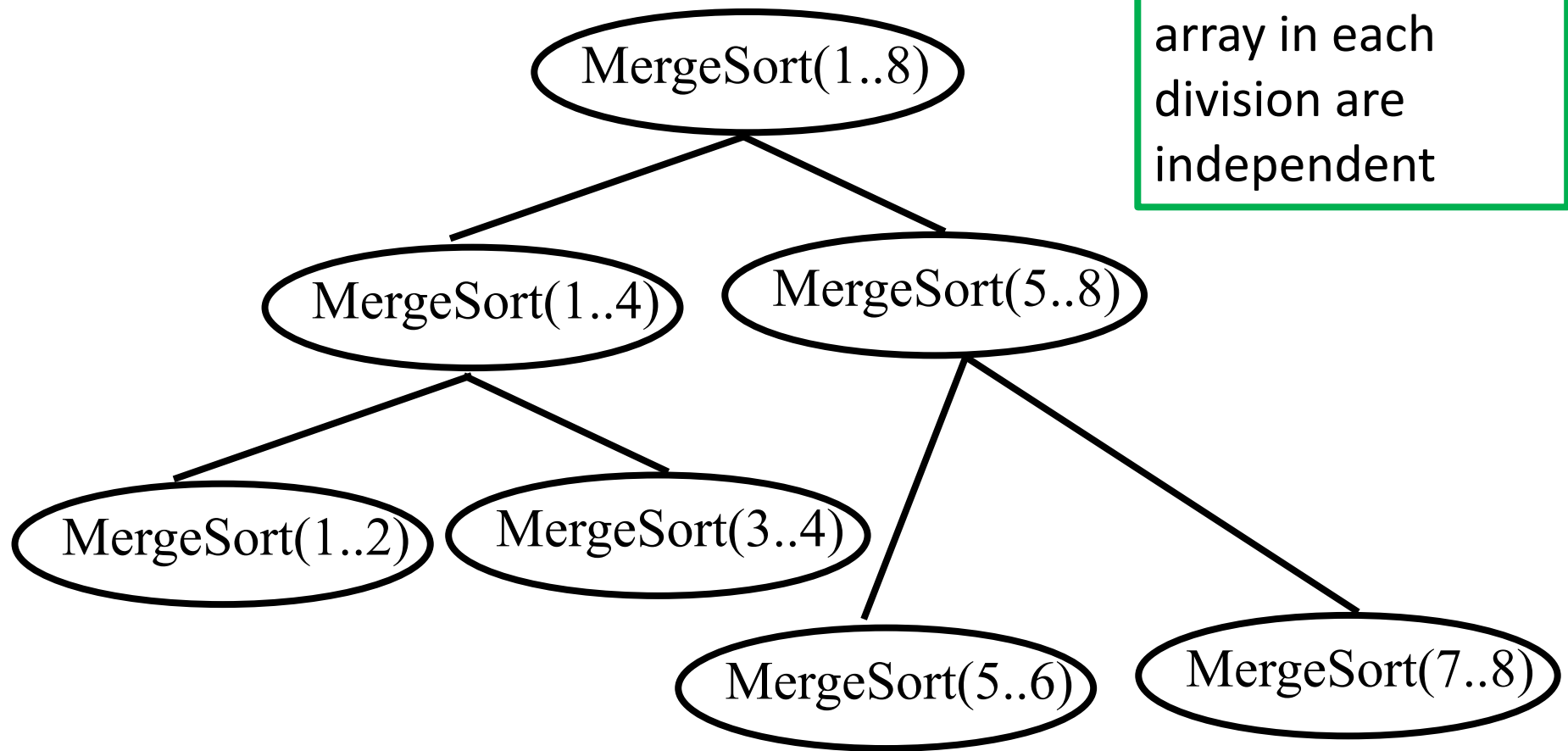
1. Concepts of dynamic programming
2. Longest common subsequence
3. Chain matrix multiplication
4. 0/1 Knapsack problem

# What is Dynamic Programming?

- It is a problem solving paradigm
- To a certain extent, it is similar to divide-and-conquer
- What do we do in divide-and-conquer?
  - Divide a problem into *independent* subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - Example: MergeSort

# Example: MergeSort(8)

The 2 half sections of the array in each division are independent



# What is Dynamic Programming?

- Dynamic programming:
  - Divide a problem into ***overlapping*** subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - ***Do not compute the answer to the same subproblem more than once***
  - Example: computing Fibonacci numbers

# FIBONACCI SEQUENCE

The Fibonacci sequence is defined recursively as:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 2$$

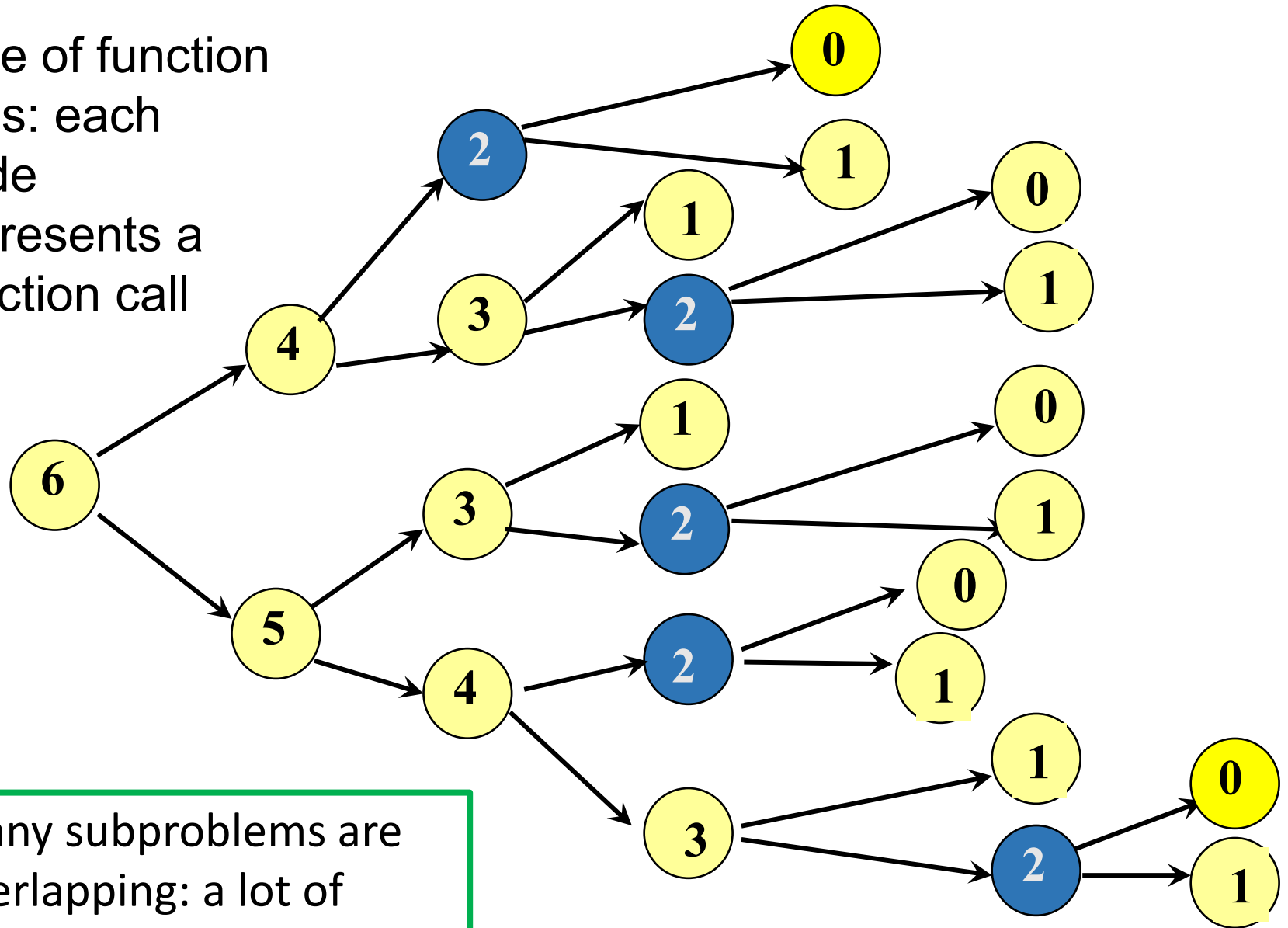
$$F_0 = 0, F_1 = 1$$

This series occurs frequently in algorithm analysis.

Divide-and-conquer: Recursive Fibonacci function

```
int fib(n)
{
    if (n == 0 || n == 1) return n;
    else return fib(n - 1) + fib(n - 2);
}
```

Tree of function calls: each node represents a function call



Many subproblems are overlapping: a lot of recomputation

- Example of repetition is given in the shaded nodes
- Notice that this is a full binary tree up to depth 3 (i.e.  $n/2$ )
- The deepest level is 5 (i.e.  $n-1$ )
- The number of recursive calls  $R$  is such that

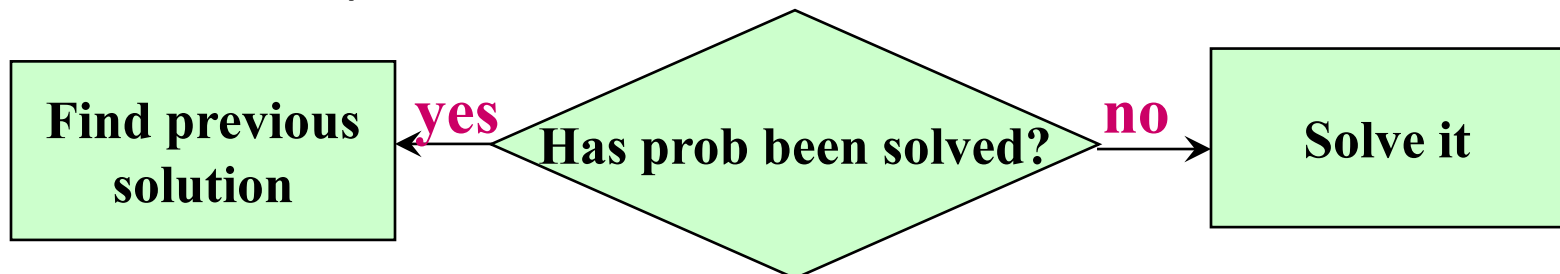
$$2^4 - 1 < R < 2^6 - 1$$

In general,

$$2^{\frac{n}{2}+1} - 1 < R < 2^n - 1$$

So this is an exponential time algorithm:  $O(2^n)$

- The main feature of dynamic programming is that it replaces an exponential-time computation by a polynomial-time computation
- It is done by this: **Memorize the solutions and do not recompute**
- Recall that a DFS on a graph only explores edges to undiscovered vertices, and it checks the other edges.
- This strategy may be applied to only solve unsolved subproblems, and checks and retrieves solutions to the solved subproblems





# Dynamic programming (Top Down)

1. Formulate the problem  $P$  in terms of smaller versions of the problem (recursively), say,  $Q_1, Q_2, \dots$
2. Turn this formulation into a recursive function to solve problem  $P$
3. Use a dictionary to store solutions to subproblems
4. In the recursive function to solve  $P$ 
  - ❖ Before any recursive call, say on subproblem  $Q_i$ , check the dictionary to see if a solution for  $Q_i$  has been stored
    - If no solution has been stored, make the recursive call
    - Otherwise, retrieve the stored solution
  - ❖ Just before returning the solution for  $P$ , store the solution in the dictionary - memorization

# The top-down approach

A dynamic programming version of fib(n)

```
int fibDP(n)
{   int f1, f2;
    if (n == 0 || n == 1) {
        store(Soln, n, n);
        return n;   }
    else {
        if (not member(Soln, n - 1))
            f1 = fibDP(n - 1);
        else f1 = retrieve(Soln, n - 1);
```

**Store,**  
**member,**  
**retrieve** are all  
methods of the  
Dictionary

```
if (not member(Soln, n - 2))
    f2 = fibDP(n - 2);
else f2 = retrieve(Soln, n - 2);

f1 += f2;
store(Soln, n, f1);
return f1;    }
}
```

- Before calling fibDP, the dictionary Soln has to be initialized.  
E.g.

	0	1	2	3	4	5	6
Soln	-1	-1	-1	-1	-1	-1	-1

- member(Dictionary, j):  
**return Dictionary[j] <> -1**
- store(Dictionary, j, s):  
**Dictionary[j] = s**
- retrieve(Dictionary, j):  
**return Dictionary[j]**

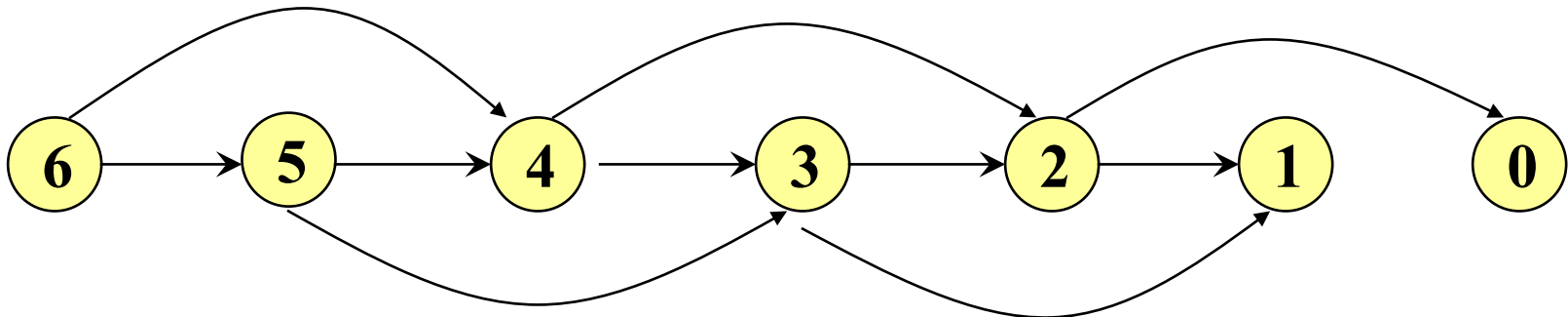
<pre> int fibDP(n) { int f1, f2;   if (n == 0    n == 1) {     store(Soln, n, n);     return n;  }   else {     if (not member(Soln, n - 1))       f1 = fibDP(n - 1);     else f1 = retrieve(Soln, n - 1); </pre>	<pre>     if (not member(Soln, n - 2))       f2 = fibDP(n - 2);     else f2 = retrieve(Soln, n - 2);      f1 += f2;     store(Soln, n, f1);     return f1;  } } </pre> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-top: 20px;"> <b>Complexity: <math>O(n)</math></b> </div>
---	--

The total computational time in each function call, excluding that of the calls to fibDP() on subproblems is bounded by a constant. The total computational cost is thus proportional to the number of calls to fibDP() when solving fibDP(n) – n+1 times. So  $O(n)$ .

# Dynamic programming (Bottom Up)

## Subproblem graphs

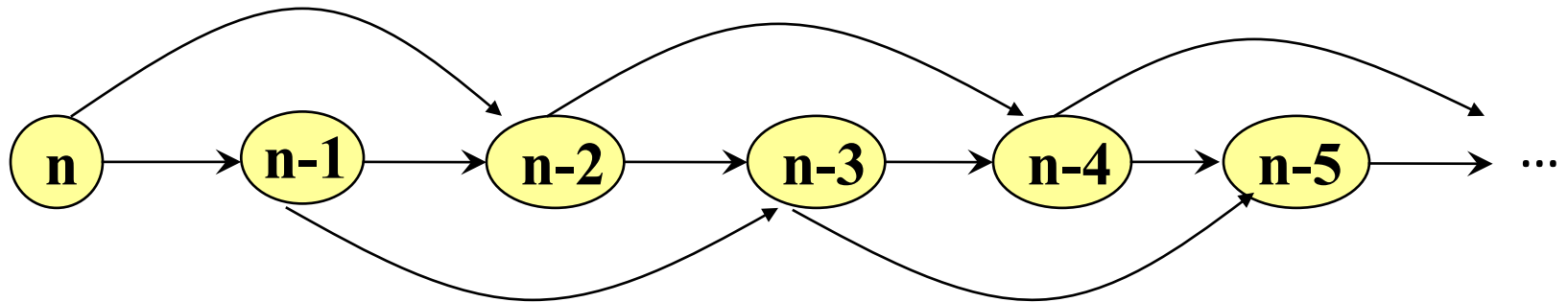
- For a recursive algorithm A, the subproblem graph for A is a directed graph whose vertices are the instances for this problem. The directed edges (I, J) for all pairs that indicate: if A is invoked on problem I, it makes a recursive call directly on instance J.
- E.g. the subproblem graph for fib(6):



# Dynamic programming (Bottom Up)

1. Formulate the problem  $P$  in terms of smaller versions of the problem (recursively), say,  $Q_1, Q_2, \dots$
2. Turn this formulation into a recursive function to solve problem  $P$
3. Draw the subproblem graph and find the dependencies among subproblems
4. Use a dictionary to store solutions to subproblems
5. In the iterative function to solve  $P$ 
  - ❖ compute the solutions of subproblems of a problem first
  - ❖ The solution to  $P$  is computed based on the solutions to its subproblems and is stored into the dictionary

## The subproblem graph of fib(n)



- Observation 1: Since we have a sequence of subproblems for fib(n), we will use an one-dimensional array to memorize the solutions of subproblems. E.g. fib(6)

	0	1	2	3	4	5	6
Soln	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

- Observation 2: Seeing the dependencies among the solutions – fib(n) needs the solutions of fib(n-1) and fib(n-2), we can compute the elements in this array in a correct order.



```
soln[0] = 0;  
soln[1] = 1;  
for j = 2 to n  
    soln[j] = soln[j-1] + soln[j-2];
```

**Complexity:  $O(n)$**

	0	1	2	3	4	5	6
Soln	0	1	1	2	3	5	8

# Longest Common Subsequence

- Given a sequence  $s = \langle s_1, s_2, \dots, s_n \rangle$ , a **subsequence** is any sequence  $\langle s_{i_1}, s_{i_2}, \dots, s_{i_m} \rangle$ , with  $i_j$  strictly increasing.

Example:  $s = \text{ACTTGCG}$

*ACT, AG, ATTC, T, ACTTGC* are all subsequences.

*TTA, AGGC* are not subsequences.

- Given two sequences  $x = \langle x_1, x_2, \dots, x_n \rangle$ ,  $y = \langle y_1, y_2, \dots, y_m \rangle$ , a **common subsequence** is a subsequence of both  $x$  and  $y$ .

- A **longest common subsequence** (LCS) is a common subsequence of maximum length

Example:  $x = AAACCGTGAGTTATTCGTTCTAGAA$

$y = CACCCCTAAGGTACCTTTGGTTC$

Common subsequences:  $ACGG$ ,  $CAGTTTC$

LCS =  **$ACCTAGTACTTTG$**  (LCS may not be unique)

- LCS has many applications including document analysis and computational biology – the similarity between two sequences is measured by the length of LCS

- How similar are these two species?



DNA:

AGCCCTAAGGGCTACCTAGCTT



DNA:

GACAGCCTACAAGCGTTAGCTTG

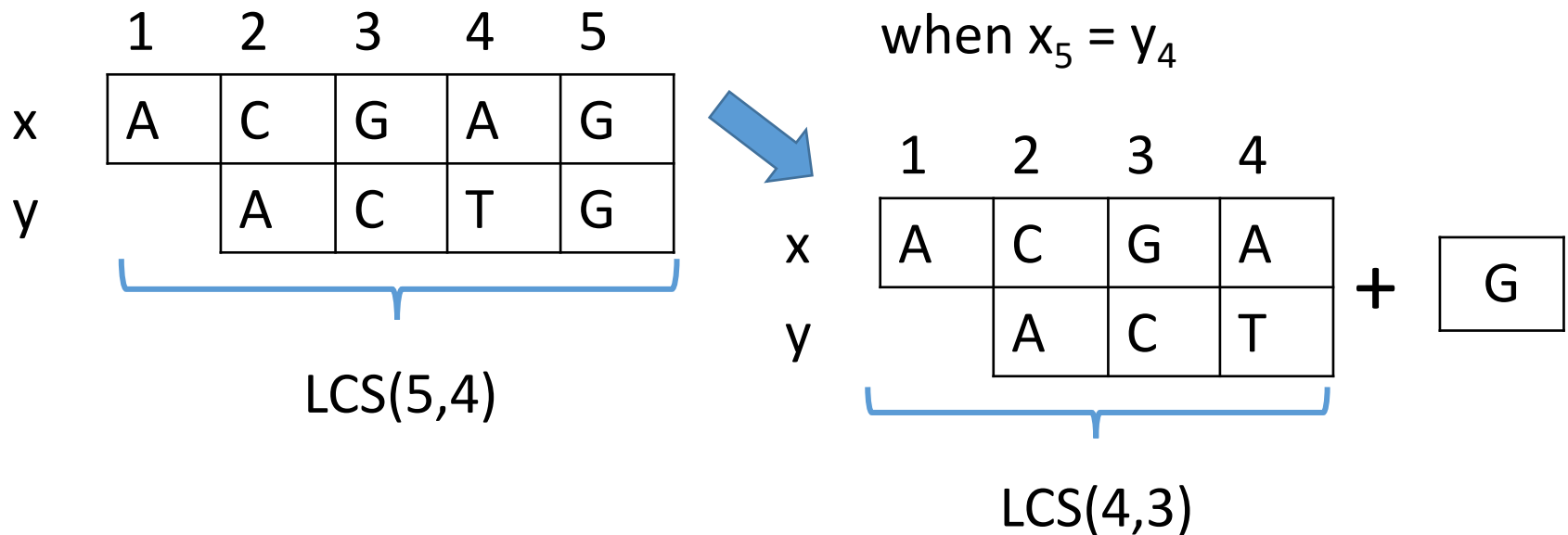
Problem definition: Given two sequences  $x = \langle x_1, x_2, \dots, x_n \rangle$ ,  $y = \langle y_1, y_2, \dots, y_m \rangle$ , compute  $\text{LCS}(n, m)$  that gives the length of the longest common subsequence

- A trivial algorithm: find all subsequences of  $x$  (there are up to  $2^n$  of them) and check whether they are subsequences of  $y$ .
- What is the complexity of this trivial algorithm?
- This is an optimization problem
- Dynamic programming is a powerful tool to solve optimization problems that satisfy the *Principle of Optimality*
- A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

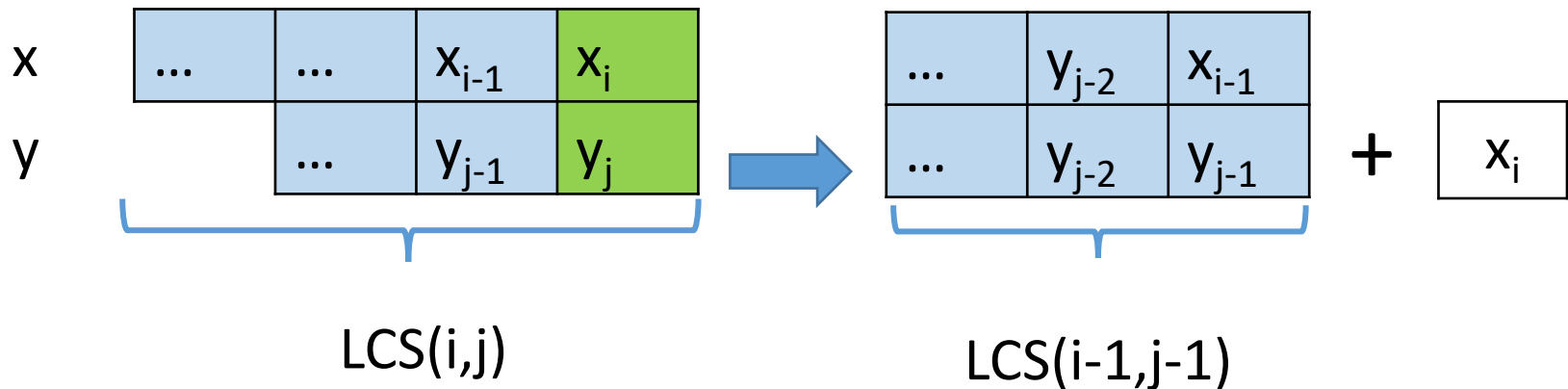
- Does the LCS problem satisfy this principle?

Step 1: Formulate the problem P in terms of smaller versions of the problem

- Consider two sequences  $x = \langle x_1, x_2, \dots, x_i \rangle$ ,  $y = \langle y_1, y_2, \dots, y_j \rangle$ . Take them as character strings. E.g.



- If  $x_i = y_j$ , then this character is the last character in the longest common subsequence. The longest common subsequence is the longest common subsequence of  $\langle x_1, x_2, \dots, x_{i-1} \rangle$ , and  $\langle y_1, y_2, \dots, y_{j-1} \rangle$  followed by  $x_i$ .



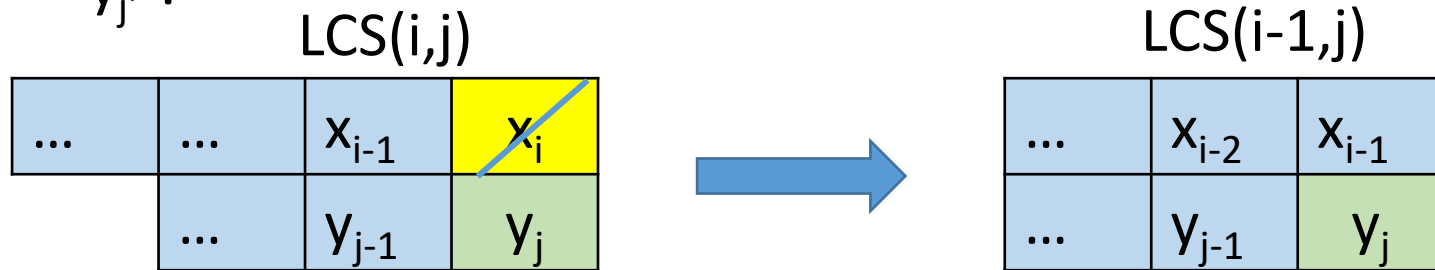
$LCS(i-1, j-1)$  is a subsolution of  $LCS(i, j)$  when  $x_i = y_j$ .

$LCS(i-1, j-1)$  is an optimal solution.

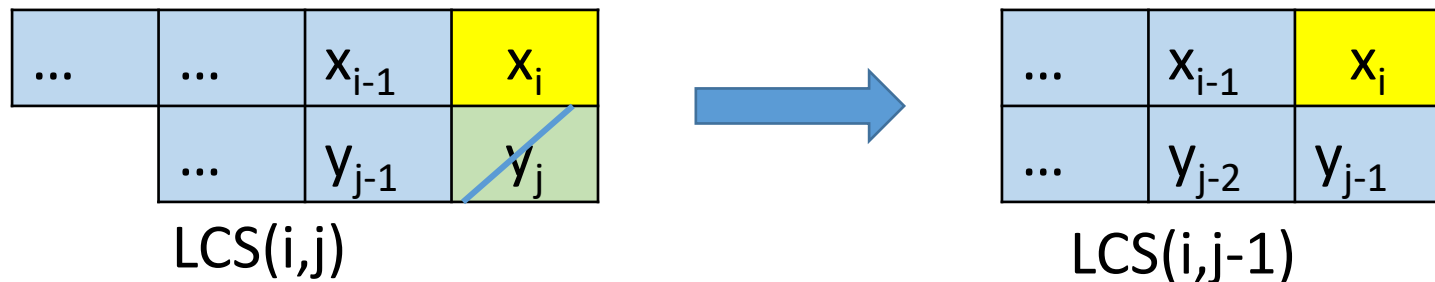
Otherwise  $LCS(i, j)$  cannot be an optimal solution

- If  $x_i \neq y_j$ , then either  $x_i$  is not in the LCS or  $y_j$  is not in the LCS (or both of them are not in the LCS).

If  $x_i$  is not in the LCS, we just need to find the longest common subsequence of  $\langle x_1, x_2, \dots, x_{i-1} \rangle$  and  $\langle y_1, y_2, \dots, y_j \rangle$ .

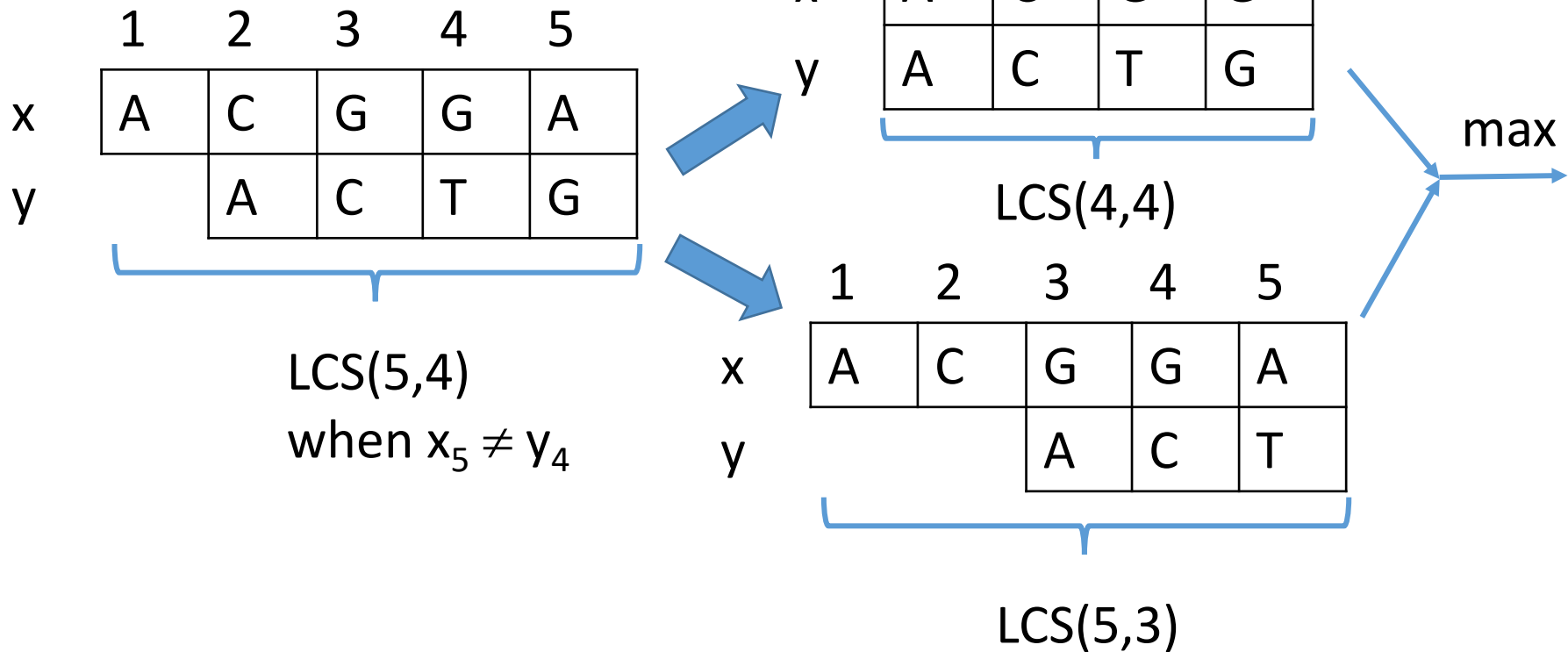


If  $y_j$  is not in the LCS, we just need to find the longest common subsequence of  $\langle x_1, x_2, \dots, x_i \rangle$  and  $\langle y_1, y_2, \dots, y_{j-1} \rangle$ .



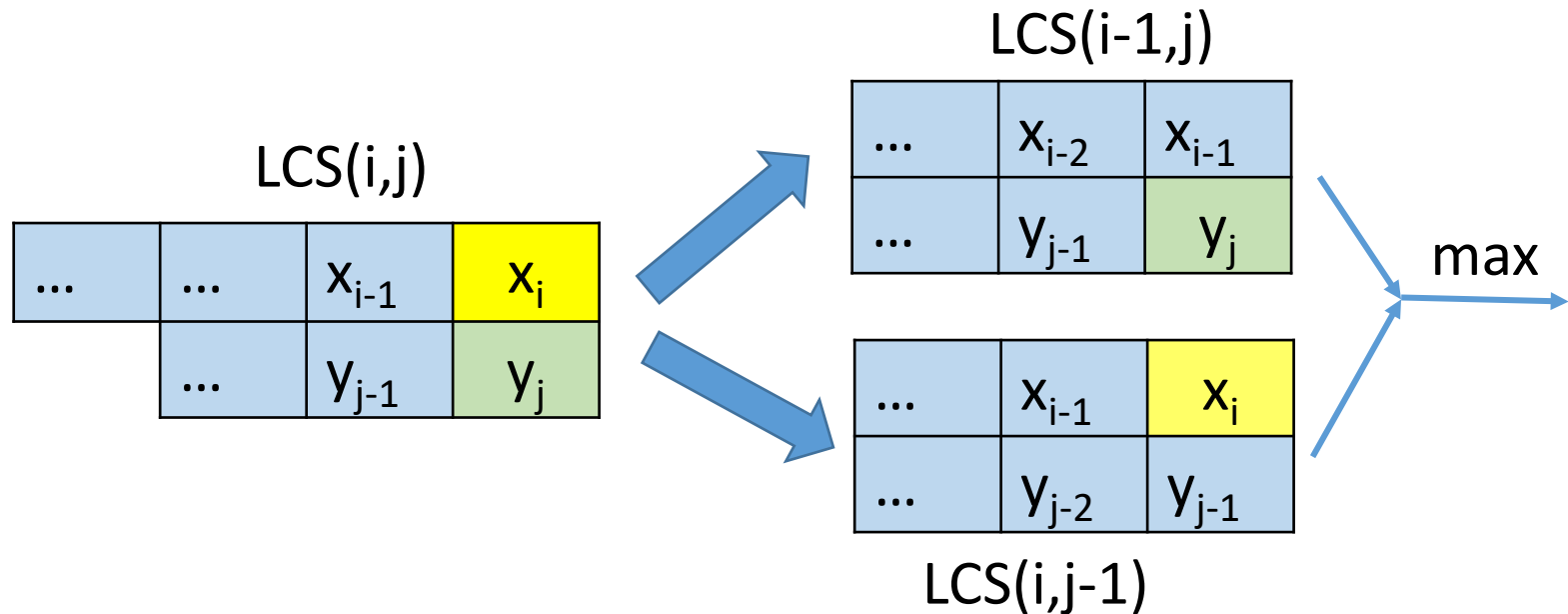


○ E.g.,



- The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

Therefore, when  $x_i \neq y_j$ ,



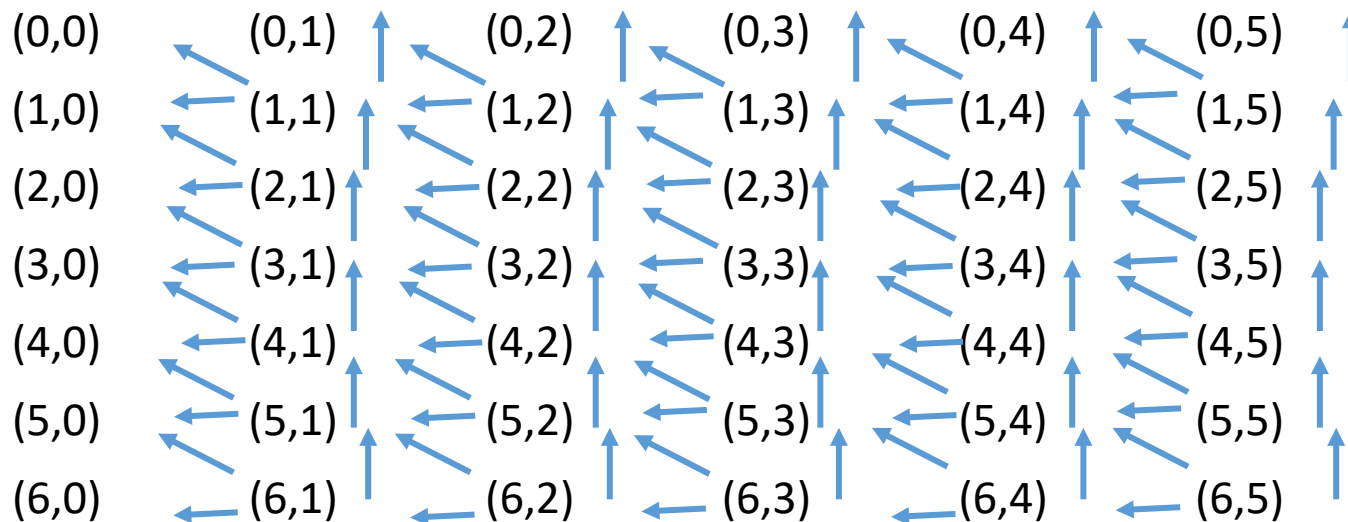
- $LCS(i-1,j-1)$ ,  $LCS(i-1,j)$  and  $LCS(i,j-1)$  are the optimal solutions for the respective subproblems. Otherwise  $LCS(i,j)$  cannot be optimal – principle of optimality.

Step 2: Turn this formulation into a recursive function to solve the longest common subsequence problem:

$$\begin{array}{ll} \text{LCS}(i,j) = 0 & \text{if } i=0 \text{ or } j=0 \\ \text{LCS}(i,j) = \text{LCS}(i-1,j-1) + 1 & \text{if } i,j > 0, x_i = y_j \\ \text{LCS}(i,j) = \max(\text{LCS}(i-1,j), \text{LCS}(i,j-1)) & \text{if } i,j > 0, x_i \neq y_j \end{array}$$

- The top down approach using a recursive function will be very inefficient.

Step 3 (bottom up approach): Draw the subproblem graph and find the dependencies among subproblems  
E.g, the subproblem graph of LCS(6,5)



Step 4: the dictionary is a  $n+1$  by  $m+1$  array.

Initialise row 0, column 0.

Compute from row 1 to row  $n$ , column 1 to column  $m$  within each row.

## Step 5

```
Int LCS(n, m)
{
    for i = 0 to n  c[i][0] = 0;
    for j = 1 to m  c[0][j] = 0;
    for i = 1 to n
        for j = 1 to m
            if x[i] == y[j]
                c[i][j] = c[i-1][j-1] + 1;
            else if c[i-1][j] >= c[i][j-1]
                c[i][j] = c[i-1][j];
            else c[i][j] = c[i][j-1];
    return c[n][m];
}
```

```
Int LCS(n, m)
```

```
{
```

```
    for i = 0 to n    c[i][0] = 0;
```

```
    for j = 1 to m    c[0][j] = 0;
```

```
    for i = 1 to n
```

```
        for j = 1 to m
```

```
            if x[i] == y[j]
```

```
                c[i][j] = c[i-1][j-1] + 1;
```

```
            else if c[i-1][j] >= c[i][j-1]
```

```
                c[i][j] = c[i-1][j];
```

```
            else c[i][j] = c[i][j-1];
```

```
    return c[n][m];
```

```
}
```

**Space Complexity:**  
(n+1)x(m+1) array  
 **$O(nm)$**

Total no. of  
iterations:  $nm$

Bounded by a  
constant time

**Time Complexity:**  
 **$O(nm)$**

## Example 1

for i = 0 to n   c[i][0] = 0;  
for j = 1 to m   c[0][j] = 0;

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

		A	C	T	G
	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

## Example 1

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

	A	C	T	G
	0	0	0	0
A	0	1	1	1
C	0	1	2	2
G	0	1	2	3
G	0	1	2	3
A	0	1	2	<b>3</b>

$$\text{LCS}(5,4) = 3$$

```
for i = 1 to n
  for j = 1 to m
    if x[i] == y[j]
      c[i][j] = c[i-1][j-1] + 1;
    else if c[i-1][j] >= c[i][j-1]
      c[i][j] = c[i-1][j];
    else c[i][j] = c[i][j-1];
```



- To find the longest common subsequence, a hint array is used in  $\text{LCS}()$  function to indicate for  $\text{LCS}(i,j)$  where the optimal subsolution is from :  $\text{LCS}(i-1, j-1)$ ,  $\text{LCS}(i-1, j)$  or  $\text{LCS}(i, j-1)$ .
  - For the hint array cell  $h[i][j]$  where  $i \neq 0$  and  $j \neq 0$ ,
    - If we do  $\text{LCS}(i,j) = \text{LCS}(i-1, j-1) + 1$ ,  $h[i][j] = \backslash$
    - If we do  $\text{LCS}(i,j) = \text{LCS}(i-1, j)$ ,  $h[i][j] = |$
    - If we do  $\text{LCS}(i,j) = \text{LCS}(i, j-1)$ ,  $h[i][j] = -$
  - First column of the hint array will be filled with  $|$ .
  - First row of the hint array will be filled  $-$ .

```

Int LCS(n, m)    // with hints to find the sequence
{
    for i = 0 to n { c[i][0] = 0; h[i][0] = '|'; }
    for j = 1 to m { c[0][j] = 0; h[0][j] = '—'; }
    for i = 1 to n
        for j = 1 to m
            if x[i] == y[j]
                { c[i][j] = c[i-1][j-1] + 1; h[i][j] = '\'; }
            else if c[i-1][j] >= c[i][j-1]
                { c[i][j] = c[i-1][j]; h[i][j] = '|'; }
            else { c[i][j] = c[i][j-1]; h[i][j] = '—'; }
    return c[n][m];
}

```

**Time Complexity:**  
 **$O(nm)$**

- To obtain the longest common subsequence computed, we start from  $h[n][m]$ .
- For each element of the hint array,  $h[i][j]$ ,
  - If  $h[i][j] = '\backslash'$ , it means  $x_i = y_j$  and this character is the last character of the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$ . This character is preceded by the longest common subsequence of  $x_1..x_{i-1}$  and  $y_1..y_{j-1}$ .
  - If  $h[i][j] = '|'$ , it means the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$  is the longest common subsequence of  $x_1..x_{i-1}$  and  $y_1..y_j$ .
  - If  $h[i][j] = '-'$ , it means the longest common subsequence of  $x_1..x_i$  and  $y_1..y_j$  is the longest common subsequence of  $x_1..x_i$  and  $y_1..y_{j-1}$ .
- After reaching the 1<sup>st</sup> row/column of the hint array, end

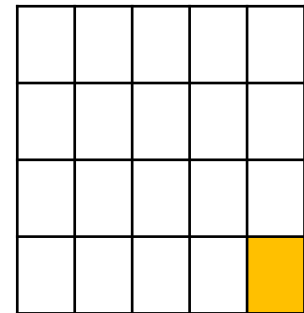
```

getSequence(n m)    // get the LCS from hint array
{
    s = empty stack; // s stores the characters in LCS
    i = n;
    j = m;
    while (i ≠ 0 and j ≠ 0)
        if (h[i][j] == '\')
            { s.push(x[i]); i--; j--; }
        else if (h[i][j] == '|')
            i--;
        else j--;
    pop and output from s;
}

```

**Maximum no. of iterations:  $n+m$ .**  
**Complexity:  $O(n+m)$**

Bounded by a constant time



# Example 1

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

		A	C	T	G
	—	—	—	—	—
A		\	—	—	—
C			\	—	—
G					\
G					\
A		\			

	1	2	3	4	5
x	A	C	G	G	A
y	A	C	T	G	

$h(5,4) = '|'$

$h(4,4) = '\backslash' \rightarrow \boxed{G}$

$h(3,3) = '|'$

$h(2,3) = '—'$

$h(2,2) = '\backslash' \rightarrow \boxed{C}$

$h(1,1) = '\backslash' \rightarrow \boxed{A}$

end

The sub sequence:

**A C G**

Example 2:

<b>x</b>	C	G	G	T	A	T
<b>y</b>	A	G	T	T	G	C

		A	G	T	T	G	C
C	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
G	0	0	1	1	1	2	2
T	0	0	1	2	2	2	2
A	0	1	1	2	2	2	2
T	0	1	1	2	3	3	<b>3</b>

		A	G	T	T	G	C
C	—	—	—	—	—	—	—
G							\
G			\	—	—	\	
T				\	\		
A		\					
T				\	\	—	—

$$\text{LCS}(6,6) = 3$$

Example 2:

<b>x</b>	C	G	G	T	A	T
<b>y</b>	A	G	T	T	G	C

		<b>A</b>	<b>G</b>	<b>T</b>	<b>T</b>	<b>G</b>	<b>C</b>
	—	—	—	—	—	—	—
<b>C</b>							\
<b>G</b>			\	—	—	\	
<b>G</b>			\			\	—
<b>T</b>				\	\		
<b>A</b>		\					
<b>T</b>				\	\	—	—

$LCS(6,6) = 3$

$h(6,6) = \text{—}$

$h(6,5) = \text{—}$

$h(6,4) = \backslash$

$h(5,3) = |$

$h(4,3) = \backslash$

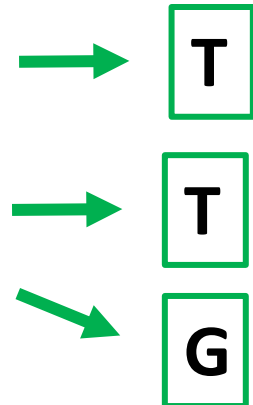
$h(3,2) = \backslash$

$h(2,1) = |$

$h(1,1) = |$

$h(0,1) = \text{—}$

end



The subsequence:

**G T T**

# Chain Matrix Multiplication

- The Order problem

Consider  $A_1$  x  $A_2$  x  $A_3$  x  $A_4$   
 $30 \times 1$     $1 \times 40$     $40 \times 10$     $10 \times 25$

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$2 \times 3 \times 4$

Many possibilities. For examples,

$$((A_1 A_2) A_3) A_4 \longrightarrow 30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20,700 \text{ multiplications}$$

$$A_1 (A_2 (A_3 A_4)) \longrightarrow 40 \times 10 \times 25 + 1 \times 40 \times 25 + 30 \times 1 \times 25 = 11,750 \text{ multiplications}$$

$$(A_1 A_2) (A_3 A_4) \longrightarrow 30 \times 1 \times 40 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41,200 \text{ multiplications}$$

$$A_1 ((A_2 A_3) A_4) \longrightarrow 1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1,400 \text{ multiplications}$$



Problem definition: given matrices  $A_1, A_2, \dots, A_n$  where dimensions of  $A_i$  are  $d_{i-1} \times d_i$  (for  $1 \leq i \leq n$ ), what order should the matrix multiplications be computed in order to incur minimum cost? Cost is the number of multiplications.

$d_0 \quad d_1 \quad d_2 \quad d_3 \quad \dots \quad d_{n-1} \quad d_n$

- There are  $(n-1)!$  ways for  $n$  matrices
- Matrix multiplication is associative:  $(AB)C = A(BC)$ . So different ways give the same result
- This is an optimization problem

Step 1: formulate the matrix multiplication cost problem in terms of smaller versions of the same problem

Consider a sequence of 6 matrices:

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$  matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions



Suppose the last matrix multiplication were at  $A_3$ ; then

- 1) We need to multiply  $A_1 \times A_2 \times A_3$  to create  $B_1$ , a  $d_0 \times d_3$  matrix
- 2) We need to multiply  $A_4 \times A_5 \times A_6$  to create  $B_2$ , a  $d_3 \times d_6$  matrix

Cost would be the cost of (1)+(2)+ cost of( $B_1 \times B_2$ )

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$  matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions

The last multiplication may be at each of the 5 matrices.

$$\text{Cost}((A_1 A_2 A_3 A_4 A_5)(A_6)) = \text{Cost}(A_1 A_2 A_3 A_4 A_5) + \text{Cost}(A_6) \\ + d_0 \times d_5 \times d_6$$

$$\text{Cost}((A_1 A_2 A_3 A_4)(A_5 A_6)) = \text{Cost}(A_1 A_2 A_3 A_4) + \text{Cost}(A_5 A_6) \\ + d_0 \times d_4 \times d_6$$

$$\text{Cost}((A_1 A_2 A_3)(A_4 A_5 A_6)) = \text{Cost}(A_1 A_2 A_3) + \text{Cost}(A_4 A_5 A_6) \\ + d_0 \times d_3 \times d_6$$

$$\text{Cost}((A_1 A_2)(A_3 A_4 A_5 A_6)) = \text{Cost}(A_1 A_2) + \text{Cost}(A_3 A_4 A_5 A_6) \\ + d_0 \times d_2 \times d_6$$

$$\text{Cost}((A_1)(A_2 A_3 A_4 A_5 A_6)) = \text{Cost}(A_1) + \text{Cost}(A_2 A_3 A_4 A_5 A_6) \\ + d_0 \times d_1 \times d_6$$

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$  matrices

$d_0 \times d_1 \quad d_1 \times d_2 \quad d_2 \times d_3 \quad d_3 \times d_4 \quad d_4 \times d_5 \quad d_5 \times d_6$  dimensions

The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

The optimal cost of multiplying the 6 matrices:

$$\text{OptCost}(A_1 A_2 A_3 A_4 A_5 A_6) = \text{Min}(\begin{aligned} &\text{OptCost}(A_1 A_2 A_3 A_4 A_5) + \text{OptCost}(A_6) + d_0 \times d_5 \times d_6, \\ &\text{OptCost}(A_1 A_2 A_3 A_4) + \text{OptCost}(A_5 A_6) + d_0 \times d_4 \times d_6, \\ &\text{OptCost}(A_1 A_2 A_3) + \text{OptCost}(A_4 A_5 A_6) + d_0 \times d_3 \times d_6, \\ &\text{OptCost}(A_1 A_2) + \text{OptCost}(A_3 A_4 A_5 A_6) + d_0 \times d_2 \times d_6, \\ &\text{OptCost}(A_1) + \text{OptCost}(A_2 A_3 A_4 A_5 A_6) + d_0 \times d_1 \times d_6 ) \end{aligned}$$

$$\text{OptCost}(A) = 0$$

Step 2: Turn this formulation into a recursive function to solve the chain matrix multiplication problem.

Suppose we use array **d** to store the dimensions of the matrices.

$d_0$	$d_1$	$d_2$	...		
-------	-------	-------	-----	--	--

Let  $\text{OptCost}(i,j)$  be the optimal cost of multiplying matrices with dimensions  $d_i \times d_{i+1}$ ,  $d_{i+1} \times d_{i+2}$ , ...,  $d_{j-1} \times d_j$ .

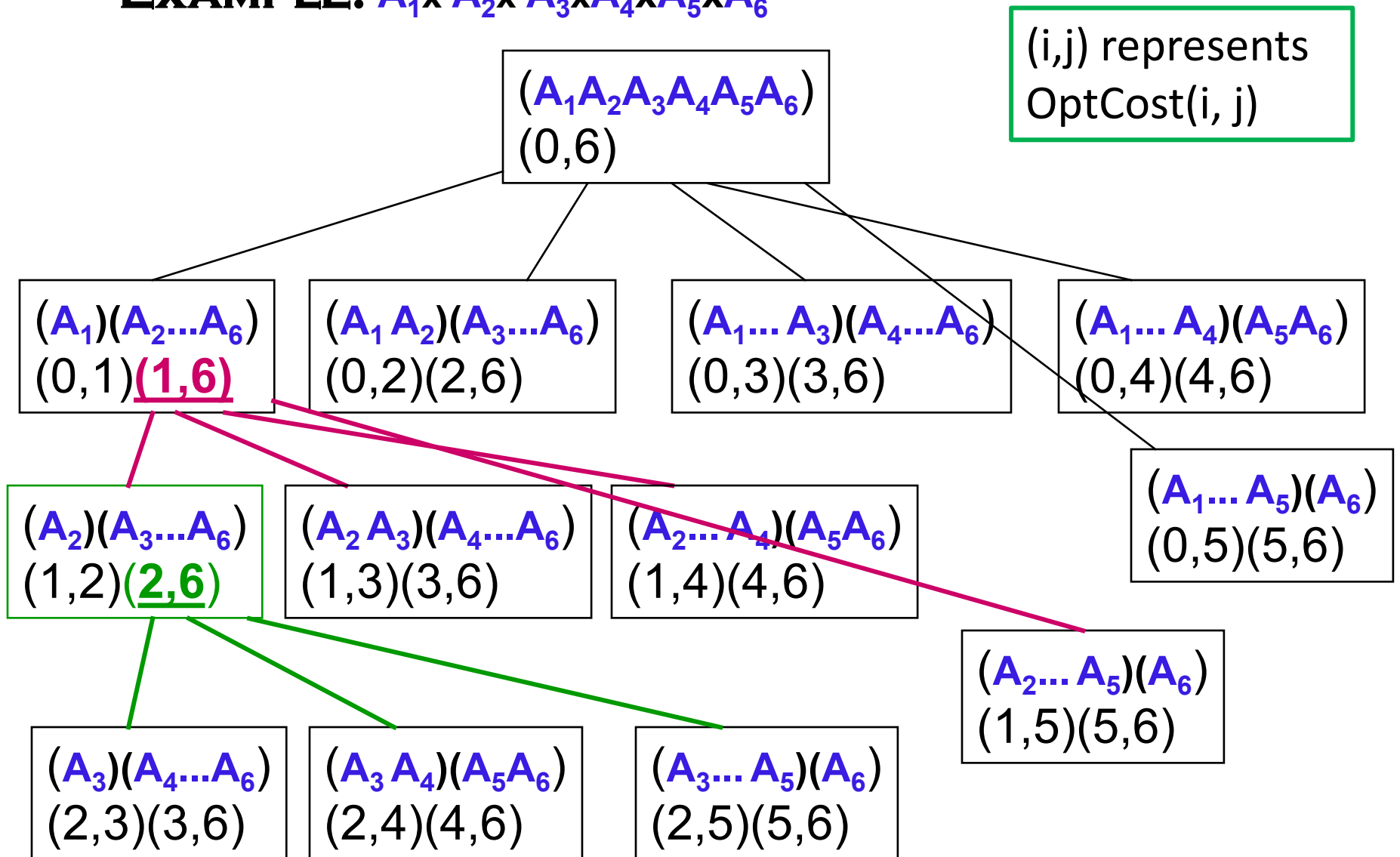
$$\text{OptCost}(i, j) = 0 \quad \text{if } j-i=1$$

$$\begin{aligned} \text{OptCost}(i, j) \\ = \min_{i+1 \leq k \leq j-1} (\text{OptCost}(i,k) + \text{OptCost}(k, j) + d_i \times d_k \times d_j) \end{aligned} \quad \text{if } j-i > 1$$

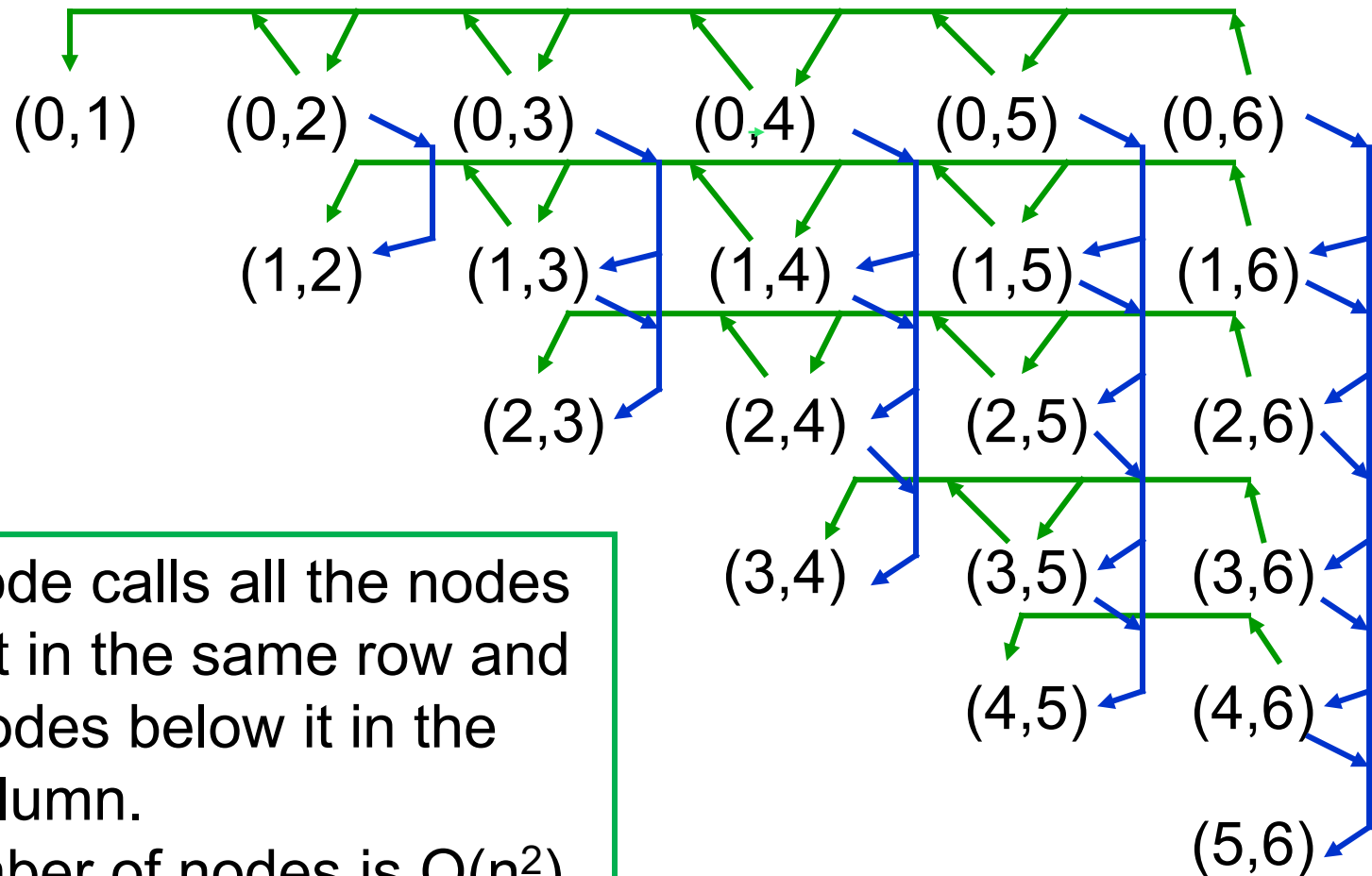
The optimal cost of multiplying  $n$  matrices is  $\text{OptCost}(0, n)$ .

- The chain matrix multiplication problem satisfies the principle of optimality
  - $\text{OptCost}(i, k)$  and  $\text{OptCost}(k, j)$  for  $k = i+1, \dots, j-1$  are the subsolutions of  $\text{OptCost}(i, j)$
  - They are the optimal solutions for the subproblems
  - Proof by contradiction

**EXAMPLE:**  $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$



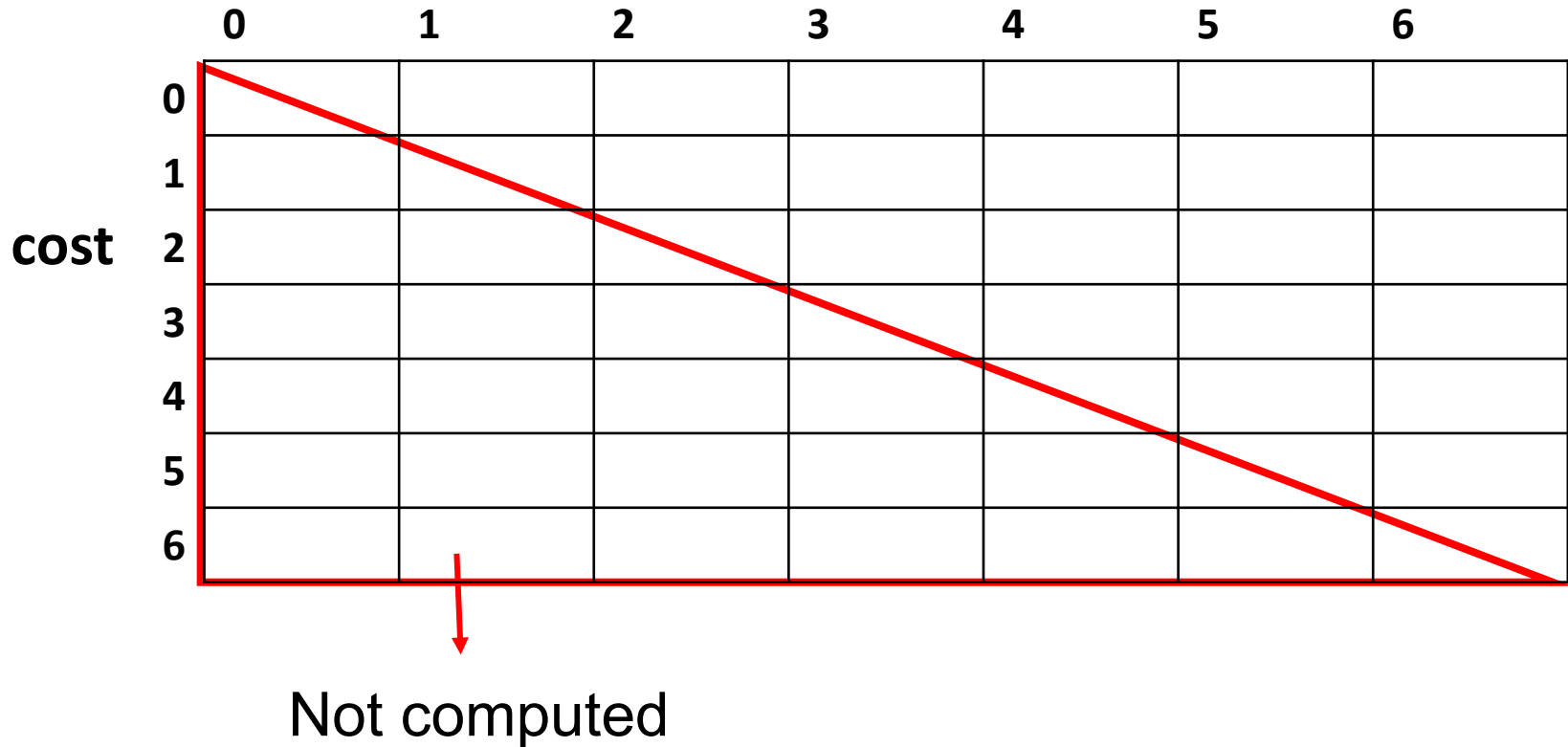
Step 3: Draw the subproblem graph and find the dependencies among subproblems



Every node calls all the nodes on its left in the same row and all the nodes below it in the same column.  
The number of nodes is  $O(n^2)$ .

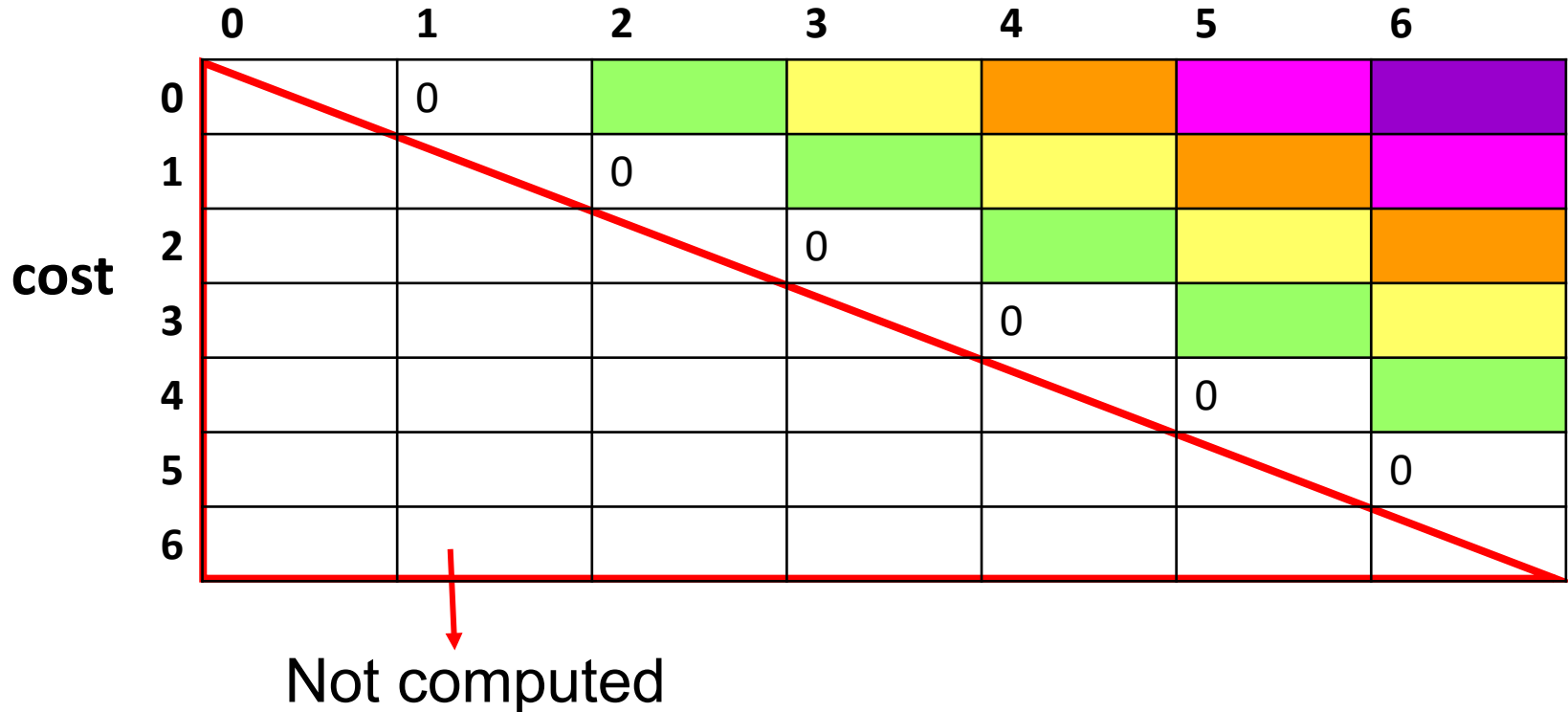


## Step 4: Dictionary: $\text{cost}[n+1][n+1]$



## Step 5




Order to solve the subproblems



Use another array,  $last[n+1][n+1]$  to represent the index of the last multiplication to be done for a subproblem

# Find the pattern

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

-  Row number and column number differ by 2, row goes from 0 to 4
-  Row number and column number differ by 3, row goes from 0 to 3
-  Row number and column number differ by 4, row goes from 0 to 2

## Find the pattern

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



Row number and column number differ by 5, row goes from 0 to 1



Row number and column number differ by 6, row goes from 0 to 0

Thus, row number and column number differ by 2 to 6, within each difference, row goes from 0 to n minus this difference

```
int matrixOrder(int [] d, int n)
```

```
{  for i = 0 to n-1    cost[i][i+1] = 0;
```

```
    for l = 2 to n
```

```
        for i = 0 to n-l
```

```
            j = i + l;
```

```
            cost[i][j] = ∞;
```

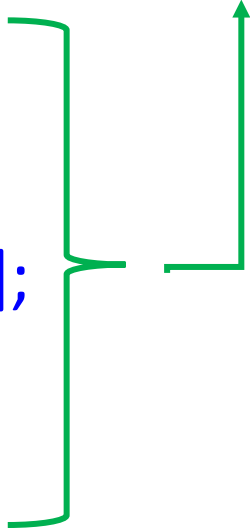
```
            for k = i+1 to j-1
```

```
                c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
```

```
                if (c < cost[i][j])
```

```
                    cost[i][j] = c; last[i][j] = k;
```

```
}
```

$$\min_{i+1 \leq k \leq j-1} (\text{OptCost}(i,k) + \text{OptCost}(k, j) + d_i \times d_k \times d_j)$$


```
int matrixOrder(int [] d, int n)
```

```
{  for i = 0 to n-1    cost[i][i+1] = 0;
```

```
  for l = 2 to n
```

```
    for i = 0 to n-l
```

```
      j = i + l;
```

```
      cost[i][j] =  $\infty$ ;
```

```
      for k = i+1 to j-1
```

```
        c = cost[i][k] + cost[k][j] + d[i]*d[k]*d[j];
```

```
        if (c < cost[i][j])
```

```
          cost[i][j] = c; last[i][j] = k;
```

```
}
```

**Complexity of  
computing the  
optimal order:  
 $O(n^3)$**

Repeated  $O(n^2)$  times

Repeated  $O(n^3)$  times

## Example

$$\mathbf{A}_1 \quad \mathbf{x} \quad \mathbf{A}_2 \quad \mathbf{x} \quad \mathbf{A}_3 \quad \mathbf{x} \quad \mathbf{A}_4$$

$$30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25$$

Array d

<b>30</b>	<b>1</b>	<b>40</b>	<b>10</b>	<b>25</b>
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>

Call to matrixOrder(d, 4)

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>0</b>		0			
<b>1</b>			0		
<b>2</b>				0	
<b>3</b>					0
<b>4</b>					

**for i = 0 to n-1**  
**cost[i][i+1] = 0;**



<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200		
1			0		
2				0	
3					0
4					

	0	1	2	3	4
0			1		
1					
2					
3					
4					

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**last[i][j] = k;**

**l=2, i=0, j=2, k=1**

k=1:

Cost[0][1]+

Cost[1][2] + 1200

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200		
1			0	400	
2				0	
3					0
4					

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

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**last[i][j] = k;**

**l=2, i=1, j=3, k=2**

k=2:

Cost[1][2]+

Cost[2][3] + 400

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200		
1			0	400	
2				0	10000
3					0
4					

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

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**last[i][j] = k;**

**l=2, i=2, j=4, k=3**

k=3:

Cost[2][3]+

Cost[3][4] + 10000

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200	700	
1			0	400	
2				0	10000
3					0
4					

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

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**last[i][j] = k;**

**l=3, i=0, j=3, k=1,2**

k=1:  
Cost[0][1]+  
Cost[1][3] + 300  
k=2:  
Cost[0][2]+  
Cost[2][3] + 12000

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200	700	
1			0	400	650
2				0	10000
3					0
4					

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

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**last[i][j] = k;**

**l=3, i=1, j=4, k=2,3**

k=2:  
Cost[1][2]+  
Cost[2][4] + 1000  
k=3:  
Cost[1][3]+  
Cost[3][4] + 250

<b>d</b>	30	1	40	10	25
	0	1	2	3	4

**cost**

	0	1	2	3	4
0		0	1200	700	1400
1			0	400	650
2				0	10000
3					0
4					

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

**last**

**for l = 2 to n**

**for i = 0 to n-l**

**j = i + l;**

**cost[i][j] = ∞;**

**for k = i+1 to j-1**

**c = cost[i][k] + cost[k][j]  
+ d[i]\*d[k]\*d[j];**

**if (c < cost[i][j])**

**cost[i][j] = c;**

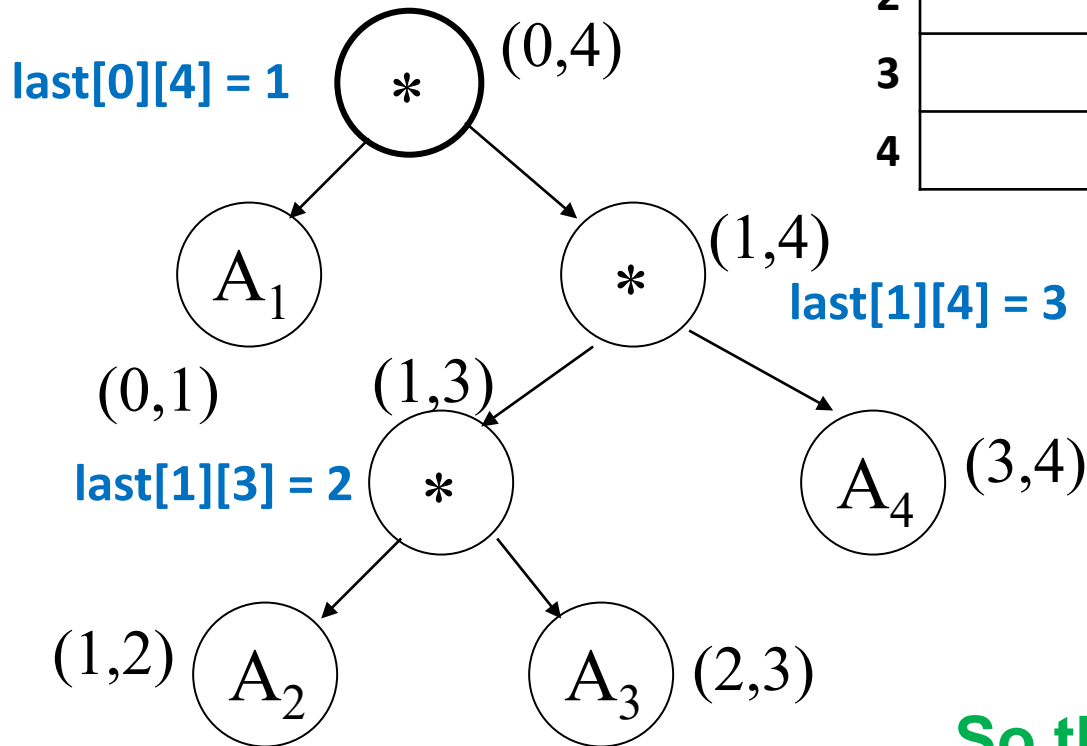
**last[i][j] = k;**

**l=4, i=0, j=4, k=1,2,3**

k=1:  
Cost[0][1]+  
Cost[1][4] + 750  
k=2:  
Cost[0][2]+  
Cost[2][4] + 30000  
k=3:  
Cost[0][3]+  
Cost[3][4] + 7500

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

**last**



**So the best sequence is  
(A1 x ((A2 x A3 ) x A4 ))**

# 0/1 Knapsack problem

Problem definition: We have a knapsack of capacity weight  $C$  (a positive integer) and  $n$  objects with weights  $w_1, w_2, \dots, w_n$  and profits  $p_1, p_2, \dots, p_n$  (all  $w_i$  and all  $p_i$  are positive integers), find the largest total profit of any subset of the objects that fits in the knapsack.

- We have to take whole objects.
- There are  $2^n$  subsets of  $n$  objects: examining all subsets takes  $O(2^n)$  time

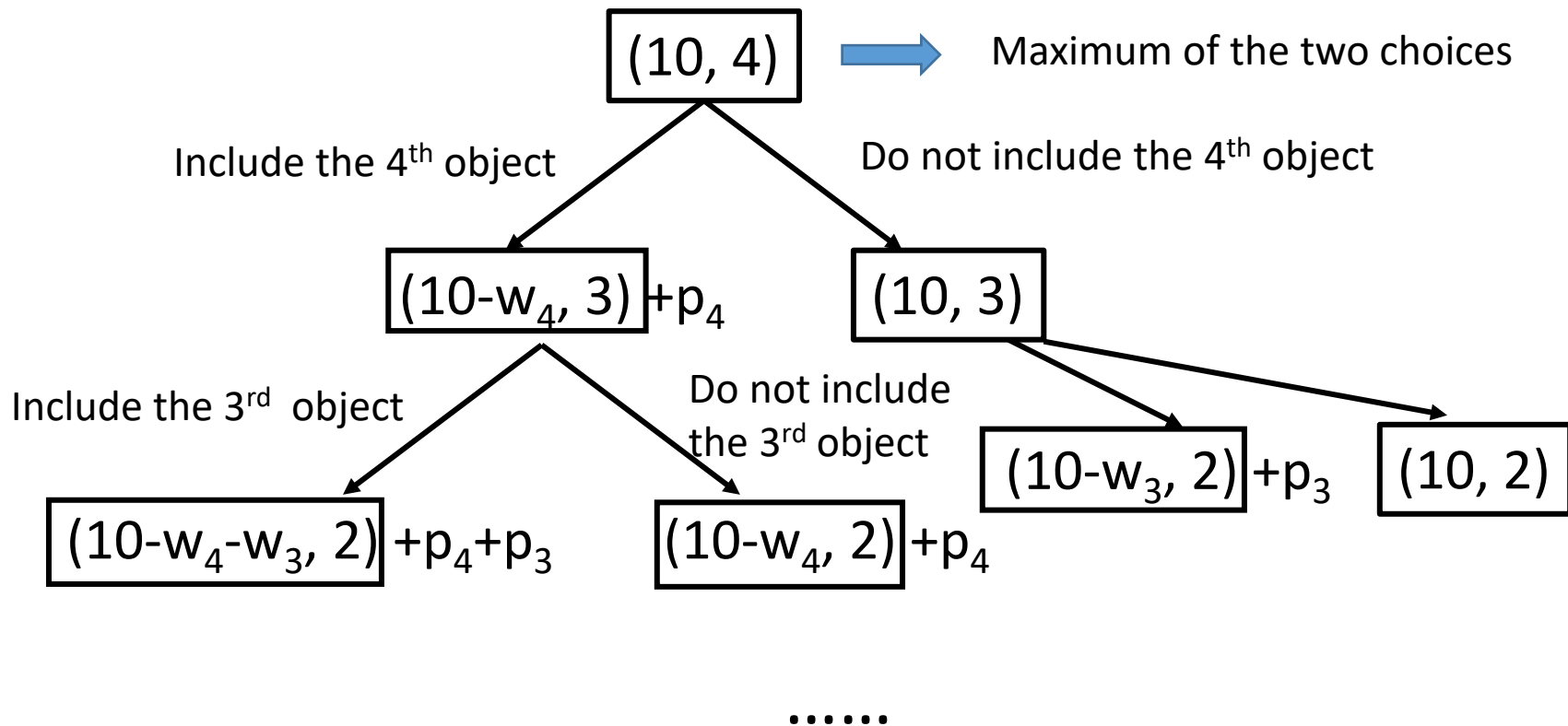
E.g. application:  $C$  is amount of money to invest, weights,  $w_1, \dots$  are investment amounts and profit is the expected return on investment.





- Step 1: formulate the 0/1 knapsack problem in terms of smaller versions of the same problem
  - Consider the last object of the  $n$  objects with weights  $w_1, w_2, \dots, w_n$ .
  - If we include it in the knapsack, the available weight capacity in the knapsack will be reduced by  $w_n$ . Then our profit will be  $p_n$ , plus the maximum we can get from solving the subproblem of  $n-1$  objects and capacity of  $C - w_n$ .
  - If we do not include it in the knapsack, our profit will be the maximum we can get from solving the subproblem of  $n-1$  objects and capacity of  $C$ .
  - The dynamic programming selection rule: **when given a number of possibilities, compute all and take the best.**

For example, a knapsack of capacity 10 and 4 objects



Step 2: Turn this formulation into a recursive function to solve the 0/1 knapsack problem

Let  $P(C, j)$  be the maximum profit that can be made by selecting a subset of the  $j$  objects with knapsack capacity of  $C$ .

$$P(C, 0) = P(0, j) = 0$$

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$

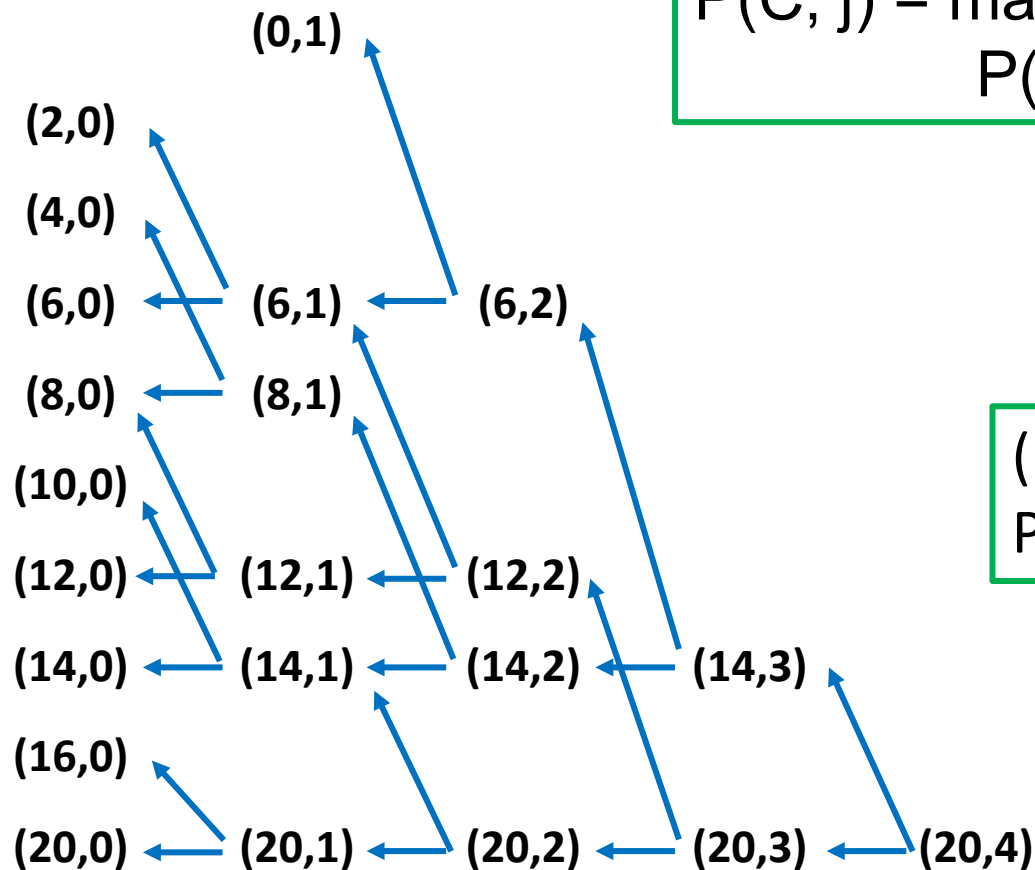
- Step 3: Draw the subproblem graph and find the dependencies among subproblems

For example,  $C = 20$

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$



$(i, j)$  represents  $P(i, j)$

## Step 4 :Dictionary: profit[C+1][n+1]

	0	1	2	...	n
0					
1					
2					
3					
4					
5					
6					
7					
...					
C					

```
int knapsack(int [] w, int [] p, int C, int n)
```

Step 5

```
{ for c = 0 to n    profit[0][c] = 0;
```

```
  for r = 1 to C    profit[r][0] = 0;
```

```
  for r = 1 to C
```

```
    for c = 1 to n
```

```
      profit[r][c] = profit[r][c-1];
```

```
      if (w[c] <= r)
```

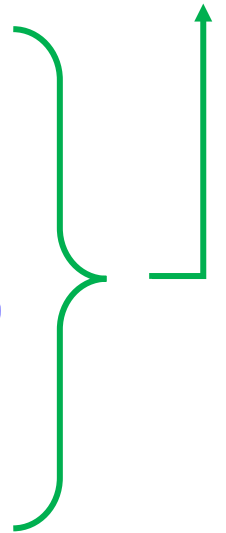
```
        if (profit[r][c] < profit[r-w[c]][c-1] + p[c])
```

```
          profit[r][c] = profit[r-w[c]][c-1] + p[c];
```

```
}
```

**Complexity :**  
 **$O(nC)$**

$$P(C, j) = \max(P(C, j-1), p_j + P(C-w_j, j-1))$$



Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	0	1	2	3	4
0	0	0	0	0	0
2	0				
4	0				
6	0				
8	0				
10	0				
12	0				
14	0				
16	0				
20	0				

for c = 0 to n  
 profit[0][c] = 0;  
 for r = 1 to C  
 profit[r][0] = 0;

Not all rows  
are shown

Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

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

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] <

profit[r-w[c]][c-1] + p[c])

profit[r][c] =

profit[r-w[c]][c-1]

+ p[c]



Example 1:  
C = 20

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	0	1	2	3	4
10	0	7	13	13	13
11	0	7	13	13	13
12	0	7	13	16	16
13	0	7	13	16	16
14	0	7	13	16	16
15	0	7	13	16	16
16	0	7	13	16	18
17	0	7	13	16	18
18	0	7	13	22	22
19	0	7	13	22	22
20	0	7	13	22	<b>22</b>

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] <

profit[r-w[c]][c-1] + p[c])

profit[r][c] =

profit[r-w[c]][c-1]

+ p[c]

Example 2:  
C = 3

	1	2	3
$w_i$	1	2	3
$p_i$	1	4	6

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

```

for c = 0 to n
    profit[0][c] = 0;
for r = 1 to C
    profit[r][0] = 0;

```

Example 2:  
C = 3

	1	2	3
$w_i$	1	2	3
$p_i$	1	4	6

	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	4	4
3	0	1	5	<b>6</b>

for r = 1 to C

for c = 1 to n

profit[r][c] = profit[r][c-1];

if (w[c] <= r)

if (profit[r][c] <

profit[r-w[c]][c-1] + p[c])

profit[r][c] =

profit[r-w[c]][c-1]

+ p[c]

- The dynamic programming algorithm has a complexity of  $O(nC)$ .
- An algorithm is polynomial time if it is a polynomial function of the size of the input.  
E.g. there are  $n$  weight numbers and  $n$  profit numbers
- An algorithm is pseudo-polynomial time if it is a polynomial function of the value of the input.  
E.g. there is only one number specifying  $C$
- So the dynamic programming algorithm for knapsack problem is pseudo-polynomial.

# String Matching

Huang Shell Ying

**References:** Introduction to Algorithms. Cormen, T.H., C.E. Leiserson. R.L. Rivest, Chapter 34  
Computer Algorithms. Sara Baase & Allen Van Gelder, Chapter 11

The problem: Given a text  $T$  of  $n$  characters and a pattern  $P$  of  $m$  characters, find the first occurrence of  $P$  in  $T$ .

We may be looking for

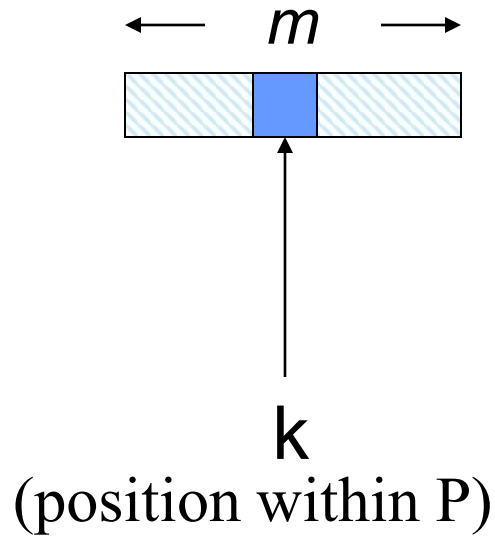
- A character string in text;
- A pattern in DNA sequences;
- A piece of coded information representing graphical, audio data, or machine code;
- A sublist in linked list.....

We will study ----

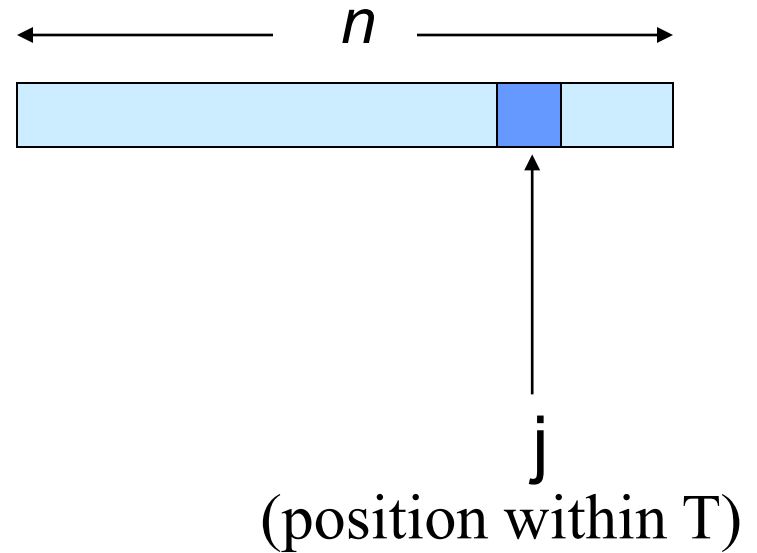
- A straightforward solution
- The Rabin-Karp Algorithm
- The Boyer-Moore Algorithm

Conventions used:

**P = PATTERN**



**T = TEXT**

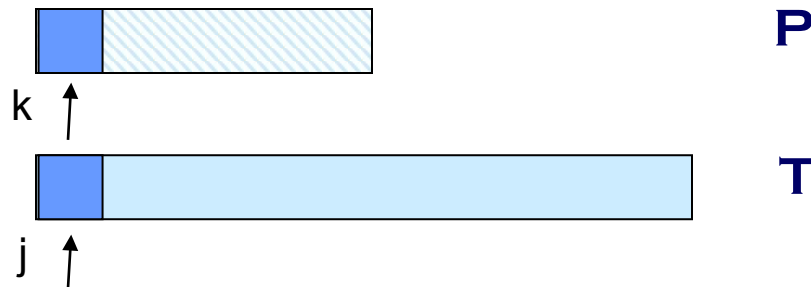


# A straightforward solution

```
int SimpleScan (char [] P, char [] T, int m)
{
    int i, j, k;

    // i is the current guess of where P begins in T;
    // j is the index of the current character in T;
    // k is the index of the current character in P;

    j = k = 0;
    i = 0;
```



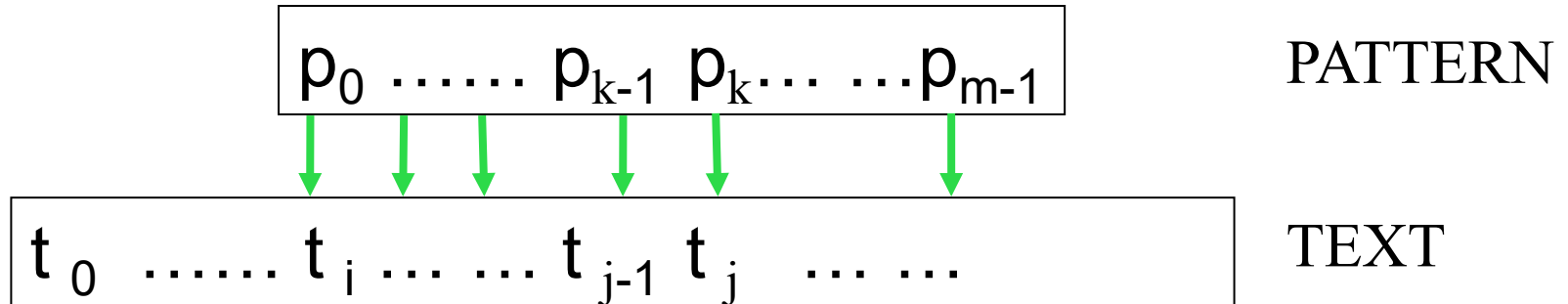


```

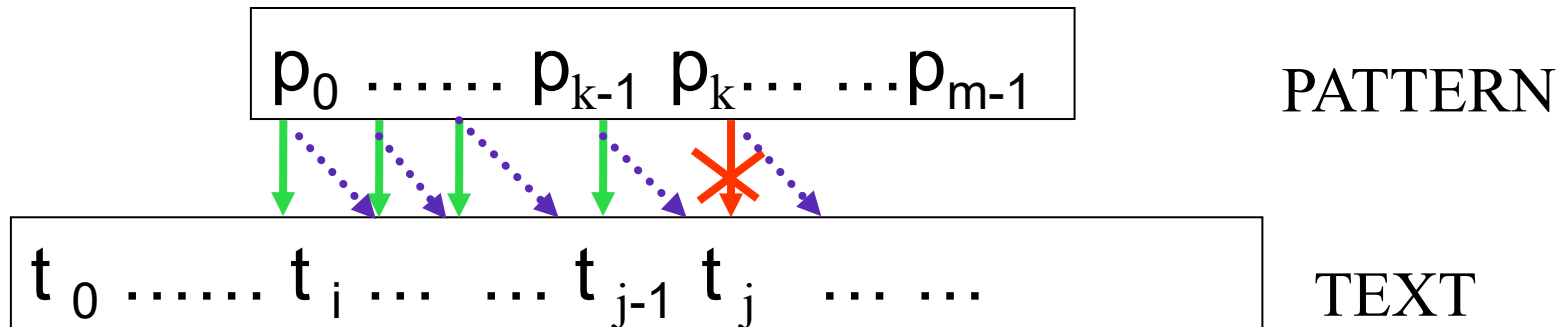
while (j < n) {
    if (T[j] == P[k]) {
        j++;
        k++;
        if (k == m)    return i; }
    else {
        j = ++i;
        k = 0;
        if (j > n-m)    break;    }

    return -1;
}

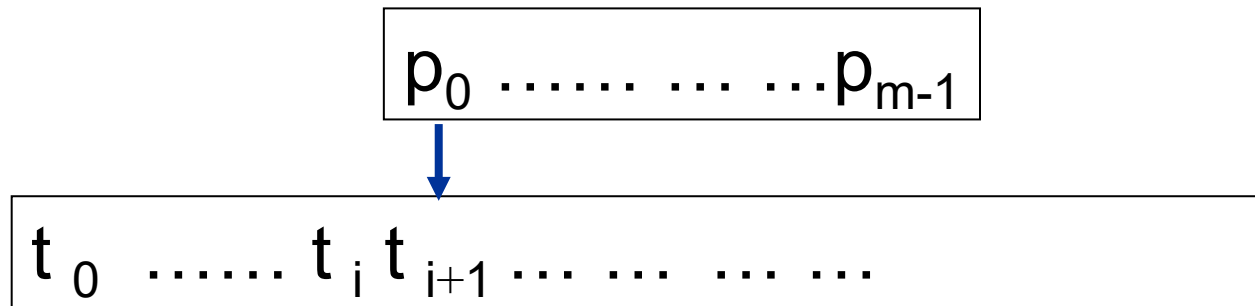
```



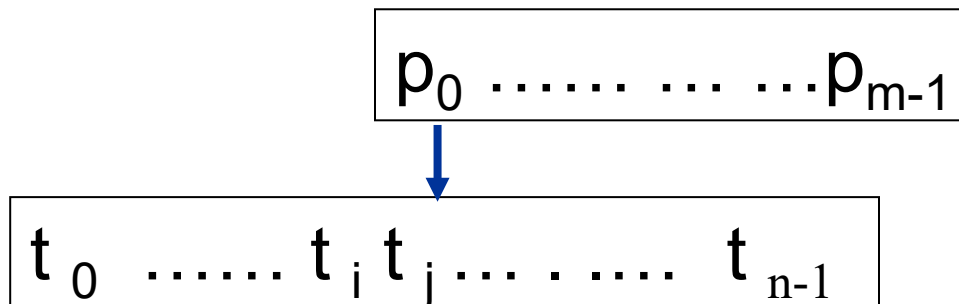
- Comparison starts with  $k=0$  and  $j=i$ . After a match between  $P[k]$  and  $T[j]$ ,  $j$  and  $k$  are incremented.
- When  $k$  reaches  $m$ , all characters have been compared and matched. Return  $i$ .



When a mismatch happens, shift the pattern right one position:  
 $j = j + 1$ ,  $k = 0$ .



After shifting, if  $n - j < m$ , there is not enough text left for comparison, then stop comparisons with no match.



# Example

P =

**ABABC**



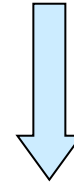
T =

**ABABA**BCCAC

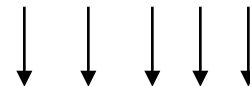
**A**BABC



**A**BABABCCAC



**ABABC**




**ABABAB**CCAC

**Match Successful!**

# Worst case

P = **AAAAC**

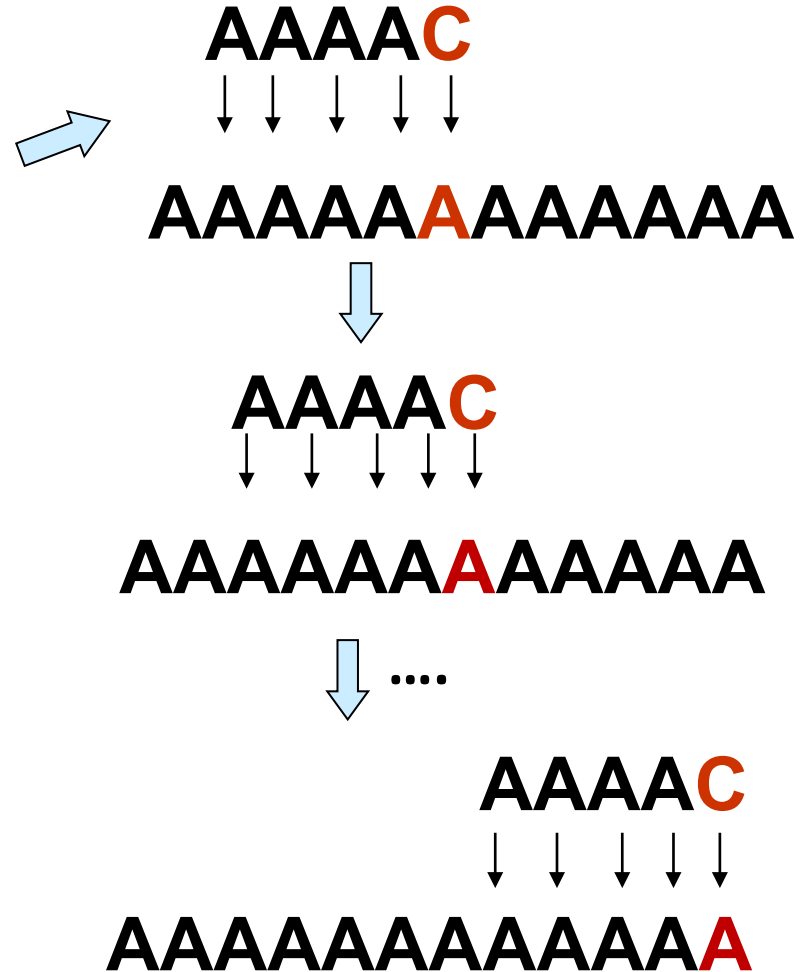


T = **AAAAA**AAAAAAAAA

From the 1<sup>st</sup> character to the 5<sup>th</sup> last character of the text T, 5 comparisons are done before a mismatch.

Total is  $m(n-m+1)$  comparisons

Worst case complexity is  $O(mn)$  where  $m$  is the length of the pattern and  $n$  is the length of the text



# The Rabin-Karp Algorithm

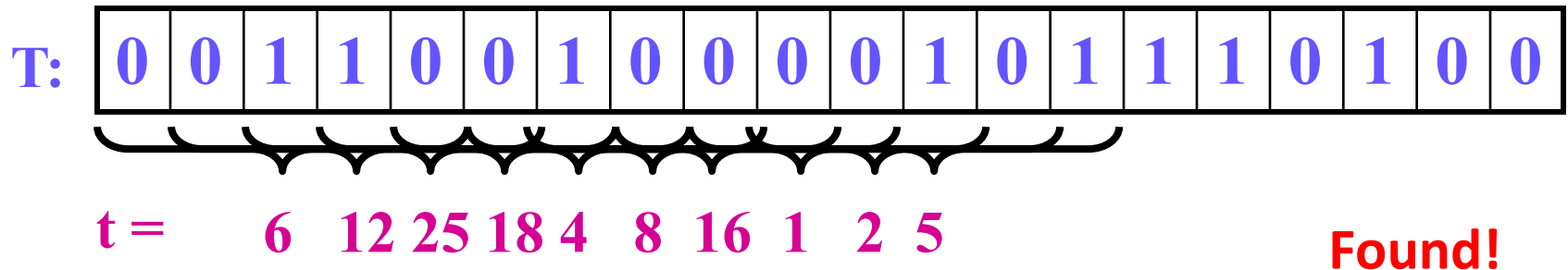
- Outline of the steps of Rabin-Karp Algorithm
  - 1) Convert the pattern (length  $m$ ) to a number,  $p$
  - 2) Convert the first  $m$ -characters (the first text window) to a number,  $t$
  - 3) If  $p$  and  $t$  are equal, pattern found and exit
  - 4) If not end-of-text, shift the text window one character right and convert the string in it to a number  $t$ , go to step 3); else pattern not found and exit

**P:**

0	0	1	0	1
---	---	---	---	---

 $m = 5, p = 5$

$p = 5$



To compute the number for the pattern and the number for the first  $m$ -character text window,

- The set of possible characters is referred to as an alphabet and denoted with sigma  $\Sigma$ . e.g.  
 $\Sigma = \{0, 1\}$  or  $\Sigma = \{0, 1, 2, \dots, 9\}$   
or  $\Sigma = \{a, b, c, \dots, z\}$
- Let  $d = |\Sigma|$

- The number  $p$  of the pattern and the number  $t$  of the first  $m$ -character text window, are calculated iteratively.

For example,  $P = \text{"36215"}\text{"}$ ,  $d = 10$

3	6	2	1	5
---	---	---	---	---

$$\begin{aligned}
 p &= 3 * 10^4 + 6 * 10^3 + 2 * 10^2 + 1 * 10^1 + 5 \\
 &= (3 * 10^3 + 6 * 10^2 + 2 * 10^1 + 1) * 10 + 5 \\
 &= ((3 * 10^2 + 6 * 10^1 + 2) * 10 + 1) * 10 + 5 \\
 &= (((3 * 10 + 6) * 10 + 2) * 10 + 1) * 10 + 5
 \end{aligned}$$

- $$p = P[0]*d^{(m-1)} + P[1]*d^{(m-2)} + \dots + P[m-2]*d + P[m-1]$$

$$= (((P[0]*d + P[1])*d + P[2])*d + \dots P[m-2])*d + P[m-1]$$

**$p = P[0];$**   
**For  $j = 1$  to  $m-1$**   
      **$p = p*d + P[j]$**

The numbers  $p$  and  $t$  can be computed in  $\theta(m)$  time.



- To compute the number  $t$  after shifting the text window, it can be done in constant time based on the number of the previous text window

For example:

36415

3	6	4	1	5	2
---	---	---	---	---	---

$i$

$$(36415 - 3 * 10^4) * 10 + 2$$

In general,  

$$\text{new} = (\text{old} - \text{MSB} * d^{m-1}) * d + \text{LSB}$$
 $d^{m-1}$  is pre-calculated as below

**$dM = 1;$**   
**For  $j = 1$  to  $m-1$**   
      **$dM = dM * d$**   
**//  $dM = d^{m-1}$**

**$t = (t - T[i] * dM) * d + T[i+m]$**   
**//  $t$  before this is the number for  $T[i .. i+m-1]$**   
**//  $t$  after this is the number for  $T[i+1 .. i+m]$**

- If the pattern is long (e.g.  $m = 100$ ), then the resulting number will be enormous. Overflow may occur.
  - For this reason, we hash the value by taking it **mod a prime number  $q$** . This prime number should be large.
- 1) **Hash** the pattern to a number,  $hp$
  - 2) **Hash** the first  $m$ -character text window to a number,  $ht$
  - 3) If  $hp$  and  $ht$  are equal, compare the pattern with the text window. If equal, pattern found and exit
  - 4) If not end-of-text, shift the text window one character right and **(re)hash** it to a number  $ht$ , go to step 3); else pattern not found and exit

Note: if  $hp = ht$ , it does not necessarily imply that  $T[i..i+m-1] = P[0..m-1]$ .

However, if  $hp \neq ht$ , definitely  $T[i..i+m-1] \neq P[0..m-1]$

P: 

0	0	1	0	1
---	---	---	---	---

$m = 5, q = 13, hp = 5$

T: 

0	0	1	1	0	0	1	0	0	0	0	1	0	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



$ht = 6$

4 8 3 1 2

$ht = 12$

$ht = 25\%13=12$

$ht = 18\%13=5$  compare P and T

$ht = 5$  compare P and T

**Found!**

- The **mod** function (% in Java) is particularly useful in this case due to several of its inherent properties:
  - $(x+y) \bmod q = [(x \bmod q) + (y \bmod q)] \bmod q$
  - $(x \bmod q) \bmod q = x \bmod q$
  - $xy \bmod q = [(x \bmod q)(y \bmod q)] \bmod q$

Example:

$$(3*10+6) \bmod 13$$

$$= ((3*10) \bmod 13 + 6 \bmod 13) \bmod 13$$

$$= ( ( (3 \bmod 13) * (10 \bmod 13) ) \bmod 13 + 6 \bmod 13) \bmod 13$$

$$= (4 + 6) \bmod 13$$

$$= 10$$

- To calculate  $hp$ , the hash value for  $P[0..m-1]$ , call  $hash(P, m, base)$ . The hash function is also used to compute the value of the first text window

```
int hash(Txt, m, d)
{
    int h = Txt[0] % q;
    for (int i = 1; i < m; i++)
        h = (h * d + Txt[i]) % q;
    return h;
}
```

E.g.  $P = \text{"36415"} , q = 7$

$h = 3$

$i = 1, h = 1$

$i = 2, h = 0$

$i = 3, h = 1$

$i = 4, h = 1$

$hash(\text{"36415"}, 5, 10) = 1$

Ref:

```
p = P[0];
For j = 1 to m-1
    p = p*d + P[j]
```

The numbers  $hp$  and  $ht$  can be computed in  $\theta(m)$  time.

- After finding ht for  $T[i \dots m-1]$ , ht for  $T[i+1 \dots m]$  can be calculated by  $\text{rehash}(T, i, m, \text{ht})$  in constant time  $\theta(1)$ .

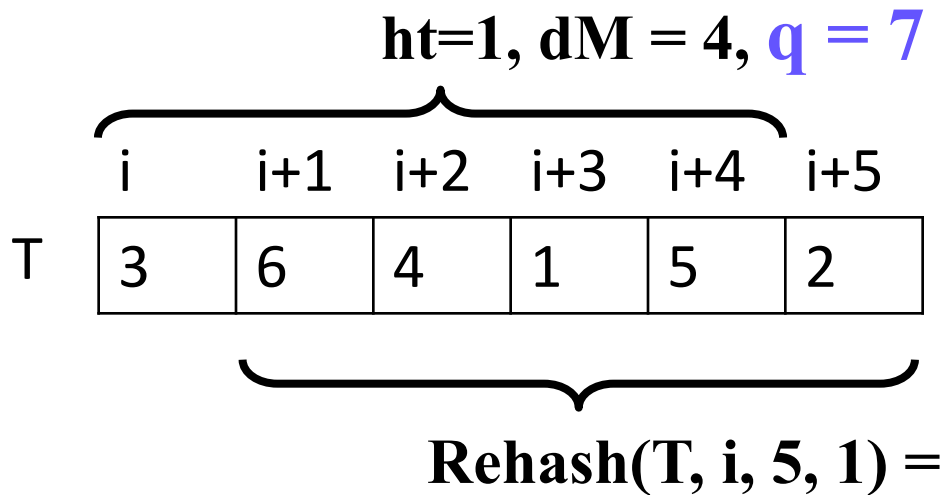
```
int rehash(T, i, m, ht)
{
    oldest = (T[i] * dM) % q;    // dM is  $d^{m-1} \bmod q$ 
    oldest_removed = ((ht + q) - oldest) % q;
    return (oldest_removed * d + T[i+m]) % q;
}
```

- Compare with no hashing:

```
t = (t - T[i]*dM)*d + T[i+m]
// dM is  $d^{m-1}$ 
// t before this is the number for  $T[i \dots i+m-1]$ 
// t after this is the number for  $T[i+1 \dots i+m]$ 
```

# Example

```
int rehash(T, i, m, ht)
{
    oldest = (T[i] * dM) % q;
    oldest_removed = ((ht + q) - oldest) % q;
    return (oldest_removed * d + T[i+m]) % q;
}
```



```
// computed once
dM = 1;
For j = 1 to m-1
    dM = dM*d % q
```

Oldest = 5  
Oldest\_removed = 3  
Return  $32 \% 7 = 4$

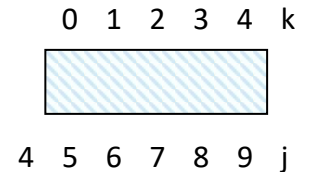
```

int RKscan(P, T)
{
    m = Length(P);
    n = Length(T);
    dM = 1;
    For j = 1 to m-1    dM = dM*d % q;    // d = |Σ|

    hp = hash(P, m, d);    // θ(m)
    ht = hash(T, m, d);

    for (j = 0; j <= n - m; j++) {
        if (hp == ht && equal_string(P, T, 0, j, m))
            return j;
        if (j < n-m) ht = rehash(T, j, m, ht);
    }
    return -1; // pattern not found
}

```



**Maximum  
n-m+1  
iterations**



- The running time of Rabin-Karp algorithm in the worst case is  $\theta((n - m + 1)m)$
- However, in many applications, the expected running time is  $O(n+m)$  plus the time required to process spurious hits.
  - $O(m)$  time for the 2 hash() calls
  - Close to  $O(n)$  time on the for loop
- The number of spurious hits can be kept low by using a large prime number  $q$  for the hash functions

# The Boyer-Moore Algorithm

- It is a very efficient algorithm for string searching
- The text being scanned is  $T$  with  $n$  characters
- The pattern we are looking for is  $P$  with  $m$  characters
- Process the text  $T[1..n]$  from left to right
- Scan the pattern  $P[1..m]$  **from right to left**
- Preprocessing to generate two tables based on which to slide the pattern as much as possible after a mismatch
- It performs even better with long patterns

```
int BMscan(char[]P char[]T, int m,  
           int[]charJump, int []matchJump )
```

```
{ int j;  int k;
```

```
  j = m; k = m;
```

```
  while (j <= n) {
```

```
    if (k < 1) return j + 1;  //match found
```

```
    if (T[j] == P[k])  {  j--; k--; }
```

```
    else {  j += max(charJump[T[j]], matchJump[k]);
```

```
           k = m;  }
```

```
}
```

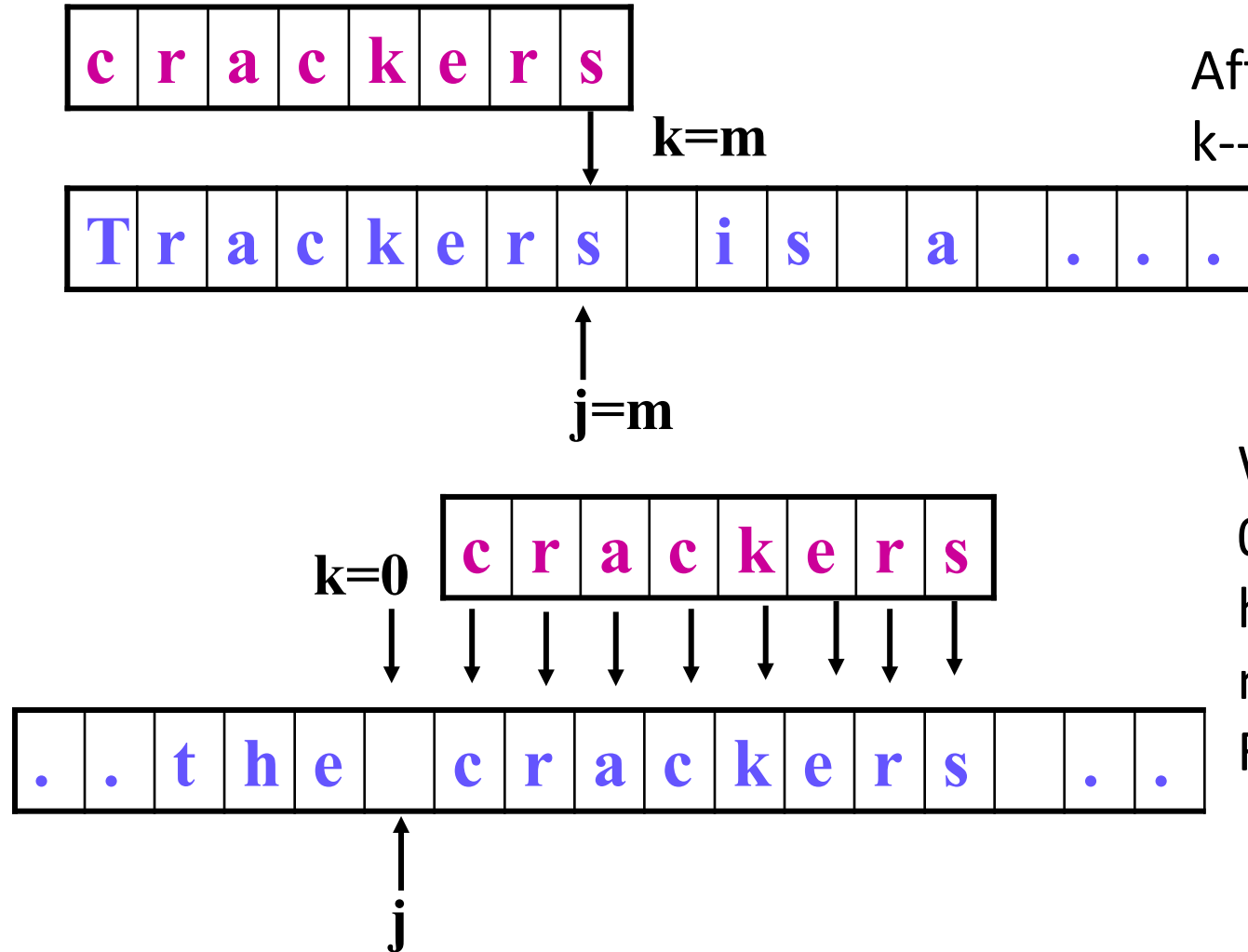
```
return -1;  // match not found
```

```
}
```

charJump and  
matchJump are the  
2 tables generated  
in a preprocessing  
step

Assume the 1<sup>st</sup>  
character is at P[1] and  
T[1] respectively

## Examples



To start.

$k = m, j = m$

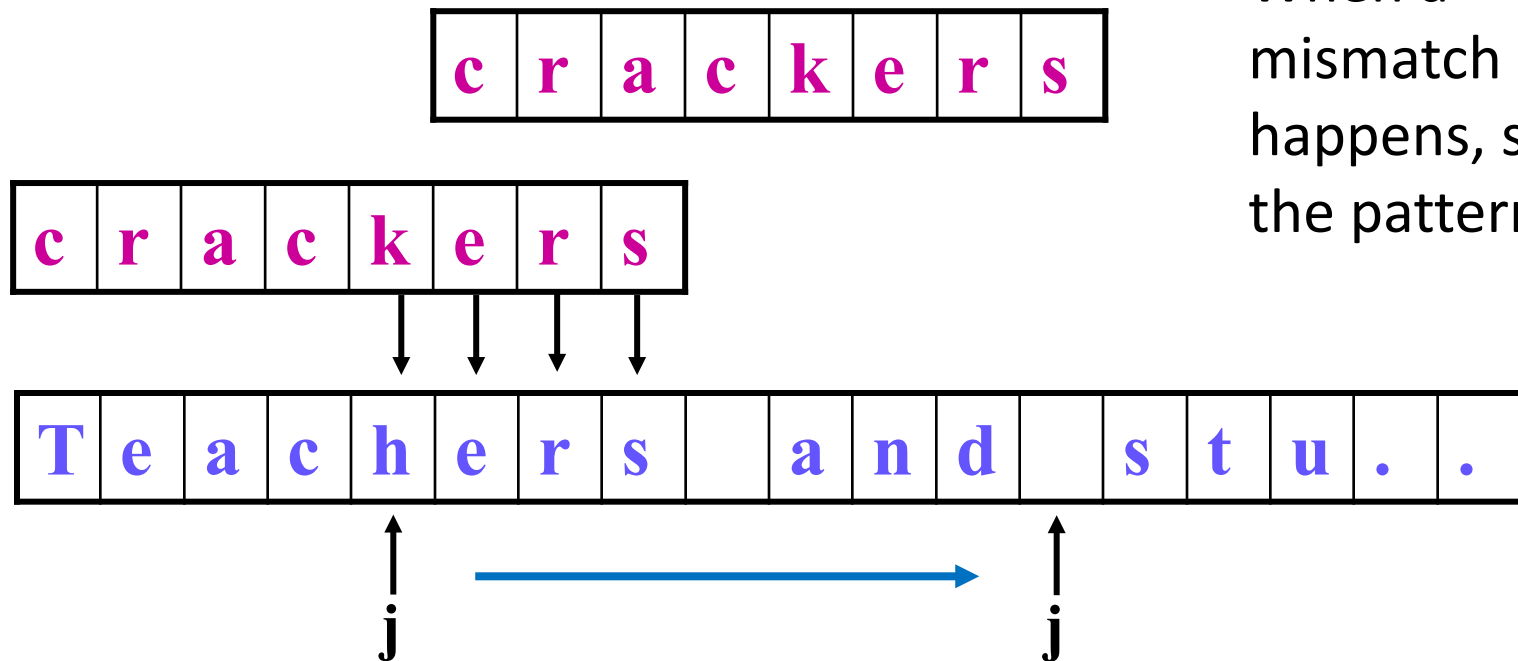
After a match,

$k--, j--$

When  $k$  reaches 0, all characters have been matched:

Return  $j+1$

## Example

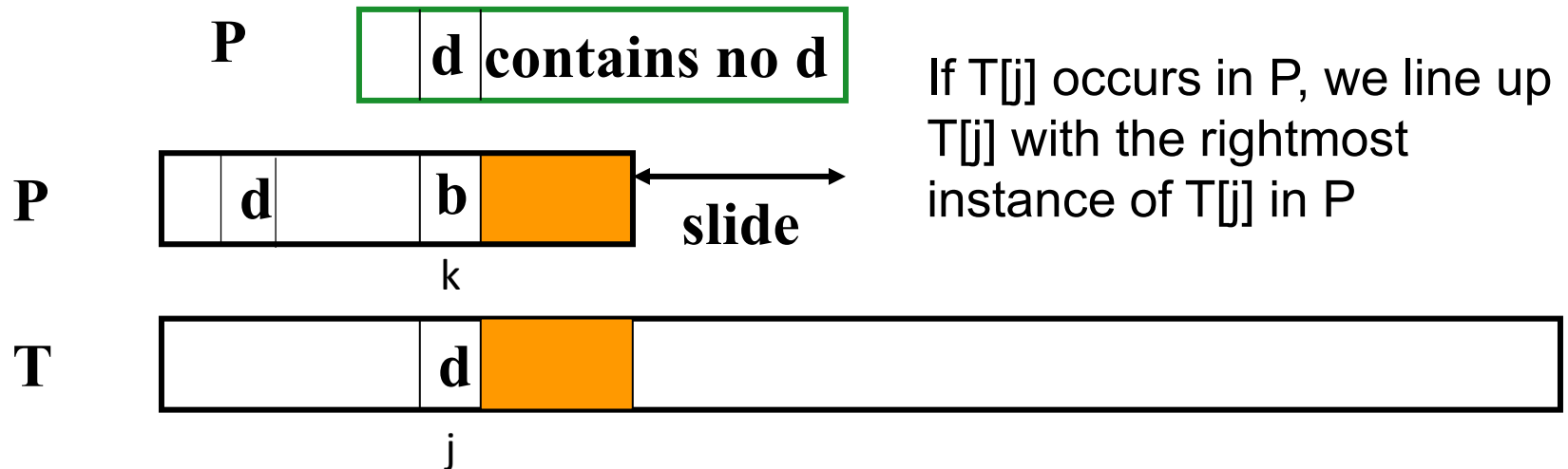
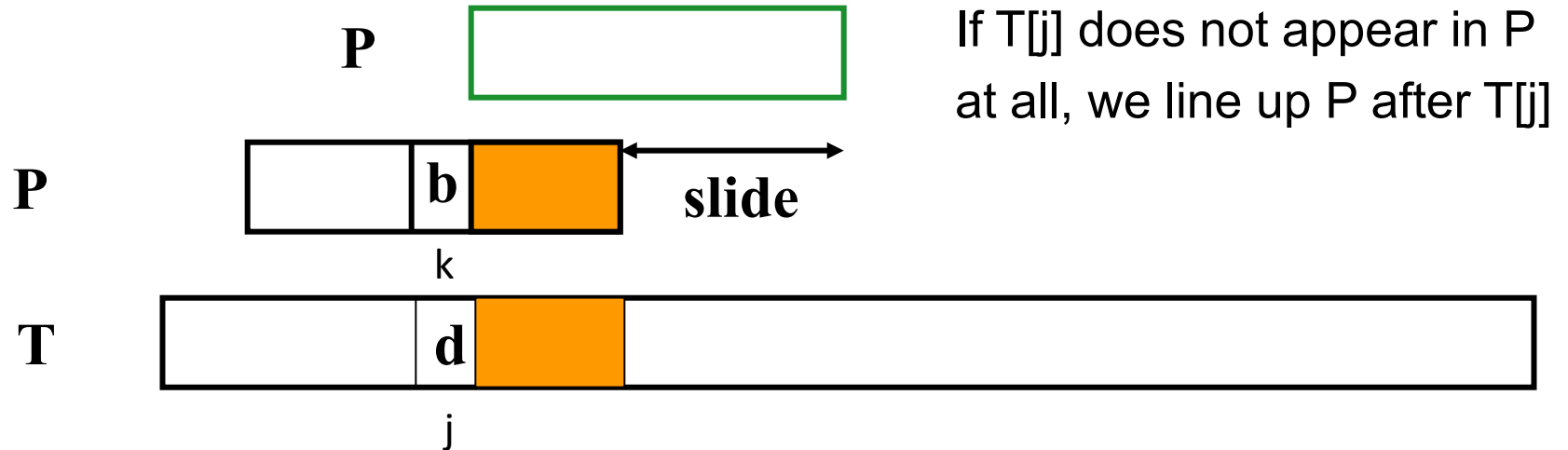


When a mismatch happens, shift the pattern

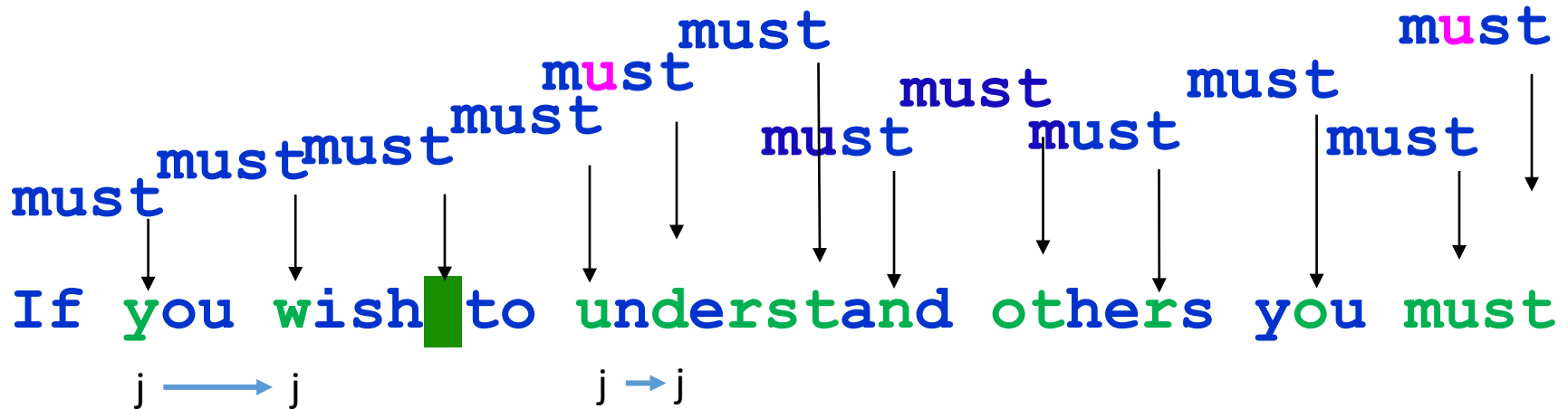
Shift the pattern as much as possible – increment  $j$  as much as possible for the next comparison:

```
{  j += max(charJump[T[j]], matchJump[k]);  
    k = m;  }
```

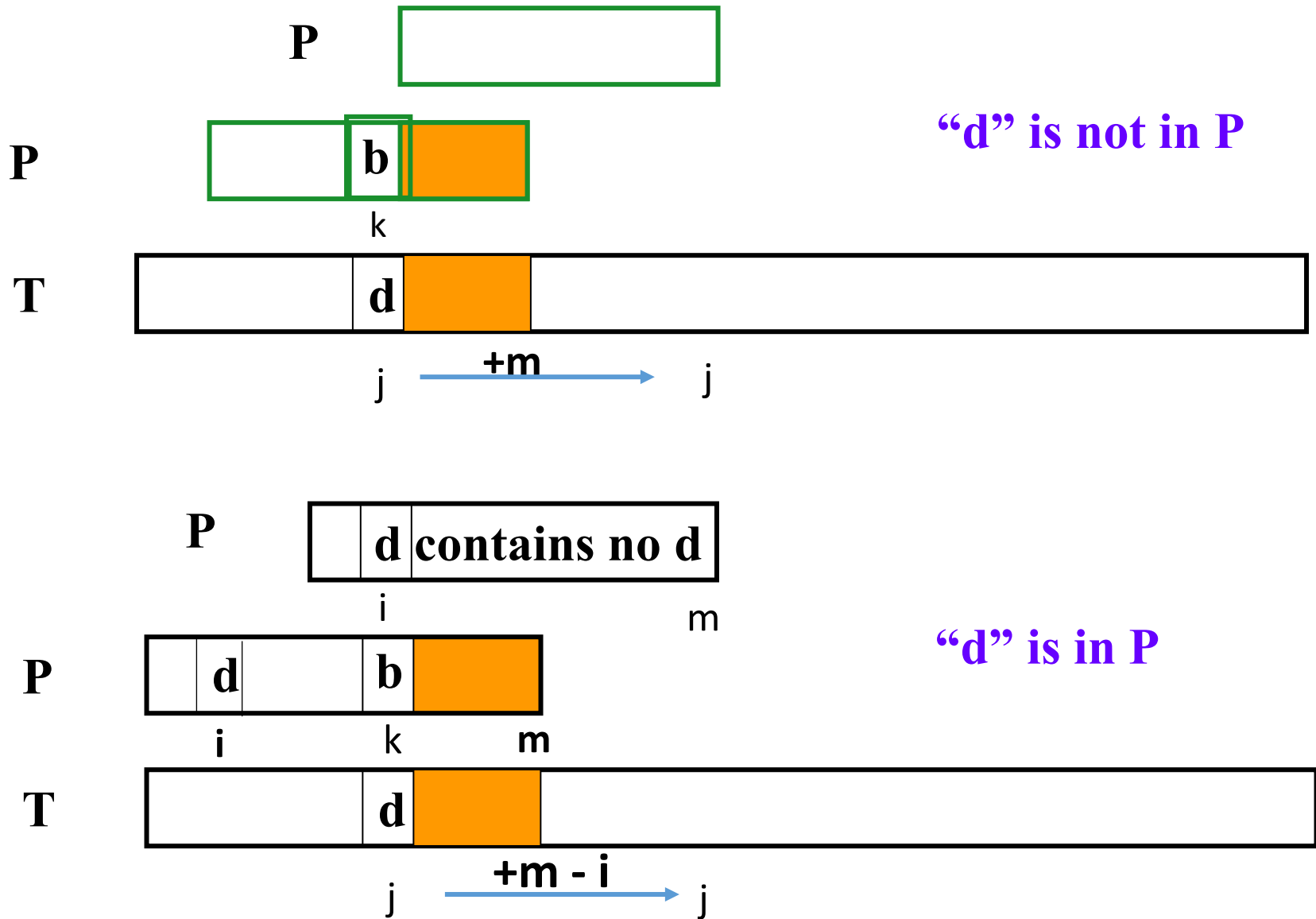
## Preprocessing to compute charJump



## Example

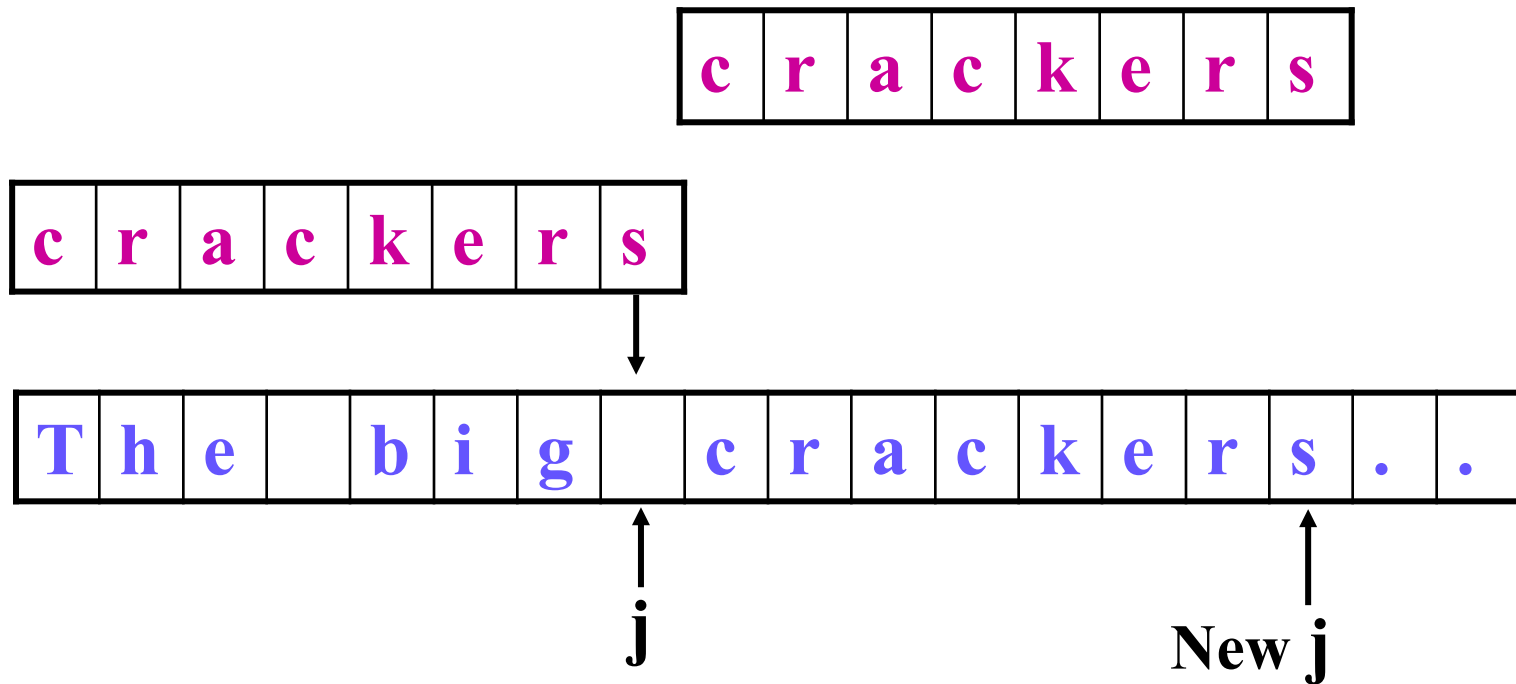


- Many of the  $n$  characters in the text are never compared – sublinear complexity
- We need to calculate how the text index  $j$  should be incremented to begin the next right-to-left scan of the pattern



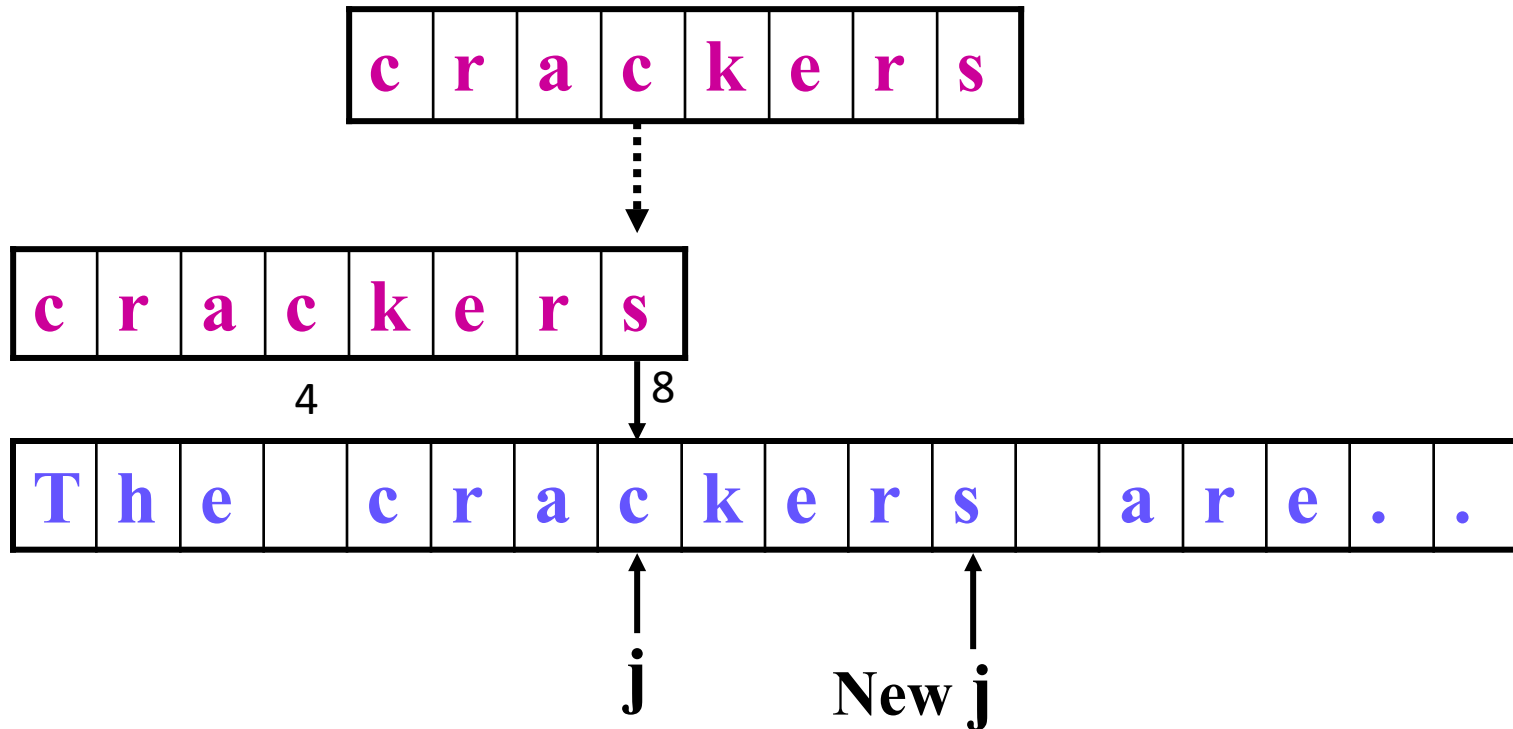


## Example



To line up P after  $T[j]$ , e.g. ' ', P is slid 8 places to the right:  
 $j = j + 8$

## Example



To line up  $T[j]$ , e.g. 'c', with the rightmost 'c' in  $P$ ,  $P$  is slid 4 places to the right:  $j = j + 8 - 4$

- Computing the jumps for all the characters:

```
void computeJumps(char [] P, int m,  
                 int alpha, int [] charJump)  
{ char ch; int k;  
  for (ch = 0; ch < alpha; ch++)  
    charJump[ch] = m;  
  for (k = 1; k <= m; k++)  
    charJump[ P[k] ] = m - k;  
}
```

Number of  
characters in  
character set

Position  
from the end

Complexity is  
 $O(|\Sigma| + m)$

Notice that if a character appears more than once, we take the right-most occurrence.

c	r	a	c	k	e	r	s
---	---	---	---	---	---	---	---

First:  $\text{charJump}['c'] = 8-1$

Then:  $\text{charJump}['c'] = 8-4$

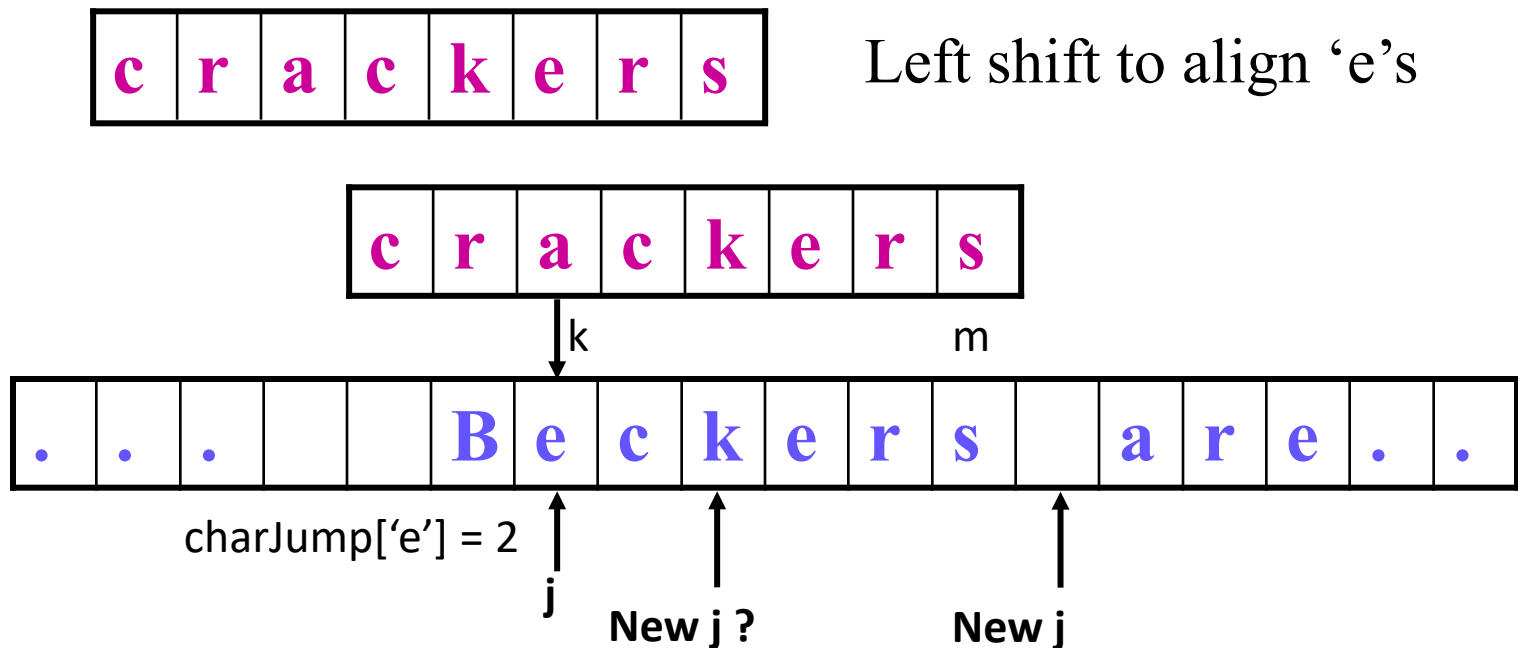
E.g. for

r	a	t	s	a	n	d	c	a	t	s
1	2	3	4	5	6	7	8	9	10	11

$m = 11,$

2	11	3	4	...	5	...	10	0	1	...
a	b	c	d	...	n	...	r	s	t	...

Sometimes this heuristic fails, for example,



## Simplified Boyer-Moore (using charJump only)

```
int simpleBMscan(char[]P char[]T, int m, int[]charJump)
{ int j; int k;
  j = m; k = m;
  while (j <= n) {
    if (k < 1) return j + 1; //match found
    if (T[j] == P[k]) { j--; k--; }
    else { j += max(charJump[T[j]], m-k+1);
           k = m; }
  }
  return -1; // match not found
}
```

E.g.

c	r	a	c	k	e	r	s
---	---	---	---	---	---	---	---

charJump

5	8	4	8	2	...	3	...	1	0	...
a	b	c	d	e	...	k	...	r	s	...

c	r	a	c	k	e	r	s
---	---	---	---	---	---	---	---

Shift  $m-k+1$  places

c	r	a	c	k	e	r	s
---	---	---	---	---	---	---	---

↓  $k=3$

.	.	.			B	e	c	k	e	r	s		a	r	e	.	.
---	---	---	--	--	---	---	---	---	---	---	---	--	---	---	---	---	---

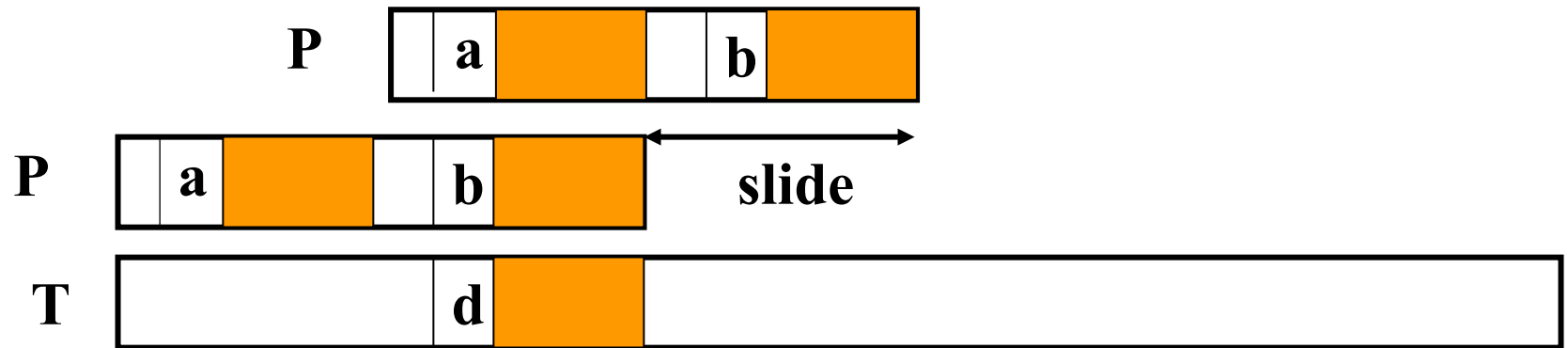
↑  $j$

↑  $\text{New } j = j + 8 - 3 + 1$

## Preprocessing to compute matchJump

This heuristic tries to derive the maximum shift from the structure of the pattern. It is defined for each of the characters in P.

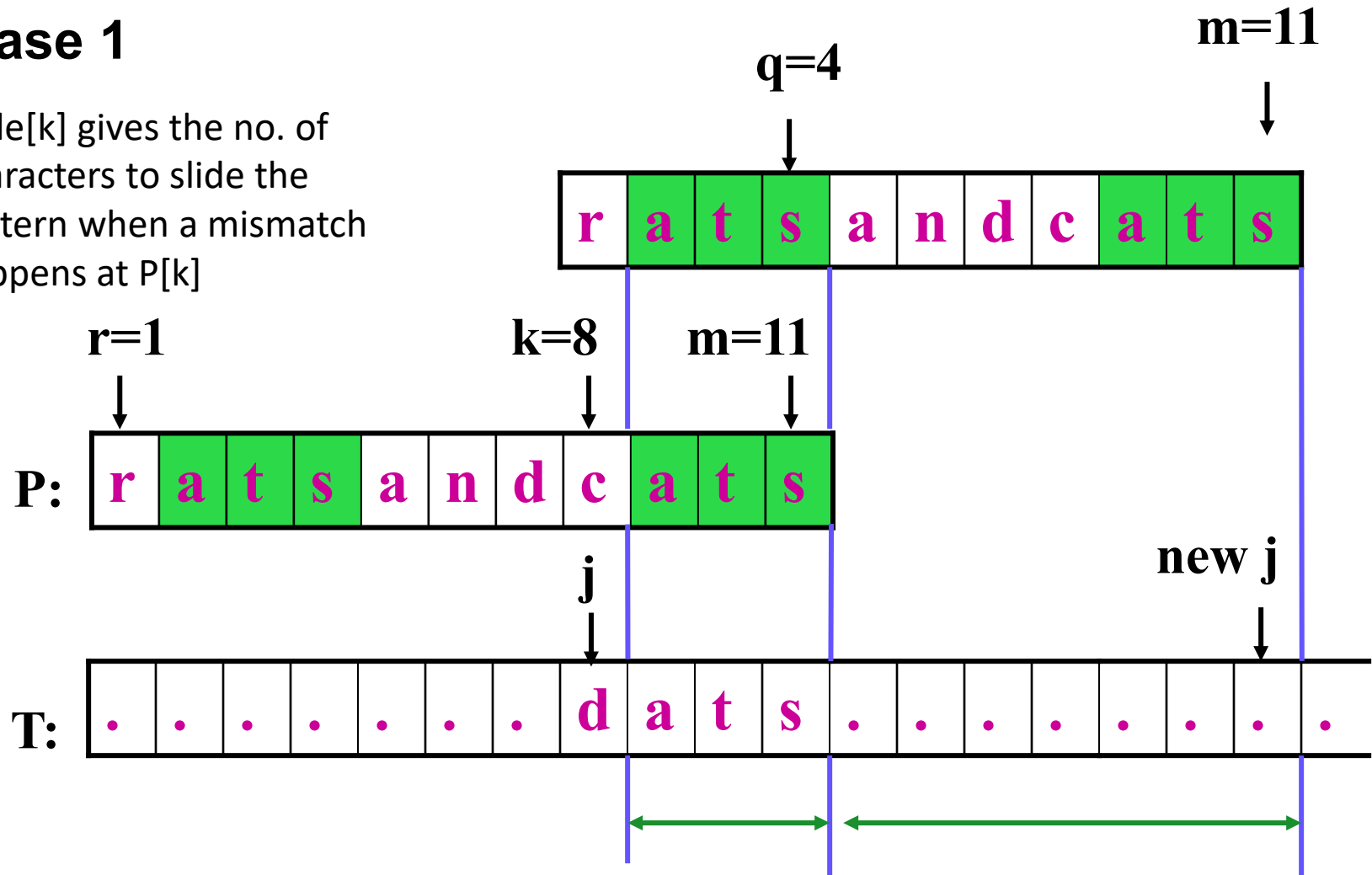
**Case 1:** The matching suffix occurs earlier in the pattern, but preceded by a different character



We line up the earlier occurrence of the suffix in P with the matched substring in T

## Case 1

Slide[k] gives the no. of characters to slide the pattern when a mismatch happens at P[k]

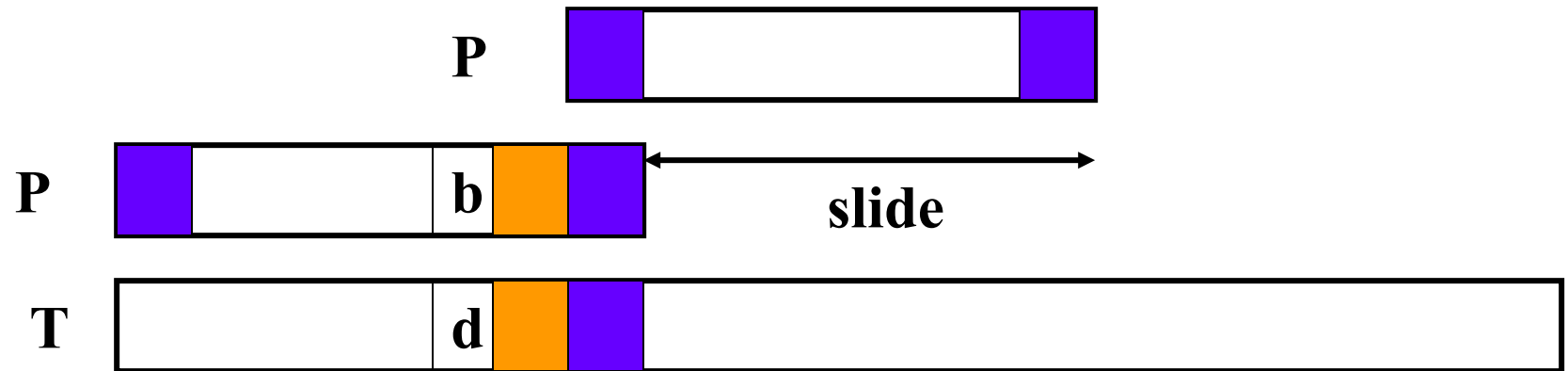


$$\text{matchJump}[k] = m - k + \text{Slide}[k]$$

$$\text{Slide}[k] = m - q \quad (P[r] \neq P[k])$$



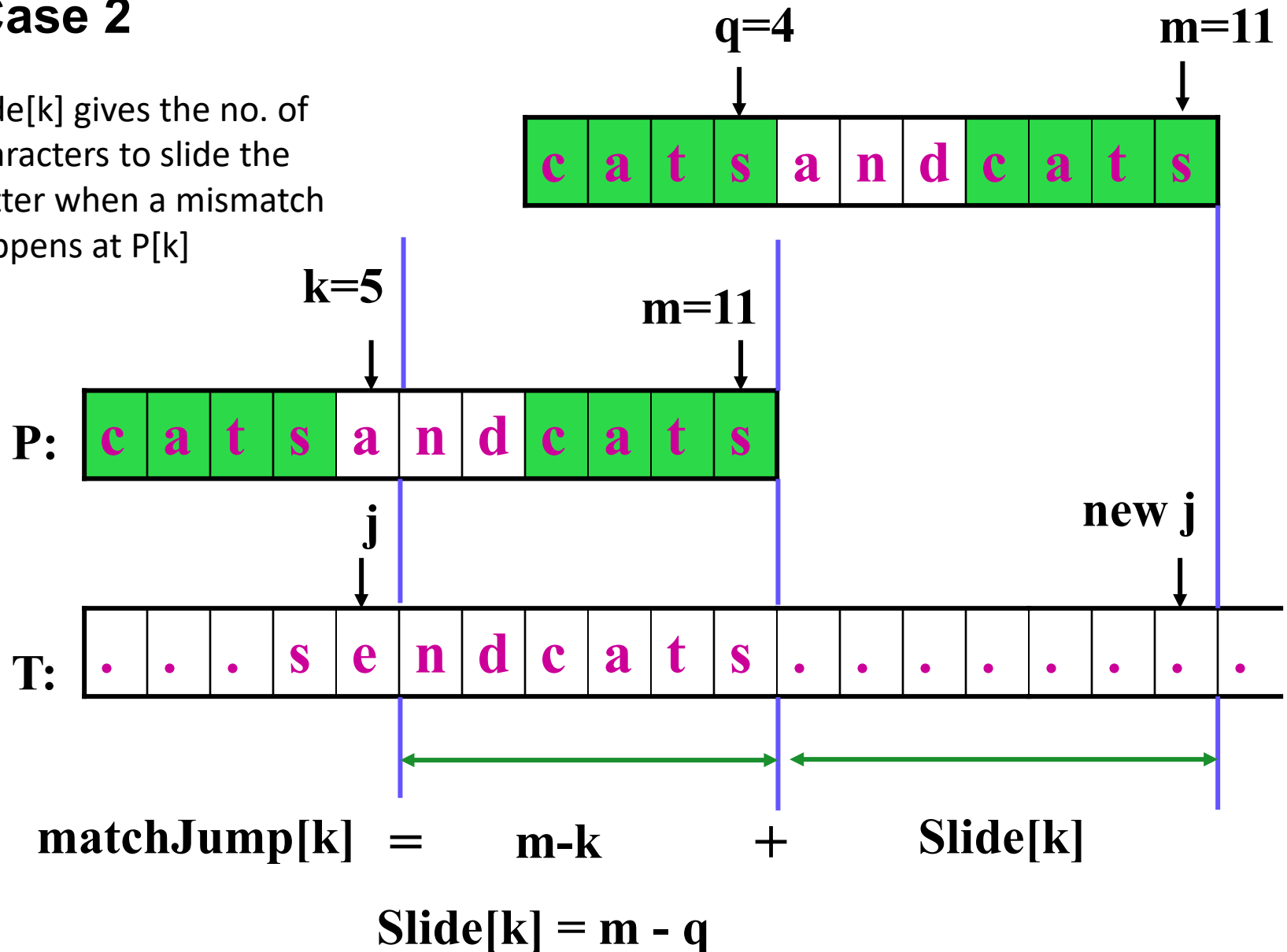
**Case 2:** Only part of the matching suffix occurs at the beginning of the pattern (a prefix).



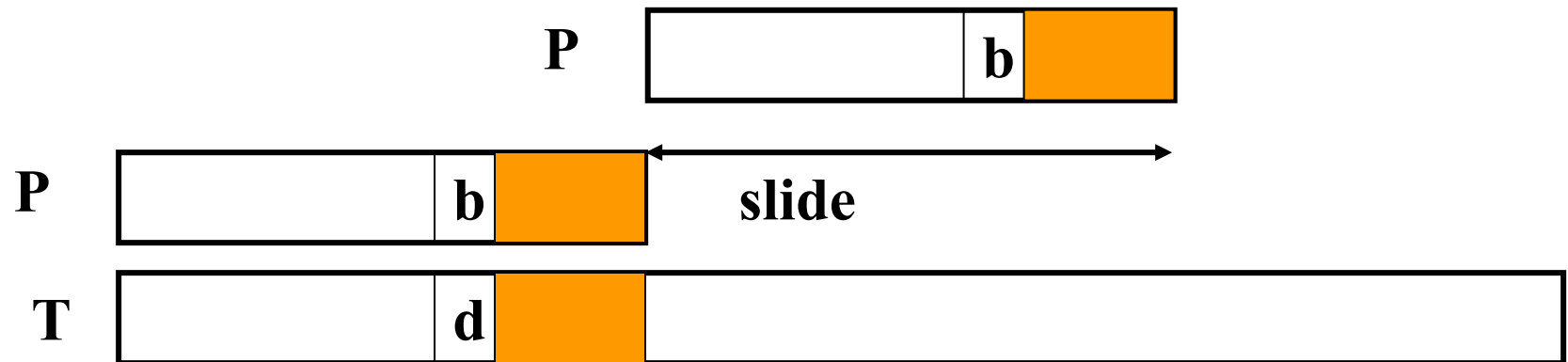
We line up the prefix in **P** with part of the matched substring in **T**

## Case 2

Slide[k] gives the no. of characters to slide the patten when a mismatch happens at P[k]



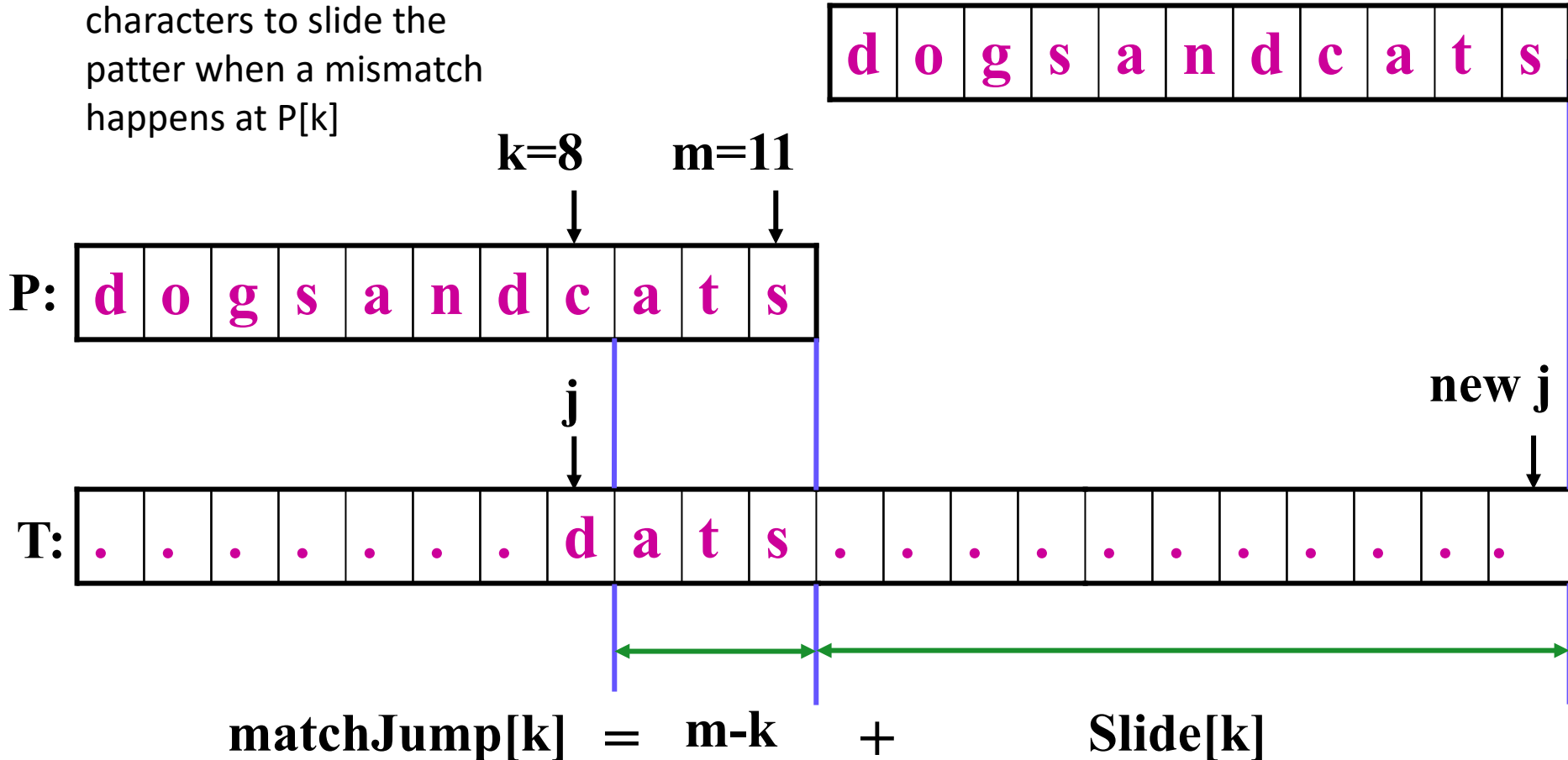
**Case 3:** There is no other occurrence of the matching suffix in the pattern. (Case 1 and Case 2 do not happen)



We line up P after the matched substring in T

## Case 3

Slide[k] gives the no. of characters to slide the patten when a mismatch happens at P[k]



$$\text{Slide}[k] = m \quad (= m - q \text{ where } q \text{ is } 0)$$

# Example: computing matchJump

Slide[m] = 1

P: 

w	o	w	w	o	w
---	---	---	---	---	---

T: 


Matched = 0 (m-k)

Slide[6] = 1

matchJump[6] = 1

P: 

w	o	w	w	o	w
---	---	---	---	---	---

T: 


Matched = 1 (m-k)

Slide[5] = 2 (m-q)

matchJump[5] = 3

P: 

w	o	w	w	o	w
---	---	---	---	---	---

P: 

w	o	w	w	o	w
---	---	---	---	---	---

T: 

					X

P: 

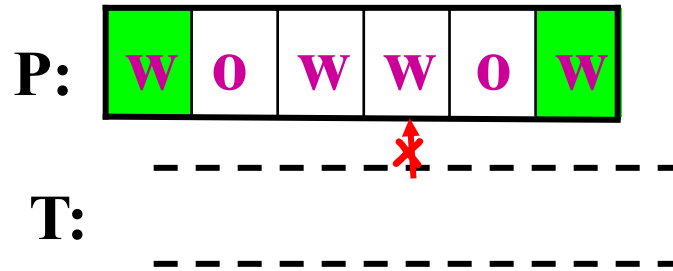
w	o	w	w	o	w
---	---	---	---	---	---

P: 

w	o	w	w	o	w
---	---	---	---	---	---

T: 

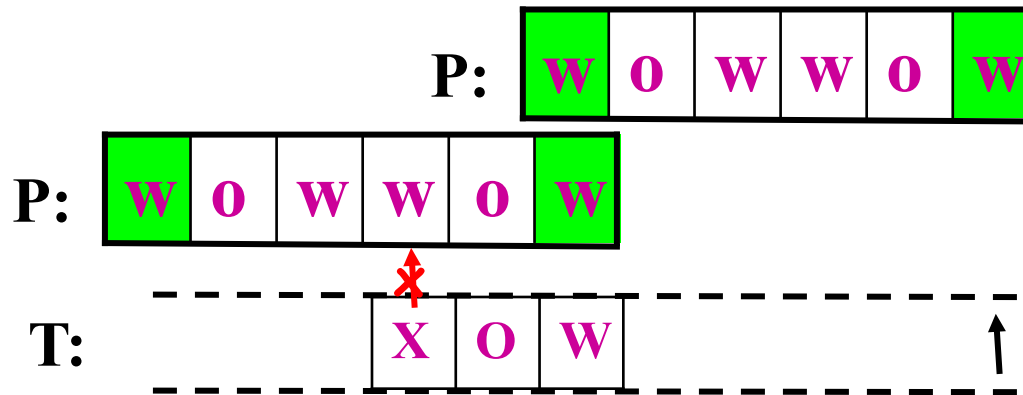
				X	w



Matched = 2 (m-k)

Slide[4] = 5 (m-q)

**matchJump[4] = 7**



P: 

w	o	w	w	o	w
---	---	---	---	---	---



T: \_\_\_\_\_

Matched = 3 (m-k)

Slide[3] = 3 (m-q)

**matchJump[3] = 6**

P: 

w	o	w	w	o	w
---	---	---	---	---	---

P: 

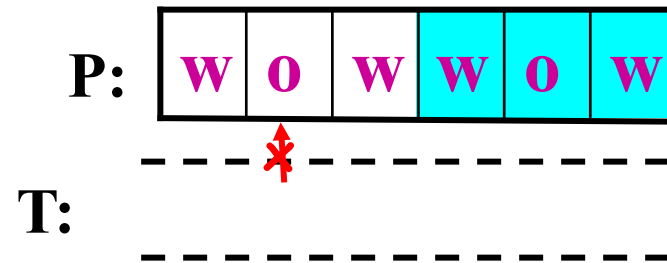
w	o	w	w	o	w
---	---	---	---	---	---



T: \_\_\_\_\_

x	w	o	w
---	---	---	---

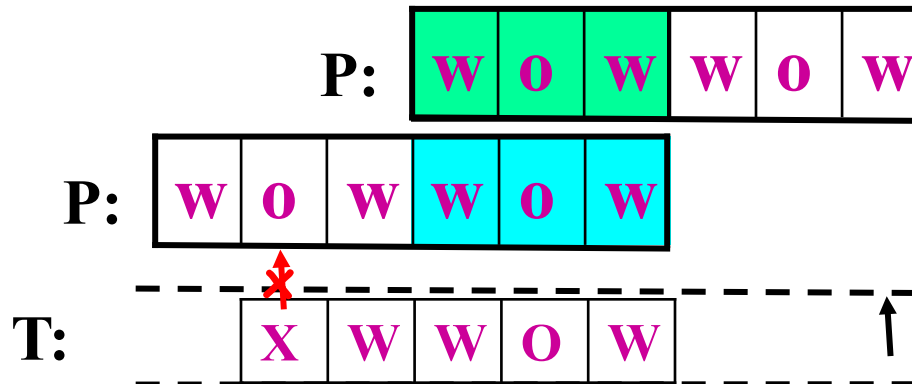




Matched = 4 (m-k)

Slide[2] = 3 (m-q)

**matchJump[2] = 7**





P: 

W	O	W	W	O	W
---	---	---	---	---	---

-----  
 \*  
 -----

T: -----

Matched = 5 (m-k)

Slide[1] = 3 (m-q)

**matchJump[1] = 8**

**matchJump**

8	7	6	7	3	1
---	---	---	---	---	---

P: 

W	O	W	W	O	W
---	---	---	---	---	---

P: 

W	O	W	W	O	W
---	---	---	---	---	---

T: 

X	O	W	W	O	W
---	---	---	---	---	---

 -----  
 \*  
 -----  
 ----- ↑

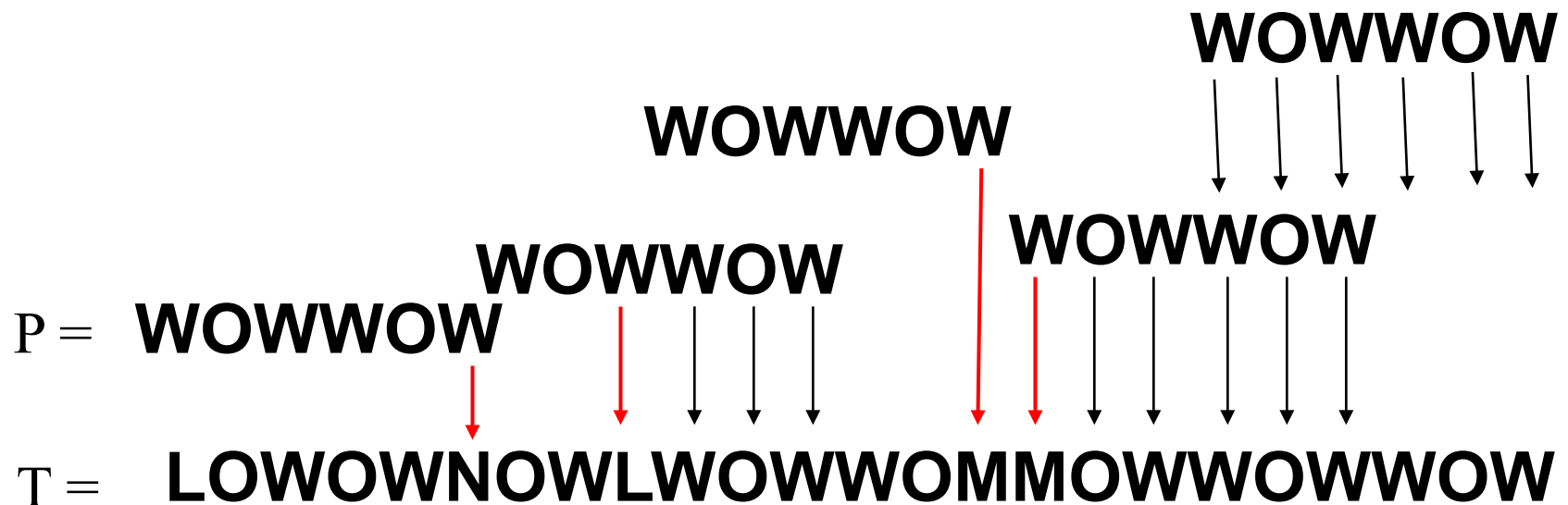
BMscan example: Pattern is WOWWOW

$\text{charjump}['O'] = 1$ ,  $\text{charjump}['W'] = 0$ ,  $\text{charjump}[X] = 6$ ,

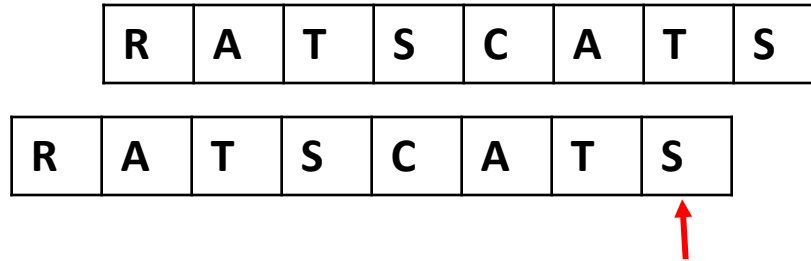
**matchJump**

8	7	6	7	3	1
---	---	---	---	---	---

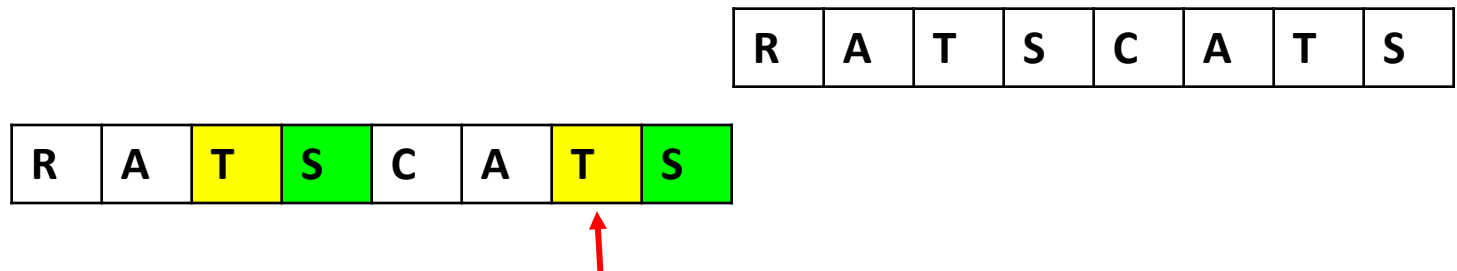
Match found  
after 18  
comparisons



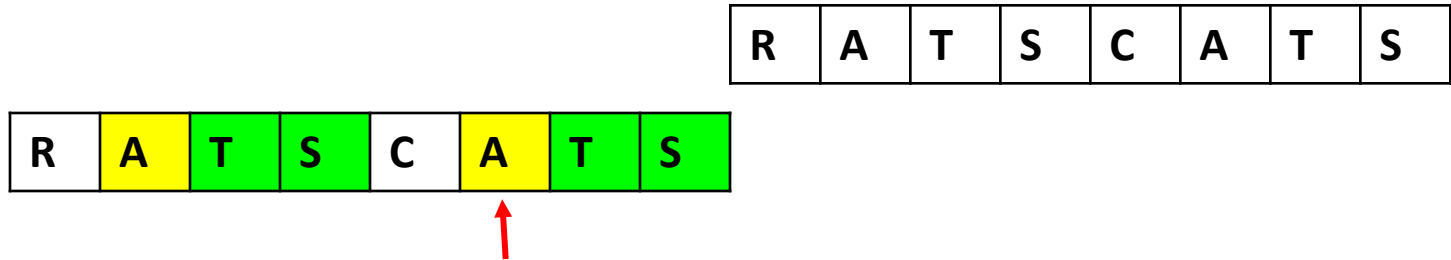
## Another example



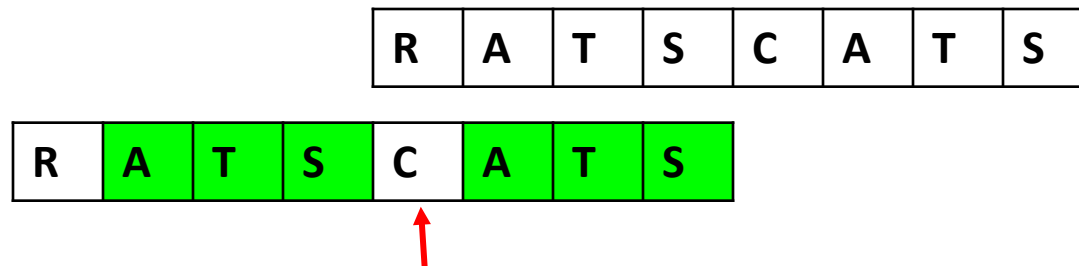
Matched =  $m - k = 0$ , Slide[8] = 1  
matchJump[8] = 1



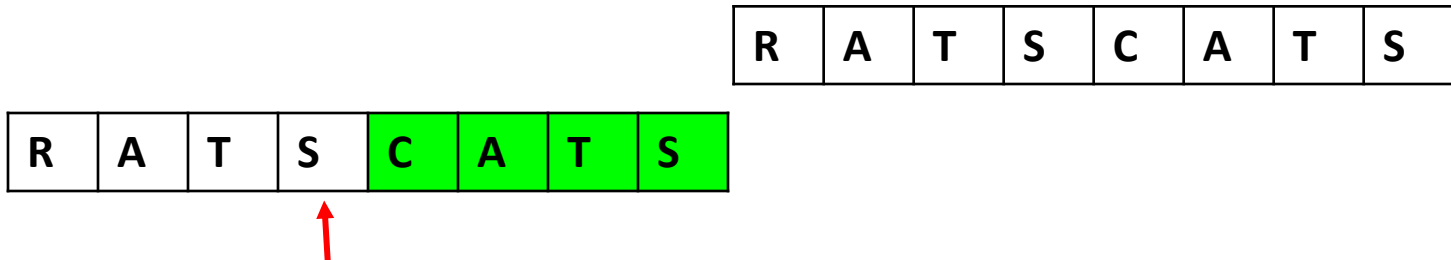
Matched =  $m - k = 1$ , Slide[7] =  $m = 8$   
matchJump[7] = 9



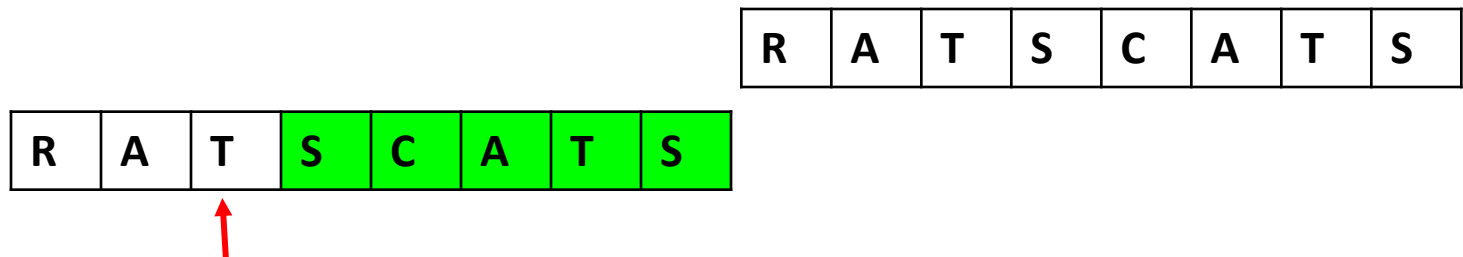
Matched =  $m - k = 2$ , Slide[6] =  $m = 8$   
 matchJump[6] = 10



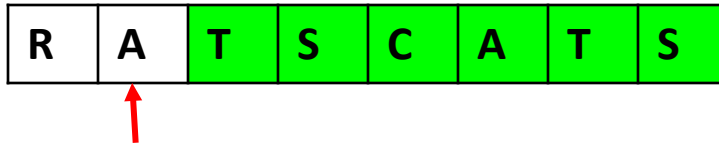
Matched =  $m - k = 3$ , Slide[5] =  $m - q = 4$   
 matchJump[5] = 7



Matched =  $m - k = 4$ , Slide[4] =  $m = 8$   
matchJump[4] = 12

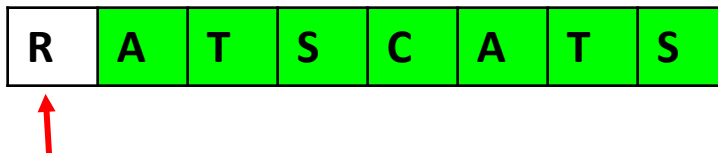


Matched =  $m - k = 5$ , Slide[3] =  $m = 8$   
matchJump[3] = 13



R	A	T	S	C	A	T	S
---	---	---	---	---	---	---	---

Matched =  $m - k = 6$ , Slide[2] =  $m = 8$   
 matchJump[2] = 14

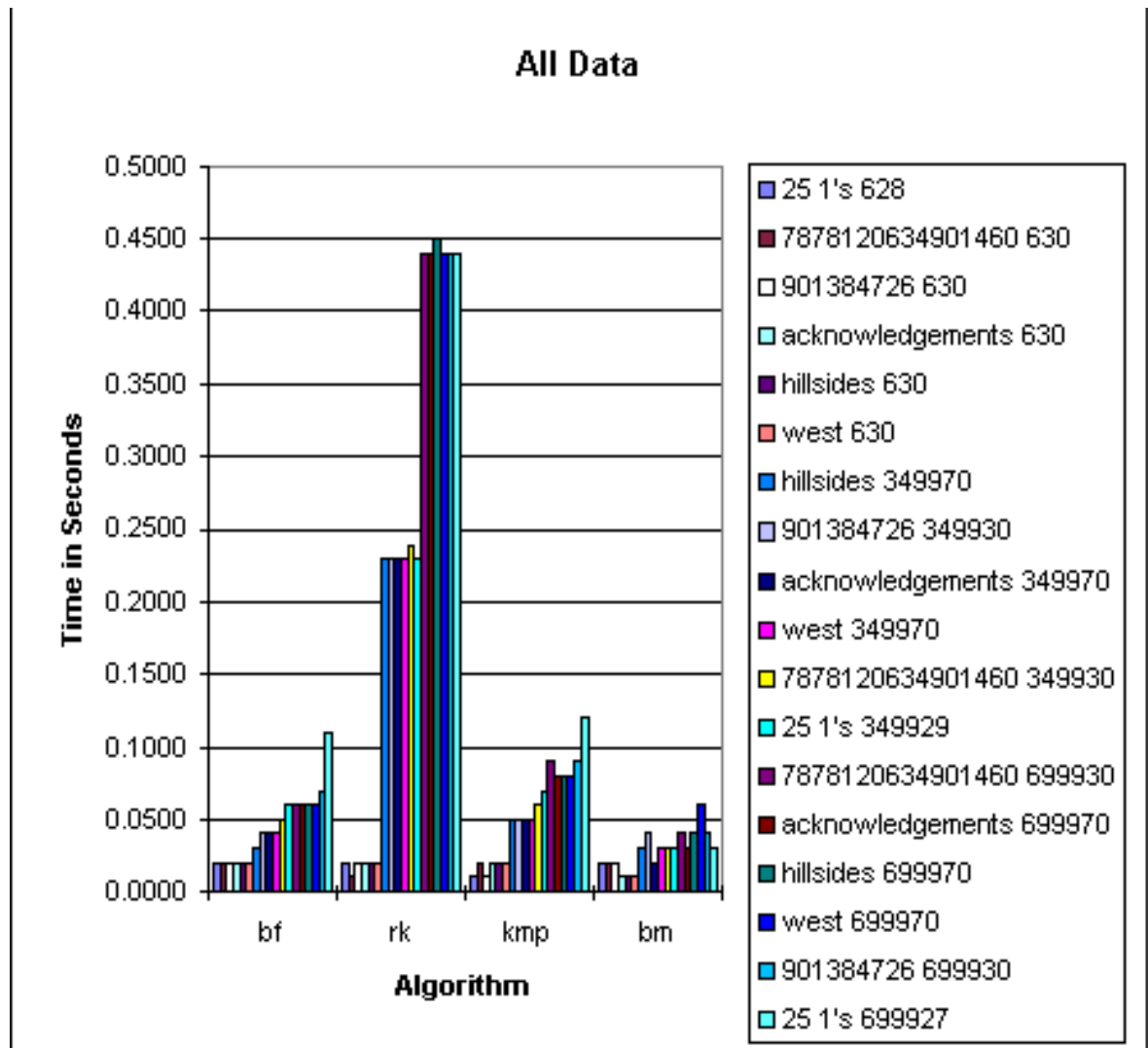


R	A	T	S	C	A	T	S
---	---	---	---	---	---	---	---

Matched =  $m - k = 7$ , Slide[1] =  $m = 8$   
 matchJump[1] = 15

	R	A	T	S	C	A	T	S
matchJump	15	14	13	12	7	10	9	1

- Brute-Force Algorithm (bf)
- Rabin-Karp Algorithm (rk)
- Knuth-Morris-Pratt Algorithm (kmp)
- Boyer-Moore Algorithm (bm)



- Brute Force behaved better than we expected
  - because worst case is not common. Worst case would occur when the pattern and the text produced a near match.
- Rabin-Karp behaved much worse
  - Rabin-Karp has several function calls. These are expensive, timewise.
  - Any division, including mod, is time expensive.
  - The conversion from character values to numeric values takes time.



- Boyer-Moore algorithm is considered the most efficient string-matching algorithm in usual applications, for example, in text editors.
- Moore says the algorithm has the peculiar property that, roughly speaking, the longer the pattern is, the faster the algorithm goes.
- The payoff is not as for binary strings or for very short patterns.
- For binary strings Knuth-Morris-Pratt algorithm is recommended.
- For the very shortest patterns, the brute force algorithm may be better.

- What else do we learn from the BM algorithm?
  - Designing algorithms to solve problems often needs insights into a problem's structure – analyse the problem carefully before thinking about its solution

# Introduction to NP

Huang Shell Yíng

**References:** *Baase & Van Gelder 13.1, 13.2, 13.4*

# Outline

1. P and NP problems
2. NP-Completeness
3. Greedy heuristic algorithms for TSP
4. Greedy heuristic algorithm for Knapsack

# P and NP problems

**Hard problems** are those that the best-known algorithm for the problem is expensive in running time.

- It is not a problem that is hard to understand
- It is not a problem that is hard to code
- It is a problem that takes exponential time or more to compute – not practical to obtain a solution for a problem of a modest size  $n$

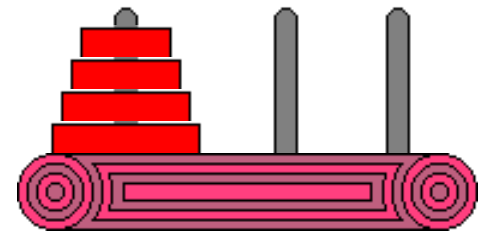
It is important to know when a problem is hard.

- We should then focus on finding an efficient algorithm to obtain a good solution instead of an optimal solution

# Problem: Towers of Hanoi

Compute a sequence of one-disk moves to transfer N disks from the first pole to the second pole. A third pole can be used in the process. Constraints: only one disk can be moved at a time and no disk is ever placed on top of a smaller one.

```
void TowersOfHanoi(int n, int x, int y, int z)
{
    if (n == 1)
        cout << "Move disk from " << x << " to " << y << endl;
    else {
        TowersOfHanoi(n-1, x, z, y);
        cout << "Move disk from " << x << " to " << y << endl;
        TowersOfHanoi(n-1, z, y, x); }
}
```



The number of disk moves/lines to print:

$$M_1 = 1;$$

$$M_n = 2M_{n-1} + 1$$

Solution for the recurrence:  $M_n = 2^n - 1$

For the computer program to solve this problem, the number of lines printed for  $N$  disks is  $2^N - 1$ .

So with 64 disks it would print  $2^{64} - 1$  lines.

Assuming the printer can print 1 thousand lines a second, the program would require 140,000 hours to complete!!



What causes this lengthy processing time in Tower of Hanoi?  
The output?

A **decision problem** is an algorithmic problem that has two possible answers: yes, or no. The purpose is merely to decide whether a certain property holds for the problem's input.

Examples:

1. Can we travel from city A to city B within  $k$  hours?
2. Can we travel from city A, visit every other city exactly once and return to city A in  $k$  hours?
3. Is it possible to supply electricity to all homes in an area by a network of power lines of less than  $k$  kilometers?
4. Is there a subset of the  $n$  objects that fits in the knapsack of capacity weight  $C$  and returns a total profit of at least  $k$ ?

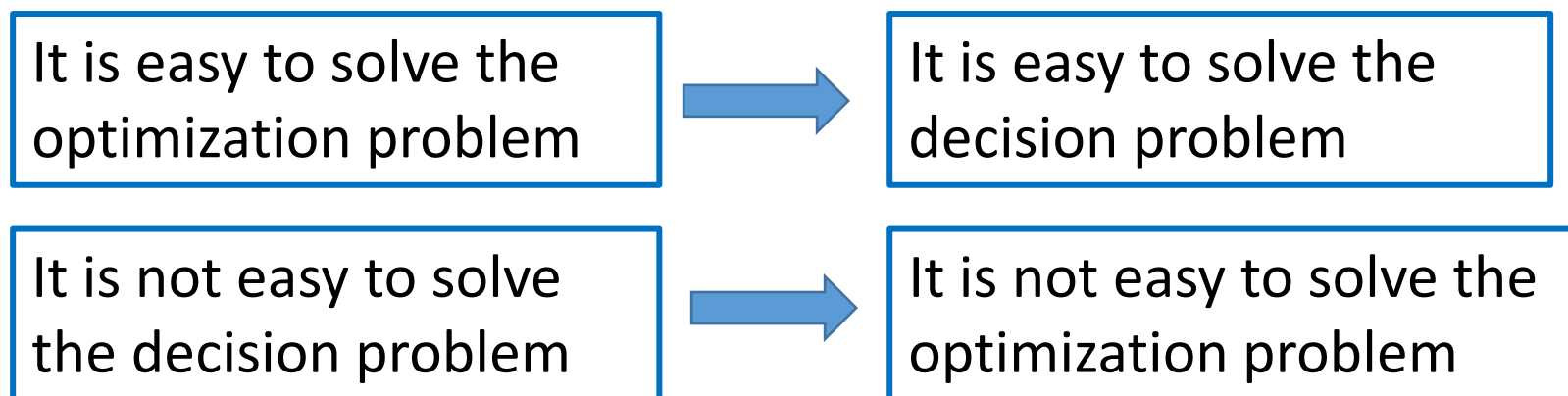


An **optimization problem** is a problem of finding the best solution from all feasible solutions.

Examples:

1. What is the shortest path from city A to city B?
2. What is the shortest path to travel from city A, visit every other city exactly once and return to city A?
3. What is the network of minimum-length power lines to supply electricity to all homes in an area?
4. What is the maximum profit if a subset of the  $n$  objects are put into a knapsack of capacity weight  $C$ ?

- In general, each decision problem has its corresponding optimization problem.
- Each optimization problem can be recast as a decision problem.
- If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.



# Problems that are not hard – P problems

## Definitions:

An algorithm is said to be **polynomially bounded** if its worst case complexity is bounded by a polynomial function of the input size.

A problem is said to be *polynomially bounded* if there is a polynomially bounded algorithm for it.

The class  $\mathcal{P}$  problems is a class of decision problems that are *Polynomially* bounded.

Examples of  $\mathcal{P}$  problems:

1. Can we travel from city A to city B within  $k$  hours?
2. Is it possible to supply electricity to all homes in an area by a network of power lines of less than  $k$  kilometers?

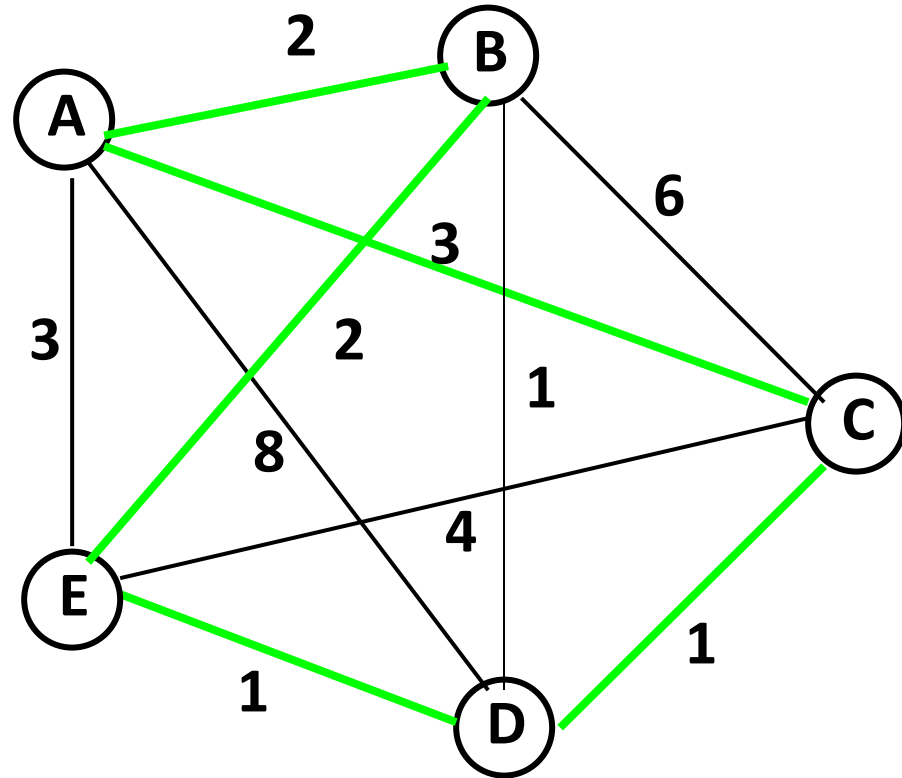
How about the problem of finding if we can travel from city A, visit every other city exactly once and return to city A in  $k$  hours?

The corresponding optimization problem is known as the **travelling salesman problem**.

# Travelling Salesman Problem

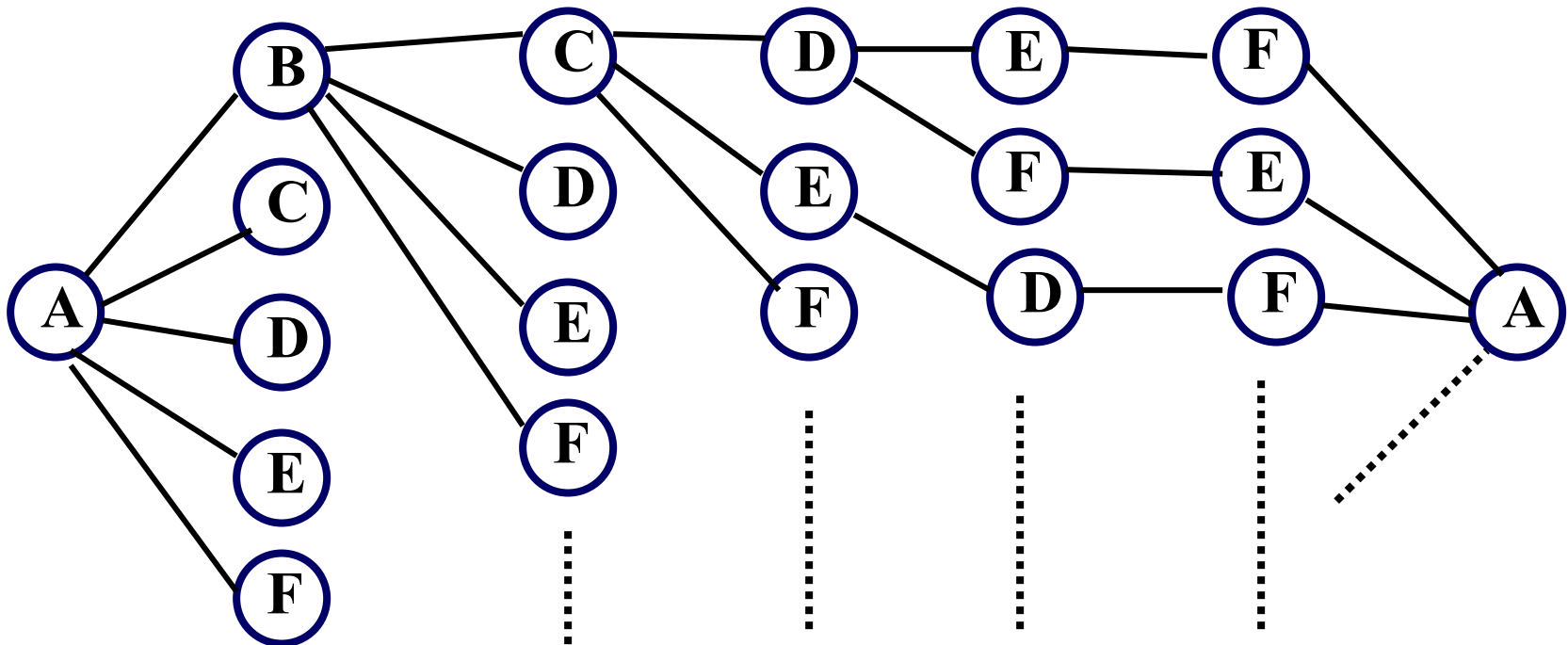
Example:

Is there a tour of all the cities with total cost of not more than 12?

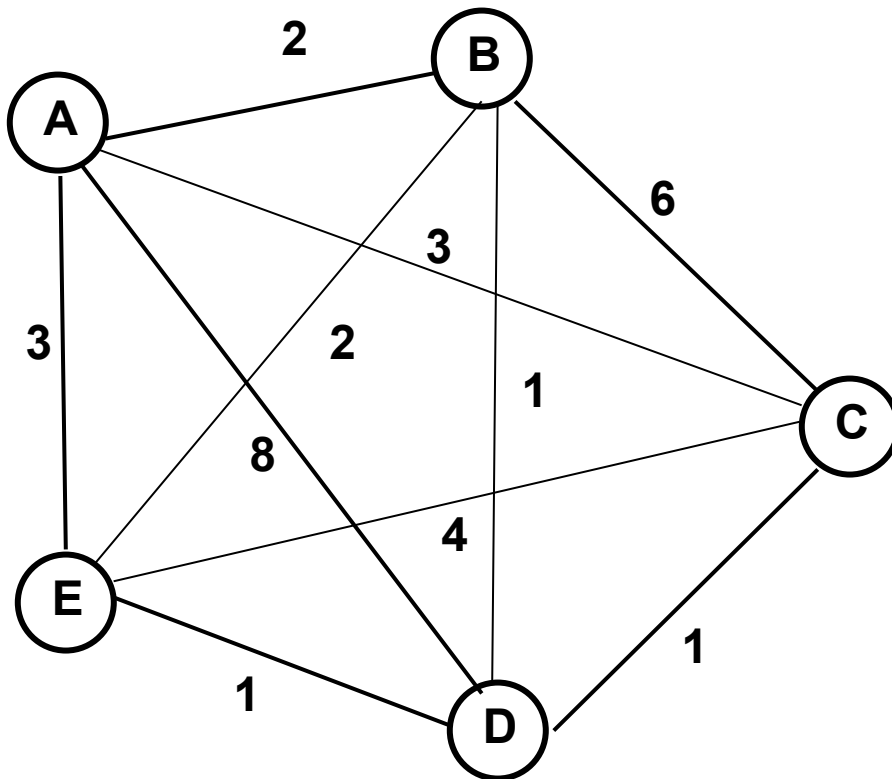


How do we find the answer to this decision problem?

Consider all possible tours, terminating after we find a tour that costs no more than 12. We have potentially 120 tours to check.



The worst case time complexity is  $O(N!)$ . If  $N = 25$ ,  $N!$  is a 26 digit number!! There is no known polynomial time algorithm.



Another characteristic of this problem:

Make a guess and check:

(1) route ABCDEA - total is 13 – answer is ‘no’.

(2) route ABDCEA - total is 11 – answer is ‘yes’

**Problem solved.**

May not get the best route which is ABEDCA - cost of 9

How do we check a solution: (1) it is a correct route (2) total length of the route. The checking can be done in  $O(n^2)$  time!

# 0/1 Knapsack problem

We have a knapsack of capacity weight  $C$  (a positive integer) and  $n$  objects with weights  $w_1, w_2, \dots, w_n$  and profits  $p_1, p_2, \dots, p_n$  (all  $w_i$  and all  $p_i$  are positive integers), is there a subset of the objects that fits in the knapsack and returns a total profit of at least  $k$ ?

- There is no known polynomial time algorithm.
- There are  $2^n$  subsets of  $n$  objects: in the worst case all subsets need to be examined which takes  $O(2^n)$  time.
- Given a guess of a subset, it takes  $O(n)$  time to check whether it satisfies the requirements.

The dynamic programming algorithm has a complexity of  $O(nC)$ .  
It is pseudo-polynomial time





Example:  $C = 20$ , is there a subset of objects that can be put into the knapsack to make a profit of at least 21?

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

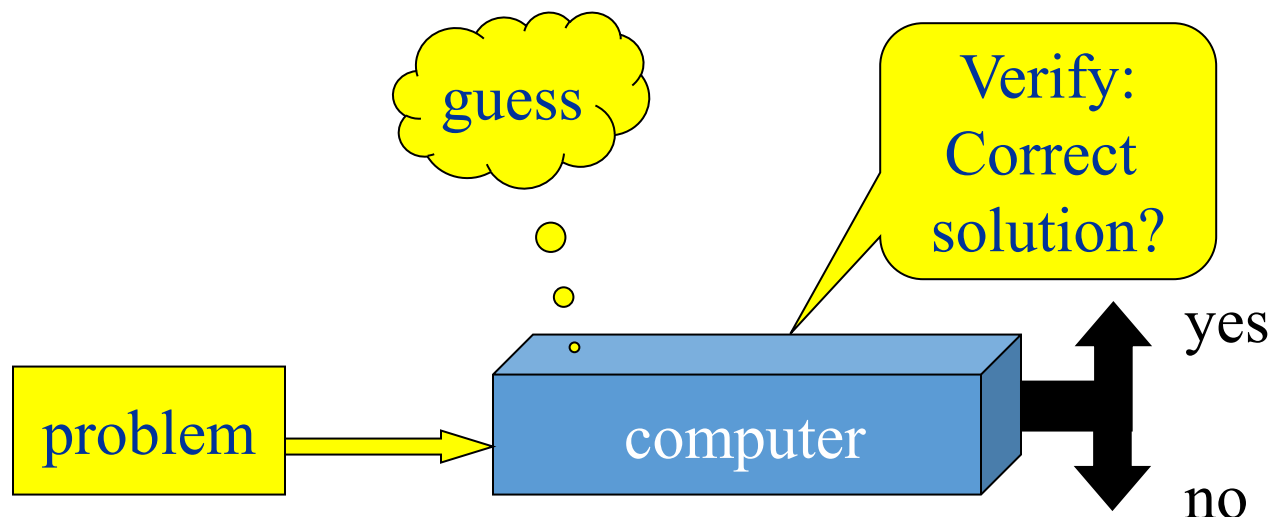
Make a guess and check: Subset:  $\{1, 3, 4\}$ .

Total weight =  $w_1 + w_3 + w_4 = 18$ .

Profit =  $p_1 + p_3 + p_4 = 21$

Answer 'yes'. (optimal is 22)

- There is a class of problems for which
  - There is no known polynomial time algorithms
  - Checking of a solution can be done in polynomial time



A ***nondeterministic algorithm*** solves a problem in two phases, 'guess' then 'verify', and an output step, 'yes' or 'no' output.

A ***nondeterministic algorithm*** can also be considered as a machine that checks all possible solutions in parallel to see which is correct.

# NP problems

## Definition:

***NP* [Nondeterministic Polynomially bounded]** : is the class of decision problems for which there is a polynomially bounded nondeterministic algorithm. i.e. it is a problem that can be solved in polynomial time on a nondeterministic machine.

Alternatively, we say that the complexity class NP is the class of problems that can be **verified** by a polynomial time algorithm.

- $P \subseteq NP$ . Every class P problem is also a class NP problem.
- There is widespread belief that  $P \neq NP$ .

- Examples of NP problems
  - Travelling salesman problem (a tour of  $n$  cities that has a total cost less than  $k$ )
  - Knapsack problem
- An example of non-NP problems: A variant of the travelling salesman problem is:

Given a network of cities  $G$  and a number  $k$ . Does every tour have total cost of not more than  $k$ ?

- We cannot verify a solution in polynomial time.
- In fact, it is a provable exponential time problem.

# NP-Completeness

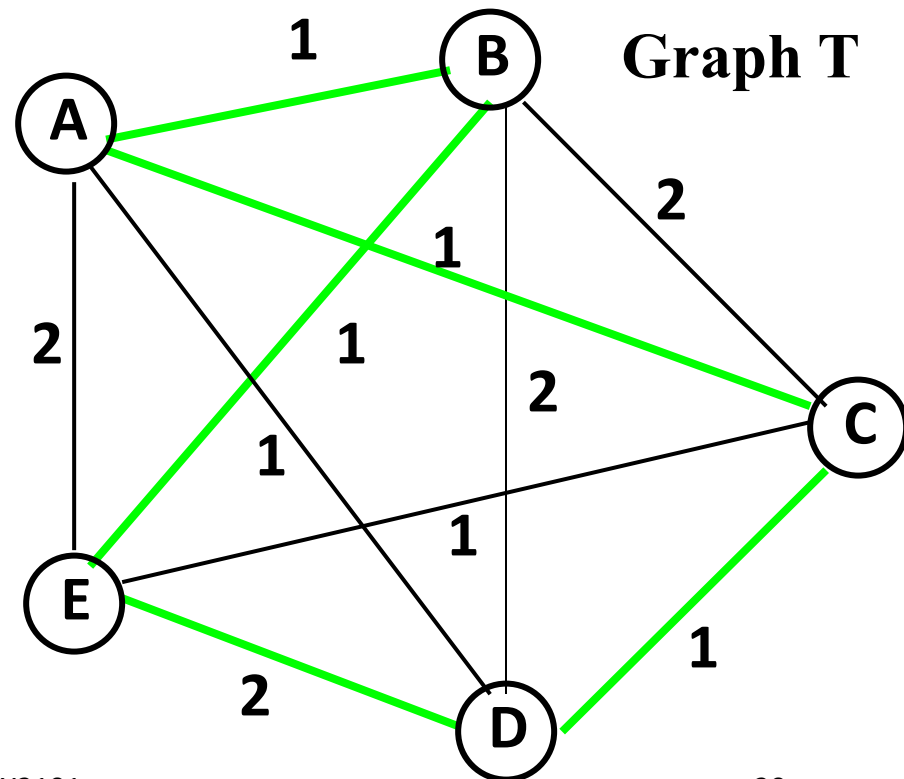
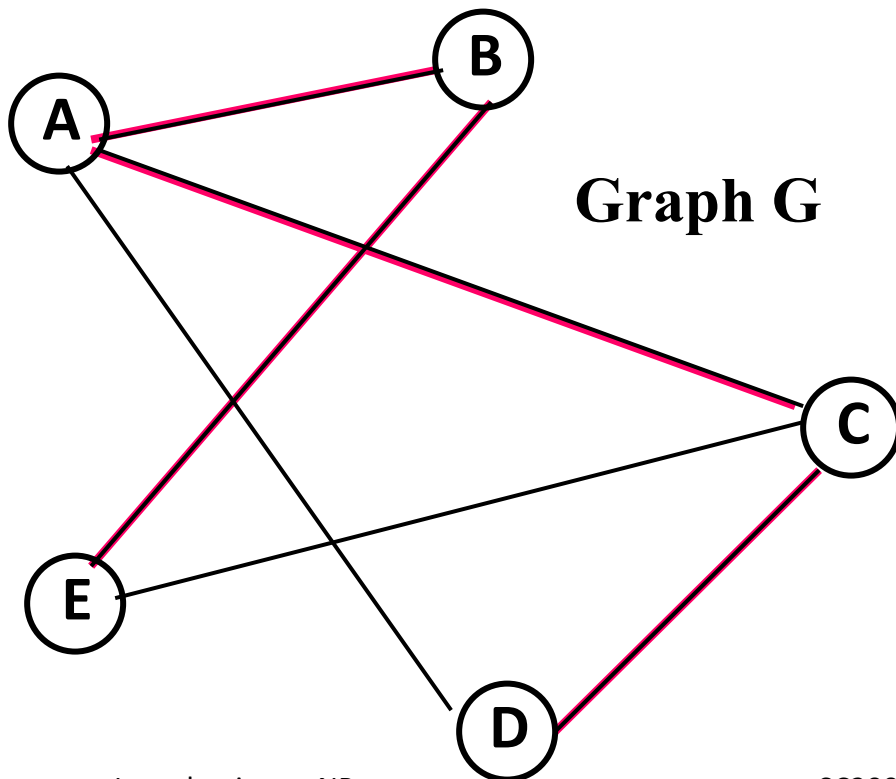
## Problem Reduction

- How do we show that a problem is hard? We reduce a known hard problem to this problem
- A reduction is a way of converting one problem  $P_1$  to another problem  $P_2$
- If  $P_2$  has an algorithm, reduction gives us a way to solve  $P_1$ .
- It also means that  $P_1$  is no harder than  $P_2$ , i.e.  $P_2$  is as hard as  $P_1$
- Example: FactorAll( $n$ ) finds all the factors of a number  $n$ . Factor1( $n$ ) finds one factor.
  - FactorAll() reduces to Factor1().
  - We use Factor1( $n$ ) to find one factor  $m$  of  $n$ . Then divide  $n$  by  $m$ , and run FactorAll on the result. Keep repeating, and we get all the factors.

- Another example of reduction

Hamiltonian path problem:

Given a graph  $G$ , is there a path, any path, that passes through all the vertices of the graph exactly once?



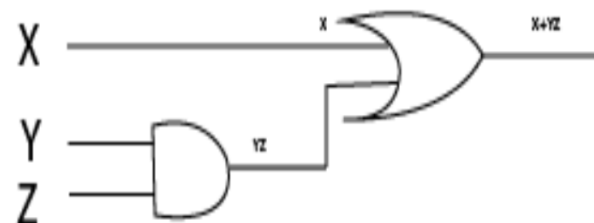
The transformed problem: Does graph  $T$ , have a travelling salesman tour that is no longer than  $N+1$ , where  $N$  is the number of vertices in  $G$ ?

- The answer to the Hamiltonian path problem is “yes” precisely when the answer to the TSP is “yes”.
  - **The transformation can be done in polynomial time.**
  - So if TSP has a polynomial time algorithm, Hamiltonian path problem can also be solved in polynomial time!
- 
- Reducibility plays an important role in classifying problems.
  - The point of reducibility of  $P1$  to  $P2$  is that  $P2$  is at least as ‘hard’ to solve as  $P1$ .

# NP-Completeness

Definition: A problem D is **NP-complete** if it is in NP and every problem Q in NP is reducible to D in polynomial time.

The circuit satisfiability problem (**CIRCUIT-SAT**) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.



**SAT** = satisfiable Boolean formulas.

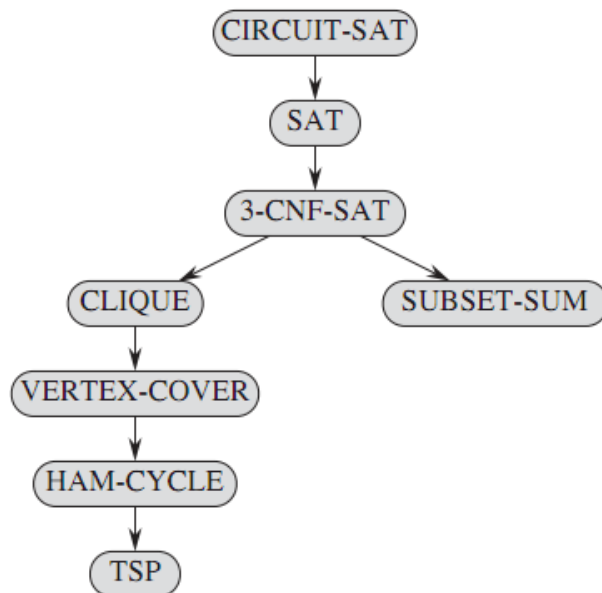
Given a Boolean formula, is there any setting for the variables which makes the Boolean formula true?

$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg C \vee \neg D)$

Setting  $A=B=C=\text{true}$ ,  $D=\text{false}$  makes the formula true.



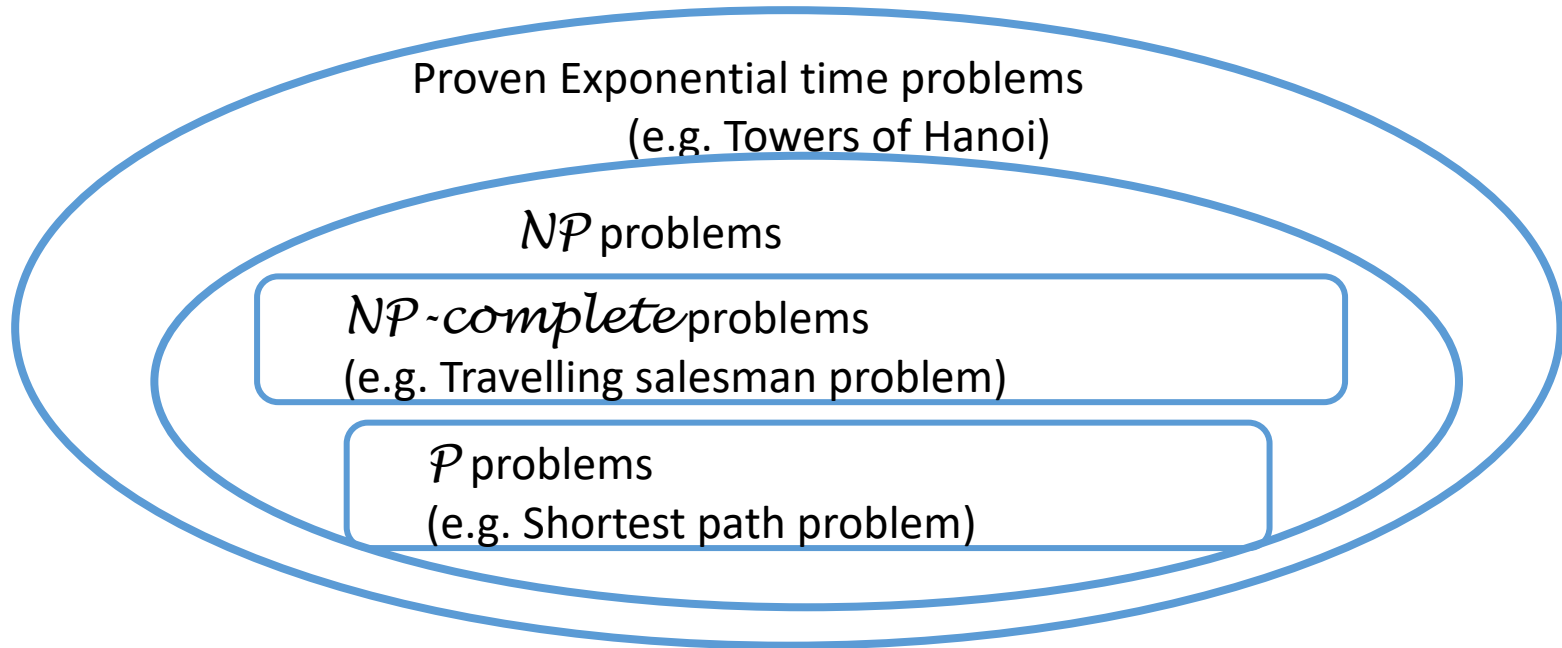
- Cook and Levin proved that **CIRCUIT-SAT** is NP-complete.
- This means every NP problem can be solved by reducing it to CIRCUIT-SAT.
- CIRCUIT-SAT can be reduced to SAT and SAT is in NP. So **SAT** is NP-complete.
- All the problems below and many others are NP-Complete.



Note that reducibility is transitive.

- NP-complete problems are equal to each other in difficulty and they are the hardest problem in NP.
- So if we want to show that a problem is hard, prove it is a NP\_complete problem.
- If anyone finds a polynomial time solution to one of these NP-complete problems, by a series of reductions, every other problem that is in **NP** can also be solved in polynomial time !!  
Then  $P = NP$ .

# Classification of problems



With so many NP- complete problems that are important applications, how do we solve them?

- Use small problem sizes
- Solve for a special instance of the problem
- heuristic algorithms
  - These are fast (polynomially bounded) algorithms
  - not guaranteed to give the best solution
  - will give one that is close to the optimal in many cases
  - But could return a very bad solution

# Greedy heuristic algorithm for TSP – Nearest Neighbour

nearestTSP( $V, E, W$ )

{ Select an arbitrary vertex  $s$  to start the cycle  $C$

$v = s$ ;

while there are vertices not in  $C$  {

    select an edge  $vw$  of minimum weight

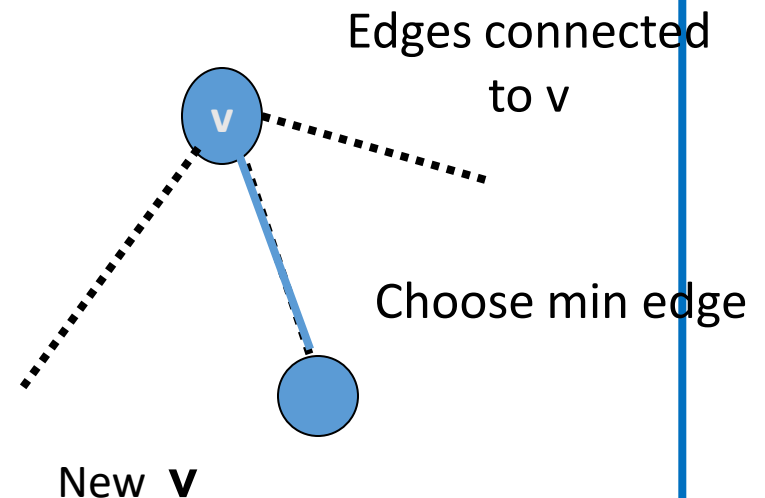
    where  $w$  is not in  $C$

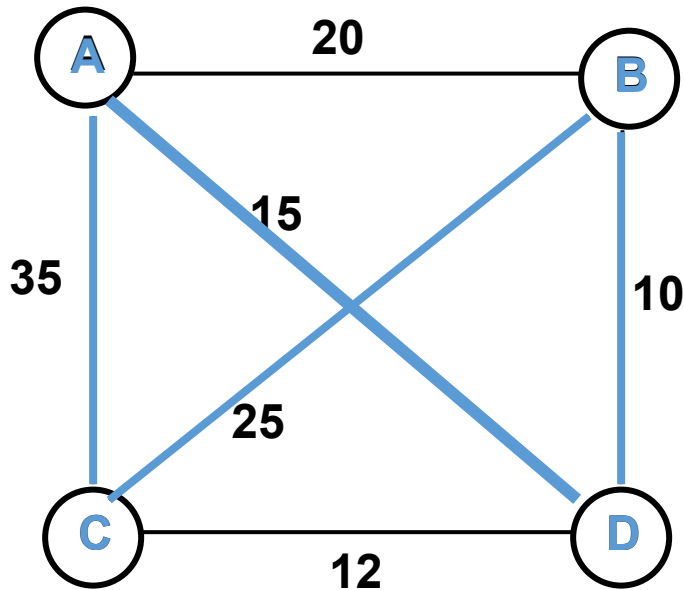
    add edge  $vw$  to  $C$ ;

$v = w$ ; }

add edge  $vs$  to  $C$ ;

return  $C$ ; }





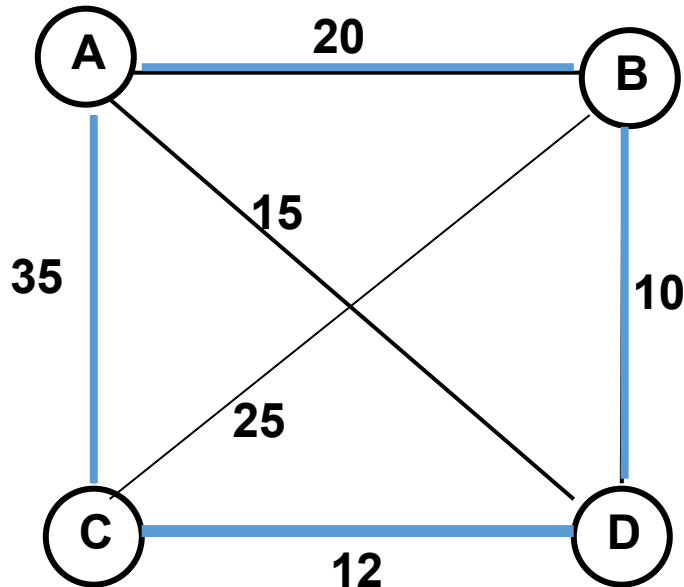
Starting at A, nearest-neighbour strategy results in:

Total distance of 85.  
This is NOT the minimum.

- Example of a greedy strategy that does not always give an optimum solution
- Will result in a very bad solution if the weight of AC is, e.g, 10,000

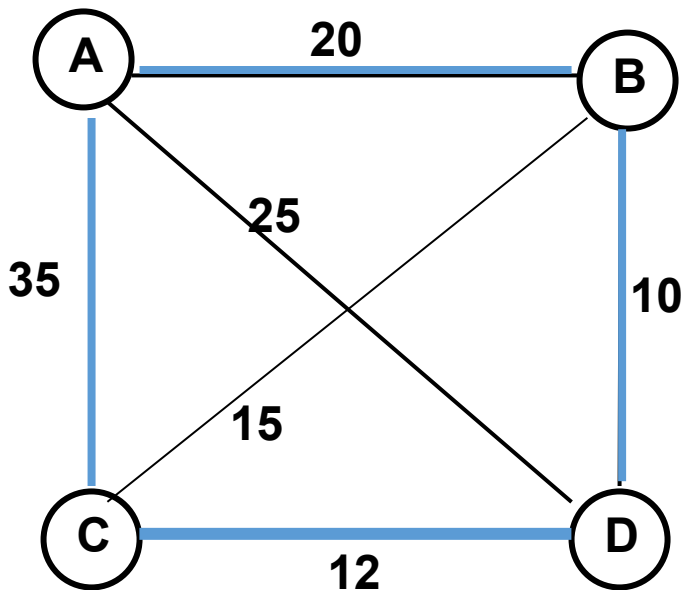
# Greedy heuristic algorithm for TSP – Shortest Link

```
shortestLinkTSP(V, E, W)
{
  R = E;
  C = empty; // C is a forest
  while (no. of edges in C < |V| - 1) {
    remove the lightest edge vw from R;
    if (vw does not form a cycle in C and vw
        would not be the third edge in C
        incident on v or w)
      add edge vw to C;
    add edge connecting the end points to C;
  }
  return C;
}
```



Using shortestLinkTSP, edges chosen are: BD, DC, AB, CA. This solution has total distance of 77.

Though better than the nearest-neighbour, it is still not optimal. Optimal is  $A \rightarrow D \rightarrow C \rightarrow B \rightarrow A$  [72]





# Greedy heuristic algorithm for Knapsack

Function greedy-knap( $w[1..n]$ ,  $p[1..n]$ ,  $C$ )

Sort the list of objects in descending order of  $p[i]/w[i]$ ;

initialize weight and value of knapsack to 0;

for  $i = 1$  to  $n$

    if ( $\text{weight} + w[i] \leq C$ )

$\text{value} += v[i]$ ;

$\text{weight} += w[i]$ ;

return value;

Example:  $C = 20$

	1	2	3	4
$w_i$	4	6	8	6
$p_i$	7	6	9	5

	1	2	3	4
$p_i/w_i$	$7/4 = 1.75$	1	$9/8 = 1.125$	$5/6 = 0.83$

Answer : choose objects 1, 3, 2 so subset = {1, 2, 3}, profit = 22

THE END

# Additional Notes (not to be examined)

- Definition of NP-hard problem – A problem  $H$  is NP-hard if every problem  $Q$  in NP is reducible to  $H$  in polynomial time
- So a NP-hard problem is as hard as any NP problem, may be even harder.
- Examples (1): All NP-complete problems are NP-hard. Note these are NP problems.
- Examples (2): the optimization problems, whose corresponding decision problems are NP-complete, are NP-hard problems. These are not NP problems because they are not decision problems.