Analysis of Algorithms

(Chapter 2)

What did we learn last lesson?

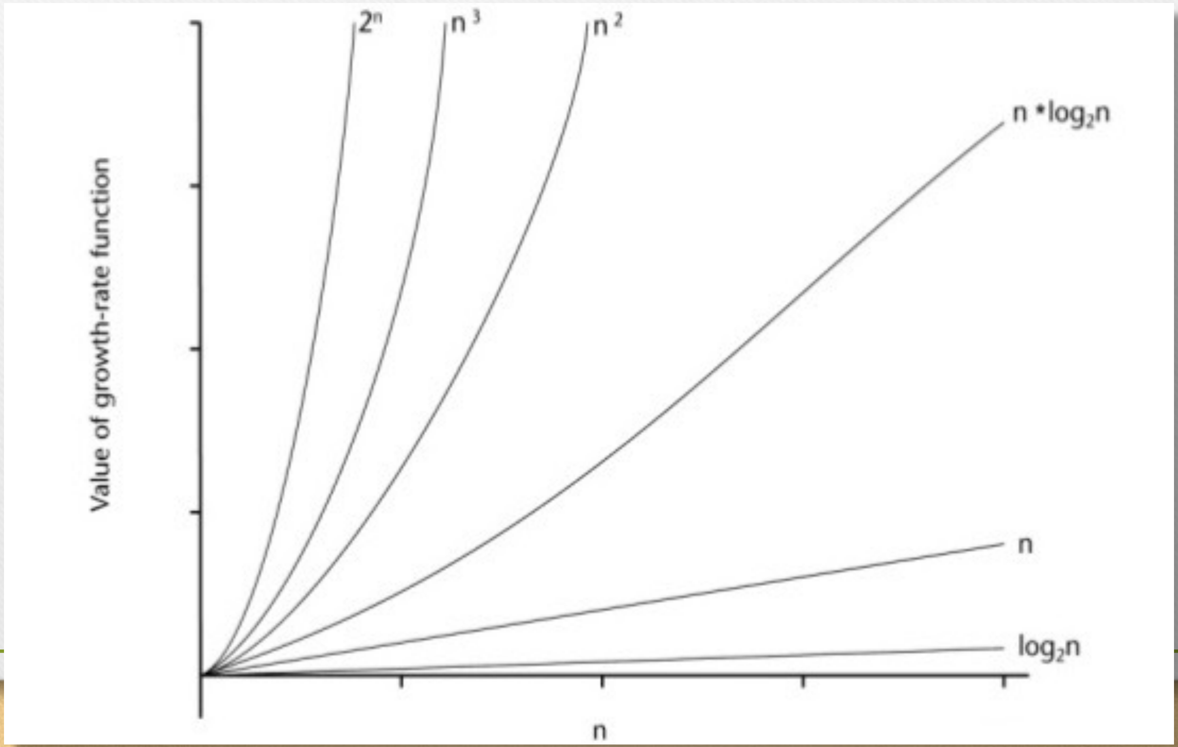
1. Efficiency of an algorithm depends on **input size**
2. Efficiency of an algorithm also depends on **basic operation**
3. Efficiency can be expressed by **counting** the basic operation

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
     A[j+1] ← A[j]
     j ← j-1
   A[j+1] ← v
```

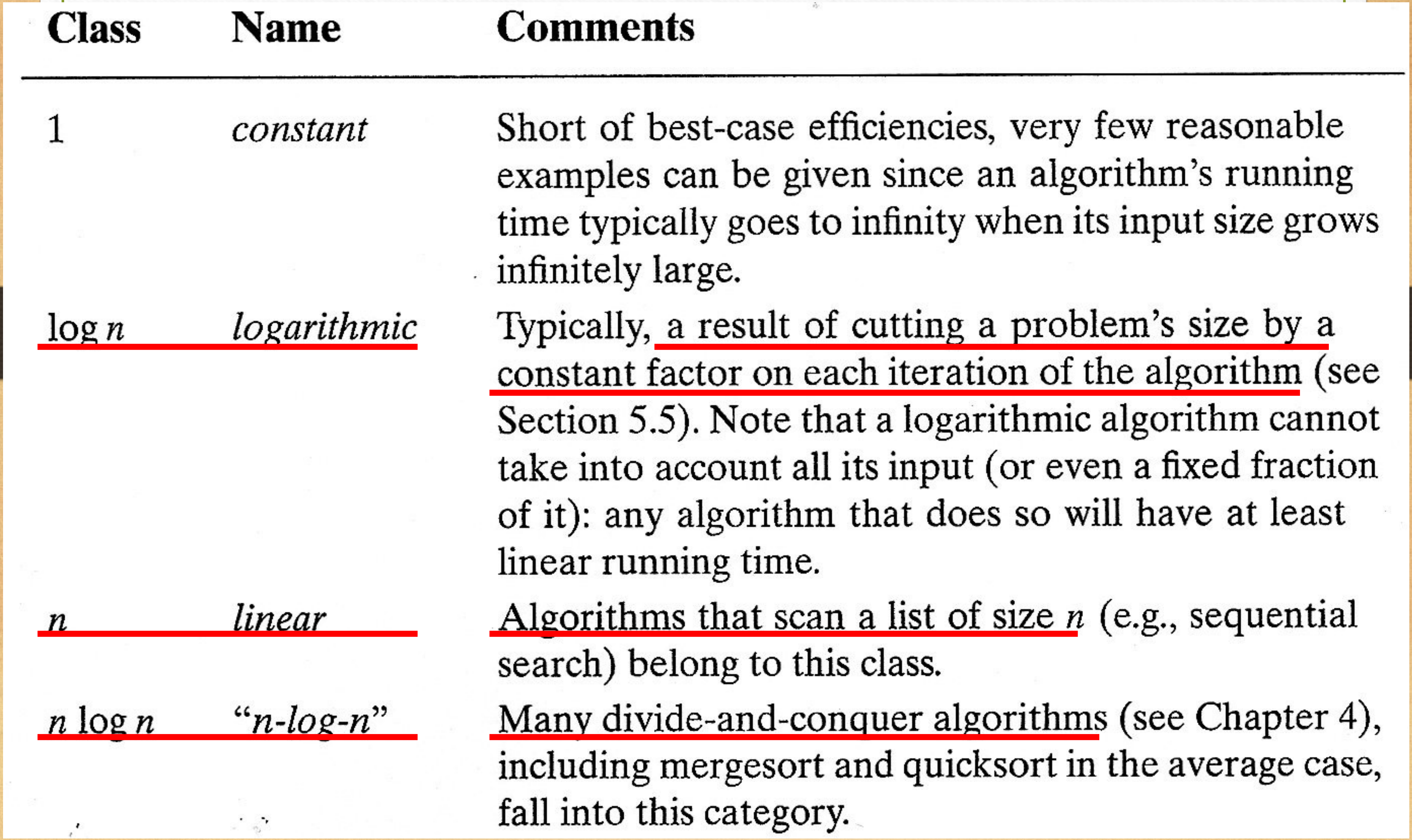
$$C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2}$$

Order of growth

- What we really care about:
  - Order of growth as $n \to \infty$

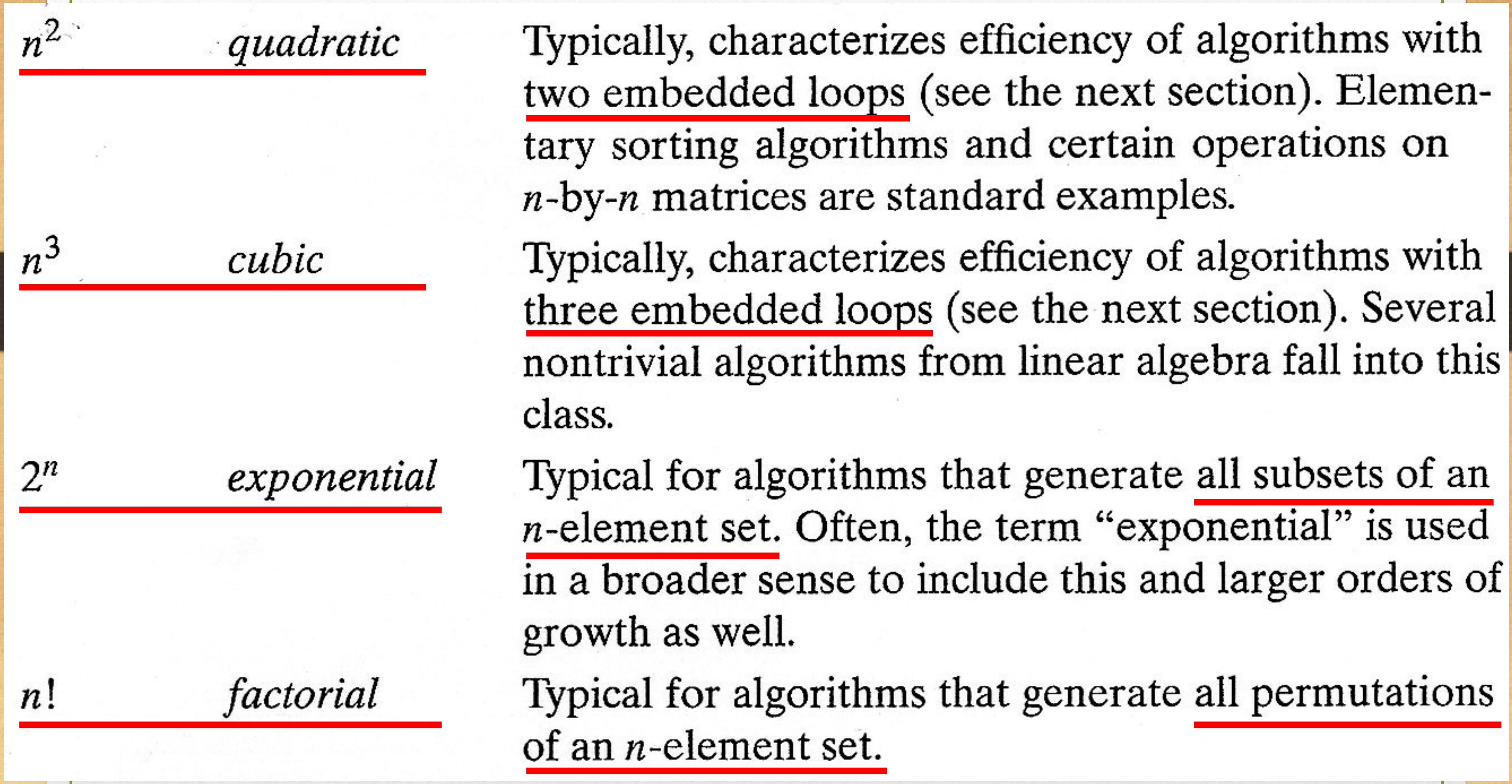


Base Efficiency Classes



Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.

Base Efficiency Classes



n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term “exponential” is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

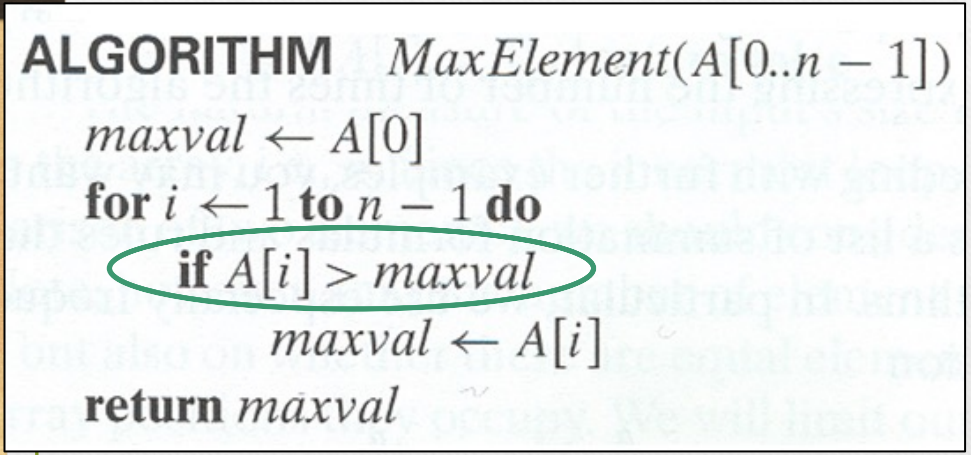
General Strategy for Analysis of Non-recursive Algorithms

This strategy is taken from page 62 of your textbook:

1. Decide on a parameter indicating the inputs size.
2. Identify the algorithms basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of the input.
    - if it depends on some other property, the best/worst/average case efficiencies must be investigated separately
4. Set up a sum expressing the number of times the basic operation is executed.
5. Use standard formulas and rules of sum manipulation to find a closed form formula c(n) for the sum from step 4 above.
6. Determine the efficiency class of the algorithm using asymptotic notations

Example1

- Problem: find the max element in a list
- Input size measure:
  - *Number of list's items, i.e. n*
- Basic operation:
  - *Comparison*

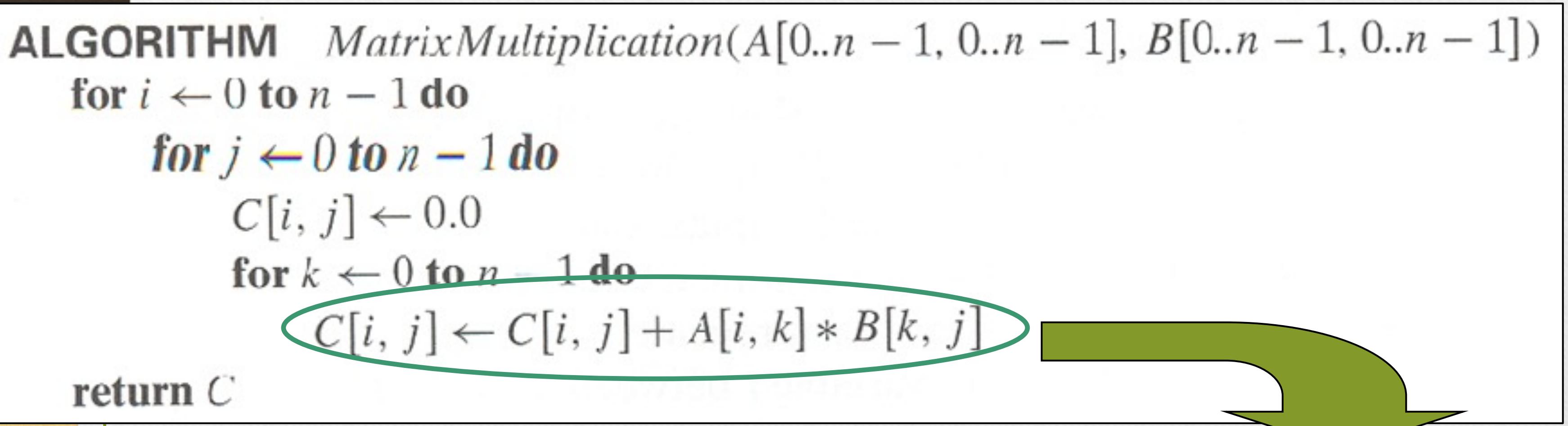


```
ALGORITHM MaxElement(A[0..n − 1])
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in O(n)$$

Example2

- Problem: *Multiplication of two matrices*
- Input size measure:
    - ~~*Matrix dimensions or total number of elements*~~
- Basic operation:



```
ALGORITHM  MatrixMultiplication($A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]$)
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ to $n-1$ do
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return $C$
```

$$C(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = n^3 \in O(n^3)$$

Brute Force

(Chapter 3)

Brute Force Technique

- No formal definition
- The "obvious and straightforward" approach for solving a problem
- Not really trying to be efficient
- Typically these are easy to implement
- "Force" comes from using computer power not intellectual power or "Just do it!"

Brute Force Example

What is a brute force vs sophisticated solution for the following?

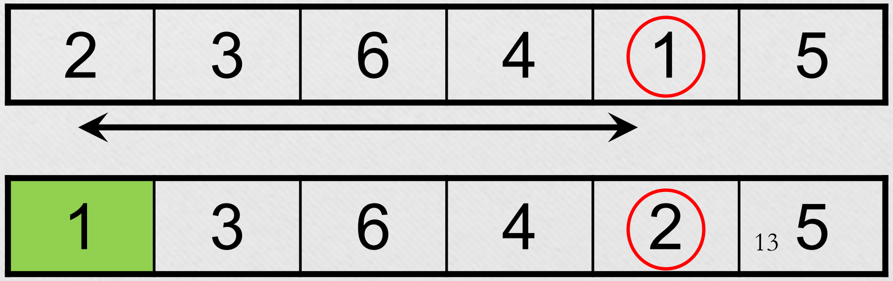
1. Finding a name in a phone book
2. Calculating a Fibonacci number

Brute Force

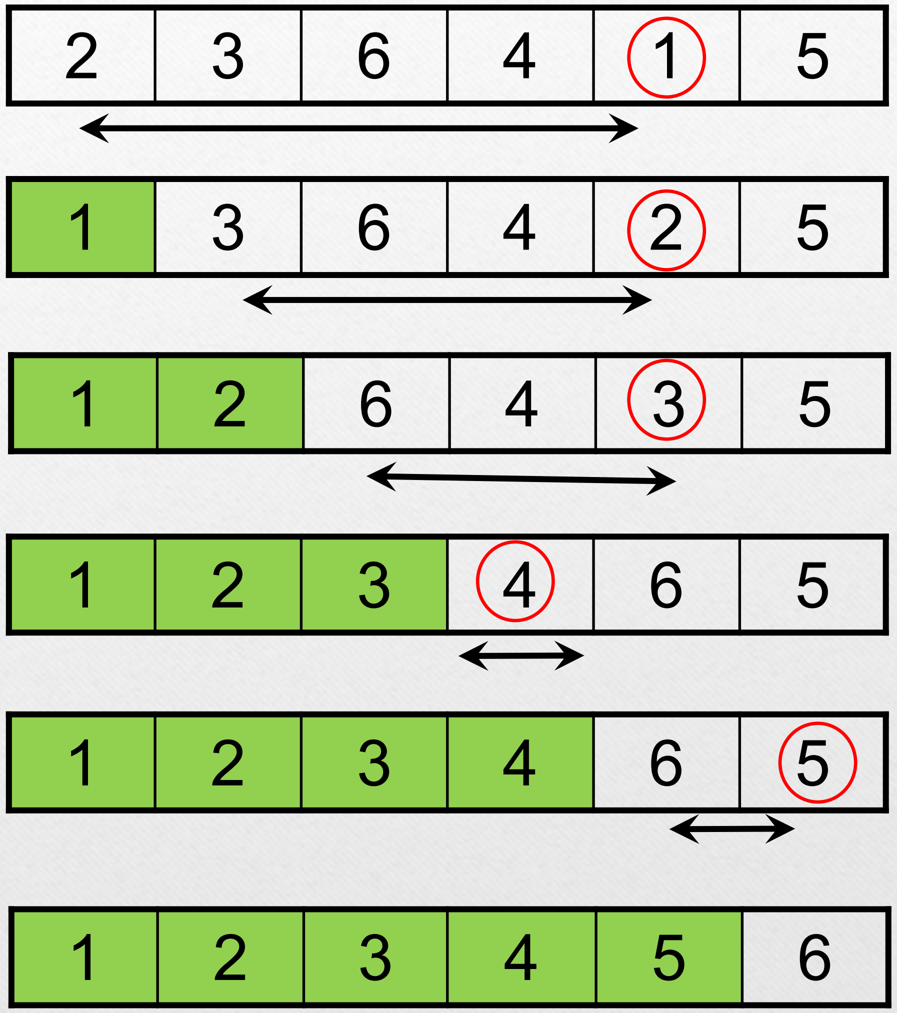
1. Sorting problem
   - Selection sort
   - Bubble sort
2. String matching
3. Optimization problem
   - Knapsack problem

Selection Sort

1. Scan the array to find the smallest element.
2. Swap it with the first element. (That index is now sorted)
3. Repeat for the second smallest element.
4. Generally: on pass *i*, find the smallest element in *A[i..n-1]* and swap it with *A[i]*.



Selection Sort



Selection Sort (pseudocode)

```
ALGORITHM  SelectionSort(A[0..n − 1])
    //Sorts a given array by selection sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]
                min ← j
        swap A[i] and A[min]
```

Efficiency? $O(n^2)$

Whiteboard

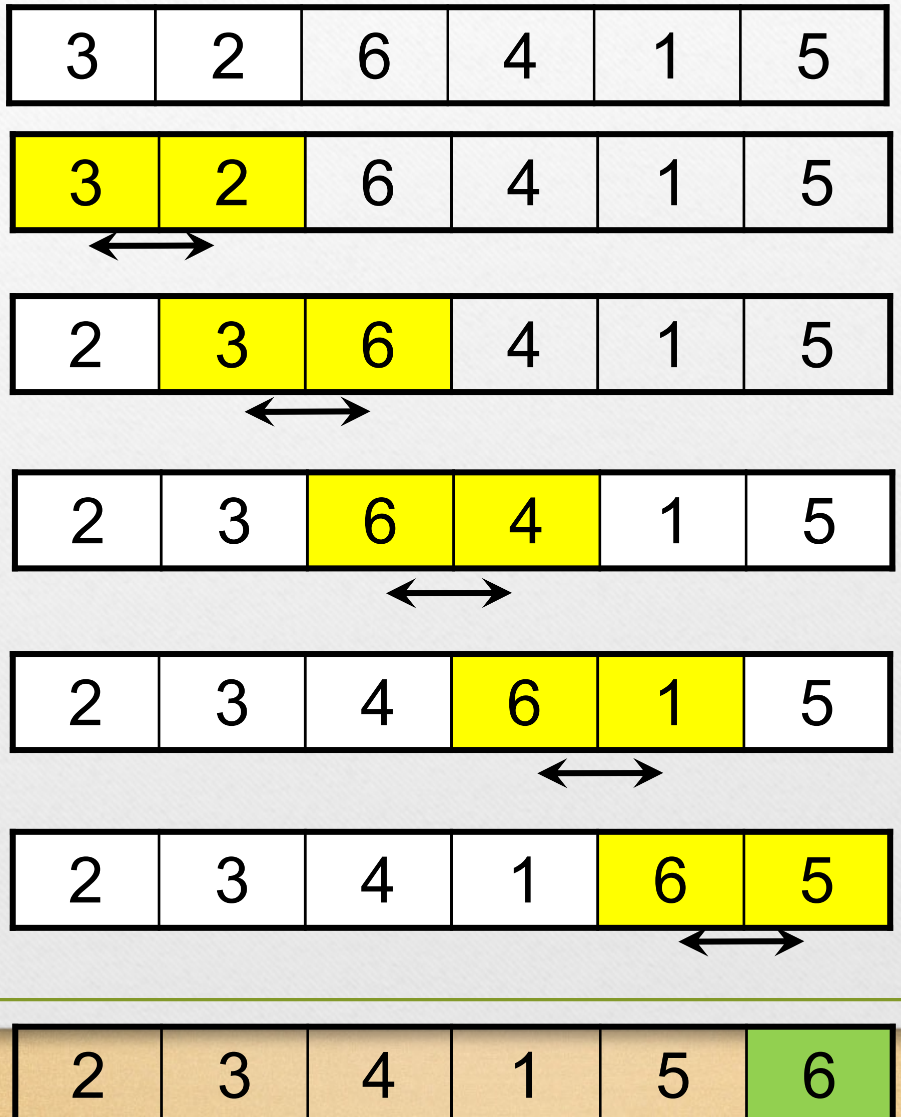
Brute Force

1. Sorting problem
   - Selection sort
   - Bubble sort
2. String matching
3. Optimization problem
   - Knapsack problem

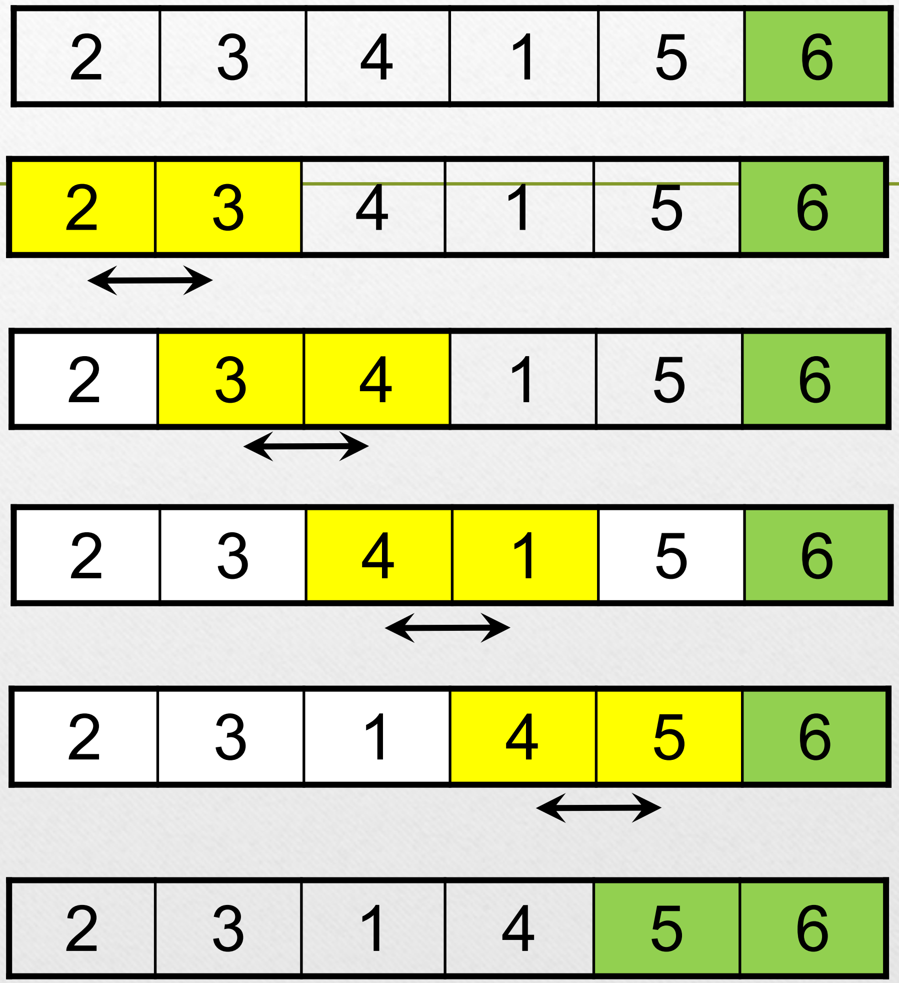
Bubble Sort

- The **bubble sort** makes multiple passes through a list. In each pass of the list, the algorithm compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

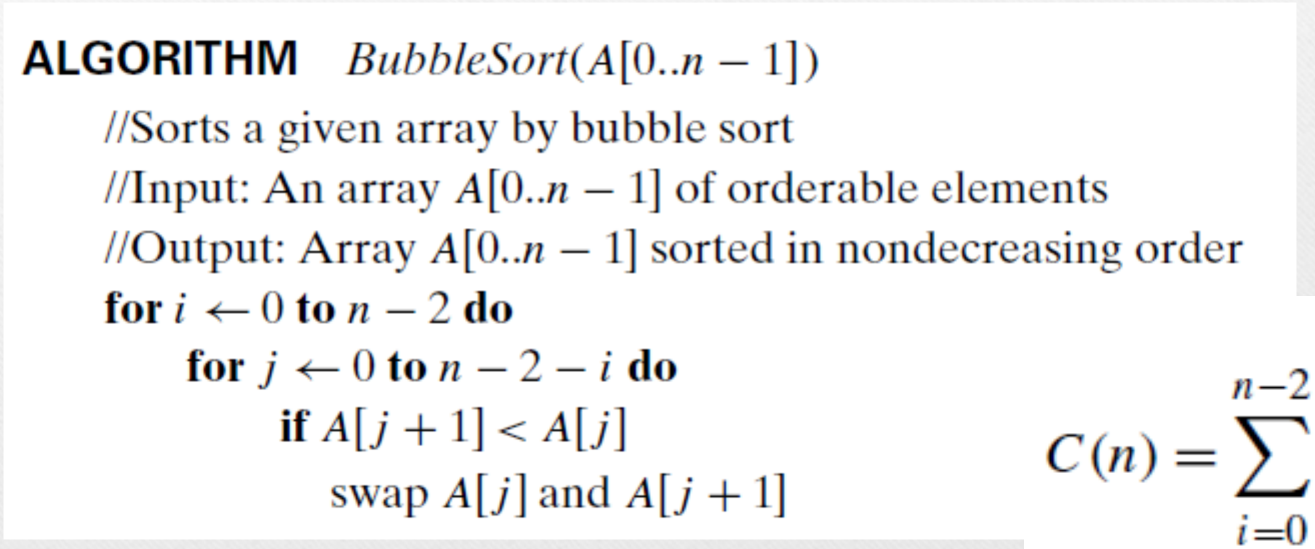
Bubble sort



Bubble sort



Bubble sort



```
ALGORITHM  BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for $i \leftarrow 0$ to $n - 2$ do
        for $j \leftarrow 0$ to $n - 2 - i$ do
            if $A[j + 1] < A[j]$
                swap $A[j]$ and $A[j + 1]$
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

Efficiency? O(n^2)

Bubble Sort

- What is it's performance based on input size?
  - What input would be the best case?
  - What input would be the worst case?
  - https://www.toptal.com/developers/sorting-algorithms/bubble-sort

Brute Force

1. Sorting problem
   - Selection sort
   - Bubble sort
2. String matching
3. Optimization problem
   - Knapsack problem

String Matching

Pattern : compress

Text : We introduce a general framework which is suitable to capture an essence of compressed pattern matching

String Matching

String Matching

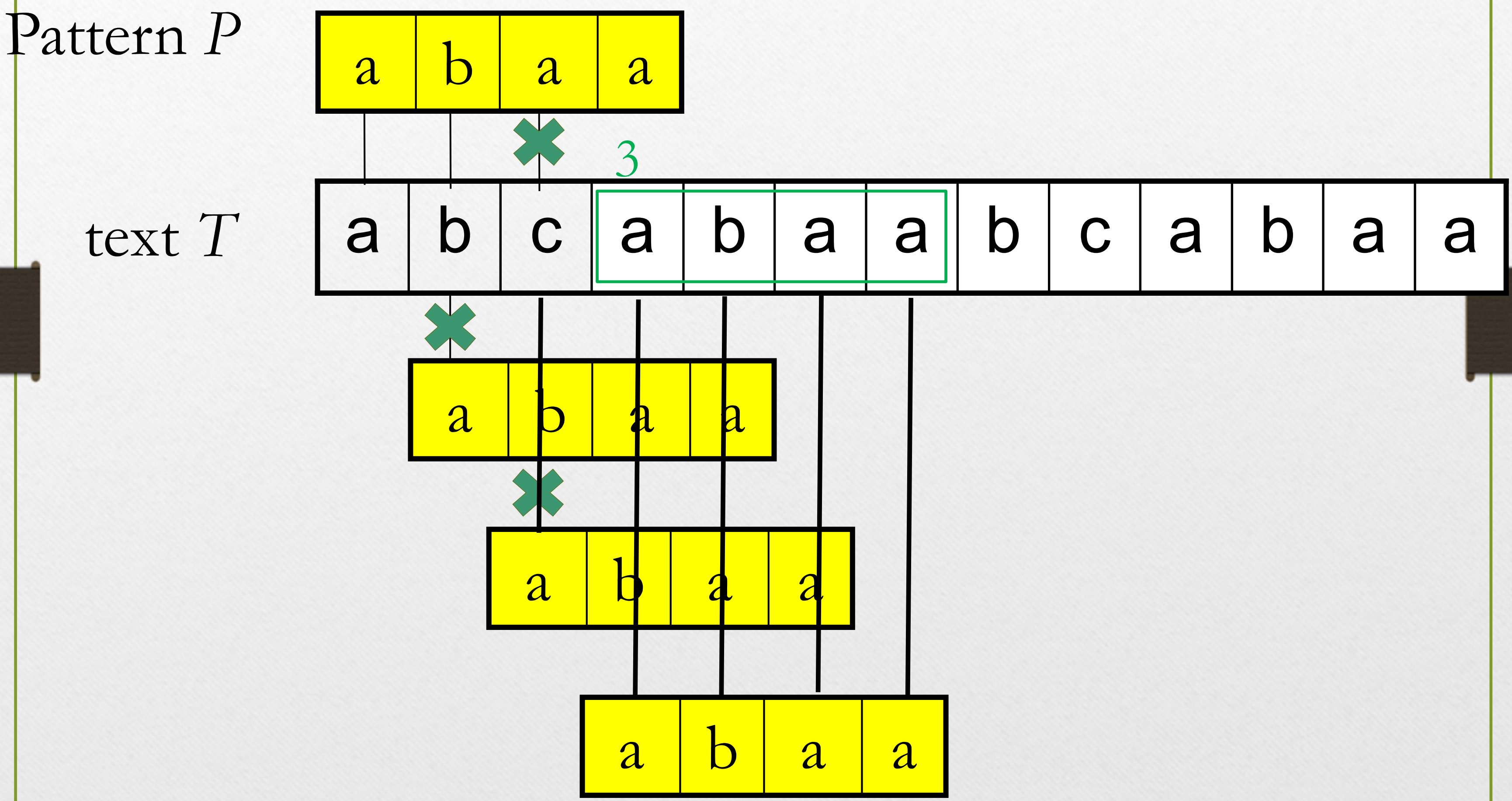
Input:

- *pattern:* A string of m characters to search for
- *text:* A longer string of n characters to search in

Problem:

Find a substring in the text that matches the pattern

String Matching

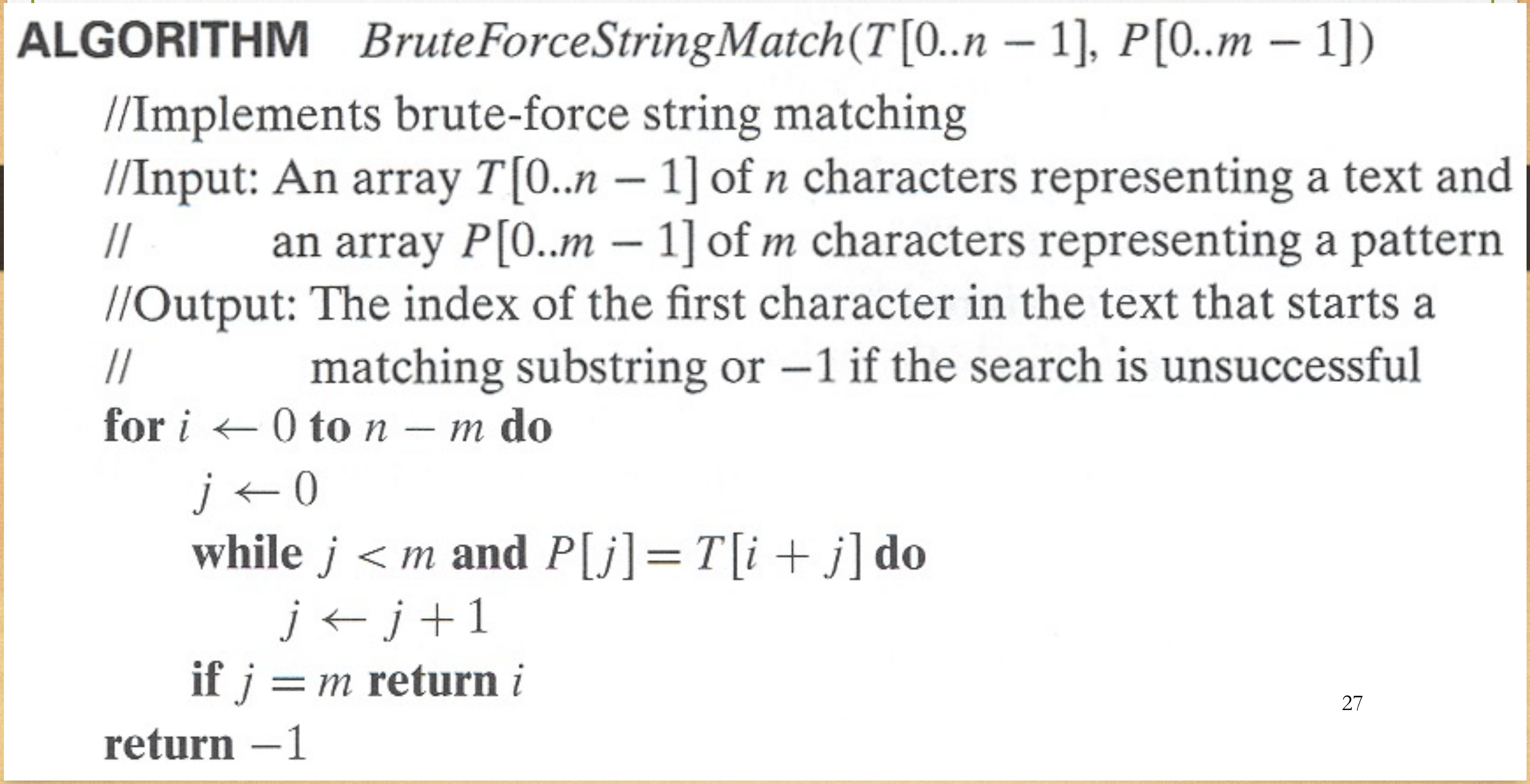


String Matching

<u>Brute-force algorithm</u>

1. Align pattern at beginning of text
2. Moving from left to right, compare each character of pattern to the corresponding character in text until
   - all characters are found to match (successful search); or
   - a mismatch is detected
3. While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

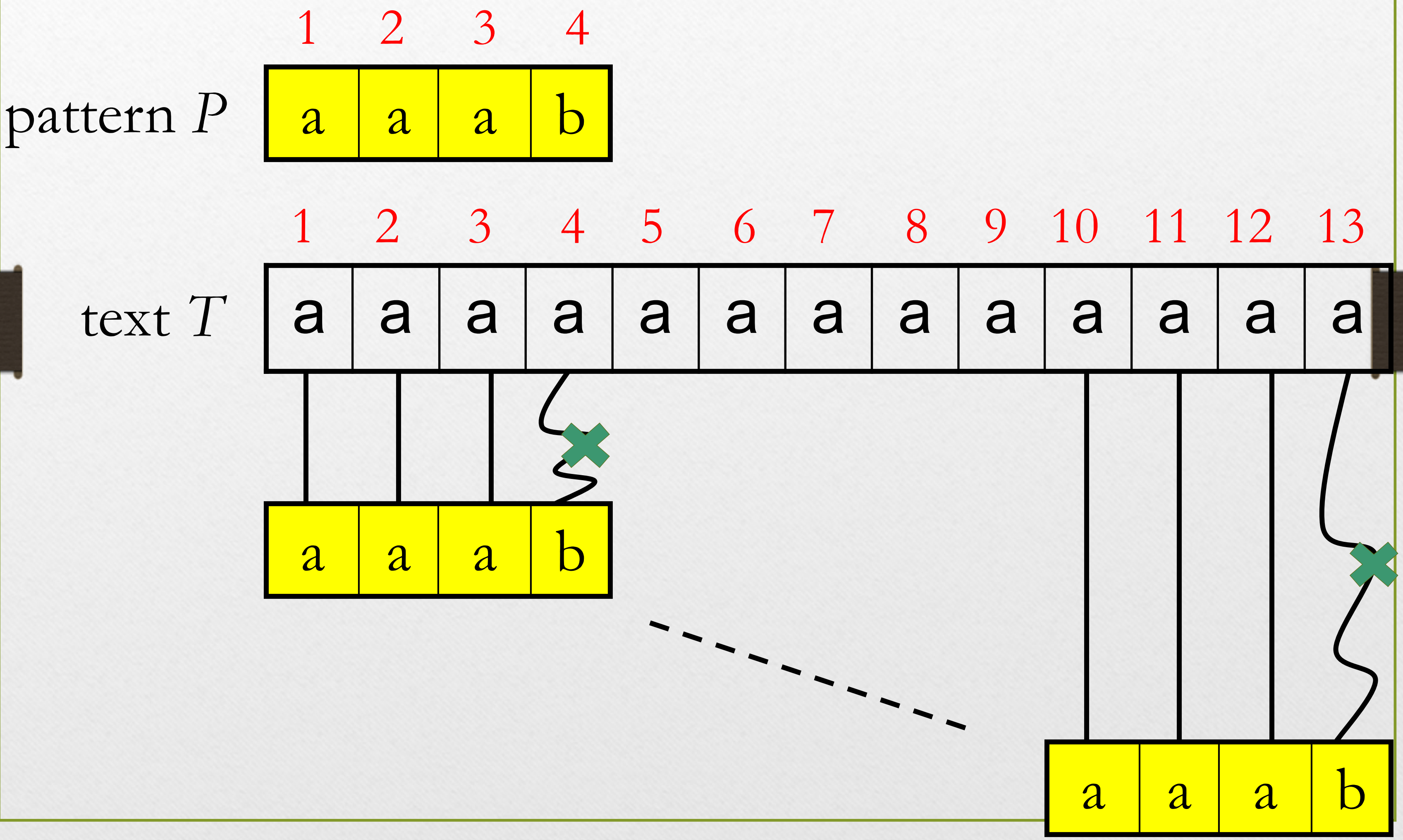
String Matching (pseudocode)



ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of $n$ characters representing a text and
//       an array $P[0..m-1]$ of $m$ characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or $-1$ if the search is unsuccessful
for $i \leftarrow 0$ to $n - m$ do
    $j \leftarrow 0$
    while $j < m$ and $P[j] = T[i + j]$ do
        $j \leftarrow j + 1$
    if $j = m$ return $i$
return $-1$
```

Analysis: Worst-case Example



Worst-case Analysis

- There are *m* comparisons for each shift in the worst case (inner loop)
- There are *n*-*m*+1 shifts (outer loop)
- So, the worst-case running time is:

  $O((n-m+1)m)$
- In the example on previous slide, we have (13-4+1)4 comparisons in total

Brute Force

1. Sorting problem
   - Selection sort
   - Bubble sort
2. String matching
3. Optimization problem
   - Knapsack problem

Brute Force

1. Sorting problem
   - Selection sort
   - Bubble sort
2. String matching
3. Optimization problems
   - Knapsack problem

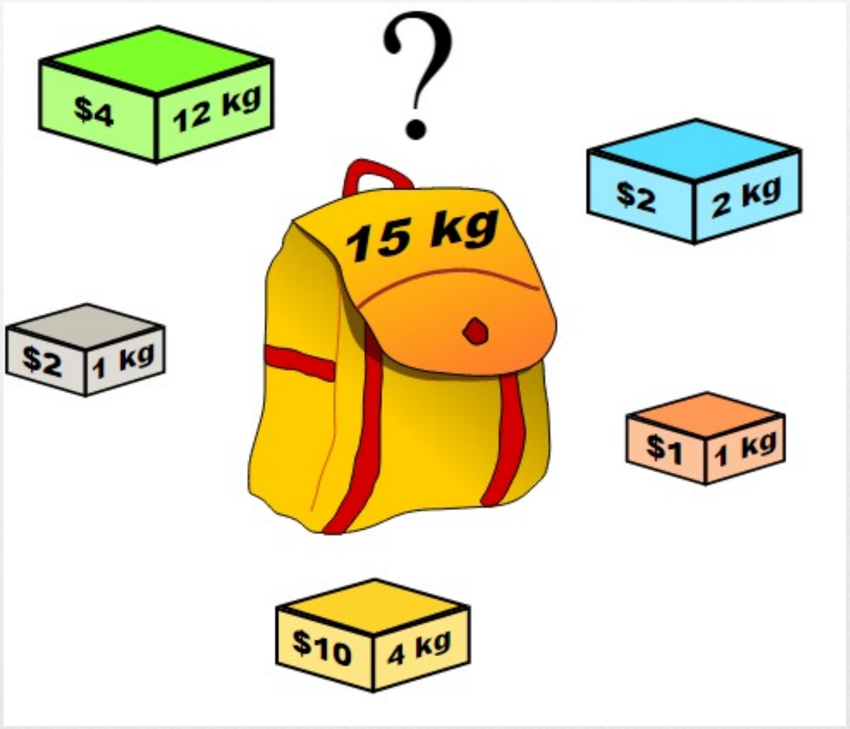
Brute Force for optimization problems

- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones, and keeping track of the best one found so far
- When search ends, announce the solution(s) found

Password Cracking

- What is the brute force mechanism for figuring out someone's password?
  - Guess every possibility
- What is the complexity of it?
  - m = number of characters to choose from (26 for lowercase alphabet
  - n = character length
  - Complexity = m^n
  - Password of 7 characters would take 26^7 (8,031,810,176) guesses

Knapsack Problem



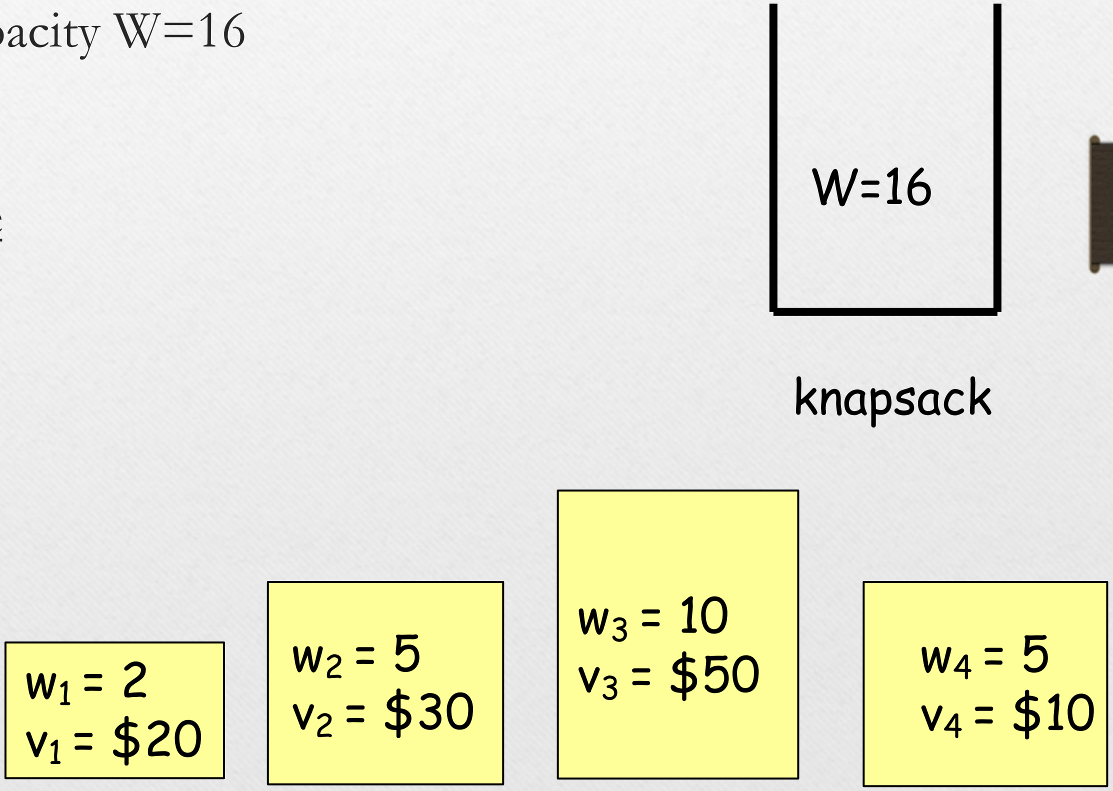
Knapsack Problem

- Input:
  - weights: $w_1 \quad w_2 \; \dots \; w_n$
  - values: $v_1 \quad v_2 \; \dots \; v_n$
  - a knapsack of capacity W
- Goal:
  - Find most valuable subset of the items that fit into the knapsack

Knapsack Problem

Example: Knapsack capacity W=16

item	weight	value
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10



Knapsack Problem

- Generate all possible subsets of the n items
- Compute total weight of each subset
- Identify feasible subsets
- Find the subset of the largest value

Knapsack Example

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Efficiency?

Need to generate *all subsets*. For n items, there are 2^n subsets. So this is a $O(2^n)$ algorithm.

Comments on brute force

- Brute force (Exhaustive-search algorithms) run in a realistic amount of time <u>only on very small instances</u>

- In many cases, exhaustive search or its variation is the only known way to get exact solution

B.F. Strengths and Weaknesses

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - matrix mult.
    - sorting
    - searching
    - string matching

- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Try it/homework

1. Chapter 3.1, page 102, questions 4, 8, 11
2. Chapter 3.1, page 107, question 5, 8