Lecture 2:
Brute Force Algorithms

COMP 3760

Textbook: 3.1, 3.2, 3.4

Today's agenda

- Brute force algorithms
- "Generate and test" subcategory
- Examples:
    - A couple of sorting algorithms
    - String matching
    - Optimization problems

Given a nonnegative integer N, return N! (factorial function)

Given a nonnegative integer N, return 2^N.

Find the combination to a bicycle lock

Brute force technique

- Direct, “obvious”, or straightforward algorithm
- The first thing you’d think of
- Try all possible solutions
- Nothing too clever
- No *over*-optimization
- “Just do it”

Important subcategory: Exhaustive search

"Generate and test":

1. Generate/enumerate all possible solutions
2. Test every one to see if it works
   - If only need *one* solution, end when you find one
   - May be looking for *all* solutions

Brute force technique

- Advantage: easy to understand, implement
- Disadvantage:
  - Maybe not the most efficient
  - But you can get lucky
- Sometimes called a *naïve* approach
  - But this is subjective and possibly misleading
  - A better word would be “straightforward”

Brute force examples

- What is the brute force solution for these problems:
  - Search for a key value in a list
  - Computing n!
  - Computing a^n ($a > 0$, n is a nonnegative integer)
  - Find the combination to a bicycle lock with 4 numbers
  - Given an unsorted list of numbers, find the two that have the largest product

Selection Sort

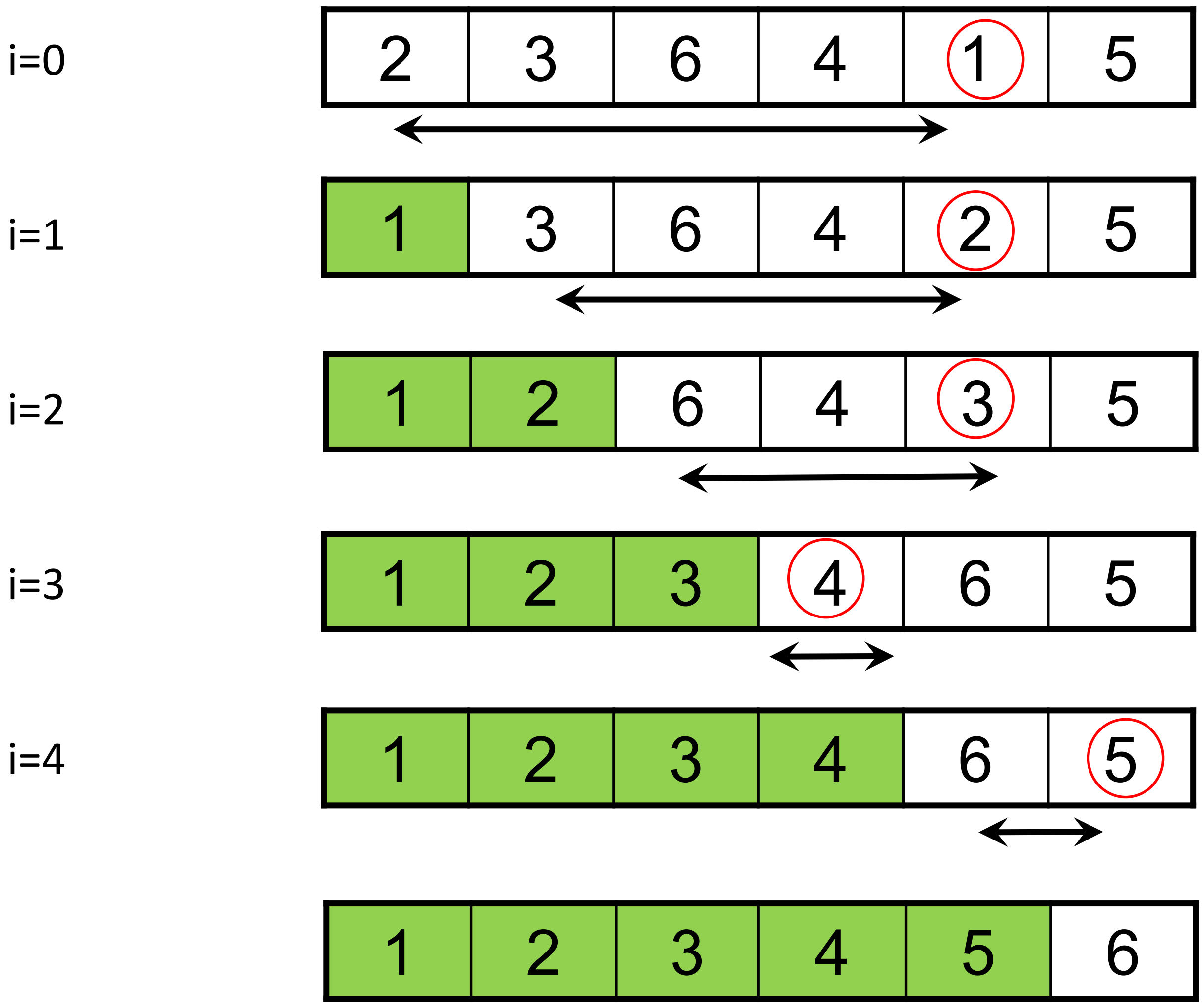
Selection sort – general idea

- “Sorted part” and “unsorted part”
- At each iteration:
    - Find the smallest item remaining in the unsorted part
    - Move it to the end of the sorted part

Selection sort: idea

- Scan the whole array to find the smallest element.
- Swap it with the 1st element (A[0]).
    - Scan A[2..n-1] for the smallest element
    - Swap it with the 2nd element (A[1])
        - Scan A[3..n-1] for the smallest element
        - Swap it with the 3rd element (A[2])
- 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 $i \leftarrow 0$ to $n-2$ do
        $min \leftarrow i$
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[j] < A[min]$
                $min \leftarrow j$
        swap $A[i]$ and $A[min]$
```

Efficiency? $O(n^2)$

Why is this brute force?

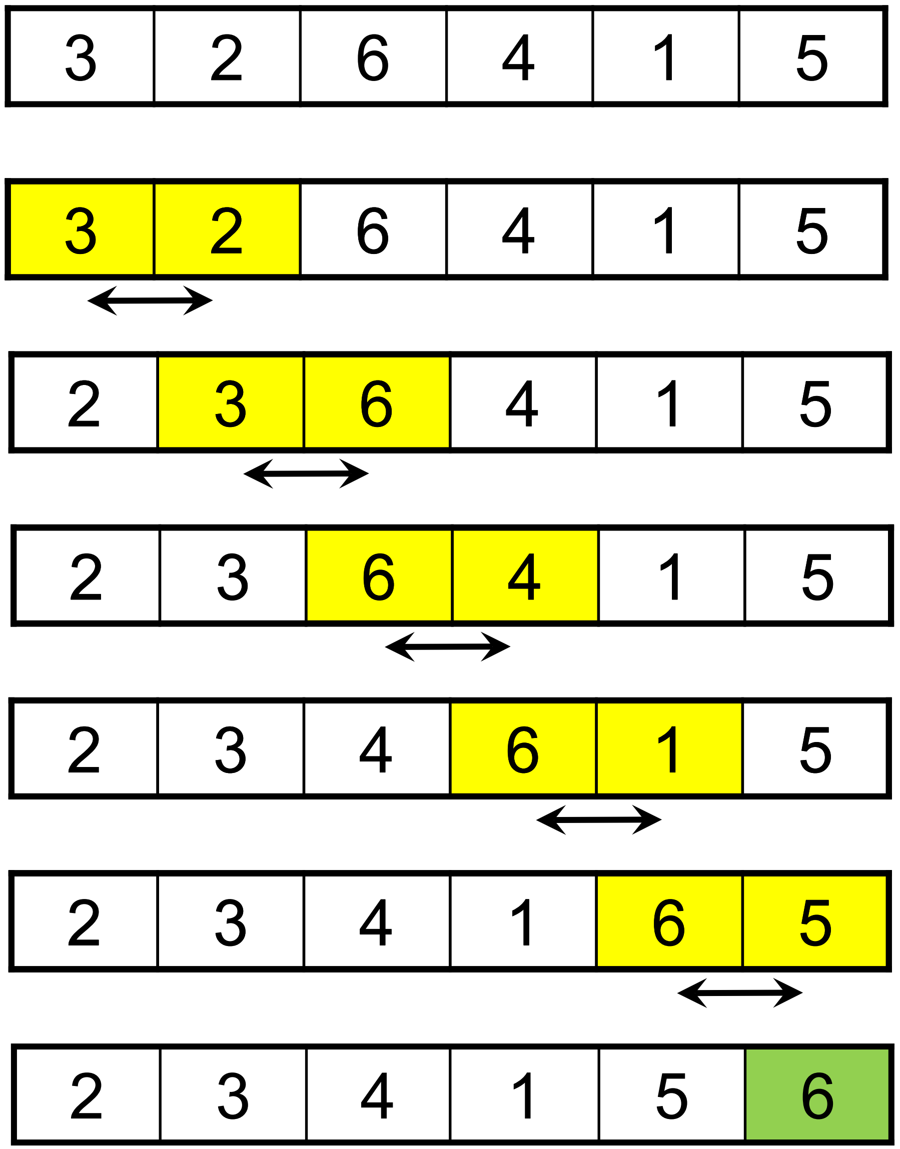
- Each main iteration involves a *sequential search* (aka linear search) for the smallest remaining item

Bubble Sort

Bubble sort – general idea

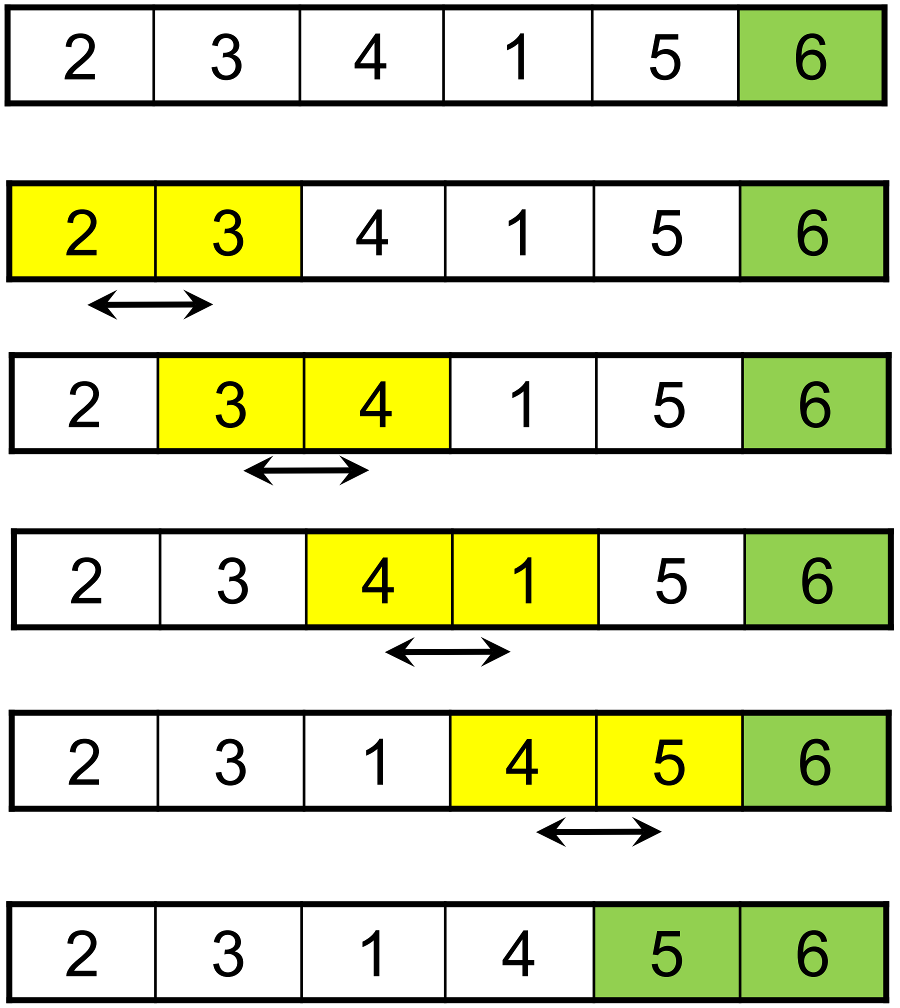
- “Sorted part” and “unsorted part”
- At each iteration:
  - Run a “bubble” across the unsorted part
  - The largest (remaining) element bubbles to the end

Bubble sort



First iteration i=0

Bubble sort



Second iteration i=1

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? $O(n^2)$

String matching

String matching

Pattern : compress

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

The String Matching Problem

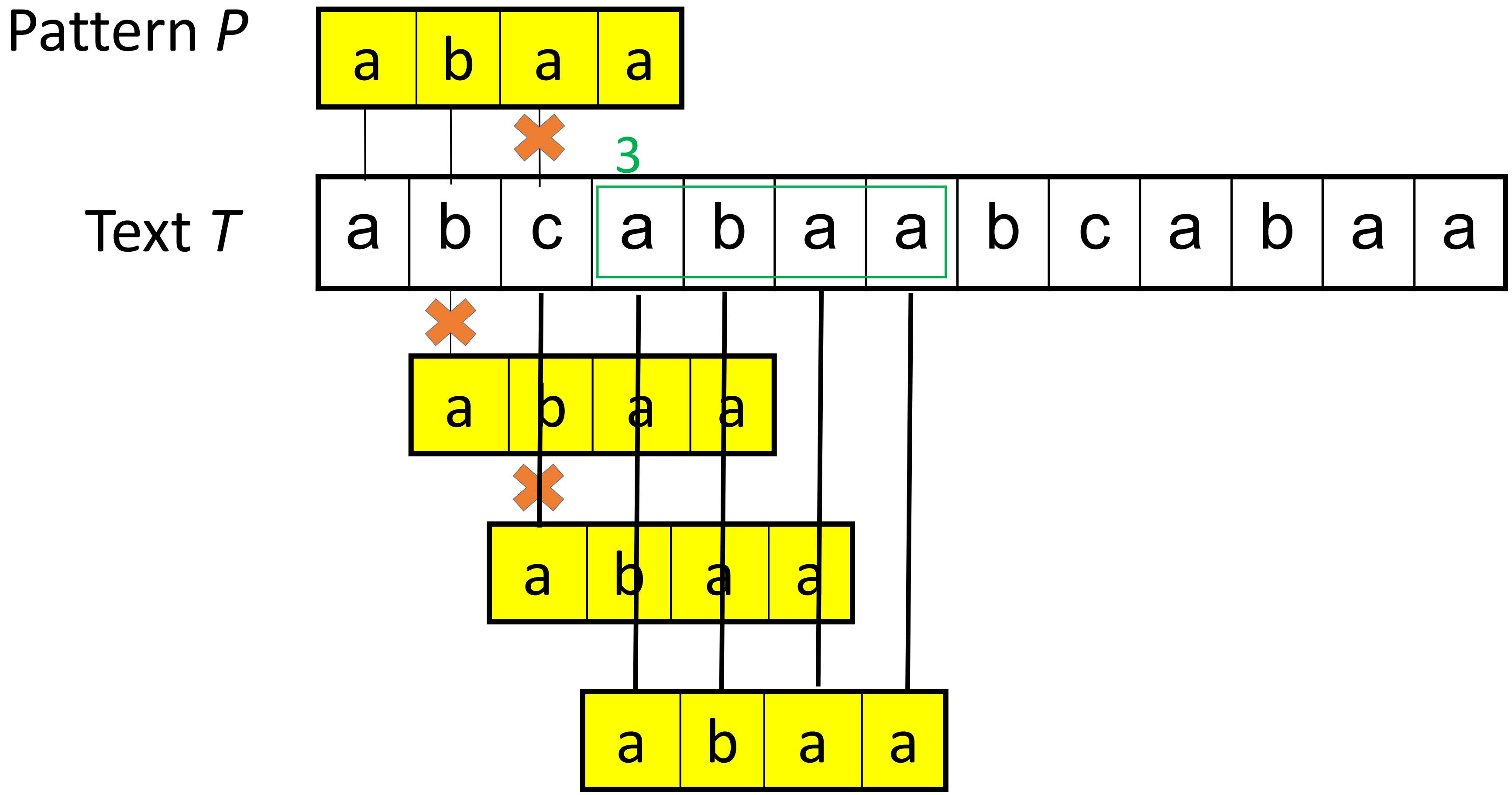
Input:

- *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

The String Matching Problem



Solution (in words)

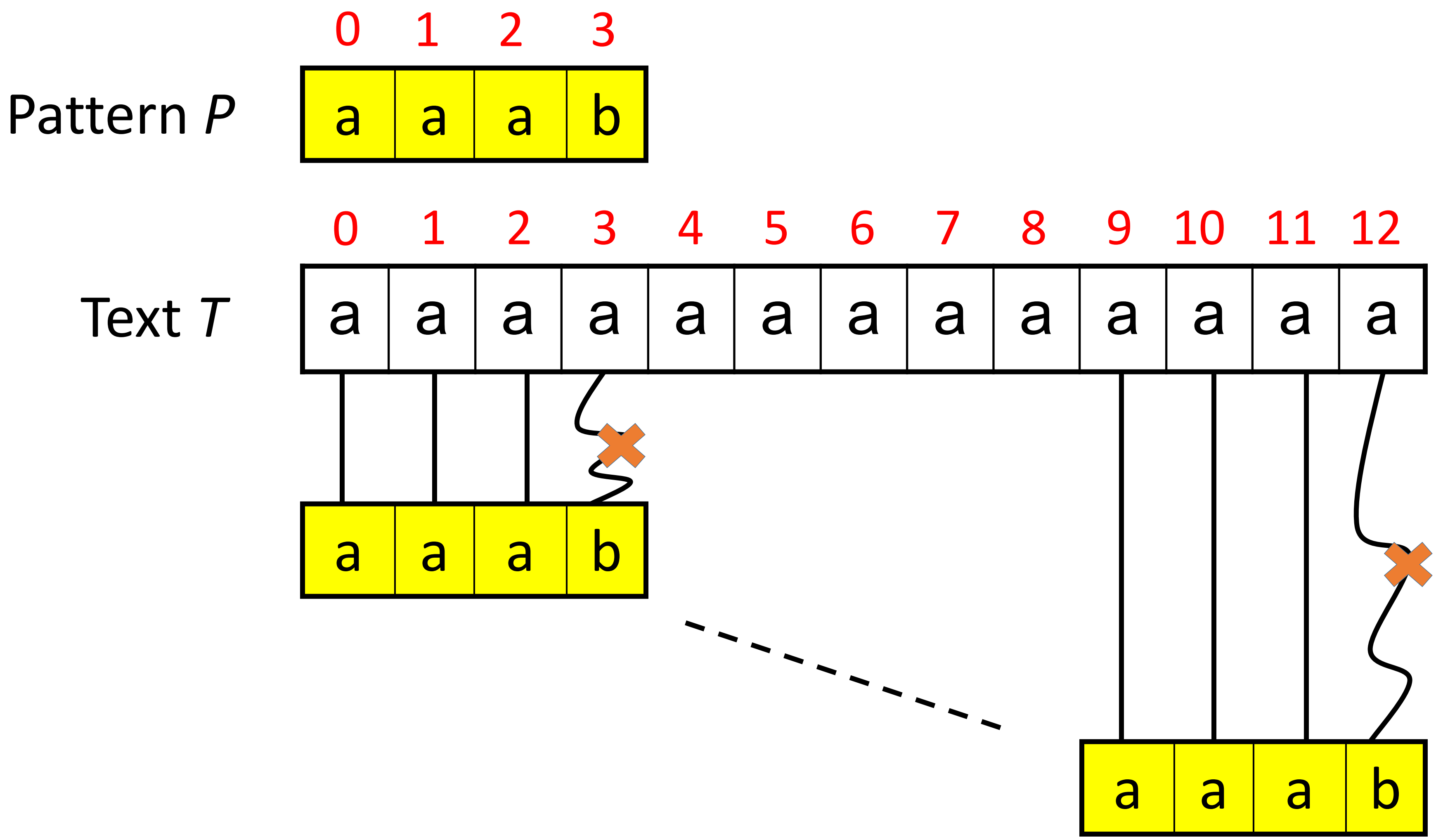
<u>Brute-force algorithm</u>

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:

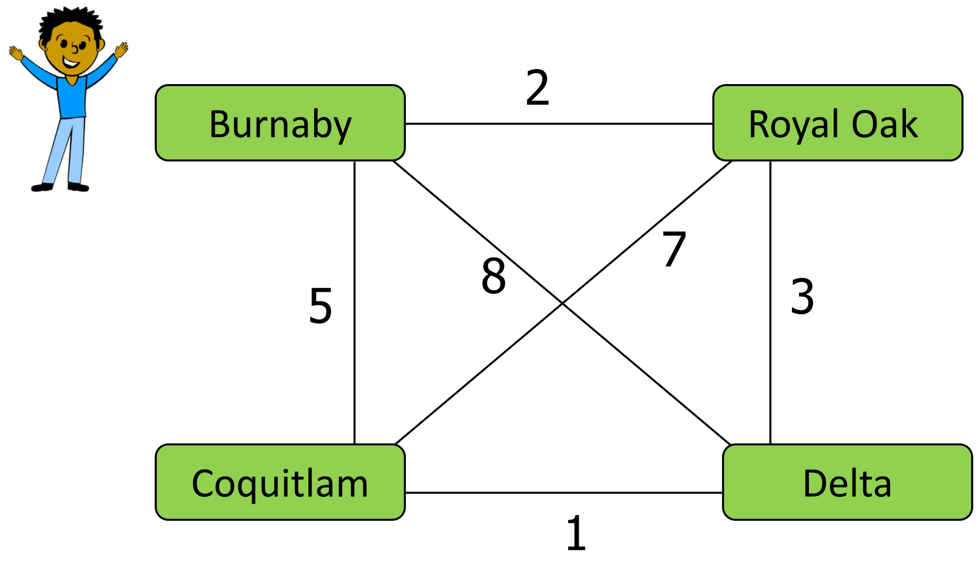
  O((n-m+1)*m)

- In the example on previous slide, we have (13-4+1)*4 comparisons in total

Traveling salesperson problem

Traveling salesperson problem

- A salesperson needs to visit n cities. You know the distance between any two cities. *Find the shortest path that visits each city exactly once and returns to the starting city*.
- Note that this has lots of applications in real life.

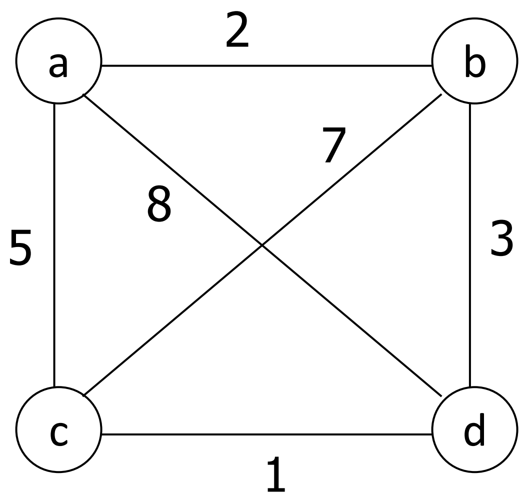


Brute force for “optimization problems”

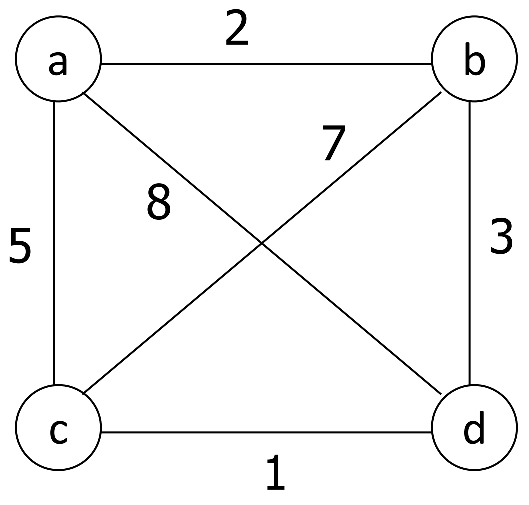
- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones, and keeping track of the best one found so far
- When search ends, announce the solution(s) found

Traveling salesperson problem (aka TSP)

- **Abstract model:**
  - A weighted graph.
  - Hamiltonian circuit: a circuit that visits every node exactly once
- **Goal:**
  - Find lowest cost Hamiltonian circuit



Traveling salesperson problem



- Assume our salesperson starts at city a. then one possible solution would be ...

  a→b→c→d→a

  ... and the length of this route is

  L = 2+7+1+8 = 18

- Other possible solutions include ...

a→b→d→c→a	**L = 2+3+1+5 = 11**
a→c→b→d→a	**L = 5+7+3+8 = 23**
a→c→d→b→a	**L = 5+1+3+2 = 11**
a→d→b→c→a	**L = 8+3+7+5 = 23**
a→d→c→b→a	**L = 8+1+7+2 = 18**

a→b→c→d→a
a→b→d→c→a
a→c→b→d→a
a→c→d→b→a
a→d→b→c→a
a→d→c→b→a

TSP example (cont'd)

Are there more tours?

No. We have to start and end at a. Any possible order of b, c, d must be considered. We have all of them listed.

Are some tours redundant?

Yes. We have counted each route twice (in each direction). We need only check half of the routes listed.

Efficiency:

The time consuming task is finding all the different permutations of b, c, d. For three things there are 3!=6. But for a big map, this is costly.

TSP efficiency

- How many possible routes – for large n?
- Remember that since we are always starting and ending at a specific city (eg: a), we only need to consider routes that start with ‘a’
  - ie: we would consider **a→b→d→c→a** but not **b→d→c→a→b**
  - this means there are only **(n-1)! permutations** to consider
- but we also notice that there are some duplicate routes, eg: **a→b→d→c→a** is the same as **a→c→d→b→a** (it is just reversed)
  - so we only consider one of them
  - every route has a reverse path, so ...
- the brute force solution requires that we generate and compute the length of (n-1)!/2 routes

TSP brute-force solution

```
for each permutation P of cities
  for each city i in P
    length ← length + weight(i,i+1)
  if length < min
    min ← length
    minroute ← P
return minroute
```

Efficiency? (n-1)!/2 = O(n!)

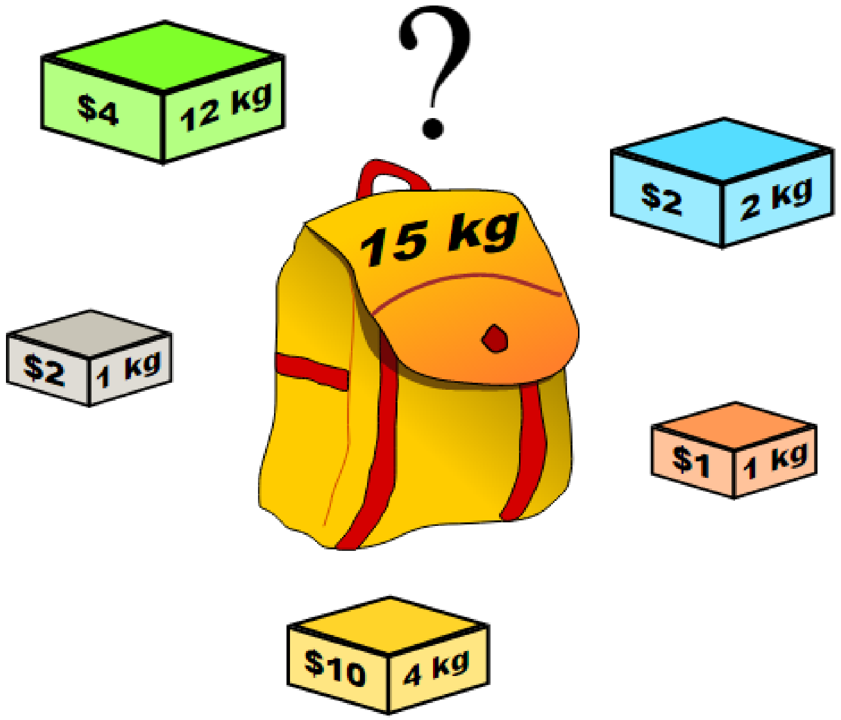
How long does it take?

- For 4 cities: 3 tours
- For 5 cities: 12 tours
- For 10 cities: 181,440 tours
- For 20 cities: 60,822,550,204,416,000 tours
- For 37 cities: 1.860×10^{41}
- For 100 cities: 4.666×10^{155}

Brute force solution **is not feasible** except for very small problems

Knapsack Problem

Knapsack problem



Knapsack problem

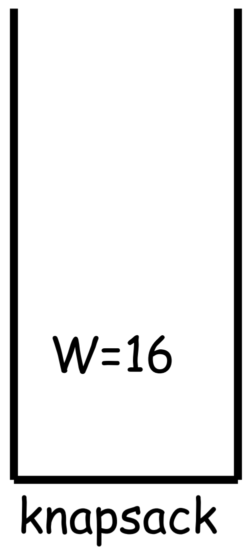
- Input:
  - weights: $w_1 \quad w_2 \ldots \; w_n$
  - values: $v_1 \quad v_2 \ldots \; v_n$
  - a knapsack of capacity W

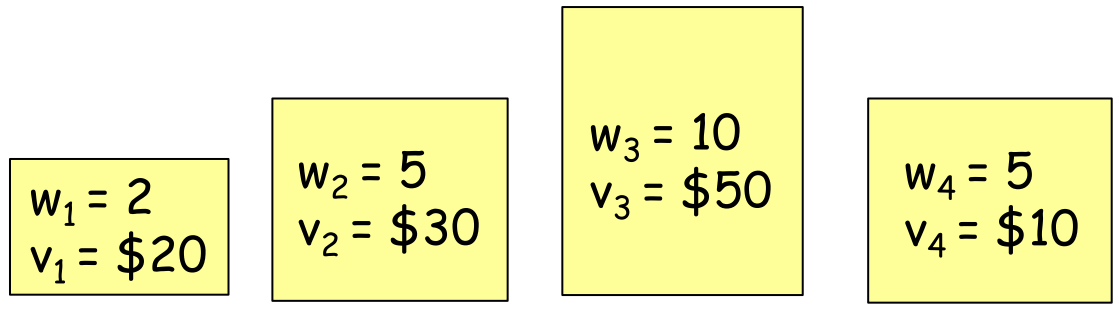
- Goal:
  - Find most valuable subset of the items that fit into the knapsack

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

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

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

best feasible subset

Efficiency?

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

Assignment Problem

Assignment problem

- There are n people who need to be assigned n jobs, and there is a (possibly different) cost for each person to do each job
- This is a classic optimization problem
- The problem is to find the combination of people and jobs that has the minimum (or maximum) overall cost

Assignment problem

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

for example, this means that the cost is "8" for person 3 to do job 2

Goal:
Find the combination of people and jobs that has the minimum overall cost

Assignment problem

- Possible solution 1: <1, 2, 3, 4>

	Job 1	Job 2	Job 3	Job 4
Person 1	[✓ 9]	2	7	8
Person 2	6	[✓ 4]	3	7
Person 3	5	8	[✓ 1]	8
Person 4	7	6	9	[✓ 4]

- in this case the total cost is 9+4+1+4 = 18

Assignment problem

- Another possible solution: <2, 1, 3, 4>

	Job 1	Job 2	Job 3	Job 4
Person 1	9	$\bigcirc$ 2	7	8
Person 2	$\bigcirc$ 6	4	3	7
Person 3	5	8	$\bigcirc$ 1	8
Person 4	7	6	9	$\bigcirc$ 4

in this case the total cost is 6+2+1+4 = 13

... and another possible solution: <2, 3, 1, 4>

	Job 1	Job 2	Job 3	Job 4
Person 1	9	$\bigcirc$ 2	7	8
Person 2	6	4	$\bigcirc$ 3	7
Person 3	$\bigcirc$ 5	8	1	8
Person 4	7	6	9	$\bigcirc$ 4

in this case the total cost is 5+2+3+4 = 14

Assignment problem

- Brute force algorithm:
  - Check every combination of assignments
  - Calculate the cost of each one
  - Find the combination with minimum cost

```
for each permutation P of job assignments
    totalcost ← sum of the job costs for P
    if totalcost < mincost
        mincost ← totalcost
        minperm ← P
return minperm
```

Efficiency?

Need to generate *all permutations*. For n jobs, there are n! permutations. So this is an O(n!) algorithm.

Comments on brute force

- Brute force (exhaustive-search) algorithms run in a realistic amount of time only on very small inputs

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

B.F. Strengths and Weaknesses

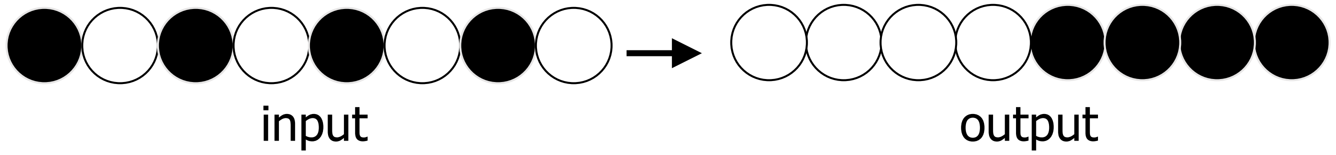
- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - matrix mult.
    - sorting
    - searching
    - string matching

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

Practice problem

You have a row of 2n discs of 2 colors, n dark and n light. They alternate dark, light, dark, light, dark, and so on. You want to get all the dark discs on the right hand side and all the light discs on the left.

The only move you are allowed to make is to interchange two adjacent discs.



(a) Design an algorithm that solves this problem

(b) Analyze the efficiency of your algorithm

Practice problems

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