Brute Force

Agenda

1. Overview
2. Sorting
3. Sequential Search & String Matching
4. Convex Hull (*)
5. Graph Search
6. Exhaustive Search
7. Summary

1. Overview

Learning objectives

1. Understand the ***Brute Force*** approach
2. Understand and apply:
   - Sorting – Selection and Bubble Sorts
   - Sequential Search and String Matching
   - Computational geometry – Convex hull problems
   - Graph Search
   - Exhaustive Search

Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Examples:

1. Computing a^n (multiply ‘a’ for n times)
2. Searching for a key of a given value in an unsorted list
3. Calculate the sum of a range

2. Sorting

Sorting

- Examples
  - Sorting Numbers (Sequential)
  - Telephone book – by surname, e.g., A, B, C, D…
  - Sorting Books in Library (Dewey system)
  - Sorting Individuals by Height (Feet and Inches)
- Why do we study sorting?
  - Important to build efficient searching algorithms and data structures, data compression.
  - Heavily studied problem in computer science, with several widely celebrated algorithms.

Sorting

Formal definition:

- Given a sequence of n elements $x_1, x_2, \ldots, x_n \in S$
- Re-arrange the elements according to some ordering criteria.

Example

A = { 3 1 6 2 1 3 4 5 9 0 }

Sorted(A) = { 0 1 1 2 3 3 4 5 6 9 }

Sorting Algorithms

- There are many sorting algorithms, such as:
    - Selection Sort
    - Insertion Sort
    - Bubble Sort
    - Merge Sort
    - Quick Sort
- The first three are the foundations for faster and more efficient algorithms

Brute Force: Selection Sort

Selection Sort is a Brute Force solution to the sorting problem.

1. Scan all n elements of the array to find the ***smallest element***, and ***swap*** it with the ***first element***.
2. Starting with the ***second element***, scan the remaining $n-1$ elements to find the smallest element and swap it with the element in the second position.
3. Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i \ldots n-1]$ and swap it with $A[i]$.

Selection Sort

Sorted **Unsorted**

23	78	45	8	32	56	Original List
8	78	45	23	32	56	After pass 1
8	23	45	78	32	56	After pass 2
8	23	32	78	45	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Selection Sort

- The list is divided into two sub-lists, *sorted* and *unsorted*, which are divided by an imaginary wall
- We find the smallest/largest element from the unsorted sub-list and swap it with the element at the beginning of the unsorted data
- After each selection and swapping, the imaginary wall between the two sub-lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones
- Each time we move one element from the unsorted sub-list to the sorted sub-list, we say that we have completed a sort pass
- A list of *n* elements requires *n-1* passes to rearrange the data

Selection Sort – Pseudocode

ALGORITHM **SelectionSort** ($A[0 \ldots n-1]$)
/* Order an array using a brute-force selection sort. */
/* INPUT : An array $A[0 \ldots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \ldots n-1]$ sorted in ascending order. */

```
for i = 0 to n − 2 do
    min = i
    for j = i + 1 to n − 1 do
        if A[j] < A[min] then
            min = j
        end if
    end for
    swap A[i] and A[min]
end for
```

No need to sort the last value
Record position

Record position of the new smallest candidate

Selection Sort - Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons)

  → So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves
- In selection sort, the outer “for” loop executes n-1 times
- We invoke swap function once at each iteration

  → Total Swaps: n-1

  → Total Moves: 3*(n-1) (Each swap has three moves)

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.

  ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2

  ➔ **So, Selection sort is $O(n^2)$**
- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **$O(n^2)$**
  - The behavior of selection sort algorithm does not depend on the initial organization of data
  - Since $O(n^2)$ grows rapidly, selection sort is appropriate only for small n
  - Although selection sort algorithm requires $O(n^2)$ key comparisons, it only requires $O(n)$ moves
  - Selection sort could be a good choice if data moves are costly but key comparisons are not

Selection Sort – Time Complexity

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

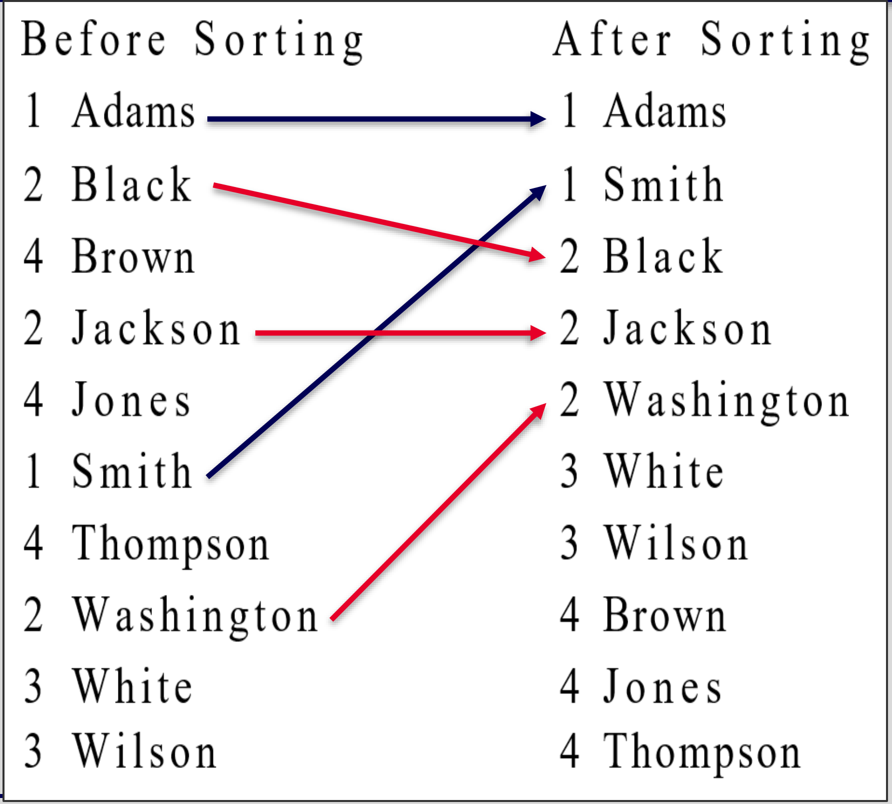
- Needs around $n^2 / 2$ comparisons and at most $n - 1$ exchanges.
- The running time is ***insensitive*** to the input, so the best, average, and worst case are essentially the same
  - Why?

Selection Sort – Why use it

- Selection sort only makes O(n) writes but O(n^2) reads.
- When writes (to array) are much more expensive than reads, selection sort may have an advantage, e.g., flash memory.
- Also, for small arrays (10 – 20 elements), selection sort is relatively efficient and simple to implement.

‘Stable’ Sorting Algorithms

- A sorting method is ***stable*** if it preserves the relative order of duplicate keys in the file.
- Why? Think about sorting last names and first names separately
- Not all sorting methods are stable.



Is Selection Sort Stable?

- Question: is Selection Sort stable?
- Consider the following example and apply Selection Sort on them:

5, 5, 3, 2

2, 5, 3, 5

2, 3, 5, 5

Another Brute Force sort…

Bubble Sort

- A **bubble sort** iteratively ***compares adjacent items*** in a list and ***swaps*** them if they are out of order.

Bubble Sort – Motivation

- One of the classic (and elementary) sorting algorithms, originally designed and efficient for tape disks, but with random access memory, it doesn't have much use these days.
- But insightful to study it and to understand why other sorting algorithms are superior in one or more aspects.
- It is simple to code.

Bubble Sort – Idea

- First iteration, ***compare each adjacent pair*** of elements and swap them if they are out of order. Eventually largest element gets propagated to the end.
- Second iteration, repeat the process,
  - but only ***from first to 2nd last element*** (last element is in its correct position). Eventually second largest element is at the 2nd last element.
- Repeat until all elements are sorted.

Bubble Sort

23	78	45	8	32	56	Original List
23	45	8	32	56	78	After pass 1
23	8	32	45	56	78	After pass 2
8	23	32	45	78	56	After pass 3
8	23	32	45	56	78	After pass 4

Bubble Sort

- The list is divided into two sub-lists: unsorted and sorted
  - Bubble sort compares adjacent integers and exchanges them if they are out of order
  - The largest element is bubbled from the unsorted list and moved to the sorted sub-list
  - After that, the wall moves one element backwards, increasing the number of sorted elements and decreasing the number of unsorted ones
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data

Bubble Sort – Pseudocode

ALGORITHM **BubbleSort** $(A[0 \dots n-1])$
/* Order an array using a bubble sort. */
/* INPUT : An array $A[0 \dots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \dots n-1]$ sorted in ascending order. */

```
for i = 0 to n − 2 do
    for j = 0 to n − 2 − i do
        if A[j + 1] < A[j] then
            swap A[j] and A[j + 1]
        end if
    end for
end for
```

Traverse to 2nd last element only

Keep traversing and swapping but leave the sorted part alone

Bubble Sort – Time Complexity

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

$$= \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

Bubble Sort – Time Complexity

- **Best case**: if original file is already sorted
  - about $n^2/2$ comparisons & 0 exchanges – $O(n^2)$.
- **Worst case**: if original file is sorted in reverse order
  - about $n^2/2$ comparisons & $n^2/2$ exchanges – $O(n^2)$.
- **Average case**: if original file is in random order
  - about $n^2/2$ comparisons & less than $n^2/2$ exchanges – $O(n^2)$.

Is bubble sort stable?

Improved Bubble Sort

- This modification attempts to reduce redundant iterations, by ***checking if any exchanges takes place*** in each pass.
- If there were no exchanges in the current iteration, the sorting is stopped after the current iteration.
- Early-Termination Bubble Sort

Improved Bubble Sort – Complexity

- **Best case** - when the original file is already sorted, only one pass is needed, $n-1$ comparisons, 0 exchanges – $O(n)$.
- **Worst case** - No improvement over the original implementation – $O(n^2)$.
- **Average case** - Depending on the data set, few iterations can be eliminated at the end of the sort.
- Therefore, the number of passes is less than $n-1$, and hence cost is lower than the original implementation. The complexity is still likely to be $O(n^2)$.

3. Sequential Search & String Matching

Sequential Search

- Sequential search or linear search involves scanning each element of the entire collection sequentially until the key is found

```
Algorithm SequentialSearch(A[0…n],K)
    i = 0;
    while (i < n && A[i] != K)
        i = i + 1;
    if i < n return i
    else return -1
```

Sentinel Technique

- A sentinel value is a special value in the context of an algorithm that uses its presence as a condition of termination, typically in a loop or recursive algorithm. (Wikipedia)

```
Algorithm SequentialSearch(A[0…n],K)
    i = 0
    A[n] = k // increase the array by 1
    while (A[i] != K)
        i = i + 1;
    if i < n return i
    return -1
```

Sequential Search – Analysis

Case	Best	Average	Worst
Item is present	1	n/2	n
Item is not present	n	n	n

What if we have an ordered list?

We can early-terminate a sequential search too

String Matching

- Given a string of ***n*** characters called the **text** and a string of ***m*** characters (m<=n) called the **pattern**, find a substring of the text that matches the pattern.
- For example:
  - “RMIT IS THE BEST UNIVERSITY” (text)
  - “UNIV” (pattern)

String Matching – Idea

- Align the pattern against the first m characters of the text.
- Start matching the corresponding pair of characters from left to right until either all the m pairs are matched,
- Or if the missing pair is found, the pattern is shifted one position to the right and character comparisons are resumed, starting again from the 1st character.

String Matching – Idea

RMIT IS THE BEST UNIVERSITY UNIV
RMIT IS THE BEST UNIVERSITY   UNIV
RMIT IS THE BEST UNIVERSITY   UNIV
RMIT IS THE BEST UNIVERSITY   UNIV
…
RMIT IS THE BEST UNIVERSITY   UNIV

String Matching – Pseudocode

Algorithm BruteForceStringMatch (T[0…n-1],P[0…m-1])

```
for i = 0 to n-m
    j = 0
    while j < m and P[j] = T[i+j] do
        j = j + 1
    if j = m return i
return -1
```

String Matching – Analysis

- **Worst case:** The algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n-m+1 tries, i.e., m(n-m+1). Therefore, the worst case is **O(nm)**.

- For example:

  - Text = AAA…AAAAH

  - Pattern = AAAAH

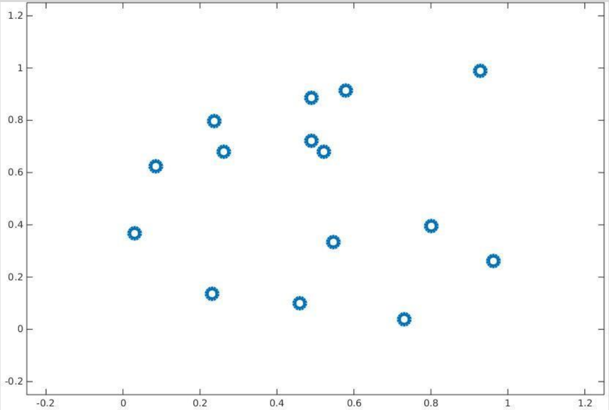
String Matching – Analysis

- **Average Case:** O(n+m) = O(n).
- **Best Case:** O(m) (if m is found in first place–m comparisons are needed), or O(n) (if m is not found–check n times)
- For example:
  - Text = AAA…AAAAH
  - Pattern = AAA or BBB

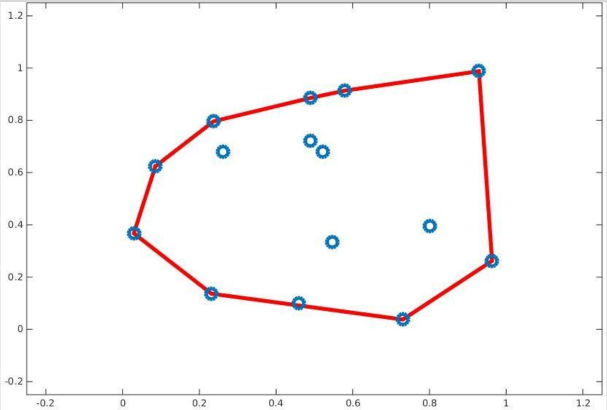
4. Convex Hull (*)

Convex hull problem

- The convex hull of a set of points is ***the smallest convex polygon that contains all the points***, i.e., all the points are "within" the polygon.



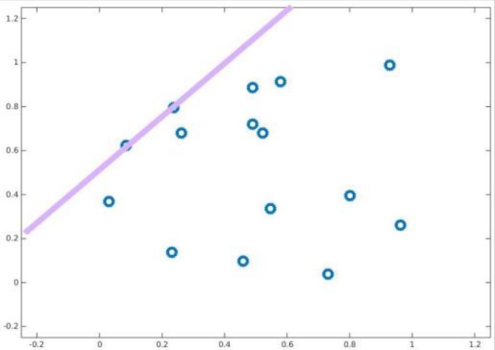
Points



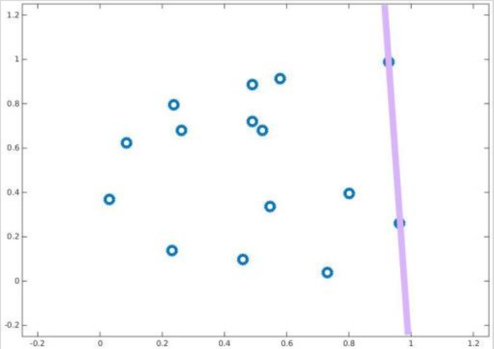
Convex hull

A Brute Force solution for Convex Hull problem

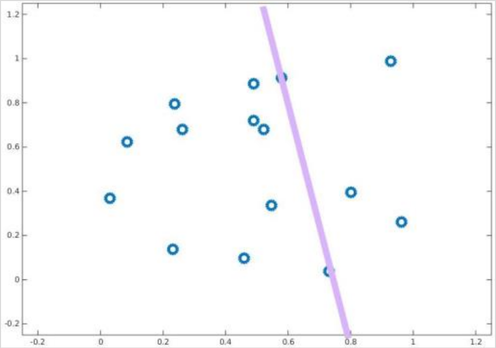
- **Idea:** If we can identify all the line segments/adjacent pairs of points that form the boundary of the convex hull, then we have the convex hull.



Part of hull



Part of hull



Not part of hull

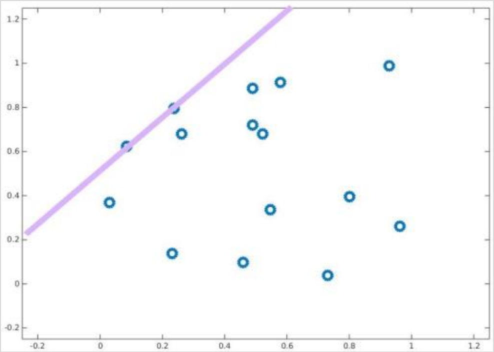
Brute Force Convex Hull - Pseudocode

```
for each point P_i
    for each point P_j where P_j ≠ P_i
        compute the line segment for P_i and P_j
        for every other point P_k where P_k ≠ P_i and P_k ≠ P_j
            if each P_k is on one side of the line segment
                label P_i and P_j in the convex hull
```

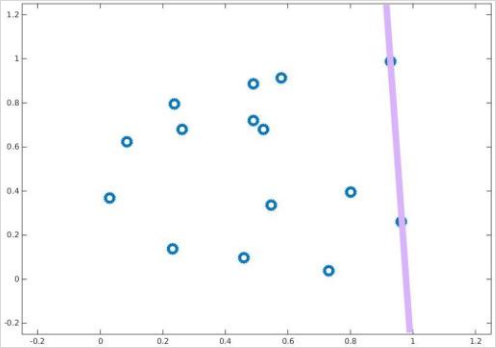
- The straight line through $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ can be defined by: $ax + by = c$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1 y_2 - x_2 y_1$
- For all the points in one of the half-plane, $ax + by > c$; For all the points in the other half-plan, $ax + by < c$; For all the points on the line, $ax + by = c$
- We check whether the expression $ax + by = c$ has the same sign at each of these points

A Brute Force solution for Convex Hull problem

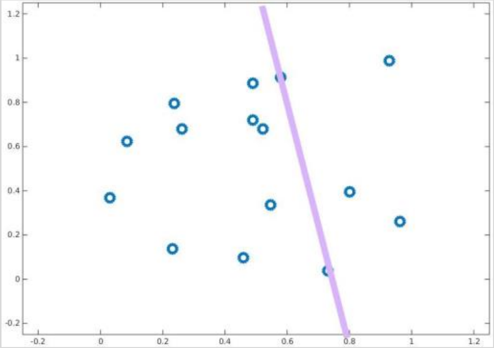
- **Complexity of algorithm:** Since there are O(n^2) pairs of points to examine, and each check requires going through O(n) remaining points, the algorithm is O(n^3).



Part of hull



Part of hull

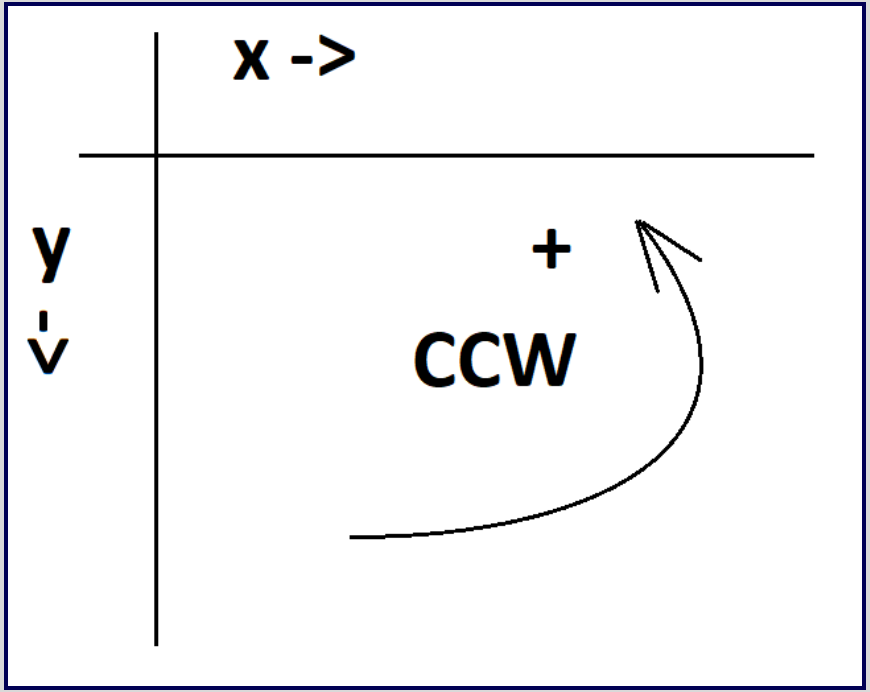


Not part of hull

The java.awt.geom package

- The java.awt.geom package defines classes for geometry objects and operations
- Point2D: represents a point(x, y) in a 2D space. The concrete classes are Point2D.Float and Point2D.Double
- Line2D: represents a line segment in a 2D space. The concrete classes are Line2D.Float and Line2D.Double

The java.awt.geom package



- The Line2D.relativeCCW() method returns 1/-1/0 when rotating a line (p1, p2) to reach a target point in counterclockwise direction or opposite direction
- Use it to determine if 2 points are on the same side of a line segment

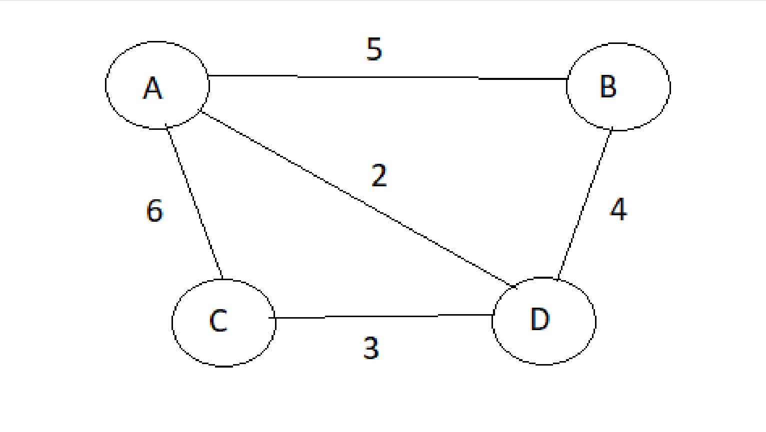
5. Graph Search

What is a Graph?

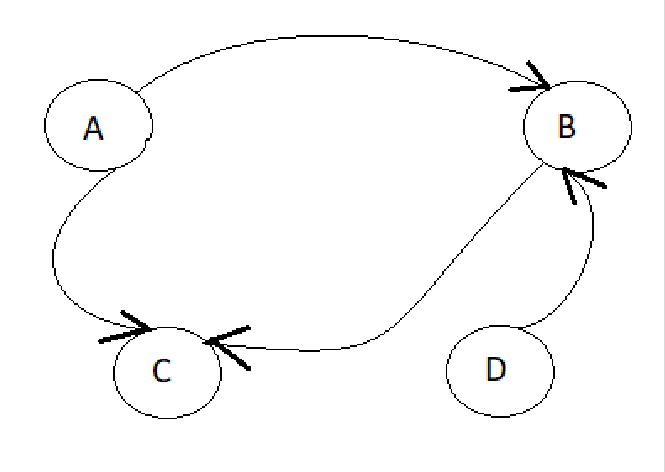
- $G = (V, E)$
- Set of vertices (nodes) V = [v1, v2, v3, etc.]
- Set of edges (arcs, connections) E = [(vs_1, ve_1), (vs_2, ve_2), etc.] in which vs_i and ve_i are members of V
- If (vs_i, ve_i) belongs to E means (ve_i, vs_i) also belongs to E, G is called undirected; otherwise, it is directed
- If each edge is assigned a numeric value, it is called weighted graph; otherwise, it is unweighted
- A graph is acyclic if it does not contain any cycles (for example, a tree); otherwise, it is cyclic

What is a Graph?

- An undirected, weighted, and cyclic graph

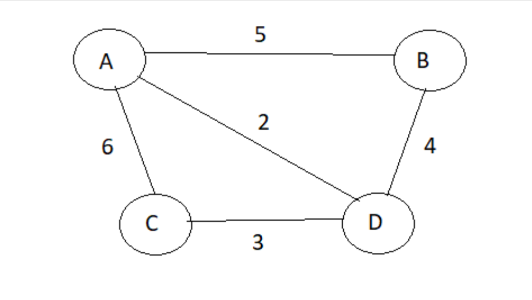


- A directed, unweighted, and acyclic graph



Adjacency Matrix

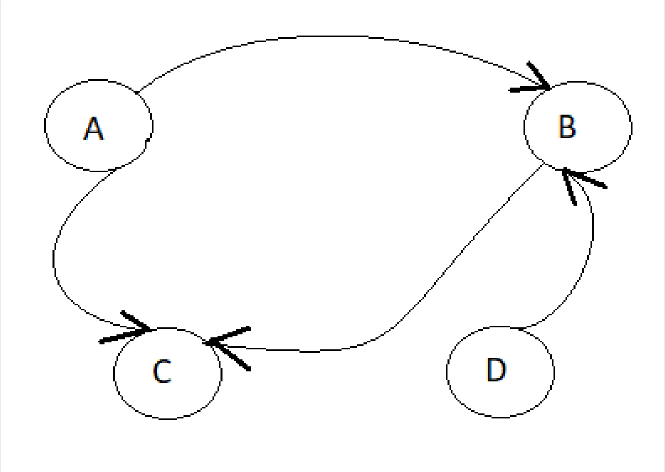
- For a graph of N vertices, use a N x N matrix E (i.e. a 2D array) where E[i, j] = 1 if vertex i and vertex j are connected (i.e., there is an edge connecting them), 0 otherwise
- If a graph is weighted, E[i, j] stores the weight of the edge connecting vertices i and j
- ```
  E = [[0, 5, 6, 2]
       [5, 0, 0, 4]
       [6, 0, 0, 3]
       [2, 4, 3, 0]]
  ```


  ```

Adjacency List

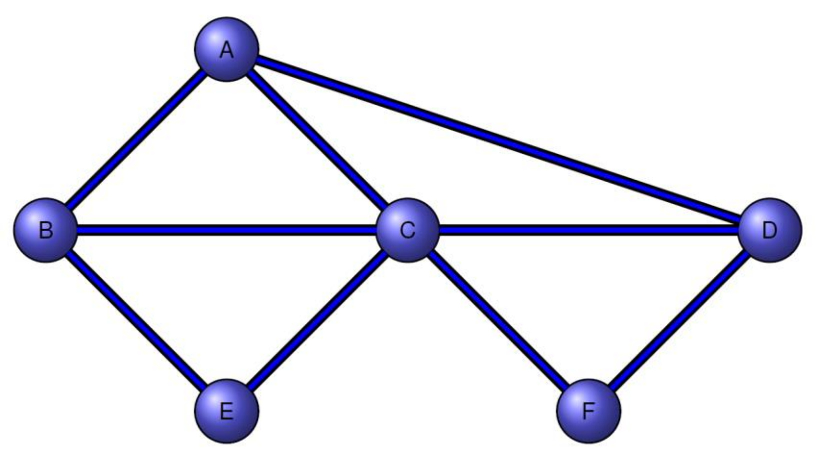
- For a graph of N vertices, use N linked lists, one per node, and each stores the list of that node's neighbours

- `E[0] -> B -> C`
- `E[1] -> C`
- `E[2] = null`
- `E[3] -> B`



Graph Traversal

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Mark a node after visiting it to prevent visiting again (we don't need this step when traversing a tree, why?)



Depth-First Search – Idea

1. Choose an arbitrary vertex and mark it visited
2. From the current vertex, proceed to an unvisited, adjacent vertex and mark it visited
3. Repeat 2^{nd} step until a vertex is reached which has no adjacent, unvisited vertices (dead-end)
4. The algorithm halts when there are no unvisited vertices

Depth-First Search – Pseudocode

ALGORITHM **DFS** (G)

/* Implement a Depth First Traversal of a graph. */

/* INPUT : Graph $G = \langle V, E \rangle$ */

/* OUTPUT : Graph G with its vertices marked with consecutive */

/* integers in initial encounter order. */

$count = 0$ ▷ number of nodes visited

for $i = 0$ **to** v **do** ▷ mark all nodes unvisited

    $Marked[i] = 0$

end for

for $i = 0$ **to** v **do** ▷ visit each unmarked node

    if not $Marked[i]$ **then**

        DFSR (i)

    end if

end for

Depth-First Search – Pseudocode

```
ALGORITHM DFSR (v)
/* Recursively visit all connected vertices. */
/* INPUT : A starting vertex v */
/* OUTPUT : Graph G with its vertices marked with consecutive */
/* integers in initial encounter order. */
count = count + 1                          ▷ increment the node visited counter
Marked[v] = 1                              ▷ mark node as visited
for v′ ∈ V adjacent to v do                ▷ recursively visit all
    if not Marked[v′] then                 ▷ unmarked adjacent nodes
        DFSR (v′)
    end if
end for
```

Depth-First Search Complexity

A DFS search can be implemented with graphs represented as:

- Adjacency matrices with complexity = **O(|V|^2)**
  - It is a graph traversal, so we need to iterate over all vertices (|V| of these).
  - For each vertex, we need to check the neighbours of it. For the matrix representation, the only way we can guarantee to find all neighbours of vertex i is to ***do a linear scan across its row in the matrix***, which has |V| elements.
  - So |V| * |V| gives $O(|V|^2)$ complexity. The traversal also needs to setup visited status, which requires $O|V|$ complexity, but the quadratic term dominates.

Breadth-First Search – Idea

1. Choose an arbitrary vertex *v* and mark it visited.
2. Visit and mark (visited) each of the adjacent(neighbour) vertices of *v* in turn.
3. Once all neighbours of *v* have been visited, select the first neighbour that was visited, and visit all its (unmarked) neighbours.
4. Then select the second visited neighbour of *v*, and visit all its unmarked neighbours.
5. The algorithm halts when we visited all vertices.

DFS vs. BFS

	DFS	BFS
Applications	connectivity, acyclicity	connectivity, acyclicity, shortest paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

6. Exhaustive Search

Another Brute Force solution…

Exhaustive Search

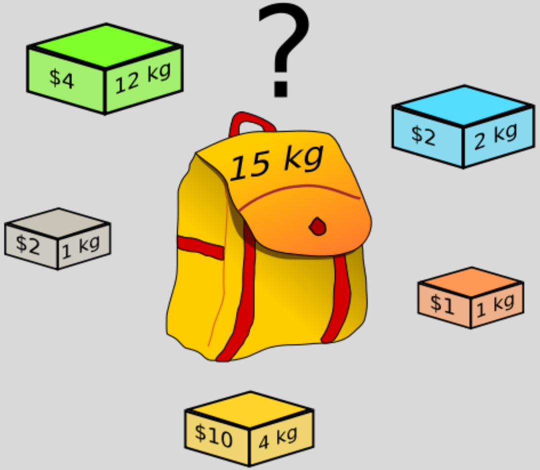
- A brute force solution involving ***enumerating/generating all possible solutions***, then selecting the "best" one.
- Typically applied to ***combinatorial problems***, and insightful to study brute force solutions to them, as some problems can only be solved optimally by exhaustive search.

Exhaustive Search – Idea

- 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 all items have been evaluated, announce the best solution(s)found.

Knapsack Problem

Given n items of known weights $w_1, \ldots, w_n$ and the values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack

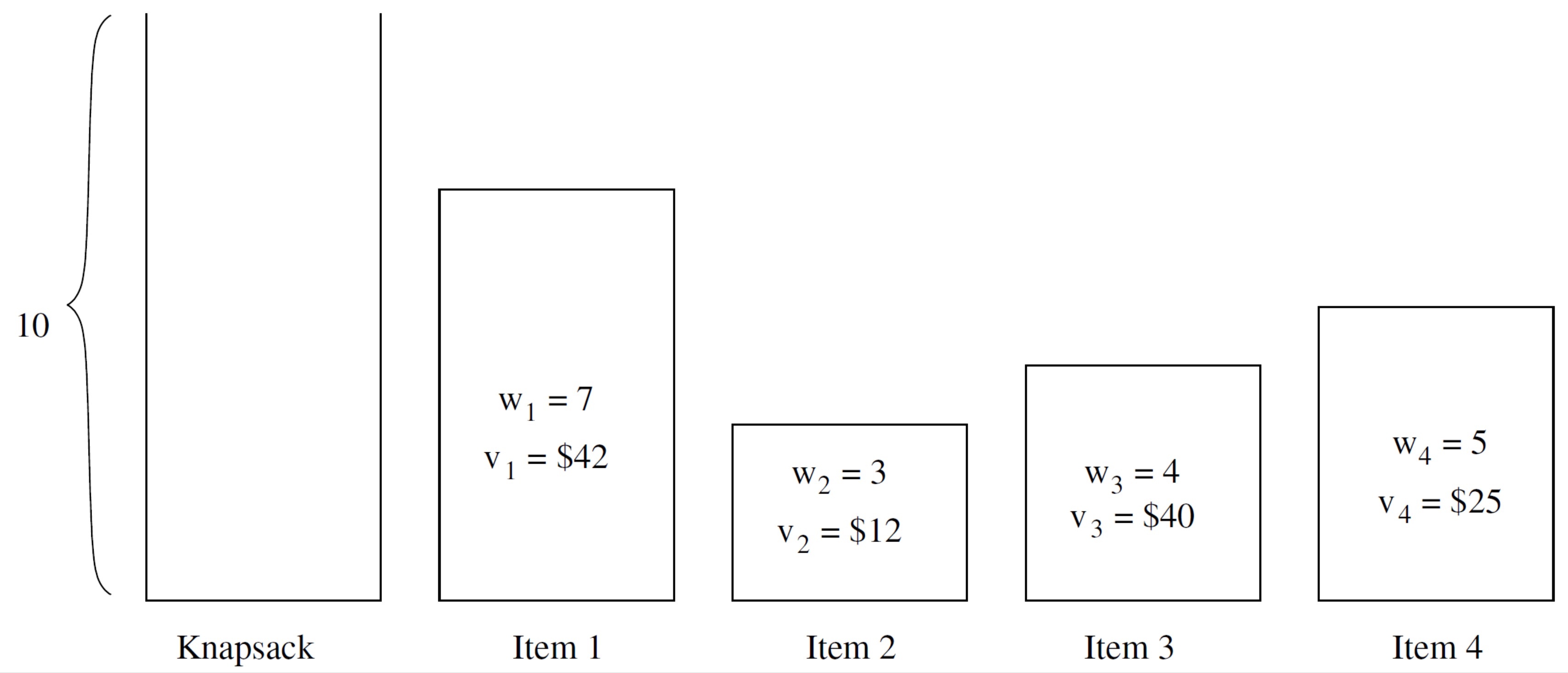


Knapsack Brute Force Algorithm

1. Consider all subsets of the set of *n* items.
2. Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).
3. Find the subset of the largest value among them.

Complexity: Since the number of subsets of an n-element set is 2^n, an exhaustive search produces an $O(2^n)$ algorithm.

Knapsack Problem



Knapsack Problem – Solution

Subset	Total Weight	Total Value
	0	\$0
$\{1\}$	7	\$42
$\{2\}$	3	\$12
$\{3\}$	4	\$40
$\{4\}$	5	\$25
$\{1,2\}$	10	\$36
$\{1,3\}$	11	Not Possible
$\{1,4\}$	12	Not Possible
$\{2,3\}$	7	\$52
$\{2,4\}$	8	\$37
$\{3,4\}$	9	\$65
$\{1,2,3\}$	14	Not Possible
$\{1,2,4\}$	15	Not Possible
$\{1,3,4\}$	16	Not Possible
$\{2,3,4\}$	12	Not Possible
$\{1,2,3,4\}$	19	Not Possible

Pruning

- When generating possible solutions, if the current state is impossible to be a part of the solution, the generating process can stop proceeding further and try new branches instead. This is called pruning
- For example, if $w1 + w2 >$ the knapsack's capacity, we should stop generating subsets that contain both item1 and item2

Generating Subsets

- Input: a list of elements [E1, E2, E3, …]
- Algorithm
  - Going through the list
    - At each position, either Select or Not Select that element
    - Then, recursively apply the same process to the next element
    - If the current element is the last element, output the current subset (if an element is Selected, it is included in the current subset; otherwise, it is not)

Generating Subsets - Pseudocode

```
generate_subset(input_arr, selected_states, cur_idx)
    if cur_idx == size
        process_subset(selected_states)

    // SELECTED
    selected_states[cur_idx] = true
    generate_subset(input_arr, selected_states,
cur_idx + 1)

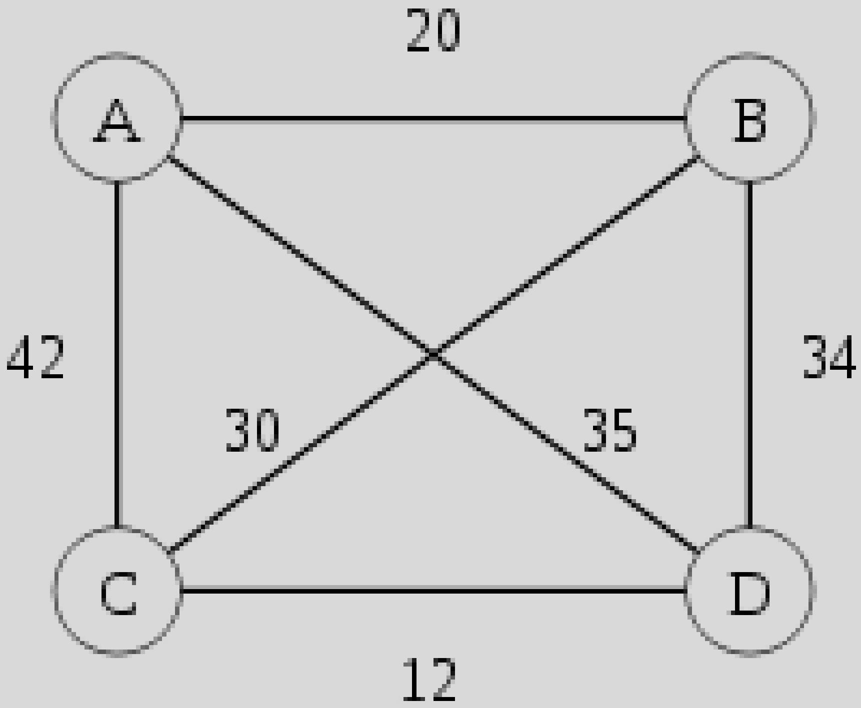
    // NOT SELECTED
    selected_states[cur_idx] = false
    generate_subset(input_arr, selected_states,
cur_idx + 1)
```

Travelling Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- Assume the list of cities is given in an array C[0..N]
- Each route is a permutation of the elements of C
- The number of permutations of a set of N element is N!, hence the complexity of this solution is O(N!)

Travelling Salesman Problem

- What is the solution to the problem instance on the right?



Generating Permutations

- Input: a list of elements [E1, E2, E3, …]
- Algorithm
  - Maintain: Remaining elements and Current permutation
  - Going through all Remaining elements, for each X
    - Move X from Remaining elements to the end of Current permutation
    - Call the method recursively
    - Move X from Current permutation to Remaining elements
  - If Remaining elements is empty, the elements in Current permutation form a valid permutation

Generating Permutations - Pseudocode

```
permute(in_arr, taken_arr, cur_arr, cur_idx)
    if cur_idx == size
        process_permutation(cur_arr)

    for i = 0 to size - 1
        if (taken[i]) continue
        cur_arr[cur_idx] = input_arr[i]
        taken[i] = true
        generate_perm(in_arr, taken_arr,
cur_arr, cur_idx + 1)
        taken[i] = false
```

8-Queens Problem

- How to place eight queens on a chess board so that no two queens can attack each other?
- Use an array col[0..7] to store the row indices of 8 columns
- Each solution must be a permutation of [0, 1, 2, 3, 4, 5, 6, 7]?
- Generate permutation and check for validity at each step

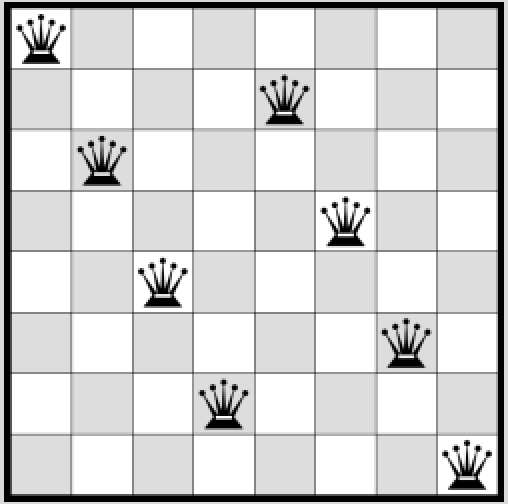


Image source: https://www.aiai.ed.ac.uk/~gwickler/eightqueens.html

To finish…

Learning objectives

1. Understand the ***Brute Force*** approach
2. Understand and apply:
   - Sorting – Selection and Bubble Sorts
   - Sequential Search and String Matching
   - Computational Geometry – Convex hull problems
   - Graph Search: DFS, BFS
   - Exhaustive Search
     - Generating Subsets
     - Generating Permutations

RMIT UNIVERSITY