Algorithms

(Brute Force)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

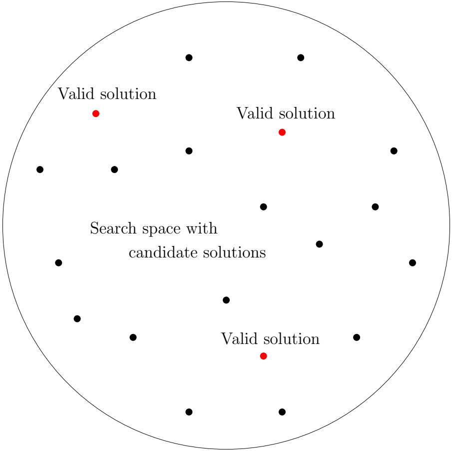
October 19, 2021

Contents

Brute force

- There are two interpretations of brute force search
  1. Extensive search
  2. Straightforward approach to solve problems

Exhaustive search




```
ExhaustiveSearch()
for every solution S in the search space do
  if solution S is valid then
    print solution S
```

String matching

Problem

- Given a text $T[0..n-1]$ and a pattern $P[0..m-1]$, find the location of the first occurrence of the pattern in the text.

String matching

Problem

- Given a text $T[0..n-1]$ and a pattern $P[0..m-1]$, find the location of the first occurrence of the pattern in the text.

Solution

- Check if the pattern matches with the text starting from the 1st index of text.
- If not, check if the pattern matches with the text starting from the 2nd index of the text.
- Repeat this process until either the pattern is found or the end of the text is reached (without finding any pattern).

String matching

StringMatching$(T[0..n-1], P[0..m-1])$

Input: Text $T[0..n-1]$ and pattern $P[0..m-1]$

Output: Return the first position in T where the pattern P occurs

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 $j = m$ **then**
    return i
return -1

String matching

Algorithm	Preprocess time	Matching time	Space
Brute force	none	$\mathcal{O}(mn)$	$\Theta(m+n)$
Trie	$\Theta(m)$	$\Theta(nodes \cdot \lvert\Sigma\rvert)$	$\Theta(nodes \cdot \lvert\Sigma\rvert)$
Suffix tree	$\Theta(n)$	$\mathcal{O}(m)$	$\Theta(n)$
Rabin-Karp	$\Theta(m)$	$\mathcal{O}(mn)$	$\Theta(1)$
Aho-Corasick	$\Theta(m)$	$\mathcal{O}(n)$	$\Theta(m)$
Boyer-Moore	$\Theta(m+\lvert\Sigma\rvert)$	$\mathcal{O}(mn)$	$\Theta(\lvert\Sigma\rvert)$
KMP	$\Theta(m)$	$\mathcal{O}(n)$	$\Theta(m)$

Closest pair

Problem

- Given n points in 2-D Euclidean space, find the closest pair of points.

Closest pair

Problem

- Given n points in 2-D Euclidean space, find the closest pair of points.

Solution

- For every two distinct points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, the distance between them can be computed as $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
- Find the points that leads to smallest such distance

Closest pair

ClosestPair$(x[1..n], y[1..n])$

Input: Arrays $x[1..n]$ and $y[1..n]$ for x- and y-coordinates
Output: Closest pair of points a and b

$minimum \leftarrow \infty$
for $i \leftarrow 1$ **to** $n-1$ **do**
  for $j \leftarrow i+1$ **to** n **do**
    $distance \leftarrow \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
    if $distance < minimum$ **then**
      $minimum \leftarrow distance$
      $a \leftarrow i$; $b \leftarrow j$
return $\{(x_a, y_a), (x_b, y_b)\}$

Closest pair

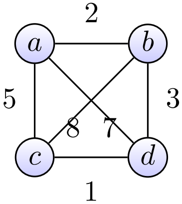
Algorithm	Time	Space
Brute force	$\Theta\left(n^2\right)$	$\Theta\left(1\right)$
D&C	$\Theta\left(n\log^2 n\right)$	$\Theta\left(n\log n\right)$
D&C improved	$\Theta\left(n\log n\right)$	$\Theta\left(n\log n\right)$

Traveling salesperson problem (TSP)

Problem

- Find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
- Given a weighted connected graph, find the shortest "Hamiltonian circuit".

Traveling salesperson problem (TSP)



No.	Tour	Length	Shortest?
1	$a \to b \to c \to d \to a$	$2+8+1+7=18$	
2	$a \to b \to d \to c \to a$	$2+3+1+5=11$	✓
3	$a \to c \to b \to d \to a$	$5+8+3+7=23$	
4	$a \to c \to d \to b \to a$	$5+1+3+2=11$	✓
5	$a \to d \to b \to c \to a$	$7+3+8+5=23$	
6	$a \to d \to c \to b \to a$	$7+1+8+2=18$	

Traveling salesperson problem (TSP)

Algorithm	Computes	Time	Space
Exact algorithms			
Brute force	opt	$\Theta\left((n-1)!\right)$	$\Theta\left(n^2\right)$
Bellman-Held-Karp DP	opt	$\Theta\left(2^n n^2\right)$	$\Theta\left(2^n n\right)$
Approximation algorithms for graphs satisfying triangle inequality			
Rosenkrantz-Stearns-Lewis	≤ 2 opt	$\Theta\left(n^2 \log n\right)$	?
Christofides	≤ 1.5 opt	$\Theta\left(n^3\right)$	?

Knapsack problem

Problem

- Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Knapsack problem

Subset	Total weight	Total value	Opt?
$\{\}$	0	$\$0$	
$\{1\}$	7	$\$42$	
$\{2\}$	3	$\$12$	
$\{3\}$	4	$\$40$	
$\{4\}$	5	$\$25$	
$\{1,2\}$	$7+3=10$	$42+12=\$54$	
$\{1,3\}$	$7+4=11$	$42+40=\$82$	
$\{1,4\}$	$7+5=12$	$42+25=\$67$	
$\{2,3\}$	$3+4=7$	$12+40=\$52$	
$\{2,4\}$	$3+5=8$	$12+25=\$37$	
$\{3,4\}$	$4+5=9$	$40+25=\$65$	✓
$\{1,2,3\}$	$7+3+4=14$	$42+12+40=\$94$	
$\{1,2,4\}$	$7+3+5=15$	$42+12+25=\$79$	
$\{1,3,4\}$	$7+4+5=16$	$42+40+25=\$109$	
$\{2,3,4\}$	$3+4+5=12$	$12+40+25=\$77$	
$\{1,2,3,4\}$	$7+3+4+5=19$	$42+12+40+25=\$119$	

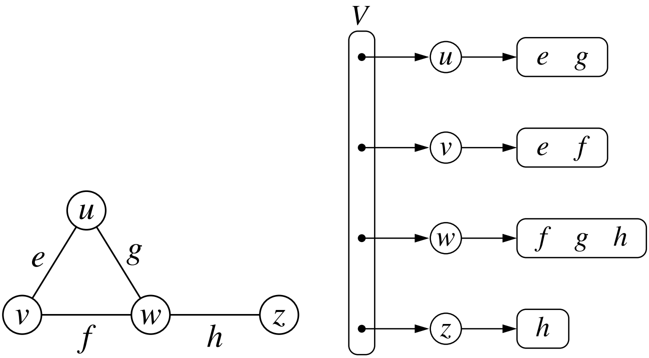
Graph traversals

- Depth first search (DFS)
- Breadth first search (BFS)

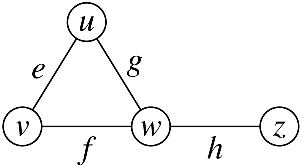
Graph representations

- Adjacency list
- Adjacency matrix

Adjacency list



Adjacency matrix



		0	1	2	3
u ⟶	0		e	g	
v ⟶	1	e		f	
w ⟶	2	g	f		h
z ⟶	3			h	

DFS and BFS

Feature	DFS	BFS
Similarities		
Works on	Trees and graphs	Trees and graphs
Time	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$
Space	$\mathcal{O}(\lvert V\rvert)$	$\mathcal{O}(\lvert V\rvert)$
Differences		
Core idea	Starts at arbitrary node and explores as far as possible along each branch before backtracking	Starts at arbitrary node and explores all nodes at the present depth prior to moving on to the nodes at the next depth level
DS	Uses stack	Uses queue

DFS

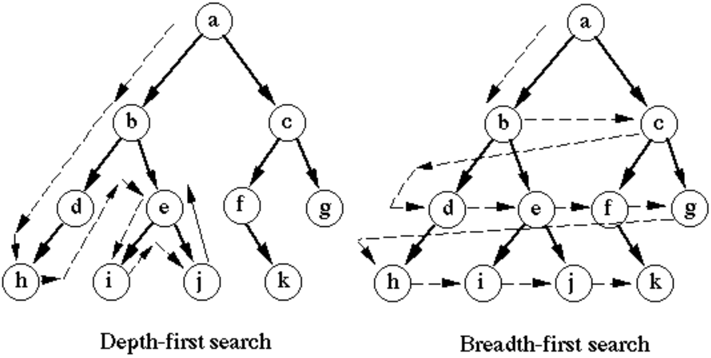


Image source: `https://vivadifferences.com/wp-content/uploads/2019/10/DFS-VS-BFS.png`

DFS

Applications:

- Finding connected components.
- Topological sorting.
- Finding the bridges of a graph.
- Finding strongly connected components.
- Determining whether a species is closer to one species or another in a phylogenetic tree.
- Planarity testing.
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.

BFS

Applications:

- Finding the shortest path between two nodes u and v, with path length measured by number of edges
- (Reverse) Cuthill–McKee mesh numbering.
- Edmonds-Karp method for computing maximum flow.
- Serialization/Deserialization of a binary tree vs serialization in sorted order.
- Construction of the failure function of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph.
- Implementing parallel algorithms for computing a graph's transitive closure.

DFS

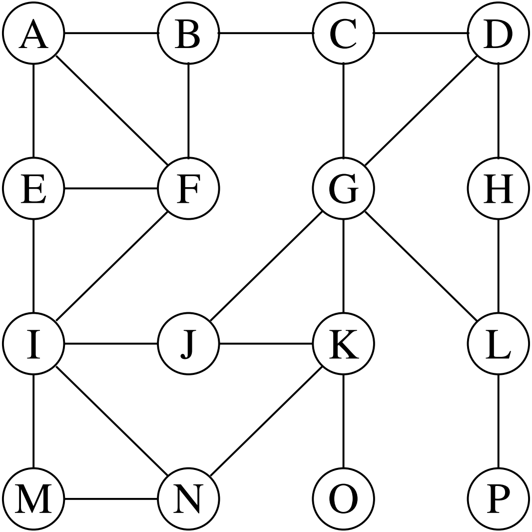
DepthFirstSearch(G)

Mark each vertex in V with 0 as a mark of being unvisited
$count \leftarrow 0$
for each vertex v in V **do**
  if v is marked with 0 **then**
    $\mathrm{DFS}(v)$

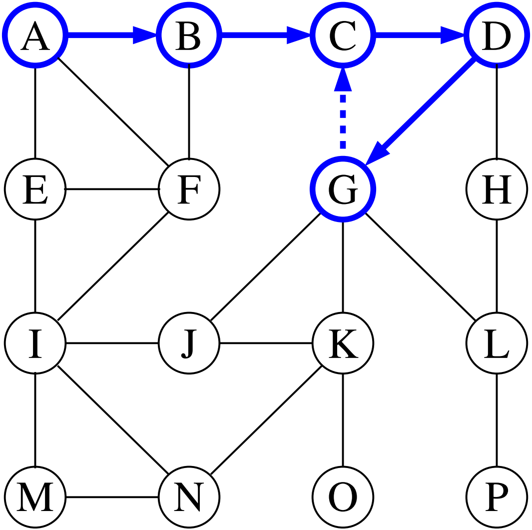
DFS(v)

$count \leftarrow count + 1$
Mark v with $count$
for each vertex w in V adjacent to v **do**
  if w is marked with 0 **then**
    $\mathrm{DFS}(w)$

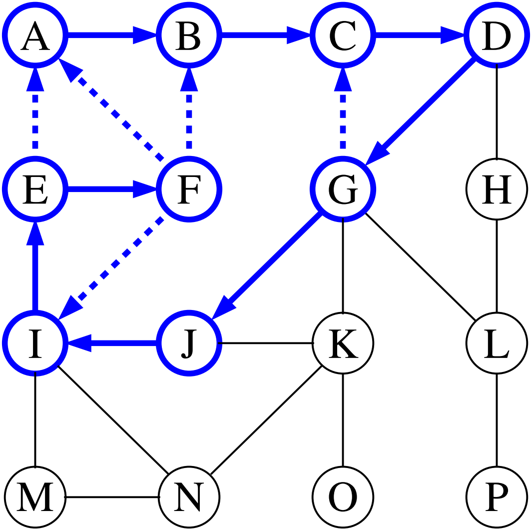
DFS



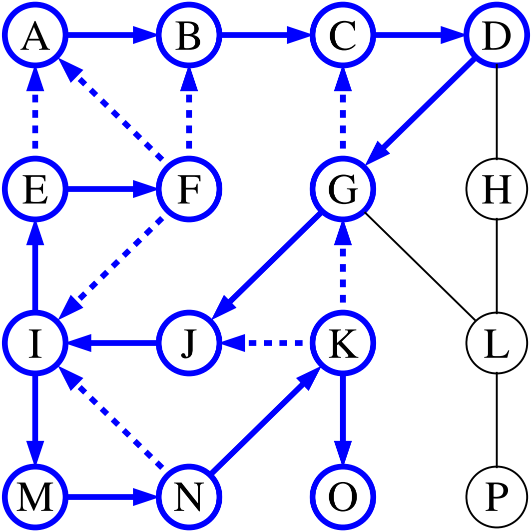
DFS



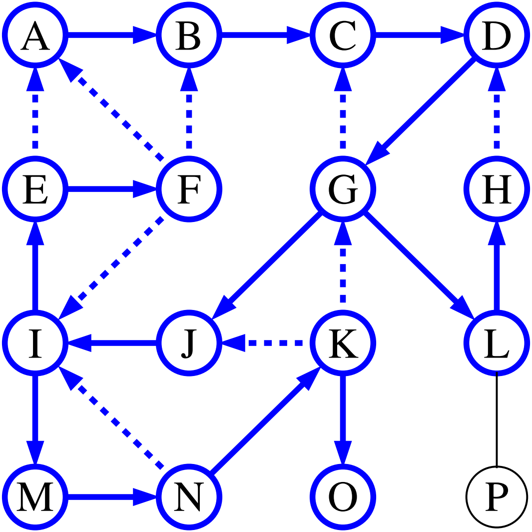
DFS



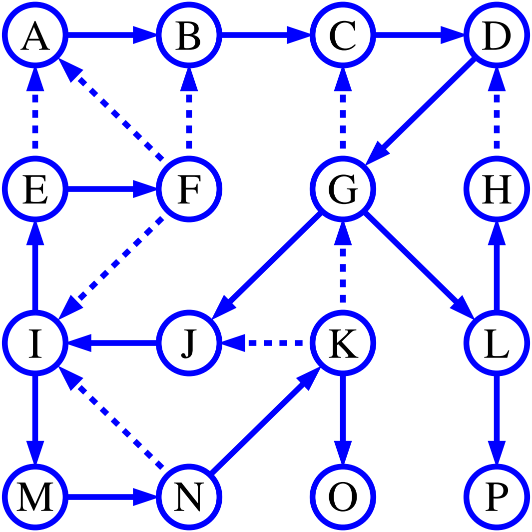
DFS



DFS



DFS



BFS

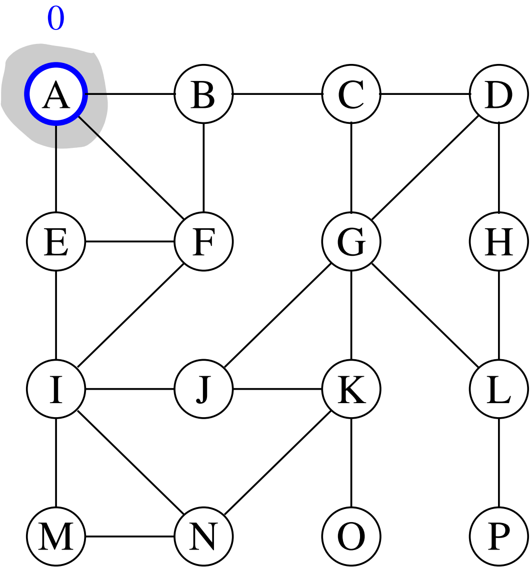
BreadthFirstSearch(G)

```
Mark each vertex in V with 0 as a mark of being unvisited
count ← 0
for each vertex v in V do
  if v is marked with 0 then
    BFS(v)
```

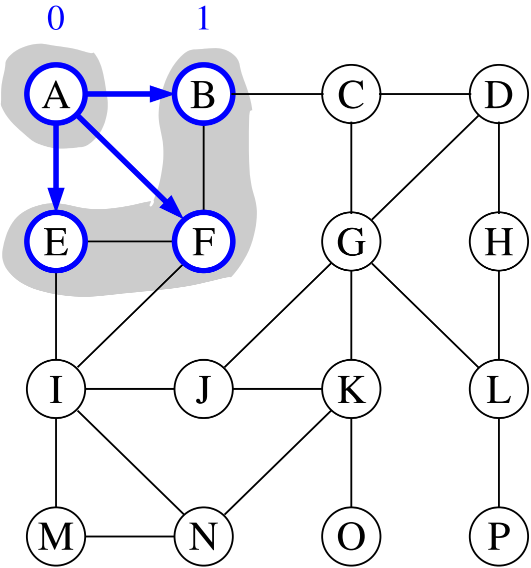
BFS(v)

```
count ← count + 1
Mark v with count
Initialize a queue with v
while queue is not empty do
  for each vertex w in V adjacent to the front vertex do
    if w is marked with 0 then
      count ← count + 1
      Mark w with count
      Add w to the queue
  Remove the front vertex from the queue
```

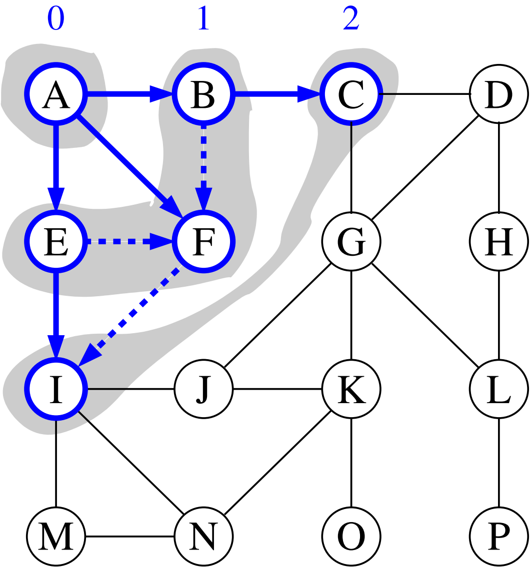
BFS



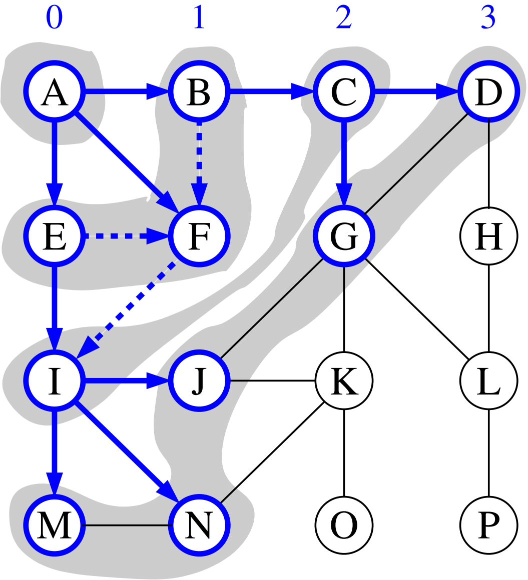
BFS



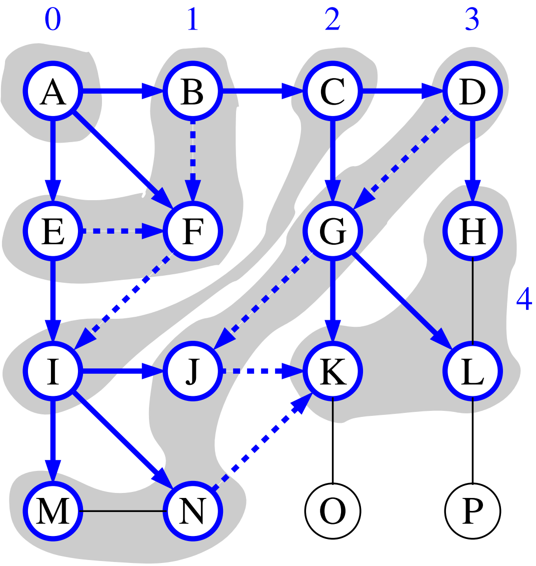
BFS



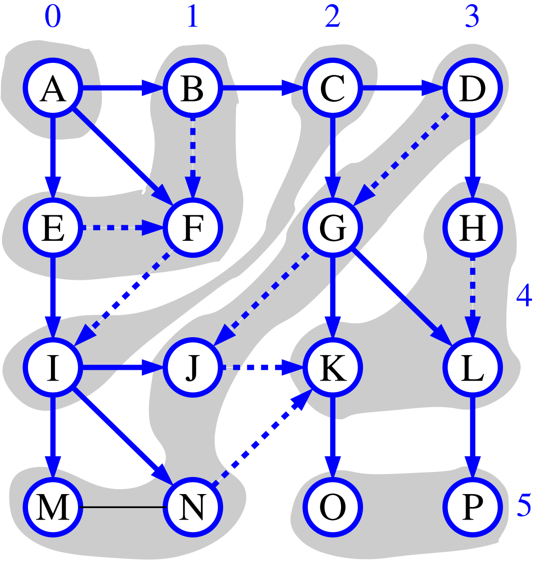
BFS



BFS



BFS



Exhaustive search

Negatives

- Combinatorial explosion or curse of dimensionality

Positives

- Might be the only technique that works for some problems (e.g. linear search)
- Might be used for benchmarking solutions
- Exhaustive search $+$ pruning $=$ Backtracking
  Backtracking is a very powerful algorithm design technique
- Might be fast for small instances of problems
  (e.g. insertion sort is used to sort subarrays of size ≤ 30)
- Used to find the shortest proofs or axioms in mathematics
- Used in computer-generated/aided proofs
- Benchmarking cryptographic algorithms using brute force attack
- Used in games where computer is a player

Random permutation generation

Problem

- Generate random permutations of $A[1..n]$.

Random permutation generation

- Does not generate uniformly random permutations

RandomPermutationGenerator($A[1..n]$)

Input: $A[1..n]$
Output: Random permutation of $A[1..n]$

1. **for** $i \leftarrow 1$ **to** $n - 1$ **do**
2. Swap($A[i], A[$Random($[1..n]$)$]$)
3. **return** $A[1..n]$

- Generates uniformly random permutations

RandomPermutationGenerator($A[1..n]$)

Input: $A[1..n]$
Output: Random permutation of $A[1..n]$

1. **for** $i \leftarrow 1$ **to** $n - 1$ **do**
2. Swap($A[i], A[$Random($[i..n]$)$]$)
3. **return** $A[1..n]$

Bubble sort

Problem

- Sort a given n-sized array in nondecreasing order.

$\textsc{BubbleSort}(A[0..n-1])$

Input: Arrays $A[0..n-1]$
Output: Sorted array $A[0..n-1]$
1. **for** $i \leftarrow 0$ **to** $n-2$ **do**
2.   **for** $j \leftarrow 0$ **to** $n-2-i$ **do**
3.   **if** $A[j+1] < A[i]$ **then**
4. $\textsc{Swap}(A[j], A[j+1])$

Selection sort

Problem

- Sort a given n-sized array in nondecreasing order.

SelectionSort$(A[0..n-1])$

Input: Arrays $A[0..n-1]$
Output: Sorted array $A[0..n-1]$

for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ **then**
      $min \leftarrow j$
  Swap$(A[i], A[min])$

Counting sort

Problem

- Sort a given n-sized array in nondecreasing order.
- Items are non-negative integers with maximum value k.

Counting sort

Problem

- Sort a given n-sized array in nondecreasing order.
- Items are non-negative integers with maximum value k.

Solution

- Create an array for indices in the range $[0, k]$
- Distribute items to these indices to compute item frequences
- Compute the cumulative frequencies of items for indices in the range $[0, k]$
- Find the sorted array

Counting sort

A	2	5	3	0	2	3	0	3	Unsorted array $A[1..n]$
C	2	0	2	3	0	1			Frequencies array $C[0..k]$
C	2	2	4	7	7	8			Cumulative frequencies array $C[0..k]$
B	0	0	2	2	3	3	3	5	Sorted array $B[1..n]$

Counting sort

```
CountingSort(A[1..n])
Input: An array A[1..n] of non-negative integers
Output: Array A[1..n] sorted in nondecreasing order
k ← maximum value in A[1..n]
B[1..n] ← new array; C[0..k] ← new array initialized to 0

[Find the frequencies of items] ......................................
[After this step, C[i] will contain #elements equal to i]
for j ← 1 to n do
  C[A[j]] ← C[A[j]] + 1

[Find the cumulative frequencies of items] ............................
[After this step, C[i] will contain #elements less than or equal to i]
for i ← 1 to k do
  C[i] ← C[i] + C[i − 1]

[Get the sorted array in B] ..........................................
for j ← n to 1 do
  B[C[A[j]] ← A[j]
  C[A[j]] ← C[A[j]] − 1

[Copy the sorted array to A] .........................................
for j ← 1 to n do
  A[j] ← B[j]
```

Counting sort

A	2	5	3	0	2	3	0	3	$A[8] = 3$
C	2	2	4	7	7	8			$C[3] = 7$
B							3		$B[7] = 3$
C	2	2	4	6	7	8			$C[3] - -$

A	2	5	3	0	2	3	0	3	$A[7] = 0$
C	2	2	4	6	7	8			$C[0] = 2$
B		0					3		$B[2] = 0$
C	1	2	4	6	7	8			$C[0] - -$

A	2	5	3	0	2	3	0	3	$A[6] = 3$
C	1	2	4	6	7	8			$C[3] = 6$
B		0				3	3		$B[6] = 3$
C	1	2	4	5	7	8			$C[3] - -$

A	2	5	3	0	2	3	0	3	$A[5] = 2$
C	1	2	4	5	7	8			$C[2] = 4$
B		0		2		3	3		$B[4] = 2$
C	1	2	3	5	7	8			$C[2] - -$

A	2	5	3	0	2	3	0	3	$A[4] = 0$
C	1	2	3	5	7	8			$C[0] = 1$
B	0	0		2		3	3		$B[1] = 0$
C	0	2	3	5	7	8			$C[0] - -$

A	2	5	3	0	2	3	0	3	$A[3] = 3$
C	0	2	3	5	7	8			$C[3] = 5$
B	0	0		2	3	3	3		$B[5] = 3$
C	0	2	3	4	7	8			$C[3] - -$

A	2	5	3	0	2	3	0	3	$A[2] = 5$
C	0	2	3	4	7	8			$C[5] = 8$
B	0	0		2	3	3	3	5	$B[8] = 5$
C	0	2	3	4	7	7			$C[5] - -$

A	2	5	3	0	2	3	0	3	$A[1] = 2$
C	0	2	3	4	7	7			$C[2] = 3$
B	0	0	2	2	3	3	3	5	$B[3] = 2$
C	0	2	2	4	7	7			$C[2] - -$