DSA Mini Textbook

Theo Park

Contents

Preface **3**

1 Runtime Analysis **4**
1.1 Power Law . . . 4
1.2 Runtime Expressions . . . 4
1.3 Asymptotic Runtime Analysis . . . 4
1.4 Recursive Relationship . . . 4

2 Intro to Data Structures **5**
2.1 Array . . . 5
2.2 Linked List . . . 5
2.3 Stack . . . 5
2.4 Queue . . . 6
2.5 Tree . . . 6
2.6 Binary Heap . . . 7
2.6.1 Building a Heap – Top-down v.s. Bottom-up . . . 7

3 Sorting Algorithms **8**
3.1 Bubble, Selection, Insertion Sort . . . 8
3.2 Shell Sort . . . 9
3.3 Heap Sort . . . 10
3.3.1 Sort Down Algorithm . . . 10
3.4 Merge Sort . . . 10
3.5 Quick Sort . . . 12
3.5.1 Partition and Pivot . . . 12
3.6 Comparison Sorting Algo Lower Bound . . . 12
3.7 Bucket Sort . . . 13
3.8 Counting Sort . . . 13
3.9 Radix Sort . . . 14
3.9.1 Lexicographic Order . . . 14
3.10 Chapter 3 Review . . . 14

4 Hash Tables **15**
4.1 Division Method . . . 15
4.2 Multiplication Method . . . 15
4.3 Collision . . . 15
4.3.1 Chaining . . . 15
4.3.2 Open Addressing . . . 15

5 Search Tree **16**
5.1 Binary Search Tree and Its Limit . . . 16
5.2 2-3 Tree . . . 16
5.3 Red-Black Tree . . . 16
5.4 Left-Leaning Red-Black Tree . . . 16
5.4.1 Deletion in LLRBT . . . 16

6 Undirected Graph **17**
6.1 Adjacency Matrix and List . . . 17
6.2 DFS . . . 17
6.3 BFS . . . 17

7 Directed Graphs **19**
7.1 Strong Connectivity . . . 19
7.1.1 Brute-force Strong Connectivity Algorithm . . . 19
7.1.2 Brute-force using Stack . . . 19
7.1.3 Strongly Connected Components and Kosaraju's Algorithm . . . 19
7.2 Directed Acyclic Graphs . . . 19
7.2.1 Topological Sort . . . 19

8 Weighted Graphs **20**
8.1 Shortest Path . . . 20
8.1.1 Dijkstra's Algorithm . . . 20
8.1.2 Bellman-Ford Algorithm . . . 20
8.2 Articulation Points . . . 21
8.3 Minimum Spanning Tree . . . 21
8.3.1 Cycle and Cut Properties . . . 21
8.3.2 Prim's Algorithm . . . 21
8.4 Union-Find . . . 21
8.4.1 Kruskal MST Algorithm . . . 21

9 Strings **22**
9.1 Brute-force String Pattern Matching . . . 22
9.2 KMP Algorithm . . . 22
9.3 Trie . . . 22
9.4 PATRICIA . . . 22
9.5 Huffman Coding . . . 22

Preface

Chapter 1

Runtime Analysis

Algorithms are any well-defined computational procedures that take some value(s) as input and produce more value(s) as output. They are **effective**, **precise**, and **finite**. There are several ways to analyze the runtime of an algorithm.

1.1 Power Law

1. For the algorithm, get a table for the input size n and the runtime $T(n)$.

n	$T(n)$
250	0.0
500	0.012
1000	0.0954
2000	0.7727
4000	6.1664

2. Make sure that the data plots are valid for the power law analysis by checking the following properties:

   - **have enough data plots.** For instance, if there are only two data plots, you should not make the power law conjecture.
   - **fits the power law.** You can verify this by finding the ratio between data plots.

n	$T(n)$	ratio
250	0.0	–
500	0.012	–
1000	0.0954	0.0954 / 0.012 = 7.95
2000	0.7727	0.7727 / 0.0954 = 8.10
4000	6.1664	6.1664 / 0.7727 = 7.98

   For the ratios we found, //TODO

1.2 Runtime Expressions

1.3 Asymptotic Runtime Analysis

1.4 Recursive Relationship

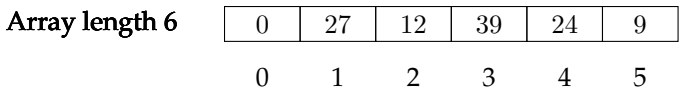
Chapter 2

Intro to Data Structures

Data structures are collections of data values, the relationships among them, and the functions or operations that can be applied to the data. All three characteristics need to be present.

2.1 Array

Array is a linear container of items.



- Access time: $\Theta(1)$
- Inserting n items in the *tail* for array size n: $\Theta(1)$ per item, $n \times \Theta(1) \in \Theta(1)$
- Inserting n items in the *tail* for array size *unknown*: $\Theta(n)$ per item, $n \times \Theta(n) \in \Theta(n)$
- Resizing: $\Theta(n)$

Lesson? **Keep track of the size and the tail!**

2.2 Linked List

- Access time: $\Theta(n)$
- Insertion: $\Theta(1)$
- Deletion:
  - Singly LL: $O(n)$
  - Singly Circular list: $O(n)$
  - Doubly LL: $\Theta(1)$
  - Doubly Circular list: $\Theta(1)$

2.3 Stack

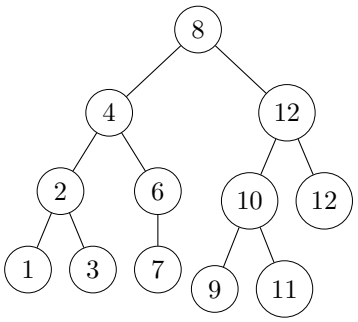
Stack takes the ”last in, first out” approach.

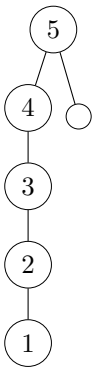
2.4 Queue

Queue takes the "first in, first out" approach.

2.5 Tree

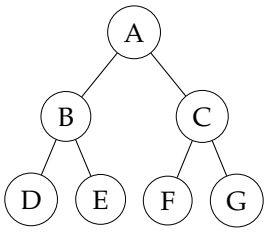
- Search: $O(\log(n))$ for a balanced tree (right), $O(n)$ for unbalanced tree (left)





- **Maximum number of leaves:** 2^h, where h is the height of the tree
- **Maximum number of nodes:** $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$, where h is the height of the tree

Following calculations and traversals will be explained using the following **full binary tree**.



Definitions:

- I (number of internal nodes) $= 3$ (A, B, C – root is an internal node unless it is a leaf)
- N (number of nodes) $= 7$
- L (number of leaves) $= 4$

Full binary tree properties:

- $L = I + 1 = \frac{N+1}{2}$
- $N = 2I + 1 = 2L - 1$
- $I = \frac{N-1}{2} = L - 1$

Traversal (applicable for all trees, including non-full-binary trees):

- **Preorder – [N]ode [L]eft [R]ight:** A B D E C F G
- **Inorder – [L][N][R]:** D B E A F C
- **Postorder – [L][R][N]:** D E B F C A
- Level – by height: A B C D E F

2.6 Binary Heap

- Max heap: Key in each node is larger than or equal to the keys in node's two children
- Min heap: Key in each node is less than or equal to the keys in node's two children

Array implementation for the above max-heap is as follow:

- LEFT-CHILD(i) = $2i + 1$
- RIGHT-CHILD(i) = $2i + 2$
- PARENT(i) = $\lfloor \frac{i-1}{2} \rfloor$, $i > 0$

2.6.1 Building a Heap – Top-down v.s. Bottom-up

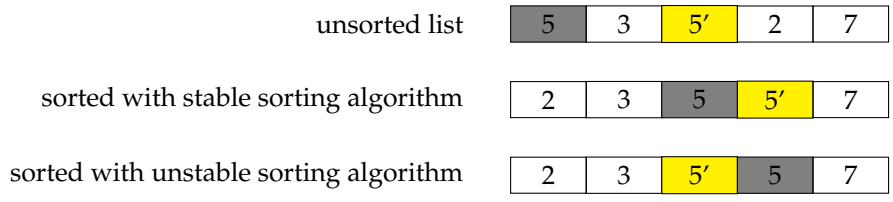
Chapter 3

Sorting Algorithms

Once you store all the items in a data structure, you might want to organize them for the future use (such as selecting nth largest element). For this, you have to *sort* the data structure (in this book, array will be assumed). *Sorting* is deciding how to permute the array elements until they are sorted.

There are couple aspects of sorting algorithms you need to consider:

- Runtime: When analyzing a runtime of a sorting algorithm, both number of compares and number of swaps are considered. **Most sorting algorithms make more comparisons than swaps**, but if a sorting algorithm makes more swaps, it must be used for the asymptotic runtime analysis
- Stability: An algorithm is stable if it preserves the input ordering of equal items.



- In-place: An algorithm is in-place if it can directly sorts the items without making a copy or extra array(s)

3.1 Bubble, Selection, Insertion Sort

The first three sorting algorithms we will discuss are BUBBLE-SORT, SELECTION-SORT, and INSERTION-SORT. They are simple to understand and implement, however, they all have the worst-case runtime of $O(n^2)$, which limits their usages.

BUBBLE-SORT goes through the array and swap elements that are out of order, and if such element is found, it re-starts from the beginning of the list.

```
1: function BUBBLE-SORT(A)                          ▷ A is an array size n
2:     repeat ← True
3:     while repeat is True do
4:         repeat ← False
5:         for i = 0 to n − 2 do
6:             if A[i] > A[i + 1] then
7:                 SWAP(A, i, i + 1)                ▷ Assume SWAP(A, i, j) swaps A[i] and A[j]
8:                 repeat ← True
9:     return A
```

In-place?	Stable?
True	True

-	NumCompares	NumSwaps
Already Sorted	$n-1$	0
Worst Case	n^2-n	$\frac{1}{2}n^2-\frac{1}{2}n$

SELECTION-SORT is a sorting algorithm closest to our "natural" thought of sorting an array. It finds the smallest, second smallest, ... elements and place them accordingly. It makes the same number of comparisons in any cases.

```
1: function SELECTION-SORT(A)                          ▷ A is an array size n
2:     for i = 0 to n − 2 do
3:         index ← i
4:         for i = i + 1 to n − 1 do
5:             if [j] < A[index] then
6:                 index ← j
7:         if i ≠ index then
8:             SWAP(A, i, index)                       ▷ Assume SWAP(A, i, j) swaps A[i] and A[j]
9:     return A
```

In-place?	Stable?
True	False

-	NumCompares	NumSwaps
Already Sorted	$\frac{1}{2}n^2-\frac{1}{2}n$	0
Worst Case	$\frac{1}{2}n^2-\frac{1}{2}n$	$\lfloor\frac{1}{2}n\rfloor$

INSERTION-SORT, takes one element at a time and places it in the correct index of the partially sorted sub-array until the array is sorted.

```
1: function INSERTION-SORT(A)                          ▷ A is an array size n
2:     for i = 1 to n − 1 do
3:         j ← i − 1
4:         while j ≥ 0 and A[j] > A[j + 1] do
5:             SWAP(A, j, j + 1)                       ▷ Assume SWAP(A, i, j) swaps A[i] and A[j]
6:             j ← j − 1
7:     return A
```

In-place?	Stable?
True	True

-	NumCompares	NumSwaps
Already Sorted	$n-1$	0
Worst Case	$\frac{1}{2}n^2-\frac{1}{2}n$	$\frac{1}{2}n^2-\frac{1}{2}n$

3.2 Shell Sort

SHELL-SORT runs insertion sort with "gaps", the index margin where insertion sort will be performed.

```
1: function SHELL-SORT(A, H)          ▷ A is an array size n, H is the array size m containing gap values
2:     for h = 0 to m − 1 do
3:         gap ← H[h]
4:         for i = 1 to n − 1 do
5:             j ← i − 1
6:             while j ≥ 0 and j + gap < n do
7:                 if A[j] > A[j + gap] then
8:                     SWAP(A, j, j + gap)
9:                 j ← j − gap
10:    return A
```

3.3 Heap Sort

HEAP-SORT uses binary max-heap to sort an array. While it's the first sorting algorithm to utilize a data structure, it's not preferred in real life due to cache issue.

```
1: function HEAP-SORT(A)              ▷ A is an array size n
2:     A ← HEAPIFY(A)
3:     for i = n − 1 down to 0 do
4:         SORT-DOWN(A, i)
5:     return A
```

The algorithm first builds the heap from the array elements (refer to section 2.5 for methods for building a heap). BOTTOM-UP is used for its runtime. Then the algorithm calls SORT-DOWN from the last heap elements down to the first.

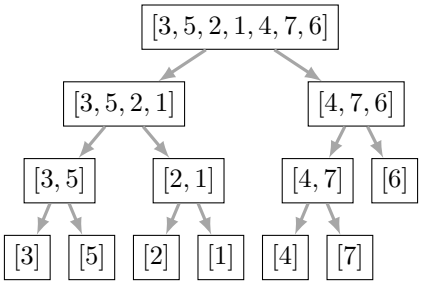
3.3.1 Sort Down Algorithm

3.4 Merge Sort

MERGE-SORT is a *divide-and-conquer* sorting algorithm that divides the array down to multiple lists with one element and merge them together.

```
1: function MERGE-SORT(A, l, r)       ▷ A is an array size l + r. Initially, l = 0, r = size of A
2:     if l < r then
3:         m ← (l + r)/2
4:         MERGE-SORT(A, l, m)
5:         MERGE-SORT(A, m + 1, r)
6:         MERGE(A, l, m, r)
7:     return A
```

The algorithm recursively calls itself with half of the current array, and once all the sub-arrays are of size 1, it begins merging them together. The diagram below shows how MERGE-SORT breaks $A = [3, 5, 2, 1, 4, 7, 6]$ down.



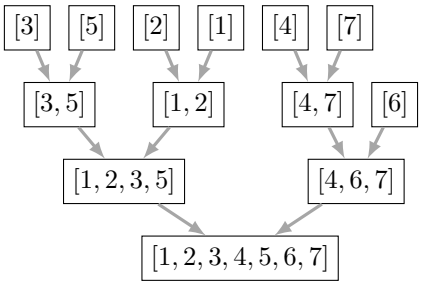

```
1:  function MERGE(A, l, m, r)                                   ▷ A is an array
2:      n1 ← m − l + 1
3:      n2 ← r − m
4:      L ← array size of (n1 + 1)
5:      R ← array size of (n2 + 1)

6:      ▷ Assign elements to each array                                    ◁
7:      for i = 0 to n1 − 1 do
8:          L[i] ← A[l + i]
9:      for i = 0 to n2 − 1 do
10:         R[i] ← A[m + j + 1]

11:     L[n1], R[n2] ← ∞
12:     i, j ← 0

13:     for k = l to r do
14:         if L[i] ≤ R[j] then
15:             A[k] ← L[i]
16:             i ← i + 1
17:         else
18:             A[k] ← R[i]
19:             j ← j + 1
20:     return A
```

MERGE algorithm merges two sorted arrays by "climbing the ladder". The diagram on the right shows how MERGE algorithm merges the array we split earlier in the recursive step of the MERGE-SORT, and the diagram on the left shows the visualization of how MERGE algorithm merges two sorted array by "climbing the ladder."



1	2	3	4	5	6	7
$\underline{1} : 4$	$\underline{2} : 4$	$\underline{3} : 4$	$5 : \underline{4}$	$\underline{5} : 6$	$\infty : \underline{6}$	$\infty : \underline{7}$

1. Sub-arrays L and R are sorted, and their last element are set to ∞. In this example, $L = [1, 2, 3, 5, \infty]$, $R = [4, 6, 7, \infty]$
2. $k = 0, i = 0, j = 0$: Since $L[0] = 1 < 4 = R[0]$, place $L[0]$ in $A[0]$ and increment i
3. $k = 1, i = 1, j = 0$: Since $L[1] = 2 < 4 = R[0]$, place $L[1]$ in $A[1]$ and increment i

4. $k = 2, i = 2, j = 0$: Since $L[2] = 3 < 4 = R[0]$, place $L[2]$ in $A[2]$ and increment i
5. $k = 3, i = 3, j = 0$: Since $L[3] = 5 > 4 = R[0]$, place $R[0]$ in $A[3]$ and increment j
6. $k = 4, i = 3, j = 1$: Since $L[3] = 5 < 6 = R[1]$, place $L[3]$ in $A[4]$ and increment i
7. $k = 5, i = 4, j = 1$: Since $L[4] = \infty > 6 = R[1]$, place $R[1]$ in $A[5]$ and increment j
8. Because the sub-array L reached its ∞ element, I will skip the rest of the steps and place rest of the elements in R to A

3.5 Quick Sort

QUICK-SORT is another divide-and-conquer sorting algorithm.

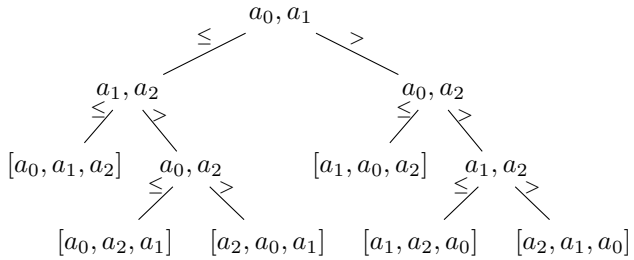
```
function QUICK-SORT(A, l, r)                          ▷ A is an array size n
    if l < r then
        m ← PARTITION(A, l, r)
        QUICK-SORT(A, l, m − 1)
        QUICK-SORT(A, m + 1, r)
    return A
```

3.5.1 Partition and Pivot

```
function PARTITION(A, l, r)                           ▷ A is an array size n
    p = A[r]
    i = l − 1
    for j = 1 to r do
        if A[j] < p then
            i ← i + 1
            SWAP(A, i, j)
    i ← i + 1
    SWAP(A, i, r)
    return i
```

3.6 Decision Tree and the Lower Bound for Comparison Sorting Algorithm

So far, we have been discussing *comparison sorting algorithms* (sorting algorithms that only reads array elements through $>, =, <$ comparison). We can draw a decision tree with all the permutations of how a comparison sorting algorithms would compare and sort the array $[a_0, a_1, a_2]$.



The decision tree for array size of 3 has $3! = 6$ leaves, and it is trivial that a decision tree for an **array with n elements will have $n!$ leaves**. The height of the decision tree represents the worst-case number of comparisons the algorithm has to make in order to sort the array.

A tree with the height h has at most 2^h leaves. Using this property, we can have the lower Big-omega bound for height of the decision tree for comparison sorting algorithms.

$$
\begin{aligned}
2^h \geq n! \therefore h \geq \log_2(n!) && (h \text{ is the height of the decision tree}) \\
h \geq \log_2(n!) = \log_2(1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n) && \\
= \log_2(1) + \log_2(2) + \cdots + \log_2(n-1) + \log_2(n) && \\
= \sum_{i=1}^{n} \log_2(i) = \sum_{i=1}^{\frac{n}{2}-1} \log_2(i) + \sum_{i=\frac{n}{2}}^{n} \log_2(i) && \\
\geq 0 + \sum_{i=\frac{n}{2}}^{n} \log_2(i) \geq 0 + \sum_{i=\frac{n}{2}}^{n} \log_2(\frac{n}{2}) && (i \text{ in the former expression is} \geq \tfrac{n}{2}) \\
= \frac{n}{2} \log_2(\frac{n}{2}) && (i = \tfrac{n}{2} \text{ to } n \text{ is exactly } \tfrac{n}{2} \text{ iterations}) \\
\in \Theta(n \log_2(n)) &&
\end{aligned}
$$

Because $h \geq \log_2(n!) \in \Theta(n \log_2(n))$, $h \in \Omega(n \log_2(n))$. Therefore, we can conclude that any **comparison sorting algorithms cannot run faster than $O(n \log_2(n))$ time** in the worst-case scenario.

3.7 Bucket Sort

BUCKET-SORT is a sorting algorithm where elements are divided into each "bucket" and a different sorting algorithm is called for each bucket. While BUCKET-SORT is a comparison sorting algorithm, it's an attempt to reduce the runtime by decreasing the number of elements that the sorting algorithm has to sort.

3.8 Counting Sort

COUNTING-SORT is a *non-comparison* sorting algorithm. It uses the extra array *count*, where its index initially represents the value of each element in A (e.g., if there are three 5's in A, $count[5] = 3$ before the "accumulation" step to determine the final index), to sort the array.

```
1:  function COUNTING-SORT(A, k)          ▷ A is an array size n, k is the max element of A
2:      count ← array size k + 1 filled with 0
3:      for i = 0 to n − 1 do             ▷ Num occurrence in each element in A, O(n)
4:          count[A[i]] ← count[A[i]] + 1

5:      for i = 1 to k do                 ▷ Accumulate the values in count from left to right, O(k)
6:          count[i] ← count[i] + count[i − 1]

7:      out ← array size n
8:      for i = n − 1 down to 0 do        ▷ Use count values to determine the index for the elements in A, O(n)
9:          out[count[A[i]] − 1] ← A[i]
10:         count[A[i]] ← count[A[i]] − 1

11:     return out
```

1. Suppose we have an array $A = [2, 5, 3, 0, 2, 3, 0, 3]$. $k = \text{MAX}(A) = 5$.

2. Initialize $count$, the array size $5+1$, with 0's. $count = [0, 0, 0, 0, 0, 0]$.
3. Count number of occurrence. $count = [2, 0, 2, 3, 0, 1]$ (e.g., 2 occurred 2 times)
4. Accumulate values of $count$ from left to right. $count = [2, 2, 4, 7, 7, 8]$ (e.g., $count[1] = 2 + 0, count[2] = 2 + 0 + 2, \ldots$)
5. Initialize out, the array size $n = 8$, with nil's. $out = [nil, nil, nil, nil, nil, nil, nil, nil]$.
6. Place each element to the out array using $count$ array
   (a) When $i = n - 1 = 7$: $A[7] = 3$ and $count[3] = 7 \Rightarrow out[7 - 1] := A[7] = 3$ and $count[3] := 7 - 1$

   out = [nil, nil, nil, nil, nil, nil, 3, nil]
   count = [2, 2, 4, 6, 7, 8]

   (b) When $i = n - 2 = 6$: $A[6] = 0$ and $count[0] = 2 \Rightarrow out[2 - 1] := A[6] = 0$ and $count[0] := 2 - 1$

   out = [nil, 0, nil, nil, nil, nil, 3, nil]
   count = [1, 2, 4, 6, 7, 8]

   (c) When $i = n - 3 = 5$: $A[5] = 3$ and $count[3] = 6 \Rightarrow out[6 - 1] := A[5] = 3$ and $count[3] := 6 - 1$

   out = [nil, 0, nil, nil, nil, 3, 3, nil]
   count = [1, 2, 4, 5, 7, 8]

   (d) ...

   out = [0, 0, 2, 2, 3, 3, 3, 5]
   count = [0, 2, 2, 4, 7, 7]

In-place?	Stable?
False	True

Because of its use for RADIX-SORT, COUNTING-SORT must be stable, and it indeed is. If there are items with the same value, it will be moved to the out array in order in the last (third) for loop.

Runtime	Space Usage
$O(n+k)$	$O(n+k)$

As the algorithm iterates both the size of the array n and the maximum element in the array k, **the algorithm runs in $O(n+k)$ time and uses $O(n+k)$ space**.

3.9 Radix Sort

RADIX-SORT is a non-comparative sorting algorithm for elements with more than one significant digits. It utilizes a stable sorting algorithm such as COUNTING-SORT to sort elements lexicographically.

```
1: function RADIX-SORT(A, k)        ▷ A is an array where the maximum dimension of an element is d
2:     for i = d down to 1 do
3:         Call a stable sorting algorithm at dimension i
4:     return A
```

3.9.1 Lexicographic Order

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d) \Leftrightarrow (x_i < y_i) \vee (x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d))$$

3.10 Chapter 3 Review

Chapter 4

Hash Tables

4.1 Division Method

4.2 Multiplication Method

4.3 Collision

4.3.1 Chaining

4.3.2 Open Addressing

Chapter 5

Search Tree

5.1 Binary Search Tree and Its Limit

5.2 2-3 Tree

5.3 Red-Black Tree

5.4 Left-Leaning Red-Black Tree

5.4.1 Deletion in LLRBT

Chapter 6

Undirected Graph

Graph is a set of vertices V and a collection of edges E that connect a pair of vertices. *Undirected graph* is a graph where edges do not have direction. *Degree of a vertex* representing how many edges is this vertex connected to.

- **Handshaking Theorem**: For any undirected graphs, $\sum_{v \text{ in } V} deg(v) = 2 \cdot |E|$
- Maximum degree of a vertex: $deg(v) \leq |V| - 1$, because a vertex cane be connected to all other vertices at most
- Maximum edge count: $|E| \leq \frac{|V|(|V|-1)}{2}$
- ***Complete graph***: A graph is said to be complete when each vertex pair is connected by a unique edge. Id est, a complete graph has the maximum number of edges ($|E| = \frac{|V|(|V|-1)}{2}$) and each vertex has the maximum degree ($deg(v) = |V| - 1$)

6.1 Adjacency Matrix and List

6.2 DFS

```
1: function DEPTH-FIRST-SEARCH(G, s)        ▷ G is a graph containing |V| vertices, s is the starting node
2:     S ← new stack
3:     Push s to S
4:     while S is not empty do
5:         u ← popped element from S
6:         if u is not visited then
7:             for w adjacent to v do
8:                 if w is not visited then
9:                     Push w to S
```

6.3 BFS

```
1: function DEPTH-FIRST-SEARCH(G, s)        ▷ G is a graph containing |V| vertices, s is the starting node
2:     Q is a new queue
```

```
3:    Enqueue s to Q

4:    Mark v as visited
5:    while Q is not empty do
6:        u ← dequeued item from Q
7:        for w adjacent to u do
8:            if w is not visited then
9:                Enqueue w to Q
10:               Mark w as visited
```

Chapter 7

Directed Graphs

7.1 Strong Connectivity

7.1.1 Brute-force Strong Connectivity Algorithm

7.1.2 Brute-force using Stack

7.1.3 Strongly Connected Components and Kosaraju's Algorithm

7.2 Directed Acyclic Graphs

7.2.1 Topological Sort

Chapter 8

Weighted Graphs

8.1 Shortest Path

8.1.1 Dijkstra's Algorithm

```
1:  function DIJKSTRA-SHORTEST-PATH(G, s)        ▷ G is a graph containing |V| vertices, s is the starting node
2:      dist ← array size |V|
3:      prev ← array size |V|
4:      Q ← a new min-heap with distance values as keys
5:      ▷ Initialization step ◁
6:      for v in Vertices do
7:          if v = s then dist[v] ← 0
8:          if v ≠ s then dist[v] ← ∞
9:          prev[u] ← −1
10:         Add a tuple (dist[v], v) to Q
11:         while Q is not empty do
12:             u ← the value with minimum dist from Q                    ▷ O(1)
```

8.1.2 Bellman-Ford Algorithm

Dijkstra should not be used on a graph with negative edge(s).

```
1:  function BELLMAN-FORD-SHORTEST-PATH(G, V)        ▷ G is the graph, V is the vertex list
2:      dist ← array size |V|
3:      prev ← array size |V|
4:      for v in V do
5:          if v = s then dist[v] ← 0
6:          if v ≠ s then dist[v] ← ∞
7:          prev[u] ← −1
8:      for i = 1 to |V| − 1 do
9:          for e in E do                        ▷ Edge e connects vertex u and v
10:             if weight[e] + dist[u] < dist[v] then
11:                 dist[v] = dist[u] + weight[e]
12:                 prev[v] = u
13:     ▷ Run the for loop once again. If the shortest distance is updated, then it means there is a negative weight
          cycle ◁
14:     for e in E do                            ▷ Edge e connects vertex u and v
```

```
15:         if weight[e] + dist[u] < dist[v] then
16:             output Negative weight edge cycle detected
17:             return
```

8.2 Articulation Points

8.3 Minimum Spanning Tree

8.3.1 Cycle and Cut Properties

8.3.2 Prim's Algorithm

8.4 Union-Find

8.4.1 Kruskal MST Algorithm

Chapter 9

Strings

9.1 Brute-force String Pattern Matching

9.2 KMP Algorithm

9.3 Trie

9.4 PATRICIA

9.5 Huffman Coding