Theo’s DSA Review

Theo Park

June 9, 2025

Contents

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

Construct a hypothesis using $T(n) = an^b$, where $b = \log(ratio)$. This section is not terribly important.

1.2 Runtime Expressions

```
for $i = 1$ to $n$ do        ▷ for (i = 1; i <= n; i++)
    A constant time operation
```

$T(n) = \sum_{i=1}^{n} c = cn$

```
for $i = 0$ to $n - 1$ do    ▷ for (i = 0; i < n; i++)
    A constant time operation
```

$T(n) = \sum_{i=0}^{n-1} c = c(n - 1 + 1) = cn$

```
1: ▷ for (int i = 0; i <= n; i += 2)    ◁
2: for $i = 0$ to $n$, increment by $2$ do
3:     $n$ time operation
```

Note that i increases by 2, so express its sequence in terms of increments by 1.

$$i = 0, 2, 4, 6, ..., n = 2 \times 0, 2 \times 1, 2 \times 2, ..., 2 \times \frac{n}{2}$$

$$T(n) = \sum_{k=0}^{\frac{n}{2}} n = n \sum_{k=0}^{\frac{n}{2}} 1 = n(\frac{n}{2} + 1) = \frac{1}{2}n^2 + n$$

```
1: for $i = 0$ to $n$ do
2:     for $j = 1$ to $n - 1$ do
3:         A constant time operation
```

$T(n) = \sum_{i=0}^{n} \sum_{j=1}^{n-1} c = \sum_{i=0}^{n} c(n-1) = c(n-1)(n+1) = cn^2 - c$

```
1: for $i = 0$ to $n$, increment by $2$ do
2:     for $j = 1$ to $n - 1$ do
3:         $n$ time operation
```

$T(n) = \sum_{k=0}^{\frac{n}{2}} \sum_{j=1}^{n-1} n = \sum_{k=0}^{\frac{n}{2}} n(n-1) = n(n-1)(\frac{n}{2} + 1) = \frac{1}{2}n^3 + \frac{1}{2}n^2 - n$

```
1: for $i = 0$ to $n$ do
2:     for $j = i$ to $n - 1$ do
3:         $n$ time operation
```

$T(n) = \sum_{i=0}^{n} \sum_{j=i}^{n-1} = \sum_{i=0}^{n} (\sum_{j=0}^{n-1} n - \sum_{j=0}^{i-1} n) = ... = n^2(n+1) - \frac{1}{2}n^2(n+1) = \frac{1}{2}n^2(n+1)$

```
for $i = 0$ to $n$, increment by 2 do
    for $j = i$ to $n - 1$ do
        $n$ time operation
```

$T(n) = \sum_{k=0}^{\frac{n}{2}} \sum j = 2k^{n-1} n = \sum_{k=0}^{\frac{n}{2}} (\sum_{j=0}^{n-1} n - \sum_{j=0}^{2k-1} n) = ... = n^2(\frac{n}{2}+1) - 2n\frac{\frac{n}{2}(\frac{n}{2}+1)}{2} = \frac{n^3}{2} + n^2 - \frac{n^3}{4} - \frac{n^2}{2} = \frac{1}{4}n^3 + \frac{1}{2}n^2$

```
for $i = 0$ to $n, i = i * 2$ do
    A constant time operation
```

Express the sequence in terms of increments by 1

- $i = 1 = 2^0$
- $i = 2 = 2^1$
- $i = 4 = 2^2$
- $i = 2^k <= n -> k <= log_2(n)$

$T(n) = \sum_{k=0}^{\log_2}(n)c = c(log_2(n)+1) = c\log_2(n)+c$

```
for $i = 1$ to $n, i = i * 3$ do
    $\log(n)$ time operation
```

$T(n) = \sum_{k=0}^{log_3(n)} \log(n) = \log(n)(\log_3(n) + 1) \approx \log(n)log(n) + \log(n)$

```
for $i = 1$ to $n$ do
    $n$ time operation
```

$T(n) = \sum_{k=0}^{\log_2(n)} = n(\log_2(n) + 1) = n\log_2(n) + n \approx n\log(n) + n$

1.3 Asymptotic Runtime Analysis

- Big-O: $f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
- Big-Omega: $f(n) \in \Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c(g)n \leq f(n)$ for all $n \geq n_0$
- Big-Theta: One number is both Big-O and Big-Omega

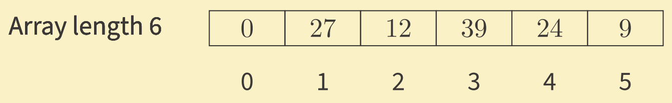
Chapter 2

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

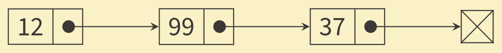
Array is a linear container of items.



- 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!**

2.2 Linked List



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

2.3 Stack

```
-       -        -        7        -      -      -
-       -        5        5        5      -      -      Error
-       3        3        3        3      3      -
init()  push(3)  push(5)  push(7)  pop()  pop()  pop()  pop()
```

Stack is a "last in, first out" (LIFO) data structure.

- `push(item)`: Inserting an item on the top – $\Theta(1)$
- `pop()`: Removes and returns the item on the top – $\Theta(1)$
- `peek()`: Returns the item on the top
- `size()`: Returns the number of item in the stack
- `isEmpty()`: Checks if size == 0

2.4 Queue

```
- - -    3 - -    3 5 -    3 5 7    - 5 7    - - 7    - - -    Error
init()   e(3)     e(5)     e(7)      d()      d()      d()      d()
```

Queue is a "first in, first out" (FIFO) data structure.

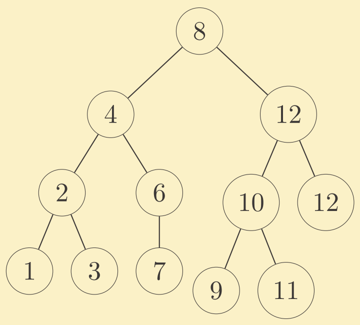
- `enqueue(item)`: Inserts an item at the end - **Theta(1)**
- `dequeue()`: Removes the item at the front of the queue and return it - **Theta(1)**
- `peek()`: Returns the item at the front of the queue w/o removing it
- `size()`: Returns the number of the items in the queue
- `isEmpty()`: Checks if size == 0

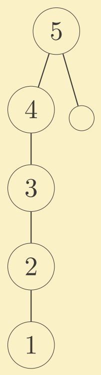
2.4.1 Implementing Queue Using a Circular Array

```
function CONSTRUCTOR($k$)                    ▷ $k$ is the initial size of the array
    $Q$ ← an array of size $k$
    size ← $0$
    head ← $0$
    tail ← $k - 1$
function ENQUEUE($n$)                        ▷ $n$ is the new item to add
    if size = $k$ then return False

    tail ← (tail + 1) $\mod k$
    Q[tail] ← $n$
    size ← size $+1$
function DEQUEUE
    if size = $0$ then return False

    tmp ← Q[head]
    Q[head] ← nil
    head ← (head + 1) $\mod k$
    size ← size $-1$
    return tmp
```

2.5 Tree

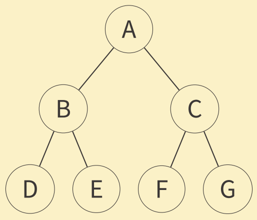
- Search: $O(\log(n))$ for a balanced tree (right), $O(n)$ for unbalanced tree (left)





- **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 Heapify

1. Define a function MAX-HEAPIFY that given an index i, it "sinks down" $A[i]$ to find its correct heap position.
2. Starting at the last non-leaf node (i.e., parent of the last node), run MAX-HEAPIFY to heapify the subtree.

$O(n)$

```python
def max_heapify(arr, i):
    n = len(arr)

    l = 2 * i + 1
    r = 2 * i + 2

    largest = i
    if l < n and arr[l] > arr[i]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, largest)


def build_heap(arr):
    n = len(arr)

    lastLeftNode = (n // 2) - 1  # floor(n / 2) - 1
    for i in range(lastLeftNode, -1, -1):
        max_heapify(arr, i)
```

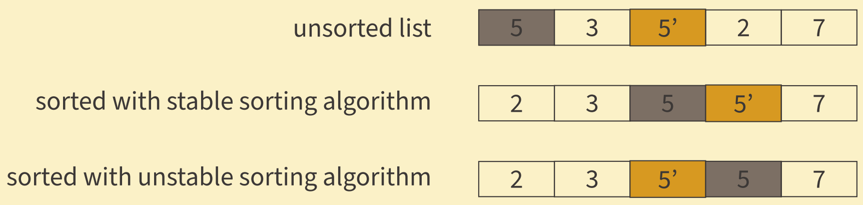
Chapter 3

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.


unsorted list
5
3
5'
2
7
sorted with stable sorting algorithm
2
3
5
5'
7
sorted with unstable sorting algorithm
2
3
5'
5
7

- 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.

```python
def bubble_sort(arr):
    n = len(arr)
    repeat = True
    while repeat:
        repeat = False

        for i in range(n - 1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                repeat = True

    return arr
```

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.

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_idx = i

        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        if i != min_idx:
            arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
```

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.

```
def insertion_sort(arr):
    n = len(arr)

    for i in range(1, n):
        j = i - 1

        while j >= 0 and arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            j -= 1

    return arr
```

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.

```
function SHELL-SORT($A, H$)    ▷ $A$ is an array size $n$, $H$ is the array size $m$ containing gap values
    for $h = 0$ to $m - 1$ do
        $gap \leftarrow H[h]$
        for $i = 1$ to $n - 1$ do
            $j \leftarrow i - 1$
            while $j \geq 0$ and $j + gap < n$ do
                if $A[j] > A[j + gap]$ then
                    SWAP($A, j, j + gap$)
                    $j \leftarrow j - gap$

    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.

```python
def heap_sort(arr):
    n = len(arr)

    build_heap(arr)

    def sortdown(i):
        # move the first element of the heap (largest) to the back
        arr[i], arr[0] = arr[0], arr[i]
        # rebuild heap with the first i elements
        max_heapify(arr, i, 0)

    for i in range(n - 1, 0, -1):
        sortdown(i)

    return arr
```

The algorithm first builds the heap from the array elements (refer to section 2.6.1 for the MAX_HEAPIFY and BUILD_HEAP functions). Then the algorithm calls SORT-DOWN from the last heap elements down to the first.

3.3.1 Sort Down Algorithm

```python
def sortdown(i):
    # move the first element of the heap (largest) to the back
    arr[i], arr[0] = arr[0], arr[i]
    # rebuild heap with the first i elements
    max_heapify(arr, i, 0)
```

Suppose we have a max-heap $A = [7, 6, 3, 5, 4, 1]$. Using the property of max-heap that the largest element is always placed in the first index, HEAP-SORT performs SORT-DOWN $n - 1$ times, which places the last element to a correct position and rebuild the heap on the last of the element.

One can call SWIM-UP instead of SWIM-DOWN to repair the heap. However, the former takes $O(n \log n)$ time and the latter takes $O(n)$ time.

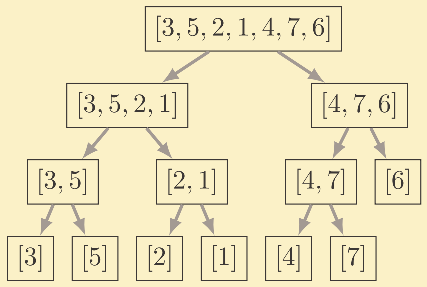
1. $i = 5$:
   SWAP$(A, 5, 0)$: [1, 6, 3, 5, 4, ~~7~~]
   SWIM-DOWN$(5 - 1)$ (rebuild heap from index 0 to 4): [6, 5, 3, 1, 4 ~~7~~]
2. $i = 4$:
   SWAP$(A, 4, 0)$: [4, 5, 3, 1, ~~6~~, ~~7~~]
   SWIM-DOWN$(4 - 1)$: [5, 4, 3, 1, ~~6~~, ~~7~~]
3. $i = 3$:
   SWAP$(A, 3, 0)$: [1, 4, 3, ~~5~~, ~~6~~, ~~7~~]
   SWIM-DOWN$(3 - 1)$: [4, 1, 3, ~~5~~, ~~6~~, ~~7~~]
4. $i = 2$:
   SWAP$(A, 2, 0)$: [3, 1, ~~4~~, ~~5~~, ~~6~~, ~~7~~]
   SWIM-DOWN$(2 - 1)$: [3, 1, ~~4~~, ~~5~~, ~~6~~, ~~7~~]
5. $i = 1$:
   SWAP$(a, 1, 0)$: [1, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~]
   SWIM-DOWN$(1 - 1)$: [1, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~]

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.

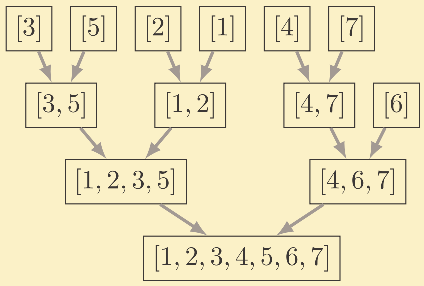
```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [None] * (n1 + 1)
    R = [None] * (n2 + 1)

    # arrssign elements to each array
    for i in range(n1):
        L[i] = arr[l + i]
    for j in range(n2):
        R[j] = arr[m + j + 1]

    L[n1], R[n2] = float("inf"), float("inf")
    i, j = 0, 0

    for k in range(l, r + 1):
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1

    return arr
```

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.



```
def merge_sort(arr, l, r):
    if l < r:
        m = (l + r) // 2
        merge_sort(arr, l, m)
        merge_sort(arr, m + 1, r)
        merge(arr, l, m, r)
    return arr
```

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

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

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.

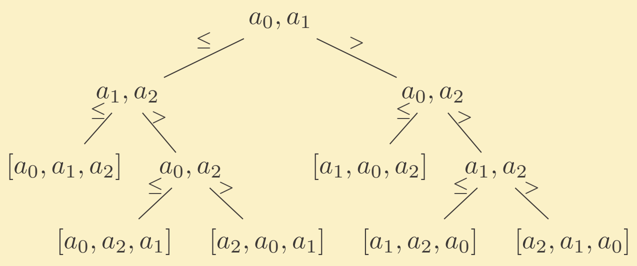
```
def quick_sort(arr, l, r):
    if l < r:
        m = partition(arr, l, r)
        quick_sort(arr, l, m - 1)
        quick_sort(arr, m + 1, r)
    return arr
```

3.5.1 Partition and Pivot

```
def partition(arr, l, r):
    p = arr[r]   # rightmost element as the pivot
    i = l - 1
    for j in range(l, r + 1):
        if arr[j] < p:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    i += 1
    arr[i], arr[r] = arr[r], arr[i]
    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.

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

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.

```
def countingSort(arr):
    n = len(arr)

    # Step 1 O(n): Find the maximum element and initialize count array
    k = max(arr)
    cnt = [0] * (k + 1)

    # Step 2 O(n): Find the number of occurences in each element in the array
    for i in range(n):
        cnt[arr[i]] += 1

    # Step 3 O(k): Convert count into a prefix sum array of itself
    for i in range(1, k + 1):
        cnt[i] += cnt[i - 1]

    # Step 4 O(n): Use the count array to find the index of each element
    out = [None] * n
    for i in range(n - 1, -1, -1):
        out[cnt[arr[i]] - 1] = arr[i]
        cnt[arr[i]] -= 1

    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.

```
function Radix-sort($A, k$)        ▷ $A$ is an array where the maximum dimension of an element is $d$
    for $i = d$ down to $1$ do
        Call a stable sorting algorithm at dimension $i$
    return $A$
```

3.9.1 Lexicographic Order

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

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.

Undirected graph has a few properties:

- **Handshaking Theorem**: $\sum_{v \in V} deg(v) = 2 \cdot |E|$ (Sum of degrees of all vertices equal to the number of edges)
- Maximum degree of a vertex: $deg(v) \leq |V| - 1$ (connected to every vertices except for itself)
- 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 ($|E| = \frac{|V|(|V|-1)}{2}$) (implies that $\forall_{v \in V}(deg(v) = |V| - 1)$)

Path and Cycle:

- Path: Sequence of vertices connected by edges
  - Euler Path: A path that visits every edge exactly once
- Cycle: A path that starts and ends on the same vertex
  - An Euler path that starts and ends on the same vertex

6.1 Adjacency Matrix and List

```
class Graph:
    # e.g.,: Graph(n = 3, edges = [[0,1],[1,2],[0,2]])
    # creates a graph with 3 nodes indexed 0, 1, 2,
    # with each connected to each other
    def __init__(self, n: int, edges: list[list[int]]):
        self.n = n

        self.adjList = [[] for _ in range(n)]
        for u, v in edges:
            self.adjList[u].append(v)
            self.adjList[v].append(u)

        self.adjMatrix = [[float("inf") for _ in range(n)] for _ in range(n)]
        for i in range(v):
            self.adjMatrix[i][i] = 0
        for u, v in edges:
            self.adjMatrix[u][v] = 1
            self.adjMatrix[v][u] = 1
```

-	Matrix	List
Space	$O(\|V\|)$	$O(\|V\| + \|E\|)$
Adding an edge	$O(1)$	$O(1)$
Checking adjacent vertices (to vertex v)	$O(1)$	$O(deg(v))$
Iteration	$O(\|V\|)$	$O(deg(v))$

6.2 BFS

For finding a shortest path in a undirected graph.
Runtime: $O(|V| + |E|)$

```
def bfs(self, v: int):  # v is the starting node
    q = deque()
    dist = [-1] * self.n
    path = [[] for _ in range(self.n)]
    visited = [False] * self.n

    q.append(v)
    dist[v] = 0
    path[v] = [0]
    visited[v] = True

    while q:
        u = q.pop()

        for w in self.adjList[u]:
            if not visited[w]:
                q.append(w)
                dist[w] = dist[u] + 1
                path[w] = path[u] + [w]
                visited[w] = True

    return dist, path, visited
```

6.3 DFS

For connectivity.
Iterative implementation using a stack:

```
def dfs_iterative(self, v: int):  # v is the starting node
    st = []
    dist = [-1] * self.n
    path = [[] for _ in range(self.n)]
    visited = [False] * self.n

    st.append(v)
    dist[v] = 0
    path[v] = [0]
    visited[v] = True

    while st:
        u = st.pop()

        for w in self.adjList[u]:
            if not visited[w]:
                st.append(w)
                dist[w] = dist[u] + 1
                path[w] = path[u] + [w]
                visited[w] = True

    return dist, path, visited
```

Using the recursive implementation to count the number of connected components. You can, in fact, change the DFS_RECURSIVE_DRIVER to BFS and the algorithm still works as expected.

```
    # helper function for finding connected components using recursive DFS
    def dfs_recursive_driver(self, v: int,
                             visited,
                             connectedComponents,
                             currConnectedCompIdx):
        visited[v] = True
        connectedComponents[currConnectedCompIdx].append(v)
        for w in self.adjList[v]:
            if not visited[w]:
                self.dfs_recursive_driver(
                    w, visited, connectedComponents, currConnectedCompIdx)

    def find_connected_comp(self):
        visited = [False] * self.n

        connectedComponents = []
        currConnectedCompIdx = -1
        for v in range(self.n):
            if not visited[v]:
                currConnectedCompIdx += 1
                connectedComponents.append([])
                self.dfs_recursive_driver(
                    v, visited, connectedComponents, currConnectedCompIdx)

        return connectedComponents
```

Chapter 7

Directed Graphs

The overall constructor is identical to that of undirected graph except that for an edge directing $u \to v$, only v is added to the adjacency list/matrix of u and not the other way around.
I also added a recursive implementation of functional DFS method that modifies external VISITED and PATH arrays.

```
class Graph:
    # e.g.,: Graph(n = 3, edges = [[0,1],[1,2],[0,2]])
    # creates a graph with 3 nodes indexed 0, 1, 2,
    # with 0 -> 1, 0 -> 2, 1 -> 2
    def __init__(self, n: int, edges: list[list[int]]):
        self.n = n
        self.edges = edges

        self.adjList = [[] for _ in range(n)]
        for u, v in edges:
            self.adjList[u].append(v)

        self.adjMatrix = [[float("inf") for _ in range(n)] for _ in range(n)]
        for i in range(v):
            self.adjMatrix[i][i] = 0
        for u, v in edges:
            self.adjMatrix[u][v] = 1

    def dfs(self, v, visited, path):
        visited[v] = True
        path.append(v)

        for w in self.adjList[v]:
            if not visited[w]:
                self.dfs(w, visited, path)
```

7.1 Strong Connectivity

A directed graph is strongly connected if every vertex is reachable from any other vertex. To test the strong connectivity, use the following algorithm, which takes $O(|V| + |E|)$.

```
1: function IS-STRONGLY-CONNECTED($G$)
2:     Pick a vertex $v$ in $G$
3:     Perform DFS($G, v$)
4:     If there is an unvisited vertex, return false
```

```
5:     Compute $G^T$ (transpose of $G$, flip all the edges)
6:     Perform DFS($G^T, v$)
7:     IF there is an unvisitied vertex, return false
8:     return true
```

7.1.1 Strongly Connected Components and Kosaraju's Algorithm

$O(|V| + |E|)$

1. Perform DFS with a global stack. For each run, append the them in the reversal order of how they would be appended to the path in the regular DFS (this is essentially picking a vertex with the lowest in-degrees)
2. Iterating through the stack, run the DFS on a transposed graph
3. The set of visited vertex from this run will form a strongly connected component

```
def kojaraju(self):
    # Run DFS to make a stack of vertices based on exit time
    visited = [False] * self.n
    indegreePriorityStack = []

    def visit(v):
        visited[v] = True

        for w in self.adjList[v]:
            if not visited[w]:
                visit(w)
        indegreePriorityStack.append(v)

    for v in range(self.n):
        if not visited[v]:
            visit(v)

    # DFS on the transposed graph, in the order of exit time
    transposedG = Graph(self.n, [[v, u] for u, v in self.edges])
    visitedTranspose = [False] * self.n
    scc = []
    while indegreePriorityStack:
        v = indegreePriorityStack.pop()
        if not visitedTranspose[v]:
            path = []
            transposedG.dfs(v, visitedTranspose, path)
            scc.append(sorted(path))

    return scc
```

7.2 Articulation Points

An articulation point is a vertex such that removing it from the graph increases the number of connected components. This is applicable for undirected graph as well.

```
1: function AP($G, s$, discovery)                    ▷ $s$ is the starting vertex
2:     Mark $s$ as visited
3:     discovery[$s$] ← $d$
```

```
4:     $\text{low}[s] \leftarrow d$
5:     for $w$ adj to $s$ do
6:         if $v$ is not visited then AP($g$, $s$, $d$)
7:         $\text{low}[s] \leftarrow \text{MIN}(\text{low}[s], \text{low}[w])$
8:         if $\text{low}[s] \geq \text{discovery}[s]$ then
9:             if $s$ is not root or $s$ has more than 1 child in the DFS tree then
10:                PRINT($s$ is the AP)
```

7.3 Directed Acyclic Graphs

DAG is a directed graph without a cycle.

7.3.1 Topological Sort

```
1: function KAHN-TOPOLOGICAL-SORT($G$)                ▷ $G$ is the graph object
2:     $H \leftarrow$ a specialized graph able to keep track of in-degrees of vertices, initialized with $G$
3:     $Ordering \leftarrow$ array of size $|V|$
4:     $n \leftarrow 0$

5:     while $H$ is not empty do
6:         $v \leftarrow$ a vertex with in-degree 0 from $H$
7:         $Ordering[n] = v$
8:         $P$.remove($v$ and all connect edges)
9:         $n \leftarrow n + 1$
```

If G is not a DAG (i.e., contains cycle(s)), H will still have edge(s) left in it because vertices in a cycle will never reach in-degree 0. This means that you can check if a graph is DAG by running the topological sort and asserting $Ordering$.length $= |V|$
Here is an actual implementation of the algorithm using queue.

```python
def kahnTopologicalSort(self):
    indeg = [0] * self.n

    for u, v in self.edges:
        indeg[v] += 1

    q = deque()
    for v in range(self.n):
        if indeg[v] == 0:
            q.append(v)

    ordering = []
    while q:
        u = q.pop()
        ordering.append(u)

        for w in self.adjList[u]:
            indeg[w] -= 1  # "removing" the edge
            if indeg[w] == 0:
                q.append(w)
    # if we do not have all the nodes in the ordering list,
    # it means that the graph is not DAG.
    # It is because if the graph is DAG, all nodes will eventually have
    # zero indegree after edge removal
    return [] if len(ordering) != self.n else ordering
```

Chapter 8

Weighted Graphs

The constructor for weighted undirected graph takes an edge list with an extra information, namely the weight of the edge. When building an adjacency list/matrix, a tuple containing the destination vertex and the weight is pushed.

```python
class Graph:
    # e.g.,: Graph(n = 3, edges = [[0,1,3],[1,2,1],[0,2,6]])
    # creates a graph with 3 nodes indexed 0, 1, 2,
    # with 0 - 1 (weight 3), 0 - 2 (weight 6), 1 - 2 (weight 1)
    def __init__(self, n: int, edges: list[list[int]]):
        self.n = n
        self.edges = edges

        self.adjList = [[] for _ in range(n)]
        for u, v, weight in edges:
            self.adjList[u].append((v, weight))
            self.adjList[v].append((u, weight))

        self.adjMatrix = [[float("inf") for _ in range(n)] for _ in range(n)]
        for i in range(v):
            self.adjMatrix[i][i] = 0
        for u, v, w in edges:
            self.adjMatrix[u][v] = weight
            self.adjMatrix[v][u] = weight
```

8.1 Shortest Path

8.1.1 Initialization and Edge Relaxation for Single-source Shortest-path Algorithms

```
1: function INITIALIZE-SINGLE-SOURCE($G, s$)          ▷ $G$ is the graph, $s$ is the starting vertex
2:     $dist \leftarrow$ array size $|V|$
3:     $prev \leftarrow$ array size $|V|$
4:     for $v \in V$ do
5:         if $v = s$ then $dist[v] \leftarrow 0$
6:         if $v \neq s$ then $dist[v] \leftarrow \infty$
7:         $prev[u] \leftarrow$ nil                          ▷ or $-1$
       return $dist, prev$
```

```
function RELAX($u, v$)
    $d \leftarrow dist[u] + weight(u, v)$
    if $dist[v] > d$ then
        $dist[v] = d$
        $prev[v] = u$
```

Simply put, relaxation is a process of greedily updating the DIST value of a vertex v to the shorter of the current DIST[v] or newly discovered edge. These two are easier to understand in action.

8.1.2 Bellman-Ford Algorithm

Bellman-Ford algorithm computes shortest-path by performing relaxation for every edge $|V| - 1$ times (maximum number of edges in a simple path). Then it attempts to relax each edge once more, and any additional relaxation would indicate the existence of a negative weight cycle (cycle involving an edge with a negative weight; one can travel this over and over to reduce the cost of path, rendering shortest path meaningless).

```
function BELLMAN-FORD-SHORTEST-PATH($G, V$)        ▷ $G$ is the graph, $V$ is the vertex list
    ▷ Initialize-Single-Source: $O(|V|)$        ◁
    $dist \leftarrow$ array size $|V|$
    $prev \leftarrow$ array size $|V|$
    for $v \in V$ do
        if $v = s$ then $dist[v] \leftarrow 0$
        if $v \neq s$ then $dist[v] \leftarrow \infty$
        $prev[u] \leftarrow -1$

    ▷ Visiting each edges and relaxing: $O(|V||E|)$        ◁
    for $i = 1$ to $|V| - 1$ do
        for $e \in E$ do        ▷ Edge $e$ connects vertex $u$ and $v$
            if $weight[e] + dist[u] < dist[v]$ then
                $dist[v] = dist[u] + weight[e]$
                $prev[v] = u$
        if If no relaxation happened in the inner loop then
            ▷ This means that every edges are relaxed to its shortest path        ◁
            return true, $dist, prev$

    ▷ Visiting each edge to check for negative weight cycle: $O(|E|)$        ◁
    for $e \in E$ do        ▷ Edge $e$ connects vertex $u$ and $v$
        if $weight[e] + dist[u] < dist[v]$ then
            output Negative weight edge cycle detected
            return false

    return true, $dist, prev$
```

8.1.3 Dijkstra's Algorithm

Dijkstra is much like BFS< but instead of regular queue, it uses min-heap priority queue to visit a vertex with minimum cost. The following pseudocode is a textbook implementation of Dijkstra's algorithm, using a min-heap priority queue that supports modifying priorities of an element after they have been enqueued.

```
function DIJKSTRA($G, s$)        ▷ $s$ is the starting vertex
    ▷ Initialize-Single-Source: $O(|V|)$        ◁
    dist $\leftarrow$ array size $|V|$
```

```
4:  $\mathrm{prev} \leftarrow$ array size $|V|$
5:  for $v \in V$ do
6:      if $v = s$ then $\mathrm{dist}[v] \leftarrow 0$
7:      if $v \neq s$ then $\mathrm{dist}[v] \leftarrow \infty$
8:      $\mathrm{prev}[u] \leftarrow -1$

9:  $Q \leftarrow$ a min-heap priority queue
10: for all $v \in G.V$ do
11:     ▷ This inserts $(0, s)$ for the starting vertex $s$ and $(\infty, v)$ for every other vertices ◁
12:     $Q$.ENQUEUE($\mathrm{dist}[v], v$)

13: while $Q \neq \emptyset$ do
14:     $(\_, u) \leftarrow Q$.DEQUEUE
15:     for all $v \in G$.ADJACENCY LIST OF $u$ do
16:         ▷ Edge relaxation ◁
17:         $d \leftarrow \mathrm{dist}[u] + G$.WEIGHT($(u, v)$)
18:         if $d < \mathrm{dist}[v]$ then
19:             $\mathrm{dist}[v] = d$
20:             $\mathrm{prev}[v] = u$
21:             $Q$.SET-PRIORITY($(d, v)$)
```

In real life, you seldom find an implementation of priority queue with SET-PRIORITY function. Instead, you can simply insert the new tuple into the queue when relaxing an edge.

```python
def dijkstra(self, v):
    dist = [float("inf")] * self.n
    prev = [-1] * self.n

    dist[v] = 0
    q = [(0, v)]

    while q:
        priority, u = heappop(q)

        for w, weight in self.adjList[u]:
            candidate = dist[u] + weight
            if candidate < dist[w]:
                dist[w] = candidate
                prev[w] = u
                heappush(q, (candidate, w))

    return dist, prev
```

This version might cause duplicate vertices in the queue. For example, if a vertex v gets relaxed to the distance value of 9 by a vertex u, then it later gets relaxed to 6 by a vertex w, then there will be both $(6, v)$ and $(9, v)$ present in the queue. It will not cause any problem since by the time $(9, v)$ is dequeued, vertices adjacent to v would have been relaxed by $(6, v)$.
To optimize this, we can check if the priority we dequeued matches the value in the dist array.

```python
def dijkstra(self, v):
    dist = [float("inf")] * self.n
    prev = [-1] * self.n

    dist[v] = 0
    q = [(0, v)]
```

```
        while q:
            priority, u = heappop(q)

            if priority == dist[u]:  # or p <= dist[u]
                for w, weight in self.adjList[u]:
                    candidate = dist[u] + weight
                    if candidate < dist[w]:
                        dist[w] = candidate
                        prev[w] = u
                        heappush(q, (candidate, w))

        return dist, prev
```

Because we only enqueue when a vertex is relaxed, $p < \text{dist}[u]$ would never happen unless there is a negative weight cycle.

8.1.4 Floyd-Warshall All Pairs Shortest Path Algorithm

Dijkstra and Bellman-Ford are examples of "single source" shortest path algorithm, computing shortest path from a given vertex to other nodes. Floyd-Warshall algorithm is an "all pairs" shortest path algorithm, computing the shortest path between all pairs of vertices. It uses an adjacency matrix to compute shortest paths in $O(|V|^3)$ time.

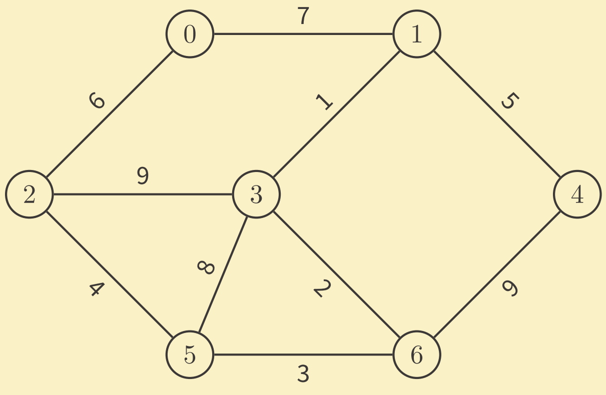
```
    def floydWarshall(self):
        # Making a copy of adjacency matrix
        dist = [row[:] for row in self.adjMatrix]

        for k in range(self.n):
            for i in range(self.n):
                for j in range(self.n):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

        return dist
```

8.2 Minimum Spanning Tree

- Minimum: $\sum weight$ is minimum
- Spanning: All vertices in the graph are connected
- Tree: No cycle

The subset of edges that connects all vertices in the minimum cost.

8.2.1 Cycle and Cut Properties



There are two fundamental properties of MST:

1. *Cycle Property*: For any cycle C in the graph, if the weight of an edge $e \in C$ is higher than any of individual weights of all other edges in C, then its edge cannot belong in the MST.
   In the example above, $(6, 4)$ in the cycle $2 - 0 - 1 - 4 - 6 - 5$ and $(2, 5)$ in the cycle $2 - 3 - 5$ cannot be in the MST.
2. *Cut Property*: For any *cut* (subdivision of graph with disjoint) C in the graph, if the weight of an edge e in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MST of the graph.
   If we remove $(0, 1)$, $(2, 3)$, $(2, 5)$, then graph separates into $(0, 1)$ and rest of the vertices. Thus, the lowest cost edge $(2, 5)$ must be in the MST.

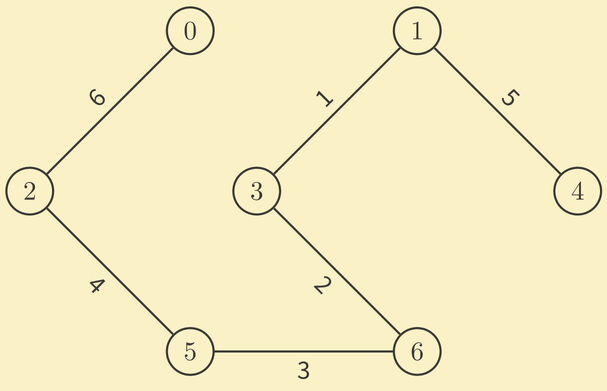
8.2.2 Prim's Algorithm

Prim's algorithm is essentially Dijkstra but for finding MST.

```
def primMST(self, s, visited):
    dist = [float("inf")] * self.n
    prev = [-1] * self.n

    dist[s] = 0
    q = [(0, s)]

    cost = 0
    mstEdges = []

    while q:
        priority, u = heappop(q)

        # Ensure that the dequed vertex has not been relaxed yet.
        # For the algorithm with implementation without set_priority method
        # See Dijkstra section for more information.
        if priority == dist[u]:
            # update the MST information
            visited[u] = True
            cost += priority
            if prev[u] != -1:
                mstEdges.append((prev[u], u, priority))

            for v, weight in self.adjList[u]:
                if not visited[v]:
                    if weight < dist[v]:
                        prev[v] = u
                        dist[v] = weight
                        heappush(q, (weight, v))

    return cost, mstEdges
```

Running the algorithm on the graph in the section 8.2.1, we get the following MST.

```
g = Graph(7, [
    [0, 1, 7],
    [0, 2, 6],
    [1, 4, 5],
    [1, 3, 1],
    [2, 3, 9],
    [2, 5, 4],
    [3, 5, 8],
    [3, 6, 2],
    [4, 6, 9],
    [5, 6, 3]
])
visited = [False] * g.n
print(g.primMST(0, visited))
...
(21, [(0, 2, 6), (2, 5, 4), (5, 6, 3), (6, 3, 2), (3, 1, 1), (1, 4, 5)])
```



The Prim's MST algorithm runs in $O(|E|\log(|V|))$.
The algorithm assumes that the graph is fully connected, so the starting node s can be any node. However, if the graph is disconnected, you need to run the algorithm on every vertices.

```
def primAllVertices(self):
    cost = 0
    mstEdges = []
    visited = [False] * self.n
    for v in range(self.n):
        if not visited[v]:
            localCost, localMstEdges = self.primMST(v, visited)
            cost += localCost
            mstEdges.extend(localMstEdges)

    return cost, mstEdges
```

8.2.3 Union-Find

8.2.4 Krustal MST Algorithm

Chapter 9

Strings