Algorithm1

April 26, 2023

Contents

1 Fundamentals **1**
1.1 Steps to Developing Algorithms . . . 1
1.2 Analysis of Algorithms . . . 1
1.3 Abstract Data Types (ADTs) . . . 2

2 Sort Algorithms **3**
2.1 Elementary Sorts . . . 3
2.2 Merge Sort . . . 4
2.3 Quick Sort . . . 4
2.4 Heap Sort . . . 5
2.5 Summary . . . 7

3 Search Algorithms **7**
3.1 Symbol Table . . . 7
3.2 Binary Search Tree . . . 7
3.3 BST Additional Operations . . . 8
3.4 Balanced Search Tree . . . 10
3.5 Hash Tables . . . 12
3.6 Summary . . . 14

4 Graph Algorithms **15**
4.1 Undirected Graphs . . . 15
4.2 Directed Graphs . . . 18
4.3 Minimum Spanning Tree . . . 21
4.4 Shortest Path . . . 24

5 String Algorithms **28**
5.1 String Sorting Algorithms . . . 29
5.2 String Search . . . 30
5.3 Substring Search . . . 33
5.4 Compression . . . 35

1 Fundamentals

1.1 Steps to Developing Algorithms

1. Modelling the problem
2. Define data structures and algorithms to solve it
3. Define and analyse cost
4. Iterate until satisfied

1.2 Analysis of Algorithms

Observation

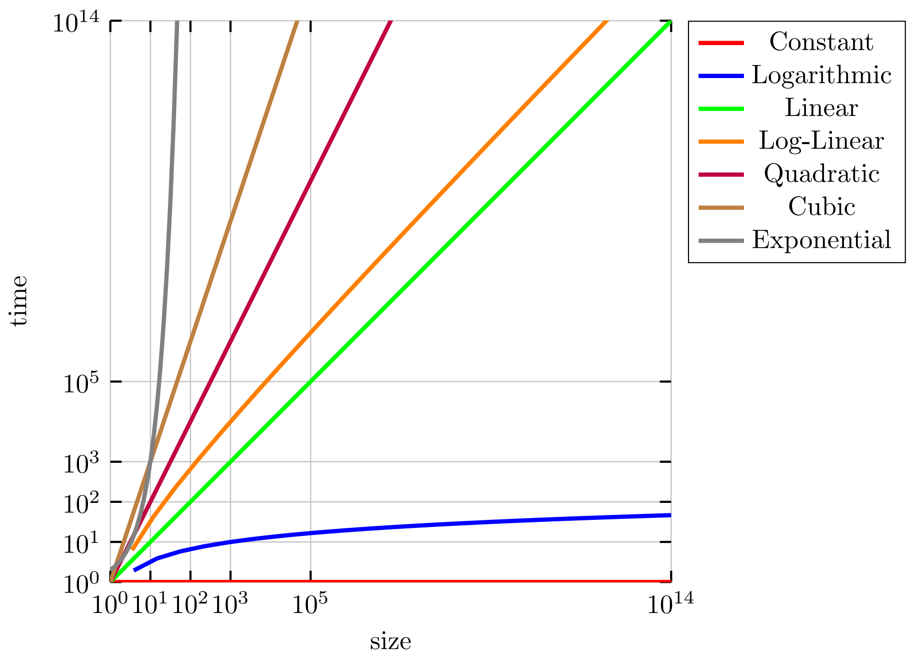
(But usually you can't afford to build the full solutions and run it multiple times)

System independent factors	**System dependent factors**
Algorithm	Hardware
Input data	Software (programming language, compiler)
	System

Mathematical Models

- focus on most costly and most frequently executed operations
- ignore lower order terms (tilde $\sim$ notation)
- we do not ignore the constant value that is associated at the leading term (i.e. $\sim 2N$)

Order of growth classifications



Types of Analyses

Best case - Lower bound	**Worst case - Upper bound**	**Average case - Expected cost**
Big Omega ($\Omega()$)	Big Oh ($O()$)	Big Theta ($\Theta()$)
Determined by "easiest" input	Determined by "most difficult" input	Need a model for "random" input
Provides a goal for all inputs	Provides a guarantee for all inputs	Provides a way to predict performance

1.3 Abstract Data Types (ADTs)

- Linear (Array, List, Stack, Queue, Set and Bag, Map, Priority Queue)
- Non-linear (Tree, Graph)

Array	**List**	**Stack**
fixed number of items, indexable	dynamic number of items	A list that last-in, first-out

`set(index, element)`
`get(index)`
	`append(item)`
	`prepend(item)`
	`head()`
	`tail()`
		`push (item)`
		`pop()`
		`isEmpty()`

Queue	**Set and Bag**
A list that first-in, first-out	Set and Bag have unindexed, unordered elements. Set's element is unrepeated. Bag's element is possibly duplicated

`enqueue (item)`
`dequeue()`
`head()`
	`insert (item)`
	`remove(item)`
	`contains(item)`

Map	**Priority Queue**
A list that can hold data in (key, value) pairs. Keys are unique, and can only hold one value	A queue where items are inserted/removed based on a given priority

`insert (key, value)`
`remove(key)`
`update(key, value)`
`lookup(key)`
	`enqueue(item, priority)`
	`dequeue()`

ADTs and Data Structures

ADTs	Common Implementations (Data Structures)
Array	array
List	array, linked list
Queue	array, linked list
Stack	array, linked list
Set and Bag	hash table
Map	hash table
Priority Queue	heap

2 Sort Algorithms

2.1 Elementary Sorts

Selection Sort

Key idea:

1. scan array from left to right
2. in iteration `i`, find the index `min` of the smallest remaining entry in the array
3. swap `a[i]` and `a[min]`

```
def selectionSort(xs: list) -> None:
for i in range(len(xs)):
    min = i
    j = i+1
    while j < len(xs):
        if xs[j] < xs[min]:
            min = j
        j += 1
    if min != i:
        xs[i], xs[min] = xs[min], xs[i]
    i += 1
```

Best case	Worst case	Average case
N^2	N^2	N^2

Insert Sort

Key idea:

- scan array from left to right
- in iteration i, swap a[i] with each larger entry to its left

```
def insertSort(xs: list) -> None:
for i in range(len(xs)):
    j = i
    while j > 0:
        if xs[j] < xs[j-1]:
            xs[j], xs[j-1] = xs[j-1], xs[j]
        j -= 1
```

Best case	Worst case	Average case
N	N^2	N^2

2.2 Merge Sort

Key idea - Divide and conquer

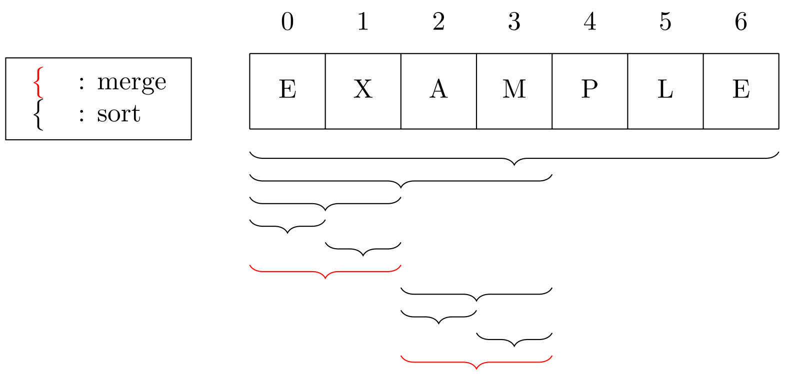
- Divide an array in two halves
- Sort each half separately
- Merge the two halves

Pseudocode

```python
def merge(xs: list, aux: list, lo: int, mid: int, hi: int) -> None:
    k = lo
    while k <= hi:
        aux[k] = xs[k]
        k += 1

    i, j, k = lo, mid+1, lo
    while k <= hi:
        if i > mid:
            xs[k], j, k = aux[j], j+1, k+1
        elif j > hi:
            xs[k], i, k = aux[i], i+1, k+1
        elif aux[j] < aux[i]:
            xs[k], j, k = aux[j], j+1, k+1
        else:
            xs[k], i, k = aux[i], i+1, k+1


def sort(xs: list, aux: list, lo: int, hi: int) -> None:
    if hi <= lo:
        return

    mid = lo + (hi-lo)//2

    sort(xs, aux, lo, mid)
    sort(xs, aux, mid+1, hi)
    merge(xs, aux, lo, mid, hi)
```

Best case	**Worst case**	**Average case**
$N \lg N$	$N \lg N$	$N \lg N$

Divide and Sort Steps




```
sort(0,6)
sort(0,3)
sort(0,1)
sort(0,0)
sort(1,1)
merge(0,1)
sort(2,3)
sort(2,2)
sort(3,3)
merge(0,3)
```

2.3 Quick Sort

Key idea

- Shuffle the array `a[]`
- Partition `a[]` so that, for some `j`
  - Entry `a[j]` is in place

  - a[i]<=a[j] for any i<j
  - a[i]>=a[j] for any i>j

- Sort each partition recursively

Pseudocode

```python
def partition(xs: list, lo: int, hi: int) -> int:
    i, j, pivot = lo, hi + 1, xs[lo]
    while True:
        i, j = i + 1, j - 1
        while xs[i] < pivot:
            i += 1
            if i == hi:
                break
        while pivot < xs[j]:
            j -= 1
            if j == lo:
                break
        if i >= j:
            break
        xs[i], xs[j] = xs[j], xs[i]
    xs[lo], xs[j] = xs[j], xs[lo]
    return j


def sort(xs: list, lo: int, hi: int) -> None:
    if hi <= lo:
        return
    j = partition(xs, lo, hi)
    sort(xs, lo, j - 1)
    sort(xs, j + 1, hi)
```

Best case	Worst case	Average case
N^2	N lg N	N lg N

Practical Improvements

- Use insertion sort on small subarrays (¡10 items)
- Best choice pivot = median value (e.g., median of 3 random items)

Some Properties

- Quick sort running time on duplicate keys could be quadratic

2.4 Heap Sort

Binary Heap Data Structure

```python
class MaxPriorityQueue:
    def __init__(self) -> None:
        self.heap = [None]
        self.tail = 0

    def enqueue(self, key) -> None:
        self.heap.append(key)
        self.tail += 1
        self._swim(self.tail)

    def dequeue(self):
        if self.tail == 0:
            return None
        max = self.heap[1]
        self.heap[1], self.heap[self.tail] = self.heap[self.tail], self.heap[1]
        self.heap[self.tail] = None
        self.tail -= 1
        self._sink(1)
        self.heap.pop()
```

```python
        return max

    def _swim(self, i: int) -> None:
        while i > 1 and self.heap[i//2] < self.heap[i]:
            self.heap[i], self.heap[i//2] = self.heap[i//2], self.heap[i]
            i //= 2

    def _sink(self, i: int) -> None:
        while 2*i <= self.tail:
            j = 2*i
            if j < self.tail and self.heap[j] < self.heap[j+1]:
                j += 1
            if self.heap[i] >= self.heap[j]:
                break
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j
```

	Best case	**Worst case**	**Average case**
enqueue	1	$\log N$	$\log N$
dequeue	$\log N$	$\log N$	$\log N$

Heap Sort

Key idea:

- Create a max-heap ordered array with all N keys
- Repeatedly remove the maximum key remaining

```python
def sink(xs: list, i: int, tail: int) -> None:
    while 2*i <= tail:
        j = 2*i
        if j < tail and xs[j] < xs[j+1]:
            j += 1
        if xs[i] >= xs[j]:
            break
        xs[i], xs[j] = xs[j], xs[i]
        i = j


def heapSort(xs: list) -> None:
    k = len(xs)//2
    while k >= 1:
        sink(xs, k, len(xs)-1)
        k -= 1
    n = len(xs)-1
    while n > 1:
        xs[1], xs[n] = xs[n], xs[1]
        n -= 1
        sink(xs, 1, n)

#The index of the heap starts from 1, therefore a dummy element is added to the beginning of
    the list.
test = [None]+[5, 9, 54, 11, 3, 6, 2, -7, -3, -9, 4, 8, -6]
heapSort(test)
print(test)
```

Best case	**Worst case**	**Average case**
$N \lg N$	$N \lg N$	$N \lg N$

2.5 Summary

	In place	Stable	Worst	Average case	Best case
Selection	✓		N^2	N^2	N^2
Insertion	✓	✓	N^2	N^2	N
Merge-sort		✓	$N \lg N$	$N \lg N$	$N \lg N$
Quick-sort	✓		N^2	$N \lg N$	$N \lg N$
Heapsort	✓		$N \lg N$	$N \lg N$	$N \lg N$

- An in-place sorting algorithm is one that does not require additional memory to be allocated for temporary storage during the sorting process
- A sorting algorithm is said to be stable if two items with equal keys appear in the same order in the sorted output as they appear in the input array

3 Search Algorithms

3.1 Symbol Table

Symbol Table
Abstract data type to handle key-value pairs
`put(key, value)`
`get(key)`

Common assumptions

- Keys are unique
- Values are not null
- Keys have a total order relation:
  - Antisymmetry: if $a \leq b$ and $b \leq a$, then $a = b$
  - Transitivity: if $a \leq b$ and $b \leq c$, then $a \leq c$
  - Totality: either $a \leq b$ or $b \leq a$

3.2 Binary Search Tree

A binary search tree is a binary tree in symmetric order

- A binary tree is in symmetric order if each node has a key, and every node's key is:
  - Larger than all keys in its left subtree
  - Smaller than all keys in its right subtree

Pseudocode

```
class Node:
    def __init__(self, key, value) -> None:
        self.key = key
        self.value = value
        self.left = None
        self.right = None

    def get(self, key):
        if self.key == key:
```

```
            return self.value
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            return None

    def put(self, key, value):
        if key == self.key:
            self.value = value
        elif key < self.key:
            if self.left is None:
                self.left = Node(key, value)
            else:
                self.left.put(key, value)
        elif key > self.key:
            if self.right is None:
                self.right = Node(key,value)
            else:
                self.right.put(key,value)
```

```
from BSTNode import BSTNode

class BSTree:
    def __init__(self) -> None:
        self.root = None

    def get(self, key):
        if self.root is None:
            return None
        else:
            return self.root.get(key)

    def put(self, key, value):
        if self.root is None:
            self.root = BSTNode(key,value)
        else:
            self.root.put(key, value)
```

3.3 BST Additional Operations

Min and Max

```
class BSTNode:
    def min(self):
        if self.left is None:
            return self.key
        else:
            return self.left.min()

    def max(self):
        if self.right is None:
            return self.key
        else:
            return self.right.max()
```

```
class BSTree:
    def min(self):
        if self.root is None:
            return None
        else:
            return self.root.min()

    def max(self):
        if self.root is None:
            return None
        else:
            return self.root.max()
```

Floor and Ceiling

```python
class BSTNode:
    def floor(self, key):
        if key == self.key:
            return self.key
        elif key < self.key:
            if self.left is None:
                return None
            else:
                return self.left.floor(key)
        else:
            if self.right is None:
                return self.key
            else:
                right_floor = self.right.floor(key)
                return right_floor if right_floor is not None else self.key

    def ceiling(self, key):
        if key == self.key:
            return self.key
        elif key > self.key:
            if self.right is None:
                return None
            else:
                return self.right.ceiling(key)
        else:
            if self.left is None:
                return self.key
            else:
                left_ceiling = self.left.ceiling(key)
                return left_ceiling if left_ceiling is not None else self.key
```

```python
class BSTree:
    def floor(self, key):
        if self.root is None:
            return None
        else:
            return self.root.floor(key)

    def ceiling(self, key):
        if self.root is None:
            return None
        else:
            return self.root.ceiling(key)
```

Delete

```python
class BSTNode:
    def delete(self, key):
        if self is None:
            return None
        if key < self.key:
            self.left = self.left.delete(key)
        elif key > self.key:
            self.right = self.right.delete(key)
        else:
            if self.left is None:
                return self.right
            if self.right is None:
                return self.left
            temp = self
            self = min(self.right)
            self.right = temp.right._deleteMin()
            self.left = temp.left
        return self


    def _deleteMin(self):
        if self.left is None:
            return self.right
        self.left = self.left._deleteMin()
        return self
```

```python
class BSTree:
    def delete(self, key):
        if self.root is None:
```

```
            return None
        else:
            self.root = self.root.delete(key)
```

Worst Case			Average Case		
Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
N	N	N	$c \lg N$	$c \lg N$	$\sqrt{N}$

Get	**Put**	**Min/max**	**Floor/ceiling**
$h \sim \lg N$	$h \sim \lg N$	$h \sim \lg N$	$h \sim \lg N$
Delete	**Ordering iteration**		
$h \sim \lg N$	N		

3.4 Balanced Search Tree

Two Three Search Tree

A 2-3 search tree is a tree where each node can be of two types:

- 2-node: one key, two children
- 3-node: two keys, three children

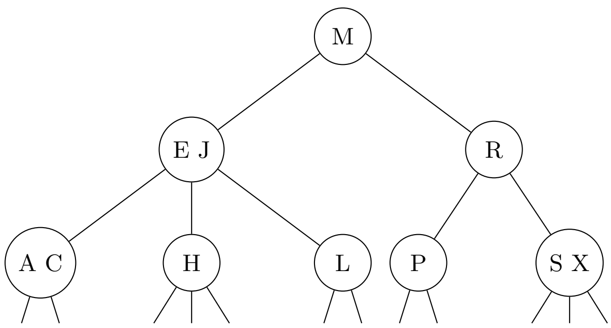
Feature

- Symmetric order. In-order traversal yields keys in ascending order
- Perfect balance. Every path from the root to a null link has the same length

Insert operation - put (new key)

- Case 1: insert the new key in a 2-node
  - Transform the 2-node into a 3-node
- Case 2: insert the new key in a 3-node
  - Add the new key to a 3-node to create a temporary 4-node
  - Move the middle key of the 4-node into its parent node and create two 2-nodes children
  - Repeat up the tree as necessary
  - If you reach the root and it is a 4-node, split it into three 2-nodes.

Graph Example



Worst Case			Average Case		
Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$

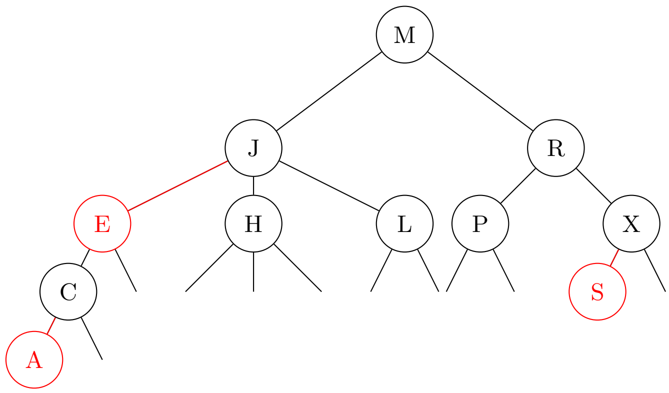
Red Black Tree

Key idea: represent a 2-3 Tree as a BST
How: transform a 3-node into a left-leaning BST (two 2-nodes connected by an "internal" red link)

Definition of a left-leaning red-black (LLRB) BST: a BST such that

- Every path from the root to null links has the same number of black links (perfect black balance)
- No node has 2 red links connected to it (or else that would be a 4-node)
- Red links lean left

Same Example



Pseudocode

```python
class LLRBNode:
    def __init__(self, key, value) -> None:
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self.color = False  # True for red and False for black

    def get(self, key):
        if self.key == key:
            return self.value
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            return None
```

```python
from LLRBNode import LLRBNode

class LLRBTre:
    def __init__(self) -> None:
        self.root = None

    def get(self, key):
        if self.root is None:
            return None
        else:
            return self.root.get(key)

    def put(self, key, value):
        if self.root is None:
            self.root = LLRBNode(key, value)
        else:
            self.root = self._put(self.root, key, value)

    def _put(self, n: LLRBNode, key, value) -> LLRBNode:
        if n is None:
            return LLRBNode(key, value)
        if key == n.key:
            n.value = value
```

```python
        elif key < n.key:
            n.left = self._put(n.left, key, value)
        else:
            n.right = self._put(n.right, key, value)

        if self._isRed(n) and not self._isRed(n.left):
            n = self._left_rotation(n)
        if self._isRed(n.left) and self._isRed(n.left.left):
            n = self._right_rotation(n)
        if self._isRed(n.left) and self._isRed(n.right):
            self._color_flip(n)
        return n

    def _isRed(self, n: LLRBNode) -> bool:
        if n is None:
            return False
        return n.color

    def _left_rotation(self, n: LLRBNode) -> LLRBNode:
        x = n.right
        n.right = x.left
        x.left = n
        x.color = n.color
        n.color = True
        return x

    def _right_rotation(self, n: LLRBNode) -> LLRBNode:
        x = n.left
        n.left = x.right
        x.right = n
        x.color = n.color
        n.color = True
        return x

    def _color_flip(self, n: LLRBNode):
        n.color = True
        n.left.color = False
        n.right.color = False
```

Worst Case			**Average Case**		
Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
2 lg N	2 lg N	2 lg N	1 lg N	1 lg N	1 lg N

3.5 Hash Tables

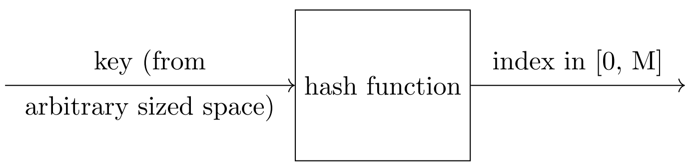
If our keys are simple, such as primitive data type, integer or strings, then we can use a hash table to store and retrieve data efficiently. Hash tables can only efficiently perform insertion and search operations, but cannot sort or perform sequential traversal operations.

Key idea: reduce the symbol table ADT into an array ADT
How: use a hash function to transform keys to integers used as indexes in an array where the corresponding values are

Hash Function

Properties: scramble keys to produce an index in [0,M] so that:

- The mapping is deterministic
- The mapping can be computed very fast
- The probability of collisions is very low



```python
def simpleHash(s: str, M: int) -> int:
    hash = 0
    P = 31
    for i in range(len(s)):
        hash = (P * hash + ord(s[i]))
    return hash % M
```

Separate Chaining

Data structure: array of $M(< N)$ linked lists
Operations

- Hash <key>: map key to integer i in $[0, M - 1]$
- Insert <key, value>: put value at the front of ith chain (if not already there)
- Search <key>: linearly scan the ith chain

How to choose M

- M too large $\rightarrow$ space waste (too many empty chains)
- M too small $\rightarrow$ search time blows up (chains too long)
- $M \approx N/5$

```python
class Node:
    def __init__(self, key, value) -> None:
        self.key = key
        self.value = value
        self.next = None
```

```python
from Node import Node

class SeparateChainingHashTable:
    def __init__(self, n: int):
        self.m = n // 5
        self.table = [None] * self.m

    def get(self, key):
        index = hash(key) % self.m
        current = self.table[index]
        while current is not None:
            if current.key == key:
                return current.value
            current = current.next
        return None

    def put(self, key, value):
        index = hash(key) % self.m
        current = self.table[index]
        if current is None:
            self.table[index] = Node(key, value)
            return
        while current is not None:
            if current.key == key:
                current.value = value
                return
            if current.next is None:
                current.next = Node(key, value)
                return
            current = current.next
```

Worst Case			Average Case		
Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
lg N	lg N	lg N	$3 - 5$	$3 - 5$	$3 - 5$

Linear Probing

Data structure: two arrays of size M > N (one for keys, one for values)
Operations

- Hash <key>: map key to integer i in $[0, M-1]$
- Insert <key, value>: put value at index i if available, or else try $i+1, i+2$, etc.
- Search <key>: access table index i; if occupied but no match, try $i+1, i+2$, etc.

How to choose M

- M too large $\rightarrow$ space waste (too many empty array entries)
- M too small $\rightarrow$ search time blows up
- $M \approx 2 \times N$

```python
class LinearProbingHashTable:
    def __init__(self, n: int) -> None:
        self.m = n * 2 + 1
        self.key_table = [None] * self.m
        self.value_table = [None] * self.m

    def get(self, key):
        index = hash(key) % self.m
        while self.key_table[index] is not None:
            if self.key_table[index] == key:
                return self.value_table[index]
            index = (index + 1) % self.m
        return None

    def put(self, key, value):
        index = hash(key) % self.m
        while self.key_table[index] is not None:
            if self.key_table[index] == key:
                self.value_table[index] = value
                return
            index = (index + 1) % self.m
        self.key_table[index] = key
        self.value_table[index] = value
```

Worst Case			Average Case		
Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
N	N	N	$3-5$	$3-5$	$3-5$

3.6 Summary

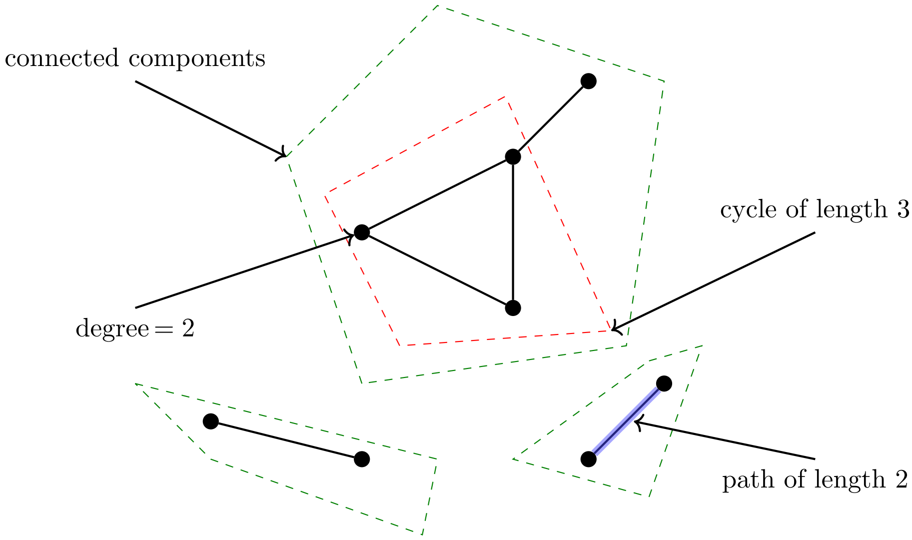
	Worst Case			Average Case		
	Search	**Insert**	**Delete**	**Search (hit)**	**Insert**	**Search**
BST	N	N	N	$c \lg N$	$c \lg N$	$\sqrt{N}$
2-3 Search Tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$
LLRB BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1 \lg N$	$1 \lg N$	$1 \lg N$
Separate Chaining	$\lg N$	$\lg N$	$\lg N$	$3-5$	$3-5$	$3-5$
Linear Probing	N	N	N	$3-5$	$3-5$	$3-5$

4 Graph Algorithms

4.1 Undirected Graphs

Terminology

- Degree. Number of edges that are incident to the vertex.
- Path. Sequence of vertices connected by edges.
- Cycle. Path whose first and last vertices are the same.
- Connected Component. Set of vertices connected to each other.




```python
class Graph:
    def __init__(self, num_vertices: int):
        self.num_vertices = num_vertices
        self.adj_list = [[] for _ in range(num_vertices)]

    def get_num_vertices(self) -> int:
        return self.num_vertices

    def add_edge(self, v: int, w: int) -> None:
        self.adj_list[v].append(w)
        self.adj_list[w].append(v)

    def get_adj_list(self, v: int) -> list:
        return self.adj_list[v]

```

Depth-First Search

Challenge: Find all vertices v connected to s (and their paths back to s)
How: To visit DFS a vertex v:

- Mark vertex v as visited
- Recursively visit all unmarked vertices w adjacent to v
- Return (retrace steps) when no unvisited options left

```python
class Stack:
    def __init__(self) -> None:
        self.stack = []

    def push(self, v) -> None:
        self.stack.append(v)

    def pop(self) -> object:
        if not self.is_empty():
            return self.stack.pop()
```

```
            return None

    def is_empty(self) -> bool:
        return len(self.stack) == 0

    def to_list(self) -> list:
        return self.stack
```

```
from Graph import Graph
from Stack import Stack

class DepthFirstPaths:
    def __init__(self, graph: Graph, start_vertex: int):
        self.visited = [False] * graph.get_num_vertices()
        self.edge_to = [-1] * graph.get_num_vertices()
        self.start_vertex = start_vertex
        self.dfs(graph, start_vertex)

    def dfs(self, graph: Graph, vertex: int):
        self.visited[vertex] = True
        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.edge_to[neighbor] = vertex
                self.dfs(graph, neighbor)

    def has_path_to(self, v: int) -> bool:
        return self.visited[v]

    def path_to(self, v: int) -> Stack:
        if not self.has_path_to(v):
            return None
        path = Stack()
        current = v
        while current is not self.start_vertex:
            path.push(current)
            current = self.edge_to[current]
        path.push(self.start_vertex)
        return path
```

Breadth-First Search

Challenge: Find all vertices v connected to a vertex s (and their distance back to s)
How:

- Put s onto a (FIFO) queue and mark s as visited
- Repeat until the queue is empty:
  - Dequeue vertex v from the front of the queue (i.e., remove the least recently added vertex v rom the queue)
  - Enqueue all of v's unvisited adjacent vertices to the queue and mark them as visited

```
class Queue:
    def __init__(self) -> None:
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self) -> object:
        if not self.is_empty():
            return self.queue.pop(0)
        return None

    def is_empty(self) -> bool:
        return len(self.queue) == 0
```

```
from Graph import Graph
from Queue import Queue
from Stack import Stack

class BreadthFirstPath:
    def __init__(self, graph: Graph, start_vertex: int) -> None:
```

```python
        self.dist_to_source = [-1] * graph.get_num_vertices()
        self.edge_to = [-1] * graph.get_num_vertices()
        self.start_vertex = start_vertex
        self.bfs(graph, start_vertex)

    def bfs(self, graph: Graph, vertex: int) -> None:
        queue = Queue()
        queue.enqueue(vertex)
        self.dist_to_source[vertex] = 0
        while not queue.is_empty():
            vertex = queue.dequeue()
            for neighbor in graph.get_adj_list(vertex):
                if self.dist_to_source[neighbor] == -1:
                    queue.enqueue(neighbor)
                    self.dist_to_source[neighbor] = self.dist_to_source[vertex] + 1
                    self.edge_to[neighbor] = vertex

    def has_path_to(self, v: int) -> bool:
        return self.dist_to_source[v] != -1

    def min_path_length_to(self, v: int) -> int:
        return self.dist_to_source[v]

    def shortest_path_to(self, v: int) -> Stack:
        if not self.has_path_to(v):
            return None
        path = Stack()
        current = v
        while current != self.start_vertex:
            path.push(current)
            current = self.edge_to[current]
        path.push(self.start_vertex)
        return path
```

Connected Components

Goal: Preprocess a graph G so to answer queries of the form “is v connected to w?” in constant time

```python
from Graph import Graph

class ConnectedComponents:
    def __init__(self, graph: Graph) -> None:
        self.visited = [False] * graph.get_num_vertices()
        self.cc = [-1] * graph.get_num_vertices()
        self.count = 0

        for vertex in range(graph.get_num_vertices()):
            if not self.visited[vertex]:
                self.dfs(graph, vertex)
                self.count += 1

    def dfs(self, graph: Graph, vertex: int) -> None:
        self.visited[vertex] = True
        self.cc[vertex] = self.count

        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.dfs(graph, neighbor)

    def get_count(self) -> int:
        return self.count

    def get_cc_id(self, vertex: int) -> int:
        return self.cc[vertex]

    def is_same_cc(self, vertex1, vertex2) -> bool:
        return self.cc[vertex1] == self.cc[vertex2]
```

Summary

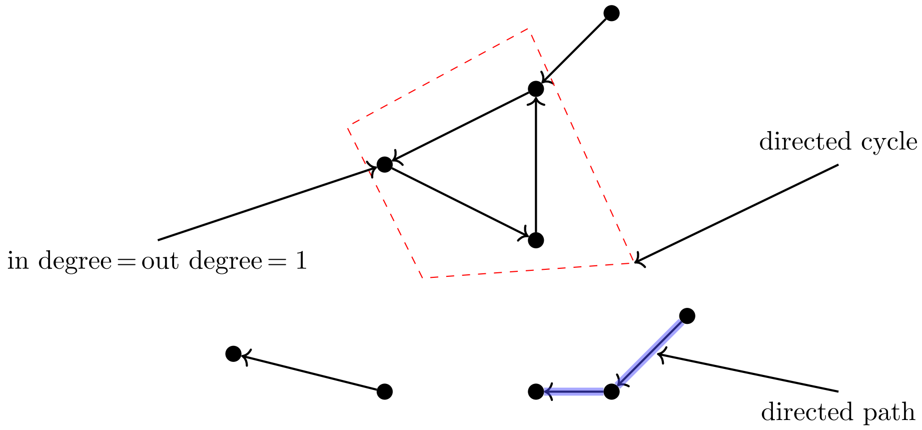
	Intuition	Programming Paradigm	Auxiliary Data Structure	Supported Graph Challenges
DFS	Maze exploration	Recursive algorithm	Use stack for unvisited vertices	Support efficient implementation of **Connected Components**
BST	Explore in increasing distance order	Iterative algorithm	Use queue for unvisited vertices	Automatically computes **Shortest Paths**

4.2 Directed Graphs

Digraph: Set of vertices connected pairwise by directed edges.

Terminology

- Directed Edge
- Directed path: Sequence of vertices connected by directed edges.
- Directed Cycle: Directed path whose first and last vertices are the same.
- In/out degree: Number of incoming / outgoing directed edges




```python
class Digraph:
    def __init__(self, num_vertices: int):
        self.num_vertices = num_vertices
        self.adj = [[] for v in range(num_vertices)]

    def add_edge(self, vertex1: int, vertex2: int) -> None:
        self.adj[vertex1].append(vertex2)

    def get_adj_list(self, vertex: int) -> list:
        return self.adj[vertex].copy()

    def get_num_vertices(self) -> int:
        return self.num_vertices
```

DFS and BFS

```python
from Digraph import Digraph

class DirectedDFS:
    def __init__(self, graph: Digraph, start_vertex: int) -> None:
        self.visited = [False] * graph.get_num_vertices()
        self.edge_to = [-1] * graph.get_num_vertices()
```

```python
        self.start_vertex = start_vertex
        self.dfs(graph, start_vertex)

    def dfs(self, graph: Digraph, vertex: int) -> None:
        self.visited[vertex] = True
        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.edge_to[neighbor] = vertex
                self.dfs(graph, neighbor)
```

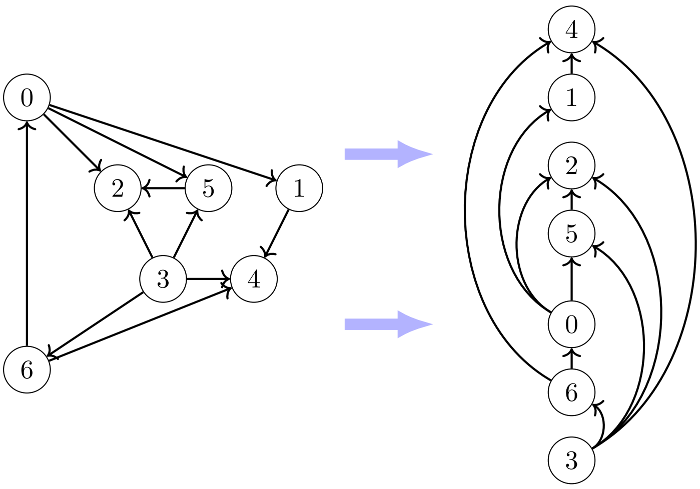
```python
from Digraph import Digraph
from Queue import Queue

class DirectedBFS:
    def __init__(self, graph: Digraph, start_vertex: int) -> None:
        self.dist_to_source = [-1] * graph.get_num_vertices()
        self.edge_to = [-1] * graph.get_num_vertices()
        self.start_vertex = start_vertex
        self.bfs(graph, start_vertex)

    def bfs(self, graph: Digraph, vertex: int) -> None:
        queue = Queue()
        queue.enqueue(vertex)
        self.dist_to_source[vertex] = 0
        while not queue.is_empty():
            vertex = queue.dequeue()
            for neighbor in graph.get_adj_list(vertex):
                if self.dist_to_source[neighbor] == -1:
                    queue.enqueue(neighbor)
                    self.dist_to_source[neighbor] = self.dist_to_source[vertex] + 1
                    self.edge_to[neighbor] = vertex
```

Topological Order

Precedence scheduling: Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?
How:

- Check the graph is a DAG
- Run depth-first search
- Return vertices in reverse post-order

Challenge: Given a directed acyclic graph (DAG), redraw it so that all edges point upwards.




```python
from Digraph import Digraph
from Stack import Stack

class TopologicalSort:
    def __init__(self, graph: Digraph):
        self.visited = [False] * graph.get_num_vertices()
        self.reverse_post = Stack()
```

```
        for vertex in range(graph.get_num_vertices()):
            if not self.visited[vertex]:
                self.dfs(graph, vertex)

    def dfs(self, graph: Digraph, vertex: int):
        self.visited[vertex] = True
        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.dfs(graph, neighbor)
        self.reverse_post.push(vertex)

    def get_topological_order(self) -> list:
        topo_order = []
        while not self.reverse_post.is_empty():
            topo_order.append(self.reverse_post.pop())
        return topo_order
```

Directed Cycle Detection

Challenge: Is a given digraph G a DAG?
How:

- Visit G using DFS
- Keep track of vertices whose recursive `dfs()` call has not completed yet
- If a call is made to a vertex with an open `dfs()` call, G is not a DAG

```
from Digraph import Digraph

class DirectedCycle:
    def __init__(self, graph: Digraph) -> None:
        self.visited = [False] * graph.get_num_vertices()
        self.on_call_stack = [False] * graph.get_num_vertices()
        self.cycle_detected = False
        for vertex in range(graph.get_num_vertices()):
            if not self.visited[vertex]:
                self.dfs(graph, vertex)

    def dfs(self, graph: Digraph, vertex: int) -> None:
        self.visited[vertex] = True
        self.on_call_stack[vertex] = True
        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.dfs(graph, neighbor)
            elif self.on_call_stack[neighbor]:
                self.cycle_detected = True
                return
            self.on_call_stack[vertex] = False

    def has_cycle(self) -> bool:
        return self.cycle_detected
```

Strongly-Connected Component

Definition: Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v
Definition: A strong component is a maximal subset of strongly-connected vertices. Strong connectivity is an equivalence relation

Kosaraju-Sharir Algorithm

Reverse graph: Strong components in G are same as in G^R.
Two phase approach:

- Compute topological order (i.e., reverse post-order) in G^R
- Run DFS in G, visiting unmarked vertices in reverse post-order of G^R

```python
class Digraph:
    def __init__(self, num_vertices: int):
        self.num_vertices = num_vertices
        self.adj = [[] for _ in range(num_vertices)]

    def add_edge(self, vertex1: int, vertex2: int) -> None:
        self.adj[vertex1].append(vertex2)

    def get_adj_list(self, vertex: int) -> list:
        return self.adj[vertex].copy()

    def get_num_vertices(self) -> int:
        return self.num_vertices

    def reverse(self) -> 'Digraph':
        new_adj = [[] for _ in range(self.num_vertices)]
        for vertex in range(self.num_vertices):
            for neighbor in self.get_adj_list(vertex):
                new_adj[neighbor].append(vertex)
        return Digraph(self.num_vertices)._from_adj_lists(new_adj)

    def _from_adj_lists(self, adj: list) -> 'Digraph':
        self.adj = adj
        return self
```

```python
from Digraph import Digraph
from TopologicalSort import TopologicalSort

class StronglyConnectedComponents:
    def __init__(self, graph: Digraph):
        self.visited = [False] * graph.get_num_vertices()
        self.scc = [-1] * graph.get_num_vertices()
        self.count = 0
        dfs_order = TopologicalSort(graph.reverse()).get_topological_order()
        for vertex in dfs_order:
            if not self.visited[vertex]:
                self.dfs(graph, vertex)
                self.count += 1

    def dfs(self, graph: Digraph, vertex: int):
        self.visited[vertex] = True
        self.scc[vertex] = self.count
        for neighbor in graph.get_adj_list(vertex):
            if not self.visited[neighbor]:
                self.dfs(graph, neighbor)

    def same_scc(self, vertex1: int, vertex2: int) -> bool:
        return self.scc[vertex1] == self.scc[vertex2]
```

4.3 Minimum Spanning Tree

Greedy Algorithm

A cut in a graph is a partition of its vertices into two (nonempty) sets.
A crossing edge connects a vertex in one set with a vertex in the other.
Property: given any cut, the crossing edge of min weight is in the MST
Key idea:

- Start with all edges colored grey
- Find a cut with no black-crossing edges
- Color its min-weight edge black
- Repeat until $V - 1$ edges are colored black

```python
class Edge:
    def __init__(self, vertex1: int, vertex2: int, weight: float) -> None:
        self.vertex1 = vertex1
        self.vertex2 = vertex2
        self.weight = weight

    def get_vertex1(self) -> int:
```

```
        return self.vertex1

    def get_vertex2(self) -> int:
        return self.vertex2

    def get_other_vertex(self, vertex: int) -> int:
        if vertex == self.vertex1:
            return self.vertex2
        if vertex == self.vertex2:
            return self.vertex1

    def __lt__(self, other: 'Edge') -> bool:
        return self.weight < other.weight

    def __gt__(self, other: 'Edge') -> bool:
        return self.weight > other.weight

    def __eq__(self, other: 'Edge') -> bool:
        return self.weight == other.weight

    def __le__(self, other: 'Edge') -> bool:
        return self.weight <= other.weight

    def __ge__(self, other: 'Edge') -> bool:
        return self.weight >= other.weight
```

```
from Edge import Edge

class EdgeWeightedGraph:
    def __init__(self, num_vertices: int) -> None:
        self.num_vertices = num_vertices
        self.adj = [[] for _ in range(num_vertices)]

    def add_edge(self, edge: Edge) -> None:
        vertex1 = edge.get_vertex1()
        vertex2 = edge.get_vertex2()
        self.adj[vertex1].append(edge)
        self.adj[vertex2].append(edge)

    def get_num_vertices(self) -> int:
        return self.num_vertices

    def get_adj_list(self, vertex: int) -> list:
        return self.adj[vertex]
```

Kruskal's algorithm

Key idea:

- Consider edges in ascending order of weight
- Add the next edge to the MST, unless doing so would create a cycle
- Stop when all the edges have been considered (or when $V - 1$ edges have been added)

```
class MinPriorityQueue:
    def __init__(self) -> None:
        self.heap = [None]
        self.tail = 0

    def enqueue(self, key) -> None:
        self.heap.append(key)
        self.tail += 1
        self._swim(self.tail)

    def dequeue(self):
        if self.tail == 0:
            return None
        min = self.heap[1]
        self.heap[1], self.heap[self.tail] = self.heap[self.tail], self.heap[1]
        self.heap[self.tail] = None
        self.tail -= 1
        self._sink(1)
        self.heap.pop()
        return min
```

```python

    def _swim(self, i: int) -> None:
        while i > 1 and self.heap[i//2] > self.heap[i]:
            self.heap[i], self.heap[i//2] = self.heap[i//2], self.heap[i]
            i //= 2

    def _sink(self, i: int) -> None:
        while 2*i <= self.tail:
            j = 2*i
            if j < self.tail and self.heap[j] > self.heap[j+1]:
                j += 1
            if self.heap[i] <= self.heap[j]:
                break
            self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
            i = j

    def is_empty(self) -> bool:
        return len(self.heap) == 1
```

```python
from Edge import Edge

class UnionFind:
```

```python
from EdgeWeightedGraph import EdgeWeightedGraph
from Queue import Queue
from MinPriorityQueue import MinPriorityQueue
from UnionFind import UnionFind
from Edge import Edge

class KruskalMST:
    def __init__(self, graph: EdgeWeightedGraph) -> None:
        self.mst = Queue()
        self.pq = MinPriorityQueue()
        for edge in graph.get_adj_list():
            self.pq.enqueue(edge)

        uf = UnionFind(graph.get_num_vertices())
        while not self.pq.is_empty() and self.mst.size() < graph.num_vertices-1:
            edge = self.pq.dequeue()
            vertex1 = edge.get_vertex1()
            vertex2 = edge.get_vertex2()
            if not uf.connected():
                uf.union(vertex1, vertex2)
                self.mst.enqueue(edge)

    def edges(self):
        return self.mst
```

Worst Case	**Build the priority queue**	**Delete min**	**Union**	**Find (connected)**
Frequency	1	E	V	E
Cost per operation	$E \log E$	$\log E$	$\log V$	$\log V$

Prim's Algorithm

Key idea:

- Start with vertex 0 and greedily grow tree T
- Add to T the min-weighted edge with exactly one endpoint in T
- Stop when $V-1$ edges have been added

```python
from EdgeWeightedGraph import EdgeWeightedGraph
from Queue import Queue
from MinPriorityQueue import MinPriorityQueue
from Edge import Edge

```

```python
class LazyPrimMST:
    def __init__(self, graph: EdgeWeightedGraph) -> None:
        self.visited = [False] * graph.get_num_vertices()
        self.mst = Queue()
        self.pq = MinPriorityQueue()
        self.visit(graph, 0)

        while not self.pq.is_empty() and self.mst.size() < graph.get_num_vertices() - 1:
            edge = self.pq.dequeue()
            vertex1 = edge.get_vertex1()
            vertex2 = edge.get_vertex2()
            if self.visited[vertex1] and self.visited[vertex2]:
                continue
            self.mst.enqueue(edge)
            if not self.visited[vertex1]:
                self.visit(graph, vertex1)
            if not self.visited[vertex2]:
                self.visit(graph, vertex2)

    def visit(self, graph: EdgeWeightedGraph, vertex: int):
        self.visited[vertex] = True
        for edge in graph.get_adj_list(vertex):
            if not self.visited[edge.get_other_vertex(vertex)]:
                self.pq.enqueue(edge)

    def get_mst_edges(self) -> Queue:
        return self.mst
```

Worst Case	**Delete min**	**Insert**
Frequency	E	E
Cost per operation	$\log E$	$\log E$

4.4 Shortest Path

Problem Formulation

Shortest path from one vertex s to any other in G

- Simplifying assumption: shortest paths from s to each vertex v in G exist

Generic algorithm to compute SPT from s:

1. Initialise `distTo[s]=0`
2. Initialise `distTo[v]=`∞ for all other vertices
3. Repeat `relax(e)` for any edge $e : v \rightarrow w$, until there are no more edges e for which `distTo[v] + e.weight() < distTo[w]`

```python
class WeightedEdge:
    def __init__(self, start_vertex: int, end_vertex: int, weight: float) -> None:
        self.start_vertex = start_vertex
        self.end_vertex = end_vertex
        self.weight = weight

    def get_start_vertex(self) -> int:
        return self.start_vertex

    def get_end_vertex(self) -> int:
        return self.end_vertex

    def get_weight(self) -> float:
        return self.weight

    def __lt__(self, other: 'WeightedEdge') -> bool:
        return self.weight < other.weight

    def __gt__(self, other: 'WeightedEdge') -> bool:
        return self.weight > other.weight
```

```python
    def __eq__(self, other: 'WeightedEdge') -> bool:
        return self.weight == other.weight

    def __le__(self, other: 'WeightedEdge') -> bool:
        return self.weight <= other.weight

    def __ge__(self, other: 'WeightedEdge') -> bool:
        return self.weight >= other.weight
```

```python
from WeightedEdge import WeightedEdge

class EdgeWeightedDigraph:
    def __init__(self, num_vertices: int) -> None:
        self.num_vertices = num_vertices
        self.adj = [[] for _ in range(num_vertices)]

    def add_edge(self, edge: WeightedEdge) -> None:
        vertex = edge.get_start_vertex()
        self.adj[vertex].append(edge)

    def get_num_vertices(self) -> int:
        return self.num_vertices

    def get_adj_list(self, vertex: int) -> list:
        return self.adj[vertex]
```

Dijkstra's Algorithm

Assumption: Digraph has non-negative weights
Key idea:

- Consider vertices in increasing order of distance from s
- Add vertex to the SPT and relax all its outgoing edges

```python
    class MinPriorityQueue:
    def __init__(self) -> None:
        self.key = [None]
        self.value = [None]
        self.tail = 0

    def enqueue(self, key, value) -> None:
        self.key.append(key)
        self.value.append(value)
        self.tail += 1
        self._swim(self.tail)

    def dequeue(self) -> object:
        if self.tail == 0:
            return None
        minimum = self.value[1]
        self.key[1], self.key[self.tail] = self.key[self.tail], self.key[1]
        self.value[1], self.value[self.tail] = self.value[self.tail], self.value[1]
        self.key[self.tail] = None
        self.value[self.tail] = None
        self.tail -= 1
        self._sink(1)
        self.key.pop()
        self.value.pop()
        return minimum

    def _swim(self, i: int) -> None:
        while i > 1 and self.key[i//2] > self.key[i]:
            self.key[i], self.key[i//2] = self.key[i//2], self.key[i]
            self.value[i], self.value[i//2] = self.value[i//2], self.value[i]
            i //= 2

    def _sink(self, i: int) -> None:
        while 2*i <= self.tail:
            j = 2*i
            if j < self.tail and self.key[j] > self.key[j+1]:
                j += 1
            if self.key[i] <= self.key[j]:
                break
```

```python
            self.key[i], self.key[j] = self.key[j], self.key[i]
            self.value[i], self.value[j] = self.value[j], self.value[i]
            i = j

    def is_empty(self) -> bool:
        return len(self.key) == 1

    def contains(self, value: int) -> bool:
        return value in self.value

    def decrease_key(self, key, value: int) -> None:
        for i in range(len(self.value)):
            if self.value[i] == value:
                self.key[i] = key
                self._swim(i)
                break
```

```python
from EdgeWeightedDigraph import EdgeWeightedDigraph
from WeightedEdge import WeightedEdge
from Stack import Stack
from MinPriorityQueue import MinPriorityQueue
import math

class DijkstraSP:
    def __init__(self, graph: EdgeWeightedDigraph, start_vertex: int) -> None:
        self.dist_to = [math.inf] * graph.get_num_vertices()
        self.edge_to = [None] * graph.get_num_vertices()
        self.dist_to[start_vertex] = 0

        self.pq = MinPriorityQueue()
        self.pq.enqueue(0, start_vertex)

        while not self.pq.is_empty():
            vertex = self.pq.dequeue()
            for edge in graph.get_adj_list(vertex):
                self.relax(edge)

    def get_dist_to(self, vertex: int) -> float:
        return self.dist_to[vertex]

    def get_path_to(self, vertex) -> list:
        path = Stack()
        edge = self.edge_to[vertex]
        while edge is not None:
            path.push(edge)
            edge = self.edge_to[edge.get_start_vertex()]
        return path

    def relax(self, edge: WeightedEdge):
        start_vertex = edge.get_start_vertex()
        end_vertex = edge.get_end_vertex()
        if self.dist_to[start_vertex] + edge.get_weight() < self.dist_to[end_vertex]:
            self.dist_to[end_vertex] = self.dist_to[start_vertex] + \
                edge.get_weight()
            self.edge_to[end_vertex] = edge
            if self.pq.contains(end_vertex):
                self.pq.decrease_key(self.dist_to[end_vertex], end_vertex)
            else:
                self.pq.enqueue(self.dist_to[end_vertex], end_vertex)
```

PQ implementation	**insert()**	**delMin()**	**decreaseKey()**	**total**
Binary heap	$\log V$	$\log V$	$\log V$	$E \log V$

Edge-Weighted DAG Algorithm

Assumption: Digraph is acyclic (DAG)
Key idea:

- Consider vertices in topological order
- Add vertex to the SPT and relax all its outgoing edges

```python
from WeightedEdge import WeightedEdge
from EdgeWeightedDigraph import EdgeWeightedDigraph
from Stack import Stack

class TopologicalSort:
    def __init__(self, graph: EdgeWeightedDigraph):
        self.visited = [False] * graph.get_num_vertices()
        self.reverse_post = Stack()
        for vertex in range(graph.get_num_vertices()):
            if not self.visited[vertex]:
                self.dfs(graph, vertex)

    def dfs(self, graph: EdgeWeightedDigraph, vertex: int):
        self.visited[vertex] = True
        for edge in graph.get_adj_list(vertex):
            if not self.visited[edge.get_end_vertex()]:
                self.dfs(graph, edge.get_end_vertex())
        self.reverse_post.push(vertex)

    def get_topological_order(self) -> list:
        topo_order = []
        while not self.reverse_post.is_empty():
            topo_order.append(self.reverse_post.pop())
        return topo_order
```

```python
from EdgeWeightedDigraph import EdgeWeightedDigraph
from WeightedEdge import WeightedEdge
from TopologicalSort import TopologicalSort
from Stack import Stack
import math

class AcyclicSP:
    def __init__(self, graph: EdgeWeightedDigraph, start_vertex: int) -> None:
        self.dist_to = [math.inf] * graph.get_num_vertices()
        self.edge_to = [None] * graph.get_num_vertices()
        self.dist_to[start_vertex] = 0

        topological_order = TopologicalSort(graph).get_topological_order()

        while len(topological_order) > 0:
            vertex = topological_order.pop(0)
            for edge in graph.get_adj_list(vertex):
                self.relax(edge)

    def get_dist_to(self, vertex: int) -> float:
        return self.dist_to[vertex]

    def get_path_to(self, vertex) -> list:
        path = Stack()
        edge = self.edge_to[vertex]
        while edge is not None:
            path.push(edge)
            edge = self.edge_to[edge.get_start_vertex()]
        return path

    def relax(self, edge: WeightedEdge):
        start_vertex = edge.get_start_vertex()
        end_vertex = edge.get_end_vertex()
        if self.dist_to[start_vertex] + edge.get_weight() < self.dist_to[end_vertex]:
            self.dist_to[end_vertex] = self.dist_to[start_vertex] + \
                edge.get_weight()
            self.edge_to[end_vertex] = edge
```

Cost: Dominated by computation of topological sort on DAG $(E + V)$

Bellman-Ford Algorithm

Assumption: Digraph has non-negative cycles
Key idea

- Repeat V times: relax all edges

```python
from EdgeWeightedDigraph import EdgeWeightedDigraph
from WeightedEdge import WeightedEdge
from Stack import Stack
```

```python
import math

class BellmanFordSP:
    def __init__(self, graph: EdgeWeightedDigraph, start_vertex: int) -> None:
        self.dist_to = [math.inf] * graph.get_num_vertices()
        self.edge_to = [None] * graph.get_num_vertices()
        self.dist_to[start_vertex] = 0

        for _ in range(graph.get_num_vertices()):
            for vertex in range(graph.get_num_vertices()):
                for edge in graph.get_adj_list(vertex):
                    self.relax(edge)

    def get_dist_to(self, vertex: int) -> float:
        return self.dist_to[vertex]

    def get_path_to(self, vertex) -> list:
        path = Stack()
        edge = self.edge_to[vertex]
        while edge is not None:
            path.push(edge)
            edge = self.edge_to[edge.get_start_vertex()]
        return path

    def relax(self, edge: WeightedEdge):
        start_vertex = edge.get_start_vertex()
        end_vertex = edge.get_end_vertex()
        if self.dist_to[start_vertex] + edge.get_weight() < self.dist_to[end_vertex]:
            self.dist_to[end_vertex] = self.dist_to[start_vertex] + \
                edge.get_weight()
            self.edge_to[end_vertex] = edge
```

Practical improvement:

- If `distTo[v]` does not change during pass i, no need to relax any outgoing edge from v in pass $i + 1$
- Maintain a queue of vertices whose `distTo[]` changed

Cost: Proportional to $E \times V$, but only in worst case

Single-Source Shortest Path Summary

Algorithm	Restriction	Typical case	Worst case	Extra space
Dijkstra (binary heap)	No negative weights	$E \log V$	$E \log V$	V
Edge-weighted DAG (topological sort)	No cycles	$E + V$	$E + V$	V
Bellman-Ford	No negative cycles	EV	EV	V
Bellman-Ford (queue based)	No negative cycles	$E + V$	EV	V

5 String Algorithms

String: Sequence of characters.
Character: Digit from a fixed alphabet of size R (radix).
String operations:

- Length: Number of characters.
- Indexing: Get the ith character.
- Substring extraction: Get a contiguous subsequence of characters.
- String concatenation: Append a string to the end of another string.

5.1 String Sorting Algorithms

Key-Index Counting

Assumption: Keys are integers in $[0, R-1]$. (One character long)
Goal: Sort an array `a[]` of N integers with values in $[0, R-1]$.
Steps:

1. Count frequencies of each value in R using key as index
2. Compute frequency cumulates (to specify destinations)
3. Access cumulates using key as index and move items
4. Copy back into original array

```python
def key_indexed_counting(char_list: list, radix: int):
    N = len(char_list)
    count = [0] * (radix + 1)
    aux = [''] * N

    for i in range(N):
        count[ord(char_list[i]) + 1] += 1

    for r in range(radix):
        count[r + 1] += count[r]

    for i in range(N):
        aux[count[ord(char_list[i])]] = char_list[i]
        count[ord(char_list[i])] += 1

    for i in range(N):
        char_list[i] = aux[i]
```

	In place	Stable	Time	Space
Key-Index Counting		✓	$\sim N + R$	$\sim N + R$

LSD Radix Sort

Least-significant-digit first string (radix) sort. (means from right to left)

Assumption: Keys all have the same (small) length D Basic Idea

- Consider each character in turn, from right to left
- At each pass, use key-indexed counting on dth character to sort (stably)

```python
def LSDsort(str_list: list, radix: int):
    N = len(str_list)
    W = len(str_list[0])

    for d in range(W-1, -1, -1):
        count = [0] * (radix + 1)
        aux = [''] * N

        for i in range(N):
            count[ord(str_list[i][d]) + 1] += 1

        for r in range(radix):
            count[r + 1] += count[r]

        for i in range(N):
            aux[count[ord(str_list[i][d])]] = str_list[i]
            count[ord(str_list[i][d])] += 1

        for i in range(N):
            str_list[i] = aux[i]
```

	In place	Stable	Time	Space
LSD Radix Sort		✓	$\sim W \times (N+R)$	$\sim (N+R)$

MSD Radix Sort

Most-significant-digit first string (radix) sort (left to right)

Basic Idea:

- Partition input into R pieces according to first character (use key-indexed counting to sort)
- Recursively sort all strings that start with the same character

```
def MSDsort(str_list: list, aux, lo: int, hi: int, d: int, radix: int):
    if hi <= lo:
        return
    count = [0] * (radix + 2)

    for i in range(lo, hi):
        count[ord(str_list[i][d]) + 2] += 1

    for r in range(radix+1):
        count[r+1] += count[r]

    for i in range(lo, hi):
        aux[count[ord(str_list[i][d]) + 1]] = str_list[i]
        count[ord(str_list[i][d]) + 1] += 1

    for i in range(lo, hi):
        str_list[i] = aux[i - lo]

    for r in range(radix):
        MSDsort(str_list, aux, lo + count[r], lo + count[r+1] - 1, d+1, radix)
```

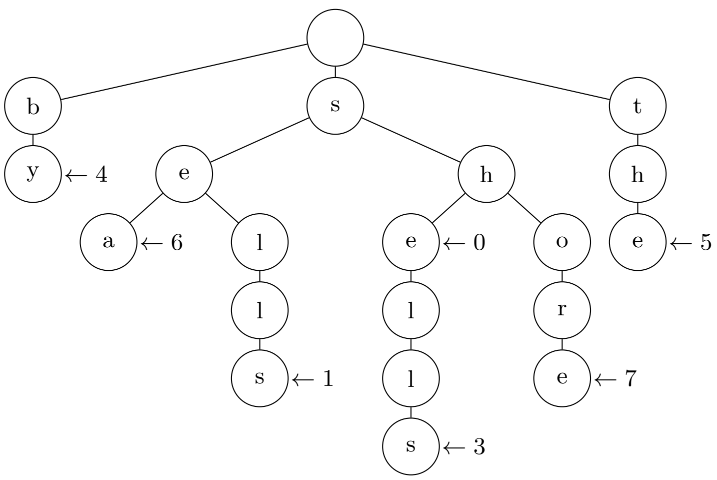
	In place	Stable	Worst case time	Average case time	Space
LSD Radix Sort		✓	$\sim W \times N$	$\sim N \log_R N$	$\sim N + D \times R$

5.2 String Search

R-way Tries

Key idea: tree representation of symbol table

- Each node represents (i.e., stores) one character, not full keys
- Each node has exactly R children, one for each possible character



```python
class Node:
    def __init__(self, radix: int) -> None:
        self.value = None
        self.children = [None] * radix

    def set_children(self, i: int, node: 'Node') -> None:
        self.children[i] = node

    def get_children(self, i: int) -> 'Node':
        return self.children[i]

    def get_value(self) -> object:
        return self.value

    def set_value(self, value) -> None:
        self.value = value
```

```python
from Node import Node

class TrieST:
    def __init__(self, radix: int) -> None:
        self.radix = radix
        self.root = Node(radix)

    def put(self, key: str, value) -> None:
        self.root = self._put(self.root, key, value, 0)

    def _put(self, node: Node, key: str, value, d: int) -> Node:
        if node is None:
            node = Node(self.radix)
        if d == len(key):
            node.set_value(value)
            return node

        c = ord(key[d])
        node.set_children(c, self._put(node.get_children(c), key, value, d+1))
        return node

    def get(self, key: str) -> object:
        node = self._get(self.root, key, 0)
        if node is None:
            return None
        return node.get_value()

    def _get(self, node: Node, key: str, d: int) -> Node:
        if node is None:
            return None
        if d == len(key):
            return node

        c = ord(key[d])
        return self._get(node.get_children(c), key, d+1)
```

Search (hit)	**Search (miss)**	**Space**
L	sub L	$N \times R$

3-way Tries

Key idea

- Each node represents (i.e., stores) one character, not full keys
- Each node has exactly 3 children: smaller (left), equal (middle), larger (right)

```python
class Node:
    def __init__(self):
        self.value = None
        self.char = ''
        self.left = None
        self.mid = None
        self.right = None
```

```python
from Node import Node

class TernarySearchTrieST:
    def __init__(self):
        self.root = Node()

    def put(self, key, value):
        self.root = self._put(self.root, key, value, 0)

    def _put(self, node, key, value, d):
        if not node:
            node = Node()
            node.char = key[d]

        c = key[d]
        if c < node.char:
            node.left = self._put(node.left, key, value, d)
        elif c > node.char:
            node.right = self._put(node.right, key, value, d)
        elif d < len(key) - 1:
            node.mid = self._put(node.mid, key, value, d + 1)
        else:
            node.value = value
        return node

    def get(self, key):
        node = self._get(self.root, key, 0)
        if node is None:
            return None
        return node.value

    def _get(self, node, key, d):
        if node is None:
            return None
        if d == len(key) - 1:
            return node

        c = key[d]
        if c < node.char:
            return self._get(node.left, key, d)
        elif c > node.char:
            return self._get(node.right, key, d)
        else:
            return self._get(node.mid, key, d + 1)
```

Time	**Space**
As fast as hashing, faster on search miss	$4N$

Extended API

`keys()`: iterate through all keys in sorted order
How to:

- Do an in-order traversal of the trie
- Keep track of matched characters on the path from root to current node
- Add matched keys to a queue

```python
from Queue import Queue

class TrieST:
    def keys(self) -> Queue:
        queue = Queue()
        self.__collect(self.root, '', queue)
        return queue

    def __collect(self, node: Node, prefix: str, queue: Queue) -> None:
        if node is None:
            return
        if node.get_value() is not None:
            queue.enqueue(prefix)
```

```python
        for c in range(self.radix):
            self.__collect(node.get_children(c), prefix + chr(c), queue)
```

`keysWithPrefix(s)`: find all keys in the symbol table starting with `s`
How to:

- Do an in-order traversal of the trie, starting at the node x matching `s`
- Keep track of matched characters on the path from x to current node
- Add matched keys to a queue

```python
class TrieST:
    def keys_with_prefix(self, s: str) -> Queue:
        queue = Queue()
        node = self._get(self.root, s, 0)
        self.__collect(node, s, queue)
        return queue
```

`longestPrefixOf(s)`: find the longest key in the symbol table that is prefix of `s` How to:

- Search for query string `s`
- Keep track of longest key encountered

```python
class TrieST:
    def longest_prefix_of(self, prefix: str):
        length = self._search(self.root, prefix, 0, 0)
        return prefix[:length]

    def _search(self, node: Node, query: str, d: int, length: int):
        if node is None:
            return length
        if node.get_value() is not None:
            length = d
        if d == len(query):
            return length
        c = ord(query[d])
        return self._search(node.get_children(c), query, d+1, length)
```

5.3 Substring Search

Goal: Find a pattern of length M in a text of length N (usually $M << N$).

Brute Force Algorithm

Key idea: Use two indices i and j to scan the text and the pattern respectively. Compare the jth character in the pattern with the $(i + j)$th character in the text

- In case of character match, move the index j in the pattern of 1
- In case of character mismatch, move the search index i in the text of 1 and restart the index j in the pattern from the beginning

```python
def substring_search(pattern: str, text: str) -> int:
    pattern_len = len(pattern)
    text_len = len(text)

    for i in range(text_len - pattern_len + 1):
        j = 0
        while j < pattern_len:
            if text[i+j] != pattern[j]:
                break
            j += 1
        if j == pattern_len:
            return i
    return -1
```

Cost (worst case)$\sim N \times M$

Knuth-Morris-Pratt algorithm

Key idea: Use a DFA as a string-searching machine.

- Start in the initial state
- Read one character at a time from the input stream
- Consult the DFA transition table to know what state to go next (there is exactly one transition for each character in the alphabet)
- Pattern found if transition leads to final state

How to build the DFA from pattern

1. Init. Create one state per character in the pattern, plus a final state
2. Match transitions. If in state j and next char `c == pattern[j]`, go to state $j + 1$
3. Mismatch transitions.
   - If in state j and next char `c != pattern[j]`, then the last $j-1$ characters in input are `pattern[1..j-1]`, followed by `c`.
   - To fill in `DFA[c][j]`, simulate having `pattern[1..j-1]` in input to the DFA, then take transition `c`.

```python
class KnuthMorrisPrattSS:
    def __init__(self, pattern: str, radix: int) -> None:
        self.pattern_length = len(pattern)
        self.radix = radix
        self.DFA = [[0] * radix for _ in range(self.pattern_length)]
        self.build_DFA(pattern)

    def build_DFA(self, pattern: str) -> None:
        self.DFA[0][ord(pattern[0])] = 1
        X = 0
        for j in range(1, self.pattern_length):
            for c in range(self.radix):
                self.DFA[j][c] = self.DFA[X][c]
            self.DFA[j][ord(pattern[j])] = j+1
            X = self.DFA[X][ord(pattern[j])]

    def substring_search(self, text: str) -> int:
        text_length = len(text)
        i, j = 0, 0
        while i < text_length and j < self.pattern_length:
            j = self.DFA[j][ord(text[i])]
            i += 1
        if j == self.pattern_length:
            return i - self.pattern_length
        else:
            return -1

def substring_search(pattern: str, text: str) -> int:
    pattern_len = len(pattern)
    text_len = len(text)

    for i in range(text_len - pattern_len + 1):
        j = 0
        while j < pattern_len:
            if text[i+j] != pattern[j]:
                break
            j += 1
        if j == pattern_len:
            return i
    return -1
```

Construction of DFA	**Substring search**
$\sim R \times M$	$\sim M + N$

Boyer-Moore Algorithm

Key idea

- Scan pattern from right to left
- Upon character mismatch, skip as many as M text characters

How much to skip?

- Case 1: mismatched character not present in pattern. Move i beyond the mismatched position.
- Case 2: mismatched character in pattern. Precompute index of rightmost occurrence of mismatched character in pattern. Skip i forward of (j - rightmost index)

```python
class BoyerMooreSS:
    def __init__(self, pattern: str, radix: int) -> None:
        self.pattern = pattern
        self.pattern_len = len(pattern)
        self.radix = radix
        self.index = [-1] * self.radix
        for i, c in enumerate(pattern):
            self.index[ord(c)] = i

    def substring_search(self, text: str) -> int:
        text_len = len(text)
        i = 0
        while i <= text_len - self.pattern_len:
            skip = 0
            j = self.pattern_len - 1
            while j >= 0:
                if self.pattern[j] != text[i + j]:
                    skip = max(1, j - self.index[ord(text[i + j])])
                    break
                else:
                    j -= 1

            i += skip
            if skip == 0:
                return i

        return -1
```

Pre-processing	**Substring matching (Average)**	**Substring matching (Worst)**
$\sim M$	$\sim N/M$	$\sim M \times N$

Summary

Brute force algorithm	**Knuth-Morris-Pratt algorithm**	**Boyer-Moore algorithm**
$\sim N \times M$	$\sim N$	$\sim N/M$ (but worst case $N \times M$)

5.4 Compression

Lossless Compression
Goal:

- Message. Given in input binary data (bit-stream) B.
- Compress. Generate a “compressed” representation $C(B)$.
- Expand. Reconstruct the original bitstream B without loss.

Run-length Coding

Key idea. Use counts to represent sequences of 0/1 bits
Representation. Use n-bit counts (e.g. $n = 8$) to represent alternating runs of 0s and 1s.

Huffman Compression

Key idea. Instead of encoding every character in alphabet using the same number of bits, use fewer bits for characters that appear more often, so to lower the total number of bits used.

Issue: ambiguity
Solution: Generate prefix-free code

How to: compression

- Trie construction
- Trie transmission
- Compression based on leaves-to-root trie traversal

How to: prefix-free codes represented as binary trie

- Chars in leaves
- Codeword is bit sequence path from root to leaf

How to: expansion

- Start at the root of the trie and read one bit from the input stream at a time
- Go left if bit is 0, go right if 1
- Once on a leaf, emit corresponding key and restart the root