# Data Structures & Algorithms

Ryan Hou

2025-02-17

# Table of contents

# Preface

Notes on Data Structures & Algorithms, illustrated in C++.

# Resources

Some relevant resources:

- EECS 280 - Programming and Intro Data Structures (University of Michigan)
- EECS 281 - Data Structures and Algorithms (University of Michigan)
- EECS 376 - Foundations of Computer Science
- Data Structures & Algorithms - Google Tech Dev Guide
- EECS 281 References
- The Algorithms

Practice Resources:

- LeetCode
- NeetCode
- Blind 75
- Grind 75
  - About Grind 75
- Codeforces

Interview Resources:

- Leetcode Patterns
- Learning Resources - Reddit-wiki-programming
- AlgoMonster
- Tech Interview Handbook
  - Study Cheatsheet
- Zero To Mastery

C++ Guides:

- learncpp.com

Books:

- Algorithms - Jeff Erickson
- Cracking the Coding Interview

# 1 Introduction

## 1.1 Perspective

> Note 1: Definition - Definition goes here
>
> **Definition** is defined by: definition.

This notebook contains programming basics, data structures, and algorithms. The language of choice is C++, and concepts from C++ STL are also covered.

## 1.2 Data Structures

## 1.3 Algorithm Families

- Brute-Force
- Greedy
- Divide and Conquer
- Dynamic Programming
- Backtracking
- Branch and Bound

# 2 Foundations

## 2.1 Resources

# 3 Pointers

# 4 Arrays

# 5 Strings

# 6 Streams and IO

# 7 Abstract Data Type

> **Note 2: Definition - Abstract Data Type**
>
> An **Abstract Data Type (ADT)** combines data with valid operations and their behaviors on stored data

- e.g. insert, delete, access
- ADTs define an **interface**

Meanwhile, a **data structure** provides a concrete implementation of an ADT.

# 8 Object-Oriented Programming

# 9 Dynamic Memory

# 10 Linked List

Linked list is a **non-contiguous** data structure with efficient **insertion** and **deletion**. Often used to implement other data structures like stacks, queues, or deques.

Main types of linked lists include:

- Singly linked list
- Doubly linked list
- Circular linked list

## 10.1 Summary

| | **Array** | **Linked List** |
|---|---|---|
| **Access** | **best** time complexity **O(1)**<br>**average** time complexity **O(1)**<br>**worst** time complexity **O(1)** | **best** time complexity **O(1)**<br>**average** time complexity **O(n)**<br>**worst** time complexity **O(n)** |
| **Insertion** | **best** time complexity **O(1)**<br>**average** time complexity **O(n)**<br>**worst** time complexity **O(n)** | **best** time complexity **O(1)**<br>**average** time complexity **O(n)**<br>**worst** time complexity **O(n)** |
| **Memory Overhead** | If array size is chosen well, the array uses less memory | Additional memory is required for pointers |
| **Memory Efficiency** | May contain unused memory | Can change dynamically in size resulting in no wasted memory |

## 10.2 C++ Example

Structure of a singly linked list node:

```
class IntList {
private:

    struct Node {
```

```
        int datum; // contains the element of the node
        Node *next; // points to the next node in the list
    }

public:
    Node *first;
    Node *last;
};
```

We can set the next pointer of the last node to `nullptr` as a sentinel value.

A more complete implementation:

```
template <typename T>
class LinkedList {
private:
    struct Node {
        T datum; // contains the element of the node
        Node *next; // points to the next node in the list
    }

    Node *first;
    Node *last;

public:
    // Constructor builds an empty list
    LinkedList() : first(nullptr) {}

    bool isEmpty() const {
        return first == nullptr;
    }

    // Return by reference the first element;
    T & front() {
        assert(!empty());
        return first->datum;
    }

    // Push a new node to the front
    void push_front(T datum) {
        Node *p = new Node;
        p->datum = datum;
```

```
        p->next = first;
        first = p;
    }

    void push_back(T datum) {
        Node *p = new Node;
        p->datum = datum;
        p->next = nullptr;
        if (empty()) {
            first = last = p;
        } else {
            last->next = p;
            last = p;
        }
    }

    // Pop the front node
    void pop_front() {
        assert(!empty());
        Node *victim = first;
        first = first->next;
        delete victim;
    }

    // Printing the linked list to os
    void print(ostream &os) const {
        for (Node *np = first; np; np = np->next) {
            os << np->datum << " ";
        }
    }

    void pop_all() {
        while (!empty()) {
            pop_front();
        }
    }

    void push_all(const LinkedList &other) {
        for (Node *np = other.first; np; np = np->next) {
            push_back(np->datum);
        }
    }
```

```
    // - The Big Three for LinkedList -

    // Destructor
    ~LinkedList() {
        pop_all();
    }

    // Copy constructor
    LinkedList(const LinkedList &other) : first(nullptr), last(nullptr) {
        push_all(other);
    }

    // Assignment Operator
    LinkedList & operator=(const LinkedList &rhs) {
        if (this == &rhs) { return *this; }
        pop_all();
        push_all(rhs);
        return *this;
    }

};
```

# 11 Iterators

## 11.1 References

- Geeks for Geeks Iterators in STL

## 11.2 Iterators in C++ STL

### Iterator Declaration

```
<type>::iterator it;
```

Can then be initialize by assigning some valid iterator. If we already have an iterator to be assigned at the time of delcaration, then we can skip the type declaration using the auto keyword.

`auto it = iter`

Example:

```
vector<int> arr = {1, 2, 3, 4, 5};

vector<int>::iterator first = arr.begin();

vector<int>::iterator last = arr.end();

while(first != last) {
    cout << *first << " ";
    first++;
}
```

Iterator Function

Return Value

begin()

Returns an iterator to the beginning of container.

end()

Returns an iterator to the theoretical element just after the last element of the container.

cbegin()

Returns a constant iterator to the beginning of container. A constant iterator cannot modify the value of the element it is pointing to.

cend()

Returns a constant iterator to the theoretical element just after the last element of the container.

rbegin()

Returns a reverse iterator to the beginning of container.

rend()

Returns a reverse iterator to the theoretical element just after the last element of the container.

crbegin()

Returns a constant reverse iterator to the beginning of container.

crend()

Returns a constant reverse iterator to the theoretical element just after the last element of the container.

### 11.2.1 Iterator Operations

```
*it;            // Access
*it = new_val;  // Update
it++;           // post-increment
++it;           // pre-increment
it--;
--it;
it + int_val;   // can add or subtract by int val or another iterator
// Comparing two iterators also works (e.g. !=, <=, etc)
```

# 12 Recursion

## 12.1 Recurrence Relations

**Recurrence relation** describes the way a problem depends on a subproblem

### 12.1.1 Solving Recurrences: Linear

$$T(n) = \begin{cases} c_0, & \text{if } n = 0 \\ T(n-1) + c_1, & \text{if } n > 0 \end{cases}$$

Recurrence: $T(n) = T(n-1) + c$

Complexity: $\Theta(n)$

### 12.1.2 Solving Recurrences: Logarithmic

$$T(n) = \begin{cases} c_0, & \text{if } n = 0 \\ T(\frac{n}{2}) + c_1, & \text{if } n > 0 \end{cases}$$

Recurrence: $T(n) = T(n/2) + c$

Complexity: $\Theta(logn)$

### 12.1.3 Master Theorem

A.k.a Master Method

Let $T(n)$ be a monotonically increasing function that satisfies:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$T(1) = c_0$ or $T(0) = c_0$

where $a \geq 1$, $b > 1$. If $f(n) \in \Theta(n^c)$, then:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

**When NOT to use Master Theorem:**

- $T(n)$ is not monotonic, such as $T(n) = sin(n)$
- $f(n)$ is not a polynomial, e.g. $f(n) = 2^n$
- $b$ cannot be expressed as a constant, e.g. $T(n) = T(\sqrt{}(sin(n)))$

### 12.1.4 Fourth Condition

A fourth condition that allows polylogarithmic functions:

# 13 Function Objects, Functors

# 14 Error Handling & Exceptions

# 15 Stacks, Queues, Deque, Priority Queue

## 15.1 Stack

### 15.1.1 Stack - Interface

Stack is a data structure that supports insertion/removal in Last In, First Out (LIFO) order

Stack ADT Interface:

| Method | Description |
|---|---|
| `push(object)` | Add object to top of the stack |
| `pop()` | Remove top element |
| `object &top()` | Return a reference to top element |
| `size()` | Number of elements in stack |
| `empty()` | Checks if stack has no elements |

### 15.1.2 Stack Implementation

A stack can be implemented with an array/vector or linked list

### 15.1.3 Stack in STL

```
#include <stack>
std::stack<>
```

The underlying containers are `std::deque<>` (by default), and `std::list<>, std::vector<>` (optional).

## 15.2 Queue

### 15.2.1 Queue - Interface

Queue is a data structure that supports insertion/removal in First In, First Out (FIFO) order

### 15.2.2 Queue Implementation

### 15.2.3 Queue in STL

```
#include <queue>
std::queue<>
```

The underlying containers are `std::deque<>` (by default), and `std::list<>` (optional).

## 15.3 Deque

### 15.3.1 Deque - Interface

Deque is an abbreviation of Double-Ended Queue

```
#include <deque>
std::deque<>
```

Main methods:

- `push_front()`
- `pop_front()`
- `front()`
- `push_back()`
- `pop_back()`
- `back()`
- `size()`
- `empty()`
- Random Access: `[]` or `.at()`

STL incudes constant time `operator[]()`

### 15.3.2 Deque - Implementation

Circular Buffer

Doubly-linked list

## 15.4 Priority Queue

Each datum in the priority queue is paired with a priority value (usually numbers, should be comparable). Supports **insertion**, **inspection** of data, and **removal** of datum with highest priority.

For elements of the same priority, they are served according to their order in the queue (following FIFO order).

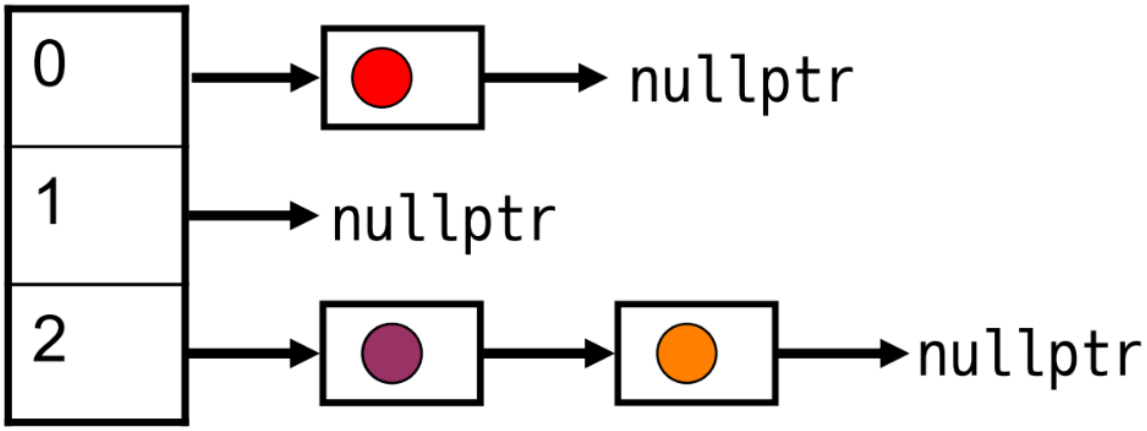


Figure 15.1: Example Priority Queue. Source: EECS 281. Lower numbers here indicate higher priority. Top element would be red node. At priority 2, orange node would be served before purple.

### 15.4.1 ADT - Interface

| Method | Description |
|---|---|
| `push(object)` | Add object to the priority queue |
| `pop()` | Remove highest priority element |
| `const object &top()` | Return a reference to highest priority element |
| `size()` | Number of elements in priority queue |
| `empty()` | Checks if priority queue has no elements |

### 15.4.2 Priority Queue Implementations

Priority queues can be implemented with many data structures. Heap is a common implementation.

| | **Insert** | **Remove** |
|---|---|---|
| Unordered sequence container | Constant | Linear |
| Sorted sequence container | Linear | Constant |
| Heap | Logarithmic | Logarithmic |
| Array of linked lists (for priorities of small integers) | Constant | Constant |

### 15.4.3 C++ Priority Queue

`std::priority_queue<>`

By default, uses `std::less<>` to determine priority. A default priority queue is a "max-PQ", where the largest element has highest priority. To implement a "min-PQ", use `std::greater<>`. Custom comparator (function object) needed if the elements cannot be compared with std less/greater.

Max PQ (`std::less<>`):

```
std::priority_queue<T> myPQ;
```

PQ with custom comparator type, `COMP`:

```
std::priority_queue<T, vector<T>, COMP> myPQ;
```

Manual priority queue implementation with standard library functions:

```
#include <algorithm>
std::make_heap();
std::push_heap();
std::pop_heap();
```

# 16 Generating Permutations

We can generate permutations by "juggling with stacks and queues"

Essentially, given $N$ elements, we want to generate all $N$ element permutations.

Main ingredients:

- One recursive function
- One stack
- One queue

Technique: move elements between the two containers

```
// Helper function for printing
template <typename T>
ostream &operator<<(ostream &out, const vector<T> &v) {
    // display contents of a vector on a single line
    for (auto &el : v) {
        out << el << ' ';
    }
    return out;
}

// Implementation
template <typename T>
void genPerms(vector<T> &perm, deque<T> &unused) {
    if (unused.empty()) {
        // Base case: we have reached a permutation when unused is empty
        //    i.e. a full permutation has been formed
        cout << perm << '\n';
        return;
    }

    for (size_t k = 0; k != unused.size(); ++k) {
        perm.push_back(unused.front());   // Pick the first element from unused
        unused.pop_front();               // Remove this element from unused
        genPerms(perm, unused);           // Recursively generate permutation
```

```
        unused.push_back(perm.back());    // Restore this element to unused
        perm.pop_back();                  // Remove it from the permutation
    }
}

// Example Usage
int main() {
    size_t n = 16;

    vector<size_t> perm;
    deque<size_t> unused(n);
    iota(unused.begin(), unused.end(), 1); // Fills unused with consecutive numbers starting
    genPerms(perm, unused);

    return 0;
}
```

Explanation of `genPerms()`:

- The function iterates over `unused` and chooses each element one by one as it fills up a permutation
- The chosen element is moved from `unused` to `perm` (backtracking)
- The function is recursively called to generate the remaining permutation (as each call picks another element from `unused`)
- After the recursion returns, the removed element is restored to `unused`
- Time complexity: $O(n!)$ since it generates all permutations

### Another Implementation of `genPerms`

```
template <typename T>
void genPerms(vector<T> &path, size_t permLength) {
    if (permLength == path.size()) {
        // Do something with the path
        return;
    }
    if (!promising(path, permLength))
        return;
    for (size_t i = permLength; i < path.size(); ++i) {
        swap(path[permLength], path[i]);
        genPerms(path, permLength + 1);
        swap(path[permLength], path[i]);
    }
}
```

### 16.0.1 STL next_permutation()

The STL has function `std::next_permutation()`

Example Usage

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::string s = "aba";

    do
    {
        std::cout << s << '\n';
    }
    while (std::next_permutation(s.begin(), s.end()));

    std::cout << s << '\n';
}
```

# 17 Complexity Analysis

## 17.1 References

- AlgoMonster - Runtime Summary

## 17.2 Runtime Overview

## 17.3 Common Time Complexities

| Notation | Name |
|---|---|
| O(1) | Constant |
| O(log n) | Logarithmic |
| O(n) | Linear |
| O(n log n) | Loglinear, Linearithmic |
| $O(n^2)$ | Quadratic |
| $O(n^3)$, O(n ), … | Polynomial |
| O(c ) | Exponential |
| O(n!) | Factorial |
| O(2^(2 )) | Doubly Exponential |

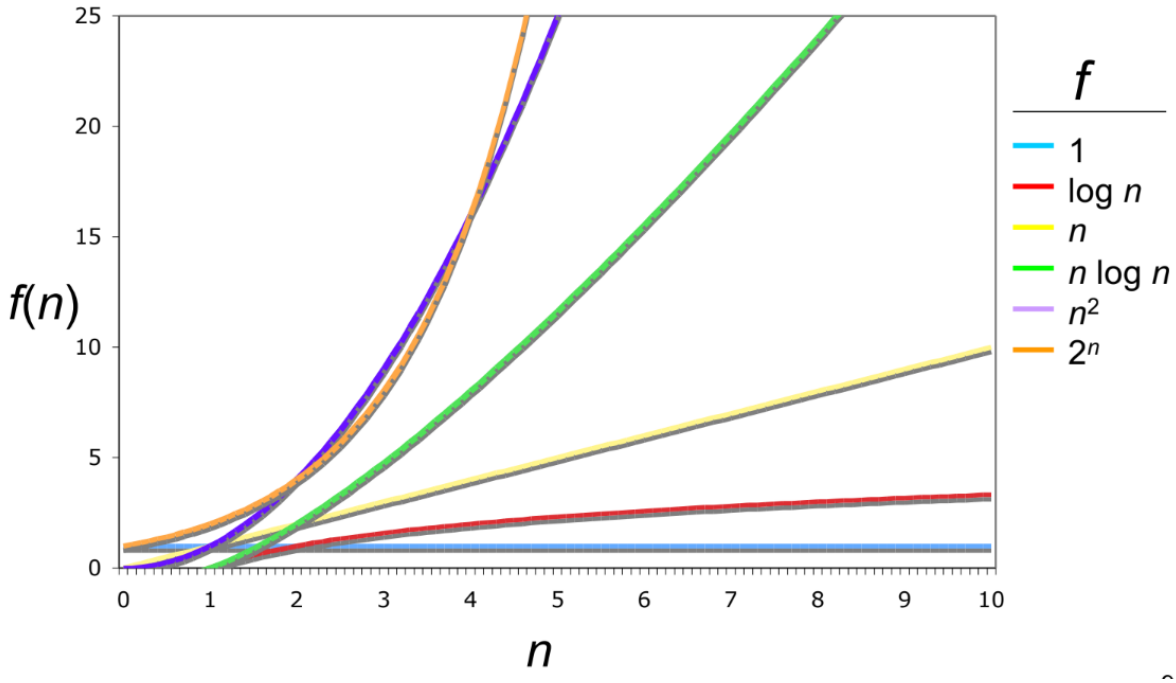


Figure 17.1: Graphing Complexities. Source: EECS 281.

### 17.3.1 O(1) - Constant

Constant time complexity. Could be

- Hashmap lookup
- Array access and update
- Pushing and popping elements from a stack
- Finding and applying math formula

### 17.3.2 O(log(N)) - Logarithmic

$log(N)$ grows very slowly

In coding interviews, log(N) typically means:

- Binary search or variant
- Balanced binary search tree lookup
- Processing the digits of a number

Unless specified, typically log(N) refers to $log_2(N)$

Example C++:

```
int N = 100000000;
while (N > 0) {
  // some constant operation
  N /= 2;
}
```

Many mainstream relational databases use binary trees for indexing by default, thus lookup by primary key in a relational database is log(N).

### 17.3.3 O(N) - Linear

Linear time typically means looping through a linear data structure a constant number of times. Most commonly, this means:

- Going through array/linked list
- Two pointers
- Some types of greedy
- Tree/graph traversal
- Stack/Queue

Example C++:

```
for (int i = 1; i <= N; i++) {
  // constant time code
}

for (int i = 1; i < 5 * N + 17; i++) {
  // constant time code
}

for (int i = 1; i < N + 538238; i++) {
  // constant time code
}
```

### 17.3.4 O(K log(N))

- Heap push/pop K times. When you encounter problems that seek the "top K elements", you can often solve them by pushing and popping to a heap K times, resulting in an O(K log(N)) runtime. e.g., K closest points, merge K sorted lists.
- Binary search K times.

Since K is constant this kind of isn't its own time complexity and can be grouped with O(log(N))

### 17.3.5 O(N log(N)) - Log-Linear

- Sorting. The default sorting algorithm's expected runtime in all mainstream languages is N log(N). For example, java uses a variant of merge sort for object sorting and a variant of Quick Sort for primitive type sorting.
- Divide and conquer with a linear time merge operation. Divide is normally log(N), and if merge is O(N) then the overall runtime is O(N log(N)). An example problem is smaller numbers to the right.

### 17.3.6 O(N^2) - Quadratic

- Nested loops, e.g., visiting each matrix entry
- Many brute force solutions

```
for (int i = 1; i <= N; i++) {
  for (int j = 1; j <= N; j++) {
    // constant time code
  }
}
```

### 17.3.7 O(2^N) - Exponential

Grows very rapidly. Often requires memoization to avoid repeated computations and reduce complexity.

- Combinatorial problems, backtracking, e.g. subsets
- Often involves recursion and is harder to analyze time complexity at first sight

E.g.: A recursive Fibonacci algorithm is $O(2^N)$

```
int Fib(int n) {
  if (n == 0 || n == 1) {
    return 1;
  }
  return Fib(n - 1) + Fib(n - 2);
}
```

### 17.3.8 O(N!) - Factorial

Grows very very rapidly. Only solvable by computers for small N. Often requires memoization to avoid repeated computations and reduce complexity.

- Combinatorial problems, backtracking, e.g. permutations
- Often involves recursion and is harder to analyze time complexity at first sight

## 17.4 Big-O, Big-Theta, and Big-Omega

| | **Big-O (O)** | **Big-Theta (Θ)** | **Big-Omega (Ω)** |
|---|---|---|---|
| **Defines** | Asymptotic upper bound | Asymptotic tight bound | Asymptotic lower bound |
| **Definition** | $f(n) = O(g(n))$ if and only if there exists an integer $n_0$ and a real number $c$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$ | $f(n) = \Theta(g(n))$ if and only if there exists an integer $n_0$ and real constants $c_1$ and $c_2$ such that for all $n \geq n_0$ : $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ | $f(n) = \Omega(g(n))$ if and only if there exists an integer $n_0$ and a real number $c$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$ |
| **Mathematical Definition** | $\exists n_0 \in \mathbb{Z}, \exists c \in \mathbb{R}: \forall n \geq n_0, f(n) \leq c \cdot g(n)$ | $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$ | $\exists n_0 \in \mathbb{Z}, \exists c \in \mathbb{R}: \forall n \geq n_0, f(n) \geq c \cdot g(n)$ |
| $f_1(n) = 2n + 1$ | $O(n)$ or $O(n^2)$ or $O(n^3)$... | $\Theta(n)$ | $\Omega(n)$ or $\Omega(1)$ |
| $f_2(n) = n^2 + n + 5$ | $O(n^2)$ or $O(n^3)$... | $\Theta(n^2)$ | $\Omega(n^2)$ or $\Omega(n)$ or $\Omega(1)$ |

## 17.5 Amortized Time Complexity

# 18 STL

# 19 Trees

A **graph** consists of **nodes/vertices** connected together by **edges**. Each node can contain some data.

A **tree** is

1. A connected graph (nodes + edges) without cycles
2. A graph where any 2 nodes are connected by a unique shortest path

The two definitions above are equivalent.

Types of Trees:

- (Simple) tree
- Rooted tree

In a **directed tree**, we can identify **child** and **parent** relationships between nodes.

In a **binary tree**, a node has at most two children.

**Terminology:**

- **Root**: the topmost node in the tree
- **Parent**: Immediate predecessor of a node
- **Child**: Node where current node is parent
- **Ancestor**: parent of a parent (closer to root)
- **Descendent**: child of a child (further from root)
- **Internal node**: a node with children
- **Leaf node**: a node without children

```
template <class Item>
struct Node {       // a binary tree node
    Node *left;     // pointer to left child
    Node *right;    // pointer to right child
    Item item;      // data or KEY
}
```

**Tree Properties**

Height:

height(empty) = 0

height(node) = max(heght(left_child), height(right_child)) + 1

Size:

size(empty) = 0

size(node) = size(left_child) + size(right_child) + 1

Depth:

depth(empty) = 0

depth(node) = depth(parent) + 1

## 19.1 Complete (Binary) Trees

> Note 3: Definition - Complete Binary Tree
>
> **Complete Binary Tree:** every level, except possibly the last, is completely filled, and all nodes are as far left as possible

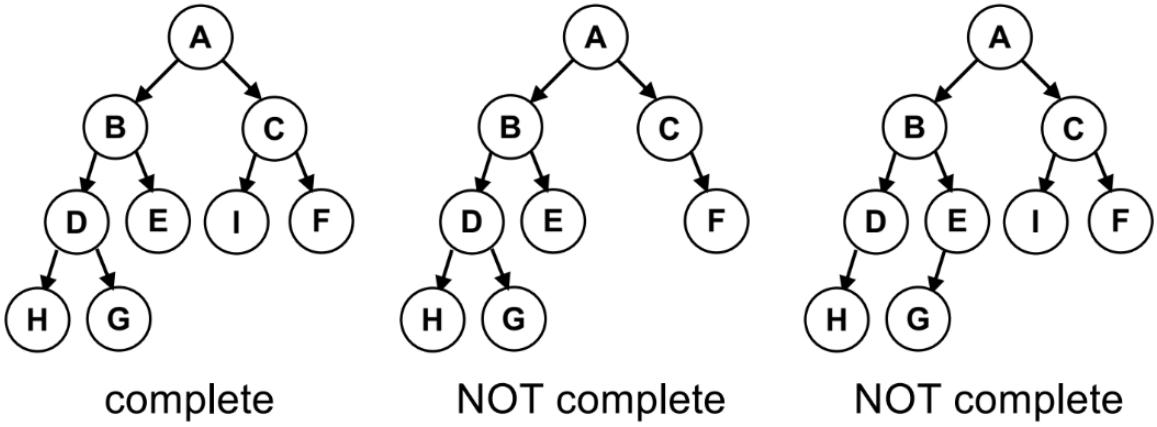


Figure 19.1: Source: EECS 281.

A complete binary tree can be stored efficiently in a growable array (vector) by indexing nodes according to level-ordering

- Completeness ensures no gaps in the array, thus easily indexable
- Index start at 1 (to make math easier)
  - To achieve 1-based indexing with 0-indexed vector, just add a dummy element at position 0 and ignore it

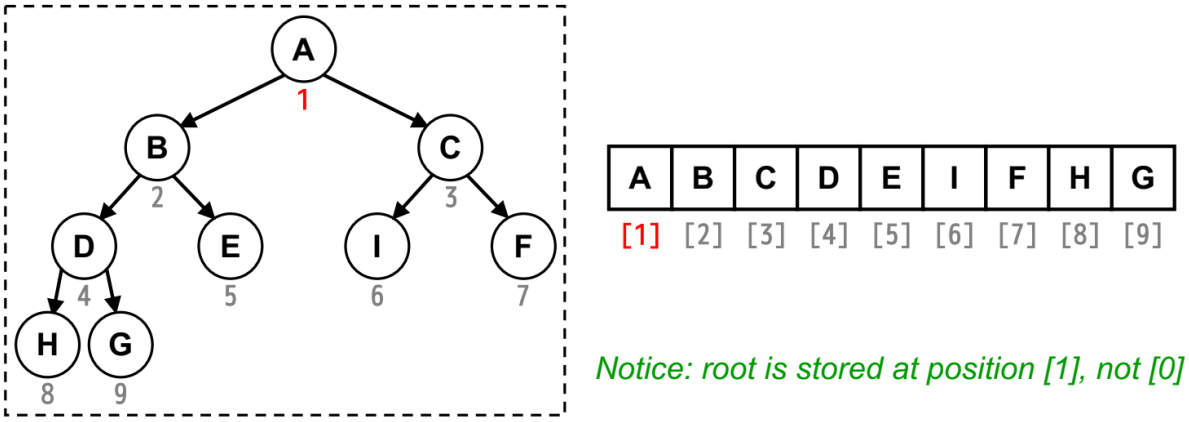


Figure 19.2: Source: EECS 281.

## 19.2 Binary Search Trees (BST)

Each node in a BST has at most two children, with the left child containing values less than the parent and the right child containing values greater than the parent.

Allows for efficient searching, insertion, and deletion.

E.g. finding the max/min is as simple as finding the rightmost or leftmost node. E.g., rightmost can be found by keeping on accessing the right child until a node no longer has a right child.

## 19.3 Tree Traversals

**In-Order** traversal visits:

1. left subtree
2. current node
3. right subtree

**Pre-Order** traversal visits:

1. current node
2. left subtree

3. right subtree

**Post-Order** traversal visits:

1. left subtree
2. right subtree
3. current node

**Level-Order** traversal visits nodes in order of increasing depth in tree. Same as bradth-first traversal.

C++ example:

```
template <class KEY>
struct Node {
    KEY key;
    Node *left = nullptr;
    Node *right = nullptr;
};  // Node{}

void preorder(Node *p) {
    if (!p) return;
    visit(p->key);
    preorder(p->left);
    preorder(p->right);
}  // preorder()

void postorder(Node *p) {
    if (!p) return;
    postorder(p->left);
    postorder(p->right);
    visit(p->key);
}  // postorder()

void inorder(Node *p) {
    if (!p) return;
    inorder(p->left);
    visit(p->key);
    inorder(p->right);
}  // inorder()
```

## 19.4 AVL Tree

# 20 Heaps

## 20.1 References

- AlgoMonster - Heap Fundamentals
- Geeks for Geeks - STL Heap

## 20.2 Heap? Priority Queue?

Priority Queue is an **Abstract Data Type**, and Heap is the concrete data structure we use to implement a priority queue. Source

## 20.3 Heap - Definitions & Properties

Definition: A tree is **(max) heap-ordered** if each node's priority is not greater than the priority of the node's parent

Definition: A **heap** is a set of nodes with keys (priorities) arranged in a complete heap-ordered tree, represented as an array

Property: No node in a heap has a key (priority) larger than the root's key

A heap has two crucial properties (representation invariants):

1. Completeness
2. Heap-ordering

These two properties can be leveraged to create an efficient priority queue and an efficient sorting algorithm using a heap!

Loose definition: data structure that gives easy access to the most extreme element, e.g., maximum or minimum

**"Max Heap":** heap with largest element being the "most extreme"

**"Min Heap":** heap with smallest element being the "most extreme"

Heaps use complete (binary) trees* as the underlying structure, but are often implemented with arrays

*Note: Not to be confused with binary* search* trees.

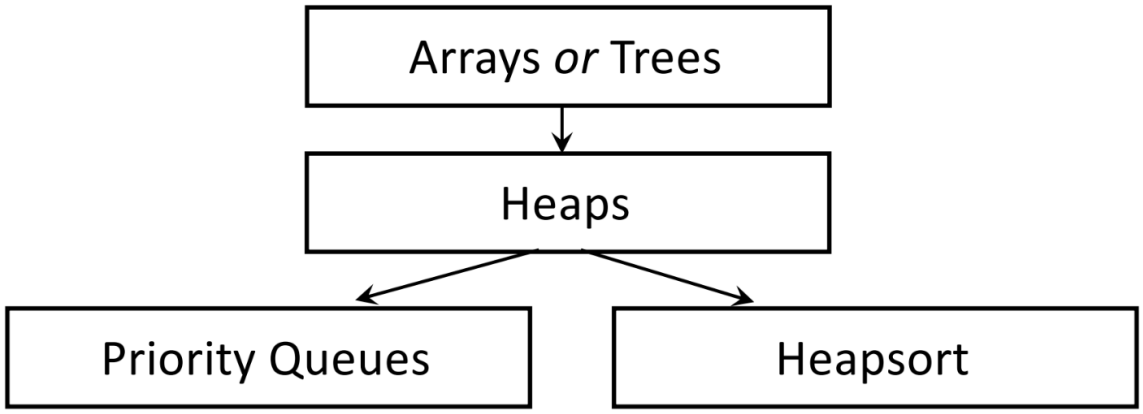


Figure 20.1: Source: EECS 281

## 20.4 Modifying Heaps

### 20.4.1 Increasing Node Priority

If the priority of a node in the heap is increased, we need to fix **bottom-up**:

1. Pass index k of array element with increased priority
2. Swap the node's key with the parent's key until:
    - a. the node has no parent (it is the root), or
    - b. the node's parent has a higher (or equal) priority key

```
Code goes here
```

### 20.4.2 Decreasing Node Priority

fix **top-down**

## 20.5 Heap in C++

### 20.5.1 make_heap()

`std::make_heap()` is used to convert the given range in a container to a heap. Max heap by default. Use a custom comparator to change it to a min heap.

```
// Initializing a vector
std::vector<int> v1 = { 20, 30, 40, 25, 15 };

// Converting vector into a heap
std::make_heap(v1.begin(), v1.end());

std::cout << v1.front() << '\n'; // Displays max element of heap
```

### 20.5.2 push_heap()

`std::push_heap(begin_iterator, end_iterator)` sorts the heap after insertion. Can you `push_back()` of vector class to insert.

```
vector<int> vc{ 20, 30, 40, 10 };
make_heap(vc.begin(), vc.end());

vc.push_back(50);
push_heap(vc.begin(), vc.end()); // now the heap is sorted
```

Time Complexity: $O(logN)$

Note: The `push_heap()` function should only be used after the insertion of a single element at the back. Undefined for random insertion or for building a heap.

### 20.5.3 pop_head()

`pop_heap()` is used to move largest element of the heap to the end of the heap, so then a `pop_back()` can be used to remove the element

```
pop_heap(vc.begin(), vc.end()); // moves largest element to the end
vc.pop_back(); // removes element at the end
```

Time Complexity: $O(logN)$

### 20.5.4 sort_heap()

`sort_heap()` is used to sort the heap in ascending order using heapsort.

```
sort_heap(v1.begin(), v1.end());
```

Time Complexity: $O(NlogN)$

### 20.5.5 is_heap()

`is_heap()` can be used to check whether a given range of the container is a heap. Default checks for max heap.

### 20.5.6 is_heap_until()

`is_heap_until()`

## 20.6 Priority Queue - Heap Implementation

**Definition:** A **priority queue** is a data structure that supports the following three basic operations:

- push(): **insertion** of a new node
- top(): **inspection** of highest priority node
- pop(): **removal** of highest priority node

Priority queue is useful for many algorithms: e.g., Dijkstra's, heapsort, sorting in reverse order

Priority queues are often implemented using heaps because the same time complexity of insertion/removal.

### 20.6.1 Insertion & Deletion

TODO

### 20.6.2 Priority Queue - Summary

Another summary can be found in the Stacks, Queues, Priority Queues page.

| Priority Queue Type | Insertion Complexity | Inspection Complexity | Removal Complexity | Notes |
|---|---|---|---|---|
| **Unordered Array PQ** | O(1) | O(n) | O(n), or O(1) | - |
| **Sorted Array PQ** | O(n) | O(1) | O(1) | - |
| **Heap** | O(log n) using `fixUp()` | O(1) | O(log n) using `fixDown()` | Must maintain heap property |

## 20.7 Heapify

## 20.8 Heapsort

# 21 Searching

## 21.1 Linear Search

## 21.2 Binary Search

Searching in a sorted container.

Asymptotic Complexity = O(log n)

### 21.2.1 C++ Implementation

Basic Implementation:

```
int bsearch(double a[], double val, int left, int right) {
    // n = right - left = size of a[]
    while (right > left) {                      // loops at most k times
        int mid = left + (right - left) / 2;    // find midpoint of current range
        if (val == a[mid]) return mid;          // midpoint is the search val, return it

        if (val < a[mid]) right = mid;          // target val is less than midpoint, update r
        else left = mid + 1;                    // target val is greater than midpoint, updat
    }

    return -1;      // val not found
}
```

Optimization - 2 comparisons half the time:

```
int bsearch(double a[], double val, int left, int right) {
    // n = right - left = size of a[]
    while (right > left) {                      // loops at most k times
        int mid = left + (right - left) / 2;    // find midpoint of current range

        if (val > a[mid]) left = mid + 1;
```

```
        else if (val < a[mid]) right = mid;
        else return mid;
    }

    return -1;      // val not found
}
```

Optimization - always 2 comparisons per loop (requires `val` to be present in `a[]`):

```
int bsearch(double a[], double val, int left, int right) {
    // n = right - left = size of a[]
    while (right > left) {
        int mid = left + (right - left) / 2;    // find midpoint of current range

        if (val > a[mid]) left = mid + 1;       // target val greater than midpoint, update l
        else right = mid;                       // target val <= midpoint, update right bound
    }

    return left;
}
```

### 21.2.2 STL Binary Search

`binary_search()` returns a `bool`

To find locations (iterators):

- `lower_bound()` First item not less than target
- `upper_bound()` First item greater than target
- `equal_range()` Pair(lb, ub), all items equal to target

## 21.3 Tree Search

### 21.3.1 Depth-First Search (DFS)

### 21.3.2 Breadth-First Search (BFS)

### 21.3.3 Dijkstra's Algorithm

### 21.3.4 Floyd's Algorithm

### 21.3.5 Bellman-Ford

Can handle negative weight edges that Dijkstra's cannot.

# 22 Sets

## 22.1 Set Operations

- Union ($\cup$): in one set or the other (OR)
- Intersection ($\cap$): in both sets (AND)
- Set Difference ($\setminus$): in one and not the other (AND-NOT)
- Symmetric Difference ($\div$): in only one (XOR)
- addElement ($+$)
- isElement ($\in$)
- isEmpty

### 22.1.1 set_union()

| Method | Asymptotic Complexity |
|---|---|
| `initialize()` | O(1) or O(n log n) |
| `clear()` | O(1) or O(n) |
| `isMember()` | O(log n) |
| `copy()` | O(n) |
| `set_union()` | O(n) |
| `set_intersection()` | O(n) |

## 22.2 Set in C++

### 22.2.1 References

Geeks for Geeks - C++ STL Set

```
std::set<data_type> set_name;

set<int> my_set;
set<int> my_set = {1, 4, 5, 12};
```

```cpp
my_set.insert(10);

for (auto& num : my_set) {
    std::cout << num << ' ';
}

set<int>::iterator it;

my_set.erase(12);

// returns an iterator to the element, otherwise returns my_set.end()
my_set.find(5);
```

## 22.3 Union-Find

# 23 Sorting

| Elementary Sorts | High-Performance Sorts |
|---|---|
| Bubble Sort | Quicksort |
| Selection Sort | Merge Sort |
| Insertion Sort | Heapsort |
| Counting Sort *(int, few different keys)* | Radix Sort *(int, char, ...)** |

```
bool operator<(const T&, const T&) const
```

STL sorting is done with iterators:

```
#include <algorithm>
vector<int> a{4, 2, 5, 1, 3};
std::sort(begin(a), end(a)); // sorts in ascending order
```

## 23.1 Types

**Non-Adaptive Sort**

**Adaptive Sort**

## 23.2 Bubble Sort

## 23.3 Selection Sort

## 23.4 Insertion Sort

## 23.5 Counting Sort

## 23.6 Quick Sort

High-level idea:

1. Choose a pivot element from the array
2. Partition array into two subarrays
    1. One containing elements smaller than the pivot
    2. One containing elements larger than the pivot
3. Recursively apply quick sort to each subarray until the entire array is sorted

Average time complexity: O(n log n) Worst-case time complexity: O(n^2)

Worst case comes when the pivot element is selected as the max/min element of the entire array, thus the pivot doesn't do much. Can avoid by picking a good pivot, such as by using the median or a random element.

Quick sort is an ***in-place*** algorithm, requiring no additional memory beyond the original array.

## 23.7 Merge Sort

## 23.8 Heapsort

Refer to heaps notes.

## 23.9 Radix Sort

# 24 Dictionary and Hash Maps

aka Hash Tables

## 24.1 Dictionary Abstract Data Type

A container of items (key/value pairs) that supports two basic operations:

- Insert a new item
- Search (retrieve) an item with a given key

Operations:

## 24.2 Hash Maps / Hash Tables

### 24.2.1 Hashing - Definition

### 24.2.2 Collision Resolution

### 24.2.3 Hash Maps in C++

C++ hash table containers:

- `unordered_set<>`, `unordered_multiset<>`
- `unordered_map<>`, `unordered_multimap<>`

Example:

```
#include <unordered_map> // among other includes...
unordered_map<string, int> months;
string month;

months["January"] = 31;
months["February"] = 28;
...
```

```
// Looking up an item
month = "March";
auto it = months.find(month);
if (it == months.end()) {
    // month not found
} else {
    cout << it->first << " has " << it->second << " days\n";
}
```

**When NOT to use hash tables:**

- Significant space overhead of nested containers
- Every access computes hash function
- O(n) worst-case time (STL implementations)
- When keys are small ints (use bucket arrays)
- For static data, set membership, or lookup (consider sorting + binary search)
- Key-value storage but traversal is needed and not lookup

# 25 Graphs

## 25.1 Definitions & Terminologies

## 25.2 Representations

Adjacency Matrix, Distance Matrix, Adjacency List

## 25.3 Shortest Path

Single-source shortest path, can solve with:

- DFS
  - Only works on trees
- BFS
  - Works for unweighted edges (or when all edges have same weight)
- Dijkstra's
  - Works for weighted edges

## 25.4 Traveling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

I.e. given an undirected weighted graph, find a path that visits every node and returns to the first node with minimum total cost.

## 25.5 Minimum Spanning Tree

Given: edge-weighted, undirected graph $G = (V, E)$

Find: subgraph $T = (V, E'), E' \subseteq E$ such that:

- All vertices are pair-wise connected
- The sum of all edge weights in $T$ is minimal

If there's a cycle in T, remove edge with highest weight

Therefore T must be a tree (since it must have no cycles)

Two algorithms to find MST:

- Prim's
- Kruskal's

### 25.5.1 Prim's Algorithm

### 25.5.2 Kruskal's Algorithm

# 26 Brute-Force & Greedy Algorithms

## 26.1 Brute-Force

Definition:

## 26.2 Greedy Algorithms

Definition: At every decision step, makes the locally optimal decision.

To guarantee correctness/optimality, must show that locally optimal solutions lead to globally optimal solution

## 26.3 Summary

Brute-force

- Solve problem in simplest way
- Generate entire solution set, pick best
- Will give optimal solution with (typically) poor efficiency

Greedy

- Make local, best decision, and don't look back
- May give optimal solution with (typically) 'better' efficiency
- Depends upon 'greedy-choice property'
- Global optimum found by series of local optimum choices

# 27 Divide and Conquer, Dynamic Programming

## 27.1 Divide and Conquer

Repeatedly divide a problem into smaller *non-overlapping* problems (preferably of equal size)

- Often recursive
- Often involve log n complexities
- Top down approach
- E.g. binary search, quicksort

Alternatively, there is also "Combine and Conquer" Algorithms, that are bottom-up: start with smallest subproblem possible, then combine increasingly large subdomains until the given problem size.

- E.g. Merge Sort

## 27.2 Dynamic Programming

Remembers partial solutions (memoization) of *overlapping* subproblems

Solve small subproblems first and store the results, then use these results when needed.

Can use bottom-up or top-down approach

### 27.2.1 Simple Example - Fibonacci

Top-Down DP:

```
uint64_t fib(uint32_t n) {
    // Array of known Fibonacci numbers. Start out with 0, 1,
    // and the rest get automatically initialized to 0.
    // MAX_FIB + 1 used to account for 0-indexing
    static uint64_t fibs[MAX_FIB + 1] = { 0, 1 };

    // Doesn't fit in 64 bits, so don't even bother computing
    if (n > MAX_FIB)
        return 0;

    // Is already in array, so look it up
    if (fibs[n] > 0 || n == 0)
        return fibs[n];

    // Currently unknown, so calculate and store it for later
    fibs[n] = fib(n - 1) + fib(n - 2);
    return fibs[n];
}
```

Bottom-Up DP:

```
uint64_t fibBU(uint32_t i) {
    uint64_t f[MAX_N];
    i = min(i, MAX_N - 1);
    f[0] = 0;
    f[1] = 1;
    for (size_t k = 2; k <= i; k++)
        f[k] = f[k - 1] + f[k - 2];
    return f[i];
}
```

### 27.2.2 Simple Example - Binomial Coefficient

### 27.2.3 General Approach

Top-Down:

- Save known values as they are calculated
- Generally preferred because it can be more naturally transformed from recursive solution, the order of computing the subproblems takes care of itself, and it may not need to compute all subproblems

Bottom-Up:

- Precompute values from base case up towards the solution. Note it will compute all subproblems, regardless if it is needed.

### 27.2.4 Example - Knight Moves

### 27.2.5 Example - Knapsack Problem

#### 27.2.5.1 Problem Definition

A knapsack has capacity $M$. An item has various weights and values.

**Problem:** Find the maximum value of items that can be packed into the knapsack with $N$ items that does not exceed capacity $M$.

Many variations of the problem exist:

- Each item is unique (the standard formulation), aka 0-1 Knapsack Problem. Must take (1) or leave (0) an item
- Finite amount of each item
- Infinite number of copies of each item
- Fractional amount of item can be taken

#### 27.2.5.2 Bottom-Up Example

Approach:

- For each item:
  - Can fit AND improves overall value? Take the item
  - Too large to fit in current capacity (or doesn't improve value)? Leave behind

Subproblem: What is the maximum value of items at this index with this capacity?

Thus, will use two nested loops:

1. Loop through all items
2. Look through knapsack sizes from 0 to $M$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

Safe:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Size** | 1 | 2 | 5 | 6 | 7 |
| **Value** | 1 | 6 | 18 | 22 | 28 |

Knapsack:

| Safe Item # \ Knapsack Size | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 3 | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| 4 | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| 5 | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 35 | 40 |

**C++ implementation:**

```cpp
uint32_t knapsackDP(const vector<Item> &items, const size_t m) {
    const size_t n = items.size();  // Get the number of items
    vector<vector<uint32_t>> memo(n + 1, vector<uint32_t>(m + 1, 0)); // DP table, initializ

    for (size_t i = 0; i < n; ++i) {          // Iterate over items
        for (size_t j = 0; j < m + 1; ++j) { // Iterate over capacities (0 to m)
            if (j < items[i].size)            // If item doesn't fit in knapsack
                memo[i + 1][j] = memo[i][j]; // Inherit value from previous row
            else                              // If item fits in the knapsack
                memo[i + 1][j] = max(memo[i][j],
                                     memo[i][j - items[i].size] + items[i].value);
                                              // Either take it or leave it (maximize value)
        }
    }

    return memo[n][m];  // Maximum value obtainable with `n` items and capacity `m`
}
```

Time complexity: $O(MN)$

Explanation:

- Table construction:
  - $i$ represents the number of items that is being considered, i.e. considering all items 0 to index $i$
  - $j$ represents the current knapsack capacity
  - Each cell of the DP table stores the max value that can be obtained using the first $i$ items within capacity $j$
- Algorithm:
  - Start with no items and an empty knapsack
  - Go through each item and each capacity (double for loop)
    - If the item fits and increases overall value: take it
      - `memo[i][j]` is the value of the previous row, i.e. not taking the item
      - `memo[i][j - items[i].size()] + items[i].value` is the value of taking this item (value of knapsack with capacity that just reaches current capacity if you take this item + this item's value)
    - Otherwise, don't take the item: keep value from previous row
- Subproblem:
  - `memo[i][j]` stores the best value possible using only the first $i$ items and knapsack of size $j$

Reconstructing the solution:

- included items improve a smaller solution, while excluded items don't
- if a smaller solution + an item is $\geq$ a full solution without the item, it is included. Otherwise, exclude
- the items taken can be reconstructed from the completed memo by working backwords from $(n, m)$ to $(0, 0)$

C++ implementation of the reconstruction:

```
vector<bool> knapDPReconstruct(const vector<Item> &items,
                               const vector<vector<uint32_t>> &memo,
                               const size_t m) {
    const size_t n = items.size();  // Number of items
    size_t c = m;                   // Current capacity (tracking remaining space)
    vector<bool> taken(n, false);   // Boolean vector to track selected items

    for (int i = n - 1; i >= 0; --i) {  // Iterate in reverse to reconstruct solution
```

```
        if (items[i].size <= c) {       // If the item fits in the remaining capacity
            if (memo[i][c - items[i].size] + items[i].value >= memo[i][c]) {
                taken[i] = true;        // Mark item as taken
                c -= items[i].size;     // Reduce the remaining capacity
            }
        }
    }

    return taken; // Return boolean array indicating chosen items
}
```

# 28 Backtracking & Branch and Bound Algorithms

Can be used to solve the following types of algorithm problems:

- **Constraint satisfaction problems**
  - Stop when a satisfying solution is found
    - one solution is sufficient
  - Can rely on **backtracking algorithms**
  - E.g. sorting, mazes, spanning tree, puzzles, GRE/analytical
- **Optimization problems**
  - Usually cannot stop early (since we need to find a global min/max)
  - Must develop a set of possible solutions
    - "feasibility set"
    - Then pick best solution
  - Can rely on **branch and bound algorithms**
  - E.g. giving change, MST
-

## 28.1 Backtracking Algorithms

Consider all possible outcomes of each decision, but prune searches that do not satisfy constraints

This prevents us from having to perform exhaustive search, however the search space is still large.

- Branch on every possibility
- Maintain one or more "partial solutions"
- Check every partial solution for validity
  - If a partial solution violates some constraint, prune it since it makes no sense to go further, i.e. backtrack

### 28.1.1 General Form

Pseudocode:

```
Algorithm checknode(node v)
    if (promising(v))
        if (solution(v))
            write solution*
        else
            for each node u adjacent to v
                checknode(u)
```

Alternate Form:

```
Algorithm checknode(node v)
    if (solution(v))
        write solution*
    else
        for each node u adjacent to v
            if (promising(u))
                checknode(u)
```

`solution(v)` - checks 'depth' of the solution (constraint satisfaction)

`promising(v)` - checks whether should be pruned. Different for each application

`checknode(v)` - called only if partial solution is both promising and not a solution (i.e. branch to check further)

Often the most difficult part is determining `promising()`

### 28.1.2 Example: M-Coloring

Given:

- n (number of nodes)
- m (number of colors)
- W[0…n)[0…n) (adjacency matrix), where W[i][j] is true iff i is connected to node j

Find all possible colorings of graph represented by `int vcolor[0...n)`, where `vcolor[i]` is the color associated with node `i`

Pseudocode:

```
Algorithm m_coloring(index i = 0)
    if (i == n)
        print vcolor(0) thru vcolor(n - 1)
        return
    for (color = 0; color < m; color++)
        vcolor[i] = color
        if (promising(i))
            m_coloring(i + 1)

bool promising(index i)
    for (index j = 0; j < i; ++j)
        if (W[i][j] and vcolor[i] == vcolor[j])
            return false
```

promising is essentially where we check partial solution for validity. If it is not promising, we don't go further, i.e. prune this branch.

### 28.1.3 Example: n Queens

Can $n$ queens be placed on a $1 \times 1$ board so that it doesn't threaten another?

## 28.2 Branch and Bound Algorithms

Essentially extends backtracking to optimization problems

Minimizing a function with this property:

- a partial solution is pruned if its cost $\geq$ cost of best known complete solution
- e.g. length of a path

The property is esentially saying that if the cost of a partial solution is too big, drop this partial solution (because cannot be optimal)

### 28.2.1 General Form

Pseudocode:

```
Algorithm checknode(Node v, Best currBest)
    Node u
    if (promising(v, currBest))
        if (solution(v)) then
            update(currBest)
        else
            for each child u of v
                checknode(u, currBest)
    return currBest
```

`solution()` - checks 'depth' of solution (constraint satisfaction)

`update()` - if the new solution is better than the current solution, update the known current best solution

`checknode()` - called only if promising and not solution (extend and go further to this branch)

`lowerbound()` - an estimate of the solution based on the cost so far and the underestimate of the remaining cost (bound)

`promising()` - return true when `lowerbound() < currBest`, i.e. it is promising to look into further since it is potentially (by the lowerbound) the optimal solution. Return of false results in pruning

The key idea to B&B is the **bound**, by bounding away (pruning) unpromising partial solutions

Minimizing With B&B:

- Start with an "infinity" bound
- Find first complete solution – use its cost as an upper bound to prune the rest of the search
- Measure each partial solution and calculate a lower bound estimate needed to complete the solution
- Prune partial solutions whose lower bounds exceed the current upper bound
- If another complete solution yields a lower cost – that will be the new upper bound
- When search is done, the current upper bound will be a minimal solution

Maximizing With B&B

- Start with a "zero" bound
- Find first complete solution – use its cost as a lower bound to prune the rest of the search
- Measure each partial solution and calculate an upper bound estimate needed to complete the solution
- Prune partial solutions whose upper bounds are less than the current lower bound

- If another complete solution yields a larger value – that will be the new lower bound
- When search is done, the current lower bound will be a maximal solution

### 28.2.2 Summary

## 28.3 Traveling Salesperson Problem (TSP)

### 28.3.1 Definition

Hamiltonian Cycle -

TSP is a optimization problem that asks to find the Hamiltonian cycle with least weight

We can apply both backtracking and B&B to TSP.

# 29 Trie

A.k.a. Prefix Tree, Radix Tree, or Digital Tree

# 30 Summary

In summary...

# 31 Tips on Solving DS&A Questions

## 31.1 Problem Solving Flowchart

## 31.2 DS&A Roadmap

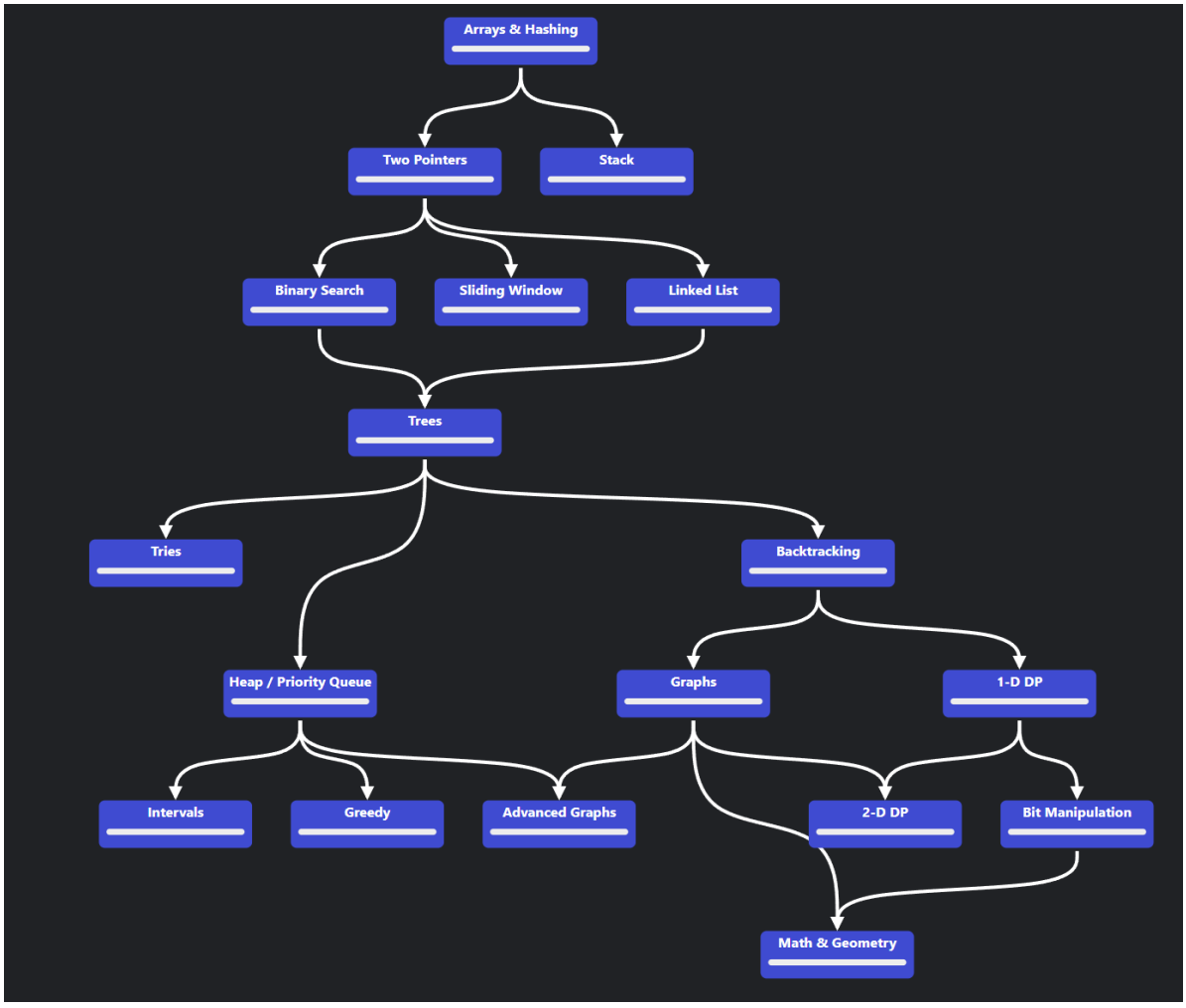


Figure 31.2: A Roadmap for studying. Source

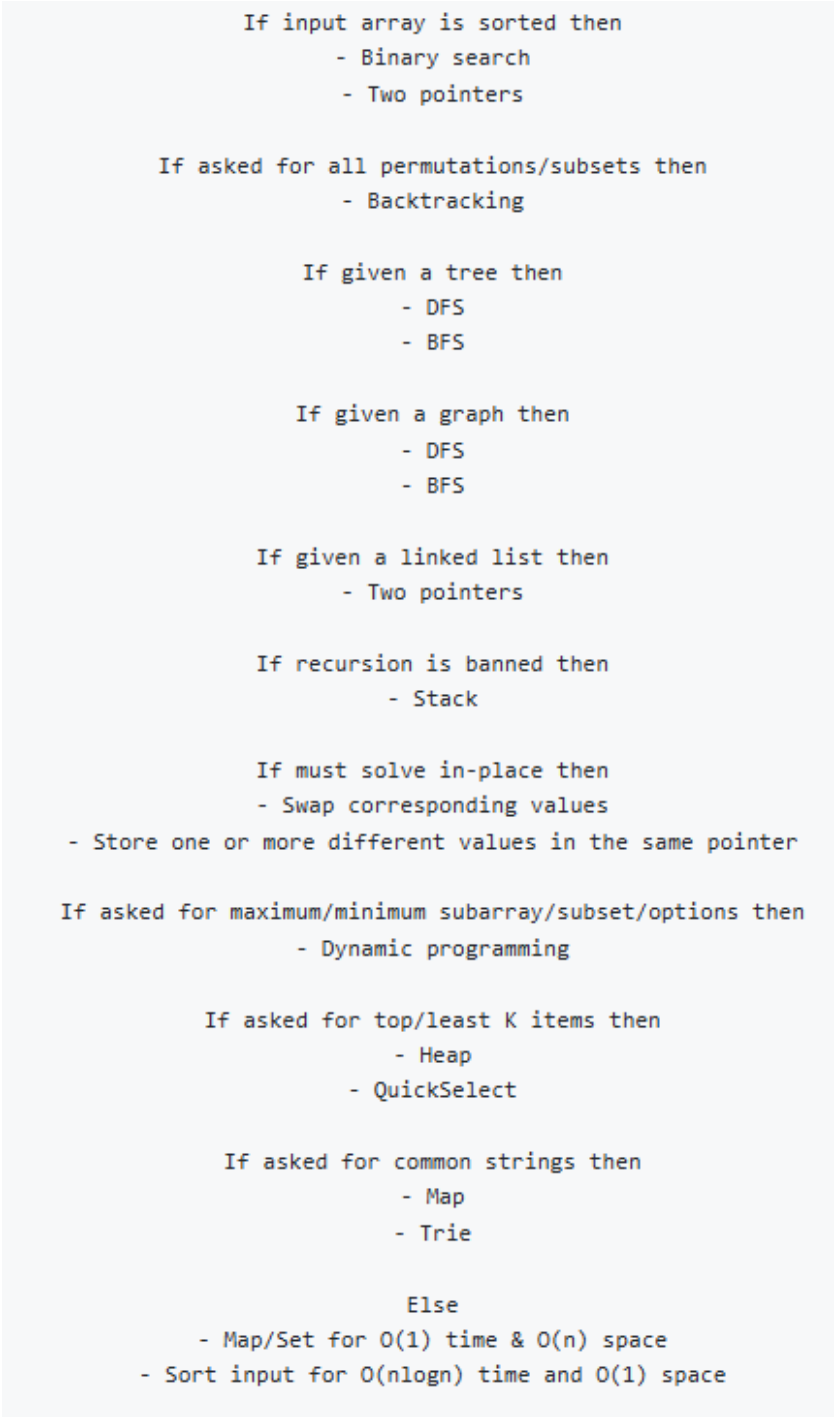


Figure 31.1: Tips on problem approach.Image Source

## 31.3 Problem Flowchart

## 31.4 ROI

## 31.5 "Academic" Algorithms

According to AlgoMonster, some **algorithms** that are very rarely/almost never asked in interviews:

- Minimal spanning tree: Kruskal's algorithm and Prim's algorithm
- Minimum cut: Ford-Fulkerson algorithm
- Shortest path in weight graphs: Bellman-Ford-Moore algorithm
- String search: Boyer-Moore algorithm

## 31.6 Keyword to Algo

AlgoMonster provides a convenient "Keyword to Algorithm" summary:

"Top k"

- Heap
  - E.g. K closest points

"How many ways.."

- DFS
  - E.g. Decode ways
- DP
  - E.g. Robot paths

"Substring"

- Sliding window
  - E.g. Longest substring without repeating characters

"Palindrome"

- two pointers: Valid Palindrome
- DFS: Palindrome Partitioning
- DP: Palindrome Partitioning II

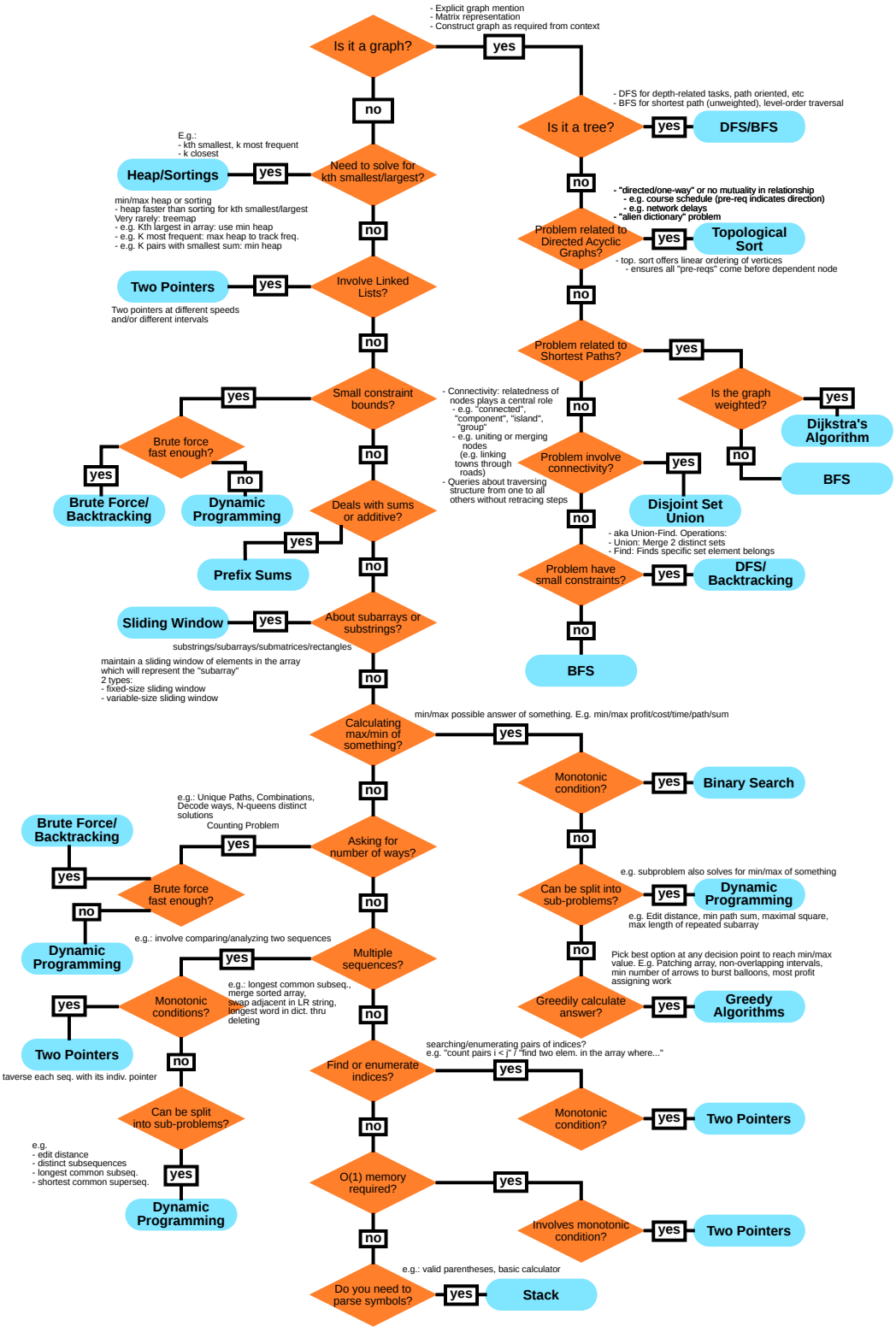


Figure 31.3: A problem solving flowchart based on AlgoMonster's flowchart. Open the SVG in a new tab and zoom in/out for better viewing.

| Topic | Difficulty to Learn |
|---|---|
| Two Pointers | Easy |
| Sliding Window | Easy |
| Breadth-First Search | Easy |
| Depth-First Search | Medium |
| Backtracking | High |
| Heap | Medium |
| Binary Search | Easy |
| Dynamic Programming | High |
| Divide and Conquer | Medium |
| Trie | Medium |
| Union Find | Medium |
| Greedy | High |

Figure 31.4: Studying to Maximizing ROI according to AlgoMonster.

"Tree"

- shortest, level-order
  - BFS: Binary Tree Level-Order Traversal
- else: DFS: Max Depth

"Parentheses"

- Stack: Valid Parentheses

"Subarray"

- Sliding window: Maximum subarray sum
- Prefix sum: Subarray sum
- Hashmap: Continuous subarray sum

Max subarray

- Greedy: Kadane's Algorithm

"X Sum"

- Two pointer: Two sum

"Max/longest sequence"

- Dynamic programming, DFS: Longest increasing subsequence
- mono deque: Sliding window maximum

"Minimum/Shortest"

- Dynamic programming, DFS: Minimal path sum
- BFS: Shortest path

"Partition/split … array/string"

- DFS: Decode ways

"Subsequence"

- Dynamic programming, DFS: Longest increasing subsequence
- Sliding window: Longest increasing subsequence

"Matrix"

- BFS, DFS: Flood fill, Islands
- Dynamic programming: Maximal square

“Jump”

- Greedy/DP: Jump game

“Game”

- Dynamic programming: Divisor game, Stone game

“Connected component”, “Cut/remove” “Regions/groups/connections”

- Union Find: Number of connected components, Redundant connections

Transitive relationship

- If the items are related to one another and the relationship is transitive, then chances are we can build a graph and use BFS or Union Find.
  - string converting to another, BFS: Word Ladder
  - string converting to another, BFS, Union Find: Sentence Similarity
  - numbers having divisional relationship, BFS, Union Find: Evaluate Division

“Interval”

- Greedy: sort by start/end time and then go through sorted intervals Interval Pattern

# 32 Problems & Explanations

## 32.1 Array

### 32.1.1 Two Sum

Difficulty: Leetcode Easy

(Blind 75)

Classic problem

- Naive Solution: Brute force approach - O(n^2)
- Optimization: Use hash table

Approach 1: Brute Force

Approach 2: Hash Table

High-level idea: use the hash table to find whether the number needed to meet the target with the current number is already in the array (that we have iterated through so far).

Example 1 - Two-pass hash table

Example 2 - One-pass hash table

### 32.1.2 Contains Duplicate

Difficulty: Leetcode Easy

(Blind 75)

Approaches:

- Brute force: compare every pair of numbers in the array
- Sort the array, then check for consecutive duplicates
- Hashing
  - Build hash set
  - Build hash table, find duplicates of the same key
  - The above two methods have same time complexity

### 32.1.3 Maximum Subarray

Difficulty: Leetcode Medium

(Blind 75)

At a glance:

- Notice that this is an optimization problem: Finding the sum of the **maximum** subarray
- Notice how this could be split into overlapping subproblems

Clarification: Note that a subarray must be a contiguous part of the array, unlike the longest common subsequence problem (where characters can be deleted).

Approach:

- Dynamic programming
  - Use a 1D DP table, at each index store the sum of max subarray up until that index in the array
    - At each index, add current number to the sum
      - If current sum $>$ max sum so far, update max sum
      - If the current sum becomes negative, then reset to 0 to start a new subarray
      - This is because you always want to start a subarray at a positive number to maximize the sum

### 32.1.4 3 Sum

Difficulty: Leetcode Medium

(Blind 75)

Concepts: Array, Two Pointers, Sorting

Approaches:

- Naive approach: Brute force by testing all triplets in the array
- Set + Two Pointer Approach
  - Sort the array
  - Create a set from the array to *store* unique triplets found that sum to 0
  - Iterate through the array with index $i$ from 0
    - Initialize two pointers, one pointer $j$ at $i + 1$ and the other $k$ at the end of the array
    - While

        - check sum of $i, j, k$ is 0. If it is add the triplet to the set, increment $j$ and decrement $k$
        - if the sum is <0, increment $j$. if the sum is >0, decrement $k$
        - *these work because the array is sorted
    - Turn the set to a vector for final output
- Hash map approach
    - Hash indices of all elements into a hash map

## 32.2 Binary / Bit Manipulation

### 32.2.1 Sum of Two Integers

### 32.2.2 Number of 1 Bits

### 32.2.3 Counting Bits

### 32.2.4 Missing Number

### 32.2.5 Reverse Bits

## 32.3 Dynamic Programming

### 32.3.1 Coin Change

Difficulty: Leetcode Medium

(Blind 75)

Leetcode

Solution Reference

Problem: Given integer array `coins` (which contain coins of different denominations) and integer `amount` of money, return the **fewest number of coins needed to make up the amount.** If the amount cannot be made up, return -1. Assume infinite amount of each kind of coin.

At first glance:

- Notice that this is an optimization problem: finding the **fewest** number of coins
- Then, notice how this could be split into subproblems: finding the fewest number of coins needed to make up amount $m$, for an amount $\leq$ `amount` (target amount)

- Notice how this is very similar to 0-1 knapsack problem.
- Strategy: Dynamic programming, memoize subproblems. A first solution could use a 2D DP table, then an optimization could make this 1D to save memory

Approach:

- Since we want to find the *fewest*, we know we can initialize the memo with `INT_MAX` (doesn't necessarily need to be implemented this way, but easy to think about)
- Build a 1D memo, where `memo[i]` indicates the fewest number of coin needed to make up amount $i$
- Key insight: to build up amount $i$ with minimum number of coin, we can look at $memo[i - coin[j]]$ for each coin $j$
  - This tells us the fewest number of coins needed to build up an amount before we take this coin, and that taking this coin can build up to this amount
  - We take the minimum of $memo[i - coin[j]] + 1$, where $+1$ is to include taking this current coin denom. Note that we need to prevent out of bounds access for `coin[j]` incase it is too large

Example Walkthrough:

Assume coins `[1, 2, 5]` and `amount = 6`

1. Initialize memo: `{0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX}`
2. At index `i=1`, we try to build up amount 1 with all the coins we have
   1. Try using coin of val 1 (coin[0]). To build up `i-coin[0]` (amount 0), we need 0 coins.
   2. No other coin values are in bounds, so use coin[0], adding a 1 to this amount
   3. `{0, 1, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX}`
3. At index `i=2`, build up amount 2 with all the coins we have
   1. Try using coin[0]. To build up amount `2-coin[0]` (amount 1), we needed 1 coin. Thus to make up 2 with a value 1 coin, we use this coin and add 1
   2. Try using coin[1]. To build up amount `2-coin[1]` (amount 0), we needed 0 coins. Thus to make up 2 with a value 2 coin, we use this coin and add 1
   3. We cannot coin[2].
   4. Minimum of the coins is to use 1 coin. Thus 1 is the value for this amount
   5. `{0, 1, 1, INT_MAX, INT_MAX, INT_MAX, INT_MAX}`
4. At index `i=3`
   1. `{0, 1, 1, 2, INT_MAX, INT_MAX, INT_MAX}`
5. At index `i=4`
   1. `{0, 1, 1, 2, 2, INT_MAX, INT_MAX}`

6. {0, 1, 1, 2, 2, 1, INT_MAX}
7. {0, 1, 1, 2, 2, 1, 2}

If the final cell has INT_MAX, it means it cannot be made up with these coins, and we should instead return -1 per the problem spec.

C++ Implementation:

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        // Create DP table
        int memo[amount + 1];
        memo[0] = 0; // initialize first val to 0 since 0 coins make up amount 0

        // Sort the coins so we can access them from small to large values
        sort(begin(coins), end(coins));

        // Walk through the DP table, for each amount i
        for (int i = 1; i < amount + 1; i++) {
            memo[i] = INT_MAX;  // default value, since we want to find min val, we start wi
            for (int c : coins) {
                if (i - c < 0) break;       // if this coin is too large to make up current a

                if (memo[i - c] != INT_MAX) // Only use previously used cell
                    memo[i] = min(memo[i], 1 + memo[i - c]);
                    // Choose the minimum between using the current coin c, otherwise keep I
            }
        }

        // Return -1 if amount cannot be made up
        return memo[amount] == INT_MAX ? -1 : memo[amount];
    }
};
```

### 32.3.2 Longest Common Subsequence

Difficulty: Leetcode Medium

(Blind 75)

Leetcode

Leetcode Solution Reference

**Subsequence**: a string that can be generated from an original string with some (or no) characters deleted without changing the relative ordering of the remaining characters

Problem: Given two strings, return the length of the **longest common subsequence**

At first glace:

- Notice that this is an optimization problem: finding the **longest**
- The problem can be divided into overlapping subproblems -> use dynamic programming

Approach:

- Define the DP table, use a 2D table where each dimension is the length of each string (text1.size() by text2.size()):
  - At each cell, store the longest common subsequence up to that position/index of each string
    - e.g. `memo[i][j]` represents the longest common subsequence length of subtrings `text1[0:i-1]` and `text2[0:j-1]`
  - Fill the table in a bottom-up approach
    - Recurrence Relation:
      - if character at `[i-1]` and `[j-1]` match, add 1 to the length `memo[i][j] = memo[i-1][j-1] + 1`
      - If character does not match, use longest length from either substring's previous char: `memo[i][j] = max(memo[i-1][j], memo[i][j-1])`
- Bottom right cell of the table `memo[length1][length2]` contains the answer

Complexity: O(`length1` $\times$ `length2`) in both time and space

C++ Implementation:

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int text1Length = text1.size();
        int text2Length = text2.size();

        // Create 2D DP table to store lengths of common subsequence at each index
        int memo[text1Length + 1][text2Length + 1];

        // Initialize the 2D array with zeros
        memset(memo, 0, sizeof memo);

        // Loop through both strings and fill the dp array.
```

```
        for (int i = 1; i <= text1Length; ++i) {
            for (int j = 1; j <= text2Length; ++j) {
                // If current characters match, add 1 to the length of the sequence
                if (text1[i - 1] == text2[j - 1]) {
                    memo[i][j] = memo[i - 1][j - 1] + 1;
                } else {
                    // If current characters do not match, take the max len
                    // of either skipping the current char of either strings
                    memo[i][j] = max(memo[i - 1][j], memo[i][j - 1]);
                }
            }
        }

        // bottom-right cell which contains answer
        return memo[text1Length][text2Length];
    }
};
```

## 32.4 Graph

### 32.4.1 Course Schedule

Concepts: DFS, BFS, Graph, Topological Sort

### 32.4.2 Number of Islands

Approach:

- Starting from land, all cells of the island can be found by doing DFS / BFS

### 32.4.3 Longest Consecutive Sequence

## 32.5 Interval

### 32.5.1 Merge Intervals

## 32.6 Linked List

### 32.6.1 Reverse a linked list

Classic problem

### 32.6.2 Merge K Sorted Lists

## 32.7 Matrix

## 32.8 String

### 32.8.1 Longest Substring without Repeating Characters

### 32.8.2 Valid Palindromes

Classic

### 32.8.3 Valid Parentheses

Classic

## 32.9 Tree

## 32.10 Heap

### 32.10.1 Merge K Sorted Lists

### 32.10.2 Top K Frequent Elements

### 32.10.3 Find Median from Data Stream