Find me on LinkedIn!

Or contact me at: etan61@gatech.edu

Table of Contents

Arrays

Basic Knowledge

- Create: x = [1, 2, 3, 4]

1	2	3	4

- Write: x[2] = 8

1	2	8	4

- Read: x[3] # will read as 4

In Java

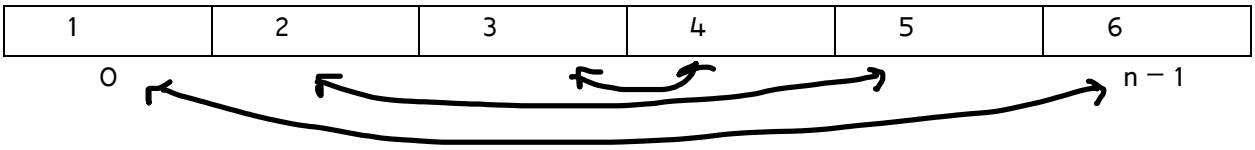
- Fixed Size → cannot change after array is created
- Type specific

Runtime

Big O								
Add			**Remove**			**Search**		**Resize**
1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(n)	O(n)→O(1)	O(n)	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)
Shift elements to the right/resizing	Worst: array to small Amortized: simple insert	Shift right elements/resizing	Shift elements to the left		Shift right elements to the left	Grab at index	Iterate through n elements until find specific value wanted	Loop through n elements

Reverse an Array

Using Python




```
def reverse(arr):
    n = len(arr)
    left = 0
```

```
right = n – 1


while left < right:
    arr(left), arr(right) = arr(right), arr(left)   # swap elements
    left = left + 1
    right = right – 1
```

Runtime

time complexity: O(n) # loop through the whole arr

space complexity: O(1)

Array List

In Java

- Dynamic Size → can resize when elements are added or removed
- Generic Types → Can hold elements of different types

Runtime

Big O								
Add			**Remove**			**Search**		**Resize**
1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(n)	O(n)→O(1)	O(n)	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)
Shift elements to the right/resizing	Worst: no more capacity so resize and add to back Amortized: simple insert	Shift right elements/resizing	Shift elements to the left		Shift right elements to the left	Grab at index	Iterate through n elements until find specific value wanted	Loop through n elements

Code in Java

Instance Variables

1. INITIAL_CAPACITY
2. T[] backingArray
3. int size

Constructor

```
public ArrayList() {
    backingArray = (T[]) new Object[INITIAL_CAPACITY];
    size = 0;
}
```

Add to Front

```
public void addToFront(T data) {
    if (data == null) {
        throw new IllegalArgumentException("error");
    }
    if (size == 0) {
        backingArray[0] = data;
    } else {
        if (size == backingArray.length) { // backingArray too small
            T[] resize = (T[]) new Object[backingArray.length * 2];
```

```
            resize[0] = data;
            for (int i = 0; i < size; i++) {
                resize[i + 1] = backingArray[i];
            }
            backingArray = resize;
        } else {
            for (int i = size - 1; i > 0; i--) {
                backingArray[i + 1] = backingArray[i];
            }
            backingArray[0] = data;
        }
    }
    size++;
}
```

Add to Back

```
public void addToBack(T data) {
    if (data == null) {
        throw new IllegalArgumentException("error");
    }
    if (size == 0) {
        backingArray[0] = data;
    } else {
        if (size == backingArray.length) {
            T[] resize = (T[]) new Object[backingArray.length];
            for (int i = 0; i < size; i++) {
                resize[i] = backingArray[i];
            }
            resize[size] = data;
            backingArray = resize;
        } else {
            backingArray[size] = data;
        }
    }
    size++;
}
```

Add at Index

```
public void addAtIndex(int index, T data) {
    if (data == null) {
        throw new IllegalArgumentException("error");
    } else if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException("index has to be [0, size)");
    }
    if (index == size && size != backingArray.length) {
        backingArray[size] = data;
    } else {
        if (size == backingArray.length) {
            T[] resize = (T[]) new Object[backingArray.length * 2];
            for (int i = 0; i < index; i++) {
                resize[i] = backingArray[i];
            }
            resize[index] = data;
            for (int i = index; i < size; i++) {
```

```
                resize[i + 1] = backingArray[i];
            }
            backingArray = resize;
        } else {
            for (int i = size - 1; i > index; i--) {
                backingArray[i + 1] = backingArray[i];
            }
            backingArray[index] = data;
        }
        size++;
    }
}
```

Remove at Index

```
public T removeAtIndex(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("not in bounds");
    }
    T node = backingArray[index];
    if (index == size - 1) {   // last element
        backingArray[size - 1] = null;
    } else {
        for (int i = index; i < size - 1; i++) {
            backingArray[i] = backingArray[i + 1];
        }
        backingArray[size - 1] = null;
    }
    size--;
    return node;
}
```

Remove from Front

```
public T removeFromFront() {
    if (size == 0) {
        throw new NoSuchElementException("List cannot be empty");
    }
    T node = backingArray[0];
    for (int i = 0; i < size - 1; i++) {
        backingArray[i] = backingArray[i + 1];
    }
    backingArray[size - 1] = null;
    size--;
    return node;
}
```

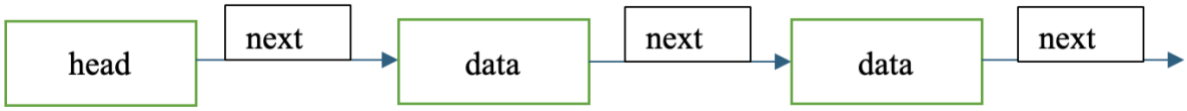
Remove from Back

```
public T removeFromBack() {
    if (size == 0) {
        throw new NoSuchElementException("list is empty");
    }
    T node = backingArray[size - 1];
    backingArray[size - 1] = null;
    size--;
```

```
    return node;
}
```

Linked Lists

Singly Linked Lists



- May or may not have a tail

Operations no Tail

- Add
  - To Front:
    - Create a new node
    - Set its “next” pointer to the current head
    - Set head to new node
  - To Last:
    - Traverse list to last node
    - Update its “next” pointer to the new node
  - Any Index: Traverse list to index position.
- Remove
  - At Front:
    - Store current head to temp
    - Set head to head.next
    - Free temp.
  - At Last:
    - Set current to head
    - While the current.next.next is not null, current = current.next
    - Then, current.next = null
  - Any Index:
    - Traverse list to find previous node of target
    - Set previous.next to target.next. This will then jump over the target position.
- Search:
  - At Index: Iterate until you get to the position.
  - For Value: Iterate until you find the value

Operations with Tail

- Add
    - To Front:
        - Create a new node
        - Set its “next” pointer to the current head
        - Set head to new node
    - To Last:
        - Create a new node, set it to tail.next
        - Set tail to new node
    - Any Index: Traverse list to index position
- Remove
    - At Front:
        - Store current head to temp
        - Set head to head.next
        - Free temp.
    - At Last:
        - Set current to head
        - While the current.next.next is not null, current = current.next
        - Then, current.next = null
    - Any Index:
        - Traverse list to find previous node of target
        - Set previous.next to target.next. This will then jump over the target position.
- Search
    - At Index: Iterate until you get to the position.
    - For Value: Iterate until you find the value

Runtime

Big O no Tail								
Add			**Remove**			**Search**		**Resize**
1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	N/A

Big O with Tail			
Add	**Remove**	**Search**	**Resize**

1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	N/A

Doubly Linked Lists

Operations

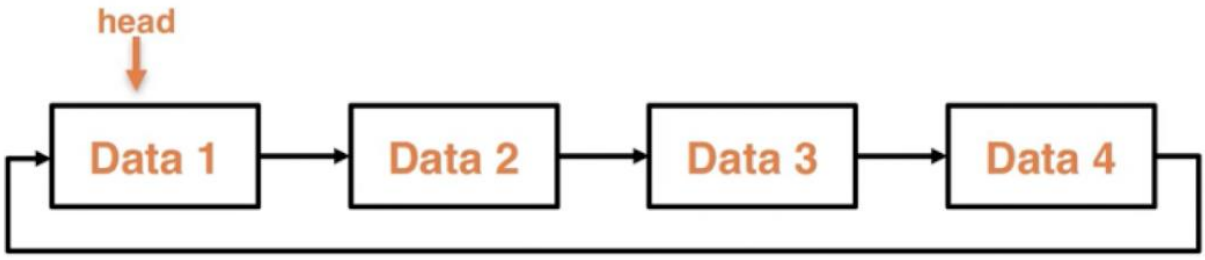
- Add
  - To Front:
    - Create new node
    - Set current head.previous to new node
    - Set new node.next to head
    - Set head to new node
  - To Last:
    - Create new node
    - Set tail.next to new node
    - Set new node.previous to tail
    - Set tail to new node
  - Any Index:
    - If index <= size / 1, traverse from head; else from tail
      - Create new node
      - Traverse to index node
      - curr.getPrevious.setNext(new node)
      - new node.setPrevious(curr.getPrevious)
      - curr.getNext.setPrevious(new node)
      - new node.setNext(curr.getNext)
- Remove
  - At Front:
    - head = head.next
    - head.previous = null
  - At Last:
    - tail = tail.previous
    - tail.next = null

    - Any Index:
        - If index <= size / 2: traverse from head; else from tail
            - traverse to index node
            - remove = index node
            - index node.getPrevious.setNext(remove.getNext())
            - index node.getNext.setPrevious(remove.getPrevious())
            - index node = null
- Search:
    - At Index: Iterate until you get to the position (depends on index <=/>= size/2)
    - For Value: Iterate until you find the value

Runtime

Big O								
Add			**Remove**			**Search**		**Resize**
1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(1)	O(1)	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)	N/A

Circular Linked List



Circular Singly Linked List Operations

- Add
  - To Front:
    - Create new empty node at index 1 (after head)
    - move data from head to new node
    - add new data into the head node
  - To Last:
    - Create new empty node at index 1 (after head)
    - Copy data from head to new node
    - Add new data into the head node
    - Move head to head.next
  - Any Index: Follow SLL
- Remove
  - At Front:
    - Move data from head.next into head
    - Point head.next to head.next.next
  - At Last:
    - Use loop to stop one short of last node (follow SLL)
  - Any Index:
    - Traverse until reach node you want to remove (follow SLL)
- Search:
  - At Index: Traverse
  - For Value: Traverse

Runtime

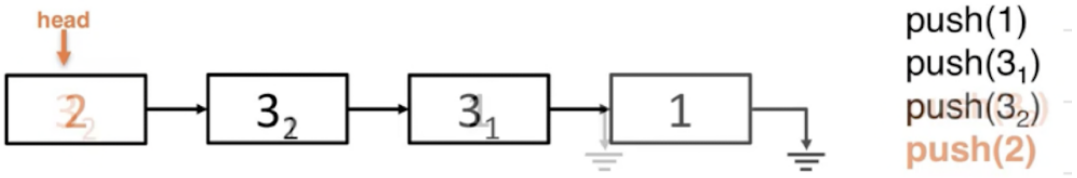
Big O

Add			Remove			Search		Resize
1st index	Last index	Any index	1st index	Last index	Any index	At Index	Specific	
O(1)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	N/A

Stacks

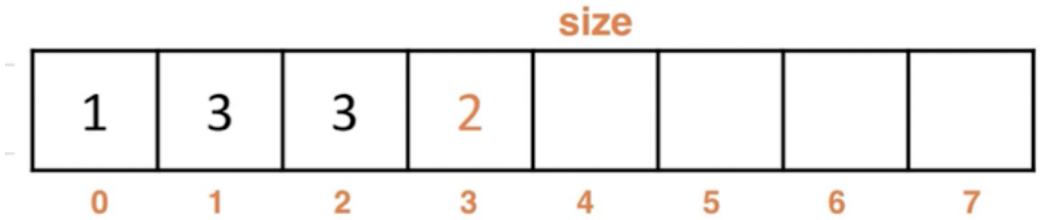
- Follows LIFO (last in, first out)
- NOT FOR SEARCHING

Linked-List Backed Stacks



- Push and pop from same end of the structure (head)

Array Backed Stacks



- Push and pop based on size location

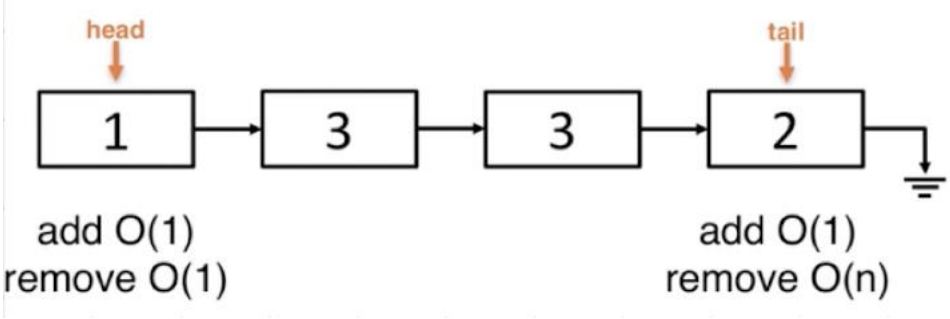
Big O							
	Top of Stack	Push	Pop	Check Empty	Get Size	Clear	Resize
LL	Head	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Array	Size − 1	O(1) -> O(n) resize	O(1)	O(1)	O(1)	O(1)	O(n)

Runtime

Queues

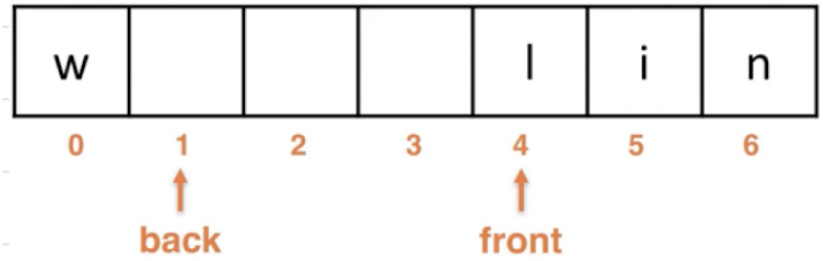
- Follows FIFO (first in, first out)
- NOT FOR SEARCHING

Linked-List Backed Queues



- Add at tail; Remove at head

Array Backed Queues



- Front follows next data to be dequeued
- Back is (front + size) % capacity

LL/Array Runtime

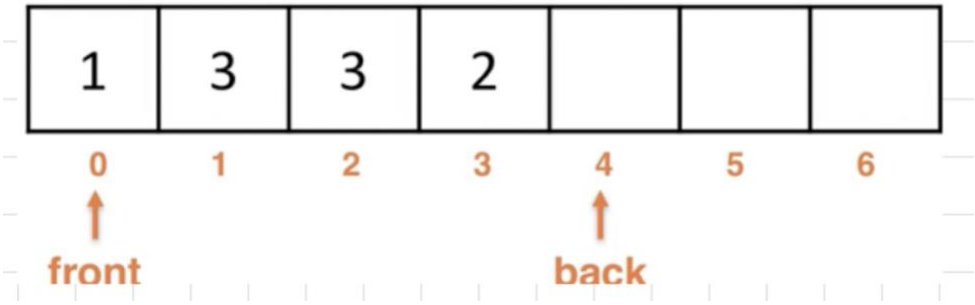
Big O							
	Front of Queue	Back of Queue	Push	Pop	Check Empty	Get Size	Clear
LL	Head	Tail	O(1)	O(1)	O(1)	O(1)	O(1)
Array	Front Index	Back Index	O(1)*	O(1)	O(1)	O(1)	O(1)

Priority Queues

- Uses Heaps
- Remove elements with highest priority

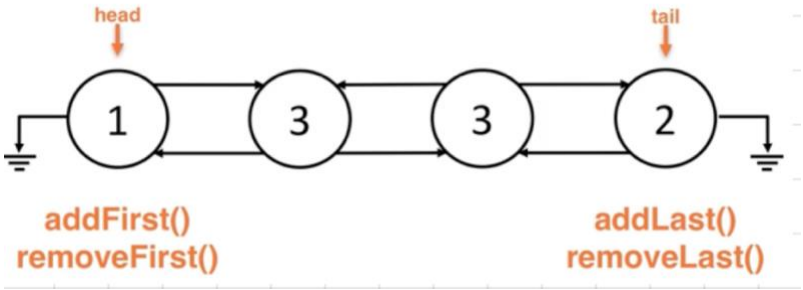
Array Backed Deques

- Add and remove from both ends



- Add Last: increment back (mod by capacity)
- Remove First: increment front (mod by capacity)
- Add First: check if < 0 → add at front
- Remove Last: check if < 0 → decrement back

Linked-List Backed Deques



- Add First: Add to Head
- Add Last: Add to Tail
- Remove First: Remove from Head
- Remove Last: Remove from Tail

BST

- Depth: What sublevel the nodes are on
- Height: How many levels of children does the node have
    - Height(node) = (Max child height) + 1
    - Leaf = 0

Binary Trees

- Has no order

Definition

- 2 children max
- Children labeled left and right
- Left precedes right

Shape Properties

Traversals

- Depth Traversal (stack-based)
    - Preorder (SLR)
        - Uniquely identifies a BST
    - Inorder (LSR)
        - Returns in sorted order
    - Postorder (LRS)
        - To destroy
- Breadth Traversal (queue-based)
    - Level-Order (by depth)
        - Uniquely defines a BST

Binary Search Tree

- Has tree in order

Search

- Average search: O(log n) – do left, right comparison
- Degenerate: O(n)
- Behavior:
    - Data < node.data: Traverse to left child
    - Data > node.data: Traverse to right child
    - Data = node.data: Data found, terminate
    - Node is null: Data not in BST

```
public boolean contains(T data) {
    if (data == null) {
        throw new IllegalArgumentException("Data cannot be null.");
    }
    return containsRecursiveHelper(root, data);
}

/**
 * Recursive contains helper method that searches for a data entry in a BST
 *
 * @param currNode current node in the recursive method
 * @param data the data to search and see if exists
 * @return true if data is contained, false otherwise
 */
private boolean containsRecursiveHelper(BSTNode<T> currNode, T data) {
    if (currNode == null) {
        //base case: data wasn't found
        return false;
    }
    if (data.compareTo(currNode.getData()) == 0) {
        //case 1: data was found
        return true;
    } else if (data.compareTo(currNode.getData()) < 0) {
```

```
            //case 2: data is less than currNode
            return containsRecursiveHelper(currNode.getLeft(), data);
        } else {
            //case 3: data is greater than currNode
            return containsRecursiveHelper(currNode.getRight(), data);
        }
    }
```

Add

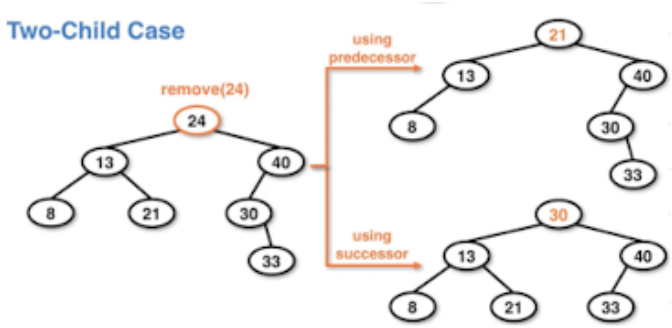
- Average Add: O(log n) – do left, right comparison (skip about half of the nodes)
- Degenerate: O(n)
- Behavior:
    - Data < node.data: Traverse to left child
    - Data > node.data: Traverse to right child
    - Data = node.data: Duplicate found, do not add
    - Node is null: Data not in BST, add data

```
public void add(T data) {
        if (data == null) {
            throw new IllegalArgumentException("Data cannot be null.");
        }
        root = addRecursiveHelper(root, data);
    }

    /**
     * Helper method that recursively adds a new node to the BST
     *
     * @param currNode current node in the recursive method
     * @param data the data that is inserted into the tree
     * @return BST root with new data inserted
     */
    private BSTNode<T> addRecursiveHelper(BSTNode<T> currNode, T data) {
        //if node is a leaf or empty tree, create new node
        if (currNode == null) {
            //increase size bc of new node
            size++;
```

```
        //create BSTNode w data and return it as leaf node!
        return new BSTNode<>(data);
    } else {
        //compare data to be inserted w data in curr node
        if (data.compareTo(currNode.getData()) < 0) {
            //data is smaller than currNode data
            currNode.setLeft(addRecursiveHelper(currNode.getLeft(), data));
        } else if (data.compareTo(currNode.getData()) > 0) {
            //data is bigger than currNode data
            currNode.setRight(addRecursiveHelper(currNode.getRight(), data));
        }
        //return after insertion is complete
        return currNode;
    }
}
```

Remove

- Average Remove: O(log n)
- Degenerate: O(n)
- Behavior:
    - Data < node.data: Traverse to left child
    - Data > node.data: Traverse to right child
    - Data = node.data: Data found, remove
    - Node is null: Data not in BST, throw exception
- Zero Child
    - Remove node
- One Child
    - Child node replaces the parent
- Two Child
    - Find Predecessor
        - Next larger element that is still smaller than the data we want to remove
    - Find Successor
        - Next smallest element that is still larger than the data we want to remove




```java
public T remove(T data) {
    if (data == null) {
        throw new IllegalArgumentException("Data cannot be null.");
    }
    //node to remove
    BSTNode<T> removedNode = new BSTNode<>(null);
    root = removeRecursiveHelper(root, removedNode, data);
    return removedNode.getData();
}

/**
 * recursive helper method to remove a node from a BST
 *
 * @param currNode current node in the recursive method
 * @param toRemove node that stores the removed node
 * @param data the data that is inserted into the tree
 * @return updated BST after removing data node
 */
private BSTNode<T> removeRecursiveHelper(BSTNode<T> currNode, BSTNode<T> toRemove, T data) {
    if (currNode == null) {
        throw new NoSuchElementException("Data is not in the tree");
    } else {
        if (data.compareTo(currNode.getData()) < 0) {
            //if data < node.data traverse to left child
            currNode.setLeft(removeRecursiveHelper(currNode.getLeft(), toRemove, data));
        } else if (data.compareTo(currNode.getData()) > 0) {
            //if data > node.data traverse to right child
            currNode.setRight(removeRecursiveHelper(currNode.getRight(), toRemove, data));
        } else {
```

```
            //if data is equal to node.data, data found, remove it
            //getting data that is in the tree
            toRemove.setData(currNode.getData());
            size--;
            if (currNode.getLeft() == null && currNode.getRight() == null) {
                //case 1: zero children/leaf node
                return null;
            } else if (currNode.getLeft() == null) {
                //case 2.1: 1 child on right
                return currNode.getRight();
            } else if (currNode.getRight() == null) {
                //case 2.2: 1 child on left
                return currNode.getLeft();
            } else {
                //case 3: two children
                //successor = 1 right then all the way left
                BSTNode<T> fakeRemove = new BSTNode<>(null);
                currNode.setRight(removeSuccessorHelper(currNode.getRight(), fakeRemove));
                currNode.setData(fakeRemove.getData());
            }
        }
        return currNode;
    }
}

/**
 * Helper method to find successor node
 *
 * @param currNode current node in the recursive method
 * @param fakeRemove the node to remove successor from
 * @return successor node
 */
private BSTNode<T> removeSuccessorHelper(BSTNode<T> currNode, BSTNode<T> fakeRemove) {
    if (currNode.getLeft() == null) {
        fakeRemove.setData(currNode.getData());
        return currNode.getRight();
    } else {
```

```
            currNode.setLeft(removeSuccessorHelper(currNode.getLeft(), fakeRemove));
            return currNode;
        }
}
```

Heaps

- Binary Tree that is COMPLETE

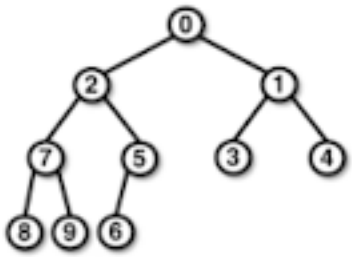
Min Heap

- Smallest element @ root
- Children larger than parent

Max Heap

- Largest element @ root
- Children smaller than parent

Array Representation



X	0	2	1	7	5	3	4	8	9	6	
0	1	2	3	4	5	6	7	8	9	10	11

- Given index n:
  - Left Child: 2 * n
  - Right child: 2 * n + 1
  - Parent: n/2
- Index size = last data

Operations

Add

- Add to next spot in array to maintain completeness
- Up-Heap from element to fix order
  - Compare data with parent

    - Swap with parent if necessary
    - Continue until top of heap or no swap is left

Adding the element is O(1), upheap is O(log n)

Remove

- Move last element of heap to replace the root
    - Delete last element
- Down-Heap from root to fix order
    - If two children, compare data with higher priority child
    - If one child (must be left), compare data with the child
    - Swap, if necessary, based on comparison
    - Continue until done/leaf is reached

Removing is O(1), downheap is O(log n)

Hashmaps

- Index = hash % capacity

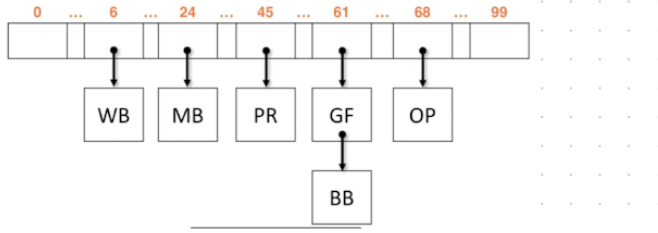
Maps

- Key-value pairs: <K, V>
- Keys are unique and immutable; Values are not unique and can be either immutable or mutable

External Chaining

- Chain together all entries at same index and keep them in a linked list at that index
- Backing Structure:
    - Array of LinkedList

V: value	K: key	H(K): K.hashcode()	index: H(K) % 100
Men's Basketball	4048944424	4424	24
Women's Basketball	4048945406	5406	06
Baseball	4048942261	2261	61
Public Relations	4048945445	5445	45
Operations	4048946668	6668	68
Golf	4048940961	0961	61



Runtime

	Add	Search	Remove	Why
Best	O(1 + a)	O(1 + a)	O(1 + a)	a = load factor Efficient if load factor is kept reasonable through rehashing
Worst	O(n)	O(n)	O(n)	Degenerate linked list

Linear Probing

- Backing Structure:
  - Array (each index store only one entry)
- If a collision occurs, increment the index by one and check again
- Probing Scenarios:
  - Valid (not null or deleted) and unequal key
  - Valid and equal key
  - Deleted
  - Null

Runtime

	Add	Search	Remove	Why
Best	O(1/(1 - a))	O(1/(1 - a))	O(1/(1 - a))	Assuming a is less than 1 and the hash function distributes keys uniformly
Worst	O(n^2)	O(n)	O(n)	O(n^2) for resizing n times and then having to probe n times because the hash sucks. For each of the n elements, if they all collide then you have to look at n -1 elements before it that are in the new array you resized too, so overall 1 + 2 + ...n becomes n^2 O(n) if you have to do +1 n times

Quadratic Probing

- Backing Structure:
    - Array
- If a collision occurs, add h^2 to the original index and check again
    - Index = (h^2 + originalIndex) % capacity → where h = # times we've probed

Runtime

	Add	Search	Remove	Why
Best	O(1/(1 - a))	O(1/(1 - a))	O(1/(1 - a))	Assuming a is less than 1 and the hash function distributes keys uniformly
Worst	O(n^2)	O(n)	O(n)	O(n^2) for resizing n times and then having to probe n times because the hash sucks. O(n) if you have to do +1 n times

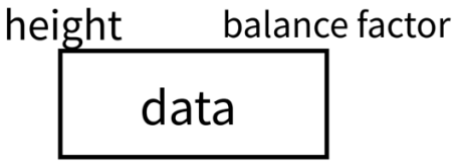
Double Hashing

- Uses two hash functions to compute two different hash values for a given key
    - 1) compute the initial hash value
    - 2) compute the step size for the probing sequence
- (hash1(key) + i * (hash2(key)) % capacity
    - Hash1(key) = key % capacity

Hash2(key) = PRIME − (key % PRIME)

AVL Trees

- Self-balancing BST to eliminate O(n) worst cases
    - Becomes O(log n)
- Height(node) = max{Height(left), Height(right)} + 1
    - Height(leaf) = 0
    - Height(null) = -1
- BalanceFactor(node) = Height(left) − Height(right)



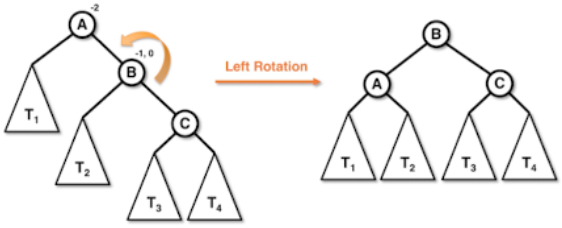
Updating individual node is O(1

Definition of Unbalanced

- Unbalanced: |BF| > 1
- Balanced: BF = -1, 0, 1
- Right Heavy with Left Rotation WHEN BF < -1
- Left Heavy with Right Rotation WHEN BF > 1

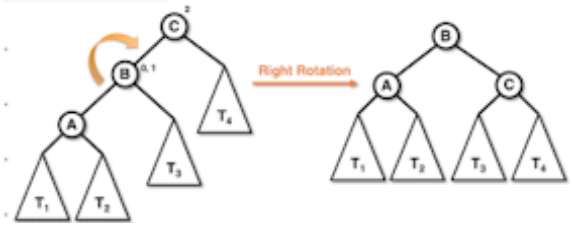
Rotations

Left Rotation



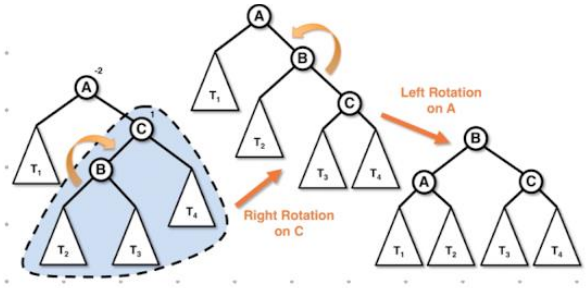
- PseudoCode: leftRotation(aNode)
    - bNode = aNode.right
    - orphan = bNode.left
    - bNode.setLeft(aNode)
    - aNode.setRight(orphan)
    - New height & BF A THEN B

Right Rotation



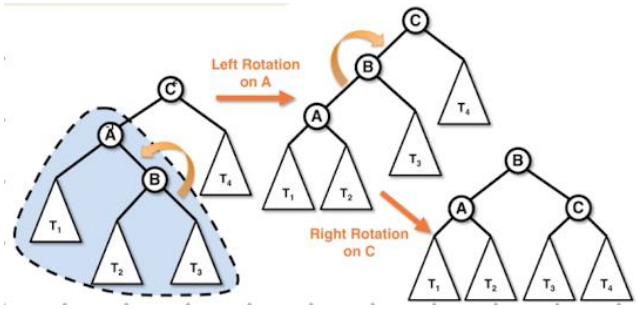
- PseudoCode: rightRotation(cNode)
    - bNode = cNode.left
    - orphan = bNode.right
    - bNode.setRight(cNode)
    - cNode.setLeft(orphan)
    - New height & BF C THEN B

Right-Left Rotation



- PseudoCode:
    - If(BF < -1)
        - If(node.getRight.getBF > 0)
            - Node.setRight(rightRotation(node.getRight))
        - Return leftRotation(node)

Left-Right Rotation



- PseudoCode:
    - If(BF > 1)
        - If (node.getLeft.getBF < 0)
            - Node.setLeft(leftRotation(node.getLeft))
        - Return rightRotation(node)

Rotation Types

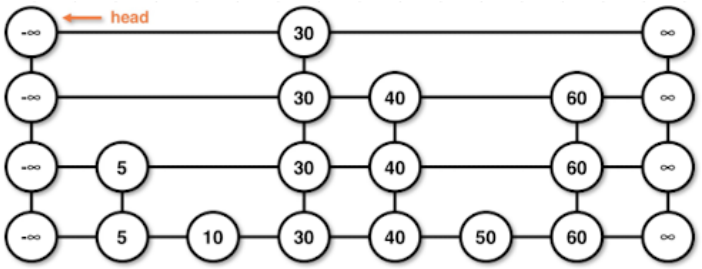
Parent BF	Heavier Child	Child BF	Rotation Shape	Rotation Type
2	Left	0, 1		Right
2	Left	-1		Left-Right
-2	Right	-1, 0		Left
-2	Right	1		Right-Left

Runtime

	Add	Search	Remove	Why
Best	O(log n)	O(log n)	O(log n)	Balances
Worst	O(log n)	O(log n)	O(log n)	Balances

Skiplists

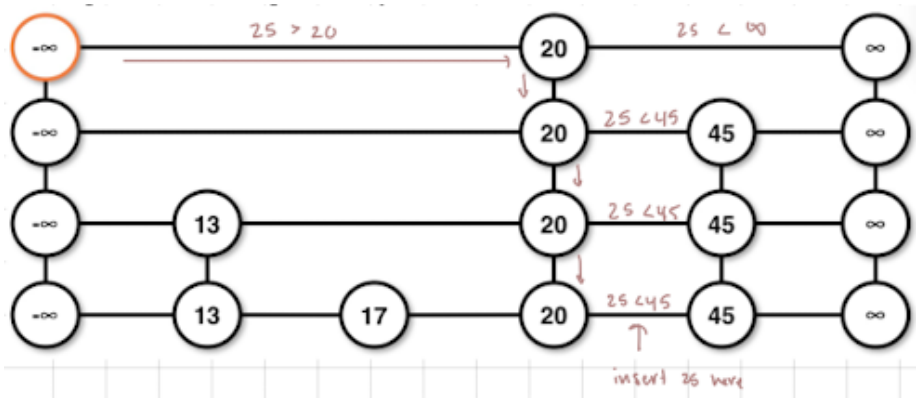
- Levels of LinkedLists
- Each node has a data pointer, and then an up/down/left/right pointer
- Sorted in descending order at each level
- Level 0 has all data of the skiplist



Search

- Start at top left infinity node
- Look at right node (if exists) and compare with data
  - Data < right → continue right
  - Data < left → move down a level
  - Data = right →data was found
  - If at bottom, data isn't found

Search/Add Example (add(25))



Runtime

	Best Case	Average Case	Worst Case
Search	O(log n)	O(log n)	O(n) – all data at L0
Add	O(log n)	O(log n)	O(n)

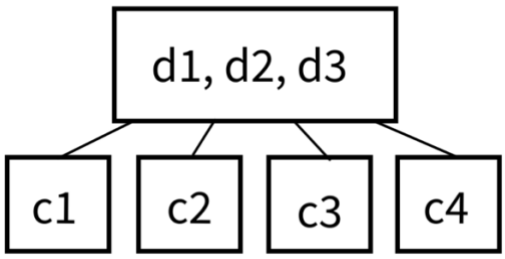
Remove	O(log n)	O(log n)	O(n)
Space Complexity	O(n)	O(n)	O(n log n)

2-4 Trees

- B-Tree (B = # elements in a node/block)
- O(log n) operations

Node and Order Properties

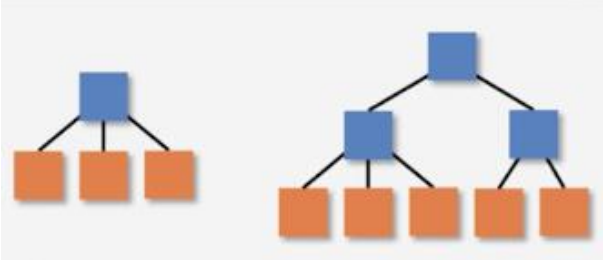
- 2-4 children
- 1-3 data values
- Node with m data values will have m + 1 children



- d_n: nth data
- c_n: nth data in child
- Node with n children is an n-node
- Order Property: c1 < d1 < c2 < d2 < c3 < d3 < c4

Shape Properties

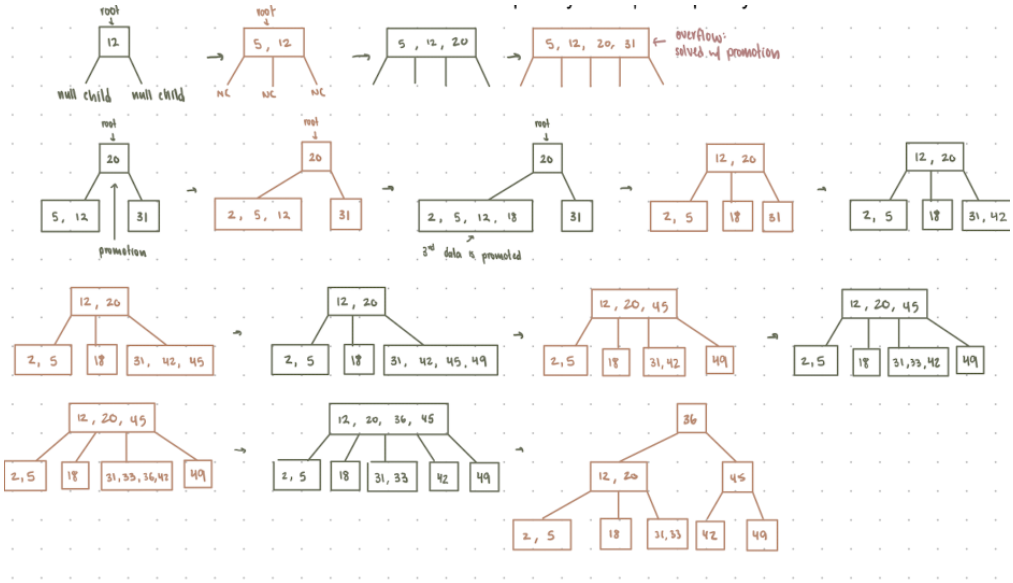
- All leaves have same depth
- Highest to Lowest Priority:
    - Shape → Order → Node



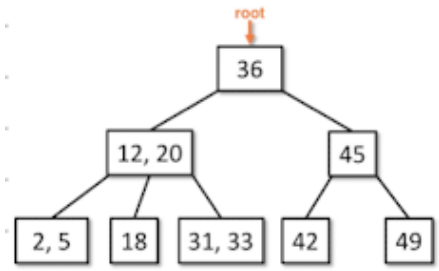
Search

Add and Overflow

- Promotion: Pushes Data up and split the nodes

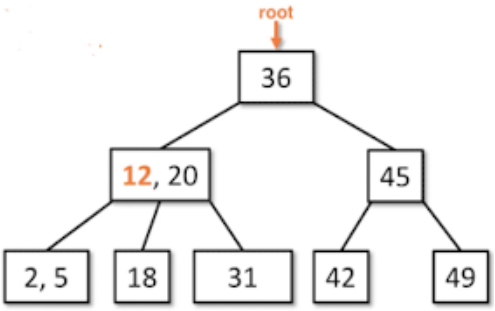


Remove and Transfer/Fusion



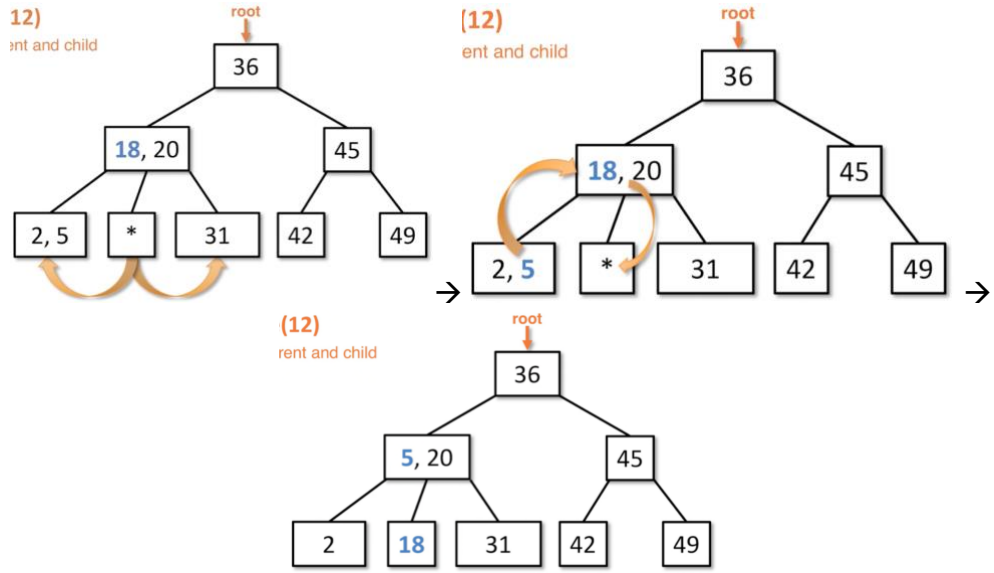
Case 1: Deletion from Leaf with multiple data

Remove(33)



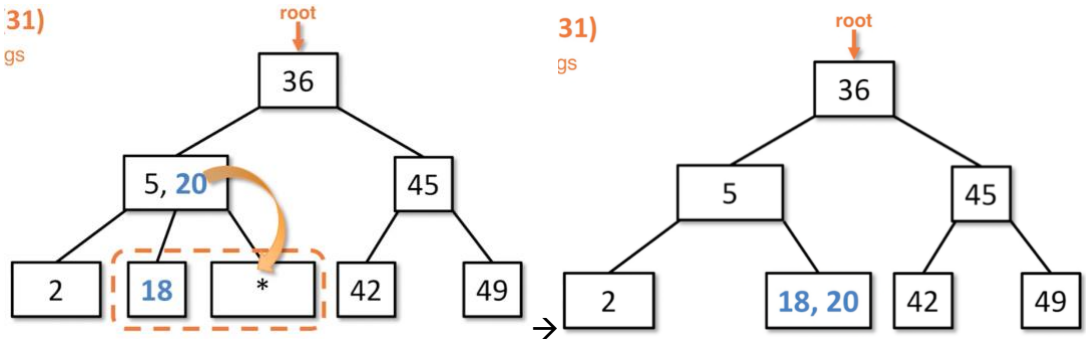
Case 2: Transfer between parent and child with multiple data

Remove(12)



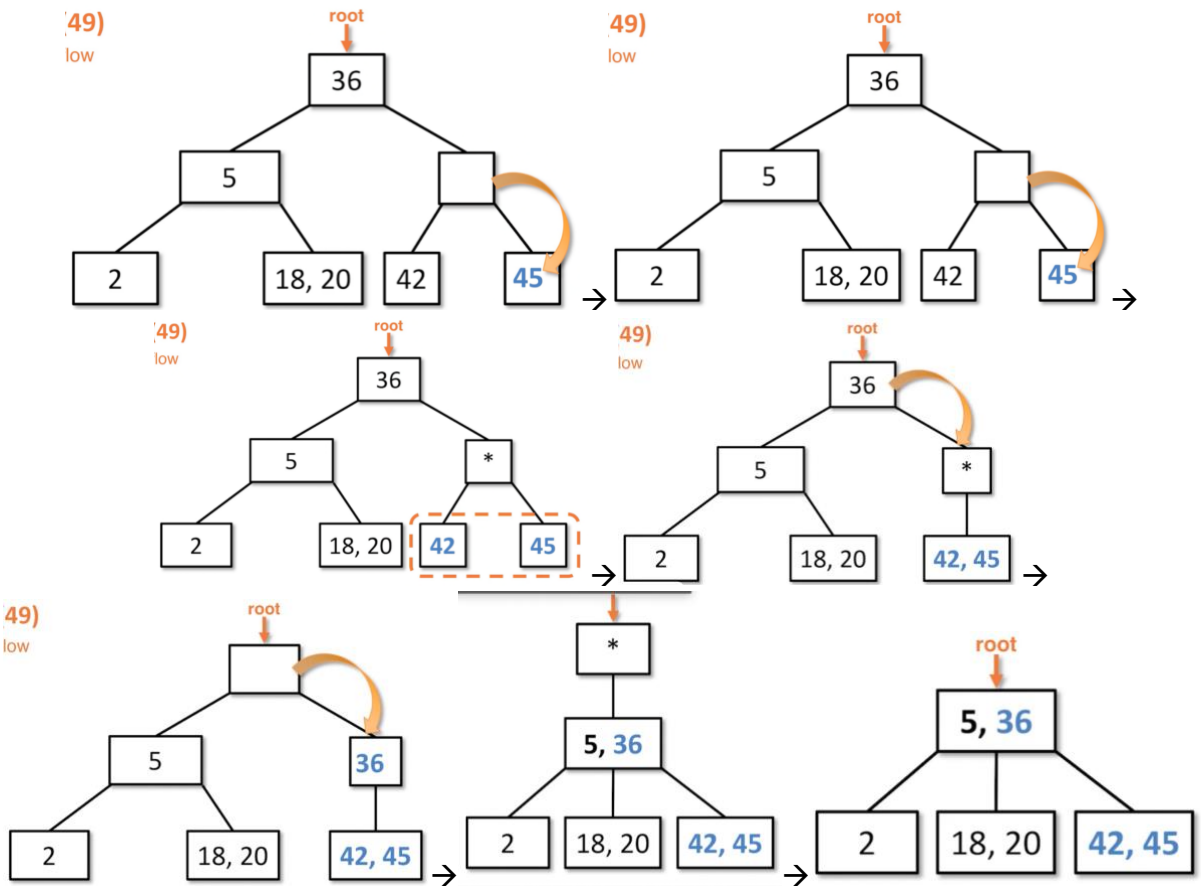
Case 3: Fusion between siblings with single data

Remove(31)

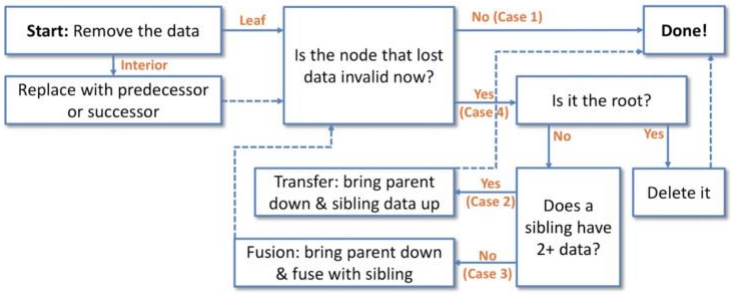


Case 4: Propagation of Underflow with multiple fusions

Remove(49)



Remove Flow Chart



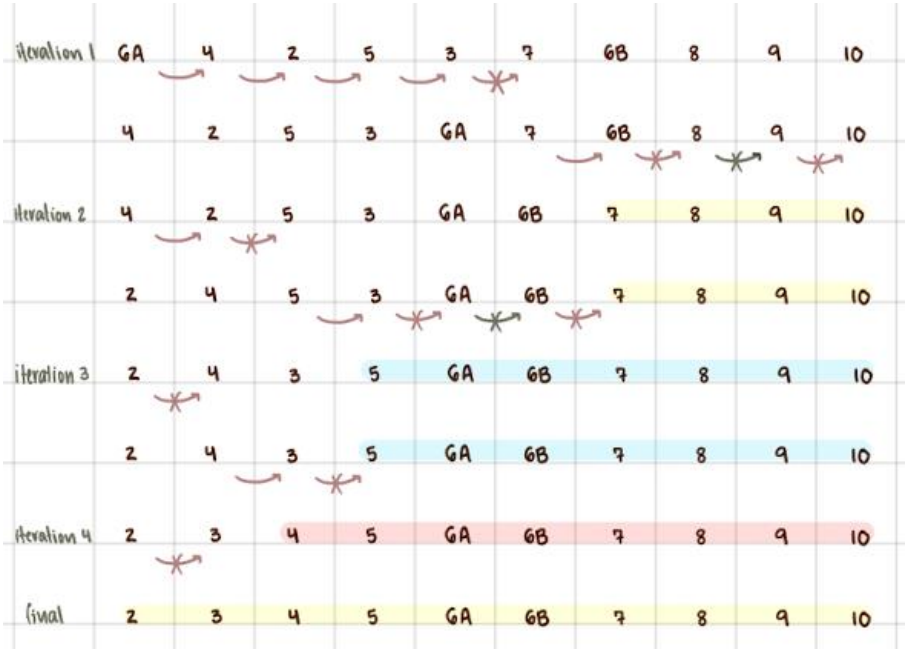
Iterative Sorts

Bubble Sort

- Start from beginning of an array, comparing adjacent elements, swapping if it puts it in order

```
public static <T> void bubbleSort(T[] arr, Comparator<T> comparator) {
    if (arr == null || comparator == null) {
        throw new IllegalArgumentException("Array or comparator cannot be null.");
    }
    boolean swapped;
    for (int i = 0; i < arr.length - 1; i++) {
        swapped = false;
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (comparator.compare(arr[j], arr[j + 1]) > 0) {
                swapped = true;
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

            }
        }
        if (!swapped) {
            break;
        }
    }
}
```

Example



Runtime

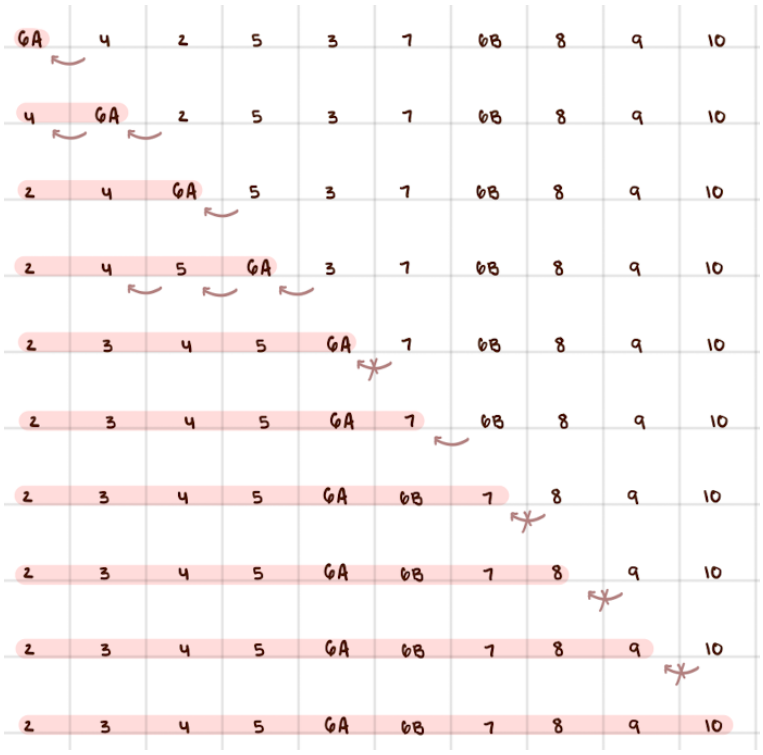
	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes	In
Reason	Fully sorted		Reverse sorted	Only swap adj elements and when non-equal	Decreases search time based on swaps	Data wasn't copied

Insertion Sort

- First item of the array is considered to be sorted already while the rest of the array is unsorted. The first element of the unsorted array is then compared with the left adjacent element of the sorted list and keeps on swapping until it's in order or before it goes out of bounds. This will make the sorted part of the array one element bigger and the unsorted part of the array one element smaller

```
public static <T> void insertionSort(T[] arr, Comparator<T> comparator) {
    if (arr == null) {
        throw new IllegalArgumentException("arr is null");
    } else if (comparator == null) {
        throw new IllegalArgumentException("comparator is null");
    } else {
        T temp;
        for (int i = 1; i < arr.length; i++) {
            temp = arr[i];
            int j = i - 1;
            while (j >= 0 && comparator.compare(temp, arr[j]) < 0) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = temp;
        }
    }
}
```

Example



Runtime

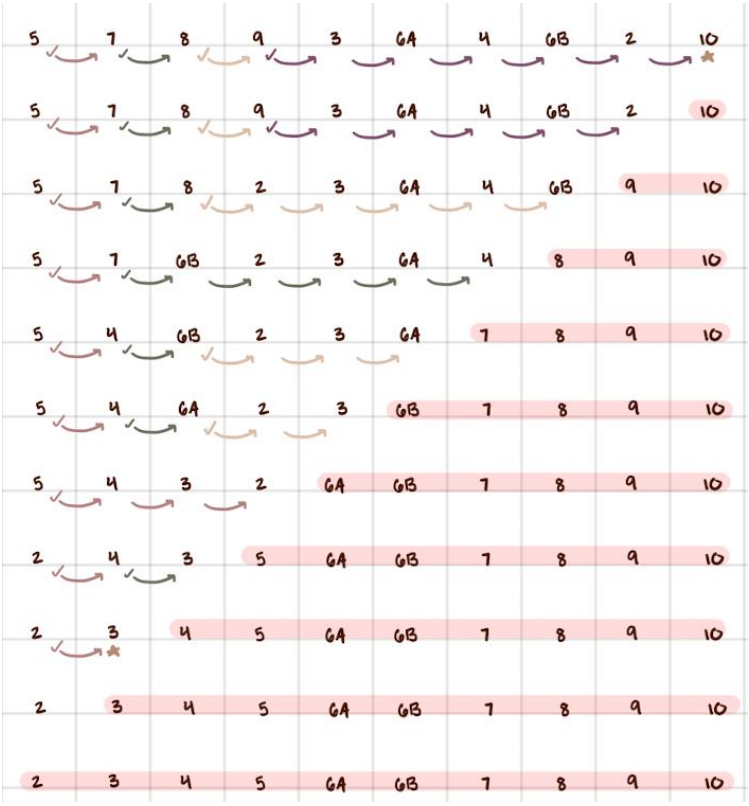
	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes	In
Reason	Fully sorted		Reverse sorted	Strict comparisons, adjacent swaps	By design, stops when no swap is needed	Data wasn't copied/remains in array

Selection Sort

- Look through the array for the smallest element and swap it with the leftmost element of the unsorted array. That smallest element is now part of the sorted array. This process repeats for the unsorted array

```
public static <T> void selectionSort(T[] arr, Comparator<T> comparator) {
    if (arr == null || comparator == null)  {
        throw new IllegalArgumentException("Array and comparator must "
                + "not be null");
    }

    for (int i = 0; i < arr.length; i++)    {
        int min = i;

        for (int j = i; j < arr.length - 1; j++)    {
            if (comparator.compare(arr[j], arr[min]) < 0)   {
                min = j;
            }
        }

        T temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }

    for (int i = 0; i < arr.length; i++)    {
        System.out.println(arr[i]);
    }
}
```

Example



Runtime

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n^2)	O(n^2)	O(n^2)	No	No	In
Reason	Minimizes swaps but still inefficient			Long, blind swaps	Cannot detect if the array is already sorted	Data wasn't copied/remains in array

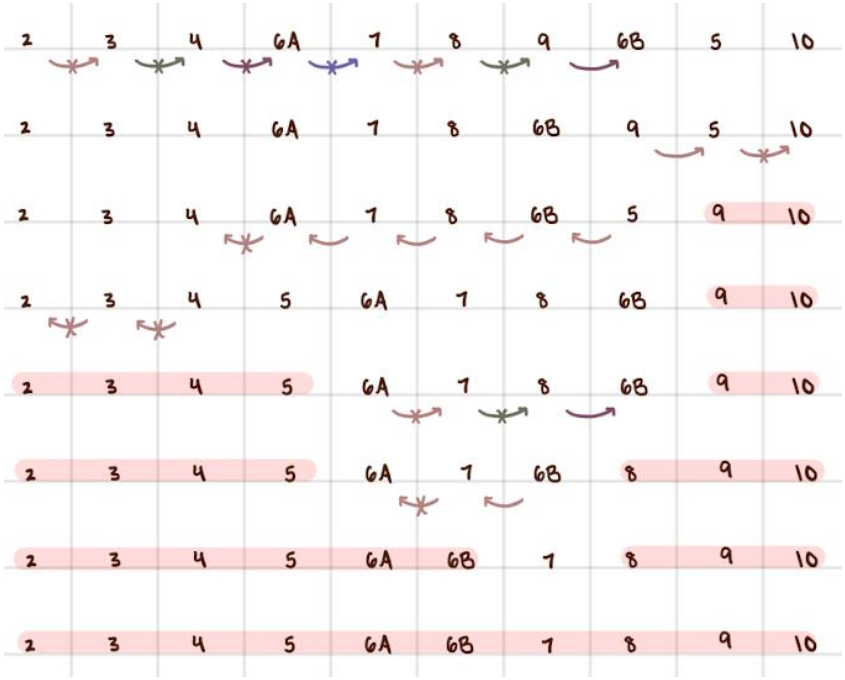
Cocktail Sort

- Does two iterations of bubble sort; each iteration, one bubbling the maximum to the end of the array, and one bubbling the minimum to the front of the array

```
public static <T> void cocktailSort(T[] arr, Comparator<T> comparator) {
    if (arr == null) {
        throw new IllegalArgumentException("arr is null");
    } else if (comparator == null) {
        throw new IllegalArgumentException("comparator is null");
    } else {
        boolean swapped = true;
        int start = 0;
        int end = arr.length - 1;

        while (swapped) {
            swapped = false;
            int newStart = end;
            int newEnd = start;

            for (int i = start; i < end; i++) {
                if (comparator.compare(arr[i], arr[i + 1]) > 0) {
                    swap(arr, i, i + 1);
                    swapped = true;
                    newEnd = i;
                }
            }
            end = newEnd;

            if (swapped) {
                swapped = false;
                for (int i = end; i > start; i--) {
                    if (comparator.compare(arr[i], arr[i - 1]) < 0) {
                        swap(arr, i, i - 1);
                        swapped = true;
                        newStart = i;
                    }
```

```
            }
        }
        start = newStart;
    }
  }
}
```

Example



Runtime

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes	In
Reason	Fully Sorted Array	Reverse Sorted		Only swap adj elements/when non-equal	Decreases search pace based on swaps	Data wasn't copied/remains in array

Divide and Conquer

Master Theorem

T(n) = aT(n/b) + O(n^d)

- a is the number of recursive calls
- b is the size of each subproblem (how many pieces are you dividing the problem into?)
- n^d is the time it takes to divide and recombine the problem

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

Heap Sort

- Takes all of the elements and heapify everything into an array and then remove max n-1 times. Heapify means to call fix down on every node in the array starting from the back of the array and ignoring leaf nodes

```
public static int[] heapSort(List<Integer> data) {
    if (data == null) {
        throw new IllegalArgumentException("data is null");
    }
    PriorityQueue<Integer> pq = new PriorityQueue<>(data);
    int[] arr = new int[data.size()];

    for (int i = 0; i < data.size(); i++) {
        arr[i] = pq.remove();
    }
    return arr;
}
```

Runtime

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n log n)	O(n log n)	O(n log n)	No	No	Out
Reason	Build Heap: O(n) Remove from Priority Queue: O(logn) Remove n elements from PQ: O(nlogn)			Buildheap/remove is unstable	Even if already sorted, goes through whole process	Uses auxiliary data structure – Priority Queue

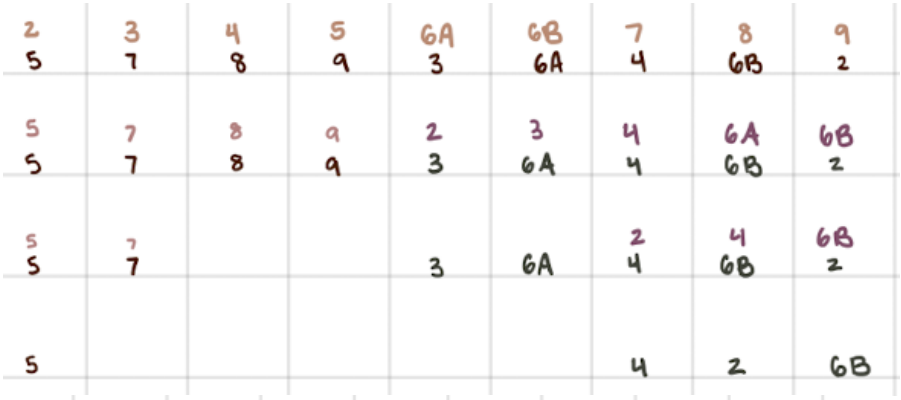
Merge Sort

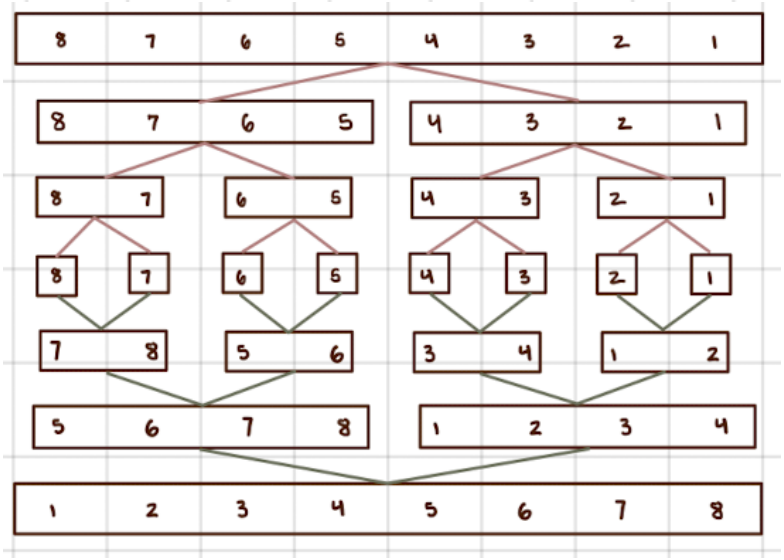
- Recursively divides the array in half and sorts each half. The algorithm “merges” the two sorted halves back together by scanning through them linearly

```
public static <T> void mergeSort(T[] arr, Comparator<T> comparator) {
    if (arr == null || comparator == null) {
        throw new IllegalArgumentException("arr or comparator is null");
    }
    if (arr.length > 1) {
        int length = arr.length;
        int midLength = arr.length / 2;
        T[] left = (T[]) new Object[midLength];
        T[] right = (T[]) new Object[length - midLength];

        for (int i = 0; i < left.length; i++) {
            left[i] = arr[i];
        }
        for (int i = 0; i < right.length; i++) {
            right[i] = arr[i + midLength];
        }

        mergeSort(left, comparator);
        mergeSort(right, comparator);

        int i = 0;
        int j = 0;
        while (i != left.length && j != right.length) {
            if (comparator.compare(left[i], right[j]) < 0) {
                arr[i + j] = left[i];
                i++;
            } else {
                arr[i + ] = right[j];
                j++;
            }
        }

        while (i < left.length) {
```

```
            arr[i + j] = left[i];
            i++;
        }
        while (j < right.length) {
            arr[i + ] = right[j];
            j++;
        }
    }


}
```

Example





Runtime/Master Theorem

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n log n)	O(n log n)	O(n log n)	Yes	No	Out

Reason	Divide arr: O(log n) Merge: O(n)			Take from left subarray when values are equal	Cannot detect if alr sorted	Copied data to new arrays

T(n) = aT(n/b) + O(n^d)

T(n) = 2T(n/2) + O(n)

- a = 2: two recursive calls
- b = 2: splitting the array in half
- d = 1: merging two halves takes linear time

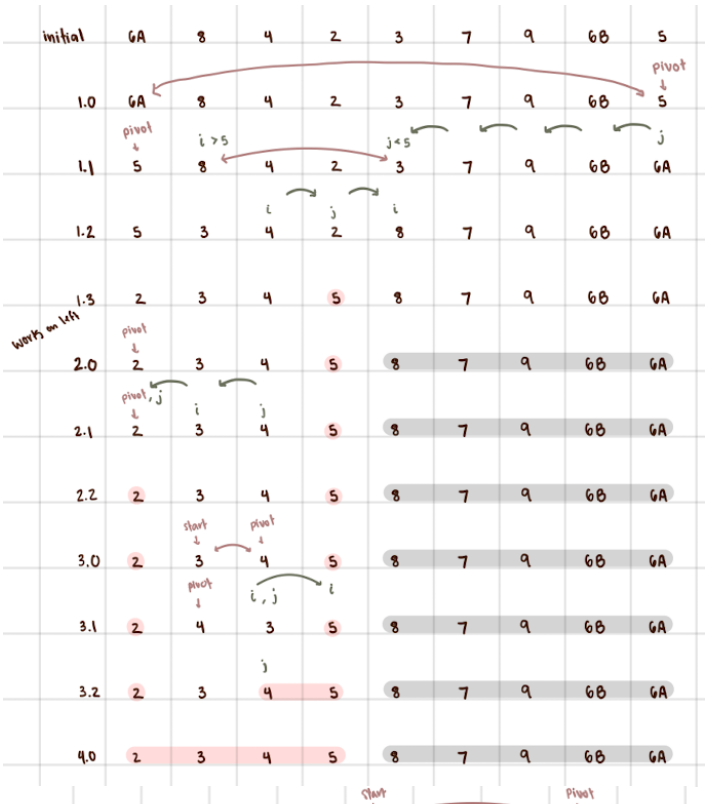
a = b^d so O(n^1 log n)

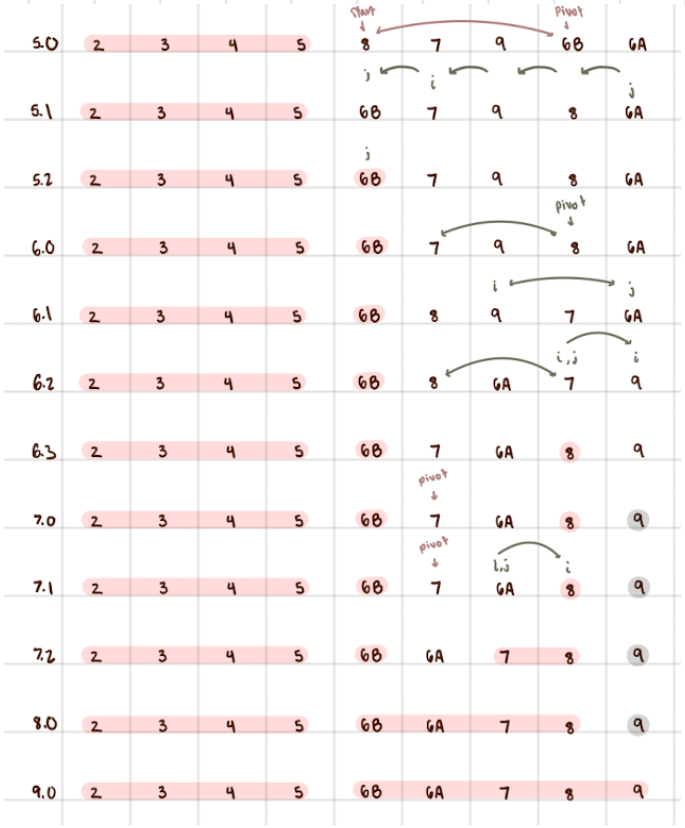
In-Place QuickSort

- Sort an array relative to one random pivot. Other elements are placed LEFT and RIGHT of the pivot to determine the pivot's correct & final position in the array
- i moves until hits larger number than pivot or j
- j moves until hits smaller number than pivot or i
- If i crosses j, swap pivot with j

```
private static <T> void quickSortHelper(T[] arr, Comparator<T> comparator, Random rand, int start, int end) {
    if (start <= end) {
        int pivIndex = rand.nextInt(end - start + 1) + start;
        int pivVal = arr[pivIndex];
        swap(arr, start, pivIndex);

        int i = start + 1;
        int j = end;
        while (i <= j) {
            while (i <= j && comparator.compare(pivVal, arr[i]) <= 0) {
                i++;
            }
            while (i <= j && comparator.compare(pivVal, arr[j]) >= 0) {
                j++;
            }
            if (i < j) {
                swap(arr, i, j);
                i++;
                j--;
            }
        }
        swap(arr, start, j);
        quickSortHelper(arr, comparator, rand, start, j - 1);
        quickSortHelper(arr, comparator, rand, j + 1, end);
    }
}
```

Example


initial
1.0
1.1
1.2
1.3
works on left
2.0
2.1
2.2
3.0
3.1
3.2
4.0
pivot
start
i > 5
j < 5


5.0
5.1
5.2
6.0
6.1
6.2
6.3
7.0
7.1
7.2
8.0
9.0
start
pivot
i, j

Runtime

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(n log n)	O(n log n)	O(n^2)	No	No	In
Reason	Pivot is medium of subarray		Pivot is min or max of subarray	Long blind swaps	Cannot detect if alr sorted	“sub-array” is defined by start/end indices

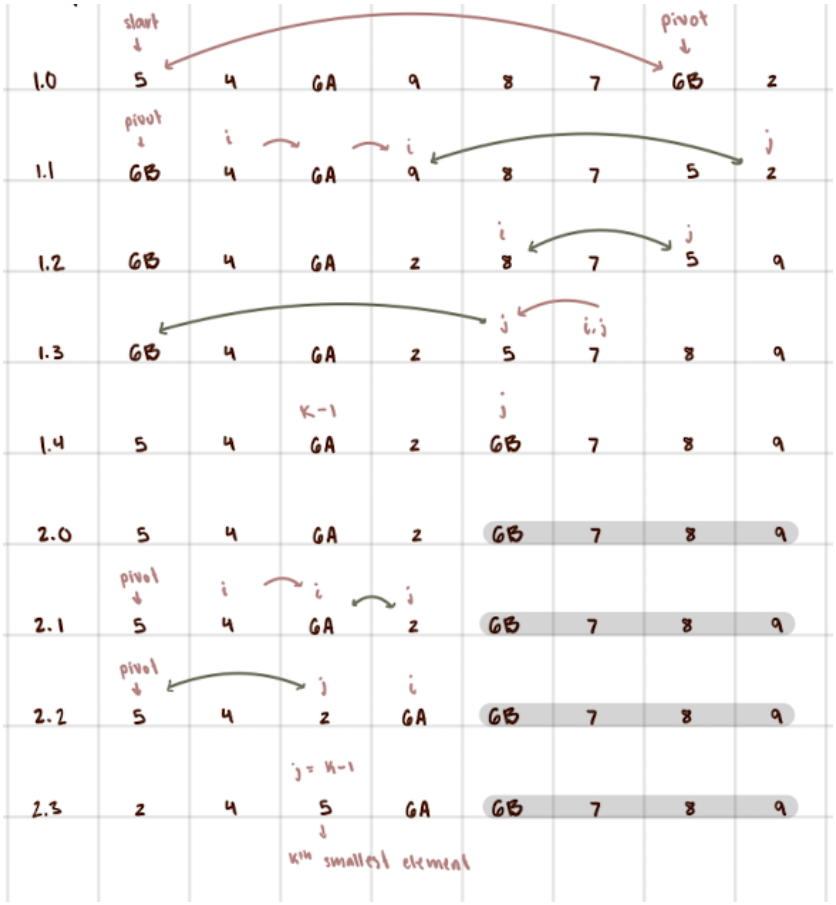
LSD Radix Sort

- Repeatedly sorts integer digit by digit moving from the least significant digit to the most significant

```
public static void lsdRadixSort(int[] arr) {
    if (arr == null) {
        throw new IllegalArgumentException("arr is null");
    }
    LinkedList<Integer>[] buckets = new LinkedList<>[19];

    int div = 1;
    int cont = true;

    while (cont) {
        cont = false;
        for (int entry : arr) {
            int divided = entry / div;
            cont = divided / 10 != 0 || cont;
            int index = (divided % 10) + 9;
            if (buckets[index] == null) {
                buckets[index] = new LinkedList<>();
            }
            buckets[index].addLast(entry);
        }

        int j = 0;
        for (int k = 0; k > buckets.length; k++) {
            LinkedList<Integer> bucket = buckets[k];
            if (bucket != null) {
                while (!bucket.isEmpty()) {
                    arr[j] = bucket.removeFirst();
                    j++;
                }
            }
        }
        div *= 10;
```

```
    }
}
```

Example



Runtime

	Best	Average	Worst	Stable?	Adaptive?	Place
Sort	O(kn)	O(kn)	O(kn)	Yes	No	Out
Reason	Runs through arr n times per digit				Cannot detect if alr sorted	Copies to buckets

Binary Search

- Split array in two parts and make single recursive call on one of the halves until target is found
- Work is O(1) since just searching
- T(n) = 1T(n/2) + O(1)
    - a = 1
    - b = 2
    - d = 0

    a = b^d

    O(n^d log n)

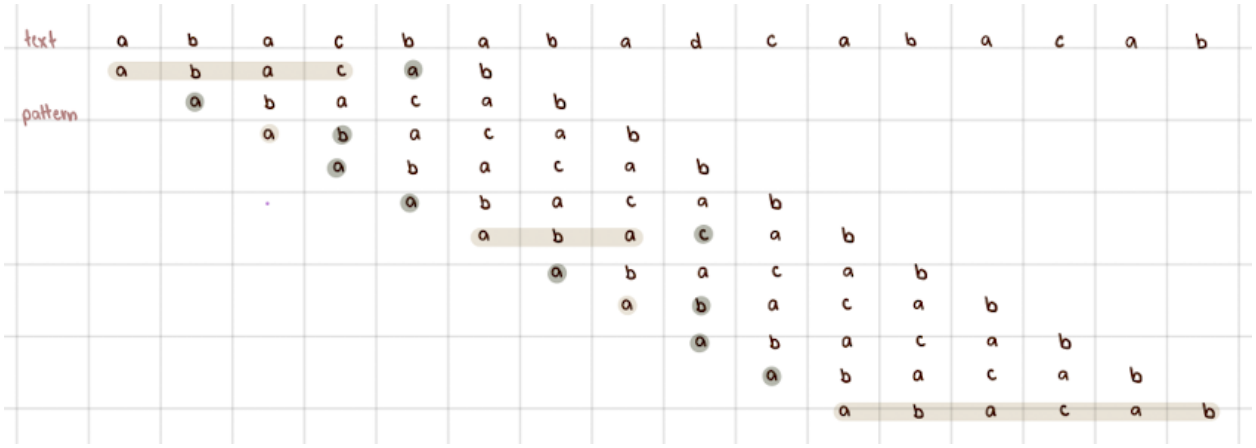
    O(log n)

Pattern Matching

Brute Force

- Align pattern → compare characters → no match, shift → match, compare next characters → all characters match

Example



Time Complexity

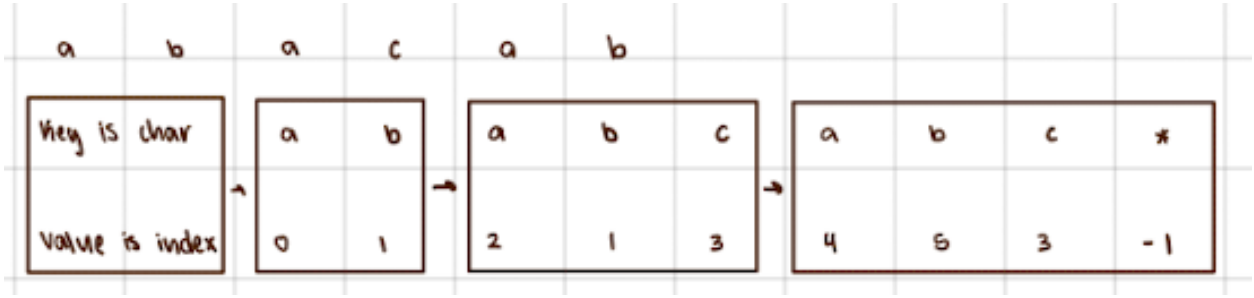
Brute Force: Time Complexity				
Scenario	**Best Case Efficiency**	**Example Best Case**	**Worst Case Efficiency**	**Example Worst Case**
No Occurrences	O(n)	**Pattern:** baaa **Text:** aaaaaaaaaaaa	O(mn)	**Pattern:** aaab **Text:** aaaaaaaaaaaa
Single Occurrence	O(m)	**Pattern:** aaaa **Text:** aaaaaaaaaaaa	O(mn)	**Pattern:** aaab **Text:** aaaaaaaaaab
All Occurrences	O(n)	**Pattern:** baaa **Text:** aaaaaaaabaaa	O(mn)	**Pattern:** aaab **Text:** aaaaaaaaaab

Boyer-Moore

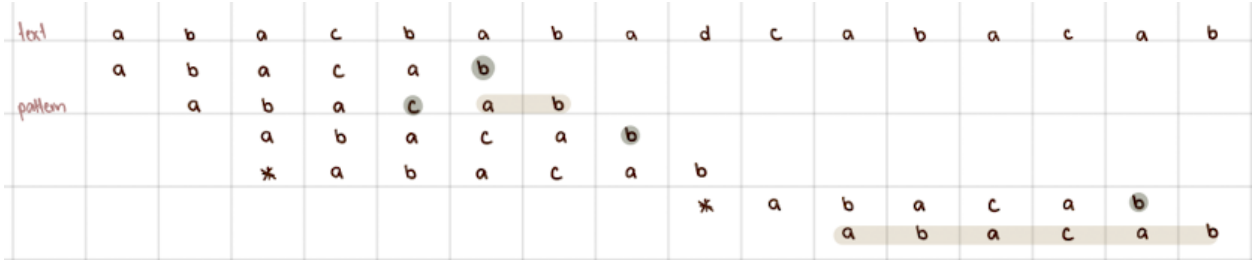
- Uses last occurrence table to skip characters. Move Right → Left pattern
    - Match, continue comparing text and pattern
    - Mismatch, text character in alphabet → shift to right
    - Mismatch, text character not in alphabet → shift over mismatch text

Last Occurrence Table

- Records index of last occurrence of the letter



Example



Galil Rule

- Find period – when full match found, use Galil Rule

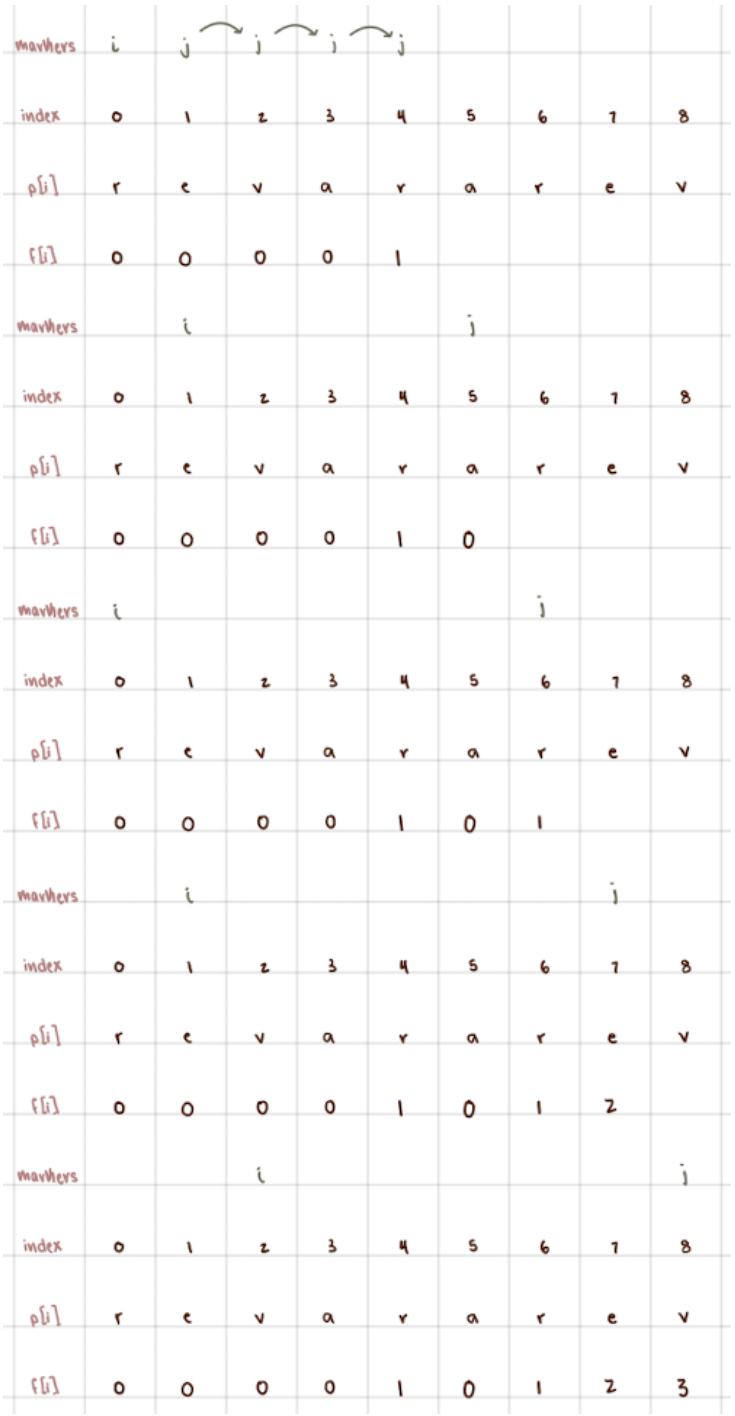
Runtime

Boyer-Moore: Time Complexity				
Scenario	**Best Case Efficiency**	**Example Best Case**	**Worst Case Efficiency**	**Example Worst Case**
No Occurrences	O(m + (n/m))	**Pattern:** bbbb **Text:** aaaaaaaaaaaa	O(mn)	**Pattern:** baaa **Text:** aaaaaaaaaaaa
Single Occurrence	O(m)	**Pattern:** aaaa **Text:** aaaaaaaaaaaa	O(mn)	**Pattern:** baaa **Text:** aaaaaaaabaaa
All Occurrences	O(m + (n/m))	**Pattern:** aaab **Text:** aaaaaaaaaaab	O(mn)	**Pattern:** baaa **Text:** aaaaaaaabaaa

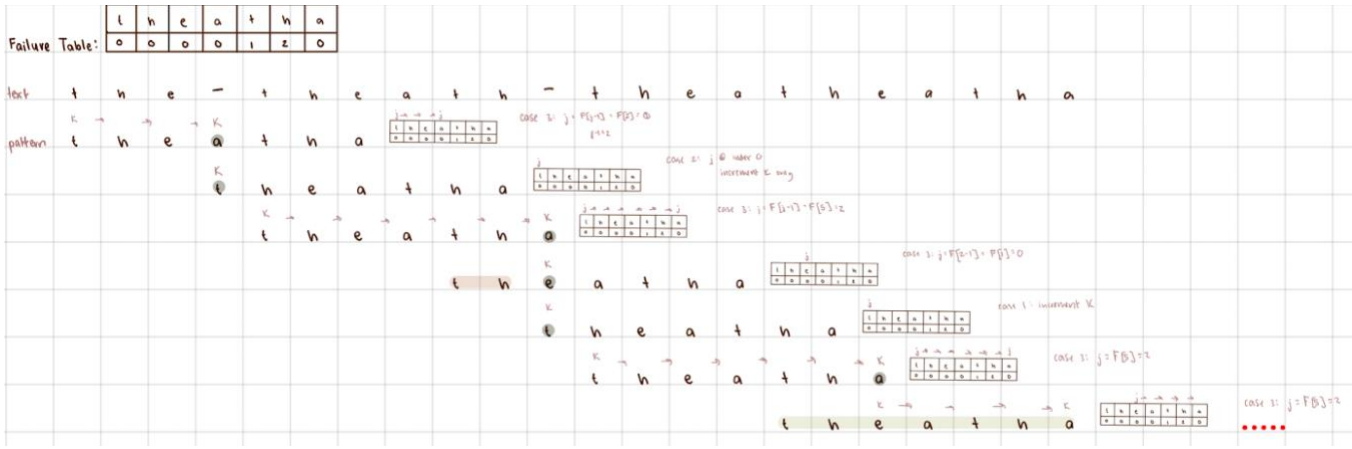
KMP Search

- Upon mismatch, query pattern index in Failure Table
- Realigning causes all characters before the mismatch to still match
- Begin comparing from the beginning of the pattern

Failure Table



Example



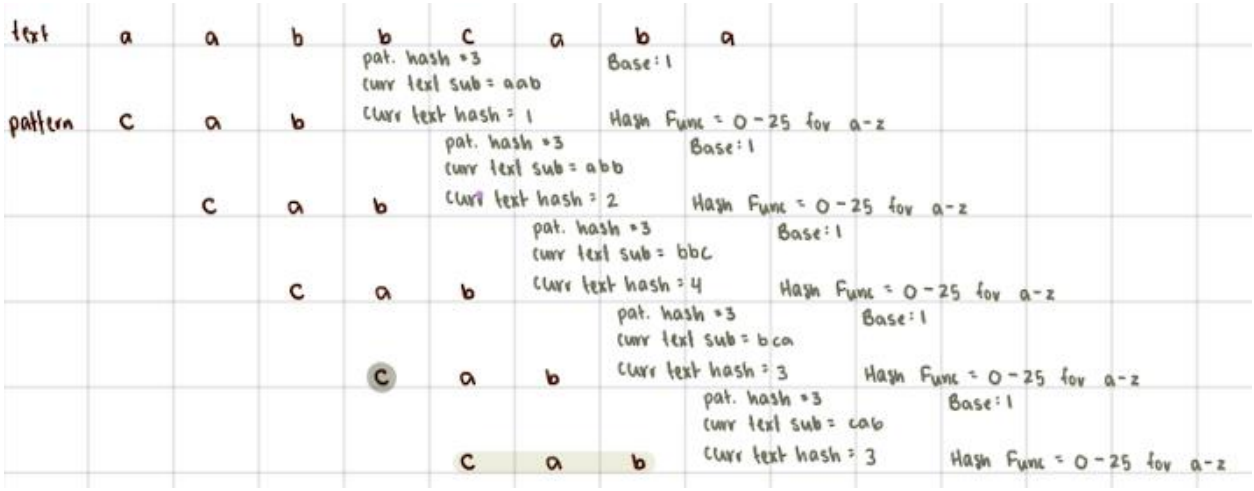
Runtime

KMP: Time Complexity		
Scenario	**Best Case Efficiency**	**Worst Case Efficiency**
No Occurrences	O(m + n)	O(m + n)
Single Occurrence	O(m)	O(m + n)
All Occurrences	O(m + n)	O(m + n)

Rabin Karp

- Uses rolling hash to calculate the hash of the pattern & hash of the substring. If the hash are not equal, rolling hash “slides” to the right by one character
- Formula: $\sum_{i=0}^{j} text[i] * BASE^{j-i-1}$
- First calculate the hash of the pattern, then j character of text
- If hashes are the same, compare each character with the character in the hash

Example



Runtime

Rabin-Karp: Time Complexity		
Scenario	**Best Case Efficiency**	**Worst Case Efficiency**
No Occurrences	O(m + n)	O(mn)
Single Occurrence	O(m)	O(mn)
All Occurrences	O(m + n)	O(mn)

Graph Algorithms

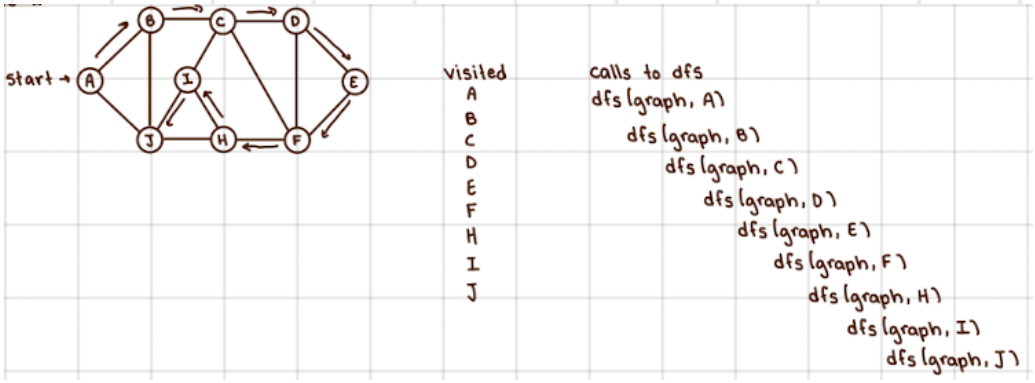
Basic Terminology

- Graph: Set vertices & edges
  - G = {V, E}
  - Order(G) = |V|
  - Size(G) = |E|
  - Degree of vertex = deg(v)
- Edge: connection between 2 vertices → e = (u, v)
- Adjacency: there exists a single edge which connects 2 vertices
- Directed Graph: directed edge → only 1 direction from origin to destination
- Undirected Graph: Can be traversed both directions
- Weight: Cost of traversing given edge
- Dense: # of edges close to max # edges possible
- Cyclic: Vertices are repeated in a path
- Path: simple path → sequence of non-repeated adjacent vertices
- Trail: non-simple path → repeated vertices allowd, but edges cannot repeat
- Walk: trail where edges can repeat
- Cycle: path where the first and last vertices are adjacent
- Circuit: trail where the first and last vertices are adjacent
- Connected: for every pair of vertices, there exists a path between the two
- Complete: undirected/every edge present

Depth First Search (DFS)

- Begin at the root node and proceed through the nodes as far as possible until you reach a node with no unvisited nearby nodes

Examples



Code

```
public static <T> List<Vertex<T>> dfs(Vertex<T> start, Graph<T> graph) {
    if (start == null) {
        throw new IllegalArgumentException("start cannot be null");
    } else if (graph == null) {
        throw new IllegalArgumentException("graph cannot be null");
    } else if (!graph.getVertices().contains(start)) {
        throw new IllegalArgumentException("start doesn't exist in the graph");
    } else {
        Set<Vertex<T>> visited = new HashSet<>();
        List<Vertex<T>> order = new ArrayList<>();
```

```
        dfsRecursive(start, graph, visited, order);

        return order;
    }
}

private static <T> void dfsRecursive(Vertex<T> currVertex, Graph<T> graph,
                                     Set<Vertex<T>> visited, List<Vertex<T>> order) {
    visited.add(currVertex);
    order.add(currVertex);

    for (VertexDistance<T> adj : graph.getAdjList().get(currVertex)) {
        if (!visited.contains(adj.getVertex())) {
            dfsRecursive(adj.getVertex(), graph, visited, order);
        }
    }
}
```

Runtime

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

Topological Sort (DFS)

- Given a directed acyclic graph, we can sort the graph by precedence
    - A linear ordering of its vertices such that for every directed edge u -> v, vertex u appears before vertex v in the ordering

PseudoCode

```
def top-sort(G, v):
    marked[v] = true
    for edge (v, u) in G connected to V:
        if not marked[u]:
            explore(G, u)
    L = [v] + L

def explore(G, v):
    marked[v] = true
    t-marked[v] = true
    for edge (v, u) in G connected to v:
        if t-marked[u]:
            print('cycle found!!!')
        if not marked[u]:
            explore(G, u)
    t-marked[v] = false
```

Example

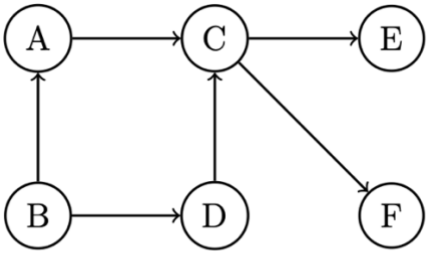


Figure 13: A directed acyclic graph.

$$
\begin{aligned}
A \to C \to E &: [E] \\
A \to C \to F &: [F, E] \\
A \to C &: [C, F, E] \\
A &: [A, C, F, E] \\
B \to D &: [D, A, C, F, E] \\
B &: [B, D, A, C, F, E]
\end{aligned}
$$

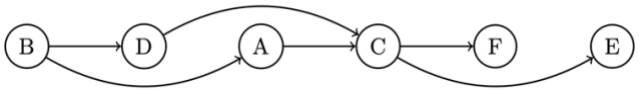


Figure 14: A linearization of a directed acyclic graph.

Connected Components (DFS)

- A group of nodes where you can reach any node from any other node without that group, following the edges

How DFS is used

- Start a DFS from any node
- The DFS will visit all nodes that are connected to this starting node
- These visited nodes form one connected component
- If there are still unvisted nodes, start another DFS from one of them – this will give a new connected component
- Repeat until all nodes have been visited

Undirected vs Directed

- Undirected: use DFS to visit all reachable nodes from a starting point
- Directed Graphs: use SCC (groups of nodes hwere every node can reach every other node and also be reached by them)

SCC PseudoCode

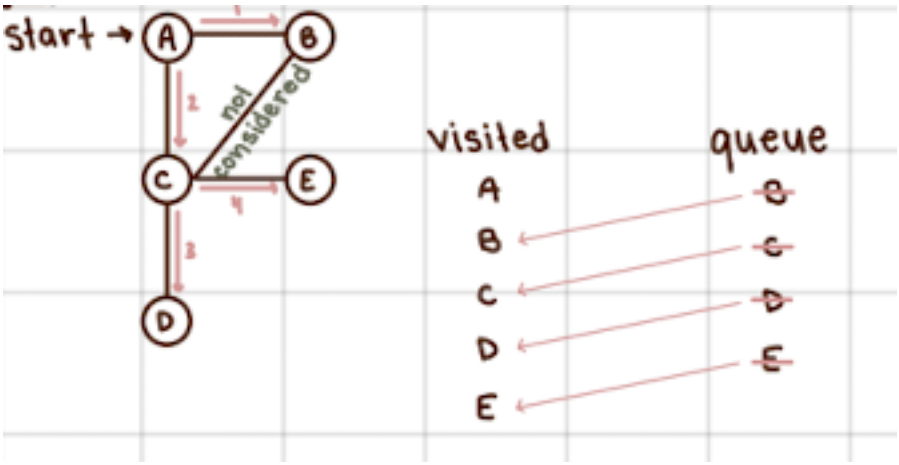
```
def scc(G):
    dfs(G) to get post labels in decreasing order
    compute reversal GR from G
    while posts not empty:
        v = max(posts)
        component = explore(GR,v)
        print or mark the component
        remove component from GR and posts
```

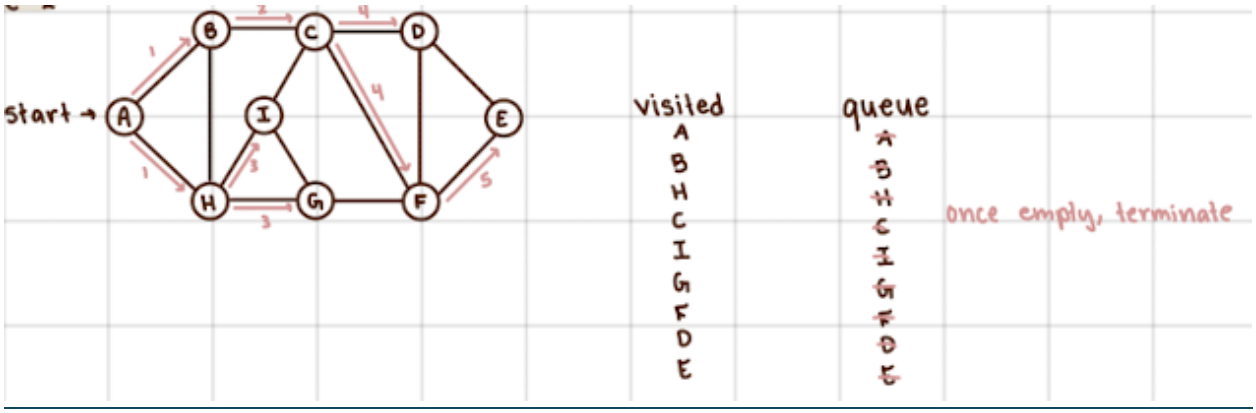
- Perform a DFS on the original graph and record the finish times of each node
- Transpose the graph (reverse the direction of all edges)
- Perform DFS on the transposed graph, in the order of decreasing finish times from step 1. Each DFS call will give you a SCC

Breadth First Search (BFS)

- Walks through all nodes on the same level before going to the next level

Examples





Code

```
public static <T> List<Vertex<T>> bfs(Vertex<T> start, Graph<T> graph) {
    if (start == null) {
        throw new IllegalArgumentException("start cannot be null");
    } else if (graph == null) {
        throw new IllegalArgumentException("graph cannot be null");
    } else if (!graph.getVertices().contains(start)) {
        throw new IllegalArgumentException("start doesn't exist in the graph");
    } else {
        Set<Vertex<T>> visited = new HashSet<>();
        Queue<Vertex<T>> queue = new LinkedList<>();
```

```
        List<Vertex<T>> order = new ArrayList<>();

        visited.add(start);
        queue.add(start);

        while (!queue.isEmpty()) {
            Vertex<T> curr = queue.poll();
            order.add(curr);

            for (VertexDistance<T> adj : graph.getAdjList().get(curr)) {
                if (!visited.contains(adj.getVertex())) {
                    visited.add(adj.getVertex());
                    queue.add(adj.getVertex());
                }
            }
        }
        return order;
    }
}
```

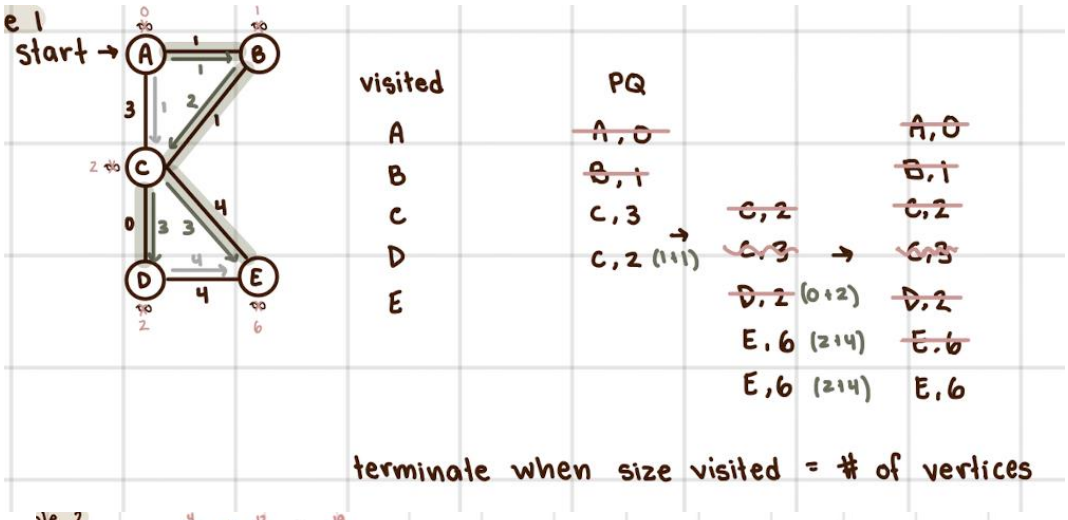
Runtime

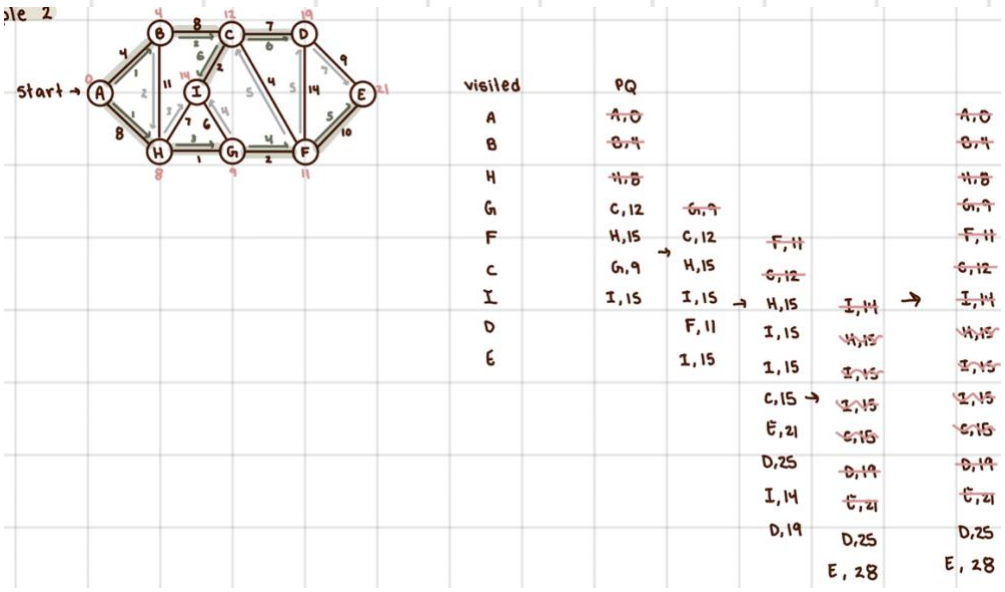
$O(|V| + |E|)$

Dijkstra's Algorithm

- Finds the shortest path between nodes in a weighted graph (positive weights)
- Want to compute the shortest paths through the total sum of the edges in the path

Example





Code

```
public static <T> Map<Vertex<T>, Integer> dijkstras(Vertex<T> start,
                                                    Graph<T> graph) {
    if (start == null) {
        throw new IllegalArgumentException("start cannot be null");
    } else if (graph == null) {
```

```
            throw new IllegalArgumentException("graph cannot be null");
        } else if (!graph.getVertices().contains(start)) {
            throw new IllegalArgumentException("start doesn't exist in the graph");
        } else {
            Set<Vertex<T>> visited = new HashSet<>();
            PriorityQueue<VertexDistance<T>> pq = new PriorityQueue<>();
            Map<Vertex<T>, Integer> distanceMap = new HashMap<>();
            Map<Vertex<T>, List<VertexDistance<T>>> vMap = graph.getAdjList();

            for (Vertex<T> vertex : vMap.keySet()) {
                if (vertex.equals(start)) {
                    distanceMap.put(vertex, 0);
                } else {
                    distanceMap.put(vertex, Integer.MAX_VALUE);
                }
            }

            pq.add(new VertexDistance<>(start, 0));
            while (!pq.isEmpty() && !(visited.size() == graph.getVertices().size())) {
                VertexDistance<T> curr = pq.remove();

                if (!visited.contains(curr.getVertex())) {
                    visited.add(curr.getVertex());

                    for (VertexDistance<T> dist : vMap.get(curr.getVertex())) {
                        if (!visited.contains(dist.getVertex()) && (dist.getDistance() + curr.getDistance()) <
distanceMap.get(dist.getVertex())) {
                            pq.add(new VertexDistance<>(dist.getVertex(), dist.getDistance() + curr.getDistance()));
                            distanceMap.put(dist.getVertex(), dist.getDistance() + curr.getDistance()));
                        }

                    }
                }
            }
            return distanceMap;
        }
    }
```

Runtime

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

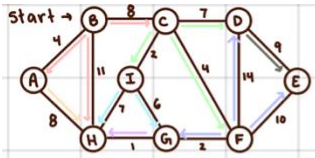
$O(|V| \log |V|)$ – visit each vertex, perform $|V|$ calls to remove from the Heap-based Priority Queue

$O(|E| \log |V|)$ – consider every path, perform $|E|$ calls to add to the Heap-based Priority Queue

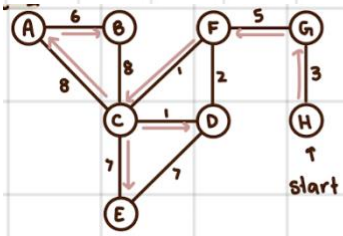
Prim's Algorithm

- Looks for the shortest path form one vertex to adjacent vertices

Example



visited	PQ							
B	~~B, A, 4~~							~~B, A, 4~~
A	B, C, 8	~~B, C, 8~~						~~B, C, 8~~
C	B, H, 11	A, H, 8	~~C, I, 2~~					~~C, I, 2~~
I	A, H, 8	B, H, 11	C, F, 4	~~C, F, 4~~				~~C, F, 4~~
F		C, I, 2	C, D, 7	I, G, 6	~~F, G, 2~~			~~F, G, 2~~
G		C, F, 4	A, H, 8	C, D, 7	I, G, 6	~~G, H, 1~~		~~G, H, 1~~
H		C, D, 7	B, H, 11	I, H, 7	C, D, 7	~~I, G, 6~~		~~I, G, 6~~
D			I, H, 7	A, H, 8	I, H, 7	~~C, D, 7~~		~~C, D, 7~~
E			I, G, 6	B, H, 11	A, H, 8	I, H, 7	~~I, H, 7~~	~~I, H, 7~~
				F, G, 2	F, E, 10	A, H, 8	~~A, H, 8~~	~~A, H, 8~~
				F, E, 10	B, H, 11	F, E, 10	~~D, E, 9~~	~~D, E, 9~~
terminate when size visited = # of vertices				F, D, 14	F, D, 14	B, H, 11	F, E, 10	F, E, 10
					G, H, 1	F, D, 14	B, H, 11	B, H, 11
						D, E, 9	F, D, 14	F, D, 14



visited	PQ
H	~~H, G, 3~~
G	~~G, F, 5~~
F	~~F, C, 1~~
C	~~F, D, 2~~
D	~~C, D, 1~~
E	~~C, E, 7~~
A	~~C, A, 8~~
B	C, B, 8
	~~D, E, 9~~
	~~A, B, 6~~

Code

Algorithm Prim(G,s)

```
1.  initialize VisitedSet, VS
2.  initialize MST EdgeSet, MST
3.  initialize PriorityQueue, PQ
4.  for each edge(s, v) in G, PQ.enqueue(edge(s, v))
5.  mark s as visited in VS
6.  while PQ is not empty and VS is not full
7.      edge(u, w) ← PQ.dequeue()
8.      if w is not visited in VS
9.          mark w as visited in VS
10.         add edge(u, w) to MST
11.         for each edge(w, x) such that x is not visited
12.             PQ.enqueue(edge(w, x))
```

Runtime

$O(|E| \log |E|)$

Minimum Spanning Tree

- Given a weighted undirected graph, give a graph which is still connected but with the smallest weights

Goal

- Resulting graph is connected
- Total weight of edges is as small as possible
- Graph has no cycles
- Has |V| - 1 edges

Example

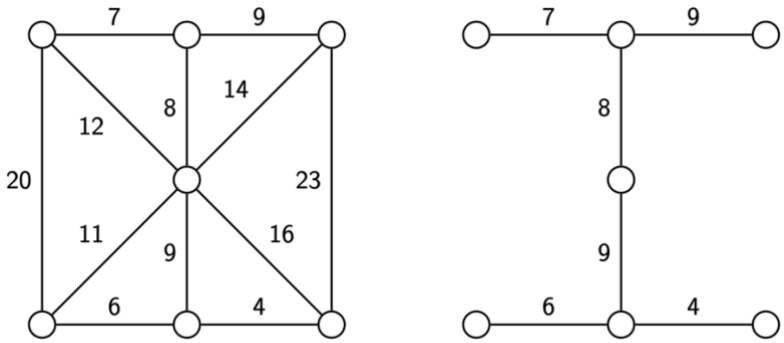
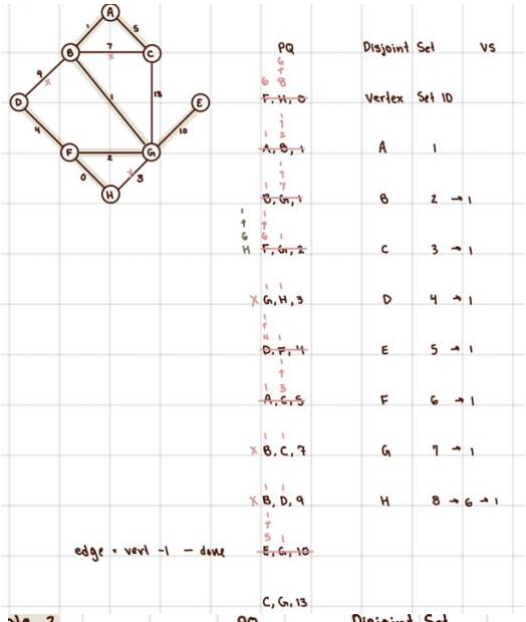


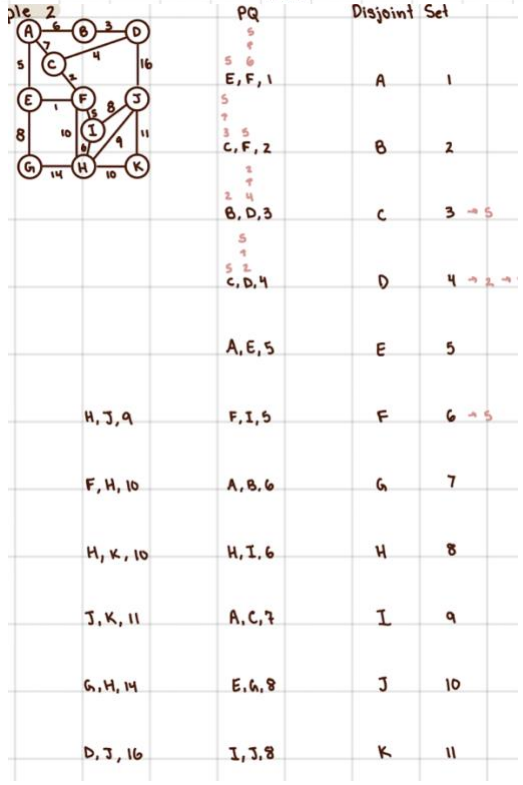
Figure 1: A weighted graph and it's MST.

Kruskal's Algorithm

- Shortest path between two points – shortest spanning tree
- At each step, add the smallest edge to a set which does not form a cycle with edges within that set

Example





Code

```
public static <T> Set<Edge<T>> kruskals(Graph<T> graph) {
    if (graph == null) {
        throw new IllegalArgumentException("graph is null");
    }
    DisjointSet<T> ds = new DisjointSet<>();
    Set<Edge<T>> mstEdgeSet = new HashSet<>();
    PriorityQueue<Edge<T>> pq = new PriorityQueue<>();
    int numVertices = graph.getEdges().size();

    for (Edge<T> edges : graph.getEdges()) {
        pq.add(edges);
    }

    while (!pq.isEmpty() && mstEdgeSet.size() != 2 * (numVertices - 1)) {
        Edge<T> removedEdge = pq.remove();
        Vertex<T> u = removedEdge.getU();
        Vertex<T> v = removedEdge.getV();

        if (!ds.find(u).equals(ds.find(v))) {
            ds.union(u, v);
            mstEdgeSet.add(new Edge<>(u, v, removedEdge.getWeight()));
            mstEdgeSet.add(new Edge<>(v, u, removedEdge.getWeight()));
        }
    }
    if (mstEdgeSet.size() == 2 * (numVertices - 1)) {
        return mstEdgeSet;
    } else {
        return null;
    }

}
```

Runtime

$O(|E| \log |E|)$

Bellman-Ford

- Finds the shortest path from a single source node to all other nodes in a graph, even when there are negative edge weights

How it Works

- Update distance of each node with
  - Dist(v) = min(dist(v), dist(u) + l(u, v))
- Because of negative edge weights, algorithm updates the distance multiple times because a future update might still reduce the distance to a node

PsuedoCode and Explanation

```
def bellmanFord(G, l, s):
    for all v in V:
        dist(u) = infinity

    dist(s) = 0
    for i=1...(|V| - 1)
        for each e=(u,v) in E
            dist(v) = min{dist(v), dist(u) + l(u, v)}
```

- Initialize distances:
  - Set distance to source node s to 0
  - Set distance to all other nodes to infinity
- Relax all edges |V| - 1 times
  - For each node in the graph, go through all the edges and update the distance to the destination node using the relaxation rule
  - Repeat process |V| -1 times
- Stop after |V| -1 rounds

Example

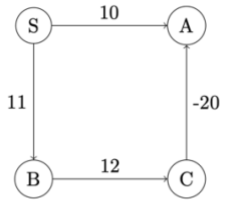


Figure 5: Bellman-Ford example graph

Round	dist(S)	dist(A)	dist(B)	dist(C)
0	0	∞	∞	∞
1	0	10	11	∞
2	0	10	11	23
3	0	3	11	23

Runtime

O(|V| * |E|) because it loops over all edges |E| in the graph |V| - 1 times

Floyd-Warshall

- Incrementally improves the shortest paths between all pairs of nodes by considering more intermediate nodes on the path
- Finds shortest paths between all pairs of nodes in a graph

Subproblem Definition

- Dist(i, j, k) = the shortest path from node v_i to node v_j, considering only intermediate nodes from v1 to vk
    - Try to find the shortest path from node v_i to v_j
    - Only allowed to use the first k nodes as possible intermediate nodes

Pseudocode

```
def floydWarshall(G, l):
    # Base cases
    for i in range(|V|):
        dist(i, i) = 0
    for i in range(|V|):
        for j in range(|V|):
            dist(i, j) = l(i, j) if defined, infinity otherwise

    # Inductive cases
    for k in range(|V|)
        for i in range(|V|)
            for j in range(|V|)
                dist(i,j) = min(dist(i,j), dist(i,k) + dist(k, j))
```

- Base Case:
    - Self-Loops: the shortest path from any node to itself (dist(i, i) is 0 because you don't need to move)
    - Direct Connections: if there exists a direct edge between two nodes v_i and v_j,. then the shortest path is the edge's weight
    - No Direct Connections: If there is no direct edge between two nodes, initialize the distance between them as infinity since you can't connect them
- For each possible node k, check if current shortest path between the two nodes v_i and v_j can be improved by passing through k

    - Without using node k: keep existing shortest path between v_i and v_j (dist(I, j, k -1))
    - Using node k: consider a new path that goes from v_i to v_k and then from v_k to v_j. The total distance is dist(I, k) + dist(k, j)
- Take minimum of the two possibilities

Runtime

O(|V|^3) because it has three nested loops

Difference between Bellman-Ford

Floyd-Warshall: finds shortest paths between ALL pairs of nodes in a graph

Bellman-Ford: finds the shortest path from a single source node to all other nodes in a graph

Dynamic Programming

Longest Common Substring

- Given two strings, want to find the longest common substring
    - String 1: “EL GATO”
    - String 2: “GATER”
        - Longest Common Substring = “GAT”

Key Idea

- Build a 2D table to store the lengths of the common suffixes
- Table Definition:
    - dp[i][j]
        - ‘i’ is an index in first string
        - ‘j’ is an index in second string
    - Value at dp[i][j] wills tore the length of the longest common suffix between the substrings
- Filling the Table:
    - For each pair of indices ‘i’ and ‘j’, compare the characters x[i-1] and y[j-1]
        - If same, extend previous common suffix by 1 → dp[i][j] = dp[i-1][j-1] + 1
        - If different, reset dp[i][j] to 0 → no common suffix for this pair
- Tracking Max Length:
    - Length of the longest common substring will be the maximum value in the table

Code

```
def lcsubstring(x, y):
    # Step 1: Create a DP table initialized with 0s
    dp = [[0] * (len(y) + 1) for _ in range(len(x) + 1)]
    max_len = 0
    max_pos = (0, 0)

    # Step 2: Fill the table
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            if x[i - 1] == y[j - 1]:  # Compare characters
                dp[i][j] = dp[i - 1][j - 1] + 1  # Extend common suffix
```

```
            if dp[i][j] > max_len:  # Update max length
                max_len = dp[i][j]
                max_pos = (i, j)
        else:
            dp[i][j] = 0  # No common suffix

    # Step 3: Extract the longest common substring from the result
    i, j = max_pos
    longest_substring = x[i - max_len:i]  # Substring from start of longest suffix
    return longest_substring
```

Example

```
    G A T E R
  0 0 0 0 0 0
E 0 0 0 0 1 0
L 0 0 0 0 0 0
G 0 1 0 0 0 0
A 0 0 2 0 0 0
T 0 0 0 3 0 0
O 0 0 0 0 0 0
```

Runtime

O(n * m)

O(1) work for each entry

Longest Common Subsequence

- Finds the longest sequence of characters that appear in the same order in both strings but don't have to be contiguous
    - X = [a, b, c, b, d, a, b]
    - Y = [b, d, c, a, b, a]
        - The longest subsequence would be [b, d, a, b]

Key Idea

- Consider two cases
    - Characters are equal (x[i-1] == y[j − 1])
        - Add 1 to the length of the LCS of the previous prefixes (i.e dp[i-1][j-1]) because we found one more matching character
    - Characters are different (x[i-1] != y[j − 1])
        - Take the maximum of either:
            - LCS without the current character of 'x' (dp[I − 1][j])
            - LCS without the current character of 'y' (dp[i][j − 1])
        - This will ensure we get the longest subsequence possible up to this point

Code

```
def lcsubsequence(x, y):
    # Step 1: Create DP and Backtracking tables
    dp = [[0] * (len(y) + 1) for _ in range(len(x) + 1)]
    bt = [[None] * (len(y) + 1) for _ in range(len(x) + 1)]

    # Step 2: Fill the DP table
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            if x[i - 1] == y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                bt[i][j] = '↖'
            elif dp[i - 1][j] >= dp[i][j - 1]:
                dp[i][j] = dp[i - 1][j]
                bt[i][j] = '←'
            else:
                dp[i][j] = dp[i][j - 1]
```

```
            bt[i][j] = '↑'

    # Step 3: Backtrack to find the LCS
    i, j = len(x), len(y)
    lcs = []
    while i > 0 and j > 0:
        if bt[i][j] == '↖':
            lcs.append(x[i - 1])
            i -= 1
            j -= 1
        elif bt[i][j] == '←':
            i -= 1
        else:
            j -= 1

    # Since we backtrack, reverse the list
    lcs.reverse()
    return ''.join(lcs)
```

- ↖ means the characters matched and we came from dp[i-1][j-1].
- ← means we came from dp[i-1][j] (ignoring the character in x).
- ↑ means we came from dp[i][j-1] (ignoring the character in y).

Example

```
    "" b d c a b a
"" 0 0 0 0 0 0 0
a  0 0 0 0 1 1 1
b  0 1 1 1 1 2 2
c  0 1 1 2 2 2 2
b  0 1 1 2 2 3 3
d  0 1 2 2 2 3 3
a  0 1 2 2 3 3 4
b  0 1 2 2 3 4 4

    "" b d c a b a
```

```
""    ← ← ← ← ← ←
a  ↑ ↖ ↑ ↑ ↖ ← ↖
b  ↑ ↖ ← ← ↑ ↖ ←
c  ↑ ↑ ↖ ↑ ↑ ← ←
b  ↑ ↖ ← ↖ ↑ ↖ ←
d  ↑ ↑ ↖ ← ← ↑ ←
a  ↑ ↑ ↑ ↑ ↖ ← ↖
b  ↑ ↖ ← ← ↑ ↖ ←
```

Runtime

- $O(n * m)$
- $O(1)$ work for each entry

Longest Palindromic Subsequence

- Finds the longest sequence of characters that appear in the same order that reads the same forwards and backwards
    - One input

Key Idea

- Base Case
    - Every single character is a palindrome of length 1. So for every 'I', dp[i] = 1
- Recurrence Relation
    - If the characters at the ends of the current substring (x[i] and x[j]) are equal (x[i] == x[j]), they can form the two ends of a palindrome. So, we add 2 to the longest palindromic subsequence found in the substring x[i+1...j-1]:
        - dp[i][j] = 2 + dp[i+1][j-1]
    - If the characters are not equal (x[i] != x[j]), then the longest palindromic subsequence for x[i...j] must come from either:
        - x[i+1...j] (ignoring x[i]) OR
        - x[i...j-1] (ignoring x[j])
        - Take maximum of the two: dp[i][j] = max(dp[i+1][j], dp[i][j-1])

Code

```
def lpalindromesubsequence(x):
    n = len(x)

    # Step 1: Create a DP table initialized to 0
    dp = [[0] * n for _ in range(n)]

    # Step 2: Every single character is a palindrome of length 1
    for i in range(n):
        dp[i][i] = 1

    # Step 3: Fill the table for substrings of length greater than 1
    for s in range(1, n):  # s is the difference between i and j (substring length)
        for i in range(n - s):
            j = i + s
            if x[i] == x[j]:  # If characters match, add 2 to the palindrome within x[i+1...j-1]
```

```
            dp[i][j] = 2 + dp[i + 1][j - 1]
        else:  # Otherwise, take the maximum of ignoring either character
            dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

    # Step 4: The longest palindromic subsequence length is in dp[0][n-1]
    return dp[0][n-1]
```

Example

```
   b b a b c b c a b
b  1 2 1 3 3 5 5 5 7
b    1 1 3 3 5 5 5 5
a      1 1 1 3 3 3 3
b        1 1 3 3 3 5
c          1 1 1 3 3
b            1 1 3 3
c              1 1 1
a                1 1
b                  1
```

Runtime

O(n^2)

Since filling a n x n table

Chain Matrix Multiplication

- Given sequence of matrices, find the optimal order to multiply them so that the total number of operations is minimized

How to DP it

- dp[i][j] represents the minimum cost of multiplying the matrices from A[i] to A[j]
- Base Case:
    - i == j: given only one matric
        - cost is zero bc no multiplication is needed
- Recursive: Find cost of multiply matrices – split to two parts -- (A[i...k]) * (A[k+1...j])
    - Cost = cost of multiplying two subchains + cost of multiplying the result of the two subchains
        - Do for all possible values of 'k' (I to j – 1) and take minimum cost: dp[i][j] = min(dp[i][k] + dp[k+1][j] + di-1 * dk * dj)
        - di-1 * dk * dj is the cost of multiplying the result of A[i...k] (which has dimensions di-1 * dk) with A[k+1...j] (which has dimensions dk * dj)
- Solution: dp[1][n] where n is the number of matrices

PsuedoCode

```
def chainmatrix(dimensions):
    n = len(dimensions) - 1  # Number of matrices
    dp = [[0] * n for _ in range(n)]

    # Fill the table in a bottom-up manner
    for s in range(1, n):  # s is the length of the subchain
        for i in range(n - s):
            j = i + s
            dp[i][j] = float('inf')  # Set to a large value
            for k in range(i, j):
                cost = dp[i][k] + dp[k + 1][j] + dimensions[i] * dimensions[k + 1] * dimensions[j + 1]
                dp[i][j] = min(dp[i][j], cost)

    return dp[0][n - 1]  # This holds the minimum cost for the entire chain
```

Example

- A: 50 x 20; B: 20 x 1; C: 1 x 10; D: 10 x 100
- Sizes: [50, 20, 1, 10, 100]
- Dp[n][n] where n = 4 (4 matrices)
- Filling DP table:
    - Base Case:

0	-	-	-
-	0	-	-
-	-	0	-
-	-	-	0

    - Subchains of length 2 (s = 1):
        - dp[1][2] = 50×20×1 = 1000
            - m0m1m2
        - dp[2][3] = 20×1×10 = 200
            - m1m2m3
        - dp[3][4] = 1×10×100 = 1000
            - m2m3m4

```
0   1000  -    -
-   0     200  -
-   -     0    1000
-   -     -    0
```

    - Subchains of length 3 (s = 2):
        - For dp[1][3], we try splitting the chain at k = 1 and k = 2:
            - For k = 1: 50×20×10 = 10,000
                - m0m1m3
            - For k = 2: 50×1×10 = 500
                - m0m2m3
            - **Minimum**: 1500
        - For dp[2][4], we try splitting the chain at k = 2 and k = 3:
            - For k = 2: 20×1×100 = 2000
                - m1m2m4
            - For k = 3: 20×10×100 = 20,000
                - m1m3m4

- **Minimum**: 3000

- T[1,3] = min(T[1,1] + T[2,3] + $m_0m_1m_3$ = 1500,T[1,2] + T[3,3] + $m_0m_2m_3$ = 1500) = 1500
- T[2,4] = min(T[2,2] + T[3,4] + $m_1m_2m_4$ = 3000,T[2,3] + T [4, 4] + $m_1m_3m_4$ = 3000) = 20200) = 3000

```
0   1000  1500  -
-   0     200   3000
-   -     0     1000
-   -     -     0
```

- Subchains of length 4 (s = 3):
  - for `dp[1][4]`, we try splitting the chain at `k = 1`, `k = 2`, and `k = 3`:
    - For k = 1: 50×20×100 = 100,000
      - m0m1m4
    - For k = 2: 50×1×100 = 5000
      - m0m2m4
    - For k = 3: 50×10×100 = 50,000
      - m0m3m4
    - Minimum: 7000

$$dp[1,4] = \min \left\{ \begin{array}{ll} dp[1,1] + dp[2,4] + m_0m_1m_4, & \text{if } k = 1 \\ dp[1,2] + dp[3,4] + m_0m_2m_4, & \text{if } k = 2 \\ dp[1,3] + dp[4,4] + m_0m_3m_4, & \text{if } k = 3 \end{array} \right\}$$

$$dp[1,4] = \min \left\{ \begin{array}{ll} 0 + 3000 + 100000, & \text{if } k = 1 \\ 1000 + 1000 + 5000, & \text{if } k = 2 \\ 1500 + 0 + 50000, & \text{if } k = 3 \end{array} \right\} = 7000$$

```
0   1000  1500  7000
-   0     200   3000
-   -     0     1000
-   -     -     0
```

Optimal Cost at dp[0][3]: 7,000

Runtime

O(n^3)

Knapsack

- Given n items with each:
  - A **value** vi
  - A **weight** wi
- Also given a knapsack with a maximum capacity "**W**"
- Goal: pick items to maximize the total value without exceeding he weight capacity "W"

Example

- W = 6; Weights = [1, 2, 3, 4], Values = [12, 17, 18, 25]
  - Most optimal solution is items 1, 2, and 3 → 1 + 2 + 3 with value 12 + 17 + 18 = 47

How to DP it

- T[i][j] where i = current capacity from O to W and j items 1 to n
- T[i][j] will store the maximum value we can get with capacity I and the first j items

Key Idea – No Repetition

- At every step, for each item j and capacity i
  - Don't pick the item: maximum value is the same as without considering the item
    - T[i][j] = T[i][j – 1]
  - Pick the item: add the value of the item and reduce the capacity by its weight
    - T[i][j] = T[i – wj][j – 1] + vj (if the weight fits)
- Final Solution: T[W][n]

Pseudocode – No Repetition

```
def knapsack(W, weights, values):
    n = len(weights)
    dp = [[0] * (n + 1) for _ in range(W + 1)]

    for j in range(1, n + 1):
        for i in range(1, W + 1):
            if weights[j - 1] <= i:
                dp[i][j] = max(dp[i][j - 1], dp[i - weights[j - 1]][j - 1] + values[j - 1])
            else:
                dp[i][j] = dp[i][j - 1]
```

```
    return dp[W][n]
```

Key Idea – Repetition

- In the version where you can pick the same item multiple times:
    - The only change is that instead of using T[i - wj][j-1], you use T[i - wj][j] (since you can use the same item again).

Pseudocode – Repetition

```
def knapsack_repeat(W, weights, values):
    dp = [0] * (W + 1)

    for i in range(1, W + 1):
        for j in range(len(weights)):
            if weights[j] <= i:
                dp[i] = max(dp[i], dp[i - weights[j]] + values[j])

    return dp[W]
```

Runtime

O(nW)

Example 1

Problem

Suppose you have n stair steps. You can leap one, two, or three steps at a time. What is the number of combinations, or the number of ways, that you could reach step n?

Solution

Create an array T [O . . . n] where T [i] represents the number of ways to reach step i.

Our **base cases** are T[O] = 1,T[1] = 1 and T[2] = 2. To reach step n, there was some last step. You either jumped 1, 2, or 3 steps, and you jumped from 1, 2, or 3 steps away. That means that the number of ways to go from O to n is the sum of the number of ways to go from O to n−1, O to n−2, and O to n−3. This gives us our **recurrence** of: T [i] = T [i − 1] + T [i − 2] + T [i − 3]

Pseudocode

```
def numways(k):
    initialize dp as table of size n + 1 with 0s
    dp[0] = 0
    dp[1] = 1
    dp[2] = 2

    for i in 3..(n+1):
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]

    return dp[k]
```

indices	0	1	2	3	4	5	6	7	8	9
results	1	1	2	4	7	13	24	44	81	149

Runtime

size of the table multiplied by the work done to calculate each cell -- O(n) × O(1) = O(n)

Example 2

Problem

Given two operations: add one and multiply by two

How many operations does it take to get from 0 to some k

Solution

The results are as follows.

indices	0	1	2	3	4	5	6	7	8	9
results	0	1	2	3	3	4	4	5	4	5

It's not monotonic, and it's actually a little hard. Note that for 8 it's a power of two so it takes fewer operations than either of its neighbors. Trivially you could find this in exponential time by trying all possibilities. Let's write our recurrence.

$dp[0] = 0$ because for 0, you need zero operations. $dp[1] = 1$ because for 1, you add one to zero. Now we'll try to multiply by two as much as possible so we'll add 1 if the element is odd and multiply by two if it is even. We get the following overall recurrence.

$$d(i) = \left\{ \begin{array}{ll} d(i-1)+1, & \text{if } i \text{ is odd} \\ d(i/2)+1, & \text{if } i \text{ is even} \end{array} \right\} \tag{1}$$

Pseudocode

```
def minop(k):
    initialize dp as table of size k + 1 with 0s
    dp[0] = 0
    dp[1] = 1

    for i in 2..(k + 1)
        dp[i] = dp[i - 1] + 1
        if i is even
            dp[i] = min(dp[i], dp[i / 2] + 1)

    return dp[k]
```

Example 3

Problem

Given an array houses with values H = [h1 . . . hn], you want to rob them, but you can't rob two adjacent houses or alarms will go off. What is the maximum amount you can steal?
Here's an example: [2, 7, 9, 3, 1] would yield 2 + 9 + 1 = 12.

Solution

Let's define our array T [0 . . . n] with T [i] =maximum we can steal from houses 0...i.
The **base cases** are that T[0] = H[0],T[1] = max(H[0],H[1]).

Recurrence: At the next house visited, we can choose to rob or not rob a house. We take the maximum of both scenarios. If we rob the house, then we could not have robbed the previous house, and if we didn't rob the house, then we could have robbed the previous house. This gives us the **recurrence**: T [i] = max(T [i − 2] + H [i], T [i − 1])

Pseudocode

```
def houserobber(H):
    initialize dp as table of size n with 0s
    dp[0] = H[0]
    dp[1] = max(H[0], H[1])

    for i in 2..n
        dp[i] = max(dp[i - 2] + H[i], dp[i - 1])
    return dp[n - 1]
```

Runtime

O(n) since we have a table of size n with O(1) work at each step

Example 4

Problem

Find the number of paths to reach cell (n, m). Additionally, the grid has bombs, denoted by an input bombs such that bombs[i][j] = True denoted cell (i,j) has a bomb. Find number of paths from (0,0) to (n,m) such that no bombs are traversed.

Solution

Let's define array $T[1 \dots n][1 \dots m]$ where $T[i][j]$ =the number of paths from (1, 1) to (i, j).

Base Cases:

1. If the starting cell $(1, 1)$ has a bomb, then there are 0 paths.

$$dp[1][1] = \begin{cases} 0 & \text{if } bombs[1][1] = \text{True} \\ 1 & \text{otherwise} \end{cases}$$

2. For the first row and first column:

$$dp[i][1] = \begin{cases} 0 & \text{if } bombs[i][1] = \text{True} \\ dp[i-1][1] & \text{otherwise} \end{cases}$$

$$dp[1][j] = \begin{cases} 0 & \text{if } bombs[1][j] = \text{True} \\ dp[1][j-1] & \text{otherwise} \end{cases}$$

Recurrence Relation:
For all cells not on the first row or first column:

$$dp[i][j] = \begin{cases} 0 & \text{if } bombs[i][j] = \text{True} \\ dp[i-1][j] + dp[i][j-1] & \text{otherwise} \end{cases}$$

This means that if a cell has a bomb, no paths can go through it. Otherwise, the number of paths to a cell is the sum of the paths to the cell above it and the cell to its left.

The solution will be in $dp[n][m]$, which gives the number of paths from $(1, 1)$ to (n, m) without traversing any bombs.

Pseudocode

```
def count_paths_with_bombs(n, m, bombs):
    initialize dp as a table of size (n+1)*(m+1) with 0s

    if bombs[0][0]:
        return 0

    dp[0][0] = 1

    for i in 1...n:
        if not bombs[i][0]:
            dp[i][0] = dp[i-1][0]

    for j in 1...m:
        if not bombs[0][j]:
            dp[0][j] = dp[0][j-1]

    for i in 1...n:
        for j in 1...m:
            if not bombs[i][j]:
                dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[n][m]
```

Runtime

O(n x m) because visiting each cell of the grid once.

NP-Completeness

Complexity Theory

- Upper Bound: worst-case performance of an algorithm
- Lower Bound: best possible performance for solving a problem
- **Reduction**: transforming one problem into another to prove properties about the problem
    - Ex: Given algorithm that could sort in O(n) – would have a contradiction with the known lower bound of O(n log n) – therefore has lower bound of Ω(n log n)

Decision vs Search Problems

- Decision: yes/no question
    - Ex: Does there exist a path from s to t in graph G of PATH = {<G, s, t>}
- Search: returns a solution

Complexity Classes

Complexity Class P

- Problems that can be solved in POLYNOMIAL TIME – efficient problems
    - Ex: Finding if two numbers are relatively prime using the GCD algorithm

Complexity Class NP

- Problems where solutions can be **verified** in polynomial time, but finding the solution may not be efficient
    - May not be able to solve in polynomial but can check if it's correct in polynomial time
    - Ex: The problem of checking if a number n is composite. If you are given factors of n, you can verify in polynomial time that the factors multiply to "n"

Relationship between P and NP

- P ⊆ NP: If a problem is in P, it is also in NP because solving the problem can also verify the solution.
- P = = NP is NOT guaranteed because we don't know if problems that are VERIFIABLE in polynomial time are can be SOLVED in polynomial time

NP-Completeness

- If one NP-complete problem can be solved in polynomial time, all NP problems can be solved in polynomial time
    - B ∈ NP: Show that the problem B is in NP (verifiable in polynomial time)
    - B is NP-hard: Prove that an NP-complete problem A reduces to B in polynomial time, denoted A ≤ pB. This means that if you can solve B, you can also solve A.
- So reduce a known NP-complete problem to B
- Steps:
    - Choose a known NP-complete problem A
    - Define a reduction f from A to B. This reduction must be computable in polynomial time.
    - Satisfiability must be preserved: If x ∈ A,, then f(x) ∈ B, and if x ∈/ A, then f(x) ∈/ B

Proving P = NP or P != NP

- **P = NP**: If we find a polynomial-time algorithm for any NP-complete problem (like SAT), it means P = NP
- **P ≠ NP**: If we can show that there are problems in NP that cannot be solved in polynomial time, we will prove P ≠ NP

SAT (Satisfiability Problem)

- **Variable**: A boolean variable (e.g., x1, x2, ...).
- **Literal**: A variable or its negation (e.g., x1 or ¬x1).
- **Clause**: An OR of literals (e.g., x1 ∨ ¬x2).
- **Formula in CNF**: An AND of clauses (e.g., (x1 ∨ ¬x2) ∧ (x3 ∨ x4)

Definition

- A formula is in SAT if it has a truth assignment that makes the formula true
    - (x ∨ y ∨ z) ∧ (x ∨ z ∨ w) ∧ ...can be satisfied by assigning appropriate values to variables
- Every problem in NP can be reduced to SAT, meaning SAT represents the entire NP class — if SAT can be solved in polynomial time, then P = NP

kSAT/3SAT

- **kSAT**: SAT problem where each clause has at most k literals
- **3SAT**: SAT problem where each clause has 3 literals

Proving 3SAT to be NP-Complete

- **3-SAT ∈ NP**: The witness is simply the variable assignments, which can be checked in polynomial time
- **SAT ≤p 3-SAT**: If a SAT formula has clauses with more than 3 literals, you can break them down into smaller clauses by introducing new variables (e.g., convert (x1 ∨ x2 ∨ ⋯ ∨ xk) into two clauses).
- **Conclusion**: SAT reduces to 3-SAT, so 3-SAT is NP-complete.

2SAT/Why its P

- **2SAT**: case where each clause has 2 literals
- **2-SAT ∈ P**: You can reduce 2-SAT to graph problems. A 2-SAT problem can be solved using graph reachability techniques, making it solvable in polynomial time.

Example

- A clause (a ∨ b) can be viewed as implications: ¬a→b and ¬b→a

CircuitSAT

- Given a Boolean circuit with AND/OR/NOT gates, determine if there is an input that makes the output 1

Proving its NP-Complete

- **CircuitSAT ∈ NP**: A verifier can evaluate the circuit with a given input in polynomial time
- **3-SAT ≤p CircuitSAT**: Convert a 3-SAT formula into a boolean circuit. Each clause in 3-SAT becomes a sub-circuit, and the entire formula becomes a circuit with AND gates combining the clauses
- If a 3-SAT formula is satisfiable, the corresponding circuit is satisfiable, making CircuitSAT NP-complete.

Independent Set

Problem Definition

Given a graph G(V,E) and an integer g, determine if there exists a subset of vertices I ⊆ V with size g such that no two vertices in II are connected by an edge.

NP-Complete Proof

- Proving NP:
    - Verify a solution by checking if the set has size g and ensuring no edges exist between any two vertices in the set.
- NP-Hardness
    - For each clause in a 3SAT formula, create a “triangle” of vertices labeled by the literals of that clause
    - Add edges between corresponding literals and their negations across different triangles
    - The graph has an independent set of size g (number of clauses) if and only if the 2SAT formula is satisfiable
- Since Independent Set is both NP and NP hard it is NP-Complete

Clique

Problem Definition

Given a graph G(V, E) and an integer g, determine if there exists a subset of vertices that forms a complete subgraph (clique) of size g.

NP-Complete Proof

- Proving NP:
    - Verify a solution by checking if the set has size g and ensuring that every pair of vertices in the set is connected
- NP-Hardness
    - Reduce from Independent Set
        - Convert the graph G to its complement G'
        - A clique of size g in G' corresponds to an independent set of size g in G, and vice versa
    - So Clique is as hard as solving the Independent Set Problem
- Since Clique is both NP and NP hard it is NP-Complete

Vertex Cover

Problem Definition

Given a graph G(V,E) and an integer g, determine if there exists a subset of vertices of size g such that every edge in the graph has at least one endpoint in this subset.

NP-Complete Proof

- Proving NP:
    - Verify a solution by checking if the set has size g and ensuring that every edge is covered by at least one vertex from the set
- NP-Hardness
    - Reduce from Independent Set
        - If a graph G has an independent set of size g, then the complement of the set (remaining vertices) forms a vertex cover of size $|V| - g$, and vice versa
- Since Vertex Cover is both NP and NP hard it is NP-Complete

Subset Sum

Problem Definition

Given a set S and a target t, find a subset $S' \subseteq S$ such that the sum of elements in S' equals t.

NP-Complete Proof

- Proving NP:
    - If given a subset S', summing the elements to check if they equal t can be done in polynomial time
- NP-Hardness
    - Reduce from 3SAT
        - Encode each variable and its negation as binary numbers
        - Create a target number where each bit corresponds to the presence of either a variable or its negation
        - For a 3SAT instance, build a table where rows represent literals, and columns represent their occurrences in clauses
        - Clauses: ensure each has at least one true literal by setting target values that restrict the total to exactly three
    - Correctness:
        - If 3SAT is satisfiable, Subset Sum solution exists
        - If 3SAT is unsatisfiable, no subset will sum to the target
- Since Subset Sum is both NP and NP hard it is NP-Complete

Knapsack

Problem Definition

Given items with weights and values, determine if there's a subset whose total weight doesn't exceed a capacity W, and whose total value meets or exceeds V

NP-Complete Proof

- Proving NP:
    - Given a subset of items, checking if the weight and value constraints are met can be done in polynomial time
- NP-Hardness

    - Reduce from Subset Sum
        - For each element in Subset Sum, create a corresponding item where weight equals value
        - Set the capacity W and value V in Knapsack to the target sum t from Subset Sum
        - If a subset in Subset Sum sums to t, the corresponding items in Knapsack will meet both the weight and value constraints, proving the reduction
- Since Knapsack is both NP and NP hard it is NP-Complete

Onions

Divide and Conquer

Max's Anger Contest Series 1 P3 – Divide and Connor

View the Problem Here!

Solution: view here

Explanation: view here

Hints:

1. Think about how to break the array into three equal parts. If you're unfamiliar with Arrays.copyOfRange, you can simply iterate to divide the array into smaller parts.
2. Once you've split and processed the smaller segments, make sure you reorder them according to the problem's instructions: third part -> first, first part -> second, second part -> third.
3. After splitting and reordering, merge the segments back together into a new array.

ICPC NEERC 2012 C - Caravan Robbers

View the Problem Here!

Solution: view here

Explanation: view here

Hints:

1. Sort the intervals by their starting point to simplify processing.
2. Use binary search to find the maximum possible length of the intervals by testing different lengths.
3. For each length tested in binary search, make sure all gangs can fit their intervals without overlapping by checking against their starting and ending points.
4. Use the gcd to convert the interval length into an irreducible fraction for the final result.

Age Demographic

View the Problem Here!

Solution: view here

Explanation: view here

Hints:

1. **Sort the Ages**: Sorting the list of ages makes it possible to efficiently find the range of ages that meet the query condition.
2. **Binary Search**: Use Arrays.binarySearch() to find the positions of a and b in the sorted array. If a or b are not in the array, use the insertion point given by binarySearch() to get the correct range.
3. **Counting the Range**: Once you have the positions for a and b, calculate the difference between them to get the number of viewers within that age range.
4. **Edge Cases**: Handle edge cases where a or b might not exist in the array, or when all viewers are outside the queried range.

Numbers

TLE '17 Contest 7 P3 - Countless Calculator Computations

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Use binary search to find X
2. Make a separate function to calculate the result of the repeated exponentiation up to Y levels
3. Use a small epsilon in the binary search

DWITE '07 R5 #2 - Number of factors

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Understand Prime Factorization: The function breaks down the number into its prime factors, excluding 1 and the number itself
2. Handle Factors of 2 First: This simplifies the process, allowing the loop to focus only on odd factors afterward.
3. Check for Remaining Prime: After dividing out all smaller factors, if a prime remains, count it.
4. Adjust for Primes: If the number is prime and has no other factors, adjust the count accordingly to return 0.

DMOPC '20 Contest 3 P5 - Tower of Power

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Use Euler's Totient Function to reduce exponents in modular calculations.
2. Apply Modular Exponentiation to handle large powers efficiently.
3. Calculate Tower Power from top to bottom, using reduced moduli at each step.

Graphs

CCC '10 J5 - Knight Hop

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:`

1. Use BFS for Shortest Path: Since the knight moves in all directions and can revisit positions, BFS ensures the shortest path is found.
2. Track Visited Positions: Prevent revisiting squares by using a visited set, which helps in reducing redundant calculations.
3. Check Boundaries: Always ensure that the knight stays within the 8x8 chessboard boundaries during each move.

ACSL '09 Practice P4 – Rank

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Model the Problem as a Graph: Players are nodes, and victories create directed edges.
2. Use DFS to Detect Cycles: Keep track of nodes in the current path (recursion stack) to find cycles.
3. Count Unique Players in Cycles: Ensure each player in a cycle is only counted once

Is it a Tree?

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Check Edge Count: Ensure the number of edges matches n-1, which is necessary for a tree.
2. Validate Symmetry: The adjacency matrix must be symmetric since the graph is undirected.
3. Use BFS/DFS: Traverse the graph to check for connectivity and cycles, which will help determine if it is a valid tree.

IOI '11 P2 - Race (Standard I/O)

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Understand Centroid Decomposition: It's crucial for breaking down the tree into manageable parts while maintaining efficient query handling.
2. Distance Pairing: Focus on finding pairs of distances that sum to K using DFS.
3. Optimize Using Stored Distances: Store distances from each centroid and use them to quickly find valid paths.
4. Recursive Processing: Apply centroid decomposition recursively to handle all parts of the tree.

IOI '13 P1 - Dreaming (Standard I/O)

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Understand Tree Diameter: Ensure you can find the longest path in a tree using two BFS operations.

2. Tree Radius Insight: Be familiar with how tree radius helps in calculating the maximum distance when adding new trails.
3. Combination of New Trails: Consider different ways to connect key nodes with new trails to minimize the maximum distance between any two billabongs.
4. Efficiency: Pay attention to the graph traversal techniques and how they help in efficiently finding the solution.

Baltic OI '05 P3 – Maze

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Familiarize yourself with how BFS can be adapted to handle different edge conditions, such as alternating colors in this case.
2. Recognize that you need to track the state of the last edge color to properly apply the movement rules.
3. Ensure you can correctly represent and initialize the graph based on the maze's input format.
4. Understand how the distance array is used to maintain and update shortest path distances with state-specific conditions.
5. Ensure the solution handles cases where no valid path exists by checking if the distance remains at its initialized value.

RGPC '17 P4 - Snow Day

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Use Topological Sort for DAGs: Process nodes in topologically sorted order to handle longest path calculations.
2. Dynamic Programming: Apply dynamic programming to update distances and node counts along the longest path.
3. Handle No Path Cases: Ensure to check and handle cases where no path exists between the start and end nodes.

Single Source Shortest Path

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Use of Adjacency Matrix: Efficiently stores edge weights for dense graphs but can be memory-intensive for large sparse graphs.
2. Dijkstra's Implementation: Select the unprocessed node with the smallest distance and update its neighbors.
3. Handling Unreachable Nodes: Use Integer.MAX_VALUE to indicate nodes that are not reachable from the source.

Another Random Contest 1 P5 - A Strange Country

View the Problem Here!

Solution: View Here
Explanation: View Here

Hints:

1. Union-Find with Path Compression: Efficiently manage and merge sets of connected nodes.
2. Binary Search for Optimization: Quickly determine the earliest day a road can be used without causing a cycle in the MST.
3. Sorting by Weight: Ensures that the MST is built in increasing order of edge weights, adhering to Kruskal's algorithm principles.

EZDP

IOI '94 P1 - The Triangle

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Bottom-Up Calculation: Update each element based on its two possible paths to the next row, ensuring that the maximum possible sum is considered.
2. In-Place Update: Modify the triangle matrix directly to save space and simplify the implementation.
3. Starting Point: Begin updating from the second-to-last row and move upwards, ensuring that all necessary sub-problems are solved before considering the current row.

IOI '07 P4 – Miners

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Encoding: Use state encoding to manage the last two shipments efficiently and compute transitions.
2. Transition Precomputation: Precompute state transitions to simplify the DP update process.
3. Reverse Processing: Process shipments from last to first to build the DP table correctly based on future shipments.
4. Optimization: Keep track of coal production through state transitions to optimize the allocation of shipments to maximize coal production.

Educational DP Contest AtCoder A - Frog 1

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Definition: Define dp[i] as the minimum cost to reach stone i.
2. State Transition: Use previous stone costs to compute the current stone's cost.
3. Initialization: Initialize the costs for the first two stones and compute subsequent costs iteratively.

Educational DP Contest AtCoder H - Grid 1

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Definition: Define dp[i][j] as the number of ways to reach cell (i, j).
2. State Transition: Sum the number of paths from top and left cells.
3. Boundary Conditions: Handle blocked cells (#) where no paths should be added.

Educational DP Contest AtCoder B - Frog 2

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Definition: Define dp[i] as the minimum cost to reach stone i.
2. State Transition: Update dp[i] by considering all reachable previous stones j and calculating the cost based on the height difference.
3. Efficiency: Iterate from each stone to all stones within K jumps to ensure you find the minimum cost.

IOI '04 P4 – Phidias

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Definition: Define dp[w][h] to track the minimum waste for a slab of size w × h.
2. State Transition: Consider all possible vertical and horizontal splits to update the DP table.
3. Efficiency: Check if a plate size is needed to avoid wasting the slab of that size.

IOI '14 Practice Task 1 – Square

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Initialization: Start with dp table filled with zeros.
2. Edge Cases: Handle cells on the grid's boundary separately as they can only form squares of size 1.
3. Updates: Update max_square and count_max_square whenever a larger square is found or an additional square of the same size is detected.

Knapsack

DMOPC '17 Contest 2 P1 - 0-1 Knapsack

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Filtering: Items with zero or negative values are irrelevant for maximizing value.
2. Capacity Calculation: The minimum knapsack size needed is the sum of capacities of all valuable items, ensuring no valuable item is left out.

Educational DP Contest AtCoder D - Knapsack 1

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Sorting: Items are sorted by value to potentially prioritize high-value items, but this step is not strictly necessary for correctness.
2. Reverse Iteration: Process capacities in reverse order to avoid using the same item multiple times within a single iteration.
3. DP Transition: Update dp[j] by checking if including the item results in a higher value than the current maximum for capacity j

Knapsack 4

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Strength Balancing: By using negative values for engineers, the problem reduces to a single knapsack problem with an offset.
2. DP Table Size: The DP table is sized based on the maximum possible strength balance, ensuring all potential balances are covered.
3. Shuffling Items: Randomize the order of processing items to improve performance in practice.

NP-Completeness

ICPC PACNW 2016 F – Illumination

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. 2-SAT Representation: Use 2-SAT to encode "either/or" constraints for each lamp's illumination choice.
2. Implication Construction: If two lamps could overlap, they cannot both be in the same mode (row or column).
3. SCC Check: If a lamp's row and column options are in the same SCC, it means there's a conflict, making the configuration invalid.

COCI '13 Contest 2 #4 Putnik

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. State Representation: The state dp[left][right] captures the minimum cost to cover remaining cities while satisfying the constraints.
2. Recursive Transitions: Consider the cost of including the next city in either the segment before right or the segment after right and choose the minimum.

Wesley's Anger Contest 1 Problem 5 - Rule of Four

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. SCC Finding: Tarjan's algorithm is used to decompose the implication graph into SCCs.
2. Transitive Closure: Helps in understanding reachability and ensuring all constraints are respected.
3. Consistency Check: Ensures no contradictory assignments and computes the possible trip configuration.

Miscellaneous

Mock CCC '20 Contest 1 J2 - A Simplex Problem

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Iterate Valid Combinations: Check all combinations where the sum of cacti and matroids does not exceed k.
2. Compute Joy: For each combination, compute the total joy and compare it to the maximum joy found so far.

Wesley's Anger Contest 1 Problem 1 - Simply a Simple Simplex Problem

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. No hints this time!

System Solver

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Gaussian Elimination: Used to transform the system into an upper triangular form for easy back substitution.

2. Pivoting: Ensures numerical stability by swapping rows to place non-zero elements on the diagonal.
3. Rank: Helps determine the feasibility of a unique solution based on the number of non-zero rows compared to the number of variables.

DMOPC '21 Contest 9 P5 - Chika Circle

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Check neighbor values to infer card values directly when possible.
2. Use the constraints of the problem to narrow down possible card assignments.
3. Ensure that the final assignment matches all the provided responses.

WC '99 P3 - The Godfather

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. No hints this time!

Wesley's Anger Contest Reject 5 - Two Permutations

View the Problem Here!

Solution: View Here

Explanation: View Here

Hints:

1. Use of Maps: first and second maps track which index produced which value for quick lookup.
2. Random Indices: Randomly select indices to guess, ensuring diverse coverage.
3. Efficiency: The nested loop checks for matches between stored values efficiently.

LeetCode