Computer Science Compendium

Scott Tolksdorf

January 8, 2014

Contents

1 Introduction

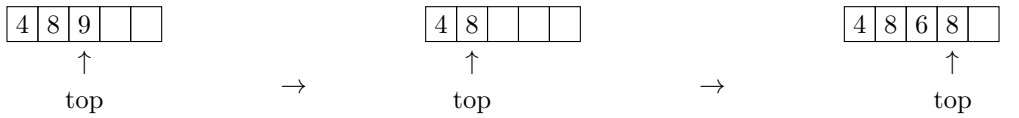
2 Data Structures

2.1 Stacks

CLRS pg.232 A Stack is a dynamic set of data. It follows a **Last-In First-Out** ordering system. Imagine a stack of plates, where you can only add to the stack and the top and only remove plates from the top. **Used in** algorithms in converting decimal numbers to binary, evaluating math expressions, maze back-tracking solutions. Most languages used stacks to resolve operations. **Insert/Delete**: $O(1)$.

Method: The data structure uses **Push** to insert new data and **Pop** to remove data. It uses a **Top** pointer to keep track of the data structures position in memory.

```
Push(x){
    S.top = S.top + 1
    S[S.top] = x
}

Pop(){
    S.top = S.top - 1
    return S[S.top + 1]
}
```



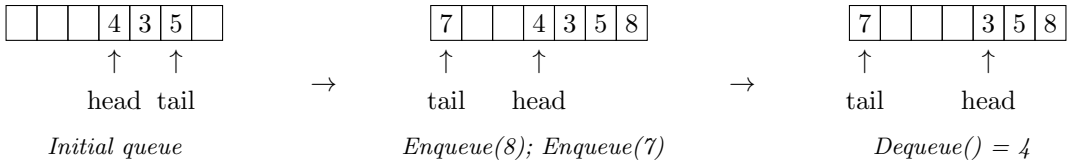
Initial stack *Pop() = 9* *Push(6); Push(8)*

2.2 Queues

CLRS pg.232 A Queue is a dynamic set of data. It follows a **First-In First-Out** ordering system. Imagine a checkout at a store, you enter at the end of the queue and only leave at the front. To maximize memory use, queues are sometimes implemented cicular in nature. **Used as** priority queues, where elements are added with a priority and removed in order of their priority. Dijkstra's Algoirthm and A* use priority queues to track the most efficient ways to solve their problem. **Insert/Delete**: $O(1)$.

Method: The data structure uses **Enqueue** to insert new data and **Dequeue** to remove data. It uses a **Head** and **Tail** pointer to keep track of the data structures position in memory.

```
Enqueue(x){
    Q[Q.tail] = x
    if Q.tail == Q.length
        Q.tail = 1
    return Q.tail = Q.tail + 1
}
Dequeue(){
    x = Q[Q.head]
    if Q.head == Q.length
        Q.head = 1
    else
        Q.head = Q.head + 1
    return x
}
```



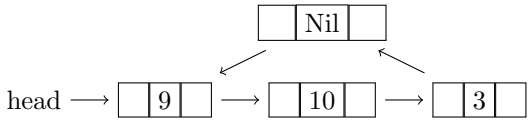
Initial queue → *Enqueue(8); Enqueue(7)* → *Dequeue() = 4*

2.3 Linked Lists

CLRS pg.236

In Linked Lists data is stored linearly and sequenitially. Each node contains a pointer to the next node in the list. In a **Double Linked List** each node also has a pointer to it's parent. Better then dynamic arrays for inserts/deletes and since it uses pointers, the data structure can be spread across memory. However, you can not random access a linked list, in order to get an element you must transverse the list to get there. Linked lists use slightly more memory per node to track the pointers. **Search:** $O(n)$; **Insert/Delete**: $O(1)$.

```
Search(k){
    x = L.head
    while x != Nil && x.key != k
        x = x.next
    return x
}
Insert(x){
    x.next = L.head
    if L.head != Nil
        L.head.prev = x
    L.head = x
    x.prev = Nil
}
Delete(x){
    x.prev.next = x.next
    x.next.prev = x.prev
}
```



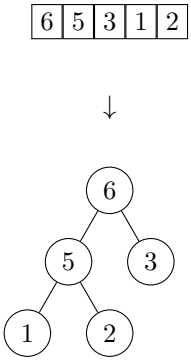
2.4 Heaps

CLRS pg.151

Heaps are a data structure which create a tree-like structure using sequential memory. They are defined by a **Heap Property** which dictate how the heap is formed, eg. max-heap property: Parent nodes are always larger then their children. The internal structure of a heap may be largely unordered, but the heap property ensures the top of the heap is the max element of the data structure. Heaps are useful for any scenerio where simply knowing the largest element is needed, eg. **Priority Queues**. Using properties of sequiential access, node navigation can be done using math on the node's index.

Method: Heapify ensures that the node at the given index is following the heap property, if not it will "float down" the data structure until it does. When extracting the max value from a heap, replace it with the last value in the heap, and Heapify from the top.

```
Parent(i) => floor(i/2)
Left(i) => 2i
Right(i) => 2i + 1

BuildHeap(A){
    for i = floor(A.length/2) -> 1
        Heapify(A, i)
}
Heapify(A, i){
```



```
    if A[Left(i)] > A[i]
        largest = Left(i)
    else
        largest = i
    if A[Right(i)] > A[largest]
        largest = Right(i)
    if largest != i
        swap A[i] with A[largest]
        Heapify(A, largest)
}
Extract(A){
    max = A[0]
    A[0] = A[A.length]
    Heapify(A,0)
    return max
}
```

2.5 Hashtables

CLRS pg.257 Hashtables is a data structure that map keys to values in an associated array using a **Hash Function**. A hash function should be choosen that limits clumping

A hash tables **Load Factor** is the ratio of the number of elements in the data structure against it maximum size. When a hash table has a high load factor it has to be resized in order to avoid collisions. Resizing can be done one of two ways; **Rebuilding** the entire hash table at once using a bigger size, or **Incremental** where a new table is allocated, new values are only inserted into the new table, as well as moving over n other elements from the old array. When the old table is empty it's removed from memory.

A technique to speed up deletes is to use **Tombstones**. A tombstone is a special entry that is inserted in place of an element you wish to delete. It's ignored during lookups, and replaced during inserts.

Collisions resolution could be done one of several ways: **Chaining**, **Open Addressing**, **Cuckoo**

2.6 Problems

1. Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures

2. Given two strings, write a method to decide if one is a permutation of the other.

3. Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees. Can you do this in place?

4. Write an algorithm such that if an element in an MxN matrix is 0, its entire row and column are set to 0.

5. Implement an algorithm to find the kth to last element of a singly linked list.

6. Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.

7. Implement a function to check if a linked list is a palindrome

8. Implement a queue class which implements a queue using two stacks.

9. Write a program to sort a stack in ascending order (with biggest items on top). You may use at most one additional stack to hold items, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

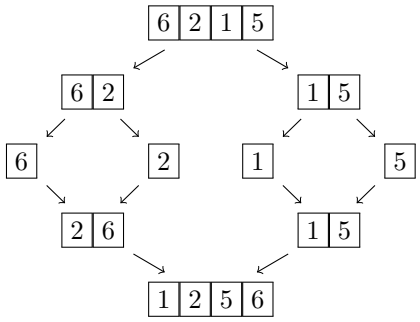
3 Sorting

3.1 MergeSort

CLRS pg.31 MergeSort is a **Divide and Conquer algorithm**, it's recursive and can be parallelized well between multiple processors. It has **Stable Sorting**, so items with the same value perserve their order from the original list. **Avg/Worst Case**: $O(n \log n)$; **Aux. Space**: $\Omega(n)$. **Best used** when accessing data sequentially is important, eg. parallel/external sorting. On average slower then HeapSort and QuickSort.

Method: MergeSort splits its list until it's down to 1 element, then rejoins them recursively inorder.

```
MergeSort(list){
    if length(list) <= 1
        return list
    split list -> A, B
    A = MergeSort(A), B = MergeSort(B)
    loop through each A, B and merge in sorted order
    return result
}
```

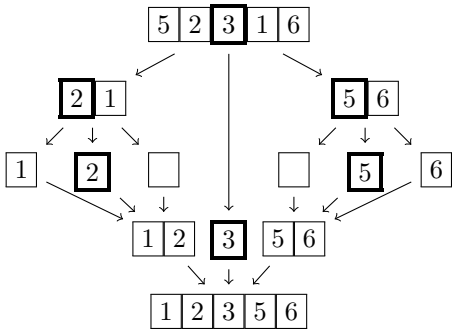


3.2 QuickSort

CLRS pg.170 QuickSort is a **Divide and Conquer algorithm**, it's recursive and can be parallelized well between multiple processors. It has **Non-Stable Sorting**, so items with the same value do not perserve their order from the original list. **Avg Case**: $O(n \log n)$; **Worst Case**: $O(n^2)$; **Aux. Space**: $\Omega(\log n)$. **Best used** when speed is top priority. Can have cases where running time is very slow, bad for very large data sets and possible for attacks.

Method: QuickSort chooses a pivot, then creates two new lists, one containing elements less then the pivot and one greater then the pivot. Recursively applies this to the new lists. Rejoins them with the pivot.

```
QuickSort(list){
    if length(list) <= 1
        return list
    select and remove a pivot from list
    create empty lists -> left, right
    for each x in array
        if x <= pivot
            append x to left
        else
            append x to right
    return join(QuickSort(left), pivot, QuickSort(right))
}
```

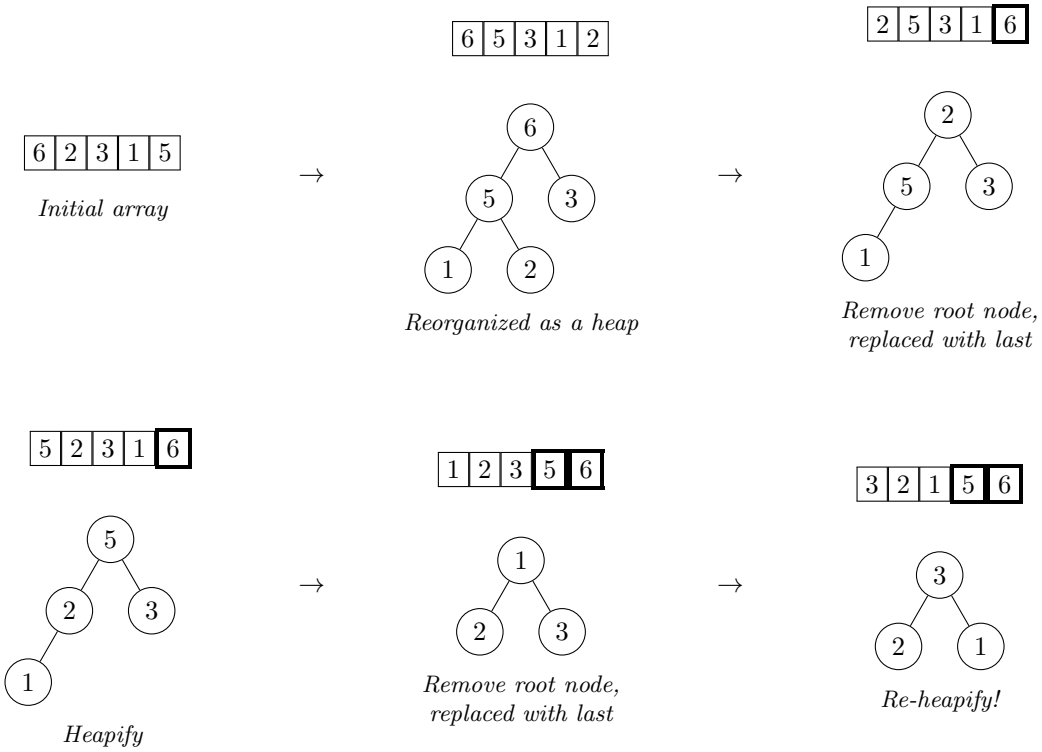


3.3 HeapSort

CLRS pg.151 Heapsort is a comparison-based sorting algorithm. It has **Non-Stable Sorting**, so items with the same value do not perserve their order from the original list. **Avg/Worst Case**: $O(n \log n)$; **Aux. Space**: $\Omega(1)$. **Best used** when space is a concerned, eg. embedded systems. On average runs slower then QuickSort, but faster then MergeSort.

Method: HeapSort first builds a heap out of the data, the iteratively removes the largest elements from the heap and stores it in an array, then rebuilds the heap. Repeat until the heap is exhausted.

```
HeapSort(A){
    BuildHeap(A)
    for length(A)
        result = Extract(A)
        replace with last in heap
        reduce heap size by 1
        Heapify(A)
    return result
}
```



Example of heaping an array

3.4 Problems

1. Write a method to sort an array of strings so that all the anagrams are next to each other.
2. Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.
3. Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.
4. A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

```
Example
Input (ht,wt): (65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)
Output:The longest tower is (56, 90) (60,95) (65,100) (68,110) (70,150) (75,190)
```

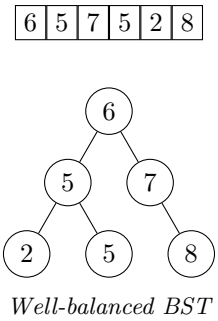
4 Trees

4.1 Binary Search Trees

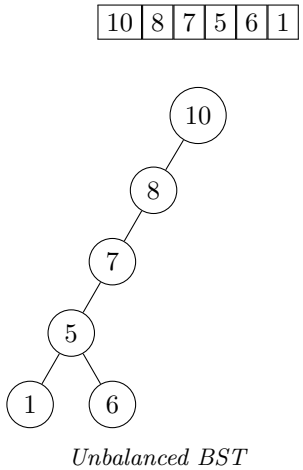
CLRS pg.287

Binary Trees are family of data structures efficient at lookups. Data is stored that $Right \leq Node \leq Left$. BSTs are **unbalanced**, meaning one part of the tree may become much larger then the other, hurting the efficiency. **Search/Insert:** $O(n \log n)$; **Delete:** $O(n)$.

```
Search(node, val){
    if node == Nil or node.key == val
        return node
    if val > node.key
        return Search(node.left, val)
    else
        return Search(node.right, val)
}
//TODO: chage to recursive insert
Insert(T, val){
    x = T.root
    while x != Nil
        if val > x.key
            x = x.left
        else
            x = x.right
    if val > x.parent.key
        x.left = val
    else
        x.right = val
}
Delete(T, node){
    if node.left == Nil and node.right == Nil
    //TODO
}
```



Well-balanced BST



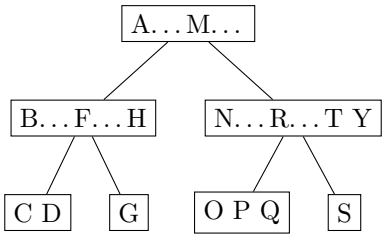
Unbalanced BST

4.2 B-Trees

CLRS pg.488

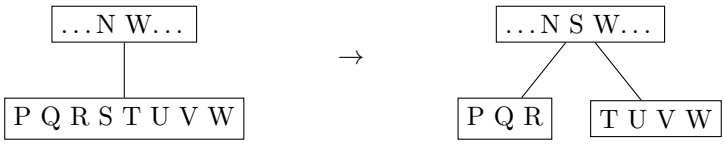
B-Trees are an extreme form of **k-ary Trees**, where k is very large. Each node has a minimum and maximum number of children it can have; $t-1$ and $2t-1$ respectively, where t is the **order** of the tree. B-Trees are achieve extremely large tree structures with a small height when the order of the tree is high, number of nodes is $2t^h - 1$ where h is the height of the tree. **Search/Insert/Delete:** $O(\log n)$.

Best Used for when lookups are very expensive and when lookups are best done in large sequiential blocks. Mostly used for storing data on **physical storage**.



B-tree with order of 2

Inserting a node is slightly more complicated. When inserting a new value into a node that is full, we must split the node into smaller chunks. With deleting a value, you may have to rearrange a node's children.

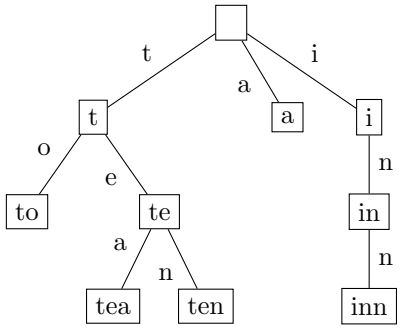


Splitting a child node into 2 on S because it is too large

4.3 Tries

A Trie is an unique tree data structure where a node's location within the tree determines its value. Each edge of the tree has a value given to it and as you traverse the tree you build the value of the node. Very useful for **storing strings** and **binary values**.

Tries have a **faster worst case than hash tables** because they don't need to rebuilds, they can produce **alpha-ordering** which may be useful for some applications like spell checking, and there is **no chance for collisions**. However, Tries may need more memory, long entires can create useless nodes, and they require **lots of random access memory** to operate. **Search/Insert/Delete:** $O(k)$ where k is the length of the longest node value.

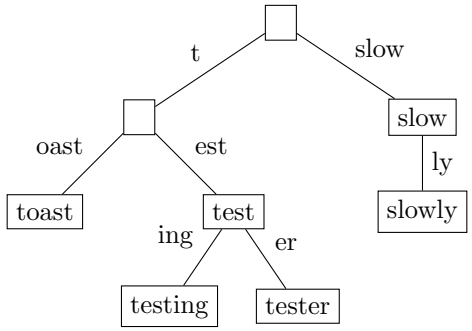


Building a Trie for "to", "tea", "ten", and "inn"

4.4 Radix Tree

CLRS pg.304 Radix Tress are space optimized **Trie** where each node with only one child is merged with its parent, and its edge is updated accordingly. They are **more efficient for small sets** especially if the strings are long and for sets of strings that share long prefixes. **Search/Insert/Delete:** $O(k)$ where k is the length of the longest edge key.

Best used in IP Routing, where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of IP addresses. They are useful for **Inverted Indexes** in text documents for very fast searching through the document.

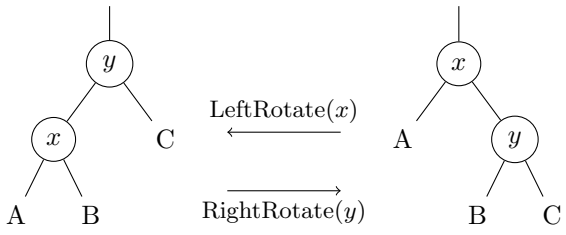


Building a Radix Trie for "toast", "test", "testing", "tester", "slow", and "slowly"

4.5 Self-Balancing Trees

CLRS pg.308 A Self-Balancing Tree is a binary search tree that keeps its height small during inserts and deletes. Binary Search Trees have a very bad worst-case; Self-Balancing Trees attempt to correct this by using **Tree Rotations** to maintain tree properties that ensure a predictable height of the tree.

Whenever a node is inserted or deleted, we need to check the node's children for consistency of the height property. **Avg/Worst** of **Search/Insert/Deletes:**$O(\log n)$. The two most common Self-Balancing Trees are **AVL Trees** and **Red-Black Trees**.



4.5.1 AVL Trees

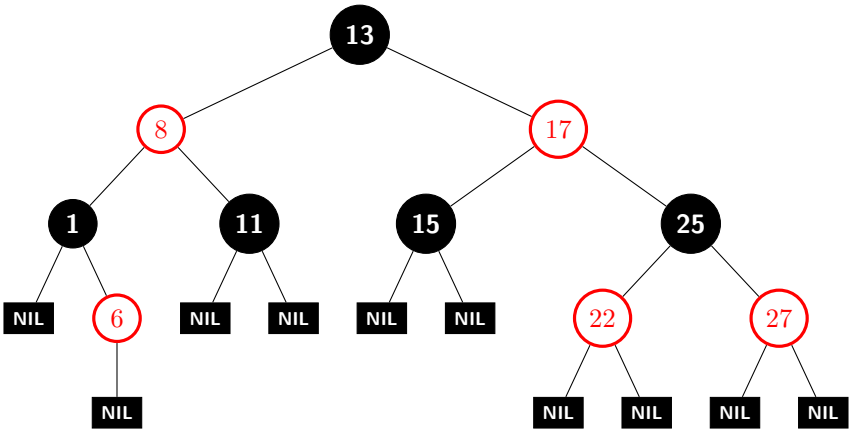
The height property of an AVL Tree is that **the heights of the two child subtrees of any node differ by at most one**. AVL Trees are more rigidly balanced than Red-Black Trees, leading to **slower inserts and deletes** but **faster search**.

```
balanceFactor(node) => height(node.left) - height(node.right)
Insert(node, val){
    if node is Nil
        node.key = val
        return node
    if val > node.key
        node.left = Insert(node.left, val)
        if balanceFactor(node) > 1
            if val > node.left.key
                LeftRotate(node)
            LeftRotate(node)
    if val <= node.key
        node.right = Insert(node.right, val)
        if balanceFactor(node) < -1
            if val < node.right.key
                RightRotate(node)
            RightRotate(node)
    return node
}

Delete(z){
    //finish later, very complex
}
```

4.5.2 Red-Black Trees

Red-Black Trees are less rigidly balanced than AVL Trees, leading to **slower search** but **faster inserts and deletes**. The height properties of a Red-Black Tree are as follows:

1. A node may be red or black
2. All leaves are black
3. Every red node must have two black child nodes
4. Every path from the root to a leaf must have the same number of black nodes.



```
//TODO: Make recursive
Insert(z){
    find leaf where node should be attached -> y
    z.parent = y
    if z.key > y.key
        y.left = z
    else
        y.right = z
    z.color = red

    //Maintain height property
    while z.parent is red
        if z.parent is a left-child
            if z.uncle is red
                z.parent.color = black
                z.uncle.color = black
                z.grandparent.color = red
                z = z.grandparent
            if z.uncle is black and z is a right-child
                z = z.parent
                LeftRotate(z)
            if z's uncle is black and z is a left-child
                z.parent.color = black
                z.grandparent.color = red
                RightRotate(z.grandparent)
        else
            Same as above but switch left and right rotate
    root.color = black
}
Delete(z){
    //finish later, very complex
}
```

4.6 Bloom Filter

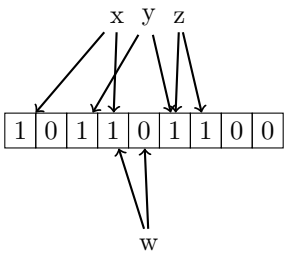
A Bloom Filter is a space efficient probalistic data structure that's used to test whether an element is in a set. False positives are possible, but **false negatives** are not. An empty Bloom Filter is a bit array of m bits all set to 0. Whenever you add an element to the set use k hash functions and flip all the corresponing entires to 1. To test if an entry belongs to the set, simply hash it and check the corresponding bit entires. If any of them are 0 the element is not part of the set. The probability of a **false positive** is $(1 - e^{-kn/m})^k$, where n is the number of elements encoded in the set.

Used in Chrome to detect unsafe urls from a master list, BigTable to skip unnecessary table lookups, and in url shortners.

4.7 Problems

1. Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

2. Implement a function to check if a binary tree is a binary search tree.



x, y, z are in the set, where w is not

3. Write an algorithm to find the'next'node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.

4. You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path does not need to start or end at the root or a leaf.

5. You have two very large binary trees: Tl, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide if T2 is a subtree of Tl.

   A tree T2 is a subtree of Tl if there exists a node n in Tl such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

6. Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.

5 Bit Manipulation

Bit Manipulation is the act of manipualting individual bits within data. It can reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are **processed in parallel**, but the code can become rather more difficult to write and maintain. **Used in** low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization.

5.1 Operations

Bit manipulation supports a number operations: OR |, AND & , XOR ^ , SHIFT ¡¡ , and NOT ~ .

```
0110 | 0001 = 0111
1000 & 1011 = 1000
1100 ^ 1010 = 0110
1101 >> 2   = 0011
~1001       = 0110
1011 & (~0 << 2) = 1011 & 1100 = 1000
```

Arthemetic operations are also supported.

```
0110 + 0010 = 1000
0110 - 0011 = 0101
0011 * 0101 = 1001
```

5.2 Common Tasks

The following are common tasks you would need to implement while manipualting bits.

```
//Retrieve the value of a bit at the ith position
getBit(n, i){
    return (n & (1 << i)) != 0
}
    getBit(11001101, 3)
    = (11001101 & (1 << 3)) != 0
    = (11001101 & 00001000) != 0
    = (11001101 & 00001000) != 0
    = 00001000 != 0
    = 1

//Sets the value of a bit at the ith position to 1
setBit(n, i){
    return n | (1 << i)
}
    setBit(11001101, 1)
    = 11001101 | (1 << 1)
    = 11001101 | 00000010
    = 11001111

//Sets the value of a bit at the ith position to 0
clearBit(n, i){
    return n & ~(1 << i)
}
    clearBit(11001101, 6)
    = 11001101 & ~(1 << 6)
    = 11001101 & ~(01000000)
```

```
    = 11001101 & 10111111
    = 10001111

//Sets the value of a bit at the ith position to u
updateBit(n, i, u){
    mask = ~(1 << i)
    return (n & mask) | (u << i)
}
    updateBit(11001101, 5, 1)
    = (11001101 & ~(1 << 5)) | (1 << 5)
    = (11001101 & 11011111) | 00100000
    = 11001101 | 00100000
    = 11101101
```

5.3 Problems

1. Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

2. Explain what the following code does: `((n & (n-1)) == 0)`.

3. Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit 0 and bit 1 are swapped, bit 2 and bit 3 are swapped, and so on.

4. Write a function that adds two numbers. You should not use any arithmetic operators.

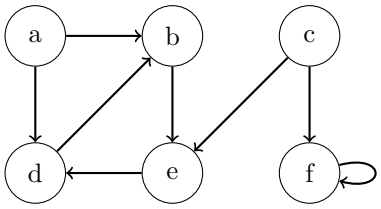
5. A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte.The screen has width w, where w is divisible by 8 (that is, no byte will be split across rows).The height of the screen, of course, can be derived from the length of the array and the width. Implement a function `drawHorizontalLine(byte[] screen, int width, int xl, int x2, int y)` which draws a horizontal line from (xl, y) to (x2, y).

6 Graph Algorithms

6.1 Representation

CLRS pg.308

A graph is a collection of nodes with values that are connected to eachother. **Undirected Graphs** simply have links betwen two nodes, where **Directed Graphs** have one-way links between nodes. We can represent a graph in one of two ways; **Adjacency Lists** are useful for when the graph is **sparse**, few edges compared to nodes. **Adjacency Matricies** are useful for when the graph is **dense**, many edges compared to nodes.



Directed Graph

$a \to b, d$
$b \to e$
$c \to e, f$
$d \to b$
$e \to d$
$f \to f$

Adjacency List

	a	b	c	d	e	f
a	0	1	0	1	0	0
b	0	0	0	0	1	0
c	0	0	0	0	1	1
d	0	1	0	0	0	0
e	0	0	0	1	0	0
f	0	0	0	0	0	1

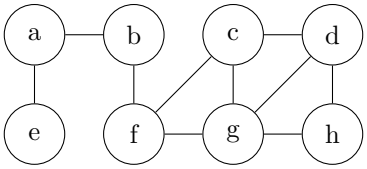
Adjacency Matrix

6.2 Breadth-First Search

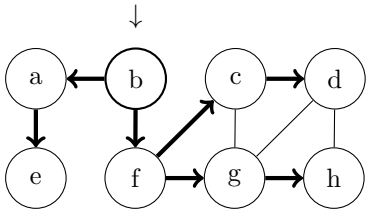
CLRS pg.594

Breadth-First Search is a strategy for searching a graph. It starts at the root node, visits all of it's neighbours, then for each neighbour, goes to their neighbours, etc. As it runs it produces a **Breadth-First Tree**, where the depth of the tree indicates how far apart the root and that node is in the graph.

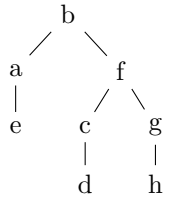
Worst Case: $O(V + E)$, where $O(E)$ may vary between $O(V)$ and $O(V^2)$ depending on how dense the graph is. **Aux. Space** is Adj. List:$\Omega(V + E)$ and Adj. Matrix:$\Omega(V^2)$. **Used for** finding the shortest path between nodes and asserting if two nodes are connected to eachother. **Useful** for when needing to find a specific node, without visitng the entire tree.



Initial undirected graph



BFS starting at node b



Breadth-first tree

```
BFS(start, goal){
    create queue Q
    create list R
    Q.enqueue(start)
    R.push(start)

    while Q is not empty
        t = Q.dequeue()
        t.visited = true
        if t is goal
            return R
        for all neighbours v of t
```

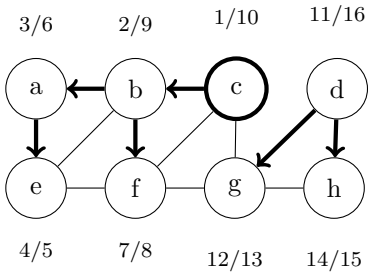
```
            if v.visited == false
                v.visited = true
                R.push(v)
                Q.enqueue(v)
}
```

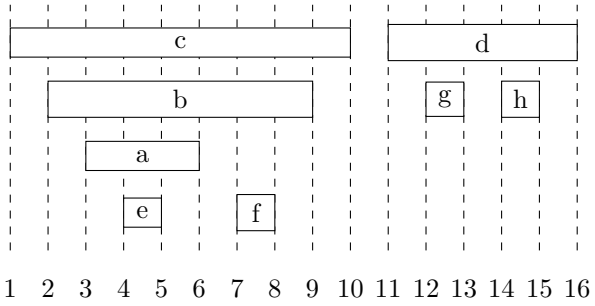
6.3 Depth First Search

CLRS pg.603

Depth-First Search is a strategy for searching a graph. Its starts at the root node and explores as deep as possible before backtracking. By tracking and applying a **timestamp** to each node as it's visited for the first and last time. You can then sort the nodes by **pre-order** and **post-order**, respectively. The timestamp increments when a node is visited and when its neighbours have been exhausted. Post-ordering the nodes provides you with a **Topological Sort** of the graph.

Used in AI for limiting the depth of decisions trees and evaluating branches based on heurstics, maze generation, and topological sorting. **Useful** when needing to visit all nodes within a graph.





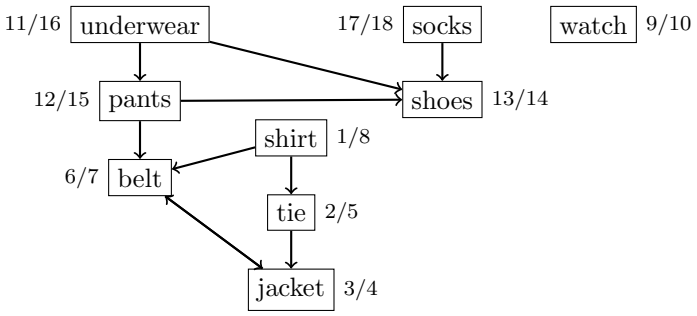

```
DFS(start){
    create stack S
    S.push(start)
    timestamp = 0
    while S is not empty
        node = S.pop()
        node.begin = timestamp
        for all neighbours v of node
            if v.visited == false
                S.push(v)
        timestamp = timestamp + 1
        node.visited = true
        node.end = timestamp
}
```

6.4 Topological Sort

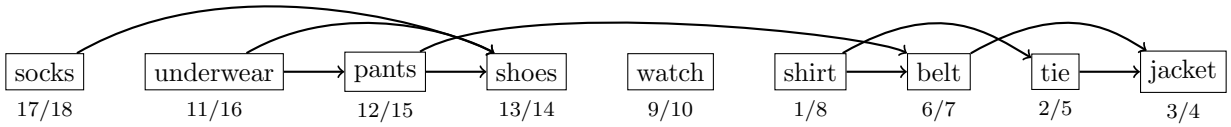
CLRS pg.613

The Toplogical Sort of a **directed graph** is a linear ordering of the nodes, such that every node comes before the node it's connected to. An example is that if each node is a task and the edges between nodes represent constriants on finishing tasks before starting another, topologically sorting this graph would ensure an order to complete the tasks which would provide no conflicts.

Topological Sort is a modified Depth-First Search alogithm, where we **postorder** the nodes, or sort the descending by the last when they were visited.



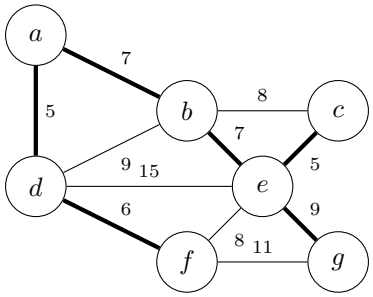
Directed graph of how to get ready in the morning with timestamps



Topologically sorted so there's no conflicts

6.5 Minimum Spanning Tree

CLRS pg.625 A Minimum Spanning Tree is a subgraph of a graph that contains all nodes and has lowest combined weight of all of it's edges. It's possible to have multiple Minimum Spanning Trees per graph, if the edge weights are not unique. We'll look at two algorithms to solve this problem.



Minimum Spanning Tree of a weighted graph

6.5.1 Prim's Algorithm

CLRS pg.634 Prim's Alogithm is a **Greedy Algorithm** using **Priority Queues**. **Avg. Case:** $O(E + V \log V)$ and best used for **dense graphs**. At each step it grows the tree by one node, selecting the smallest edge available.

```
PrimMST(G){
    add all nodes in G to Queue, Q with key = \infty
    while Q is not empty
        x = Q.extractMin()
        for all neighbours v of x
            if v is in Q and weight(x,v) < v.key
                v.parent = x
                v.key = weight(x,v)
}
```

6.5.2 Krushal's Algorithm

CLRS pg.631 Krushal's Algorithm is a **Greedy Algorithm** using **Disjoint Sets**. **Avg. Case:** $O(E \log V)$ and best used for **sparse graphs**. It first creates a set for each node and then a set of all edges ordered by weight. For each edge, if both nodes aren't in the same set together, it merges the sets together and adds that edge to the solution set. It ends when all nodes belong to a single set.

```
KrushalMST(G){
    make Result an empty set
    make a set for each node that contains it
    foreach (u, v) in G.Edges ordered by weight(u, v)
        if Find-Set(u) != Find-Set(v)
            add (u,v) to Result
            Join-Sets(u, v)
    return Result
}
```

6.6 Dijsktra's Algorithm

CLRS pg.658 Dijsktra's Algorithm finds the **single-source shortest path** in a weighted graph. As Dijsktra's Algorithm tranverses the graph, it follows the path of lowest expected total distance. It keeps track of how good each branch is using a **Min-Priority Queue**. **Avg. Case:** $O(E + V \log V)$.

A greedy algorithm is used to choose the next node for each branch and will switch to a different branch if it no longer is the fastest. The variable *distance* on each node in our code tracks the shortest path value to get to that node from the start.

```
Dijsktra(G, start, goal){
    create Priority Queue Q
    add all nodes to Q with distance of \infty
    start.distance = 0
    while Q is not empty
        current = Q.extractMin()
        if current is goal
            goal.parent = current
            return ReconstructPath(goal, empty list)
        for each neighbour of current
            temp = current.distance + weight(current, neighbour)
            if temp < neighbour.distance
                neighbour.distance = temp
                neighbour.parent = current
}
ReconstructPath(node, result){
    if node has parent
        ReconstructPath(node.parent, result)
    result.push(node)
    return result
}
```

6.7 A*

CLRS pg.308 The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored by using an additional **heuristic** unique to the problem. The heuristic function helps guide the path in which the algorithm takes next. For example if the nodes have a coordinate location, the

heuristic function could be the euclidean distance between the current node and the goal, making sure that the algorithm tries the closest branches first.

Avg. Case: $O(\log h(x))$, where $h(x)$ is the optimal heuristic function.

```
AStar(G, start, goal){
    create Priority Queue Q
    add all nodes to Q with score of \infty
    start.score = 0
    while Q is not empty
        current = Q.extractMin()
        if current is goal
            goal.parent = current
            return ReconstructPath(goal, empty list)
        for each neighbour of current
            temp = current.score + weight(current, neighbour) + heuristic(neighbor,
                goal)
            if temp < neighbour.score
                neighbour.score = temp
                neighbour.parent = current
}
ReconstructPath(node, result){
    if node has parent
        ReconstructPath(node.parent, result)
    result.push(node)
    return result
}
```

6.8 k-Means Clustering

K-means clustering aims to partition a number of points in k groups/clusters. We have the optiminal distribution of clusters when every point is closest to the center of it's cluster then any other center of the clusters. The problem is **NP-hard**; however, there are efficient heuristic algorithm scan converge quickly to a local optimum. **Used in** reducing an image color pallete to k colors and feature analysis for machine learning.

As a heuristic algorithm, there is no guarantee that it will converge to the global optimum, and the result may depend on the initial clusters. As the algorithm is usually very fast, it is common to run it multiple times with different starting conditions. **Worst Case:** $2^{\Omega}(n)$ to find globally optimal solution.

Method: Choose k "mean" points and place them randomly. Produce an **Voronoi Diagram** using those "mean" points. Find the center of each region and move the "mean" point there. Repeat until the clusters settle. Run the algorithm multiple times with different initial "mean" points to assure the optimal solution is found.

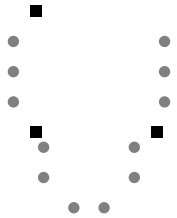
6.9 Convex Hull

CLRS pg.1029

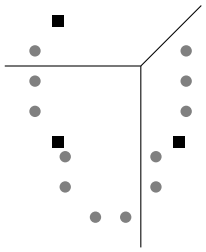
The Convex Hull of a set of points is the **smallest polygon that contains all points**. Imagine it like an elastic band wrapped around a number of pegs in a board. **Used with** bezier curves, pattern recogintion, image processing, statistics, and static code analysis. We'll look at two algorithms to solve this problem.

6.9.1 Graham's Scan

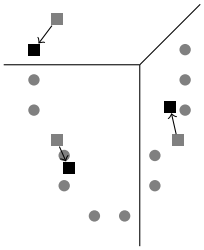
Graham's Scan starts at the lowest point in the set and iterates over the points in a counterclockwise direction, relative to this point. At each step it calculates the angle between the last two points and a the



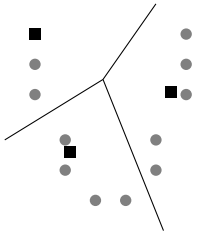
Place three mean points randomly



Create regions based on the mean points



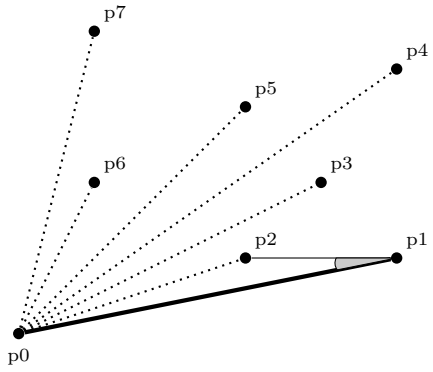
Find the center of the points in each region and move the mean points there



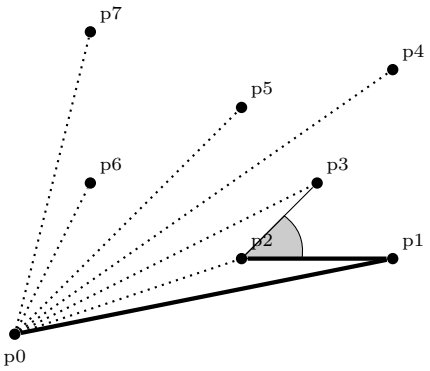
Re-draw the Voronoi Diagrams using the mean points and repeat

next point to try. If that angle is towards the left, we add the test point to our solution set. If that angle is to the right, we remove the last point from our solution set and repeat. **Running time:** $O(n \log n)$.

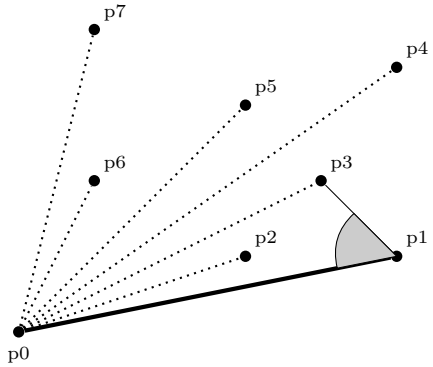
```
GScan(P){
    root = lowest point in P
    S = all points in P sorted by relative angle to root
    Hull = empty list

    Hull.push(root)
    Hull.push(S.pop())

    while S is not empty
        next = S.top()
        if angleDirection(Hull.secondLast(), Hull.last(), next) is left
            Hull.push(S.pop())
        else
            Hull.pop()
    return Hull
}
```



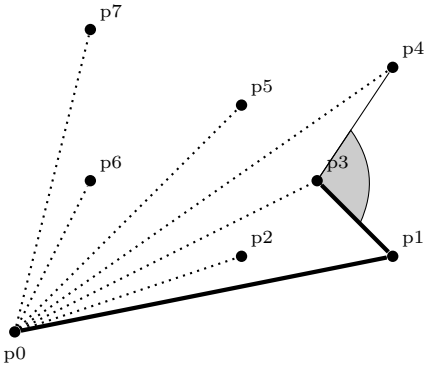
Add p2 to solution set, since ∠ p0 p1 p2 is facing left



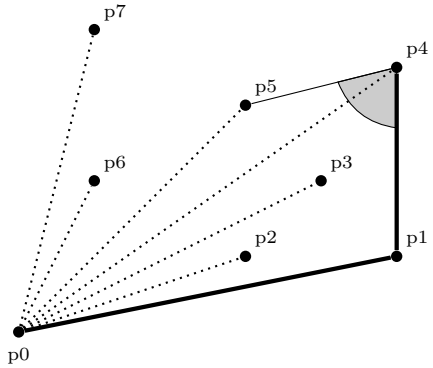
∠p1 p2 p3 is facing right, remove p2



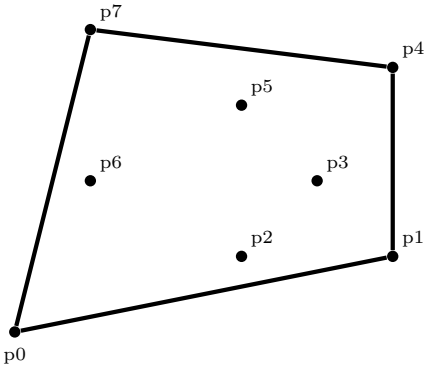
∠p0 p1 p3 is good, so we add p3



∠p1 p3 p4 is bad, remove p3



∠p1 p4 p5 is good, add p4, etc.



Completed scan

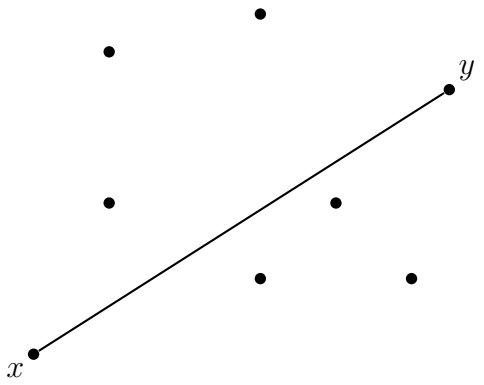
An example of Graham's Scan

6.9.2 QuickHull

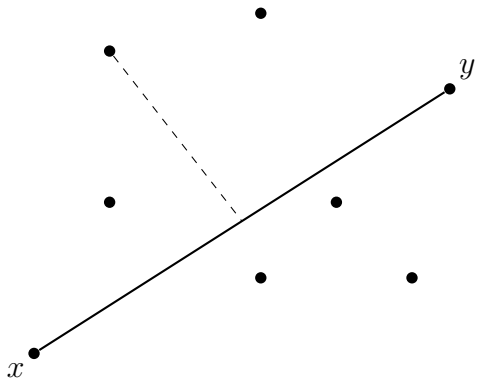
QuickHull is based off of QuickSort in that we use divide and conquer style. At each step we divide the current set by drawing a line between two points. For each side of the line we find the point farthest from the line, create a triangle between these three points. We remove all the points inside the triangle from the working set and repeat the same process on the two other triangle sides.

Just like QuickSort, QuickHull has a **Avg Case**: $O(n \log n)$ and **Worst Case**: $O(n^2)$.

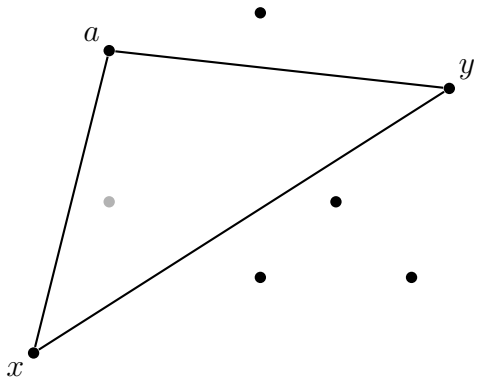
```
QuickHull(P){
    r = right most point in P
    l = left most point in P
    S = []
    S.push( (r,l) )
    while S is not empty
        (x,y) = S.pop()
        s = all points in P above line(x,y)
        if s is not empty
            p = point farthest from line(x,y) in s
            for each point r in triangle(x,y,p)
                remove r from P
            S.push( (x,p) )
            S.push( (p,y) )
    return P
}
```



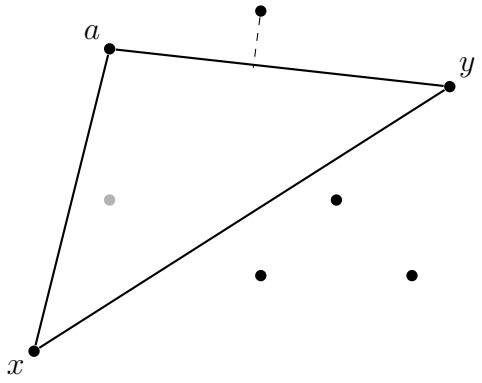
Find points x and y farthest to the right and left, and split the set of points



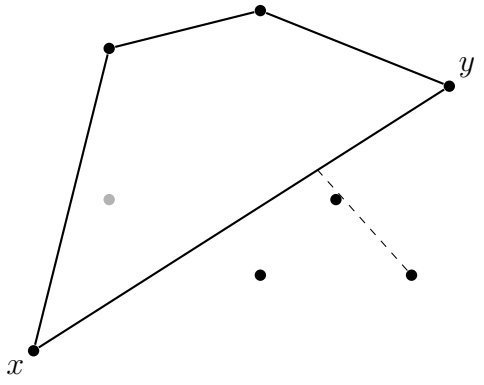
Find the point farthest from line (x,y) on one side



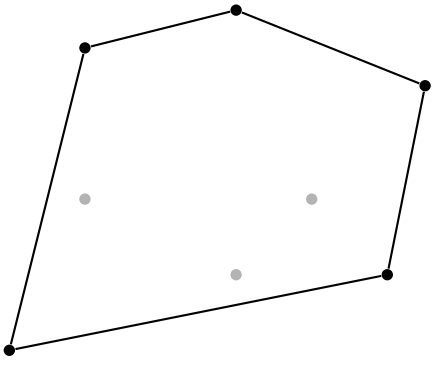
Create a triangle and remove all points from within it



Repeat steps with line (x,a) and (a,y)



Repeat the process with the other side of the line (x,y)



Completed scan

An example of QuickHull

7 Dynamic Programming

7.1 Greedy Algorithms

7.2 Bottom-up Approach

Bottom-up recursion is often the most intuitive. We start with knowing how to solve the problem for a simple case, like a list with only one element, and figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can build the solution for one case off of the previous case

7.3 Top-Down Approach

Top-Down Recursion can be more complex, but it's sometimes necessary for problems. In these problems, we think about how we can divide the problem for case N into sub- problems. Be careful of overlap between the cases

7.4 Problems

1. A child is running up a staircase with n steps, and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

2. Write a method to compute all permutations of a string.

3. Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

7.5 Bresenham's line algorithm

7.6 BoyerMoore String Search Algorithm

7.7 NP-Complete

7.7.1 Travelling Salesman

7.7.2 Knacksack Problem

8 Systems

8.1 Scheduling

8.2 Threads

8.3 Semaphores

8.4 Deadlocks

In order for a deadlock to occur, you must have all four of the following conditions met: 7. Mutual Exclusion: Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.) 2. Hold and Wait: Processes already holding a resource can request additional resources, without relinquishing their current resources. 3. Wo Preemption: One process cannot forcibly remove another process'resource. 4. Circular Wait: Two or more processes form a circular chain where each process is waiting on another resource in the chain.

8.5 Problems

1. Design a class which provides a lock only if there are no possible deadlocks.

2. In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

9 Big Data

9.1 Hadoop

9.2 Databases?

9.3 MapReduce

10 Cryptography

10.1 Assymetric Cryptosystems

RSA is one of the first practical assymeteric cryptosystems and is widely used to secure data currently. Not only does it encrypt data from being read by the wrong party, but it also verifies that the data came from the intended sender. It does this by using a **Public and Private Key Encryption**.

10.1.1 Key Generation

1. Choose two distinct prime numbers p and q, where $n = pq$
   $p = 61, q = 53, n = 61 \times 53 = 3233$
2. $\phi(n) = (p-1)(q-1)$, where ϕ is Euler's totient function.
   $\phi(3233) = (61-1)(53-1) = 3120$
3. Choose an integer *public* such that $gcd(public, \phi(n)) = 1$, therefore *public* and $\phi(n)$ are coprime.
   $public = 17$
4. Determine *private* as $private^{-1} \equiv public \bmod \phi(n)$, therefore *private* is the modular multiplicative inverse of *public* mod $\phi(n)$
   $private = 2753$

Encryption of message M is done by $c = M^{public} \bmod n$, where c is the ciphertext.

$$\textbf{encrypt}(M) = M^{17} \bmod 3233 = c \tag{1}$$

Decryption of c is done by $M = c^{private} \bmod n$.

$$\textbf{decrypt}(c) = c^{2753} \bmod 3233 = M \tag{2}$$

10.1.2 Sending Messages

Bob wants to send a secure message M to Alice. Not only is Bob worried about someone else reading the message, but other people may pretend to be him and send messages as him to Alice. They agree to use RSA and both create a public and private key for themselves. They share their publics keys with eachother and keep their private keys a secret. This is how Bob would prepare his message.

1. Ensures only Alice can open it
   $\textbf{encrypt}(M, public_{alice}) = c_1$
2. Ensures only Bob could have sent it
   $\textbf{encrypt}(c_1, private_{bob}) = c_2$

Bob prepares his message to Alice

1. Verifies that Bob sent it
   $\textbf{decrypt}(c_2, public_{bob}) = c_1$
2. Opens it with her private key
   $\textbf{decrypt}(c_1, private_{alice}) = M$

Alice opens her message

10.2 Cryptographic Hash Functions

A cryptographic hash function is a hash function that takes data of arbitrary size and returns a fixed-size bit string. The ideal cryptographic hash function has four main properties:

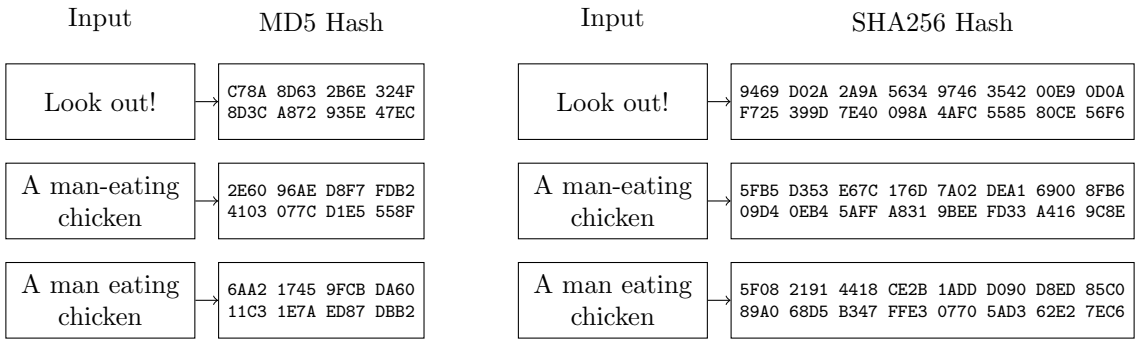
Ease : Easy to compute the hash value for any given message

One-way : Can not generate a message from a hash

Avalanche : Small changes in the data drastically change the hash

Collision Resistant : Infeasible to find two different messages with the same hash.

Used as digital signatures, for data fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. We'll look at two common cryptographic hash functions.



Notice that small changes in the input cause the hash to change significantly; Avalanche effect

10.2.1 MD5

MD5 produces a 128-bit hash value, usually expressed as a 32 digit hexadecimal number. It is **not collision resistant**, making it not suitable for security applications, like SSL certificates or digital signatures. **Used mostly** to verify message integrity when transmitting data.

Method: The MD5 hash algorithm is broken down into 4 rounds of 16 operations. These operations manipulate four 32-bit words, A, B, C, and D, which are initialize with default values. Each round uses a different function on words B, C, and D, which then get mixed with a 32-bit block of the input data and a 32-bit constant. That result is then left bit rotated and stored in word A. All words get rotated by one position ($A \rightarrow B$) and the process is repeated.

10.2.2 SHA-2

SHA-2 comes in two variants, that produce 256 and 512 bit hash values. SHA-2 belongs to the family of hash algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Security flaws were discovered in SHA-1, leading to SHA-2's widespread adoption. **Used in** security applications and protocols, including TLS and SSL, PGP, SSH, S/MIME, Bitcoin and IPsec.

10.3 Security Attacks

If we wanted to store private information, such as user's password, one technique we could use is to keep a hash of the user's password, rather than storing it directly. This prevents leaking of your user's passwords, since the hash can only be used to verify. We'll look at a few methods people use to bypass/break a system like this.

10.3.1 Brute-force Attack

A Brute-force attack consists of systematically checking all possible keys or passwords until the correct one is found. In the worst case, this would involve traversing the entire search space. Longer keys are exponentially more difficult to break then shorter ones. A cipher with n bits has a worst-case of 2^n. Breaking a symmetric 256-bit key by brute force requires 2^{128} times more computational power than a 128-bit key. 50 supercomputers that could check a billion billion (10^{18}) keys per second, would require about 30^{51} years to exhaust the 256-bit key space.

To counter a brute-force attack you can limit the number of attempts that a passphrase can be tried, by introducing time delays between successive attempts, or by increasing the attempt's complexity by requiring a CAPTCHA or a second-step verification.

10.3.2 Dictionary Attack

A Dictionary Attack is similar to Brute-force, but take advantage of the fact that most people use shorter passpharse that uses a common set of words with slight variation, such as appending a digit, eg. *'password1'*. They are not guaranteed to succeed, and can be easily **countered** by including random characters within the passpharse, sppliting up common words, or simply by increasing the length of your passphrase.

10.3.3 Rainbow Tables

A Rainbow Table is a **pre-computed** look-up table for reversing cryptographic hash functions. Before an attack a rainbow table will be created by exploring the passphrase solution space and saving it's corresponding hash. There are **space vs. time trade-offs** as the possible passphrase solution space could be very large, so Rainbow Tables don't map it entirely. Using techniques from a Dictionary-Attack, you could create a Rainbow Table of very common passpharses to quickly reverse those. You can use a "salt" in your hash to **counter** a Rainbow tables, as it makes the solution space infeasibly big.

```
Searching for hash(apple) in users' hash list... : Matches [alice3, 0bob0, charles8]
Searching for hash(blueberry) in users' hash list... : Matches [usr10101, timmy, john91]
Searching for hash(letmein) in users' hash list... : Matches [wilson10, dragonslayerX, joe1984]
Searching for hash(s3cr3t) in users' hash list... : Matches [bruce19, knuth1337, john87]
Searching for hash(z@29hjja) in users' hash list... : No users used this password
```

10.3.4 Hash Salting

Rainbow tables only work because each password is hashed the exact same way. If two users have the same password, they'll have the same password hashes. We can prevent these attacks by randomizing each hash, so that when the same password is hashed twice, the hashes are not the same. To counter this we generate a new random string of characters called a "salt" each time we save a password . We append the salt to the password, hash it, and store the result with the salt for that user. To verify a password, we lookup the salt, append it to the password, hash it and match it to what we have stored.

```
hash("hello")                    = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

11 Web

11.1 SSL and HTTPS

11.2 EcmaScript 5

11.3 AJAX

11.4 Web RPC

11.5 Full Request

12 Behavourial Questions

Prep for these, you shouldn't have to think about them hardest bug current project favourite Open source project cleverest solution what have you contributed to OSS

13 Designing Algorithms

13.1 Design Patterns

13.1.1 Singleton

The Singleton pattern ensures that a class has only one instance and ensures access to the instance through the application. It can be useful in cases where you have a "global" object with exactly one instance. For example, we may want to implement Restaurant such that it has exactly one instance of Restaurant.

13.1.2 Factory

The Factory Method offers an interface for creating an instance of a class, with its subclasses deciding which class to instantiate. You might want to implement this with the creator class being abstract and not providing an implementation for the Factory method. Or, you could have the Creator class be a concrete class that provides an imple- mentation for the Factory method. In this case, the Factory method would take a param- eter representing which class to instantiate.

13.1.3 Facade

13.1.4 Flyweight

13.1.5 Proxy

13.1.6 Observer

13.1.7 Template Method

13.2 Five Algorithm Approaches

The following are five techniques for quickly figuring out an algorithm for a solution.

Examplify Create multiple examples of the problem and try to find a pattern.

Pattern Matching Find what problems the algorithm is similar to and modify them.

Simplify and Generalize Change the contraints of the problem to make it simplier and solve it. USe the solution to create a more generalized apporach.

Base Case and Build Solve the problem for a few base cases (n = 1,2,3), then build a solution for n where we know the solution for $n - 1$.

Data Structure Brainstorm Run through your list of data structures and try to apply each one until one fits.

13.3 Object-Oriented Design

13.4 Test Cases

In general, you should think about the following types of test cases:

The normal case Does it generate the correct output for typical inputs? Remember to think about potential issues here. For example, because sorting often requires some sort of partitioning, it's reasonable to think that the algorithm might fail on arrays with an odd number of elements, since they can't be evenly partitioned. Your test case should list both examples.

The extremes What happens when you pass in an empty array? Or a very small (one element) array? What if you pass in a very large one?

Nulls and "illegal" input It is worthwhile to think about how the code should behave when given illegal input. For example, if you're testing a function to generate the nth Fibonacci number, your test cases should probably include the situation where n is negative.

Strange input What happens when you pass in an already sorted array? Or an array that's sorted in reverse order?

13.5 Problems

1. Design the data structures for a generic deck of cards. Explain how you would subclass the data structures to implement blackjack.
2. Design a parking lot using object-oriented principles.
3. Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve.
4. Design and implement a hash table which uses chaining (linked lists) to handle collisions.
5. How would you test an ATM in a distributed banking system?