Chapter 1

Getting_Started

The Computer Science handbook is a handbook designed to explain algorithms and data structures in way that anyone can understand. Many websites (eg Wikipedia) contain lengthy and wordy explanations that are full of technical jargon. We have tried our hardest to simplify all language to make it easy to read without any math or computer science background. We hope to share our knowledge with you and we ask only one thing from you. You must learn something before you leave!

1.1 Format

Each article with have multiple sections to help you understand the content.

1.1.1 Introduction

The introduction section gives a brief overview of what the article is about. It will usually come with a prerequisite section which will contain topics that will be recommended to have been read before the article.

1.1.2 Implementation

The implementation section will be an implementation of the article in Java. It is recommended that you try to implement things yourself first before looking at the implementation. If you truly understand the concept, then you will never have to memorize a single line of code. The code will come from your understanding of how it works.

1.1.3 Applications

The applications section will be about real world applications of the concept that show why the concept is important.

1.1.4 Exercises

The exercises section contains practice problems to test if you truly understand the concept. Many of these questions come from real interview questions.

Chapter 2

Fundamentals

There are some fundamental topics about writing computer programs that we must learn before we can move on to the basic theory.

2.1 Basics

First we must learn how to write a basic Java program with control structures and variables.

2.1.1 Data Types

We use closets or drawers to store our clothes and garages to store our cars. Similarly, we store different types of data in different kinds of data types. Picking the right data type for the type of information is important. Most programming languages will support these data types.

Data type Number of bits Range
boolean 1 bit true or false
byte 8 bits -128 to 127
short 16 bits -32,768 to 32,767
int 32 bits 2,147,483,648 to 2,147,483,647
long 64 bits 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807
char 8 bits 256 bits
float 32 bits 3.4e038 to 3.4e+038
double 64 bits 1.7e308 to 1.7e+308

2.1.2 Arithmetic Operations

Operation Description Example

z = x + y Addition z = 1 + 2, z = 3
z = x - y Subtraction z = 3 - 1, z = 2
z = x * y Multiplication z = 3 * 2, z = 6
z = x / y Integer division z = 5 / 2, z = 2 or z = 9 / 3, z = 3
z = x Modulus (or remainder) z = 5
x++ Increment by 1 x=5, x++, x = 6
x– Decrement by 1 x=5, x–, x = 4

2.1.3 Control Structures

If statement:

```
if (expression){
  do action;
}
```

Loops:

```
while(expression){
  do action;
}
```

2.2 Runtime and Memory

Next, we analyze how long a program will take to perform a calculation and how much memory a program will used based on the inputs.

2.3 Boolean Algebra

2.4 Logic

Chapter 3

Recursion

Next: Advanced Recursion

Recursion is process that repeats itself in a similar way. Anything that has its definition nested inside itself is considered to be recursive. For example GNU stands for GNU not Unix!. Expanding this acronym gives us ((GNU not Unix) not Unix!). As you can see this will go on forever and GNU's definition is nested inside itself so it is recursive. The Fibonacci sequence is also recursive: F(n) = F(n-1)+F(n-2). Inside the function F, we see two more F's!

In computer science infinite looping is bad because computers do not know how to terminate so we need some way to stop it. We will call a stopping point the base case. A base case is the case where the recursion will stop. Everything must eventually reduce to a base case. For the Fibonacci sequence, the base case is F(0) = 1 and F(1) = 1 and we can see that for all N¿1, the Fibonacci sequence will reach the base case.

So for something to be recursive in computer science, it needs:

a recursive definition (contained in itself) and a base case that all valid inputs will eventually reach

Template for recursion:

```
recursion(parameter)
    if base\_case (parameter):
        stop
    recursion( operation(parameter) )
```

recursion is the recursive function base_case is the check if the parameter has reached the base case operation reduces the parameters towards the base case

3.1 Factorial

Let's look at a simple recursive function: factorial function.

1! = 1

n! = 1 * 2 * 3 .... n.

Or we could write it as n! = (n-1)! * n. We now see that in this form, the factorial function is defined within itself which makes it recursive. Our base case is 1! = 1.

Example:

4!

= 3! * 4

= 2! * 3 * 4

= 1! * 2 * 3 * 4

= 1 * 2 * 3 * 4

= 24

3.1.1 Formalization

```
Let f(n) be the nth factorial number where n is a positive integer.

Base case:
f(1) = 1

Recurrence:
f(n) = f(n-1) * n

Example:
f(4)
= f(3) * 4
= f(2) * 3 * 4
= f(1) * 2 * 3 * 4   [Base Case]
= 1 * 2 * 3 * 4
= 24
```

3.1.2 Implementation

```
int factorial(int n){
    if(n == 1)return 1;
    return factorial(n - 1) * n;
}
```

3.2 Sum of digits of a string

We can use recursion in many places and we can apply it to simple problems that you probably have not thought about. Summing the digits of a string can be done using simple loop but we can also use recursion to do it.

Let’s say we have the string ’23528’. The total is equal to the first digit + the sum of the rest of the digits. 2 + ’3528’. We can do the exact same thing for the rest of the digits: the second digit + the sum of the rest of the digits and keep doing this until we have no more digits.

’23528’ ’3528’ + 2 ’528’ + 5 ’28’ + 10 ’8’ + 12 ” + 20 20

3.2.1 Formalization

```
Let sum(string) be the sum of digits in a string
Let n be the length of the string

For simplicity, let string[x..y] be the substring of the string from index x t
Example:
string = 'abcd'.
string[1..3] = 'abc'

Base case:
sum(empty) = 0

Recurrence:
sum(string) = sum(string[2..n]) + int(string[1])

Example:
sum('23528')
= sum('3528') + 2
= (sum('528') + 3) + 2
= ((sum('28') + 5) + 3) + 2
= (((sum('8') + 2) + 5) + 3) + 2
= ((((sum('') + 8) + 2) + 5) + 3) + 2
= ((((0 + 8) + 2) + 5) + 3) + 2
= 20
```

3.2.2 Implementation

```
int sum(string str){
```

```
    int n = str.length();
    if(n == 0){
        return 0;
     }
    else{
      int charToNum = (str.charAt(0)-'0');
      return sum(str.substring(1,n)) + charToNum;
    }
}
sum('23528');
```

3.3 Count

Let's say we have an string of N characters and we want to find the number of times the letter 'c' appears. This time we need to add some logic to the problem.

If the first letter of the string is a 'c', then we can add 1 to the count. Otherwise, we don't add anything. We can do the exact same thing for the substring without the first letter. If the second letter of the string is a 'c', we add 1 other we don't add anything etc.

For example we have the string 'cacaec'. Since the first letter is a 'c' we add 1 to the count. Then we remove the first letter and get 'acaec', the first letter is not a 'c' so we don't add to the count. We keep reducing this way until we get an empty string and we return the count.

'cacaec' 'acaec' +1 'caec' + 1 'aec' + 2 'ec' + 2 'c' + 2 '' + 3 3

3.3.1 Formalization

```
Let count(string) be the number of 'c's in the string

For simplicity, let string[x..y] be the substring of the string from x
Example:
string = 'abcd'.
string[1..3] = 'bcd'

Base case:
count(empty) = 0

Recurrence:
count(string) = {if string[n]=='c'  count(string[2..n])+1
```

```
                    {else                    count(string[2..n])
```

```
Example:
count('cacaec')
= count('acaec')+1
= (count('caec')+0)+1
= ((count('aec')+1)+0)+1
= (((count('ec')+0)+1)+0)+1
= ((((count('c')+0)+0)+1)+0)+1
= ((((count('')+1)+0)+0)+1)+0)+1
= ((((0+1)+0)+0)+1)+0)+1
= 3
```

3.3.2 Implementation

```
int count3(string str){
   int n = str.length()
   if(n == 0)return 0;
   if(str.charAt(0) == 'c'){
      return count3( str.substring(1, n) ) + 1;
   }
   else{
     return count3( str.substring(1, n) );
   }
}

count3('cacaec');
```

3.4 Calculate Exponential

Let's say we wanted to find x^n and for the sake of this problem we want the last four digits of that number. Note that 267^4 mod 10000 = (267^3 mod 10000) * 267 mod 10000. This is important because it means we can take the last 4 digits in each step instead of having to compute the giant exponent and then taking the last 4 digits. We can do this problem very easily by using a simple loop, but we can do better by using recursion. By definition of exponents, x^a * x^a = x^{2a}. Using this we can see that if n is divisible by 2, then x^n = x$^{n/2}$ * x$^{n/2}$.

For example 5^4 = 5^2 * 5^2

But let's take a look at x$^{n/2}$. If n is even we can do the exact same thing! x$^{n/2}$ = x$^{n/4}$ * x$^{n/4}$. We have a recurrence relation and the base case is simple: if $x^1 = x$.

But what if n is odd and not 1? Then we have $x^n = x^{n/2} * x^{n/2} * x$ and we can now solve this problem using recursion.

3.4.1 Formalization

```
Let exp(b,n) be b^n

Base case:
exp(b,1) = 1   [Since b^1 = b]

Recurrence:
exp(b,n) = {if n divisible by 2     exp(b,n/2)^2 % 10000
           {else                    (exp(b,n/2)^2)*b % 10000

Example: (for simplicity, leave out the modulus)
exp(3,10)
= exp(3,5)^2
= (exp(3,2)^2))*3)^2
= (((exp(3,1)^2)*3)^2)  [Base case]
= (((3^2)^2)*3)^2)
= ((9^2)*3)^2)
= (81*3)^2
= (243)^2
= 59049
```

3.4.2 Implementation

```
int exponent(int b,int n){
    if(n==1)return b;
    if(n%2==0){
      int x = exponent(b,n/2);
      return (x * x)%10000;
    }
    else{
      int x = exponent(b,n/2);
      return (x*x*b)%10000;
    }
}
```

3.5 Exercises

Given an array of N integers, write a recursive function to get the sum Given a string S, write a recursive function to determine if it is a palindrome. Given a number N, write a recursive function to output the number in binary. Given a string S, write a recursive function to return a reversed string

Chapter 4

Sorting

Sorting is arranging an array of n elements in either increasing or decreasing order by some property. It is very useful in computer science for efficiency in other algorithms that usually require a search.

A stable sort is a sort that can preserve sorting of other properties. For example if we have

(3,G), (1,G), (3, A), (6 K), (1,B)

and we want to sort by the first property in increasing order, we will have:

(1, G) (1,B) (3,G), (3,A) (6 K).

If we sort again by the second property we have (1,B) (3, A) (1, G) (3, G) (6,K) then the sort is stable as the first sort is preserved. However if we had (1,B) (3, A) (3, G) (1, G) (6, K), then the sort would be unstable.

4.1 Slow Sorts

Slow sorts are sorts that have the runtime of $O(n^2)$. Although they should almost never be used, it is good to know how to implement a simple sort.

Name Runtime

Bubble Sort $O(n^2)$

Selection Sort $O(n^2)$

Insertion Sort $O(n^2)$

4.2 Fast Sorts

Fast sorts are sorts that have a runtime of O(nlogn). They are very fast and you will usually use merge sort or quick sort for sorting in the real world.

Most language will have their own implementation of a quick sort or merge sort in their standard libraries.

Name Runtime
Heap Sort O(nlogn)
Merge Sort O(nlogn)
Quick Sort O(nlogn)

4.3 Super Slow Sorts

Just for fun, these are some silly sorts that you should never use.

Name Runtime
Bozo Sort O(?)
Permutation Sort O(n!)
Miracle Sort O(?)

4.4 Exercises

Given two sorted arrays of N numbers, merge the two arrays into an single array of size 2N. Find the minimum number of swaps to sort an array Find the minimum number of adjacent swaps to sort an array

Chapter 5

Data_Structures

Data structures are a way of storing data such that it can be used in an efficient way. Although many of these data structures are already built into various languages, it is important to understand how they work. By understanding the implementations we can have a sense of which data structure to use in different problems as well as determine how efficient they are.

An abstract data type is a conceptual model for representing data. An abstract data type tells what it should do as opposed to how it should work. It will tell us what operations it should have but should not tell us how to implement them.

For example, a bottle should be able to hold water and allow us to drink from. This tells us what it should do but we don't need to know how it works or how it is made. A plastic water bottle is an implementation of a bottle. It holds water in its interior and allows us to drink by unscrewing the cap and letting us pour water down our throat. A thermos is also an implementation of the bottle, it holds fluid inside it, but this thermos has a lid that can be popped open and water can come from it. A thermos and plastic water bottle are different implementations as they are made differently and used differently, but they fundamentally do what a bottle is supposed to do: store liquid, and provide a way to drink. A bottle does not actually exist, but types of bottles do.

Some implementations of abstract data types are better than others for different purposes. For example plastic water bottles are very cheap whereas a thermos is more expensive but a thermos can hold hot water and keep it warm for a long period of time. When thinking of a implementation for an abstract data type we need to know what we need it for.

For more intermediate data structures, read the advanced data structures page.

5.1 Arrays

Imagine you had a row of parking spaces where each was labelled with a number. If you wanted to know what the license plate of a car at parking space 4 was, all you have to do is go to the parking space and read off the license plate. If you wanted to park your car at parking space 5, you would go to parking space 5 and put your car in there only if there was nothing there.

Let's say that you had cars at parking spaces 1, 2, and 3. If you wanted to insert a new car at parking space 1 and keep the rest of the cars in the same order, you would have to shift the cars in the parking spaces from 1, 2 and 3 to 2, 3 and 4 by getting in each car and parking them in the new spaces which would take some time. This type of structure is an array.

An array is the most basic data structure that stores elements of the same type in a fixed block. The fact that it is in one block and the same type is important because it allows accessing elements very quickly if you have the index. All you have to do is go to the index and retrieve the element. However, inserting elements in the array is slow because you would have to shift all the elements and also if you want to shift past the fixed size you will get an error. (Imagine the parking spaces are full and you wanted to insert a car somewhere, there will still be one car that will have no parking space).

Arrays can be multidimensional meaning you can have an array of array of objects. (Imagine a parking lot with multiple rows of parking spaces).

Operation Create Get Set

Time Complexity O(n) O(1) O(1)

5.2 Stack

Imagine a stack of plates at a buffet, the plates are taken from the top and are also replaced from the top. The first plate to go in will be the last plate to come out. The last plate to go in will be the first to come out. This structure is called a stack.

A stack is an abstract data type with the property that it can remove and insert elements following a FILO (First In Last Out) structure. The first element to be inserted must be the last element to be removed and the last element to be inserted must be the first element to be removed. Sometimes, removal is called "pop" and insertion is called "push".

Stacks are used for function calls on the memory stack. Whenever a function is called, it is placed on the memory stack with its variables and when it is returning a value, it is popped off the stack.

A stack is usually implemented as a vector.

5.2.1 Vector

A vector is a stack that is implemented as an array. It is very similar to an array, but it is more flexible in terms of size. Elements are added and removed only from the end of the array. When more elements are added to the vector and the vector is at full capacity, the vector resizes itself and reallocates for 2*N space. When using an vector we can keep adding elements and let the data structure handle all the memory allocation.

Operation Create Get Set Push to back Delete

Time Complexity O(n) O(1) O(1) O(1) O(n)

5.3 Queue

Imagine you are standing in line for a restaurant. Whoever is first in line will be served first and whoever is last in line will be served last. People can be served while more people join the line and the line may get very long because it takes a while to serve one person while more people join the queue. This is called a queue.

A queue is an abstract data type with two functions, pop and push. Removal from the front is called "pop" or "dequeue". Insertion from the back is called "push" or "enqueue". A queue follows a First In First Out (FIFO) structure meaning the first element pushed should be the first element popped and the last element pushed should be the last element popped.

Queues are often used for buffer systems, for example a text message service. The messages that arrive at the server first are relayed first and the messages that arrive later are relayed later. If there are too many text messages in the system such that the rate texts are received overwhelm the number of texts that are sent the buffer may overflow and messages will get dropped. Most of the time this won't happen because the systems are designed to handle large loads, but if there were an emergency that caused everyone to start texting many texts could be dropped.

Example of push:

Example of pop:

5.3.1 Linked List

Imagine you had some train cars that were linked together where each was labelled on the inside with a letter. If you had to find a specific letter you would have to look inside the first train to check and then walk into the next train car to check the letter and so forth until you found the train car you wanted. If you wanted to insert a train car somewhere, all you would have to do it unlink the position where you wanted to insert it and then relink the new train car with the other cars. If you wanted to remove a train car all you would have to do is unlink that car from the other cars and then relink the cars that were adjacent to it. This sort of structure is called a linked list.

A pointer is something that holds the memory location of another object.

A linked list is similar to an array but it is different such that it is not stored in one block of data. Each element can be stored in a random place in memory but each element contains a pointer to the next element thus forming a chain of pointers. Think of a pointer as a link that links two train cars. Since the elements aren't in a block, accessing an element must be done by traversing the entire linked list by following each pointer to the next. However, this also allows insertion to be done more quickly by simply changing the point of the previous element and setting to the pointer of the current element to the next element. Deletion is also done by taking the previous element and changing its pointer to two elements ahead. In a linked list the links only go forward and you cannot move backward.

A doubly linked list is a linked list that has pointers going backwards as well as forwards.

Operation Get Add at node Delete at node Add Delete

Time Complexity O(n) O(1) O(1) O(n) O(n)

5.4 Trees

Trees are data structures that follow a hierarchy, each node has exactly one or zero parents and each node has children. Trees are recursive structures meaning that each child of a tree is also a tree. A tree within another tree is called a subtree.

A child is a node that is below another node. A parent is a node that is above another node.

The element at the top of the tree with no parents is called a root. The node at the bottom of the tree with no children is called a leaf.

Each node can hold different kinds of information depending on the tree. A node can hold the children it has, the parent it has, a key associated with the node and a value associated with the node.

5.4.1 Binary Tree

A binary tree is a tree where each node has at most two children.

5.5 Sets

Imagine you have a grocery list that you use to keep tracking of things you need to buy. You want to make sure there are no duplicate items in the list, you can add items to the list and that you can remove items from your list. This structure is similar to what a set does.

Sets are abstract data types which are able to store values and are used for three operations: insertion, deletion and membership test.

Insertion places an element into the set, deletion removes an element from the set and a membership test is checking whether an element exists within the set.

5.5.1 Hash Set

Hash sets are sets that use hashes to store elements. A hashing algorithm is an algorithm that takes an element and converts it to a smaller chunk called a hash. For example let our hashing algorithm be (x mod 10). So the hashes of 232, 217 and 19 are 2,7, and 9 respectively.

For every element in a hash set, the hash is computed and elements with the same hash are grouped together and stored in a linked list. The linked list is called a bucket.

If we want to check if an element already exists within the set, we first compute the hash of the element and then search through the linked list associated with the hash to see if the element is contained.

Let use the example of the hashset of the elements of 3242, 3523, 123, 235 and 538. The hash set looks like this when computed:

If we wanted to check if 7238 was in the hash set, we would get the hash (7238 mod 10 = 8). So we get the bucket associated with the hash 8 and we get the list of (538). When we iterate through this short list, we see that 7238 is not a member of the set.

Similarly, if we wanted to insert 7238 into the hash set, we would check if it exists and if it did not we would append the element to the end of the bucket. For deletion we would find 7238 check if it existed in the set and remove it from the bucket.

Hash sets are very efficient in all three set operations if a good hashing algorithm is used. When the objects are that being stored are large then hash sets are effective as a set.

Operation Membership Insertion Deletion

Time Complexity O(1) O(1) O(1)

5.5.2 Tree Set

A tree set is a set which stores the values in a binary search tree. To store elements in a tree set, they must be able to be sorted by a property. To insert an element, it is added to the binary tree. To delete an element, it is removed from the binary tree. To check for membership, we do a binary search for the element in the binary tree.

The advantage of tree sets is that they are maintained in a sorted order.

Operation Membership Insertion Deletion

Time Complexity O(log n) O(log n) O(log n)

5.6 Maps

Imagine you had a English dictionary. If you look up a word, you can find it's definition and read it out. For example if you looked up the word 'cat' in the English dictionary, you would look through the dictionary alphabetically until you found the word 'cat' and then you would look at the definition: 'a feline animal'. If you really wanted to, you could also add your own words into the dictionary and the definitions of your words. This type of structure is called a map.

Maps (also called dictionaries) are abstract data types that store pairs of key-values and can be used to look up values from the keys. The keys are like the words in an English dictionary and the definitions can be seen as the values. Maps are able to support insertion of key-value pairs, retrieve values from keys, and delete key-value pairs.

5.6.1 Hash Map

Hash maps use hash sets to store the keys which then map to their values. The advantage of a hash map is that it is very fast but a disadvantage is that it is not sorted unlike a tree set.

Operation Membership Insertion Deletion

Time Complexity O(1) O(1) O(1)

5.6.2 Tree Map

Tree maps use tree sets to store the keys which then maps to their values.
Operation Membership Insertion Deletion
Time Complexity O(log n) O(log n) O(log n)

5.7 Priority Queue

Consider a waiting list for lung donors. The patients are given a score when they are placed on the waiting list by how much they need a lung based on their whether they smoke, risk factors, age, expected time left etc. When a lung is available, the patient with the highest score will get removed from the waiting list. During this time, more patients could be added to the queue. The behaviour is similar to a queue but instead of the first person getting in the queue getting a lung first, the person with the highest score will get it. This means that if Sam has a score of 60 and gets placed in the queue after Bob who has a score of 40, Sam will get the lung first even though Bob was in the queue before him.

A priority queue is an abstract data type with two operations: push and pop. Push adds an element into the priority queue and pop removes the highest or lowest element.

A priority queue is usually implemented as a heap because it is the most efficient because of its structure.

5.7.1 Heap

Heaps are trees which have the property that a parent node must either be greater than all the elements in its left and right subtrees (a max heap) or less than all the elements in its left and right subtrees (a min heap). Priority queue's are most efficiently implemented as heaps.
Operation Push Pop
Time Complexity O(log n) O(log n)

Chapter 6

Graph_Theory

Next: Advanced Graph Theory

Graphs are a set of objects where some pairs of objects called nodes or verticies are usually connected by links called edges. The nodes here can be seen numbered from 1 to 6. There are edges connecting these various nodes.

A undirected graph is a graph where an edge from A to B is the same as the edge from B to A for all edges. The above graph is undirected.

A directed or bidirectional graph is a graph where edges have direction meaning if there is an edge from A to B then there may not be an edge from B to A.

A subgraph is a subset of edges and vertices within a graph.

A directed acyclic graph (DAG) is a graph with no directed cycles (see topological sorting).

A weighted graph is a graph that contains weights or values assigned to each edge or node. Usually these weights act as the cost to reach/use that node.

6.1 Representations

A graph can be represented as a adjacency matrix or a adjacency list.

6.1.1 Adjacency Matrix

Adjacency matrixes use a matrix to store the edges between nodes. A 1 means a connection between x and y and a 0 means no connection. The edge weight could also be used instead of a 1. $O(n^2)$ where n is the number of nodes. For example the matrix for graph below is:

1 2 3 4 5 6
1 0 1 0 0 1 0
2 1 0 1 0 1 0
3 0 1 0 1 0 0
4 0 0 1 0 1 1
5 1 1 0 1 0 0
6 0 0 0 1 0 0

6.1.2 Adjacency List

Adjacency lists use an array of linked lists to store all the edges. At x, you have a linked list of nodes that connect to that node. 1 connects to nodes 2 and 5, 2 connects to 1, 3 and 5 and so forth. O(m) storage where m is number of edges.

Node edges
1 2, 5
2 1 3 5
3 2 4
4 3 5 6
5 1 2 4
6 4

6.2 Tree

A tree is a special graph with no cycles. It has the special property that there will be only one path from one node to another node.

A subtree is a child tree of a tree.

Note that trees have two meanings in computer science. It can either refer to a tree data structure or it can refer to a tree in graph theory.

6.2.1 Spanning Tree

A spanning tree of a graph is a connected tree that spans all the nodes of the graph.

A minimum spanning tree is the spanning tree that requires the minimum of some property such as total weight or total edges.

Spanning tree algorithms are essential in networking to ensure no loops occur when sending data through a network.

Algorithm Desc Time Space
Prim's Using greedy method O(n log n) $O(n^2)$

Kruskal Using connected components O(n log n) O(n^2)

6.3 Shortest Path

The shortest path is defined as a path from one node to another while trying to minimize a certain property (least number of nodes, smallest total weight). However, shortest paths may have negative weights which leads to cycles.

6.3.1 Algorithms

Algorithm Desc Time Space Detect cycles?

Floyd Warshall Computes shortest path between all pairs of nodes O(n^3) O(n^2) Yes

Bellman Ford Computes shortest path between a pairs of nodes O(n^2) O(n) Yes

Dijkstra's Computes shortest path between a pair of nodes using the Greedy method O(n log n) O(n log n) No

6.4 Cycle detection

A cycle occurs when you start at a node and you can reach the same node from a path.

Note that trees cannot contain cycles.

A cycle can be detected using a depth first search on each unvisited node to check if the DFS tree has a backwards edge.

6.5 Topological Sorting

A topological sort or topological order of a directed graph is an order in which every node will come after its ancestors.

For example topological orders could be:

(A, B, C, D, E, F G) (B, A, D, C, F, E, G)

But (B, A, C, F, D, E, G) is not a topological ordering because D is an ancestor of F and it comes after F.

6.6 Strongly Connected

A graph is strongly connected if all nodes in the graph follows the conditions:

There exists a path from that node to every other node All other nodes can visit that node.

In other words, all nodes in a strongly connected graph can visit each other.

To determine if a graph is strongly connected, we first pick a node and check that we can reach all nodes from it and this checks the first property. To check the second property, we can reverse all the edges of a node (A-¿B becomes B-¿A) and then check again that we can reach all nodes from the same node.

6.7 Connected Components

A connected component is a subgraph where all the vertices in the subgraph connect to each other. Finding the number of distinct connected components can be done using a breadth first search or a depth first search.

6.7.1 Strongly Connected Components

A strongly connected component is a connected component but has the property that each vertex can visit any other vertex in the strongly connected component from any path.

Chapter 7

Searches

Searches are used to find solutions to problems and there many ways to search for a solution. Here are some generic searches that can be applied to many different problems.

7.1 Binary Search

Binary search is a type of search that is able to find an object in a sorted list in O(log n). In binary search we first start at the middle element and we keep trying to halve the problem until we find the element we need.

7.2 Ternary Search

Ternary search is a type of search that finds the maximum value of a increasing or decreasing function by breaking it into 3 parts.

7.3 Depth First Search

Depth first search or DFS is a method of search that goes as far as possible before backtracking. DFS is implemented using a stack and most of the time it uses an function stack for recursion.

7.4 Breadth First Search

Breadth first search or BFS is a method of search that takes the closest things first then the farthest. BFS is implemented with a queue.

7.5 Flood Fill

Flood fill is a search that fills a grid. It can be implemented with either DFS or BFS. We first start at some starting position and then we expand in the directions that we can (eg: up, left, down, right).

7.6 Backtracking

Backtracking is a search that enumerates every single possible solution by using partial solutions.

Chapter 8

Dynamic_Programming

Prerequisites: Advanced Recursion

Next: Advanced Dynamic Programming

Dynamic programming uses memoization by solving subproblems to solve the more complex problem. Dynamic programming uses recursion but instead of working backwards, it builds up the answer and reduces the number of duplicate computations.

Like recursion, dynamic programming requires two things:

A base case and A subproblem that can be reduced into smaller subproblems

8.1 Fibonacci Sequence

The Fibonacci sequence is determined by f(n) = f(n-1)+f(n-2) where f(0) = 1 and f(1) = 1.

If we calculate f(5) we have:

```
f(5)
= f(4) + f(3)
= f(3) + f(2) + f(2) + f(1)
= f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)
= f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)
= 1 + 1 + 1 + 1 + 1 + 1 + 1
= 8
```

However we are computing multiple values more than once. When we compute f(5) we need to compute f(4) and f(3) but f(3) is already computed when we compute f(4) and thus we have to recompute it again. We can avoid this redundancy by "building up". We can calculate f(2), then f(3) then f(4) and finally f(5) and we won't have duplicate calculations.

```
f(0) = 1
f(1) = 1
f(2) = f(1) + f(0) = 2
f(3) = f(2) + f(1) = 3
f(4) = f(3) + f(2) = 5
f(5) = f(4) + f(3) = 8
```

8.1.1 Formalization

Recursion

```
Let f(n) be the nth Fibonacci number

Base case:
fib(0) = 1, fib(1) = 1

Subproblem:
fib(n) = fib(n-1) + fib(n-2)

Example:
f(5)
= f(4) + f(3)
= f(3) + f(2) + f(2) + f(1)
= f(2) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)
= f(1) + f(0) + f(1) + f(1) + f(0) + f(1) + f(0) + f(1)
= 1 + 1 + 1 + 1 + 1 + 1 + 1
= 8
```

Dynamic Programming

```
Let fib[n] be the nth Fibonacci number

Base case
fib[0] = 1
fib[1] = 1

for x from 1 to N
    fib[x] = fib[x-1] + fib[x-2]
```

```
Example:
N = 10
fib[0] = 1
fib[1] = 1
fib[2] = 2
fib[3] = 3
fib[4] = 5
fib[5] = 8
fib[6] = 13
fib[7] = 21
fib[8] = 34
fib[9] = 55
fib[10] = 89
```

8.1.2 Code

Recursion

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

Dynamic Programming

```
public int fib(int n){
    int fibArr[] = new int[n];
    fibArr[0] = 1
    fibArr[1] = 1
    for(int x=2; x<n; x++){
        fibArr[x] = fibArr[x-1] + fibArr[x-2];
    }
    return fibArr[n];
}
```

8.2 Coin Problem

Let's say that you wanted to make change for \$51 using the smallest amount of bills (\$1, \$2, \$5, \$10, \$20). We can use a greedy approach by always taking the highest bill that can be subtracted to find the smallest amount of change. 51 - 20 = 31 - 20 = 11 - 10 = 1. So the smallest amount of change would be comprised of 2 x \$20 + 1 x \$10 + 1 for a total of 5 bills. This solution seems very easy to implement, but what if the bills were not so nice?

Imagine that an alien currency was in denominations of \$3, \$5, \$7 and \$11. What would be the smallest amount of bills to make change for \$13? Note that a greedy approach does not work for this alien currency. For example: 13 - 11 = 2. It is impossible to make change using the greedy approach. Note that we can make change with 2 x \$5 + 1 x \$3 = \$13.

Let's define the problem more formally: Given a list of bills each with a positive denominations, find the lowest amount of bills required to make C dollars or return impossible if it cannot be done.

The base case for 0 dollars is very simple. There are 0 bills to make 0 dollars.

Lets try to simply this problem. Assume we only have one bill worth d dollars. If we have C dollars and we want to use the bill, then we will have C-d dollars left. So the minimum bills to make C dollars is the minimum number of bills to make (C - d) bills plus the 1 for the bill we used. For example, we want to make 10 dollars and we only have 2 dollar bills. If we use a 2 dollar bill, then the minimum bills required to make 10 dollars is the minimum number of bills required to make 8 dollars plus that 2 dollar bill. To find the minimum number of bills required to make 8 dollars, we can find the minimum number of bills to make 6 dollars plus one more 2 dollar bill. As you can see it is the same as before and it is a recursive function!

Recursion

```
Let d be the denomination
Let bills(C) be the minimum number of bills to make C dollars.

Base Case:
bills(0) = 0
bills(C) = impossible if C < 0

Subproblem:
bills(C) = bills(C-d) + 1 if bills(C-d) is possible
           impossible if bills(C-d) is impossible

Example:
C = 10
d = 2

bills(10)
```

```
= bills(8) + 1
= bills(6) + 1 + 1
= bills(4) + 1 + 1 + 1
= bills(2) + 1 + 1 + 1 + 1
= bills(0) + 1 + 1 + 1 + 1 + 1
= 5
```

We can implement this using dynamic programming but it will not give us much benefit since the recursion has no recalculations.

```
bills[0] = 0

for c from 1 to C
    if c-d >= 0 and bills[c-d] is not impossible
        bills[c] = bills[c-d] + 1
    else
        bills[c] = impossible

Example:
C = 10
d = 2

bills[0] = 0
bills[1] = impossible
bills[2] = 1
bills[3] = impossible
bills[4] = 2
bills[5] = impossible
bills[6] = 3
bills[7] = impossible
bills[8] = 4
bills[9] = impossible
bills[10] = 5
```

Now let's consider the problem with multiple bills of denominations d1,d2 .... dn and we want to make C dollars. If we use a d1 bill, then we will have C-d1 dollars left and similarly if we use a d2 bill then we will have C-d2 dollars left. More generally, if we use a dn bill, then we will have C-dn dollars left. If we want to find the minimum bills to make C dollars we should try to use every bill and see which requires the minimum number of bills. So the minimum bills to make C dollars is the minimum of C - dn bills plus one more dn bill for all bills of denomination dn. For example, if we have \$7 and we have bills \$3,\$4, and \$5, the minimum number of bills to make \$7 is the minimum of the minimum number of bills to make \$4 plus one more \$3 bill, minimum number of bills to make \$3 plus one more \$4 or the minimum number of bills to make \$2 plus one more \$5 bill.

```
Let denom be a list of denominations
Let bills(C) be the minimum number of bills from denominations to make

Base Cases:
bills(0) = 0
bills(C) = impossible if C < 0

Subproblem
bill(C) = {  minimum of bills(C - d)+1 for d in denom if bills(C-d) is
             {  impossible if bills(C - d) is impossible for all d in 

Example:
denom = [3,4,5]

bill(7)
= min(bill(4)+1,bill(3)+1,bill(2)+1))
= min(min(bill(1)+1,
          bill(0)+1,
          bill(-1)+1)+1,
      min(bill(0)+1,
          bill(-1)+1,
          bill(-2)+1) +1,
      min(bill(-1)+1),
          bill(-2)+1,
          bill(-3)+1)+1)
= min( min(impossible, 1, impossible) +1,
          min(1,impossible,impossible) +1,
          min(impossible,impossible,impossible)+1)
= min(2,2,impossible)
= 2
```

However note, that we are recomputing multiple things. For example we are recomputing bill(0) multiple times and bill(-1) multiple times. If instead we worked the solution up instead of down, we can find bill(C) more efficiently. We can do this by computing bill(c) as c goes from 0 to C. Eg. we compute bill(0) then bill(1) then bill(2) .... until bill(C).

Putting it all together:

```
Let bills[C] be the smallest amount of bills to make the amount C, or impossib
Let denom be an array of denominations

Base case:
 bills[0] = 0

Subproblem:
for c from 1 to C
     bills[c] = impossible
     for d in denom
          if c-d >= 0 and bills[c-d] is not impossible:
               bills[c] = min(bills[c], bills[c-d]+1)

Example:
denom = [3,4,5]
C = 7
 bills[0] = 0
 bills[1] = impossible (bills[-2],bills[-3],bills[-4])
 bills[2] = impossible
 bills[3] = 1 (bills[0]+1)
 bills[4] = 1 (bills[0]+1)
 bills[5] = 1 (bills[0]+1)
 bills[6] = 2 (bills[3]+1)
 bills[7] = 2 (bills[3]+1 or bills[4]+1)
```

8.3 Knapsack Problem

Imagine you are a robber and you have found a large stash of valuables. Each valuable has a value and a weight. You can only hold 10kg in your bag and you want to find the highest valued haul you can get away with.

Necklace: $10, 1kg Stack of cash: $270, 3kg Jewelry: $665, 7kg Rare painting: $900, 9kg

Let's try a greedy approach: we will take the items with the highest value to weight ratio.

Necklace: $10/kg Stack of cash: $90/kg Jewelry: $95/kg Rare painting: $100/kg

The greedy approach will choose the rare painting and the necklace for a total of $910. However if we take the jewelry and the stack of cash we will get \$935 and still fit it into the bag. How can we solve this problem? The answer is dynamic programming.

Let's first write a more formal definition of the problem:

Given unlimited quantities of N items, each associated with a positive weight and value, and a maximum total weight W that we can hold, what is the maximum value we can hold.

Let's write a more specific version of the problem: we want to find the maximum value that a bag with weight capacity W can carry out of N items of positive values and weights.

The base case for this is trivial. With zero weight, the maximum value you can have is 0.

Let's try simplifying the problem by using only one item with weight w and value v and a knapsack with maximum weight capacity W. Suppose we want to add the item into the knapsack then we have W-w capacity remaining and the value of the item. So the maximum value of knapsack with capacity W is the maximum value of W-w plus one more the item's value v. The maximum value of W-w is the maximum value of W-2w + 2v. As we can see, its the same subproblem as before and we have found a recursive relation.

For example if we had an item with value \$5 and weight 4kg and a knapsack with capacity 9kg. The maximum value that knapsack of 8kg can contain is the maximum value of a knapsack of 5kg plus \$5 for the item we put in the bag and so forth.

```
Let knapsack(W) be the maximum value of items that can fit into maximum
Let w be the weight of one item and v be the value of one item.

Base Case:
knapsack(0) = 0
knapsack(W) = 0 if W < w

Recursion:
knapsack(W) = knapsack(W-w) + v

Example:
v = 5
w = 4
knapsack(9)
= knapsack(5) + 5
= knapsack(1) + 5 + 5
```

= 10

Note that we are recomputing multiple values multiple times. We can avoid this by using dynamic programming and working up with our solution instead of backwards.

```
Let knapsack[W] be the maximum value with weight capacity W
Let w be the positive integer weight of the item and v be the value of an item

for k from 0 to w-1
    knapsack[k] = 0

for weight from 1 to W
    knapsack[weight] = knapsack[weight-w]+v

Example:
W = 9
w = 4
v = 4

knapsack[0] = 0     base case
knapsack[1] = 0     base case
knapsack[2] = 0     base case
knapsack[3] = 0     base case
knapsack[4] = 5     (knapsack[0]+5)
knapsack[5] = 5     ^
knapsack[6] = 5     ^
knapsack[7] = 5     ^
knapsack[8] = 10    (knapsack[4]+5)
knapsack[9] = 10    ^
```

Now let's go back to the original problem. We have N items each of positive weight and value and we want to find the maximum value to be put into a knapsack of capacity W.

If no items are left to fit in our remaining capacity, then the maximum value must be $0. This is our base case.

If we use an item, we will have capacity W-wi and the added value will be vi. We want to try all items to place into the knapsack so we try every single item and find the maximum value out of the items. So the maximum value of capacity W is the maximum value of W-wi plus the added value of vi for all items. To find the maximum value of capacity W-wi we can do the exact same thing by trying to place each item and finding the maximum value out of the items.

For example we have a knapsack of capacity 10kg and necklaces worth \$30 with weight 3kg, paintings worth \$50 with weight 9kg and silver bars worth \$60 with weight 7kg. If we choose a necklace, we will have 7kg capacity left with the added value of \$30. If we choose a painting, we will have 1kg capacity left and added value of \$50. If we choose a silver bar, we will have 3kg capacity left and added value of \$60. With the remaining capacity, we can choose another item and do the exact same thing.

8.3.1 Formalization

```
Let knapsack(W) be the maximum value with maximum capacity W
Let weights be an array of weights where wi is the weight of the ith it
Let values be an array of values where vi is the value of the ith item

knapsack(W) = 0 if wi > W for all items

knapsack(W) = max(knapsack(W-wi) + vi for all items with wi <= W)
```

```
Let knapsack[w] be the maximum value with maximum capacity W
Let weights be an array of weights where wi is the weight of the ith it
Let values be an array of values where vi is the value of the ith item

for w from 0 to W
    knapsack[w] = 0

for w from 1 to W
    maxVal = 0
    for i in items
        if w-wi >= 0
            knapsack[w] = max(knapsack[w], knapsack[w-wi] + vi)
```

```
Example:
```

8.4 Number of Paths

We first examined the number of paths problem in advanced recursion. However, now that we know how to use dynamic programming, we can see that the recursive solution was very inefficient because we were recomputing values many times.

```
Let path(x,y) be the number of ways to get to the intersection at x and y

Base case:
paths(1,y) = 1
paths(x,1) = 1

Recurrence:
paths(x,y) = paths(x−1,y) + paths(x,y−1)

Example:
paths(3,5)
= paths(2,5) + paths(3,4)
= paths(1,5) + paths(2,4) + paths(2,4) + paths(3,3)
= 1 + paths(1,4) + paths(2,3) + paths(1,4) + paths(2,3) + paths(2,3) + paths(3
= 1 + 1 + paths(1,3) + paths(2,2) + 1 + paths(1,3) + paths(2,2) + paths(1,3) +
= 1 + 1 + 1 + paths(1,2) + paths(2,1) + 1 + 1 + path(1,2) + paths(2,1) + 1 + p
= 15
```

Instead of recomputing multiple values, we can build our solution upwards starting from 1,1

```
Let paths[x][y] be the number of ways to get from (1,1) to (x,y).

paths[1][1] = 1

for x from 1 to N
    for y from 1 to M
        paths[x][y] = paths[x−1][y] + paths[x][y−1]

Example:
N = 3
M = 5

1 1 1 1  1
1 2 3 4  5
1 3 6 10 15
```

8.5 Exercises

Given an array of N integers, find the largest sum that can be using consecutive integers. Given an array of N integers, find the longest increasing subsequence Given a matrix of NxN integers, find the maximum sum of a submatrix

Chapter 9

Strings

String problems are more important than ever before with the enormous amount of text and information that is now available. For example, if we search for keywords on Google out of the millions of articles, how can we do it in such a way that the retrieval is relevant, accurate and efficient? If we misspell the word ”shooting” as ”sohoting” how can we come up with a list of autocorrected words?

9.1 Pattern Matching

When we press ctrl+f to search for a word on a page which may contain tens of thousands of words, how can we do it quickly? More formally: if we have two strings A and B, how can we search for instances of A inside B in the quickest way?

9.2 String Distance

Given two strings A and B, how can we tell how similar the strings are?
Levenshtein Distance
Hamming Distance

9.3 Exercises

Given a string, count the number of palindromes greater than 1 character contained in it. E.g. abacca has 3: aba, cc, acca. Given a sentence, reverse the order of the sentence without using additional memory. For example: There are three blue cows reversed is cows blue three are There

Chapter 10

Basics

10.1 Data Types

We use closets or drawers to store our clothes and garages to store our cars. Similarly, we store different types of data in different kinds of data types. Most programming languages will support these data types and picking the right data type is important.

To define a variable of a data type we use the follow syntax:

datatype variable_name = init_val;

datatype is the type of variable variable_name is the name of the variable used init_val is the initial value of the variable

Example:

```
int x = 3;
double y = −4.5;
```

x is an integer, and we initialize it with the value 3. y is a double floating point number (essentially a decimal number) and we initialize it with -4.5

10.1.1 Boolean

A boolean is stored in a bit that is either true or false. Booleans are usually used as flags to store if something is one one state or the other.

Data type Number of bits Range

bool 2 bits true or false

10.1.2 Integer

An integer is any number that does not contain decimals. It is stored as binary number in memory. For example: 0, -5, 6 are integers.

Data type Number of bits Range
byte 8 bits -128 to 127
short 16 bits -32,768 to 32,767
int 32 bits 2,147,483,648 to 2,147,483,647
long 64 bits 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

10.1.3 Character

A character is any letter or symbol. For example: 'a','B','8','!'.

Characters are usually stored as a number and then displayed as a character by the computer. An encoding is a computer translation from a number to a character. We store simple characters such as lower case and upper case letters, numbers and common punctuation, in 8 bits (0-255) and we can use it to encompass the English language. For these 8 bits, we use an encoding called ASCII. For example: '0' is 48 and 'B' is 66. There are other encodings like Unicode which uses more bits to convert to more languages such as Chinese or Russian. For the most part, we will just use ASCII.

Data type Number of bits Range
char 8 bits 256 bits

10.1.4 Float

A float is a decimal stored as binary in memory. We use scientific notation to represent the decimal. Scientific notation is a decimal number ¡ 10 times some exponent of 10. For example: $8.23 * 10^4$ is in scientific notation. Decimals can be stored in 32 bits or 64 bits.

In a 32bit float, we have 1 bit for the sign (positive or negative), 23 bits for the significant figures (7 digits) and 8 bits for the exponent.

In a 64bit double we have 1 bit for the sign (positive or negative), 52 bits for the significant figures (16 digits) and 11 bits for the exponent.

Data type Number of bits Range
float 32 bits 3.4e038 to 3.4e+038
double 64 bits 1.7e308 to 1.7e+308

10.2 Arithmetic Operations

Java includes many built in operations for basic operations.

Operation Description Example

z = x + y Addition z = 1 + 2, z = 3

z = x - y Subtraction z = 3 - 1, z = 2

z = x * y Multiplication z = 3 * 2, z = 6

z = x / y Integer division z = 5 / 2, z = 2 or z = 9 / 3, z = 3

z = x Modulus (or remainder) z = 5

x++ Increment by 1 x=5, x++, x = 6

x– Decrement by 1 x=5, x–, x = 4

Boolean operators are operations between two booleans (true or false). Boolean operators:

Operation Description Example

x —— y Or (One have to be true) True —— False = True, True —— True = True, False —— False = False

x && y And (Both have to be true True && True = True, True && False = False, False && False = False

!x Not !False = true, !true = false

x == y Equality 1 == 1 = true, 1 == 3 = false

x != y Inequality 1 != 1 = false, 1 != 3 = true

x ¡ y Less than 2 ¡ 3 = true, 2 ¡ 2 = false, 2 ¡ 1 = false

x ¡= y Less than or equal to 2 ¡= 3 = true, 2 ¡= 2 = true, 2 ¡= 1 = false

x ¿ y Greater than 3 ¿ 2 = true,3 ¿ 3 = false, 3 ¿ 4 = false

x ¿= y Greater than or equal to 3 ¿= 2 = true,3 ¿= 3 = true, 3 ¿= 4 = false

exp ? x : y Ternary (if exp is true use x otherwise use y) true ? 3 : 4 = 3, (1 ¿ 2) ? 3 : 4 = 4

Assignment assigns a value to a variable.

```
int x = 3;
```

Operation Description Example

x = y Set x to y x = 4 (x = 4)

x += y Addition x += 4 (x = 7)

x -= y Subtraction x -= 3 (x = 0)

x *= y Multiplication x *= 3 (x = 9)

x /= y Division x /= 2 (x = 1)

x Modulus x

10.3 Conditional Statements

Let's say we wanted to send a greeting to someone depending on the time of day. If it's the morning, we should say 'Good Morning' otherwise we will say 'Hello'. Let's say we have the 24 hour time as an integer. How can we set a condition for morning and night? We use sometime called an "if statement".

```
if (test exp){
     do something;
else{
     do something else;
}
```

If test exp evaluates to true, then the computer will do something, otherwise, it will do something else. So for example:

```
if( time < 12){
    greeting = "Good Morning";
}else {
    greeting = "Hello";
}
```

If the current time in 24 hour format for hours is less than 12 (noon) then the greeting will be "Good Morning". Otherwise (anything greater than or equal to 12) the greeting will be "Hello".

Let's say we wanted to make this more complex. If the time is morning (7am - 12pm) we should set the greeting to "Good Morning", if the time is in the afternoon (12pm - 5pm), we should say "Good Afternoon", if the time is at night (5pm - 12am) we should set the greeting to "Good Night" and from 12am to 7am we should set the greeting to "Go to sleep!".

We can do this by using an "else if" which has the following structure:

```
if (test exp){
     do something;
}
else if(test exp2){
     do something2;
}
else if(test exp3){
     do something3;
}else {
     do something else
}
```

If test exp is true, the computer will do something, otherwise if test exp2 is true, the computer will do something2, otherwise if test exp3 is true, the computer will do something3 otherwise the computer will do something else. Note that the computer will only execute one of the "do somethings". The computer evaluates each of the test expressions from top to bottom. As soon as one of the test expressions evaluate to true, the computer will "do something" and then stop checking the other test expressions.

So for example:

```
if(time >= 7 \&\& time <= 12){
    greeting = "Good Morning";
}
else if(time >= 13 \&\& time <= 17){
   greeting = "Good Afternoon";
}
else if(time>=18 \&\& time <= 24){
   greeting = "Good Night";
}
else {
   greeting = "Go to sleep!";
}
```

10.4 Loops

Let's say we wanted to calculate the 4th power of two without using the math power function. We can do this easily with code.

```
int x = 2*2*2*2;
```

How about 10?

```
int x = 2*2*2*2*2*2*2*2*2*2;
```

How about 100? We could copy and paste this more times but this get very tedious. Most programming languages have something called a loop which does the same action a number amount of times.

A for loop structure is as follows:

```
for( first exp; test exp; next exp){
    //do something
}
```

Here are the steps for a for loop:

Execute first exp Evaluate test exp If test exp evaluates to true, go to step 4, otherwise break do something Execute next exp Go back to 3

```
int i = 0;
int x = 2;
for(i=1;i<100;i++){
    x = x * 2;
}
```

A while loop is similar to a for loop but simpler:

```
while(test exp){
     //do something
}
```

While the test expression is true, the computer will do something.

```
int n = 10;
while(n>0){
    System.out.println(n);
    n--;
}
```

We also have a do while loop which is similar to a while loop but it runs at least once:

```
do{
    //do something;
}while(test expression);
```

The do-while loop will first do something and then as long as test expression is true, the computer will do something. Do while loops are useful if we need to get the value of something first before we keep looping.

```
char c;
do{
   c = getChar();
}while(c=='N')
```

10.5 Exercises

(FizzBuzz) Print all numbers from 1 to 100. If the number is divisible by 3, print "Fizz". If the number is divisible by 5, print "Buzz". If it is divisible by both, print "FizzBuzz".

Example (from 10 to 16)

10 Buzz 11 12 Fizz 13 14 15 FizzBuzz 16

Chapter 11

Data_Types

We use closets or drawers to store our clothes and garages to store our cars. Similarly, we store different types of data in different kinds of data types. Most programming languages will support these data types and picking the right data type is important.

To define a variable of a data type we use the follow syntax:

datatype variable_name = init_val;

datatype is the type of variable variable_name is the name of the variable used init_val is the initial value of the variable

Example:

```
int x = 3;
double y = -4.5;
```

x is an integer, and we initialize it with the value 3. y is a double floating point number (essentially a decimal number) and we initialize it with -4.5

11.1 Boolean

A boolean is stored in a bit that is either true or false. Booleans are usually used as flags to store if something is one one state or the other.

Data type Number of bits Range

bool 2 bits true or false

11.2 Integer

An integer is any number that does not contain decimals. It is stored as binary number in memory. For example: 0, -5, 6 are integers.

Data type Number of bits Range
byte 8 bits -128 to 127
short 16 bits -32,768 to 32,767
int 32 bits 2,147,483,648 to 2,147,483,647
long 64 bits 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

11.3 Character

A character is any letter or symbol. For example: 'a','B','8','!'.

Characters are usually stored as a number and then displayed as a character by the computer. An encoding is a computer translation from a number to a character. We store simple characters such as lower case and upper case letters, numbers and common punctuation, in 8 bits (0-255) and we can use it to encompass the English language. For these 8 bits, we use an encoding called ASCII. For example: '0' is 48 and 'B' is 66. There are other encodings like Unicode which uses more bits to convert to more languages such as Chinese or Russian. For the most part, we will just use ASCII.

Data type Number of bits Range
char 8 bits 256 bits

11.4 Float

A float is a decimal stored as binary in memory. We use scientific notation to represent the decimal. Scientific notation is a decimal number ¡ 10 times some exponent of 10. For example: $8.23 * 10^4$ is in scientific notation. Decimals can be stored in 32 bits or 64 bits.

In a 32bit float, we have 1 bit for the sign (positive or negative), 23 bits for the significant figures (7 digits) and 8 bits for the exponent.

In a 64bit double we have 1 bit for the sign (positive or negative), 52 bits for the significant figures (16 digits) and 11 bits for the exponent.

Data type Number of bits Range
float 32 bits 3.4e038 to 3.4e+038
double 64 bits 1.7e308 to 1.7e+308

Chapter 12

Runtime_and_Memory

Computers are super fast at making calculations compared to humans, but humans have much more "memory" than computers currently do. For example, computers can add two 100-digit numbers together much more quickly than any human possibly can. However, the human brain can contain much more memory than humans.

Brains have trillions of connections between neurons and the estimated storage capacity is about 2.5 petabytes. That is approximately 340 years of TV shows that you could watch! Computers on the other hand, have much less memory than human brains. Standard computers have around 8GB of RAM and some higher end machines may have 16-32GB. Although hard-disks can store terabytes of memory, we use RAM (flash memory) when analyzing computer memory because it is much faster. As an analogy, RAM can be thought of as grabbing an object in another room whereas disk memory is driving 20 min away to get that object.

The neurons and connections in our brains allow us to store an immense amount of information in our brains and allows us to easily recognize patterns much better than computers. For example it is very easy for us to identify different objects around us (e.g. we can easily differentiate between an apple and an orange), but it is difficult for a computer to do this for the countless number of objects in the world. However this is currently changing as more research is being done in "deep learning".

Brain Computer

Memory 2.5 petabytes (approx. 340 years of TV shows) 16GB RAM

Processing 60Hz 2.7GHz Quad Core (1,000,000,000 times faster than humans)

12.1 Limits

There are many different methods of implementing different things but most of the time we care most about implementations that are the fastest and use the least amount of memory. Let's go through some basic benchmarks about computers that you should know. Adding two numbers takes a nanosecond (1 billionth of a second) for an average computer to process. For practical purposes, we'll assume that the average computer program can hold up to 1GB or RAM or about 250 million integers. We use RAM because it is flash memory and read/write is super fast in comparison to disk read/writes.

Memory Operations (per seconds)

1 GB (250,000,000 ints) 100,000,000 operations

12.2 Big O Notation

When we compare how efficient algorithms or data structures are to another, we want to be able to describe them such that they can be quantified. We use something called Big O notation to do this. Many algorithms and data structures will have its time and space complexities depend on the size of the inputs. The Big O notation takes the largest factor of an input to compare computation times / memory usage. When we take the largest factor, we ignore smaller factors, coefficients and constants because they do not matter at very large values. For example: $O(3n^2 + 12n + 20)$ is simply $O(n^2)$ because it is the largest factor.

Here is a list of common Big O notations based on complexity:

Big O Limit of N for 1 second (Standard processor)

O(1) Constant Time runtime independent of N

O(log N) Sublinear time a very big number

O(N) Linear Time 100,000,000

O(N log N) 5,000,000

$O(N^2)$ Quadractic Time 10000

$O(N^3)$ Cubic Time 450

$O(2^N)$ Exponential Time 27

O(N!) Factorial Time 11

Keep in mind that in a couple of years as technology improves, this chart will be outdated.

12.3 Runtime Analysis

Let us examine a function that takes in an array of size N and returns the maximum element. In a program we are usually concerned with two complexities: memory and time.

Code:

```
int findMax(int[] array){
  int maxVal = array[0]; //O(1) memory to store max element, O(1) time for assi
  for(int i=1;i<array.length;i++){ //O(n) loop runtime
      if(maxVal < array[i]){ //O(1) to compare runtime
         maxVal = array[i]; //O(1) to assign new value runtime
     }
  }
}
```

Memory: The array takes O(N) memory and storing the max value is O(1) more memory. but usually when analyzing programs, we ignore the input memory sizes and take into account additional memory that is required to produce the output. So the memory footprint of the function is O(1).

Time: The first assignment of maxVal takes O(1). The loop runs N-1 times and of each of those N-1 times it checks if the current array element is greater than the current max which takes O(1). If it is greater, then we reassign maxVal which is O(1). When analyzing time complexity, we usually take worst case so we have: O(1+(N-1)(2)) and this simplifies to O(N) since it is the largest factor.

12.3.1 Example

```
void zeroArrays(int[][] grid){
  for(int i=0; i<grid.length;i++){
    for(int j=0;j<grid[i].length;j++){
      grid[i][j] = 0;
    }
  }
}
```

Chapter 13

Advanced_Recursion

Prerequisites: Recursion

Next: Intermediate Recursion, Backtracking, Dynamic Programming

Recursion can sometimes be very difficult to wrap your head around because when you try to go deeper, it only keeps getting deeper. It is sometimes hard to figure out the recurrence relation, but once you do, the problem becomes simple to solve.

13.1 Number of paths

Let's say we have a grid of N rows and M columns. How many ways are there to get from the bottom left corner to the top right corner using the intersection points and only going upwards or rightwards?

We see that the only way to reach an intersection is to come from the left intersection or come from the bottom intersection. So the number of ways to reach an intersection is the number of ways to reach the intersection to the left plus the number of ways to reach the intersection on the bottom.

So more accurately, the number of ways to get to intersection N x M is the sum of the ways to get to the intersection N-1 x M and the intersection N x M-1. However, the number of ways to reach the intersection N-1 x M is the sum of the number of ways to the intersection N-2 x M and N-1 x M-1. As we can see, it is a recurrence relation.

The base case is the starting intersection in the bottom left hand corner (the intersection 1x1) which is the only way to reach that intersection. We can also see that there is only one way to reach the intersections on the left side. There is only one way to reach the intersections on the bottom side as well.

```
Let path(x,y) be the number of ways to get to the intersection at x anc

Base case:
paths(1,y) = 1
paths(x,1) = 1

Recurrence:
paths(x,y) = paths(x-1,y) + paths(x,y-1)

Example:
paths(3,5)
= paths(2,5) + paths(3,4)
= paths(1,5) + paths(2,4) + paths(2,4) + paths(3,3)
= 1 + paths(1,4) + paths(2,3) + paths(1,4) + paths(2,3) + paths(2,3) +
= 1 + 1 + paths(1,3) + paths(2,2) + 1 + paths(1,3) + paths(2,2) + paths
= 1 + 1 + 1 + paths(1,2) + paths(2,1) + 1 + 1 + path(1,2) + paths(2,1)
=

int paths(int n,int m){
    if(n==1||m==1){
        return 1;
    }
    return ways(n-1,m) + ways(n,m-1);
}
```

13.2 Towers of Hanoi

There are three poles and N disks where each disk is heaver than the next disk. In the initial configuration, the discs are stacked upon another on the first pole where the lighter discs are above the heavier discs. We want to move all the discs to the last pole with the following conditions:

Only be moved from one pole to another one at a time. The discs have to be stacked such that all the lighter discs are on top of the heavier ones.

Lets' try to make this problem simpler. To move N discs from the first pole the the last pole we need to move N-1 discs to the middle pole, then move the Nth disc to the last pole, and then move all N-1 discs from the middle pole back on to the last pole.

Let the starting pole be the first pole, the helper pole be the middle pole and the destination pole the third pole.

To move N discs from the starting pole to the destination pole:

Step 1:

We need to move N-1 discs from the starting pole to the helper pole.

Step 2:

We need to move the Nth disc from the starting pole to the destination pole.

Step 3:

We need to move N-1 discs from the helper pole to the destination pole.

We can see that Step 2 is easy, all we have to do is move that one disc. But for Step 1 and Step 3, we have to move N-1 discs. How can we move N-1 discs to the middle pole?

We see that we can use the same reasoning: we need to move N-2 discs to the third pole. Then we need to move the N-1 disc to the second pole and then move N-2 discs from the third pole to the second pole.

In this case, the start pole is the first pole, the helper pole is the third pole and the destination pole is the middle pole.

To move N-1 discs from starting pole to destination pole:

Step 1:

We need to move N-2 discs from starting pole to helper pole

Step 2:

We move the N-1th disc from the starting pole to the destination pole

Step 3:

We move N-2 discs from the helper pole to the destination pole.

As we can see, the steps to move N discs are the exact same as to move N-1 discs! The only difference is that the actual poles are different but if we reassign poles to new roles every time we move down a disc, we can build a recurrence relation.

In Step 1, when we move N-1 discs from start to helper, the new helper is the old destination and the new destination is the old helper.

In Step 3, when we move N-1 discs from the helper to the destination, the new helper is the start pole, and the new start pole is the helper

13.2.1 Formalization

```
Let f(N,start,helper,dest) be the steps to move N discs from the start to dest

Base case:
f(1,start,helper,dest)= Move from (start) to (dest)
```

```
Recurrence:
f(N, start, helper, dest)

=

f(N-1, start, dest, helper)
Move from start to dest
f(N-1, helper, start, dest)

Example:
Let A, B, C be pole 1, 2, 3

f(4,A,B,C)

=

f(3,A,C,B)
Move from A to C
f(3,B,A,C)

=

f(2,A,B,C)
Move from A to B
f(2,C,A,B)
Move from A to C
f(2,B,A,C)
Move from B to C
f(2,A,B,C)

=

f(1,A,C,B)
Move from A to C
f(1,B,A,C)
Move from A to B
f(1,C,B,A)
Move from C to B
f(1,A,C,B)
Move from A to C
```

```
f(1,B,C,A)
Move from B to C
f(1,A,B,C)
Move from B to C
f(1,A,C,B)
Move from A to C
f(1,B,A,C)

=

Move from A to B
Move from A to C
Move from B to C
Move from A to B
Move from C to A
Move from C to B
Move from A to B
Move from A to C
Move from B to A
Move from B to C
Move from A to C
Move from B to C
Move from A to B
Move from A to C
Move from B to C
```

13.3 Permutations

A permutation is an arrangement of the original set of elements.

For example permutations of A,B,C,D,E,F:

D,E,F,C,B,A F,C,D,B,A,E B,D,A,E,F,C A,B,C,D,E,F

Given a string S of length N, how can we generate all permutations?

Let's assume that we have a list of permutations for the substring of S of N-1 characters. Then to get the permutations for the string of length N, all we need to do is insert the Nth character in between all the positions of each permutation of N-1 characters. For example permutation of A,B,C,D.

We can manually find the permutations of A,B,C.

A,B,C A,C,B B,A,C B,C,A C,A,B C,B,A

If we insert the letter D between each letter for every permutation we get:

D,A,B,C A,D,B,C A,B,D,C A,B,C,D D,A,C,B A,D,C,B A,C,D,B A,C,B,D .... etc

And we can guarantee that every new permutation will be unique. (Try to prove that to yourself).

But let's look at how we can get the permutation of A,B,C. We can also get all the permutations of the string by taking the permutations of A,B and inserting C in all the positions for all substrings.

Substrings of A,B

A, B B, A

Insert C for all positions for all permutations

C,A,B A,C,B A,B,C C,B,A B,C,A C,B,A

We see that we have a recurrence relation. To get the permutations of a string N, we take the string[1..N-1] and we insert the Nth character at every position for each permutation of the N-1 substring. The base case of an empty string is simply an empty string. Permutation of an empty string is an empty list.

For simplicity, S[i..j] will mean the substring from including i to excluding j.

```
Let P(S) be the list of permutations for the string S of length N

Base case:
P('') = []

Recurrence:
Let N = length of string S
P(S) = ps[0..i] + S[N] + ps[i..N] for i in 0..N, for ps in P(S[0..N−1]

Example:
P('ABC')
=
```

13.3.1 Implementation

```
Vector<String> permutation(String s){
    int n = s.length;
    Vector<String> vec = new Vector<String>();
    if(s.length==0)return vec;
    Vector<String> subvec = permutation(s.substring(0,n−1));
    for(int i=0;i<subvec.size();i++){
```

```
        String ps = subvec.get(i);
        for(int j=0;j<n;j++){
            vec.push(ps.substring(0,j)+s.charAt(n)+ps.substring(j,n-1));
        }
    }
    return vec;
}
```

13.4 Exercises

Given a string S, write a recursive function to generate all substrings Write a solution for hanoi towers but with the restriction that discs can only be moved from adjacent poles. (You can move a disc from A to B but not A to C because they are not adjacent)

Chapter 14

Bubble_Sort

Bubble sort is one of the most basic sorting algorithms, one of the first you should learn.

Its name describes how the algorithm works: bigger bubbles float to the top. This will become more clear when we see the implmentation.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

14.1 Implementation

```
//sorts integers from smallest to greatest
public static void BubbleSort (int a[]){
    for (int i=0;i<a.length;i++){
        for (int j=1;j<a.length;j++){
            //if the current bubble is smaller than the bubble that's
            //below then the bigger bubble floats up
            if (a[j]<a[j-1]){
                //swap the bubbles
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

Chapter 15

Selection_Sort

Selection Sort is similar to Bubble Sort in terms of complexity and arguably more intuitive.

Selection Sort works by selecting the smallest element and setting it as the first item in the array, then selecting the second element and setting that as the second item in the array and so on. This process repeats until the least smallest element (the largest element) is set as the last item in the array.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

15.1 Implementation

```
//sorts integers from smallest to greatest
public static void SelectionSort (int a[]){
    //iterates through the array selecting the smallest elements
    for (int i=0;i<a.length-1;i++){
        int minIndex = i;//var for storing where the ith smallest element is
        //iterates through the array looking for the ith smallest element
        for (int j=i+1;j<a.length;j++){
            //if the current element is smaller than the current smallest elem
            if (a[j]<a[minIndex]){
                //updated the location of the smallest element
                minIndex = j;
            }
            //swap the current i with the smallest element
            int temp = a[i];
```

```
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }
    }
}
```

Chapter 16

Insertion_Sort

Insertion Sort is the another n¡sup¿2¡/sup¿ sorting algorithm.

It works by starting with an empty sorted sequence of numbers and then inserting every element in the array into the sorted sequence maintaining the order after each insert.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

16.1 Implementation

```
//sorts integers from smallest to greatest
public static void InsertionSort (int a[]){
    //iterates through the array selecting the smallest elements
    for (int i=0;i<a.length;i++){
        //element that needs to be inserted into the sorted list
        int val = a[i];
        //keeps track of where to insert the element
        int index = i;

        //loops through the already sorted list to find where to
        //insert the current element
        while (index > 0 \&\& a[index-1] > val){
            //switches the current item at index with the one before
            //to check if the element that needs to be inserted will
            //go in the next spot
            a[index] = a[index-1];
            index--;
```

```
        }
        a[index] = val;
    }
}
```

Chapter 17

Heap_Sort

Prerequisites: Heap

Heap sort is a sort that inserts all the element in an array into a min heap and then pops all the element outs. Since a heap guarantees that the root node will be the smallest element, we can store the entire array in a heap and the order that they are popped out of the heap will be in order.

17.0.1 Implementation

```
public void heapSort(int[] arr){
    PriorityQueue<Integer> pq; //built in heap
    for(int i=0;i<arr.length;i++){
        pq.push(arr[i]);
    }
    for(int i=0;i<arr.length;i++){
        arr[i] = pq.pop(i);
    }
}
```

Chapter 18

Merge_Sort

Preqrequistes: Recursion

Merge sort works by breaking down the sorting into smaller pieces. If we want to sort N elements, we can sort the first half of the elements, sort the second half and then merge the results together. To sort the first half, we can do the exact same thing of sorting the first quarter and the second quarter and merging the results.

18.1 Implementation

Merge sort work be breaking down the problem into smaller and smaller parts and then combing those parts to solve the big problem.

We can keep splitting the list into half until there each piece has 1 element or no elements. We can then combine the results of each piece repeatedly until the entire list is sorted.

Example:

18.1.1 Formalization

```
Let merge(arr1,arr2) combine two sorted arrays into one sorted array

merge(arr1, arr2)= { arr2 if arr1 is empty
                     { arr1 if arr2 is empty
                     { arr1.first + merge(arr1[1..arr1.length],arr2) if arr1.f
                     { arr2.first + merge(arr1[0..arr.length],arr2[1..length])

Let sort(arr) sort an array
```

```
Let middle be (0+arr.length)/2
sort(arr) = merge(sort(arr[0..middle]) ,sort(arr[middle..arr.length])
```

18.1.2 Code

For our Java implementation, instead of returning a new array every time we merge two arrays, we can create a temporary array to store the merged results and then move it back.

```
/**
 * Merges two segments of the same array. The two segments should be a
 * @param arr Array containing segments
 * @param start1 Start index of first segment
 * @param end1 End index of first segment
 * @param start2 Start index of second segment
 * @param end2 End index of second segment
 */

public static void merge(int arr[],int start1,int end1, int start2,int
    int arr2[] = new int[end2-start1+1];
    int begin = start1;
    int n = 0;

    //Pick smallest element one by one
    while(start1 <= end1 \&\& start2 <= end2){
        if(arr[start1] <= arr[start2]){
            arr2[n] = arr[start1];
            start1++;
        }else{
            arr2[n] = arr[start2];
            start2++;
        }
        n++;
    }
    //If first segment still has elements
    while(start1 <= end1){
        arr2[n] = arr[start1];
        n++;
        start1++;
    }
    //if second segment still has elements
```

```
    while(start2 <= end2){
        arr2[n] = arr[start2];
        start2++;
        n++;
    }
    //Copy merged array back
    for(int i=0;i<n;i++){
        arr[begin+i]=arr2[i];
    }
}
/**
 * Sorts an array using merge sort
 * @param arr Array to be sorted
 * @param start Index at beginning of array
 * @param end Index at end of array
 */
public static void mergeSort(int arr[],int start,int end){
    if(end-start<=0){
        return;
    }
    int mid = (start+end)/2;
    mergeSort(arr,start,mid);
    mergeSort(arr,mid+1,end);
    merge(arr,start,mid,mid+1,end);
}

public static void main(){
    int arr[] = {5,7,2,6,8,5};
    mergeSort(arr,0,arr.length-1);
}
```

Chapter 19

Quick_Sort

Quick Sort is another fast sorting algorithm that uses divide and conquer. It is also known as partition-exchange sort or as Hoare's quicksort (named after the author).

In Quick Sort an element is selected as a "pivot". The list is then divided into two sublists: a list of elements less than (or equal to) the pivot and a list of elements greater than the pivot. Each sublist is sorted (conquered) and then appended together along with the origin pivot.

In the best case (if the pivot that is chosen is exactly the middle element), then the runtime is O(n log n). However, in the worst case, the runtime for Quick Sort is n^2. Despite its worst case behaviour, quick sort is still popular and in widespread use. The average case behaviour is quite good and there are straightforward methods that can be used to improve the selection of the pivot.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

19.1 Implementation

```
//sorts integers from smallest to greatest
public static void QuickSort (int a[], int first, int last){
    //first is the starting index of the list
    //last is the last index of the list
    //QuickSort can be called by QuickSort(a, 0, a.length-1)

    if (last <= first) return; //if the array size <= 1
```

```
    int pivot = a[first]; //picks the first element as the pivot
    // the location to put the next integer that's larger than the pivo
    int index = last;

    //iterate through the list, sorting it by the first pivot
    for (int i=last;i>=first+1;i--){
        if (a[i]>=pivot){
            //swap the current element to be on the side that's
            //larger than the pivot
            int temp = a[index];
            a[index] = a[i];
            a[i] = temp;

            //decrements index for the next switch
            index--;
        }
    }

    //swap the element at index with the pivot so that it's
    //in the right place
    int temp = pivot;
    a[first] = a[index];
    a[index] = pivot;

    //recursively sort the lists less than and greater than the pivot
    QuickSort (a, first, index-1);
    QuickSort (a, index+1, last);
}
```

Chapter 20

Bozo_Sort

Bozo sort is a sort that keeps randomly arranging an array until it is sorted. If you are very lucky bozo sort will be very fast! But for the most part this is a horrible sort.

20.1 Implementation

```
public void bozoSort(int[] arr){

   boolean sorted = false;
   int i = 0;

   while(!sorted){
        sorted = true;
        for(i=1;i<arr.length;i++){
            if(arr[i]<arr[i-1]){
                sorted = false;
                break;
            }
        }
        if(sorted)return;

        for(i=0;i<arr.length;i++){
             int x = Math.randInt(arr.length);
             int y = Math.randInt(arr.length);
             int temp = arr[x];
             arr[x] = arr[y];
```

```
            arr[y] = temp;
        }

    }
}
```

Chapter 21

Permutation_Sort

Permutation sort is a sort that keeps permutating the array until it is sorted. It is the slowest sort that will guarantee that the array will be sorted.

21.0.1 Implementation

```
void permuteSort(int[] arr){
  while(!sorted(arr)){
    permute(arr);
  }
}
```

Chapter 22

Miracle_Sort

Prerequisites: Miracles, Sense of Humour

Miracle sort is a sort that truly requires a miracle. We keep checking the array until it is sorted. It requires that some external force (a miracle?) changes some bits in the computer in a way that it becomes sorted.

22.1 Implementation

```
public void miracleSort(int[] arr){
    boolean sorted = false;
    do{
        sorted = true;
        for(int i=1;i<arr.length;i++){
            if(arr[i]<arr[i-1]){
                sorted = false;
                break;
            }
        }
    }while(!sorted);
}
```

Chapter 23

Arrays

Source on Github

Imagine you had a row of parking spaces where each was labelled with a number. If you wanted to know what the license plate of a car at parking space 4 was, all you have to do is go to the parking space and read off the license plate. If you wanted to park you car at parking space 5, you would go to parking space 5 and put your car in there if there was nothing there. Let's say that you had cars at parking spaces 1, 2, and 3. If you wanted to insert a new car at parking space 1 and keep the rest of the cars in the same order, you would have to shift the cars in the parking spaces from 1, 2 and 3 to 2, 3 and 4 by getting in each car and parking them in the new spaces which would take some time. This type of structure is an array.

An array is the most basic data structure that stores elements of the same type in a fixed block. The fact that it is in one block and the same type is important because it allows accessing elements very quickly if you have the index. All you have to do is go to the index and retrieve the element. However, inserting elements in the array is slow because you would have to shift all the elements and also if you want to shift past the fixed size you will get an error. (Imagine the parking spaces are full and you wanted to insert a car somewhere, there will still be one car that will have no parking space).

Arrays can be multidimensional meaning you can have an array of array of objects. (Imagine a parking lot with multiple rows of parking spaces).

Operation Create Get i Set i

Time Complexity O(n) O(1) O(1)

23.1 Implementation

Implementation of a very simple array in Java

23.1.1 Create array

Creating an array allocates a block of memory for us to store the elements.

```
public static int[] createArray(int size){
    int[] array = new int[size];
    return array;
}
```

23.1.2 Get

When we want to get the element of an index all we have to do is offset the index by the array. However we need to check that the index is within bounds of the array.

```
public static int getElemenent(int[] arr, int i){
    if(i<0 || i>=arr.length){
        throw new ArrayIndexOutOfBoundsException();
    }
    return arr[i];
}
```

23.1.3 Set

Setting is very similar to getting but instead we change the value in the array.

```
public static void setElemenent(int[] arr, int i,int value){
    if(i<0 || i>=arr.length){
        throw new ArrayIndexOutOfBoundsException();
    }
    arr[i] = value;
}
```

23.2 See Also

Vector Stack

Chapter 24

Stack

Imagine a stack of plates at a buffet, the plates are taken from the top and are also replaced from the top. The first plate to go in will be the last plate to come out. The last plate to go in will be the first to come out. This structure is called a stack.

A stack is an abstract data type with the property that it can remove and insert elements following a FILO (First In Last Out) structure. The first element to be inserted must be the last element to be removed and the last element to be inserted must be the first element to be removed. Sometimes, removal is called ”pop” and insertion is called ”push”.

Example of push:

Example of pop:

Stacks are used for function calls on the memory stack. Whenever a function is called, it is placed on the memory stack with its variables and when it is returning a value, it is popped off the stack.

A stack is usually implemented as a vector.

24.1 Implementation

Implementation Pop Push

Vector O(1) O(1)

24.2 Exercises

Given a string of brackets of either () or [], determine if the bracket syntax is legal (every opening bracket has a closing bracket from left to right).

Legal syntax:

([( ) [ ]]) ( ) ( ) [ ] ( ) ( )
Illegal syntax:
(( )] ( ) [( ])

Chapter 25

Vector

Prerequisites: Arrays, Stack

Source on Github

A vector is a stack that is implemented as an array. It is very similar to an array, but it is more flexible in terms of size. Elements are added and removed only from the end of the array. When more elements are added to the vector and the vector is at full capacity, the vector resizes itself and reallocates for 2*N space. When using an vector we can keep adding elements and let the data structure handle all the memory allocation.

Operation Get Push Pop Insert Delete

Time Complexity O(1) O(1) O(1) O(n) O(n)

25.1 Implementation

There is a builtin Vector class already, but we will go through the implementation of a simple integer vector class to understand how the data structure works.

25.1.1 Class

In our vector class, we need to store the element and the size of the current vector.

```
class Vec{
    private int[] arr; //Storage of elements
    private int size; //Current size
}
```

```
//Constructor
publc Vec(int startSize){
    arr = new int[startSize];
    this.size = 0;
}
```

25.1.2 Resize

Resize will be used to resize the current size of elements. We create a new array of two times the size of the old one and copy the old array over.

```
public void resize(){
    int[] newArr = new int[2*arr.length];
    for(int i=0;i<end;i++){
        newArr[i] = arr[i];
    }
    arr = newArr;
}
```

25.2 Add Element

Add element will add elements to the end of the vector. If the array is full, the vector will resize itself.

```
public void add(int x){
        if(size>=arr.length){
        resize();
    }
    arr[size] = x;
    end++;
}
```

25.3 Pop

Removes the element at the end of the vector. We decrease the size of the vector and return the last element.

```
public int pop(){
    if(size==0){
        throw new NoSuchElementException();
```

```
    }
    int ret = arr[size];
    size--;
    return ret;
}
```

25.4 Remove

Removes element at the index idx. It will throw an exception if the index is out of bounds.

We shift everything to the right of the index to the left by one to fill in the missing element.

```
public int remove(int idx){
    if(idx<0||idx>=size){
        throw new ArrayIndexOutOfBoundsException();
    }
    int ret = arr[idx];
    while(idx+1<size){
        arr[idx]=arr[idx+1];
        idx++;
    }
    size--;
    return ret;
}
```

25.5 Get Element

Returns the element at the specified index. It will throw an exception if the index is out of bounds. Note this function is exactly as the same as in the array

```
public int get(int idx){
    if(idx<0||idx>=size){
        throw new ArrayIndexOutOfBoundsException();
    }
    return arr[idx];
}
```

25.6 Insert Element

Insert the new number x at the index. We need to make space at the index for the new element so we shift everything to the right of the index by 1.

```
public void insert(int idx,int x){
    if(idx<0||idx>size){
        throw new ArrayIndexOutOfBoundsException();
    }
    size++;
    if(size>=arr.length){
        resize();
    }
    //Shift elements to the right or idx by 1
    int idx2 = size;
    while(idx2>idx){
        arr[idx2]=arr[idx2-1];
        idx2--;
    }
    arr[idx] = x;
    }
```

25.7 Exercises

Implement removeAtIndex(int index) for Vector

Chapter 26

Queue

Imagine you are standing in line for a restaurant. Whoever is first in line will be served first and whoever is last in line will be served last. People can be served while more people join the line and the line may get very long because it takes a while to serve one person while more people join the queue. This is called a queue.

A queue is an abstract data type with two functions, pop and push. Removal from the front is called "pop" or "dequeue". Insertion from the back is called "push" or "enqueue". A queue follows a First In First Out (FIFO) structure meaning the first element pushed should be the first element popped and the last element pushed should be the last element popped.

Queues are often used for buffer systems, for example a text message service. The messages that arrive at the server first are relayed first and the messages that arrive later are relayed later. If there are too many text messages in the system such that the rate texts are received overwhelm the number of texts that are sent the buffer may overflow and messages will get dropped. Most of the time this won't happen because the systems are designed to handle large loads, but if there were an emergency that caused everyone to start texting many texts could be dropped.

Example of push:

Example of pop:

26.1 Implementation

We can implement a queue most efficiently using a linked list because it has an efficient memory allocation.

Implementation Pop Push

Linked List O(1) O(1)

26.2 Exercises

Given a list of letters representing instructions where the first instruction is executed, output what the final list should look like after N instructions are executed.

First instruction:

A. Add B to the end of the list of instructions B. Do nothing C. Add two A's to the front of the list of instructions

Example:

ABC BCB CB AAB ABB BBB BB B

Chapter 27

Linked_List

Prerequisites: Queue

Source on Github

Imagine you had some train cars that were linked together where each was labelled on the inside with a different letter. If you had to find a specific letter you would have to start at the first train car and look inside to check the letter and then walk into the next train car to check the letter and so forth until you found the train car you wanted. If you wanted to insert a train car somewhere all you would have to do it unlink the position where you wanted to insert it and then relink the new train car with the other cars. If you wanted to remove a train car all you would have to do is unlink that car from the other cars and then create a new link to the cars that were adjacent to it. This sort of structure is called a linked list.

A pointer is something that holds the memory location of another object.

A linked list is similar to an array but it is different such that it is not stored in one block of data. Each element can be stored in a random place in memory but each element contains a pointer to the next element thus forming a chain of pointers. Think of a pointer as a link that links two train cars. Since the elements aren't in a block, accessing an element must be done by traversing the entire linked list by following each pointer to the next. However, this also allows insertion to be done more quickly by simply changing the point of the previous element and setting to the pointer of the current element to the next element. Deletion is also done by taking the previous element and changing its pointer to two elements ahead. In a linked list the links only go forward and you cannot move backward.

A doubly linked list is a linked list that has pointers going backwards as well as forwards.

Operation Get Push Delete Insert
Time Complexity O(n) O(1) O(1) O(1)

27.1 Implementation

In Java, there already exists a LinkedList class but we will implement our own.

27.1.1 Link Class

The Link class for each "link" in the Linked List. In each Link we only need the value and location of the previous and next node.

```
class Link{
    int value;
    Link next;
    public Link(int value){
        this.value = value;
        this.next = null;
    }
}
```

27.1.2 Class

Create the linked list by initializing the starting node as null and setting the size to empty.

```
class LinkedList{
    Link head;
    Link end;
    int size;
    public LinkedList(){
        start = end = null;
        size = 0;
    }
}
```

27.1.3 Push

Create a new node with the value given and add it to the end. We have to set the current head previous node to the new node and the new next next to the last node.

```
/*
 * Adds new node to head of linked list
 */
public void push(int value){
    Link newLink = new Link(value);
    if(size==0){
        head = end = newLink;
    }else{
        end.next = newLink;
        end = newLink;
    }
    size++;
}
```

27.1.4 Pop

Pops off the node at the head.

```
/*
 * Adds new node to head of linked list
 */
public int pop(){
    if(head==null){
        throw new NoSuchElementException();
    }
    int ret = head.value;
    head = head.next;
    size--;
    if(size==0){
        end = null;
    }
    return ret;
}
```

27.1.5 Get

Get retrieves the value at the specified index. We have to loop through the entire list to get the index we want because the nodes are not in the same block of memory.

```
/*
* Gets the value at index
*/
public int get(int index){
    int i = 0;
    Link curNode = head;
    while(curNode!=null){
        if(index==i){
            return curNode.value;
        }
        curNode = curNode.next;
        i++;
    }
    throw new NoSuchElementException();
}
```

27.1.6 Delete

To delete the current node we set the previous node next link to the link after.

```
/*
 * Deletes node after specified
 */
public void deleteNext(Link node){
    if(node.next==end){
        end = node;
    }
    node.next = node.next.next;
    size--;
}
```

27.2 Exercises

Implement a doubly linked list. A game is player by always eliminating the kth player from the last elimination and played until one player is left. Given N players each assigned to a number, find the number of the last player.

For example, you have 5 players (1,2,3,4,5) and the 3rd player is eliminated.

1, 2, 3, 4, 5 1, 2, 4, 5 (1, 2, 3 is eliminated) 2, 4, 5 (4, 5, 1 is eliminated) 2, 4 (2, 4, 5 is eliminated) 2 (2, 4, 2 is eliminated) Player 2 is the last one standing.

Chapter 28

Trees

Trees are data structures that follow a hierarchy, each node has exactly one or zero parents and each node has children. Trees are recursive structures meaning that each child of a tree is also a tree. A tree within another tree is called a subtree.

A child is a node that is below another node.

A parent is a node that is above another node.

The element at the top of the tree with no parents is called a root. The node at the bottom of the tree with no children is called a leaf.

Each node can hold different kinds of information depending on the tree. A node can hold the children it has, the parent it has, a key associated with the node and a value associated with the node.

A is the root of the tree and E,F,G,H are leaves. The parent of E is B. C,F,G is a subtree of the original tree.

28.1 Binary Tree

A binary tree is a tree where every node has at max two children.

Chapter 29

Binary_Tree

A binary tree is a tree such that each node has at most 2 children.

29.1 Binary Search Tree

A binary search tree is a type of binary tree where all the nodes in a left subtree will be smaller than the node and all the nodes in a right subtree will be greater than the node. It has a recursive structure such that each subtree is also a binary search tree.

Chapter 30

Sets

Imagine you have a grocery list that you use to keep tracking of things you need to buy. You want to make sure there are no duplicate items in the list, you can add items to the list and that you can remove items from your list. This structure is similar to what a set does.

Sets are abstract data structures which are able to store values and are used for three operations: insertion, deletion and membership test.

Insertion places an element into the set, deletion removes an element from the set and a membership test is checking whether an element exists within the set.

30.1 Implementation

Type Membership Insertion Deletion

Tree Set O(log n) O(log n) O(log n)

Hash Set O(1) O(1) O(1)

30.2 Exercises

Given a list of words, determine how many of them are anagrams of each other. An anagram is a word that can have its letters scrambled into another word.

For example silent and listen are anagrams but banana and orange are not.

Given two lists of friends, find the number of mutual friends. Given a list of numbers, find the number of tuples of size 4 that add to 0.

For example in the list (10,5,-1, 3, 4, -6) the tuple of size 4 (-1,3,4-6) adds to 0.

Chapter 31

Hash_Set

Prerequisites: Sets

Source on Github

Hash sets are sets that use hashes to store elements. A hashing algorithm is an algorithm that takes an element and converts it to a smaller chunk called a hash. For example let our hashing algorithm be (x mod 10). So the hashes of 232, 217 and 19 are 2,7, and 9 respectively.

For every element in a hash set, the hash is computed and elements with the same hash are grouped together and stored in a linked list. The linked list is called a bucket.

If we want to check if an element already exists within the set, we first compute the hash of the element and then search through the linked list associated with the hash to see if the element is contained.

Operation Membership Insertion Deletion

Time Complexity O(1) O(1) O(1)

31.0.1 Prerequisites

Sets Linked List

31.1 Implementation

Let use the example of the hashset of the elements of 3242, 3523, 123, 235 and 538. The hash set looks like this when computed:

If we wanted to check if 7238 was in the hash set, we would get the hash (7238 mod 10 = 8). So we get the bucket associated with the hash 8 and we get the list of (538). When we iterate through this short list, we see that 7238 is not a member of the set.

Similarly, if we wanted to insert 7238 into the hash set, we would check if it exists and if it did not we would append the element to the end of the bucket. For deletion we would find 7238 check if it existed in the set and remove it from the bucket.

Hash sets are very efficient in all three set operations if a good hashing algorithm is used. When the objects are that being stored are large then hash sets are effective as a set.

31.1.1 Class

Inside our implementation of a hash set we will store the buckets using an array of linked lists, the number of buckets, and the number of elements in the set.

The collision chance is the threshold for resizing the hash set. When the ratio of elements in the set to number of buckets is greater than the threshold, then the chance of collision will be high enough that it will slow down the operations. The lower this ratio, the better performing a hash set will be.

```
public class HashSet {

    public LinkedList<Integer>[] buckets;
    public int bucketsSize = 10;
    public int size = 0;
    public static final double COLLISION\_CHANCE = 0.3;

    public HashSet(){
        buckets = new LinkedList[10];
        for(int i=0;i<bucketsSize;i++){
            buckets[i] = new LinkedList<Integer>();
        }
        size = 0;
    }
```

31.1.2 Hash code

The hash code is the result of the hashing algorithm for an element. In our hash set implementation, we will use a simple hash: modulus of the integer by the number of buckets.

For the most part if the numbers are all random then the hash function is fine. However, if the number of buckets was 10 and we added the elements 20,30,40,50,60,70, they will all end up in the same bucket and results in poor performance.

```
public int getHash(int x,int hashSize){
    return x % hashSize;
}
```

31.1.3 Resize

A hash set must be able to resize. When the ratio of number of elements to number of buckets, the chance of collision will increase more and more. So we must able to resize the number of buckets to support the number of elements to lower the chance of collision.

To resize efficiently, we can create two times the number of buckets and set them to empty and then insert all the elements in the old buckets to the new buckets.

```
public void resize(){
    int newBucketsSize = bucketsSize*2;
    LinkedList<Integer>[] newBuckets = new LinkedList[newBucketsSize];
    for(int i=0;i<newBucketsSize;i++){
        newBuckets[i] = new LinkedList<Integer>();
    }
    for(int i=0;i<bucketsSize;i++){
        for(Integer y:buckets[i]){
            int hash = getHash(y,newBucketsSize);
            newBuckets[hash].push(y);
        }
    }
    buckets = newBuckets;
    bucketsSize = newBucketsSize;
}
```

31.1.4 Insert

To insert an element in a hash set, we get the hash code from our hashing algorithm and insert the element into the corresponding bucket.

The function will return method or not the operation was successful. If the bucket already contains the element the operation will stop because we do not want to add duplicate elements into the set. If the bucket does not contain the element, we will insert it into the bucket and the operation is successful.

```
public boolean insert(int x){
    int hash = getHash(x, bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.contains(x)){
        return false;
    }
    curBucket.push(x);
    if( (float)size/bucketsSize>COLLISION\_CHANCE){
        resize();
    }
    size++;
    return true;
}
```

31.1.5 Contains

To check if a hash set contains an element, we get the hash code from our hashing algorithm and check if the corresponding bucket contains the element.

```
public boolean contains(int x){
    int hash = getHash(x, bucketsSize);
    LinkedList<Integer> curBucket = buckets[hash];
    return curBucket.contains(x);
}
```

31.1.6 Remove

To remove an element from a hash set, we get the hash code from our hashing algorithm and remove the element from the corresponding bucket.

The function will return whether or not the operation was successful. If the bucket contains the element we can remove it from the linked list and the operation is successful. If the element is not in the bucket then the operation fails because we cannot remove something that is not there.

```
public boolean remove(int x){
```

```
    int hash = getHash(x, bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.remove((Integer)x)){
        return true;
    }
    return false;
}
```

31.2 Exercises

Try to come up with a better hashing algorithm Calculate the probability of a collision occurring given the number of buckets and number of elements in the hash set Given an array of numbers, find the number of pairs of numbers that sum to 0. Given an array of numbers and a number A, find the number of pairs of numbers that sum to A. Given an array of numbers and a number A, find the number of quadruples that sum to A.

Chapter 32

Tree_Set

Prerequeisites: Sets, Binary Search Tree

Source on Github

A tree set is a set which stores the values in a binary search tree. To store elements in a tree set, they must be able to be sorted by a property. To insert an element, it is added to the binary tree. To delete an element, it is removed from the binary tree. To check for membership, we do a binary search for the element in the binary tree.

The advantage of tree sets is that they are maintained in a sorted order.

Operation Membership Insertion Deletion

Time Complexity O(log n) O(log n) O(log n)

32.1 Implementation

Tree Sets are implemented using binary search trees.

32.2 Exercises

Given a list of names, output all the unique names in alphabetical order

Chapter 33

Maps

A map is an abstract data type that stores key-value pairs.

Imagine you had a English dictionary. If you look up a word, you can find it's definition and read it out. For example if you looked up the word 'cat' in the English dictionary, you would look through the dictionary alphabetically until you found the word 'cat' and then you would look at the definition: 'a feline animal'. If you really wanted to, you could also add your own words into the dictionary and the definitions of your words. This type of structure is called a map.

Maps (also called dictionaries) are abstract data types that store pairs of key-values and can be used to look up values from the keys. The keys are like the words in an English dictionary and the definitions can be seen as the values. Maps are able to support insertion of key-value pairs, retrieve values from keys, and delete key-value pairs.

33.0.1 Prerequisites

Sets

33.1 Implementation

Type Get Insertion Deletion

Tree Map O(log n) O(log n) O(log n)

Hash Map O(1) O(1) O(1)

33.2 Exercises

Given a list of N strings, output the strings in alphabetical order and the number of times they appear in the list. 2, Given two list of N strings, output the

Chapter 34

Hash_Map

Prerequisites: Map, Hash Set

Source on GitHub

Hash maps are maps that use hash sets to store the keys.

34.1 Implementation

Here is a Java implementation of a hash map which is a modified version of a hash set.

34.1.1 Class

Inside our implementation of a hash map we will store the buckets using an array of linked lists, the number of buckets, and the number of elements in the set.

The collision chance is the threshold for resizing the hash set. When the ratio of elements in the set to number of buckets is greater than the threshold, then the chance of collision will be high enough that it will slow down the operations. The lower this ratio, the better performing a hash set will be.

```
public class HashMap {

    public LinkedList<Pair>[] buckets;
    public int bucketsSize = 10;
    public int size = 0;
    public static final double COLLISION\_CHANCE = 0.3;
```

```
    public HashMap(){
        buckets = new LinkedList[10];
        for(int i=0;i<bucketsSize;i++){
            buckets[i] = new LinkedList<Pair>();
        }
        size = 0;
    }
}
```

34.1.2 Hash

The hash code is the result of the hashing algorithm for an element. In our hash set implementation, we will use a simple hash: modulus of the integer by the number of buckets.

For the most part if the numbers are all random then the hash function is fine. However, if the number of buckets was 10 and we added the elements 20,30,40,50,60,70, they will all end up in the same bucket and results in poor performance.

```
public int getHash(int x,int hashSize){
    return x % hashSize;
}
```

34.1.3 Resize

A hash map must be able to resize. When the ratio of number of elements to number of buckets, the chance of collision will increase more and more. So we must able to resize the number of buckets to support the number of elements to lower the chance of collision.

To resize efficiently, we can create two times the number of buckets and set them to empty and then insert all the elements in the old buckets to the new buckets.

```
public void resize(){
    int newBucketsSize = bucketsSize*2;
    LinkedList<Pair>[] newBuckets = new LinkedList[newBucketsSize];
    for(int i=0;i<newBucketsSize;i++){
        newBuckets[i] = new LinkedList<Pair>();
    }
    for(int i=0;i<bucketsSize;i++){
        for(Pair p:buckets[i]){
```

```
                int hash = getHash(p.key, newBucketsSize);
                newBuckets[hash].push(p);
            }
        }
        buckets = newBuckets;
        bucketsSize = newBucketsSize;
}
```

34.1.4 Insert

To insert an element in a hash set, we get the hash code from our hashing algorithm and insert the element into the corresponding bucket.

The function will return method or not the operation was successful. If the bucket already contains the element the operation will stop because we do not want to add duplicate elements into the set. If the bucket does not contain the element, we will insert it into the bucket and the operation is successful.

```
public boolean insert(Pair p){
    int hash = getHash(p.key, bucketsSize);

    LinkedList<Pair> curBucket = buckets[hash];
    if(curBucket.contains(p.key)){
        return false;
    }
    curBucket.push(p);
    if( (float)size/bucketsSize>COLLISION\_CHANCE){
        resize();
    }
    size++;
    return true;
}
```

34.1.5 Get

To get the value from a hash set from a key, we get the hash code from our hashing algorithm of the key and find the key-value pair in the corresponding bucket.

```
public Pair get(int key){
    int hash = getHash(key, bucketsSize);
```

```
    LinkedList<Pair> curBucket = buckets[hash];
    for(Pair p:curBucket){
        if(p.key==key){
            return p;
        }
    }
    return null;
}
```

34.1.6 Remove

To remove an element from a hash set, we get the hash code from our hashing algorithm and remove the element from the corresponding bucket.

The function will return whether or not the operation was successful. If the bucket contains the element we can remove it from the linked list and the operation is successful. If the element is not in the bucket then the operation fails because we cannot remove something that is not there.

```
public boolean remove(int key){
    int hash = getHash(key,bucketsSize);

    LinkedList<Pair> curBucket = buckets[hash];
    for(Pair p: curBucket){
        if(p.key==key){
            curBucket.remove(p);
            return true;
        }
    }
    return false;
}
```

34.2 Exercises

Create a hash map for the English dictionary (word as keys, definition as value). You will need to create a hash function for strings.

Chapter 35

Tree_Map

A tree map is a map that stores the key value pairs in a tree set.
Operation Membership Insertion Deletion
Complexity O(log n) O(log n) O(log n)

35.0.1 Prerequisites

Tree Set Map

35.1 Implementation

Here is a Java implementation of a tree map:

35.1.1 Class

A pair is a key with a value. In this implementation we will use the value as a string.

```
class Pair{
    int key;
    String value;
    public Pair(int key,String value){
        this.key = key;
        this.value = value;
    }
}
```

A node is a node contained in the binary search tree. The node must store the child nodes and for simplicity of the implementation, we will store the parent node as well. Each node will also have a key value pair associated with it.

```
class Node{
    Pair pair;
    Node left;
    Node right;
    Node parent;
    public Node(Pair p){
        this.pair = p;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
    public void replaceChild(Node child,Node replacement){
        if(left==child){
            left = replacement;
                if(replacement!= null){
                    replacement.parent = this;
                }
            }
            if(right==child){
                right = replacement;
                if(replacement!= null){
                    replacement.parent = this;
                }
            }
        }
    }
}
```

In our tree map, we will store the root node (ancestor of all nodes) and the number of numbers.

```
public class TreeMap {

    int size;
    Node root;

    public TreeMap(){
        size = 0;
        root = null;
```

```
    }
}
```

35.1.2 Insert

To insert a key-value pair into the tree set, we first find where the key should be. If the key already exists,

```
public boolean insert(Pair p){
    if(root==null){
        root = new Node(p);
        return true;
    }
    Node curTree = root;
    while(curTree != null){
        if(p.key == curTree.pair.key){
            return false;
        }else if(p.key < curTree.pair.key){
            if(curTree.left == null){
                Node newTree = new Node(p);
                newTree.parent = curTree;
                curTree.left = newTree;
                return true;
            }
            curTree = curTree.left;
        }else {
            if(curTree.right == null){
                Node newTree = new Node(p);
                newTree.parent = curTree;
                curTree.right = newTree;
                return true;
            }
            curTree = curTree.right;
        }
    }
    return false;
}
```

35.1.3 Get

To get the value from a key stored in a tree set, we binary search for the key and then retrieve the key-value pair located at the node.

```
public Pair get(int key){
    Node curTree = root;
    while(curTree!=null){
        if(key==curTree.pair.key){
            return curTree.pair;
        }else if(key<curTree.pair.key){
            curTree = curTree.left;
        }else{
            curTree = curTree.right;
        }
    }
    return null;
}
```

35.1.4 Remove

Removing an element is a much more complex because we need to maintain the tree structure of the tree set when removing elements. First we locate the element that we want to remove. If the element is not there then the operation failed and we return false. If the element is there then are three cases we need to consider.

Case 1: Node is a leaf node

If the node we want to remove is the leaf node, we can simply remove it.

Case 2: Node has one child

If the node we want to remove has a child, we can replace that node with its' only child.

Case 3: Node has two children

We need to replace the node with the rightmost of the left subtree or the leftmost of the right subtree to maintain the order.

It does not matter which side we pick so we will use the rightmost of the left subtree. First we copy the value of the rightmost of the left subtree into the node that will be deleted.

Then we replace the rightmost of the left subtree with its left subtree.

```
public boolean remove(int key){
    //Get node to remove
```

```
Node curNode = root;
while(curNode!=null){
    if(key==curNode.pair.key){
        break;
    }else if(key<curNode.pair.key){
        curNode = curNode.left;
    }else{
        curNode = curNode.right;
    }
}
if(curNode==null){
    return false;
}
//Case 1: leaf node
if(curNode.left==null \&\& curNode.right==null){
    //if root
    if(curNode==root){
        this.root = null;
    }else{
        curNode.parent.replaceChild(curNode,null);
    }
}
//Case 2: one child
else if(curNode.left==null){
    //If root
    if(curNode==root){
        root = curNode.right;
        root.parent = null;
    }else {
        curNode.parent.replaceChild(curNode,curNode.right);
    }
}
else if(curNode.right==null){
    //If root
    if(curNode==root){
        root = curNode.left;
        root.parent = null;
    }else {
        curNode.parent.replaceChild(curNode,curNode.left);
    }
```

```
    }
    //Case 3: two children
    else {
        //Get rightmost of left subtree
        Node rightmost = curNode.left;
        while(rightmost.right!=null){
            rightmost = rightmost.right;
        }
        curNode.pair = rightmost.pair;
        rightmost.parent.replaceChild(rightmost, rightmost.left);
    }
    size--;
    return true;
}
```

35.2 Exercises

Given a list of N numbers, output the first M unique numbers.

Chapter 36

Priority_Queue

Prerequisites: Queue, Heap

Consider a waiting list for lung donors. The patients are given a score when they are placed on the waiting list by how much they need a lung based on their whether they smoke, risk factors, age, expected time left etc. When a lung is available, the patient with the highest score will get removed from the waiting list. During this time, more patients could be added to the queue. The behaviour is similar to a queue but instead of the first person getting in the queue getting a lung first, the person with the highest score will get it. This means that if Sam has a score of 60 and gets placed in the queue after Bob who has a score of 40, Sam will get the lung first even though Bob was in the queue before him.

A priority queue is an abstract data structure with two operations: push and pop. Push adds an element into the priority queue and pop removes the highest or lowest element.

A priority queue is usually implemented as a heap because it is the most efficient because of its structure.

36.1 Implementation

Implementation Push Pop

Heap O(log n) O(log n)

36.2 Applications

Priority queues are very efficient in its opertions O(log n) and it is used in many other algorithms such as:

Dijkstra's Prim's Kruskal Line Sweeping

36.3 Exercises

Given a list of N numbers, find the M largest numbers. (Note you can do better than O(N log N)) Given N lists of N numbers, find the N largest numbers.

Chapter 37

Heap

Prerequisites: Queue

Source on GitHub

Heaps are data structures that are able to pop the maximum or minimum value or push a value very quickly. Heaps are implemented as trees which have the property that a parent node must either be greater than all the elements in its left and right subtrees (a max heap) or less than all the elements in its left and right subtrees (a min heap). Priority queue's are most efficiently implemented as heaps. This guarantees that the maximum or minimum element is the root node.

Heaps store their data level by level in a binary tree. This allows us to store heaps in an array. The root index is 0. For every node, the left index can be found by using the formula 2=ind+1 and the right index can be found by using the formula 2=ind + 2. The parent of a node can be found by integer division with (ind-1)/2.

```
root = 0
left = index * 2 + 1
right = index * 2 + 2
parent = (index -1)/2
```

Indexes of a heap

Example Heap:

A heap has two operations: push and pop. Pushing an element into a heap adds it into the heap and the heap needs to ensure that the properties of the heap still hold. Popping removes an element from the top of the heap and the heap needs to ensure that the properties of the heap still hold.

Operation Heapify Resize Push Pop

Time Complexity O(n) O(n) O(log n) O(log n)

37.1 Implementation

Here is an implementation of a max heap. A heap needs to be able to resize, push an element and pop an element.

37.1.1 Class

```
public class Heap {

    public int[] arr;
    public int size;

    public Heap(int startSize){
        arr = new int[startSize];
        size = 0;
    }
}
```

37.1.2 Heapify

Heapify takes a random array of N elements and transforms it into a heap. The runtime of heapify is O(N).

```
public void heapify(int arr[]){
    this.arr = arr;
    for(int i=0;i<Math.floor(arr.length/2.0);i++){
        int idx = i;
        while(idx<size){
            int left = idx*2+1;
            int right = idx*2+2;
            if(left<size \&\& arr[left]>arr[idx]){
                int swap = arr[left];
                arr[left] = arr[idx];
                arr[idx] = swap;
                idx = left;
            }else if(right<size \&\& arr[right]>arr[idx]){
                int swap = arr[right];
                arr[right]=arr[idx];
                arr[idx] = swap;
                idx = right;
            }else {
```

```
                break;
            }
        }
    }
}
```

37.1.3 Resize

When the heap gets too full, we can resize it to make it bigger

```
public void resize(){
    int[] newArr = new int[arr.length*2];
    for(int i=0;i<size;i++){
        newArr[i] = arr[i];
    }
    arr = newArr;
}
```

37.1.4 Push

Pushes the number x into the priority queue. We can do this by adding it to the bottom of the heap and then keep swapping it upwards if it is greater than the parent.

```
public void push(int x){

    if(size>=arr.length){
        resize();
    }
    arr[size] = x;
    size++;

    //Make sure parent is > child from the last element
    int idx = size-1;
    int parent = (idx-1)/2;
    while(idx>0 \&\& arr[parent]<arr[idx]){
        int swap = arr[parent];
        arr[parent] = arr[idx];
        arr[idx] = swap;
        idx = parent;
        parent = (idx-1)/2;
```

```
    }
}
```

37.1.5 Pop

Popping removes the greatest element in the priority queue by removing the root which is guaranteed to be the greatest as property of a heap. After removing the root, we replace it with the element at the bottom of the heap and we can keep swapping it with its children until the heap property is satisfied.

```
public int pop(){
    if(size==0)return 0;
    int ret = arr[0];
    arr[0] = arr[size-1];
    size--;

    int idx = 0;

    while(idx<size){
        int left = idx*2+1;
        int right = idx*2+2;
        if(left<size \&\& arr[left]>arr[idx]){
            int swap = arr[left];
            arr[left] = arr[idx];
            arr[idx] = swap;
            idx = left;
        }else if(right<size \&\& arr[right]>arr[idx]){
            int swap = arr[right];
            arr[right]=arr[idx];
            arr[idx] = swap;
            idx = right;
        }else {
            break;
        }
    }
}
```

37.2 Applications

Heaps are very efficient in its opertions O(log n) and it is used in many other algorithms such as:
Dijkstra's Prim's Kruskal Line Sweeping

37.3 Exercises

Implement a min heap Prove heaps work

Chapter 38

Advanced_Graph_Theory

Prerequisites: Graph Theory

Advanced topics on graph theory.

38.1 Bipartite Graph

A bipartite graph is a graph which can be partitioned into two sets such that no nodes in a set connect to another node in the same set.

38.2 Special Paths

A path is a certain order of visiting objects.

38.2.1 Hamiltonian Path

A Hamiltonian Path is a path that visits every node exactly once.

38.2.2 Eulerian Path

A Eulerian Path is a path that visited every edge exactly once.

38.3 Special Cycles

A cycle is a path that ends up at the same starting position.

38.3.1 Hamiltonian Cycle

A Hamiltonian cycle is a cycle that visits every node exactly once at ends back at the start.

38.3.2 Eulerian Cycle

A Eulerian Cycle is a cycle that visited every edge exactly once at ends back at the start.

38.3.3 Travelling Salesman Problem

The Travelling salesman is the problem where a salesman wants to find a cycle that minimizes the total cost of weights used of edges.

38.4 Special Nodes

Some graphs may have nodes that have special properties.

38.4.1 Root

A node in a directed acyclic graph that has no ancestors is a root.

38.4.2 Center

The center of a undirected tree is the node that minimizes the sum of the distance to every other node. The center can be found be continuously stripping away leaf nodes (nodes with only one edge) layer by layer until either 1 or 2 nodes remain. The longest path in a tree will contain the center.

38.5 Network Flow

38.5.1 Max Flow Problem

38.5.2 Min Cut Problem

38.5.3 Ford-Fulkerson

Chapter 39

Adjacency_Matrix

An adjacency matrix is a method of storing a graph using a two dimensional array. Given n nodes, the adjacency matrix can be stored in a n x n matrix.

Array[i][j] represents the weight between the node i and node j.

Example

1 2 3 4 5 6

1 0 1 0 0 1 0

2 1 0 1 0 1 0

3 0 1 0 1 0 0

4 0 0 1 0 1 1

5 1 1 0 1 0 0

6 0 0 0 1 0 0

39.1 Implementation

```
class edge{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}
public static int[][] getAdjMatrix(Vector<edge> edges){
    int n = 0;
    int adjMatrix[][] = new int[n][n];
```

```
    for(int i=0;i<n;i++)for(int j=0;j<n;j++)adjMatrix[i][j] = 0;

    for(int i=0;i<edges.size();i++){
        edge e = edges.get(i);
        adjMatrix[e.source][e.dest] = e.weight;
        adjMatrix[e.dest][e.source] = e.weight;
    }
    return adjMatrix;
}
```

Chapter 40

Prim's

Prerequisites: Priority Queue, Minimum Spanning Tree

A minimum spanning tree is a tree in a graph that connects all the nodes using the smallest cost total cost of edges.

40.1 Implementation

Prim's algorithm finds the minimum spanning tree using a greedy fashion. It works as such:

Pick an arbitrary node Find the closest node to that node Find the closest node to the 2 nodes Find the closest node to the 3 nodes ... Find the closest node to n-1 nodes

The closest node is the node with the lowest cost edge to the already connected nodes.

40.1.1 Example

40.1.2 Java Code

In Java, we need to specify a comparison for the Priority Queue to order. We do this by implementing the Comparable class and overriding the compareTo method.

adjList is an adjacency list that is an array of arrays that store the graph.

```
class node implements Comparable<node>{
    int weight,index;
    public node(int weight,int index){
```

```
        this.weight = weight;
        this.index = index;
    }
    public int compareTo(node e){
        return weight-e.weight;
    }
}
public static int Prims(Vector<Vector<node>> adjList){
    int cost = 0;
    int n = adjList.size();
    PriorityQueue<node> pq = new PriorityQueue<node>();
    boolean visited[] = new boolean[n];
    for(int i=0;i<n;i++){
        visited[i] = false;
    }
    int inTree = 1;
    visited[0] = true;
    for(int i=0;i<adjList.get(0).size();i++){
        pq.add(adjList.get(0).get(i));
    }
    while(!pq.isEmpty()\&\&inTree<n){
        node cur = pq.poll();
        if(visited[cur.index])continue;
        inTree++;
        visited[cur.index]=true;
        cost+=cur.weight;
        for(int i=0;i<adjList.get(cur.index).size();i++){
            pq.add(adjList.get(cur.index).get(i));
                }
    }
    //Graph is not connected
    if(inTree<n)return -1;
    return cost;
}
```

40.2 Applications

In networking, if a cable company wanted to connect all the houses with the least amount of wiring, the minimum spanning tree can be found that finds the least total cost of wire.

Minimum spanning trees can also be used in generating mazes.

40.3 Exercises

Prove that Prim's algorithm works Extends Prim's to output all the edges used Given a weighted graph with n nodes, find the smallest total cost to connect all nodes into 3 separate groups. (A single node can be a group) Same as 3, but a group must contain at least 3 other nodes.

Chapter 41

Kruskal

Prerequisites: Sorting, Minimum Spanning Tree

A minimum spanning tree is a tree in a graph that connects all the nodes using the smallest cost total cost of edges.

Kruskal's algorithm finds the minimum spanning tree using connected components.

If implemented efficiently using a priority queue to get the edges with minimum weight or a sorting the edges, the runtime is O(n log n).

41.1 Implementation

Uniquely label each node Take the edge with the minimum weight If the edge connects nodes A and B with different labels, all nodes with label B will be labeled with A. Otherwise, throw the edge away Repeat 2-3 until all the nodes have the same label

41.1.1 Example

41.1.2 Code

```
class edge implements Comparable<edge>{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
    public int compareTo(edge e){
```

```
        return weight-e.weight;
    }
}
public static int getParent(int parents[],int x){
    if(parents[x]==x)return x;
    parents[x] = getParent(parents,parents[x]);
    return parents[x];
}
public static int Kruskal(Vector<Vector<edge>> adjList){
    int n = adjList.size();
    int parents[] = new int[n];
    for(int i=0;i<n;i++)parents[i] = i;
    int sum = 0;

    PriorityQueue<edge> edges = new PriorityQueue<edge>();

    for(int i=0;i<n;i++){
        for(int j=0;j<adjList.get(i).size();j++){
            edges.add(adjList.get(i).get(j));
        }
    }

    while(!edges.isEmpty()){
        edge e = edges.poll();
        if(getParent(parents,e.source)!=getParent(parents,e.dest)){
            parents[e.source] = getParent(parents,e.dest);
            sum+=e.weight;
        }
    }

    return sum;
}
```

41.2 Applications

41.3 Exercises

Prove Kruskal's Algorithm works Extends Kruskals's to output all the edges used Given a weighted graph with n nodes, find the smallest total cost to connect all nodes into 3 separate groups. (A single node can be a group)

Same as 3, but a group must contain at least 3 other nodes.

Chapter 42

Floyd_Warshall

Prerequisites: Shortest Path, Dynamic Programming

Floyd Warshall is a algorithm for finding the shortest distances between all pairs of nodes in a graph. Floyd Warshall can be used to find negative cycles in the graph.

Description Time Space Detect cycles?

Computes shortest path between all pairs of nodes $O(n^3)$ $O(n^2)$ Yes

42.1 Implementation

Floyd-Warshall uses a dynamic programming approach to finding the shortest path between node A and node B. Every path from node A to node B can be rewritten as a path from A to some node in between plus the path from the node in between to node B. The shortest path from A to B can be found by finding a node C such the shortest path from A to C plus the shortest path from C to B is minimized.

42.1.1 Formalization

Recursion

```
Given a directed graph with N nodes and edges between nodes:
Let edge(i,j) be the weight of the edge from node i to node j in the graph
Let shortestPath(i,j) be the shortest path from i to j

Base Case:
shortestPath(i,i) = 0
shortestPath(i,j) = edge(i,j)
```

```
Recursion:
shortestPath(i,j) = minmum of shortestPath(i,k) + shortestPath(k,j) fo
```

42.1.2 Code

```
class edge{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}
public static final int UNDEFINED = Integer.MIN\_VALUE;

public static int[][] FloydWarshall(Vector<Vector<edge>> adjList){
    int n = adjList.size();
    //dist[i][j] is the minimum distance from i to j
    int[][] dist = new int[n][n];

    //initialize dist[i][j]
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            dist[i][j] = UNDEFINED;
        }
    }

    //dist[i][i] = 0
    for(int i=0;i<n;i++){
        dist[i][i]=0;
    }

    //initialize weights, dist[i][j] = edge from i to j
    for(int i=0;i<n;i++){
        for(int j=0;j<adjList.get(i).size();j++){

            edge e = adjList.get(i).get(j);
            dist[e.source][e.dest] = e.weight;
```

```
            System.out.println(e.source+" "+e.dest);
        }
    }

    for(int k=0;k<n;k++){
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                //If dist[i][k] and dist[k][j] have been set then use those va
                if(dist[i][k]!=UNDEFINED\&\&dist[k][j]!=UNDEFINED){
                    //If the new distance is less than current or not used, th
                    int newDist = dist[i][k]+dist[k][j];
                    if(dist[i][j] > newDist || dist[i][j]==UNDEFINED){
                        dist[i][j] = newDist;
                    }
                }
            }
        }
    }
    for(int i=0;i<n;i++){
        if(dist[i][i]<0){
            System.out.println("negative cycle");
        }
    }

    return dist;
}
```

42.2 Applications

Floyd Warshall is useful when you want to find the shortest distance between all possible pairs of nodes.

42.3 Exercises

Prove Floyd Warshall works Extend Floyd Warshall to reconstruct the paths from each pair of nodes

Chapter 43

Bellman_Ford

Prerequisites: Shortest Path

Bellman Ford is an algorithm that finds the shortest path from one source node to every other node in the graph. The running time is $O(n^2)$ and is slower than Dijkstra's but it is able to find negative cycles.

43.1 Implementation

Bellman Ford can be done using backtracking to find the shortest path in a graph. We first start at the starting node with starting cost of 0 and 0 edges used. For each node thats connected to that node, we repeat and add to the cost of the node.

We will do an example of the Bellman Ford algorithm on the above graph. At each node we have the node index and the current weight to reach that node. We start at node 0 with a weight of 0.

From node 0, we can reach node 1 and node 3. At node 1, we have an accumulative weight of 3. At node , we have an accumulative weight of 5.

From node 1, we can reach node 2 and node 4 with respective accumulative weights of 10 and 5.

From node 3 we can reach node 4 with an accumulative weight of 9.

From node 2, we can reach node 5 with an accumulative weight of 19.

From node 4, we can reach node 5 with an accumulative weight of 11.

From node 4, we can reach node 5 with an accumulative weight of 15.

```
Let N be the number of nodes in the graph
Let edges an adjacency list of the graph where:
    edges[source] contains all edges of the graph where source is the source e
An edge is represented as an object where:
```

```
    edge.weight is the weight of the edge
    edge.target is the target node of the edge
    edge.source is the source node of the edge
Let start be the starting node
Let shortestPath[target] be the shortest path from the source node to t

Let bellmanFord(target,n,w) be the shortest path from the source node t

bellmanFord(target, n , w)

Base Case:
bellmanFord(target, N , w):
    stop

Recurrence:
bellmanFord(source, n, w):
    shortestPath[source] = min(shortestPath[source], w)
    bellmanFord(edge.dest, n + 1, w + edge.weight) for edge in edges[so

Init:
shortestPath = [0] * N
bellmanFord(start,0,0)
```

We can rewrite this solution using dynamic programming without recursion.

```
class edge{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}

public static int BellmanFord(Vector<Vector<edge>> adjList,int startNod

    int n = adjList.size();
    //dist[i] is minimum distance from start to i
    int[] dist=new int[n];
```

```
//used[i] is if dist[i] has been initialized
boolean[] used = new boolean[n];

//initialize dist[i]=0 and used[i]=false
for(int i=0;i<n;i++){
    dist[i] = 0;
    used[i] = false;
}
used[startNode] = true;
dist[startNode] = 0;
for(int i=0;i<n-1;i++){
    //Iterate through adjacency list
    for(int j=0;j<n;j++){
        for(int k=0;k<adjList.get(j).size();k++){
            if(!used[j])continue;
            edge e = adjList.get(j).get(k);
            //If dist[e.source] has been used
            if(used[e.source]){
                //If new dist < cur dist or not used, then update
                int newDist = dist[e.source]+e.weight;
                if(newDist<dist[e.dest] || !used[e.dest]){
                    used[e.dest]= true;
                    dist[e.dest] = newDist;
                }
            }
        }
    }
}

for(int j=0;j<n;j++){
    for(int k=0;k<adjList.get(j).size();k++){
        edge e = adjList.get(j).get(k);
        //If negative cycle
        if(dist[e.source]+e.weight < dist[e.dest]){
            System.out.println("Negative cycle");
        }
    }
}

//If no path exists
```

```
    if(!used[endNode]){
        System.out.println("No path from start to end");
    }

    //Return distance from start to end
    return dist[endNode];
}
```

43.2 Applications

Arbitrage occurs when you can exchange currencies for another and make a profit. For example given a currency exchange table:
USD CAD EURO
USD / 1.12 0.72
CAD 0.90 / 0.64
EURO 1.38 1.56 /
Notice that 1 USD -¿ 1.12 CAD -¿ 1.008 USD. Bellman Ford can be used to find methods of arbitrage by using the vertex as currency and edges as transactions, and the weight as the exchange rate. All that is needed is to find a path that maximizes product of weights and finding a negative cycle.

43.3 Exercises

Write a program that detects a path for arbitrage to occur Prove Bellmand-Ford works

Chapter 44

Dijkstra's

Prerequisites: Shortest Path, Priority Queue, Greedy Algorithm

Dijkstra's is a greedy approach to find the shortest path in a graph with positive weights. It has many useful applications in networking and it can be extended to a variety of problems.

Dijkstra works by beginning at the starting node repeatedly picking the next closest node of those already visited.

If Dijkstra's is implemented using a priority queue, the run time is O(n log n).

If a negative cycle exists within the graph, then the algorithm breaks as it will repeatedly try to take the negative edges. See Bellman Ford to find negative cycles in a graph.

A naive fix for negative cycles would be to offset all edges by the largest negative edge and then subtract it from the resulting total but this does not work. Consider an example where you have start node A and end node B. The first route from A-¿B has length 2 and the second route has lengths 1,1,-2. Clearly the second route has less cost. If we try to make the length positive by adding all costs by 2, we will have the first path of length 4 and the second path of lengths 3,3,0 and the first route becomes the smallest total cost which is wrong.

44.1 Implementation

At each node we visit we keep track of the minimum cost it takes to reach to reach that node from the starting node.

Start at the starting node Find an unvisited node that has the least cost to reach from the visited nodes. Mark that node as visited Repeat until all nodes are visited

When we reach a node for the first time, it will be the shortest path to that node (Try to prove this to yourself).

We first start at the starting node. The distance from the starting node to the starting node is obviously 0.

From the starting node we have two nodes we can reach. The top node has a cost of 3 to reach and the bottom node has a cost of 5 to reach.

We pick the smallest node the reach and we mark it as visited. Once we visit a node, we can guarantee that it is the smallest cost to reach it. The next nodes minimum cost to reach is 10, 5 and 5.

We are indifferent to both 5's as they are both the minimum and we can choose either. We mark the node as visited and we find the minimum costs to other nodes which are 5,10,11.

We take the next smallest which is 5 and we mark the node as visited. The next costs are 10 and 11.

We take the smallest which is 10 and we now only have one last node to reach at a cost of 11.

At each node we have the minimum cost to get from the start node to each node.

44.1.1 Java

Implementation of Dijkstra in Java using a priority queue.

```
class node implements Comparable<node>{
    int weight,index;
    public node(int weight,int index){
        this.weight = weight;
        this.index = index;
    }
    public int compareTo(node e){
        return weight-e.weight;
    }
}

public static int dijkstra(int[][] adjMatrix,int start,int end){
    int n = adjMatrix.length;
    PriorityQueue <node> pq = new PriorityQueue<node>();
    boolean visited[] = new boolean[n];
    for(int i=0;i<n;i++)visited[i] = false;
    pq.add(new node(0,start));
```

```
    while(!visited[end] \&\& !pq.isEmpty()){
        node curNode = pq.poll();

        if(visited[curNode.index])continue;
        visited[curNode.index] = true;
        if(curNode.index==end){
            return curNode.weight;
        }
        for(int i=0;i<n;i++){
            if(adjMatrix[curNode.index][i]>0 \&\& !visited[i]){
                int newWeight = curNode.weight+adjMatrix[curNode.index][i];
                pq.add(new node(newWeight,i));
            }
        }
    }
    return -1;
}
```

44.2 Applications

In general, Dijkstra is usually the goto method for finding the minimum cost between two nodes in any kind of network. For example, Dijkstra can be used in networking to find the shortest path between two hosts. It can also be used in flight networking to find the cheapest cost to get from one airport to another airport.

44.3 Practice Exercises

Extend Dijkstra's to find the exact path from start to end (the order of nodes of the shortest path A-¿B-¿C) Extend Dijkstra's to find the best three minimal costs with unique paths from the start node to the end node Prove that Dijkstra's algorithm works

Chapter 45

Cycle_detection

A cycle occurs in a graph when a duplicate node is encountered when traversing a tree using a depth first search. In other words, a cycle occurs when you can reach the same node again.

An undirected graph where the number of edges is greater than or equal to the number of nodes will always have cycles.

45.1 Implementation

```
public static boolean hasCycle(int [][] adjMatrix){
        int[] visited = new int[adjMatrix.length];
        for(int i=0;i<adjMatrix.length;i++)visited[i] = 0;
        for(int i=0;i<adjMatrix.length;i++){
            if(hasCycleAt(adjMatrix,i,visited))return true;
        }
        return false;
    }

    public static boolean hasCycleAt(int[][] adjMatrix,int i, int visited[]){
        if(visited[i] == 1)return true;
        visited[i] = 1;
        for(int j = 0; j < adjMatrix.length;i++){
            if(adjMatrix[i][j] > 0){
                if(hasCycleAt(adjMatrix,j,visited))return true;
                visited[j] = 0;
            }
        }
```

```
        visited[i] = 2;
        return false;
    }
```

Chapter 46

Topological_Sorting

A topological sort or topological order of a directed graph is an order in which every node will come after its ancestors.

For example topological orders could be:

(A, B, C, D, E, F, G) (B, A, D, C, F, E, G) (B, A, D, G, F, C, E)

But (B, A, C, F, D, E, G) is not a topological ordering because D is an ancestor of F and it comes after F.

46.0.1 Prerequisites

Graph Theory Depth First Search

46.1 Implementation

Topological sort can implemented in O(n) time using DFS for a directed acyclic graph (a digraph with no cycles). How it works:

Start with an empty top order Pick any unmarked node Get the DFS preordering from that node for unvisited nodes Add the DFS to the head of the current order Mark every node that has been visited

Example:

Pick C DFS preorder from C is (C,E) Add DFS preorder to head [C,E] Pick F DFS preorder from F is (F) Add DFS preorder from F to head [F,C,E] Pick B DFS preorder from B is (B,D,G) Add DFS preorder from B to head [B,D,G,F,C,E] Pick A DFS preorder from A is (A) Add DFS preorder from A to head [A,B,D,G,F,C,E] Done, all nodes visited

A DFS order from a node is guaranteed to be a topological order. Since we add everything to the head of the order, a child of a node cannot appear before it.

Chapter 47

Connected_Components

A connected component is a subgraph where all the vertices in the subgraph connect to each other.

Finding the number of distinct connected components can be done using a breadth first search or a depth first search.

47.1 Implementation

Set each node's parent to itself Pick an unused edge in the graph that is from node A to node B, if the parent of A is not the same as parent of B then set all nodes whose parent is B to parents of A Repeat for all edges

47.1.1 Example

We select a random edge from node 1 and node 2. We set the parent node of 2 to node 1.

We select another random edge from node 5 to node 6.

We pick another random edge and we set the parent to 5.

We take another random edge and we set the parents to 7.

We take another random edge and we have an interesting case, two connected components are to be connected. We take all the nodes whose parent is 1, and we set the new parent to 5.

We take another random edge and set its parent to 1.

We take the last edge and set the last node parent to 7.

47.1.2 Java Code

```
public static int getParent(int x,int[] parent){
    if(parent[x]==x)return x;
    parent[x] = getParent(parent[x],parent);
    return parent[x];
}

public static void connected(int adjMatrix[][]){
    int n = adjMatrix.length;
    int[] parent = new int[n];
    int i,j;
    for(i=0;i<n;i++){
        parent[i] = i;
    }
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            if(adjMatrix[i][j]>0){
                int pi = getParent(i,parent);
                int pj = getParent(j,parent);
                if(pi!=pj){
                    parent[pj] = pi;
                }
            }
        }
    }
}
```

Chapter 48

Binary_Search

48.1 Binary Search

Binary search is a type of search that is able to find an object in a sorted list in O(log n). In binary search we first start at the middle element and we keep trying to halve the problem until we find the number.

48.1.1 Example

For example if I told you I had a number form 1 to 100 and I told you if your guess was higher or lower than my number you could use binary search to find it.

Eg: my number = 17

You guess: 50 I say lower.

So we know that:1number¡50. Since the number is ¡50 then we know we can eliminate all the numbers above 50. We just made the problem half as hard! The reason we picked 50 is important because it is the middle and it tells us the most information. If we picked 80 and the reply was higher it would narrow down the problem a lot, but if the reply was lower it would barely reduce the problem. Picking the middle works best because it tells us the most information if we get a ”lower” or ”higher” reply. So we should also guess the middle between 1 and 50.

You guess: 25. I say lower.

So we know that 1number¡25. Once again we made the problem half as hard again. Note that at every step we will make the problem half as hard. We need to pick the next middle number which is either 12 or 13, but we are indifferent because it will still tell us the most information (unless you get lucky).

You guess 13. I say higher.
So we know that 13¡number¡25.
You guess 19. I say lower.
So we know that 13¡number¡19.
You guess 16. I say higher.
So we know that 16¡number¡19
You guess 17. I say correct!

48.2 Implementation

This is a generic implementation of a binary search.

48.2.1 Generic Binary Search

```
void binarySearch(int ans,int minBound,int maxBound){
    while(maxVal>=minVal){
        int mid = (minVal+maxVal)/2;
        if(mid==ans)return;
        if(mid<ans)minVal = mid;
        else maxVal = mid;
    }
}
```

48.3 Exercises

Given a sorted array, find whether or not the number N exists Given a sorted array, find the the number of elements between the number A and number B inclusive. Example: 1,2 4, 6, 8, 10, 16, 20. Given A=5 and B=15, the number of elements between A and B is 3 (6, 8, 10) Given two sorted arrays, find the number of duplicate elements. Given two decimal numbers A and B, how do you find A/B without using division?

Chapter 49

Ternary_Search

Ternary search is a search that find local minimum or maximum values in a function given the interval between A and B.

If there are multiple local minimum and maximum values, ternary search will find one of them but not necessarily the maximum value of all points.

49.1 Implementation

Let's say we have a function f(x) with only one max point between A and B.

Let m1 by 1/3 of the way from A and B and let m2 be 2/3 of the way from B.

49.1.1 Proof

Case 1 : f(m1) ¡ f(m2)

Case 1.1: m1 ¡ m2 ¡ x

Case 1.2: m1¡x¡m2

x ¡ m1 ¡ m2 is not possible.

If f(m1) ¡ f(m2) then the max point must be in the middle third or last third.

Case 2: f(m1) ¿= f(m2)

Case 2.1: m1 ¡ x ¡ m2

Case 2.2: x ¡ m1 ¡ m2

m1 ¡ m2 ¡ x is not possible

If f(m1) ¿= f(m2) then the max point must be in the first third or middle third.

49.1.2 Example

Using the above proof, we can use ternary search to find the maximum point.

49.1.3 Formalization

```
Let f(x) be the function
Let A,B be interval

Let tern(a,b) return x where f(x) = maximum

Let m1 = a+(b-a)/3,
m2 = a+(b-a)*2/3

tern(a,b) = { if |f(a)-f(b)|<epsilon      (a+b)/2
            { if f(a) < f(b)               tern(m1,b)
            { else                         tern(a,m2)
```

49.1.4 Code

```
public double tern(double a,double b){
    if(Math.abs(f(a)-f(b))<0.0001){
        return (a+b)/2.0;
    }
    double m1 = a+(b-a)/3.0;
    double m2 = a+(b-a)*2/3;
    if(f(a)<f(b)){
        return tern(m1,b);
    }else {
        return tern(a,m2);
    }
}
```

Chapter 50

Depth_First_Search

Prerequisites: Recursion, Stack

A depth first search on a tree is a traversal of a tree that goes as far down as a branch as possible and then back.

A DFS can also be used on a graph to transverse it by ignoring nodes that have already been visited.

A DFS requires a stack but most the time DFS is implemented with recursion which uses the system stack. So most of the time you do not need an explicit stack.

50.1 Implementation

Most of the time, DFS is implemented using recursion and it is very short and simple to code.

```
public class Tree {
    int value;
    Tree left;
    Tree right;
}
```

50.1.1 Binary Tree Transversal

Implementation for outputting a binary tree in order from left to right using DFS

```
/**
```

```
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    DFS(cur.left);
    System.out.println(cur.value);
    DFS(cur.right);
}
```

50.1.2 Binary Tree Preorder

Implementation for outputting a binary tree in DFS pre order

```
/**
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    System.out.println(cur.value);
    DFS(cur.left);
    DFS(cur.right);
}
```

50.1.3 Binary Tree Postorder

Implementation for outputting a binary tree in DFS post order

```
/**
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    DFS(cur.left);
    DFS(cur.right);
    System.out.println(cur.value);
}
```

50.1.4 Graph Transversal

This is a DFS implementation for traversing a bidirectional graph with positive weights.

```
/**
 * Performs a DFS on a graph
 * adjMatrix - Adjacency matrix for a graph with positive edges. 0 indicates n
 * cur - Current node the DFS is at
 * visited - Mutable array which keeps track of which nodes have been visited
 */
public static void DFS\_graph(int[][] adjMatrix,int cur,boolean[] visited){
    if(visited[cur])return;
    visited[cur] = true;
    System.out.println(cur);
    for(int i=0;i<adjMatrix.length;i++){
        if(adjMatrix[cur][i]>0)DFS\_graph(adjMatrix,i,visited);
    }
    return;
}
```

50.2 Memory

Since DFS goes as deep as possible before going back, the stack we use will need to store at least the depth of the tree.

50.3 Exercises

Given a binary tree, find the height of it (the longest path from root to leaf) Given a graph, determine if it contains a cycle. Given a node and a binary tree, find the next node for post order, pre order and normal order.

Chapter 51

Breadth_First_Search

Prerequisites: Recursion, Queue

A breadth first search is a search that transverses level by level. For example in a tree, it will transverse everything from the first layer, to the second layer, the third layer and all the way down to the last layer. BFS is implemented with a queue.

Push root into queue Pop element from queue and push all non visited neighbours Repeat 2 until queue is empty

51.1 Implementation

Printing a binary tree using BFS:

```
void bfs(Node root){
    Queue<Node> q = new Queue<Node>();
    q.push(root);
    while(q.isEmpty()==false){
        Node cur = q.pop();
        System.out.println(cur.value);
        if(cur.left)q.push(cur.left);
        if(cur.right)q.push(cur.right);
    }
}
```

51.2 Exercises

Given a grid of squares with walls at certain locations and two locations A and B, find the minimum distance (going up/left/right/down) between the locations or impossible otherwise. For example if A is at (1,1) and B is at (3,1) but there is a wall at (2,1) then the minimum distance would be 4 (down, left, left, up). Given a tree of letters (A is the root), output the tree using BFS with separators between levels

Example: A-¿B, B-¿D,B-¿C, C-¿G will output A — B — C D — G

Given a tree of letters, and two letters X and Y determine if X is an ancestor of Y or Y is a ancestor of X or neither. An ancestor of a node is another node that is the root of a subtree that contains that node. Or simply parent of the node, grand parent, great grandparents etc.

Example: A-¿B, B-¿C, B-¿D, D-¿G, A is a parent of both C and D but G and C are not ancestors of each other

Given a binary tree and a node in the binary tree, find the next node in BFS order of the tree.

Chapter 52

Flood_Fill

Prerequisites: Depth First Search, Breadth First Search

Flood fill is a search that fills a grid from a start point to find the areas connected to the start point. For example the "bucket fill" in Photoshop or MS Paint uses flood fill to fill in the connecting areas with the same colour.

Flood fill can be implemented using a BFS or DFS.

52.1 General solution

Most flood fill solutions follow the same basic layout. There is a general DFS solution and a general BFS solution.

52.1.1 General DFS

```
//marked is initially a n by m boolean array of false
void floodFill(int i,int j){
    if(outOfBounds(i,j))return;
    if(visited(i,j))return;
    markVisited(i,j);
    floodFill(i+1,j);
    floodFill(i,j+1);
    floodFill(i-1,j);
    floodFill(i,j-1);
}
```

52.1.2 General BFS

```
void floodFill(int i,int j){
  Queue<Point> q;
  q.push(new Point(i,j));
  while(!q.isEmpty()){
    Point cur = q.pop();
    if(outOfBounds(cur))continue;
    if(visited(cur))continue;
    markVisited(cur);
    q.push(new Point(cur.x+1,cur.y));
    q.push(new Point(cur.x-1,cur.y));
    q.push(new Point(cur.x,cur.y+1));
    q.push(new Point(cur.x,cur.y-1));
  }

}
```

52.2 Bucket Fill

Given an n x m matrix and a start point and two colors (src and dst), we want to replace all the cells in the matrix that are connected to the start point with the color src and change to color dst as well as output the number of cells changed.

Example:

In a numeric representation of the colors:

52.2.1 DFS Solution

This is the DFS approach to the problem.

Assume that n,m are global integers that are the width and height of the image Assume that image is a global integer n by m matrix for the image Assume that visited is a global boolean n by m matrix that is initially all false

```
/* Changes all pixels that are equal to src to tar that are connected t
 * and returns the number of pixels changed
 */
public int FloodFillDFS(int x,int y,int src,int tar){
   if(x<0 || x>=n || y<0 || y>=m)return 0;
   if(visited[x][y])return 0;
   visited[x][y] = true;
   if(image[x][y]!=src)return 0;
```

```
    image[x][y] = tar;
    int sum = 0;
    sum+=FloodFillDFS(x+1,y,src,tar);
    sum+=FloodFillDFS(x-1,y,src,tar);
    sum+=FloodFillDFS(x,y+1,src,tar);
    sum+=FloodFillDFS(x,y-1,src,tar);
    return sum;
}

floodFillDFS(1,1,1,2);
```

52.2.2 BFS Solution

This is the BFS approach to the problem.

Assume that n,m are global integers that are the width and height of the image Assume that image is a global integer n by m matrix for the image Assume that visited is a global boolean n by m matrix that is initially all false

```
/* Changes all pixels that are equal to src to tar that are connected to start
 * and returns the number of pixels changes
 */
public int FloodFillBFS(int x,int y,int src,int tar){
    LinkedList<Point> q = new LinkedList<Point>();
    q.push(new Point(x,y));
    int total = 0;
    while(q.isEmpty()==false){
        Point cur = q.pop();
        if(cur.x<0||cur.x>=n||cur.y<0||cur.y>=m)continue;
        if(visited[cur.x][cur.y])continue;
                visited[cur.x][cur.y] = true;
        if(image[cur.x][cur.y]!=src)continue;
        image[cur.x][cur.y] = tar;
        total++;
        q.push(new Point(cur.x+1,cur.y));
        q.push(new Point(cur.x-1,cur.y));
        q.push(new Point(cur.x,cur.y+1));
        q.push(new Point(cur.x,cur.y-1));
    }
    return total;
```

```
}
FloodFillBFS(1,1,1,2);
```

52.3 Exercises

Given a grid and list of start points and walls, find the distance to the closest start point for every non-wall grid space without passing through a wall. Given a grid and list of cell coordinates, find the perimeter around the cells

Example: a single cell has a perimeter of 4, two cells joined side by side have a perimeter of 6

Chapter 53

Backtracking

Backtracking is a search that find all possible solutions by enumerating on a partial solution. Backtracking can be done using DFS or BFS. Generally, DFS will be better than BFS because backtracking is used to enumerate a large amount of solution. Since BFS requires storing each "level" of solutions and DFS requires storing each "height" of the solution.

Backtracking is similar to recursion, but instead of generating all the solutions, we will generate each solution one by one. In this way, we only need to store the current solution in memory whereas in normal recursion we need to store all the solutions into memory.

Since backtracking requires enumerating through all solutions, it is usually slow with runtimes usually O(n!) or O(2^n).

53.1 General Solution

```
Base case:
When a solution has been generated

Reject:
Check if partial solution needs to be rejected

Recurrence:
Generate next partial solution thats growing to full solution
```

```
backtrack(solution):
```

```
    if reject(solution)
         stop

    if base case
        stop

    backtrack( next\_solution ) for next\_solution from solution
```

```
Initial:
backtrack( empty\_solution)
```

53.2 List all sets

Given a set of numbers S of length N, output all subsets of S.

For example S=[1,2,3,4]. The subsets encodings of [1,2,3,4]:

1, 2, 3, 4 1,2, 1,3, 1,4, 2,3, 2,4, 3,4 1,2,3, 1,2,4,1,3,4,2,3,4 1,2,3,4

We want to be able to enumerate all the subsets of S so we need to find a way to encode a subset of an array. We can use a binary number of length N to encode a subset of an array of length N. For example a 1 represents we use a number in the set and a 0 means we don't use a number in the set.

So above:

= 0000 1 = 1000 2 = 0100 3 = 0010 4 = 0001 1,2 = 1100 1,3 = 1010 1,4 = 1001 2,3 = 0110 2,4 = 0101 3,4 = 0011 1,2,3 = 1110 1,2,4 = 1101 1,3,4 = 1011 2,3,4 = 0111 1,2,3,4 = 1111

At each position we either take or don't take the number in the set and we can do this for each number. We can enumerate through all these encoding by first starting with 0 or 1 and appending more 0's or 1's.

53.2.1 Recursive Method

Here is the way we would do this problem with recursion:

```
Start with []
Add 1 and 0 to the right of each binary number in the array
Repeat until N
[0,1]
[00,01,10,11]
[000,001,010,011,100,101,110,111]
```

```
[0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,11
Let S be an array of N integers
Let subsets(arr,n) be subsets of S from 1 to n

Base case
S(set,0) = set

Recurrence
S(set,n) = subsets([sub+0 for sub in set] + [sub+1 for sub in set],n)

Example

subsets([],4)
subsets([0,1],3)
subsets([00,01,10,11],2)
subsets([000,001,010,011,100,101,110,111],1)
subsets([0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101
=
[0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,11
```

Here is the way we would do this problem with backtracking, however it takes a lot of memory to store ALL the solutions. Instead of building all the solutions, we can build each solution one by one.

53.2.2 Formalization

```
Base case
subset(0, binary): print binary

Recurrence
subset(n−1,binary+'0')
subset(n−1,binary+'1')

Example
subset(4,'')

subset(3,'0')
subset(3,'1')

subset(2,'00')
```

```
subset(2,'01')
subset(2,'10')
subset(2,'11')

subset(1,'000')
subset(1,'001')
subset(1,'010')
subset(1,'011')
subset(1,'100')
subset(1,'101')
subset(1,'110')
subset(1,'111')

subset(0,'0000')
subset(0,'0001')
subset(0,'0010')
subset(0,'0011')
subset(0,'0100')
subset(0,'0101')
subset(0,'0110')
subset(0,'0111')
subset(0,'1000')
subset(0,'1001')
subset(0,'1010')
subset(0,'1011')
subset(0,'1100')
subset(0,'1101')
subset(0,'1110')
subset(0,'1111')
```

53.2.3 Implementation

```
void subsets(int arr[],bool use[],int i){
    if(i>=arr.length){
        for(int j=0;j<n;j++){
            System.out.print("%d ",arr[i]);
        }
        System.out.println();
        return;
     }
```

```
        use[i] = false;
        subsets(arr,use,i+1);
        use[i] = true;
        subsets(arr,use,i+1);
}
subsets([1,2,3,4],[0,0,0],0);

subsets([1,2,3,4],[0,0,0],0);

subsets([1,2,3,4],[1,0,0],0);
```

53.3 Permutation

Permutation can be

53.4 N Queen Problem

Find the number of ways to place N queens on a NxN board without any of them attacking each other. Queens attack each other by being along the same row, column or diagonal.

First we need to be able to encode a solution. There must be only one queen in each row, each column, each positive diagonal and each negative diagonal. We need a way to encode each row, column and diagonal to make it easy to check that they are all unique.

If we guarantee that all rows are unique and we have all columns filled, then we can guarantee that all the columns are unique as well.

To encode a diagonal we can use a clever equation to represent the diagonal. We can notice that every positive diagonal has the same value with row+col.

d1 = row+column

N=6:

Every square with the same d1 is in the same diagonal (/). For example on a 6x6 board: (0,2), (1,1), (2,0) all have d1 = 2 and are all along the same diagonal.

We can also notice that every negative diagonal has the same value of (N-row)+column. The second diagonal can also be found using another equation:

d2 = (N-row-1)+column

N=6:

Everything with the same d2 will be on the same diagonal. For example on a 6x6 board (0,0),(1,1),(2,2) have d2 = 5 and are all along the same diagonal.

Now that we get can encode rows, columns, and diagonals, we can use it to make checking solutions more easy. We can keep sets that track with rows/columns/diagonals are currently filled and check our solution to make sure we do not have anything in the same row/column/diagonal.

We can place a queen in each row and make sure that each column / diagonals is unfilled.

53.4.1 Formalization

```
Let N be a NxN board where we want to place N queens
Let queen(n,columns,d1,d2) be a placing of N queens across a board

Base case
queen(0,cols,d1,d2): print solution

Reject solution
reject(cols,d1,d2):    { false if duplicates in cols or d1 or d2
                              { true otherwise
Recurrence
queen(row,cols,d1,d2) = queen(row-1,cols with col,d1 with row+col,d2 w

Examples
N=4
queen(4,[],[],[])

queen(3,[0],[3],[6])
queen(3,[1],[4],[5])
queen(3,[2],[5],[4])
queen(3,[3],[6],[3])

queen(2,[0,0],[3,2],[6,5]) x— reject
queen(2,[0,1],[3,3],[6,4]) x— reject
queen(2,[0,2],[3,4],[6,3])
queen(2,[0,3],[3,5],[6,2])
queen(2,[1,0],[4,2],[5,5]) x— reject
queen(2,[1,1],[4,3],[5,4]) x— reject
queen(2,[1,2],[4,4],[5,3]) x— reject
```

```
queen(2,[1,3],[4,5],[5,2])
queen(2,[2,0],[5,2],[4,5])
queen(2,[2,1],[5,3],[4,4])  x— reject
queen(2,[2,2],[5,4],[4,3])  x— reject
queen(2,[2,3],[5,5],[4,2])  x— reject
queen(2,[3,0],[6,2],[3,5])
queen(2,[3,1],[6,3],[3,4])
queen(2,[3,2],[6,4],[3,3])  x— reject
queen(2,[3,3],[6,5],[3,2])  x— reject

queen(1,[0,2,0],[3,4,1],[6,3,4])  x— reject
queen(1,[0,2,1],[3,4,2],[6,3,3])  x— reject
queen(1,[0,2,2],[3,4,3],[6,3,2])  x— reject
queen(1,[0,2,3],[3,4,4],[6,3,1])  x— reject
queen(1,[0,3,0],[3,5,1],[6,2,4])  x— reject
queen(1,[0,3,1],[3,5,2],[6,2,3])
queen(1,[0,3,2],[3,5,3],[6,2,2])  x— reject
queen(1,[0,3,3],[3,5,4],[6,2,1])  x— reject
queen(1,[1,3,0],[4,5,1],[5,2,4])
queen(1,[1,3,1],[4,5,2],[5,2,3])  x— reject
queen(1,[1,3,2],[4,5,3],[5,2,2])  x— reject
queen(1,[1,3,3],[4,5,4],[5,2,1])  x— reject
queen(1,[2,0,0],[5,2,1],[4,5,4])  x— reject
queen(1,[2,0,1],[5,2,2],[4,5,3])  x— reject
queen(1,[2,0,2],[5,2,3],[4,5,2])  x— reject
queen(1,[2,0,3],[5,2,4],[4,5,1])
queen(1,[3,0,0],[6,2,1],[3,5,4])  x— reject
queen(1,[3,0,1],[6,2,2],[3,5,3])  x— reject
queen(1,[3,0,2],[6,2,3],[3,5,2])
queen(1,[3,0,3],[6,2,4],[3,5,1])  x— reject
queen(1,[3,1,0],[6,3,1],[3,4,4])  x— reject
queen(1,[3,1,1],[6,3,2],[3,4,3])  x— reject
queen(1,[3,1,2],[6,3,3],[3,4,2])  x— reject
queen(1,[3,1,3],[6,3,4],[3,4,1])  x— reject

queen(0,[0,3,1,0],[3,5,2,0],[6,2,3,3])  x— reject
queen(0,[0,3,1,1],[3,5,2,1],[6,2,3,2])  x— reject
queen(0,[0,3,1,2],[3,5,2,2],[6,2,3,1])  x— reject
queen(0,[0,3,1,3],[3,5,2,3],[6,2,3,0])  x— reject
queen(0,[1,3,0,0],[4,5,1,0],[5,2,4,3])  x— reject
```

```
queen(0,[1,3,0,1],[4,5,1,1],[5,2,4,2])  x— reject
queen(0,[1,3,0,2],[4,5,1,2],[5,2,4,1])  SOLUTION
queen(0,[1,3,0,3],[4,5,1,3],[5,2,4,0])  x— reject
queen(0,[2,0,3,0],[5,2,4,0],[4,5,1,3])  x— reject
queen(0,[2,0,3,1],[5,2,4,1],[4,5,1,2])  SOLUTION
queen(0,[2,0,3,2],[5,2,4,2],[4,5,1,1])  x— reject
queen(0,[2,0,3,3],[5,2,4,3],[4,5,1,0])  x— reject
queen(0,[3,0,2,0],[6,2,3,0],[3,5,2,3])  x— reject
queen(0,[3,0,2,1],[6,2,3,1],[3,5,2,2])  x— reject
queen(0,[3,0,2,2],[6,2,3,2],[3,5,2,1])  x— reject
queen(0,[3,0,2,3],[6,2,3,3],[3,5,2,0])  x— reject
```

53.5 Exercises

Given a sequence of numbers, output all increasing subsequences Given a NxN chessboard with certain squares that can have no pieces placed, output the number of configurations that can be made from rooks without attacking each other. Given a sudoku grid, output a solution for the grid if it exists. Note: this question is popular for technical interviews.

Chapter 54

Advanced_Dynamic_Programming

Dynamic programming is very powerful and more efficient than recursion for problems that recompute multiple values. However sometimes it is difficult to find an efficient dynamic programming solution and we will examine problems where we will need higher dimensions.

54.0.1 Prerequistes

Dynamic Programming

54.1 Longest Common Subsequence

A subsequence is a subset of the original sequence that is in the same order. For example in the string "abcdefghi", "aeg" is a subsequence but "eaq" is not because it is not in order.

The longest common subsequence between two strings A and B is the longest subsequence in A that is also in B.

For example given A="xyaaaabcdeg", B="bcaaaaefgxy" the longest common subsequence is "aaaaeg"

```
xyaaaabcde\_g
bcaaaa\_\_\_efgxy
```

If we try to use greedy we will see it doesn't work. For example if we use try to take as much as B as we can, we see that we will get BCEG or if we try to take as much as A we get XY.

Let first write a formal definition of the problem, given two strings A and B each with lengths N and M respectively we want to find the longest common subsequence between them.

[Note to make reading easier I have used short forms for substring and index. For example A[3] means the 3rd character of A and A[1..4] means the substring of A from the first character to the fourth character inclusive. A[0] represents the null substring of A]

The base is simple. The LCS of A[1..x] (where x is from 1 to N) and B[0] = 0. The LCS of B[1..x] and A[0] = 0.

We need to break this problem into subproblems.

If A[N] = B[M] and then the new LCS is the LCS of A[1..N-1] and B[1..M-1]. Note that if two strings have the same character at the end of their string it has to be part of the LCS. Try to prove this to yourself.

```
LCS of matched = (LCS of A from 1 to N-1, B from 1 to M-1) + A[N]
```

If A[N] != B[M] then we try to match A[N] with B[M-1] or A[N-1] with B[M]. Thus we take the LCS of A[1..N-1] and B[1..M] and the LCS of A[1..N] and B[1..M-1].

```
LCS of not matched = max ( (LCS of A[1..N], B[1..M-1]) , (LCS of A[1..
```

Putting it all together we have:

```
Let LCS[i][j] be the length of longest common subsequence of A[1..i] an

Base case:
LCS[i][0] = 0 where 0< i        n
LCS[0][j] = 0 where 0< j        m

Subproblem:
if A[i] == B[j], LCS[i][j] = LCS[i-1][j-1]+1 where 0< i        n and 0 <
if A[i] != B[j], LCS[i][j] = max( LCS[i-1][j], LCS[i][j-1] ) where 0<
```

54.2 Zero-one Knapsack Problem

In the Dynamic Programming section, we examined the knapsack problem:

Given an unlimited amount of N items with positive weights and values, we want to find the maximum value we can hold with a capacity.

Let's change the problem slightly such that there is only one of each object. The problem becomes slightly more difficult because we to take into account whether or not we have used an object before.

Given one of each N items with positive weights and values, we want to find the maximum value we can hold with a capacity W.

```
Let knapsack(N,W) be the maximum value of iterating through N items with a maxi
Let weights be an array of weights
Let values be an array of values

knapsack(i,0) = 0
knapsack(i,w) = 0 if no items can fit
knapsack(i,w) = 0 where i <= 0

knapsack(n,w) = max(knapsack(n-1,w-weight[n])+value[n], knapsack(n-1,w))
```

```
Let knapsack[N][W] be the maximum value of iterating through N items with maxim
Let weights be an array of weights where weights[i] is the weight of the ith it
Let values be an array of values where values[i] is the value of the ith item

for i from 0 to W
    knapsack[0][i] = 0

for n from 1 to N
    knapsack[n][0] = 0
    for w from 1 to W
        knapsack[n][w] = max(knapsack[n-1][w-weight[n]]+value[n], knapsack[n-1
```

Chapter 55

Coin_Problem

Let's say that you wanted to make change for \$51 using the smallest amount of bills (\$1, \$2, \$5, \$10, \$20). We can use a greedy approach by always taking the highest bill that can be subtracted to find the smallest amount of change. 51 - 20 = 31 - 20 = 11 - 10 = 1. So the smallest amount of change would be comprised of 2 x \$20 + 1 x \$10 + 1 for a total of 5 bills. This solution seems very easy to implement, but what if the bills were not so nice?

Imagine that an alien currency was in denominations of \$3, \$5, \$7 and \$11. What would be the smallest amount of bills to make change for \$13? Note that a greedy approach does not work for this alien currency. For example: 13 - 11 = 2. It is impossible to make change using the greedy approach.

55.1 Solution

Let's define the problem more formally: Given a list of bills each with a positive denomination d, find the lowest amount of bills required to make C dollars or return impossible if it cannot be done.

The base case 0 for this is very simple. There are 0 bills to make 0 dollars.

We can reduce this problem into subproblems. Let's assume that we have found out the lowest amount of bills required to make all the dollar amounts from 0 to C-1 or determined if it is impossible to do so. Lets take a look at an arbitrary bill b with denomination d. We know the minimum number of bills to make C-d (or if its impossible) based on our assumption that we have solved from 0 to C-1. Thus if use the bill b to make C then it is just the minimum number of bills to make C-d with 1 more bill so we add 1 more to that value. If we take the minimum value for all bills (if its possible to make C-d), we will get the lowest amount of bills required to make C.

Putting it all together:

```
Let bills[C] be the smallest amount of bills to make the amount C, or i

Base case:
bills[0] = 0

Subproblem:
bills[C] = min(bills[C-d]+1) for all bills where d is the denominator o
if bill[C-d] = impossible for all bills, then bills[C] is impossible
```

Example of previous problem where the bills are ($3, $5, $7, $11) and we want to find the minimum number of bills to make 13.

Let -1 be "impossible".

C 0 1 2 3 4 5 6 7 8 9 10 11 12 13

bills[C] 0 -1 -1 1 0 1 2 -1 2 3 2 1 4 3

55.2 Implementation

55.3 Exercises

Given a list of bills each with unique denomination d, find the number of ways to make C dollars.

For example given bills: $2, $3, $5, there are 2 ways to make $7 (2+5, 2+2+3)

Given a list of N integers, separate the list into two sets such that the difference is minimized and output the difference.

For example given integers: 1, 4, 10, 12, we can separate them into (4+10=14) and (1+12=13) so the minimum difference is 1.

Given a list of lengths, find the smallest area that can be created if the lengths are used to make a triangle.

For example, given lengths: 2,4,6,8,10 we can make a triangle with minimum area 43.3 if we use the sides (2+8=10, 4+6 = 10, 10).

Chapter 56

Knapsack_Problem

Imagine you are a robber and you have found a large stash of valuables. Each valuable has a value and a weight. You can only hold 10kg in your bag and you want to find the highest valued haul you can get away with.

Necklace: $10, 1kg Stack of cash: $270, 3kg Jewelry: $665, 7kg Rare painting: $900, 9kg

Let's try a greedy approach: we will take the items with the highest value to weight ratio.

Necklace: $10/kg Stack of cash: $90/kg Jewelry: $95/kg Rare painting: $100/kg

The greedy approach will choose the rare painting and the necklace for a total of $910. However if we take the jewelry and the stack of cash we will get $935 and still fit it into the bag. How can we solve this problem? The answer is dynamic programming.

56.1 Solution

Let's first write a more formal definition of the problem:

Given n objects, each associated with a positive weight and value, and a maximum total weight W that we can hold, what is the maximum value we can hold. In the zero/one knapsack problem, there is only one of each object so we either take it or leave it.

Let's write a more specific version of the problem: we want to find the maximum value that a bag with maximum weight W can hold of N objects with positive weight and value which we can either take or not take.

The base case for this is trivial. With zero weight, the maximum value you can have is 0.

We now have to break this problem down into subproblems.

Now we want to find the maximum value for a bag of maximum weight W and assessing all N objects. Since we have already assessed up to N-1 objects we only need to assess the Nth object. For the Nth object we can either take it or leave it.

If we leave it, then it is the same as a bag of maximum weight W with N-1 objects because we are just ignoring the Nth object.

If we take it, then we need to find the maximum value thats possible while making room for that object and add that to the value of the object.

If we want the maximum value of assessing N objects and maximum weight W then we want the max of leaving the Nth object and taking the Nth object so:

```
max value = max( maximum value taking , maximum value leaving)
max value = max( maximum value of N−1 objects with weight W, (maximum 
```

Putting it all together we have:

```
Let weight[i] be the weight of object i
Let value[i] be the value of object i

Let knapsack[i][j] be the maximum value that a knapsack of maximum wei

Base case:
knapsack[0][0] = 0

Subproblem:
knapsack[i][j] = max(knapsack[i−1][j−weight[i]], knapsack[i−1][j−1])
```

56.2 Implementation

56.3 Exercises

Given a list of n objects with a positive weight and value, find the maximum value that can be obtained with maximum weight W. However, each object can be used more than once.

Chapter 57

Pattern_Matching

Pattern matching is finding if a certain sequence of elements exists in another larger sequence of elements. For example we want to find if the string "abc" is in the string "abdabdbdacbaabcasd" (which the answer is yes).

57.1 Knuth Morris Pratt

If we have the needle string "abcxabcy" and haystack string "abcyabcxabcy" then our first search will be putting the needle at position 0 as we see the search fails at 'x'. However we note that we do not need to set the needle at position 1 because we have already done the search for the prefix "abc". Thus we can search starting by setting the needle at the next 'a'.

KMP uses this type of optimization for pattern matching by precomputing a table for the needle string.

57.2 Rabin Karp

57.3 Finite State Automata

57.4 Boyer Moore

Chapter 58

Boolean

A boolean is stored in a bit that is either true or false. Booleans are usually used as flags to store if something is one one state or the other.

Data type Number of bits Range

bool 2 bits true or false

58.1 Example

```
boolean w = false;
boolean x = true;
boolean y = (w || x);
boolean z = (1 == 4);
```

Chapter 59

Integer

An integer is any number that does not contain decimals. It is stored as binary number in memory. For example: 0, -5, 6 are integers.

Data type Number of bits Range
byte 8 bits -128 to 127
short 16 bits -32,768 to 32,767
int 32 bits 2,147,483,648 to 2,147,483,647
long 64 bits 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

59.1 Example

```
short x = 5;
int y = 10;
long z = 100;
```

Chapter 60

Character

A character is any letter or symbol. For example: 'a','B','8','!'.

Characters are usually stored as a number and then displayed as a character by the computer. An encoding is a computer translation from a number to a character. We store simple characters such as lower case and upper case letters, numbers and common punctuation, in 8 bits (0-255) and we can use it to encompass the English language. For these 8 bits, we use an encoding called ASCII. For example: '0' is 48 and 'B' is 66. There are other encodings like Unicode which uses more bits to convert to more languages such as Chinese or Russian. For the most part, we will just use ASCII.

Data type Number of bits Range

char 8 bits 256 bits

Chapter 61

Float

A float is a decimal stored as binary in memory. We use scientific notation to represent the decimal. Scientific notation is a decimal number ¡ 10 times some exponent of 10. For example: 8.23 * 10^4 is in scientific notation. Decimals can be stored in 32 bits or 64 bits.

In a 32bit float, we have 1 bit for the sign (positive or negative), 23 bits for the significant figures (7 digits) and 8 bits for the exponent.

In a 64bit double we have 1 bit for the sign (positive or negative), 52 bits for the significant figures (16 digits) and 11 bits for the exponent.

Data type Number of bits Range

float 32 bits 3.4e038 to 3.4e+038

double 64 bits 1.7e308 to 1.7e+308

61.1 Example

```
double x = 1.0/4.0;
double y = 1.78e5;
double z = −10.535246;
```

Chapter 62

Exercises

62.1 Interview Preparation

From personal experience, I know that programming interviews can be quite stressful. It is a much more different environmnet than you are probably used to which can throw you off your game. I fumbled through easy problems in my first few interviews as I was a nervous wreck from the time pressure as well as the glare of the interviewer. However, in later interviews, I became more accustomed to the process and became more comfortable with the interviewer. As you do more interviews in your life, you will be less nervous and become more confident in your abilities. The more prepared you are, the more likely your chances of success in the interview room. Below you will find a small guide to rocking the interview!

62.2 Before the Interview

Prepartion is key for interviews. You can be charming and confident as you want, but if you cannot solve their problems, your application will quickly be discarded. When you are given a problem, there is a chance you have solved or read about a solution to the problem. Hence, the more prepared you are, the more likely you will already have a solution. The sections in this handbook that come up often are: all of Data Structure, Advanced Recursion, Binary Search and Strings. If you cover those topics, you should be set for most interviews. However, if you are interviewing for higher end companies, you may need to learn a lot more. In some interviews, you will be asked to code your solution, therefore I suggest you get very familiar with a language and become very profficient in it. You should know how to do input/output, use the standard library and how to debug properly. Java is a good language to use as it has many built in data structures and has stack traces for debugging.

There are two types of interviews I have encountered: phone/Skype interviews and in person. If given a choice, in person is always preferrable. When you doing an in person interview, you are able to better communicate with your interviewer and build a rapport. You will also be show your thought process on paper which is easier to do than via phone or Skype. Additionally, depending on the company, if they fly you down to their headquarters for the interview, you'll get a free trip!

If the company you are interviewing for is a startup, you can dress casually. Otherwise, if the company is more corporate, then you might want to dress a little more formally.

62.3 During the Interview - Part 1: Behavioural

The interview usually starts with introductions and then the interviewer will usually ask you about projects you've worked on as well as past places you have worked. A common question is: what was the hardest part about that project and how did you solve it. (If you do not already have a personal project, then I highly recommend starting one. If personal projects do not interest you, then this field may not be for you. Hackathons are a good way of starting projects as you are able to dedicate lots of your time towards it). During this first part of the interview, try to build a rapport with the interviewer. You should be passionate about the work you have done and hopefully your interviewer is equally passionate and be able to relate to you. Like any normal interview, try to maintain eye contact and good posture.

62.4 During the Interview - Part 2: Technical

The second part of the interview usually consists of technical problems. If you are expected to code the solution, you should expect 1 medium to difficult problem (30-45 min) and possibly an easy problem (5-10 minutes). (Exception of Google, does 3-4 problems). If you are not coding, then you should expect 3-4 problems where you need to describe the solution.

62.4.1 Step 1 - Analyzing the Problem

1-2 minutes

When given the problem, make sure you read through the problem carefully and that you understand the specifications. Ask your interviewer if you are unsure of anything.

62.4.2 Step 2 - Find a solution

5-10 minutes

Now that you understand the problem, ask yourself if you've seen this problem before or any similar problem. If you have, then you should also remember the solution and you're in luck! However, if you don't remember the solution or have not seen the problem before, then you will need to think out the solution. First start with the naive solution which is usually bruteforce. An inefficient solution is better then no solution. Once you have that, start working towards a more optimal solution. You should be thinking aloud and letting the interviewer see your thought process. You can write down your solutions on paper so that it can be more clear to the interviewer. You should be able to explain why your solution is more efficient than another and why it works. If your solution is wrong, the interviewer may stop you and you should quickly find the mistake and determine why it is wrong. Keep going as far as you can without letting the interviewer help you. If you are stuck, the interviewer will likely drop a hint and you should be able to pick up from that. Once you have an optimal solution, you should be confident enough to prove that is optimal. If the interviewer asks if you can do better, then there may be the case that there is a better solution.

62.4.3 Step 3 - Start coding

10-20 minutes

Now that you have a solution, you need to prove to the interviewer that you can implement it. If you are profficient enough, you should be able to code and debug easily enough. You should comment your code and name your variables properly so your interviewer knows whats going on.

62.5 Practice Problems

I have included a mix of real interview problems that are used and some problems that I interesting. Good luck!

Chapter 63

Binary_Search_Tree

A binary search tree is a binary tree with special properties. The left children of a node will always be less than the node and the right children of a node will always be more than the node. It has a recursive structure such that every subtree is also a binary search tree.

Operation Membership Insertion Deletion

Time Complexity O(log n) O(log n) O(log n)

63.0.1 Prerequisites

Set Binary Tree

63.1 Implementation

In a binary search tree, everything to the left of a node is smaller than that node and everything to the right of that node is greater than that node.

This implementation of a binary search will be unbalanced. For a balanced binary search tree, see AVL trees and Red Black trees.

63.1.1 Class

A Node is a node in our tree set. Each node will contain a left subtree, a right subtree, the parent of the tree and the value stored at that node. It is unnecessary to store the parent but for this implementation it will be easier to keep track of the parent.

replaceChild replaces the left or right child node specified with the replacement node.

```
public class Node {
    int value;
    Node left;
    Node right;
    Node parent;
    public Node(int val,Node parent){
        this.value = val;
        this.left = null;
        this.right = null;
        this.parent = parent;
    }
    public void replaceChild(Node child,Node replacement){
        if(left==child){
            left = replacement;
            if(replacement!= null){
                replacement.parent = this;
            }
        }
        if(right==child){
            right = replacement;
            if(replacement!= null){
                replacement.parent = this;
            }
        }
    }
}
```

In our class we will store the number of element in the set and the root of the tree. From the root of the tree we can traverse the rest of the tree.

```
public class TreeSet {

    int size;
    Node root;

    public TreeSet(){
        size = 0;
        root = null;
    }
```

63.1.2 Insert

To insert an element in the tree set we search for the element that we are trying to insert. If it is already there then the operation fails because sets contain unique elements. Otherwise we will insert the new element into the set.

```
public boolean insert(int x){
    if(root==null){
        root = new Node(x,null);
        size = 1;
        return true;
    }
    Node curTree = root;
    while(curTree != null){
        if(x == curTree.value){
            return false;
        }else if(x < curTree.value){
            if(curTree.left == null){
                curTree.left = new Node(x,curTree);
                size++;
                return true;
            }
            curTree = curTree.left;
        }else {
            if(curTree.right == null){
                curTree.right = new Node(x,curTree);
                size++;
                return true;
            }
            curTree = curTree.right;
        }
    }
    return false;
}
```

63.1.3 Contains

To check if the tree set contains an element, we search for it in the binary tree by starting at the root. If the number is less than the current, we search the left child. If the number if greater than the current, we search the right child.

We return true if it exists and false otherwise.

```
public boolean contains(int x){
    Node curTree = root;
    while(curTree!=null){
        if(x==curTree.value){
            return true;
        }else if(x<curTree.value){
            curTree = curTree.left;
        }else{
            curTree = curTree.right;
        }
    }
    return false;
}
```

63.1.4 Remove

Removing an element is a much more complex because we need to maintain the tree structure of the tree set when removing elements. First we locate the element that we want to remove. If the element is not there then the operation failed and we return false. If the element is there then are three cases we need to consider.

Case 1: Node is a leaf node

If the node we want to remove is the leaf node, we can simply remove it.

Case 2: Node has one child

If the node we want to remove has a child, we can replace that node with its' only child.

Case 3: Node has two children

We need to replace the node with the rightmost of the left subtree or the leftmost of the right subtree to maintain the order.

It does not matter which side we pick so we will use the rightmost of the left subtree. First we copy the value of the rightmost of the left subtree into the node that will be deleted.

Then we replace the rightmost of the left subtree with its left subtree.

```
public boolean remove(int x){
    //Get node to remove
    Node curNode = root;
```

```
while(curNode!=null){
    if(x==curNode.value){
        break;
    }else if(x<curNode.value){
        curNode = curNode.left;
    }else{
        curNode = curNode.right;
    }
}
if(curNode==null){
    return false;
}
//Case 1: leaf node
if(curNode.left==null \&\& curNode.right==null){
    //if root
    if(curNode==root){
        this.root = null;
    }else{
        curNode.parent.replaceChild(curNode,null);
    }
}
//Case 2: one child
else if(curNode.left==null){
    //If root
    if(curNode==root){
        root = curNode.right;
        root.parent = null;
    }else {
        curNode.parent.replaceChild(curNode,curNode.right);
    }
}
else if(curNode.right==null){
    //If root
    if(curNode==root){
        root = curNode.left;
        root.parent = null;
    }else {
        curNode.parent.replaceChild(curNode,curNode.left);
    }
}
```

```
    //Case 3: two children
    else {
        //Get rightmost of left subtree
        Node rightmost = curNode.left;
        while(rightmost.right!=null){
            rightmost = rightmost.right;
        }
        curNode.value = rightmost.value;
        rightmost.parent.replaceChild(rightmost, rightmost.left);
    }
    size--;
    return true;
}
```

63.1.5 Print Tree

Since tree sets are stored as a binary search tree, we can print the elements in order.

```
public String dfs(Node curTree){
    if(curTree == null)return "";
    String ret = "";
    ret += dfs(curTree.left);
    ret += curTree.value;
    ret += ",";
    ret += dfs(curTree.right);
    return ret;
}

public String toString(){
    String ret = "";
    if(root!=null){
        ret += dfs(root);
    }
    return ret.substring(0,ret.length()-1);
}
```

63.2 Exercises

Write a function to determine if a binary tree is a binary search tree

Chapter 64

hash_set

Prerequisites: Sets

Source on Github

Hash sets are sets that use hashes to store elements. A hashing algorithm is an algorithm that takes an element and converts it to a smaller chunk called a hash. For example let our hashing algorithm be (x mod 10). So the hashes of 232, 217 and 19 are 2,7, and 9 respectively.

For every element in a hash set, the hash is computed and elements with the same hash are grouped together and stored in a linked list. The linked list is called a bucket.

If we want to check if an element already exists within the set, we first compute the hash of the element and then search through the linked list associated with the hash to see if the element is contained.

Operation Membership Insertion Deletion

Time Complexity O(1) O(1) O(1)

64.0.1 Prerequisites

Sets Linked List

64.1 Implementation

Let use the example of the hashset of the elements of 3242, 3523, 123, 235 and 538. The hash set looks like this when computed:

If we wanted to check if 7238 was in the hash set, we would get the hash (7238 mod 10 = 8). So we get the bucket associated with the hash 8 and we get the list of (538). When we iterate through this short list, we see that 7238 is not a member of the set.

Similarly, if we wanted to insert 7238 into the hash set, we would check if it exists and if it did not we would append the element to the end of the bucket. For deletion we would find 7238 check if it existed in the set and remove it from the bucket.

Hash sets are very efficient in all three set operations if a good hashing algorithm is used. When the objects are that being stored are large then hash sets are effective as a set.

64.1.1 Class

Inside our implementation of a hash set we will store the buckets using an array of linked lists, the number of buckets, and the number of elements in the set.

The collision chance is the threshold for resizing the hash set. When the ratio of elements in the set to number of buckets is greater than the threshold, then the chance of collision will be high enough that it will slow down the operations. The lower this ratio, the better performing a hash set will be.

```
public class HashSet {

    public LinkedList<Integer>[] buckets;
    public int bucketsSize = 10;
    public int size = 0;
    public static final double COLLISION\_CHANCE = 0.3;

    public HashSet(){
        buckets = new LinkedList[10];
        for(int i=0;i<bucketsSize;i++){
            buckets[i] = new LinkedList<Integer>();
        }
        size = 0;
    }
```

64.1.2 Hash code

The hash code is the result of the hashing algorithm for an element. In our hash set implementation, we will use a simple hash: modulus of the integer by the number of buckets.

For the most part if the numbers are all random then the hash function is fine. However, if the number of buckets was 10 and we added the elements 20,30,40,50,60,70, they will all end up in the same bucket and results in poor performance.

```
public int getHash(int x,int hashSize){
    return x % hashSize;
}
```

64.1.3 Resize

A hash set must be able to resize. When the ratio of number of elements to number of buckets, the chance of collision will increase more and more. So we must able to resize the number of buckets to support the number of elements to lower the chance of collision.

To resize efficiently, we can create two times the number of buckets and set them to empty and then insert all the elements in the old buckets to the new buckets.

```
public void resize(){
    int newBucketsSize = bucketsSize*2;
    LinkedList<Integer>[] newBuckets = new LinkedList[newBucketsSize];
    for(int i=0;i<newBucketsSize;i++){
        newBuckets[i] = new LinkedList<Integer>();
    }
    for(int i=0;i<bucketsSize;i++){
        for(Integer y:buckets[i]){
            int hash = getHash(y,newBucketsSize);
            newBuckets[hash].push(y);
        }
    }
    buckets = newBuckets;
    bucketsSize = newBucketsSize;
}
```

64.1.4 Insert

To insert an element in a hash set, we get the hash code from our hashing algorithm and insert the element into the corresponding bucket.

The function will return method or not the operation was successful. If the bucket already contains the element the operation will stop because we do not want to add duplicate elements into the set. If the bucket does not contain the element, we will insert it into the bucket and the operation is successful.

```
public boolean insert(int x){
    int hash = getHash(x, bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.contains(x)){
        return false;
    }
    curBucket.push(x);
    if( (float)size/bucketsSize>COLLISION\_CHANCE){
        resize();
    }
    size++;
    return true;
}
```

64.1.5 Contains

To check if a hash set contains an element, we get the hash code from our hashing algorithm and check if the corresponding bucket contains the element.

```
public boolean contains(int x){
    int hash = getHash(x, bucketsSize);
    LinkedList<Integer> curBucket = buckets[hash];
    return curBucket.contains(x);
}
```

64.1.6 Remove

To remove an element from a hash set, we get the hash code from our hashing algorithm and remove the element from the corresponding bucket.

The function will return whether or not the operation was successful. If the bucket contains the element we can remove it from the linked list and the operation is successful. If the element is not in the bucket then the operation fails because we cannot remove something that is not there.

```
public boolean remove(int x){
```

```
    int hash = getHash(x, bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.remove((Integer)x)){
        return true;
    }
    return false;
}
```

64.2 Exercises

Try to come up with a better hashing algorithm Calculate the probability of a collision occurring given the number of buckets and number of elements in the hash set Given an array of numbers, find the number of pairs of numbers that sum to 0. Given an array of numbers and a number A, find the number of pairs of numbers that sum to A. Given an array of numbers and a number A, find the number of quadruples that sum to A.

Chapter 65

Line_Sweeping

65.1 Intersection of line segments

65.1.1 1D intersection of line segments

Given a list of line segments in 1D, find the area of overlap. For example: (1,5), (6, 7), (2, 6). The area of overlap is 3 since (1,5) overlaps with (2,6) at (2,5) so the area is 3.

Every line segment has a starting point and ending point. Let the starting point will be the smaller coordinate and the ending point be the larger coordinate. We can then put all these points in a priority queue

When we reach a starting point we will increase a counter and when we reach an ending point we will decrease the counter. When the counter is ¿ 0 then we can add the distance between points to the total length.

65.1.2 2D intersection of line segments

65.2 Intersection of rectangles

65.3 Exercises

Given a list of points, find the minimum distance between a pair of points.

Chapter 66

Minimum_Spanning_Tree

Prerequisites: Graph Theory

A spanning tree of a graph is a tree that spans all the nodes of the graph but only using some of the edges to connect all the nodes.

A minimum spanning tree is the spanning tree that requires the minimum of some property such as total weight or total edges.

Spanning tree algorithms are essential in networking to ensure no loops occur when sending data through a network.

66.1 Implementations

Algorithm Desc Time Space

Prim's Using greedy method O(n log n) O(n^2)

Kruskal Using connected components O(n log n) O(n^2)

66.2 Exercises

Given a weighted graph with n nodes, find the smallest total cost to connect all nodes into 3 separate groups. (A single node can be a group) Same as 3, but a group must contain at least 3 other nodes.

Chapter 67

Shortest_Path

The shortest path is defined as a path from one node to another while trying to minimize a certain property (least number of nodes, smallest total weight). However, shortest paths may have negative weights leads to cycles.

67.0.1 Prerequisites

Graph Theory

67.1 Algorithms

Algorithm Desc Time Space Detect cycles?

Floyd Warshall Computes shortest path between all pairs of nodes O(n^3) O(n^2) Yes

Bellman Ford Computes shortest path between a pairs of nodes O(n^2) O(n) Yes

Dijkstra's Computes shortest path between a pair of nodes using the Greedy method O(n log n) O(n log n) No

Chapter 68

Greedy_Algorithm

A greedy algorithm is an algorithm that always selects the optimal next step towards the solution based on some property. Greedy problems are generally straightforward and easy to solve once you determine what property to maximize. However the most difficult part is proving that greedy is the optimal solution for all cases. It is much easier to provide a counter example to a faulty solution than to prove that a solution works for all cases or even to determine if a greedy solution exists.

68.1 Coin Problem

A cashier is given a \$20 bill from their customer whose total comes to \$14.67. How can the cashier give the change in the least amount of coins? Let's say our currency has denominations 1 cent, 5 cent, 10 cent, 25 cents, \$1 and \$2.

We can greedily use the biggest denomination as possible until we have no more coins. We need to make change for \$5.33.

\$5.33 - 2=\$2
= \$1.33 - 1=\$1
= \$0.33 - 1=\$0.25
= \$0.08 - 1=\$0.05
= \$0.03 - 3=\$0.01
= 0

The coin problem can only be solved greedily if the denominations fit nicely inside each other. Otherwise, the solution becomes more complicated. (See Dynamic Programming)

```
int minCoins(int value){
    int denoms[] = {1,5,10,25,100,200};
```

```
    int totalCoins = 0;
    for(int i = denoms.size; i >= 0; i--){
        int coins = value / denoms[i];
        totalCoins += coins;
        value -= denoms[i] * coins;
    }
    return totalCoins;
}
```

68.2 Interval Scheduling

Let's say you are at a festival and there are several special events that you want to attend, however the events may overlap at certain times. How can you pick the events that you go to such that you attend as many events as possible?

For example, we have the events below shown as a intervals on a timeline.

The first approach may seem to be greedily choosing the shortest intervals that do not overlap:

Another approach may be taking the earliest available event:

The above approaches seem to work, but they do not offer the optimal solution. We notice that we want to fit as many events as possible so we want to pick events that end as quickly as possible. So we can greedily select events with the earliest ending time.

Note that although this approach does not give us the longest time spent at events, it gives us the most events that can be visited.

```
class Interval implements Comparable<Interval>{
    int start, end;
    public int compareTo(interval i){
        return end - i.end;
    }
}

int maxEvents(ArrayList<Interval> intervals){
    Collections.sort(intervals);
    int curTime = 0;
    int events = 0;
    for(int i=0;i<intervals.size();i++){
        while(i < intervals.size() \&\& intervals[i].start < curTime){
            i++;
```

```
        }
        curTime = intervals[i].end;
        events++;
    }
    return events;
}
```

68.3 Couple Matching Problem

You are organizing a dance with a group of 10 men of varying heights and a group of 10 women of varying heights. The dance is done in pairs (a man and a women) and it is best performed when the pair is as close height as possible.

We can minimize the average height difference by sorting men and woman by height and then matching them.

68.4 Gas Tank Problem

You are driving from Boston to New York (350km) and there gas stations along the way that sell gas at different prices. Your gas tank has a capacity of 80L and you use 1L/km. How do you arrive at New York in the cheapest way possible? Assume that we can reach the next stop with an almost empty tank and there's enough gas stations spaced out along the way to make the trip.

Let's change the problem such that we are not constrained by real life limitations. Instead of paying for the gas as soon as its in our tank, we will pay for the gas as it is used. Also, instead of the gas mixing in the tank, we will be able to separate the gas by their price.

We can solve the problem using a greedy approach. As we travel from one stop to the next, we only use the cheapest fuel and pay for it as we go. Every time a gas station is reached, we "dump" out all the gas that is more expensive and then we will the gas tank to the top. We can do this because we only pay for gas when we use it so we do not lose anything.

This approach works because when we "dump" the gas, it is equivalent to not buying the gas in the first place. We fill the tank with the most efficient gas and dump the rest that we don't need so it is the optimal way to solve the problem.

We start with a full tank of 80L ($0) at 0km.

We drive 70km to the next gas station and we use 70L so we use 70L of $0 gas. We refill the tank at the pump and get 10L ($0) and 70L ($10).

We drive 20km more to the next gas station using 20L. We use the cheapest gas which is 10L ($0) and 10L ($10). So far, the total cost of the trip is $10 from the 10L ($10) we used. In our tank we have 60L ($10) but the current pump is at $5. So we can dump all the gas and fill our tank with 80L ($5).

We drive 30km more to the next gas station using 30L. We use and pay for the 30L ($5). Since the pump sells gas higher than any remaining fuel, we refill the remainder of our tank (30L) with the more expensive gas.

68.5 Exercises

Given N intervals, each with an end time and start time, how can you select intervals to give you the largest amount of time covered? Given the future schedule of a stock, and starting with C dollars, what is the maximum amount of money you can make given that there is no commission?

For example you are given $1000, the future prices of a stock are: 100, 103, 105, 106, 101, 99, 95, 97.

Buy 10 x $100 [$0] Sell 10 x $106 [$1060] Buy 11 x $95 [$15] Sell 11 x $97 [$1082]

You are on a road trip that is D km and there are N gas stations along the way and your gas tank has capacity C. Assuming that you start with a full tank, travel 1km/min, use 1L/km and the gas station fill rate is 5L/min, how do you finish in the quickest way?

For example, if the total distance is 200km apart with tank capacity 80L and with 5 gas stations at 10km, 40km, 50km, 90km, 150km, the minimum time it will take is 224 minutes.

Chapter 69

Longest_Common_Subsequence

A subsequence is a subset of the original sequence that is in the same order. For example in the string "abcdefghi", "aeg" is a subsequence but "eaq" is not because it is not in order.

The longest common subsequence between two strings A and B is the longest subsequence in A that is also in B.

For example given A="xyaaaabcdeg", B="bcaaaaefgxy" the longest common subsequence is "aaaaeg"

```
xyaaaabcde\_g
bcaaaa\_\_\_efgxy
```

If we try to use greedy we will see it doesn't work. For example if we use try to take as much as B as we can, we see that we will get BCEG or if we try to take as much as A we get XY.

69.0.1 Prerequisites

Dynamic Programming

69.1 Solution

Let first write a formal definition of the problem, given two strings A and B each with lengths N and M respectively we want to find the longest common subsequence between them.

[Note to make reading easier I have used short forms for substring and index. For example A[3] means the 3rd character of A and A[1..4] means the substring of A from the first character to the fourth character inclusive. A[0] represents the null substring of A]

The base is simple. The LCS of A[1..x] (where x is from 1 to N) and B[0] = 0. The LCS of B[1..x] and A[0] = 0.

We need to break this problem into subproblems.

If A[N] = B[M] and then the new LCS is the LCS of A[1..N-1] and B[1..M-1]. Note that if two strings have the same character at the end of their string it has to be part of the LCS. Try to prove this to yourself.

```
LCS of matched = (LCS of A from 1 to N-1, B from 1 to M-1) + A[N]
```

If A[N] != B[M] then we try to match A[N] with B[M-1] or A[N-1] with B[M]. Thus we take the LCS of A[1..N-1] and B[1..M] and the LCS of A[1..N] and B[1..M-1].

```
LCS of not matched = max ( (LCS of A[1..N], B[1..M-1]) , (LCS of A[1..N
```

Putting it all together we have:

```
Let LCS[i][j] be the length of longest common subsequence of A[1..i] an

Base case:
LCS[i][0] = 0 where 0< i     n
LCS[0][j] = 0 where 0< j     m

Subproblem:
if A[i] == B[j], LCS[i][j] = LCS[i-1][j-1]+1 where 0< i     n and 0 < j
if A[i] != B[j], LCS[i][j] = max( LCS[i-1][j], LCS[i][j-1] ) where 0<
```

69.2 Implementation

69.3 Exercises

Given three strings of length N, find the longest common subsequence Given two strings of length N, find the longest common substring (adjacent characters) Given a string S, find the least number of operations (add letter, remove letter) to turn it into a palindrome.

Example: abcdc -¿ abcdcb -¿ abcdcba (2 operations)

Given a string A and string B, find the least number of operations (change letter, add letter, remove letter) to turn A into B.

Example: A = abcdefh, B = bcefg, abcdefh-¿bcdefh-¿bcefh-¿bcefg