Arrays

O(1) retrieving

arrays have Fixed length

```
int[] array = new int[10]
                    ↑
              fixed size of 10
```

resizable arrays can grow as needed

some languages (such as Python) only have resizable arrays

=

Use resizable when you don't know how many elements you need upfront

Underneath it is an array and acts like it, but when the array gets full we create a new array that's double the size of the old array and copy the elements over

Copying all elements is really slow, but cost is ammortized, only happens when we hit capacity so in average adding is still constant time, even though sometimes is linear

Strings

sequence of characters, almost like an array

You can access the ith element immediately like with an array

unlike arrays however, you cannot change the string (in many languages)

When we concatenate a bunch of strings:

```
str = a      "coding" + "style" + "is" + "fun"
str += b       ↑a       ↑b      ↑c     ↑d
str += c
str += d  ← really inefficient! create a new string every time
```

Create an array or arraylist of all the strings and every time you concatenate one just add it to that list, tracking all the while the total number of characters you need. When actually need the string out, go and merge all the strings together at once

```
StringBuilder sb = new StringBuilder()
sb.append(a)        ↑ most languages
sb.append(b)          should have
sb.append(c)          something like this
sb.append(d)
```

Hash tables

* For any problem, have hash tables at the top of your mind as a possible technique to solve the problem

A hash table, at a high level, is a key value lookup.

So it gives you a way of, given a key, associating a value with it. For very very quick lookups.

```
hashTable.put("Katy", new Person(...))
hashTable.get("Katy")
```

In a hashTable, the key as well as the value can be basically any type of data structure, as long as it has a hash function.

=

At a high level we want to store the objects in an array

A hash function takes a string (generally an object), converts it into some sort of integer, and then remaps that integer into an index into that array. That's where we can find our value

The hashcode is not actually the index in this array, we map from the key to the hashcode, and then over to the index

string → hashcode → index

That's because the array that actually stores the data from the hashtable, might be much much smaller than all the available potential hash codes

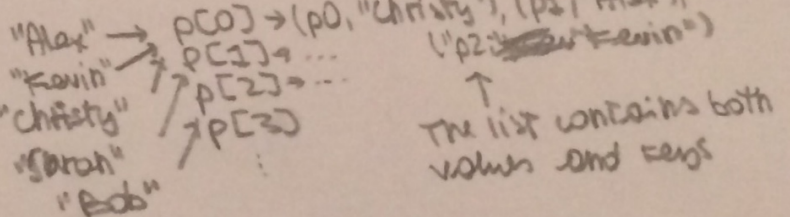
Two strings can have the same hashcode, that's because there are an infinite number of strings, but a finite number of hashcodes. Additionally since we're remapping the hashcode into an even smaller index, two things with different hashcodes could actually wind up mapped to the same index

That's called a collision!

Chaining is the most common way to solve collisions,

with chaining, when there is collisions, just store them in a linked list

When looking for the value of "Alex", you actually need to look all the values in that linked list and pull out the value for "Alex".



O(1) for a "good" hashtable

O(n) for a terrible hashtable ← what we assume

↑ lots of collision. worst case!

Linked Lists

A linked list is essentially a sequence of elements where each element links to the next element, which links to the next element and so on...

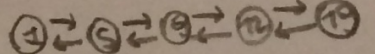
It can contain pretty much any type of data, elements can be sorted or unsorted, it can contain duplicate elements or all unique elements

In an array elements are indexed, a[4]
In a linked list you have to start with the head and work your way through, until you get to the fourth element!
↑
That takes linear time

Advantages
insertion and deletion can be quick!

O(1) prepend
O(n) append

DOUBLY LINKED LIST



Each element also links to the previous element

Stacks and Queues!

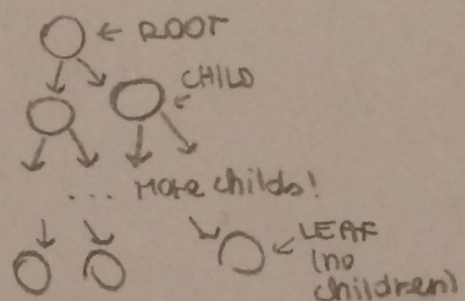
- linear data structures
- Flexible size

The main difference is in how elements are removed:

A stack is LIFO

A queue is FIFO

Trees

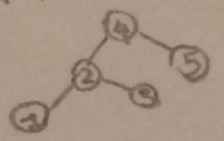


Binary Tree
Each node has no more than two child nodes

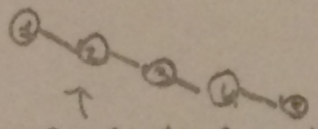
Binary search tree
A binary search tree is a binary tree where on any subtree the left nodes are less than the root node, which is less than all of the right nodes

This makes finding a node O(lg n)

Our tree could get really unbalanced!



An unbalanced tree has slower inserts and finds



Depending on the order of insertion

Traversing

Inorder
left then root then right

Preorder
root then left then right

Postorder
left then right then root

Inorder gives us the elements in sorted order in a binary search tree!

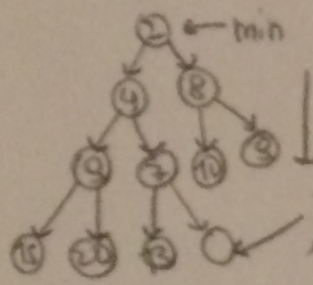
Insert

start with some element you want to insert. Check whether is bigger or smaller than the the root, if bigger go to the right, else to the left, Do this over and over until you get to an empty spot and insert the element.

MIN or MAX

Heaps

In a min heap, the elements are always smaller than their children

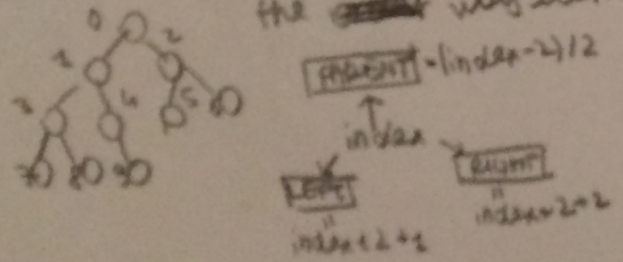


inserting an element, it always goes in the next empty spot looking top to bottom left to right.

.. And then bubble it up until we get to the right spot.
- Compare with parent ...
- if out of order, swap them
- keep going up the tree

Removing the minimum element,
- Remove the min element
- Swap empty root with the last element added
- Bubble down root
  - compare with left and right child
  - swap it with the smaller
  - keep going down until the heap property is restored

An heap is visualized as a binary tree, but we can actually use an array to store the values since the heap never has gaps (because of the way elements are added)

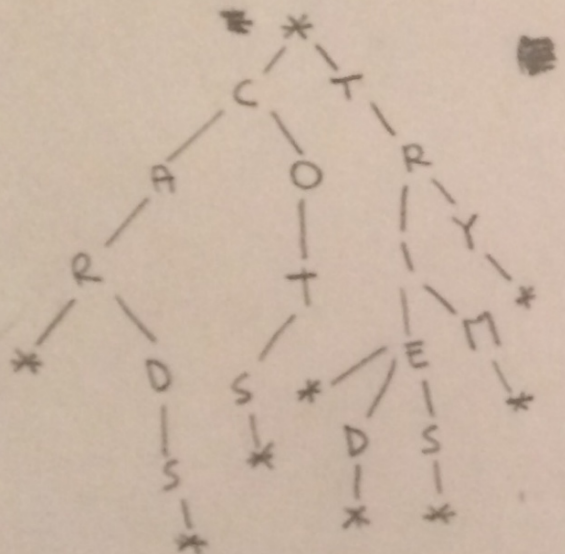


Tries

- Data structure which is actually a type of tree
- Often used to store characters

Each nodes stores a character and the path from the root to that node is a word or part of a word

Very quick lookups of a particular kind



Words starting with "car-"
car card cards!

Use when you need some sort of word validation

- Walk through scrabble board and find all the words
- Given list of strings, find all celebrity names

Instead of looking up from the root, every time when checking a string, we want these calls to build on each other
We can keep states in various ways

By:

- Keeping state within trie
- Return the node reference

```
class Node {
    HashMap<Character, Node> children;
    boolean isCompleteWord;
```

Bubble sort

[redacted]

Really naive algorithm, not one we would want to use in the real world

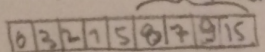
Walk through the array and everytime we see elements that are out of order, we swap them

By doing it enough times, our array will become sorted $O(n^2)$ but in place

For example, you have a sorted array, and one element gets incremented, then one walkthrough a bubblesort could then sort the array efficiently

Quicksort! very efficient

- pick some element to be the pivot element (randomly)
- walk through the array and swap elements around such that the elements less than the pivot come before elements bigger than it
- repeat the process for the left and right partition



[redacted] choose new pivot and repeat for each half

- Eventually we will wind up with a sorted array as it gets more and more sorted each loop

[redacted]

IF we pick a bad pivot everytime (very first or last element in subarray) $O(n^2)$ runtime

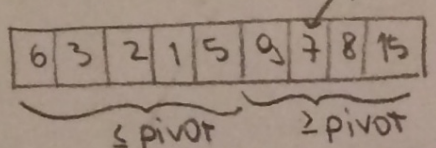
As long as we're smart about how we pick the pivot element, we get [redacted] runtime efficient [redacted]

$O(n \log n)$
each element gets quicksort called on it, $\log n$ times

QUICKSORT

Really popular and efficient sorting algorithm

- Take array of, say integers
- Pick a pivot element
  (For now we assume is picked randomly)
- Walk through the array and swap elements around such that all elements less than the pivot, come before the elements bigger than it



- Repeat the process to the left and right partition over and over and eventually our array just becomes sorted

EFFICIENCY

In an ideal world in quicksort, we're dividing the array in half each time. We pick a great pivot that really is roughly the median, and then half the elements gets pivoted to one side of the array and half of the get pivoted to the other and then apply quicksort to each half.

∴ In that case we get O(nlogn)
(each element gets quicksort called on it logn times, and each one of those was one swap. So n elements go through logn swaps)

However in the bad case, we pick a really bad pivot, like, everytime we pick the pivot happens to be the lowest element in that subarray, then we have n² calls to quicksort, therefore O(n²)

As long as we're smart about how we pick the pivot element, we get a pretty efficient runtime, and that's why we implement quicksort in the real world

MERGESORT

Pretty efficient sorting algorithm.

The best way to conceptualize merge sort is recursively

Suppose we have this large array, and we want to sort it
What if we could magically sort the left half and the right half?
Then if we wanted to make the whole array sorted, all we have to do is merge those in sorted order

What we do to sort the left and right half is just apply merge sort again. By doing it over and over we'll get a sorted array eventually.

```
mergesort(array) {
    mergesort(array's left half)
    mergesort(array's right half)
    merge left and right half in sorted order
```

Always gives O(nlogn)

But merging two arrays together generally requires extra space → O(n) space

copy all the elements into a new array, merge them and copy them back

Mergesort

10 5 2 7 4 9 | 12 1 8 6 11 3

magic sort | magic sort

2 4 5 7 9 10 | 1 3 6 8 11 12

merge

1 2 3 4 5 6 7 8 9 10 11 12

mergesort([redacted] array) {
  mergesort (array's left half)
  mergesort (array's right half)
  merge left and right half in sorted order

Always gives $\Theta(n \log n)$ runtime.
Merging two arrays together generally requires extra space

Binary Search

nos to be sorted

1 3 4 5 13 20 25 40 42 43

13>4 | 13>5 | 13<20 | 13<15

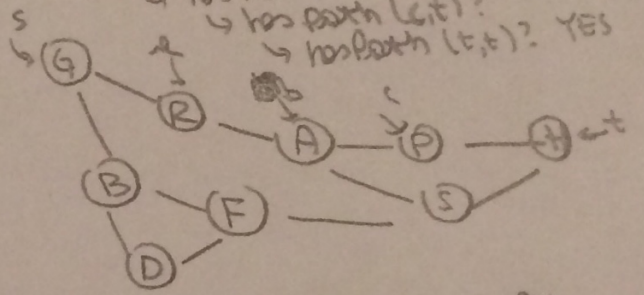
① n elements
② $\frac{n}{2}$ elements
③ $\frac{n}{4}$ elements
⋮
? 1 element

$O(\log_2 n)$ steps

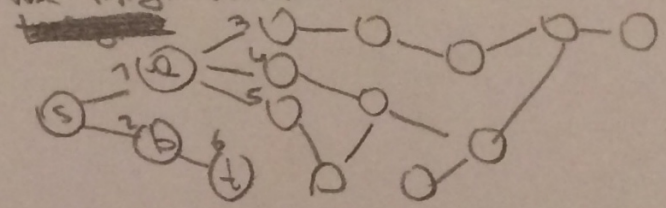
Graph [redacted] search, DFS and BFS

Depth First Search
Goes deep (to children) before going broad (to neighbours)

hasPath(s,t)?
↳ hasPath(a,t)?
↳ hasPath(b,t)?
↳ hasPath(c,t)?
↳ hasPath(t,t)? YES



We might run really far away.



[redacted] that's why we might often prefer to use BFS instead

Breadth First Search
Goes broad (to neighbours) before going deep (to children)

In both we need a flag or some way to prevent visiting nodes again

BFS, [redacted] iterative, using queue. Pull first element from the queue, check if it is this. Find element and if not go add all its children to the queue

If we want to find if there's actually a path or the shortest path, BFS is preferred

Binary search must be sorted

1 3 4 5 73 20 25 40 42 44 53

↑ 73>4 ↑ 73>5 ↑ 73<80 ↑ 73<20

comparisons

- ① n elements
- ② n/2 elements
- ③ n/4 elements
- ⋮
- (log n) 1 element

} O(log₂ n)

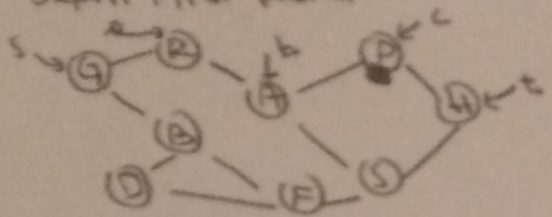
```
public static boolean binarySearchRecursive(int[] array,
                                            int x, int left, int right) {
    if (left > right) return false;
    int mid = (left + right) / 2;   // left + ((right-left)/2) for int overflow
    if (array[mid] == x) return true;
    else if (x < array[mid]) {
        return binarySearchRecursive(array, x, left, mid-1);
    } else {
        return binarySearchRecursive(array, x, mid+1, right);
    }
}

public static boolean binarySearchIterative(int[] array, int x) {
    int left = 0;
    int right = array.length - 1;
    while (left <= right) {
        int mid = left + ((right-left)/2);
        if (array[mid] == x) return true;
        else if (x < array[mid]) right = mid-1;
        else left = mid+1
    }
    return false;
}
```

can also be applied to other data like strings

Graph search: DFS and BFS

collections of nodes where nodes might point to other nodes through edges that are either directed (one way) or undirected

Depth First Search



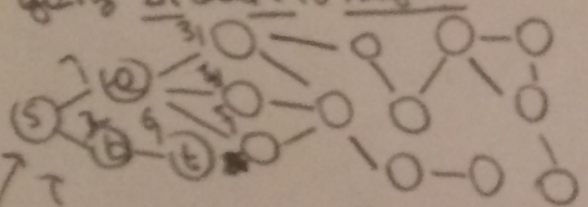
hasPath(s,t)? ↑

hasPath(a,t)? ↑

hasPath(b,t)? ↑

hasPath(c,t)? Yes ↑

goes deep (to children) before going broad (to neighbors)



DFS have gone really far away when t is actually close. That's why we often prefer to use breadth first search

Breadth First Search

goes broad (to neighbors) before going deep

DFS is implemented with a recursive algorithm. The only trick is you have to mark them to use a is_visited flag so that you don't wind up in some sort infinite loop

With BFS the main trick to remember is you want to use a queue (iterative). When seeing a node if it has a path to t, you're gonna add all of its children to the queue. Pull out the first element from the queue and check and then if it is t, if not go add all of its children to the queue

queue [c, b, c, d, e] (previous example)

Big O

Describes how runtime scales with respect to some input variables

```
boolean contains(array, x){
  for each element in array{
    if element == x {
      return true
    }
  }
}
```

O(N) where N is the size of the array

```
void printPairs(array){
  foreach x in array{
    foreach y in array{
      Print x,y
    }
  }
}
```

* will print (5,2) and (2,5) - both orders?

$O(N^2)$

Many Big O times do NOT use "N". It's just a variable! Any letter will do!

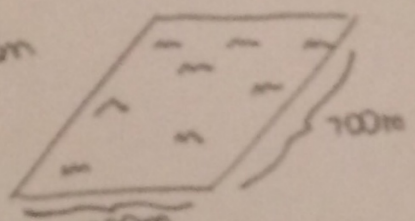
Runtime to draw a square plot of given area

option ①

$O(a)$ ~ area of given

option ②

$O(s^2)$ ~ length of one side



Both options are equivalent since $a = s^2$. Variables have meanings and it matters what they are

① Different steps get added

```
Function something(){
  doStep1(); // O(a)
  doStep2(); // O(b)
}
```

O(a+b)

② Drop constants

```
Function minMax1(array){
  min, max = null
  for each a in array
    min = MIN(a, min)
  for each a in array
    max = MAX(a, max)
}
```

O(n)

```
Function minMax2(array){
  min, max = null
  for each a in array
    min = MIN(a, min)
    max = MAX(a, max)
}
```

O(n)

$O(2n) \Rightarrow O(n)$

We're just looking for how things scale roughly

③ Different inputs ⇒ different variables

```
int intersection(arrayA, arrayB){
  int count = 0
  for a in arrayA{
    if a == b{
      count = count + 1
    }
  }
  return count
}
```

$O(N^2)$ ← what is N?

$O(a*b)$ ← length of arrayA, length of arrayB

④ Drop non-dominant terms

```
Function whyWouldYouDoThis(array){
  max = null
  for each a in array{
    max = MAX(a, max)
  }
  print max
}
```

O(n)

```
for each a in array{
  for each b in array{
    Print a,b
  }
}
```

$O(n^2)$ $O(n^2 + n) \Rightarrow O(n^2 + n^2) \Rightarrow O(2n^2)$

If n^2 and n^2 are equivalent (see Rule 2), then either is fine

$O(n + n^2) \Rightarrow O(n^2)$

Recursion

↓ joke.

to understand recursion, one must first understand recursion

Say you wanna know how many .txt files are on a computer

You can go to the root directory and count how many are there.

But then it has folders, and each of these folders has subfolders and so on and so on.

You need to go out and see all of them, and find all the .txt files across the whole computer

Recursive approach:

Count files in root directory and then add up how many are in each folder and we go and ask them in the very same process, and they then know to go out and ask their subfolders and so on

```
int countTxt(Directory dir){
    if(dir == null) return 0;
    int count = 0;
    for (File file: dir){
        if (file.endsWith(".txt")) count++;
    }
    for(Directory subdir: dir){
        count += countTxt(subdir);
    }
    return count;
}
```

Every recursion must have what's called a base case, or like a stopping point.

In this case is when we have a folder with no subfolders, then we'll return once we count how many files it has.

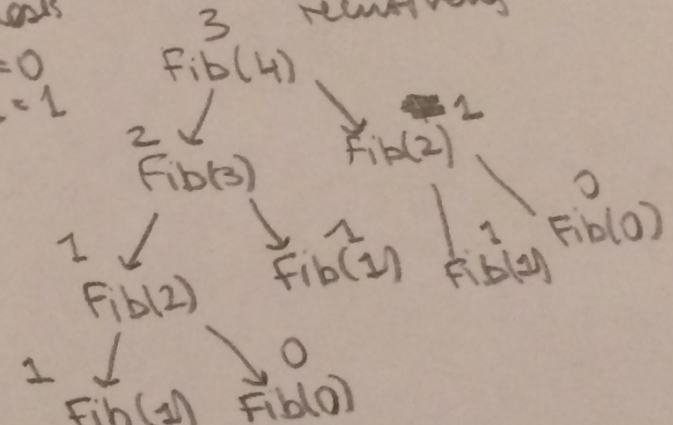
Fibonacci sequence is one of the simplest examples of recursion

$F_n = F_{n-1} + F_{n-2}$ ← we often say it is defined recursively

↑ base cases

$F_0 = 0$

$F_1 = 1$



We're repeating work, this can get very very inefficient

We can optimize it with memoization

Anything that is implemented recursively can be implemented iteratively

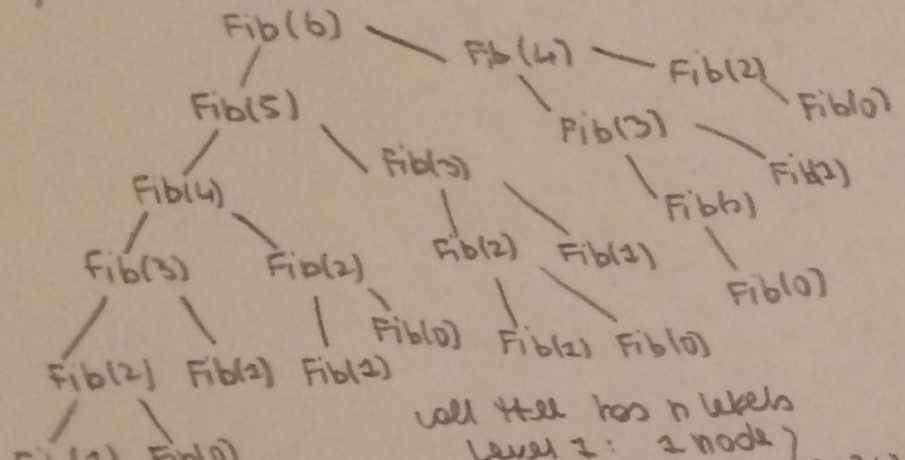
For certain problems however, the recursive implementation is a lot easier to read and understand and write

INTERVIEW: Discuss + decide whether to implement a solution iteratively or recursively

Memoization + Dynamic Programming

Fib(n) = Fib(n-1) + Fib(n-2)

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



call tree has n levels

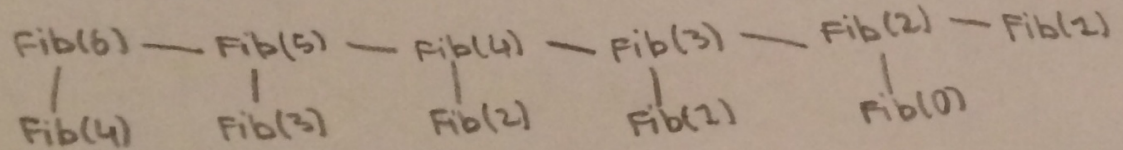
Level 1: 1 node
Level 2: 2
Level 3: 4
Level 4: 8

1+2+2+...+2
$O(2^n)$

We keep doing things repeatedly!

```
int Fib(int n, int[] memo) {
  if (n<=0) return 0;
  else if (n==1) return 1;
  else if (memo[n]) memo[n]=Fib(n-1)+Fib(n-2);
  return memo[n];
}
```

MEMOIZATION → O(n) time



Space complexity (or memory usage)

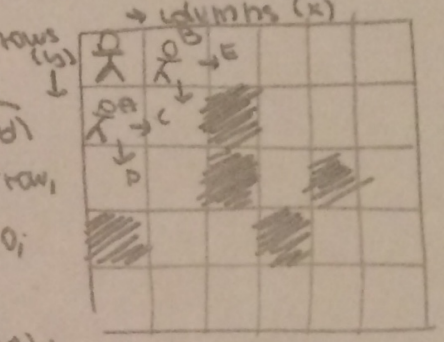
Non-memoized: O(n) "The biggest your call stack ever gets"

Memoized: O(n) O(n) for call stack (like previous) and O(n) for the memo table

pathsFrom(start, end) =

paths(A, end) + paths(B, end)

paths(D, end) + paths(C, end) | paths(C, end) + paths(E, end)




```
int countPaths(boolean[][] grid, int row,
               int col) {
  if (!validSquare(grid,row,col)) return 0;
  if (isAtEnd(grid,row,col)) return 1;
  return countPaths(grid,row+1,col)
       + countPaths(grid,row,col+1);
}
```

TIP Use row and column, not x and y
grid[row][column] is equivalent to
grid[y][x], NOT grid[x][y]

```
int countPaths(boolean[][]grid,int row,int col,int[][]paths){
  if (!validSquare(grid,row,col)) return 0;
  if (isAtEnd(grid,row,col)) return 1;
  if (paths[row][col]==0){
    paths[row][col]=countPaths(grid,row+1,col)+
                    countPaths(grid,row,col+1);
  }
  return paths[row][col];
}
```

Recursive $O(2^{n^2})$ runtime
Memoization $O(N^2)$ runtime

Bit manipulation

base 10 $\overset{1}{5}\ \overset{1}{8}\ \overset{}{3}$ $\}\ 5*10^2+8*10^1+3*10^0$

$+\ 6\ 3\ 7$

$1\ 2\ 2\ 0$

base 2 $\overset{1}{1}\ \overset{1}{0}\ \overset{}{1}$ $\}$ [redacted] $1*2^2+0*2^1+1*2^0$

$+\ 0\ 1\ 1$

$1\ 0\ 0\ 0$

Two's complement

Imagine we're on the land of 8-bit integers

base 10	base 2 (using two's complement)
78	0 1 0 0 1 1 1 0
-78	1 0 1 1 0 0 1 0

sign bit
0 ⇒ positive
1 ⇒ negative

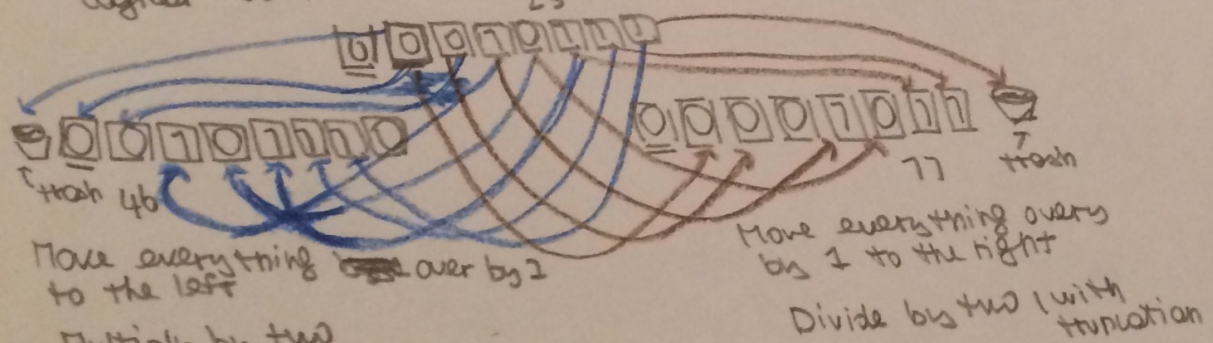
0 0 1 0 0 1 1 0
+ 1 1 0 1 1 0 1 0
1 0 0 0 0 0 0 0 0

the remaining bits are filled in with the two's complement (numbers that sum to zero to obtain the positive version of the negative)

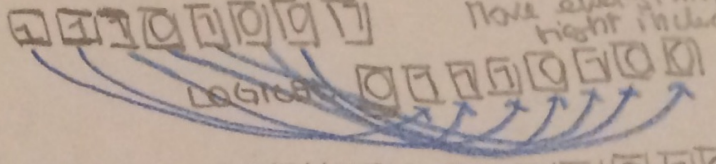
remember the +1

Shifting

logical vs Arithmetic

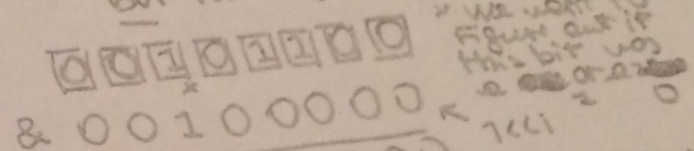


-23



Moves everything over to the right including the sign bit but then it fills in the sign bit with the original sign bit

The result does still bear that sort of, divide by two relationship but rounding to -12 this time



we want to figure out if this bit was a 1 or a 0

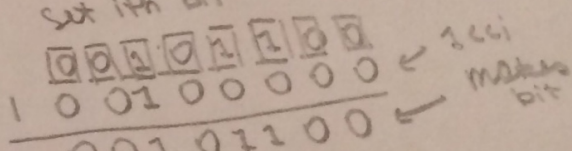
that indicates whether that bit is a 1 or a 0

(x & (1<<i)) != 0

Masks

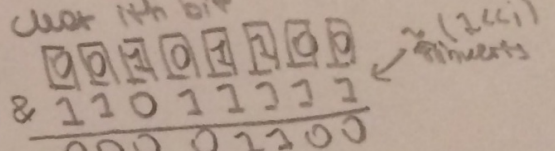
0 & 0 ⇒ 0
0 & 1 ⇒ 0
1 & 1 ⇒ 1
0 | 0 ⇒ 0
0 | 1 ⇒ 1
1 | 1 ⇒ 1
0 ^ 0 ⇒ 0
0 ^ 1 ⇒ 1
1 ^ 1 ⇒ 0

Set ith bit



makes sure that ith bit is set

Clear ith bit



inverts

Interview evaluations

As an interviewer, I'm

[redacted]

- [redacted] looking at everything you do in the interview and taking a step back and saying "What does this tell me about you as a software engineer"
- Suppose you write some code, I'm looking at it as a sample [redacted] and trying to extrapolate information about that code sample, figure out what do I think about you as a software engineer
- Some mistakes, compilation errors, are a big deal. Others aren't bugs
  Say you're using the built in linked list class and you call [redacted] list.insert instead of list.add. That doesn't say you're a bad software engineer, just that you don't use that the built-in linked list probably class every day

Another issue is mixing up single equals and double equals. pitfalls

- problem solving, I'm not timing, 20 minutes, thumbs up if you solve it, note, thumbs down", it's all about what is it telling me

It's pretty very common I can interview two candidates, one takes 10 minutes, the other 45 minutes.

The candidate that took 45 minutes might've actually done much better because the reason it took long is because we've had this big discussion about hashtables and collisions, this leads deep discussion that I didn't have with the other one, that maybe was struggling so much that I basically spoon-feed him the algorithm. The other did the problem pretty much independently, demonstrated much better skills overall

- How long does it take to solve the problem
- How optimal is the answer
- Is the answer even correct

Things to take into account

Communicate your thought process as much as possible!

- Sit in a room alone and get comfortable talking out loud and saying your thought process
- Whenever you notice yourself being quiet, try to take a step back and at least give the headline of your thought process

"Im thinking about how to optimize this piece"

"I feel there's some way of using the fact the data is sorted"

Try to give what you can. It is much better than to say nothing at all as it gives some idea

[redacted]

[redacted]

ooo

I might think you're stuck even though you really not so I might jump in to give you this hint that you don't really need

Focus on your communication, think of this as a collaborative exercise and not you just submitting homework at the end of the interview

Behavioural questions

Preparation: prepare a quick walkthrough through your resume for something like "tell me about yourself" or "walk me through your background" or something like that. Stop talk from the beginning of your career and walk through chronologically from there.

Go fairly quickly, this is not the time to spend [redacted] of details. It [redacted] boring for [redacted] and the interviewer [redacted] stops paying attention

1 Think about quick ways that you can show that you've been successful

...that can be [redacted]
- a feature you built
- a promotion
- an award

concrete things and try to slip them in there really quickly

2 Think about what you want the interviewer to ask you questions on [redacted] and drive him on that direction

"I spend last four years at company B, been focusing a lot on rebuilding our caching infrastructure in-house"

[redacted] it's natural for the interviewer to ask follow-up questions [redacted] about that project

3 Hobbies like [redacted] activities, competitions, projects on the side - anything technical should be mentioned.

For non-technical hobbies, think about if there's a way of framing them in a way that makes them relevant

How do they show teamwork, a drive to learn, willingness to take risk and anything like that

Then sort through your past background and think of a couple of projects you're gonna be particularly well prepared to talk about. Pick these out based on what you were doing more so than the rest of your team.

Think of the [redacted] technical and non technical questions they might ask

① - What were the biggest challenges
- What was the architecture
- What were the technologies
- What were the implications of picking those things

and...

② - What was the teamwork dynamic like
- How did you show influence or leadership

Make sure that you're focusing on your role [redacted] and your contributions

Don't always say "we" "us" "the team" and also [redacted] think about what you do differently

Solving Algorithms

7 steps to solve algorith problems:

① Listen carefully to the problem
Generally speaking any detail in a question is necessary to solve that problem.
- either to solve it at all
- ... or to solve it optimally

IF there's some detail you haven't used yet, think about how you can put that to use so you might need that to solve the problem optimally

Q Given two arrays (sorted + distinct), Find # elements in common

② Come up with a good example.
Most people's examples (for Q):
A: 1, 5, 15, 20 B: 2, 5, 13, 30
Not very useful as this example is so small and kind of a special case
Better example
A: 1, 5, 15, 20, 30, 37 B: 2, 5, 23, 30, 32, 35, 37, 42
Larger and you've avoided special cases

③ Come up with a brute force algorithm
IF the first thing you have is something really slow and terrible, that's okay. It's so much better to start off with something slow than to start off with nothing at all

DON'T CODE IT
- state brute force algorithm
- state runtime
- immediately go to optimizing

A good chunk of the time on algorithm interview questions will often be spent on optimizations

④ Optimize ↑

⑤ Take a step back and make sure you know exactly what you're going to do in your code.
So many people code prematurely when (you!) they haven't really really comfortable with what they're about to do, it ends in disaster. 80% understanding of what you're about to write is really not enough

⑥ Start coding.
whiteboard:
- write straight lines
- use space wisely

whiteboard or computer, braces → nothing / conventions
- coding style matters (stick to a style, no matter which) descriptive variable names

- MODULARIZE up front any little conceptual chunks of code, push that off to another function

⑦ Test your code. One mistake a lot of people do here is that they take their big example from step ② and throw that as a test case.
The problem with that is that it's really large so it will take you a long time to run through and you already used it to develop your code so if there's an oversight there, the problem will probably repeat itself here. ANALYSE
Better process: walk through your code line by line and think about each line up front is it really doing the right thing. Double check things that look tricky.
Use test cases - start with small test cases rather than big ones. They work as effectively but they're so much faster to run through. Because they are faster people tend to be more thorough and they're more likely to find bugs
- go to edge cases after that
- throw in some big test cases if you have time

Make sure that when you're testing you're really thinking about what you're doing
- really testing your code, not your algorithm
- think before you fix bugs. Don't panic!