University of Central Florida
COP 3502: Computer Science 1
Study Union Review

Author: Mujahid Jaffer
Spring 2020

Table of Contents

Recursion

What is Recursion and why do we use it?

Recursion is when you have a function that calls itself. Every recursive function must have 2 things: A Base Case, and a Recursive Call. People use recursion only when it is very complex to write iterative code. For example, tree traversal techniques like preorder, postorder, and inorder can be made both iteratively and recursively. But usually, we use recursive functions because of its simplicity once you get used to writing them. It is important that when you make the recursive call, you change some of the parameters. Otherwise, you'll end up in an infinite loop. *Onion Example*

Example of Sum Function and Tracing

Big O Run-Time Calculation

What is Big O Run-Time Calculation and why do we use it?

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm. Many companies will use this to test how long something will take to occur on their website/app if they have a lot of activity on the network to ensure their website/app won't crash. To find the Big O Runtime of a function, identify the line that executes the most in the function and see how it changes relative to its input.

Time Complexities

O(1): No matter what our input is, the function will always take the same amount of time to run.
O(log n): Our input is being halved after every iteration.
O(n): O(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set
O(nlogn): This usually happens when there is a call made to a log n function that occurs n times
O(n^2): O(n^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This usually happens when there is a nested for loop
O(n^3): O(n^3) represents an algorithm whose performance is directly proportional to the cube of the size of the input data set. This usually happens when there is are 2 for loops nested within a for loop.
O(2^n): O(2^n) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2N) function is exponential

Practice Problems

Big O.c

```c
// Big O Runtimes
#include <stdio.h>
#include <math.h>

int function1 (int n)
{
  return 5;
}
Runtime:



int function2(int n) {
  int i, counter = 0;

  for(i = 0; i < n; i++)
    counter++;

  return counter;
}
Runtime:



int function3(int n) {
  int i, counter = 0;

  for(i = 0; i < 100; i++)
    counter++;

  return counter;
}
Runtime:

```

Big O.c

```c

int function4(int n) {
  int i, counter = 0;

  for(i = 0; i < 5*n; i++)
    counter++;

  return counter;
}
Runtime:



int function5 (int n)
{
  int i, counter;
  for (i = 0; i < n*n; i++)
    counter++;

  return 5;
}
Runtime:



int function6 (int n)
{
  int i, counter;
  for (i = 0; i < n; i++)
    counter++;

  for (i = 0; i < n; i++)
    counter++;

  return counter;
}
Runtime:


```

Big O.c

```
int function7 (int n)
{
  int i, counter;
  for (i = 0; i < n; i++)
    counter++;

  for (i = 0; i < n*n; i++)
    counter++;

  return counter;
}
Runtime:



int function8 (int n)
{
  int i, j, counter;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      counter++;

  return counter;
}
Runtime:



int function9 (int n)
{
  int i, j, counter;
  for (i = 0; i < 50; i++)
    for (j = 0; j < n; j++)
      counter++;

  return counter;
}
Runtime:
```

Big O.c

```

int function10 (int n)
{
  int i, j, k, counter;
  for (i = 0; i < n; i++)
    for (j = 0; j < n*n; j++)
      for (k = 0; k < math.pow(2, n))
        counter++;

  return counter;
}
Runtime:



int function11 (int n)
{
  int i = n, x = 0;

  while (i > 0)
  {
    i = i / 2;
    x++;
  }
  return n;
}
Runtime:



int function12 (int n)
{
  int i, counter;
  for (i = 0; i < n; i++)
    counter += function11(n);

  return counter;
}
Runtime:
```

Algorithm Analysis

Main Things to Know

Most Algorithm analysis questions can be solved within 3 steps with the following formula: **T(n) = c * O(n)**, where T(n) represents the time it takes for a function to run given input n, c is a constant, and O(n) is whatever runtime you're given.

1. Plug in what you know to the formula T(n) = c * O(n)
2. Solve for c
3. Use the value found for c and plug it in to the formula again for the second function with what you know and solve for the unknown.

Practice Problems

Fall 2019 **Algorithms and Analysis Tools Exam, Part A**

2) (5 pts) ANL (Algorithm Analysis)

An algorithm to process input data about n cities takes O(n!) time. For $n = 10$, the algorithm runs in 10 milliseconds. How many ***seconds*** should the algorithm take to run for an input size of $n = 12$?

Summer 2019 **Algorithms and Analysis Tools Exam, Part A**

2) (5 pts) ANL (Algorithm Analysis)

An algorithm to process a query on an array of size n takes $O(\sqrt{n})$ time. For $n = 10^6$, the algorithm runs in 125 milliseconds. How many ***seconds*** should the algorithm take to run for an input size of n = 64,000,000?

Summations

5 Rules of Summations

1) Summation Split

ex. $\sum_{i=3}^{6}(7i+4) = \sum_{i=3}^{6} 7i + \sum_{i=3}^{6} 4$

2) Pulling out a constant

ex. $\sum_{i=2}^{9} 8i = 8\sum_{i=2}^{9} i$

3) Summation of 1's

ex. $\sum_{i=a}^{b} 1 = b-a+1$

ex2. $\sum_{i=7}^{30} 1 = 30-7+1 = \boxed{24}$

4) Summation of i that starts at 0 or 1

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

5) Summation of i that doesn't start at 0 or 1

$\sum_{i=a}^{b} i = \sum_{i=1}^{b} i - \sum_{i=1}^{a-1} i$

Practice Problem

$$\sum_{i=6}^{25} 4i+9$$

Recurrence Relations

Why do we use Recurrence Relations and what are the steps to solve them?

Recurrence Relations are used as a methodical way of finding the Big O Runtime of any given recursive function. There is a 7-step method of solving Recurrence Relations (Mujahid's Unofficial Steps).

1. If you're given code, create the recurrence relation by looking at the base cases and the return statement. If you are already given the recurrence relation, skip this step and go to step 2!
2. Do iterative substitution until you find a pattern, which normally takes around 3 iterations
3. Derive a generalized form using the variable 'k' which represents the iteration you are on.
    a. If there's a summation, you can usually solve it here
4. Since you know that your recursive calls are going to hit your base case eventually, set your base case equal to whatever is inside your recursive call in the generalized form
5. Replace all the k's in your generalized form with n's
6. Derive the closed form, which shouldn't have any k's
7. Find the highest order of n to find the Big O Runtime.

Practice Problem

Spring 2018 **Algorithms and Analysis Tools Exam, Part A**

3) (10 pts) ANL (Summations and Recurrence Relations)

Using the iteration technique, find a tight Big-Oh bound for the recurrence relation defined below:

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2, \text{ for } n > 1$$
$$T(1) = 1$$

Hint: You may use the fact that $\sum_{i=0}^{\infty}(\frac{3}{4})^i = 4$ and that $3^{log_2 n} = n^{log_2 3}$, and that $log_2 3 < 2$.

Sorting

Bubble Sort

Process: Starting from the beginning, each number compares itself with the next one and swaps with it until it can't anymore. After k iterations, the last k numbers will be sorted
Best Case Runtime: O(n) | Average Case Runtime: $O(n^2)$ | Worst Case Runtime: $O(n^2)$

k = 0	9	8	17	6	5
k = 1					
k = 2					
k = 3					
k = 4					

Selection Sort

Process: Select the smallest element and swap that with the element in the left-most part of the array that isn't sorted yet. After k iterations, the first k elements will be sorted and in the right order.
Best Case Runtime: O(n) | Average Case Runtime: $O(n^2)$ | Worst Case Runtime: $O(n^2)$

k = 0	9	8	17	6	5
k = 1					
k = 2					
k = 3					
k = 4					

Insertion Sort

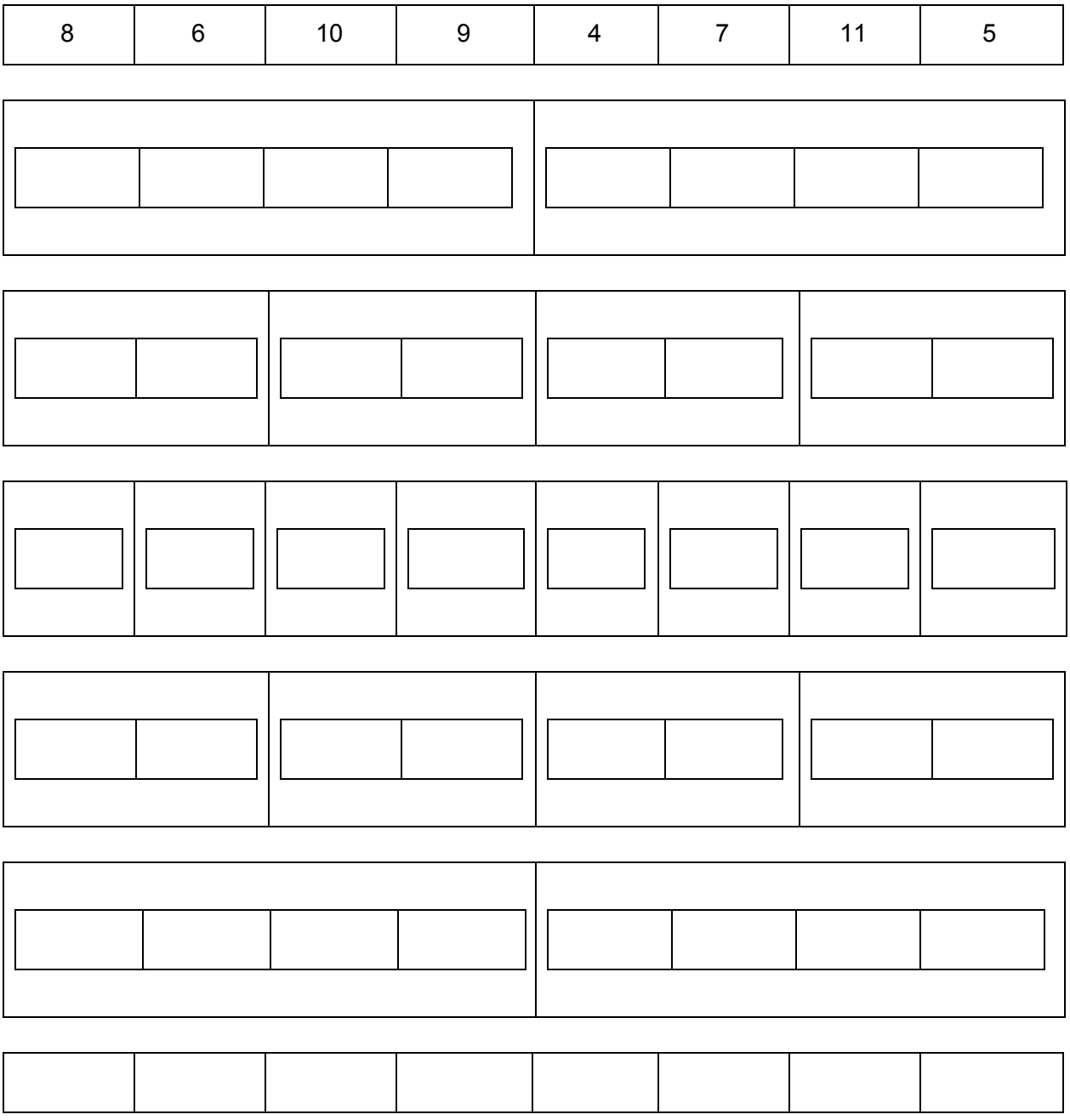
Process: Insert each element into a partition that is already sorted. After k iterations, the first k elements will be sorted but not in their right positions

Best Case Runtime: O(n) | Average Case Runtime: $O(n^2)$ | Worst Case Runtime: $O(n^2)$

k = 0	9	8	17	6	5
k = 1					
k = 2					
k = 3					
k = 4					
k = 5					

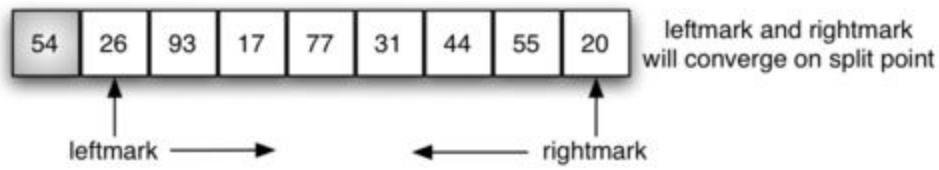
Merge Sort

Process: Break the array down log n times until you have all elements in their own array (size = 1). Then, combine them together using the merge algorithm which is an O(n) operation. Since it's an O(n) operation occurring log n times, the algorithm is O(nlogn).
Best Case Runtime:O(nlogn) | Average Case Runtime:O(nlogn) | Worst Case Runtime:O(nlogn)



Quick Sort

Best Case Runtime: O(nlogn) | Average Case Runtime: O(nlogn) | Worst Case Runtime: $O(n^2)$
Process: Select an element as your pivot, usually the first element. Set a variable “left pointer to the next index” and another variable “right pointer” to the last element. Move the left pointer up until you find an element greater than the pivot, then stop. Move the right pointer down the array until you find an element less than or equal to the pivot, then stop. If at any time, the left and right pointers cross over, stop right away and swap the pivot with the element at the right pointer. Otherwise, swap the left and right pointers once they’ve stopped. At this point in time, the pivot is in its correct position and everything smaller than it is towards it’s left and everything greater is to its right. Call Quick Sort again on the left and right sections of the array.



Left Pointer: > Pivot
Right Pointer: <= Pivot

Binary Search Trees

What are BST's and why do we use them?

A data structure (useful for holding data) that has good runtimes for searching for data by using the idea of binary sort. For any node, all values less than it will be on its left-hand side and all values greater than it will be on its right-hand side.

Typical Struct for BST Node

```
typedef struct node {
        int data;
        struct node* left;
        struct node* right;
} node;
```

Insertion, Searching, and Deletion

- Insertion
  - Every number is inserted at a leaf node.
  - The first number is always the root.
  - For each number afterward, if it is smaller than the root, go to the left child of the root. If it is greater than the root, go to the right child. Continue until you find a leaf node and insert it there.
- Searching
  - Compare the value you're searching for with the root.
  - If the value is equal to root, return
  - If the value is greater than root, search the right subtree
  - If the value is less than root, search the left subtree
- Deletion
  - Search for the value in the BST using the algorithm above.
  - If the node has no children, get rid of it.
  - If the node has 1 child, that child moves up to take its place.
  - If the node has 2 children, replace node with the greatest value in its left subtree.

Tree Traversals

- Preorder Traversal:
  - Print out data of root node, then traverse left subtree and then right subtree
- Inorder Traversal:
  - Traverse left subtree with inorder, print data at root, then traverse right subtree with inorder
- Postorder Traversal:
  - Traverse left subtree with postorder, then right subtree with postorder, then print data at root

Practice Problems

Spring 2019 **Data Structures Exam, Part B**

1) (5 pts) DSN (Binary Trees)

Write a **recursive** function to print a postorder traversal of all the integers in a binary tree. The node struct and function signature are as follows:

```
typedef struct node
{
  struct node *left;
  struct node *right;
  int data;
} node;

void print_postorder(node *root)
{
```

Fall 2018 **Data Structures Exam, Part B**

1) (10 pts) DSN (Binary Search Trees)

Complete writing the function shown below ***recursively***, so that it takes in a pointer to the root of a binary search tree of strings, *root*, and a string, *target*, and returns 1 if the string is contained in the binary search tree and false otherwise. You may assume all strings stored in the tree contain lowercase letters only. In order to receive full credit, your function must run in O(h) time, where h is the height of the binary search tree storing all of the words.

```
typedef struct bstNode {
   struct bstNode *left, *right;
   char str[100];
} bstNode;

int search(bstNode *root, char* target){
```

AVL Trees

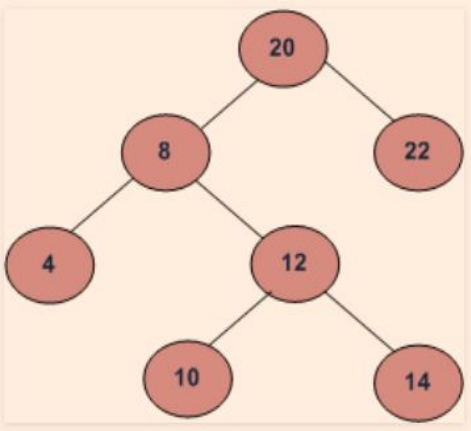
What are AVL Trees and why do we use them?

AVL Trees are self-balancing trees and they prevent the worst-case runtime of Binary Search Trees.

Notes on AVL Trees and Balancing

1. A tree is a valid AVL Tree IF each node in the tree is balanced.
2. To be balanced, a node has to have a balance factor (BF) of -1, 0, or 1.
3. To find out the sign of the BF, check to see if it's left heavy, right heavy, or neither.
    a. If it's left heavy, the BF is positive
    b. If it's right heavy, the BF is negative
    c. If neither, the BF is 0
4. To find out the number of the BF, see how many levels further the tree goes on the side it's heavier on than the side its lighter on.

Example



Insertion

To insert, add in the value like it's a Binary Search Tree. Then, check all the balance factors. If any are invalid, correct them using rotations.

Deletion

To delete, search for the value like it's a Binary Search Tree. If it has 0 children, get rid of it entirely. If it has 1 child, that child moves up to take its spot. If it has 2 children, take the greatest value in its left subtree to replace it. Then, check all the balance factors. If any are invalid, correct them using rotations.

Rotations

If the balance factor of a node is -2, it means it is right heavy so look to its right child. If the child is -1, perform a single left rotation at the bad node. If the right child has a balance factor of +1, first perform a right rotation at the child then left rotation at the bad node.
If the balance factor of a node is +2, it means it is left heavy so look to its left child. If the child is +1, perform a single right rotation at the bad node. If the left child has a balance factor of -1, first perform a left rotation at the child then right rotation at the bad node.

Practice Problem

3) (10 pts) DSN (AVL Trees)

(a) (8 pts) Create an AVL tree by inserting the following values into an initially empty AVL Tree in the order given: 7, 8, 54, 13, 35, 66, 50, and 12. Show the state of the tree after each insertion.

(b) (2 pts) Draw the state of the tree after the deletion of the node containing the value 7.

Heaps

What are Heaps and why do we use them?

Heaps are a tree-like data structure, and we have 2 types: Min Heaps and Max Heaps.
A Max Heap is a tree where a parent node's value is larger than that of any of its descendant nodes. Use it whenever you need quick access to the largest smallest item, because that item will always be the first element in the array or at the root of the tree.
A Min Heap is a tree where a parent node's value is smaller than that of any of its descendant nodes. Use it whenever you need quick access to the smallest item, because that item will always be the first element in the array or at the root of the tree.
However, the remainder of the array is kept partially unsorted. Thus, instant access is only possible for the largest (smallest) item. Insertions are fast, so it's a good way to deal with incoming events or data and always have access to the earliest/biggest. An application for heaps is a priority queue, where the earliest item is desired. Heaps are stored as arrays, not as a tree as they may seem. To insert an element into a heap, put it in the bottom-most left-most position and then percolate up as necessary.

Practice Problem

Spring 2019 **Data Structures Exam, Part B**

2) (10 pts) ALG (Minheaps)

a) Show the result of inserting the value 24 into the following minheap.

b) Show the result of deleting the root of the following minheap.

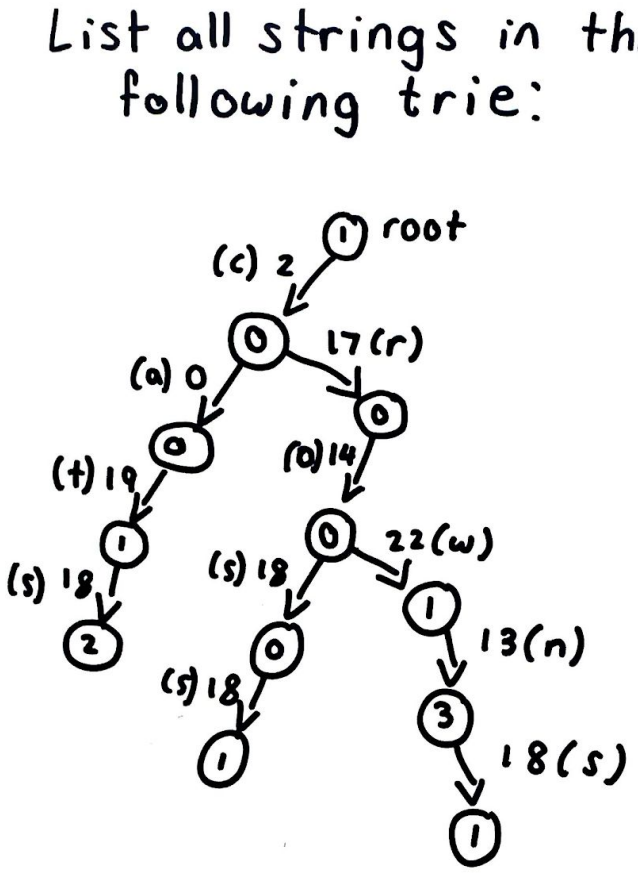
Tries

What are Tries and why do we use them?

A Trie is a tree that stores strings. The number of children of a node is equal to the size of the alphabet. With Tries, we can insert and find strings in ***O(L)*** time where *L* represents the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do with hashing). Another advantage of tries is that we can easily print all words in alphabetical order which is not easily possible with hashing.

Typical Struct for Trie Node

```
typedef struct trieNode {
        int count;
        struct trieNode* children [26];
} trieNode;
```



List of Strings:

Practice Problem

3) (10 pts) DSN (Tries)

Write a recursive function that takes the root of a trie, *root*, and a single character, *alpha*, and returns the number of strings in the trie that contain that letter. You may assume that letter is a valid lowercase alphabetic character ('a' through 'z'). At some point, you will probably find it enormously helpful to write a helper function to assist with part of this task.

Note that we are *not* simply counting how many times a particular letter is represented in the trie. For example, if the trie contains only the strings "apple," "avocado," and "persimmon," then the following function calls should return the values indicated:

```
countStringsWithLetter(root, 'p') = 2
countStringsWithLetter(root, 'm') = 1
```

The TrieNode struct and function signature are as follows:

```
typedef struct TrieNode {
    struct TrieNode *children[26];
    int numwords;
    int flag; // 1 if the string is in the trie, 0 otherwise
} TrieNode;

int countStringsWithLetter(TrieNode *root, char alpha) {
```

Hash Tables

What are Hash Tables and why do we use them?

Hash Tables are a data structure that allow us to have great runtimes if used properly for insertion, deletion, and search. They are also a great tool to use in interviews! To insert in a hash table, we plug in the number into a hash function which gives us the index of where to insert the number: Number → Hash Function → Index. If there is already a number at the given index, we have a collision. There are 3 collision resolution mechanisms:

1. Linear Probing: Keep on going through the array until you find an open space to insert the number at
2. Quadratic Probing: Find the next free index in the array by adding quadratic values ($+1^2$,$+2^2$, $+3^2$, $+4^2$...etc.) to the hash index
3. Separate Chaining: Have separate linked lists, one for each possible value from the hash function and do a head first insertion on the list that needs to be inserted into.

Practice Problem

Insert the following numbers into the following hash table with the hash function mod by table size:

Part 1: Linear Probing

Given the following hash table of size 17, insert the given values with the hash function “mod by table size”. Use linear probing for resolving collisions.
Values: 20, 15, 35, 48, 64, 5, 2, 84, 36, 53, 16

20 % 17 = 3, 15 % 17 = 15, 35 % 17 = 1, 48 % 17 = 14, 64 % 17 = 13, 5 % 17 = 5,
2 % 17 = 2, 84 % 17 = 16, 36 % 17 = 2, 53 % 17 = 2, 16 % 17 = 16.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Part 2: Quadratic Probing

Insert the above values using quadratic probing for resolving collisions

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Part 3: Separate Chaining

Insert the same values using separate chaining for resolving collisions

Bitwise Operator

What are Bitwise Operators and why do we use them?

These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bitwise operators work on these bits.
Bitwise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (XOR), << (left shift) and >> (right shift). << moves each bit over to the left by one and fills the right bits with 0's. Left bit-shifting by n digits is the same as multiplying the number by 2^n. Right bitshifts move every bit to the right. If the number is unsigned, the left bits will be padded with 0's. If it is signed, most compilers will pad the left bit with whatever was in the rightmost bit.

Table *Evaluation of bitwise operators on 1 bit values*

Inputs		and	or	xor
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Input	not
a	~a
0	1
1	0

Base Conversion

From Base 10 to another Base: Divide the given number by the base like you're in elementary school so you get a quotient and remainder. Divide the quotient by the new base to get another set of a quotient and remainder. Repeat until you get a quotient of 0. Read the bits from bottom up to get your new number in the given base.
From another Base to Base 10: Above each of the digits in your number, list the power of the base that the digit represents. Now, there's a simple process of multiplication and addition to determine your base 10 number.

Example

1) Convert $(151)_{10}$ to Base 4

2) Convert $(2113)_4$ to Base 10

2's Complement and Bitmasks

2's Complement is how computers store signed integers. For 2's complement, the leftmost bit tells us if the number is positive or negative. If it is 1, you add -2^{31} to the total of the other bits. To convert a number in binary to 2's complement, simply flip all the bits and add 1.

Practice Problem

Summer 2018 **Algorithms and Analysis Tools Exam, Part B**

3) (5 pts) ALG (Bitwise Operators)

What is the output of the following program?

```
#include <stdio.h>

int main() {

    int n = 182, i = 0;

    while (n > 0) {
        if ((n & (1<<i)) > 0) {
            printf("%d\n", (1<<i));
            n ^=(1<<i);
        }
        i++;
    }
    return 0;
}
```

BackTracking

What is Backtracking and why do we use it?

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem. Usually, recursion is used to trace the path we are taking, so we can back up whenever we find an error in the current solution.

Steps for Solving BackTracking Questions

The following steps are taken from Professor Szumlanski's notes for COP3502:

1. **Base Case**
   We generally implement backtracking using recursion. Your base case generally checks whether you've succeeded at finding a solution to the problem. If so, it might perform some action and/or return some value (a boolean indicating success or a value associated with the solution you just found). Either way, the base case gives your algorithm the ability to return to the previous function call.
2. **(Optional) Check for Duplicate States**
   Sometimes, you want to explicitly check whether you're in a state that you've already seen. If so, return from this function call rather than allowing yourself to get stuck in an infinite loop where you constantly bounce back and forth between states you've already seen. (This can be thought of as a base case as well.)

   Sometimes, instead of checking for duplicate states explicitly, we just set up the for-loop that makes our recursive calls in a way that ensures only new, unique states are passed to the function with each recursive call. Hence, this step that checks for duplicate states is optional in some implementations.
3. **Generate Possible Moves**
   This is generally implemented as a for-loop that changes the parameters passed to this function in some way in order to generate all possible states you could reach from your current state. The steps involved with this are:
   a. **Check Validity of Move**
      In this for-loop, if you're attempting to make some sort of move (i.e., moving a character in a maze), you might first want to check whether it's valid (i.e., does it send your character into a wall, or a fiery pit of doom, or out-of-bounds in your maze array?). If so, skip to the next iteration of this for-loop. (This could also be coded up as a base case in some applications.)
   b. **Change State**
      Make your recursive call to the backtracking algorithm, passing it the new state you're in with this move (i.e., passing it the parameters you just modified in order to make the current move).

c. **Perform Recursive Descent**
   Make your recursive call to the backtracking algorithm, passing it the new state you're in with this move (i.e., passing it the parameters you just modified in order to make the current move).
d. **(Optional) Process the Return Value of the Recursive Call**
   When your recursive function call returns, you might want to capture its return value. It might be returning a boolean indicating whether it found a path to the solution you want. If it did, you might be able to stop making recursive calls. Alternatively, if you're looking for every possible solution to this problem (as with printing all the strings in a trie or finding all the exits in a maze), you might continue searching even if this recursive call already found one solution.
e. **Undo State Change**
   If you return from a recursive call, you need to undo the state change you did in step (b) above before you can go on to the next iteration of this move-generating loop. (This step might be optional in some problems, depending on how you're representing your states.)

****NOTE****
For a majority of backtracking problems, there will be an array that contains all the used positions in the array. If that position hasn’t been used, the value will be 0. But when you recurse down and go to that position, you need to set that value in the used array to 1. If you have to return from that recursive call, remember to turn that value back to 0 to undo the move.

Practice Problem

Summer 2017 **Algorithms and Analysis Tools Exam, Part B**

3) (10 pts) DSN (Backtracking)

For the purposes of this question we define a Top Left Knight's Path to be a sequence of jumps taken by a single knight piece, starting at the top left corner of a R by C board, which visits each square exactly once. The knight may end on any square on the board. Recall that the way a knight jumps is by either moving 1 square left or right, followed by 2 squares up or down, OR by moving 2 squares left or right, followed by 1 square up or down. In this question you'll complete a recursive function that counts the total number of Top Left Knight's Paths for a particular R and C (which will be constants in the code.) Your code should use the constants R and C and should still work if the values of these constants were changed. Complete the recursive function below so that it correctly solves this task. Just fill out the blanks given to you in the recursive function.

```
#include <stdio.h>
#include <stdlib.h>
#define R 6
#define C 6
#define NUMDIR 8
const int DR[] = {-2,-2,-1,-1,1,1,2,2};
const int DC[] = {-1,1,-2,2,-2,2,-1,1};

int main() {
    printf("There were %d Top Left Knight Paths.\n", countTours());
    return 0;
}
int countTours() {
    int used[R][C], i, j;
    for (i=0; i<R; i++)
        for (j=0; j<C; j++)
            used[i][j] = 0;
    return countToursRec(used, 0, 0, 0);
}
int countToursRec(int used[][C], int numMarked, int curR, int curC) {

    if (numMarked == ____________________ )

        return  ____;

    int i, res = 0;
    for (i=0; i<NUMDIR; i++) {

        int nextR = ____________________________;

        int nextC = ____________________________;

        if (inbounds(nextR, nextC) && ____________________________________________ ) {
            used[nextR][nextC] = ____ ;
            res += countToursRec(used, _________________, _________ , _________ );
            used[nextR][nextC] = ____ ;
        }
    }
    return res;
}

int inbounds(int nextR, int nextC) {
    return nextR >= 0 && nextR < R && nextC >= 0 && nextC < C;
}
```

Helpful Links

AVL Trees

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

Backtracking

https://www.geeksforgeeks.org/backtracking-introduction/

Big O Runtimes

https://bigocheatsheet.com

Bitwise Operators

http://ecomputernotes.com/what-is-c/operator/c-bitwise-operators

More Code on Recursion

https://github.com/mujahidrj/SARC-Code/blob/master/Recursion.c

Sorting Algorithms

https://www.geeksforgeeks.org/sorting-algorithms/

Tries

http://simplestcodings.blogspot.com/2012/11/trie-implementation-in-c.html

Quick Sort

https://www.youtube.com/watch?v=7h1s2SojIRw