Week1

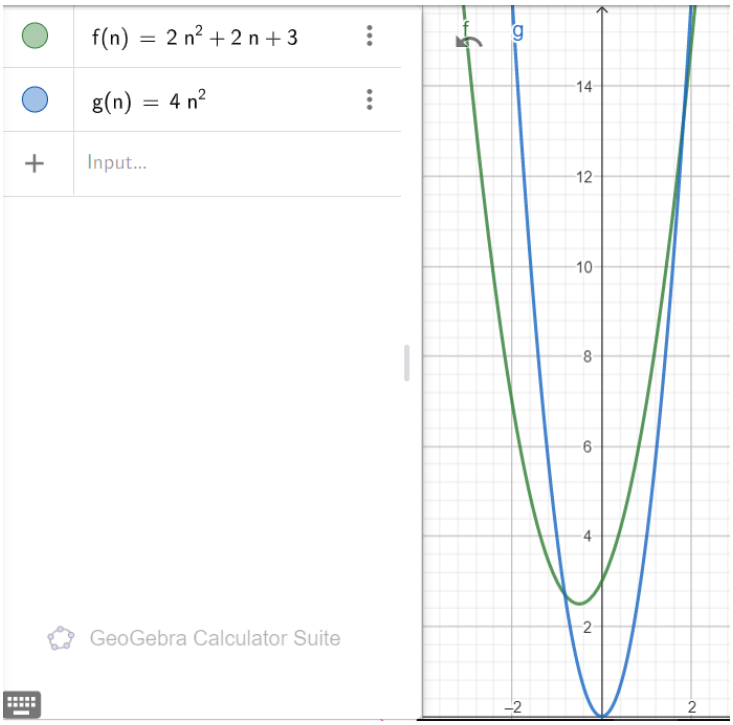
- *NameError* can occur in Python code, if the variable/function cannot be found in the namespace, when it is used.

Week2

- https://www.youtube.com/watch?v=zS1dLE66smA
- Example of calculating time complexity follows:

```
# Time complexity of the following function is
# 1 + (n + 1) + (n *(n + 1)) + (n * n) + 1 = 2n^2 + 2n + 3
def add_in_two_loops(n):
    s = 0 # 1
    for i in range(n): # (n + 1), including the final iteration check
        for j in range(n): # n * (n + 1), including the final iteration check
            s += 1 # 1 * n * n
    return s # 1
```

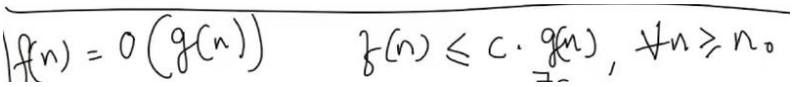
- Complexity of the above function can be represented as f(n) = 2n^2 + 2n + 3. To calculate the asymptotic complexity of f(n), we will use another function g(n) that is above f(n), beyond a specific value of n (n_0) and when multiplied by a constant factor c.
- This is represented graphically as follows:



Here, g(n) = n^2, c = 4 and $n_0 = (1 + \sqrt{7})/2 = 1.82$

- Thus, we conclude that f(n) = O(g(n)) = O(n^2)

- Generally, it can be said



- Find time-complexity when using a *while* loop:

```
# Following code has a time complexity of log2(s)
s = 32
while s > 1:
    s /= 2
print(s)
```

This is because, each iteration in the loop, s is halved and iterates only 5 times.

- How to find time complexity of recursive algorithms by unwinding?

For example, to find the time complexity of a recursive function f with the following complexity for the initial call.

f(n) = 2*f(n-1) + 1

Substitute f(n-1) with 2*f(n-2) + 1.
This leads to f(n) = 2*2*f(n-2) + 2 + 1 = 2^2 *f(n-2) + 2^1 + 2^0

Substitute f(n-2) with 2f(n-3) + 1.
This leads to f(n) = 2*2*2*f(n-3) + 4 + 2 + 1 = 2^3 *f(n-3) + 2^2 + 2^1 + 2^0
…
Substitute f(n-k) with 2*f(n-k) + 1
This leads to f(n) = 2^k *f(n-k) + (2^k-1)

Assuming 1 as the base case, n – k = 1 => k = (n – 1)
Thus, $2^{(n-1)}$ *f(1) + ($2^{(n-1)}$ – 1)

Since f(1) = 1,

The final answer will be $2^{(n-1)}$ *1 + ($2^{(n-1)}$ – 1) = 2^n – 1 (approximately 2^n)

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Search algorithms

Naïve search - O(1), O(n), O(n)

say, we're searching for v in l.
for each item in the list {
    If the item equals v, then you found what you're searching for.
}
You didn't find it anywhere!

Binary search

Implementation 1 - O(1), O(logn), O(logn)

```
say, you're searching for v in l, starting at s and ending at e.
binary_search(v,l,s,e) {

    if e <= s then you've just 1 element in the list, return True if it matches v, else False
    divide the list at the midpoint //m
    If v  is equal to element at m, then you've found it (Best case)
    else {
        if v is more than element at m, then make a recursive call, with s = m + 1.
        if v is less than element at m, then make a recursive cal, with e = m - 1.
    }
}
```

NOTE: This algorithm will work only if the list is sorted.

- Recurrence relation for binary search is T(n) = T(n/2) + 1

Implementation 2 - O(1), O(logn), O(logn)

```
say, you're searching for v in l.
Initialize "start" = 0, and "end" = length(l-1)
loop as long as "start" is less than/equal to "end" {
    divide the list (from start to end) at the midpoint //"mid"
    if element at "mid" equals v, then you've found it (Best case)
    if element at "mid" is more than v, modify "start"= "mid"+ 1
    if element at "mid" is more than v, modify "end"= "mid" – 1
}
You didn't find it anywhere!
```

Sorting algorithms

Selection sort - $O(n^2)$, $O(n^2)$, $O(n^2)$

```
say, you're sorting l (assume l has n elements)
if there's no element in l, return empty list
else {
    loop over all elements in l {
        find the least among remaining elements in the list //"least"
        if "least" is less than current element, then swap them.
    }
}
```

Insertion sort - O(n), $O(n^2)$, $O(n^2)$

```
say, you're sorting l (assume l has n elements)
if there's no element in l, return empty list
```

```
else {
    loop over all elements in l {
        loop over all its previous elements { //while
            if the current element is lesser than previous element, swap them.
            Make previous element, the current element.
        } // At the end of this loop, current element is placed in the "earliest" position.
    }
}
```

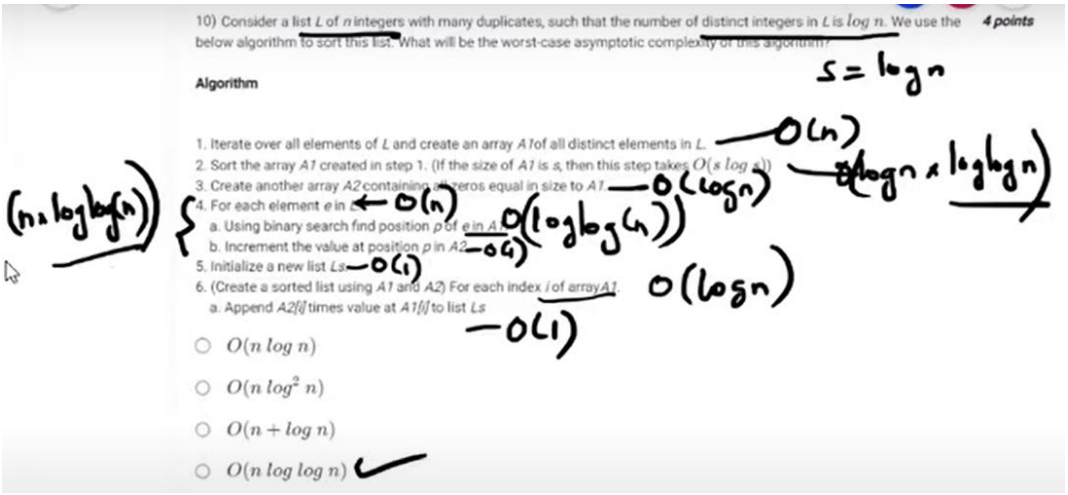
Merge sort - O(nlogn), O(nlogn), O(nlogn)

```
say, you're sorting l (assume l has n elements)
mergesort(L) {
    Divide the list into two equal halves – a left half and a right half.
    Call this recursively until it can't be divided (there's only one element in each half)
    merge(A, B) { // into a new list C
        say, you've two lists(halves) A and B
        while there are elements in A and B {
            If the current element in A is less than current element in B, pick former, else pick
            ---latter, and append to C.
        }
        If there are no more elements in A, add all elements in B to C.
        If there are no more elements in B, add all elements in A to C.
    }
}
```

- If the swap operation was costlier, *selectionsort* would be preferred over *insertionsort*. This is because, *insertionsort* could (potentially) perform n swaps for each iteration, whereas *selectionsort* would perform swap only once for each iteration.
- Time complexities of known algorithm, given n is the input size.
    - Time complexity of Python's slicing operation is *O(n)*.
    - *Binary search* has the best time complexity of *O(1)*, if the middle element is what you're searching for; else O($log_2 n$) always.
    - *Selection sort* has a time complexity of O(n^2) always. Best, worst and average remains the same.
    - *Insertion sort* has the best time complexity of *O(n),* if all elements in the list are already sorted, or are equal; else *O(n^2)* always.
    - *Merge sort* has a time complexity of O($n\log_2 n$) and has a recurrence relation T(n) = 2T(n/2)+ O(n). Best, worst and average remains the same.
    - *Merge* function (which accepts two sorted lists with m and n elements respectively, and merges into a third list) has a time complexity of O(m + n)
    - In a linked list, *append* method takes *O(1)* time, where *delete* method takes *O(n)* time.
    - In *quicksort*, worst case occurs if the partition process picks the largest/smallest element as the pivot. This typically happens, when the input list is sorted in ascending/descending order. In this case, the time complexity is O(n^2) and the recurrence relation is T(n) = T(n-1)+

O(n). In all other cases, the time complexity is $O(n\log_2 n)$ and the recurrence relation is $T(n) = 2T(n/2)+ O(n)$.

- *selectionsort* and *quicksort* result in unstable sorting, while other algorithms result in stable sorting.
- In the case of *mergesort,* sorting is not *in-place,* whereas it's *in-place* for all other sorting algorithms.
- In the case of *insertionsort*, after *m* iterations of the loop, first *m* elements in the list are in sorted order. But, they're not necessarily the smallest elements in the list.
- Here's a classic question on time complexity.



- Only 1 box is open and remaining 99 are locked.
  There is a unique key inside the open box which will open one of the locked boxes.
  Only one locked box has the actual gift, remaining boxes contain a unique key similar to the key in the open box.
  The keys are arranged in such a way that the actual gift can be seen only after all the boxes are opened.
- For the above problem, the time complexity is $O(n^2)$. It's obtained as follows
  $(n - 1) + (n - 2) + (n - 3) + ... + 2 + 1 = n * (n - 1) /2 = O(n^2)$

Week3

Quick sort - O(nlogn), O(nlogn), $O(n^2)$

say, you're sorting l (assume l has n elements)
quicksort(L, l, r)
    If there is only one element between l and r, return the list.
    Identify the pivot element. Typically the first element in the list //l
    //"lower" points to the end of lower section; "upper" points to the end of upper section.
    Initialize "lower" and "upper" to pivot index.
    Loop until there are "unclassified" elements {
        If the next "unclassified" element is greater than pivot, increment upper.
        Else {
            swap it with "lower" element //insert after current "end of lower" section
            increase "lower" and "upper"
        }

```
    }
    Swap pivot element with end of lower section //lower-1
    Reduce "lower" by 1
    Recursively call quicksort for all elements between l and lower //inclusive
    Recursively call quicksort for all elements between lower+1 and upper //inclusive
}
```

Linked Lists

```
say, you're inserting a new node to the start of a linked list
add_head(L, e)
    newest = Node(e)
    newest.next = L.head
    L.head = newest

say, you're inserting a new node to the end of a linked list
add_tail(L, e)
    newest = Node(e)
    newest.next = None
    L.tail.next = newest
    L.tail = newest

say, you're removing the head of a linked list
remove_head(L)
    if L.head == None {
        error
    }
    L.head = Lhead.next
```

- Using linked-lists results in more memory consumption.
- In order to remove collisions in hash tables, either of the following two techniques can be used.
    - Open addressing (close hashing). It works by taking the original hash index and adding successive values linearly/quadratically, until a free slot is found. Note that this *probing* mechanism loops over to the beginning of the list.
    - Closed addressing (open hashing). It works by maintaining separate linked lists for each possible generated index by the hash function.
- Performing a binary search using an array implementation is $O(\log_2 n)$, but using a linked-list is O(n).
- In a singly linked list with head and tail, following operations takes O(1) time
    - adding a *new* head
    - adding a new tail
    - deleting head

  However, deleting the tail will take O(n) time, since in order to set the previous node's next to None, *previous* must be located first, which takes O(n) time. In the case of a doubly-linked list, this operation will take O(1), since *previous* is readily available for each node.

- A stack can be used to evaluate pre-fix/post-fix expressions.

Week4 (Graphs)

BFS (Breadth-First Search)

Given an adjacency list and a node, search breadth first.
Initialize the "visited" dictionary for all keys in the list to False.
Consider the first key, add it to the queue. Mark it "visited"
Repeat as long as there are elements in the queue {
    Take next element from the queue.
    Get the list of values for the key from the adjacency list. These represent the incident nodes.
    Append them in the same order into the queue, only if the they're not already "visited"
}
After the queue is emptied, return the visited dictionary.

DFS (Depth-First Search)

Given an adjacency list and a node, search depth first.
Initialize the "visited" dictionary for all keys in the list to False.
Consider the first key, add it to the stack. DO NOT mark "visited"
Repeat as long as there are elements in the stack {
    Pop out the last element from the stack. Mark it "visited"
    Get the list of values for the key from the adjacency list. These represent the incident nodes.
    Append them in the reverse order into the stack., only if they're not already "visited"
}
After the stack is emptied, return the visited dictionary.

Find Components (BFS)

Given adjacency list, find the components associated with each node.
Initialize "components" dictionary for all nodes to -1
Start with "component_id" of 0
Repeat as long as there are keys in the adjacency list {
    Locate the minimum key that has component = -1
    Get the visited dictionary for the BFS traversal using this key.
    Assign the same "component_id" to all nodes visited during this traversal.
}
After all nodes have been completed, return the components dictionary.

Pre-post order (DFS) – Recursive solution.

Given adjacency list, find the pre-post associated with each node.
Initialize "pre" and "post" dictionary for all nodes to -1
DFSPrePost(AList, v, count) {
    Mark v as "visited" and assign a "pre" count.
    Increment "count"
    Repeat for all neighbors of v {

```
            If not already visited, call DFSPrePost recursively for the neighbor
        }
        At this point, all neighbors have been completed; assign "post" count.
        Increment and return "count"
}
```

Topological sort

```
Given adjacency list, find the topological sort
Initialize indegree of all keys in adjacency list to 0.
Calculate indegree of all nodes (incident on keys) in the adjacency list //indegree
Add all nodes whose indegree is 0 to the queue //zerodegreeq
Repeat as long as there are items in zerodegreeq {
        Take next element from the queue, and add to "toposortlist"
        Reduce indegree for the node
        Reduce indegree for all its neighbors
        If the indegree is 0, then add to queue //zerodegreeq
}
return "toposortlist"
```

Longest path (Very similar to topological sort)

```
Given adjacency list, find the topological sort
Initialize indegree of all keys in adjacency list to 0.
Initialize longest path to all keys in adjacency list to 0.
Calculate indegree of all nodes (incident on keys) in the adjacency list //indegree
Add all nodes whose indegree is 0 to the queue //zerodegreeq
Repeat as long as there are items in zerodegreeq {
        Take next element from the queue, and add to "toposortlist"
        Reduce indegree for the node
        Longest path to the node is the larger of the current longest path and 1 more than longest path to
        its parent.
        Reduce indegree for all its neighbors
        If the indegree is 0, then add to queue //zerodegreeq
}
return "toposortlist"
```

- Number of vertices in a graph is termed O*rder* of the graph; Number of edges in a graph is termed *Size* of the graph
- In a complete graph, every pair of distinct vertices is connected by an edge, and the degrees of all vertices are equal.
- In a connected graph, there is a path between every pair of vertices.
- The maximum (and minimum) number of edges in a complete graph with n vertices is n(n - 1)/2.
- The maximum number of edges in a connected graph with n vertices is (n -1)(n - 2)/2.

- The minimum number of edges in a connected graph with n vertices is (n – 1). This happens when the graph is a tree.
- It is possible but not necessary that a complete graph also be a connected graph. For example, a complete graph with only one vertex is a connected graph, but a complete graph with two or more disconnected components is not a connected graph.
- Maximum number of edges in a directed acyclic graph (DAG) is n(n-1)/2.
- Maximum number of edges in an undirected acyclic graph is (n – 1)
- Maximum path length in a DAG is (n – 1).
- In any graph (both directed and undirected), the sum of the degrees of all the vertices is equal to twice the number of edges. Thus, $2m = \sum deg(v)$, This is known as the handshaking lemma.
- In an undirected connected graph
    - Sum of degrees of all vertices is even.
    - Number of vertices with an odd degree is even.
- The minimum number of colors needed to color a planar graph with n vertices is 4.
- Complexity of DFS is $O(n^2)$ using adjacency matrix, and O(m + n) using adjacency list.
- BFS/DFS can be used to identify the count of connected components.
- In a depth-first traversal of a graph G with n vertices, k edges are marked as tree edges. The number of connected components is (n - k). See https://youtu.be/iPd5Q_MRmgM?&t=6553
- DFS can be used to detect cycles in a graph using pre/post numbering during traversal.
- While performing pre/post numbering using DFS algorithm in a graph, if (u, v) is an edge of the graph such that [pre(v), post(v)] contains [pre(u), post(u)], then the graph has cycles. v->u is a back-edge.
- DAG will have at least one vertex without incoming edges, but might have more than one too (though not necessary)
- DAG will have at least one topological sort sequence, but might have more than one too (though not necessary)
- It is not possible to topologically sort a graph with cycles. Thus, *topological sort* can be used as a mechanism to identify if the graph has cycles.
- DFS will always produce same number of tree edges, irrespective of the order in which vertices are considered. If the graph is connected, it'll always produce (n-1) edges, if there are n vertices.
- Time complexity of topological sort using adjacency list, given that m=#edges and n = #vertices, is O(m + n). Use of adjacency matrix will increase this to $O(n^2)$.
- Time complexity of finding the longest path in DAG using adjacency list, given that m=#edges and n = #vertices, is O(m + n). Use of adjacency matrix will increase this to $O(n^2)$.
- Time complexity of an algorithm to compute incoming edges for each vertex, given that m=#edges and n = #vertices, is O(m + n)
- If (u, v) is an edge of G that is not in the tree T generated as part of a BFS, and d is the shortest distance of the vertex from the starting point, then the possible values of d(u) - d(v) are -1, 0, or 1. If (u, v) is not an edge in G, then u is a leaf in T.

Week5

- Dijkstra's algorithm to find shortest path will not *always* work, when the graph has negative weights. Use Bellman-Ford algorithm in this case.
- Bellman-Ford algorithm can work with negative weight edges, but not with negative weight cycles.
- Bellman-Ford iterates through all edges in a set, each time relaxing the edges.
- Bellman-Ford is run for (n – 1) iterations, each time identifying paths that could reach each vertex through one additional hop. Thus, the first iteration will check for all paths to each vertex in one hop. Second iteration will check for all paths to each vertex in two hop. Until, we've covered (n - 1) iterations.
- Bellman-Ford algorithm can detect negative weight cycles, by running it nth time. If the weights reduce even after (n – 1) iterations, there's a cycle.
- Dijkstra and Bellman-Ford can work on directed or undirected graphs.
- Shortest path in a graph with n vertices can have 1 to (n – 1) edges.
- While finding the all-pairs shortest path using Floyd-Warshall algorithm on a directed weighted graph, if any of the diagonal elements contain a negative weight at the end of the process, then there exists a negative weighted cycle.
- Dijkstra has a time complexity of $O(V^2)$ using simple array, and O((E+V)logV) using a binary heap for priority queue implementation.
- Bellman-Ford has a time complexity of O(VE). Normally, this is larger than Dijkstra's since E > V
- All-pairs shortest paths obtained using Floyd-Warshall algorithm has a time complexity of $O(n^3)$
- Formula to compute shortest path from vertex i to j (with k representing an intermediate vertex)

$$SP^k[i,j] = min[SP^{k-1}[i,j],\ SP^{k-1}[i,k] + SP^{k-1}[k,j]]$$

Spanning trees

- In a graph with n vertices, a spanning tree is a subset of the graph with the n vertices, and (n – 1) edges.
- Number of spanning trees that can be constructed from a graph with n vertices and m edges is mC_{n-1} - #cycles
- Maximum number of spanning trees that can be constructed from a graph is $n^{(n-2)}$. This happens when the graph is complete.
- Adding an edge to a spanning tree will create a cycle.
- In a spanning tree, every pair of nodes is connected by a unique path.
- For a weighted graph, multiple spanning trees could be constructed, but only one of them will be minimum cost. Use Prim's or Kruskal's algorithm might yield different MST, but the cost of both trees will be same.
- Both Kruskal and Prim can be used with arbitrary (including negative) weights and negative weight cycles. Only shortest path algorithm is affected by negative weight cycles.
- It only makes sense to find minimal spanning trees for undirected graphs.
- Prim's algorithm:
    - Selected the edge with least weight/cost
    - Select an edge connected to the last edge, with the least cost.
    - Repeat this, until (n – 1) edges have been selected.

    - Works very similar to Dijkstra, except the relaxation step, which assigns distances[v] with min(d, distances[v]) instead of min(d + distances[u], distances[v])
- In a disconnected graph with multiple components, spanning trees doesn't exist.
- If there are multiple components in the graph, Kruskal's algorithm might be able to find the minimum cost spanning tree for one of the components. Note that this isn't possible using Prim's.
- Kruskals's algorithm:
    - Sort edges in the increasing order of weights, and select the one with least weight/cost
    - Repeat this until (n – 1) edges have been selected, each time selecting the least weight/cost. It's not necessary that the selected edges are connected.
    - Make sure, at each selection, that no cycles are formed at any point.
- Time complexity for Kruskal/Prim's algorithm is $O(n^2)$. Using *minheap,* this can be reduced to *O(nlogn).*
- Kruskal's could find the "missing" edge costs. https://youtu.be/4ZlRH0eK-qQ?t=1001
- Kruskal's might be able to find multiple "minimum cost spanning trees", but the costs of all such will be equal.
- In general, when the edge weights are unique and the graph is connected, both Prim's and Kruskal's algorithms will produce the same minimum spanning tree. However, in certain cases where the edge weights are not unique or the graph is not connected, the algorithms may produce different trees.

Week6

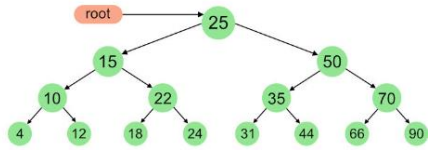
- In Kruskal,
    - Cost of *union (of components)* operation per pair of components is n. Considering that this has to be repeated (n – 1) times, the total cost is n^2 ; we can improve this to nlogn.
    - This improvement can be achieved by maintaining a *members* and *sizes* dictionary.
    - Sorting of edges is mlogm time, using merge sort.
    - Total time complexity can be improved to (m + n) logn time.
- In Prim,
    - The major bottleneck is to find the minimum cost edge from the graph, which is O(n). Considering that this has to be repeated (n – 1) times, the total cost is n^2 ;
    - This can be reduced by keeping the costs in a 2-dimensional queue priority structure. Each row in the matrix is kept sorted. An additional column keeps track of the number of items in each row.
    - Time complexity to insert into the matrix, while keeping a sorted row is O(logn)
    - Time complexity to remove the maximum from the matrix is O(logn)
    - Performing these operation n times, time complexity is O(nlogn)
    - We can store the edge weights in a binary tree (heap), in order to improve the time complexity.
    - Using heap in Prim is advantageous over sorted arrays, since the distances are recalculated each time an edge is added to the MCST. Since this step is missing in Kruskal, using heap in Kruskal isn't advantageous.
- Heaps are complete binary trees, with following constraints,

    - Structural constraint, wherein the tree has to be filled up level by level, starting with level0. Within each level, the nodes must be filled up from left to right. In other words only the lowest level (and in the RHS) can be incomplete.
    - Value constraint, wherein the nodes in the previous levels must be at least as high as the lower levels.
- In the max-heap, value of each node (except the leaves) >= its children.
- In the min-heap, value of each node (except the leaves) <= its children.
- While inserting a node into the heap, every node must be reconciled with its parent for its priority. If the node's priority is more than the parent, it must be swapped with the parent.
- The number of nodes that fill up k levels is $2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^k = 2^{k+1} - 1$
- If we have n nodes in the binary tree, the number of levels in the tree cannot exceed log(n + 1). This also means that this is the maximum height you'll have to navigate and perform swap operations and hence the time complexity for inserting a node into a binary tree is O(logn), if n represents the number of nodes.
- To "get" the highest priority node (*delete_max*) from a max_heap, follow these steps:
    - Remove the root node and return.
    - Swap the last node in the tree (right-most at the last level) into the root's position.
    - Find the largest among root's children. Swap root with the highest among them.
    - Repeat this until you reach the leaf node.
- The time complexity for the *delete_max* operation is also O(logn)
- If the nodes in a heap are stored in a list from left to right, then (2i + 1) and (2i + 2) are the indices to the children of the i^{th} node. Similarly, parent of i^{th} node has an index (i – 1)//2
- Since *insert* and *delete_max* operation are done N times in the Prim's, the total complexity is nlog(n)
- To build a heap from an unsorted array, repeatedly "insert" each element into the list of nodes. This can be done in O(n) time, although technically "heapify" gets called repeatedly (and "heapify" has O(logn) time complexity.
- A binary tree (not binary search tree) is a structure that has at most two child per parent. At the extreme case, this can be a linear structure.
- A binary tree is said to be complete, when each level is complete and can hold no more children.
- A binary tree with n leaves will have (n – 1) internal nodes include root.
- A heap is to be *almost complete binary tree,* since, except for the bottom-most level (leaves), all other levels are complete in a heap. In a heap, it's mandatory that at every level, nodes must be inserted from left to right and don't allow holes in between.
- Searching through a heap for an item will take O(n) time, because there's no order to storing the elements in a heap other than what parent-child relationship requires. Note that searching through a "balanced" binary search tree is O(logn).
- Extracting the largest element in a BST is O(n), since elements are not necessarily arranged in sorted order.
- Heap and Binary search tree are two different data structures, and it's not necessary that a heap is a BST or a BST is a heap.
- Binary search trees are best implemented using recursion. There are 3 traversal mechanisms: in-order, pre-order and post-order, as depicted in the picture below.

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



- In-order is left-root-right, Pre-order is root-left-right, Post-order is left-right-root.
- In the case of BST, we can reconstruct the original tree, in the following cases.
    - if only pre-order traversal is known, first node of which is always the root.
    - if only post-order traversal is known, last node of which is always the root.
- Note that it's not possible to reconstruct the tree, if only in-order traversal is known
- In-order traversal of a BST is always sorted.
- However, in the case of general binary tree, it's not possible to reconstruct original tree, from pre-order or post-order traversals, unless also supplied with in-order traversal order.
- Note that in the case of a traversal (pre-order, in-order or post-order), the time complexity remains O(n).
- Every node in the binary search tree has a value, a left tree and a right tree. All leaves of the tree have an empty node below it, in order to make it simpler for the recursion to work (serves as a base case).
- Inserting a value into the binary search tree tries to locate the value to be inserted in the tree. If the value is found, then it returns without doing anything (No duplicates are allowed in BST). If not found, it sets its value and creates a new tree (node) on the left or the right of the current node.
- It's possible to have an unbalanced tree , for the which the complexity is O(n). If the tree is balanced, all operations (including insert and delete) are O(logn)
- A strict binary tree (each node has two children) with n leaves has (n – 1) internal nodes.

Week7

- Minimum number of nodes in AVL tree of height h is S(h) = S(h-2) + S(h-1) + 1 where S(0) = 0 and S(1) = 1. In order to obtain this easily, use the formula S(h) = fib[h+2] – 1. For example, the minimum number of nodes required to construct an AVL tree with height 12, S(12) = fib(14) – 1. Since, fib(14) = 377. S(12) = 376.
- Similarly, maximum number of nodes in an AVL tree is $2^h – 1$. For example, the maximum number of nodes in an AVL tree with height 12 is $2^{14} – 1 = 16383$
- Huffman encoding uses variable length encoding of letters.
- In this scheme, no letter will have other letters as prefixes. Otherwise, decoding isn't possible.
- Following is the calculation involved.

- Measure frequency $f(x)$ of each letter
  - Fraction of occurrences of x over large text corpus
  - Number of times x appears divided by total number of letters
- $A = \{x_1, x_2, \ldots, x_m\}$
  - $f(x_1) + f(x_2) + \cdots + f(x_m) = 1$
  - $f(x)$ is "probability" that next letter is x corpus
- Message M to be transmitted has n symbols
  - Each letter x occurs $n \cdot f(x)$ times
- Suppose we have the following frequencies for our earlier example

x	a	b	c	d	e
$E(x)$	11	01	001	10	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.25
  - $(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 3) + (0.18 \cdot 2) + (0.05 \cdot 3)$
- Fixed length encoding would require 3 bits per letter — $2^2 < 5 \leq 2^3$
  - 25% saving using variable length code

- Each x is encoded as $E(x)$ with length $|E(x)|$
- Total message length is

$$\sum_{x \in A} n \cdot f(x) \cdot |E(x)|$$

- Average number of bits per letter in encoding

$$\sum_{x \in A} f(x) \cdot |E(x)|$$

- A better encoding

x	a	b	c	d	e
$E(x)$	11	10	01	001	000
$f(x)$	0.32	0.25	0.20	0.18	0.05

- Average number of bits per letter is 2.23
  - $(0.32 \cdot 2) + (0.25 \cdot 2) + (0.20 \cdot 2) + (0.18 \cdot 3) + (0.05 \cdot 3)$
- Given a set of letters A and frequences $f(x)$ for each x, produce the most efficient prefix code possible
  - Minimize $ABL(A)$ — Average Bits per Letter

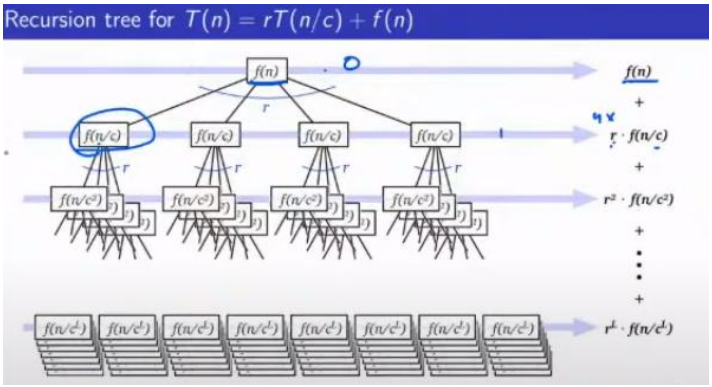
- Binary trees can be used to represent encoding, where letters are leaves and path to leaf describes encoding – 0 is left and 1 is right.
- In the above representation, every node in the tree has 0 or 2 children (no node with a single child), thus constructing it as a full tree. In other words, none or two letters can be represented with a certain number of bits.

- Following 3 rules apply to Huffman encoding:
  - Any optimal prefix code produces a full tree
  - In an optimal tree, if leaf *x* is at lesser depth than leaf *y*, *f (x) >= f (y)*. In other words, all letters with lowest frequencies are pushed to the bottom of the tree.
  - In an optimal tree, for any leaf at maximum depth, its sibling is also a leaf.

Week8

- Given two sets of symbols, inversion is the set of all pairs *(i , j)* such that *i < j*, but *j* occurs before *i* in either set. The maximum number of possible inversions is n(n – 1)/2, given n is the number of elements in either set.
- In order to find all inversions, it'll take $O(n^2)$ time using the naïve method. With the divide and conquer approach (like merge-sort), it'll take O(nlogn) time.
- Naïve algorithm used for Integer multiplication has a recurrence relation of $4T(n/2) + n = O(n^2)$
- Karatsuba's algorithm used for integer multiplication has a recurrence relation of $3T(n/2) + n = O(n^{\log 3})$

- Quick-select uses “wall index” returned by partitioning algorithm. This is the kth least in the array.
- The time complexity of Quick-select algorithm is O(n) on average and O(n^2) in the worst case, and is dependent on the partitioning strategy.
- The worst case time complexity of Fast-select algorithm is O(n).
- Time complexity of the brute force closest pair problem is $O(n^2)$, and using the divide-conquer approach is O(nlogn)
- Time complexity of recurrence based algorithms at each level can be generally represented as $r^i * f(n/c^i)$, where r is the number of recurrence calls at the level *i*, c is the division factor and f(n) represents the time spent on non-recursive work.



- Leaves correspond to the base case T(1).
- If L represents the number of levels, number of leaves is r^L

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- Decreasing Each term is a constant factor smaller than previous term
  - Root dominates the sum, $T(n) = O(f(n))$
- Equal All terms in the series are equal
  - $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$ — $\log_c n$ is asymptotically same as $\log n$
- Increasing Series grows exponentially, each term a constant factor larger than previous term
  - Leaves dominate the sum, $T(n) = O(n^{\log_c r})$

- For a recurrence relation T(n) = 2T(n/8) + O(n), the time complexity is O(n). This is the *decreasing* case above.
- For a recurrence relation T(n) = 4T(n/4) + O(n), the time complexity is O(nlogn). This is the *equal* case above. Example: 4-way merge sort.
- For a recurrence relation T(n) = T(n – 1) + O(n), the time complexity is $O(n^2)$. Example: Worst case of *quicksort*.
- For a recurrence relation T(n) = T(n/2) + O(1), the time complexity is O(logn). Example: *binary search*.

- For a recurrence relation T(n) = T(n/2) + O(n), the time complexity is O(n). This is the *decreasing* case above. Example: Find kth smallest (largest) element using fastselect. In this case, algorithm uses divide and conquer, and uses MoM to find the middle element.