Pre-Placement Grind

Hardik Rajpal

November 27, 2023

Contents

Chapter 1

Misc.

TODO: pragma O3 stuff

1.1 Misc. Confirmed Optimizations

1. While using `cout`, `cerr` is fine during debugging, remember to remove them before submission as they greatly affect runtime.
2. When writing custom comparators, avoid having memory accesses (`a[i][j]`) inside the comparator. Instead, modify the data structures whose elements are being compared to pass values that can be used in the comparator without memory accesses.
3. For maps,

```
auto iter = m.find(k);
if(iter!=m.end()){return iter->second;}
//is much faster than:
if(m.count(k)){return m[k];}
```

4. Replace sets that check for inclusion by bit operations with an integer if the number elements < 32 for ints and 64 for long longs.
5. Pass params by reference where possible.
6. Instead of a reference parameter, consider using a pointer to that variable stored in a class data member.
7. Replace data types by smaller data types where possible:
   - long long by int
   - int by char
8. Replace fixed-length vectors by array<type,fixed-length>.
9. Replace maps by vectors if they are indexed by integers within a fixed range. Ex: the alphabet as indices.
10. That die roll problem optimization. (Needs to be phrased more mathematically.)
11. Use of suffix arrays + descending order (or its prefix counterpart) to cut of paths in backtracking.

1.2 Binary Representations

1.2.1 Unsigned numbers

Represent n using k bits $b_{k-1}b_{k-2}...b_0$:

$$n \in \{0, 1, ...2^k - 1\}, n = \sum_{i=0}^{k-1} b_i \times 2^i \tag{1.1}$$

1.2.2 1's complement

Represent n using k bits $b_{k-1}b_{k-2}...b_0$:

$$n \in -(2^{k-1} - 1), ... - 0, 0, 1, ...2^{k-1} - 1, n = (-1)^{b_{k-1}}(\sum_{i=0}^{k-2}(c_i) \times 2^i) \tag{1.2}$$

- Where if $b_{k-1} == 1$, $c_i = 1 - b_i$, else $c_i = b_i$.
- It has two representations of zero: 1* and 0*.
- MSB = 1 $\implies$ n ≤ 0, else n ≥ 0.
- Bits to n is equivalent to
  - Identify sign based on MSB.
  - If MSB ==1, flip all bits.
  - Find the unsigned integer they represent and combine the sign.
- To take one's complement of a number is to find its negative counterpart.
- It is equivalent to just flipping every bit.

1.2.3 2's complement

Represent n using k bits $b_{k-1}b_{k-2}...b_0$:

$$n \in \{2^{k-1}, 2^{k-1} - 1, ... - 1, 0, 1, ...2^{k-1} - 1\}, n = (-1)^{b_{k-1}} \sum_{i=0}^{i=k-1} b_i \times 2^i \tag{1.3}$$

- To take 2's complement of a number is to flip its bits and add 1 to it.
- This is equivalent to finding its negative in the 2's complement notation.

1.2.4 Applications

Using the fact that computers use 2's complement and bit operations, we can check certain facts about a number in O(1) time:

- $n = 2^k \iff (n > 0)\&\&(n\&n - 1 == 0)$
- $n = 2^{2k} \iff (n > 0)\&\&(n\&n - 1 == 0)\&\&(n\&mask == n)$, where $mask = 0x55555555$.
- Additionally, for any prime p, we can check if n is p^k, by checking if $ub\%n == 0$, where ub is the largest power of p that fits within the integer upper bound in the language.

1.3 Modular Arithmetic

- a % b returns the remainder when a is divided by b.
- $a\%b = a - floor(a/b) * b$.
- $a \equiv bmodn \iff a\%n == b\%n \iff (a-b)\%n == 0$
- The congruence relation modulo n splits integers into n residue (equivalence) classe $\{C_i\}_{i=0}^{i=n-1}$:

$$C_i = \{z \| z\%n = i\}; \tag{1.4}$$

- Representative of C_i is $r_i \in C_i, 0 \leq r_i \leq n-1$
- Operations:
  - $a + b\%m = ((a\%m) + (b\%m))\%m$
  - $a - b\%m = ((a\%m) - (b\%m))\%m$
  - $(a * b)\%m = ((a\%m) * (b\%m))\%m$
  - $(a/b)\%m = ((a\%m) * (b^{-1}\%m))\%m$
- Note: in c++, a % m where a<0 returns -((-a)%m), which is not equal to the mathematical result of (m - ((-a) % m)).
- If a < 0, a mod m should be calculated as (m + (a % m)).
- Or the following function extends to both negative and positive a's : (m + (a % m)) % m.
- Exponentiation.
- $a == b \mod m \implies a^e = b^e \mod m$.
- Note: the converse is NOT true. It results in an algebraic equations with possibly multiple roots.
- $a^e = a^c \mod n \iff e == c \mod \phi(n)$ and gcd(a,n) == 1.
- If n is prime, then $a^e = a^{e \mod \phi(n)} \mod n \forall a$
- TODO: CRT, problems.

1.4 Master Theorem for Recurrence

$$T(n) \leq aT(n/b) + cn^d \implies T(n) \leq a^{log_b(n)}T(0) + c.n^d.(\sum_{i=1}^{log_b(n)-1} \frac{a}{b^d}) \tag{1.5}$$

$$\implies T(n) = \begin{cases} O(n^d log(n)) & a = b^d \\ O(n^d) & a < b^d \\ O(n^{log_b(a)}) & a > b^d \end{cases} \tag{1.6}$$

1.5 Useful Built-In C++ functions

1. `swap()`
2. `lcm, gcd`
3. `cout<<std::boolalpha<<var_boolean;` Prints true or false instead 1 or 0.

1.6 Design Rules

1. One-Definition Rule: TODO

1.7 Watch out for these bugs

- When reusing a pointer to heap data, remember to set it to NULL after you `delete` it.
- Prefer using (small.size()¿large.size()+k) where k¿=0, as opposed to (small.size() - large.size() ¿ k). The two return values are unsigned and thus always yield a non-negative difference.
- Separate out the iterating condition (limit, checks, increments to values) from a map before updating it, to avoid infinite loops and incorrect results.

Chapter 2

Week 1

2.1 Searching Algorithms

Notes from GFG

These are algorithms to check for the existence of an element or to retrieve it from a data structure. The retrieval can also involve only returning the position (index) or a pointer to the element. There are two types:

1. Sequential search: check every element based on a pre-determined sequence (ex. linear, alternating, etc.), and return the matches.
2. Interval search: Designed for searching in **sorted** data structures. They involve **repeatedly** dividing the search space into intervals which can be excluded entirely after certain checks (ex. binary search).

Some search algorithms are discussed below:

1. **Linear Search:** Straighforward for-loop iterating over all elements in an array.
   Time: O(n)
   Space: O(1)
2. **Sentinel Linear Search:** Reduces the number of comparisons by eliminating the need to check if the index is within bounds. This is accomplished by appending the target element to the end of the array, and treating its index in the result as "not found."
   Time: O(n)
   Space: O(1)
3. **Binary Search:** It's used for sorted arrays. It involves comparing the element at the center of the interval (defined initially as the entire array), with the target element. One of the halves of the interval is picked based on this comparison. The interval shrinks until the target is found or an interval of size one is not equal to the element. It can be implemented recursively or iteratively, each involving a step similar to $m = l + \frac{(r-l)}{2}$ while $l \leq r$
   Time: .
   Space: O(log(n)) O(1)
4. **Meta Binary Search:** Seems unimportant but check it here
   Time: O(log(n))
   Space: O(1)
5. **K-ary Search:** The search space is divided into k intervals in each step and one of them is picked to proceed further by comparing the target element to the interval markers.
   Time: O(log(n)). The reduction is of a constant term: $log_k 2$
   Space: O(1)

6. **Jump Search:** The sorted array is examined in jumps of the optimal size $\sqrt{n}$,until the element being examined is greater than the target element. The interval is then shrunk to the previous interval. The shurnken interval can be examined linearly or with another jump search.
   Time: O($2\sqrt{n} = O(\sqrt{n})$), or $O(n^{1/2} + n^{1/4} + n^{1/8}...) = O(\sqrt{n})$
   Space: O(1)

7. **Interpolation Search:** It improves over binary search only if the data is uniformly distributed. It involves selecting the splitting point of the current search space by comparing the target value to the current lower and upper bounds of the space. Linear interpolation involves the following equations:
   $slope = (arr[r] - arr[l])/(r - l)$
   $m = l + slope \times (x - arr[l])$
   Time: O(log(log(n))) on average, O(n) WCS.
   Space: O(1)

8. **Exponential or Unbounded (Binary) Search:** We examine the search space from the lower end l, comparing $l + 2^k - 1$ with the target element x, where k is the number of comparisons so far, until $x < arr[l + 2^k - 1]$. Then, we examine the interval bounded by $l + 2^{k-1} - 1$ and $l + 2^k - 1$, using binary search.
   Time: O(log(n)), where n is the length of the array or where the first occurrence of the target element exists in an unbounded array.
   Space: O(1)

9. **Fibonacci Search:** The array must be sorted. We first find the Fibonacci number $f(m)$ that exceeds the length of the given array. We compare the target element to the element at $arr[f(m-2)]$. We pick an interval based on the outcome.
   Time: O(log(n))
   Space: O(1)

Misc

- The preferred formula for evaluating the middle point of the interval in binary search is $m = l + (r - l)/2$, and not $m = (l + r)/2$, as the latter can suffer overflow.
- Global variables can also be used to maintain a "best value yet" while searching through a space with binary search. For ex. find the first element $\geq$ x in an array.
- Problems where an array can be mapped to a boolean variable and is guarranteed to have either
  - F...FT...T or
  - T...TF...F

  and our aim is to find the boundary between true and false values can be translated to a binary search problem, with the target as the point where the variable changes: arr[i] != arr[i+1].
- Remember the `break` statement in iterative binary search if the middle point element is equal to the target.
- One can also binary search for a target range's starting point, instead of just a target. See this problem.
- In some cases, we might want to keep the current middle point `m` in the search space, here we resort to replacing either one of `r = m - 1` or `l = m + 1` by `= m` and change the loop invariant `l <= r` to `l < r`.

2.2 Sorting Algorithms

These algorithms rearrange a given array in ascending order. Various other orders can be achieved by modifying the comparison operator. A sorting algorithm is **stable** if it preserves the relative order of equal elements.

Merge Sort

The first part of the algorithm recursively handles halves of the given array. The second part merges the halves sorted by the first part. It takes O(nlog(n)) time in the **all cases**. O(n) space is necessary for the merging side of affairs. Implemented recursively. It's advantages include stability, parallelizability and lower time complexity. It's disadvantages include higher space complexity and not being in-place, and that it's not always optimal for small datasets.

Quick Sort

It involves recursively picking an element (**the pivot**) from the unsorted array, placing it so that all elements less than it are before and all those greater than it are after. Then calling this function on the sub-arrays after and before the chosen element.

The Others

1. **Selection Sort:** The given array is viewed in two parts; sorted and unsorted. Every iteration involves **selecting** the minimal element and swapping it with the first element of the unsorted part. Hence, the boundary of the sorted part is expanded and that of the unsorted part has contracted. All of this happens inplace. It isn't stable.
   Time: O(n^2)
   Space: O(1)

2. **Bubble Sort:** This involves repeatedly traversing the array, swapping any two **adjacent** elements if they are in the incorrect (descending) order, until we encounter a run with no swaps. It is stable. With each iteration, the last elements of the array are sorted in ascending order.
   Time: O(n^2)
   Space: O(1)

3. **Insertion Sort:** It involves iterating over the array once, and in each iteration, if the current element is less than its left neighbour, we move it leftwards until its left neighbour is lower than it. It is in-place and stable. Best case happens when the array is sorted: O(n). Worst case is when it's in descending order: O(n^2). Average time is O(n^2).
   Time: O(n^2)
   Space: O(1)

Chapter 3

Week 2

See the other notes.pdf

Chapter 4

Week 3

4.1 Complete Search

Subset Processing

We use the function below with 0. (n = size of given set.)

```
void search(int k) {
    if (k == n) {
        // process subset
        subsets.push_back(subset);
    }
    else{
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

Listing 4.1: Subset Generation

```
for (int b = 0; b < (1<<n); b++) {
    //b runs from 00..00 to 11...11
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)){
            subset.push_back(i)
        };
    }
}
```

Permutation Generation

Permutation of a vector can be generated as follows:

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(),permutation.end()));
```

Backtracking En General

If the dimensions of inputs are smaller than usual, backtracking is an option. As with other algorithms, you want to optimize this as much as possible. Optimizations are possible by:

1. Transforming the inputs so as to reduce the search space.
   Example: If you are searching for a subset whose sum is a given target, Searching the space of frequency map is better than searching subsets in the untransformed set, at least when duplicates are abundant.
2. Cutting off fruitless search paths as soon as possible. (Pruning the search tree.)
3. Specifying "min" requirements before taking a path, and equivalently, specifying "max" allowed values in a path to be explored further.
4. Optimizing the data structures used to record the current state and restrictions. Particularly,
   - Using vectors instead of maps where possible.
   - Using bitmap `int`s when only inclusion is to be checked.
5. Instead of using min/max to bring index values within range, which will likely incur repeated searches at the boundary, use an if block to disregard paths associated with values that exceed the bounds.
6. A modification of the needle may speed up the search. For example, the search for a word may be sped up by searching for its reversed word if the end letter is less frequent than the letter at the start.

The abstract code for backtracking looks like ths:

```
    //declare global/class member variables.
    void search(int p){
        //p signifies path/position being inspected
        //in the search space.
        if(checkTerminalConditions()){
            if(globalVarSolutionValid){
                //update collection of solutions.
            }
        }
        else{
            for(possible path of exploration){
                //(1)update global vars so as to take this path.
                search(p+1);
                //(2)undo the updates made to global variables.
                //(not necessary if (1) overrides/uses previous updates.)
            }
            //undo any leftover changes made to global variables.
        }
    }
```

Pruning: A way of adding intelligence to the backtracking algorithm and reducing the time spent in fruitless paths. Additionally, we can leverage symmetries of the search space to check only a fraction of the entire possible solution set. Clearly, optimizations at the start of the search tree save a lot more time than those at the end.

Meet in the Middle

Another name for **Divide and Conquer**. It refers to splitting the search space up into two halves and combining the results of the two halves. It works if there is an efficient way to combine the results. Even 1 level of splitting (and extracting solutions from the halves using brute force) can have worthwhile optimizations: $O(2^n) \implies O(2^{n/2})$.

Chapter 5

Week 4

5.1 Greedy Algorithms

Reading Notes

Greedy Solutions focus on looking at the problem in smaller steps, and at each step we select the option that offers the most obvious and immediate benefit. It's sort of like assuming there's only one maximum point in the search space, and hence, we just move in the direction with the most inclination. Some popular greedy algorithms are:

- Dijkstra's shortest path.
- Kruskal's minimum spanning tree.
- Prim's minimum spanning tree.
- Huffman encoding.

With greedy algorithms, we often have to repeatedly pick the minimal element from a collection; hence using a `priority_queue` or a `multiset` is often helpful.

5.1.1 Union Find

The data structure can also show up in greedy algorithms. Given below is the most optimized implementation of `find` and `combine`.

```
vector<T> items;//given vector of items.
vector<int> root;//representative roots of trees array.
vector<int> rank;//for combine optimization.
int find(int u){
    if(root[u]==u){return u;}
    //instead of return find(root[u]), do:
    root[u] = find(root[u]);//path compression
    return root[u];
}
void combine(int u, int v){
    int ru, rv;
    ru = find(u);rv = find(v);
    if(ru!=rv){
        //u, v in different trees.
        if(rank[ru] < rank[rv]){
            root[ru] = rv;
            rank[rv] += rank[ru];
            //combined tree has least possible height.
        }
        else{
            root[rv] = ru;
```

```
            rank[ru] += rank[rv];
        }
    }
}
```

Variations of `root` array

The usual union-find implementation's root elements satisfy `root[r] == r`. However, we can also use negative numbers at `root[r]` (which can't be the index of any parent), and check for `root[r] < 0` when searching for the root. Such a setup allows for recording information in the domain of negative numbers at the root, say, the size of the tree, but negated. The combine function then simply sets the combined tree's root value to the confluence of values at `rv` and `ru`.

Kruskal's MST Algorithm

1. Have a min-heap of all edges.
2. Iterate through the heap, merging the trees of the vertices of each edge. For each non-trivial merge, update a counter. Additionally, add the edge to the list of edges for the MST or its weight to the weight of the MST.
3. Once the merge counter is at $|V|$ - 1, break.

Prim's MST Algorithm

1. Pick a starting vertex. Initialize an empty min-heap of edges. Maintain a count of visited vertices.
2. Mark current vertex as visited.
3. Add all edges going out of the current vertex to the heap.
4. Iterate through the heap until an edge to an unvisited point is found.
5. Set this point as the current point. Iterate until count of visited vertices = $|V|$.

Dijkstra's Shortest Path

1. Pick a starting vertex. Maintain an array of minimum distances to reach any vertex from a visited vertex. For visited vertices, this should be -1.
2. Update distances of array elements as min(old distance, distance from current point which is INT_MAX if they are not neighbours). While iterating, record the array element with minimum distance to it.
3. Set the recorded element as the current vertex and continue until the current vertex is the target vertex.

Modifications can be made to record the predecessors in the paths or calculate the weights of the paths.

```
int distance(vector<int> &pi, vector<int> &pj);
int dijkstras(vector<vector<int>>& ps, int s, int target) {
    int n = ps.size(), res = 0, i = s;
    vector<int> min_d(n, INT_MAX);
    while (i != target) {
        min_d[i] = -1;
        int min_j = i;
        for (int j = 0; j < n; ++j){
            if (min_d[j] != -1) {//visited vertices.
                min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
                min_j = min_d[j] < min_d[min_j] ? j : min_j;
            }
        }
        res += min_d[min_j];
        i = min_j;
```

```
    }
    return res;
}
```

Listing 5.1: Shortest Path

```
int distance(vector<int> &pi, vector<int> &pj);
int dijkstras(vector<vector<int>>& ps, int s, int target) {
    int n = ps.size(), res = 0, i = s,connected = 0;
    vector<int> min_d(n, INT_MAX);
    while (connected < n) {
        min_d[i] = -1;
        connected++;
        int min_j = i;
        for (int j = 0; j < n; ++j){
            if (min_d[j] != -1) {//visited vertices.
                min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
                min_j = min_d[j] < min_d[min_j] ? j : min_j;
            }
        }
        res += min_d[min_j];
        i = min_j;
    }
    return res;
}
```

Listing 5.2: Dijkstra's MST

- Dijkstra's can be modified to count the number of minimum cost paths between two fixed points too.
- This involves maintaining an array of `ways` denoting the number of min-cost ways to visit a vertex. The answer is `ways[dst]`. See this problem.

Stack Based Questions

These usually involve finding the (lexicographically) minimal subsequence. We maintain a stack to track the sequence selected so far. To reverse a stack to get the subsequence, the most optimal method is:

```
while(s.size()){
    ans.push_back(s.top());
    s.pop();
}
reverse(ans.begin(),ans.end());
```

Heap+Queue

Honestly I've only seen one question with this paradigm. However, it's worth a shot if you realize you have to process numbers starting always with the largest/smallest element, and have to track elements being available/unavailable over time. I know that's a very vague and oddly specific situation, but I couldn't just walk by a problem and not make this note.
Additionally, in scheduling problems, consider trying to find a way to arrange the given tasks, which might result in a closed form solution.

5.1.2 Greedy Matching

Given two arrays to match elements such that the matching function can be put into a total order over the elements (ISTG I will word this better, later), we can sort two arrays and take the first matches offered by traversing one array, selecting the first matched element with the element being traversed.

5.1.3 Misc Data Structures

Multiple problems tagged "Greedy" are really just a matter of organizing the input data in a structure such as a (frequency) map or a heap. Or we're just sorting the input array. So, consider this when thinking of approaching a question greedily.

5.2 2 pointers

- Problems that apparently involve an O(n^2) search space of solutions may be done in O(n), if a we can find some reason to ignore certain sets of coordinates.
- With two-pointer approaches, we try to see if there's a reason to move any one particular pointer instead of the other.
- An example problem would be this and this.

Chapter 6

Week 5

6.1 Dynamic Programming

A common optimization to look out for when writing the code for dynamic programming problems, try to ensure that

1. The code doesn't compute paths that aren't going to be useful.
2. The code doesn't recompute any path more than once.

As per this article, the approach to most dynamic programming problems can be broken down to:

1. Find recursive relation.
2. Recursive (top-down).
3. Add Memoization.
4. Iterative + memoization (bottom-up).
5. Further optimizations.
    - Discarding paths
    - Reducing space complexity.

6.1.1 Common Patterns

Min (Max) Path to Reach Target

Distinct Ways

- It might help to convert direction problems into terms involving sums and bounds, which are easier to reason about and enforce.

Merging Intervals

DP on Strings

Decision Making

Chapter 7

OOP (in C++)

7.1 OOPs

- Access-specifiers:
  1. private: can only be accessed inside the class.
  2. protected: can be accessed inside the class and inside derived classes.
  3. public: can be accessed everywhere.

7.2 Inheritance from TutorialsPoint and LearnCpp.com

```
1   class DerivedClass: access-specifier BaseClass{
2       //access-specifier is one on public/private/protected.
3   };
```

Listing 7.1: Syntax

- Allows us to define a class in terms of another class.
- Derived classes inherit properties of base classes.
- Inheritance implements "is-a" relationship. Ex: mammal is-a animal, dog is-a mammal =¿ dog is-a animal also holds.
- Derived classes inherit all properties of base classes except:
  1. Constructors, destructors and copy constructors.
  2. Overloaded operators.
  3. Friend functions.
- The base classes can be inherited through public, protected or private inheritance, which is specified by the access specifier before its name in the declaration of the derived class. The results are:
  1. Public: access permissions of public and protected members of the base class are carried forward in the inherited class.
  2. Protected: access permissions of public and protected members of the base class are lowered to protected.
  3. Private: access permissions of public and protected members of the base class are lowered to private.

7.2.1 Multiple Inheritance

```
class DerivedClass: access-specifier baseA, access-specifier baseB ... {
//access-specifier is one of public/protected/private.
};
```

Listing 7.2: Syntax

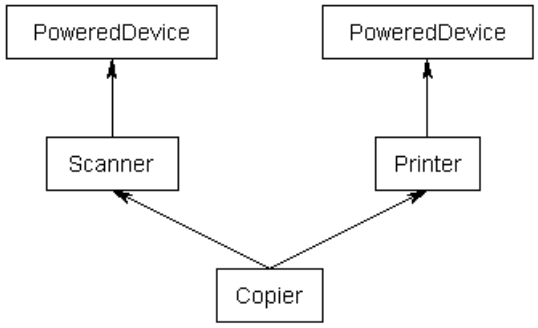
- **Mixins**: a small class that can be inherited from (in combination with other classes) to add properties to the derived class.
- The constructors of parent base classes are called in the order that they are declared and before the constructor of the derived class.
- Note that destructors are called in the completely reverse order of constructors. (Think of it as a stack of objects of base classes, with the derived class at the top).

```
class Derived: public Base1, public Base2...
//base1 constructor.
//base2 constructor.
//derived constructor.
...
//derived destructor.
//base 2 destructor.
//base 1 destructor.
```

- If two parent classes contain members with the same signature (name, args), a call to the signature from their common child class' object raises a compilation error. This is resolved using scope resolution operators.

Diamond Problem of Multiple Inheritance

- When two parent classes that share a base class are used to derive a child class, the inheritance tree looks like this:



- Each parent class has its own copy of the base class data members (resulting in redundant copies), and we can't call public members of the base class from the new derived class.
- The diamond problem refers to our liking for a single instance of the base class in such cases of multiple inheritance, which is different from what happens when we try to implement such a hierarchy.
- The solution is to use the keyword virtual while declaring the parent classes to identify them as virtual base classes:

```
class base{};
class b1: virtual public base{};//
class b2: virtual public base{};//without virtual in both of them, copies are made.
class derived: public b1, public b2{};
```

  - Without `virtual`, each parent class maintains its copy of variables from the base class, and sizeof(derived) = (sizeof(b1)+size(b2)).
  - With `virtual`,
    sizeof(derived) = (sizeof(b1 without b data)+sizeof(b2 without b data) + sizeof(b)+16B)
  - The 16B are for book keeping. It can also be 8B on some systems.
  - Note that all parent classes are prefixed with `virtual` and share ancestors, have a single copy of the ancestor's variables in the derived class.
  - The book-keeping info grows with 8B for each new parent class. It doesn't grow with the size of the base class (or any class).

- The construction of the base class becomes the responsibility of the derived class:

```
class Copier: public Scanner, public Printer{
public:
Copier():PoweredDevice()/*base class*/, Scanner(), Printer(){
    //Order of constructor defintions run.
    //Base
    //base1
    //base2
    //Derived
}
}
```

- The point above is true because of Printer, Scanner being virtual base classes. The order of constructors holds **even when single inheritance is done from a virtual base class**.
- Accessing members of the root class, using an object of a class derived from two non-virtual sibling descendants of the root, leads to compilation errors. To avoid this, either declare the siblings as virtual descendants or use scope resolution operators (of the sibling classes, not the root!).

7.3 Overloading

TODO order of usage around operators study.

- A single identifier (function name/operator) corresponds to two different implementations, based on the argument list supplied to it.
- Overload resolution refers to the compiler's task of selecting the most appropriate implementation when it encounters a call to an overloaded function.
- Note: operators can be overloaded outside classes too:

```
        //As a member function:
        Box operator+(const Box&);
        //Not as a member function:
        Box operator+(const Box&, const Box&);

```

- In general, use const and & for operands to
  - Avoid accidentally modifying them in the operation.
  - Avoid time spent copying them around.
- Most operators can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Operators that can't be overloaded:
  1. ::
  2. .*
  3. .
  4. ?:
- Unary operators include: ++ (post,pre), – (post,pre), -, and !

7.3.1 Overloading ++ (and –)

```
class Digit{
//postfix: a++ : returns an rval.
Digit operator++(int){
    ... return digit;
}
//prefix: ++a : returns an lval. (original variable)
Digit& operator++(){
    ... return *this;
}
}
```

- Both definitions have something unique:
  1. & in the prefix definition, to ensure it returns an lvalue.
  2. (int) in the postfix definition is necessary to distinguish it from the prefix definition, as c++ doesn't support return-value-based overloading.
- Also note that though operator++(int) looks like ++a (prefix), it's actually for a++ (postfix).

7.4 Polymorphism

Polymorphism means a call to a member function (after resolution of overloads), can lead to different implementations being called, based on the type of object that invokes the function.

7.4.1 Static Linkage

```
class Shape {
   public:
      int area() {
         cout << "Parent class area :"...
      }
};
class Rectangle: public Shape {
   public:
      int area () {
         cout << "Rectangle class area :"...
```

```
    }
};
class Triangle: public Shape {
   public:
      int area () {
         cout << "Triangle class area :"...
      }
};
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
   shape = &rec;
   shape->area();
   shape = &tri;
   shape->area();
   //both calls print "Parent class area:..."
   return 0;
}
```

Without any prefixes in the functions defined in the derived classes (that are identifiable with functions in the base class), the compiler assumes that any calls to these functions from an object of type base*/base always needs the implementation from the parent class. This is known as static linkage or static resolution (of the function call) or early binding, as the implementation for the area function is fixed at runtime to that of the base class (for objects of or pointers to the base class).
Note that calls from `rec` or `tri` would have called their respective functions, not the base class' function.

7.4.2 Dynamic Linkage

- Prefixing function identifiers that are shared across derived and base classes with `virtual`, allows for dynamic linkage, or dynamic resolution or late binding.
- With the said prefix, the compiler identifies the right implementation to call by the contents of the object or pointer being used to invoke the function. This is polymorphism.
- Note that a public function may be overridden by a private function, making the function inaccessible from objects of the derived classes.

```
class Base{
    public:
    virtual int func(){
        cout<<"base"<<endl;
        return 0;
    }
};
class Derived: public Base{
    private:
    int func(){
        cout<<"derived, private"<<endl;
        return 1;
    }
};
int main(){
    Derived d;
    d.func();//compiler error: func is not accessible.
    Base &b = d;
    b.func();//compiles, prints "derived, private"=> allows accessing private function.
}
```

Pure Virtual Functions

If a function is always intended to be overridden in the derived classes and there's no meaningful definition in the base class, we can just declare the function in the base class and set it to zero to avoid compilation errors about no definition being found for the function in the base class.

```
class Shape{
    //virtual int area();
    //Above line compiles if there are no to area() from any objects of shape/derived classes.
    //In the presence of such calls, even if area() is defined in derived classes and their
    //objects are used to call area() (from the right pointer, or a pointer of type Shape*)
    //compilation errors ensue.
    //However,
    virtual int area()=0;//goes through compilation successfully.
};
```

Note:

- We say a function demonstrates polymorphism if we can use a pointer of the base class to access functions of different derived classes and have different implementations being used based on the derived class to which the object belongs.
- Without the virtual keyword, even if functions with identical identifiers (name and arguments considered) are declared in derived and base classes, polymorphism is not observed. Function calls from pointers of the base class' type call its own implementation, not that of derived classes.
- Once a function is declared as virtual in a class, it demonstrates polymorphism across all derived descendant classes, even without the virtual keyword being present intermediate classes.
- The `final` keyword can be used to throw a compilation error if a function that we don't want any base classes to override is overridden:

```
class Rectangle:public Shape{
    public:
    int area()final{
        cout<<"Rect Area"<<endl;
        return 0;
    }
};
//Now if a class called square attempts to override area, a compilation error is thrown.
```

- Using the final keyword in a non-virtual function (that was not declared to be virtual in any of the ancestors) throws a compilation error.
- **Object Slicing:** When an object of a derived class is assigned to an object (not a pointer) of a base class, only members inherited from the base class are kept and the others are discarded. Thus, all functions that may have been overridden are reverted to their definitions in the base class.

```
void printarea(Shape s){
    s.area();
}
void printareaReference(Shape &s){
    s.area();//NO slicing.
}
Shape s; Rect r;
s = r;//slicing.
Shape &s = r;
printarea(r);//slicing. prints "Shape area..."
printareaReference(r);//No slicing. prints "Rectangle area..."
```

  In JAVA, and other languages where each non-primitive variable is actually a reference, object slicing doesn't happen.

7.5 Data Abstraction and Encapsulation

The idea is to write classes with a well-defined boundary between:

1. Implementation: how the class works, the variables and functions it needs for its work.

2. Interface: the function calls and variables accessible to the users of the class.

Data abstraction allows:

- Implementation of a class to evolve without affecting code that uses it.
- Prohibiting users from possibly disturbing the state of the objects of the class, which may affect correctness of its functions. Ex. A user sets the `top` pointer inside a stack's implementation to the start of the array, without updating the length, which is non-zero. This results in a segmentation fault.

It is enforced using access specifiers. Data encapsulation is about bundling all the related data and functions that use it into one class, keeping as much implementation detail from the user as possible.

7.6 Abstract Classes a.k.a C++ Interfaces

An abstract class is a class with at least one **pure virtual function**. Such classes define an interface that all derived classes have to support (have an implementation of).

7.7 Notes from interview questions

- In multilevel inheritance (A-¿B-¿C), any function calls from an object of type C are linearly searched for up the hierarchy, and the first implementation is taken.
- Pointers of a parent type can hold a child, but child pointers being assigned to parent objects raises compilation errors.
- Pointers of a parent type can only access members (variables and functions) declared and declared public in the parent.
- When a derived class defines a function with the same name as some function its base class, all functions (even with different signatures) of the base class with the same name become inaccessible to objects.
- However, using a pointer of the base type to point to the object of the derived class, both the overridden method and the unoverridden overload of a method with the same name can be accessed.
- Or, using a scope resolution operator:

```
1         d.Base::fun(5);//goes through.
2         Base &b = d;
3         b.fun(5);//goes through.
4
```

Initializer Lists

- Initializer lists of a derived class can't include members of the base class. They need to be initialized using the contructor of the base class.

7.7.1 Destructors

- The default destructor of a base class is accessible to everyone.
- If it is overridden, it must be specified at least as protected.
- If the destructor is declared as protected, calls to `delete baseptr` throw compilation errors.
- Ideally, if `baseptr`s are intended to be used for polymorphism and we want to avoid `delete baseptr` calls, we should set the destructor to protected.
- Finall, if a the destructor is public, and `delete baseptr` is called, the destructor of any derived class object held in baseptr is not called, and the chain of destructors starts being called from that of the baseptr's class.

7.7.2 New and Delete

TODO

7.7.3 Copy Constructors

```
class Sample{
    int id;
    Sample(Sample &t)
    {
        id=t.id;
    }
};
//defines what do to do when:
Sample a,b;
a = b;//calls copy constructor of a.
```

- Used to intialize members of a newly created object by copying members of an already existing object.
- It takes a reference parameter of an object of the same class.
- This is known as copy initialization, a.k.a. member-wise initialization.
- If not defined explicitly by the programmer, the compiler defines it for us.
- The following definition of a copy constructor makes the object of this class uncopyable:

```
class Derived{
    Derived(const Derived&) = delete;
};
```

- Attempting to copy `Derived` objects in the code throws a compilation error.
- Hence, `throw d` also throws a compilation error, as `throw` implies copy operations. (See Exceptions section.)

Types of Copy Constructors

1. Default Copy Constructor: The implementation offered by the compiler which copies the bases and members of an object in the same order that a constructor would intialize the bases and members of the object.
2. User Defined Copy Constructor: needed when an object owns pointers or non-shareable references, such as to a file. A destructor and assignment operator should also ideally be written in this case to assist in transfer/destruction of said references.

A copy constructor is called when:

- An object of the class is **returned by value.**
- An object of the class is **passed by value** as an argument.
- An object is constructed based on another object of the same class. Ex: `Shape s1 = 1,2; Shape s2(s1)` or `Shape s2 = s1;`
- The compiler generates **a temporary object.**

Note that it's not called when a previously declared object is assigned another object. This calls the assignment operator.

```
Shape s1,s2;
s1 = {1,3};
Shape s3(s1);//calls copy constructor.
Shape s4 = s1;//calls copy constructor.
s2 = s1;//calls assignment operator.
```

Note that a call to the copy constructor is not guarranteed as the compiler performs optimizations like **return value optimization** and **copy elision** to avoid unnecessary copies where possible. (TODO)
Other points:

- Use a user-defined copy constructor when the default copy constructor results in a shallow copy (say, if some members are pointers). Deep-copy is only possible in a user-defined constructor.
- Copy constructors can be made private, and this makes objects of the class non-copyable. It's particularly useful (as a lazy technique to avoid shallow copies) if the class has pointers of dynamically allocated resources. The right way is to write a deep-copy-constructor and make it public.
- A copy constructor which takes the object argument by value leads to a compilation error, as at runtime it would have lead to an infinite chain of copy constructor calls.
- Use const in the argument to make sure:
  1. The source object isn't accidentally modified.
  2. The copy-constructor can be called with temporary objects created by the compiler, which can't be bound to non-const references.

```
        //if copy constructor doesn't say const Shape &s1,
        Shape s2 = fun();//fun returns s2 by value.
        //the above code throws a compilation error, at the last line.
        //If const is present, it compiles.

```

- In default constructors, default constructors of parents are called before those of derived classes, but, in copy constructors, the parent's default constructors (not copy constructors are called), unless the implementation of the derived class' copy constructor specifically calls their copy constructors.

Copy Elision (a.k.a. Copy Omission)

The compiler avoids making copies of objects (in pass by value/return by value scenarios) where possible.

7.8 From LearnCPP.com

7.8.1 14.4 Const objects

```
const Date today {2020, 10, 14};//valid.
const Date today = {2020, 10, 14};//valid.
//Note that const objects must be initialized, unless a default
//constructor is defined.
```

Objects that are declared with `const` keyword (as a local variable, or a function argument) impose certain restrictions (that upon violation lead to compiler errors.):

- Their members variables can't be changed, neither via direct access nor calls to member functions that change them.
- Additionally, const objects can't call non-const member functions. `const` before the definition body indicates that the member function doesn't modify the members of the class; it doesn't impose any restrictions on the returned value or aruments.

```
class Date{
    //can't be called by a const object, despite not
    //altering any variables in its definition:
    void print(){
        cout<<"non-const member function.";
    }
    //can be called by const objects:
    void print2() const {
        cout<<"const member function.";
    }
}
```

- If the declaration and definition are written separately, `const` must be present after the function signature in both places.

```
class Date{
    void print() const;
};
void Date::print()const{
    ...
}
```

- An attempt to modify the class inside a const function raises a compilation error, even inside unreachable if-blocks.
- Within the definition of a const member function, `this` is a const pointer to a const object.
- No constructor can be declared as a constant, as they need to modify the member variables, regardless of what their implementation says.
- It is perfectly fine to call const member functions from non-const objects.
- Functions can be overloaded based on whether they are const or not. So, const objects call the const variant while non-const objects call the non-const variant. This is usually done if constness changes the return value.

```
//These are valid overloads:
int fun(){

};
int fun()const{

};
//These are invalid as const keyword specifiers return type.
int fun(){}
const int fun(){}
//Also note that
const int fun(){};
//is exactly the same as:
int const fun(){};
//and the two are different from
int fun()const{};
```

7.8.2 14.6 Access functions

- Trivial member functions to access selectd private data members.
- Of two types: Setters (mutators) and getters (accessors).
- Getters are made const so they can be called on const objects while setters have to be non-const.
- For efficiency, getters can be written to return constant lvalue references, instead of returning by value:

```
const std::string &getName()const{return m_name;}
```

- Note that such functions' returned references become invalid the moment the object is destroyed. So, the references should not be stored (and accessed) beyond the lifetime of the object.

```
const std::string & ref = createEmployee().getName();
//we store the reference to a property in the rvalue implicit
//temporary object, created by the compiler.
//accessing ref later leads to undefined behaviour.
```

- References returned from functions to private members should be constant; otherwise, they permit direct modification of private members.

Ref-qualifier overloads

```
const std::string& getName() const & { return m_name; } // when called on an lvalue (single &), return member by reference
const std::string getName() const && { return m_name; } // when called on an rvalue (double &&), return member by value
// Alternately, we can disable the use of a member functions for rvalue objects
const std::string& getName2() const && = delete;         // when called on an rvalue, emit a compilation error
```

7.9 Friends

A friend is a class or function (member or non-member) that has been granted full access to the private and protected members of another class. Using friends, classes can selectively give full access to their members without unnecessary impacts.
The friendship is established by (declared in) the class removing its access control for some other entity.

7.9.1 Friend (non-member) Functions

With non-member functions, an object of the class must be accepted as an argument to access the relevant data.

```
class Shape{
    int area;
    friend double paintcost(const Shape& shape);
};
double paintcost(const Shape& shape){
    return shape.area*costunit;//accesses private member and compiles.
}
```

- Note that the friend function can also be defined inside the class, and remain a non-member function because of the `friend` keyword.
- A function can be a friend of multiple classes, all of which appear in its argument list. These are used when it makes less syntactic sense to make the function a member of either class.

```
friend void printWeather(const Temperature& temp, const Humidity& hum){
    //access private members of both temp and hum at once.
}
```

- Access specifiers make no difference to the availability of friend functions as they are non-members anyways.
- Friend functions should also use the class' interface where possible, instead of directly accessing data, as this insulates them from future change in the class.

7.9.2 Friend Classes and Member Functions

- Friend classes can access private and protected members of another class.

```
class Storage{
    //private members here.
    friend class Display;
    //Display declared as a friend of storage.
    //Display accesses all members of storage.
    //Note: no forward declaration required.
};
class Display{
    void print(const Storage& storage);
}
```

- Friendship is not reciprocal.
- Friendship is not transitive.
- Frienship is not inherited. Classes derived from a friend are not friends.

Friend Member Functions

- One point worth prattling about is that the compiler needs to have seen the declaration of a member function before it can be declared as a friend somewhere using the syntax below:

```
class Storage{
    //private members.
    friend void Display::displayStorage(Storage &storage);
    //compiler should have seen displayStorage in Display
    //before this line.
};
```

- This implies the compiler should have encounted the **full definition of the class** to which the member function belongs, not just a forward declaration.
- Additionally, to use members of the class where friendship is declared (`Storage`), it should have been declared before the definition of the friend member function.
- A better solution is of course to split the code up into separate files.

Chapter 8

C Stuff

8.1 Calloc vs Malloc

Malloc	Calloc
`(void*) malloc(NumBytes)`	`(void*) calloc(NumElems, ElemSizeInBytes)`
Stands for memory allocation.	Stands for clear memory allocation.
Returned memory block has garbage values.	Returned memory block is initialized to 0.
Faster than calloc.	Slower than malloc.
Doesn't involve memory overheads.	Involves some memory overheads.

Chapter 9

More C++ Stuff

9.1 Templates from LearnCpp.com

- Templates are declarations of functions of classes with placeholder types.
- Placeholder types aka, type template parameters, aka template types.
- A placeholder type is not known at the definition of the template, but is decided at a call to the template.
- Once a template is written, the compiler can use it to generate as many overloaded entities as needed, using concrete types.
- C++ allows three kinds of template parameters:
  1. Type template parameter: represents a type.
     - Use keywords `typename` or `class`.
     - `typename` can be replaced by any type, primitive or a class.
     - TODO: are the two equivalent?
  2. Non-type template paramater: represents a constexpr value.
  3. Template template paramater: represents a template.
- Templates are analogous to stencils.
- Templates can work with types that didn't even exist when they were written.
- It's common convention to use uppercase letters for template parameters.

9.1.1 Function Templates

- Example:

```
template<typename T>
T max(T x, T y){
    return (x<y)?y:x;
}
// signature is now: T max<T>(T x, T y);
```

- A function call looks like: `max<int>(a,b);`
- It is necessary to specify T in every call to `max<T>`.
- When the compiler encounters a call to `max<int>`, and it doesn't have a corresponding definition yet, it uses the template to create a function for the type selected.

- This is known as **(Function) (Template) Instantiation**.
- **Implicit instantiation** refers to instantiation due to a function call statement.
- Instantiated functions are called a **function instance** or **template function**.
- Function instances are normal functions in all regards.
- Function instantiation is a compiler task, and thus happens during compilation.

Template Argument Deduction

- If all the template type parameters appear in the function's argument, and the arguments we pass to the function have the right type (without the need for coercion), then we can omit the type specification in the function call statement:

```
max<int>(a,b);
max<>(a,b);//Compiler only studies templates for finding an overload.
max(a,b);//compiler first studies non-template declarations of max, followed by templates.
```

- The order of checking for overloads before templates allows for type-specific optimizations in generic functions.
- So, if the type parameters are deducible from the arguments and we want to use the type-specific, non-template overload of the function, use `max(a,b)`, but use `max<>(a,b)` if you want to use the template overload for some reason.
- For instantiated functions to successfully compile, it is necessary for their bodies to not invoke any functions or operators not defined for the types being inferred from the function calls.
- Note that the compiler generates instantiated functions so long as they make sense syntactically. It is our responsibility to ensure semantic sanity is maintained.
- We can disallow particular template overloads using **function template specialization** like so:

```
template <typename T>
T addOne(T x)
{
    return x + 1;
}

// Use function template specialization to tell the compiler that addOne(const char*) should emit a compilation error
template <>//empty type parameter list, NOT <char*>
const char* addOne(const char* x) = delete;
```

- Attempt to use deleted function instances raises compilation faults.
- It is common practice to put the entire template declaration and definition in header files, rather than just the declaration, as all cpp files need to know the template exists for their function calls to instantiate the relevant overloads from the template.
- Functions implicitly instantiated from templates are implicitly inline; inline functions can be defined in multiple files, so long as their definitions are identical.
- TODO: inline functions.
- The template themselves are not inline, which is a concept that only applies to functions and variables.
- Generic programming refers to the practice of using templates and associated template parameters (generic types).

- There are some drawbacks to template programming:
  - Code bloat and slow compile times: As the compiler instantiates a function for each call with a unique set of arguments, even compactly written templates cause the two problems.
  - Error messages from using templates are much more convoluted than regular error messages.

Multiple Template Types

- Note that a function call with different types being passed in place of T without explicitly mentioning `<T>` raises compilation errors.

```
template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
int main()
{
    std::cout << max(2, 3.5) << '\n';  // compile error
}
```

- The compiler doesn't know which overload to refer to and which argument to coerce.
- Type coercion is done only in function overloads, **not in template argument deduction**.
- Note about function overloading:
  - This compiles but isn't sound:

```
int mymax(int x,int y)...
mymax(2,2.5);//returns 2
mymax(2.1,2.5);//returns 2
```

  - This compiles and is sound:

```
double mymax(double x,double y)...
mymax(2,1);//returns 2. Note that double arguments are trivially sound.
mymax(2,2.5);//returns 2.5.
```

  - This doesn't compile:

```
int mymax(int x,int y)...
double mymax(double x,double y)...
mymax(2,2.5);//compiler error, ambiguous overload.
```

- There are three ways to allow this mixing of types in templates.
  1. Static Casting the arguments in a function call, without specifying the template type.
     - Must use static cast `static_cast<T>(var)` instead of dynamic casting `(type T)(var)`, which happens at runtime, because we need the type to be casted at compile time, to find the right template.
  2. Just provide the template type and avoid the need for template argument deduction.
  3. Declare function template types with multiple type parameters. (Discussed below.)
- Allow multiple types in the argument list and use auto to ensure arithmetic conversions don't lose data.

```
template <typename T, typename U>
auto max(T x, U y)
{
    return (x < y) ? y : x;
}
//now this works for (T,T) and (T,U) instances.
//C++ 20 introduces abbreviated function templates:
```

```
//Below is a shorthand for the above beast:
auto max(auto x, auto y)
{
    return (x < y) ? y : x;
}
//should be used only if x and y are allowed to be of different types.
```

9.1.2 Non-type Template Parameters

- Serve as placeholders for constexpr values passed in during a function call.
- A non-type template parameter can have be one of:
  1. An integral type
  2. An enumeration type
  3. `std::nullptr_t`
  4. A floating point type (C++ 20)
  5. A pointer or reference to:
     - An object
     - A function or member function.
     - A literal class type (C++ 20).
- Example of usage:

```
bitset<8> b;
```

- Syntax:

```
template <int N> // declare a non-type template parameter of type int named N
void print()
{
    std::cout << N << '\n'; // use value of N here
}
```

- On seeing a call to `print<5>()`, the compiler instantiates a print function with N=5.
- One application is in creating data structures whose size is fixed at compile time.
- One other application is in static asserts for values passed as non-type template parameters. For example, in a sqrt function.
- Type conversion of non-int parameters is often performed:

```
print<'c'>();//compiles with casting 'c' to int.
```

- Though only some types of constexpr conversions are allowed:
  - Integral promotions (ex. char to int).
  - Integral conversions (char to long or int to char).
  - User-defined conversions (some custom class to int).
  - Lval to Rval conversions.

9.2 Exceptions from LearnCpp.com

- The primary issue with error-handling without exceptions is that the error-handling code ends up intricately linked to the normal control flow of code; constraining both how the code is laid out and how errors can be reasonably handled.
- Exceptions allow decoupling of error-handling code from the regular flow of control.
- Three C++ keywords are used for this:
  1. `throw` It is followed by an object of any kind (int/double/class).
  2. `try` Encloses the block of code that can raise an error.
  3. `catch` Specifies the object to be expected if an error is thrown and what to do with it.
- Each try block can have many catch blocks, each expecting a different type of object to be thrown, but must have at least one.
- Execution resumes normally at the end of the catch block.
- Exceptions of non-primitive types should be caught in const references to avoid unnecessary copies.
- The variable name of a parameter can be excluded if it is not used inside the catch block. Though it's type must be specified nonetheless.
- The compiler searches for matching enclosing "catch" blocks for a thrown exception up the program, until it is caught. If no catch block is found, the program fails with an exception error.
- The compiler only ever casts derived classes of exceptions to their parents classes, but it never casts between primitive types.
- The catch block can serve one of four purposes:
  1. Convey the caught error to the user in a neat way, say, by printing to the console.
  2. Return a value or error code back to the current function's caller.
  3. Throw another exception, to be caught by an outer try-catch block.
  4. Terminate the program cleanly.

9.2.1 Stack unwinding

- If the current function doesn't have a try-catch block enclosing the thrown exception, the current searches down the call stack to find the first enclosing try-catch block.
- On finding said block, the stack unwinds (locals are popped off) until the control can resume in the stack's state of the function that contains said catch block.
- If no catch block is found all the way to main, the program terminates **without** unwounding the stack.
- This may lead to trouble if some locals have non-trivial destructors.
- The stack not being unwound allows retention of all the debug info related to the exception.
- Behold the catch-all handler, that catches all exceptions:

```
    try{
        //exception throwing code.
    }
    catch(int a){

    }
    catch(...){
        cout<<"All non-int exceptions lead here"<<endl;
    }
```

- The catch-all handler should be the last block in the catch block chain, to allow for specific error-handling as much as possible.

9.2.2 Exceptions within classes

- If a constructor of class A throws an exception, all the initialized members call their respective destructors, but A() is never called, as construction never succeeded.
- This can lead to missed cleanup for resources that had been allocatd in A() before the exception.
- However, if each resource that demands cleanup was declared inside a member of A, that had its own destructor with the requisite cleanup code, the cleanup would go through.
- However, creating a class to manage each resource, and be a member in A, isn't efficient.
- STL provides classes that have destructors for cleanup:
  - `std::unqiue_pointer<T>` : for pointers that are freed on destruction.

9.2.3 Classes for Exceptions

- Seeing as primitive types convey vague information about the error, it is preferable to define classes that are meant to be thrown around.
- A hierarchy of such classes is useful and often done.
- Catch blocks with an argument of a base class also catch all errors of derived classes without slicing, if the argument is a reference argument, and with slicing otherwise.
- If the base catch block precedes the derived catch block, the base catch block executes.
- Otherwise, the derived catch block executes.
- Hence, catch blocks for derived classes should be written before (above) those for base classes.
- All of the exceptions thrown by STL in C++ are of classes derived from `std::exception`.

```
catch(std::exception &e){
    cout<<e.what()<<endl;//prints the error description.
}
```

- Note that `e.what()` messages vary across compilers.
- A common derived class to use is `std::runtime_error("message for what")`.
- We can of course further extend the STL class hierarchy in our codebase.
- Rethrowing exceptions is a common practice.
- If an exception is caught by an argument that is of a class less-derived than the exception, the right way to rethrow the exception is `throw;` (an empty throw statement).
- A statement like `throw baseException;` slices the derived exception class.

9.2.4 Storing Exceptions

- When an exception is thrown, its data is initially on the local stack.
- If the stack needs to be unwound for it to be caught, the compiler copies its data to memory location reserved for exceptions off the stack, until a catch block is encountered.

9.2.5 Function Try Blocks

- One use case of function try blocks is when we want to catch an exception thrown by the constructor of a base class, in the constructor of the derived class.

```
class B : public A
{
public:
    B(int x) try : A{x} // note addition of try keyword here
    {
        //usual constructor stuff
    }
    catch (...) // note this is at same level of indentation as the function itself
    {
            // Exceptions from member initializer list or constructor body are caught here
            std::cerr << "Exception caught\n";
            throw; // rethrow the existing exception
    }
};
```

- Note that function level catch blocks **for constructors** must throw an exception. They are not allowed to resolve and just consume exceptions.
- Implicit rethrows are called at the end of the catch-block, hence it is better to generally avoid letting control get to the end of these catch blocks.

Function type	Can resolve exceptions via return statement	Behavior at end of catch block
Constructor	No, must throw or rethrow	Implicit rethrow
Destructor	Yes	Implicit rethrow
Non-value returning function	Yes	Resolve exception
Value-returning function	Yes	Undefined behavior

- We shouldn't use the catch block to clean up resources of a failed constructor, as the object was never created if an exception was thrown.

9.2.6 Common Design Mistakes

- Cleanup calls to resources established and used inside a try block should be outside the catch block, so they're called in normal execution, and in exceptional execution. This is assuming that control exits the catch block.

```
try
{
    openFile(filename);
    writeFile(filename, data);
}
catch (const FileException& exception)
{
    std::cerr << "Failed to write to file: " << exception.what() << '\n';
}
// Make sure file is closed
closeFile(filename);
```

- Additionally, the variables must be in scope for the cleanup calls, and must be declared outside the try block.

- Or as discussed earlier, we can uses STL classes like `unique_pointer<T>` or other resource classes with destructors that perform the cleanup.
- Generally abstain from using exceptions in destructors as an exception thrown during stack unwounding "confuses" the compiler between continuing the unwound or handling the exception. Instead, write the error to a log file.
- It should be noted that exceptions come with performance costs such as:
  1. Mainly: the expensive operation of unwinding the stack on an exception and finding the suitable catch block.
  2. An heavier executable.
  3. The program runs slower due to additional checks.
- Some compilers implement **zero-cost exceptions**, where the runtime cost of exception-handling is zero for a no-exception scenario, but incurs large penalties in exceptional scenarios.

9.2.7 Exception Specifications

- Language mechanism to document what kind of exceptions a function can throw, in its declaration.
- The `noexcept` keyword in a function declaration guarrantees that the function won't throw an exception.

```
void doSomething() noexcept{
    //body that can't throw any exception out.
}
```

- If a `noexcept` function is overridden, the overriding function should also be declared as `noexcept`.
- If at runtime, an exception is emitted from a `noexcept` function, `std::terminate()` is called immediately, even if there exist handlers around the function call.
- The promise of `noexcept` is contractual; it is not enforced by the compiler.
- Functions differing only in there exception values (like those differing only in their return values) can not be overloads of each other; they are all the same.
- The noexcept operator can be used to store variables conveying whether a function or expression can potentially throw errors or not:

```
void foo() {throw -1;}
void boo() {};
void goo() noexcept {};
struct S{};
constexpr bool b1{ noexcept(5 + 3) }; // true; ints are non-throwing
constexpr bool b2{ noexcept(foo()) }; // false; foo() throws an exception
constexpr bool b3{ noexcept(boo()) }; // false; boo() is implicitly noexcept(false)
constexpr bool b4{ noexcept(goo()) }; // true; goo() is explicitly noexcept(true)
constexpr bool b5{ noexcept(S{}) };   // true; a struct's default constructor is noexcept
    by default
```

9.3 The Inlined

9.3.1 Global Constants

- Some constants might be used across files and we want to define them only once.
- One way of doing this (pre C++17) was:

```cpp
#ifndef CONSTANTS_H
#define CONSTANTS_H
// define your own namespace to hold constants
namespace constants
{
    // constants have internal linkage by default
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double myGravity { 9.2 };
}
#endif
```

- And we `#include<constants.h>` wherever we need the constants.
- Each separate object file that uses `constants.h` gets a separate copy of the constants, but the compiler removes them away later.
- Downsides of this:
  1. Each object file using `constants.h` has to be recompiled whenever the constants are updated.
  2. If the constants aren't optimized away by the compiler for some reason and are large, they use a lot of memory.
- The alternative is to:
  1. Put forward declarations of the constants in a header file that can be included in multiple files.
  2. Define them only once in a .cpp file.
- Note: `constexpr` variables can't be forward declared, as the compiler must know their values at compile time. We use `const` in this case:

```cpp
/*constants.cpp contents:*/
#include "constants.h"

namespace constants
{
    // actual global variables
    extern const double pi { 3.14159 };
    extern const double avogadro { 6.0221413e23 };
    extern const double myGravity { 9.2 }; // m/s^2
}

/*constants.h contents*/
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace constants
{
    // since the actual variables are inside a namespace, the forward declarations need
    to be inside a namespace as well
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}
#endif
```

- Downside of this method is that the values of `constants::pi` and the others aren't treated as constants in files other than `constants.cpp`. This keeps several helpful optimizations from happening.
- `inline` variables are a better solution from `C++ 17`
- An inline variable, declared a single header file, can be used in multiple cpp files without generating redundant copies.
- Inline global variables have external linkage by default.

9.4 Unions

Cpp Reference Page

9.5 Misc. (Courtesy of Sony)

- `cout.width(len)` prefixes the next output (with whitespace) sent to it so that its length is equal to len
- If len ¡ output.size(), the function call has no effect.
- Size of objects of a union class is the maximum size of any one field.

Chapter 10

DSA Revision Notes

10.1 Stacks, Queues, Linked Lists

- Implementation of stack, queue, deque usually involve maintaining a fixed-size array and a pointer to the top/front/back of the queue that moves around.
- The growth strategies for a stack's underlying array are:
  - Growth: f(N) = 2N;
  - Tight: f(N) = N+c;
- Push operation that call growing the underlying array cost f(N)+N+1 units.

10.1.1 Queues

- A common use of queues (or deques) is to explore a search space in steps, for example BFS.
- Each step involves expanding the frontier of the space by one unit, while popping the previous frontier from the queue.

10.2 Hash Tables

- Load factor $\alpha = n/m$, where the hash table has m slots (unique indices) and holds n elements. α = average number of elements in each slot.
- A hash function maps the keys of a hash table to its indices.
- Hash functions are the composition of:
  1. Hash codemap: $KeySet \Longrightarrow Z$
  2. Compression map: $Z \Longrightarrow [0, 1, ...N-1]$
- Both have to deterministic and dependent on the key.
- Strings are usually hashed using polynomials, with the characters as coefficients. Common points of evaluation of polynomials are x = 33, 37, 39 or 41.
- Some common compression maps: beginitemize
- $h(z) = z\%m$, where m is prime (not too close to powers of 2.) m=b^e makes for a bad compression map. $x\%2^e$ gives last e bits of x.
- $h(z) = floor(m(fractional(zA)))$, where A $\in$ (0,1) and m = size of the table.

- Often use m = 2^p for above.
- Fibonacci hashing involves using A = $\frac{\sqrt{5}-1}{2}$
- $h(z) = |ak + b| mod m$ where a,N are coprime.

10.2.1 Universal Hashing

- A solution to the idea of adversarial choices of elements for a determined hash function.
- A collection H of hash functions is universal if
  For a randomly chosen h ∈ H and two keys k and l,
  $P(h(k) = h(l)) < 1/m$

10.2.2 Collision Resolution

Chaining

- Each key is mapped to a linked list with elements that share the key.
- Chaining is the most time efficient collision resolution method.
- Linked lists can be sorted if necessary.
- Search time = O(α).

Open Addressing

- Allows storage of at most m elements.
- Introduce NULL elements. Use an array of size m, initialized to NULLs.
- Systematically probe slots when searching for an element.
- Modify the hash function to take the probe number i as the second parameter.
  $h :\Longrightarrow Keyset \times 0, 1, 2...m - 1$
- Hash function determines the sequence of slots to be examined for a given key.
- $h(k, 0), h(k, 1)...h(k, m - 1)$ is a permutation of $0, 1, ...m - 1$
- Variations of probing include:
  1. Linear probing
     - if h(k) is used, find next empty slot for insertion.
     - Use a tombstone marker to delete elements, so later search queries can read the tombstone and check further elements.
     - For deletion, search all next slots where h(slot value) = h(k) or slot value == tombstone.
     - Rehash if there are too many tombstones.
     - Inserts are allowed to use tombstones.
  2. Double hashing
     - set h1(k) = initprobe and h2(k) = offset.
     - Search for empty slots starting at initprobe, in offsets of h2(k) mod m.
     - Avoid functions h2(k) which can be give zero.
     - If m is prime, this method examines every slot in the table.
     - Expected number of probes to find an empty slot = $\frac{1}{1-\alpha}$ (Or an unsuccessful search.)

10.3 Trees

- In a tree, degree of a node may refer to the number of children it has.
- **Binary Tree**: An ordered tree with each node having at most two children.
- **Complete Binary Tree**: Each level i has 2^i nodes, i = 0, 1, ... h.
  - # Leaves = $l = 2^h$; Total nodes = n = $2^{h+1} - 1$; # Internal nodes = m = $2^h - 1 = l - 1$
  - height = h = $log_2(l) = log_2((n+1)/2)$
- If a binary tree has n nodes,

$$n \leq 2^{h+1} - 1 \implies n - 1 \leq h \leq log_2((n+1)/2)$$

$$1 \leq l \leq m + 1, l = n - m \implies l \leq \frac{(n+1)}{2}$$

10.3.1 Binary Search Tree

- Structure used for ordered dictionaries, where the keys are stored in a binary tree.

```
Tree search(Tree t,Value v){
    if(t->value==v){
        return t;
    }
    else if(t->value < v){
        if(t->left){
            return search(t->left,v);
        }
        return NULL;
    }
    else{
        //t->value > v
        if(t->right){
            return search(t->right,v);
        }
        return NULL;
    }
}
```

- Inorder traversal of a BST gives a sorted projection of all the keys. This is called BST-sort.
- Has the same complexity cases as quick-sort:
  - WCS: O(n^2)
  - BCS: O($nlogn$)
  - Average over $n!$ permutations: O($nlogn$)
- Method implementations:
  1. Replace: Given x,y and parent(x), replace x by y as a child of parent(x).

```
void replace(Node*x, Node*y, Node*px){
    if(px->left==x){
        px->left = y;
    }
    else{
        //(px->right==x)
        px->right = y;
    }
}
```

  2. Search: as given above.

3. Insert: search for node in tree, insert as a child where NULL is found.
4. Successor:

```
Node *n successor(Node *x, Node *n){
    if(x->right){
        return leftMostChild(x->right);
    }
    else{
        Node *y = parent(x,n);
        while(y!=NULL && x==y->right){
            x = y;
            y = parent(x,n);
        }
        //y == root or y is closest ancestor such that x is in y->right.
        return y;
    }
}
```

5. Deletion:

```
void delete(Node *x, Node *n){
    if(!(x->left&&x->right)){
        //x has at most one child.
        Node *y = parent(x,n);
        if(y->left==x){
            y->left = oneChildOf(x);//may return null if x has no children.
        }
        else if(y->right==x){
            y->right = oneChildOf(x);
        }
    }
    else{
        //x has two children.
        Node *s = Successor(x,n);
        delete(s,n);
        Node *y = parent(x,n);
        //transfer the children
        s->right = x->right;s->left = x->left;
        if(y->left==x){
            y->left = s;
        }
        else if(y->right==x){
            y->right = s;
        }
    }
}
```

10.3.2 Traversals

1. Pre-order: Do me pre my children; process each node before its children are processed.

```
void preorder(Node n){
    process(n);
    for(Node &u:n.children){
        preorder(u);
    }
}
```

   - Used in reading documents, webpages.

2. Post-order: Do me post my children; process each node after its children are processed.

```
void postorder(Node n){
    for(Node &u:n.children){
        postorder(u);
    }
    process(n);
}
```

- Used by the disk-usage command and evaluation of arithmetic expressions.

3. In-order: Do me between my children; process left children first, followed by the given node, followed by the right children.

```
void inorder(Node n){
    for(Node &u:n.leftchildren){
        inorder(u);
    }
    process(n);
    for(Node &u:n.leftchildren){
        inorder(u);
    }
}
```

4. Eulerian: Generic traversal combining pre-, post- and in-order traversals.

```
void eulerian(Node n){
    process(n);//First process
    for(Node &u:n.leftchildren){
        eulerian(u);
    }
    process(n);//Second process
    for(Node &u:n.leftchildren){
        eulerian(u);
    }
    process(n);//Third process.
}
```

10.3.3 AVL Trees

- They are aka height-balanced binary search trees.
- If $h(n)$ denotes the height of the tree rooted at n:

$$h(n) = 1 + max(h(n \to left), h(n \to right))$$
$$h(n) = 1 \implies \text{n is a leaf.}$$

- Define the balance factor of a node as:

$$BF(n) = h(n \to right) - h(n \to left)$$

- $\forall n \in \{InternalNodes\}, |h(n \to right) - h(n \to left)| \leq 1$
- $h(root) \leq log_{\phi}(n)$
- If the leaf closest to the root is at level k:
  1. $h(root) \leq 2k - 1$
  2. All nodes at levels 1...k-2 have 2 children.
  3. 2 $\implies$ n $\geq 2^k - 1$
  4. 1, 3 $\implies$ $2^{k-1} \leq n \leq 2^{2k-1}$
  5. 4 $\implies$ $2^{(h-1)/2} \leq n \leq 2^h$
- In an AVL tree of height h,
  1. The leaf closest to the root is at level $\geq (h+1)/2$.
  2. On the first $(h-1)/2$ levels, AVL tree is complete (full).
  3. $2^{(h-1)/2} \leq n \leq 2^h$

- Method implementations:
  1. Update: (private) method to update height values and balance factor values across the tree (or subtree passed as argument.)
  2. Rotaion: operation used in insertion and deletion for balance restoration.

     – Either structure being a valid BST implies the other is a valid BST.

```
void rotate(Node* u,Node* v){
    //u is the parent to v.
    Node *p = parent(u);
    if(u->left==v){
        u->left = v->right;
        v->right = u;
        if(p->left==u){
            p->left = v;
        }
        else{
            p->right = v;
        }
    }
    else{
        u->right = v->left;
        v->left = u;
        if(p->left==u){
            p->left = v;
        }
        else{
            p->right = v;
        }
    }
}
```

  3. Insertion: O(logn)

```
void insert(Node *n, Value x){
    PathTraversed path = bstInsert(n,x);
    //path.last = x's node.
    //find closest ancestor of x that is imbalanced.
    //imbalance is possible because of a height increment.
    Node *top, *mid, *bot;
    bot = path.back();//x's node.
    mid = parent(bot);
    top = parent(mid);
    //top will point to the imbalanced node.
    //top->mid->bot belongs to the path.
    while(balanced(top) && parent(top)!=top){
        //root check condition.
        path.pop_back();
        back = mid;
        mid = top;
        top = parent(top);
    }
    if(!balanced(top)){
        if(BF(top)*BF(mid)>=0){
            //left-left or right-right cases.
            rotate(top,mid);
            //balanced.
        }
```

```
        else if(BF(top)*BF(mid)<0){
            //left-right or right-left cases
            rotate(mid,bot);
            rotate(top,bot);
            //balanced.
        }
    }
}
```

4. Deletion: O(logn) We follow a similar strategy to deletion in BSTs, followed by recursive balancing.

```
void delete(Node*n, Node* x){
    Node *z = parent(x,n);
    //if its the root, the pointer is modified.
    if(!(x->left && x->right)){
        replace(x,oneChildOf(x),z);
    }
    else{
        //x has two children.
        Node *s = successor(n,x);
        delete(n,s);
        replace(x,s,parent(x));
        z = parent(s,n);
    }
    while(parent(z)!=z){
        while(balanced(z) && parent(z)!=z){
            //find first imbalanced ancestor until root.
            z = parent(z);
        }
        y = childWithLargerHeight(z);
        x = childWithLargerHeight(y);
        if(BF(z)*BF(y)>=0){
            rotate(z,y);
        }
        else{
            rotate(y,x);
            rotate(z,x);
        }
    }
}
```

   - Insertion is quicker because there's only one point where rebalancing is necessary, whereas in deletion, the imbalance may be transferred higher up the tree.

10.4 String Stuff

The common problem of finding a pattern P in a text T (—T— ¿ —P—) can be solved efficiently by:

1. Either preprocessing the pattern (KMP)
2. Or preprocessing the text (Tries)

The latter is preferred if multiple patterns are searched for a in a single piece of text.

10.4.1 KMP

```
vector<int> kmp(string t, string p){
    int l = p.size();
    int n = t.size();
    //this lps implementation is inefficient. O(l^3).
    vector<int> lps(l,0);//lps[0,1,...l-1].
    //lps[0] = 0.
    for(int i=2;i<=l;i++){
        for(int j=1;j<i;j++){
```

```
            if(p.substr(0,j)==p.substr(i-j,j) && (i==l || (p[j]!=p[i]))){
                lps[i-1] = max(lps[i-1],j);
            }
        }
    }
    int j = 0, i=0;
    vector<int> matches = {};
    for(;i<l && j<n;){
        if(t[j]!=p[i]){
            //mismatch
            if(i==0){
                j+=1;
            }
            else{
                i = lps[i-1];
            }
        }
        else{
            //match=>increment both.
            i++;
            j++;
            if(i==l){
                i = lps[i-1];
                matches.push_back(j-l);
            }
        }
    }
    return matches;
}
```

Linear Time LPS

It's complicated man. Maybe do this later.

10.4.2 Tries

- Tries are meant to store data indexed by strings from a fixed set of characters.
- Said data can be the occurrences of the index string in a large body of text.
- A generic implementation is give below:

```
    class Trie{
        int charindex(char c);//each char mapped to an integer,
        vector<Trie*> chars = vector<Trie*>(numChars,NULL);//integers indexing a Trie
    pointer,
        Data data;//initialized to empty.
        void insert(string index,Data data){
            int i = 0;
            Trie* t = this;
            while(i<index.size()){
                if(!t->chars[charindex(index[i])]){
                    t->chars[charindex(index[i])] = new Trie;
                }
                t = t->chars[charindex(index[i])];
                i++;
            }
            t->data = data;
        }
        Data* get(string index){
            int i = 0;
            Trie* t = this;
            while(i<index.size()){
                if(!t->chars[charindex(index[i])]){
                    return NULL;
                }
```

```
                t = t->chars[charindex(index[i])];
                i++;
            }
            return &(t->data);
        }
        void remove(string index){
            int i = 0;
            Trie* t = this;
            while(i<index.size()){
                if(!t->chars[charindex(index[i])]){
                    return;
                }
                t = t->chars[charindex(index[i])];
                i++;
            }
            return t->data = NULL;
        }
    };

```

- This implementation's space complexity is O(—W—J), where J is the number of unique ordered pairs (i,c) such that data has been indexed using a string which has c at index i. W is the character set.
- A more space-efficient, and thus access time-inefficient implementation would be to use maps from chars to Trie pointers.
- A suffix Trie can be constructed in O(—T—) time, which means pattern matching can be done in O(—P—+—T—+k) time, where k is the number of times the pattern occurs in T.

10.5 Heap

- Implemented as a binary tree to order items by a parameter.
- All levels except the last one are full. The last level is left-filled.
- Structural Property: ∀node, Priority(node)≤Priority(node.parent)
- $\implies$ the root has the maximum priority.
- Height of a heap with n nodes is $h = floor(log_2(n+1))$

10.5.1 Array Implementation

```
vector<int> a(vals.begin(),vals.end());
//zero indexed.
int parent(i){
    return (i-1)/2;
    //parent(0) returns 0.
}
int left(i){
    return 2*i + 1;
}
int right(i){
    return 2*i + 2;
}
//condition: for all i, a[i] <= a[parent(i)]
```

10.5.2 UpwardsHeapify

```
void upwardsHeapify(int i){
    while(i!=0){
        //MAX HEAP => parent should be >. Swap if not.
        if(a[parent(i)] < a[i]){
            swap(a[parent(i)],a[i]);
        }
        i = parent(i);
    }
}
```

10.5.3 Insertion

```
void insert(int v){
    a.push_back(v);
    upwardsHeapify(a.size()-1);
    return;
}
```

10.5.4 Heapify

- Function used to convert any given binary tree into a heap.

```
heapifyNode(int i){
    int vl=INT_MIN,vr=INT_MIN;
    if(left(i)<n){
        vl = a[left(i)];
    }
    if(right(i)<n){
        vr = a[right(i)];
    }
    if(a[i] < max(vl,vr)){
        if(a[left(i)] > a[right(i)]){
            swap(a[i],a[left(i)]);
            heapifyNode(a[left(i)]);//continues down the tree.
        }
        else{
            swap(a[i],a[right(i)]);
            heapifyNode(a[right(i)]);//continues down the tree.
        }
    }
}
heapifyTree(int i){
    if(i<0 || i > n-1){return;}
    heapifyTree(left(i));
    heapifyTree(right(i));
    heapifyNode(i);
}
```

- Note that heapify continues down the tree, because the branch whose head is swapped with the given node is not guarranteed to be a heap with the new node.
- `heapifyNode` takes O(logn) time.
- `heapifyTree` takes O(nlogn) time, since it visits each node once, calling `heapifyNode`.

10.5.5 Deletion (of root)

```
void deleteMin(){
    arr[0] = arr[arr.size()-1];
    heapifyNode(0);//Note: deletion is O(logn)
    return;
}
```

10.5.6 Inplace Heapsort

- Works in O(nlogn) time and O(1) space.
- The key idea is to implement max heap helper functions and keep swapping a[0] with a[n-1], reducing n each time.

```
void inplaceheapsort(vector<int>& nums){
    int n = nums.size();
    vector<int> ans;
    maxHeapifyTree(nums,0,n);
    while(--n){
        swap(nums[0],nums[n]);//max elem moved to end of array repeatedly.
        maxHeapifyNode(nums,0,n);
    }
    return;
}
```

10.5.7 Other modifications

- If an element in a max heap is incremented, we must call `upwardsHeapify` to ensure the array remains a heap; if it is decremented, we must call `heapify`. Vice-versa for elements in min heaps.

10.5.8 Array to Heap in O(n)

- `heapifyTree(0)` and repeated insertions run in O(nlogn) time.
- A linear solution is below:

```
//Note: only first half of the elems are used in the loop
for(int i=n/2;i>-1;i--){
    heapifyNode(i);
}
```

- The running time is linear because the heapify calls incur low costs for of the nodes.
- In particular, there are at most $n/2^i$ nodes that incur i swaps (or operations) in heapify.
- The sum: $\sum_{i=1}^{logn} n \times i/2^i$ is strictly bounded by $2n$

10.6 Quick Sort

- Involves two procedures: Partition and Quicksort
- Partion takes an array, picks a pivot element in it, and rearranges the elements such that:
  - A prefix of the array contains all elements less than (or equal to) the pivot.
  - A suffix of the array contains all elements greater than (or equal to) the pivot.
- It returns the first index of a valid suffix where all values are $\geq$ the chosen pivot.

```
int partition(vector<int> &a, int s, int e){
    //s,e inclusive.
    int pivot = a[pivotCriteria()];
    int i = s-1, j = e+1;//outer points to allow do-while
    while(i<j){
        do{j--;}
        while(j>s-1 && a[j]>=pivot);//note the equality.
        do{i++;}
        while(i<e+1 && a[i]<=pivot);//note the equality.
        if(i<j){
            swap(a[i],a[j]);
        }
    }
    return i;//first index of a valid suffix.
}
```

- Quicksort partitions the array into two arrays, and makes recursive calls on each partition.

```
void quicksort(vector<int> &a, int s, int e){
    if(e<=s){return;}
    int q = partition(a,s,e);
    //first index of suffix.
    quicksort(a,s,q-1);//sort the prefix
    quicksort(a,q,e);//sort the suffix.
}
```

10.6.1 Quick Select

- The partition function splits the array into two parts whose elements have an upper (for prefix) and lower (for suffix) bounds on their ranks in the sorted array.
- This allows us to use it to find the kth (smallest) element in an array. The following algorithm is called quick select:

```
vector<int> a;
int partition(int s, int e){
    int pivot = a[e];
    int i,j;
    i = s-1;
    j = e;
    while(i<j){
        do{j--;}
        while(j>s-1 && a[j]>=pivot);
        do{i++;}
        while(i<e && a[i]<=pivot);
        if(i<j){
            swap(a[i],a[j]);
        }
    }
    j++;
    swap(a[e],a[j]);
    return j;
    //return the start of the suffix, ensuring nums[suffixStart] = pivot.
}
int qselect(int s, int e, int k){
    int q;
    q = partition(s,e);
    if(q<k){
        return qselect(q+1,e,k);
    }
    else if(q>k){
        return qselect(s,q-1,k);
    }
    else{
        return a[q];
    }
}
```

Chapter 11

Operating Systems

Find separate notes here.

Chapter 12

Networks

Find separate notes here.

Chapter 13

DBMS

13.1 SQL

- Notation:
  - `T1, T2, ...`: Data types
  - `A1, A2, ...`: Column names and `A`: entire set of columns.
  - `a1, a2, ...`: An example of values for A1, A2,... and `a`: a set of values of each column.
  - `R1, R2, ...`: Table (relation) names.
  - `E1(A), E2(B), ...`: Expressions using column names from `A` and `B`.
  - `C1, C2, ...`: Condition statements.
  - `abc|def`: either abc or def can be used here.
- Select statements:

```
select (A1, A2, A3) from R1 where C1;
```

Listing 13.1: The Usual

  - Examples of C1:

```
A1 = a1
A1 > a1
A1 < a1
A1 <= a1
A1 >= a1
A1 <> a1
A1 LIKE pattern
A1 in (a1,a1',a1'')
A1 between a1 and a2
A1 IS NULL
```

  - Each condition operand can be preceeded by NOT to negate it. Ex: `A NOT LIKE pattern`.
  - In patterns, _ replaces 1 character and `%` replaces 0, 1 or more characters.

```
select (A1, A2, A3) from R1 order by A1,A2 desc|asc;
select distinct (A1, A2, A3) from R1 where E1(A);
select (A1, A2, A3) from R1 limit n offset k;//skip first k rows and return the next n
    rows.
select A1, aggregate(A2) from R1 group by A1;
select A1, aggregate(A2) from R1 group by A1 having E1(aggregate(A2));
```

Listing 13.2: Variations of Select

- Joins:
  1. `(Inner) Join`: return rows where both tables have an entry for the common column (ignore matchless columns from both tables).
  2. `Left (outer) Join`: return all rows from table 1 (on the left), and if the row's column has a matching row in table 2, append its data too.
  3. `Right (Outer) Join`: return all rows from table 2 (on the right), and if the row's column has a matching row in table 1, append its data too.
  4. `Full (Outer) Join`: return all rows from both tables, correlating the ones with matching entries in the common column.
  5. cross join: every row is joined with every row. This is a costly operation and is rarely used. It is invoked in the following query:

```
select A1,A2 from R1,R2;
\\equivalent to
select A1,A2 from R1 cross join R2;
```

  6. The generic join sytax is:

```
select * from R1 (join-specification) R2 on C;
```

- Set operations can be used to combine the results from two separate select queries:

```
select * from R1 UNION select * from R2;
select * from R1 INTERSECT select * from R2;
select * from R1 MINUS select * from R2;
```

Listing 13.3: Set Operation

Chapter 14

Javascript Notes from InterviewBit

14.1 Data Types, typeof and instanceof from W3Schools

- There are five different data types to hold values:
  1. string
  2. number
  3. boolean
  4. object
  5. function
- There are 6 types of objects:
  1. Object
  2. Date
  3. Array
  4. String
  5. Number
  6. Boolean
- There are 2 data types that cannot contain values:
  1. null (typeof returns object though)
  2. undefined
- The `typeof` operator returns the data type of a variable.

```
// Returns "string"
typeof "John"
//Returns number:
typeof 3.14
typeof NaN
// Returns "boolean:"
typeof false
// Returns "object:"
typeof [1,2,3,4]
typeof {name:'John', age:34}
typeof new Date()
typeof null
// Returns "function:"
typeof function () {}
//Returns "undefined:"
typeof myCar //variable not yet defined.
typeof undefined//keyword undefined.
```

- `typeof` can't be used to determine if an object is an array or a date.
- Primitive Data: single simple data value with no additional properties or methods. typeof returns one of "string", "number", "boolean" or "undefined".
- Non-primitive, aka Reference Data: typeof returns "function" or "object".
- To check if an object is an array (and similarly for Date):

```
function isArray(myArray) {
  return myArray.constructor === Array;
}
```

- Empty Values: need not be undefined. Ex. an empty sring has both a legal value and a type.
- undefined and null are equal in value but different in type.

```
null === undefined // false
null == undefined // true
```

- `instanceof` operator returns true if an object is an instance of the specified object.
- Note some peculiarites below:

```
const cars = ['a','b','c'];
cars instanceof Array; //true
cars instanceof Object; //true
cars instanceof string[];
/*Syntax error, rhs of instanceof is not an Object*/
let a = cars[0];
a instanceof string;
/*Syntax error, rhs of instanceof is not an Object*/
a instanceof String;//false
//a has primitive type string, not String.
a = Object(cars[0])
a instanceof Object;//true
a instanceof String;//true

//Additionally
a = {c:'b'};
typeof a; // returns object.
a instanceof Object;//return true
```

- `void` operator takes a piece of code and ensures voids the return value (to return undefined)

14.2 Notes from Questions

- Initially used for building dynamic web pages.
- Hoisting: all variable and function declarations are moved to the top of the code by the interpreter. This means order of declaration is irrelevant.
- `"user strict"`, with quotes at the start of a file turns off hoisting.
- `==` compares values of two variables while `===` compares values and types.
- var vs let:
  1. var variables are hoisted, let variables are not.
  2. var variables have function scope, let variables have block scope.
  3. let variables are closer to other programming languages in terms of the first two points.
- Implicit type coercion: when data types of two operands don't match, implicit type coercion takes place.

1. String coercion: 3+"3" = "33";
2. Number coercion: 3-"3" = 0, because there's no string operator corresponding to -.
3. Boolean coercion: takes place when using logical operators, ternary operators, if statments and loop checks. Involves understanding:
   (a) Falsy values: values coerced to false. The exhaustive set is false, 0, 0n, -0, "", null, undefined, NaN.
   (b) Truthy values: values not coerced to false.
4. Equality coercion: happens with == operator. Both variales are converted to the same type and their values are compared. No coercion happens with ===.

- Logical operators (|| and &&) return one of the operands, not true or false, based on which is truthy and falsy, following lazy evaluation.
- It's a dynamically typed language, so variable types are checked during runtime and can change over runtime. Statically typed languages have type checks during compile time.
- `NaN` indicates a value that is not a legal number.
  - typeof NaN returns number.
  - use isNaN(x) to check if x is NaN.
  - only non-numeric and non-empty strings (empty $\Longrightarrow$ 0), undefined and NaN return true for isNaN()
  - Even '131n' return true for isNaN.
- Mutability: the property of a variable to be modified without using the assignment operator.
- Primitive data types are immutable.
- Non-primitive data types, aka reference data types, are mutable. These include Objects and Functions.
- All `objects` are on the heaps, and the variables that hold them are actually pointers that reside on the stack.

```
const staff2 = staff;//where staff is an Object.
//staff2 as a pointer holds the same address as staff, so:
staff2.name = '';//also modifies staff.
```

- To copy an object, use

```
const staff2 = Object.assign({},staff);//cast if necessary, like Array.from...
//or
const staff2 = {...staff};
```

- Note that both ... and Object.assign only create shallow copies; the reference properties of the objects inside are still copied by reference:

```
let b = [1,2,3];
let d = {j:b};
d.j.push(2);//b is modified too.
let c = {...d};
c.j.push(2);//b is modified too.
let e = Object.assign({},d);
e.j.push(2);//b is modified too.
```

```
let b = [1,2,3];
let d = {j:b};
```

- References in javascript:
  - Primitive data types are passed by value, non-primitive data types are passed by reference.

- Objects can be made immutable by using:
  1. `Object.preventExtensions(staff)`: disallows adding new properties.
     - Properties can still be deleted or modified.
     - New property definitions using '.' and assignment fail silently.
     - New property definitions using `Object.defineProperty` throw a TypeError.
  2. `Object.seal(staff)`: Only permits modifying existing properties.
     - Deletion and addition of properties is disallowed.
     - `Object.isSealed(staff)` used to check if an object is sealed.
     - '.' operations fail silently
     - `defineProperty` for new properties throw a TypeError.
     - Deletions fail silently.
  3. `Object.freeze(staff)` disallows all modifications, including deletion and insertion of properties.
     - Check using `Object.isFrozen(staff)`
     - Same rules as errors/silent failing as above.
     - It's not a recursize freeze. Properties of reference properties of a frozen object can still be modified. A recursive deep-freeze function may be written to avoid this fall.
- Variables declared using `const` may be mutable or immutable; but they can't be assigned to later.

14.3 Anti-patterns

Summarized from blogs one and two.

- Array operations:
  - Use `map` only to create a new array from a given array.
  - Use `for-loops` only for sequential element processing.
  - Use `forEach` only for independent element processing.
- `let, const and var`:
  - `let` and `const` variables are blockscoped, while `var` are not. Prefer using block-scoped variables as they reduce the chances of bugs related to shadowing.
  - For all constant variables, use `const`;
- Promises: offer several code management advantages, such as:
  - No more callback pyramid of doom:

```
        doSomething(function (result) {
            doSomethingElse(result, function (newResult) {
                doThirdThing(newResult, function (finalResult) {
                console.log(`Got the final result: ${finalResult}`);
                }, failureCallback);
            }, failureCallback);
        }, failureCallback);
        //replaced by:
        doSomething()
        .then(function (result) {
          return doSomethingElse(result);
        })
        .then(function (newResult) {
          return doThirdThing(newResult);
        })
        .then(function (finalResult) {
          console.log(`Got the final result: ${finalResult}`);
        })
        .catch(failureCallback);

```

- Note the importance of the `return` statments in the chain. A floating promise in such a chain is one without a return statement, whose output is thus not passed to the following then clauses.
- Floating promises can lead to race conditions. Ex:

```
        const listOfIngredients = [];
        doSomething()
        .then((url) => {
            // Missing ‘return‘ keyword in front of fetch(url).
            fetch(url)
            .then((res) => res.json())
            .then((data) => {
                listOfIngredients.push(data);
            });
        })
        .then(() => {
            console.log(listOfIngredients);
            // listOfIngredients will always be [], because the fetch request hasn't completed yet.
        });

```

- Using async/await allows for writing code with promises that resembles synchronous code closely.
- `.then(callback)` attached after `.catch` is both with and without execution of the catch statement.

- Promise composition:
  - `Promise.all([p1,p2,1,abc])` await all promises (non-sequentially) and return array with values returned by them. It rejects with the error from the first promise being rejected.
  - For sequential promise resolutions:

```
        [func1, func2, func3]
        .reduce((p, f) => p.then(f), Promise.resolve())
        .then((result3) => {
            /* use result3 */
        });
        //Think of Promise.resolve as a blank promise that returns undefined immediately.
        //Promise.resolve("value").then((v)=>{console.log(v);})//"value".

```

    Or using async/await:

```
        let result;
        for (const f of [func1, func2, func3]) {
        result = await f(result);
        }
        /* use last result (i.e. result3) */


```

  - `Promise.allSettled([p1,p2,p3])` works like `Promise.all`, except it never rejects. It waits for all promises to be resolved or rejected and returns an array of status-value objects:

```
        Promise.allSettled([
        Promise.resolve(33),
        new Promise((resolve) => setTimeout(() => resolve(66), 0)),
        99,
        Promise.reject(new Error("an error")),
        ]).then((values) => console.log(values));
        /* [
           { status: 'fulfilled', value: 33 },
           { status: 'fulfilled', value: 66 },
           { status: 'fulfilled', value: 99 },
           { status: 'rejected', reason: Error: an error }
         ] */

```

  - `Promise.any([p1,p2,p3])` resolves with the value from the first (quickest) to be resolved. If an empty array is passed or if all the promises are rejected, it rejects with an array containing reject-values for each promise.
  - `Promise.race([p1,p2,p3])` resolves when the first (quickest) promise rejects or resolves.

- Promise error handling: (References one and two)

```
P.then((v)=>{return transform(v);}).catch((err)=>{handle(err);});
//handle will be called for errors from P and for errors in transform(v).
P.then((v)=>{return transform(v);},(err)=>{handle(err)}).catch((err)=>{
    handleTransformError(err)});
//handle will be called only for errors from P.
```

- Nesting: convert nested if-checks to flattened if (invalid), return statements.
- Naming convention: adjectives prefixing nouns, camelCase.
- When defininining useful constants (that can be written in terms of other constants),
  - Use uppercase snake_case for names.
  - Have the definition convey be meaningful in terms of other constants.
- DOM manipulation calls (`document.appendChild`) should not be done in loops.
- Avoid creating extra objects in memory during accumulation operations:

```
    const users = [
  { name: "Medhat", admin: true },
  { name: "Adam", admin: false },
  { name: "Karma", admin: true },
]

// spread operator is creating a new Obj
users.reduce(
  (acc, item) => ({
    ...acc,
    [item.name]: item.admin,
  }),
  {}
)
//better:
users.reduce((acc, item) => {
  acc[item.name] = item.admin
  return acc
}, {})
```

Chapter 15

Python

15.1 Misc.

- The else block at the end of a for loop is executed only if the for loop was exited without using a break statement.
- Same rule applies to an else block at the end of a while loop.
- The try-except-else-finally block works such that:
  - Try block is always executed.
  - Except block catches particular errors if thrown and hence executes only if errors arise.
  - Else block executes if NO error was encountered.
  - Finally block is always executed.
- TODO list:
  - try-catch-finally better reads.
  - with block.
  - functions with args, kwargs.
  - sys.argv etc.
  - match case statement.
  - python pass by reference or value.

Chapter 16

Solid Principles from Stackoverflow

- Solid is a set of principles distilled from the writings of Robert C. Martin in the early 2000s. It is a way to think specifically about:
  - the quality of OOP
  - how the code should be split up
  - which parts should be internal/exposed
  - how code should use other code.
- Some major changes in the industry after the development of the solid principles are:
  - Dynamically-typed languages
  - Non-OOP paradigms: functional programming.
  - Open source software has proliferated.
  - Microservices, SAAS: services that talk to other services.
- These changes imply that many things SOLID really cared about: classes, interfaces, data hiding, polymorphism, are not what programmers deal with every day.
- What hasn't changed:
  - Code is written and modified by people over and over again. There will always be a need for well-documented code and APIs.
  - Code is organized into modules. Each language provides some way of organizing code into distinct, bounded units. Thus, there will always be a need to decide how best to group code together.
  - Code can be internal or external: some code is written for use within the team, while some is written for use by consumers or other teams. There is a need to decide what is visible and hidden.
- Up we discuss SOLID principles as applicable to modern programming.

16.1 Single Responsibility Principle

- Each class (module) should do one job, and do it well. Our code shouldn't mix multiple roles or purposes together: high cohesion.
- We shouldn't have to modify the code for that class unless there are changes to its specific function, or the interface it provides, or the interfaces it uses.

16.2 Open-closed Principle

- Software entities (classes) should be open for (allow) extension (via inheritance), but closed for (disallow) modification. (Once the code is sent into production).
- This limits the dependency on the author of the class. Additionally, multiple changes could result in the class incorporating too many concerns.
- It also protects the code from unskilled hands.
- We should be able to use and add to a module (via inheritance), without rewriting it.
- In the FP world, this involves defining explicit hook points in the base method to allow for before and after modifications, as well as a way to override the base behaviour.

```
// library code

const saveRecord = (record, save, beforeSave, afterSave) => {
    const defaultSave = (record) => {
    // default save functionality
    }

    if (beforeSave) beforeSave(record);
    if (save) {
    save(record);
    }
    else {
    defaultSave(record);
    }
    if (afterSave) afterSave(record);
}

// calling code

const customSave = (record) => { ... }
saveRecord(myRecord, customSave);
```

16.3 Liskov Substitution Principle

- Objects of a subclass should be allowed to be substituted in methods involving objects of the superclass, without altering any desirable properties of the program (breaking anything).
- This allows for/encourages polymorphism in OOPS.
- In FP, it encourages function parameters, as passed to filter and map in javascript.

16.4 Interface Segregation Principle

- Many client-specific interfaces are better than one general-purpose interface.
- Don't show your client (this could be another piece of code using an interface/type), more than what they need to see.

```
interface PrintRequestModifier {
   public void createRequest();
   public void deleteRequest();
}

interface PrintRequestWorker {
   public void workOnRequest()
}

```

```
10 class PrintRequest implements PrintRequestModifier, PrintRequestWorker {
11    public void createRequest() {}
12    public void deleteRequest() {}
13    public void workOnRequest() {}
14 }
```

- This decreases coupling and ensures that a client doesn't need to know about, or depend on, features that it has no intention of using.

16.5 Dependency Inversion Principle

- Depend on abstractions, not concretions.
- In OO, this means that clients should depend on interfaces rather than concrete classes as much as possible; this minimizes the surface area of the code that the client depends on.

Chapter 17

Aptitude Test Pointers

17.1 Number Sequence

These questions are hella annoying. So, I've written down a list of possibilities here that I can refer to while cheating in the test. (That's a joke.)

1. A.P, G.P, AGP. HP.
2. Check difference, 2nd difference ...
3. Incorporating a well-known series.
4. Consider $x^{f(x)}$ for monotonic series with large gaps.
   (a) Primes
   (b) Fibonacci (spot by nth difference - n-1th difference)
5. Two series of alternating numbers interleaved or alternating next() function.
   (a) The next function alternates between constant difference and constant factor.
   (b) The next function alternates between f(x)=ax+d and g(x)=bx+c
6. If the numbers go up and down, it's a result of interleaving or the relation between neighbours keeps switching between two functions.
7. Given alphabet series,
   (a) Convert to positions and reverse positions.
   (b) Consider vowel and consonant relations.
8. The sequence might involve manipulating a permutation of the previous element lmao.