Pre-Placement Grind

Hardik Rajpal

October 1, 2023

Contents

Chapter 1

Misc.

1.1 Misc. Confirmed Optimizations

1. 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];}
```

2. Replace sets that check for inclusion by bit operations with an integer if the number elements < 32 for ints and 64 for long longs.

3. Pass params by reference where possible.

4. Instead of a reference parameter, consider using a pointer to that variable stored in a class data member.

5. Replace data types by smaller data types where possible:

   - long long by int
   - int by char

6. Replace fixed-length vectors by array<type,fixed-length>.

7. Replace maps by vectors if they are indexed by integers within a fixed range. Ex: the alphabet as indices.

8. That die roll problem optimization. (Needs to be phrased more mathematically.)

9. 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 $\leq$ 0, else n $\geq$ 0.

- Bits to n is equivalent to

  - Identify sign based on MSB.
  - Flip all other bits and find the unsigned integer they represent.

- 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.

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 due to 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. //TODO pseudo code

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.)

```cpp
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

```cpp
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

TODO: write up permutation ideas. TODO: selection of subsets satisfying a property.

Backtracking En General

If the dimensions of inputs are smaller than usual, backtracking is an option. As with other algorithms, you want to maximize 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:

```
1    //declare global/class member variables.
2    void search(int p){
3        //p signifies path/position being inspected
4        //in the search space.
5        if(searchTerminalConditions()){
6            if(globalVarSolutionValid){
7                //update collection of solutions.
8            }
9        }
10       else{
11           for(possible path of exploration){
12               //(1)update global vars so as to take this path.
13               search(p+1);
14               //(2)undo the updates made to global variables.
15               //(not necessary if (1) overrides/uses previous updates.)
16           }
17           //undo any leftover changes made to global variables.
18       }
19   }
```

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`.

```
1  vector<T> items;//given vector of items.
2  vector<int> root;//representative roots of trees array.
3  vector<int> rank;//for combine optimization.
4  int find(int u){
5      if(root[u]==u){return u;}
6      //instead of return find(root[u]), do:
7      root[u] = find(root[u]);//path compression
8      return root[u];
9  }
10 void combine(int u, int v){
11     int ru, rv;
12     ru = find(u);rv = find(v);
13     if(ru!=rv){
14         //u, v in different trees.
15         if(rank[ru] < rank[rv]){
16             root[ru] = rv;
17             rank[rv] += rank[ru];
18             //combined tree has least possible height.
19         }
20         else{
21             root[rv] = ru;
```

```
22          rank[ru] += rank[rv];
23        }
24      }
25  }
```

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.

```
1  int distance(vector<int> &pi, vector<int> &pj);
2  int dijkstras(vector<vector<int>>& ps, int s, int target) {
3      int n = ps.size(), res = 0, i = s;
4      vector<int> min_d(n, INT_MAX);
5      while (i != target) {
6          min_d[i] = -1;
7          int min_j = i;
8          for (int j = 0; j < n; ++j){
9              if (min_d[j] != -1) {//visited vertices.
10                 min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
11                 min_j = min_d[j] < min_d[min_j] ? j : min_j;
12             }
13         }
14         res += min_d[min_j];
15         i = min_j;
```

```
16        }
17        return res;
18 }
```

Listing 5.1: Shortest Path

```
1  int distance(vector<int> &pi, vector<int> &pj);
2  int dijkstras(vector<vector<int>>& ps, int s, int target) {
3      int n = ps.size(), res = 0, i = s,connected = 0;
4      vector<int> min_d(n, INT_MAX);
5      while (connected < n) {
6          min_d[i] = -1;
7          connected++;
8          int min_j = i;
9          for (int j = 0; j < n; ++j){
10             if (min_d[j] != -1) {//visited vertices.
11                 min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
12                 min_j = min_d[j] < min_d[min_j] ? j : min_j;
13             }
14         }
15         res += min_d[min_j];
16         i = min_j;
17     }
18     return res;
19 }
```

Listing 5.2: Dijkstra's MST

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:

```
1  while(s.size()){
2      ans.push_back(s.top());
3      s.pop();
4  }
5  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.

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

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

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

```
1   class DerivedClass: access-specifier baseA, access-specifier baseB ... {
2   //access-specifier is one of public/protected/private.
3   };
```

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).

```
1   class Derived: public Base1, public Base2...
2   //base1 constructor.
3   //base2 constructor.
4   //derived constructor.
5   ...
6   //derived destructor.
7   //base 2 destructor.
8   //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:
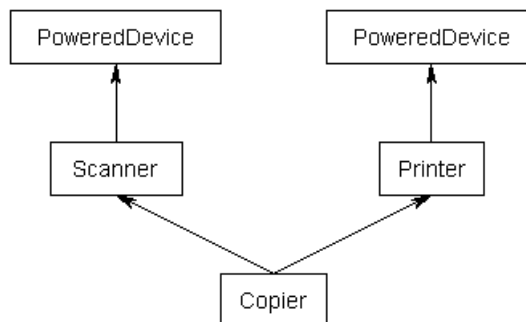
- Each parent class has its own copy of the base class data members (resulting in redundant copies), and we can't call public members of the base class from the new derived class.

- 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:

```
1   class base{};
2   class b1: virtual public base{};//
3   class b2: virtual public base{};//without virtual in both of them, copies are made.
4   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 −)

```cpp
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

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

```
11          }
12  };
13  class Triangle: public Shape {
14      public:
15          int area () {
16              cout << "Triangle class area :"...
17          }
18  };
19  int main() {
20      Shape *shape;
21      Rectangle rec(10,7);
22      Triangle  tri(10,5);
23      shape = &rec;
24      shape->area();
25      shape = &tri;
26      shape->area();
27      //both calls print "Parent class area:..."
28      return 0;
29  }
```

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.

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.

```
1  class Shape{
2      //virtual int area();
3      //Above line compiles if there are no to area() from any objects of shape/derived classes.
4      //In the presence of such calls, even if area() is defined in derived classes and their
5      //objects are used to call area() (from the right pointer, or a pointer of type Shape*)
6      //compilation errors ensue.
7      //However,
8      virtual int area()=0;//goes through compilation successfully.
9  };
```

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:

```
1  class Rectangle:public Shape{
2      public:
3      int area()final{
4          cout<<"Rect Area"<<endl;
5          return 0;
6      }
7  };
8  //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.

```
1  void printarea(Shape s){
2      s.area();
3  }
4  void printareaReference(Shape &s){
5      s.area();//NO slicing.
6  }
7  Shape s; Rect r;
8  s = r;//slicing.
9  Shape &s = r;
10 printarea(r);//slicing. prints "Shape area..."
11 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 Copy Constructors

```
1  class Sample{
2      int id;
3      Sample(Sample &t)
4      {
5          id=t.id;
6      }
7  };
8  //defines what do to do when:
9  Sample a,b;
10 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.

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.

```
1  Shape s1,s2;
2  s1 = {1,3};
3  Shape s3(s1);//calls copy constructor.
4  Shape s4 = s1;//calls copy constructor.
5  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.
     ```
     1      //if copy constructor doesn't say const Shape &s1,
     2      Shape s2 = fun();//fun returns s2 by value.
     3      //the above code throws a compilation error, at the last line.
     4      //If const is present, it compiles.
     5
     ```

- 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

```
1  const Date today {2020, 10, 14};//valid.
2  const Date today = {2020, 10, 14};//valid.
3  //Note that const objects must be initialized, unless a default
4  //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.

```
1  class Date{
2      //can't be called by a const object, despite not
3      //altering any variables in its definition:
4      void print(){
5          cout<<"non-const member function.";
6      }
7      //can be called by const objects:
8      void print2() const {
9          cout<<"const member function.";
10     }
11 }
```

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

```
1  class Date{
2      void print() const;
3  };
4  void Date::print()const{
5      ...
6  }
```

- 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.

```
1  //These are valid overloads:
2  int fun(){
3  
4  };
5  int fun()const{
6  
7  };
8  //These are invalid as const keyword specifiers return type.
9  int fun(){}
10 const int fun(){}
11 //Also note that
12 const int fun(){};
13 //is exactly the same as:
14 int const fun(){};
15 //and the two are different from
16 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.

```
1  friend void printWeather(const Temperature& temp, const Humidity& hum){
2      //access private members of both temp and hum at once.
3  }
```

- 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.

```
1  class Storage{
2      //private members here.
3      friend class Display;
4      //Display declared as a friend of storage.
5      //Display accesses all members of storage.
6      //Note: no forward declaration required.
7  };
8  class Display{
9      void print(const Storage& storage);
10 }
```

- 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:

```
1  class Storage{
2      //private members.
3      friend void Display::displayStorage(Storage &storage);
4      //compiler should have seen displayStorage in Display
5      //before this line.
6  };
```

- 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

More C++ Stuff

8.1 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.

8.1.1 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:

```
1    try{
2        //exception throwing code.
3    }
4    catch(int a){
5
6    }
7    catch(...){
8        cout<<"All non-int exceptions lead here"<<endl;
9    }
```

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

Chapter 9

DSA Revision Notes

9.1 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.

-

9.2 Hash Tables

- Load factor $\alpha = n/m$, where the hash table has m slots (unique indices) and holds n elements. $\alpha =$ 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: $KeySet \implies Z$
  2. Compression map: $Z \implies [0, 1, ...N - 1]$

- 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.

9.2.1 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 $\in$ H and two keys k and l,
  $P(h(k) = h(l)) < 1/m$

9.2.2 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.)

9.3   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,

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

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

9.3.1   Binary Search Tree

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

```
1  Tree search(Tree t,Value v){
2      if(t->value==v){
3          return t;
4      }
5      else if(t->value < v){
6          if(t->left){
7              return search(t->left,v);
8          }
9          return NULL;
10     }
11     else{
12         //t->value > v
13         if(t->right){
14             return search(t->right,v);
15         }
16         return NULL;
17     }
18 }
```

- 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. Search: as given above.
  2. Insert: search for node in tree, insert as a child where NULL is found.
  3. Successor:

```
1  Node *n successor(Node *x, Node *n){
2      if(x->right){
3          return leftMostChild(x->right);
4      }
5      else{
6          Node *y = parent(x,n);
7          while(y!=NULL && x==y->right){
8              x = y;
```

```
 9              y = parent(x,n);
10          }
11          //y == root or y is closest ancestor such that x is in y->right.
12          return y;
13      }
14  }
```

4. Deletion:

```
 1  void delete(Node *x, Node *n){
 2      if(!(x->left&&x->right)){
 3          //x has at most one child.
 4          Node *y = parent(x,n);
 5          if(y->left==x){
 6              y->left = oneChildOf(x);//may return null if x has no children.
 7          }
 8          else if(y->right==x){
 9              y->right = oneChildOf(x);
10          }
11      }
12      else{
13          //x has two children.
14          Node *s = Successor(x,n);
15          delete(s,n);
16          Node *y = parent(x,n);
17          //transfer the children
18          s->right = x->right;s->left = x->left;
19          if(y->left==x){
20              y->left = s;
21          }
22          else if(y->right==x){
23              y->right = s;
24          }
25      }
26  }
```

9.3.2   Traversals

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

```
 1  void preorder(Node n){
 2      process(n);
 3      for(Node &u:n.children){
 4          preorder(u);
 5      }
 6  }
```

- Used in reading documents, webpages.

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

```
 1  void postorder(Node n){
 2      for(Node &u:n.children){
 3          postorder(u);
 4      }
 5      process(n);
 6  }
```

- 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.

```
1  void inorder(Node n){
2      for(Node &u:n.leftchildren){
3          inorder(u);
4      }
5      process(n);
6      for(Node &u:n.leftchildren){
7          inorder(u);
8      }
9  }
```

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

```
1  void eulerian(Node n){
2      process(n);//First process
3      for(Node &u:n.leftchildren){
4          eulerian(u);
5      }
6      process(n);//Second process
7      for(Node &u:n.leftchildren){
8          eulerian(u);
9      }
10     process(n);//Third process.
11 }
```

9.3.3   AVL Trees

- They are aka height-balanced trees.

- They are binary (searchable,sorted) trees, with the added property of balance.

- If $h(n)$ denotes the height of the tree rooted at n:

$$h(n) = 1 + max_{u \in n.children}(h(u))$$
$$h(n) = 1 \implies \text{n is a leaf.}$$

- $\forall n \in \{InternalNodes\}, |h(u) - h(v)| \leq 1$, where u and v are children of n.

- $h(root) \leq log_\phi(n)$

- 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. Insertion:

9.4   KMP

9.4.1

Chapter 10

Aptitude Test Pointers

10.1 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.