Pre-Placement Grind

Hardik Rajpal

August 27, 2023

Contents

Chapter 1

Week 1

1.1 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: $\mathrm{O}(2\sqrt{n} = O(\sqrt{n}))$, 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`.

1.2 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

Quick Sort

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.
   Time: O(n^2)
   Space: O(1)

Chapter 2

Week 2

TODO move notes over from the other notebook.

Chapter 3

Week 3

3.1 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 3.1: 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:

```
//declare global/class member variables.
void search(int p){
    //p signifies path/position being inspected
    //in the search space.
    if(searchTerminalConditions()){
        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 4

Week 4

4.1 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.

4.1.1 Union Find

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

```cpp
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;
```

```
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 4.1: 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 4.2: 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.

4.1.2 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.

4.1.3 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 5

Week 5

5.1 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.

5.1.1 Common Patterns

Min (Max) Path to Reach Target

Distinct Ways

Merging Intervals

DP on Strings

Decision Making

Chapter 6

Aptitude Test Pointers

6.1 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.