Pre-Placement Grind

Hardik Rajpal

July 27, 2023

Contents

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: 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)