Things to know for each searching/sorting algos

Worst case time complexity (when)
Expected time complexity (when)
Best case time complexity (when)
Invariant
good in which case
Bad in which case
Implementation (at least in pseudo code)
Runtime
Space usage
Stability

Check List

- **Big-O notation**
  - Definition: Upper bound
    - n0> 0; c > 0
    - For all n > n0 -> T(n) <= cf(n)
  - Definition: Lower bound
    - n0> 0; c > 0
    - For all n > n0 -> T(n) >= cf(n)
- **Recurrences**
- **Edge cases of algos**
- **Binary Search**
  - **Conditions**
    - While begin < end
    - Key <= begin + (end - begin)/2
    - Begin = mid + 1 → no array out of bounds because division always round down
    - End = mid
    - Return arr[begin] == key ? begin : -1
- BST
  - Insert
  - Delete: 3 cases
  - Find min
  - Find max
  - Find successor
  - Find predecessor
  - inorder/preorder/post-order
  - search

Common Recurrences

1. T(n) = 2T(n/2) + n → O(nlogn) (e.g merge sort)
2. T(n) = 2T(n/2) + 1 → O(n) (e.g in-order traversal)
3. T(n) = T(n/2) + 1 → O(logn) (e.g binary search)
4. T(n) = T(n/2) + n → O(n)
5. T(n) = T(n - 1) + T(n - 2) + 1 → O(2^n) (e.g. fibonacci)

The more common Big O Notation

Summary

Name	Best Case	Average Case	Worst Case	Extra Memory	Stable?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

Big-O Notation

Order or size:

Function	Name
5	Constant
loglog(n)	double log
log(n)	logarithmic
$\log^2(n)$	Polylogarithmic
n	linear
nlog(n)	log-linear
n^3	polynomial
$n^3\log(n)$	
n^4	polynomial
2^n	exponential
2^{2n}	
n!	factorial

O(log(n!)) = O(nlog(n))

Recurrences

T(n) = 1 + T(n - 1) + T(n - 2)
Big O will be O(2^n)
Example would be the recurrence relation of fibonacci algorithm

SEARCHING

Binary Search

- **Implementation**

1. Start at the middle
2. If that is the search element → return element
3. If the search element less than middle → search recursively on left (update new high)
4. If the search element more than middle → search recursively on right (update new low)

- Things to take note
  - Middle is low + (high - low) / 2
  - Stop when low > high

Binary Search

Sorted array: A[0..n-1]

2	4	4	5	6	7	8	9	11	17	23	28

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if  key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

- Preconditions
  - Array of size n
  - Array must be sorted
- Postcondition
  - If element is in array then A[begin] = key
- Loop invariant
  - A[begin] <= key <= A[end]

Binary search algorithm

Sorting algorithm

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. Wikipedia

Worst complexity: O(log n)
Average complexity: O(log n)
Best complexity: O(1)
Space complexity: O(1)
Data structure: Array
Class: Search algorithm

Peak Finding (Key idea is binary search)

To find local maximum

- Implementation
  1. Start at the middle
  2. If A[mid] is peak → return A[mid]
  3. Else if left > A[mid] → search left
  4. Else if right > A[mid] → search right
- Invariant
  - If we search right then peak is in right
  - If we search left then peak is in left (can prove using induction)
  - There exits a peak in the range [begin, end] and the peak in [begin, end] is also peak in [1, n]
- Recurrence
  - T(n) = T(n/2) + O(1)
  - O(1) is comparing the middle with the left and right element

SORTING

Bogosort

- Implementation

1. Choose random permutation of A
2. Return that permutation if A is sorted

- Time complexity
  - O(n * n!)
- Best complexity
  - O(n)
- Worst complexity
  - infinite
- Space complexity
  - O(1)
- Stability
  - Not stable

Quick sort

- Implementation (Partition)

1. Choose pivot
2. Find element that is less than pivot → move to left side of pivot
3. Find element that is greater than pivot → move to right side of pivot

- Time complexity
  - Runtime partition: O(n)
- Invariant partition
  - For all i > high, A[i] > pivot
  - For all i < j< low, A[j] < pivot

```
partition(A[1..n], n, pIndex)       // Assume no duplicates, n>1
    pivot = A[pIndex];              // pIndex is the index of pivot
    swap(A[1], A[pIndex]);          // store pivot in A[1]
    low = 2;                        // start after pivot in A[1]
    high = n+1;                     // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high--;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low-1]);
    return low-1;
```

-
- Implementation for Quick sort

Prev PgUp | 12-9. Demonstration | Next PgDn

Try Quick Sort on example array [27, 38, 12, 39, 27, 16]. We shall elaborate the first partition step as follows:
We set **p = a[0] = 27**.
We set **a[1] = 38** as part of **S2** so **S1 = {}** and **S2 = {38}**.
We swap **a[1] = 38** with **a[2] = 12** so **S1 = {12}** and **S2 = {38}**.
We set **a[3] = 39** and later **a[4] = 27** as part of **S2** so **S1 = {12}** and **S2 = {38,39,27}**.
We swap **a[2] = 38** with **a[5] = 16** so **S1 = {12,16}** and **S2 = {39,27,38}**.
We swap **p = a[0] = 27** with **a[2] = 16** so **S1 = {16,12}**, **p = {27}**, and **S2 = {39,27,38}**.

After this, **a[2] = 27** is guaranteed to be sorted and now Quick Sort recursively sorts the left side **a[0..1]** first and later recursively sorts the right side **a[3..5]**.

- Stability
  - Not stable
- Recurrence
  - General: T(n - k) + T(k) + n
- Time complexity
  - O(nlogn)
- Best case
  - O(nlogn) → when the partition always divide the element into 2 equal halves
  - T(n) = T(n /2) + T(n/2) + n
  - n=cost of partition on n elements
- Worst case
  - O(n^2) → when each partition only reduce the number of element by one
  - T(n) = T(n - 1) + T(1) + n
  - n=cost of partition on n elements
  - T(1) = cost of quicksort on one element
- Space complexity
  - O(logn) → quicksort calls itself logn time. Each time allocation a new space
- Bad pivot
  - First element
  - Last element
  - Middle element
  - Basically every single thing that is the same position
  - Runtime: O(n^2)
- Good pivot
  - Median element
  - Random element
  - Runtime: O(nlogn)
- QuickSelect
  - Runtime: O(n) → select stuff using the idea of partition

Order Statistics (Search for kth element)

- **Implementation**

1. Choose random pivot
2. Do partition
3. If the kth element is on the left → recursive left find the kth element in left
4. Else if kth element is on the right → recurse right, find the k - nth element on the right

- Time complexity
  - O(n)

Note: the main idea is we just do recursion on just one side instead of 2 sides. This is a lot faster

TREES

Binary Tree

- Factors that determine the order of BST
  - Order of insertion
- Traversals
  - Inorder: L-Middle-R
  - Preorder: Middle-L-R

Bubble sort

- Implementation

1. If A[j] > A[j + 1] → swap
2. Repeat n times, each time with one less element (the last element of each

- Recurrence
  - T(n) = T(n - 1) + n
- Time complexity
  - O(n^2)
- Best complexity
  - O(n) → when all are sorted
- Worst complexity
  - O(n^2) → when it's reverse sorted
- Space complexity
  - O(1)
- Stability
  - Yes → only swap elements that are different
- Loop invariant
  - At the end of j iteration → last j element is sorted
  - Last j element are also the biggest j elements

Selection sort

- Implementation

1. Find minimum element A[j] im A[1..n]
2. Swap A[j] with A[k]

- Recurrence
  - T(n) = T(n - 1) + n
- Time complexity
  - O(n^2)
- Best complexity
  - O(n^2)
- Worst complexity
  - O(n^2) → when it's inversely sorted
- Space complexity
  - O(1)
- Stability
  - Not stable → swap changes order
- Loop invariant
  - After j iteration → the smallest j elements are sorted

Insertion sort

- Implementation
  - J from 2 to n - 1 insert A[j] into sorted array
- Recurrence
  - T(n) = T(n - 1) + n
- Time complexity
  - O(n^2)
- Best complexity
  - O(n) → when it's already sorted
- Worst complexity
  - O(n^2) → when it's inversely sorted
- Space complexity
  - O(1)
- Stability
  - Yes
- Loop invariant
  - First j elements are sorted

Note: Insertion sort is very fast on sorted array!

Merge sort

- implementation
  - Split array into 2 halves
  - Recursively sort the 2 halves
  - Combine both
- Recurrence
  - T(n) = 2T(n/2) + cn (c is a fix constant)
- Time complexity
  - O(nlogn)
- Best complexity
  - O(nlogn)
- Worst complexity
  - O(nlogn)
- Space complexity
  - O(n) → put the elements into new array
- Stability
  - Stable

Successor Queries

Background: when you search for a key that is not in the tree, you will either get its successor or predecessor

If you want to search for successor:

1. Search for that key that is not in the tree
2. Check if the obtained key is successor
3. If it is → return that key
4. Else, search for successor of that key

Background: If you want to search for successor and the key is in that tree

1. Just search for that node normally
2. After finding the node, if the node has right child → search rightchild.min()
3. Else search parent of that node until the child is not the parent's right tree (the child is the parent's left tree)

Deleting node

- Case1: No children
  - Just delete
- Case 2: one child
  - Delete that node
  - Then, connect the child to the parent of that node
- Case 3: 2 children
  - Find successor of the node that you want to delete
  - Delete that node
  - Remove successor
  - Successor take that position instead

Binary Search Tree

delete(v) Running time: O(height)

Three cases:

1. No children:
   - remove v
2. 1 child:
   - remove v
   - connect child(v) to parent(v)
3. 2 children
   - x = successor(v)
   - delete(x)
   - remove v
   - connect x to left(v), right(v), parent(v)

Binary Search Tree

Modifying Operations
- insert: O(h)
- delete: O(h)

Query Operations:
- search: O(h)
- predecessor, successor: O(h)
- findMax, findMin: O(h)
- in-order-traversal: O(n)

Height-balanced Tree (AVL)

Note

1. **A height-balanced tree with n nodes has at most height h < 2logn**
2. **A height-balanced tree with height h has at least n > 2^(h/2) nodes**

Trie

Trie Tradeoffs

Time:
- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Even faster if string is not in trie!

Trie Tradeoffs

Time:
- Trie tends to be faster: O(L).
- Does not depend on size of total text.
- Does not depend on number of strings.

Space:
- Trie tends to use more space.
- BST and Trie use O(text size) space.
- But Trie has more nodes and more overhead.

Dynamic Order Statistics

1. If you search kth item and you are going to the right
2. When you search right, you are searching the k - (left_weight + 1)
3. If you are searching left, you continue to search the kth item

Merge sort is more efficient and works **faster than quick sort** in case of larger array size or datasets. **Quick sort** is more efficient and works **faster than merge sort** in case of smaller ar size or datasets

Insertion Sort is preferred for fewer elements. It becomes fast when data is already **sorted o** nearly **sorted** because it skips the **sorted** values. Efficiency: Considering average time compl of both algorithm we can say that **Merge Sort** is efficient in terms of time and **Insertion Sort** i efficient in terms of space.

Insertion sort is **faster** for small n because **Quick Sort** has extra overhead from the recursiv function calls. **Insertion sort** is also more stable **than Quick sort** and requires less memory.

GRAPHS

Adjacency list

Adjacency matrices

*Note: Both DFS and BFS do not explore all possible path

E.g. cannot use DFS/BFS to find longest path

BFS

```
frontier = {s}
while frontier is not empty:
    next-frontier = {}
    for each node u in the frontier:
      for each edge (u,v) in the graph:
          if v has not yet been visited, add v to next-frontier
    frontier = next-frontier
```

- **Steps**
  - Look at the current frontier
  - Once the queue in the current frontier is empty, go to the next frontier
- Runtime (assume adjacency list)
  - O(V+E)
    - Each vertex is explored once
    - Each edge is being gone through once
- Can be used to
  - Find shortest path
  - Parent pointers store shortest paths

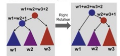
DFS

Augmented Trees

1. Updating weight takes constant time (when rebalancing)
2. You only update the weight of the 2 affected nodes (blue and orange). The rest remains the same

Augmented Trees

Maintain weight during rotations:



Interval Search

- Runtime
  - O(logn)
- If the thing we are searching is more than the interval then go right. Else go left.

Orthogonal Range Searching (One Dimension)

- Implementation
  1. Find split node (take note that the circle nodes contain the maximum element on the left subtree)
  2. Do left traversal
  3. Do right traversal
- Time complexity
  - Finding split node: O(logn)
  - We recurse at most O(logn) time
  - If k is the number of queries found → overall time complexity: O(k + logn)

Note:

1. do all-leaf traversal-right if key >= low
2. Do all-leaf traversal-left if key <= high

Symbol Table

- Key should be immutable
- Using <key, value> pair

Direct Access Tables

- Time complexity:

  Search: O(1)

  Insert: O(1)
- Problems
  - Too much space
  - If keys are not integers then it's a problem (how to arrange them in a table)

Hashing

Depth-First Search

```
DFS(Node[] nodeList){
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);

  for (start = i; start<nodeList.length; start++) {
    if (!visited[start]){
        visited[start] = true;
        DFS-visit(nodeList, visited, start);
```

- **Steps**
  - Start at a node
  - Go all the way until cannot go further
  - Backtrack until find a node that can go further
  - Do recursively until there is no more nodes

Runtime
  - O(V+E) (adjacency list) → each node is visited once, each edge also visited once
  - O(V^2) (adjacency matrices) → each node is visited once. Need to check all the possible neighbouring nodes

Topological order
- Only have forward edges
- Only in DAG
- Post-order DFS

SSSP
- **Bellman Ford**
  - Can be used with
    - Negative weight edges
    - Can detect existence of negative weight cycle → if something still change at the last iteration, there is negative weight cycle
  - Runtime
    - O(VE)
    - Run the algo V time, each time visit all the edges

Dijkstra's
- **Steps**
  - Start from source
  - Relax based on the priority in the min priority queue
  - Until there is no more nodes in the queue
- Priority queue is implemented using AVL tree
- All operational costs: deleteMin() insert() decreaseKey() has cost of log(n) per operation
- **Runtime**
  - Each node is insert once → O(VlogV)

Hash table		
Type	Unordered associative array	
Invented	1953	
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	O(n)[1]	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

- How to deal with collision
  - Chaining
    - Put both items in the same bucket
    - Form a linked list
  - Open Addressing
- Implementation (for hashing with chaining)
  - Insert
    1. Calculate h(key)
    2. Look up h(key) and add (key, value) to the linked list

    Runtime: O(1 + cost(h))

    Runtime (expected/worst): O(1)/O(1)
  - Search
    1. Calculate h(key)
    2. Search for (key, value) in the linked list

    Runtime: O(n + cost(h))

    Runtime (expected/worst): O(1)/O(n)
- Java Hash Functions
  - Always returns the same value, if the object hasn't changed
  - If two objects are equal, then they return the same hashCode
  - Must redefine .equals to be consistent with the hashCode

Java Hash Functions

Every object supports the method:

```
boolean equals(Object o)
```

```
boolean equals(Object p){
  if (p == null) return false;
  if (p == this) return true;

  if (!(p instanceOf Pair)) return false;
  Pair pair = (Pair)p;

  if (pair.first != first) return false;
  if (pair.second != second) return false;
  return true;
}
```

- Properties of good hash function
  1. h(key, i) should enumerate all the buckets → if not it'll return table full when there's still space left
  2. Simple uniform hashing assumption → every key is equally likely to map to every permutation, independent of other keys

- Each edge is decrease once → O(ElogV)
- Each node is deleted once → O(VlogV)
- Total: O(ElogV)

- Can be used:
  - Find shortest path when there is no negative weight cycle
  - Find the longest path by negating the edges. (Only when no cycle)

Shortest Path Summary:

Graph Type	Algorithm	Time
No negative weight cycles	Bellman-Ford	O(VE)
No negative edges	Dijkstra	O(E log V)
No directed cycles	TopoSort + Relax	O(E)
No cycles	DFS + Relax	O(V)
Planar...		
Bounded arboricity...		
Minor-Free Graphs...		

Heap

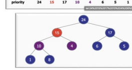
- **Insert**
  - Add at left-most leaf
  - Then bubble up until the condition is satisfied
- **Increase key**
  - Just increase then bubble up
- **Decrease key**
  - Just decrease and then bubble down
  - Bubble down to the side that has greater root
- **Delete**
  - Swap with the latest leaf node
  - Bubble down the leaf node
- **Extract max**
  - Do as if we are deleting the max node
  - Swap max node with the latest leaf node
  - Bubble down the swapped leaf node

	Time Complexity	Best Time	Worst Time	Space	Stability	Invariant	Good when?	Bad when?	Recurrence
Bubble	O(n^2)	O(n)	O(n^2)	O(1)	Stable	Last j elements sorted	Small dataset Almost sorted array	Reversed array Smallest element in last position	T(n) = T(n-1) + n
Selection	O(n^2)	O(n^2)	O(n^2)	O(1)	Unstable	First j element sorted First j also the smallest j	Check if already sorted Least swap	Large array Inversely sorted	T(n) = T(n-1) + n
Insertion	O(n^2)	O(n)	O(n^2)	O(1)	Stable	First j element sorted	Almost sorted array	Inversely sorted Large array	T(n) = T(n-1) +n
Merge	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Stable	Subarray sorted	Sorting linkedlist When we want to maintain the order of array	Almost sorted array	T(n) = 2T(n/2) + cn
Quicksort	O(nlogn)	O(nlogn)	O(n^2)	O(logn)	Unstable	Pivot is correct position Element on left less than pivot Element on right greater than pivot	When pivot will divide elements into 2 equal size groups	When pivot only remove one element	T(n) = T(n - k) + T(k)
Quickselect	O(nlogn)	O(n)	O(n^2)	-	-				

Store Tree in an Array

```
left(x)  = 2x+1
right(x) = 2x+2
```

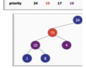
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Arr

```
parent(x)  = floor((x-1
```

array slot	0	1	2	3
priority	24	15	17	10



- **Sorted-list to heap**
  - **Keep extract max**
  - runtime : O(nlogn) → each extract max is logn
- **Unsorted-list to heap**
  - Use recursion
  - Runtime: O(n) because the height decreases

Union-find

Union-Find Summary

Path Compression **without** weighted union?

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	α(m, n)	α(m, n)

MST (every MST contains exactly V-1 edges)

***all spanning tree contains v - 1 edges**

- For every cycle
  - Max weight edge is not in the MST
- For every cut
  - Min weight edge is always in the MST
  - The minimum outgoing edge is always part of MST
- **Prim's**
  - **Each node is added/removed once → O(VlogV)**
  - **Each edge is visited once → O(ElogV)** (to decrease node
  - **Total: O(ElogV)**
- **Kruskal's**

Kruskal's Algorithm

```
// Sort edges and initialize
Edge[] sortedEdges = sort(G.E());
ArrayList<Edge> mstEdges = new ArrayList<Edge>();
UnionFind uf = new UnionFind(G.V());

// Iterate through all the edges, in order
for (int i=0; i<sortedEdges.length; i++) {
        Edge e = sortedEdges[i]; // get edge
        Node v = e.one(); // get node endpoints
        Node w = e.two();

        if (!uf.find(v,w)) { // in the same tree?
              mstEdges.add(e); // save edge
              uf.union(v,w); // combine trees
        }
}
```

- Use union-find
- If poll from queue and still not union → add the node in the MST then union
- Queue sort edges by weight
- union : each O(logV)
- Find: each O(logV)
- Sort the edges O(ElogE) = O(ElogV)
- Runtime: O(ElogV)
- Total: O(ElogV)

- **Boruvka's**
  - **Each Boruvka's step cost : O(V+E)**
    - **Store component identifier for all nodes O(V)**
    - **Find the min edges for each connected components using BFS/DFS + E)**
    - **Scan all nodes to merge components O(V)**
  - **There is at most O(logV) Boruvka's step**
  - Total: O(ElogV)

Steiner
- **Steps (This does not guarantee optimal Steiner Tree)**
  - For every vertex (v, w) calculate the minimum distance between those vertices → use Dijkstra's V times
  - Form a new graph from those vertices
  - Run MST on new graph

Kahn's
- **For topological ordering**
- **Queue stores** node that has 0 incoming edges

DYNAMIC PROGRAMMING

1. **Optimal substructure**
   a. **Optimal solution can be constructed from optimal solutions to smaller sub-problems**
2. **Overlapping subproblem**
   a. **The same smaller problem is used to solve multiple different bigger proble**

***Solve and memoize, only compute each solution once**