FA EXAM PART 1

Contents

1 Complexity Rules

1.1 Rules for Estimating O

- $O(c \cdot f(n)) = O(f(n))$
- $O(f_1(n)\, f_2(n)) = O(f_1(n)) \cdot O(f_2(n))$ (in nested loops)
- $O(f_1(n) + f_2(n)) = O(f_1(n)) + O(f_2(n))$ (in consecutive loops)

Complexities for Common Operations

- Searching: $O(\log n)$
- Selection: $O(n)$
- Sorting: $O(n \log n)$
- The base of log in CS is typically 2.

2 Complexity of Divide & Conquer Algorithms

- a: number of recursive calls,
- b: division factor of the input,
- c: degree of the polynomial describing the running time without recursive calls
- Typically, $b > 1$.
- $f(n)$: captures the non-recursive work per level (often $f(n) = n^c$).

2.1 Master Theorem

$$T(n) = \begin{cases} T_0 & \text{if } n < n_0, \\ a\, T\left(\frac{n}{b}\right) + n^c & \text{otherwise.} \end{cases}$$

Cases:

1. If $a < b^c$, then $T(n) = O(n^c)$.
2. If $a = b^c$, then $T(n) = O(n^c \log_b n)$.
3. If $a > b^c$, then $T(n) = O(n^{\log_b a})$.

2.2 Correctness

- **Partial correctness**: If the preconditions are met, the postcondition is correct when the algorithm terminates.
- **Total correctness**: Partial correctness + algorithm termination.

3 Sorting

3.1 Heaps

A **Heap** can be represented as an array, viewed logically as a complete binary tree, with:

- *Max-heap*: every node $\geq$ children.
- *Min-heap*: every node $\leq$ children.

3.1.1 Heapify – $O(\log n)$

```
Heapify(H, i):
  largest = index of max(H[i], H[left(i)], H[right(i)])
  if (largest != i): // one child larger than root at least
    swap H[i], H[largest] // swap w. largest child
    Heapify(H, largest) // continue down the heap
```

3.1.2 Build-Heap – $O(n)$

```
Build-Heap(H):
  heap_size[H] = length(H)
  for i = floor(length(H)/2) down to 1: // from the first non-leaf, to the root
    Heapify(H, i) // put node at index i as root to 2 heaps
```

3.1.3 HeapSort – $O(n \log n)$ – Optimal

```
HeapSort(H):
  Build-Heap(H)
  for i = length(H) downto 2: //do for all array positions from last to sec.
    swap H[1], H[i] //swap root with last element in current heap
    heap_size[H] = heap_size[H] - 1 //decrease heapsize
    Heapify(H, 1) //repair heap
```

3.1.4 Pop-Heap (delete) – $O(\log n)$

```
POP-HEAP(H):
  if heap_size[H] < 1: // empty heap
    return
  max = H[1] // save root (max)
  H[1] = H[heap_size[H]] // move bottom element to root
  heap_size[H] = heap_size[H] - 1 // decrease heap size
  Heapify(H, 1) // push element in root position down, to restore heap property
  return max
```

3.1.5 Push-Heap (insert) – $O(\log n)$

```
PUSH-HEAP(H, key):
  heap_size[H] = heap_size[H] + 1 // increase heap size
  H[heap_size[H]] = key
  i = heap_size[H]
  while (i > 1) and (H[parent(i)] < H[i]):
    swap H[i], H[parent(i)]
    i = parent(i)
```

3.1.6 Bottom-up vs. Top-down Heap Construction

- **Bottom-up**: $O(n)$ – sorting.
- **Top-down (successive insertions)**: $O(n \log n)$ – priority queues.

3.2 Quicksort

```
QuickSort(A, p, r):
  if p < r: // if non empty array
    q = Partition(A, p, r) //q is an index, boundary between
       the two partitions
    QuickSort(A, p, q)
    QuickSort(A, q+1, r)
```

- Best: $O(n \log n)$
- Average: $O(n \log n)$
- Worst: $O(n^2)$

3.3 Partition (Hoare)

```
Hoare-Partition(A, p, r):
  x = A[p]        // select pivot
  i = p - 1       // initialize i
  j = r + 1       // initialize j
  while true:
    repeat
      j = j - 1
    until A[j] <= x // find element larger than or equal to
       pivot
    repeat
      i = i + 1
    until A[i] >= x // find element smaller than or equal to
       pivot
    if i < j:
      exchange A[i] with A[j] // swap the elements
```

```
    else:
      return j // return the partition index
```

3.3.1 QuickSelect

```
QuickSelect(A, p, r, i):
// p- rst, r- last, i desired rank
  if p = r: // we are on the correct array position
    return A[p]
  q = Partition(A, p, r) // q- index where partition ends
  k = q - p + 1 //k- length of the <= partition
  if i <= k:
    return QuickSelect(A, p, q, i)
  else:
    return QuickSelect(A, q + 1, r, i - k)
```

Best/Average: $O(n)$,**Worst**: $O(n^2)$.

3.3.2 Median-of-Medians (Akl-Select)

A pivot-selection method guaranteeing $O(n)$ worst-case for selection.

```
    AklSelect(A[1..n], i):

Split the array into i sub-arrays of size a, each.
Directly sort each Ai, and find its median.
Generate the array of medians, and call AklSelect() on the
new array, to select the median of medians (M = m[n / 2a]).
Partition the input array into elements < M and >= M.
Assume there are k elements <= M.

  If i = k:
    return M
  If i < k:
    AklSelect(A[1..k-1], i)
  Else:
    AklSelect(A[k+1..n], i-k)
```

3.4 Quicksort Improvements: Randomization + Hybridization

Optimal

```
QuickSortV3(A, p, r):
  if (r - p) < someSmallThreshold:
    DirectSort(A, p, r)
  else:
    q = RandomPartition(A, p, r)
    QuickSortV3(A, p, q)
    QuickSortV3(A, q+1, r)
```

```

RandomPartition(A, p, r):
    i = random(p, r)
    exchange A[p] with A[i]
    return Hoare-Partition(A, p, r)
```

3.5 MergeSort – $O(n \log n)$

```
MergeSort(A, p, r):
  if p >= r:
    return
  q = (p + r) / 2
  MergeSort(A, p, q)
  MergeSort(A, q+1, r)
  Merge(A, p, q, r)
```

3.6 Counting Sort – $O(n + k)$, Stable

```
CountingSort(A, B, k):
  for i = 1 to k:
    C[i] = 0
  for j = 1 to length[A]:
    C[A[j]] = C[A[j]] + 1
  // C[j] = # of elements == j
  for j = 2 to k:
    C[j] = C[j] + C[j-1]
  // C[j] = # of elements <= j
  for j = length[A] downto 1:
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

3.7 Radix Sort

$O(d \cdot (n + b))$, where d is the number of digits, b is the base.

3.8 Bubble Sort – $O(n^2)$

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
```

```
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

3.9 Selection Sort – $O(n^2)$

```
void selectionSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        swap(arr[i], arr[min_idx]);
    }
}
```

3.10 Insertion Sort – $O(n^2)$ worst-case – $O(n)$ best-case

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

4 Elementary Data Structures (DS)

4.1 Stacks (LIFO)

Array-based:

```
Stack-Empty(S):
  if top[S] = 0:
```

```
    return true
  else:
    return false

Push(S, x):
  top[S] = top[S] + 1
  S[top[S]] = x

Pop(S):
  if Stack-Empty(S):
    // error stack underflow
  else:
    top[S] = top[S] - 1
    return S[top[S] + 1]
```

4.2 Queues (FIFO)

Circular array:

```
EnQ(Q, x):
  if (tail[Q] + 1) mod length[Q] = head[Q]:
    return "Queue Full"

  Q[tail[Q]] = x
  tail[Q] = (tail[Q] + 1) mod length[Q]

DeQ(Q):
  if head[Q] = tail[Q]:
    return "Empty"

  x = Q[head[Q]]
  head[Q] = (head[Q] + 1) mod length[Q]
  return x
```

4.3 Linked Lists

4.3.1 Search (Doubly Linked) – $O(n)$

```
List-Search(L, k):
  x = head[L]
  while x != nil and key[x] != k:
    x = next[x]
  return x
```

4.3.2 Insert (Doubly Linked) – $O(1)$

```
List-Insert(L, x):
  next[x] = head[L]
  if head[L] != nil:
    prev[head[L]] = x
  head[L] = x
  prev[x] = nil
```

4.3.3 Delete (Doubly Linked) – $O(1)$

```
List-Delete(L, x):
  if prev[x] != nil:
    next[prev[x]] = next[x]
  else:
    head[L] = next[x]

  if next[x] != nil:
    prev[next[x]] = prev[x]
  else:
  tail[x] = prev[x]
```

4.4 Arrays vs. Linked Lists: Summary

	Array	Linked List
Access	direct (index) $O(1)$	sequential $O(n)$
Insert	at end: $O(1)$ in middle: $O(n)$	at end: $O(1)$ in middle: $O(1)$ (except for search)
Delete	at end: $O(1)$ in middle: $O(n)$	at end: $O(1)$ in middle: $O(1)$ (except for search)
Space	data	data + pointers

5 Hash Tables

5.1 Collision Resolution Methods

- **Chaining**: each slot has a linked list of colliding keys.
- **Open Addressing**: if collision, probe within the table.

5.2 Chaining (Linked Lists)

- $h(k)$ = hash function
- $\alpha = n/m$ = load factor (average number of keys per slot)
- Expected: $O(1 + \alpha)$
- Worst: $O(n)$ (if all collide in one slot)

5.3 Open Addressing

average unsuccessful search time is $1/(1-\alpha)$
average successful search time is $1/\alpha * ln(1/(1-\alpha)$

5.3.1 Linear Probing

$$h(k,i) = (h'(k) + i) \bmod m$$

5.3.2 Quadratic Probing

$$h(k,i) = \left(h'(k) + c_1 \cdot i + c_2 \cdot i^2\right) \bmod m$$

5.3.3 Double Hashing

$$h(k,i) = \left(h'(k) + i \cdot h''(k)\right) \bmod m$$

5.3.4 Universal Hashing

Randomly choose from a family of hash functions at runtime.

6 Tree Data Structures and Advanced Operations

6.1 Basic Tree Operations

$$n = \text{number of nodes in } T, \quad h = \text{height of } T.$$

6.1.1 Common Operations

- **Traversal**: $O(n)$.
- **Search**:
  - General binary tree: $O(n)$.
  - BST: $O(h)$.
- **Insert**: $O(h)$
- **Min/Max/Predecessor/Successor (BST)**: $O(h)$
- **Remove**:
  - General binary tree: $O(n)$.
  - BST: $O(h)$.

6.1.2 Recursive Tree Traversals

```
tree_walk(x, order)
    if x != nil then
        if order = pre then
            write key[x]    // Preorder: process node first

        tree_walk(left[x], order)

        if order = in then
            write key[x]    // Inorder: process node between left and right

        tree_walk(right[x], order)

        if order = post then
            write key[x]    // Postorder: process node last
```

6.1.3 Iterative Traversal (Sketch)

```
printTree(T)
    d <- 1
    node <- root[T]
    repeat
        if d = 1 then
            // "Preorder" position
            if left[node] != NIL then
                d <- 1
                node <- left[node]
            else
                d <- 2
        else if d = 2 then
            // "Inorder" position
            if right[node] != NIL then
                d <- 1
                node <- right[node]
            else
                d <- 3
        else if d = 3
            // "Postorder" position
            if parent[node] != NIL then
                if node = left[parent[node]] then
                    d <- 2
                node <- parent[node]

    until (node = root[T] and d = 3)
```

6.2 Binary Search Trees (BST)

$$\forall x: \text{ keys in left}[x] < \text{key}[x] < \text{keys in right}[x].$$

6.2.1 BST Search

```
r_tree_search(x, k) //x=root; k=searched
    if x = nil or k = key[x] then
        return x
    else if k < key[x] then
        return r_tree_search(left[x], k)
    else
        return r_tree_search(right[x], k)
```

6.2.2 BST Insert

```
tree_insert(T, z) //x=root; z=new node, already allocated
    y <- nil  // y= x's parent; stays 1 step behind x;
    x <- root[T]
    while x != nil //loop to find the position to insert
        do y <- x // y=x at the prev step
        if key[z] < key[x] then
            x <- left[x]
        else
            x <- right[x]

    parent[z] <- y  // position found; x=nil; y=new node
       (z)'s parent
    if y = nil then //if the tree was empty before this
        root[T] <- z
    else if key[z] < key[y] then
        left[y] <- z
    else
        right[y] <- z
```

6.2.3 BST Delete

```
tree_delete(T, z)
    if left[z] = nil or right[z] = nil then //z=node to
       delete; y physically deleted
        y <- z //Case 1 OR 2; z has at most 1 child =>
           del z
    else
        y <- tree_successor(z) //find replacement=min(
           right)

    if left[y] != nil then //find replacement=min(right)
        x <- left[y] //y has no child to the right; x=y's
            child
    else
        x <- right[y]

    if x != nil then //y is not a leaf;
        parent[x] <- parent[y] // y's child redirected to
            y's parent = x's parent

    if parent[y] = nil then //means y were the roo
        root[T] <- x //y's child becomes the new root
    else if y = left[parent[y]] then //link y's parent to
        x which becomes its child
        left[parent[y]] <- x
    else
```

```
        right[parent[y]] <- x

    if y != z then
        key[z] <- key[y]

    return y //outside the procedure: copy y's info into
        z; dealloc y
```

7 Find-Min and Find-Max Operations

7.1 Finding the Minimum $O(h)$

```
find_tree_min(x)
    while left[x] != nil
        x <- left[x]
    return x
```

7.2 Finding the Maximum $O(h)$

```
find_tree_max(x)
    while right[x] != nil
        x <- right[x]
    return x
```

8 Finding Predecessor and Successor $O(h)$

The predecessor and successor of a node are defined based on the in-order traversal.

- **Predecessor** (pred): The maximum node in the left subtree of x.
- **Successor** (succ): The minimum node in the right subtree of x.

```
find_pred(x)
    return find_tree_max(left[x])

find_succ(x)
    return find_tree_min(right[x])
```

8.1 Find-Succ Code (all cases)

```
find_tree_successor(x)
    if right[x] != nil // Regular case; the succ belongs to the same subtree
        return find_tree_min(right[x])

    y <- p[x] // y keeps a pointer one level above x
    while y != nil and x == right[y] // Traverse upwards until x is a left child
        x <- y
        y <- p[y]
    return y
```

- Finding the predecessor is symmetric: replace `right` with `left` and `min` with `max`.

9 Perfect Balanced Trees (PBT)

- **Balance Condition:** Balance refers to the number of nodes, not the heights of the subtrees.

$$h = \log n$$

- **Insertion Complexity:**
  - Insert as in a regular BST: $O(h) = O(\log n)$.
  - Requires up to n rotations to rebalance.
  - Overall complexity: $O(n)$.
- **Deletion Complexity:**
  - Delete as in a regular BST: $O(h) = O(\log n)$.
  - Requires up to n rotations to rebalance.
  - Overall complexity: $O(n)$.

10 AVL Trees

- **Insertion Complexity:**
  - Insert as in a regular BST: $O(h) = O(\log n)$.
  - Requires at most $1/2$ rotations: $O(1)$.
- **Deletion Complexity:**
  - Delete as in a regular BST: $O(h) = O(\log n)$.
  - Requires at most $O(\log n)$ rotations.
- **Height Property:**

$$h \leq 1.45 \log n$$

- **Maintenance:**
  - Easy to maintain for insertion.
  - Structure ensures $O(h) = O(\log n)$ time complexity.
  - Requires at most $O(\log n)$ rotations to preserve balance.
- **Balance Property:** Ensure that the balance factor remains within $\{-1, 0, 1\}$.
- **Self-Balancing:**
  - **Single Rotation:** Simple rotation to rebalance the tree. $O(1)$
  - **Double Rotation:** Combination of rotations to restore balance. $O(1)$
- **Post-Insertion:** After an insertion, at most one rotation is required to maintain balance.

11 Order Statistic (OS) Tree

$$\dim[x] = \dim[\text{left}[x]] + \dim[\text{right}[x]] + 1$$

11.1 OS_Select Operation $O(h)$

```
OS_Select(x, i)
    r <- dim[left[x]] + 1 // Number of nodes in the left
       subtree + root
    if i == r then
        return x
    else if i < r then
        // The ith smallest is in the left subtree
        return OS_Select(left[x], i)
    else
        // The ith smallest is in the right subtree
        return OS_Select(right[x], i - r)
```

Case #1: Node is a right child of its parent

$$\text{rank}(\text{Key2}) = \dim(\text{RL}) + 1 + \dim(\text{L}) + 1$$

Case #2: Node is a left child of its parent

$$\text{rank}(\text{Key1}) = \dim(\text{LL}) + 1$$

11.2 OS_Rank Operation $O(h)$

```
OS_Rank(T, x)
    r <- dim[left[x]] + 1
    y <- x
    while y != root[T] do
```

```
        if y == right[parent[y]] then
            r <- r + dim[left[parent[y]]] + 1
        y <- parent[y]
    return r
```

11.3 Augmented Trees with Successor/Predecessor (Type 2)

- Each node has `succ` and `pred`.
- Forms a doubly linked list of nodes in sorted order.
- **O(1)** for successor/predecessor/min/max.
- BST Insert/Delete still $O(h)$ but must fix `succ`/`pred`.

12 Augmented Trees – Insert O(h) = O(log n)

Perform the standard BST insertion for node x.

```
if x = right[p[x]] then // node inserted = right child
    pp[x] <- succ[p[x]]
    dl_list_ins_after(p[x], x)
```

```
else // node inserted = left child
    pp[x] <- pred[p[x]]
    dl_list_ins_after(pp[x], x)
```

13 Augmented Trees – Delete O(h) = O(log n)

Apply the regular delete operation in a BST.

```
if right[y] = nil then // no child to the right
    x <- left[y]    // x = y's only child
    while x != nil do // along x's right branch
        pp[x] <- pp[y] // update pp
        x <- right[x]
    dl_list_del(y)
else // symmetric on the left
    x <- right[y]
    while x != nil do
        pp[x] <- pp[y]
        x <- left[x]
    dl_list_del(y)
```

14 Augmented Trees – Min

```
if x = left[p[x]] then
    return succ[pp[x]] // on the leftmost branch, pp[x]=nil
else
    return succ[p[x]]
```

15 Augmented Trees – Max

```
if x = left[p[x]] then
    return pred[p[x]] // on the rightmost branch, pp[x]=nil
else
    return pred[pp[x]]
```

16 Red-black trees O(log n)

- Each node is either **red** or **black**.
- The root of the tree is always **black**.
- All leaves (represented as `NIL` nodes) are **black**.
- If a node is **red**, then both of its children are **black**.
- For each node, all paths from the node to its descendant leaves contain the same number of black nodes.

16.1 Insertion O(log n)

1. **Case 1:** Uncle is **red**.
    - Recolor the parent and uncle to **black**, and the grandparent to **red**.
    - Move up the tree and continue fixing from the grandparent.
2. **Case 2:** Uncle is **black** and the new node is on the **opposite side**.
    - Perform a **rotation** to align the tree for Case 3.
3. **Case 3:** Uncle is **black** and the new node is on the **same side**.
    - Perform a **rotation** and **recoloring** to fix the properties.

16.2 Deletion O(log n)

1. **Case 1:** Sibling is **red**.
   - Perform a rotation and recolor to transform the situation into one of the other cases.
2. **Case 2:** Sibling is **black** and both of sibling's children are **black**.
   - Recolor the sibling to **red** and move up the tree.
3. **Case 3:** Sibling is **black**, sibling's left child is **red**, and sibling's right child is **black**.
   - Perform a rotation and recolor to prepare for Case 4.
4. **Case 4:** Sibling is **black** and sibling's right child is **red**.
   - Perform a rotation and recolor to fix the properties.

16.3 Rotations O(1)

```
left_rotate(T, x):
    y <- right[x]                // y saves Q
    right[x] <- left[y]          // right of P goes on BRB-insert
    if left[y] != NIL then
        p[left[y]] <- x
    p[y] <- p[x]                 // Q's parent becomes P's parent
    if p[x] == NIL then
        root[T] <- y
    else if x == left[p[x]] then
        left[p[x]] <- y
    else
        right[p[x]] <- y
    left[y] <- x                  // P becomes the left child of Q
    p[x] <- y                     // Q becomes the parent of P
```

16.4 Disjoint Sets (Union-Find)

```
Find_Set(x)
    if x != p[x] then
        p[x] <- Find_Set(p[x])
    return p[x]

Union(x, y)
    rx <- Find_Set(x)
    ry <- Find_Set(y)
```

```
    if rx != ry then
        if rank[rx] > rank[ry] then
            p[ry] <- rx
        else
            p[rx] <- ry
            if rank[rx] = rank[ry] then
                rank[ry] <- rank[ry] + 1
```

16.5 Binomial Trees and Binomial Heaps

16.5.1 Binomial Trees

- B_k: 2^k nodes, height k.

16.5.2 Binomial Heaps

- A set of binomial trees with at most one tree of each degree.
- Operations (Find-Min, Union, Extract-Min, Key-Delete) typically $O(\log n)$.
- Make-heap $O(1)$

16.6 Fibonacci Heaps

- More relaxed structure than binomial heaps.
- Multiple trees of same degree allowed.
- **Amortized**:
  - Insert: $O(1)$
  - Find-Min: $O(1)$
  - Union: $O(1)$
  - Extract-Min: $O(\log n)$
  - Decrease-Key: $O(1)$

16.7 B-Trees

- Minimum degree t:
  - Each node has at most $2t - 1$ keys and at least $t - 1$ keys (except root).
  - Height is $O(\log n)$ with big branching factor.

16.7.1 B-Tree Search/Insert/Delete

- **Search**: $O(t \cdot h)$ but typically $O(\log n)$ with large t.
- **Insert**: split any full node on the top-down path, then insert key into leaf.
- **Delete**: if a node would underflow ($t - 1$ keys), fix by borrowing or merging with sibling.

16.7.2 B-Tree Delete Cases

1. Key in leaf with enough keys: remove directly.
2. Key in internal node: swap with predecessor/successor in a leaf, then remove from leaf.
3. Underflow ($< t - 1$):
   - Borrow from sibling with $\geq t$ keys,
   - Or merge with sibling if both have $t - 1$ keys.

16.8 Complexity Summary (Trees and Heaps)

Data Structure	Search	Insert	Delete	Find-Min/Max
BST (unbalanced)	$O(h)$	$O(h)$	$O(h)$	$O(h)$
AVL / RB Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binomial Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\text{min} : O(\log n)$
Fibonacci Heap	$O(1)$ (amort.)	$O(1)$ (amort.)	$O(\log n)$	$\text{min} : O(1)$
B-Tree (degree t)	$O(t \cdot h)$	$O(t \cdot h)$	$O(t \cdot h)$	$O(t \cdot h)$