CS2040S Cheatsheet AY23/24 —— @Jin Hang

Asymptotic

Recurrence Relation

$T(n) = T(n-1) + O\left(n^k\right)$	$O\left(n^{k+1}\right)$
$T(n) = T(n-1) + O(\log n)$	$O(n \log n)$
$T(n) = T(n-1) + O(n \log n)$	$O(n^2 \log n)$
$T(n) = T(n/k) + O(1)$	$O(\log n)$
$T(n) = T\left(\frac{n}{2}\right) + O(n)$	$O(n)$
$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$
$T(n) = aT\left(\frac{n}{a}\right) + O(n)$	$O(n \log n)$
$T(n) = aT(n-1) + O(1)$	$O\left(a^n\right)$

Master Theorem: For $T(n) = aT(n/b) + f(n)$
1. $\exists \epsilon > 0$ s.t. $f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $\exists k \geq 0$ s.t. $f(n) = \Theta(n^{\log_b a} \lg^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
3. $\exists \epsilon > 0$ s.t. $f(n) = \Omega(n^{\log_b a + \epsilon})$
   if $\exists c < 1$ s.t. $af(n/b) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$

Order of Big-O Notation: $O(1) < O(\log(\log n)) < O(\log n) < O(\log^m n) < O(n^k) < O(2^n) < O(n!) < O(n^n)$
$\log n! = \sum_{i=0}^{n-1} \log(n-i) \leq O(n \log n)$
$n \log m + m \log n = O(n \log m + m \log n)$

- $T(n) = O(f(n))$ if $\exists c > 0, n_0 > 0$ s.t. $n > n_0$: $T(n) \leq cf(n)$
- $T(n) = \Omega(f(n))$ if $\exists c > 0, n_0 > 0$ s.t. $\forall n > n_0$: $T(n) \geq cf(n)$
  For $T(n) = O(n^2) \rightarrow T(n) = \Omega(n)$ or $\Omega(n^2)$
- $T(n) = \Theta(f(n))$ iff. $T(n) = O(f(n)) \& T(n) = \Omega(f(n))$

Precondition: Fact that is true when the function begins.
Postcondition: Fact that is true when the function ends.
Loop Invariant: relationship between variables that is true at the beginning (or end) of each iteration of a loop.

Binary Search

Functionality: If element is in the array, return index of element, **else**, return -1.
Preconditions: 1. Array is of size n **2.**Array is sorted
Postcondition: If element is in the array, `A[begin] = key`
invariant 1. (Correctness) `A[begin] <= key <= A[end]`
2. (Performance) (end - begin) $<= n/2k$ in iteration k.

Peak Finding (Find local maximum)

1-D Peak Finding $T(n) = T(n/2) + O(1) = O(\log n)$
Output a local maximum in A, where `A[i-1] <= A[i]` and `A[i+1] <= A[i]`. Assume that `A[-1] = A[n] = -MAX_INT`

```
FindPeak(A, n)
if A[n/2] is a peak then return n/2
else if A[n/2+1] > A[n/2] then Search for peak in right half.
else if A[n/2−1] > A[n/2] then Search for peak in left half.
```

Property: If we recurse in the right half, then there exists a peak in the right half.
Correctness: 1. There exists a peak in the range [begin, end]
2. Every peak in [begin, end] is a peak in $[1, n]$

2-D Peak Finding

Output: a peak in A[n,m] that is not < (at most) 4 neighbors.
Algorithm 1 $O(n \log m)$
1. Each column requires $O(n)$ time to find max
2. 1-D Peak Finding Algo. for m columns: $O(\log m)$

Algorithm 2 Divide-and-Conquer

```
Find MAX element of middle column.
if found a peak, done.
else if (left neighbor is larger), then recurse on left half.
else if (right neighbor is larger), then recurse on right half.
```

Running time: $T(n,m) = T(n, m/2) + O(n) = O(nlogm)$
Algorithm 3 (Most Efficient) Reduce-and-Conquer

```
Find MAX element on border + cross.
if found a peak, DONE.
else: Recurse on quadrant containing element bigger than MAX.
```

Running time $T(n,m) = T(n/2, m/2) + O(n+m) = n\sum_{i=1}^{k} \frac{1}{2^i} + \sum_{i=1}^{k} \frac{1}{2^i} \leq 2n + 2m = O(n+m)$

Sorting Jumble

Bubble Sort

Method: Repeatedly swaps adjacent elements that are out of order until there are no swaps in an iteration, or after n iterations.
invariant: At the end of iteration j, the **biggest** j items are correctly sorted in the **final** j positions of the array.
Time Complexity
- Best Case: $O(n)$ for ascend or already sorted array.
- Worst Case: $O(n^2)$ for descend array
- Average Case: $O(n^2)$, assume inputs are chosen at random

Space Complexity: $O(1)$
Satbility: Stable, only swap element that are different.
Applicable conditions: Traditional bubble sorting is generally not used or is not used directly. But if the array is already sorted and there is nothing else to choose, we can use bubble sort.

Selection Sort

Method: Maintains a sorted prefix, and repeatedly finds the **smallest** element in the unsorted remainder and swaps it with the first element in the remainder.
invariant: At the end of iteration j, the **smallest** j items are correctly sorted in the **first** j positions of the array.
Complexity: $T(n) = O(n^2)$ for all cases || $S(n) = 1$
Satbility: Unstable. As there is **swap** in sort i.e. 2**2**1 → 12**2**

Insertion Sort

invariant: After j iteration, **First** j items are sorted.(不一定是最小的j个元素) **And** rest elem. remain its position
Time Complexity
- Best Case: $O(n)$ for already sorted array.
- Worst Case: $O(n^2)$ for inverse sorted array
- Average Case: $O(n^2)$, assume inputs are chosen at random

Space Complexity: $O(1)$
Satbility: Stable as long as we implement it properly, i.e. `while(i > 0) and (a[i] > key)`(始终是不等关系，防止相同元素乱序)
Applicable conditions: When the list is mostly sorted, InsertionSort is faster than MergeSort.
- limited space
- mostly sorted
- when size n is small (i.e. $n <= 50$)

Merge Sort

Method
1. Divide: split array into two halves.
2. Recurse: sort the two halves.
3. Combine: merge the two sorted halves.

Merge Running Time: Given two lists: A of size $n/2$ and B of size $n/2$. In each iteration, move one element to final list. After n iterations, all the items are in the final list. Each iteration takes $O(1)$ time to compare two elements and copy one. $O(n) = cn$
Time Complexity: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$
Space Complexity: $O(n)$ for **Min Space** Method
Satbility: Stable
Applicable conditions:
- large space ($S(n) = O(n)$)
- large size
- Note that we may also choose Quick Sort under this situation

It involves a certain amount of data movement, so we may combine it with insertion sort to first obtain a sequence of a certain length and then merge it, which will improve efficiency.
Min Space: Use only one temporary array to min the amount of extra space needed. On termination, items in range [begin,end] are sorted in A. The tempArray is used for workspace.

```
MergeSort(A, begin, end, tempArray)
    if (begin=end) then return;
    else:
        mid = begin + (end−begin)/2
        MergeSort(A, begin, mid, tempArray);
        MergeSort(A, mid+1, end, tempArray);
        Merge(A[begin..mid], A[mid+1, end], tempArray);
Copy(tempArray, A, begin, end);
```

Hence, $S(n) = 2S(n/2) + O(1) = O(n)$
Iterative: sort the array in groups of power of 2, sort in pairs then merge into 4's, 8's, and so on.

Quick Sort

Method
- Divide: Partition the array into two **sub-arrays** around a pivot x s.t. elem.s in lower subarray $\leq x \leq$ elem.s in upper subarray.
- Conquer: Recursively sort the two sub-arrays.
- Combine: Trivial, do nothing.

Partition: $O(n)$
1. **Invariants:** A[high] > pivot at the end of each loop
   At the end of every loop iteration:
   - For $1 < i <$ low: $B[i] <$ pivot; For $j >$ high : $B[j] >$ pivot
   - In the end, every elem. from A is copied to B $\Rightarrow B[i] =$ pivot.
2. **Choose the pivot:** As long as we have a **fixed** pivot choice, $T(n) = O(n^2)$ as always possible to find a bad input.
   - first element: $A[1]$
   - middle element: $A[n/2]$
   - last element: $A[n]$
   - median of $(A[1], A[n/2], A[n])$

   Good Pivot: A pivot is good if it divides the array into two pieces, each of which is size at least $n/10$. $\Rightarrow$ **Paranoid Q.S.**
   Probability of choosing a good pivot: $p = 8/10$
   Expected number of times to repeatedly choose a pivot to achieve a good pivot: $E(X) = 1/p = 1/(8/10) = 10/8 < 2$

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

Space Complexity: $O(n \log n)$ **Satbility:** Unstable
Paranoid QuickSort Method: Every time we recurse, we reduce the problem size by at least (1/10).

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else repeat
        pIndex = random(1, n)
        p = partition(A[1..n], n, pIndex)
    until p > (1/10)n and p < (9/10)n
        x = QuickSort(A[1:p−1], p−1)
        y = QuickSort(A[p+1:n], n−p)
```

Time Complexity: Note that **partitation** will cost O(n) time, hence basicly $T(n) <= T(9/10n) + T(1/10n) + O(n) = O(n \log n)$
More generally, for $k \in [1/10n, 9/10n]$, we have
$T(n) = T(k) + T(n-k) + T(1) = T(k) + T(n-k) + O(n) = O(n \log n)$
Space Complexity: $O(n \log n)$
Satbility: All partition are **unstable**, make stable by using associating array to record the total order of elements
3-way Partition:

< pivot	= pivot	> pivot

Duplicate Elem In an array of size n with $k(k < n)$ kinds of distinct keys, 3-way balanced partition: $O(n \log k)$, worst case $O(nk)$

Number of Pivots $O(n \log k)$ for Partition with k pivots

1. $O(k \log k)$ time to sort the pivots (e.g. using MergeSort)
2. $O(n \log k)$ time to place each item in the correct bucket (e.g. via binary search among the pivots).

Hence, for QuickSort of n elem. with k pivots:
$T(n) = kT(\frac{n}{k}) + O(n \log k) = O(n \log k) = O(n \log n)$

Quick Select $T(n) = T(n/k) + O(n) = O(n)$ $S(n) = O(\log n)$

Tree

BST Impt. Property:

- all in left sub-tree < key < all in right sub-right
- same keys ≠ same shape, order of insrertion determine the shape of tree [ways of insertion $(n!)$ > shapes of BST(about 4^n)]
- On a **balanced** BST, all operations run in $O(\log n)$ time
- A BST is **balanced** if $h = O(\log n)$
- Node $n \le \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ for all BST

Height: Number of edges on longest path from root to leaf.

- $h(v) = 0$ (if v is a leaf)
- $h(v) = \max(\text{h(v.left), h(v.right)}) + 1$
- $\log n + 1 - 1 = O(\log n) \le h \le n$, for tree with total n nodes.

Operation

- Search: $O(h)$ • Insert: $O(h)$ • Traverse: $O(n)$
- Successor Queries: $O(h)$

```
Search for key in the tree.
if (result > key), then return result.
if (result <= key), then search for successor of result.
node has a right child: successor(Node) = right.searchMin()
node has no right child: Find its next in order node
```

- Delete: $O(h)$
  1. **No children:** delete node directly
  2. **1 child:** Remove the node and link its parent and child node
  3. **2 children:** Find `successor(Node)` → swap it with its successor. → Delete node v

  Claim: successor of deleted node has at most 1 child

AVL Trees (* H.B. = height-balance for following)

Invariant: A node v is **H.B.** if |v.left.height – v.right.height| ≤ 1

Claim: A **H.B.** tree with n nodes has at most height $h < 2\log(n)$.

Lemma 1: A **H.B.** tree with height h has at least $n > 2^{\frac{h}{2}}$ nodes

proof. Let n_h be **min. num.** of nodes in a **H.B.** tree of height h.
$$n_h \ge 1 + n_{h-1} + n_{h-2} \ge 2n_{h-2} \ge \ldots \ge 2^k n_0$$
where $h - 2k = 0 \Rightarrow k = \frac{h}{2}$. Hence, $h < 2\log(n)$

Lemma 2: 高度为$h(h \ge 1)$的AVL树最少节点数递推公式
$$S(h) = S(h-1) + S(h-2) + 1 \Rightarrow S(h) = \text{Fib.}(h+2) - 1$$

Tree Rotations

- Only fix lowest out-of-balance node.
- Only need at most two rotations to fix
- Using rotations, you can create every possible "tree shape."

1. **LL Rotation:** 插入左孩子的左子树
3. **RR Rotation:** 插入右孩子的右子树
2. **LR Rotation:** 插入左孩子的右子树
4. **RL Rotation:** 插入右孩子的左子树

Delete: After deletion, for every ancestor of the deleted node

1. Check if it is height-balanced
2. If not, perform a rotation
3. Continue to the root
4. Up to $O(\log n)$ Rotations

Order Statistics !**Weight:** size of the tree rooted at that node.

- w(leaf) = 1 • **rank** $= w_{左} + 1$ • w(v) $= w_{左} + w_{右} + 1$

Select(k): $O(h) \Leftrightarrow O(\log n)$

```
rank = m.left.weight + 1;
    if (k == rank) then return v;
    else if (k < rank) then return m.left.select(k);
    else if (k > rank) then return m.right.select(k−rank);
```

rank(node): $O(h) \Leftrightarrow O(\log n)$

```
rank = node.left.weight + 1;
while (node != null) do
    if (node is left child) then do nothing
    else if (node is right child) then
        rank += node.parent.left.weight + 1;
    node = node.parent;
return rank;
```

Insert/Delete

1. Insert/Delete
2. 节点→根遍历
3. 路径中所有节点 v.weight+1($O(\log n)$)
4. 翻转调整AVL树
5. 翻转后更新节点 weight ($O(1)$)

Maintain weight during rotations: $O(1)$ Time (翻转后只用改两个)

Interval Queries We need to maintain MAX after every rotation

1. **Search** for interval: $O(\log n)$

```
c = root;
while (c != null && x is not in c.interval) do
    if (c.left == null) then c = c.right;
    else if (x > c.left.max) then c = c.right;
    else c = c.left;
return c.interval;
```

2. **Search** for all interval that overlap the node: $O(k \log n)$ for k overlapping intervals (Best Sol.: $O(k + \log n)$, Not Cover now)

```
Repeat until no more intervals:
    1. Search for interval.
    2. Add to list.
    3. Delete interval.
Repeat for all intervals on list: Add interval back to tree.
```

3. **Insert / Delete**: After insert / delete, Conduct Rotation if the tree is out of balance ⇒ maintain MAX after every rotation.

Claims of Interval Search

1. If search goes right, then no overlap in left subtree.
2. If search in left subtree fails, then search also would fail in right subtree! ⇔ If search goes left and fails, then key < every interval in right sub-tree.
3. Either search finds key in subtree or it is not in the tree.

Orthogonal Range Searching $S(n) = O(n)$

Strategy: Preprocessing (buildtree): $O(n \log n)$

1. Use a binary search tree.
2. Store all points in the leaves of the tree. (Internal节点只存拷贝)
3. Each internal node v stores the MAX of any leaf in left subtree.

Operations

1. FindSplit(low, high): $O(\log n)$, find split node.

```
v = root;    done = false;
while !done {
    if (high <= v.key) then v=v.left;
    else if (low > v.key) then v=v.right;
    else (done = true);
}
return v;
```

2. LeftTraversal(v, low, high): $O(\log n + k)$, Left Traverse. At every step, we either:
   - Output all right sub-tree and recurse left: $O(k) + O(\log n)$
   - Recurse right: $O(\log n)$

```
if (low <= v.key)
    All_Leaf_Traversal(v.right);
    LeftTraversal(v.left, low, high);
else LeftTraversal(v.right, low, high);
```

3. RightTraversal(v, low, high)

```
if (v.key <= high)
    All_Leaf_Traverasal(v.left);
    RightTraversal(v.right, low, high);
else RightTraversal(v.left, low, high);
```

Invariant: The search interval for a left-traversal at node v includes the **maximum** item in the subtree rooted at v.

Dynamic: Need to fix rotations after inster and delete operations

(a,b)-Tree $2 \le a \le (b+1)/2$ | **Rule 1: (a,b)-child policy**

Node type	#Keys		#Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

Rule 3: Depth All leaf nodes must all be at the same depth.

Property An (a,b)-tree is balanced with $\log_b n \le h \le \log_a n$

Operations

1. **Search:** An (a, b)-tree with n nodes has $O(\log_a n)$ height. → Binary search for a key at every node takes $O(\log_2 b)$ time ⇒ $O(\log_a n \cdot \log_2 b) = O(\log n)$
2. **Split:** Find medиean v_m, split LHS($v < v_m$) and insert v_m to the parent node. For split operation, we need to copy $\frac{b}{2}$ elements from one node to other and cost b to insert a key into a key list ⇒ $O(b)$.
3. **Insert:** $O(b \cdot \log_a n) = O(\log_a n)$
4. **Delete:** $O(\log_a n)$
5. **Merge and Share:** $O(b)$

```
1. Search for node w which contains key x
2. If w is not a leaf:
    1. Determine predecessor key px and node pw
    2. Swap(w, x, pw, px)
    3. Update w as pw
3. Delete key x from w
4. Repeat until w satisfy rule 1
    1. Merge/Share w with its smallest adjacent sibling
    2. Update w as its parent
```

kd-Tree

Operations

1. **Search:** $O(\log n)$, If it is a horizontal / vertical split, then compare the x / y to the split value, and branch left or right.
2. **Build:** Use QuickSelect to find median of the data by the x or y as the split value, and then partition the points among the left and right children. $T(n) = 2T(n/2) + O(n) = O(n \log n)$
3. **Find the minimum** x**:**
   - horizontal split ⇒ recurse on the left child
   - vertical node ⇒ recurse on both children (minimum could be in either the top half or the bottom half)

   $T(n) = 2T(\frac{n}{4}) + O(1) = O(\sqrt{n})$

Supplyment

Mathematical

- $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$
- $\frac{\log_b n}{\log_a n} = \log_b a$
- $E[X] = \frac{1}{p}$

Queick Sort

After every partition, the pivot should be in the correct position

1. 找选项中第一个元素在他应该在的位置p
2. 检查是否符合$e_{上} > p \quad \& \quad e_{下} < p$

Partition with pIndex

1. swap(pIndex, 0)
2. start after pivot in A[0]
3. Define: A[n+1] = $+\infty$
4. Partition like before

Duplicates Elem.

one-way Q.S. $O(n^2)$, Every partition arr divided to $[1 : n-1]$

Partition

```
partition(A[1..n], n, pIndex)
pivot = A[pIndex];
swap(A[1], A[pIndex]);
low = 2;
high = 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;
```

Tree

Perfectly balanced: Both children of each node have an equal number of nodes and are perfectly balanced.

String

- Compare 2 Strings: $O(L_{\max})$
- Append 2 Strings: $O(L_1 + L_2)$