CS2040S Data Structures and Algorithms

AY 24/25 Sem 2 — github/omgeta

1. Asymptotic Analysis

Preconditions are conditions which must be true for the algorithm to work correctly. Postconditions are conditions guaranteed to be true after the algorithm ends.

Invariants are conditions that remain unchanged and consistently true throughout the algorithm.
Loop invariants are invariants for each loop iteration.

Runtime $T(n)$ is given if $\exists c, n_0 > 0$ s.t. $\forall n > n_0$:

i. $T(n) \leq cf(n) \iff T(n) = O(f(n))$
ii. $T(n) \geq cf(n) \iff T(n) = \Omega(f(n))$
iii. $T(n) = O(f(n)), \Omega(f(n)) \iff T(n) = \Theta(f(n))$

Order of growth is:
$O(1) < O(\log\log n) < O(\log n) < O(\log^2 n) < O(\sqrt{n}) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(2^{2n}) < O(n!)$

Common recurrences:

i. $T(n) = T(n/2) + O(1) = O(\log n)$
ii. $T(n) = T(n/2) + O(n) = O(n)$
iii. $T(n) = 2T(n/2) + O(1) = O(n)$
iv. $T(n) = 2T(n/2) + O(n) = O(n\log n)$
v. $T(n) = T(n-1) + O(1) = O(n)$
vi. $T(n) = T(n-1) + O(n) = O(n^2)$
vii. $T(n) = 2T(n-1) + O(1) = O(2^n)$

Mathematical Properties

Useful properties:

i. $\log(xy) = \log x + \log y$ (Product rule)
ii. $\log(x/y) = \log x - \log y$ (Quotient rule)
iii. $\log(x^k) = k\log x$ (Power rule)
iv. $\log_b x = \frac{\log_a x}{\log_a b}$ (Change of base)
v. $x^a \cdot x^b = x^{a+b}$ (Product rule)
vi. $\frac{x^a}{x^b} = x^{a-b}$ (Quotient rule)
vii. $(x^a)^b = x^{ab}$ (Power rule)

Master Theorem

Dividing function $T(n) = aT(n/b) + f(n)$ where $a > 1, b > 1, f(n) = \Theta(n^k \log^p n)$ has cases:

i. $\log_b a > k \implies T(n) = \Theta(n^{\log_b a})$
ii. $\log_b a = k \wedge p > -1 \implies T(n) = \Theta(n^k \log^{p+1} n)$
$\wedge\, p = -1 \implies T(n) = \Theta(n^k \log\log n)$
$\wedge\, p < -1 \implies T(n) = \Theta(n^k)$
iii. $\log_b a < k \wedge p \geq 0 \implies T(n) = \Theta(n^k \log^p n)$
$\wedge\, p < 0 \implies T(n) = \Theta(n^k)$

Decreasing function $T(n) = aT(n-b) + f(n)$ where $a > 0, b > 0$ has cases:

i. $a < 1 \implies T(n) = O(f(n))$
ii. $a = 1 \implies T(n) = O(n \cdot f(n))$
iii. $a > 1 \implies T(n) = O(a^{n/b} \cdot f(n))$

2. Searching

Binary search, $O(\log n)$, in a sorted array cuts search space in half until the target is found or range exhausted.

```
int binarySearch(T[] arr, T x) {
  int lo = 0, hi = arr.length - 1;
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] < x) lo = mid + 1;
    else hi = mid;
  }
  return lo;
}
```

1D Peak Finding, $O(\log n)$:

i. Binary search on side with rising neighbour if not peak

2D Peak Finding on $n \times m$, $O(n \log m)$:

i. Take global max of column, binary search on columns with rising neighbour if not peak.
ii. Optimise: Quadrant DnC. $O(n+m)$.

kth Smallest, $O(n)$:

i. QuickSelect, recurse on half with k

3. Sorting

BubbleSort repeatedly swaps inverted adjacent elements up to form a globally sorted suffix.

i. After i iterations, last i elements are correct
ii. Time: $O(n)$ (Best), $O(n^2)$ (Worst)
iii. In-place and stable

SelectionSort selects the smallest unsorted element and places it in the globally sorted prefix.

i. After i iterations, first i elements are correct
ii. Time: $O(n^2)$ (Best & Worst)
iii. In-place but unstable

InsertionSort adds elements to a locally sorted prefix.

i. After i iterations, first i elements are sorted
ii. Time: $O(n)$ (Best), $O(n^2)$ (Worst)
iii. In-place and stable

MergeSort divides the array into halves, sorts each half, and merges the sorted halves.

i. Locally sorted prefix in power of 2
ii. Time: $O(n\log n)$ (Best & Worst)
iii. Space: $O(n\log n)$
iv. Out-of-place and stable

QuickSort partitions the array around $\leq$ and $>$ pivot, then recursing on both partitions.

i. Elements before pivot are $\leq$, after pivot are $>$
ii. Time: $O(n\log n)$ (Best), $O(n^2)$ (Worst)
iii. In-place but unstable
iv. Optimise: randomized pivot, 3-way partition for $O(n\log n)$ worst-case, insertion sort for small n

CountingSort counts number of objects with distinct keys.

i. Time: $O(n+k)$ where k is number of keys
ii. Space: $O(n+k)$

RadixSort sorts integers by individual digits.

i. Time: $O(nd)$ where d is number of digits
ii. Space: $O(n+k)$

4. Trees

Binary Search Trees (BSTs)

Binary search tree maintains that for any node, all left descendants are $\leq$ and all right descendants are $>$.
Operations are $O(h)$ where h is the height of the tree.

BSTs are balanced if $h = O(\log n)$.
If balanced, (non-traversal) operations are $O(\log n)$. In the worst case, they are $O(n)$.

i. Height, h is given by $\log n - 1 \leq h \leq n$
ii. `search`: Traverse left/right based on comparisons.
iii. `insert`: Find position via search and add node.
iv. `delete`: If not leaf, replace with successor.
v. Traversal yields sorted elements. Time: $O(n)$

AVL Trees

Self-balancing BSTs which are height-balanced, i.e. height difference between left/right subtrees differ by at most 1.

i. Balance factor $= |height_{right} - height_{left}| \leq 1$.
Nodes are left-heavy with balance factor ≤ -1, or right-heavy with balance factor ≥ 1.
ii. $h < 2\log n \iff 2^{\frac{h}{2}} \leq n \leq 2^{h+1} - 1$
iii. `insert`: After insertion, walk up tree and fix lowest unbalanced node (max 2 rotations)
iv. `delete`: After deletion, fix all unbalanced nodes until root (up to $\Theta(\log n)$ rotations)

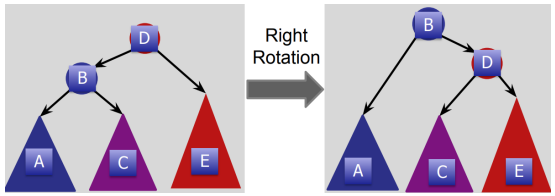
To balance left-heavy node `v`:

i. `v.left` balanced or left-heavy
$\implies$ `rightRotate(v)`
ii. `v.left` right-heavy
$\implies$ `leftRotate(v.left); rightRotate(v)`

To balance right-heavy node `v`:

i. `v.right` balanced or right-heavy
$\implies$ `leftRotate(v)`
ii. `v.right` left-heavy
$\implies$ `rightRotate(v.right); leftRotate(v)`

Rotations




```
Node leftRotate(B)          Node rightRotate(D)
  D = B.right                 B = D.left
  D.par = B.par               B.par = D.par
  B.par = D                   D.par = B
  B.right = D.left            D.left = B.right
  D.left = B                  B.right = D
  return D                    return B
```

Problems

Order Statistics, $O(\log n)$ if balanced

i. Augment nodes with subtree size $= size_{left} + size_{right} + 1$
ii. `select`: find kth smallest element.
iii. `rank`: get node position in sorted order.

Counting Inversions, $O(n \log n)$:

i. After inserting each element, add `i - tree.rank(ele)` to running count.

Interval Query, $O(\log n)$

i. Nodes store interval, key is lower interval bound
ii. Augment nodes with subtree max upper bound
iii. `interval`: If left subtree exists with $max \geq x.low$, recurse on left subtree, else on right subtree

1D Range Query, $O(\log n + k)$ for k elements in range:

i. Leaves are elements. Internal nodes are max of left subtree.
ii. `range`: Find split node $O(\log n)$ and traverse leafs

2D Range Query, $O(\log^2 n + k)$ for k elements in range:

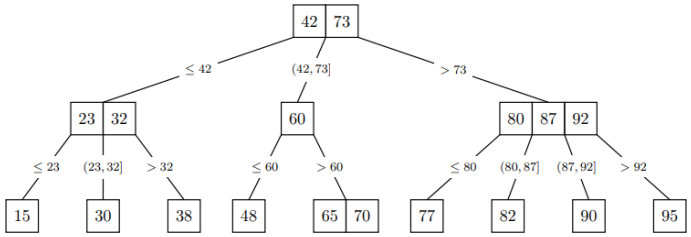
i. Build 1D x-tree for each x, then for each internal node, build y-tree for all y
ii. `range`: $O(\log n)$ y-tree searches each of $O(\log n)$

Tries

Tree storing strings where each node is a character, and each path is a key. Operations are $O(L)$, where L is length of the key. Faster than $O(Lh)$ of strings in bBST. Both use $O(\text{text size})$ space but Trie has node overhead.

i. `prefixSearch`: Find all keys with prefix.
Time: $O(L + k)$, where k matches

(a,b)-Trees



Self-balancing search trees where non-root nodes have between a, b children for $2 \leq a \leq \frac{b+1}{2}$

i. Root has $[2, b]$ children.
Internal nodes have $[a, b]$ children.
ii. Nodes with k children have $k - 1$ keys.
Leaves have $[a - 1, b - 1]$ keys.
iii. All leaves at same depth
iv. `search`: Time: $O(b \log_a n) = O(\log n)$
v. `insert`: Proactively split full nodes ($b - 1$ keys) during search, guaranteeing parent of split node won't have excess keys after insertion of split node.
Time: $O(\log n)$
vi. `split`: Split about median node z with $\geq 2a$ keys, s.t. left has $\geq a - 1$ keys and right has $\geq a$ keys.
Then, z offered to parent with $\leq b - 2$ keys.
Time: $O(b)$ for splitting node with b children
vii. `delete`: Passively, delete node then recursively check parents for violation, carrying out merge/share with smallest adjacent sibling. For internal nodes, replace with successor.
Time: $O(\log n)$
viii. `merge`: Demote split node and join when $< b - 1$ keys. Time: $O(b)$
ix. `share`: `merge` then `split` when $\geq b - 1$ keys

5. Hash Tables

Hash tables map n key-value pairs to m buckets.

i. Load $= \alpha = \frac{n}{m}$ = expected items per bucket
ii. `hashcode` computes hash, by default memory address

Linear Chaining, colliding pairs are added to linked list:

i. `get`: Time: $O(1+\alpha)$ (Expected), $O(n)$ (Worst)
ii. `insert`: Time: $O(1)$
iii. `delete`: Time: $O(1+\alpha)$ (Expected), $O(n)$ (Worst)
iv. Expected max chain length: $O(\log n)$ (Expected)
v. Space: $O(m+n)$
vi. Assumption: Follows SUHA

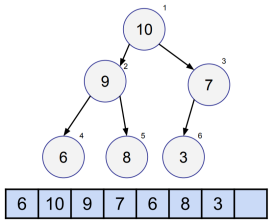
Linear Probing, probe following sequence of buckets:

i. `get`: Time: $O(1)$ (Expected $\alpha < 1$; worse as $\alpha \to 1$)
ii. `insert`: Time: $O(1)$ (Expected), $O(n)$ (Worst)
iii. `delete`: Use tombstone instead of set `null`. Time: $O(1)$ (Expected), $O(n)$ (Worst)
iv. `resize`: Resize to keep operations amortised $O(1)$.
When $n == m$, grow $m = 2m$.
When $n < \frac{m}{4}$, shrink $m = \frac{m}{2}$.
Time: $O(m_1 + m_2 + n) = O(n)$
v. Space: $O(m)$
vi. Assumption: Violates UHA with expected cluster size $= \Omega(\log n)$ when $a = \frac{1}{4}$
vii. Clusters take advantage of caching.
viii. More sensitive to hash function quality.

Growth	Resize	Insert n Items
Increment by 1	$O(n)$	$O(n^2)$
Double	$O(n)$	$O(n)$ (Amortised $O(1)$)
Square	$O(n^2)$	$O(n)$

6. Heaps

Binary Heap (MaxHeap)



Binary heaps are complete binary trees with heap order.

i. Shape: Complete binary tree (all levels filled left-to-right)
ii. Heap: $priority_{parent} \geq priority_{child}$
iii. `insert`: Insert as leaf and bubble up.
Time: $O(\log n)$
iv. `extractMax`/`delete`: Remove (root), replace with last node and bubble down with larger child.
Time: $O(\log n)$
v. `increaseKey`/`decreaseKey`: Keep HashMap mapping id to index to access in $O(1)$. Then, update and bubble up/down with larger child.
Time: $O(\log n)$
vi. `contains`: Use HashMap. Time: $O(1)$

Array implementation, Space: $O(n)$:

i. Size at index 0
ii. Parent of node i at $\lfloor \frac{i}{2} \rfloor$
iii. Left child of node i at $2i$
iv. Right child of node i at $2i+1$

Heapify, Time: $O(n)$:

i. For each level bottom-up, bubble down from right-to-left.

HeapSort builds a max-heap in $O(n)$ and repeatedly extract max to end of sorted array.

i. Time: $O(n \log n)$
ii. In-place but unstable

7. Union-Find Disjoint Sets

UFDS manages disjoint sets.

i. `find`: Check if x, y belong to same set
ii. `union`: Merges set of x, y

Variations:

i. Quick-Find keeps track of node's set componentId.
`find`: $O(1)$, `union`: $O(n)$
ii. Quick-Union keeps set in tree, with arbitrary union.
`find`: $O(n)$, `union`: $O(n)$
iii. Weighted-Union keeps set in tree, with tree union connecting smaller tree to root of larger tree.
`find`: $O(\log n)$, `union`: $O(\log n)$
iv. Path-Compression on find reparents all nodes on path to root, to the root.
`find`: $O(\log n)$, `union`: $O(\log n)$
v. WC+PC for m operations.
`find`: $a(m,n)$, `union`: $a(m,n)$

8. Amortised Analysis

Amortised analysis gives the average time per operation over a worst-case sequence of operations, even if some operations are expensive.

i. Guarantees avg. cost over k operations $\leq kT(n)$

Methods:

i. Aggregate: Total cost $= T \Rightarrow$ am. cost $= T/k$
ii. Accounting: Overcharge cheap ops to "pay" for expensive ops later

Hash Table Resizing:

i. Resize cost $O(n)$, spread over n ops $\Rightarrow O(1)$ am.

9. Graphs

Graphs G consist of vertices V connected by edges E.

i. Degree (node): Number of edges incident on a node
ii. Degree (graph): Max. degree of any node
iii. Diameter: Max. shortest path between nodes
iv. Clique: Complete graph
v. Dense: $|E| = \Theta(V^2)$
vi. Adjacency Matrix: Fast query neighbour (u, v) Space: $O(V^2)$
vii. Adjacency List: Fast enumerate neighbours Space: $O(V + E)$

Traversal, $O(V + E)$ (Adj. list) $O(V^2)$ (Adj. matrix):

i. BFS: Explore by level, add nodes to frontier queue.
ii. DFS: Explore deeply before backtrack, uses stack.

SSSP

Conserved across adding constant to edge.

BFS/DFS handle trees. BFS handles unweighted graphs:

i. Relax edges in BFS/DFS order
ii. Time: $O(V + E)$

Djikstra handles positive-weighted graphs

i. Using PQ for closest node, $|V|$ `insert/extractMin` and $|E|$ `relax/decreaseKey`
ii. Time: $O((V + E) \log V) = O(E \log V)$, or $O(E + V \log V)$ (Fib. Heap)

Bellman-Ford handles negative-weights and detects negative cycles:

i. $|V| - 1$ iterations of relaxing all $|E|$ edges, terminating early when estimates aren't changed
ii. Time: $O(VE)$

DAG_SSSP handles DAGs:

i. TopoSort, then relax edges in topological order
ii. Time: $O(V + E)$
iii. For longest path, negate edges/ modify relax

Topological Order

Topological sorting on DAGs:

i. Post-order DFS: Time: $O(V + E)$
ii. Kahn's Algorithm: Add nodes with no incoming edges, then remove their outgoing edges and repeat. Time: $O(V + E)$

Minimum Spanning Trees

Minimum spanning trees contain all vertices with minimum total edge weight. Conserved across adding/multiplying constant to edge.

i. Cutting an MST produces two MSTs
ii. Cycle: Max. weight edge of cycle not in MST
iii. Cut: Min. weight edge across cut is in MST

Generic MST Algorithm:

i. Color max. edge in each cycle red, and min. edge in each cut blue; greedily apply this rule until no more edges uncolored with blue edges forming MST

Prim's Algorithm:

i. Using PQ for lightest edge, grow MST by adding lighted edge to unvisited nodes
ii. Time: $O((V + E) \log V) = O(E \log V)$

Kruskal's Algorithm:

i. Using UFDS to keep track of cycles, sort edges by weight, adding if endpoints are not connected.
ii. Time: $O(E \log E) = O(E \log V)$

Boruvka's Algorithm:

i. Each component adds min. outgoing edge. Repeat $\log V$ times
ii. Time: $O(E \log V)$

Directed Graph (with root):

1. For each node except root, add min. incoming edge
2. Time: $O(E)$

10. Dynamic Programming

DP solves problems with optimal substructure and overlapping subproblems.

i. Top-down (Memoization): Recursive with cache
ii. Bottom-up (Tabulation): Iteratively fill table

Problems

Fibonacci:

i. $F(n) = F(n-1) + F(n-2)$
ii. Time: $O(n)$ with memoization or bottom-up

Longest Increasing Subsequence (LIS):

i. $dp[i] = \max(dp[j] + 1)$ for $j < i$ and $A[j] < A[i]$
ii. Time: $O(n^2)$ or $O(n \log n)$ with binary search

Knapsack 0/1:

i. $dp[i][w] = \max(dp[i-1][w], dp[i-1][w - w_i] + v_i)$
ii. Time: $O(nW)$

Matrix Chain Multiplication:

i. $dp[i][j] = \min_{i \le k < j}(dp[i][k] + dp[k+1][j] + cost)$
ii. Time: $O(n^3)$

Floyd-Warshall (All-Pairs Shortest Path):

i. $dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])$
ii. Time: $O(V^3)$

Prize Collecting (on Graph):

i. $dp[v][k] = \max(dp[u][k-1] + prize(u, v))$
ii. Time: $O(kE)$

Vertex Cover on Tree:

i. $dp[v][0]$ = min cover if v not included
$dp[v][1]$ = min cover if v included
ii. Recurrence:
- $dp[v][0] = \sum dp[c][1]$ for all children c
- $dp[v][1] = 1 + \sum \min(dp[c][0], dp[c][1])$

iii. Time: $O(V)$