CS2040S

AY24/25 Sem 2

by ngmh

Document Distance

- Binary Similarity: Identical texts
- Scalar: Number of common words, ratio, etc.
- Vector Space Model: Each document is a high-dimensional vector, where dimensions are word frequencies
- To compare 2 words, convert them to vectors, calculate vector norms, dot product, and angle which represents similarity

Big-O Notation

- T(n) = Running time on inputs of size n
- O = Upper Bound
- Θ = Tight Bound
- Ω = Lower Bound
- $T(n) = O(f(n))$ if T grows no faster than f
- There exists $c > 0$ and $n_0 > 0$ such that for all $n > n_0$, $T(n) \leq cf(n)$
- $T(n) = \Omega(f(n))$ if T grows no slower than f
- There exists $c > 0$ and $n_0 > 0$ such that for all $n > n_0$, $T(n) \geq cf(n)$
- $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))s$
- If $T(n)$ is a polynomial of degree k then $T(n) = O(n^k)$
- If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then $T(n) + S(n) = O(f(n) + g(n))$ and $T(n) * S(n) = O(f(n) * g(n))$
- Sterlings Approximation: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$
- Appending to String is $O(n)$ in Java

Solving Recurrences

- Guess and Check using substitution
- Drawing recursion tree
- Master Theorem

$$T(n) = aT(n/b) + cn^k$$
$$T(1) = c,$$

onstants, solves to:

$$T(n) \in \Theta(n^k) \text{ if } a < b^k$$
$$T(n) \in \Theta(n^k \log n) \text{ if } a = b^k$$
$$T(n) \in \Theta(n^{\log_b a}) \text{ if } a > b^k$$

$$T(n) = 2T(\tfrac{n}{2}) + O(n) \Rightarrow O(n \log n)$$
$$T(n) = T(\tfrac{n}{2}) + O(n) \Rightarrow O(n)$$
$$T(n) = 2T(\tfrac{n}{2}) + O(1) \Rightarrow O(n)$$
$$T(n) = T(\tfrac{n}{2}) + O(1) \Rightarrow O(\log n)$$
$$T(n) = 2T(n-1) + O(1) \Rightarrow O(2^n)$$
$$T(n) = 2T(\tfrac{n}{2}) + O(n \log n) \Rightarrow O(n(\log n)^2)$$
$$T(n) = 2T(\tfrac{n}{4}) + O(1) \Rightarrow O(\sqrt{n})$$
$$T(n) = T(n-c) + O(n) \Rightarrow O(n^2)$$
$$T(n) = T(n-c) + O(1) \Rightarrow O(n)$$

- Fibonacci: $O(\phi^n)$
- Tutorial 2: $T(n) = T(n-1) + T(n-2) + O(1)$
  - Guess $T(i) = F(i) - 1$
  - $T(n) = T(n-1) + T(n-2) + 1 = F(n-1) - 1 + F(n-2) - 1 + 1 = F(n-1) + F(n-2) - 1 = F(n) - 1$
- $log(n!)$ tight bound:
  - $log(n!) = log(1) + log(2) + ... + log(n) \leq nlogn = O(nlogn)$
  - $log(n!) = log(1) + log(2) + ... + log(n) \geq log(n/2) + log(n/2+1) + ... + log(n) \geq log(n/2) + log(n/2) + ... + log(n/2) = \frac{n}{2}log(\frac{n}{2}) = \Omega(nlogn))$
- log v/s $sqrt$ crap:
  - $log(n) < \sqrt{n}$
  - $log^2(n) < n$
  - $n^{log(n)} < 2^n$
- Chicken Rice Median with QuickSelect: For each level, pivot has $n-1$ bites, and median plate has expected cost of $\frac{1}{n} \cdot n + (1 - \frac{1}{n}) \cdot 1 \leq 2$, total is $O(2logn) = O(logn)$ by expectation

Binary Search

- Preconditions: Condition that is true before running, e.g. Array is sorted and of size n
- Invariants: Relationship between variables that is always true, e.g. $A[begin] \leq key \leq A[end]$
- Postconditions: Condition that is true after running, e.g. If element is in array: $A[begin] = key$
- Can be augmented by binary searching on a monotonic function instead

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

Peak Finding

- Find local maximum in A ($\leq$)
- Use divide and conquer to recurse
- Invariant: There always exists a peak in the half we recurse on
- Correctness: There exists a peak in [begin, end] and every peak in [begin, end] is a peak in [0, n-1]

```
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
```

Steep Peaks

- Find local maximum in A ($<$)
- Cannot use old method, consider case where both sides are equal and we do not know where to recurse
- Degenerates to $O(n)$ if we recurse on both half

2D Peak Finding

Slow Algorithm

- Find global max in each column
- Find peak in array of max elements
- $O(mn + log(m))$

Fast Algorithm

- Find peak in array of peaks with lazy evaluation of columns
- Find max of middle column
- Recurse left / right half by comparing max of adjacent columns
- $O(nlogm)$

Sorting

BogoSort

- Choose a random permutation
- Check if sorted
- $O(n \cdot n!)$

BubbleSort

- Repeatedly swap adjacent elements
- Best Case: Sorted, $O(n)$
- Average / Worst Case: $O(n^2)$
- Invariant: At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array
- Loop through the whole array n times, swapping any adjacent elements that are out of order

SelectionSort

- Repeatedly find the minimum element and add it to the prefix
- Always $O(n^2)$
- Invariant: At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array
- Loop through thte whole array n times, finding the minimum and appending it to the prefix by swapping with the end of prefix

InsertionSort

- Insert the current element into the correct position in the prefix from the back
- Best Case: Sorted, $O(n)$
- Average / Worst Case: $O(n^2)$
- Invariant: At the end of iteration j, the first j items in the array are in sorted order
- Loop through indexes of the array, each time bubbling the current element from the back down to its correct position

```
while (i > 0) and (A[i] > key)
    A[i+1] = A[i]
    i = i-1
A[i+1] = key
```

MergeSort

- Uses Divide and Conquer
- $T(n) = 2T(n/2) + cn$
- $logn$ levels, n for each level, $O(nlogn)$
- Sort both halves of the array, then merge them together

QuickSort

- Repeatedly partition the array by a pivot and recurse on both halves
- Average Case: $O(nlogn)$
- Worst Case: $O(n^2)$
- Maintain a low and $high$ pointer
- Non In-Place Partition
  - If an element is lower than pivot, add to prefix
  - If an element is higher than pivot, add to suffix
  - Invariant: For every $i < low$, $B[i] < pivot$ and for every $j > high$, $B[j] > pivot$
- In-Place Partition
  - Invariant: $A[high] > pivot$ at the end of each loop
  - Invariant: For all $i >= high$, $A[i] > pivot$ and for all $1 < j < low$, $A[j] < pivot$

```
swap(A[1], A[pIndex])
while (low < high)
    while (A[low] <= p) and (low < high) low++
    while (A[high] > p) and (low < high) high--
    if (low < high) swap(A[low], A[high])
swap(A[1], A[low-1])
```

- 3-Way Partitioning:
  - Two pass: Regular partition, then pack duplicates using high low pointers
  - One pass: Maintain four regions of array (< pivot, = pivot, in progress, > pivot)
  - If $A[i] < pivot$, swap with start of = pivot
  - If $A[i] == pivot$, increment pivot end by 1
  - If $A[i] > pivot$ swap with start of > pivot

Paranoid Quicksort

- Partition until a certain partition factor is met (e.g. $9/10$)
- Probability of good pivot = $8/10$
- $E[Choices] = 1/p = 10/8 < 2$
- By expectation, we only have to partition twice
- $E[T(n)] = E[T(k)] + E[T(n-k)] + E[Choices](n) \leq E[T(k)] + E[T(n-k)] + 2n = O(nlogn)$
- $T(n) = T(n/10) + T(9/10) + O(n) = O(nlogn)$

Properties of Sorting Algorithms

- In-Place v/s Not In-Place
- MergeSort and QuickSort is not in-place
- Stability: Preserving initial order of equal elements
- SelectionSort and QuickSort are not stable due to swaps
- Insertion and Bubble Complexity: Swap first and last element

QuickSelect

- Select the k^{th} smallest element in an unsorted array
- Randomly partition, with paranoia
- Recurse only on correct half
- $E[T(n)] \leq E[T(9n/10)] + E[Partitions](n) \leq E[T(9n/10)] + 2n \leq O(n)$

Trees

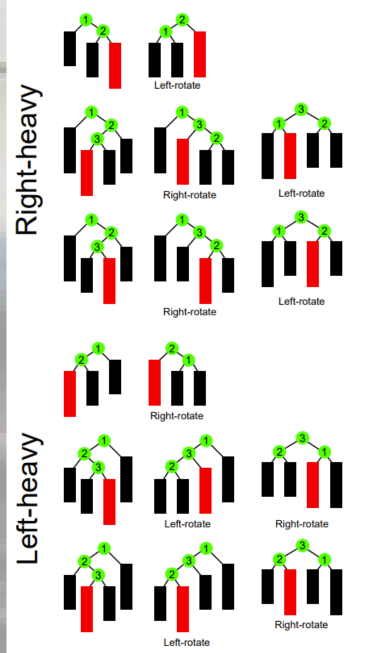
- Binary Search Tree: Keys in left < Key < Keys in right
- Height: 0 at leaf, increases by 1 for each parent, maximum of children
- Insertion: Traverse until respective child does not exist on node and just add
- In-Order: LSR; Pre-Order: SLR; Post-Order: LRS
- Level-Order: Greatest height to lowest Height, from left to right
- Successor:
  - Search for key
  - If $result > key$ return $result$
  - If $result \leq key$ search for $succesor(result)$
  - $successor()$:

```
if (righTree != null) return rightTree.min()
TreeNode parent = parentTree
TreeNode child = this
while parent != null && child == parent.right
    child = parent
    parent = child.parentTree
return parent
```

- Deletion:
  - 1 Child: Remove v, Connect child(v) to parent(v)
  - 2 Children: Delete successor and replace node with it

AVL Trees

- Balanced Tree: $h = O(logn)$
- Height Balanced Tree: $|v.left.height - v.right.height| \leq 1$, has at most $h < 2log(n)$ nodes and at least $n > 2^{h/2}$
- Weight Balanced Tree: $v.left.w, v.right.w \leq \alpha\ v.w$
- Height / Weight Balanced implies Balanced but not the other way round
- Maximally Imbalanced: AVL tree with minimum possible number of nodes given its height
- Left / Right Heavy: Which child has larger height
- Rotations
  - Direction is where root of subtree goes
  - Requires a child opposite of rotation direction
  - v Left Heavy: Check Left child:
    - Balanced: Right-Rotate(v)
    - Left Heavy: Right-Rotate(v)
    - Right Heavy: Left-Rotate(v.left), Right-Rotate(v)
  - v Right Heavy, Check Right child:
    - Balanced: Left-Rotate(v)
    - Left Heavy: Right-Rotate(v.right), Left-Rotate(v)
    - Right Heavy: Left-Rotate(v)
  - Basically if middle heavy rotate child away from middle first before rotating root of subtree
  - Credits to Way Yan (for tracing):



- Insertion:
  - Only need to fix lowest unbalanced node after deletion, with at most two rotations
  - Since rotating reduces height by 1, the height after rotation will be correct
- Deletion
  - Swap node with ancestor if it has 2 children
  - Delete node from tree, reconnecting children
  - Recurse upwards and rotate if needed, up to $O(logn)$ needed
  - Since height after rotation might be less than height before deletion
- Potential Modification: Only store difference in height from parent

Scapegoat Tree

- 2/3 Weight-Balanced tree
- Unbalanced: Child has more than $2/3$ of the total subtree weight
- Rebuild tree at highest unbalanced node

Augmenting Data Structures

- Choose underlying data structure, Additional information, Maintain updates of information, Develop new operations
- Order statistics: Rank and Select, use subtree size information to determine where to recurse
- Counting Inversions: At each index, the inversion count is tree size minus rank of current element

Order Statistic Trees

- Augment with weight of each node
- $select(k)$: Finds node with rank k

```
rank = left.weight+1
if k == rank return v
if k < rank return left.select(k)
if k > rank return right.select(k-rank)
```

- $rank(node)$ Computes rank of a node v

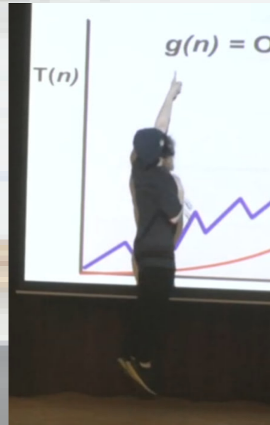
```
rank = left.weight+1
while node != null
    if node == par.right
        rank += par.left.weight + 1
    node = par
return rank
```

Tries

- Store letters in nodes
- Words are represented as paths from root to leaf
- Use special mark for terminal node / character

(a, b)-trees

- Restriction: $2 \leq a \leq (b+1)/2$.
- Rule 1: (a, b)-child policy
  - Root node must have between 1 and $b-1$ keys
  - Internal / Leaf node must have between $a-1$ and $b-1$ keys
  - Leaf node has no children
- Rule 2: Key ranges
  - A non-leaf node has one more child than number of keys
  - Each subtree has a key range
- Rule 3: Leaf depth
  - All leaf nodes must be at the same depth
- B-Tree: $(B, 2B)$-tree, where B is block size of a storage device
- Min / Max Height: $O(log_b n), O(log_a n) + 1$
- Searching: At each node, binary search for corresponding key range to recurse on, taking $O(log_2 b \cdot log_a n) = O(logn)$ time
- Insertion:
  - Insert at leaf node
  - Redistribute keys if it violates size limit by performing a split
    - Find median key v_m
    - Separate all keys $\leq v_m$ to make a new node y
    - Give y the final child from the left of v_m
    - Insert v_m into parent
    - Connect v_m to the new node y we created
    - Recurse upwards if needed
  - Works as LHS $= \lfloor (b-1)/2) \rfloor = \lfloor a - 1/2 \rfloor$ and RHS $= \lceil (b-1)/2 \rceil = \lceil a - 1/2 \rceil$, and $\lfloor a - 1/2 \rfloor \geq a - 1$
  - Proactive: Preemptively split any node at full capacity $(b-1)$ during search phase
    - Must have $b \geq 2a$
    - After split, left sibling has $\lfloor (b-2)/2 \rfloor = \lfloor b/2 \rfloor - 1$ keys
    - After splitting, it must have $\geq a - 1$ keys
    - $\lfloor b/2 \rfloor - 1 \geq a - 1, \lfloor b/2 \rfloor \geq a, b \geq 2a$
  - Passive: Perform insertion first then check parent for violation and recurse
- Deletion:
  - Search for key, then delete from keylist
  - However, number of keys might fall below $a-1$
  - Suppose z is offending, and y is smallest sibling
  - Merge with sibling if necessary
  - $y + z < b - 1$:
    - In parent, delete key v separating siblings
    - Add v to keylist of y
    - In y, merge in z's keylist and treelist
    - Remove z
  - $y + z \geq b - 1$
    - Perform sharing by $merge(y, z)$ followed by splitting merged node
    - By definition of (a, b)-trees, they will have at least $b+1$ keys after taking one from the parent
    - LHS has $\lfloor (b+1)/2 \rfloor \geq a$, RHS has $\lceil (b+1)/2 \rceil \geq a$
  - Proactive: Preemptively merge and share any node at minimal capacity $(a-1)$ during search phase
  - Passive: Delete first then check parent for violation and recurse
  - If deleting a key results in orphaned children, swap out key with a leaf node key (successor) before deleting



pls give me A+ eldon thanks