CS 240 Midterm Review (Module 1–7)

Asymptotic Analysis

- Order notations
  - $f(n) \in O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le f(n) \le cg(n)\ \forall\ n \ge n_0$
    - f "grows no faster than" g ($\le$)
  - $f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le cg(n) \le f(n)\ \forall\ n \ge n_0$
    - f "grows no slower than" g ($\ge$)
  - $f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\ n \ge n_0$
    - f and g grow at the same rate ($=$)
  - $f(n) \in o(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \le f(n) < cg(n)\ \forall\ n \ge n_0$
    - f is "*strictly* upper-bounded" by g ($<$)
  - $f(n) \in \omega(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \le cg(n) < f(n)\ \forall\ n \ge n_0$
    - f is "*strictly* lower-bounded" by g ($>$)
  - Suppose $L = \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$
    - If $L = 0$ then $f \in o(g)$
    - If $0 < L < \infty$ then $f \in \Theta(g)$
    - If $L = \infty$ then $f \in \omega(g)$
  - If $f \in O(g)$ and $f \in \Omega(g)$, then $f \in \Theta(g)$
- Loop analysis
  - Begin from the innermost nested loop; use $\sum$ for each outer loop
- Recurrence relations analysis
  - **In general**, $\{T(n) = T(n/2) + c\} \in \Theta(\log n)$

$$
\begin{aligned}
&\text{Let } n = 2^k, \text{i.e. } k = \log n \\
T(2^k) &= T(2^{k-1}) + c \\
&= T(2^{k-2}) + 2c \\
&= \ldots \\
&= T(2^0) + kc \\
&= c + (\log n)c \\
&\in \log n
\end{aligned}
$$

Priority Queues and Heaps

- **Priority queue**: an **abstract data type** containing a collection of items each with a priority

- **Heap**: binary tree with 2 properties
  - Structural property: all levels of filled except the lowest, which is left-justified
  - Ordering property: the parent of any node has greater value than the node itself
- The height of a heap with n nodes is $\Theta(\log n)$
  - Since $2^k \leq n$ (# of nodes on all levels above) and $n \leq 2^{k+1} - 1$ (# of nodes including this level)
- **Bubble-up algorithm**: used for heap <u>insertion</u>
  - If node.key > node.parent.key then swap
  - Brings a <u>large</u> value from a leaf node <u>up</u>
- **Bubble-down algorithm**: used for heap <u>deletion</u>
  - If node.key < node.largest_child.key then swap
  - Brings a <u>small</u> value from the root node <u>down</u>
- Heapify with bubble-up: insert each item, total runtime = $\Theta(n \log n)$
- Heapify with bubble-down (given an unordered array): since leaf nodes can't bubble-down, start bubbling down from second-last level up ($n/2$ nodes)
  - Total runtime = $\Theta(n)$
- **Heapsort**: heap-insert n times (or just heaptify), then delete-max n times – always $\Theta(n \log n)$
  - <u>Not stable</u>

Sorting, Selection, Randomized Algorithms

- A sorting algorithm is <u>stable</u> if the order of equal (tied) keys are preserved (from the original order in the input)
- **Selection problem**: find the k-th largest element within n elements
- **Quickselect**:
  - Choose pivot = $\Theta(1)$
  - Partition:
    - Go from outermost pair inwards, swap any pairs that are in the wrong order ($\Theta(n)$)
    - Return index of pivot; array is now partitioned by the pivot value
  - Recursively call partition on one of the two partitions, until pivot index = desired index (like binary search)
  - Worse case: every recursive call paritions off 1 element = $\Theta(n^2)$
  - Best case: desired element is returned on first call = $\Theta(n)$
- **Quicksort**:
  - Same as quickselect, except recurse on both partitions instead of just one
  - Worse case = $\Theta(n^2)$

    - Best case = average case = $\Theta(n \log n)$
    - Not stable
- **Randomized algorithm**: algorithm whose output depends on the input as well as some random numbers
    - $T(I, R)$ = runtime given input I and set of random numbers R
    - **Expected runtime** $= T^{exp}(I) = \sum_R T(I, R) \times P(R)$
        - For uniform distribution, $P(R) = 1/\#$ possibilities generated by R
    - For quickselect and quicksort, randomizing the pivot makes the expected time = average time
    - **Monte Carlo algorithm**: always fast, not always correct
    - **Las Vegas algorithm**: always correct, not always fast
- **Comparison model**:
    - Data can only be accessed by:
        - Comparing two elements
        - Moving elements around
    - **Theorem**: any correct comparison-based sorting algorithm is $\Omega(n \log n)$
- **Countsort**: input of size n only contains numbers in a consecutive range of size k
    - Count the # of occurrences of each element (i.e. a histogram)
    - Place elements back into array in-order, based on their # of occurrences
    - $\in \Theta(n + k) \in \Theta(n)$ if $k \in O(n)$
    - Is stable
- **Radix sort**: represent all elements in base r (radix)
    - Sort by each digit (count sort), starting from most/least significant digit (m digits)
    - $\in \Theta(m(n + r)) \in \Theta(n)$ if $m, r \in O(n)$
    - MSD is not stable, LSD is stable

Balanced Search Trees

- **Binary search tree**
    - Left[right] subtree nodes <[>] root node
    - Search, insert = $\Theta(\log n)$
    - Delete = $\Theta(\log n)$
        - If is leaf, just delete
        - If has one child, replace with child
        - If has two children, swap with *predecessor* or *successor* then delete

- Worse-case height = $\Theta(n)$

- **AVL tree**
  - **Balance** of each node = height(right subtree) – height(left subtree)
  - Height of empty subtree = –1
  - If |balance| > 1, tree is out of balance
  - **Right/left rotation**: when node balance = ± 2 and a child has balance = ± 1 of the same sign
  - **Double right/left rotation**: when node balance = ± 2 and a child has balance = ± 1 of the opposite sign
  - Rotations are $\Theta(1)$
  - Insert & delete normally as in BSTs, update balances from bottom up and rotate if any subtree is out of balance ($\Theta(\log n)$)
  - # of nodes in a subtree of a given height is at least = $N(h) = 1 + N(h-1) + N(h-2) \geq 2N(h-2)$
    - From this recurrence we get $N(h) \geq 2^{\lfloor h/2 \rfloor}$ or $h \in O(\log n)$
  - # of nodes is also at most $N(h) = 2^{h+1} - 1$, or $h \in \Omega(\log n)$
  - Therefore the height of an AVL tree is $\Theta(\log n)$

Dictionaries

- **Dictionary**: a collection of **key-value pairs** (KVP)
- **Optimal static ordering**: elements are stored in decreasing order by probability of access
  - Given L of n elements, expected access cost is $E(L) = \sum_{i=1}^{n} P(x_i)T(x_i)$
  - If P is uniform then $E(L) = \sum_{i=1}^{n} \frac{i}{n} \in \Theta(n)$
- **Dynamic ordering**:
  - **Move-to-front (MTF)**: move searched item to the front of list
  - Transpose: swap searched item with item preceding it
- **Skip list**:
  - A series of lists $S_0 \dots S_h$ containing keys in increasing order
    - Each starts and ends special keys $-\infty$ and $+\infty$
    - Each level contains a subset of the level below; i.e. $S_h \subseteq S_{h-1} \subseteq \dots \subseteq S_0$
    - S_h only contains $-\infty$ and $+\infty$
  - Skip search: navigate down and to the right from the top-left
    - Peek ahead on current level; if went over target, descend one level
  - Height of an inserted element is <u>randomly computed</u>
    - e.g. height = k = # of heads flipped before a tail

    - $P(\text{height} = k) = (1/2)^k$
    - Expected # of nodes on level k is $\dfrac{n}{2^k}$
    - Expected total # of nodes for C levels is $\sum_{k=0}^{C} \dfrac{n}{2^k} = 2n$ as $C \to \infty$
  - Expected space $= O(n)$
  - Expected height $= O(\log n)$
  - Expected search, insert, delete $= O(\log n)$

Tries

- **(Binary) Trie/radix tree**: a bitwise binary tree
  - Left child $= 0$, right child $= 1$
  - A node is flagged if the binary string generated by the path from root to it is in the dictionary
  - Insert: search for node
    - If it exists, flag it
    - If not, extend from last matching node by creating new nodes
  - Delete: search for node
    - If it's not a leaf, unflag it
    - If it's a leaf, delete it and all ancestors until a flagged node or node with 2 children is reached
  - Search, insert, delete $\in \Theta(h) = \Theta(|x|)$ where $|x| = \#$ of bits
- **Compressed trie/Patricia trie**
  - Reduce each path through unflagged nodes with one child to a single edge
  - Each node stores the next index/digit to be tested
  - Time complexity same as uncompressed; space complexity is improved

Hashing

- **Theorem**: any comparison-based search on a size-n dictionary is $\Omega(\log n)$
- **Direct addressing**:
  - Each key k is integer $0 \leq k < M$ for some M
  - Each value v corresponding to k is stored at $A[k]$
  - Search, insert, delete $\in \Theta(1)$
  - Total storage $\in \Theta(n)$
- **Hash function** $= h : U \to \{0, 1, \ldots, M-1\}$
  - Any key $k \in U$ is mapped to some index in an array of size M (a hash table)

    - i.e. value v for key k is stored at $A[h(k)]$
- **Load factor** $= \alpha = \dfrac{n}{M}$
    - If load factor is too high/too low, increase/decrease M and rehash, i.e. recreate hash table ($\Theta(M+n)$)
- **Closed addressing**: each entry in the hash table can hold more than one KVP (bucket)
    - Can use unordered linked list (chaining)
    - Average bucket size/chain length $= \alpha$
    - Thus search has average-case $\Theta(1+\alpha)$, worst-case $\Theta(n)$
- **Opening addressing**: if collision occurs, search linearly until a slot is available
    - $h(k, i) = h(k) + i \bmod M$
    - Must distinguish between empty and deleted slots
- **Double hashing**:
    - $h(k, i) = h_1(k) + h_2(k)i \bmod M$
- **Cuckoo hashing**: always insert new element at $h_1(k)$
    - If collision, kick out old element and re-insert at $h_2(k)$
    - Repeat for at most n times; rehash if exceeds n times
    - Any particular element is guaranteed to be at either $h_1(k)$ or $h_2(k)$

Multi-Dimensional Data

- **Range search**: return all elements within a certain range of values
    - Nodes are either *boundary*, *inside*, or *outside* based on the paths from root to the left & right boundary values
    - One-dimensional range search $= O(\log n + k)$ for k reported elements
- **Quad trees**: divide elements into 4 quadrants until each quadrant contains only 1 element
    - Each node has 4 children representing 4 quadrants
    - **Spread factor** $= \beta = \dfrac{d_{max}}{d_{min}}$, where $d_{max/min}$ = max/min distance between 2 points
    - Height $= h \in \Theta(\log \beta)$
    - Range search $\in \Theta(nh)$
- **kd-trees**: split elements into 2 equal regions until each region contains only 1 element
    - Sort by x-coordinate; divide elements by median
        - Median is stored at root; elements with x $\leq$[>] median go in left[right] subtree
    - Sort each subset by y-coordinate; repeat; alternate sorting between x- and y-coord
    - Height $= h \in \Theta(\log n)$
    - Rnage search $\in O(k + \sqrt{n})$

- **Range trees**:
  - Construct T w.r.t. x-coords
  - Every node in T has an associated tree T_{assoc} of its subtree in T w.r.t. to y-coords
  - Range search:
    - Perform 1D range search on T w.r.t. x-coords
    - For outside nodes, do nothing
    - For boundary nodes, check individually
    - For inside nodes that are immediate children of boundary nodes, perform 1D range search on their T_{assoc} w.r.t. y-coords
    - Time complexity $\in O(k + \log^2 n)$
    - Space complexity $\in O(n \log n)$

String Matching

- Given a text T of length n and a pattern/word P of length m, we try to find the first i such that $P[j] = T[i + j]$ for $0 \le j \le m - 1$
- Brute-force $\in \Theta(mn)$
- Deterministic finite automata
  - Matching time $\in \Theta(n)$
  - Preprocessing time $\in O(m|\Sigma|)$
- **Knuth-Morris-Pratt (KMP)**:
  - Instead of shifting P by 1 and re-checking every time P is not matched (as in brute-force), shift P to the next spot where at least some of P is matched
    - How much to shift? Find the longest prefix that is a **strict** suffix
  - **Failure array**:
    - $F[j] \leftarrow$ length of the longest strict suffix ($P[1..]$) that is a prefix ($P[0..]$)
    - $F[0] = 0$ always
    - $F[j - 1]$ gives the index in P to start checking again after a failed match
      - Everything before that index is known to match, due to it being the longest prefix that is a suffix
    - $\in O(m)$
  - Matching using the failure array:
    - If match succeeds, move to next position in T and P
    - If match fails at first position ($T[i] \neq P[0]$), just try next position ($i + 1$
    - If match fails at $j > 0$, stay at the same place in T and start checking at $P[F[j-1]]$
    - Alternative method: each time, shift P forward by $j - F[j]$

- $\in O(n)$

- **Boyer-Moore**:
  - Check match right-to-left, starting with last character in P
  - **Bad character heuristic**:
    - Suppose match fails at $T[i] = c$
      - If $c \in P$, shift P forward to align the last occurrence of c in P with $T[i]$
      - If $c \notin P$, shift P past c to align $P[0]$ with $T[i+1]$
    - **Last-occurrence function**:
      - For each c in Σ, $L(c)$ = largest index of c in P, or -1 if $c \notin P$
  - **Good suffix heuristic**:
    - Suppose $P[j..m] = t$ (a suffix of P) is matched, but fails at $P[j-1] = c$
    - Shift P forward to align the last occurrence of t in P in the same location
      - Or if t doesn't exist, align the longest suffix of t that is a prefix of P
      - Or if no suffix exists, shift P past t completely (like in bad character)
    - **Suffix skip array**:
      - $S[i] \leftarrow$ largest j such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ and $P[i] \neq P[j]$
      - $S[i]$ gives the index before the last occurrence of t not prefixed by c in P
      - If $t \in P$, extend the indices into negatives by prepending P to itself, but overlapping the longest prefix that's a suffix
  - On mismatch, choose the one that provides a bigger jump
    - Shift the end of P to index $= i + m - 1 - min(L(T[i], S[j])$
    - i.e. shift P forward by $1 - max(L(T[i], S[j])$
  - Worst case $\in O(n + |\Sigma|)$

- **Rabin-Karp Fingerprint**
  - Allow a substring of the length of P (m) to be hashed as a number
  - At every position in T, hash the m digits at that location and compare with the hash of P ($\Theta(1)$)
    - If hash does not match, move to next position
    - If hash matches, check for match linearly ($\Theta(m)$))
    - Rolling hash: assume $h(t) = t \mod k$
      - Only hash the first position fully (e.g. $h(abcde)$)
      - At subsequent positions, (this is $\Theta(1)$)

$$h(bcdef) = (h(abcde) - a \times (10^5 \mod k)) \times 10 + e \mod k$$

- **Suffix trees**:

- If multiple patterns are searched for in T, preprocess T instead
- Store all suffixes of T terminated by end-of-string character (\$) in a trie
- Compress trie into a tree:
  - Each node holds the length of substring generated by the path from root to it
  - Also holds l, r such that the node's substring corresponds to $T[l..r]$
    - If node is a leaf, $r = n - 1$ (goes to the end of T)
    - Otherwise, $T[l..r]$ is the first occurrence of the substring in T
  - There are $n + 1$ leaves, with $[0, n] \dots [n, n]$
- Only need to find a prefix in the tree

Compression

- **Compression ratio** $= \dfrac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$ where S = source text, C = encoded text
- Fix/variable-length codes: the encoding of each character of text has the same/different length
- **Huffman encoding**: $\Sigma_C = \{0, 1\}$
  - Binary trie where Σ_S is stored in leaf nodes
    - Path from root to each leaf generates the binary encoding of each character
    - Begin with single nodes containing each character and its weight (# of occurrences in S)
    - Combine tries with lowest weights until only one trie remains
    - Results in characters with higher frequencies being closer to root node
  - Dictionary trie must be sent along with C for decoding
- **Run-length encoding**: $\Sigma_S = \Sigma_C = \{0, 1\}$
  - Set $C[0] = S[0]$ to indicate the alternating sequence of 0's and 1's in S
  - For a run of 0/1 of length k:
    - $C = [\lfloor \log k \rfloor \text{ 0's}][\text{binary representation of } k]$
  - Only efficient for run lengths of $k \geq 6$
- Adaptive encoding: expand the dictionary as text is being encoded/decoded
- **Lempel-Ziv/LZW**:
  - Effective on English text
  - Encoding:
    - Each time, read & encode the longest substring of S that exists in the dictionary
    - For every substring x in S encoded, add xc to the dictionary where c is the character that follows x in S
  - Decoding:

- For each decoded substring y and the last decoded substring x, add xc to the dictionary where $c = y[0]$
- If decoding a substring that has yet to be added to the dictionary, let $y = x + x[0]$

- **Text transformation**: change source text to make it more compressible
  - Transformation must be reversible
- **Move-to-Front**:
  - Encoding outputs sequence of indices accessed by reading S (i.e. $|\Sigma_C| = |\Sigma_S|$)
  - Decoding outputs characters by accessing dictionary at indices given by C
  - Encoding & decoding work the same way: read one character/index and move it to front of the array
- **Burrows-Wheeler Transform**:
  - Produces runs of characters
  - Encoding:
    - Sorts all cyclic shifts (i.e. in-place rotations) of S; store in L
    - C = list of the last character of every string in L
- Decoding:
  - Generate list $(C[i], i)$ pairs from C and sort them by $C[i]$
  - Store the sequence of i's in the sorted list in an array N
  - Begin with j = index of \$ in C
  - Generate S by setting $j = N[j]$ and appending $C[j]$

Memory Data Structures

- **B-tree/2-3 tree**:
  - Each internal node either has 1 KVP and 2 children or 2 KVP and 3 children
  - All leaves do not hold KVPs and are on the same level
  - Search: if node has 2 KVPs, traverse to one of 3 children based on $x < a, a < x < b, b < x$
  - Insertion: find the lowest internal node and insert KVP
    - If node has 3 KVPs, promote the middle to parent node (recurse if necessary), split the other 2 to individual nodes
  - Deletion: swap KVP with in-order *successor* so it's in a leaf node (N)
    - If N has 2 KVPs, just delete
    - Otherwise if immediate sibling M (they share a parent P) has 2 KVPs, replace $N[0] \leftarrow U$ and $U \leftarrow M[1]$ and remove $M[1]$ (perform a transfer)
      - Where U = intermediate value between N and M in P (i.e. successor/predecessor of N)

            - Similar to a rotation in AVL
        - Otherwise if M has 1 KVP, remove N and merge intermediate value in parent P into sibling M; recurse upwards and promote if necessary
    - An **(a,b)-tree** is a tree where:
        - The root has at least 2 children
        - Each internal node has $a \leq k \leq b$ children
        - A node that has k children stores $k-1$ KVPs
        - Leaves store no KVPs and are on the same level
    - A **B-tree** of <u>order M</u> is a $(\lceil M/2 \rceil, M)$-tree
        - Height with n nodes $\in \Theta\left(\frac{\log n}{\log M}\right)$
        - Search, insert, delete $\in O(\log n)$
- **Extendible hashing**:
    - Use a B-tree with height 1 and max S KVPs in a node
    - Root node contains array (<u>directory</u>) of size 2^d where d = order
    - Each entry points to a block, which holds $\leq S$ items
    - Each block B has a <u>local depth</u> $k_B \leq d$
        - All values in B have the same first k_B bits
    - Search for k: lookup first d bits of $h(k)$ and load the block into memory
        - Search block for k
    - Insertion: search for k and find block B
        - If B has space, insert
        - Otherwise if B has S items and $k_B < d$, split B and increment k_B (if necessary)
        - Otherwise if B has S items and $k_B = d$, double the directory (increment d) and split B
    - Deletion: search and remove k
        - If possible, merge neighbour blocks (reverse of block split)
        - If every block has $k_B < d$, shrink the directory (reverse of directory grow)