CS 240 Midterm Review (Module 1–7)

Asymptotic Analysis

- Problem instance (I) – *input* for the specified problem
- Problem solution – *output* for the specified problem instance
- Problem size – Size(I) = size of instance I
- Algorithm - a step-by-step process for carrying out a series of computations
  - An algorithm A solves a problem P if, for every instance I of P, A computes a valid solution for I in finite time
- RAM model
  - Assume any memory access & primitive operation is constant time
  - Assume infinite amount of memory
  - Sequential operation
  - Running time is determined by the # of memory accesses & primitive operations
- 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
    - f is "upper-bounded" by 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
    - f is "lower-bounded" by 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
  - e.g. mergesort:
  - Step 1: split array of length n into two subarrays, of lengths $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ ($T = \Theta(n)$)
  - Step 2: recursively run mergesort on subarrays ($T = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$)
  - Step 3: merge sorted subarrays into a single sorted array ($T = \Theta(n)$)
  - Thus the recurrence relation is

$$
\begin{aligned}
T(n) &= \Theta(1) && \text{if } n = 1 \\
T(n) &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) && \text{if } n > 1 \\
&= 2T(\frac{n}{2}) + cn \\
&= 2(2T(\frac{n}{4}) + \frac{cn}{2}) + cn \\
&= \dots \\
&= 2^k T(\frac{n}{2^k}) + kcn && \text{where } k = \log n \\
&= nT(1) + \log n (cn) \\
&\in \Theta(n \log n)
\end{aligned}
$$

  - **In general**, $\{T(n) = T(n/2) + c\} \in \Theta(n \log n)$

Priorty Queues and Heaps

- **Priority queue**: an **abstract data type** containing a collection of items each with a priority
- **Heap**: binary tree with 2 structures
  - 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 = $\Theta(n)$

Sorting, Selection, Randomized Algorithms

- Every problem has an intrinsic cost/problem complexity = $C(n)$
- If a problem has complexity $\Omega(C(n))$, and an algorithm has worst-case runtime $O(C(n))$, then the algorithm is optimal
- **Selection problem**: find the k-th largest element within n elements
  - Using sorted array = $\Theta(n \log n)$
  - Using heap: heapify, then remove max from heap k times = $\Theta(n + k \log n)$
  - Using quick-select = $\Theta(n)$
- **Quickselect**:
  - Choose pivot = $\Theta(1)$
  - Partition:
    - Go from outermost pair inwards, swap any pairs that are in the wrong order
    - i.e. $++i$ and $--j$ until $A[i] > pivot$ and $A[j] < pivot$, then swap i and j
    - Return index pivot; array is now partitioned by the pivot value
    - $\Theta(n)$
  - Recursively call partition on one of the two partitions, until pivot index = desired index
  - Worse case: every recursive call paritions off 1 element = $\Theta(n^2)$
  - Best case: desired element is returned on first call = $\Theta(n)$
  - Average case: $\sum$ all runtimes for all permutations of the array / # of permutations ($n!$)
- **Quicksort**:
  - Same as quickselect, except recurse on both partitions instead of just one
  - Worse case = $\Theta(n^2)$
  - Best case = average case = $\Theta(n \log n)$
- **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 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 $\Omega(n \log n)$ (at least $n \log n$)
- Non-comparison based sorts can achieve faster than $\Omega(n \log n)$
- **Countsort**: input is array of size n which only contain numbers in a consecutive key set of k elements
  - Count the # of occurrences of each element (i.e. a histogram)
  - Calculate where each first key in key set
  - $\in \Theta(n + k) \in \Theta(n)$ if $k \in O(n)$
- **Radix sort**: represent all elements in base r (radix)
  - Pad with leading 0s so all elements have m digits
  - Sort elements into <u>buckets</u> (using count sort) based on their most/least significant digit
  - Make subsequent passes through every digit (r digits)
  - $\in \Theta(m(n + r)) \in \Theta(n)$ if $m, r \in O(n)$
- A sorting algorithm is <u>stable</u> if the order of equal (tied) keys are preserved (from the original order in the input)

Balanced Search Trees

- **Binary search tree**
  - A node's left subtree all have key values less than the root node
  - A node's right subtree all have key values greater than the root node
  - Search: start with root, binary search = $\Theta(\log n)$
  - Insert: search for closest existing node, insert as new leaf = $\Theta(\log n)$
  - 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
  - # 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 $E(L) = \sum_{i=1}^{n} P(x_i)T(x_i) = \sum_{i=1}^{n} P(x_i)i$
  - 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 $S_0 \ldots S_h$ containing keys in increasing order
    - Each starts and ends special keys $-\infty$ and $+\infty$
    - $S_h \subseteq S_h - 1 \subseteq \ldots \subseteq S_0$
  - Height at which new elements are inserted is $i = \#$ of heads flipped before a tail
    - $P = (1/2)^i$
  - Expected height $= O(\log n)$