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 $0 \leq f(n) \leq cg(n)\ \forall\ n \geq n_0$
    - f "grows no faster than" g
    - f is "upper-bounded" by g ($\leq$)
  - $f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)\ \forall\ n \geq n_0$
    - f "grows no slower than" g
    - f is "lower-bounded" by g ($\geq$)
  - $f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n \geq n_0$
    - f and g grow at the same rate
  - $f(n) \in o(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \leq f(n) < cg(n)\ \forall\ n \geq n_0$
    - f is "*strictly* upper-bounded" by g ($<$)
  - $f(n) \in \omega(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \leq cg(n) < f(n)\ \forall\ n \geq n_0$
    - f is "*strictly* lower-bounded" by g ($>$)
  - Suppose $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$
    - 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 $n \geq 2^{k+1} - 1$ (# of nodes including this level)
- **Bubble-up algorithm**: used for heap insertion
  - If node.key > node.parent.key then swap
  - Brings a large value from a leaf node up
- **Bubble-down algorithm**: used for heap deletion
  - If node.key < node.largest_child.key then swap
  - Brings a small node from the root down
- 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 ($n/2$ nodes)

- Total runtime = $\Theta(n)$

Sorting and Randomized Algorithms