MATH182 DISCUSSION 2 WORKSHEET

1. Sorting by selection and selection by sorting

Selection sort works as follows: Find the smallest element of the given array. Put it in the first position. Then find the next-smallest element, and put it in the second position. Continue until you've exhausted the array.

Write pseudocode for finding (the index of) the smallest element of an array between two indices. Then write pseudocode to implement selection sort. What are the loop invariants in each? What are their running times? Is your implementation of selection sort stable, and if not, can you modify it to be stable? In general, would you prefer to use insertion sort or selection sort?

Selection sort is a sort of dual to insertion sort. Both work by building up a sorted sub-array until it becomes the whole array. In insertion sort, we choose the next element based on its position in the unsorted array, then figure out where to put it in the sorted array. In selection sort, we choose the next element based on where it will be in the sorted array, then have to figure out where it is in the unsorted array. Is there a similar "dual" to merge sort? (*Hint:* the answer is Quicksort.)

Finding the position of a given value in an arbitrary array of length n takes $\Omega(n)$ time in the worst case—you need to test every element of the array. If the array is *sorted*, however, we can do better. The binary search algorithm searches a sorted array A for a given value v by comparing v to the middle element. If v is greater than the middle element, then it can only be in the right half of the array; if v is less than the middle element, then it can only be in the left half of the array. Show that binary search on an array of length n takes time $O(\lg n)$. (It may seem as though using binary search in insertion sort could improve its performance to $O(n \lg n)$. Unfortunately, you would still need to shift old elements out of the way, so this doesn't work. There are other data structures for which binary search works and insertion can be done quickly; this idea can lead to tree-based sorting.)

2. We could discuss asymptotics forever

Unless otherwise noted, all named functions in this section are assumed to be asymptotically positive.

Let $f, g, h : (1, +\infty) \to \mathbb{R}$ be asymptotically positive functions such that $f = O(g)$ and $h(x) \to +\infty$ as $x \to +\infty$ (i.e. $h(x) = \omega(1)$). Show that $f(h(x)) = O\big(g(h(x))\big)$. Does this also hold if O is replaced by o, Θ, Ω, or ω?

Suppose $f(n) = O(g(n))$. Define $F(m) = \sum_{n=1}^{m} f(n)$ and $G(m) = \sum_{n=1}^{m} g(n)$. Show that $F(m) = O(G(m))$. If $f(n) = o(g(n))$ and $g(n) \geq 1$ for sufficiently large n, show that then $F(m) = o(G(m))$.

Show that $O(f) \cdot O(g) = O(f \cdot g)$ and $O(f) + O(g) = O(f + g)$. Do corresponding equations hold for o, Θ, Ω, and ω? (*Warning:* The addition formula is not true if f and g are not assumed asymptotically positive—take e.g. $f = -g$.)

Suppose $f : \mathbb{N} \to \mathbb{R}$ is an increasing function and $f(2n) = O(f(n))$. Show that (i) $f(n) = O(n^{O(1)})$ (i.e. $f(n) = O(n^c)$ for some constant c), and (ii) $\sum_{k=1}^{n} f(k) = \Theta(nf(n))$.

Show that if $f(2n) = O(f(n))$, $g(2n) = O(g(n))$, and $\alpha > 0$, then $f(2n)g(2n) = O(f(n)g(n))$ and $f(2n)^\alpha = O(f(n)^\alpha)$. Verify that $f(n) = n$ and $f(n) = \log n$ satisfy $f(2n) = \Theta(f(n))$. Use the above to give asymptotic approximations to (i) $\sum_{k=1}^{n} \log^2 k$, (ii) $\sum_{k=2}^{n} \frac{k^2}{\log k}$, and (iii) $\sum_{k=1}^{n} \sum_{j=1}^{k} j^{3/2} \sqrt{\log j}$.

Show that for any $d \in \mathbb{N}$ and $c > 1$, $\sum_{k=1}^{n} k^d c^k = \Theta(n^d c^n)$

3. Let's play a game

The game of Nim is a game played between two people, with the following rules: There are several piles of stones on the ground. Each player, on their turn, may choose one pile, and take any positive number of stones from it. The player who takes the last stone wins.

We call a set of pile sizes a *position*. We say that a position is a *winning position for player 1* if the player with the first turn can ensure that they win. Otherwise, we call it a *winning position for player 2*. A position is a winning position for player 1 if and only if there is a legal move—a choice of pile and number of stones to take—which brings the game to a winning position for player 2.

Write a program (or pseudocode) to determine whether a given position is a winning position for player 1. As test cases, you are given that [3,4,5], [1,2,3,5], and [4,6,3] are winning positions for player 1, while [3,4,7] and [1,2,2,4,5] are winning positions for player 2. (*Hint:* You'll want to store information about smaller positions, like we did with FibonacciFast. However, dealing with an array of variable dimension takes some work, so it's easier to keep a table with values computed so far, and have the program attempt a table lookup before anything else, and store the result in the table after computing it.)