1 Notes on the programming assignment

1.1 General advice

1. Whereas the code is important, the write-up is equally important.
2. Remember that this assignment is experimental in nature — there is no fixed set of tests to run, rather you should run enough tests to answer the questions with some degree of sophistication.
3. **Test your code on small examples** (small enough that you can work them out by hand) to check for bugs.
4. It may take some time for your code to run and for you to get it to a performance-reasonable state; don't leave this problem set to the last minute.

1.2 Some specifics

1. Important decisions to be made before writing the program:
   (a) Programming language.
   (b) Graph representation: adjacency matrix versus adjacency list.
   (c) Prim's versus Kruskal's algorithm (both are entirely doable).
2. Be careful when (and how often) you seed the random number generator.

2 Heaps

Heaps are data structures that make it easy to find the element with the most extreme value in a collection of elements. A Min-Heap prioritizes the element with the smallest value, while a Max-Heap prioritizes the element with the largest value. Because of this property, heaps are often used to implement priority queues. As we saw in class, this is useful for Prim's algorithm (among many other things).

You can find more about heaps by reading pages 151–169 in CLRS.

2.1 Heap Representation

While a heap can be represented as a regular tree, it is often more efficient to store a binary heap as an array. We call the first element in the heap element 1. Now, given an element i, we can find its left and right children with a little arithmetic:

Exercise.

- $\textsc{Parent}(i) =$

- $\textsc{Left}(i) =$
- $\textsc{Right}(i) =$

The completeness requirement makes sure this representation of heaps is compact.

2.2 Heap operations

2.2.1 Max-Heapify

Max-Heapify(H, N): Given that the children of the node N in the Max-Heap H are each the root of a Max-Heap, rearranges the tree rooted at N to be a Max-Heap.

Max-Heapify(H, N):
Require: $\textsc{Left}(N)$, $\textsc{Right}(N)$ are each the root of a Max-Heap
  $(l, r) \leftarrow (\textsc{Left}(N), \textsc{Right}(N))$
  if $\textsc{Exists}(l)$ and $H[l] > H[N]$ **then**
    $largest \leftarrow l$
  else
    $largest \leftarrow N$
  end if
  if $\textsc{Exists}(r)$ and $H[r] > H[largest]$ **then**
    $largest \leftarrow r$
  end if
  if $largest \neq N$ **then**
    $\textsc{Swap}(H[N], H[largest])$
    Max-Heapify$(H, largest)$
  end if
Ensure: N is the root of a Max-Heap

Exercise.

- *Run* Max-Heapify *with* $N = 1$ *on*

$$H = [14,\ 16,\ 10,\ 8,\ 7,\ 9,\ 6,\ 2,\ 4,\ 1]$$

- *What is* Max-Heapify*'s run-time?*

2.2.2 Build-Heap

Build-Heap(A): Given an unordered array, makes it into a max-heap.

BUILD-HEAP(A):
Require: A is an array.
  for $i = \lfloor$ length$(A)/2 \rfloor$ downto 1 **do**
    MAX-HEAPIFY(A, i)
  end for

Exercise.

- *Run* BUILD-HEAP *on* $A = [2, 1, 4, 3, 6, 5]$
- *Running time (loose upper bound):*
- *Running time (tight upper bound):*

2.2.3 Extract-Max

Extract-Max(H): Remove the element with the largest value from the heap.

EXTRACT-MAX(H):
Require: H is a non-empty MAX-HEAP
  $max \leftarrow H[root]$
  $H[root] \leftarrow H[$SIZE$(H)]$ {last element of the heap.}
  SIZE$(H) -= 1$
  MAX-HEAPIFY$(H, root)$
  return max

Exercise.

- *Run* EXTRACT-MAX *on* $H = [6, 3, 5, 2, 1, 4]$.
- *What is* EXTRACT-MAX*'s run time?*

2.2.4 Insert

Insert(H, v): Add the value v to the heap H.

INSERT(H, v):
Require: H is a MAX-HEAP, v is a new value.
  SIZE(H) $+= 1$
  $H[\text{SIZE}(H)] \leftarrow v$ {Set v to be in the next empty slot.}
  $N \leftarrow$ SIZE(H) {Keep track of the node currently containing v.}
  while N is not the root and $H[\text{PARENT}(N)] < H[N]$ **do**
    SWAP$(H[\text{PARENT}(N)], H[N])$
    $N \leftarrow$ PARENT(N)
  end while

Exercise.

- *Run* INSERT*(H, v) with $v = 8$ and*
$$H = [6, 3, 5, 2, 1, 4]$$
- *What is* INSERT*'s runtime?*

Exercise. *Suppose that we use a heap for sorting. What is the runtime? What's a disadvantage of this sorting method?*

Exercise. *Explain how to solve the following two problems using heaps.*

1. *Give an $O(nlogk)$ algorithm to merge k sorted lists with n total elements into one sorted list.*

2. *Say that a list of numbers is* k-close *to sorted if each number in the list is less than* k *positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an* $O(nlogk)$ *algorithm for sorting a list of* n *numbers that is* k*-close to sorted.*

Exercise. *Describe an efficient algorithm that, given an undirected graph* G*, determines a spanning tree of* G *whose largest edge weight ("bottleneck") is minimum over all spanning trees of* G*.*