University of Waterloo
CS240 Spring 2016
Assignment 2

Due Date: Wednesday, June 1, at 5:00pm

Please read `http://www.student.cs.uwaterloo.ca/~cs240/s16/guidelines.pdf` for guidelines on submission. This assignment contains both written problems and a programming problem. Submit your written solutions electronically as a PDF with file name a01wp.pdf using MarkUs. We will also accept individual question files named a02q1w.pdf, a02q2w.pdf, ... , a02q4w.pdf if you wish to submit questions as you complete them.

Problem 5 contains a programming question; submit your solution electronically as a file named `countSort.cpp`.

Problem 1 [6 marks]

Module 2, Slide 12 describes an implementation of *heapify* that calls *bubble-down* for array indices $\lfloor n/2 \rfloor$ down to 0, in that order. Consider *bad-heapify* which calls *bubble-down* on the same array indices, except from 0 up to $\lfloor n/2 \rfloor$. Prove that the order of *bubble-down* operations is critical to the correctness of the *heapify* algorithm. In particular, provide an array A of size 10 such that *bad-heapify*(A) is not a heap and explain why *bad-heapify*(A) is not a heap. Also, draw the corresponding binary tree for array A.

```
bad-heapify(A)
A: an array
1.  n ← size(A) − 1
2.  for i ← 0 to ⌊n/2⌋ do
3.      bubble-down(A, i)
```

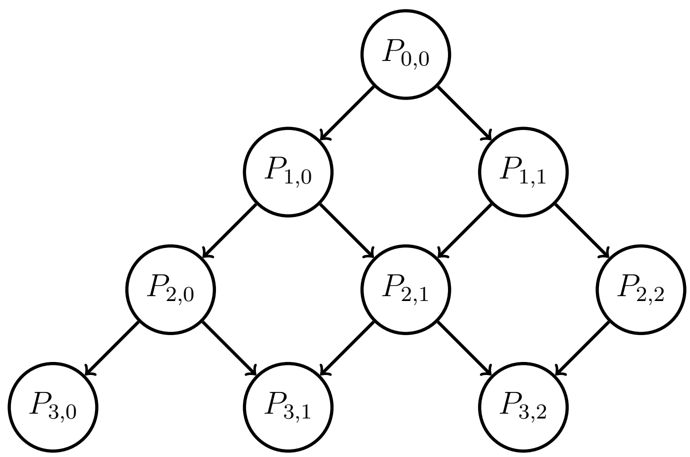
Problem 2 [3+(2)+4+5+5+8+(5)=25 marks]

A *pyramid* is a data structure similar to a heap that can be used to implement the priority queue ADT.

As with a heap, a pyramid is defined by two properties:

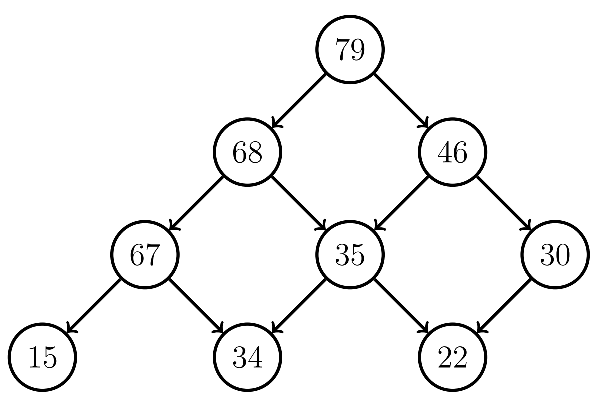
- **Structural property**: A pyramid P consists of $\ell \geq 0$ levels. The ith level, for $0 \leq i < \ell$, contains at most $i + 1$ entries, indicated as $P_{i,j}$ for $0 \leq j \leq i$. All levels but the last are completely filled, and the last level is left-justified.

  For example, the following diagram shows the structure of a pyramid with 4 levels and 9 nodes, labelled as described above.



- **Ordering property**: Any node $P_{i,j}$ has at most two *children*: $P_{i+1,j}$ and $P_{i+1,j+1}$, if those nodes exist. The priority of a node is always greater than or equal to the priority of either child node.

  For example, the following diagram shows the ordering property for a pyramid with 4 levels and 9 nodes. Priorities have been placed in the nodes and the arrows indicate "$\geq$" relationships.



a) [3 + (2) marks] A pyramid P with n nodes can be stored in an array A of size n, similar to an array-based heap. For example, the top of the pyramid $P_{0,0}$ will be stored in $A[0]$, followed by level 1, then level 2, and so on.

Give formulas for the array index that the following pyramid entries will have. Assume that n, i, and j are such that all indicated pyramid nodes actually exist. You do not need to justify your answers.

1. $P_{i,j}$
2. Left and right children of $P_{i,j}$
3. Left and right parents of $P_{i,j}$
4. **[1 bonus mark]** Left and right children of the node corresponding to $A[i]$
5. **[1 bonus mark]** Left and right parents of the node corresponding to $A[i]$

b) [4 marks] Give upper and lower bounds for the number of nodes n in a pyramid P with ℓ levels, where $\ell \geq 1$. For example, a pyramid with 2 levels has at least 2 and at most 3 nodes. Your bounds should be tight.

c) [5 marks] Give pseudocode for the *delete-max* operation that takes a pyramid stored in an array A of size n, removes the largest priority from the pyramid, and returns it.

Explain why your algorithm preserves the structural and ordering properties of the pyramid; i.e. show that the resulting tree is a pyramid (according to the definition above).

Show that your algorithm has time complexity $O(\sqrt{n})$.

d) [5 marks] Give pseudocode for the *insert* operation that takes a pyramid stored in an array A of size n and a priority x and inserts the new element into the pyramid.

Explain why your algorithm preserves the structural and ordering properties of the pyramid; i.e. show that the resulting tree is a pyramid (according to the definition above).

Show that your algorithm has time complexity $O(\sqrt{n})$.

e) [8 marks] Consider the *contains* problem: Given an array A of size n and an number x, determine whether x is an element of A.

For example, if A is sorted, the *contains* problem can be solved in $O(\log n)$ time by using a binary search.

Give pseudocode for an algorithm to solve the *contains* problem when the input array A is a pyramid as described above.

Show that the running time of your algorithm is $\Theta(\sqrt{n} \log n)$.
(Hint: A pyramid contains some sorted lists.)

f) **[5 Bonus marks]** Prove that any algorithm that solves the *contains* problem on an ordinary max-heap (i.e., as defined in class) must take $\Omega(n)$ time.

Problem 3 [6+6=12 marks]

The organizers of EURO 2016 are interested in buying footballs for the tournament. They have the option of buying them from n different companies. The main two factors for the organizers are the price and weight of the ball. In general, they prefer cheaper and lighter balls. They hire you as a computer scientist to come up with an algorithm to select the set of *reasonable* balls. A ball x is *not reasonable* if there exists another ball y in the input such that y is cheaper and lighter than x. You are given two arrays P and W as input where the ith ball has price $P[i]$ and weight $W[i]$. $P[i]$ and $W[i]$ are arbitrary real numbers.

a) Provide an efficient algorithm that finds all reasonable balls. Give a pseudocode of your algorithm and analyze its running time.

b) Assume that we additionally know that $P[i]$ is an integer between 1 and 200 for each $0 \leq i < n$. Provide a more efficient algorithm for the problem and analyze its running time.

Problem 4 [4+6+(5)=10 marks]

A *deterministic* algorithm is one whose execution depends only on the input. By contrast, the execution of a *randomized* algorithm depends also on some randomly-chosen numbers. A *Las Vegas* randomized algorithm always produces the correct answer, but has a running time which depends on the random numbers chosen (randomized quick-select and quick-sort are of this type). Informally, such algorithms are always correct, and probably fast. A *Monte Carlo* randomized algorithm has running time independent of the random numbers chosen, but may produce an incorrect answer. Informally, such algorithms are always fast, and probably correct.

Given an array A of length n, an element x is said to be *dominant* in A if x occurs at least $\lfloor n/2 \rfloor + 1$ times in A. That is, copies of x occupy more than half of the array.

a) Given an array A that contains a dominant element, describe a Monte Carlo randomized algorithm to find the dominant element. Show that your algorithm has worst-case running time $O(1)$ and returns the correct answer with probability at least 1/2.

b) Given an array A that contains a dominant element, describe a Las Vegas ran- domized algorithm to find the dominant element. Show that your algorithm always returns the correct answer, and has expected-case running time $O(n)$.

c) [**Bonus**] Given an array A that contains a dominant element, describe a deterministic algorithm to find the dominant element. Show that your algorithm has worst-case running time $O(n)$.

For partial marks, you may assume that array elements are comparable, and you may use any algorithm from class as a subroutine.

For full marks, you may only assume that array elements can be tested for equality/inequality (not greater/less than). (Note: this is a tricky algorithm.)

Problem 5   [10 marks]

For this problem you will implement a slightly modified version of the `counting-sort` algorithm as presented in the course slides. The input to the algorithm will be an array A of n integers such that the difference between the maximum and minimum integers in A is at most k. Note that

- The value of k is not an input parameter of the function and is not known at the start of the function.
- The numbers in the input array do not necessarily lie between zero and k.

Your implementation should have runtime $\Theta(n+k)$ when sorting an array with n elements. Thus, your implementation should be $\Theta(n)$ when $k \in O(n)$.

Implement your algorithm in a file called `countSort.cpp`. Your program should read n+1 integers from `cin`, each on their own line. The first number read is the size of the array, n, and the next n numbers are the elements of the array. Your program should then write to `cout` the n elements of the array in sorted order (increasing order), each on it own line.

The following is an example of input and the correct output from `countSort`:

Input

```
5
17
7
13
2
9
```

$\xrightarrow{countSort}$

Output

```
2
7
9
13
17
```

Submit the code for your `main` function, along with any helper functions, in a file called `countSort.cpp`.

Note: All marks for this problem will be correctness marks of the auto-testing (provided your code implements counting sort). You do not need to submit any written justification or analysis of your algorithm.