University of Waterloo
CS240 - Spring 2014
Assignment 3

Due Date: Wednesday June 18 at 09:15am

Please read `http://www.student.cs.uwaterloo.ca/~cs240/s14/guidelines.pdf` for guidelines on submission. Problems 1 – 6 are written problems; submit your solutions electronically as a PDF with file name `a03wp.pdf` using MarkUs. We will also accept individual question files named `a03q1w.pdf`, `a03q2w.pdf`, . . . , `a03q6w.pdf` if you wish to submit questions as you complete them.

There are 67 marks available.

Problem 1 [$5 + 5$ marks]

A *stable* sorting algorithm is one in which the relative order of all identical elements (or keys) is the same in the output as it was in the input.

1. Is Heapsort a stable sorting algorithm ?

   If yes, explain the stability and give a formal proof. Otherwise, provide a counterexample with an explanation.

   Heapsort was determined to not be a stable sorting algorithm. The following counterexample illustrates the instability of heapsort specifically using the heapify algorithm described in class:

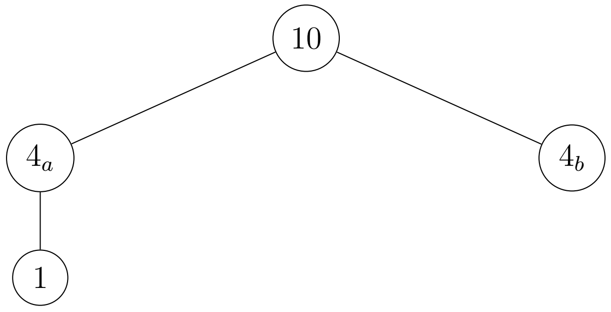
   ```
   heapify(A)
   A: an array
   1. n = size(A) - 1
   2. for i = floor(n/2) down to 0 do
   3.    bubble_down(A,i)
   ```

   The instability of heapsort is in its decision to bubble down when its children are the same value.

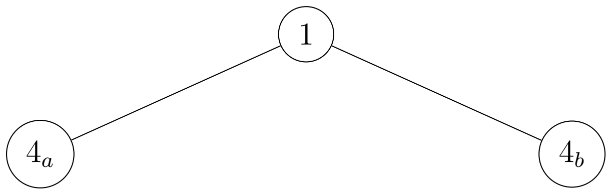
   For example:

   Let array M = [10, 4_a, 4_b, 1]

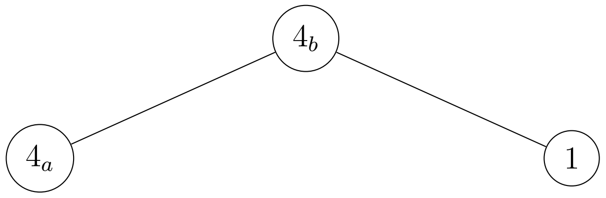
   The bubble downs have no effect on position 1 or 0, since this heap example is already ordered. We then begin the deletes:



Then:



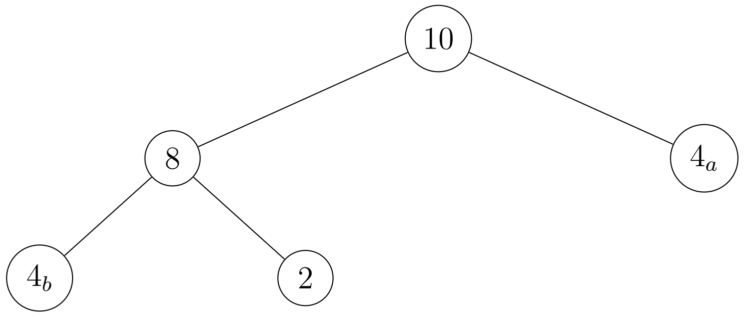
If order is to be preserved, we must pick to swap on the right side resulting in:



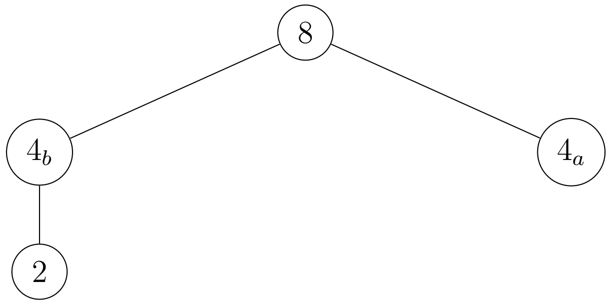
and producing a final array of: [1,4_a, 4_b,10] which has preserved order.

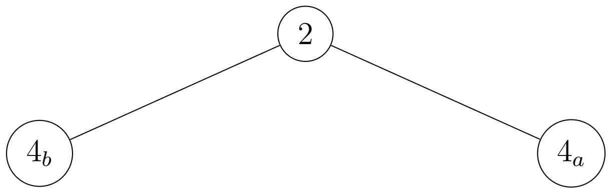
But if we were to similarly select the right side on another heap:

B = [10, 8, 4_a, 4_b, 3] Since the bubble downs on 2,1,0 have no effect, they are skipped.

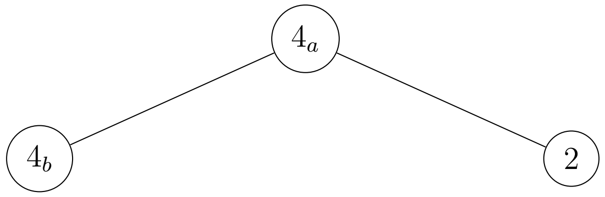


Then:





Now, picking the right side again we get:



And a final array of: [2,4_b,4_a,8,10]

It was necessary to pick the right side in the first heap to maintain stability and the left side in the second heap. Therefore regardless of which side is chosen, there exist heaps such that the heap stability cannot be ensured. Heapsort therefore is not a stable sorting algorithm.

2. Consider the Quicksort1 algorithm described in class that uses the partitioning method of slide 5 of module 3 and takes $A[0]$ as pivot. Is Quicksort1 a stable sorting algorithm ?

   If yes, explain the stability and give a formal proof. Otherwise, provide a counter-example with an explanation.

   Quicksort is also not a stable sorting algorithm. Example:

   A = [1_a,1_b,4,2,1_c]

   with 1_a as pivot, the first partition results in i at position 2 and j at position 4. Swapping we get:

   [1_a,1_b,1_c,2,4]

   Then i = 3, j = 2, so swap A[0] and A[j] since j < i

   [1_c,1_b,1_a,2,4] returning j = 2

   recursively call quicksort on [1_c, 1_b] and [2,4]

for [1_c, 1_b], i = 2, j = 1, $j < i$ so swap A[0], A[j], to get 1_b, 1_c and returning j = 1
for [2,4], i = 1, j = 0, j < i, swap A[0], A[j], which remains 2,4, returning j = 0

Now quick sort will return since all partitions are of size 1 and the sorted array is:

[1_b, 1_c, 1_a, 2, 4] which is not in the same order as we began. Therefore this quicksort is not stable.

Problem 2 [10 marks]

Given an array $A[0, \ldots, n-1]$ of numbers with the following property: $A[i] \geq A[i-j]$ for all $j \geq \log n$.

1. Show how to sort the array A in $O(n \log\log n)$ time.

Hint: Partition A into contiguous blocks of size $(\log n)$, i.e., first $(\log n)$ elements are in the first block, next $(\log n)$ elements are in the second block, and so on. Then, establish a connection between the elements within two blocks, which are separated by another block.

First begin by partitioning A into contiguous blocks 1 to n of size $\log n$. By the property described, we know that all elements in block n are greater than all elements in block n-2. We begin the sort by inserting all elements in block n into a heap using heapInsert, followed by all elements in block n-1. There are then a total of $2\log n$ elements in this heap. The bubble ups as a result of heapInsert could then at most cost the height of the tree, which would be $\log 2 \log n$.

In the next step, use deleteMax to remove $\log n$ elements from the current heap. This delete operation is the same complexity as the inserts, which is again $\log 2 \log n$. We now total $2 \log 2 \log n$ operations to insert and then delete an element, which is O($\log\log n$). The result of $\log n$ deletions produces the sorted top $\log n$ elements.

After removing $\log n$ elements, we insert the next largest block, which can bubble up again with complexity $\log 2 \log n$ according to heapInsert. After inserting these elements, we delete again, with complexity $\log 2 \log n$. We now have the top $2 \log n$ elements sorted, and continue this process until no more blocks remain.

Since the complexity of inserting and then deleting elements from the $\log 2n$ sized heap is O($\log\log n$) for each element and this must be done n times, the total cost is O(n$\log\log n$).

Problem 3 [5 + 5 **marks**]

A sorting algorithm is said to sort *in place* if only a constant number of elements of the input are ever stored outside the array. In class we showed that all comparison based sorting algorithms require $\Omega(n \log n)$ comparisons to sort an array of length n. But suppose are given an array $A[0, \dots, n-1]$ that contains a permutation of the first n non-negative integers $0, \dots, n-1$.

1. Allowing non comparison based algorithms, give an $O(n)$ in place algorithm to sort A. For this question, you only have to give the algorithm without justification.

```
1. let A be an array of size n
2. for i = 0 to n - 1 do
3.    while A[i] != i
3.      swap A[i], A[A[i]]
```

2. Analyze the worst-case running time of your method and prove that it is $O(n)$.

Note: For simplicity we are assuming A is filled with integer keys. Your algorithm must easily extend to work for an array A that is filled with (key,element) pairs, each integer key in the range $0, \dots, n-1$ occurring exactly once.

Time complexity:

The worst case time complexity is when all of the elements are out of order by 1 position. After each swap, an element is now in the correct place and will never be swapped again. At most there could then be n-1 swaps.

Each element in the array is compared at least once, and if a swap is required, it will be compared again afterwards to see if the swap resulted in the element now being at its correct location. There are then at least n comparisons in the worst case to compare all swaps, then an additional n-1 comparisons to iterate over the elements in the array.

Worst case:
1234567890 compare, then swap = 1+c complexity
2134567890 compare, then swap = 1+c complexity
3124567890 compare, then swap = 1+c complexity
4123567890 compare, then swap = 1+c complexity
5123467890 compare, then swap = 1+c complexity
6123457890 compare, then swap = 1+c complexity
7123456890 compare, then swap = 1+c complexity
8123456790 compare, then swap = 1+c complexity
9123456780 compare, then swap = 1+c complexity

0123456789 compare, 1 complexity

Followed by n-1 comparisons as it iterates over the remaining now sorted elements.

This totals complexity T(n):

$$\begin{aligned}T(n) &=(\sum_{i=1}^{n-1} 1 + c) + \sum_{i=1}^{n-1} 1 \\ &= (n-1)(1+c) + n - 1 \\ &= n - nc - 1 + n - 1 \\ &\therefore T(n) \in O(n)\end{aligned}$$

Problem 4 [10 **marks**]

You are given an unsorted array $A[0,\ldots,n-1]$ filled with distinct integers. For a given k, $1 \le k \le n$, we want to rearrange the array so that $A[0,\ldots,k-1]$ contains the k smallest integers in increasing order.

1. Describe an in-place algorithm for this problem. If $k \in O(n/\lg n)$ your algorithm should have running time $O(n)$.

This sorting algorithm is composed of several different steps:

1. First begin by quickselecting the kth smallest element. Quickselect is used to find the element in position k of a sorted array, which is useful for our algorithm. According to Module 3 slide 2, this can be done in O(n) time.
2. The next step is to use the partition algorithm to partition(A, p) from slide 5 of module 3, on array A and the pivot returned from quickselect in order to get all elements smaller than the kth smallest element to its left in the array. This would require at most the ceiling of $\frac{n}{2}$ swaps, which is O(n) complexity.
3. Finally, we then take all elements to the left of the kth smallest element and perform any fast comparison based sorting algorithm, such as quick sort, with an average time complexity of O($(k-1)\log(k-1)$). This is O(k$\log k$) time complexity.

If k $\le \frac{n}{\log n}$, then the total complexity of this algorithm would then total T(n):

$$
\begin{aligned}
T(n) &= O(n) + O(n) + O(k \log k) \\
&= O(n) + O(n) + O(\frac{n}{\log n} \log \frac{n}{\log n}) \\
&= O(n) + O(n) + O(\frac{n}{\log n} \log(\log n - \log\log n)) \\
&= O(n) + O(n) + O(n) - O(n\frac{\log\log n}{\log n}) \\
&= 3O(n) - O(n) \\
&\therefore T(n) \in O(n)
\end{aligned}
$$

Problem 5 [$8 + 6 + 6 = 20$ **marks**]

A tug of war is a contest in which two teams of players pull on a rope in opposite directions. The team with the greater combined strength wins (we assume that strengths are perfectly additive, and that there is no element of chance). For this problem, you are given n players, that are either weak or strong. All strong players have exactly the same strength, and all weak players similarly have exactly the same strength. We also assume that there is at least one weak and one strong player.

The task at hand is to determine which players are strong and which are weak. Your tool to determine this is to assign players to teams and have contests. The outcome of a single contest can either be a tie, one team winning, or the other team winning.

1. [6 marks] Give a precise (not big-Omega) lower bound for the number of contests required in the worst case to determine which players are strong and which are weak.

   The worst case is where every player but the last is the same. Begin with player one and compare them with every other player to determine player one. Iterating over every player, there is guaranteed to be someone who will not tie with player one, in this case the last one. In this worst case it would take n-1 comparisons of the first player with the others. After player one has been determined, everyone else is known since they have already been compared with player one. Therefore the precise lower bound for the number of contests in the worst case would be n-1.

2. [7 marks] Describe an algorithm called `find-strong` to determine the strong players when $n = 4$. Use the names P_1, P_2, P_3, P_4 for the four players, and the function

   $$\texttt{contest}\left(\{\text{first-team}, \text{second-team}\}\right),$$

   which returns either "first wins", "second wins", or "tie". Your function should return a set or list of the strong players.

Give an exact worst-case analysis of the number of contests required in your algorithm. For full marks, this should match exactly the lower bound from Part 1 when $n = 4$.

```
find-strong(P)
  firstPlayer = unknown
  A= empty array of strong players
  B= empty array of players that were tied
  for i = 2 to n do
    if contest({P1, Pi}) == "tie"
      if firstPlayer == unknown
        B.push(Pi)
      else if firstPlayer == strong
        A.push(Pi)
    else if contest({P1, Pi}) == "first wins"
      if firstPlayer == unknown
        firstPlayer = strong
        A.push(P0)
        A = A + B
    else if contest({P1, Pi}) == "second wins"
      if firstPlayer == unknown
        firstPlayer = weak
      A.push(Pi)
  return A
```

Setting up array P for our worst case when n=4, P=[1,1,1,0] where 1 is strong and 0 is weak

The algorithm woulds step through, contest P1 with P2, resulting in a tie and pushing P2 to the unknown bucket. It would repeat this for P3, and then contest P1 with P4, determining that P1, P2, and P3 must all be strong, returning these three players. This took n-1 = 4-1 comparisons, matching the lower bound from the previous question.

3. [7 marks] Describe an algorithm to determine the strong and weak players, for any n. Use the names $P_1, P_2, \ldots, P_n$ and the contest subroutine from Part 2. Show that your algorithm is asymptotically optimal, meaning that the big-O cost should match the lower bound from Part 1.

The algorithm works by taking the first player and comparing him to all other players. The big-O cost T(n) for this algorithm would be:

$$\begin{aligned}T(n) &= \sum_{i=2}^{n} c\\ &= (n+1-2)c\\ &= (n-1)c\end{aligned}$$

Where c is the constant complexity of one contest in combination with a fixed number of comparisons and assignments to the arrays.
By the definition of big-O this would be O(n-1) complexity, matching the lower bound of n-1 from the worst case above.

the algorithm is defined as:

```
find-strong(P)
  firstPlayer = unknown
  A= empty array of strong players
  B= empty array of players that were tied
  C= empty array of weak players
  for i = 2 to n do
    if contest({P1, Pi}) == "tie"
      if firstPlayer == unknown
        B.push(Pi)
      else if firstPlayer == strong
        A.push(Pi)
      else
        C.push(Pi)
    else if contest({P1, Pi}) == "first wins"
      if firstPlayer == unknown
        firstPlayer = strong
        A.push(P1)
        A = A + B
    else if contest({P1, Pi}) == "second wins"
      if firstPlayer == unknown
        firstPlayer = weak
        C.push(P1)
        C = C + B
      A.push(Pi)
```

Problem 6 [7 marks]

Assume that you are given an array A of n integers such that $0 \leq A[i] < k$ for all $0 \leq i < n$. We would like to create data structures to answer the queries of the following type:

"How many integers in A are in the range $[a, b]$, given integers a and b with $0 \leq a \leq b < k$?"

1. Describe a preprocessing algorithm that runs in time $O(n + k)$ and creates a data structure that allows answering the queries in constant time.

Begin by following the counting sort procedure derived in class. For n,k start by creating a separate array B of size k. For each value in array A, increase the corresponding index of array B by 1 as counting sort would. Then step through B, making each element the sum of all previous elements.

At this point, stop applying the counting sort procedure. The complexity would then be O(n) + O(k) or O(n+k) to create the data structure B.

For a given a,b the access time would then be the time to access A[b] and A[a-1], since the answer, the number of integers in the range [a,b], is given by A[b] - A[a-1]. This would only require two constant time accesses to the array.

The correct number is A[b] = A[a-1] since we want the number of integers between a and b. A[b] includes all integers b added on the upper bound, but to include all integers a on the lower bound you need to subtract A[a-1] instead of A[a], as the latter would remove occurrences of a.