Homework 4
Due: Thursday 3 November 2022 by 11:59 PM

Instructions. Type your answers to the following questions and submit as a PDF on Gradescope by the due date and time listed above. (You may write your solutions by hand, but remember that it is at your own risk as illegible solutions will not receive credit.) Assign pages to questions when you submit. **For all questions, you must show your work and/or provide a justification for your answers.**

Note on Academic Dishonesty. Although you are allowed to discuss these problems with other people, your work must be entirely your own. It is considered academic dishonesty to read anyone else's solution to any of these problems or to share your solution with another student or to look for solutions to these questions online. See the syllabus for more information on academic dishonesty.

Grading. Some of these questions may be graded for completion and some for accuracy.

Question 1.
Consider an array-based Stack that is dynamically resized. For each of the following, you can assume that the arrays start as size 1. You should show your work using a summation, but you also need to explain the context.

(a) Show mathematically that the amortized cost of *push* is $O(1)$ if the size of the stack is multiplied by a constant *c* each time it becomes full (i.e. the sizes will be $1,\ c,\ c^2,\ c^3$...).

(b) Show mathematically that the amortized cost of *push* is $O(N)$ if the size of the stack is increased by a constant amount c each time it becomes full (i.e. the sizes will be $1,\ 1+c,\ 1+2c,\ 1+3c$,...).

(a) Assuming that after *N* pushes, the array is of size *N* (and full), the total number of array accesses is

$$N + 2(1 + c + c^2 + c^3 + \ldots + N/c) = N + 2\sum_{k=0}^{\log_c(N/c)} c^k = N + 2\left(\frac{c^{\log_c(N/c)+1}-1}{c-1}\right)$$

$$= N + 2\left(\frac{c^{\log_c N - \log_c c + 1}-1}{c-1}\right) = N + 2\left(\frac{N-1}{c-1}\right).$$

Explanation of the math: The first *N* is counting one access for each of the *N* pushes. The other part (the summation) is counting 2 accesses for copying over the elements whenever the array is resized. The first time this happens, 1 element is copied over, then *c* elements, then c^2elements, and so on. Since the

array is assumed to be full after the *N* pushes, the last time it was resized was when there were N/c elements to copy over. The summation is a geometric series with *c* as the base.

To get the amortized cost of each push, we should take that total and divide it over the *N* pushes. It's clear that the total number of accesses is O(*N*), so dividing that over *N* pushes, will give an amortized cost that is O(1).

(b) Assuming that after *N* pushes, the array is of size *N* (and full), the total number of array accesses is

$N + 2(1 + (1 + c) + (1 + 2c) + (1 + 3c) + \ldots + (N - c)) = N + 2\sum_{k=0}^{\frac{N-c-1}{c}}(1 + kc) =$

$= N + 2\left(\frac{N-c-1}{c} + 1 + c\left(\frac{\frac{N-c-1}{c}\left(\frac{N-c-1}{2}+1\right)}{2}\right)\right)$. (There is no real need to simplify this out as it is clear that the overall total is O(N^2).

Explanation of the math: The first *N* is counting one access for each of the *N* pushes. The other part (the summation) is counting 2 accesses for copying over the elements whenever the array is resized. The first time this happens, 1 element is copied over, then *1+c* elements, then *1+2c* elements, and so on. Since the array is assumed to be full after the *N* pushes, the last time it was resized was when there were $N - c$ elements to copy over. The summation is arithmetic.

To get the amortized cost of each push, we should take that total and divide it over the *N* pushes. It's clear that the total number of accesses is O(N^2), so dividing that over *N* pushes, will give an amortized cost that is O(*N*).

Question 2.
Imagine that you are implementing a queuing system for a theater company. For some of the more popular shows, they need a reliable queuing system for handling ticket requests when they go on sale. At this particular theater, people can be members or non-members, and members always get precedence over non-members when it comes to buying tickets. Other than that, the requests should be handled on a first-come first-serve basis. Describe an implementation of this system and discuss your implementation in terms of space and runtime efficiency. You may write pseudocode if you want, but it is not required.

You could do this pretty simply with two queues. The first queue is the "VIP" queue for members and the second is the regular queue for non-members. If a member requests a ticket, put them in Queue 1, and if a non-member requests a ticket, put them in Queue 2. When handling the

requests, always handle requests in Queue 1 first and look at Queue 2 only if/when Queue 1 is empty.

Assuming the Queues are growable and can enqueue and dequeue in O(1) time (possibly amortized), the space is proportional to the number of requests and the time is proportional to the number of requests as well.

Question 3.
Explain the best and worst case runtimes for each of the following algorithms. Justify your answer with respect to how the algorithm works and with mathematical analysis. Note that although it's good if you know the "trivial" best cases, we want the non-trivial best cases here.

(a) Insertion Sort
(b) Selection Sort
(c) Mergesort
(d) Heapsort
(e) Quicksort

(a) Best case: O(N). This is because the algorithm scans each element and inserts it into the sorted array (the previous elements in the array). If the array is already sorted, each new element will be compared with the previous element, and it will be found to already be in order, so we just move on. That is basically just a single scan through the array. Worst case: $O(N^2)$.This is because in the worst case the array is sorted in reverse direction, and the second element has to compare/swap once, the third element has to compare/swap twice, etc...and the N^{th} element has to compare/swap N-1 times. That is a total of $1 + 2 + 3 + \ldots + N - 1 = \sum_{k=1}^{N-1} k = \frac{(N-1)(N)}{2}$ times, which is $O(N^2)$.

(b) The best and worst case are both $O(N^2)$ because this algorithm does not perform differently on different inputs. An outer loop scans through the positions in the array, and an inner loop scans the remaining elements in the array to find the minimum. To find the minimum, you have to scan all of the remaining elements in order to guarantee that you have the minimum. So the total amount of work is $N + N - 1 + N - 2 + \ldots + 1$, which sums to an $O(N^2)$ function.

(c) The best and worst case is O(NlogN). Regardless of the input, the algorithm recursively divides the input into halves and merges the halves together. The recurrence relation for this is T(N) = 2T(N/2) + N with a base case of T(1) = 1. This is because an array of one element requires only constant work because it is already sorted and cannot be further divided. The work on N elements is O(N) for the merging plus the work required for doing two recursive calls, each on half of the input. At the next level,

you have the work required for merging two arrays of size N/2, which is N/2 + N/2 = N plus the work required for the 4 recursive calls, each on size N/4. In a recursion tree, we can see that at each level, O(N) work is being done for the merging, and the height of the tree is logN because it takes logN levels to get from N to 1 when dividing by 2 each time. That means the total amount of work is O(NlogN). This can also be shown using the Master Theorem.

(d) First, bottom-up heap construction takes O(N) time because given N nodes, the amount of work required for merging the heaps together is always going to be O(N). Second, once the heap is constructed, we do sinkdown. In the worst case, every time we swap the maximum value with the last value, that value would have to sink all the way down the heap, but the heap is shrinking by 1 each time. So the work required would be $log(N-1) + log(N-2) + \ldots + log2 = log((N-1)!)$, which is O(NlogN).

The best case is typically O(NlogN) as well. However, there is a very specific edge case that results in O(N) runtime, and that is when every element in the array is the same. That means that the time for bottom-up heap construction is simply ~N/2 because you just have to scan half the elements and nothing sinks, and the time for sinkdown is O(N) as well because nothing ever has to sink.

(e) In the best case, the pivot value is always the median, which divides the input evenly into subarrays of size N/2. This would give us a recurrence relation of T(N) = 2T(N/2) + N. The +N is because the amount of work for dividing up the array is linear because it requires scanning through the array. From the Master Theorem (and other justifications), we know that T(N) is O(NlogN). In the worst case, the pivot value chosen is always the maximum or the minimum, which means that the array is divided into subarrays of 0 elements and N-1 elements. This indicates a recurrence relation of T(N) = T(N-1) + N. If you expand that out, you get a total runtime of N + N-1 + N-2 + … + 1, which is an arithmetic series, summing to a quadratic function. So the worst case is $O(N^2)$.

Question 4.

Write and analyze a version of Quicksort called Five-way Quicksort that uses two pivots (p1 and p2) and divides the array into five subarrays–less than p1, equal to p1, between p1 and p2, equal to p2, and greater than p2. You should write your solution in pseudocode, and your analysis should include discussions of both best and worst case and should give the runtimes as recurrence relations and in big-Oh.

```
Algorithm FiveQuick
Input: An array A of size N
Output: A, sorted
pivot(A, 0, A.length-1)
```

```
procedure pivot (Array A, int i, int j):
      if(i >= j)
          return
      if(i < 0 || i >= A.length || j >= A.length)
          return

      int r1 = a random integer between i and j
      int r2 = a different random integer betwee i and j
      x1 = min(A[r1], A[r2])
      x2 = max(A[r1], A[r2])
      swap A[i] and x1
      swap A[j] and x2

      int p1 = i;
      int p2 = j;
      int f1 = i + 1;
      int f2 = j - 1;
      int k = i + 1;
      while(k <= f2 && f1 <= f2) {
          int kVal = A[k];
          if(kVal < x1) {
            swap A[f1] and A[k]
            swap A[p1] and A[f1]
            p1++;
            f1++;
            k++;
          } else if(kVal == x1) {
            swap A[f1] and A[k]
            f1++;
            k++;
          } else if(kVal > x2) {
            swap A[k] and A[f2]
            swap A[p2] and A[f2]
            p2--;
            f2--;
          } else if(kVal == x2) {
            swap A[f2] and A[k]
            f2--;
          } else {
            k++;
          }
      }
      pivot(A, i, p1-1);
```

```
      pivot(A, f1, f2);
      pivot(A, p2+1, j);
    }
```

Analysis:
In the best case, the pivots evenly divide the array into 3 subarrays, so the runtime is T(N) = 3T(N/3) + N; T(1) = T(2) = 1. This is because, it would do three recursive calls on N/3 and the additional +N is for pivoting. Using the Master Theorem, this comes out to be O(NlogN).

In the worst case, the pivots do not divide the array up evenly and instead make for two subarrays of size 0 and one subarray of size N-2. The runtime would then be T(N) = T(N-2) + N; T(1) = T(2) = 1. This is because it would end up doing one recursive call on N-2 (the data minus the two pivots) and the +N comes from pivoting. This runtime, when expanded out, comes out to be $O(N^2)$.

Question 5.

Prove by induction that the worst-case runtime of Quicksort is $O(N^2)$. Hint: Let $N_0 = 1$ and $c = 2$.

The recurrence relation for the worst case of Quicksort is
$T(N) = T(N-1) + N; T(1) = 1.$

Conjecture: $T(N) \leq cN^2 \; for \; some \; c > 0 \; and \; for \; all \; N \geq 1. \; Let \; c = 2.$

Basis Step: $T(1) = 1 \leq 2(1)^2 = 2$

Inductive Step:

Assume as the inductive hypothesis that $T(k-1) \leq 2(k-1)^2$ for some integer $k - 1 \geq 1$.We must show that $T(k) \leq 2k^2$.

$T(k) = T(k-1) + k$

$\leq 2(k-1)^2 + k \; by \; the \; inductive \; hypothesis$

$= 2(k^2 - 2k + 1) + k$

$= 2k^2 - 4k + 2 + k$

$= 2k^2 - 3k + 2$

$\leq 2k^2$ because since $k - 1 \geq 1, k \geq 2$, so $-3k + 2 \leq -4.$