This exam is closed book, notes, computer and cell phone.
Show partial solutions to get partial credit.
Make sure you have answered all parts of a question.
If your solutions are not written legibly, you won’t get full credit.
Clarity and succinctness will be rewarded.

Question 1: __________(out of 20)
Question 2: __________(out of 10)
Question 3: __________(out of 15)
Question 4: __________(out of 15)
Question 5: __________(out of 10)
Question 6: __________(out of 10)
Question 7: __________(out of 10)
Question 8: __________(out of 10)
Total:———(out of 100)
1. Short questions:

(a) Give 4 operations efficiently implementable with the array implementation of a heap. For each operation give the cost in $O$-notation as a function of the size $n$ of the heap.

- **ExtractMax**: $O(1)$
- **DeleteMax**: $O(\log n)$
- **InsertKey**: $O(\log n)$
- **ChangeKey***: $O(\log n)$
- **Delete***: $O(\log n)$

* for these operations you need to know the index of the key to be changed/deleted

(b) What is the comparison tree lower bound for sorting and what is the key operation the sorting algorithm is allowed to use for determining the order of keys?

- **Key operation**: comparison
- **Lower bound**: $\log_2(n!) = \Omega(n \log n)$

(c) Which of the following sorting algorithms are stable: InsertionSort, MergeSort, QuickSort, CountingSort, HeapSort?

- **All but HeapSort and Quicksort**.

(d) What is the worst-case and average-case running time of QuickSort? When does the worst-case occur?

- **Worst-case**: $O(n^2)$; occurs when pivot is always largest or smallest element
- **For QuickSort this occurs when keys are sorted**
- **Average case**: $O(n \log n)$

(e) Give an algorithm for finding the largest and the second largest key among distinct $n$ keys? The straightforward algorithm requires $2n - 3$ comparisons. Give a 3 sentence description of an algorithm that requires fewer (how many?) comparisons.

- **Hint**: Find largest using a tournament and ask yourself how many more comparisons it takes to find 2nd largests.

- **The tournament takes $n - 1$ comparisons. The second largest must have lost directly against the largest. There are $\leq \lceil \log_2 n \rceil$ keys that were compared directly to the largest. In $\leq \lceil \log_2 n \rceil - 1$ additional comparisons you can find the largest among the direct losers.**
- **Total**: $n + \lceil \log_2 n \rceil - 2$. 

3
2. Show by induction that the number of leaves of a ternary tree of height $h$ is at most $3^h$.

Definition:
- In a ternary tree a node has at most 3 children.
- The height of a tree is the longest number of edges on any path from the root to a leaf.

$P(h)$: Any tree of height $h$ has $\leq 3^h$ leaves.

I) Strong induction:

Base case: single leaf, height 0, $P(0)$ clearly holds since $3^0 = 1$.

Assume $P(h')$ holds for $0 \leq h' \leq h - 1$. To show $P(h)$.

Given any tree of height $h$. Removing the root leaves us with up to 3 subtrees, all of which have height $\leq h - 1$. (Note that up to two of the subtrees might be empty.) By the inductive hypothesis each such subtree can contribute at most $3^{h'} \leq 3^{h-1}$ leaves, where $h' \leq h - 1$ is the height of the subtree. Since there are at most 3 subtrees, the number of leaves is $\leq 3 \cdot 3^{h-1} = 3^h$.

II) Weak induction:

Base case is the same.

Assume $P(h - 1)$ holds for $0 \leq h - 1$. To show $P(h)$.

Given a tree $T$ of height $h$. Let $T'$ be $T$ with all leaves of the bottom layer removed. Clearly $T'$ has height $h - 1$. By the inductive hypothesis $T'$ has $\leq 3^{h-1}$ leaves. Each leaf of $T'$ either stays a leaf of $T$ or it becomes an internal node and begets up to 3 leafs of $T$. Each leaf of $T$ either was a leaf of $T'$ or up to three leaves share a parent that was a leaf of $T'$. Thus $T$ has $\leq 3 \cdot 3^{h-1} = 3^h$ leaves.

III) Simplified: Any ternary tree of height $h$ can be “filled out” to a “complete ternary tree” of height $h$ by adding nodes. In such complete trees all internal nodes have degree 3 and all leaves are at the bottom level. This filling out process only increases the number of leaves. Thus it suffices to show the claim only for complete trees. For complete trees, you only need to show that the bottom layer of such trees has $3^h$ nodes. 3 nodes on the bottom layer share a parent on the previous layer. By induction there are $3^{h-1}$ parents on the previous layer. Thus the bottom layer has exactly $3 \cdot 3^{h-1} = 3^h$ nodes.
3. Describe an algorithm that does the following. It is given \( n \) integers in the range 0 to \( k \). After preprocessing its input the algorithm must be able to answer any query of the form:

Input: Integers \( a, b \), such that \( 0 \leq a, b \leq k \).
Question: How many of the \( n \) integers fall into a range \([a..b]\)?

Your algorithm is allowed to use \( O(n + k) \) preprocessing time and after this, each query must cost \( O(1) \) time.

Hint: The setup of the problem is similar to the one-dimensional interval summing problem covered in class. For the solution, you might use part of CountSort.

(a) Give your preprocessing procedure in pseudo-code and reason that it requires \( O(n + k) \) time.

Assume the \( n \) integers are in array \( A(1..n) \)

Initialize array \( C(0..k) \) to 0. \( O(k) \)

for \( j = 1 \) to \( n \)
\( C(A(j)) = C(A(j)) + 1 \) \( O(n) \)

\( C(-1) = 0 \)

for \( i = 0 \) to \( k \)
\( C(i) = C(i) + C(i - 1) \) \( O(k) \)

Total time \( O(n + k) \)

(b) Show how to process the range queries in \( O(1) \) time.

If \( a > b \) then return 0
else return \( C(b) - C(a - 1) \) \( O(1) \)
4. Consider the following sorting algorithm for sorting A(1..n):

\[ \text{Bubblesort}(A, n) \]
1. for \( i = 1 \) to \( n-1 \)
2. for \( j = n \) downto \( i+1 \)
3. if \( A(j) < A(j-1) \)
4. exchange \( A(j) \) with \( A(j-1) \)

You are to prove the correctness of the algorithm using a loop invariant. Start by seeing what the algorithm does on some small examples. For the sake of simplicity assume the keys are distinct.

(a) The outer loop varies \( i \). What is the invariant (call it \( P(i) \)) that holds at the beginning of this loop?

\[ P(i) : A \text{ contains permutation of the original array and } A(1..i-1) \text{ contains the } i-1 \text{ smallest elements of the original array in sorted order.} \]

(b) Complete the inductive argument by showing the Initialization, Maintenance and Termination part of the correctness proof.
 Initialization: For \( i = 1 \), \( A \) is unchanged and \( A(1..0) \) is empty and contains the 0 smallest elements.

Maintenance: At the beginning of the \( i \)-loop the \( i-1 \) smallest elements are in the sorted array \( A(1..i-1) \). Since the array contains a permutation of the original, the \( n-i+1 \) largest elements are in \( A(i..n) \). During the \( i \) loop the smallest element of \( A(i..n) \) (i.e. the \( i \)-th largest of the original array) is swapped to position \( i \). Thus at the end of the loop, the \( i \) smallest elements are in the sorted array \( A(1..i) \). Since only swaps were used, \( A \) still has a permutation of the original array.

Termination: At the beginning of loop \( i = n \), the loop terminates. \( P(n) \) guarantees that \( A(1..n-1) \) contains the \( n-1 \) smallest elements. Since \( A \) contains a permutation of the original array, \( A(n) \) contains the largest element. We conclude that the whole array is sorted.

(c) What is the running time of this algorithm in \( \Theta \) notation and why?

The inner loop is \( O(1) \) per iteration and the \( i \)th outer loop costs \( \Theta(n-i) \). Therefore the total time is

\[
\sum_{i=1}^{n} \Theta(n - i) = \sum_{j=1}^{n} \Theta(j) = \Theta(n^2).
\]
Note that this algorithm has the same time bound in $\Theta$ notation no matter whether you consider worst-case or average case time (the same set of comparisons are always used for each execution).
5. Show how to sort the following array of numbers using RadixSort:

(a) Give the status of the array after each of the three passes:

428
525
841
315
152
582
421
141

Solution:
428 841 315 141
525 421 421 152
841 141 525 315
315 152 428 421
152 582 841 428
582 525 141 525
421 315 152 582
141 428 582 841

(b) What is the running time of the algorithm if you are to sort $n$ $d$-digit numbers in the range from 0 to $k$.

$O(d(n + k))$
6. Show that $\sum_{i=1}^{n} \log_2(i + 2) = \Theta(n \log n)$.

We suggest to reason the upper and lower bound separately.

\[
\sum_{i=1}^{n} \log_2(i + 2) \leq \sum_{i=1}^{n} \log_2(n + 2) \\
\geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \log_2(\lfloor \frac{n}{2} \rfloor + 2) \\
= \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \log_2 \left( \frac{n}{2} \right) \\
\geq \frac{n}{2}((\log_2 n) - 1) \\
= \frac{n}{2} \log_2 n - \frac{n}{2} \\
= \frac{n}{4} \log_2 n + \frac{n}{4} \log_2 n - \frac{n}{2} \\
\geq \frac{n}{4} \log_2 n \\
= \Omega(n \log n)
\]
7. Show using the Substitution Method that \( T(n) = O(n^2) \), where

\[
T(n) = \begin{cases} 
1 & \text{for } n = 1 \\
4T(\lfloor \frac{n}{2} \rfloor) + n & \text{otherwise.}
\end{cases}
\]

What is your exact conjecture? Don’t forget the base case!
For the sake of simplicity you can assume \( n \) is a power of 2, i.e. \( \lfloor \frac{n}{2} \rfloor = \frac{n}{2} \).
Hint: First try to prove the conjecture \( T(n) \leq cn^2 \), and then try to refine the conjecture.

Conjecture 1: \( T(n) \leq cn^2 \)
Base case: \( T(1) = 1 \leq c \cdot 1^2 \)
True if \( c \geq 1 \)

Inductive assumption: Assume conjecture holds for \( 1 \leq n' < n \).
To show it holds for \( n \):

\[
T(n) = 4T(\lfloor \frac{n}{2} \rfloor) + n \\
\leq 4c(\frac{n}{2})^2 + n \\
= cn^2 + n \\
\]

\text{STUCK}

Conjecture 2: \( T(n) \leq cn^2 - bn \), for some \( b, c \geq 0 \) to be determined later
Base case: \( T(1) = 1 \leq c \cdot 1^2 - b \cdot 1 \)
True if \( c \geq b + 1 \)

Assume it holds for \( 1 \leq n' < n \). Then:

\[
T(n) = 4T(\frac{n}{2}) + n \\
\leq 4c(\frac{n}{2})^2 - 4bn + n \\
\overset{b \geq 1}{\leq} c n^2 - bn + n \\
\leq c n^2 - bn
\]

Good choices for the constants: \( b = 1, c = 2 \).
Much trickier with floors:

\[ T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n \]
\[ = 4c\left(\left\lfloor \frac{n}{2} \right\rfloor \right)^2 - 4b\left\lfloor \frac{n}{2} \right\rfloor + n \]
\[ \leq 4c \left( \frac{n}{2} \right)^2 - 4b\left( \frac{n}{2} - 1 \right) + n \]
\[ = cn^2 - 2bn + 4b + n \]
\[ \leq cn^2 - bn - bn + 4b + n \]
\[ \leq cn^2 - bn - 2n + 4b + n \]
\[ = cn^2 - bn - n + 4b \]
\[ \leq cn^2 - bn. \]

**Good choices for the constants:** \( b = 2, \ c = 3, \ n_0 = 4. \)

8. You are to show how BuildHeap works on the following 10 keys:

\[ 3 \ 8 \ 9 \ -1 \ 5 \ 6 \ 10 \ 1 \ 0 \ 9 \]

Draw the following binary trees
(a) for the initial array

\[
\begin{array}{cccc}
3 & 8 & 9 \\
-2 & 5 & 6 & 10 \\
1 & 0 & 9 \\
\end{array}
\]

(b) after nodes 5 and 4 have been processed

\[
\begin{array}{cccc}
\text{Build-Max-Heap} & \text{Build-Min-Heap} \\
3 & 3 \\
8 & 9 & 8 & 9 \\
1 & 9 & 6 & 10 & -2 & 5 & 6 & 10 \\
-2 & 0 & 5 & 1 & 0 & 9 \\
\end{array}
\]

(c) after 3 and 2 have been processed

\[
\begin{array}{cccc}
3 & 3 \\
9 & 10 & -2 & 6 \\
1 & 8 & 6 & 9 & 0 & 5 & 9 & 10 \\
-2 & 0 & 5 & 1 & 8 & 9 \\
\end{array}
\]

(d) the final heap

\[
\begin{array}{cccc}
10 & -2 \\
9 & 9 & 0 & 6 \\
1 & 8 & 6 & 3 & 1 & 5 & 9 & 10 \\
-2 & 0 & 5 & 3 & 8 & 9 \\
\end{array}
\]

(e) How expensive is the BuildHeap operation as a function of \( n \)?

\( O(n) \)

(f) How expensive is BuildHeap if you inserted the \( n \) keyes one at a time?

\[
\sum_{i=1}^{n} O(\log i) = O(n \log n)
\]