King Fahd University of Petroleum and Minerals
College of Computer Science and Engineering
Information and Computer Science Department

ICS 353: Design and Analysis of Algorithms

First semester 2016-2017
Major Exam #1, Sunday, October 23, 2016.

Name:

ID#:

Instructions:

1. The exam consists of 7 pages, including this page, containing 5 questions. You have to answer all 5 questions.
2. The exam is closed book and closed notes. No calculators or any helping aides are allowed. Make sure you turn off your mobile phone and keep it in your pocket if you have one.
3. The maximum number of points for this exam is **100**.
4. You have exactly **90** minutes to finish the exam.
5. Make sure your answers are **readable**.
6. If there is no space on the front of the page, feel free to use the back of the page. Make sure you indicate this in order for me not to miss grading it.

Question Number	Maximum # of Points	Earned Points
1	20	
2	20	
3	20	
4	20	
5	20	
Total	**100**	

***. Some Useful Formulas:**

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} \qquad \sum_{i=1}^{n} i.c^i = \Theta(1) \text{ for } 0 < c < 1 \qquad 2^{\lg n} = n$$

,

$$log_b a = \frac{log_c a}{log_c b} \text{ where } c, b \neq 1 \qquad \log a^b = b \log a \qquad \log ab = \log a + \log b$$

Q1. (20 points)

a. (8 points) Using the definition of $O()$ notation, prove that $4n^3 + 8n^2 + 100n + 10$ is in $O(n^3)$.

To find $c > 0$ and $n_0 \in \mathbb{N}$ such that $4n^3 + 8n^2 + 100n + 10 \leq cn^3 \quad \forall n \geq n_0$.
Since
$4n^3 + 8n^2 + 100n + 10 \leq 4n^3 + 10n^3 + 100n^3 + 10n^3 = 122n^3 \; \forall n \geq 1$
with $c = 122, n_0 = 1$, we can conclude that $4n^3 + 8n^2 + 100n + 10 = O(n^3)$.

b. (12 points) Express the following function

$$f(n) = n \log n^5 + \log^2 \sqrt{n}$$

in terms of the Θ-notation in the simplest form. Make sure you prove your answer.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n \log n^5}{\log^2 \sqrt{n}} &= \lim_{n\to\infty} \frac{5n \log n}{(\log \sqrt{n})(\log \sqrt{n})} \\
&= \lim_{n\to\infty} \frac{5n \log n}{\left(\frac{1}{2}\log n\right)\left(\frac{1}{2}\log n\right)} \\
&= \lim_{n\to\infty} \frac{5n \log n}{\frac{1}{4}\log^2 n} \\
&= 20 \lim_{n\to\infty} \frac{n \log n}{\log^2 n} \\
&= c_1 \lim_{n\to\infty} \frac{n}{\log n} && c_1 = 20 \\
&= c_1 \lim_{n\to\infty} \frac{n}{\frac{\ln n}{\ln 2}} \\
&= c_2 \lim_{n\to\infty} \frac{n}{\ln n} && c_2 = 20 \ln 2 \\
&= c_2 \lim_{n\to\infty} \frac{1}{\frac{1}{n}} && \text{using L'Hospital's Rule} \\
&= c_2 \lim_{n\to\infty} n = \infty
\end{aligned}$$

Therefore, the numerator ≫ the denominator, and hence $f(n) = \Theta(n \log n)$.

Q2. **(20 points)**

a. (15 points) For the following code segments:

```
Algorithm SomeCount (int n) {
count = 0;
for (i=1; i<=n; i=i*2)
   for (j=i; j>=1; j--)
      count++;
}
```

i. (6 points) Formulate the cost of running the above code in summation form (note that it will be equal to the value of `count`). You may assume that *n* is a power of 2.

ii. (3 points) Evaluate the summation in part "i".

iii. (2 points) Express the cost of this code segment in terms of Θ() notation.

iv. (4 points) Is this algorithm linear, with respect to input size? Clearly justify your answer.

i. The values of i: $1, 2, 2^2, 2^3, \ldots, 2^{\log n}$.
Consider $r = \log i$. Since j ranges from i down to 1, the value of *count* can be formulated as

$$count = \sum_{r=0}^{\log n} \sum_{j=1}^{i} 1$$

ii.

$$count = \sum_{r=0}^{\log n} \sum_{j=1}^{i} 1 = \sum_{r=0}^{\log n} i = \sum_{r=0}^{\log n} 2^r = \frac{2^{\log n+1} - 1}{2 - 1} = 2n - 1$$

iii. The cost is $\Theta(n)$.

iv. No. Since we only have one input, n, the value of *count* depends on the value of n. Hence, the algorithm is exponential with respect to the input size.

b. (5 points) Consider following recursive Algorithm:

```
Algorithm myRecursive(int n, char from, char to, char temp){
  if (n == 1)
     print(from + " --------> " + to);
  else {
     myRecursive(n - 1, from, temp, to);
     print(from + " --------> " + to);
     myRecursive(n - 1, temp, to, from);
  }
}
```

Let $T(n)$ denote the number of times the print statement is executed.

i. (1 points) What is the value of $T(1)$?

$$T(1) = 1$$

ii. (4 points) Derive the recurrence equation describing the value of $T(n)$ for $n > 1$.

$$T(n) = 2T(n-1) + 1$$

Q3. **(20 points)** Consider a max-heap H of size $n > 2$ of distinct elements.

a. (7 points) What is the maximum number of element comparisons that are needed to find the second largest element in H? Clearly justify your answer.

One comparison. Find the maximum of the children of the root. This will be the second largest element.

b. (13 points) What is the minimum number of element comparisons that are needed to find the minimum value in the max-heap? Clearly justify your answer by outlining the algorithm and analyzing its cost.

The minimum will be in a leaf. Therefore, the algorithm should linearly scan all the leafs and find the minimum. Since the index of the last internal node equals $\left\lfloor \frac{n}{2} \right\rfloor$, leaves start at index $\left\lfloor \frac{n}{2} \right\rfloor + 1$ and end at index n. Hence, the number of element comparisons $= n - \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) = n - \left\lfloor \frac{n}{2} \right\rfloor - 1$, which is obviously $\Theta(n)$.

Q4. **(20 points)** Consider the following array:

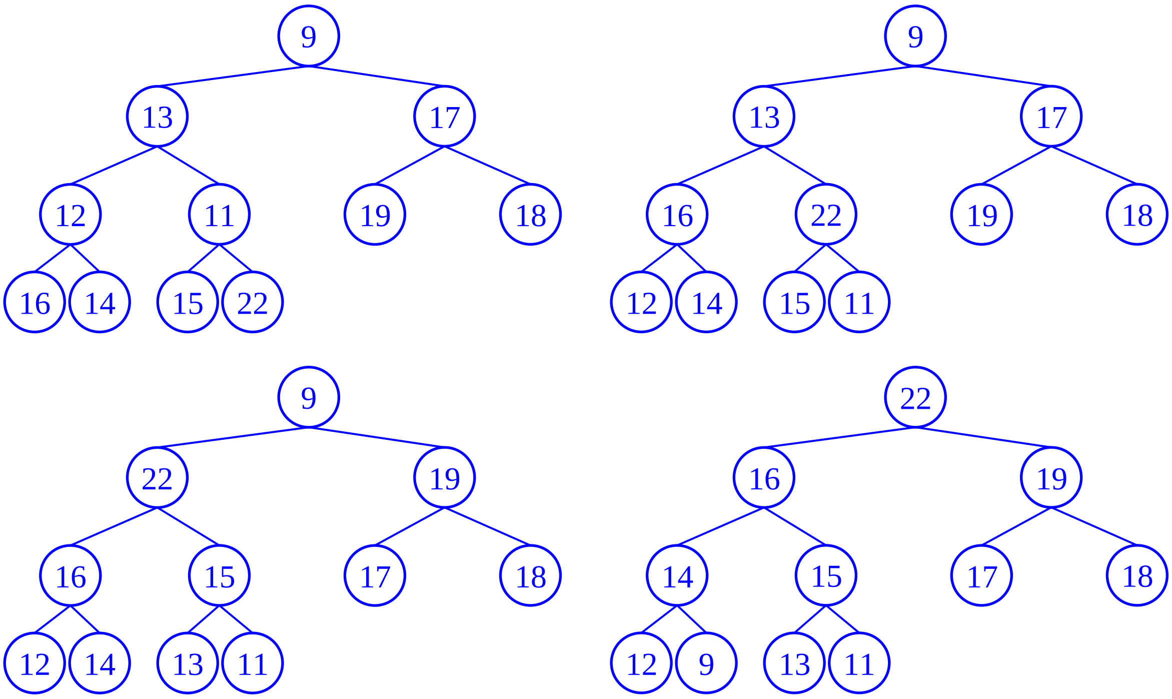
9 , 13, 17, 12, 11, 19, 18, 16, 14, 15, 22

a. (12 points) Illustrate the operation of Algorithm MAKEHEAP on the above array to build a max-heap. (Show the intermediate steps).

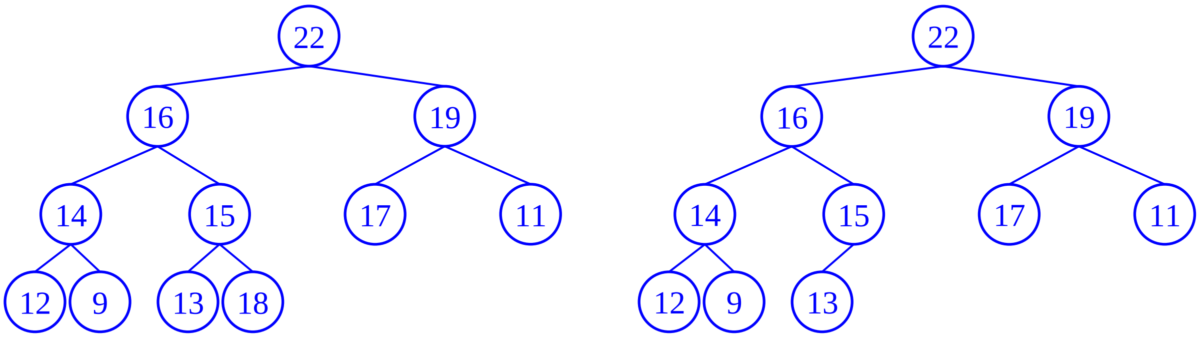
b. (4 points) Show the max-heap after deleting the element with key value 18 from the heap H in part "a.". (Show the intermediate steps).

c. (4 points) Show the max-heap after inserting an element of value 21 into the heap H in part "a.". (Show the intermediate steps).

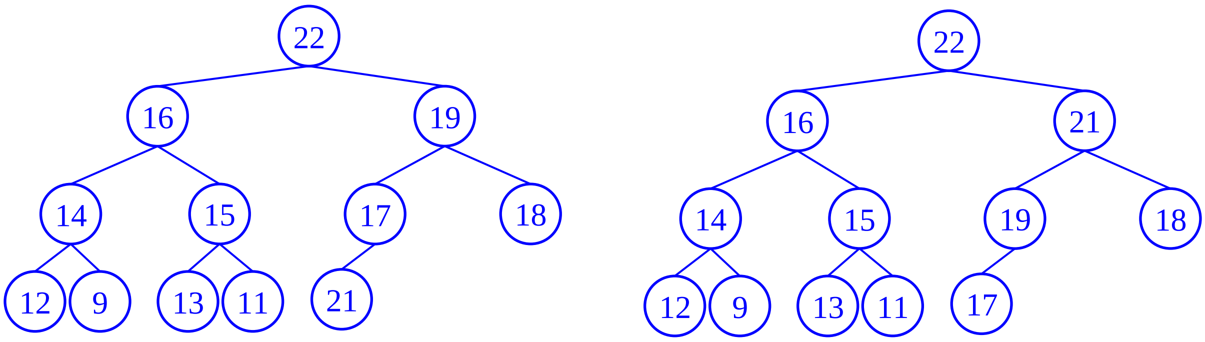
a.



b.



c.



Q5. **(20 points)** Consider the linear search problem. Assume that a key *x* is to be found in an array *A* of size *n*, where the probability that the key is found in the first position is $\frac{1}{2}$, in the second position is $\frac{1}{3}$, in the third position is $\frac{1}{9}$ and it is equally likely to find the key in the rest of the positions of the array. Assume that the key *x* will always exist in the array.

a. (3 points) Analyze the best case time complexity of the above algorithm, finding it in terms of $\Theta()$ notation.
b. (3 points) Analyze the worst case time complexity of the above algorithm, finding it in terms of $\Theta()$ notation.
c. (3 points) Analyze the worst case space complexity of the above algorithm, finding it in terms of $\Theta()$ notation.
d. (11 points) Analyze the average case time complexity of the above algorithm, finding it in terms of $\Theta()$ notation.

a. The best case of linear search occurs when the key is at the first position, requiring one element comparison, for a cost of $\Theta(1)$.
b. The worst case of linear search occurs when the key is at the last position, requiring n element comparison, for a cost of $\Theta(n)$.
c. The worst case space complexity of linear search is the same as the best case, which only requires an iterator over the array and an integer to hold the position, independent from the input size. Hence, it is $\Theta(1)$.
d. Probability of the key being in any position other than the first, second or third is equal to $\frac{\left(1-\frac{1}{2}-\frac{1}{3}-\frac{1}{9}\right)}{n-3} = \frac{1}{18(n-3)}$. Hence, the average case time complexity is equal to

$$\sum_{i=4}^{n} \frac{1}{18(n-3)} \times i = \frac{1}{18(n-3)} \sum_{i=4}^{n} i$$

$$= \frac{1}{18(n-3)} \times \left(\sum_{i=1}^{n} i - \sum_{i=1}^{3} i \right)$$

$$= \frac{1}{18(n-3)} \times \left(\frac{n(n+1)}{2} - \frac{3 \times 4}{2} \right)$$

$$= \frac{1}{18(n-3)} \left(\frac{n(n+1)}{2} - 6 \right)$$

$$= \Theta(n)$$