Note 1. For the remainder of the course "Give an algorithm" means: describe an algorithm, show that it works as claimed, analyze its worst-case running time and give the expression for its worst-case running time.

Note 2. For most of the problems summary solutions are provided.

Note 3. Several of the first few problems refer to concepts from undergraduate classes that you should have picked (e.g. the prerequisites) or from CS 506 that is also offered at NJIT as a bridge course. Handouts 3 and 4 supplement this material.

Problem 1.
(a) How many bits do we need to minimally represent $2^m - 1$? How many for $2^m - 1$? How many for all integers between $2^m - 1$ and $2^m - 1$? Express the answer as a function of $m$.
(b) Use the result from (a) to show that every positive integer $L$ can be represented by $\lceil \lg (L + 1) \rceil$ bits.

Problem 2.
You are given the following function definition in pseudo C++/C. How much is $f(91)$? How much is $f(92)$? How much is $f(99)$? How much is $f(100)$? Explain.

```c
int f(int x) // x is integer
return( (x>100) ? (x-10) : f(f(x+11)));
```

Problem 3.
Consider a modification of merge-sort in which out of $n$ keys $n/k$ subsequences of length $k$ are formed and each one is sorted by insertion sort and then all subsequences are merged. The optimal $k$ will be determined after analyzing the worst case performance of the algorithm. In other words, this new algorithm works as follows.
(i) Form $n/k$ subsequences each of length $k$. Sort each one with insertion sort. What is the total (worst-case) running time for sorting all $k$ subsequences using insertion sort in asymptotic notation? Explain.
(ii) The $k$-long sorted subsequences can then be merged in $\Theta(n \lg (n/k))$ time. Show how.
(iii) Show that the (worst-case) running time of the whole algorithm (steps (i) and (ii)) is $O(nk + n \lg (n/k))$.
(iv) What is the (asymptotically) largest value of $k$ for which this algorithm is as good as merge-sort? Explain.

Problem 4.
Can you sort 4 keys such as $a,b,c,d$ with at most 5 comparisons in the worst case? If yes, show your algorithm. Assume all keys are distinct, if it helps.

Problem 5.
Use mathematical induction to show that when $n$ is an exact power of two, the solution of the recurrence

$$T(n) = \begin{cases} 8 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n + 3n$.

Problem 6.
(a) Show that $(n - 3)^2 = \Theta(n^2)$ by providing the $n_0, c_1, c_2$ in the definition of the $\Theta$ notation.
(b) Rank the following function by order of growth: that is, find an arrangement $g_1, g_2, g_3, \ldots, g_k$ of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), g_3 = \Omega(g_4), \ldots, g_7 = \Omega(g_8)$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$. (8 points)

$$n, \ (n - 1)!, \ \lfloor \lg n \rfloor, \ lg \ n, \ lg (n!), \ 2^{\lg n}, \ n^{\lg n}, \ n!.$$
Problem 8.
Solve exactly using the iteration method the following recurrence. You may assume that \( n \) is a power of 4, i.e. \( n = 4^k \).

\[ T(n) = 4T(n/4) + n^2, \text{ where } T(1) = 4. \]

Problem 9.
Solve the following recurrences; make your bounds as tight as possible. Use strong induction if necessary.

\[ T(n) = T(n/2) + \log n, \text{ where } T(2) = 1. \]
\[ T(n) = T(n/2) + T(n/3) + 32n, \text{ where } T(1) = 100. \]

Problem 10.
(a) What is the minimum and maximum number of elements in a heap of height \( h \)?
(b) Show that an \( n \) element heap has height \( \lfloor \log n \rfloor \).
(c) Show that in any subtree of a MAX-HEAP, the root of the subtree contains the largest value occurring anywhere in that subtree.

Problem 11.
Min-Max. Give a divide-and-conquer algorithm that finds both the MINIMUM and the MAXIMUM of \( n \) keys \( x_1, x_2, \ldots, x_n \) by performing at most \( 3n/2 - 1 \) comparisons. You may assume that \( n \) is a power of two.

Problem 12.
\( d \)-HEAPS. Analysis of \( d \)-ary heaps. (Assume MAXheaps in the remainder).
A \( d \)-ary heap is like a binary heap but non-leaf nodes have \( d \) children rather than 2.
(a) How would you represent a \( d \)-ary heap in an array?
(b) What is the height of a \( d \)-ary heap of \( n \) elements in terms of \( n \) and \( d \)?
(c) Give an efficient algorithm that extracts the maximum element from a \( d \)-ary max-heap. Analyze its running time in terms of \( n \) and \( d \).
(d) Implement \( \text{Insert} \) for \( d \)-ary heaps and analyze its running time.
(e) Given an efficient implementation of an operation \( \text{Increase}(A, i, \text{value}) \) which first sets \( A[i] = \max\{A[i], \text{value}\} \) and then properly updates the \( d \)-ary max-heap. Analyze the worst-case running time of \( \text{Increase} \).

Problem 13.
The following two methods are proposed to find \( M_k \), the \( k \)-th smallest of \( n \) keys stored in array \( A[0..n-1] \).
(a) Sort \( A \) and then output the \( k \)-th element of the sorted \( A \). This is \( M_k \).
(b) Build a MIN-heap out of the \( n \) elements of \( A \), and then perform \( k \) \( \text{EXTRACT-MIN} \) operations. The last (i.e. \( k \)-th) \( \text{EXTRACT-MIN} \) operation returns \( M_k \).
Which of the two algorithms is asymptotically faster? Explain, i.e. provide details of efficient implementation of the various steps, and analyze their worst-case running time using asymptotic notation. Use case analysis, if necessary.

Problem 14.
You are given \( n \) arbitrary keys. We would like to identify which key appears most often (i.e. the highest-frequency key). Give an efficient algorithm that finds the highest-frequency key of an input of \( n \) keys. If two or more keys have the highest-frequency, then you are allowed to output any single one of these keys attaining the highest-frequency.
(Example. If \( \langle 10, 2, 4, 7, 4, 2, 10, 2 \rangle \) is an input key sequence, then the answer will be key 2. If \( \langle \text{Tom}, \text{Jerry}, \text{Jerry}, \text{Tom} \rangle \) is the input, your answer can be \text{Tom}; an equally correct answer is \text{Jerry}).

Problem 15.
You are given \( n \) keys each one of which is an A,B,C, or D (say A means freshperson, B sophomore and so on). Sort the keys in worst-case linear time \( \Theta(n) \) so that in the output sequence the As are before (on the left of) the Bs, which are before the Cs, which are before the Ds.

Problem 16.
Solution Outline

Problem 1.
(a) (b) $2^{m-1}$ is an 1 followed by $m-1$ zeroes, i.e. we need $m$ bits. $2^{m-1}$ is a sequence of m 1's. So both integers need $m$ bits to be represented. If the extremes $A = 2^{m-1}$ and $B = 2^m - 1$ need $m$ bits so do all integers $L$ between $A$ and $B$ such that $A \leq L \leq B$.

(b) Every positive integer $L$ can be sandwiched between two $A, B$ as defined in part (a), for some integer value $m$. Thus

$$2^{m-1} \leq L \leq 2^m - 1$$
$$2^{m-1} + 1 \leq L + 1 \leq 2^m$$
$$2^{m-1} < L + 1 \leq 2^m$$
$$\lg (2^{m-1}) < \lg (L + 1) \leq \lg (2^m)$$
$$m - 1 < \lg (L + 1) \leq m$$

The term $\lg (L + 1)$ is strictly greater than $m - 1$ and at most $m$. Its ceiling $\lceil \lg (L + 1) \rceil = m$, which is an integer can not thus be $m - 1$, because $\lg (L + 1)$ is greater than $m - 1$, and therefore $\lceil \lg (L + 1) \rceil = m$ must be the next greater integer, i.e. $m$. However, by part (a), $m = \lceil \lg (L + 1) \rceil$ is the number of bits in $L$. This completes the proof of the problem, i.e. number of bits of $L$ is $m = \lceil \lg L \rceil + 1$. Therefore $m = \lceil \lg L \rceil + 1 = \lceil \lg (L + 1) \rceil$.

Problem 2.
We start establishing $f(91)$ by using the two conditions $A$: $(x - 10)$ if $x > 100$ and $B$: $f(x) = f(f(x + 11))$ otherwise.

$$f(91) = \text{Condition } B \quad f(f(91 + 11))$$
$$= f(f(102))$$
$$= \text{Condition } A \quad f(102 - 10)$$
$$= f(92)$$

We have just proved that $f(91) = f(92)$. We next find $f(92)$.

$$f(92) = \text{Condition } B \quad f(f(92 + 11))$$
$$= f(f(103))$$
$$= \text{Condition } A \quad f(103 - 10)$$
$$= f(93)$$

We have just proved that $f(91) = f(92) = f(93)$. We next find that $f(91) = f(92) = f(93) = f(94)$, and proceeding similarly that $f(91) = f(92) = \ldots = f(100)$. We thus find now $f(100)$.

$$f(100) = \text{Condition } B \quad f(f(100 + 11))$$
$$= f(f(111))$$
$$= \text{Condition } A \quad f(111 - 10)$$
$$= f(101)$$
$$= \text{Condition } A \quad 101 - 10$$
$$= 91$$

Therefore $f(91) = f(92) = \ldots = f(99) = f(100) = 91$.

Problem 3.
i. Each sequence of length $k$ can be sorted by insertion sort in $O(k^2)$ worst case time. We have $n/k$ sequences, each one requires $O(k^2)$ time, therefore total time for all sequences is

$$(n/k) \cdot O(k^2) = O(nk).$$
depending on whether $d_4$. If $y$ need to determine whether $c$, we binary search/insert $c$.

Comparison 1. We first compare $a, b$ using our first comparison. The output (left-to-right smallest-to-largest) is either $a, b$ or $b, a$ depending on this outcome. Let us call this output generically $x_1, x_2$.

Comparisons 2.3. Now that we have settled $a$ versus $b$, we deal with $c$. For $c$ we binary search/insert $c$ in the sorted sequence of $a, b$ denoted $x_1, x_2$. That is, we first compare $c$ to $x_2$ and then if necessary $c$ to $x_1$. This requires one or two comparisons, i.e. two in the worst-case. Why? If $c$ is greater than $x_1$ we are done: the output is $x_1, x_2, c$. Otherwise we need to determine whether $c$ goes to left or right of $x_1$ by utilizing comparison 3. Let us again call the three sorted keys $y_1, y_2, y_3$, a permutation of $a, b, c$.

Comparison 4.5. In the final step we binary search/insert $d$ into the sorted sequence $y_1, y_2, y_3$. We can use again 2 comparisons, as many as previously, even if the keys are three and not two. How? We first compare $d$ to $y_2$ (comparison 4). If $d$ is greater than $y_2$, we compare $d$ to $y_3$ (comparison 5) and thus $d$ is inserted either on the left of $y_3$ or on the right depending on whether $d$ is smaller or larger than $y_3$. If however the result of comparison 4 says $d$ is less than $y_2$, then in comparison 5 we compare $d$ to $y_1$ and decide whether to insert on the left or right of $y_1$.

In 5 comparisons we managed to sort 4 keys in the worst-case. This is the best we can. We will prove later in class that no matter what we do and what algorithm we invent, sorting 4 keys in the worst-case requires at least 5 comparisons if the only operation allowed is comparison/swapping of keys. How do 4 (number of keys) and 5 (number of comparisons) compare? Well 5 is the smallest integer greater than $\lg (4!)$. 

Problem 5.

We show the claim by induction. Our proposition $P(n)$ is: $T(n) = 3n + n \lg n$.

Since we only need to show $P(n)$ not for all $n$ but only for $n$ being a power of two, we can transform $P(n)$ into $Q(k) = P(2^k)$ using $n = 2^k$.

$Q(k)$ is thus given by the following expression.

$$Q(k) : T(2^k) = 3 \cdot 2^k + k2^k = (k + 3)2^k.$$ 

The recurrence $T(n) = 2T(n/2) + n$ can also be written in terms of $k$ as

$$T(2^k) = 2T(2^{k-1}) + 2^k.$$ 

We show $Q(k)$ by induction on $k$. 

---

### Table

<table>
<thead>
<tr>
<th>Level</th>
<th>SizePerSeq</th>
<th>#ofSeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n/2**0</td>
<td>* 2**0 = n</td>
</tr>
<tr>
<td>1</td>
<td>n/2**1</td>
<td>* 2**1 = n</td>
</tr>
<tr>
<td>2</td>
<td>n/2**2</td>
<td>* 2**2 = n</td>
</tr>
<tr>
<td>3</td>
<td>n/2**3</td>
<td>* 2**3 = n</td>
</tr>
</tbody>
</table>

**Figure 1.**

```
(n) (n/2) (n/4) (n/8) (n/16) (n/32) (n/64) (n/128) (n/256) (n/512) (n/1024) (n/2048) (n/4096) (n/8192) (n/16384) (n/32768) (n/65536) (n/131072) (n/262144) (n/524288) (n/1048576) (n/2097152) (n/4194304) (n/8388608) (n/16777216)
```

...... intermediate levels
Base case. For \( k = 1 \), we have \( n = 2^1 = 2 \). We substitute 1 for \( k \) in \( T(2^k) \). Thus \( T(2^k) = T(2) = 8 \), which is 8 by the base case of the recurrence that was given to us.

Starting from the right side of \( T(2^k) \) which is \((k + 3)2^k\) for \( k = 1 \) we get \((1 + 3)2^1 = 8\) as well. Thus for \( k = 1 \) \( T(2^k) \) is indeed equal to \((k + 3)2^k\).

Therefore we have just shown that \( Q(0) \) is true because its left side \( T(2^k) \) gives 8 by the base case and its right side \((k + 3)2^k\) also gives 8 for \( k = 1 \) and thus \( T(2^k) = (k + 3)2^k \) for the base case \( k = 1 \).

Inductive Hypothesis. We show that if \( Q(k - 1) \) is true, then \( Q(k) \) will also be true.

Inductive Step. If \( Q(k - 1) \) is true, then \( T(2^{k-1}) = (k + 2)2^{k-1} \). We expect to establish that \( Q(k) \) is true, i.e. \( T(2^k) = (k + 3) \cdot 2^k \), given that \( Q(k - 1) \) is true.

We use the recurrence \( T(2^k) = 2T(2^{k-1}) + 2^k \) to show that \( Q(k) \) is true then. We start from the recurrence above and we substitute there for \( T(2^{k-1}) \) the expression obtained by the inductive hypothesis for \( Q(k - 1) \).

\[
T(n) = T(2^k) = 2T(2^{k-1}) + 2^k
= 2((k + 2)2^{k-1}) + 2^k
= k2^k + 3 \cdot 2^k = (k + 3)2^k
\]

We have thus proved that \( T(2^k) = (k + 3)2^k \). We express the answer as a function of \( n \) now. Since \( n = 2^k \) we also have that \( k = \lg n \), and thus

\[
T(2^k) = (k + 3) \cdot 2^k
T(n) = (\lg n + 3) \cdot n
T(n) = n \lg n + 3n
\]

We have thus proven that \( T(n) = n \lg n + 3n \) for all \( n = 2^k \), where \( k \geq 1 \).

Problem 6.

Part a. We show first that \((n - 3)^2 = O(n^2)\) and then that \((n - 3)^2 = \Omega(n^2)\).

Case 1. Show that \((n - 3)^2 = O(n^2)\)

For all \( n \geq 1 \) we have that

\[
(n - 3)^2 = n^2 - 6n + 9
\leq n^2 + 0 \cdot n^2 + 9
\leq n^2 + 0 \cdot n^2 + 9 \cdot n^2
\leq 10n^2
\]

Therefore there exist constant \( n_2 = 1 \) and \( c_2 = 10 \) such that \((n - 3)^2 \leq c_2 n^2 \) for all \( n \geq n_2 \). This proves the claim.

Technique 1. What we use in this proof is the fact that \( n^2 \pm An \pm B \) is bounded above, for positive \( A, B \), by \( n^2 \pm An \pm B \leq (1 + A + B)n^2 \).

Case 2. Show that \((n - 3)^2 = \Omega(n^2)\)

Technique 1 can not be used in this case. However, since \( 4 \geq 0 \) we can eliminate 4 easily.

\[
(n - 3)^2 = n^2 - 6n + 9
\geq n^2 - 6n
\]

The next step is non-trivial. We bound \( n^2 - 6n \) from below by \( n^2/2 \). This is so as long as

\[
n^2 - 6n \geq n^2/2 \iff 
n^2/2 \geq 6n \iff 
n/2 \geq 6 \iff 
n \geq 12
\]

We can do this as long as \( n \) is not zero; this is true since for all cases we assume that at least \( n \geq 1 \). Therefore there exist constant \( n_1 = 12 \) and \( c_1 = 1/2 \) such that \((n - 3)^2 \geq n^2/2 \) for all \( n \geq n_1 = 12 \). This proves the claim.
In order to show that \((n-3)^2 = \Theta(n^2)\) we need to establish \(c_1, c_2\) and \(n_0\). \(c_1\) and \(c_2\) are 1/2 and 10 respectively. \(n_0 = \max(n_1, n_2) = 12\). For these values the problem is thus shown.

### Part b. Some preliminaries

1. Case 1 of the Master method applies and thus equivalence classes with commas, and eliminating set notation for single-element sets/equivalence-classes.
2. \(n \lg n = \omega(n)\).
3. \(n \lg n = n^2 \cdot \lg \frac{n}{\epsilon} = \Theta(n)\).
4. Since \(2^\lg n = n\) and \(2^\lg 2 = 2\), we have \(n^\lg n = (2^\lg n)^\lg n = 2^\lg n^\lg n = (\lg n)^\lg n\).
5. \(n - 1)! = \Theta(n^\lg n)\).
6. \(n! = n \cdot (n - 1)!, therefore n! = \omega((n - 1)!))\).

A ranking of the functions would be (leftmost is asymptotically smallest, rightmost is asymptotically largest), separating equivalence classes with commas, and eliminating set notation for single-element sets/equivalence-classes.

\[\{n, 2^\lg n\}, \{n \lg n, \lg(n!}\}, \{n^\lg n, (\lg n)^\lg n\}, (n - 1)!, n!\].

### Problem 7.

One can use the master method for all three of them.

a. \(T(n) = 4T(n/2) + 3n^2\). It has \(a = 4 \geq 1, b = 2 > 1\), \(f(n) = 3n^2\). \(n^{\log_b a} = n^4 = n^2\). Obviously \(n^2 = f(n) = \Omega(n^{\log_b a})\). Case 2 of the Master method applies and thus \(T(n) = \Theta(n^2 \lg n)\).

b. \(T(n) = 32T(n/4) + 3n^2\). Note that \(32 \cdot 32 = 1024 = 2^{10}, 4^5 = 1024 i.e. 4^{5/2} = 1024^{1/2} = 32\). It has \(a = 32 \geq 1, b = 4 > 1\), \(f(n) = 3n^2\). \(n^{\log_b a} = n^5, 32 = n^{5/2}\). Obviously \(n^2 = f(n) = O(n^{\log_b a - \epsilon}) = O(n^2)\), by choosing \(\epsilon = 1/2 > 0\). Case 1 of the Master method applies and thus \(T(n) = \Theta(n^{5/2})\).

c. \(T(n) = 4T(n/2) + n^3\). It has \(a = 4 \geq 1, b = 2 > 1, f(n) = n^3\). \(n^{\log_b a} = n^4 = n^2\). Obviously \(n^3 = f(n) = \Omega(n^{\log_b a + \epsilon}) = O(n^3)\), by choosing \(\epsilon = 1 > 0\). Case 3 of the Master method applies since also \(af(n/b) = 4(n/2)^3 = n^3/2 \leq cn^3 = (1/2)n^3\), by choosing \(c = 1/2 < 1\), and thus \(T(n) = \Theta(n^3)\).

### Problem 8.

\(T(n) = 4T(n/4) + n^2\). We use the iteration method. This first example will be done in fine detail showing all calculations.

\[
T(n) = 4T(n/4) + n^2
\]

\[
= 4 \left( 4T \left( \frac{n}{4^2} \right) + \left( \frac{n}{4} \right)^2 \right) + n^2
\]

\[
= 4^2 T \left( \frac{n}{4^2} \right) + n^2 \frac{n}{4^2} + n^2 \frac{1}{4^2}
\]

\[
= 4^2 \left( 4T \left( \frac{n}{4^3} \right) + \left( \frac{n}{4^3} \right)^2 \right) + n^2 \frac{n}{4^3} + n^2 \frac{1}{4^3}
\]

\[
= 4^3 T \left( \frac{n}{4^3} \right) + n^2 \frac{n}{4^2} + n^2 \frac{1}{4^2}
\]

\[
= \ldots
\]

\[
= 4^i T \left( \frac{n}{4^i} \right) + n^2 \frac{n}{4^{i-1}} + \ldots + n^2 \frac{n}{4^1} + n^2 \frac{1}{4^0}
\]

\[
= 4^i T \left( \frac{n}{4^i} \right) + n^2 \left( \frac{1}{4^{i-1}} + \ldots + \frac{1}{4^1} + \frac{1}{4^0} \right)
\]

\[
= 4^i T \left( \frac{n}{4^i} \right) + n^2 \left( \frac{1}{4^{i-1}} - \frac{1}{4^i} \right)
\]

\[
= 4^i T \left( \frac{n}{4^i} \right) + n^2 \left( \frac{1 - \frac{1}{4^i}}{1 - \frac{1}{4}} \right)
\]

\[
= 4^i T \left( \frac{n}{4^i} \right) + \frac{4n^2}{3} \left( 1 - \frac{1}{4^i} \right)
\]
The base case is \( T(1) = 4 \). We set \( n/4^i = 1 \), i.e. \( n = 4^i \). If we solve this for \( i \), we get \( i = \log_4 n = \lg n/2 \). Then \( T(n/4^i) = T(1) = 4 \). In addition \( 4^i = n \). Therefore the formula for \( T(n) \) becomes

\[
T(n) = 4^i T \left( \frac{n}{4^i} \right) + \frac{4n^2}{3} \left( 1 - \frac{1}{4^i} \right)
\]

\[
= nT(1) + \frac{4n^2}{3}(1 - 1/n)
\]

\[
= 4n + \frac{4n^2}{3} (1 - 1/n)
\]

\[
= \frac{4n^2}{3} + \frac{8n}{3}
\]

**Note.** It is easy to verify for example that \( T(1) = \frac{4 \cdot 1^2}{3} + \frac{8 \cdot 1}{3} = 4 \). This can serve as a checking mechanism that the solution that you have established satisfies at least the base case.

**Problem 9.**

a. \( T(n) = T(n/2) + \lg n \). We use the iteration method, noting that \( T(2) = 1 \).

\[
T(n) = T \left( \frac{n}{2} \right) + \lg n
\]

\[
= \left( T \left( \frac{n}{2^2} \right) + (\lg \frac{n}{2}) \right) + \lg n
\]

\[
= T \left( \frac{n}{2^2} \right) + \lg \frac{n}{2} + \lg \frac{n}{2^2}
\]

\[
= \ldots
\]

\[
= T \left( \frac{n}{2^i} \right) + \lg \frac{n}{2^{i-1}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0}
\]

The base case is \( T(2) = 1 \). We set \( n/2^i = 2 \), i.e. \( n = 2^{i+1} \). If we solve this for \( i \), we get \( i = \lg n - 1 \). Then \( T(n/2^i) = T(2) = 1 \). In addition \( 2^i = n/2 \). Therefore the formula for \( T(n) \) becomes

\[
T(n) = T \left( \frac{n}{2^j} \right) + \lg \frac{n}{2^{j-1}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0}
\]

\[
= T \left( \frac{n}{2^{\lg n-1}} \right) + \lg \frac{n}{2^{\lg n-2}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0}
\]

\[
= T(2) + (\lg n - (\lg 2^{\lg n-1})) + (\lg n - (\lg 2^{\lg n-2})) + \ldots + (\lg n - \lg 2^1) + (\lg n - \lg 2^0)
\]

\[
= 1 + \lg n - (\lg n - 2) + \ldots + (\lg n - (\lg n - 3)) + \ldots + (\lg n - 1) + (\lg n)
\]

\[
= \sum_{j=1}^{\lg n} j
\]

\[
= \lg n (\lg n + 1)/2
\]

Therefore, \( T(n) = \lg n(\lg n + 1)/2 \). It is easy to verify that \( T(2) \) is indeed 1.

b. Since \( T(n) = T(n/2) + T(n/3) + 32n \) we have for \( n = 1 \) from the base case that \( T(1) = 100 \). We show that \( T(n) = \Omega(n) \). By the recurrence we have that \( T(n) \geq 32n \) for all \( n > 1 \) since \( T(n/2), T(n/3) \) are non-negative and the other term is \( 32n \). In addition, \( T(1) = 100 \geq 32 \cdot 1 = 32 \). Therefore \( T(n) \geq 32n \) for \( n = 1 \) as well. Therefore \( T(n) = \Omega(n) \).

We now prove that \( T(n) = O(n) \) or equivalently that there exist positive constants \( n_2, c_2 \) such that \( T(n) \leq c_2n \) for all \( n \geq n_2 \). Proof is by induction on \( n \).

**Base Case.** \( T(1) = 100 \leq c_2 \cdot 1 \) is true provided that \( c_2 \geq 100 \). This completes the base case.

**Inductive Step.** Let \( T(i) \leq c_2i \) be true for all \( 1 \leq i < n \). We want to show that \( T(n) \leq c_2n \). Since \( T(i) \leq c_2i \) for all \( i < n \), it is also true for \( i = n/2 \) and \( i = n/3 \) as well, and thus by the inductive hypothesis we obtain \( T(n/2) \leq c_2n/2 \) and \( T(n/3) \leq c_2n/3 \). Then we use these two inequalities in the recurrence to obtain.

\[
T(n) = T(n/2) + T(n/3) + 32n
\]

\[
\leq c_2n/2 + c_2n/3 + 32n
\]

\[
= 5c_2n/6 + 32n
\]
In order to complete the inductive step we need to prove that \( T(n) \leq c_2 n \) or equivalently \( 5c_2 n/6 + 32n \leq c_2 n \). The latter is equivalent to \( 32n \leq c_2 n/6 \) or \( c_2 \geq 192 \). Therefore the inductive step is completed and \( T(n) \leq c_2 n \) for any \( c_2 \geq 192 \). In conclusion, combining the inductive step and the base case we obtain that \( T(n) \leq c_2 n \) for all \( n \geq 1 \) as long as \( c_2 \geq 192 \). We can fix \( c_2 = 192 \) and thus show that \( T(n) = O(n) \).

Therefore since \( T(n) = O(n) \) and \( T(n) = \Omega(n) \) we obtain that \( T(n) = \Theta(n) \). The constants in the definition are 1, 192 and for \( n_0 \) we can still use 1.

Problem 10.

(a) A heap of height \( h \) has its first \( h - 1 \) levels completely filled and in the last level there exists at least one element and at most \( 2^h \). Therefore the number of elements is \( 1 + 2 + \ldots + 2^{h-1} + m = 2^h - 1 + m \), where \( 1 \leq m \leq 2^h \). Therefore the minimum number of elements is \( 2^h \) and the maximum number of elements is \( 2^h + 1 - 1 \).

(b) An \( n \) element heap has height \( h \) such that, by way of part (a), \( 2^h \leq n \leq 2^h + 1 - 1 < 2^{h+1} \). Therefore by taking logarithms base 2 of both sides \( h \leq \log n < h + 1 \). Height \( h \) is always an integer. By the definition of the floor function we derive that \( h = \lfloor \log n \rfloor \).

(c) Let \( S(u) \) be the subtree of a heap rooted at some node \( u \). Suppose by way of contradiction that the maximum value \( M \) stored in \( S(u) \) is not at \( u \). Let the left and right subtrees of \( u \) be \( LS(u) \) and \( RS(u) \). Then the maximum of \( S(u) \) must be in either \( LS(u) \) or \( RS(u) \). Let the index of the heap node of \( S(u) \) with the maximum \( M \) be \( m \) (note that if there are more than one nodes storing \( M \) we take the node with the minimum index value \( m \)). As the node indexed \( m \) cannot be node \( u \) it must be an internal node of \( S(u) \) and therefore it must have a parent \( p \). The index of the parent will be floor\((m/2)\) i.e. smaller than \( m \). The value stored in \( p \) must also be less than \( M \) as \( m \) is the lowest indexed node storing the maximum value \( M \) and therefore the value of the parent whose index is less than \( M \) must be smaller. Therefore the MAX heap-property fails for \( p \) and \( m \), a contradiction to the assumption that \( S(u) \) is a subtree of a heap.

Problem 11.

```java
// MinMax(A,1,r) returns (min,max) , min=min(A[1..r]), max=max(A[1..r]),
// for the subequence of A between A[1] and A[r] (inclusive) of size r-1+1
// A a sequence of size n, n is a power of two ie n=2**k.
MinMax(A,1,r)
1. if lengthof[A[1..r]] is 2
      else return (A[r],A[1]);
}
   // Split A into two halves as in mergesort
3. (min1,max1) = MinMax(A,1,(1+r)/2);  // Call this half A[1...(1+r)/2] A1
4. (min2,max2) = MinMax(A,(1+r)/2+1,r); // and this one A2
   // and recursively find min,max of two halves.
   // Compare minima of two halves to find minimum
5. if min1 <= min2 then min= min1;
   else min= min2;
   // Compare maxima of two halves to find maximum
6. if max1 <= max2 then max= max2;
   else max= max1;
7. return((min,max));
```

The top-level recursive call is

\[
\text{MinMax}(A,1,n)
\]

The number of \textit{key} comparisons \( T(n) \) of \( \text{MinMax}(A,1,n) \) is derived as follows. Step 2 requires one comparison; \( A \) is split as in lines 2-4 of Merge-Sort (page 13). Step 3 requires \( T(n/2) \) comparisons and so does Step 4. We call the half of \( A \) in step 3, \( A1 \) and that of step 4 \( A2 \). In steps 5 and 6 we perform a total of two (2) comparisons. Note that the minimum of \( A \) is the minimum of the minima of \( A1 \) and \( A2 \) and the minimum of \( A \) is the maximum of the maxima of \( A1 \) and \( A2 \). We don’t need to compare the minima to the maxima. Therefore, the recurrence for the number of comparisons becomes:

\[
T(n) = 2T(n/2) + 2T(n/4) + 2 = 2^2T(n/2^2) + 2^2 + 2 = 2^3n^{-1}T(2) + 2^3n^{-1} + \ldots + 2 = n/2 + (-1) + n - 1 = 3n/2 - 2.
\]

with \( T(2) = 1 \) by way of lines 1-2. By the iteration/recursion tree method we obtain that \( T(n) = 3n/2 - 2 \), as required. [Note that the solution of the recurrence was compressed into one line.]
Problem 12.
(a) A d-ary heap is to be represented by an one dimensional array $A$. $A[1]$ stores the root and $A[2], \ldots A[d+1]$ store the $d$ children of the root. The $j$-th child of node indexed $i$ is $d(i-1) + j + 1$ where $1 \leq j \leq d$. Therefore the children of $i$ can have potential indexes $d(i-1) + 2$ to $di + 1$.

If the indexing starts from 0 i.e. the root is at $A[0]$, then the children of $i$ have indices starting with $di + 1$ through $di + d$.

The parent of node $i$ is floor($((i - 2)/d + 1$), in other words floor($((i + d - 2)/d$) if the indexing starts from 1. If the indexing starts from zero, it is floor($((i - 1)/d$). A binary heap is a special case for $d = 2$.

(b) A heap of height $h$ has $n = 1 + d + d^2 + \ldots + d^{h-1} + m = (d^h - 1)/(d - 1) + m$ elements, where $1 \leq m \leq d^h$. Therefore its height is $O(\log n/\log d)$.

(c) Extract-Max works the same way as Heap-Extract-Max for binary heaps except that Heapify($A,i$) on $d$-ary heaps works a little bit differently. It assumes that the $d$ subtrees of node indexed $i$ have the heap property and the maximum is taken over the value stored at $i$ and the values stored in all $d$ children of $i$. The maximum operation takes time $O(d)$ now, and Extract-Max may call Heapify($O(h)$ times. By way of part (b,b) $O(h) = O(\log n/\log d)$ and therefore the time for Heapify will be $O(d\log n/\log d)$.

(d) Insert for binary heaps works fine for $d$-ary heaps as well (remember to use the new definition of a parent though). An upward movement is independent of the number of children of a parent. Therefore its running time is $O(h) = O(\log n/\log d)$.

(e) This is similar to insertion. If the value $k$ is at most the value of $A[i]$ nothing changes, otherwise the value of $A[i]$ is changed and propagated upwards just as we did in Heap-Insert. The observation is that since the value $k$ is propagated upwards, we only need to change the value of the final destination only, just as we did in Heap-Insert. The following program demonstrates this. Note that parent(i) follows the definition of part (b,a) not that of a binary heap.

Heap-Increase-Key ($A,i,k$)
1. If $A[i] >= k$ then return ;
2. while $i>1$ and $A[parent(i)] < k$
4. $i=parent(i);$  
5. $A[i]=k;$

Problem 13.
The two algorithms are

a. Find_M($k$) a ($A[1..n]$,$n$)
1. HeapSort($A$);
2. return($A[k]$);

b. Find_M($k$) b ($A[1..n]$,$n$)
1. BuildMINheap($A,n$);
2. for(i=1;i<k;i++){  
3. $z=EXTRACTMIN(A);$  
4. }  
5. return($z$);

The two pseudocodes are given above.

Algorithm Find_M($k$) a requires $O(n \log n)$ worst-case running time for step 1 (HeapSort can be replaced by MergeSort) and step 2 requires constant time. The total is $O(n \log n)$.

Algorithm Find_M($k$) b requires $O(n)$ worst-case running time for the build-heap operation of step 1, and $O(1)$ time for step 5. The loop involves $k$ EXTRACT-MIN operations. Each one takes $O(\log n)$ time for a total of $O(k \log n)$. Note that the $z$ returned in step 5, is the result of the $k$-th EXTRACT-MIN operation, the $k$-th smallest of the $n$ keys of the input. Total running time is $O(n + k \log n)$.

Let us asymptotically compare the performance of the two algorithms for various values of $k$. To make thing less confusing consider algorithm (a) has performance $n \log n$ and (b) $n + k \log n$.

Test Case 1. Let us start with small values of $k$. Let $k = \Theta(1)$, i.e. $k$ be constant. The running time of (a) is $n \log n$ and (b) is $n + 1 \log n = O(n)$. The latter algorithm is asymptotically faster.

How much can we increase $k$ beyond a constant and still get the same performance? For what values of $k$ is the term $k \log n$ comparable to $n$? This is for $k \log n = O(n)$, i.e. $k = O(n/\log n)$. For such values (b) is $\Theta(n)$ and (a) is $n \log n$ and (b) is asymptotically faster.

Test Case 2. Suppose that $k = \omega(n/\log n)$. Algorithm (b) has performance $n + k \log n = n + \omega(n/\log n) \log n$ which is $\omega(n + (n/\log n) \log n)$ i.e. it is $\omega(n)$.

Test Case 3. However as long as $k = o(n)$, the term $n + k \log n$ is still $n + o(n \log n) = o(n \log n)$, i.e. algorithm (b) is still asymptotically faster that algorithm (a).

Case. A.
Cases 1, 2, and 3 can become one case: if \(k = o(n)\) then (b) is asymptotically faster than (a).

**Case B.** If \(k\) is not \(k = o(n)\), then \(k = \Theta(n)\), and algorithm (b) has performance \(n + k \lg n = O(n \lg n)\) and both algorithms have the same asymptotic performance.

**Problem 14.**

Sort the keys with merge-sort. This takes \(O(n \lg n)\) worst-case running time. Duplicate keys appear next to each other in the sorted sequence. The most frequent of them is the desired key, i.e. we need to count the occurrences of a duplicate key. To do so we do a linear scan left to right of the output of merge-sort (i.e. \(O(n)\) additional running time for the scan) on the sorted array to determine the longest repetition of a given key. We use counters; every time we move from key \(x\) to a new key \(y\), \(x \neq y\) we reset the counters and check whether \(x\) is more frequent than the current most frequent key stored in a maxkey variable and update maxkey as needed; then we proceed to examining the frequency of \(y\). If we go from \(x\) to \(y = x\) we update the counters (increment a frequency counter by 1). Total time is \(O(n \lg n) + O(n)\) i.e. \(O(n \lg n)\).

If pseudocode is desired, the following is a good approximation. We keep track of the frequency of the current key in count, the maximum frequency in maxfreq and the index of the key attaining it (in the sorted sequence) in maxkeyindex (i.e. the max frequency key has value \(A[\text{maxkeyindex}]\)). The key at \(A[0]\) is used initially as the current best (line 2) and its index (0) stored in maxkeyindex along with its current frequency (1) in maxfreq and count.

In line 3 we go through the remaining \(n-1\) keys (C++/C notation for arrays). If the key value for \(A[i]\) is the same as before (start) we increment the counter count by one. Otherwise we have a new key with possibly higher frequency. If this is so (test of line 6) we update our relevant info to reflect this (lines 7-9) otherwise we restart with this new key (count, start are updated as in line 13). The code of lines 1-16 works fine unless the highest-frequency key is the last one, i.e \(A[n-1]\). For this we need lines 17-19, to detect and correctly record that key.

```cpp
HighFrequency(A[0..n-1],n) //A C++ version
1. A= Sort(A,n)
2. count=1; maxkeyindex = 0;maxfreq=1; start=A[0];
3. for(i=1;i< n;i++) {
4.    if (A[i] == start) count++;
5.    else {
6.        if (count > maxfreq ) {
7.            maxfreq=count;
8.            maxkeyindex=i-1;
9.            count=1;
10.           start=A[i];
11.        }
12.        else {
13.            count=1; start=A[i];
14.        } 
15.    }
16. }
17. if (count >maxfreq ) {
18.    maxkeyindex=n-1; maxfreq=count;
19. }
20. print(A[maxkeyindex] has frequency maxfreq); // print is generic not C++/C or Java!
```

The pseudocode is also easy to analyze. Line 1 is \(O(n \lg n)\). The loop of line 3 runs for \(n-1\) times each time executing a constant number of operations (lines 4-15), i.e. total cost for these lines is \(O(n)\). Lines 17-20 cost \(O(1)\). Total cost is \(O(n \lg n) + O(n) + O(1) = O(n \lg n)\).
Problem 15.

We count the As, Bs, Cs and Ds. We go through the keys once \((O(n)\) time) and count the 1’s etc in lines 2-7. Then we print enough As etc in linear time \(O(n)\) (lines 8-16).

The solution is very simple, very inelegant but it works! More on this problem called count-sort later in class.

```
Count1234 (A,n)
1. c1=c2=c3=c4=0;
2. for(i=1;i<=n;i++) { //Very naive but it works!
3.   if (A[i] == A) c1++;// We don’t care here about syntactice correctness cf A vs ’A’
4.   if (A[i] == B) c2++;// How can you optimize these rather sloppy lines 3-6?
5.   if (A[i] == C) c3++;
6.   if (A[i] == D) c4++;
7. }
8. c=1; // What can you do to optimize lines 9-17?
9. for(i=1;i<=c1;i++) // print as many A as c1
10.  A[c++]=A;
11. for(i=1;i<=c2;i++) // print the Bs
13. for(i=1;i<=c3;i++) // and the Cs
15. for(i=1;i<=c4;i++) // and the Ds
17. return(A)
```

Problem 16.

No solution is provided. It’s similar to some other problem (aka 4)!