Math 391: Midterm 1.0  
Spring 2016

James Skon  
skonjp@kenyon.edu

Test date: October 5  
Submission Instructions:

• Give all your assumptions.  
• Show all your work.  
• Be as complete as possible.

Grading

• Problem 1 (8 points):

• Problem 2 (8 points):

• Problem 3 (9 points):

• Problem 4 (15 points):

• Problem 5 (15 points):

• Problem 6 (10 points):

• Problem 7 (10 points):

• Problem 8 (15 points):

• Problem 9 (10 points):

• Total Score (100 points):
Problem 1 (8 points): For each of the functions \( f(N) \) given below, indicate the tightest bound possible (in other words, giving \( O(2^N) \) as the answer to every question is not likely to result in many points). Unless otherwise specified, all logs are base 2. You MUST choose your answer from the following (not given in any particular order), each of which could be re-used (could be the answer for more than one of a) - h)):

\[ O(N^2), O(N^3 \log N), O(N \log N), O(N), O(N^2 \log N), O(N^3), O(2^N), O(N^3), O(\log N), O(1), O(N^4), O(N^N) \]

You do not need to explain your answer.

\[
f(N) = N \cdot (100N + 200N^3) + N^3 \quad O(N^4)
\]

\[
f(N) = N^2 \log N + N^3 + 1000^4 \quad O(N^3)
\]

\[
f(N) = 100N + (N/2) \log(N/2) + N/4 \quad O(N \log N)
\]

\[
f(N) = (N/4) + N^{3/2} \quad O(N)
\]

\[
f(N) = N \log(N^4) + 3N^2 \quad O(N^2)
\]

\[
f(N) = (N^3 + N^1)/N \quad O(N^2)
\]

\[
f(N) = 400N^2 - 20N \quad O(N^2)
\]

\[
f(N) = N^2 \log N + N \log(N^2) \quad O(N^2 \log N)
\]
**Problem 2 (8 points):** Big-Oh and Run Time Analysis: Describe the worst case running time of the following pseudocode functions in Big-Oh notation in terms of the variable $n$. Showing your work is not required (although showing work may allow some partial credit in the case your answer is wrong don't spend a lot of time showing your work.). You MUST choose your answer from the following (not given in any particular order), each of which could be re-used (could be the answer for more than one of I. IV.): $O(n^2), O(n^3 \log n), O(n \log n), O(n), O(n^2 \log n), O(n^3), O(2^n), O(n^3), O(\log n), O(1), O(n^4), O(n^n)$

I. void silly(int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            System.out.println(j = + j);
            for (int k = 0; k < j; ++k)
                System.out.println(k = + k);
        }
    }
}

$O(n^3)$

II. void silly(int n, int x, int y) {
    for (int i = 0; i < n; ++i) {
        if (x > y) {
            for (int j = 0; j < n; ++j)
                System.out.println(j = + j);
            for (int k = 0; k < n * n; ++k)
                System.out.println(k = + k);
        } else
            System.out.println(i = + i);
    }
}

$O(n^3)$

III. void silly(int n, int m) {
    if (m > n) return;
    System.out.println(m = + m);
    silly(n, m+2);
}

$O(n)$

IV. void silly(int n) {
    for (int i = 0; i < n; i = i + 10) {
        for (int j = 0; j < i; ++j) {
            System.out.println(i = + i);
            System.out.println(j = + j);
        }
    }
}

$O(n^2)$
Problem 3 (9 points): a) Give (e.g. draw) an adjacency-list representation for a complete binary tree on 7 vertices. b) Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

Assume the tree below:

```
1
/  \
/    \
2    3
\    /  /
4  5  6  7
```

a)  

b)  

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Problem 4 (15 points): Give proofs for each of the following.

(a) Show that for any real constants $a$ and $b$, where $b > 0$,
\[(n + a)^b = \Theta(n^b)\].

(b) Is $2^{n+1} = O(2^n)$?

(c) Is $2^{2n} = O(2^n)$?

(a) To show that $(n + a)^b = \Theta(n^b)$ we want to find constants $c_1, c_2, n_0 > 0$ such that
\[0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b\] for all $n \geq n_0$.

Note that
\[n + a \leq n + |a| \leq 2n\] when $|a| \leq n$.

and
\[n + a \geq n - |a| \geq \frac{1}{2}n\] when $|a| \leq \frac{1}{2}n$.

Thus when $n \geq 2|a|$.

\[0 \leq \frac{1}{2}n \leq n + a \leq 2n.\]

Since $b > 0$, the inequality still holds when all parts are raised to the power $b$:
\[0 \leq \left(\frac{1}{2}n\right)^b \leq (n + a)^b \leq (2n)^b,\]
\[0 \leq \left(\frac{1}{2}n\right)^b \leq (n + a)^b \leq 2^b n^b.\]
Thus, $c_1 = (1/2)^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ satisfy the definition.

(b) To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that
\[0 \leq 2n + 1 \leq c \cdot 2^n\] for $n \geq n_0$.

Since $2n + 1 = 2 \cdot 2^n$ for all $n$ we can satisfy the definition with $c = 2$ and $n_0 = 1$.

(c) To show that $2^{2n} \neq O(2^n)$, assume $c, n_0 > 0$ such that
\[0 \leq 2^{2n} \leq c \cdot 2^n\] for all $n \geq n_0$.

Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. But no constant is greater than all $2^n$, and and so the assumption leads to a contradiction.
**Problem 5 (15 points):** Relative asymptotic growths

Indicate, for each pair of expressions \((A, B)\) in the table below, whether \(A\) is \(O, o, \Omega, \omega, \Theta\) of \(B\). Assume that \(k \geq 1, \epsilon > 0,\) and \(c > 1\) are constants. Your answer should be in the form in the table with yes or no written in each box.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>O</th>
<th>o</th>
<th>Ω</th>
<th>ω</th>
<th>Θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>(\lg^k n)</td>
<td>(n^\epsilon)</td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>on</td>
</tr>
<tr>
<td>b.</td>
<td>(n^k)</td>
<td>(c^n)</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>c.</td>
<td>(\sqrt{n})</td>
<td>(n^{\sin n})</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>d.</td>
<td>(2^n)</td>
<td>(2^{n/2})</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e.</td>
<td>(n^{\lg c})</td>
<td>(c^{\lg n})</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>f.</td>
<td>(\lg(n!))</td>
<td>(\lg(n^n))</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

**Reasons:**

a) Any polylogarithmic function is little-oh of any polynomial function with a positive exponent.

b) Any polynomial function is little-oh of any exponential function with a positive base.

c) The function \(\sin n\) oscillates between \(-1\) and 1. There is no value \(n_0\) such that \(\sin n\) is less than, greater than, or equal to 1/2 for all \(n \geq n_0\), and so there is no value \(n_0\) such that \(n^{\sin n}\) is less than, greater than, or equal to \(cn^{1/2}\) for all \(n \geq n_0\).

d) Take the limit of the quotient: \(\lim_{n \to \infty} 2^n / 2^{n/2} = \lim_{n \to \infty} 2^{n/2} = \infty\).

e) By equation \(a^{\log_b c} = c^{\log_b a}\), these quantities are equal.

f) By equation \(\lg(n!) = \Theta(n \log n)\). Since \(\lg(n^n) = n \lg n\), these functions are \(\Theta\) of each other.
Problem 6 (10 points): Binary Min-heap problem

a. Draw the binary min heap that results from inserting 4, 9, 3, 7, 2, 5, 8, 6 in that order into an initially empty binary min heap. You do not need to show the array representation of the heap. You are only required to show the final tree, although drawing intermediate trees may result in partial credit. If you draw intermediate trees, please circle your final result for any credit.

b. Given the binary min heap below, show what it will look like after removing 4 (minimum) nodes:
Problem 7 (10 points): Prove that a complete graph with \( n \) vertices contains \( n(n - 1)/2 \) edges.

Proof: This is easy to prove by induction. BASE: If \( n = 1 \), zero edges are required, and \( 1(1 - 0)/2 = 0 \).

INDUCTION: Assume that a complete graph with \( k \) vertices has \( k(k - 1)/2 \). When we add the \((k + 1)st\) vertex, we need to connect it to the \( k \) original vertices, requiring \( k \) additional edges. We will then have \( k(k - 1)/2 + k = (k + 1)((k + 1) - 1)/2 \) vertices, and we are done.
Problem 8 (15 points): Prove that the following algorithm is correct. The purpose of the algorithm is to find the maximum value in an array and return the index of that number.

```c
int find_max(int a[], int size) {
    assert size>0;
    int max_val = a[0];
    int max_loc = 0;
    int i=1;
    while (i<size) {
        if (a[i]>max_val) {
            max_val = a[i];
            max_loc = i;
        }
        i++;
    }
    return max_loc;
}
```

(a) Give a loop invariant for this algorithm:

(b) Base Case:

(c) Inductive Case:

(d) Termination:
**Loop invariant**: We can claim that the following is a loop invariant that always holds at the loop test:

- $\text{max_val} = a[\text{max_loc}]$, and
- $\text{max_val} = \text{max over all entries in the array from } a[0..i-1]$

**Base Case**: When we first arrive at the loop test, $i=0$, so we are considering array entries from $a[0]$ through $a[0]$. Obviously, $\text{max_val}=a[0]=\text{the max of } a[0]$, and $\text{max_loc}=0$, so the loop invariant holds.

**Inductive Case**: Assume the loop invariant holds, and also that the loop test passes, i.e., that $i < \text{size}$. There are two cases: either $a[i] > \text{max_val}$ or it isn’t. If $a[i] > \text{max_val}$, then clearly it is the largest value in the array from $a[0]$ through $a[i]$. Therefore, if we set $\text{max_val}$ to $a[i]$ and $\text{max_loc}$ to $i$, and then increment $i$, the loop invariant still holds. Similarly, if $a[i] \leq \text{max_val}$, then $\text{max_val}$ and $\text{max_loc}$ don’t have to change, so we can increment $i$, and the loop invariant also still holds.

**Termination**: After the loop terminates, $i==\text{size}$. Combined with the loop invariant, this means that $\text{max_val} = \text{max over all entries in the array from } a[0..\text{size-1}]$, in other words, the entire array, and that $\text{max_val}=a[\text{max_loc}]$. So, $\text{max_loc}$ is the location of a largest element of the array.

The loop always terminates because $i$ increases by 1 each iteration, so it must eventually reach $\text{size}$. 
Problem 9 (10 points): Greedy Algorithms

There are \( n \) white dots and \( n \) black dots, equally spaced, in a line. You want to connect each white dot with some one black dot, with a minimum total length of wire.

Example:

![Image of connected dots]

Do you see a greedy algorithm for doing this?

(a) Describe a greedy algorithm in english for this problems. Then give your estimate as to whether you approach finds the optimal solution or not. (proof not needed)

(b) Looking at your algorithm, can you give it reliable big-oh estimate for complexity based on the description of the algorithm? If you believe you can, do so, if not, explain why not.

(a) There are many possible greed approaches to this problem, some better then others. What I will be looking for is a method that uses a greedy approach, e.g. makes greedy local choices, and never goes back. 
Example: Scan from left to right one dot at a time. For each dot encountered look back (right to left from this dot) to check if there is an unmatched dot of the opposite color. If so, connect the current with the nearest (looking right) opposite color dot. If there isn’t a previous unmatched opposite color dot, then scanning right. If there are a matching number of black and white dots, when we reach the end all will be paired.

(b) Consider that we are scanning left to right, and then scanning back each time a color changes. This could certainly lead to estimated bound of \( O(n^2) \). In the worst case we have all one color then the other. However, if we keep count of the number of nodes of each color that have been scanned but not matched, then we only scan back when we know there are unmatched nodes waiting. But what will this reduce us too. 
Ultimately it is difficult to me a tight estimate unless we fully consider the data structures and algorithms we use to implement the underlying representations and operations. So, though \( O(n^2) \) is certainly a likely upper bound, this is probably as close as we can get it without careful design of the entire solution.