CS 435, Online Algorithms	**Homework 1 Algorithms Analysis**	Dr. David Nassimi

1. Prove the following polynomial is $\Theta(n^3)$. That is, prove $T(n)$ is both $O(n^3)$ and $\Omega(n^3)$.

$$T(n) = 2n^3 - 10n^2 + 100n - 50$$

   (a) Prove $T(n)$ is $O(n^3)$: By definition, you must find positive constants C_1 and n_0 such that

$$T(n) \le C_1 n^3, \quad \forall n \ge n_0.$$

   (b) Prove $T(n)$ is $\Omega(n^3)$: By definition, you must find positive constants C_2 and n_0 such that

$$T(n) \ge C_2 n^3, \quad \forall n \ge n_0.$$

   Note: Since the highest term in $T(n)$ is $2n^3$, it is possible to pick n_0 large enough so that C_1 and C_2 are close to the coefficient 2. (The definitions of $O()$ and $\Omega()$ are not concerned with this issue.) For this problem, you are required to pick n_0 so that C_1 and C_2 fall within 10% of the coefficient 2. That is,

$$C_2 n^3 \le T(n) \le C_1 n^3, \quad \forall n \ge n_0$$

   where $C_2 \ge 1.8$ and $C_1 \le 2.2$.

2. (a) Compute and tabulate the following functions for $n = 1, 2, 4, 8, 16, 32, 64$. The purpose of this exercise is to get a feeling for these growth rates and how they compare with each other. (All logarithms are in base 2, unless stated otherwise.)

$$\log n, \ n, \ n\log n, \ n^2, \ n^3, \ 2^n.$$

   (b) Order the following complexity functions (growth rates) from the smallest to the largest. That is, order the functions asymptotically. Note that $\log^2 n$ means $(\log n)^2$.

$$n^2 \log n, \ 5, \ n\log^2 n, \ 2^n, \ n^2, \ n, \ \sqrt{n}, \ \log n, \ \frac{n}{\log n}$$

   The comparison between some of the functions may be obvious (and need not be justified). If you are not sure how a pair of functions compare, you may use the **ratio test** described below.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{if } f(n) \text{ is asymptotically smaller than } g(n), \\ \infty & \text{if } f(n) \text{ is asymptotically larger than } g(n), \\ C & \text{if } f(n) \text{ and } g(n) \text{ have the same growth rate.} \end{cases}$$

   Note: For any integer constant k, $\log^k n$ is a smaller growth rate than n. This may be proved using the ratio test.

3. Find the exact number of times (in terms of n) the innermost statement ($X = X + 1$) is executed in the following code. That is, find the final value of X. Then express the total running time in terms of $O(\ )$, $\Omega(\ )$, or $\Theta(\ )$ as appropriate.

```
X = 0;
for k = 1 to n
   for j = 1 to n - k
      X = X + 1;
```

4. The following program computes and returns ($\log_2 n$), assuming the input n is an integer power of 2. That is, $n = 2^j$ for some integer $j \geq 0$.

```
int LOG (int n){
int m, k;
m = n;
k = 0;
while (m > 1) {
   m = m/2;
   k = k + 1; }
ruturn (k)
}
```

   (a) First, trace the execution of this program for a specific input value, $n = 16$. Tabulate the values of m and k at the beginning, just before the first execution of the while loop, and after each execution of the while loop.

   (b) Prove by induction that at the end of each execution of the while loop, the following relation holds between variables m and k. (This relation between the variables is called the *loop invariant.*)

$$m = n/2^k.$$

   (c) Then conclude that at the end, after the last iteration of the while loop, the program returns $k = \log_2 n$.

5. The following pseudocode computes the sum of an array of n integers.

```
int sum (int A[ ], int n) {
T = A[0];
for i = 1 to n − 1
   T = T + A[i];
return T;
}
```

   (a) Write a recursive version of this code.

   (b) Let $f(n)$ be the number of additions performed by this computation. Write a recurrence equation for $f(n)$. (Note that the number of addition steps should be exactly the same for both the non-recursive and recursive versions. In fact, they both should make exactly the same sequence of addition steps.)

   (c) Prove by induction that the solution of the recurrence is $f(n) = n - 1$.

6. The following pseudocode finds the maximum element in an array of size n.

```
int MAX (int A[ ], int n) {
M = A[0];
for i = 1 to n − 1
   if (A[i] > M)
      M = A[i]    // Update the max
return M;
}
```

   (a) Write a recursive version of this program.

(b) Let $f(n)$ be the number of key comparisons performed by this algorithm. Write a recurrence equation for $f(n)$.

(c) Prove by induction that the solution of the recurrence is $f(n) = n - 1$.

7. Consider the following pseudocode for insertion-sort algorithm. The algorithm sorts an arbitrary array $A[0..n-1]$ of n elements.

```
void ISORT (dtype A[ ], int n)
{ int i, j;
for i = 1 to n − 1
   { // Insert A[i] into the sorted part A[0..i − 1]
   j = i;
   while (j > 0 and A[j] < A[j − 1]) {
      SWAP (A[j], A[j − 1]);
      j = j − 1 }
   }
}
```

(a) Illustrate the algorithm on the following array by showing each comparison/swap operation. What is the total number of comparisons made for this worst-case data?

$$A = (5, 4, 3, 2, 1)$$

(b) Write a recursive version of this algorithm.

(c) Let $f(n)$ be the worst-case number of key comparisons made by this algorithm to sort n elements. Write a recurrence equation for $f(n)$. (Note that the sequence of comparisons are exactly the same for both non-recursive and recursive versions. But, you may find it more convenienet to write the recurrence for the recursive version.)

(d) Find the solution for $f(n)$ by repeated substitution.

8. Consider the bubble-sort algorithm described below.

```
void bubble (dtype A[ ], int n)
{ int i, j;
for (i = n − 1;  i > 0;  i − −)                    //Bubble max of A[0..i] down to A[i].
   for (j = 0;  j < i;  j + +)
      if (A[j] > A[j + 1])    SWAP(A[j], A[j + 1]);
}
```

(a) Analyze the time complexity, $T(n)$, of the bubble-sort algorithm.

(b) Rewrite the algorithm using recursion.

(c) Let $f(n)$ be the worst-case number of key-comparisons used by this algorithm to sort n elements. Write a recurrence for $f(n)$. Solve the recurrence by repeated substitution (i.e, iteration method).

9. The following algorithm uses a **divide-and-conquer** technique to find the maximum element in an array of size n. The initial call to this recursive function is max(arrayname, 0, n).

```
dtype Findmax(dtype A[ ], int i, int n)
{   //i is the starting index, and n is the number of elements.
dtype Max1, Max2;
if (n == 1) return A[i];
Max1 = Findmax (A, i, ⌊n/2⌋);               //Find max of the first half
Max2 = Findmax (A, i + ⌊n/2⌋, ⌈n/2⌉);       //Find max of the second half
if (Max1 ≥ Max2)   return Max1;
    else return Max2;
}
```

Let $f(n)$ be the worst-case number of *key comparisons* for finding the max of n elements.

(a) Assuming n is a power of 2, write a recurrence relation for $f(n)$. Find the solution by each of the following methods.

i. Apply the repeated substitution method.

ii. Apply induction to prove that $f(n) = An + B$ and find the constants A and B.

(b) Now consider the general case where n is any integer. Write a recurrence for $f(n)$. Use induction to prove that the solution is $f(n) = n - 1$.

10. The following divide-and-conquer algorithm is designed to return TRUE if and only if all elements of the array have equal values. For simplicity, suppose the array size is $n = 2^k$ for some integer k. Input S is the starting index, and n is the number of elements starting at S. The initial call is SAME$(A, 0, n)$.

```
Boolean SAME (int A[ ], int S, int n) {
Boolean T1, T2, T3;
if (n == 1) return TRUE;
T1 = SAME (A, S, n/2);
T2 = SAME (A, S + n/2, n/2);
T3 = (A[S] == A[S + n/2]);
return (T1 ∧ T2 ∧ T3);
}
```

(a) Explain how this program works.

(b) Prove by induction that the algorithm returns TRUE if and only if all elements of the array have equal values.

(c) Let $f(n)$ be the number of key comparisons in this algorithm for an array of size n. Write a recurrence for $f(n)$.

(d) Find the solution by repeated substitution

Additional Exercises (Not to be handed-in)

11. One of the earlier problems above presented the program to compute $\log_2 n$ when input is $n = 2^j$ for some integer j.

(a) Generalize this algorithm to compute $\lfloor \log_2 n \rfloor$ where input n is any integer, $n \geq 1$.

(b) Trace the algorithm for $n = 14$ to see it works correctly.

(c) Prove by induction that the algorithm works correctly for any integer n.

Hint: Observe that any integer n always falls between two consecutive powers of 2. (For example, for $n = 14$, $2^3 < 14 < 2^4$.) In general, for every integer n,

$$2^k \leq n < 2^{k+1}$$

for some integer k. This will be helpful in your induction proof.

12. Consider a $2^n \times 2^n$ board, with one of its four quadrants missing. That is, the board consists of only three quadrants, each of size $2^{n-1} \times 2^{n-1}$. Let's call such a board a quad-deficient board. For $n = 1$, such a board becomes an L-shape 3-cell piece called a **tromino**, as shown below.

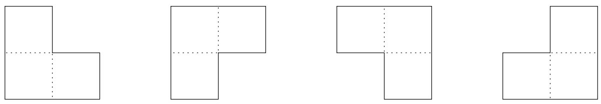

Figure 1: A Tromino, with 4 possible rotational positions.

(a) Use a **divide-and-conquer** technique to prove by induction that a quad-deficient board of size $2^n \times 2^n$, $n \geq 1$ can always be **covered** using some number of trominoes. (By covering we mean that every cell of the board must be covered by a tromino piece, and the pieces must not overlap.) Use a diagram to help describing your algorithm and the proof.

(b) Illustrate the covering produced by the algorithm for $n = 3$ (that is, $2^3 \times 2^3$ board).

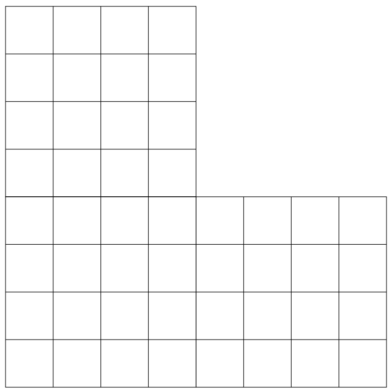

Figure 2: An 8×8 quad-deficient board.

(c) Let $f(n)$ be the total number of trominoes used to cover a $2^n \times 2^n$ quad-deficient board. Write a recurrence for $f(n)$. Solve the recurrence by repeated substitution.

13. The Fibonacci sequence [1] is defined as $F_1 = 1$, $F_2 = 1$, and

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 3.$$

[1] **Historical Note:** Originally, Fibonacci came up with this recurrence to describe the population growth of rabbits! Suppose that at the beginning of the year, a farm receives one pair of newly-born rabbits, and that every month each pair which is at least two-month old gives birth to a new pair. Let F_n be the number of pairs at the end of month n, assuming that no deaths occur. Then, the number of pairs that are born at the end of month n is F_{n-2}, thus the recurrence $F_n = F_{n-1} + F_{n-2}$. What is the number of pairs at the end of the year?

(a) Compute and tabulate F_n for $n = 0$ to 12.

(b) Prove the following **lower bound** on F_n.

$$F_n \geq 2^{\lfloor (n-1)/2 \rfloor}, \quad n > 2.$$

Hint: For $n > 2$, observe that $F_{n-1} \geq F_{n-2}$. (Why?) Using this relation, obtain a simpler recurrence: $F_n \geq 2F_{n-2}, \ n > 2$. Then apply repeated substitution.

(c) Prove the following **upper bound** for F_n.

$$F_n \leq 2^{n-2}, \quad n > 2.$$

Hint: Again, use the relation $F_{n-2} \leq F_{n-1}$ to obtain a simpler recurrence: $F_n \leq 2F_{n-1}, \ n > 2$. Then apply repeated substitution.

14. (a) Prove (without use of calculus) that

$$\log(n!) = \Theta(n \log n).$$

Hints:

- First observe that $\log(n!) = \sum_{i=1}^{n} \log i$.
- Then prove the upper bound in a trivial way.
- One way to prove the lower bound is by considering only the larger $n/2$ terms in the summation. That is, $\sum_{i=1}^{n} \log i > \sum_{i=\lceil n/2 \rceil}^{n} \log i$.

(b) Prove that

$$\sum_{i=1}^{n} (i \log i) = \Theta(n^2 \log n).$$