Kaitlin Tabernik
2889753

Fall 2025 CIS 390/550 Homework #1
(Due September 23 – before class) - 50 points

Queston 1. (*10 points*) **Euclid's Algorithm** [*hint: Lecture 1, slide 8*]

a. (*5 points*) Find gcd (67890,12345) by applying Euclid's algorithm.

67890 mod 12345 = 6165
12345 mod 6165 = 15
6165 mod 15 = 0
gcd = 15

b. (*5 points*) Estimate how many times faster it will be to find gcd (67890,12345) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from $\min\{m, n\}$ down to $\gcd(m, n)$.

$12345 - 15 + 1 \approx 12331$ candidates.
Euclid took 3 modulus steps.
Speedup $\approx 12331 / 3 \approx 4.1 \times 10^3$
~4100 times faster

Queston 2. (*10 points*) **Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and the information to put the elements in its appropriate position in the sorted array.**

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: Array A[0..n − 1] of orderable values
//Output: Array S[0..n − 1] of A's elements sorted
//    in nondecreasing order
for i ← 0 to n − 1 do
    Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
return S
```

a. (*6 points*)

Apply this algorithm to sorting the list 48, 19, 24, 58, 21, 2.

48 → 4 smaller elements

19 → 1 smaller elements

24 → 3 smaller elements

58 → 5 smaller elements

21 → 2 smaller elements

2 → 0 smaller elements

Place at index = count so, [2, 19, 21, 24, 48, 58]

Sorted output: 2, 19, 21, 24, 48, 58

b. (*2 points*) Is this algorithm stable? [*hint: An algorithm is stable if it doesn't change the position of equal elements in the input when producing the output.*]
No, it doesn't store which order the equal numbers were originally in so it will not be guaranteed to restore in the sorted array in the original order.

c. (*2 points*) Is it in-place? [*hint: An algorithm is in-place if it doesn't require much extra memory and mainly operates on the original data, without creating a significant additional data structure.*]
No, it uses extra memory to store the smaller numbers and cannot sort in the original array as it would have to in this case.

Queston 3. (*10 points*) Rank the following functions by order of growth, from slowest growing to fastest-growing. That is, find an arrangement f1, f2, ...f13 of the following functions satisfying f1 ∈ O(f2), f2 ∈ O(f3), etc. Hint: Try applying the rules of Theorem in slide. [*hint: Lecture 2, slide 19*]

$10n \log n$; 5^{100} ; $\log(\log n)$; $\log^2 n$; $2^{\log n}$; $6\sqrt{n}$; $n^{0.001}$; $n^3 \log n$
$5 \log n$; $7n$; n^3 ; n^2 ; $n!$

1. 5^{100}
2. $\log(\log n)$
3. $\log^2 n$
4. $n^{0.001}$
5. $6\sqrt{n}$
6. $5 \log n$
7. $10n \log n$

8. n^2
9. n^3
10. $n^3 \log n$
11. $2^{\log n}$
12. 7n
13. n!

Queston 4. (*20 points*) For the following algorithms, derive the big O running times.

a. (*5 points*) A proposed function representing the number of primitive operations for the algorithm in terms of the input size, addressing each line and/or loop of the algorithm. You do not need to be precise counting constant numbers of primitive operations (e.g., figuring out exactly how many primitive operations a line does). However, you should be precise about how many times a loop runs.

b. (*3 points*) For your proposed function representing the number of primitive operations f(n), provide the upper bound for the function f(n) such that (wouldn't convert to docx.)

c. (*2 points*) Clear communication: a mix of mathematical notation and explanations in words.

Algorithm 1

```
for i = 1 to n*n do
 if i is even then
 for j = 1 to n do
 x = x + 1
```

a) f(n): The outer loop runs n^2 times. We check "even" n^2 times. About half of those (≈ n^2/2) are even, and for each even i the inner loop runs n times. So:
f(n) = c1·n^2 (checks) + c2·(n^2/2)·n (increments) = O(n^3).
b) Big-O: O(n^3).
c) In words: We have n^2 iterations total, and roughly half trigger another loop of length n, which gives around (n^2/2)·n = n^3/2 work. The even checks are only n^2, so the n^3 term dominates.

Algorithm 2

```
for i = 1 to n*n do
 if n|i then
 for j = 1 to n do
 x = x + 1
```

a) f(n): The outer loop also runs n^2 times, so we do n^2 divisibility checks. Only multiples of n pass the "n|i" test, and in 1..n^2 there are exactly n of those. Each time it passes, the inner loop runs n times. So:
f(n) = c1·n^2 (checks) + c2·n·n (increments) = O(n^2).
b) Big-O: O(n^2).

c) In words: Even though the outer loop is n^2, only every n-th iteration does real work, and that work is size n each time. That's n triggers × n work = n^2 total, which matches the n^2 checks, so overall O(n^2).