Name: Section:

CSCI2300 – Introduction to Algorithms
Spring 2018, Exam I (100 Points)

No electronic devices allowed, i.e., no calculators, laptops, tablets, etc. Show all work for full credit.

1. (20 points) Answer the following:

   (a) (5 points) Is $2^{n+1} = O(2^n)$? Why or why not?

   (b) (5 points) Is $2^{2n} = O(2^n)$? Why or why not?

   (c) (5 points) Is $\lceil \log_2 n \rceil! = O(n^k)$ for some constant $k \geq 1$? Why or why not?

   (d) (5 points) True or False: If $x \cdot y \equiv 0 \mod N$, then either $x \equiv 0 \mod N$ or $y \equiv 0 \mod N$. Why or why not?

   Answer:

   (a) Yes, since $2^{n+1} = 2 \cdot 2^n = O(2^n)$.

   (b) No. To see this, consider the ratio of 2^{2n} to 2^n, we get: $\frac{2^{2n}}{2^n} = 2^{2n-n} = 2^n$. In other words 2^{2n} grows faster than 2^n.

   (c) No. We can approximate $\lceil \log_2 n \rceil!$ as $O((\log_2 n)^{(\log_2 n)})$. Let's compare this with n^k. Taking log on both terms, we get:

   $$\log_2((\log_2 n)^{(\log_2 n)}) \text{ vs. } \log_2(n^k)$$

   or

   $$\log_2 n \cdot \log_2 \log_2 n \text{ vs.} k \log_2 n$$

   Cancelling $\log_2 n$ on both sides, we find that there is no fixed constant k that exceeds $\log_2 \log_2 n$.

   (d) This is false. For example, $2 \cdot 2 = 4 \equiv 0 \mod 4$, but obviously $2 \not\equiv 0 \mod 4$.

2. (20 points) Consider the following algorithm to divide an integer x with integer y, where $x \geq y$. The method returns the quotient q and remainder r.

```
div(x,y):
    r = x
    q = 0
    while (r >= y):
        r = r - y
        q = q + 1
    return (q, r)
```

(a) (10 points) Prove that the method is correct.

(b) (10 points) What is the bit-complexity of the algorithm if x requires n bits.

Answer:

(a) This method implements the basic definition of division, i.e., it repeatedly subtracts y from r, starting from $r = x$, and tabulates how many times this is done in q, which forms the quotient.

The final remainder r is certainly less than y, since if r is greater than or equal to y, then the loop would be executed at least one more time. On the other hand, the final reminder cannot be negative, since in that case we get r by subtracting y from the previous value, say r', i.e., $r = r' - y < 0$, which implies $r' < y$, but in that case the loop would not have been be executed. Thus, the remainder $r \in [0, y - 1]$ as required.

(b) The worst case is when the value of x is $2^n - 1$ and say the value of $y = 1$. In this case, y has to be subtracted as many times as the value of x, and each subtraction takes $O(n)$ time for n-bit integers. Therefore the total time is $O(n \cdot 2^n)$.

Name: Section:

3. (20 points) Consider the RSA encryption scheme. Let $N = 319$ and $e = 3$.

   (a) (10 points) What is the value of the secret key, d?

   (b) (10 points) What is the encryption of the message $M = 100$?

Answer:
(a) We have to first factorize N into the two primes p and q. We have $N = pq = 11 \times 29 = 319$, also $(p-1)(q-1) = 280$.
The value d should be chosen so that $ed \equiv 1 \mod 280$.
Now, $280 = 93 \times 3 + 1$, or
$1 = 1 \times 280 + (-93) \times 3$, which implies $d = -93$ or $d = -93 + 280 = 187$.

(b) The encryption is $100^3 \mod 319 = 1000000 \mod 319 = 254$.

4. (15 points) Given an unsorted array A with n numbers, your task is to output the k largest numbers in sorted order in $O(n + k \log k)$ average case time. Give a (high-level) pseudo-code of your algorithm, with explanation (you can use known algorithms as subroutines). Show that its running time is indeed $O(n + k \log k)$.

Answer:

(a) Consider the following algorithm:

1. Use randomized selection to find the k-th largest element, say v
2. Use v as a pivot to find all elements larger than v
3. Sort those k elements

(b) Selecting the k-th largest item via randomized selection takes $O(n)$ time. Using it as pivot to obtain the k largest elements takes $O(n)$ time, and then sorting them takes $O(k \log k)$ time. Therefore, the total time is $O(n + k \log k)$.

5. (25 points) Consider the following sorting algorithm.

```
TSORT(A,i,j):
    base case: ?
    k = floor((j-i+1)/3) # find the location of 1/3rd
    TSORT(A,i,j-k) #sort the first two-thirds of array in-place
    TSORT(A,i+k,j) #sort the second two-thirds of array in-place
    TSORT(A,i,j-k) #sort the first two-thirds again in-place
```

The sorting method performs "in-place" sorting of an input array by sorting the first two-thirds, followed by second two-thirds and then first two-thirds again. "In-place" means the array is directly modified in each call.

(a) (10 points) Prove that the method correctly sorts an input array A with n elements, when called as TSORT$(A, 0, n-1)$.

(b) (5 points) Fill in the base case for a correct implementation.

(c) (10 points) Show the recurrence relation, and give the worst-case running time.

Hint: Create a small example array, with say 6 elements, and see what the algorithm does. This will help you with the correctness and the base case.

Answer:

(a) Let S_1, S_2, and S_3 denote the three partitions of the array. These are all place holders/slots for elements in the first-third, second-third, and last-third of the array. When we sort in-place $S_1 \cup S_2$, we can guarantee that all elements in S_2 are larger than those in S_1. Let's denote this as $S_2 > S_1$. Likewise, when we sort $S_2 \cup S_3$, we can then guarantee that $S_3 > S_2$. By transitivity then all elements in S_3 are therefore larger than (or equal to) all elements in both S_1 and S_2. Finally, when we perform the third in-place sort on $S_1 \cup S_2$, we get $S_1 < S_2$. Thus the method is correct.

(b) Base case:
$l = j - i + 1$ # length of array
if $l \leq 1$: Return
if $l \leq 2$ and $A[j] < A[i]$: Swap $A[i]$ and $A[j]$; Return

(c) The recurrence is: $T(n) = 3T(2n/3) + O(1)$ or $T(n) = 3 \cdot T(n/(3/2)) + O(1)$. By master theorem, the time is: $O(n^{\log_{3/2} 3})$ which is greater than $O(n^2)$. This method is not very efficient!