CS:3330 Exam 1, Spring 2018

Tuesday, Feb 20 2018, 6:30 pm to 8:30 pm

---

1. This problem is on understanding the growth rate of functions that represent running times of algorithms and the use of asymptotic notation.

   (a) Take the following list of functions (from natural numbers to natural numbers) and arrange them in ascending order of growth. Thus, if a function g immediately follows f in the list, then $f = O(g)$. Some of these functions in standard form, but others are described in words, or as summations, or as recurrence relations. For every function described in words or as a summation or as a recurrence, write it in standard form first before placing it in the sorted list of functions. Show your work in order to receive partial credit.

   (i) $2^{3 \log_2 n}$

   (ii) The solution to the recurrence $T(n) = T(n-1) + n$.

   (iii) The running time of the naive primality testing algorithm on an input that is n bits large. Recall that the naive primality testing algorithm considers all potential factors from 2 though the square root of the input number.

   (iv) $(\log_2 n)^2 \cdot \sum_{i=1}^{n} \Theta(1/2^i)$

   (v) $n^{2.5}/(\log_2 n)^4$

   (vi) The solution to the recurrence $T(n) = T(n/4) + \Theta(1)$.

   (b) For each statement below, write down if it is `True` or `False`. Provide a 1-2 sentence justification for your answer.

   (i) $100n^3 + 10n^2 + 15 = \Theta(n^3 \log_2 n)$.

   (ii) We prefer an algorithm running in $\Theta(\sqrt{2^n})$ time relative to an algorithm running in $\Theta(4^{\log_2 n})$ because the first algorithm is more efficient.

   (iii) The solution to the recurrence $T(n) = T(4n/9) + T(5n/9) + n$ is $\Theta(n \log_2 n)$.

   (iv) The recurrence relation that describes the running time of the binary search algorithm is $T(n) = 2T(n/2) + O(1)$.

2. Write down the worst case running time of each of the following code fragments. Use the Θ notation to express your answers and show your work to receive partial credit. For all three code fragments, you can assume that all comparisons, assignments, and arithmetic operations take $O(1)$ time each.

   (a) Express your answer as a function of n.

```
function printHello(n)
    for i ← 1 to n do
        B ← 1
        while (B < n) do
            print("hello")
            B ← 3 * B
```

   (b) Express your answer as a function of n and k.

```
    for i ← 1 to n do
        j ← 1
        while j < √n do
            print("hello")
            j ← j + 2 · k
```

(c) The arguments to this function are an element k and an array L of length n.

```
function fastScan(k, L[1..n])
    left ← 1; right ← n
    while left ≤ right do
        mid ← (left + right)/4
        if L[mid] = k then
            return mid
        else if L[mid] < k then
            left ← mid + 1
        else if L[mid] > k then
            right ← mid − 1

    return 0
```

3. Let us define a problem called ModuloFactorial as follows:

   ModuloFactorial
   Input: Positive integers n and p.
   Output: $n!$ mod p.

   Recall that $n! = n \times (n-1) \times \cdots \times 2 \cdot 1$. So if the input is $n = 4$ and $p = 7$, then the output should be 3 (check this!). Here is an algorithm that solves this problem.

```
function modFact(n, p)
    m ← 1
    for i ← 2 to n do
        m ← m * i
    return m mod p
```

   Suppose that p is guaranteed to be small and it fits in a single word of memory. (If you want to be concrete, you can assume that 1 word equals 32 bits.)

   (a) What is the input size for the problem ModuloFactorial?

   (b) How many bits does it take to represent $n!$? Express your answer in Θ notation. (**Hint:** Use Stirling's approximation: $\ln(n!) = n \ln n - n + O(\ln n)$.)

   (c) What is the running time of the `modFact` function? Express your answer using the Θ notation, first as a function of n and then as a function of the input size you came up with in (1).

   (d) Would you say this is an *efficient* algorithm? (Yes/No)

   (e) There is a property of modular arithmetic we could use to make the algorithm more efficient. Suggest a simple modification to the `modFact` function, that (based on a property of modular arithmetic) would make the algorithm more efficient.

   (f) Would you say that your algorithm from (e) is *efficient*? Answer Yes/No and justify your answer in 1-2 sentences.

4. Suppose you have k sorted arrays, each with n elements and you want to combine these into a single sorted array of kn elements.

   (a) Here's one strategy: Using the `merge` procedure (that we discussed in class), merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity if this algorithm in terms of n and k? (Recall that merging two sorted arrays with x elements in one and y elements in the second, takes $\Theta(x + y)$ time.)

(b) Here is an idea that leads to a more efficient "divide-and-conquer" algorithm for the problem.

- **Divide step:** Arbitrarily partition the collection of arrays into two groups, each group containing $k/2$ arrays.
- **Conquer step:** Make two recursive calls to merge all the arrays within each group.
- **Combine step:** Merge two sorted arrays, each of size $nk/2$.

Write a recurrence relation for the running time of this "divide-and-conquer" algorithm. Note that the total number of elements is nk and so you will writing a recurrence for $T(nk)$.

(c) Solve this recurrence by unrolling it. Note that the base case of this "divide-and-conquer" algorithm is the situation in which we have just two arrays of size n each and we merge them. So $T(2n) = \Theta(n)$ is the running time of the algorithm in the base case.

5. An array $A[1..n]$ is said to have a *majority* element if more than half the elements of the array are the same. For example, the array $A = [3, 2, 7, 3, 3, 1, 3, 3]$ has a majority element and it is 3. The MAJORITY problem is the following:

MAJORITY
INPUT: An array $A[1..n]$
OUTPUT: If A has a majority element, then output that element. Otherwise, output "no majority element."

Here is a randomized algorithm that attempts to solve MAJORITY.

```
function MAJORITY(A[1..n])
    k ← 3
    for i ← 1 to k do
        p ← random(1, n)
        count ← 0
        for j ← 1 to n do
            if A[j] = A[p] then
                count ← count + 1
        if count > n/2 then
            return A[p]
    return "no majority element"
```

(a) What is the running time of this algorithm? You can assume that all comparisons, assignments, and arithmetic operations run in $O(1)$ time each.

(b) Describe in 1-2 sentences, the situation in which this algorithm returns an incorrect answer.

(c) What is the probability that the algorithm returns an incorrect answer?

(d) We would like the error probability of this algorithm to be less than 1%. What (minor) modification would you make to the algorithm to ensure this.
