CS 3510: Design & Analysis of Algorithms (Sections A & B)

Practice Quiz 1 Solutions

Professors Dana Randall and Gerandy Brito

1.) (20 points total) For each of the following, **select only one option** that most accurately describes the relationship between $f(n)$ and $g(n)$.

(a.) (5 points) $f(n) = n^3 + 2n$, $g(n) = 12n^2 + 24\sqrt{n}$

- ◯ **A.** $f(n) = \mathcal{O}(g(n))$
- ◯ **B.** $g(n) = \mathcal{O}(f(n))$
- ◯ **C.** $f(n) = \mathcal{O}(g(n))$ **and** $g(n) = \mathcal{O}(f(n))$

Solution: B. $g(n) = \mathcal{O}(f(n))$. $f(n)$ has an n^3 term, which grows strictly faster than both the n^2 and $n^{0.5}$ terms in $g(n)$.

(b.) (5 points) $f(n) = (\log n)^2$, $g(n) = (\log n) + \sqrt{n}$

- ◯ **A.** $f(n) = \mathcal{O}(g(n))$
- ◯ **B.** $g(n) = \mathcal{O}(f(n))$
- ◯ **C.** $f(n) = \mathcal{O}(g(n))$ **and** $g(n) = \mathcal{O}(f(n))$

Solution: A. $f(n) = \mathcal{O}(g(n))$. $f(n)$ is a power of $(\log n)$, which grows slower than any power of n, and $g(n)$ has an $n^{0.5}$ term.

(c.) (5 points) $f(n) = 2\log_5(3^n)$, $g(n) = n + 4n^{0.2}$

- ◯ **A.** $f(n) = \mathcal{O}(g(n))$
- ◯ **B.** $g(n) = \mathcal{O}(f(n))$
- ◯ **C.** $f(n) = \mathcal{O}(g(n))$ **and** $g(n) = \mathcal{O}(f(n))$

Solution: C. $f(n) = \mathcal{O}(g(n))$ **and** $g(n) = \mathcal{O}(f(n))$. This requires a bit of manipulation: by log properties, $f(n) = 2 \cdot n \cdot \log_5(3)$, which is a constant times n, and in $g(n)$ the $n^{0.2}$ term is dominated by n, so these both grow like $\mathcal{O}(n)$.

(d.) (5 points) $f(n) = 8^{\log_7 n}$, $g(n) = n \log n$

- ◯ **A.** $f(n) = \mathcal{O}(g(n))$
- ◯ **B.** $g(n) = \mathcal{O}(f(n))$
- ◯ **C.** $f(n) = \mathcal{O}(g(n))$ **and** $g(n) = \mathcal{O}(f(n))$

Solution: B. $g(n) = \mathcal{O}(f(n))$. By log properties, $f(n) = \left(7^{\log_7 8}\right)^{\log_7 n} = n^{\log_7 8}$, and since $8 > 7$, we know $\log_7 8 > 1$. Since $g(n) = n \log n$, it grows faster than n, but slower than n^c for any $c > 1$.

2.) (20 points) Suppose you've discovered two schemes you could use to design a divide and conquer algorithm for a problem.

(A.) Divide a problem of size n into 4 subproblems of size $n/8$, where the cost of combining the subproblem solutions is $\mathcal{O}(\sqrt{n})$.

(B.) Divide a problem of size n into 3 subproblems of size $n/3$, where the cost of combining the subproblem solutions is $\mathcal{O}(n)$.

Which scheme do you prefer and why? Justify your answer.

Solution: I prefer scheme **A**.

In scheme A, our recurrence is $T_A(n) = 4T_A(n/8) + \mathcal{O}(\sqrt{n})$, which can be solved with the master theorem with $a = 4$, $b = 8$, $d = 0.5$, using the $a > b^d$ case, since $4 > 8^{0.5}$. This gives $T_A(n) = \mathcal{O}(n^{\log_8 4}) = \mathcal{O}(n^{2/3})$.

In scheme B, our recurrence is $T_B(n) = 3T_B(n/3) + \mathcal{O}(n)$, which can be solved with the master theorem with $a = 3$, $b = 3$, $d = 1$, using the $a = b^d$ case, since $3 = 3^1$. This gives $T_B(n) = \mathcal{O}(n \log n)$.

Since $n^{2/3}$ grows asymptotically slower than $n \log n$, I prefer scheme A, as past a certain n, using it will result in a faster algorithm than scheme B.

3.) (25 points total) Suppose we have the following algorithm.

Algorithm 1 Neil's "Fun" algorithm

```
1:  procedure FUN(n)
2:      if n = 1 then
3:          PRINT("yay!")
4:          return
5:      for i ← 1 to n do
6:          for j ← 1 to n do
7:              PRINT("nay!")
8:      FUN(n/2)
9:      FUN(n/2)
10:     return
```

(a.) (10 points) Give a recurrence relation for $T(n)$, the number of lines printed by FUN as a function of its input n.

Solution: $T(n) = 2T(n/2) + n^2$. Note that this holds even if $n = 1$, since $1^2 = 1$, which is the number of lines printed when FUN(1) is called.

(b.) (10 points) Solve the recurrence relation. Give your answer in Big-O notation. Show all work, including values of a, b, and d if you use the master theorem.

Solution: $T(n) = \mathcal{O}(n^2)$. Using master theorem with $a = 2$, $b = 2$, $d = 2$, we're in the $a < b^d$ case, since $2 < 2^2 = 4$. Thus, $T(n) = \mathcal{O}(n^2)$.

(c.) (5 points) Of these printed lines, how many are "yay!"s and how many are "nay!"s?

Solution: $\mathcal{O}(n^2)$ lines are "nay!"s, and $\mathcal{O}(n)$ lines are "yay!"s. The recurrence for "nay!"s alone is the same as in part a.), only the base case is different (we could consider $T(2) = 4$ our base case instead), but this doesn't change the Big-O of the number of lines.

For "yay!"s, our recurrence becomes $Y(n) = 2Y(n/2)$, with the base case of $Y(1) = 1$. Expanding, we can see that $Y(n) = 2^k Y(n/2^k) = 2^{\log_2 n} Y(1) = n$. Thus, $Y(n) = \mathcal{O}(n)$.

If we wanted to be precise (not required), expanding and careful analysis would reveal that we have exactly $2n(n-1)$ "nay!"s and n "yay!"s.

4.) (35 points total) You are given a **sorted** array of **distinct** integers $A = [a_1, a_2, \ldots, a_n]$, which may contain negative values, and you want to find out whether there is an index i such that $A[i] = i$. In this problem, the array A is 1-indexed.

For example, in the array $A = [-3, -2, 0, 3, 4, 6, 9, 11]$, you should return 6, since $A[6] = 6$.

(a.) (20 points) Give a divide-and-conquer algorithm for this problem with running time strictly faster than linear (e.g., $\mathcal{O}(\sqrt{n})$ or $\mathcal{O}(\log n)$). Assume that you can look up the value of $A[i]$ in $\mathcal{O}(1)$ time. Explain your algorithm **and** give pseudocode.

Solution: We propose a variant of binary search, where at each step we look at the middle element and its index. If it matches its index, we've found an i such that $A[i] = i$, and return our answer. If $A[i] > i$, we continue searching in the left half of the array, and if $A[i] < i$, we continue searching in the right half of the array.

In the following pseudocode, we use -1 to represent that no such index exists. Index i is inclusive, and index j is exclusive, so the initial call is FINDFIXEDPOINT(A, 0, n).

```
1: procedure FindFixedPoint(A, i, j)
2:     if i ≥ j then
3:         return −1
4:     mid ← (i + j)/2
5:     if A[mid] = mid then
6:         return mid
7:     else if A[mid] > mid then
8:         return FindFixedPoint(A, i, mid)
```

```
    else
        return FindFixedPoint(A, mid + 1, j)
```

(b.) (10 points) What is the running time of your algorithm? Show all your work.

Solution: At each call, we do $\mathcal{O}(1)$ steps in comparing $A[\texttt{mid}]$ against $\texttt{mid}$, and then we make a recursive call on an input of size $n/2$ (since we aren't explicitly slicing A here, the "size of an input" here refers to $j - i$).

This gives a recurrence of $T(n) = T(n/2) + \mathcal{O}(1)$. Using master theorem with $a = 1$, $b = 2$, $d = 0$, we're in the $a = b^d$ case, since $1 = 2^0 = 1$. Thus, $T(n) = \mathcal{O}(\log n)$.

(c.) (5 points) Briefly argue why your algorithm is correct.

Solution: The key here is that the given array is sorted and consists of distinct integers, which means that $A[i + 1] \geq A[i] + 1$, so moving forward exactly one index in A must increase the values by *at least* 1, and likewise for moving backwards one index.

If $A[i] > i$, then using the inequality noted above, we have $A[i + 1] \geq A[i] + 1 > i + 1$. Repeating this argument (formally, by induction), we have $A[i + k] > i + k$, which tells us that no indices $j > i$ can contain a solution, allowing us to "throw away" the right half of the input and recurse on the left half.

Likewise, if $A[i] < i$, then there can be no solution left of index i, allowing us to throw away the left half of the input.

Since at each step of the algorithm we either find an answer or eliminate parts of the array that cannot contain an answer, we will always find the answer, if it exists.