CS 331: Algorithms and Complexity (Spring 2024)
Unique Number: 50930, 50935 50940, 50945

Assignment 1

Due on Thursday, 25 January, by 11.59pm

Problem 1

(a) (3 points)

We need to find whether $\sqrt{n}$ *or* $\log(n^2)$ *grows faster.*
Solve: $\lim_{n\to\infty} \frac{\sqrt{n}}{\log(n^2)} = \lim_{n\to\infty} \frac{d}{dn}\frac{\sqrt{n}}{\log(n^2)} = \lim_{n\to\infty} \frac{1}{2\sqrt{n}}/\frac{2n}{n^2}$ *(Constants don't matter), this approaches* ∞ *as* n *increases,* $\therefore$
`O(f`$_4$`(n)) = `$\sqrt{n}$
`O(f`$_1$`(n)) = n`
`O(f`$_2$`(n)) = n`$\log\log$`n`
Since $\log\log n$ *grows slower than* n*, then* $f_2(n)$ *grows slower than* $f_3(n)$*.*
`O(f`$_3$`(n)) = n`^{2}
By the growth rate diagram, polynomial functions grow slower than exponential functions.
`O(f`$_6$`(n)) = `$2^{\log n}$
Since $\log n$ *grows slower than* $\sqrt{n}$*, then* $f_6(n)$ *grows slower than* f_5 *(n).*
`O(f`$_5$`(n)) = `$2^{\sqrt{n}}$
$\therefore$ *the order is:* $f_4(n), f_1(n), f_2(n), f_3(n), f_6(n), f_5(n)$

(b) (3 points)

Given $a_1 = 1$*,* $a_2 = 8$*, and* $a_n = a_{n-1} + 2a_{n-2}$ *when* $n \geq 3$*, show* $a_n = 3 \cdot 2^{n-1} + 2(-1)^n$ *for all* $n \in \mathbb{N}$*.*
`Define our predicate as P(n):  a`$_n$` = 3 · 2`$^{n-1}$` + 2(-1)`^{n}
Base Cases:
We use two base cases since the recurrence relation is defined using the previous two terms.

Proof. P(1): Show $a_1 = 3 \cdot 2^{1-1} + 2(-1)^1 = 1$:

$3 \cdot 2^{1-1} + 2(-1)^1$	Arithmetic
$= 3 \cdot 2^0 + 2(-1)$	Simplify
$= 3 \cdot 1 + -2$	Simplify
$= 1$	Simplify

$\therefore$ P(1) is true.

P(2): Show $a_2 = 3 \cdot 2^{2-1} + 2(-1)^2 = 8$:

$3 \cdot 2^{2-1} + 2(-1)^2$	Arithmetic
$= 3 \cdot 2^1 + 2(1)$	Simplify
$= 3 \cdot 2 + 2$	Simplify
$= 8$	Simplify

∴ P(2) is true.
∴, the base cases are true. □

Inductive Hypothesis: *Assume for n = k, P(1), P(2), ..., P(k) are all true.*

Inductive Step:
Show P(k + 1), aka $a_{k+1} = 3 \cdot 2^k + 2(-1)^{k+1}$.

Proof.

$a_{k+1} = a_k + 2a_{k-1}$	Inductive Hypothesis
$= (3 \cdot 2^{k-1} + 2(-1)^k) + 2a_{k-1}$	Factor out 2
$= 2(3 \cdot 2^{k-2} + 2(-1)^{k-1}) + 2(3 \cdot 2^{k-2} + 2(-1)^{k-1})$	Combine like terms
$= 4(3 \cdot 2^{k-2} + 2(-1)^{k-1})$	$4 = 2^2$
$= 3 \cdot 2^k + 2(-1)^{k+1}$	

∴, we have shown that P(k + 1) is true, and by induction, P(n) is true for all n ∈ ℕ. □

(c) (4 points)
`Algorithm 1(Alice)`

Proof. We need to swap the lines `Report average as sum / count;` and `Increase count by 1;` as in the first iteration, we will divide by 0, which is not the correct average. □

`Algorithm 2(Bob)`

Proof. First, we show termination.
Since each iteration we read one integer, the set of integers needed to be read is decreased by one each iteration, and since the set is finite, the algorithm will terminate.
Next, we show correctness.
Assume we have to read n integers from the input stream, which we'll call S.
Base Case: n = 1
average is updated to $\frac{(0 \cdot 0 + S_1)}{0+1} = S_1$, which is the correct average, and count is incremented by one which results in count=1.
Then we have average=S_1 and count=1, which is correct.
Inductive Hypothesis: Assume for n = k, average=$\frac{\sum_{i=1}^{k} S_i}{count}$ and count=k.
Inductive Step: Show for n = k + 1, average=$\frac{\sum_{i=1}^{k+1} S_i}{count+1}$ and count=k + 1.
In the loop, we read the integer S_{k+1} and update average to $\frac{(average*count)+S_{k+1}}{count+1}$.
We have to show that $\sum_{i=1}^{k+1} S_i$ = (average * count) + S_{k+1}.
We know that average=$\frac{\sum_{i=1}^{k} S_i}{count}$, ∴ average * count = $\sum_{i=1}^{k} S_i$.
Adding S_{k+1}, we get $\sum_{i=1}^{k+1} S_i$ = (average * count) + S_{k+1}.
∴, we can express $\frac{(average*count)+S_{k+1}}{count+1}$ as $\frac{\sum_{i=1}^{k+1} S_i}{count+1}$, which is the correct average.
count is then incremented by one, which results in count=k+1.

Problem 1 continued on next page...

∴, we have shown that the algorithm is correct.

□

Problem 2

(6 points)
For this problem, I would use a modified binary search, since I know that the array is sorted in ascending order with the target being less than the "max" values appended to the array.

```
// Finding bounds
int low = 0, high = 1;
while(E[high] < x) {
    low = high;
    high *= 2;
}
// Binary search
while (low <= high) {
    int mid = low + (high - low) / 2; // or (low + high) >>> 1
    if (E[mid] == x) return mid;
    else if (E[mid] < x) low = mid + 1;
    else high = mid - 1;
}
```

Proof. The first while loop finds the bounds of the search space, which is $O(\log n)$ since we are exponentially increasing the search space.
The second while loop is a binary search, which is $O(\log n)$.
The first part of the algorithm terminates since the high value is doubled each iteration, and won't exceed the length of the array.
The second part of the algorithm terminates since the range of the search is halved each iteration.
Correctness: We are trying to find x < max in E.
Since the array is constructed as $[\mathbf{sorted}|\mathbf{max}_{size-n}]$, and max is greater than all elements of the array up to n, then there are 3 cases in each ith iteration:
Case 1: x < E[i], then since the array is sorted, x is the lower half of the search space.
Case 2: x > E[i], then since the array is sorted, x is the upper half of the search space.
Case 3: x = E[i], then we have found the index of x.
∴, we half the search space each iteration, thus the algorithm is $O(\log n)$. □

Problem 3

Your task is to do the following:

i **(7 points)**

```
Initialize all TAs to be unassigned
Store all TA preferences in a sorted list for each course and the
number of TAs needed
while (course needs TA) {
    c = a course that needs a TA
    ta = first TA in c's preference list that c hasn't tried to assign
    if (ta is unqualified for c) {
        c rejects ta
    } else if (ta is unassigned) {
        assign ta to c
    } else if (ta prefers c to ta's current course) {
        c' = ta's current course
        unassign ta from c'
        assign ta to c
    }
    else {
        ta rejects c
    }
}
```

ii **(7 points)**

Proof. First, we show termination.
The sorted list of applicants TAs for each course is strictly decreasing each iteration, $\therefore$ the algorithm will terminate.
Next, we show correctness.
Observation 1: Courses are assigned TAs in order of preference.
Observation 2: Once a TA is assigned to a course, they will not be unassigned, only reassigned to a different course.
Claim 1: All available TA spots are filled unless unqualified.
Proof: Assume by contradiction that there is an available TA spot upon termination, in other words, there is a course c_0 that needs a TA and there is one that's qualified and unassigned.
Since there are more TAs than courses, there must be a TA t_0 that is unassigned .
Since t_0 is unassigned, then c_0 must have rejected t_0, however, since t_0 is unassigned, then t_0 must be unqualified for c_0.
$\therefore$ we have a contradiction, and all available TA spots are filled unless unqualified.
Claim 2: All TAs are assigned to a course in a stable marriage.
Proof: Assume by contradiction that there is a TA t_0 that is assigned to course c_1 and TA t_1 that is assigned to course c_0 such that c_0 prefers t_0 over t_1 and t_0 prefers c_0 over c_1.
Case 1: t_0 was never assigned to c_0.
Since c_0 prefers t_0 over t_1 and c_0 is assigned TAs based on preference, then t_0 must have been assigned to a course that is higher on t_0ś preference list than c_0, which is a

contradiction.
Case 2: t_0 was assigned to c_0 and then reassigned.
This means that t_0 rejected c_0 and was assigned to a course that is higher on their preference list than c_0.
However, since c_1 is lower on t_0ś preference list than c_0, this is a contradiction.
$\therefore$, we have shown that all TAs are assigned to a course in a stable marriage.
$\therefore$, we have shown that the algorithm is totally correct.

$\square$