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
$= 1$	

$\therefore$ 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
$= 8$	

$\therefore$ P(2) is true.

$\therefore$, the base cases are true. $\square$

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

$\therefore$, we have shown that P(k + 1) is true, and by induction, P(n) is true for all $n \in \mathbb{N}$. $\square$

(c) (4 points)

`Algorithm 1(Alice)`

Proof. We need to swap the lines

```
Report average as sum / count;
Increase count by 1;
```

as in the first iteration, we will divide by 0, which is not the correct average. $\square$

`Algorithm 2(Bob)`

Proof. Bob's algorithm is correct, shown using loop invariants.

Let S be the stream of integers indexed starting at 0.

Inductive Hypothesis: Assume at step k, `average`=avg of the first `count` elements where `count`=k.

Initialization: Before the first iteration, average=0 and count=0, which is correct, as there are no integers to calculate the average of.

Maintenance: Assume the inductive hypothesis is true at the beginning of the kth iteration.

$\therefore$, `average`=avg of the first `count` elements where `count`=k.

We read in integer S_k.

Next, we focus on the line `average = (average * count + S`$_k$`) / (count + 1)`.

Since `average`=$\frac{sum}{count}$, after multiplying by `count` and adding S_k, the expression yields the sum of the first k+1 elements.

Then, we divide by `count + 1` to get the average of the first k+1 elements.

$\therefore$, `average`=avg of the first `count+1` elements.

`count` is then incremented by one, which results in `count`=k+1.

$\therefore$, the induction hypothesis is true at the beginning of iteration n = k + 1 .

Problem 1 continued on next page...

Termination: The loop terminates when there are no more integers in the stream, in other words, when k=n, the length of the stream.
$\therefore$, `average`=avg of the first `count` elements where `count`=n, which is the average of all the integers in the stream.
$\therefore$, 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;
    if (high outOfBounds) {
        break;
    }
}
// Binary search
while (low <= high) {
    int mid = low + (high - low) / 2; // or (low + high) >>> 1
    if (mid outOfBounds) high = mid - 1;
    else if (E[mid] == x) return mid;
    else if (E[mid] < x) low = mid + 1;
    else high = mid - 1;
}
```

Proof. **Inductive Hypothesis:** Assume at step k, `low`(2^{k-1}) is the lower bound of the search space and `high`(2^k) is the upper bound of the search space where `x` is at index $\geq$ low.
Initialization: Before the first iteration, low=0 and high=1, and since all elements of E have index ≥ 0, then the inductive hypothesis is true.
The first while loop finds the bounds of the search space, which is $O(\log n)$ since we are exponentially increasing the search space.
Maintenance: Assume the inductive hypothesis is true at the beginning of the kth iteration. if `E[high] < x`, then we double the search space while ignoring the current one, as we know that x is greater than the current high value.
We set `low` to `high` and `high` to `high` * 2, which means that `low` $= 2^k$ and `high` $= 2^{k+1}$.
Since the loop only ran because E[high_{old}] < x, and `low` was set to `high`, then we know that E[low] < x.
$\therefore$, `x` is at index $\geq$ low.

$\therefore$, the induction hypothesis is true at the beginning of iteration n = k + 1.
Termination: The loop terminates when E[high] $\geq$ x, in other words, when x is in the current search space.
$\therefore$, x $\geq$ low.
$\therefore$, we have shown that the algorithm is correct.
Since the search space is doubled every time until x is in the search space or the search space exceeds the size of the array, then the algorithm is $O(\log n)$.
The second while loop is a modified binary search, which is $O(\log n)$.
We are trying to find x < max in E[low:high].
Case 1: mid > |E| $\rightarrow$ x is the lower half of the search space.
Case 2: x < E[mid] $\rightarrow$ x is the lower half of the search space.
Case 3: x > E[mid] $\rightarrow$ x is the upper half of the search space.
Case 4: x = E[mid] $\rightarrow$ return mid.
$\therefore$, since the array is increasing, then if the element at mid is greater than x, we can ignore all elements after it too, and conversely, if the element at mid is less than x, we can ignore all elements before it.
$\therefore$, we half the search space each iteration, thus the algorithm is $O(\log n)$. $\square$

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. 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.

□