Sean
Blanchard
10/5/20
comp 354

MIDTERM 1

1-1 a) Determine if each of the following algorithms is fully correct, and prove that your answer is correct.

A(x)

IN: $x \in \mathbb{R}, x \geq 1$

1 $x' = x$
2 $p = 0$
3 while $x' > 1$ do
4 | $p = p + 1$
| $x' = x'/2$
return p
OUT: $p = \log_2 x$

$x = 1$ $p = 0$ $\log_2(1) = 0$ ✓
$x = 2$ $p = 1$ $\log_2(2) = 1$ ✓
$x = 3$ $p = 2$ $\log_2(3) = 1.584$ ✗

After plugging in random values, you can see above that once $x = 3$, this will throw our $x' = x/2$ out of whack, so on uneven x values it clearly shows that $\log_2(x) \neq p$

This algorithm should hold true if x was strictly even numbers.

$x = 2^{-2} = .25$

b) B(x)

IN: $x \in \mathbb{R}$, $x \geq 1$

while $x > 1$ do

$x = x/2$;

if $x == 1$

return true;

else

return false;

$x = 2^{-1} = .5$

$x = 2^0 = 1$

true;

$x = 2^1 = 2$ $2/2 = 1$

true;

$x = 2^2 = 4$ $4/2 = 2$ $2/2 = 1$

true;

OUT: true if $x = 2^n$ for some integer n (initially); false otherwise

If look above, it will always return true if $x = 2^n$ for some integer n (initially) If and only if n is a positive integer. This disproves this algorithm as n can be negative.

1-2 Data Structure Selection

a) The Priority Queue must be able to do the following in O(log n) time:

- insert (key, value)
- extract_min
- decrease_key(old_key, value, new_key)

What data structure should be implemented to use Priority Queue ADT in O(log n) time?

We would want to use a Heap Sort because it uses Binary Heap to sort an array. We can efficiently implement Priority Queues using Binary Heap it supports insert(), delete(), and extract max(), decreasekey(), in O(log n) time. Min heap would be used as a priority queue to get the minimum distance vertex from set of not yet included vertices.

b) • Each time a player wins a game, their username is added to the end of a list
• Each time a player enters the home screen (number of wins calculated), this list of usernames is traversed and the number of occurrences of the player's username is counted.

The first issue I see is that when you place an item at the rear of the list, there is no sorting that was done before this action. Thus, once you win, it has to add you to list (low time); c) but then needs a sorting algorithm to go through the list. There are a few ways I would look to tackle this.

I think I'd use some sort of dictionary collection to store users and number of wins. The main reason I was thinking of using a dictionary collection is we can slice our lists to sequential chunks, thus making searching for the user easier and faster. Once a user wins a game, we will search for user, update, and display.

WRONG/SLOW // user friendly though

Id Try to use a Self Organizing List.
A self organizing list that re-organizes or re-arranges items for better performance.
Id implement the Idea of the more the user plays the game Id place those wins closer to the head. (Ie. more frequently accessed item closer to front). Id store a Count Method that would store each node with the number of wins, because the more you win the more likely you are to play the game. The worst case time complexity would be O(n), and drastically increased with helper functions.

1-3 Sorting lower bounds:

a) Why does the comparison model must take $\Omega(n \log n)$ time in sorting?

The easiest way for me to explain this is with a example

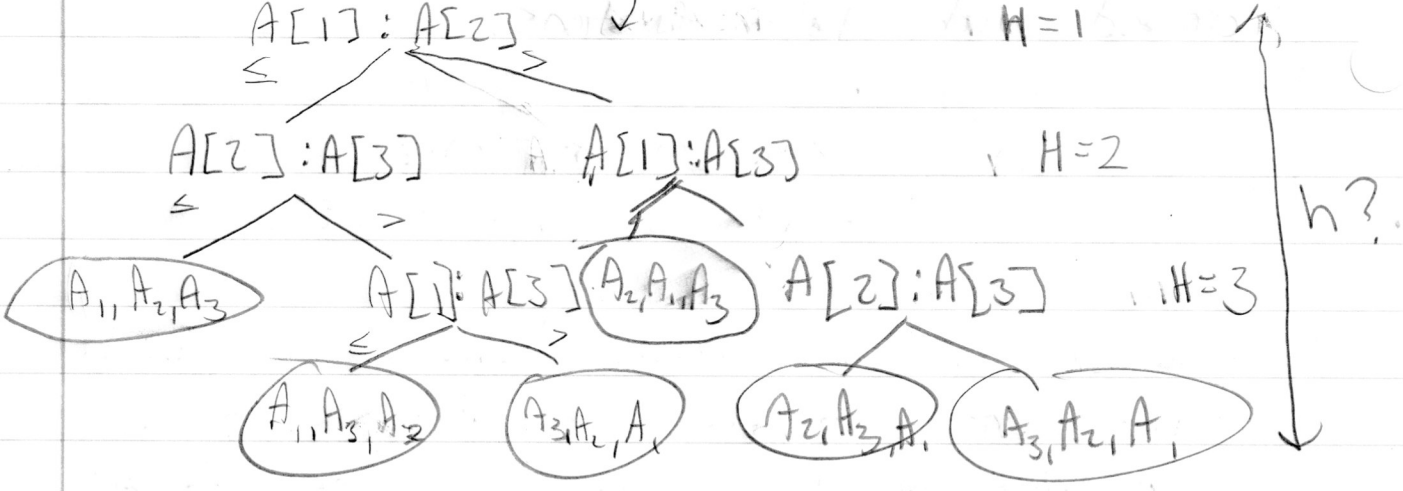
small example: we have an array A: w/ 3 items:

A[1] A[2] A[3]

How can I sort array w/ comparisons?

A <= B OR A > B 2 Forks w/ 2 possibilities

[_ , _ , _] ★ I'm interested in height of tree!



h : 3

input : 3

★ Circled output are Leaves → 6

for a tree w/ max height 3, the maximum number of leaves are 8.

- With the above information we can make the generalized formula,

of Leaves

Min = 6 = ! input

Max = 8 = 2^h

Thus, $2^h \geq n!$

We want to find $n!$ or the least work done since it is the lower bound.

MATH! take log of both sides

$h \lg(2) \geq \lg(n!)$

↳ $\lg(1) + \lg(2) + \ldots + \lg(n)$

$\sum_{i=2}^{n} \lg(i)$

$\int_{1}^{n} \lg(x)\,dx$

$\frac{1}{\ln(2)} [n \ln(n) - n + 1]$

$n \lg(n) - n \lg(e) + \lg(e)$

This shows the dominating term, and we solved for the "least" the height of this tree could be and we solved for is $n \log(n)$, and this is where the lower bound came from.

This is why sorting MUST take $\Omega(n \log(n))$ time. ▨

b) RadixSort can sort integers of bounded size in linear time. Does this contradict the result discussed in the previous part? why or why not.

No, it plays no role, because radix sort is not comparison based, it is not bounded by $\Omega(n \log(n))$ for running time.

The idea behind radix sorting is that we view the keys to be sorted sequences of numerals consisting of digits in some finite Radix. An example is a ASCII character string can be thought of as base-256 numerals.

1-4 Dictionary Management

a) Every time an element is inserted or deleted, resize the dictionary so that $m = 2n$

n = elements

m = size of dictionary

$n=1$ $m=2$

$n=2$ $m=4$

$n=3$ $m=6$

$n=4$ $m=8$

This will always gaurantee there is a open space in dictionary but cost alot to build if keep inserting and deleting. It will still remain $O(1)$ insert and delete amortized complexity, but there are advantages and disadvantages of having a huge dictionary (mostly bad).

b) After inserting, if $n > m/2$, double size of dictionary, After deleting, if $n \leq m/4$, halve the size of dictionary.

$n=1$ $m=2$

$n=2$ $m=4$

$n=3$ $m=8$

$n=4$ $m=8$

$n=5$ $m=16$

$n=6$ $m=16$

$n=7$ $m=16$

$n=8$ $m=16$

$n=9$ $m=32$

Delete

$n=9$ $m=32$

$n=8$ $m=16$

$n=7$ $m=8$

$n=6$ $m=8$

$n=5$ $m=8$

$n=4$ $m=8$

$n=3$ $m=8$

$n=2$ $m=4$

This algorithm is more smart as it saves more space in the dictionary. It still has O(1) amortized cost for deletion or insertion.

1-5 Optimization Question

Series of steps taken to optimize a slow implementation

I think the first step I would look at is which are my biggest steps taking place in my implementation. efficiency on smaller scale problems are generally not as important, as they lead to negligible losses. Some algorithms are better for small problems. I think another thing I would look for is to avoid repeating work, like removing unnecessary passes when sorting an array. Obviously a big one is to make sure the algorithm is outputting the correct desired output. We also need to pick the correct algorithm for specific events.