Data Structures and Algorithms Homework 2

Due Wednesday Sept 11; Joseph Sepich (jps6444)

1 Problem 1 Sorted Array

Given a sorted arrayAofn(possibly negative) distinct integers, you want to find out whether there is anin-dexifor whichA[i] =i. Give a divide-and-conquer algorithm that runs in timeO(logn). Provide only themain idea and the runtime analysis.

There are two very important parts to this problem. The first important part of the problem is the fact tha the array is sorted. The second important part is knowing that you are looking for a integer contained in the array that is the same value as the index. From here we can use a binary search like algorithm. If we look at the middle integer and the statement holds, A[i] == i, then we can return true. If the middle index is larger than the value contained in it, you know that none of the values below that index will work, because the index has to be larger than it. If the middle index is smaller than the value contained in the array at the position, then you know that none of the value above that index will work, because the index would have to be smaller then it. These two conditions also rely on the fact that A only contains integer values and not doubles or floating point values.

Knowing these three conditions you can recursively call one subproblem of size n/2 with constant time at each step eventually reaching a number that fulfills the requirement or a problem with size 1, that doesn't fulfill the requirement where you return false. (Accessing a value at an index is constant time). Using the master theorem we know this running time would be O(log(n)), since a is 1, b is 2, and d is 0.

Collaborators: GeeksForGeeks CountingSort page. https://www.geeksforgeeks.org/counting-sort/

2 Problem 2 Squaring vs Multiplying: Matrices

The square of a matrix A is its product with itself, AA.

2.1 Part a

Show that five multiplications are sufficient to compute the square of a 2 x 2 matrix.

Below are the standard calculations:

- AA(1, 1) = A(1,1) x A(1,1) + A(1,2) x A(2,1)
- AA(1, 2) = A(1,1) x A(1,2) + A(1,2) x A(2,2)
- AA(2, 1) = A(2,1) x A(1,1) + A(2,2) x A(2,1)
- AA(2, 2) = A(2,1) x A(1,2) + A(2,2) x A(2,2)

You can improve the second calculation to be A(1,2) x (A(1,1) + A(2,2)), and the third calculation to be A(2,1) x (A(1,1) + A(2,2)). This makes 2 multiplications in the first row, 2 multiplications in the last row, and 1 in each middle row. This is six, but A(1,2) x A(2,1) = A(2,1) x A(1,2), so the same value is use in the first and last row. This gets rid of 1 of the six multiplications for a total of five.

2.2 Part b

Using a divide and conquer approach for this algorithm does not work. While Strassen has a good solution for multipling matrices, our solution involves squaring a matrix. This implies that you are multiplygin A by A (or two identical matrices on each step), which will not be the case unless you have a case where all entries are the same value. Since matrix multiplcation is not commutative like it is for numbers, this algorithm would then not function properly in a divide and conquer scenario.

2.3 Part c

We want to use a squaring algorithm runningin O(n^{c}) time to calculte the product of two matricies. The product of A and B would be the matrix AB, so if we can find a matrix, that if squared contains AB, then we can calculate AB using this running time.

Let's work backwards and start with the following matrix:

$$\begin{pmatrix} AB & x \\ y & z \end{pmatrix}$$

If we look back at part a, we know that the top left and bottom right calculations would be a sum of the product of the diagonal and either the top left or bottom right squared. If we leave the top left and bottom right as 0 then we can get:

$$\begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix} = \begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2$$

This matrix that is being squared would have to be 2n x 2n in size, since it requires to be the width of A + the width of B wide and the height of A + the height of B tall, which are both 2n. If we plug this into our squaring running time we get $O((2n)^c = O(2^c n^c) = O(n^c)$, since 2^c is a constant that doesn't grow. Therefore this squaring algorithm could help us to multiply two matrices in the same running time. (It would still be affected by the coefficients at smaller numbers, which I am guess is related to the improved algorithms we discussed in lecture that are not used due to large coefficients).

3 Problem 3

For sorting an array of integers you can use a Counting Sort algorithm. This algorithm first takes the count of each distinct element in the input array x. This operation would take linear time O(n) by looping through the array once. After this you would make the counts cumulative by adding the sum of the previous counts, so if you had three 1 values and a single 2 value, the 1 count would be 3 and the 2 count would be 4. This second operation would takes O(m) time, since the distinct values list is only going to be as big as the difference between the largest and smallest value. The final operation is to then insert these values back into an array by inserting them at their count value (in a 1 based index), and then decreasing the count by 1 until you hit a value that has been inserted already. This last operation would take O(n) time, since you must insert n numbers.

These three steps together would then take O(n) + O(m) + O(n) time, which is equavalent to $O(2n + m) = O(n + m)$ time. The bound of $\Omega(nlog(n))$ does not apply in this case, because we are not making any comparisons. The comparisons between numbers in most algorithms are what create the tree of possible outcomes, but in this case, since we do not need to create this comparison tree, there is only going to be linearish time.

4 Problem 4

First of all, we know that we have consecutive integers in the list. Let's define the bit count of each number, since they will have leading zeros. The highest number will be N, with N+1 integers. Converting to binary digits we would have n = 2^k - 1, where k is the number of bits for each integer. This means each integer has ceiling($\log_2$(n+1)) bits. If we had for example k = 3, with the array going from 0 to 7 the integers would look like:

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

As you can see, with all the numbers filled in there are 4 1's in each column and 4 0's in each column. Looking at one bit it would be the same with 1 1 in each column and 1 0 in each column. 2 bits would be the same as well with 2 1's in each column and 2 0's in each column.

We can use this property to search the numbers. We know that for each number, in our example, greater than 4, we should have a 1 in the fist column. Since we go up to 7, we know this should be 7-4 + 1, or 4 total 1's, so if we were missing a 1, then we know to only focus in the first half, and then can use the same method on the second column, but now our subproblm as n/2 integers left, and so on until we reach the last column, which would determine our number.

```
int total = length(A) // value of n
int bitNumber = ceiling(log(n+1)/log2) // number of bits
int[] binaryNumber = list()

Findmissing(A) {
  if (length(A) == 1)
    binaryNumber.Add(A[])
    return
  bits[] zeroCount = list()
  bits[] oneCOunt = list()
  foreach first_bit in A[i]:
    if (first_bit == 0) zeroCount.add(index)
    if (first_bit == 1) oneCount.add(index)
  if zeroCount < total / 2: // not enough zeros
    binaryNumber.Add(0)
    FindMissing(A.filter(index == zeroCount)) // whose number start with zero
  else: // not enough ones
    binaryNumber.Add(1)
    FindMissing(A.Filter(index == oneCount)) // whose number start with one
}
```

This makes our running time T(n) = T(n / 2) + O(n). Using the master theorem on this we know that d = 1 is greater than 0, so the running time for finding this missing integer would be O(n).

5 Problem 5

5.1 Part a

Run DFS at node A. Run through nodes alphabetically.

List the nodes in the order you visit them.

1. A
2. B
3. D
4. E
5. G
6. F
7. C
8. H
9. I

5.2 Part b

List each node with its pre- and post-number. THe numbering starts from 1 and ends at 18.

Using the list from before should make this easy. We know that all intervals should be disjoint or contained within another.

1. A: [1, 12]
2. B: [2, 11]
3. D: [3, 6]
4. E: [4, 5]
5. G: [7, 10]
6. F: [8, 9]
7. C: [13, 18]
8. H: [14, 17]
9. I: [15, 16]

5.3 Part c

1. Edge [A, B]: Tree
2. Edge [B, D]: Tree
3. Edge [D, E]: Tree
4. Edge [B, G]: Tree
5. Edge [G, F]: Tree
6. Edge [C, H]: Tree
7. Edge [H, I]: Tree
8. Edge [A, E]: Forward
9. Edge [E, D]: Back
10. Edge [G, D]: Cross
11. Edge [C, I]: Forward

5.4 Part d

Big O notation means worst case scenario, so we want to prove that a graph with |V| vertices has at most |V|ˆ2 edges. You could look at a complete graph to prove this claim, since a complete graph will have the most possible edges. A complete graph in this case would mean that an adjacency matrix would have all 1's except in the diagonal, so each vertex would be connected to every other vertex, besides itself. If you total these 1 values up in an n by n matrix, you can get rid of the diagonal, which would have n to have nˆ2 - n total edges, but half the matrix is also stating the same information (reflexive). So you would also half this value to get $\frac{n^2-n}{2} = \frac{n(n-1)}{2}$ total edges possible in a complete graph. Since a complete graph would be the worst case considering the number of edges this implies $\frac{n(n-1)}{2} = O(n^2)$.

5.5 Part e

1. Let's start with the most basic graph, a single node. With this kind of graph you would have 0 edges, so the degree would be zero. The sum of all degrees are even.
2. If you add a second node with any number of edges between it, then you have 2 (the number of nodes) times the number of edges, which would always be an even sum.
3. If you carry this on you would always have an even degree number, because each vertex is connected to a side of each edge. This inherently requires the sum to be even. (Each edge has two vertices connected to it).