CSC421 Analysis of Algorithms Assignment #1

Matt Triano

April 12, 2017

1 Problem 1) Nuts and Bolts

1.1 Problem Statement

Given a collection of n nuts and a collection of n bolts, arranged in an increasing order of size, give an $O(n)$ time algorithm to check if there is a *nut* and a *bolt* that have the same size. The sizes of the nuts and bolts are stored in the sorted arrays $NUTS[1..n]$ and $BOLTS[1..n]$, respectively. Your algorithm can stop as soon as it finds a single match (i.e, you do not need to report all matches).

- Operation: Comparison or assignment
- Input Size: n
- Desired Performance: $O(n)$
- Worst Case Scenario: $Threadsize(nut) \neq Threadsize(bolt) \; \forall \binom{nut}{bolt}$

1.2 Solution

```
nut_i = 1
bolt_i = 1
while (nut_i <= n and bolt_i <= n):
        if (nut[nut_i] == bolt[bolt_i]):
            return "A match exists" (and break out of the loop)
    else if (nut[nut_i] < bolt[bolt_i]):
            nut_i = nut_i + 1
    else:
            bolt_i = bolt_i + 1
```

analysis

There are 2 assignments before the loop. $ops_{initial} = 2$

There are 2 comparisons in the condition of the while loop, there will be (at most) 2 comparisons and 1 assignment per loop iteration, and the loop will run at most $2n$ times. $ops_{loop} = 2n * (2 + 2 + 1) = 10n$

So the worst case total is $O(2 + 10n) = O(cn) = O(n)$.

2 Problem 2) the Subset Sum problem

2.1 Problem Statements

- Let $A[1..n]$ be an array of distinct positive integers, and let t be a positive integer.

- **(a):** Assuming that $A[]$ is sorted, show that in $O(n)$ time it can be decided if $A[]$ contains two distinct elements x and y such that $x + y = t$
- **(b):** Use part (a) to show that the following problem, referred to as the 3-Sum problem, can be solved in $O(n^2)$ time:

3-Sum:

Given an array $A[1..n]$ of distinct positive integers that is not (necessarily) sorted, and a positive integer t, determine whether or not there are three distinct elements x, y, z in $A[]$ such that $x + y + z = t$.

2.2 Solution for 2A)

```
def two_sum(A, t):
        lower_i = 1                    // the low end index
        upper_i = n                    // the high end index
    while (A[upper_i] > t):                      // this quickly trims upper_i down; O(lg n)
            if (A[floor(upper_i/2)] >= t):
                upper_i = floor(upper_i/2)
        else:
                break
        while (lower_i < upper_i):
                if (A[lower_i] + A[upper_i] > t):
                upper_i = upper_i - 1                       // This will further trim upper_i
                elif (A[lower_i] + A[upper_i] == t):
                    return "list A contains at least 1 pair of values that sums to t"
        else:
                lower_i = lower_i + 1                       // This will start incrementing lower_i
        return "list A doesn't contain a pair that sums to t"
```

In my solution, I have index variables initialized to the highest and lowest indices for $A[]$, and they will incremented towards the solution. I included a little extra unnecessary complexity for efficiency. I could have excluded the first while loop and still achieve $O(n)$ performance, but as $A[]$ only contains distinct positive integers, the maximum possible y that satisfies $x + y = t$ occurs when $x = 0$. So we want to get to i such that $A[i] <= t$ as quickly as possible. My first while loop will run in (at worst) $O(3 \lg n)$ time, and it will subtract $O(\frac{n}{2^{\lg n}})$ from runtime.

The next loop is the core of the algorithm. Each iteration, if the sum of $A[lower_i] + A[upper_i] > t$, then we reduce $upper_i$ by 1, if the sum of $A[lower_i] + A[upper_i] < t$, then we increase $lower_i$ by 1, and if $A[lower_i] + A[upper_i] == t$, then we return success. As the list is ordered, positive, and monotonically increasing, every iteration of this process will move the sum towards t (ie any decrease in $lower_i$ or increase in $upper_i$ will move the sum further from t. If $lower_i$ exceeds $upper_i$, then we know that the search failed to find a pair. It this loop, at worst, there will be 3 comparisons and 1 assignment per iteration, and in the worst case scenario, the performance is $O(4n)$.

So for the 2SUM method, the worst case performance is dominated by $O(4n)$ which is $O(n)$.

2.3 Solution for 2B)

```
def three_sum(A, t):
        lower_i = 1                    // the low end index
        mid_i = 2                    // the iterating index
        upper_i = n                    // the high end index

        while (lower_i < upper_i):
```

```
            for j in range(mid_i, upper_i):
                if (A[lower_i] + A[j] + A[upper_i] > t):
                    break
            elif (A[lower_i] + A[j] + A[upper_i] == t):
                    return "list A contains at least 1 set of 3 distinct values that sums to t"
        if (A[lower_i] + A[mid_i] + A[upper_i] > t):
                upper_i = upper_i - 1        // This will further trim upper_i
                elif (A[lower_i] + (A[mid_i] + A[upper_i] == t):
                    return "list A contains at least 1 set of 3 distinct values that sums to t"
        else:
                mid_i = mid_i + 1                // This will keep mid_i above lower_i
                lower_i = lower_i + 1        // This will start incrementing lower_i
        return "list A doesn't contain a pair that sums to t"
```

In this 3SUM solution, it's like the 2SUM solution with a low index and a high index converging in on where a solution could live, but inside the while loop, I've included a for loop that iterates through all indices between $lower_i$ and $upper_i$. In the worst case scenario, this inner loop would operate, on average, in $O(\frac{n}{2})$ time (it would have to check $n-1$ values when $lower_i$ and $upper_i$ are at their initial values but only 1 value if $upper_i = lower_i + 2$). This inner for loop would run every iteration of the while loop, so we would get complexity $O(4n * \frac{n}{2}) \to O(n^2)$.

3 Problem 3)

3.1 Problem Statement

Let $A[1..n]$ be an array of positive integers (A is not sorted). Pinocchio claims that there exists an $O(n)$-time algorithm that decides if there are two integers in A whose sum is 1000. Is Pinocchio right, or will his nose grow? If you say Pinocchio is right, explain how it can be done in $O(n)$ time; otherwise, argue why it is impossible.

3.2 Solution

As $A[]$ is unsorted, the list would have to be iterated, on average, $\frac{n}{2}$ times to find any specific number, and then for each number, you would have to iterate, on average, $\frac{n}{2}$ times again to see if there is a pair that sums to 1000. That has complexity $O(n^2)$, which is greater than $O(n)$.

Any (comparative) sorting algorithm will have at least worst case complexity $O(n \lg n)$. But per a review of tables of complexities for different searching algorithms, it seems radix sort has a complexity of $O(d(n+k))$ where d is the number of digits used to represent array values and k is the base of those values. $O(d(n+k)) = O(dn + dk)$ but as d and k are constants, we see that $O(dk)$ drops off and $O(dn) \to O(n)$.

So this sort can occur in $O(n)$ time, and as shown in the 2SUM algorithm on problem 2, the search can occur in $O(n)$ time, and these actions would happen sequentially, but $O(n) + O(n) \Rightarrow O(n)$.

So, if $A[]$ is is small enough that a radix sort algorithm could sort it in $O(n)$ time, Pinnochio is technically correct. But using any of the sorting methods we've learned about (best performance is $O(n \lg n)$), we couldn't support Pinnochio's claim.

4 Problem 4)

Suppose that we are given an array A[1..n] of integers such that $A[1] < A[2] < \; < A[n]$. Give an $O(lgn)$ time algorithm to decide if there exists an index $1in$ such that $A[i] = i$.

4.1 Solution for 4)

```
def i_spy(A, t):
        lower_i = 1                 // the low end index
        upper_i = n                 // the high end index
    if (A[lower_i]/lower_i > 1) or (A[upper_i]/upper_i < 1):
            Return "Failure: no i exists such that A[i] = i"
    condition = A[lower_i]/lower_i + A[upper_i]/upper_i
        While ( condition < 2):
                mid_i = floor((lower_i + upper_i)/2)
        if ( A[mid_i]/mid_i > 1):
                upper_i = mid_i
        elif ( A[mid_i]/mid_i < 1):
                lower_i = mid_i
        condition = A[lower_i]/lower_i + A[upper_i]/upper_i
        // the live below is a hacky way of dealing with the choice to use "floor()"
        if ((A[mid_i]/mid_i == 1) or ((A[mid_i + 1]/(mid_i + 1) == 1)):
                return "Success: at least one i exists such that A[i] = i"
        return  "Failure: no i exists such that A[i] = i"
```

For this problem, we want to see if any of the values in $A[]$ lie along the $x = y$ line. Taking advantage of the fact that the values of $A[]$ are integers and that they are monotonically increasing, we see that if $A[n]/n < 1$ or $A[1]/1 > 1$, $A[]$ will never cross that $x = y$ line. Additionally, if we know that $A[x_\alpha]/x_\alpha$ and $A[x_\omega]/x_\omega$ are both above or both below the $x = y$ line, then we know ($\forall x_\alpha$ and x_ω) that all points between x_α and x_ω are also on that side of the line (and have not crossed the line, so $A[i] \neq i$ for all included points). So we can use a binary search to either identify a region of interest in the array, or at least cut away half of the array that can't contain a solution. This continues until both ratios $A[lower_i]/lower_i$ and $A[upper_i]/upper_i$ are on the same side of the $x = y$ line. This binary search is the only part of the algorithm that doesn't occur in $O(c)$ time, and binary search takes $O(\lg(n))$ time, so this algorithm has worst case run time $O(\lg(n))$.

5 Problem 5)

- Let $A[1..n]$ be an array of numbers. To find the largest number in A, one way is to divide A into two halves, recursively find the largest number in each half, and pick the maximum between the two.
  - **(a):** Write a recursive algorithm to implement the above scheme. Write a recurrence relation describing the running time of the algorithm and solve it to give a tight bound on the running time of this algorithm.
  - **(b):** Does this recursive algorithm make fewer comparisons than an incremental algorithm that computes the largest element in A by iterating through the elements of A? Explain.

5.1 Solution for 5A)

```
def recursive_max(A):
    if (len(A) == 1):
        return A[0]
    elif (len(A) == 2):
        if (A[0] > A[1]):
            return A[0]
        else:
            return A[1]
    else:
```

```
    mid = floor(len(A)/2)
    half1 = recursive_max(A[0:mid])
    half2 = recursive_max(A[mid+1:len(A)])
    return recursive_max([half1, half2])
```

In the ideal implementation, there would be 1 comparison for every split in the tree and the number of comparisons, $T(n)$, needed to find the max value would be given by the recurrence relations below.

$$T(n) = \begin{cases} 0 & \text{if n = 1.} \\ 1 & \text{if n = 2.} \\ T(\frac{n}{2}) + T(\frac{n}{2}) + 1 & \text{if n > 2.} \end{cases} \tag{1}$$

However, my implementation has more comparisons (to handle practical realities). But it will have the same asymptotic performance. My algorithm's performance is given by the recurrence relation below.

$$T(n) = \begin{cases} 1 & \text{if n = 1.} \\ 3 & \text{if n = 2.} \\ T(\frac{n}{2}) + T(\frac{n}{2}) + 5 & \text{if n > 2.} \end{cases} \tag{2}$$

For convenience, I'm going to look at values of n that are generated by $n = 2^i$ for positive, integer values of i.

- $T(1) = 1$
- $T(2) = 3$
- $T(4) = T(\frac{4}{2}) + T(\frac{4}{2}) + 5 = T(2) + T(2) + 5 = 11$
- $T(8) = T(\frac{8}{2}) + T(\frac{8}{2}) + 5 = T(4) + T(4) + 5 = 27$
- $T(16) = T(8) + T(8) + 5 = 59$
- $T(32) = T(16) + T(16) + 5 = 123$
- ...
- $T(i) = \frac{3i}{2} + 5 * 2^{\lg(i)-1}$

This runtime $T(n)$ will be more accurate when n is close to a value of 2 raised to some integer, but for all i, this function would be a close upper bound on runtime.

5.2 Solution for 5B)

This recursive algorithm will take longer than an incremental algorithm as an incremental algorithm would only have to make $n - 1$ comparisons, while the recursive algorithm will have to make at least $\frac{3n}{2}$ comparisons.