Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 1

Least to greatest (each line is a separate equivalence class):

- Constant Time:

$$n^{\frac{1}{\log n}} \quad 3$$

- Logarithmic:

$$\log\log n \quad \log_8 n \quad \log_2 n \quad \log_2 n^3$$
$$(\log n)^2$$
$$4^{\log n}$$

- Sub-linear:

$$n^{\frac{1}{2}}$$
$$\frac{n}{\log n}$$

- Linear:

$$n - \log n$$
$$n \log n$$

- Polynomial:

$$n^2 \quad n^2 + 10^{100} n \log n$$
$$n^2 \log n$$
$$n^{\log\log n} \quad (\log n)^{\log n}$$
$$n^4$$

- Exponential:

$$4^{2n} \quad n! \quad 8^{n-1} \quad 8^n$$
$$n^n \quad 4^{n^2}$$

Problem 2

```
let n1 = the list of she-aliens
let n2 = the list of he-aliens
let c = n1 ∪ n2
let d = []
for alien in c:
    if alien.preferences is empty:
        remove alien from c
while c is not empty:
    let current = min(c, by length of alien.preferences)
    if current.preferences is empty:
        remove current from c
    let match = random.choice(current.preferences)
    add (current, match) to d
    remove current from c
    remove match from c
    for alien in c:
        remove current from alien.preferences
        remove match from alien.preferences
```

This function is $O(n^2)$ because while iterating through the list of aliens to find matches, we must go through all other aliens if we find a match in order to remove the matches from the other aliens' preferences.

Apparently, the algorithm doesn't work, but we were not able to find a counterexample. We know that the counterexample depends on four-node groups connected in a N-shape, which form a cascading sequence that this greedy algorithm will chase, but we were unable to successfully find a sequence where the greedy algorithm finds a non-optimal solution.

For finding the optimal number of pairs, one could create an $n1 \times n2$ adjacency matrix and fill in the adjacency matrix like an NQueens puzzle, using backtracking to maximize the number of pair points on the grid such that no two filled points have the same column or row.

Apparently, there is also another solution which relies on calculating the flow rate though the bipartite graph, if all of the nodes on one side were connected to a source and all of the nodes on the other side were connected to a sink. We aren't exactly

sure how this one works, but it is supposed to produce the same optimal solution, with the added benefit of not running recursively.

Problem 3

Algorithm Description

For this problem, we needed fast lookup and insertion. A self-balancing binary search tree was the best choice for this, so we implemented an AVLTree. While iterating through the array of numbers, we look in our AVLTree for both numbers that would result in a difference of t, that is, we look for $a_i + t$ and $a_i - t$.

Argument of Correctness

Our algorithm keeps its data in tuples of a value and a frequency. Because it quickly searches for either of the two numbers which could pair with the current value to make t, it is guaranteed to find every possible pairing that makes the desired value.

Running Time Estimate

Since iteration through the numbers is $O(n)$ and we must perform an insertion and 2 lookups for each number, our execution time is $n * 3\log(n)$, which is $O(n \log n)$.

Pseudocode

```
let avlt = new AVLTree()
let count = 0
for i=0 to the length of the input:
    let possibility1 = avlt.search(input[i]+t)
    let possibility2 = avlt.search(input[i]-t)
    if possibility1 exists:
        count += possibility1.frequency
    if possiblilty2 exists and t != 0:
        count += possibility2.frequency
    let currentValue = avlt.search(input[i])
    if currentValue exists:
        currentValue.frequency += 1
    else:
        avlt.insert(currentValue)
return count
```

Problem 4

Algorithm Description

For this problem, we used the AVLTree implemented in problem 3 to keep track of all the t differences. We computed the difference t between every pair of input values and stored each t difference and its frequency as a tuple inside the tree.

Argument of Correctness

This algorithm finds every possible t difference by computing the difference between every single pair of inputs. It uses an AVLTree to store the differences for fast lookup and update. Because we have every possible t difference, we can find the most frequently occurring one.

Running Time Estimate

Finding every possible t difference is $O(n^2)$. Storing them in the tree requires $O(\log n)$ time, and finding the maximum requires $O(n)$ time. This makes our complexity $n^2 * 2\log(n) + n$, which is $O(n^2 \log n)$.

Pseudocode

```
let avlt = new AVLTree()
for i=0 to the length of the input:
    for j=i to the length of the input:
        let t = new Tuple(value=input[i]-input[j], frequency=1)
        let possibility = avlt.search(t)
        if possibility exists:
            possibility.frequency += 1
        else:
            avlt.insert(t)
let max = 0
let stack = new Stack()
stack.push(avlt.root)
while stack is not empty:
    let current = stack.pop()
    if current.frequency > max.frequency:
        max = current
    stack.push(current.left)
    stack.push(current.right)
return max.value
```

Problem 5

Algorithm Description

For this problem, we first found the lexical difference between the two strings as an array of differences between each letter. This requires only a single pass. This list of differences can be represented as a series of heights/depths in a "mountain range". We will refer to the letter distances as heights, and the contiguous groups of letters as "mountains" or "valleys". For example:

$$d = [1\ 2\ 3\ 3\ 2\ 1\ 0\ 3\ 4\ 3\ 3\ 2\ 3\ 0\ \ -1\ \ -2\ \ -3\ \ -2\ \ -3\ 0]$$

In this example, our lexical difference array contains two mountains and one valley. The two mountains contain a valley, and the valley itself contains a mountain. We then recursively chunk the mountains and valleys into groups and cut away the base height or depth shared in common by the mountain/valley. For example, the d array above would be split into three "chunks".

$$\text{mountains} = \begin{bmatrix} [1\ 2\ 3\ 3\ 2\ 1] \\ [3\ 4\ 3\ 3\ 2\ 3] \\ [-1\ \ -2\ \ -3\ \ -2\ \ -3] \end{bmatrix}$$

The first mountain has a height 1 base that we can cut off, the second mountain has a height 2 base that we can cut off, and the valley has a depth 1 base that we can fill upwards. Each of these counts as a shift as defined in the problem statement, for a total of 4 shifts during this operation. After the cutoff, the arrays become:

$$\text{mountains} = \begin{bmatrix} [0\ 1\ 2\ 2\ 1\ 0] \\ [1\ 2\ 1\ 1\ 0\ 1] \\ [0\ \ -1\ \ -2\ \ -1\ \ -2] \end{bmatrix}$$

We then run the chunking function on each of these entries and add the results back into the array. Note that the second mountain, after cutoff, has split into two mountains.

$$\text{mountains} = \begin{bmatrix} [1\ 2\ 2\ 1] \\ [1\ 2\ 1\ 1] \\ [1] \\ [-1\ \ -2\ \ -1\ \ -2] \end{bmatrix}$$

We can cut off the common bases on these mountains/valleys again and add the shifts required to the total. The cut mountains/valleys are rechunked and cut again recursively until they have all been flattened out, at which point the number of operations accumulated from each mountain cut is our result.

Argument of Correctness

This algorithm minimizes the number of shifts needed to accomplish the goal by recursively finding the greatest common contiguous segment of letters that can be shifted in a single direction. This prevents any unnecessary shifts from being made if the same operation can be done on a bigger segment of letters.

Running Time Estimate

This algorithm is $O(n)$, chunking the lexical difference array is linear, while calculating the shift of a mountain is constant. In the worst case, the maximum number of times we need to recursively chunk a mountain with a valley in it is upper bounded by n, so our algorithm runs in constant n time.

Pseudocode

```
def stringDistance(s1, s2):
    result = []
    for c1 in s1, c2 in s2:
        result.append(c2 - c1)
    return result

def chunk(segment):
    let mountains = []
    let mountain = []
    let lastHeight = segment[0]
    for i=1 to segment.length:
        let currentHeight = segment[i]
        if lastHeight == 0 and currentHeight == 0:
            pass
        else if lastHeight == 0 and currentHeight != 0:
            mountain.append(currentHeight)
        else if lastHeight != 0 and currentHeight = 0:
            mountains.append(mountain)
            mountain = [currentHeight]
        else if lastHeight is opposite sign of currentHeight:
            mountains.append(mountain)
            mountain = [currentHeight]
        else:
            mountain.append(currentHeight)
        lastHeight = currentHeight
    mountains.append(mountain)

def chopMin**(chunk):
```

```
    let minValue = min*(chunk)
    for i=0 to chunk.length:
        chunk[i] -= minValue
    return minValue

let queue = new queue
let chunks = chunk(stringDistance(input1, input2))
let totalShifts = 0
for i=0 to chunks.length:
    queue.push(chunk[i])
while queue is not empty:
    let currentMountain = queue.pop()
    shifts += chopMin(currentMountain)
    queue.push(chunk(currentMountain))
return totalShifts
```

*: Note that this min() function finds the element in chunk that is closest to 0, so it is more of an "absolute value min" function.
**: The chopMin() function mutates the input array.