Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 1

Algorithm Description

For this problem, we used two passes over the array of heights to determine the maximum possible volume of the reservoir. One pass would start left to right, and the other pass would sweep right to left. Each pass kept track of the highest point that it passed over and added the difference between the subsequent lower points to the volume. If the pass reached a point higher than the apex that it was tracking, it designated the end of a reservoir, and the summed volume would be added to a heap for tracking.

Argument of Correctness

There are three possible ways for a reservoir to exist, the left wall can be higher than the right, the left wall can be lower than the right, or they can be equal. A left to right pass of this algorithm is able to determine the volume of all the reservoirs as long as the right side of the reservoir is higher than or equal to the left side. A right to left pass has the opposite condition, so by making two passes, we are able to determine the sizes of all the reservoirs.

Running Time Estimate

This algorithm requires two passes through the heights, so the execution time is a constant $2n$, or $O(n)$.

Pseudocode

```
def getMaxReservoirVolume(data):
  let highestVolume = 0
  let currentVolumeLTR = 0
  let currentVolumeRTL = 0
  let currentApexLTR = data[0]
  let currentApexRTL = data[data.length - 1]
  for i=1 to data.length:
    let LTRelement = data[i]
    let RTLelement = data[data.length - i - 1]
    if LTRelement > currentApexLTR:
      currentApexLTR = LTRelement
      highestVolume = max(highestVolume, currentVolumeLTR)
      currentVolumeLTR = 0
    else:
      currentVolumeLTR += currentApexLTR - LTRelement
    if RTLelement > currentApexRTL:
      currentApexRTL = RTLelement
      highestVolume = max(highestVolume, currentVolumeRTL)
      currentVolumeRTL = 0
    else:
      currentVolumeRTL += currentApexRTL - RTLelement
  return highestVolume
```

Problem 2

Consider the divide-and-conquer algorithm in Figure 1 that assumes access to a global string s of lower-case letters, where $s[i]$ refers to the i-th character of s.

1. State the recurrence for $T(n)$ that captures the running time of the algorithm as closely as possible.

   $$O(n)$$

2. Use the "unrolling the recurrence" or the mathematical induction technique to

find a tight bound on $T(n)$.

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2T\left(\frac{n}{2}\right) + c \\
&= 2\left(2T\left(\frac{n}{4}\right) + c\right) + c \\
&= 2\left(2\left(2T\left(\frac{n}{8}\right) + c\right) + c\right) + c \\
&= 2^k T\left(\frac{n}{2^k}\right) + kc \\
&\text{stops when } \frac{n}{2^k} = 1 \equiv k = \log(n) \\
&= 2^{\log(n)} T(1) + c\log(n) \\
&= n + c\log(n) \\
&= O(n + \log(n)) \\
&= O(n)
\end{aligned}
$$

3. What does the algorithm do?
   This algorithm calculates four statistics about the number and placement of vowels and consonants in a string. Those four statistics are:

   - `maxdif`, the maximum of `leftalignedmaxdif` and `rightalignedmaxdif` (usually, although we couldn't figure out the special cases where it wasn't)
   - `leftalignedmaxdif` is the maximum difference between the number of consonants and the number of vowels, in the left half of the input.
   - `rightalignedmaxdif` is the maximum difference between the number of consonants and the number of vowels, in the right half of the input.
   - `dif` contains the difference between the number of consonants and the number of vowels in string s. (This is the only one we're completely sure about.)

Problem 3

Algorithm Description

For this problem, we represented each number in the array as a tuple containing the number's value and the index of its original position. We then ran a merge sort on

the array. Each merge operation was a subproblem that could be handled the same way in order to count the weight of the inversions. When merge sorting two lists, hereafter referred to as $left$ and $right$, pulling a number from the left list into the result means that there was no inversion. Pulling a number from the right list means that number has an inversion with every number on the left list. Essentially, the number of inversions is incremented by the size of the left list each time we pull from the right list for every merge operation.

We made a modification to this algorithm to count the weight of the inversions instead of the number of inversions by precomputing the sum of all the elements in the left list. For this problem, we needed to find the difference in indices between the right element and ALL the elements in the left list every time we pulled from the left list. Let $l_1, l_2, \ldots, l_n$ be the original indices of the elements in the left list, and let r be the original index of the element in the right list which we are pulling out. Each time we pull out an element from the right, we need to compute the following in order to get the total weight all the inversions:

$$\begin{aligned}(r - l_1) + (r - l_2) + \cdots + (r - l_n) &= \sum_{i=1}^{n} r - l_n \\ &= \sum_{i=1}^{n} r - \sum_{i=1}^{n} l_n \\ &= rn - \sum_{i=1}^{n} l_n\end{aligned}$$

Before we begin the merging, we can compute $\sum_{i=1}^{n} l_n$ simply by summing up all the numbers in the left list. Each time we pull from the right list, we multiply the right number's original index by the size of the left list and subtract our precomputed sum to get the weight of all the inversions. Each time we pull from the list list, we subtract that number's value from the precomputed sum so that it remains equal to the sum of all the numbers in the left list. By making this calculation during each merge operation and summing up all the results, we will have the total weight of all the inversions when the list has been fully sorted.

Argument of Correctness

During each merge operation, we are guaranteed that the left and right lists are sorted due to the recursive nature of the merge sort algorithm. Since a merge sort is also stable, this allows us to keep track of the number of times a number is "swapped" when we pull from the right list. Each time a number is swapped, we know it is

inverted with every element in the left list, and we can use this to sum up whatever information we need about the inversions.

Running Time Estimate

The merge sort is $O(n \log n)$, and since we are precomputing the sum of the left side before each merge operation, we can compute the weight of the inversions during the merge in constant time. This makes the entire algorithm $O(n \log n)$.

Pseudocode

```
def calculateWeightedInversions(data):
  let inversions = 0

  def merge(left, right):
    let leftSum = sum([e.index for e in left])
    let result = []
    while left.length + right.length < result.length:
      if left.length == 0:
        result.push(right.pop())
      elif right.length == 0:
        result.push(left.pop())
      elif left.peek().index > right.peek().index:
        inversions += left.length * right.peek().index - leftSum
        result.push(right.pop())
      else:
        leftSum -= left.peek().value
        result.push(left.pop())
    return result

  def mergeSort(data):
    if data.length == 1:
      return data
    let middle = data.length / 2
    return merge(data[:middle], data[middle + 1:])

  mergeSort(data)
  return inversions
```

Problem 4

For each the following recurrences, use the Master theorem to express $T(n)$ as a Theta of a simple function. State what the corresponding values of a, b, and $f(n)$

are and how you determined which case of the theorem applies. Do not worry about the base case or rounding.

1. $T(n) = 9T(\frac{n}{3}) + n^3$

$$\begin{aligned} a &= 9 \\ b &= 3 \\ f(n) &= n^3 \\ n^3 &= O(n^{\frac{\log 9}{\log 3} - \epsilon}) \quad \epsilon > 0 \\ \therefore T(n) &= \Theta(n^{\frac{\log 9}{\log 3}} \log n) \\ &= \Theta(n^2 \log n) \end{aligned}$$

2. $T(n) = \frac{3}{2}T(\frac{2n}{3}) + 3$

$$\begin{aligned} a &= \frac{3}{2} \\ b &= \frac{3}{2} \\ f(n) &= 3 \\ f(n) &= O(n) \\ \therefore T(n) &= \Theta(n) \end{aligned}$$

3. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

$$\begin{aligned} a &= 2 \\ b &= 4 \\ f(n) &= \sqrt{n} \\ \therefore T(n) &= \Theta(\sqrt{n} \log n) \end{aligned}$$

Problem 5

Algorithm Description

For this problem, we implemented a radix sort with a variable base. By computing an optimal base to convert all the numbers into, we are able to sort all the numbers in linear time.

Argument of Correctness

Given n numbers of which the maximum is $n^2 - 1$, a base 10 radix sort would have a runtime of $n \log_{10}(n^2 - 1)$, which is:

$$O(n \log(n))$$

In general, for a radix sort of base b, the runtime is $b \times d$ where d is the number of digits in the largest number. Since we are given the constraint that the largest number can be upper bounded by n^2, d can be expressed in terms of n as $\log_b(n^2) = d$.

Note that as long as b is a function of n of the form cn, then this runtime is equivalent to linear time.

$$\begin{aligned} Let : b &= cn \\ bd &= b \log_b(n^2) \\ &= cn \log_{cn}(n^2) \\ &= 2cn \log_{cn}(n) \\ &= 2cn \frac{\log(n)}{\log(cn)} \\ \lim_{n \to \infty} \frac{\log(n)}{\log(cn)} &= 1 \end{aligned}$$

From the definitions above:

$$\begin{aligned} \log_b(m) &= d \\ \log_b(n^2) &= d \\ n^2 &= b^d \end{aligned}$$

Logically, this shows that we could do a sort in one pass if we let $d = 1$ and $b = n^2$. This will have $O(n^2)$ space complexity as it is basically just creating an array of size n^2 and placing the numbers at the index they correspond to. It makes the most sense to choose a b as close to the resultant d as possible because this minimizes the space complexity.

Running Time Estimate

From the calculations above, the running time of this optimized radix sort is linear, or $O(n)$.

Pseudocode

```
def getOptimalBaseDigits(max):
  for base = 2 to max:
    let digits = ceil(log(max) / log(base))
    if digits < base:
      return (base, digits)

def convertFromBase10(number, base, digits):
  let result = []
  while number != 0:
    result.pushFirst(number % base)
    number /= base
  return result

def radixSort(data, base, digits):
  let buckets1 = [[] for i in base]
  let buckets2 = [[] for i in base]
  for i = 0 to data.length:
    buckets1[data[i][0]].pushLast(data[i])
  let empty = buckets2
  let filled = buckets1
  for digit = 1 to digits:
    for bucket in filled:
      while not bucket.isEmpty():
        let num = bucket.popFirst()
        empty[num[digit]].pushLast(num)
    filled, empty = empty, filled
  let result = []
  for bucket in filled:
    result.pushLast(bucket.popFirst())

def main(input):
  let base, digits = getOptimalBaseDigits(max(input))
  let matrix = [convertFromBase10(d, base, digits) for d in data]
  matrix = radixSort(matrix)
  return matrix
```