Austin Baysinger
Final Project
12/11/17

1. *Given an array $\{a_1, a_2, \ldots, a_n\}$, a reverse is a pair (a_i, a_j) such that $i < j$ but $a_i > a_j$. Design a divide-and-conquer algorithm with a runtime of O(n log n) for computing the number of reverses in the array. Your solution to this question needs to include both a written explanation and a Python implementation of your algorithm.*

(a) Explain how your algorithm works, including pseudocode.

Answer:

My algorithm for this problem is a repurposed Merge Sort. Merge Sort works by dividing the arrays into "left" and "right" subarrays and comparing values from each subarray to find the proper sorting. My algorithm differs in that it maintains a counter variable throughout its execution.

Whenever a value in a left subarray is found to be greater than a value in its corresponding right subarray(i.e. $i < j$ but $a_i > a_j$), the counter is incremented by the number of spaces the larger value is displaced in the sorting procedure. This displacement corresponds to the number of elements less than the element of interest that were located at higher indices in the array or, in other words, the displacement is the number of reverses for that element in the current subarray.

The idea is that for every element that is moved to a larger index, each index between its initial unsorted index and its final sorted index represents a reverse. The reverse counter is calculated for each step of recursion and passed along to accumulate a global total of reverses once the array is completely sorted. In terms of complexity, the addition of this counter to Merge Sort only adds a constant number of operations so the $\Theta(n \log n)$ runtime of Merge Sort remains unchanged.

```
def mergeReverses(A[],p,q,r):
     int reverseCount
     n1 = q - p + 1
     n2 = r - q
     L[n1]
     R[n2]
     for i in range [0,n1):
          L[i] = A[p + i]
     for j in range [0,n2):
          R[i] = A[q + j + 1]
     i,j = 0
     k = p
     while i < n1 and j < n2:
          if L[i] <= R[j]:
               A[k] = L[i]
               i++
          else:
               A[k] = R[j]
               reverseCount += (n1 + j - k + p)
               j++
          k++
     while i < n1:
          A[k] = L[i]
          i++
          k++
     while j < n2:
          A[k] = R[j]
          j++
          k++
     return reverseCount

def numReverses(A[],p,r):
     if p < r:
          q = floor((p + r) / 2)
          left = numReverses(A[],p,q)
          right = numReverses(A[],q+1,r)
          merged = mergeReverses(A[],p,q,r)
          return left + right + merged
     return 0
```

(b) Randomly generate an array of 100 numbers and use it as input to run your code. Report on both the input to your code and on how the output demonstrates the correctness of your algorithm.

Answer:

To test my code I first generated an array of size 10 with its first element starting at 10 and counting down to 1. This was easy enough to count the number of reverses as 10 is a reverse with everything to the right in the array, 9 is a reverse with everything to the right and so on. My algorithm sorted this array and returned a count of 45 reverses which matched my own calculations.

For further robustness, I generated an array with 100 elements and populated it randomly with integers in the range [0,100). This reliably creates a large amount of reverses in the array. On average, my algorithm will report around 2500 reverses.

Going back to how I formed my first test array, we can assume a scenario where we have an array A with values: [99, 98, 97, 96, 95, … , 0]. This has the max possible reverses for an array of size 100. The number of reverses can be calculated by summating (n - i - 1) from i = 0 to n – 1. Here n is the size of the array and i represents how far into the array we are. For instance, if we look at A[0], it is a reverse with all n – 1 elements to the right in the array. i in this iteration is 0 so we have A[0] is a reverse with 100 – 0 – 1 = 99 other elements. For A[1] we have 100 – 1 – 1 = 98 and the process continues.

So for a test array with 100 integers, the max possible reverses would be 4950 which is confirmed by my algorithm. It further validates my results on arrays of random elements as 4950 is an upper bound on the average 2500 reverses found in randomized arrays.

2. *Suppose you have been sent back in time and have arrived at the scene of an ancient Roman battle. It is your job to assign n spears to n soldiers so that each soldier has a spear. It is best if your assignments minimize the difference in heights between the height of the man and the height of the spear.*

(a) Design an algorithm to find the optimal, or near optimal, solution without evaluating all possible combinations. Include an explanation and pseudocode showing how your algorithm works.

Answer:

Let S[n] be an array of spear heights and R[n] be an array of soldier heights. First sort both of these arrays then assign S[i] to R[i] for all i in range [0,n). In this way, the shortest spear is assigned to the shortest soldier then the second shortest spear is assigned to the second shortest soldier and so on. It is important to note that, for instance, S[0] may not be the most optimal assignment for R[0]. It may be that a taller spear is more optimal, but if a taller spear is assigned to R[0] then S[0] must later be assigned to a taller soldier where the difference in heights will be even less optimal. So though this algorithm may not always find the optimal assignment, it will be near optimal.

```
def assignSpears(S[],R[],n):
     A[n]
     sort(S[])
     sort(R[])
     for i in range [0,n):
          A[i] = (S[i], R[i])
     return A[]
```

(b) Compare the runtime complexity of your algorithm with the complexity of a brute force solution.

Answer:

Assuming the use of Merge Sort as my sorting algorithm of choice, assignSpears has a runtime of $\Theta(n \log n)$. This is because there are two calls to Merge Sort both with complexity $\Theta(n \log n)$ and then finally an iteration through n. Dropping insignificant terms leaves $\Theta(n \log n)$.

A brute force method would loop through all n spears for each soldier to devise some optimal assignment by evaluating every possible pairing. This clearly has a complexity of $O(n^2)$ due to the nature of nested iteration.

3. *Consider the spider-web DAG graph that shows a spider sitting at the bottom of its web, and a fly sitting at the top.*

(a) Write an algorithm to determine how many different ways the spider can reach the fly by moving along the web's lines in the directions indicated by the arrows.

Answer:

1. Topologically sort the graph G into an array A and reverse the order so that the fly vertex is at A[0].

2. Associate a path counter with each vertex in G. Initialize the fly vertex to 1 and the rest to 0.

3. Starting at A[1], loop through A. For each vertex v in A, sum the path counts of v's direct successors, i.e. the neighboring vertices between v and the fly vertex. At this point, because A is topologically sorted, every vertex's successor nodes will have already been assigned their path counts by the algorithm.

4. Once the algorithm reaches the spider vertex at the end of A, the path count assigned to the spider vertex will be equal to the total number of paths from the spider to the fly.

4. *There are n >= 3 people positioned on a field (Euclidean plane) so that each has a unique nearest neighbor. Each person has a water balloon. At a signal, everybody hurls his or her balloon at the nearest neighbor. Assume that n is odd and that nobody can miss his or her target.*

(a) Write an algorithm to answer the question: Is it true or false that there remains at least one person not hit by a balloon?

Answer:

1. Construct an n by n matrix M and populate it with the distances between every pair of people. For example if the distance between person p_i and person p_j is 10, then the value of M[i][j] equals 10. For all M[i][j] where i = j, store a value that is one greater than the max distance in the matrix. These indices represent distances between a person and him or herself. We set them to a large value so that the algorithm implementation can ignore these values.

2. Create an array A of size n and initialize all slots to the Boolean false. Each index represents a person and the Boolean represents whether or not the person was hit with a balloon.

3. Find the minimum distances of each row i in M. Use the j indices of these distances to set those same locations in A to the Boolean true. Here we are iterating through each person and setting their nearest neighbor in A to true to indicate they were hit by a balloon.

4. After processing M, iterate through A and once a false is found, return true to indicate that at least one person was not hit by a balloon.

(c) Prove that your algorithm is correct. Your proof needs to include specific features of your algorithm.

Answer:

```
1. def dryOrWet(M, n):
2.   A[n] = "Dry"
3.   for i in range(0, n):
4.        minIndex = M[i].index(min(M[i]))
5.        A[minIndex] = "Wet"
6.   return A
```

Above is a code implementation of the above algorithm. Assume M is an n x n matrix already populated with distances. Further assume n >= 3 and each person has a unique nearest neighbor. That means each row and column in M will have a unique minimum value. On line 2 an array of size n is initialized with all elements declared "Dry" to represent the set of people on the place and their initial state of being dry. On lines 4 and 5 of my algorithm, minIndex, an index into row i where that row's minimum value is located, is used to indicate that person A[minIndex] was hit with a balloon. The for loop on line 3 ensures that process happens for each row i in M. After execution, each person's unique nearest neighbor has been processed and A contains a list of who was hit and who wasn't.

(d) Analyze the runtime behavior of your algorithm.

Answer:

My algorithm has a runtime of $O(n^2)$ due to the nature of using a matrix as a data structure. For each n_i person, n distances have to be evaluated. In the code above on line 4, the min() function is essentially an inner for loop as this function has to iterate through n items to find the smallest of them.