Programming Assignment #1 - CS325

Joshua Villwock Jaron Thatcher Ryan Phillips

January 31, 2014

Introduction

Inversion counting is a common problem in computer science. In this document, we will present three different algorithms to solve the issue. Also included are proofs for the algorithms, as well as empirical and asymptotic analysis of their run time on increasingly sized inputs.

Pseudocode

Brute Force:

```
BruteForce(arr)
  count = 0
  for i in 0 to arr.length
    for j in i to arr.length
        if arr[i] > arr[j]:
        count++
  return count
```

Naive Divide and Conquer:

```
NaiveDivideAndConquer(arr)
  count = 0
  if len(arr) < 2:
    return count
  middle = length(list_in)/2
  left = arr[:middle] // slice off half of the array
  right = arr[middle:]
  // count inversions between left and right halves
  for i in range (0,len(left)):
    for j in range (0,len(right)):
      if left[i] > right[j]:
          count++
  // and count internal inversions recursively
  count += NaiveDivideAndConquer(left)
  count += NaiveDivideAndConquer(right)
  return count
```

Merge and Count:

```
MergeAndCount(arr,0)
  results = []
  // base case
  if len(x) < 2:
    return x, count
  middle =len(x)/2
  // recursive calls
  left, count = MergeAndCount(x[:middle],count)
  right, count = MergeAndCount(x[middle:],count)
  i, j = 0, 0
  while i < length(left) and j < length(right):
    if left[i] > right[j]:
        results.append(right[j])
        count += length(left) - i
        j++
    else:
        results.append(left[i])
        i++
    results += left[i:]
    results += right[j:]
    return results, count
```

Correctness Proofs

Naive Divide and Conquer:

Pre-explanation:

The Nave divide & Conquer which we were tasked to develop is fairly simple once you know how it works: It splits the list into 2 halves, then brute forces the inversions between the first half and the second half, but NOT inversions within any one half. It then continues recursively by calling itself on the 2 resulting half-lists. Eventually each list contains either 1 or 0 elements, at which point, there are no inversions in that list, and it returns.

Proof by contradiction:

Assume that the result of the master divide & conquer is incorrect. There are only 2 ways this could happen: Either the recursive calls return an incorrect result, or the checking the two halves of the list against each other is incorrect.

Base case: The only way the recursive calls could be incorrect is if the base case (when no more recursion is needed) is wrong. In this case, that is when the list has 0 or 1 elements. A single element obviously cannot have any inversions, and our code returns that.

The other case is if our code that brute-forces the halves was incorrect, or missed some numbers: Our algorithm simply does a brute-force on the entire range where i is in the first half, and j is in the second half. The recursive calls take care of where i and j are in the same half, so all the bases are covered here. We can assume that brute-forcing those numbers is correct, since this was talked about extensively in class, and is logically sound. (merely comparing every i with every j for any inversions)

Merge and Count:

Pre-explanation:

Merge & Count works by simply running mergesort on the list, and counting how many times elements are inverted. Mergesort works (code actually flows in the other direction, as it is recursive, however it makes more sense conceptually this way) by splitting the Length N list into n separate elements, then continues to combine those into n/2 lists of 2n length. It does this by comparing the existing "lists" (singlets) and moving the smallest of the first elements of either list into the new list, thereby sorting them.

Proof by contradiction:

Assume that one of the resulting lists is incorrect. If one of the lists obtained by combining smaller lists was not sorted, then mergesort would no longer work, and therefore the inversion counting would also be incorrect. The only way for this to happen is for the smaller sub-lists to be incorrectly sorted, (since we assume they are sorted, and only look at the first elements in either) and the only way for those to be incorrectly sorted is if exactly what we are talking about right now is true. Eventually we get to the base case, where each sub-list is only 1 element, but it is impossible to mess this up, since there is no way for a single-element list to not be sorted.

Asymptotic Analysis of Run Time

Brute Force: It uses two nested loops, with the lower bound of the inner set to the current value of the outer.

```
This has a runtime of:

(n^2 + n) / 2

Which can also be expressed as:

O(n^2)
```

Naive Divide and Conquer:

```
Recurrence Relation:

T(n) = O(n/2)^2 + 2T(n/2)
T(n) = (n^2)/4 + 2T(n/2) // 1st

Telescoping:

T(n/2) = ((n/2)^2 / 4) + 2T(n/4) = n^2 / 16 + 2T(n/4) // 2nd

General form:

T(n) = (1/2)*(-1+n)n + (n + c)/2

Since we have the following terms: (1/2)*n^2 + (1/2)n

This can be simplified as just: O(n^2)
```

Merge and Count:

```
T(n) = T(n/2) + O(n)
T(n) = T(n/2) + cn // 1st recursion

Telescoping:

[T(n/4) + cn] + cn = T(n/4) + 2cn // 2
[T(n/8) + cn] + 2cn = T(n/8) = 3cn // 3

General Pattern: T(n/2^n) + ncn
```

```
When T(n) = T(n/(2^n) + ncn

You have two subproblems of size n
 Plus linear time combination

AKA: O(n log n)
```

Testing

The first test for correctness was performed using the provided file “verify.txt”. It was assumed that the last value of each row was the expected number of inversions, so all 3 algorithms were run on each row of values (excluding the last), and this was compared to the expected value. This can be performed via: “test_correctness1(“verify.txt”)”.

The second test for correctness used the second provided file “test_in.txt”. Since no expected values were given, the results were just printed out. All 3 algorithms gave the same value, so this is a good indication. The results have been included below, with just a single value given (number of inversions) for each row in the test file. This test can be run by calling the function “test_correctness2(“test_in.txt”)”.

Results: 252180, 250488, 243785, 247021, 250925, 256485, 249876, 253356, 255204, 247071

Extrapolation and Interpretation

Slope of lines in log-log plot:

The equation for the best fit line on the log-log plot (calculated using numpy.polyfit()) has the following form:

$f(n) = e^{\text{y-intercept}} * n^{\text{slope}}$

Brute Force:

- slope: 2.05692581355
- y-intercept: -17.5588327907

Naive Divide & Conquer:

- slope: 2.03626014268
- y-intercept: -17.3982172216

Merge & Count:

- slope: 1.10025742067
- y-intercept: -13.1947528938

Largest input item solvable in an hour:

Extrapolation using the best fit function from the previous section:

f(n) = Runtime = 1 hr = $e^{\text{y-intercept}} * n^{\text{slope}}$

Solving for n, using the values from the above chart, yields the following numbers:

Brute Force: 265410

Naive Divide & Conquer: 299864

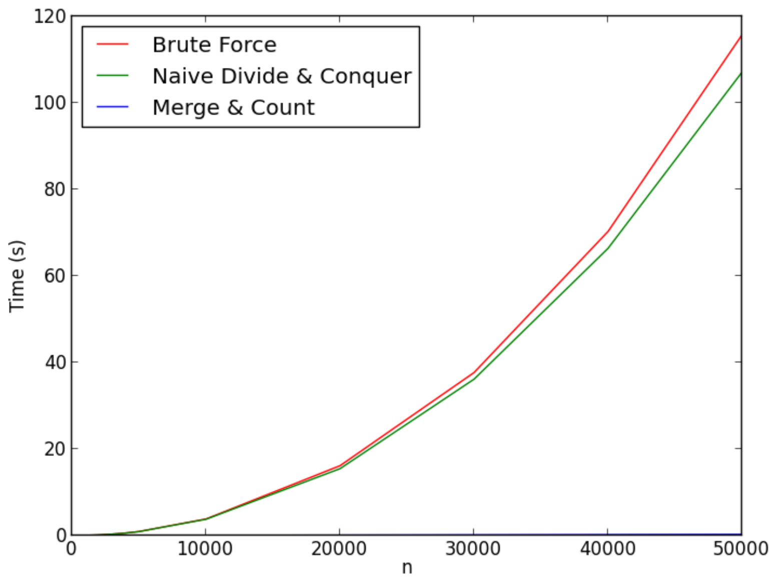
Merge & Count: 275733290

Discrepancy between actual and asymptotic:

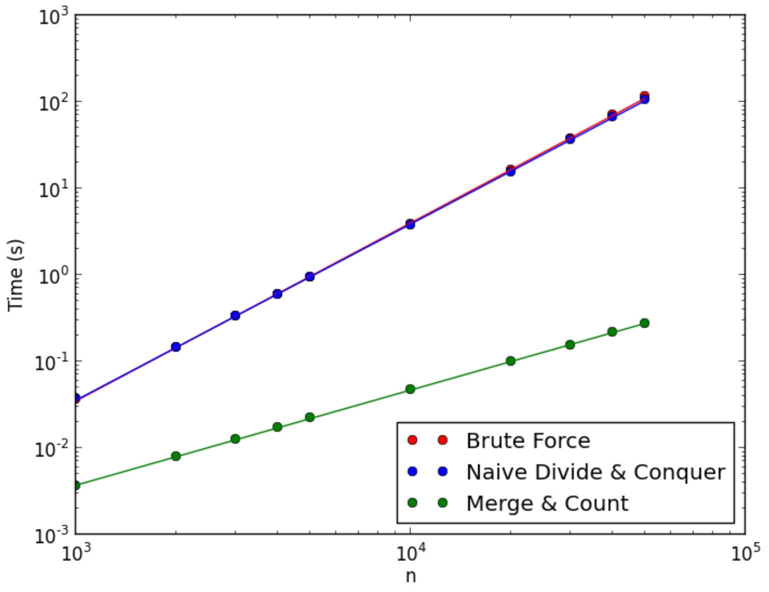
The slopes of the first two best fit lines on the log-log graph are very close to 2, and the third is just over 1.1. This matches up with what is expected - i.e. there is no real discrepency to report.

Empirical Analysis of Run Time

Linear Plot:



Log-log Plot:


Brute Force
Naive Divide & Conquer
Merge & Count
Time (s)
n
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{3}
10^{4}
10^{5}