Algorithms & Complexity: Lecture 11: Divide and Conquer II

Sam Barrett

March 17, 2021

1 Counting number of inversions

Problem 1 *(Inversion)*
Given an array of n pairwise disjoint (unique) numbers $a_1, a_2, \ldots, a_n$
Two numbers a_i *and* a_j *form an inversion if* $i < j$ *but* $a_i > a_j$

The number of inversions is a measure of the *sortedness* of an array.

The naive approach to solving this requires $O(n^2)$ time, as it checks if each of the $\binom{n}{2}$ pairs is an inversion or not.

We can instead construct an algorithm similar to MergeSort:

1. Divide

   Divide the list L into two equal sections L_1 and L_2

2. Conquer

   Count the number of inversions within L_1 and L_2

3. Combine

   Count the number of inversions where one number is from L_1 and the other is from L_2

Analysis:
Let $T(n)$ be the time needed to count the number of inversions from a given list of n numbers.

- The divide step requires $O(1)$ time
- The Conquer step requires $2 \cdot T(n/2)$ steps
- The combine step requires $O(n^2)$ steps as $|L_1| = n/2 = |L_2|$
- Our recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n^2)$

This approach doesn't appear to be any better than our naive $O(n^2)$ approach. Can we improve the combination step?

If we can get the combination step down to $O(n)$ time the our entire procedure can be reduced to the complexity of MergeSort ($T(n) = O(n \cdot \log n)$)

1. Divide

   Divide the list L into two equal sections L_1 and L_2

2. Conquer

   Sort and count the number of inversions within L_1 and L_2

3. Combine

   We can assume that both lists are **sorted** Count the number of inversions where one number is from L_1 and the other is from L_2

Analysis:

- The divide step requires $O(1)$ time
- The Conquer step requires $2 \cdot T(n/2)$ steps
- The combine step:

  It can be shown that this step now only needs $O(n)$ time as both lists are sorted

Algorithm 1: Combine(L_1, L_2)

Input: Two sorted lists L_1, L_2 of length $n/2$ each
Let $L_1 = \{b_1 < b_2 < \ldots < b_{n/2}\}$
Let $L_2 = \{c_1 < c_2 < \ldots < c_{n/2}\}$
Initialise $i, j = 1$
Initialise $c = 0$; `// counter to maintain number of inversions`
while $i \neq n/2 \cap j \neq n/2$ **do**
  ; `// Even if one list becomes empty, we can stop`
  if $b_i > c_j$ **then**
    $j++$ $c+= (n/2) - i + 1$; `// All numbers in` L_1 `after` a_i `are` $> b_j$
  else
    ; `// In this case we have` $b_i < c_j$
    $i++$
  end
end
return c

Running time:

- In each step, either i or j increases

- The while loop ends when one of the lists becomes empty
- Hence, the total running time is $|L_1| + |L_2| = O(n)$

- Our recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n)$

  Same as for MergeSort and we have solved this recurrence to $T(n) = O(n \log n)$

We can now construct an algorithm `Sort-and-Count` which takes a list L and returns the number of inversions in L and the sorted version of L.

Algorithm 2: `Sort-and-Count(`L`)`

```
1 Divide L into two lists L_1 and L_2
2 Let (r_1, L'_1) = Sort-and-Count(L_1)
3 Let (r_2, L'_2) = Sort-and-Count(L_2)
4 Let (c, L') be the output of Combine(L'_1, L'_2)
5 return c + r_1 + r_2
6 return L'
```

2 Faster integer multiplication

We will make two assumptions:

1. Multiplying 2 bits can be done in constant time
2. Adding 2 bits can be done in constant time

When we consider the process of long multiplication, we will split the problem of, for example, 12×13 into 12×10 and 12×3 and combine these results. Intuitively we can see that this process is similar to that which we have been employing to create divide and conquer algorithms.

In this formulation (for multiplying two binary strings of length n) we need to add $O(n)$ binary strings where each binary string takes $O(n)$ time to compute. This leads to a total running time of $O(n^2)$.

2.1 Attempt 1

Let x_1, x_0 be the first and last $n/2$ bits respectively.

We can therefore say that: $x = x_0 + 2^{n/2} \cdot x_1$, where we also know that both $x_{0,1}$ have length $n/2$

Let y_1, y_0 be the first and last $n/2$ bits respectively.

We can therefore say that: $y = y_0 + 2^{n/2} \cdot y_1$, where we also know that both $y_{0,1}$ have length $n/2$

We can then also see that:

$$\begin{aligned} x \cdot y &= (x_0 + 2^{n/2} \cdot x_1) \cdot (y_0 + 2^{n/2} \cdot y_1) \\ &= x_0 y_0 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + 2^n \cdot x_1 y_1 \end{aligned}$$

We therefore need to solve the following four instances of multiplication of binary strings of length $n/2$ each:

- x_0 and y_0
- x_1 and y_1
- x_0 and y_1
- x_1 and y_0

Giving us the recurrence: $T(n) \leq 4 \cdot T(n/2) + O(n)$. We know this recurrence solves to a running time of $O(n^2)$, no better than our naive algorithm

2.2 Attempt 2

We will require the following three quantities:

1. $x_0 y_0$
2. $x_1 y_1$
3. $x_0 y_1 + x_1 y_0$

We have previously shown that these can be obtained from solving four instances of size $n/2$, but can we do better?

Observe:

$$x_0 y_1 + x_1 y_0 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

We therefore only need to solve **three** instances of multiplications of binary strings of length $n/2$:

1. x_0 and y_0
2. x_1 and y_1
3. $(x_0 + x_1)$ and $(y_0 + y_1)$

This gives us the recurrence: $T(n) = 3 \cdot T(n/2) + O(n)$ which can be solved to a time of $O(n^{\log_2 3}) \equiv O(n^{1.59})$. This is an improvement over our naive and first approach.