CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 3: Divide and conquer

We will see some divide and conquer algorithms, including the interesting median of median algorithm.

Counting inversions [KT 5.3]

Input: n distinct numbers $a_1, a_2, \ldots, a_n$.

Output: number of pairs with $i<j$ but $a_i > a_j$.

For example, given $(3, 7, 2, 5, 4)$, there are five pairs of inversion $(3,2), (7,2), (7,5), (7,4), (5,4)$.

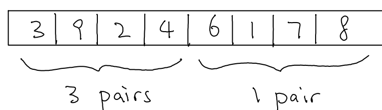
You can think of this problem is computing the "unsortedness" of a sequence.

You may also imagine that this is measuring how different are two rankings; see [KT 5.3] for an elaboration of this connection.

For simplicity, we again assume than n is a power of two.

Using the idea of divide and conquer, we try to break the problem into two halves.

Suppose we could count the number of inversions in the first half, as well as in the second half. Would it be easier to solve the remaining problem?



It remains to count the number of inversions with one number in the first half and the other number in the second half.

These "cross" inversion pairs are easier to count, because we know their relative positions and we just need to count how many.

In particular, to facilitate the counting, we could sort the first half and the second half, without worrying because we have already counted the inversion pairs within.

Now, for each number in the second half, the number of "cross" inversion pairs is precisely the number of numbers in the first half that is larger than it.

In this example, 1 is involved in 4 cross pairs, 6, 7, 8 are all involved in 1 cross pair

(with 9), and so the number of "cross" inversion pair is 7.

How to count the number of cross inversion pair involving a number a_j in the second half?

Method 1: Since the first half is sorted, we can use binary search to determine how many numbers in the first half is greater than a_j. This takes $O(\log n)$ time for one number, and $O(n \log n)$ time for all numbers in the second half.

Not too bad, but we can do better.

Method 2: Observe that we can determine this information when we merge the two sorted list as in merge sort. When we insert a number in the second half to the merged list, we know how many numbers in the first half is greater than it.

For example, | 2 | 3 | 4 | 9 | | 1 | 6 | 7 | 8 | $\Rightarrow$ | 1 | 2 | 3 | 4 | 6 | | | |, we know there is only one number in the first half which is greater than 6.

(arrows point to 9 in the first list and 6 in the second list)

As in merge sort, this can be done in $O(n)$ time.

Now we are ready to describe the algorithm we have so far.

Count ($A[1,n]$) $\leftarrow T(n)$

- if $n=1$, return.
- count ($A[1,\frac{n}{2}]$) $\leftarrow T(\frac{n}{2})$
- count ($A[\frac{n}{2}+1,n]$) $\leftarrow T(\frac{n}{2})$
- sort ($A[1,\frac{n}{2}]$) $\leftarrow O(n\log n)$
- sort ($A[\frac{n}{2}+1,n]$) $\leftarrow O(n\log n)$
- merge-and-count-cross-inversions ($A[1,\frac{n}{2}]$, $A[\frac{n}{2}+1,n]$) $\leftarrow O(n)$

Total time complexity is $T(n) = 2T(\frac{n}{2}) + O(n\log n)$.

Solving this will give $T(n) = \Theta(n \log^2 n)$. Try this using the recursion tree method!

Perhaps you have already noticed, that the sorting step is the bottleneck and not necessary, as we have already sorted them in the merge step. Alternatively, we can just think that during merge sort we can count the number of inversion pairs easily.

The final algorithm is :

Count-and-sort ($A[1,n]$) ← $T(n)$

if $n=1$, return 0.

S_1 = count-and-sort ($A[1, \frac{n}{2}]$) ← $T(\frac{n}{2})$

S_2 = count-and-sort ($A[\frac{n}{2}+1, n]$) ← $T(\frac{n}{2})$

S_3 = merge-and-count-cross-inversions ($A[1,\frac{n}{2}]$, $A[\frac{n}{2}+1,n]$) ← $O(n)$

return $S_1 + S_2 + S_3$

Total time complexity $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$ as in merge sort.

Exercise : Write a pseudocode for the merge-and-count subroutine.

Maximum Subarray [CLRS 4.1]

Input : n numbers $a_1, \dots, a_n$

Output : i, j that maximizes $\sum_{k=i}^{j} a_k$.

For example, 1, -3, 4, 5, 6, -20, 5, 7, 7, -3, 9, 10, 11, -10, 3, 3

(5, 7, 7, -3, 9, 10, 11: this is the optimal solution)

The problem is trivial if all numbers are positive.

A naive algorithm is to try all i, j and compute the sum, and this takes $O(n^3)$ time, too slow.

We can speed it up by realizing that for one i, we can compute sum from i to j for all j in $O(n)$ time, and so the total time is $O(n^2)$.

We now apply the divide and conquer approach to this problem, although it is not the best as we will discuss after.

We again assume n is a power of two.

Suppose we break the array into two halves, and find the maximum subarray within each half. Would it be easier to solve the remaining problem?

1, -3, 4, 5, 6, -20, 5, 7   7, -3, 9, 10, 11, -10, 3, 3

(4, 5, 6 marked in the first half; 7, -3, 9, 10, 11 marked in the second half)

The optimal solution is either contained in the first half, contained in the second half, or cross the mid-point.

So, after solving the two subproblems, we just need to find the maximum subarray that crosses the mid-point - i.e. $i \le \frac{n}{2} < j$.

Each such crossing subarray can be broken into two pieces - $[i, \frac{n}{2}]$ and $[\frac{n}{2}+1, j]$

Assume $[i,j]$ is an optimal solution with $i \le \frac{n}{2} < j$.

Observe that $[i, \frac{n}{2}]$ must be the maximum subarray ending at $\frac{n}{2}$. Suppose not, the sum from $[i', \frac{n}{2}]$ is even bigger, then $[i', j]$ would be bigger than $[i,j]$, contradicting to the optimality of $[i,j]$.

This observation leads to the following claim that leads to a simple algorithm.

Claim $[i,j]$ is an maximum subarray crossing $\frac{n}{2}$ if and only if
$[i, \frac{n}{2}]$ is a maximum subarray ending at $\frac{n}{2}$ and
$[\frac{n}{2}+1, j]$ is a maximum subarray starting from $\frac{n}{2}+1$.

Finding a maximum subarray with a fixed starting / ending point can be done in $O(n)$ time, like what we did in the $O(n^2)$ algorithm by computing the partial sums.

The algorithm is correct by the above argument.

The total time complexity is $T(n) = 2T(\frac{n}{2}) + O(n)$ ← for maximum "crossing" subarray, and this implies that $T(n) = O(n \log n)$.

Challenge: Design an $O(n)$ time algorithm for the maximum subarray problem.

Question: Can you extend the algorithm to work in the circular setting, where the array wraps around?

6 1 2
7 -3
-10 8 4

Finding median [CLRS 9.3]

Input: n distinct numbers $a_1, a_2, \ldots, a_n$

Output: return the median of these n numbers

It is clear that the problem can be solved in $O(n \log n)$ time, by first sorting them.
But it turns out that there is an interesting $O(n)$-time algorithm.

To solve the median problem, it is more convenient to consider a more general problem.

Input: $a_1, \dots, a_n$ and k

Output: return the k-th smallest number in $a_1, \dots, a_n$.

The reason is that the median problem doesn't reduce to itself (so can't recurse), while the k-th smallest number problem lends itself to reduction.

The idea is similar to that in quicksort (which is a divide and conquer algorithm).

We pick a number a_i. Split the n numbers into two groups, one group with numbers smaller than a_i, called in S_1, and another group with numbers greater than a_i, called it S_2.

From the sizes of S_1 and S_2, we know the ranking of a_i, called it r, i.e. a_i is the r-th smallest number in $a_1, \dots, a_n$.

If $r=k$, we are done.

If $r>k$, find the k-th smallest number in S_1.

If $r<k$, find the $(k-r)$-th smallest number in S_2.

Notice that when $r>k$, the problem size is reduced to $r-1$; when $r<k$, the problem size is reduced to $n-r$.

So, if somehow we could choose a number "in the middle" as a pivot (as in quicksort), then we reduce the problem size quickly and making good progress, but finding a number in the middle is the very question that we want to solve.

But we don't need the pivot to be exactly in the middle, just that it is not too close to the boundary.

Suppose we can choose a_i such that its ranking $\frac{n}{10} \le r \le \frac{9n}{10}$, then we know that the problem size would have reduced by at least $\frac{n}{10}$, no matter what is k.

So, the recurrence relation for the time complexity is $T(n) \le T\left(\frac{9n}{10}\right) + cn$ ← for splitting, and this implies that $T(n) = O(n)$.

But how do we find such a good pivot?

__Randomization__ If you have seen randomized quicksort before, then you probably would

guess that a random pivot would work. This is indeed the case and you can take a look at [DPV 2.4] for a proof. We won't discuss randomized algorithms here.

Deterministic Interestingly, there is a deterministic algorithm that would always find a number with $\frac{3n}{10} \le r \le \frac{7n}{10}$ in $O(n)$ time.

If this can be done, then the time complexity is $T(n) = T\left(\frac{7n}{10}\right) + P(n) + cn$.

($P(n)$: time to finding good pivot; cn: splitting)

We need $P(n) = O(n)$ for the $T(n) = O(n)$.

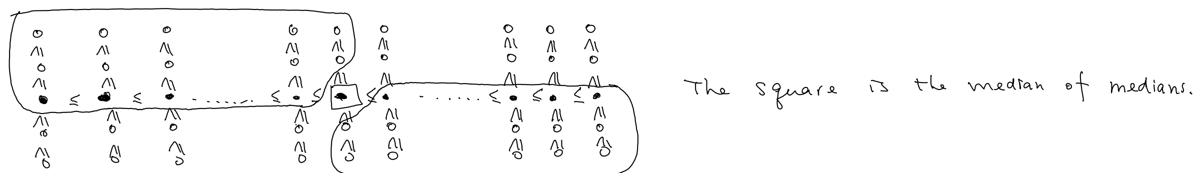
Pivoting The idea of the pivoting algorithm is to find the median of medians.

- We divide the n numbers into $\frac{n}{5}$ groups, each with 5 numbers. ← $O(n)$
- We find the median of each group. ← $O(n)$
- Then we find the median of these $\frac{n}{5}$ medians. ← $T(\frac{n}{5})$

Claim Let r be the rank of the median of the medians.

Then $\frac{3n}{10} \le r \le \frac{7n}{10}$.

proof



The square is the median of medians.

In the picture, we sort each group and order the group by an increasing order of the medians.

I emphasize that this is only for the purpose of proving the claim; we don't need to do this sorting in the algorithm.

Then it should be clear from the picture that the median of medians is greater (smaller) than the numbers in the top-left corner (bottom-right corner).

There are about $\frac{3n}{10}$ numbers at the top-left corner and the bottom-right corner, since there are $\frac{n}{10}$ groups with three numbers each.

Therefore, $\frac{3n}{10} \le r \le \frac{7n}{10}$. □

The claim shows that the pivoting algorithm is correct.

Time complexity

Time complexity

We have $P(n) = T(\frac{n}{5}) + c_2 n$.

So, $T(n) = T(\frac{7n}{10}) + P(n) + c_1 n = T(\frac{7n}{10}) + T(\frac{n}{5}) + (c_1+c_2)n$.

Now, as we have seen in L02, this implies that $T(n) = O(n)$ (could check by induction).

This completes the analysis of the median of medians algorithm.

Exercise: What happen if we divide into groups of 3 elements, 7 elements, $\sqrt{n}$ elements?

Exercise: It would be good to trace out the recursion to understand the algorithm more deeply.