CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 3: Divide and Conquer

We will study 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 as computing the "unsortedness" of a sequence.

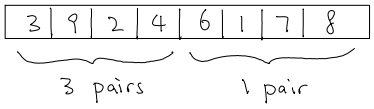
You may also imagine that this is measuring how different are two rankings; see [KT 5.3] for an elaboration of this connection.

For simplicity, we again assume that 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 then 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.

In particular, to facilitate the counting, we could sort the first half and the second half, without worrying losing information as we already counted the inversion pairs within each half

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

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 pairs is 7.

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

Idea 1: As the first half is sorted, we can use binary search to determine how many numbers in the first half are greater than a_j.

This takes $O(\log n)$ time for one a_j, and totally $O(n \log n)$ time for all numbers in the second half.

This is not too slow, but we can do better.

Idea 2: Observe that this information can be determined when we merge the two sorted list 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 that are greater than it.

For example, 

, we know that there is only one number in the first half that is greater than 6.

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

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

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

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

if $n=1$, return 0. // base case

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 is unnecessary, as we have already sorted them in the merge step.

As it turns out, we can just modify the merge sort algorithm to count the number of inversion pairs.

Final algorithm

count-and-sort ($A[1,n]$) $\leftarrow T(n)$

if $n=1$, return 0.

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

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

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 $T(n) = 2T(\frac{n}{2}) + O(n)$ ⇒ $T(n) = O(n \log n)$ as in merge sort.

After all - it is just a simple modification of the merge sort algorithm, but I hope the (long) thought process clarifies how we approach the problem and come up with the algorithm.

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

Maximum Subarray [CLRS 4.1]

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

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
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 the 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.

We again assume that 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

An optimal solution is either contained in the first half, contained in the second half, or crossed 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 that $[i, j]$ is an optimal solution with $i \le \frac{n}{2} < j$.

Observe that $[i, \frac{n}{2}]$ must be a maximum subarray ending at $\frac{n}{2}$.

Suppose not. If the sum from $[i', \frac{n}{2}]$ is even bigger, then the sum from $[i', j]$ would be bigger than that of $[i, j]$, contradicting the optimality of the sum from $[i, j]$.

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

Claim For $i \le \frac{n}{2} < j$, $[i, j]$ is a maximum subarray crossing the mid-point 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 at $\frac{n}{2}+1$.

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

The correctness of the algorithm is based on the claim above.

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

Challenge: Provide 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? E.g. 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: the median of these numbers.

It is clear that the problem can be solved in $O(n \log n)$ time, by first sorting the numbers.

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 slightly more general problem.

Input: n distinct numbers $a_1, \ldots, a_n$ and an integer $k \ge 1$.

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

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

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

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

Let r be the rank of a_i, i.e. a_i is the r-th smallest number in $a_1, \ldots, a_n$.

If $r = k$, then we are done.

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

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

Observe that when $r > k$, the problem size is reduced to $r-1$ as $|S_1| = r-1$,

and when $r < k$, the problem size is reduced to $n-r$ as $|S_2| = n-r$.

So, if somehow we could choose a number "in the middle" as a pivot as in quicksort, then we can 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 observe that 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 rank satisfies say $\frac{n}{10} \le r \le \frac{9n}{10}$, then we know that the problem size would have reduced by at least $\frac{n}{10}$, as $|S_1| = r-1 \le \frac{9n}{10}$ and $|S_2| = n-r \le \frac{9n}{10}$.

So, the recurrence relation for the time complexity is $T(n) \le T(\frac{9n}{10}) + P(n) + cn$, where $P(n)$ denotes the time to find a good pivot and cn is the number of operations for splitting.

If we manage to find a good pivot in $O(n)$ time, i.e. $P(n) = O(n)$, then it implies that $T(n) = O(n)$.

We have made some progress to the median problem, by reducing the problem of finding the number exactly in the middle to the easier problems of finding a number not too far from the middle.

It remains to figure out a linear time algorithm to find a good pivot.

Randomized solution: If you have seen randomized quicksort before, then you would have guessed that keep choosing 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 in this course.

Deterministic solution: There is an interesting deterministic algorithm that would always return a number a_i with rank $\frac{3n}{10} \le r \le \frac{7n}{10}$ in $O(n)$ time.

As seen above, this means $P(n) = O(n)$ and it follows that $T(n) = O(n)$.

Finding a good pivot: The idea of the algorithm is to find the median of the medians.

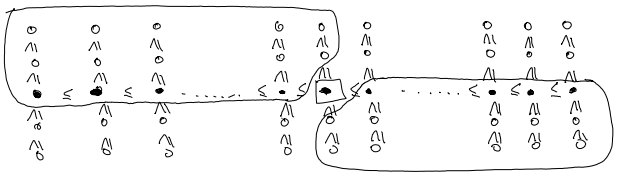
① Divide the n numbers into $\frac{n}{5}$ groups, each of 5 numbers. Time: $O(n)$.

② Find the median in each group. Call them $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.

③ Find the median of these $\frac{n}{5}$ medians $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $T(\frac{n}{5})$.

Lemma Let r be the rank of the median of medians. Then $\frac{3n}{10} \le r \le \frac{7n}{10}$.

Proof



The square is the median of the medians.

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

We emphasize that this is just for the analysis, and we don't need to do sorting in the algorithm.

It should be clear that the square is greater than the numbers in the top-left corner, and is smaller than the numbers in the bottom-right corner.

There are about $3\cdot(\frac{n}{10}) = \frac{3n}{10}$ numbers in the top-left and the bottom-right corners.

This implies that $\frac{3n}{10} \le r \le \frac{7n}{10}$. □

This lemma proves the correctness of the pivoting algorithm.

Time complexity

We have $P(n) = T(\frac{n}{5}) + c_1 n$, where c_1 is a constant.

By the reduction above, $T(n) \le T(\frac{7n}{10}) + P(n) + c_2 n = T(\frac{7n}{10}) + T(\frac{n}{5}) + (c_1+c_2)n$.

Using the recursion tree method in L02, we see that $T(n) = O(n)$. We could check it by induction.

This completes the analysis of the median of medians algorithm.

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

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