CS 787: Advanced Algorithms

Divide and Conquer

Instructor: Dieter van Melkebeek

We continue our review of some topics that are usually covered in an undergraduate algorithms course. Today's topic is the divide-and-conquer method.

1 Paradigm

An instance of the given problem is divided into easier instances of the same problem, which are solved recursively and then combined to create a solution to the original instance.

Of course, divide and conquer is not suitable for every problem. A divide and conquer approach will only work if the problem is easily divided into a small number of easier sub-problems, and the solution to the entire problem is easy once the sub-problems have been solved.

We give two examples of problems that can be efficiently solved by divide and conquer algorithms.

2 Sorting

Given: an array of n integers.

Goal: rearrange the array into non-decreasing order.

There are many algorithms for this problem. The divide-and-conquer approach we present is known as Merge Sort. The idea of Merge Sort is to break up the array into two subarrays of half the size (± 1), recursively sort each of them, and then merge the sorted subarrays to create the entire sorted list.

```
Merge-Sort($A$ - array of length $n$)
  if $n \leq 1$ then Return.
         else Merge-Sort(first half $A$).
              Merge-Sort(second half $A$).
              Merge(first half $A$,second half $A$).
              Return.
```

Theorem 1. *The* Merge-Sort *algorithm is correct and runs in time $O(n \log n)$.*

Proof. First let us mention that merging two sorted lists can easily be done in linear time. At each stage, consider the first element of each list. Take the one that is smaller, remove it, and add it to the end of the new list. At the end, the new list will be a sorted list of the elements of the two lists, and this only takes linear time.

Given that Merge is correct and runs in linear time, we can argue that Merge-Sort is correct and runs in $O(n \log n)$ time. Correctness is a simple proof by induction on the size of the list. To

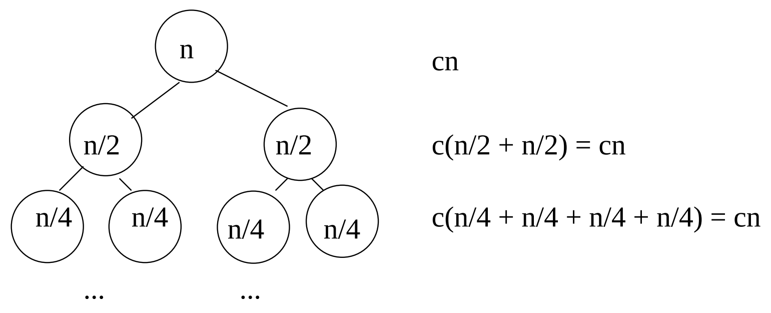

Figure 1: Recursion tree for merge sort. Each circle indicates the size of the array that must be sorted at that point. On the right is the sum of the total cost incurred at each level.

prove the running time, we look at the recursion tree for the algorithm. Figure 1 gives the recursion tree. Notice that the cost at each level is just the cost of merging the sorted arrays that are returned from lower levels. We have already argued that merging the lists is linear in the size of the two lists; therefore, the cost at each level is linear. Also notice that the size of the arrays being considered at each level shrinks by a factor of 2 at each level, and thus the number of levels before reaching an array of size 1 is logarithmic. Hence the total run time is $O(n \log n)$. $\square$

Note that the only operations that MERGE-SORT performs on the elements of the array are comparisions and copy operations. Its running time of $O(n \log n)$ is optimal (up to constant factors) among all such algorithms.

Theorem 2 (Sorting lower bound)**.** *Any sorting algorithm that only accesses the elements of the array through comparisons and copy operations must make take time $\Omega(n \log n)$.*

We refer to the handout on Moodle for a proof of this theorem. An algorithm that does not fall within the scope of the theorem is BUCKET-SORT, which runs in time $O(n + r)$, where r denotes the size of the range of the integers.

3 Selection

Given: an unsorted array A of n integers, and an integer k in the range $1 \leq k \leq n$.

Goal: find the k^{th} smallest element in A, i.e., the k^{th} entry in the sorted version of A.

We denote the solution by $\text{Select}(A, k)$. Interesting special cases include $k = 1$ (minimum), $k = n$ (maximum), and $k = \lceil n/2 \rceil$ (median).

We could just sort the list and select the k^{th} element in $O(n \log n)$ time, but we can do better and achieve a $\Theta(n)$ time algorithm. The algorithm is mainly intended as a didactic example of the divide-and-conquer paradigm with a more complicated recursion tree. The hidden constants make the algorithm unpractical for current applications and systems.

Motivation

Often times finding the median is an intermediate step in the solution of a larger problem. In particular, if we try to solve a problem on an array A of numbers using divide-and-conquer, we may want to split A into a left array L and a right array R such that both are about the same size and all elements of L come before the ones of R. For example, this is what one aims to do in quick-sort. One way to realize it is by figuring out the median m of A, and then going over A, putting the elements less than m in L, those larger than m in R, and equally distributing the ones equal to m over L and R.

In the sequel we denote by L_m the subarray of A of values less than m, and by R_m the subarray of values larger than m.

Divide-and-Conquer Approach

Suppose for a second that we can find the median m of A in linear time, and consider the subarrays $L = L_m$ and $R = R_m$ defined above. If $k \leq |L|$, then $\text{Select}(A, k) = \text{Select}(L, k)$, and we recurse on L_m. If $k \geq n - |R|$, then $\text{Select}(A, k) = \text{Select}(R, \ell)$ for ℓ such that $k = n - |R| + \ell$, so we recurse on R. Otherwise, we know that $\text{Select}(A, k) = m$.

The resulting recursion tree consists of a line where the size of a child is half the size of its parent. As the amount of work associated locally with a node is linear in its size, say at most c times its size, the total amount of work is

$$c \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots) \cdot n \leq 2c \cdot n,$$

where we used the fact that the geometric series with ratio $\frac{1}{2}$ sums to 2.

This approach seems - and is - a vicious circle, in particular if our goal is to find the median of A: In order to find it in linear time, we are assuming that we can find it in linear time…

However, notice that for the correctness of the above procedure, the fact that m is the median of A does not matter – every choice of m would work. The fact that m is the median only comes into play in the analysis of the running time. In this regard, notice that for the above recursive procedure to run in linear time, it actually suffices to guarantee that both subarrays L and R have size at most $\rho \cdot n$ for *some* constant $\rho < 1$; we do not need $\rho = \frac{1}{2}$. Indeed, the expression for the running time then becomes

$$c \cdot (1 + \rho + \rho^2 + \ldots) \cdot n \leq c \cdot \frac{1}{1 - \rho} \cdot n,$$

where we used the general expression for the sum of a geometric series with ratio $\rho < 1$. Since ρ is a constant, this is still $O(n)$, although the constant hidden in the Big-Oh notation is larger.

Thus, it would suffice to find an *approximate* median m' of A in linear time, and use m' instead of m to split the array A into L and R. By an approximate median we mean an element m' such that at most $\rho \cdot n$ elements of A are smaller than m' (the set L) and at most $\rho \cdot n$ are larger (the set R), for some constant $\rho < 1$.

Finding an approximate median

Here is the key insight. Consider breaking up the array A into consecutive segments of length w, for a constant w to be determined later. At the end, there may be one segment of length less than w left. Determine for each of these segments their median, and let A' be the array consisting of these $\lceil n/w \rceil$ medians.

Claim 1. *The median of A' is an approximate median of A with $\rho = 3/4$.*

Proof. Let m' denote the median of A'. At least half of the elements x' of A' satisfy $x' \leq m'$. As each of these elements x' are the median of their respective segments, at least half of the elements x in the segment of x' satisfy $x \leq x'$. Since the segments are disjoint, this means that at least a fraction $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$ of the elements x of A satisfy $x \leq m'$. Thus, the set R of elements that exceed m' satisfies $|R| \leq (1 - \frac{1}{4}) \cdot n = \rho \cdot n$ for $\rho = \frac{3}{4}$. A similar argument shows that the set L of elements that are smaller than m' satisfies $|L| \leq \rho \cdot n$. □

How do we find the median of A'? First, we need to construct A', i.e., we need to find the median of each segment. As the length of the segments is bounded by the constant w, we can find each individual median in constant time, resulting in $O(n)$ time overall to construct A'. Note that A' is shorter than A, so once we have constructed A', we can find its median by making *another recursive call* to our selection procedure.

Note that this isn't exactly what we set out to do, which was to design a (separate) linear-time procedure to find an approximate median of A. Instead, we compute an approximate median of A by computing the exact median of an array A' of smaller size, and do so by making a recursive call the the procedure we're designing. This also means we need to redo the analysis of the overall procedure, which becomes more complicated.

Final algorithm and analysis

Our final divide-and-conquer algorithm makes *two* recursive calls to the selection procedure, and works as follows on a non-trivial input A:

1. First, we construct the array A' and make a recursive call to find the median m' of A'. The size of this recursive call is $|A'| = \lceil n/w \rceil$.

2. Next, we use m' to construct the sets $L = L_{m'}$ and $R = R_{m'}$ as before, recursively call the selection procedure on either L or R (or neither), and return that answer (or m'). The size of this recursive call is at most $\rho \cdot n$.

The resulting recurrence for the running time $T(n)$ of our procedure for the selection problem is

$$T(n) \leq T\left(\frac{n}{w}\right) + T(\rho \cdot n) + c \cdot n,$$

where $\rho = 3/4$ and c is some constant. One can solve this recurrence explicitly, but we analyze the amount of work by aggregating the local work per level of recursion. The amount of local work associated with a node of size s is $c \cdot s$. We start with a root of size n. The sum of the sizes of the nodes at any subsequent level is at most

$$\alpha \doteq \frac{1}{w} + \rho$$

times the sum of the sizes at the level above it. Since the amount of local work attributed to a node is linear in its size, the bound on the total amount of work per level of recursion is also multiplied by an additional factor of α per level. This results in the following expression for the total amount of work:

$$c \cdot (1 + \alpha + \alpha^2 + \ldots) \cdot n, \tag{1}$$

where the number of terms is $O(\log n)$.

- For $\alpha < 1$, the amount of work reduces by a constant factor at every level and is therefore dominated by the amount of work at the top level. The underlying geometric series is converging, and the resulting running time is $O(n)$.
- For $\alpha = 1$, our bound on the amount of work per level of recursion is $c \cdot n$ at every level, resulting in a bound of $O(n \log n)$ on the running time.
- For $\alpha > 1$, the amount of work per level grows by a constant factor (up to the point where some branches die out, which happens after a logarithmic number of levels), and the total amount of work is dominated by the lower levels. The resulting upper bound on the running time is worse than $O(n \log n)$.

Note that $\alpha < 1$ if and only if $w > 4$. Thus, if we pick $w = 5$, the resulting selection algorithm runs in linear time!

Larger values of w result in smaller values of α but larger values of c in (1), the latter due to the extra work needed to construct A'. The optimal trade-off depends on the implementation details and the system parameters.