CS430

LECTURE NOTE NO.3

DIVIDE AND CONQUER

A classical approach where a problem is divided into sub-problems, each of smaller size, since typically solving problems of smaller size are easier to solve. We illustrate this method using sorting.

1 Sorting and Order statistics

Suppose we have to sort a list L of n integers. We devise two divide and conquer schemes for this problem.

1.1 Mergesort

The first algorithm is termed *MERGESORT*.

Algo MERGESORT (L)

1 Split L into two lists, L_1 and L_2.

2 Recursively sort L_1 and L_2.

3 Merge the sorted lists L_1 and L_2.

To merge the sorted lists we use the following procedure:

Algo MERGE(L_1, L_2, L)

1. If both lists are non-empty append the smallest of the first items of L_1 and L_2 to L.
2. If one list is empty simply append the other list to L.
3. Repeat the above steps until both L_1 and L_2 are empty.

The algorithm above requires $O(n \log n)$ steps. But requires extra space for construction of the new list L.

1.2 Quicksort

Another algorithm for searching first separates the numbers and sorts recursively. We assume that L is stored in an array $A[1..n]$.

Algo QUICKSORT(L)

1. Let p be a partitioning element.
2. Partition L into two parts L_1 and L_2 such that all numbers in L_1 are less than p and all numbers in L_2 are greater than or equal to p.
3. Sort L_1 and L_2 recursively.

The partitioning is done by using two pointers, l and r. Initially $l = 1$ and $r = n$. Assume that all numbers are unique. The steps are

(i) Increase l until $A[l] \geq p$ or $l = r$.

(ii) Decrease l until $A[r] < p$ or $r = l$.

(iii) If $l < r$, swap current items in $A[l]$ and $A[r]$.

(iv) Repeat the above steps until $l \geq r$.

(v) If the item at $A[l]$ is not p, swap p with $A[l]$

The algorithm requires $O(n^2)$ steps in the worst case if the partitioning element is chosen arbitrarily.

If p is the median then the algorithm requires $O(n \log n)$ steps. We next analyze the behavior of the algorithm when p is randomnly chosen.

We let $T_a(n)$ represent the average time complexity of sorting n elements. The following recurrence characterizes $T_a(n)$. Here we assume that the partition item is not part of the recurrences. Thus if the partition item is in the first position we get two recursive problems, one of size 0 and the other of size $n - 1$. Similarly if the partition item is in the ith position we get problem arrays, one of size $i - 1$ and the other of size $n - i$. Assuming the the partition item occurs in each of these posiions with equal probability we get:

$$T_a(n) = c.n + 1/n\Sigma_{0 \leq i \leq n-1}\{T_a(i) + T_a(n - i - 1)\}$$

together with the initial conditions

$$T_a(1) = 1$$
$$T_a(0) = 0$$

The recurrence can be expressed as

$$T_a(n) = c.n + 2/n\Sigma_{0 \leq i \leq n-1}T_a(i)$$

We rewrite the above recurrence as

$$nT_a(n) = c.n^2 + 2\Sigma_{0 \leq i \leq n-1}T_a(i)$$

Noting that

$$(n - 1)T_a(n - 1) = c.(n - 1)^2 + 2\Sigma_{0 \leq i \leq n-2}T_a(i)$$

we get

$$nT_a(n) - (n - 1)T_a(n - 1) = c.n^2 - c.(n - 1)^2 + 2T_a(n - 1)$$

or

$$nT_a(n) = 2cn - c + 2(n+1)T_a(n-1)$$

Dividing this by $n(n+1)$ we obtain the recurrence

$$\frac{T_a(n)}{n+1} = \frac{2cn-c}{n(n+1)} + \frac{T_a(n-1)}{n}$$

Rewrite $T_a(n)/(n+1)$ as $T'(n)$ we get the following recurrence for T'

$$T'(n) = \frac{2cn-c}{n(n+1)} + T'(n-1)$$

or

$$T'(n) = \frac{3c}{n+1} - \frac{c}{n} + T'(n-1)$$

Summing this recurrence for $1 \leq i \leq n$ gives a telescoping series and $T'(n) = O(H(n)))$ where $H(n)$ is $O(\ln n)$ and thus $T_a(n)$ is shown to be $O(n \log n)$. (details left as an exercise to the reader)

1.3 Another Analysis

In this section we will present a different analysis of the Quicksort sorting algorithm.

Suppose we are sorting $x_1, x_2 \ldots x_n$. We will count the expected number of comparisions. We will use $X(ij)$ as an indicator function, i.e. $X(ij) = 1$ if the element of rank i, x_i is compared with the element of rank j, x_j where $j > i$ and $X(ij) = 0$ otherwise.

When are two elements compared? The comparisons occur only when either x_i or x_j are chosen as a partition element with x_i and x_j belonging to the same partition. If both belong to the same partition then all the elements of rank betwen i and j also belong to the same partition. Thus the probability of choosing either x_i or x_j is atmost $2/(j-i+1)$ since at least $j-i+1$ elements are present in the partition.

Thus expected number of comparision made during Quicksort is

$$\Sigma_i \Sigma_{j>i} \frac{2}{j-i+1}$$

which is $O(nlogn)$.

1.4 HeapSort

This sorting method uses a data structure called the *binary heap*, which implements a priority queue.

A priority queue is a data structure that maintains an ordered set of items S. We assume that each item has an associated key, intermed $KEY(s)$, $s \in S$ and is ordered by these KEY values. The priority key allows one to perform the following operations:

- FIND-MIN: Find the item with the smallest value of KEY, i,e $MIN{-}KEY = \min_s KEY(s), s \in S$.
- DELETE-MIN: delete the item with minimum value of KEY.

- DECREASE-KEY: decrease the value of $KEY(s)$ for a item s.

A symmetric version where the maximum value is used is also helpful. In fact a MAX-HEAP can also be used for heapsort. For sorting, we will primarily use the second type of operation, $DELETE - MAX$. Given this operation, the sorting algorithm is reasonably simple. Simply find and remove the item with the maximum value of KEY. This will generate a sequence of items sorted in increasing order of their KEY values.

The binary heap is a complete binary tree where all the leaf nodes are at two successive levels. An implementation of the arrray stores the complete binary tree in an array. The root of the tree is stored at $A[1]$, Let a tree node v be stored at $A[i]$. Then the indices of $v's$ parent, $Parent(v)$ is

$$Parent(v) = \lfloor i/2 \rfloor$$

and that of the left and right child, $Left(v)$ and $Right(v)$ are:

$$Left(v) = 2i$$

$$Right(v) = 2i + 1$$

In a max-heap, the max-heap property is that for every node i

$$A[Parent(v)] \geq A[v]$$

or

$$A[\lfloor i/2 \rfloor] \geq A[i]$$

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no greater than that contained at the node itself.

The heapsort algorithm starts by a procedure $BUILD\text{-}HEAP$ to build a max-heap on the input array $A[1..n]$. The procedure $BUILD\text{-}HEAP$ utilizes a procedure $HEAPIFY$.

Consider a node v in the tree stored at $A[i]$, where the heap property is violated. Further assume that the heap property is true at both child nodes. $HEAPIFY$ compares $A[i]$ with $A[2i]$ and $A[2i+1]$ and the largest of the values in $A[2i]$ and $A[2i+1]$ is interchanged with the value at $A[i]$. Thus now v satisfies the heap property but one of its child nodes, say $Left(v)$ may not. In which case, $HEAPIFY$ is called recursively at that node. The recursive call will terminate when a leaf node of the tree is reached, in which case that node trivially satisfies the heap property. Thus $O(logn)$ steps are required to correct a violation of the heap property at a node.

In order to construct the heap, $BUILD-HEAP$ calls $HEAPIFY$ for all values of i ranging down from $\lfloor n/2 \rfloor$ to 1.

Once the heap has been built, the maximum element of the array is available at the root $A[1]$. This element can be placed at $A[n]$, exchanging $A[1]$ with $A[n]$. The heap specified by $A[1 \ldots n-1]$ violates the heap property at the root. $HEAPIFY$ can then update this property in $O(\log n)$ steps. We now have a max-heap $A[1 \ldots (n-1)]$ and the above process of exchanging the first and last item of the arrow and and executing $HEAPIFY$ can be repeated until the heap is reduced to a size of 1. At this stage the array is sorted.

Given that $HEAPIFY$ is called at most n times, HEAPSORT takes time $O(n \lg n)$. The intialization, $BUILD-HEAP$ takes time $O(n)$. This can be seen as follows: When $HEAPIFY$ is called at a node which is height l (or distance l from a leaf node) it requires l steps. The

number of nodes at height l in a completely balanced tree is bounded by $O(n/2^l)$ Thus the total work is:

$$\Sigma_{1 \le l \le \log n} \frac{n}{2^l} l$$

which is $O(n)$.

1.5 Lower Bound on Sorting

We next show that in the comparison tree model, sorting requries $\Omega(\log n)$ steps.

All of the sorting algorithms that we have seen have been based on element comparisons $a_i : a_j$ of the items being sorted; all of them use $\Omega(n \log n)$ such comparisons. This is no coincidence: Any sorting algorithm based on comaprisons of the elements being sorted must use that many element comparisons in both the worst case and on the average.

In this discussion we will consider only the case in which no two elements are equal, so such a comparison results in either a $>$ or $<$ answer, never $=$. Any sorting algorithm that works correctly for all inputs must work for this type of input.

For a fixed value of n, we can take such a sorting algorithm and expand it into a binary decision tree in which internal nodes are the comparisons and leaves are the sorted elements—that is, a leaf corresponds to a permutation of the input elements. For the algorithm to sort correctly, each of the possible $n!$ permutations must correspond to a unique leaf (if two or more permutations share a leaf, the algorithm cannot have determined the unique sorted order; if some permutation is missing, the algorithm cannot sort correctly on the corresponding input order).

1.6 Worst Case

Notice that the height of the decision tree is the number of comparisons used by the algorithm in the worst case. Let $S(n)$ denote the minimum number of comparisons required in the worst case by *any* sorting algorithm based on element comparisons. A binary tree of height h can have at most 2^h leaves, so

$$2^{S(n)} \ge n!$$

and hence

$$S(n) \ge \lg n! = \Omega(n \log n).$$

Thus *any* sorting algorithm based on element comparisons must use $\lg n!$ in the worst case.

1.7 Average Case

Now consider the minimum number of comparisons required in the average case by any sorting algorithm based on element comparisons. The average number of comparisons used is the average of the depths of all leaves: if we define $EPL(T)$, the *external path length* of a binary tree T, as the sum of the depths of all leaves, then $EPL(T)/n!$ gives the average number of comparisons used by the sorting algorithm represented by the decision tree T.

Lemma 1.1. *The binary tree T with the least external path length $EPL(T)$ has all of its leaves on levels l and $l+1$ for some value of l.*

Proof. Suppose not; let T be a binary tree of n leaves with minimal external path length with lowest (deepest) leaf at level L and shallowest (highest) leaf at level $l < L - 1$. Remove two sibling leaves from level L, making their parent internal node into a leaf; replace a leaf at level l with and internal node and two leaves as children. We now have a new tree T' of n leaves satisfying

$$EPL(T') = EPL(T) - 2L + 2(l + 1) < EPL(T)$$

because $l < L-1$ implies $-2L+2(l+1) < 0$. That is, $EPL(T')$ is less than the smallest possible external path length in a tree with N leaves, contradicting our choice of T as a binary tree with minimal external path length. □

Lemma 1.2. *If $l_1, l_2, \ldots, l_N$ are the depths of the leaves in a binary tree, then*

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1.$$

This is called Kraft's Inequality.

Proof. By induction. □

Now we can compute the minimum external path length in a binary tree with N leaves. Suppose, by the first lemma, that there are k leaves at level l and $N - k$ leaves at level $l + 1$, $1 \leq k \leq N$ (that is, all leaves may be at level l with none on level $l + 1$). The second lemma tells us that

$$k2^{-l} + (N - k)2^{-l-1} = 1$$

and hence

$$k = 2^{l+1} - N.$$

But $k \geq 1$ so that $2^{l+1} > N$; similarly, $k \leq N$ so $2^l \leq N$. Thus

$$l = \lfloor \lg N \rfloor.$$

Then $k = 2^{l+1} - N$ gives

$$k = 2^{\lfloor \lg N \rfloor + 1} - N$$

and the minimal external path length is thus

$$lk + (l + 1)(N - k) = N\lfloor \lg N \rfloor + 2N - 2^{\lfloor \lg N \rfloor + 1}.$$

Define $\delta(N) = \lg N - \lfloor \lg N \rfloor$ and hence $0 \leq \delta(N) < 1$. The minimal external path length is then

$$N \lg N + N(2 - \delta(N) - 2^{1-\delta(N)}).$$

The function $2 - \delta(N) - 2^{1-\delta(N)}$ is small on the interval $[0, 1]$; specifically, $0 \leq \delta(N) \leq 0.0861$ on this interval.

We conclude that for a binary tree with $N = n!$ leaves, the external path length, and hence the minimum possible average sorting time (element comparisons) must be at least $EPL(T)/n!$ where T is a tree of $n!$ leaves of least external path length; in other words, $\lg n!$.

2 Sorting in Linear Time

The $\Omega(n \log n)$ lower bounds we just proved do not apply if we can avoid element comparisons. For example, if we know we are sorting a permutation of the numbers $1, \ldots, n$, then as we encounter each number we can put it directly into its correct place in sorted order.

2.1 Radix Sorting:

In radix sorting the representation of the numbers is utilized to speed up the sorting. Suppose each number x is represented in base r as $x_d x_{d-1} \ldots x_1$, where n_i is atmost r. We sort the numbers considering the digits starting from the least significant digit.

Algorithm Radix-Sort
begin
    Let the input list be L.
    for i= 1 to d do
    begin
        Distribute the numbers into lists $L_1, L_2 \ldots L_r$
        based on the ith digit, preserving the order in L within L_j.
        Concatenate the lists to form the complete list, i.e.
        $L \leftarrow L_1, L_2 \ldots L_r$.
    end
    Output the list L.
end.

The above algorithm correctly sorts the lists. This can be proven by showing that the following invariant is valid at the iith iteration of the loop.

Property 2.1. *At the ith iteration the list, L, is sorted with respect to digits $1, 2 \ldots i$.*

We prove this by induction on the number of iterations. Assume that the property holds for upto the $i-1$st iteration. At the ith iteration the numbers are arreeanged in order of their ith digit. For numbers which have the ith digit the same, the numbers are sorted in order of digits up to the $i-1$st digit. This is because the order within a list, say L_j is preserved when the numbers are distributed into sub-lists based on their ith digit. Thus at the end of the ith iteration the numbers are sorted by digits $1, 2, \ldots i$.

The time complexity of this algorithm is $O(nd)$. This is because at each iteration we scan the list to determine the distribution into sub-lists. This scan is done once. There are d iterations. The bound follows.

2.2 Median Statistics

In this section we show how to determine the median of a set, S, of n numbers. We will use divide and conquer to find the median in $O(n)$ steps.

The algorithm we design is more general and designed to find the kth element. The algorithm is similar to quicksort. A partition element is chosen. Let this element have rank $k, k \geq n/2$.

Then we need to recursively search in a list of size k in the worst case. This leads to the recurrence

$$T(n) = T(k) + cn$$

The solution to this recurrence depends on k. If k is large the algorithm would have worst case complexity of $O(n^2)$. However if we can determine a partition element which ensures that $k = cn$ for some constant c then we can achieve $O(n)$ time complexity.

To choose a good partition element we consider a sampling strategy. We partition the set of elements into $n/(2k+1)$ sets each of size $2k+1$. For each set we find the median element. Consider the set of medians MED. The median element of this set, MOM, will provide a very good partition element. This is because the number of items in MED that are less or greater than this element is at least $\frac{n}{2(2k+1)}$. And the number of items less or greater than this in S is atleast $\frac{(k+1)n}{4k+2}$. Thus the partition generated using p is of size $n(1-\frac{1}{4+2k})$. To determine MOM we recursively apply the algorithm.

The time complexity can be evaluated via the following recurrence:

$$T(n) = T(n(1 - \frac{(k+1)}{(4k+2)})) + T(\frac{n}{(2k+1)}) + cn$$

for some constant c. This is linear for some reasonable k. Picking k=2 gives us the recurrence

$$T(n) = T(7n/10) + T(n/5) + cn$$

This can be proven to have a solution $10cn$.

3 Closest Pair Problem

Given a set, S, of points in 2-d space, the closest pair problem is to find the pair of points such that the distance between the two points is the minimum over all interpoint distances, i.e find p_1 and p_2 such that

$$d(p_1, p_2) = \min_{p,q} |\{d(p,q)|p, q \in S\}$$

where $d(p,q)$ is the euclidean distance between p and q.

We solve this proble using divide and conquer. Suppose the points are sorted by their x and y-coordinates. The first step is to divide the point set S into sets S_1 and S_2 where $S_1 = \{p|x(p) \leq x_m\}$ and $S_2 = \{p|x(p) > x_m\}$ where x_m is the median of the x-coordinates of points in S. Recursively find the closest pair in S_1 and in S_2. Let δ_1 be the closest pair distance in S_1 and let δ_2 be the closest pair distance in S_2. Let $\delta = \min(\delta_1, \delta_2)$. It now remains to find the closest pair with one point in S_1 and the other in S_2 such that the distance between the closest such pair is less than δ. We call such a pair the closest across pair.

This is accomplished as follows. Let S_l be the set of points in S_1 which lie in a vertical region R_1 defined as $\{p|x_m - \delta \leq x(p) \leq x_m\}$ and let S_r be the set of points in the region R_2 defined as $\{p|x_m \leq x(p) \leq x_m + \delta$. Assume that points in S_l and S_r are sorted by their y-coordinates and are available in a list of the same name. This sorting is easily obtained from the set S sorted by y-coordoinate in linear time. Furthermore let $pos(q), q \in S_l$ be the position of point q in the list S_r, i.e $pos(q)$ is the position of the point in S_2 with y-coordinate just greater than that of q. This position is easily determined by merging the two lists S_l and S_r. The following algorithm now finds the closest across pair.

Algorithm $CAP(S_l, S_r)$

Begin

Let $S_l = \{p_1, p_2 \ldots p_k\}$

For i = 1to k do

Find point closest to p_i amongst points

$C(p_i) = \{q | q \in S_r, |y(q) - y(p_i)| \leq \delta\}$

$CAP \leftarrow$ closest pair found above

end.

The correctness of the above algorithm is easy and left to the reader.

Let us now consider the gains obtained in time complexity by using divide and conquer. We are going to show that the time complexity of algorithm CAP is $O(n)$. We need the following lemma:

Lemma 3.1. *Let* $C(p_i) = \{q | q \in S_r, |y(q) - y(p_i)| \leq \delta\}$. *Then* $|C(p_i)| \leq 6$.

The proof of this is left as a geometric exercise for the reader.

Thus as the above lemma shows, the point in S_r which is closest to a point p in in S_l and less than δ distance away can be found in $O(1)$ time. Thus CAP requires $O(n)$ steps. This give a bound of $O(n \log n)$ steps for solving the closest pair problem.

4 Acknowledgements

The section on Lower Bounds is from notes by Ed. Reingold.