[Divide and Conquer Paradigm]

Design and Analysis of Algorithms (CSc 523)

Samujjwal Bhandari
Central Department of Computer Science and Information Technology (CDCSIT)
Tribhuvan University, Kirtipur,
Kathmandu, Nepal.

In the ancient time Roman politicians used the concept of dividing the enemy unity somehow to conquer them. The method they used was not conceived for algorithms. While designing any algorithm if we try to break the problem of larger size into smaller then we adhere to the designing technique called "Divide and Conquer". Most of the computational problems can be solved by using this technique. Analysis of such an algorithm developed using divide and conquer paradigm needs recurrence since divide and conquer algorithms use recursion.

The main elements of the Divide and Conquer Solutions are:

Divide: Divide the larger problem into the problem of smaller size.

Conquer: Solve each piece by applying recursive divide and conquer.

Combine: add up all the solved pieces to a larger solution.

Binary Search

Algorithm: An array of elements A[] has sorted elements (here in nondecreasing order)

```
BinarySearch(low, high, key){
        if(high = = low){
        if(key = = A[low]) then return low+1; //index starts from 0
        else return 0;
        }
        else{
        mid = (low + high) /2 ; //integer division
        if(key = = A[mid] then return mid+1;
        else if (key < A[mid]) then return BinarySearch(low, mid-1, key) ;
        else return BinarySearch(mid+1, high, key) ;
        }
}
```

We can see that the above algorithm terminates whenever key value is found or both the index for an array low and high equals. If result is found it is returned otherwise 0 is returned. (Students should verify the correctness in detail).

Efficiency:

From the above algorithm we can say that the running time of the algorithm is

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ &= \Theta(\log n) \text{ (verify).} \end{aligned}$$

In the best case output is obtained at one run i.e. $\Theta(1)$ time if the key is at middle.

In the worst case the output is at the end of the array so running time is $\Theta(\log n)$ time.

In the average case also running time is $\Theta(\log n)$.

For unsuccessful search best, worst and average time complexity is $\Theta(\log n)$.

Running example

Take input array A[] = {2 , 5 , 7, 9 ,18, 45 ,53, 59, 67, 72, 88, 95, 101, 104}

For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since A[mid] = = 2, return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition high = = low, since A[0] != 103, return 0(unsuccessful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since A[mid] = 67, return 9(successful).

Fast Exponentiation

Here our aim is to calculate the value of a^n. In simple way we can do this in by n-1 multiplication. However, we can improve the running time by using divide and conquer strategy. Here we can break n into two parts i and j such that we have n = i + j. So we can write $a^n = a^i * a^j$. if we use i = $\lfloor n/2 \rfloor$ then we can have, if n is even then $a^n = a^i * a^i$, and if n is odd $a^n = a^i * a^j * a$.

Algorithm:

```
Fastexp(a,n)
{
        if(n = = 1)
                return a;
        else
                x = Fastexp(a, $\lfloor n/2 \rfloor$);
                if(n is odd)
                        return x*x*a;
                else
                        return x*x;
}
```

Correctness:

In the above algorithm every time the value of x is a^i where i = $\lfloor n/2 \rfloor$, in both the conditions returned value will be a^n. hence by induction it follows.

Efficiency:

Observe the above algorithm then you will find that each time problem is divided into approximately half part and the cost of dividing and combining the problem constant. So we can write time complexity as

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Solving this relation we get Θ(logn).

If we concern about space then the stacks are called upto logn time hence Θ(logn).

Max and Min Finding

Here our problem is to find the minimum and maximum items in a set of n elements. We see two methods here first one is iterative version and the next one uses divide and conquer strategy to solve the problem.

Algorithm:

```
IterMinMax(A,n)
{
    max  = min = A[0];
    for(i = 1; i < n; i++)
    {
        if(A[i] > max)
            max = A[i];
        if(A[i] < min)
            min = A[i];
    }
}
```

The above algorithm requires 2(n-1) comparison in worst, best, and average cases Since the comparison A[i] < min is needed only A[i] > max is not true. if we replace the content inside the for loop by

```
    if(A[i] > max)
        max = A[i];
    else if(A[i] < min)
        min = A[i];
```

Then the best case occurs when the elements are in increasing order with (n-1) comparisons and worst case occurs when elements are in decreasing order with 2(n-1) comparisons. For the average case A[i] > max is about half of the time so number of comparisons is 3n/2 – 1.

We can clearly conclude that the time complexity is $\Theta(n)$.

Now let us see the divide and conquer strategy to solve the problem.

Algorithm:

```
MinMax(i,j,max,min)
{
    if(i = = j)
        max = min = A[i];
    else if(i = j-1)
    {
        if(A[i] < A[j])
        {
            max = A[j]; min = A[i];
        }
        else
        {
            max = A[i]; min = A[j];
        }
    }
    else
    {
        //Divide the problems
        mid = (i + j)/2;            //integer division
        //solve the subproblems
        MinMax(i,mid,max,min);
        MinMax(mid +1,j,max1,min1);
        //Combine the solutions
        if(max1 > max)      max = max1;
        if(min1 < min)      min = min1;
    }
}
```

The above algorithm adopts the following idea; if the number of elements is 1 or 2 then max and min are obtained trivially. Otherwise split problem into approximately equal part and solved recursively.

Analysis:

Here we analyze in terms of number of comparisons as cost because elements may be polynomials, strings, real numbers, etc. Now we can give recurrence relation as below for MinMax algorithm in terms of number of comparisons.

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2$, if $n>2$

$T(n) = 1$, if $n = 2$

$T(n) = 0$, if $n = 1$

We can simplify above relation to

$T(n) = 2T(n/2) + \Theta(1)$.

Solving the recurrence by using master method complexity is (case 1) $\Theta(n)$.

Note: Both the algorithms we have studied above have $\Theta(n)$ complexity but due to space overhead in divide and conquer approach we prefer iterative one.

Integer Multiplication

Here our problem definition gives

Inputs:

$X = x_{n-1}, x_{n-2}, \ldots, x_0$.

$Y = y_{n-1}, y_{n-2}, \ldots, y_0$. Here X and Y are n digit positive numbers.

Output:

$Z = z_{2n-1}, z_{2n-3}, \ldots, z_0$. Here Z is product X*Y of digits 2n.

Our school method solves this problem in O(n^2) time (how?). But we use divide and conquer approach algorithm (by A.A. Karatsuba, in 1962) to solve the problem asymptotically faster. The number can be of any base but for simplicity assume decimal numbers. Now we can have

$$X = \sum_{i=0}^{n-1} (x_i)*10^i \text{ and } Y = \sum_{i=0}^{n-1} (y_i)*10^i$$

So that Z = X*Y is

$$\sum_{i=0}^{2n-1} (z_i)*10^i = \sum_{i=0}^{n-1} (x_i)*10^i * \sum_{i=0}^{n-1} (y_i)*10^i$$

For example

$X = 141 = 1*10^2 + 4*10 + 1*10^0$; $Y = 123 = 1*10^2 + 2*10 + 3*10^0$.

$Z = 1*10^4 + 7*10^3 + 3*10^2 + 4*10 + 3*10^0. = 17343$

Now if we suppose,

$A = x_{n-1}, x_{n-2}, \ldots, x_{n/2}$.

$B = x_{(n/2)-1}, x_{(n/2)-2}, \ldots, x_0$.

$C = y_{n-1}, y_{n-2}, \ldots, y_{n/2}$.

$D = y_{(n/2)-1}, y_{(n/2)-2}, \ldots, y_0$.

Then we have

$X = A*10^{(n/2)} + B$

$Y = C*10^{(n/2)} + D$, and

$Z = (A*10^{(n/2)} + B)* (C*10^{(n/2)} + D)$

$= A*C*10^n + (A*D + B*C)* 10^{n/2} + B*D$.

For example if X = 2345, Y = 5684 , A = 23, B = 45, C = 56, D = 84 then

$X = 23*10^2 + 45$

$Y = 56*10^2 + 84$ then

$Z = 23*56*10^4 + (23*84 + 45*56)* 10^2 + 45*84 = 13328980 = 2345*5684$.

The terms *(A*C)*, *(A*D)*, *(B*C)*, and *(B*D)* are each products of 2 (n/2)-digit numbers. From the above facts we can generate an idea that can be applied to design an algorithm for integer multiplication. The main idea is

If the two numbers are of single digit can be multiplied immediately (Boundary).
If n > 1 then the product of 2 n digit numbers can be divided into 4 products of n/2 digit numbers (Divide and Conquer).
Calculating product of two numbers requires additions of 4 products that can be done in linear time and multiplication by the power of 10 that can also be done in linear time (Combine). (Devising algorithm as an exercise #1)

Our above discussion needs the representation of an algorithm for multiplying 2 n digit numbers by 4 products of n/2 digit number, however Karatsuba discovered how the product of 2 *n*-digit numbers could be expressed in terms of three products each of 2 *n/2* digit numbers. This concept however increases the number of steps required in combine process though the complexity is O(n).

Suppose

$U = A*C$

$V = B*D$

$W = (A+B)*(C+D)$

Then we have $A*D + B*C = W - U - V$ so

$$Z = X*Y = A*C*10^n + (A*D + B*C)* 10^{n/2} + B*D.$$
$$= U*10^n + (W - U - V)* 10^{n/2} + V.$$

Algorithm: A, B, C, D, U, V, W are as defined in our discussion

```
ProdInt(X,Y,n)
{
        if( n = = 1)
                return X[0]*Y[0];
        else
        {
                A = X[n-1] ...X[n/2];
                B = X[(n/2)-1] ... X[0];
                C = Y[n-1] ...Y[n/2];
                D = Y[(n/2)-1] ... Y[0];
                U = ProdInt(A,C,n/2);
                V = ProdInt(B,D,n/2);
                W = ProdInt((A+B),(C+D),n/2);
                return $U*10^n + (W – U – V)* 10^{n/2} + V$;
        }
}
```

Analysis:

From the above algorithm we can give recurrence relation as

$T(n) = 3T(n/2) + O(n)$.

Solving this recurrence by using master method where a = 3, b= 2, we get

$T(n) = \Theta(n^{1.58})$.

Quick Sort

Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Steps for Quick Sort

Divide: partition the array into two nonempty sub arrays.

Conquer: two sub arrays are sorted recursively.

Combine: two sub arrays are already sorted in place so no need to combine.

Definition: Partitioning

An array A[] is said to be partitioned about A[t] if all the elements on the left of A[t] are less than or equal to and the all the elements on the right are greater than or equal to A[t].

Example: A[] = { 3, 7, 6 15, 14 18, 25, 55, 32, 45, 37}

Here the array A[] is partitioned about A[5](index 0, 1, 2, ….).

Algorithm:

```
Partition(A,i,j)
{
    x = i -1; y = j + 1; v = A[i];
    while(x<y)
    {
    do {
        x++;
    }while(A[x] <= v);
    do {
        y--;
    } while(A[y] >= v);
    if(x<y)
        swap(A[x],A[y]);
    }
    A[i] = A[y]; A[y] = v;
    return y;
}
```

Analysis:

In the above algorithm either left pointer or right pointer moves at a time and scans the array at most 1 time. You can note that at most n swaps are done hence the time complexity for above algorithm is $\Theta(n)$.

Running Example

Take an array A[] = { 16, 7, 6 15, 14 18, 25, 55, 32, 45, 37}

```
(v) 16   7   6   15   14(y)   (x) 18   25   55   32   45   37
        A[i] = A[4]; A[4] = 16; i is 0 here

[14   7   6   15]   16   [18   25   55   32   45   37]
                Partitioned sub arrays.
```

Algorithm:

```
QuickSort(A,p,q)
{
    if(p<q)
    {
        r = Partition(A,p,q);
        QuickSort(A,p,r-1);
        QuickSort(A,r+1,q);
    }
}
```

Analysis:

The quick sort algorithm time complexity i.e. the time to sort an array A[] can be written as the recurrence relation

$T(n) = T(k-1) + T(n-k) + \Theta(n)$.

Where k is the result of Partition(A,1,n)

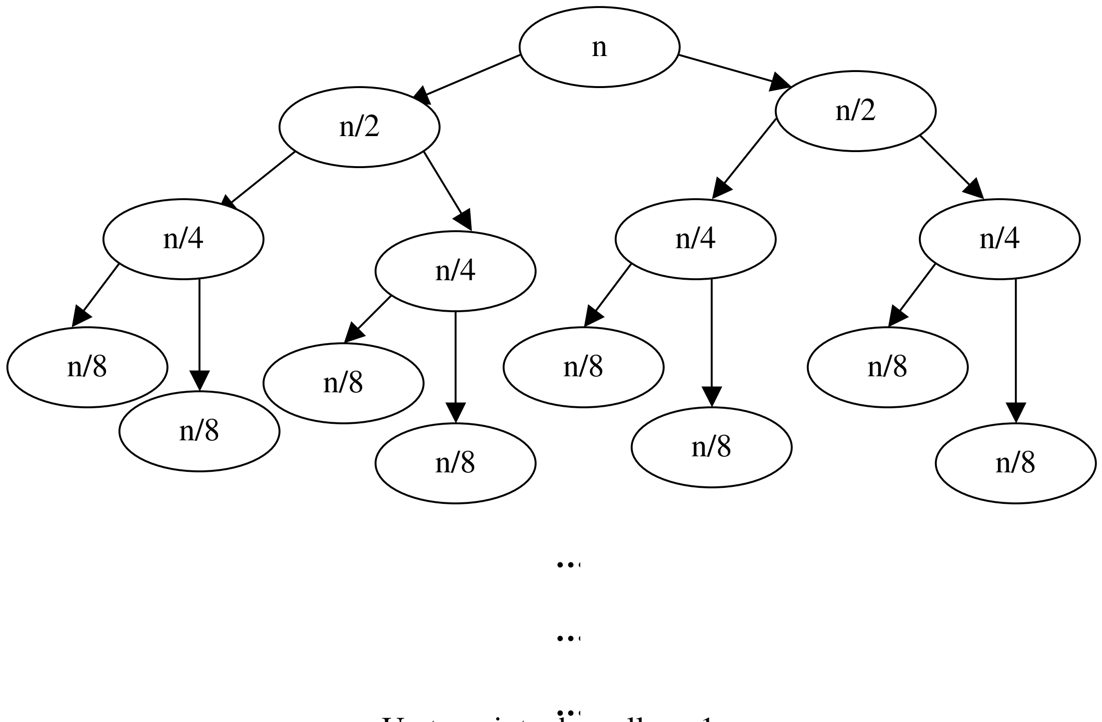
Best Case:

The best case for divide and conquer relation occurs when division is as balanced as possible. Here best base occurs if we can chose pivot as the middle part of the array at all time, so we have

$T(n) = 2T(n/2) + \Theta(n)$.

Solving this recurrence we get $T(n) = \Theta(n\log n)$.

The tree for best case is like



Up to point when all are 1

This gives the total sum of the nodes as $\Theta(n\log n)$(verify!).

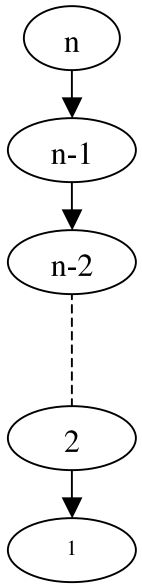
Worst Case:

Worst case occurs if the partition gives the pivot as first element (last element) all the time i.e. k =1 or k = n this happens when the elements are completely sorted, so we have

$T(n) = T(n-1) + \Theta(n)$.

Solving this relation we get $\Theta(n^2)$.

The tree for worst case is like



Adding all the levels we get

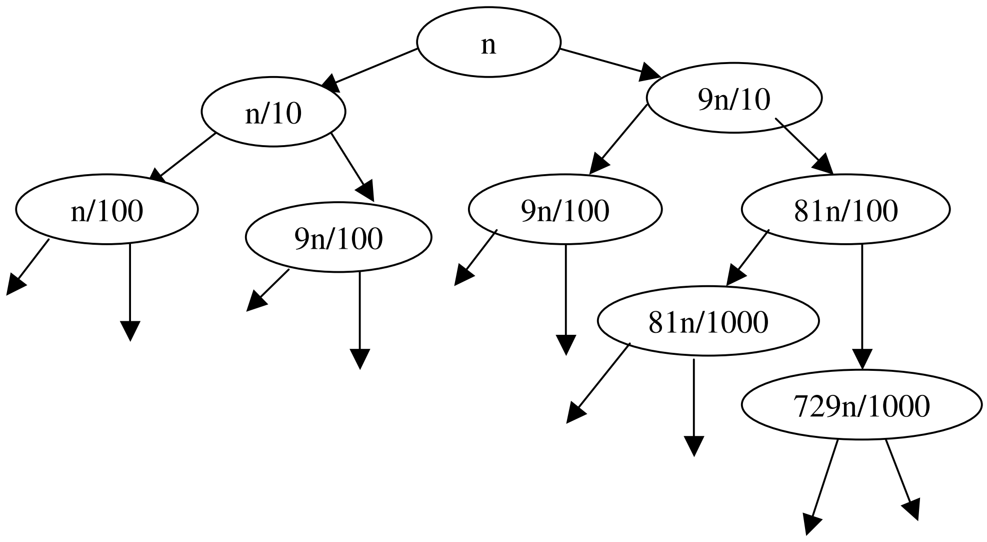
$1+2+3+ \dots + n = \Theta(n^2)$(verify!).

Case when the partition is balanced but not as of best-case partition

Lets suppose that partitioning algorithm always produce 9:1 split of the array. See here 9:1 seems unbalanced. For this situation we can write recurrence relation as

$T(n) = T(9n/10) + T(n/10) + \Theta(n)$.

Then we can show that this is $\Theta(n\log n)$ from the below recurrence tree



Adding up all the levels we get

$n + n + n + (<=n) + (<=n) + \dots + 1$

Here when the height $\log_{10}{}^{n}$ is reached number of nodes in a level is <= n. The recursion terminates at the height $\log_{10/9}{}^{n}$.

Average Case:

Assumptions:

All permutations of inputs are equally likely.

Partitions may not be in the same way at all levels.

Expect some partition be balance and some are not balanced.

When we adhere to the above assumptions then we get expected running time for quick sort as Θ(nlogn).

Randomized Quick Sort

We assumed that all permutations of inputs are equally likely but this assumption is not always true (think of already sorted array). So we use random approach for selecting pivot to relax that assumption. The algorithm is called randomized if its behavior depends on input as well as random value generated by random number generator. The beauty of the randomized algorithm is that no particular input can produce worst-case behavior of an algorithm. The worst case only depends on random number generated.

Algorithm:

```
RandPartition(A,i,j){
k = random(i,j); //generates random number between i and j including both.
swap(A[I],A[k]);
return Partition(A,i,j);
}
```

Algorithm:

```
RandQuickSort(A,p,q)
{
    if(p<q)
    {
        r = RandPartition(A,p,q);
        RandQuickSort(A,p,r-1);
        RandQuickSort(A,r+1,q);
    }
}
```

Analysis:

The analysis here works for the both versions of quick sort we have discussed.

Worst Case Revisited:

We saw that the worst case occurs when the partition selects the pivot at first or the last every time. Now assume we do not know what the worst case partition is lets say q such that,

$$T(n) = \max_{1<=q<=n-1}(T(q) + T(n-q)) + \Theta(n).$$

Here q is from 1 to n-1 since partition algorithm produces two partitions where there is at least 1 element. Since we have seen the worst case bound lets use substitution method.

Guess $T(n) = O(n^2)$.

To show $T(n) <= c\ n^2$.

Assume $T(q) <= c\ q^2$ and $T(n-q) < = c\ (n-q)^2$.

Now substituting this we get,

$$T(n) <= \max_{1<=q<=n-1}(c\ q^2 + c\ (n-q)^2) + \Theta(n).$$

$$= c \max_{1<=q<=n-1}(q^2 + (n-q)^2) + \Theta(n).$$

Taking first derivatives of $(q^2 + (n-q)^2)$ with respect to q we get,

$$2q - 2(n-q) = 2q - 2n + 2q = 4q - 2n$$

Second derivative with respect to q is 4 (positive).

We know if second derivative is positive and first derivative is zero then we have minima at q and that q is n/2. So there must be some other value of q less than (or greater than) n/2 such that the function has maxima at q. Take q at one end point (see at both end points value is same). So we can write,

$$\max_{1<=q<=n-1}(q^2 + (n-q)^2) <= (1^2 + (n-1)^2) + \Theta(n),\ q = 1 \text{(same for n-1)}.$$

$$= n^2 - 2(n-1), \text{ so}$$

$$T(n) <= cn^2 - 2c(n-1) + \Theta(n).$$

$$<= c\, n^2.$$

For constant c when $2c(n-1) >= \Theta(n)$.

Hence the worst case is $O(n^2)$.

We can eliminate the worst-case behavior of input instance by the following

Use the middle element of the subarray as pivot.

Use a *random* element of the array as the pivot.

Perhaps best of all, take the median of three elements (first, last, middle) as the pivot.

However, above care still gives the worst case running time as $\Theta(n^2)$.

Average Case:

To analyze average case, assume that all the input elements are distinct for simplicity. If we are to take care of duplicate elements also the complexity bound is same but it needs more intricate analysis. Consider the probability of choosing pivot from n elements is equally likely i.e. 1/n.

Now we give recurrence relation for the algorithm as

$$T(n) = 1/n \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n).$$

[In randomized version extra $\Theta(1)$ is added but the final relation would be same as above]

Now for some j = 1,2, …, n-1

T(q) + T(n-q) is repeated two times so we can write the relation as,

$$T(n) = 2/n\sum_{j=1}^{n-1} T(j) + \Theta(n).$$

Or, $$nT(n) = 2\sum_{j=1}^{n-1} T(j) + n^2. \text{ ---- (I)} \qquad [\text{taking } \Theta(n) = n \text{ for simplicity}]$$

For n =n-1 we have,

$$(n-1)T(n-1) = 2\sum_{j=1}^{n-2} T(j) + (n-1)^2. \text{ ---- (II)}$$

(I) – (II) gives,

$$nT(n) - (n+1)T(n-1) = 2n - 1.$$

Or, $$T(n)/(n+1) = T(n-1)/n + 2n - 1 /n(n+1).$$

Put T(n)/(n+1) as A_n then we have above relation as

$$A_n = A_{n-1} + 2n - 1 /n(n+1).$$

The above relation is written as,

$$A_n = \sum_{i=1}^{n} 2i - 1 /i(i+1). \qquad [2i - 1 \approx 2i]$$

$$\approx 2\sum_{i=1}^{n} 1/(i+1). \qquad [\textit{Harmonic series } H_n = \sum_{i=1}^{n} 1/i \approx \log n]$$

$$\approx 2\log n$$

But we have $T(n) = (n+1)\, A_n \approx 2(n+1)\log n$.

So,

$$T(n) = \Theta(n\log n).$$

Final Remark on Quick Sort

Practical implementations have shown that the quick sort is the faster sorting algorithm of all we have studied. Though worst time complexity for heap sort and merge sort are better than that of quick sort due to the large constant overhead quick sort runs faster. Another important factor is extra space also.

Sorting Comparison

Sort	**Worst Case**	**Average Case**	**Best Case**	**Comments**
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	Θ(nlogn)	Θ(nlogn)	Θ(nlogn)	Requires Memory
Heap Sort	Θ(nlogn)	Θ(nlogn)	Θ(nlogn)	*Large constants
Quick Sort	$\Theta(n^2)$	Θ(nlogn)	Θ(nlogn)	*Small constants

Median Order Statistics (Selection Problems)

ith order statistic of a set of elements gives ith largest(smallest) element. In general lets think of ith order statistic gives ith smallest. Then minimum is first order statistic and the maximum is last order statistic. Similarly a median is given by ith order statistic where i = (n+1)/2 for odd n and i = n/2 and n/2 + 1 for even n. This kind of problem commonly called selection problem. We can specify selection problem as:

Input: A set of n distinct elements and number i, with 1 <= i <= n.

Output: The element from the set of elements, that is larger than exactly i-1 other elements of the given set.

This problem can be solved in Θ(nlogn) in a very straightforward way. First sort the elements in Θ(nlogn) time and then pick up the ith item from the array in constant time. What about the linear time algorithm for this problem? The next is answer to this.

General Selection

This problem is solved by using the "divide and conquer" method. The main idea for this problem solving is to partition the element set as in Quick Sort where partition is randomized one.

Algorithm:

```
RandSelect(A,p,r,i)
{
        if(p = =r )
                return A[p];
        q = RandPartition(A,p,r);
        k = q – p + 1;
        if(i <= k)
                return RandSelect(A,p,q,i);
        else
                return RandSelect(A,q+1,r,i - k);
}
```

Analysis:

Since our algorithm is randomized algorithm no particular input is responsible for worst case however the worst case running time of this algorithm is $\Theta(n^2)$. This happens if every time unfortunately the pivot chosen is always the largest one (if we are finding minimum element).

Assume that the probability of selecting pivot is equal to all the elements i.e 1/n then we have the recurrence relation

$$T(n) = 1/n\Big(T(\max(1,n-1)) + \sum_{j=1}^{n-1} T(\max(j,n-j))\Big) + O(n).$$

Where T(max(1,n-1)) is either T(1) or T(n-1) is due to partition must at least partition a set with one element at a side, and

$\sum_{j=1}^{n-1}$ T(max(j,n – j)) is for a instance of recursive call for all except nth partition (see above is the nth partition) where,

max(j,n-j) = j, if j>=$\lceil n/2 \rceil$

and max(j,n-j) = n-j, otherwise.

Observe that every T(j) or T(n – j) will repeat twice for both odd and even value of n (one may not be repeated) one time form 1 to $\lceil n/2 \rceil$ and second time for $\lceil n/2 \rceil$to n-1, so we can write,

$$T(n) <= 1/n(T(n-1) + 2 \sum_{j=\lceil n/2 \rceil}^{n-1} T(j)) + O(n).$$

In the worst case T(n-1) = O(n^2). We have,

$$T(n) <= 2/n \sum_{j=\lceil n/2 \rceil}^{n-1} T(j) + O(n).$$

Using substitution method,

Guess T(n) = O(n)

To show T(n) <= cn

Assume T(j) <= cj

Substituting on the relation

$$T(n) \quad <= 2/n \sum_{j=\lceil n/2 \rceil}^{n-1} ck + O(n).$$

$$<= 2c/n(\sum_{j=1}^{n-1} k - \sum_{j=1}^{\lceil n/2 \rceil -1} k) + O(n).$$

$$= 2c/n(n(n-1)/2 - \lceil n/2 \rceil(\lceil n/2 \rceil-1)/2) + O(n).$$

$$<= c(n-1) – cn/4 + c/2 + O(n)$$

$$= c(3n/4 +1/2) + O(n).$$

$$<=cn.$$

For c large enough such that c c(3n/4 +1/2) dominates O(n).

Hence any order statistic is O(n).

Worst-case linear time Selection

Here we study the algorithm that guarantees the running time of the selection problem is O(n) at the worst case. In this algorithm like in RandSelect mentioned previously the input set is recursively partitioned. If we can guarantee that there will be good split of the input by choosing some pivot then we may come up with the solution. For this purpose we use median of medians as pivot point. To find median of medians we divide n input elements into $\lfloor n/r \rfloor$ groups of r elements each and n - $\lfloor n/r \rfloor$ elements are not used. When median of medians is used as a pivot we have each medians as $\lceil r/2 \rceil^{\text{th}}$ smallest element so at least $\lceil \lfloor n/r \rfloor /2 \rceil$ of the medians are less than or equal to median of medians and at least $\lfloor n/r \rfloor$ - $\lceil \lfloor n/r \rfloor /2 \rceil$ + 1 >= $\lceil \lfloor n/r \rfloor /2 \rceil$ of the medians are greater than or equal to median of medians. That means at least $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor /2 \rceil$ elements are less (greater) than or equal to median of medians.

Algorithm:

```
1. Select (A,k,l,u)
2. {
3. n = u – l + 1;
4. if(n <=r){
            sort(A,l,u);      return k^th element
5. }
6. divide(A,l,u,r); //ignore excess elements
7. Get medians from all the subsets and put on array m;
8. v = select(m, $\lceil \lfloor n/r \rfloor /2 \rceil$, 1, $\lfloor n/r \rfloor$);
9. Partition A using v as pivot.
10. If( k = =(j – l +1))         return v;          //j is the position of v
11. else if( k < (j – l +1))      return select(A,k,l,j-1);
12. else    return select(A,k - (j – l +1),j+1,u);
13. }
```

First consider r = 5 and the elements in the array are distinct. $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor /2 \rceil$ is at least $1.5\lfloor n/5 \rfloor$ then we can say at least $1.5\lfloor n/5 \rfloor$ elements are greater than or equal to v this implies there are at most n - $1.5\lfloor n/5 \rfloor$ <= .7n + a (a > 0)[Analysis shows a as 1.2, see text book] elements smaller than or equal to v. Here we are considering that the median finding is done in O(1) since there are few number to be sorted. Hence the steps other than the recursive ones take at most O(n) time. Running time for line 8 is T(n/5) since r = 5. Again .7n + 1.2 is no more than 3n/4 for n>=24. Now we can write recurrence relation as

$$T(n) = T(n/5) + T(3n/4) + O(n).$$

Guess T(n) = O(n) then we have to prove T(n) <= cn -b. Assuming guess is true for T(n/5) and T(3n/4) we get,

$$\begin{aligned} T(n) &\le cn/5 - cb + 3cn/4 - cb + O(n). \\ &= 19c/20n - 2cb + O(n). \\ &\le cn \end{aligned}$$

For large c and some b such that, 19c/20n – 2cb > = O(n).

Hence T(n) = O(n).

Matrix Multiplication

Given two A and B n-by-n matrices our aim is to find the product of A and B as C that is also n-by-n matrix. We can find this by using the relation

$$C(i,j) = \sum_{j=1}^{n} A(i,k)B(k,j)$$

Using the above formula we need O(n) time to get C(i,j). There are n^2 elements in C hence the time required for matrix multiplication is $O(n^3)$. We can improve the above complexity by using divide and conquer strategy.

Strassen's Matrix Multiplication Algorithm

Strassen was the first person to give better algorithm for matrix multiplication. The idea behind his algorithm is divide the problem into four sub problems of dimension n/2 by n/2 and solve it by recursion. The basic calculation is done for 2 by 2 matrix. The 2 by 2 matrix is solved as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{x} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where,

$P = (A_{11} + A_{22})(B_{11} + B_{22})$.

$Q = (A_{21} + A_{22}) B_{11}$.

$R = A_{11}(B_{12} - B_{22})$.

$S = A_{22}(B_{21} - B_{11})$.

$T = (A_{11} + A_{12}) B_{22}$.

$U = (A_{21} - A_{11})(B_{11} + B_{12})$.

$P = (A_{12} - A_{22})(B_{21} + B_{22})$.

And

$C_{11} = P + S - T + V$.

$C_{12} = R + T$.

$C_{21} = Q + S$.

$C_{22} = P + R - Q + U$.

See above there are total 7 multiplications and 18 additions.

We can have recurrence relation for this algorithm as

$T(n) = 7T(n/2) + O(n^2)$.

Solving above relation by master method we get case 1 so, $T(n) = \Theta(n^{2.81})$.

The fastest algorithm known for this problem is by Coppersmith and Winograd that runs in $\Theta(n^{2.376})$ but not used due to large constant overhead.

Exercises

1. Write an algorithm for integer multiplication (divide and conquer) discussed in this material but not done.
2. Trace out the steps of Quick Sort for the following input array

   A[] ={13, 19, 9, 5, 12 ,8, 7, 4, 11, 2, 6, 21}
3. Write an algorithm and analyze it that multiplies two n by n matrices and has complexity O(n^3)(brute force way).
4. 10.3.5 (pg 192)

   Given “black-box” worst-case linear time median subroutine, give a simple, linear time algorithm that solves the selection problem for an arbitrary order statistic.