Finding Min and Max element of an Array Using Divide and Conquer Approach

1. **Problem**:-

   Given an array A of n elements drawn from a totally ordered set, find a pair $< min, max >$ such that $min = MIN\{A[1], A[2], \dots, A[n]\}$ and $max = MAX\{A[1], A[2], \dots, A[n]\}$

2. **D and C Strategy**:-

   Since D and C strategy is sought we have to divide bigger problem into multiple smaller problems and solve each of them individually and independently. Finally, solutions need to be combined to get solution to bigger problem. **Since we are working on an array, it is natural to define a problem as an ordered pair of indices on which algorithm is working on at a particular call. For example, original/initial problem will be referred as $[\mathbf{1}, \boldsymbol{n}]$ since we want to work on entire array.** Also, it is better to divide problem in 2 smaller problems rather than more than 2. In that also we have many choices for deciding size of each problem. Better divide into roughly equal size i.e half the size.

   a. If current problem X is $[a, b]$ then $a \leq b$ and size of problem is the number of elements i.e $b - a + 1$.
   b. Divide problem$[a, b]$ into 2 subproblems $L = [a, \frac{b+a}{2}]$ and $R = \left[\frac{b+a}{2} + 1,\ \ b\right]$
   c. Let solutions to $L\ be < Min_L, \mathrm{Max_L} >$ and $R\ be < Min_R, \mathrm{Max_R} >$.
   d. Then solution to X is $< \min\{Min_L, Min_R\},\ \ \max\{Max_L, Max_R\} >$

   How far should a problem be divided? Since problem size will be either even or odd, smallest even and odd numbers are 1 and 2. Surely, boundary problem size is then 1 and/or 2. That is, stop dividing problem if problem is of the form $[a, a]$ or $[a, a+1]$ for some index $1 \leq a \leq n$.

   - If size=1 i.e problem is $[a, a]$ then solution is $< A[a],\ A[a] >$ since a single element is both min and max.
   - If size=2 i.e problem is $[a, a+1]$ then solution is either $< A[a], A[a+1] >$ or $< A[a+1], A[a] >$. Which one of these 2, can be found by making a comparison $A[a] < A[a+1]$.
   - If size is neither 1 nor 2 then do steps $a.$ to $d.$ as given above.

   Combining this discussion we get following D and C based algorithm.

3. **D and C based Recursive Algorithm**:-

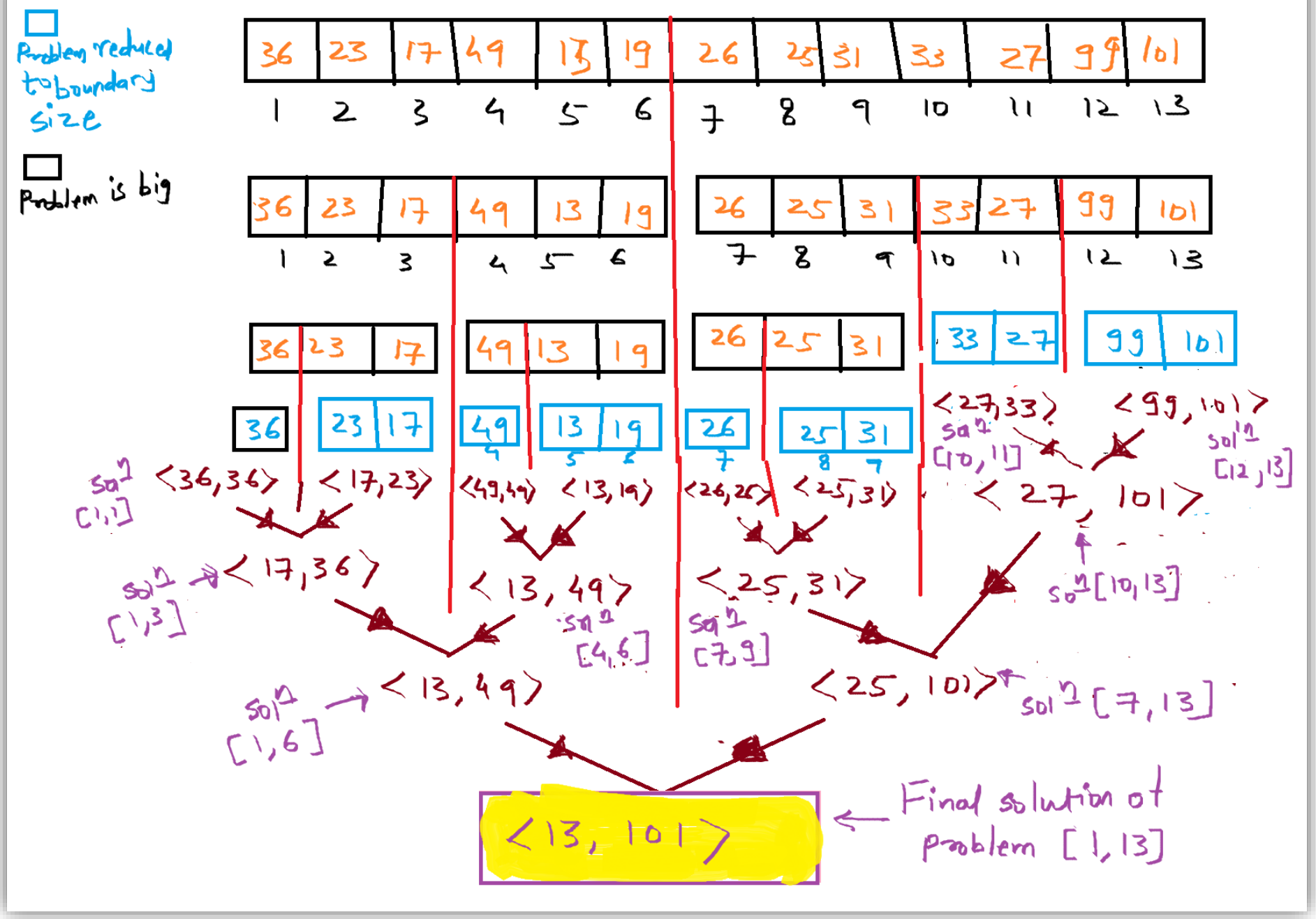
Line	Recursive Algorithm ..returns a pair <min, max>	
	< __ , __ > DCminmax(low, high)	
1	{	
2	If(low==high)	
3	Return $< A[low], A[low] >$	
4	If(high-low == 1)	
5	If($A[low] < A[high]$)	
6	Return $< A[low], A[high] >$	
7	Else	
8	Return $< A[high],\ A[low] >$	
9	$mid = \frac{low+high}{2}$	Line 9===Dividing [low, high] problem
10	$< Min_L,\ Max_L >$ = DCminmax(low, mid)	
11	$< Min_R,\ Max_R >$ = DCminmax($mid + 1,\ high$)	Lines 10—11, solving small subproblems
12	Return $< \min\{Min_L, Min_R\},\ \max\{Max_L, Max_R\} >$	Line 12= Combine phase to solve [low, high]
13	}	

   Lines 2—8, boundary condition

   Initial call to algorithm wants solution for biggest problem i.e it works on problem $[1,\ n]$ and hence initial call will be $\boldsymbol{DCminmax(1, n)}$

4. **Example**:-

   - Here, $n = 13$. Initial problem $[1,\ 13]$ .Hence initial call is $DCminmax(1, 13)$



5. **Time Complexity**:-

- We will look at counting number of comparisons required to solve simultaneous min, max problem. Why only comparisons? Why not other operations? Reason is not so much revealing and not hard to digest.
- Searching min and/or max is a very simple problem for which already $\Theta(n)$ worst case *linear search* algorithm is already known requiring exactly $n-1$ comparisons, given a random array.
- Also, given size of the array, indices always satisfy $1 \leq i \leq n$. But yet an array element $A[i]$ can be anything, right from arbitrarily large number to an arbitrary string to almost anything. Hence comparison can be quite a concern because of arbitrary nature of array elements. Given these things, counting comparisons is mostly focussed on.

It is easy to formulate a recurrence equation for the number of comparisons, given above recursive algorithm.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2$$
$$= 1\ ,\ n = 2$$
$$= 0,\ \ n = 1$$

- **Case :-** $\boldsymbol{n = 2^k}$, for some $k \in \mathbb{N}$.. hence $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$

$$T(n) = 2 + 2T\left(\frac{n}{2}\right) = 2 + 2[2 + 2T(n/4)] = 2 + 2^2 + 2^2 T\left(\frac{n}{4}\right) = 2 + 2^2 + 2^2\left[2 + 2T\left(\frac{n}{8}\right)\right] \ldots \text{ expanding till the end,}$$

$$T(n) = 2 + 2^2 + 2^3 + 2^4 + \cdots + 2^{k-1} + 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) = \ 2^k - 1 + 2^{k-1}\, T(2) = 2^k - 1 + 2^{k-1} 1 \ = n - 1 + \frac{n}{2} - 1$$

$$\boldsymbol{T(n) = \frac{3n}{2} - 2}$$

- **Case:-** $\boldsymbol{n \neq 2^k}$…
  I. By explicitly calculating number of comparisons for a few initial values, say up to $n = 32$, we observe following pattern for the equation of number of comparisons:-

$$T(n) = T(2^k) + (n - 2^k) \times 2, \ \ where\ k = \lfloor \log_2 n \rfloor\ and\ 2^k < n \leq 2^k + 2^{k-1}$$
$$= T(2^k) + n - 2^{k-1}, \qquad 2^k + 2^{k-1} < n < 2^{k+1}$$

Hence, after substituting $T(2^k) = \frac{3 \times 2^k}{2} - 2 = 3 \times 2^{k-1} - 2$, referring to Case above, we get

$$T(n) = 3 \times 2^{k-1} - 2 + (n - 2^k) \times 2, \ \ where\ k = \lfloor \log_2 n \rfloor\ and\ 2^k < n \leq 2^k + 2^{k-1}$$
$$= \ 3 \times 2^{k-1} - 2 + n - 2^{k-1}, \qquad 2^k + 2^{k-1} < n < 2^{k+1}$$

We can prove this by induction with base case, say $n = 8$. Proof is left to students, as a revision exercise of proof by induction.

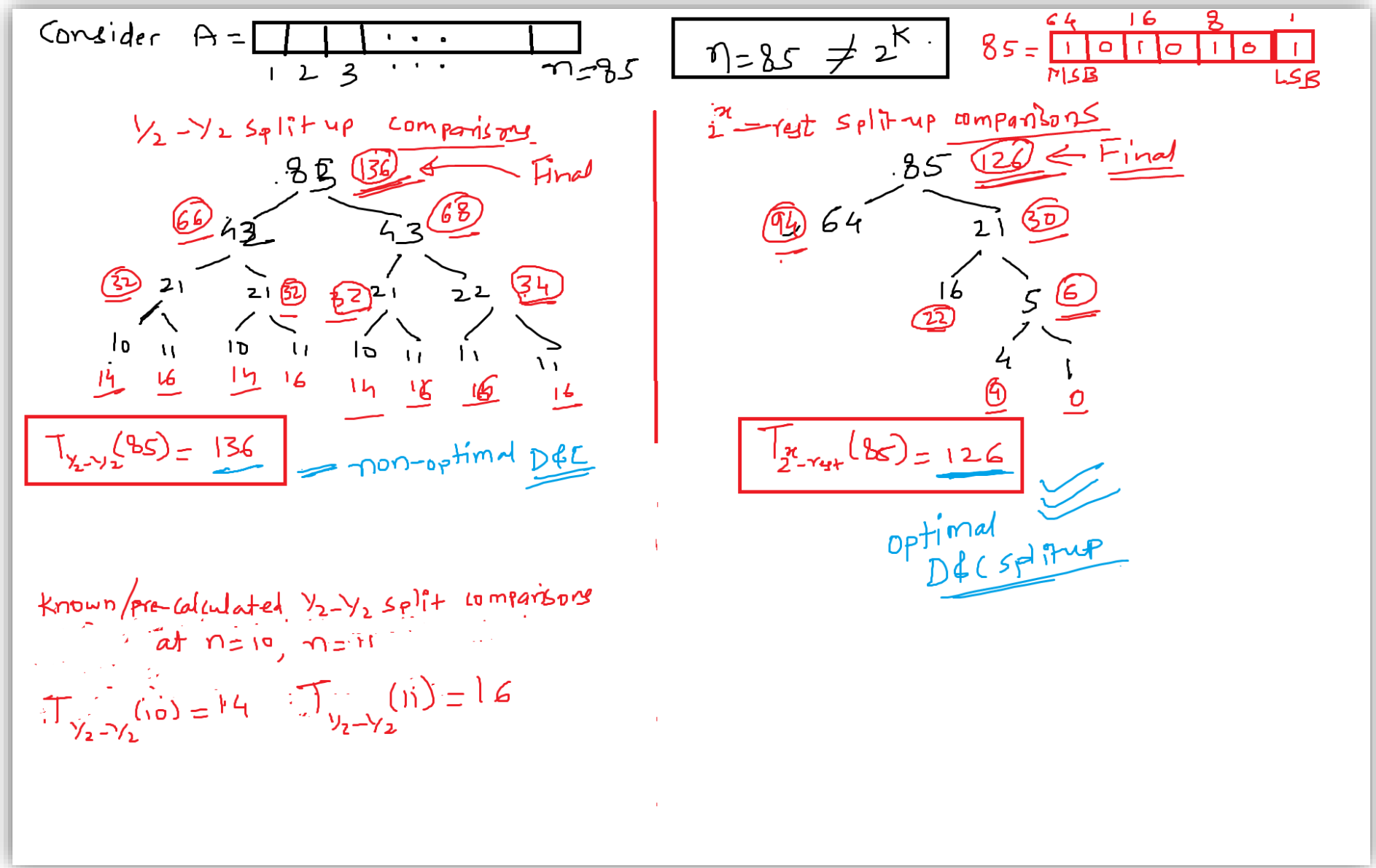
Finally, we have complete solution for number of comparisons done by $\frac{1}{2} - \frac{1}{2}$ split D&C algorithm is

$$\boldsymbol{T(n) = 3 \times 2^{k-1} - 2 + (n - 2^k) \times 2, \quad when\ k = \lfloor \log_2 n \rfloor\ and\ 2^k < n \leq 2^k + 2^{k-1}}$$
$$\boldsymbol{= \ 3 \times 2^{k-1} - 2 + n - 2^{k-1}, \qquad when \quad 2^k + 2^{k-1} < n < 2^{k+1}}$$
$$\boldsymbol{= \frac{3n}{2} - 2 = 3 \times 2^{k-1} - 2, \qquad when\ `\ n = 2^k, for\ some\ k \in \mathbb{N}} \qquad \textbf{E[1]}$$

II. However, here we see, by below example, that number of comparisons done by $\frac{1}{2} - \frac{1}{2}$ split up D and C is clearly more than any other split up. That is why D&C algorithm with $\frac{1}{2} - \frac{1}{2}$ split up is clearly not optimal when $n \neq 2^k$.

We can see reasons behind increased number of comparisons with $\frac{1}{2} - \frac{1}{2}$ split compared to $x - y$ split. $\frac{1}{2} - \frac{1}{2}$ split creates problem tree which is a complete tree of height $\log_2 n$. Such a tree creates at least $\frac{n}{2}$ problems. For every disjoint pair of problems, we require 2 comparisons to combine their solutions. More such pairs, more such comparisons to combine solutions. Maximum number of problems are created in a complete tree. Hence, $\frac{1}{2} - \frac{1}{2}$ split does maximum amount of work to combine sub problem solutions and hence maximum efforts to solve initial problem.

From below example, we can also see why $\frac{1}{2} - \frac{1}{2}$ split takes 10 comparisons more than unequal split. (*Hint*:- calculate how many more nodes are in left tree. Combine them in pairs to produce solution to parent problems. For every such pair 2 comparisons to combine. Overall effort adds up to 10 more comparisons. Since these nodes are absent in right tree, 10 comparisons are avoided.)



III. Moreover, recursive implementation has recursion stack depth of height $\leq \log_2 n$ i.e $S(n) \in O(\log_2 n)$.

IV. Above discussion must lead students to think about minimum number of comparisons required to solve simultaneous min-max problem and strategy to solve this problem. If D&C is an optimal strategy then

clearly $\frac{1}{2} - \frac{1}{2}$ split is not optimal. Which split-up $x - y$ should be chosen for optimality? If D&C is not an optimal strategy then which strategy is?

6. **C program**:-

```
#include<stdio.h>
#include<malloc.h>
//#include<stdlib.h>

struct record
{
   int max, min;
};

typedef struct record record;

void input(unsigned);
record dcminmax(unsigned, unsigned);

int *a=NULL;            // array a
unsigned comparisons=calculated=0;   // tracking comparisons done showing comparisons required

int main()
{
        record result;          // solution pair of input array
        unsigned n,i;

        scanf("%u", &n, printf("Enter size of array: "));
        if( (n>1)&&(a=(int*)malloc(n*sizeof(int))) ) // solve the problem if either u have got memory or
        {                               // problem size is at least 2
           input(n-1);
           for(i=0;i<n; i++)
              printf("%d ", a[i]);
           result=dcminmax(0,n-1);
           i=0;
           while(1)
           {
              if((1<<i)> n)
                 break;
              else
                 i++;
           }
           floorpower=1<<(i-1);
calculated=3*(floorpower>>1)-2+ (n<=(floorpower+(floorpower>>1))? ((n-floorpower)<<1):  (n-(floorpower>>1)));
           printf("\n\nMax=%d  Min=%d\n", result.max, result.min );
      printf("\ncomp[D&c]=%u  calculated=%u maxlowerbound=%u\n\n", comparisons, calculated, n/2*3-2*!(n%2) );
           free(a);
        }
        else
           printf("Not enough memory or size of array either 0 or 1\n");

        return 0;
}


void input(unsigned n)      // Generating input array randomly
{                                       // Can use rand() instead
        int * ip;
        ip==(int*)(n&(1<<5)?main:printf);
        while(n)
           a[n--]=*ip--;
        a[0]=*ip;
}

/*
void input(unsigned n)      // Generating input using rand() function
{                                       // from stdlib.h
   while(n)
      a[n--]=rand();
   a[0]=rand();
}
*/

record dcminmax(unsigned low, unsigned high)
```

```
{
        record record1, record2;
        unsigned mid;
        if(high==low)                       // boundary problem of size 1
        {
            record1.max=record1.min=a[low];
            return record1;
        }
        if( (high-low)==1)                  // boundary problem of size 2
        {
           if(a[low]>a[high])
           {
              record1.max=a[low]; record1.min=a[high];
           }
           else
           {
              record1.max=a[high]; record1.min=a[low];
           }
           comparisons++;
           return record1;
        }
                                    // big problem, needed to be divided and solved
        mid=low+(high-low)/2;               //divide phase
        record1=dcminmax(low, mid);         // solving smaller sub problems
        record2=dcminmax(mid+1, high);
        if(record1.max < record2.max)       // conquer phase to solve parent problem
        {
           record1.max=record2.max;
        }
        comparisons++;
        if(record1.min > record2.min)
        {
           record1.min=record2.min;
        }
        comparisons++;
        return record1;         //you can use record2 also. Doesn't matter, as we have to return a single record
}
```

7. **Generating a guess for** $T(n):-$

Below is the table of number of comparisons required by $\frac{1}{2}-\frac{1}{2}$ split algo. We can generate a good guess and prove it correct by induction for all values.

n	T(n)	n	T(n)	n	T(n)	n	T(n)	n	T(n)	n	T(n)
1	**0**	12	**18**	23	**36**	34	**50**	45	**72**	56	**86**
2	**1**	13	**19**	24	**38**	35	**52**	46	**74**	57	**87**
3	**3**	14	**20**	25	**39**	36	**54**	47	**76**	58	**88**
4	**4**	15	**21**	26	**40**	37	**56**	48	**78**	59	**89**
5	**6**	16	**22**	27	**41**	38	**58**	49	**79**	60	**90**
6	**8**	17	**24**	28	**42**	39	**60**	50	**80**	61	**91**
7	**9**	18	**26**	29	**43**	40	**62**	51	**81**	62	**92**
8	**10**	19	**28**	30	**44**	41	**64**	52	**82**	63	**93**
9	**12**	20	**30**	31	**45**	42	**66**	53	**83**	64	**94**
10	**14**	21	**32**	32	**46**	43	**68**	54	**84**		
11	**16**	22	**34**	33	**48**	44	**70**	55	**85**		

Value of n 2^k	**Value of n** $2^k + 2^{k-1}$	From $n = 2^k$, an arithmetic progression of consecutive even numbers continues till $n = 2^k + 2^{k-1}$. After that a sequence of further consecutive numbers continues till next power of 2 $i.e$ $n = 2^{k+1}$. Track values of $T(n)$ between 2 consecutive **yellow** boxes. You will see that the shift from AP of difference 2 to AP of difference 1 occurs at **maroon** coloured box. We already know $T(n = 2^k) = \frac{3n}{2} - 2$. From this we can create $T(n)$ for every other value of n. This observation about a very simple pattern, leads to equation **E[1]** which mentions formulae for all these 3 cases. To prove E[1] by induction, we can start with base case $n = 4$ suitably.

8.