Algorithm Design Paradigm

->Divide and Conquer

->Dynamic Programming

->Greedy Algorithms

->Branch and Bound

->Combinatorial Exploration

Divide and Conquer Paradigm:

->Base case: B(z)

->Decomposition mechanism

$<X_1,X_2,\ldots,X_k> \leftarrow D(X)$

->Substructure calls

$Y_1 \leftarrow f(X_1),\ldots,Y_k \leftarrow f(X_k)$

->Recomposition mechanism

$Y \leftarrow R(Y_1,Y_2,\ldots,Y_k)$

Complexity: $T(n) = \sum_{i=1}^{k} T(ni) + D(n) + R(k)$

SEARCHING

Searching from unordered set:

```
search_u(L,x) { // returns true or false based on x being present in L
    L={x_1,x_2,…,x_n}
    if(|L|=1){
        if(x=x_1) return true
        else    return false}
    split L into L_1,L_2 //(non-empty sets)
    if(search_u(L_1,x))
        return true
    else if(search_u(L_2,x))
        return true
    else    return false }
```

$$\boxed{T(n)=T(k)+T(n-k)+O(1)}$$

If k=1: T(n)=O(n) (from finding maximum element algorithm)

If splitting the set into one element and rest other elements (k=1) gives the best complexity solution then the approach can also have been iterative as well as recursive.

Problem lower bound

The problem lower bound is the minimum time(or steps) needed to solve the problem entirely, it is not dependent on the algorithm we choose it is fixed for a problem.

For example in the above searching from unordered set the problem lower bound is **Ω(n)**. We cannot improve the complexity below O(n). Suppose we assume that we can improve it further, then by contradiction it can be proved that if the complexity is less than n for the size n input we end up missing seeing one element and that element itself might be the element we are searching for in worst case.

The constant improvement includes

- Push up the problem lower bound and prove it
- Push down and prove algorithm upper bound

So that tight bound is achieved. If the problem lower bound and algorithm upper bound are equal then the algorithm is said to be **optimal**.

Search in an ordered set:

Advantage: Can probe at any element and throw away a chunk of data (either less than or greater than the required element) before continuing search. (Reduces the domain of search)

```
search_o(L,x){
    if(|L|=0) return false
    if($x_i$>x)
        if(search_o(L-{$x_i$,…,$x_n$},x)}
            return true
        else    return false
    else
        if(search_o(L-{$x_1$,…,$x_i$},x)}
            return true
        else    return false }
```

$$\boxed{T(n)=\mathbf{max}\{T(i-1),T(n-i-1)\} + O(1)}$$

The **max** denotes that in worst case we have to search the set that has more number of elements.

Case(i): i=1

$T(n)=T(n-1)+O(1)$

$T(n)=O(n)$

Case(ii): i=k [some constant]

$T(n)=T(n-k)+O(1)$

$T(n)=O(n)$ [The constant split is also as bad as the one and the rest split]

Case(iii): i=n/2

$T(n)=T(n/2)+O(1)$

$T(n)=\log_2 n + 1$[improvement over O(n)]

Best case

QUESTIONS:

1.Why i=n/2?

$T(n)=T(n/2)+O(1)$

$T(n)=\log_2 n + 1$ [improvement over O(n)]

Best case

2.Why not n/3 or 2n/3?

if i=n/3

$T(n)=T(n/3)+O(1)$

$T(n)=\log_3 n +1$

$\log_3 n > \log_2 n$

Similarly, if i=2n/3

$T(n)=T(2n/3)+O(1)$

$T(n)=\log_{(\frac{3}{2})} n + 1$

$\log_{(\frac{3}{2})} n > \log_2 n$

The number of probings(steps) required is more in cases other than i=n/2

3.Why not probe twice(in same step)?

$T(n)=T(n/3)+2$ [If we probe twice in a single step we have 3 subproblems of size n/3 each]

$T(n)=T(n/3^k) + 2k$

$T(n)=2\log_3 n$

If we probe three times in a single step we get 4 subproblems of size n/4 each. In such case

$T(n)= 3\log_4 n$ [worse than $2\log_3 n$]

Can complexity be less than order of logn for search in ordered set ?

If the input size is 'n' it needs logn bits to be stored in the memory. To search a size n set for an element x we need to read the size of input(logn bits). Therefore complexity cannot be less than logn. We can choose to not see every element(throw away a chunk of data either less than or greater than the required element) but we **have** to see(read) the input size.

TRY:

Search a target set S from input set L where:

i. **L is ordered, S is ordered** [Best case]
ii. **L is ordered, S is unordered**
iii. **L is unordered, S is ordered**
iv. **L is unordered, S is unordered** [|L|=n,|S|=m,O(mn), L can be sorted using tournament sort in logn time to get case(ii)]

All of the above variants are finding intersection of two sets.

Try using binary search tree data structure and array.

For binary search tree complexity is of order nlogn.

SORTING

$L=\{x_1,x_2,\ldots,x_n\}$

Approach 1: Max removal

```
Sort1(L){
        if(|L|<=1) return L
        $x_i$<-findMax(L)
        L'<-L-{$x_i$}
        M<-sort1(L')
        return ($x_i$||M) } // where || is concatenation
```

$$\boxed{T(n)=T(n-1)+O(n) \Rightarrow T(n)=O(n^2)}$$

[The decomposition step is finding the maximum element which is of order O(n).]

Called **Selection sort** because we select the max element and sort it into right place in every subproblem.

The number of operations can be improved in max finding step to (3n/2)-1, but the order of sorting remains n^2.

Two way selection sort - finding minMax in one step.

Freezing the decomposition mechanism is not the only solution, trying different mechanisms by playing around with balance and split of recursion tree can improve the complexity.

Heap Sort – Tournament structure

Balance the splits so that finding first max element reduces the complexity of finding successive maximum elements(to logn order)

Heap data structure when represented as an array every i^{th} element is greater than $(2i)^{th}$ and $(2i+1)^{th}$ elements.

Heap is always first filled towards left.