Foundations of Algorithm Design and Machine Learning: Scribe

Date: 8th January, 2025

Algorithm Design Steps:

Initial Solution

- Recursive Formulation
- Correctness
- Analysis

Exploration of Possibilities

- Decomposition
- Recursive Structure
- Recomposition

Solution Refinement

- Balance/Split
- Identical Sub-Problems

Find Solution

- Traversal of recursive structure
- Prune

Data Structuring

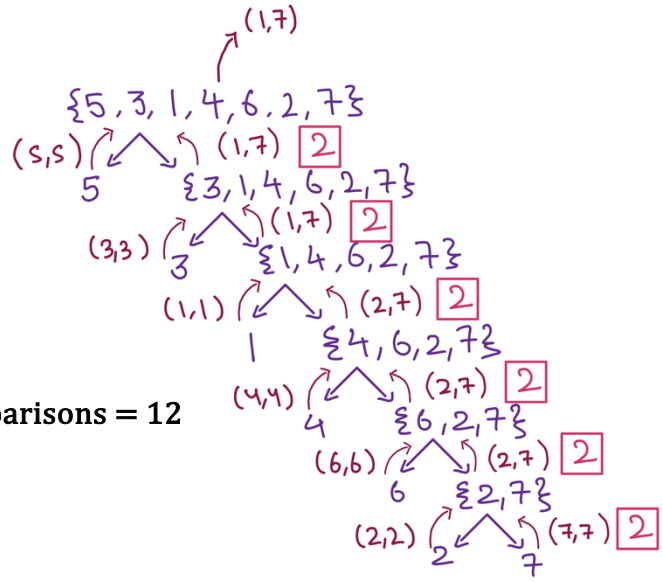
- Reuse Computation
- Space Complexity

Example 1: Finding the largest and smallest number

Initial Solution

```
<m,M> <- minMax1 (L)
        {
        If (|L|)=1) return (x1,x1)
                L`<-L-{x1}
        <m',M'><-minMax2(L)

        If (x1>m')
                m<-m'
        else
                m<-x1
        if(x1>M')
                M<-x1
        Else
                M<-M'
        }
```



#comparisons = 12

The input list is divided recursively into smaller sublists until each sublist contains only one element. Once the sublists are reduced to single elements, comparisons are made as the results are merged back up the recursion tree

Time Complexity

The recurrence relation for the number of comparisons is: $T(n)=T(n-1)+2$

Expanding this recurrence:

$$T(n)=T(n-2)+2+2$$
$$T(n)=T(n-3)+2+2+2$$
$$\vdots$$
$$T(n)=T(1)+2(n-1)$$

Since $T(1)=0$, we get:

$$T(n)=2(n-1)$$

For $n=7$, the total number of comparisons is:

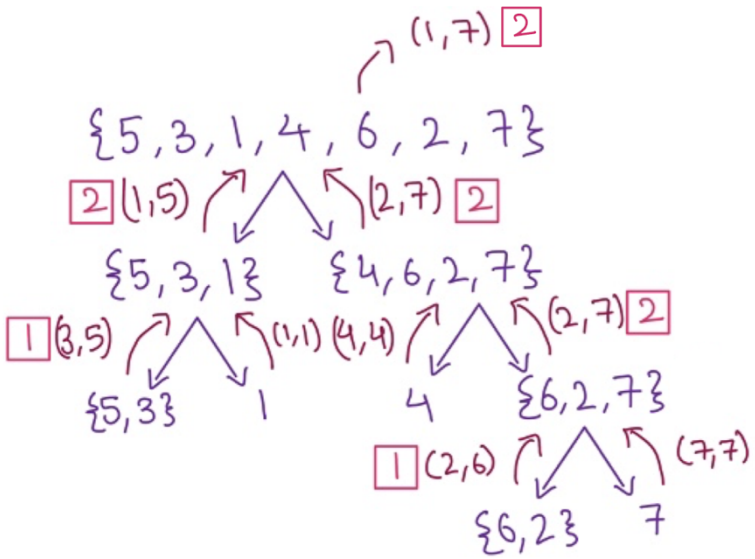
$$T(7)=2(7-1)=12$$

Alternate Solution

Adding a separate condition for |L|=2 to reducing the number of comparisons when number of elements is 2 from 2 to 1 and modifying the divide-and-conquer method to further reduce the number of comparisons.

```
<m,M> <- minMax2(L)
        {
        If (|L|=1)
                return (x1,x1)
        If (|L|=2) { if x1>x2
                        return (x1,x2)
                   Else
                        Return (x2,x1)
                   }
        Split L into two non-empty sets L1,L2
        <m1,M1>-minMax(L1)
        <m2,M2>-minMax(L2)
        If (m1>m2)
                {
                m<-m2
                If (M1>M2)
                        M<-M1
                Else
                        M<-M2
                }
        Else
                m<-m1
                If (M1>M2)
                        M<-M1
                Else
                        M<-M2
        }
```

Splitting Method 1

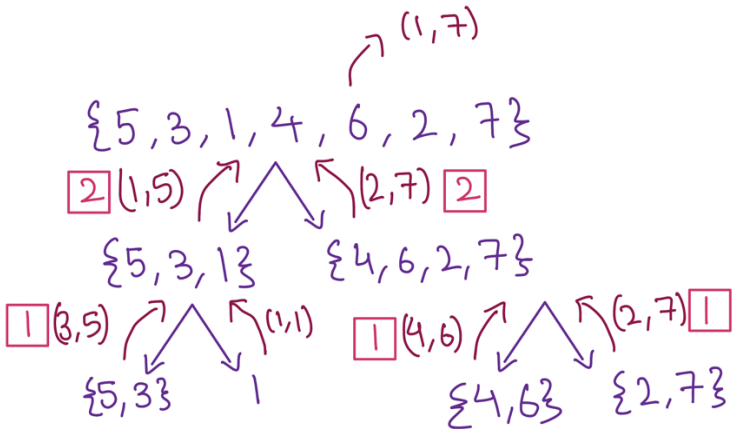


#comparisons = 10

Modifying the splitting criteria in method 1 to reduce the number of comparisons.

Splitting Method 2

Splitting L into two lists containing even number of elements



#comparisons = 9

Split Method 1: Arbitrarily into two sublists.
Split Method 2: Into two sublists with even number of elements wherever possible.

Splitting evenly reduces comparisons because it balances the recursive calls. This approach optimizes the divide-and-conquer strategy for finding the minimum and maximum in a list.

As a thumb rule, while splitting even number of elements, always split into two groups of even number of elements. While splitting odd number of elements, always split into one group of even number of elements and one of odd.

Average Time Complexity

To find the average time complexity or average number of comparisons for the splits (1 and n-1), (2 and n-2), and so on,

$$\boldsymbol{Recurrence\ Relation\ for\ T(n)}:$$

$$\textit{The recurrence relation is given as}:$$

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

$$T(n) = T(2) + T(n-2) + 2$$

$$T(n) = T(3) + T(n-3) + 2$$

$$\vdots$$

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

Summing All Terms:

$$(n-1)T(n) = \left[2i = 1\sum n-1T(i) + 2(n-1)\right] \qquad (1)$$

Writing the same statement for $n-1n-1$:

$$(n-2)T(n-1) = [2i = 1\sum n-2T(i) + 2(n-2)]. \qquad (2)$$

Subtracting $(\mathbf{2})$***from*** $(\mathbf{1})$:

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

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

Multiplying Both Sides by $\left(\frac{\mathbf{1}}{\boldsymbol{n-1}} - \frac{\mathbf{1}}{\boldsymbol{n}}\right)$:

$$\frac{T(n)}{n} - \frac{T(n-1)}{n-1} = 2\left(\frac{1}{n-1} - \frac{1}{n}\right)$$

Writing above relation for $\boldsymbol{n, n-1, n-2}$***, and so on***:

$$\frac{T(n-1)}{n-1} - \frac{T(n-2)}{n-2} = 2\left(\frac{1}{n-2} - \frac{1}{n-1}\right)$$

$$\frac{T(n-2)}{n-2} - \frac{T(n-3)}{n-3} = 2\left(\frac{1}{n-3} - \frac{1}{n-2}\right)$$

$$\vdots$$

$$\frac{T(2)}{2} - \frac{T(1)}{1} = 2\left(\frac{1}{1} - \frac{1}{2}\right)$$

Adding All Terms:

$$\frac{T(n)}{n} = 2\left(1 - \frac{1}{n}\right)$$

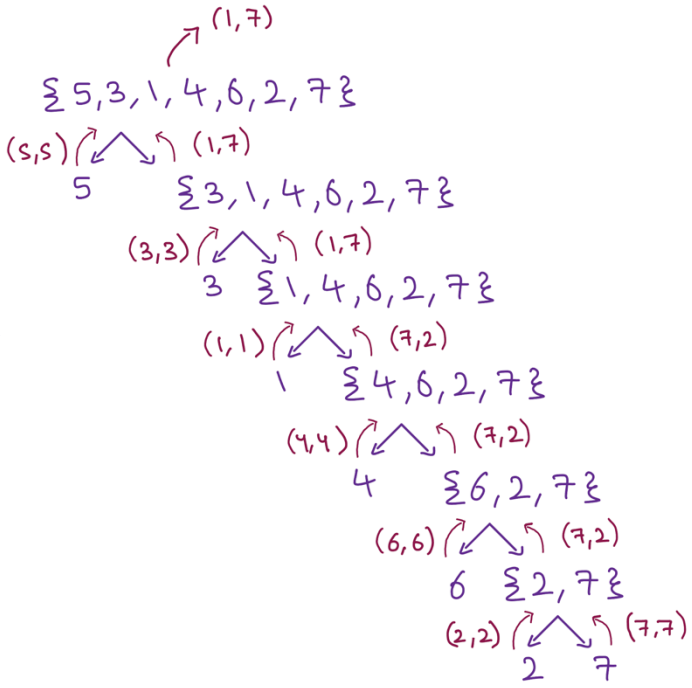
Final Result:

$$T(n) = 2(n-1)$$

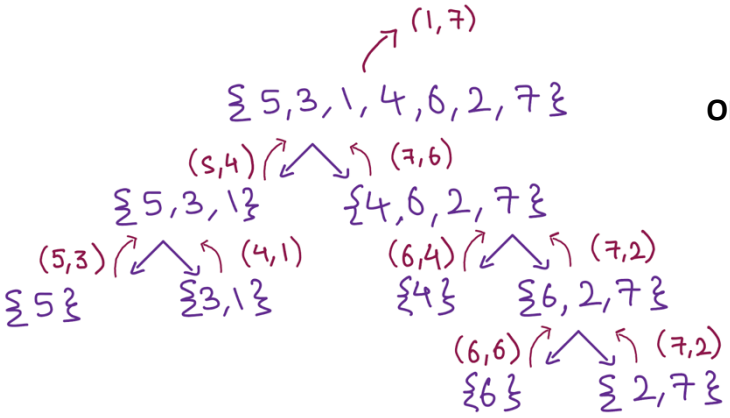
Example 2: Finding the largest and second largest number

Possible Solution

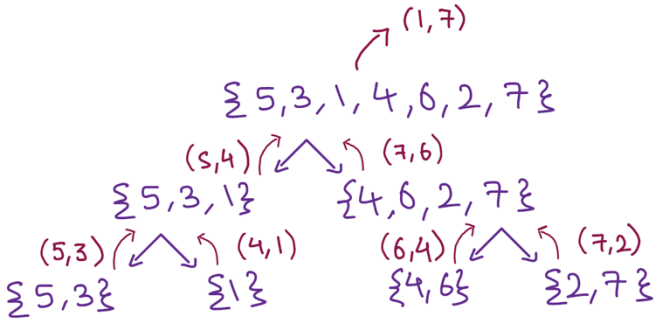
```
<m,M> <- max_and_2ndmax(L)
        {
        If (|L|=1)
                return (x1,x1)
        If (|L|=2) { if x1>x2
                        return (x1,x2)
                    Else
                        Return (x2,x1)
                    }
        Split L into two non-empty sets L1,L2
        <m1,M1> <- max_and_2ndmax (L1)
        <m2,M2> <- max_and_2ndmax (L2)
        If (m1>m2)
                {
                m <- m1
                If (m2>M1)
                        M<-m2
                Else
                        M<-M1
                }
        Else
                m <- m2
                If (m1>M2)
                        M<-m1
                Else
                        M<-M2
        }
```



OR



OR



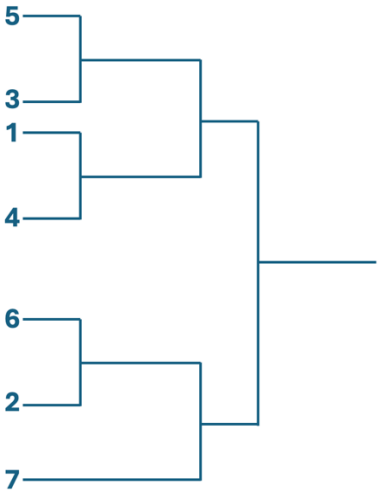
Grand Slam Tournament Structure

To find the second largest number, we can take a bottom up approach, (the grand-slam tournament structure). The unique characteristic about the structure is that en-route to becoming the champion, the second largest number would have must been beaten by the largest number.

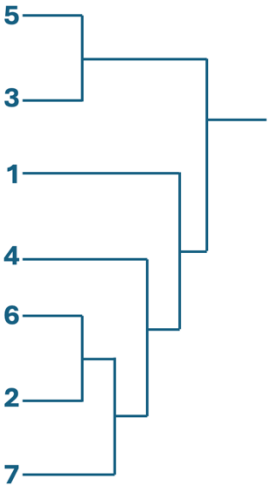
So to find the 2nd largest number:

- Maintain the hierarchical tree structure as it is.
- Replace the largest number with '-infinity' in the maintained tree structure.
- Check for the second largest number **ONLY** on the nodes where the largest number had competed.

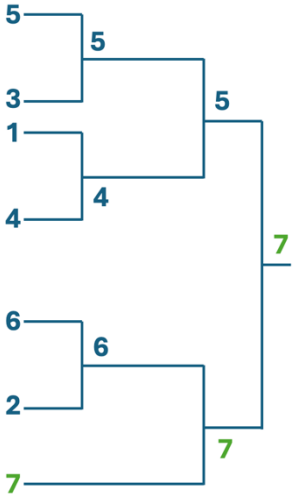
Data Structuring

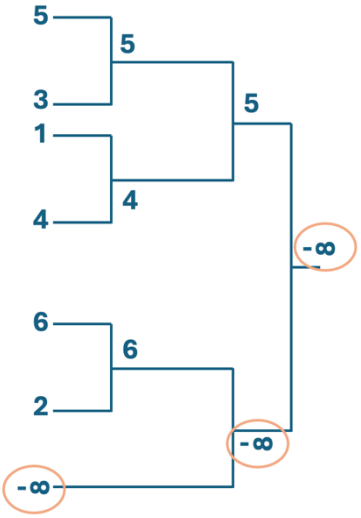


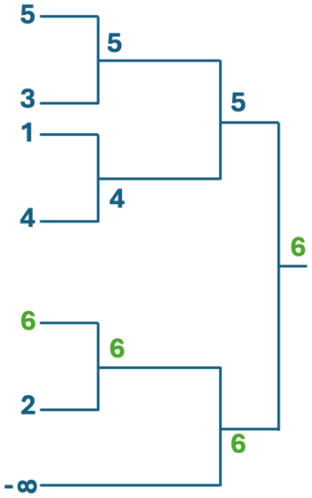
Case 1: Heap



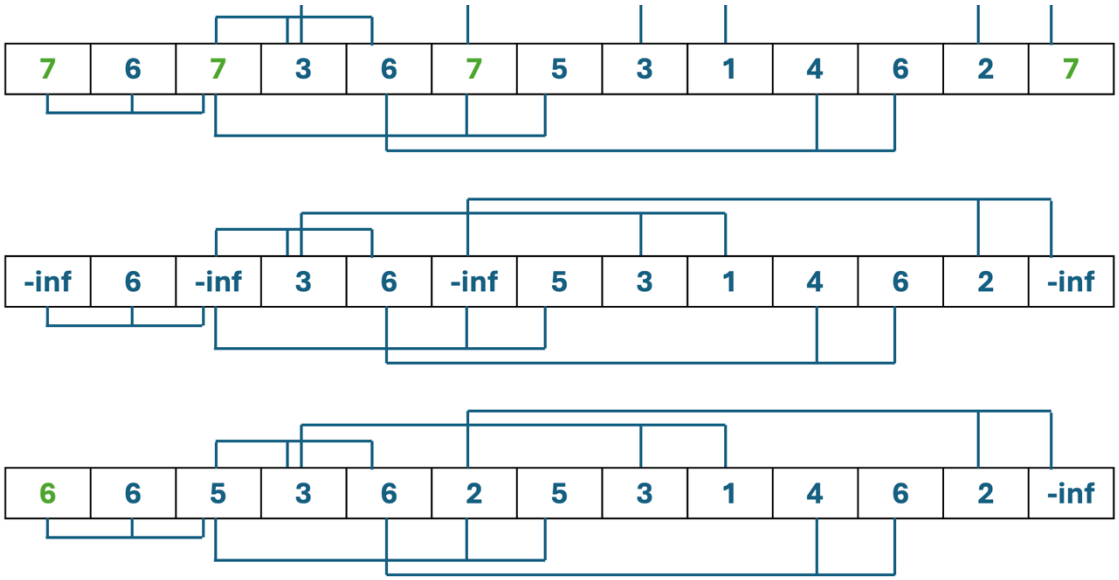
Case 2: Array







Finding Largest and Second largest elements using Heap Structure



Finding Largest and Second largest elements using Array Structure

If ii is the index of the largest element, then:

$$Compare\ i' = \left\lfloor \frac{i}{2} \right\rfloor - 1,$$

$$Then\ i'' = \left\lfloor \frac{i'}{2} \right\rfloor - 1,$$

$$and\ so\ on, until\ reaching\ level\ 0.$$

Time Complexity

Number of comparisons required to find the largest number is (n-1).

Time to traverse half the tree after replacing the largest element with '- infinity', without changing the structure of the tree is [logn].

$$Time\ required\ to\ find\ largest\ element\ =\ (n-1)$$

$$Time\ required\ to\ find\ largest\ and\ 2nd\ largest\ elements\ =\ (n-1)+\ 2\lceil logn \rceil$$

$$Time\ required\ to\ find\ largest, 2nd\ and\ 3rd\ largest\ elements\ =\ (n-1)+\ 4\lceil logn \rceil$$

$$and\ so\ on$$

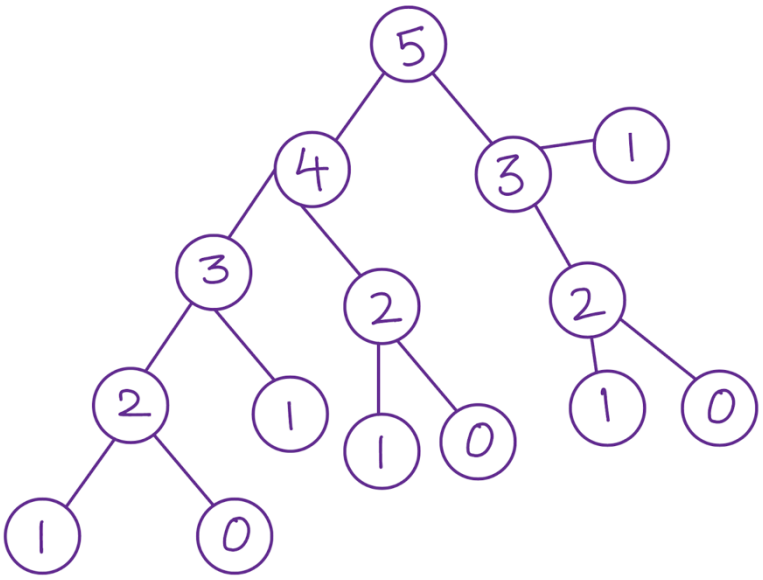
Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21,....

$$fib(n) = fib(n-1) + fib(n-2) \quad if\ n \geq 2$$
$$= n \quad if\ n < 2$$

Code:

```
eval_fib(n) {
if n<2 return n
eval_fib(n) = eval_fib(n-1) + eval_fib(n-2)
return fib(n)
}
```



As visible from the chart above, multiple computations are done multiple times, resulting in exponential complexity. Ideally we should not compute the same sub-problem again and again.

Time Complexity

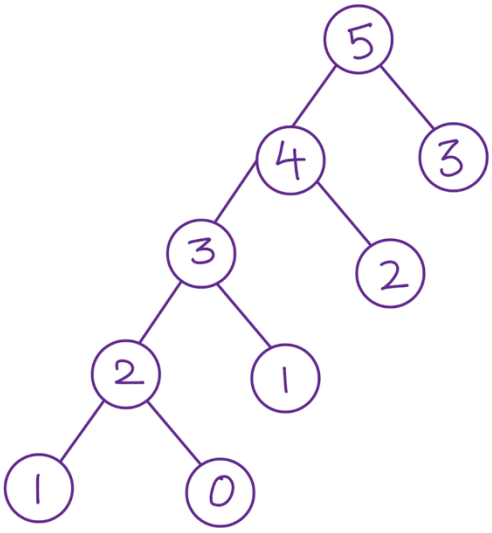
$$T(n) = T(n-1) + T(n-2) + 1$$

$$= \frac{1}{\sqrt{5}}\left[\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{\sqrt{5}-1}{2}\right)^n\right]$$

$$= O(2^n),\ Exponential$$

Solution: Memoization

The technique of caching the calculated values in a table to optimise computation of algorithms like the above, where computations are redone multiple times, is called memoization.



f(n)	0	1	1	2	3	5	8	13
Done	1	1	1	1	1	1	0	0

Values not being computed again and again; values already computed are stored in the Memoization table

A memoization table is used to store the results of previously computed subproblems in dynamic programming. This table enables efficient retrieval of these results when the same subproblem recurs, avoiding redundant calculations. Done is marked as 1 if the value for that specific sub-problem has been computed once.

This optimization reduces the time complexity of problems with overlapping subproblems, such as the Fibonacci sequence, from exponential O(2^n) to linear O(n), while trading off additional memory space for storing results.

Iterative solutions for Fibonacci series:

1. $f(n) = f\left(n - \frac{n}{2}\right) + f(n + 1), \qquad n \geq 2$

$= 1, \qquad n < 2$

2. $f(n) = f(n+1) + f\left(\frac{n-1}{2}\right), \qquad if\ n\ is\ odd$

$= f(n-1) + f\left(\frac{n}{2}\right), \qquad if\ n\ is\ even$

Generic expression –

$$f(n) = f\big(g(n)\big) + f\big(h(n)\big), \qquad if\ c(n)\ is\ true$$

$$= b(n), \qquad if\ c(n)\ is\ false$$

In case there is an existence of cycle in which case the computation needs to be terminated, the DONE contains three values:

$$DONE = 0, if\ computation\ is\ pending$$

$$DONE = 1\ if\ cycle\ is\ encountered$$

$$DONE = 2, computation\ completed$$