Algorithm Design

Scribe

08/01/2025

1 Algorithm Design

1.1 Algorithm Design Framework

Any solution to an algorithmic problem can be derived by applying a standardized framework, the details of which are outlined below.

Step 1: Initial Solution

- **Recursive Formulation:** This step involves formulating the solution of a problem according to the divide-and-conquer paradigm, i.e., breaking the problem into subproblems of similar type and recursively solving them to arrive at a solution. This involves three major steps:
  - **Divide:** Break the problem into smaller subproblems.
  - **Conquer:** Solve the subproblem and return the solution to the calling problem.
  - **Combine:** Combine the solutions of subproblems to output the solution of the problem.
- **Correctness:** Use mathematical induction method to prove the correctness of the solution.
- **Analysis:** After establishing the correctness of the solution a thorough analysis is done on parameters like time complexity, space complexity, resource utilization in terms of power, security etc.

Step 2: Exploration of Possibilities

- **Decomposition:** Taking an example, draw out a recursive structure for the analysis of the problem. The recursive structure involves decomposing the problem into multiple subproblems and exploring all the sub-problems pathwise.
- **Recursive Structure:** Explore the recursive structure until the base case/leaf node is reached.
- **Recomposition:** Start combining the solutions of the subproblems from the leaf node of the recursive structure to the root of the recursive tree.

Step 3: Solution Refinement

- **Balance/Split:** Try to identify if there are possibilities to optimize the decomposition structure of the problem so that calls made in the recursive structure are reduced.
- **Identical Subproblems:** Identify the nodes within the recursive structure that represent identical subproblems, which are computed repeatedly.

Step 4: Final Solution

- **Traversal of recursive structure:** Analyze if the optimized traversal consists of minimum number of comparisons possible.
- **Prune:** Prune the recursive tree structure of nodes that denotes identical subproblems.

Step 5: Data Structuring

- **Reuse Computation:** Create memoization table to store solutions of nodes that are computed as you traverse the tree structure.
- **Space Complexity:** Try to analyze the space complexity of the final solution by taking into account all the data structures used in developing the implementation.

1.2 Examples

1.2.1 Finding Largest and Smallest Number

Initial Solution (Pseudo-code)

$< m, M > \leftarrow minMax(L)$
$if(|L| = 1) \quad return(x_1, x_1)$
$L' \leftarrow L - x_1$
$< m', M' > \leftarrow minMax(L')$
$if(x_1 > m') \quad m \leftarrow m'$
$else \quad m \leftarrow x_1$
$if(x_1 > M) \quad M \leftarrow x_1$
$else \quad M \leftarrow M'$

Time Complexity Analysis

$$T(n) = T(n-1) + 2 = 2(n-1) \tag{1}$$

Recursive Structure
As seen in Figure 1 the number of elements in the example array is 7, thus the number of comparison becomes 2(n-1) i.e. 12. The question to ask after the analysis is can we decrease the number of comparisons by using different method for splitting the problem into smaller sub-problems.

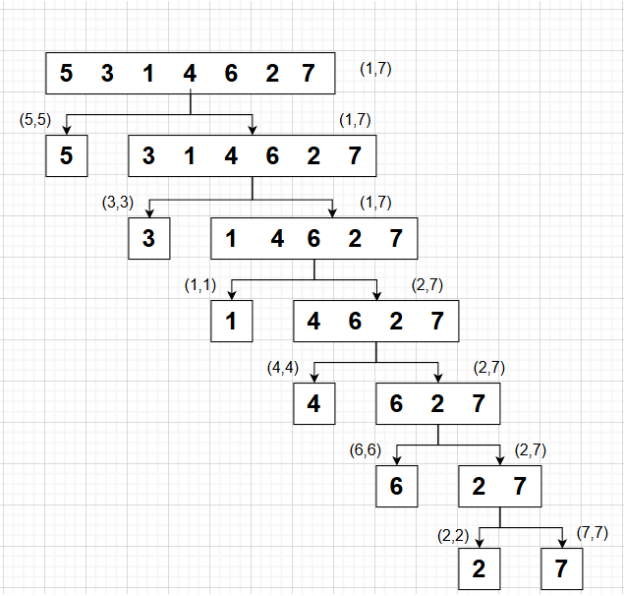

Figure 1: MinMax Recursion Structure

Alternate Solution (Pseudo-code)

$< m, M > \leftarrow minMax(L)$
  $if(|L| = 1) \quad return(x_1, x_1)$
  $if(|L| = 2)$
    $if(x_1 > x_2) \quad return(x_2, x_1)$
    $else \quad return(x_1, x_2)$
  Split L into two non empty set $L_1 and L_2$
  $< m_1, M_1 > \leftarrow minMax(L_1)$
  $< m_2, M_2 > \leftarrow minMax(L_2)$
  $if(m_1 > m_2)$
    $m \leftarrow m_2$
    $if(M_1 > M_2) \quad M \leftarrow M_1$
    $else \quad M \leftarrow M_2$
  $else$
    $m \leftarrow m_1$
    $if(M_1 > M_2) \quad M \leftarrow M_1$
    $else \quad M \leftarrow M_2$

Time Complexity Analysis

As per the recursive structure in Figure 2 the number of comparisons has reduced to 10 with red text marking the number of comparisons on each node in the figure.

Recursive Structure

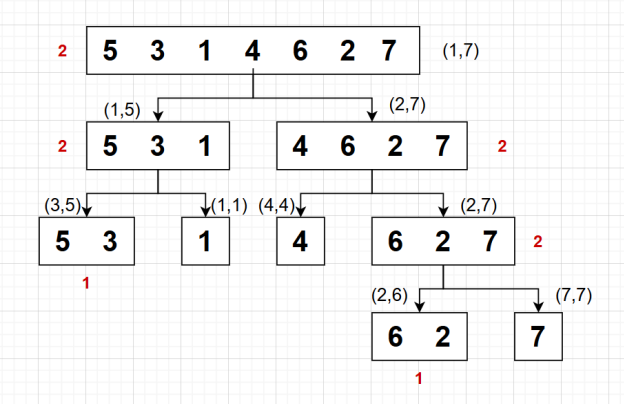

Figure 2: MinMax Recursion Structure

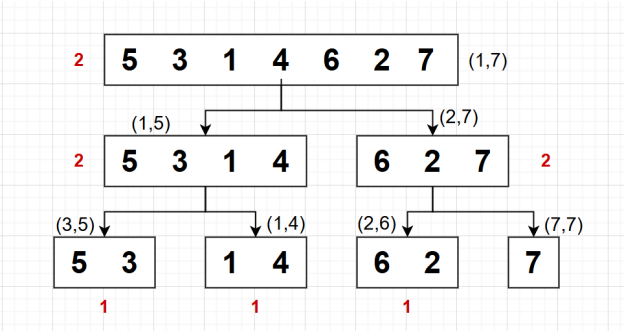

Figure 3: MinMax Recursion Structure

Can we reduce the number of comparisons further?
Let's try to create a recursive structure which splits the problem as follows:
Given n elements:

- If n is even, split the problem into two n/2 size sub-problems.
- If n is odd, split the problem into two (n+1)/2 and (n-1)/2 sub-problems respectively.

The structure is shown in figure 3. As observed from the figure the number of comparisons denoted by red text has reduced to 9 from 10. This indicates a further improvement in the algorithm. Can we reduce this further? To answer this we should find out the average case time complexity as shown below:

$$\begin{aligned} T(n) &= T(1) + T(n-1) + 2 \\ T(n) &= T(2) + T(n-2) + 2 \\ &\vdots \\ T(n) &= T(n-1) + T(1) + 2 \end{aligned}$$

Adding the above equations yields:

$$(n-1)T(n) = 2\sum_{i=1}^{n-1} T(i) + 2(n-1) \tag{2}$$

Replacing n with n-1 in equation 2 we have:

$$(n-2)T(n-1) = 2\sum_{i=1}^{n-2} T(i) + 2(n-2) \tag{3}$$

Adding (2) and (3) yields:

$$(n-1)T(n) - nT(n-1) = 2 \tag{4}$$

Dividing (4) by n(n-1) yields:

$$\frac{T(n)}{n} - \frac{T(n-1)}{n-1} = 2(\frac{1}{n-1} - \frac{1}{n}) \tag{5}$$

If we recursively solve (5) for n, n-1, n-2, ... 2 we get the following:

$$\begin{aligned}
\frac{T(n)}{n} - \cancel{\frac{T(n-1)}{n-1}} &= 2(\cancel{\frac{1}{n-1}} - \frac{1}{n}) \\
\frac{T(n-1)}{n-1} - \cancel{\frac{T(n-2)}{n-2}} &= 2(\cancel{\frac{1}{n-2}} - \frac{1}{n-1}) \\
&\vdots \\
\cancel{\frac{T(2)}{2}} - \frac{T(1)}{1} &= 2(1 - \cancel{\frac{1}{2}})
\end{aligned}$$

The above equations yields:

$$\begin{aligned}
\frac{T(n)}{n} - T(1) &= 2(1 - \frac{1}{n}) \\
\implies \frac{T(n)}{n} &= 2(1 - \frac{1}{n}) \\
\implies T(n) &= 2(n-1)
\end{aligned}$$

The above result shows that finding minimum and maximum from a given list of number will take O(n) comparisons on an average. Thus the even split solution is the best amongst the all.

1.2.2 Finding the largest and second largest

Initial Solution (Pseudo-code)

$< M_l, M_{sl} > \leftarrow max2(L)$
  $if(|L| = 1) \quad return(x_1, x_1)$
  $if(|L| = 2)$
    $if(x_1 > x_2) \quad return(x_1, x_2)$
    $else \quad return(x_2, x_1)$
  Split L into two non empty set $L_1 and L_2$ according to the even split rule

$< M_{l1}, M_{sl1} > \leftarrow max2(L_1)$
$< M_{l2}, M_{sl2} > \leftarrow max2(L_2)$
$if(M_{l1} > M_{l2})$
  $M_l \leftarrow M_{l1}$
  $if(M_{sl1} > M_{l2}) \quad M_{sl} \leftarrow M_{sl1}$
  $else \quad M_{sl} \leftarrow M_{l2}$
$else$
  $M_l \leftarrow M_{l2}$
  $if(M_{sl2} > M_{l1}) \quad M_{sl} \leftarrow M_{sl2}$
  $else \quad M_{sl} \leftarrow M_{l1}$

Recursive Structure
The recursive structure clearly shows that only 9 comparisons are required to get the solution to the problem. Thus, the solution is optimal.

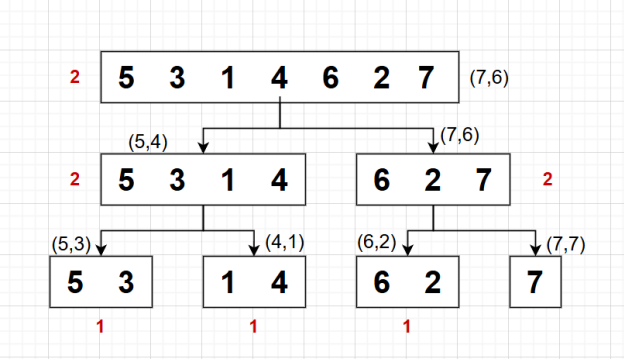

Figure 4: Max2 Recursion Structure

The same problem can be solved using a bottom up approach also called the grand-slam tournament structure. The heap structure shown below explains the flow of solution using the grand-slam tournament structure.

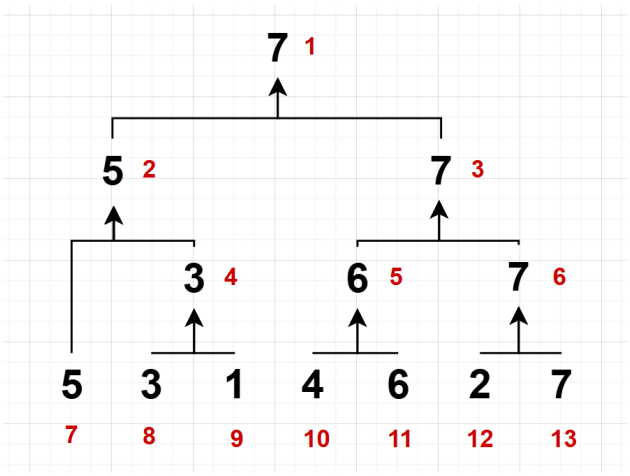

Figure 5: Heap Structure

Solution
For finding the largest we carry out the comparisons as per the heap structure shown above. As we traverse the path of the largest through these heap nodes we are ensured that largest would have beaten the second largest somewhere in the path. Thus finding the largest will take (n-1) comparisons and finding the second largest will take $\lceil logn \rceil$ as we are just traversing the path of the largest through the heap structure.

Data Structuring
The problem of find the largest can be solved by creating an array of elements whose size is equal to number of nodes in the heap data structure i.e. 13. This is shown in Figure 6.

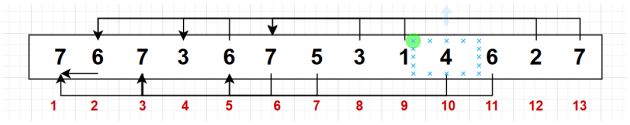

Figure 6: Array Structure

- Populate the array starting from the index l-n+1 where l is the length of the array, n is the number of elements in the given list. For our case this index is 13-7+1 = 7.
- Initialize the counter index as l-n
- Compare i and i-1 index elements for i=l downto 3.
- Populate the max element among the i and i-1 index in the counter index
- Decrease the counter index by 1.
- After the process ends the largest number in the list will be available in the index 1.

To find the second largest we need to traverse the path of largest which can be done as follows:

- Start with i=1
- Set counter as 0
- Compute 2*i
- If the counter is even number then 2*i represents the 2nd largest candidate and 2*i+1 represents the largest number. After examining set i=2*i+1
- If the counter is odd number 2*i represents the largest number and 2*i+1 represents the 2nd largest candidate.
- Increment counter by 1.

After identifying the nodes of largest one can replace the last node of the largest number by -Inf and carry out the same procedure discussed above to find the second largest.

1.2.3 Fibonacci Sequence

Fibonacci Sequence is defined by the following definition: The equation is recursive in nature, thus a natural solution arises.

Initial Solution (Fibonacci Sequence)

```
evalFib(n)
  if(n=0 or n=1) return n
  return evalFib(n-1)+evalFib(n-2)
```

Time Complexity

$$T(n) = T(n-1) + T(n-2) + 1 = \frac{1}{\sqrt{5}}(\frac{\sqrt{5}+1}{2}^{n} - \frac{\sqrt{5}-1}{2}^{n}) = O(2^n) \tag{6}$$

Recursive Structure
The recursive structure is very intuitive with base cases defined as n=0 and n=1. On careful examination there are repeated computations in the recursive structure which can be optimized. One way to optimize such a structure is to store the values of computed nodes in a table. On encountering a computation which has been solved earlier during the computation the value from the table can be reused thus reducing the computation time. This technique is called **Memoization**.

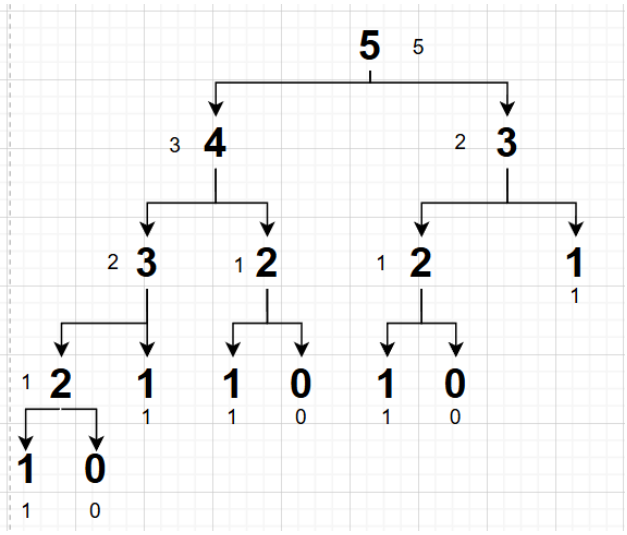

Figure 7: Fibonacci Sequence Recursive Structure

The below table shows the memoization table and the pruned recursive structure. The memoization table consists of two rows. The first row refers to the value of the Fibonacci sequence at a given n. The second row refers to the DONE index which indicates whether the value of Fibonacci sequence at a given n has been computed.

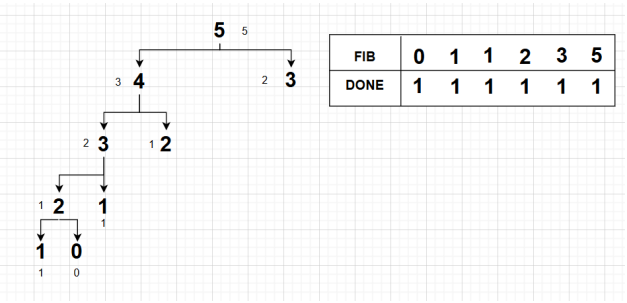

Figure 8: Fibonacci Sequence Memoization

If DONE=1 you retrieve the value from the table. Otherwise you compute the value store the same in the table and change DONE of the corresponding n to 1.

Other Variations

- $f(n) = f(n - \frac{n}{2}) + f(n + 1)$ $n \geq 2$ and $f(n) = 1$ $n < 2$
- $f(n) = f(n + 1) + f(n - \frac{1}{2})$ n is odd and $f(n) = f(n - 1) + f(\frac{n}{2})$ n is even
- $f(n) = f(g(n)) + f(h(n))$ if c(n) is true and $f(n) = b(n)$ if c(n) is false

In the above variation there might be an existence of cycle in which case the computation needs to be terminated. Therefore to accomodate such a sitution our DONE index will have three values:

- DONE=0 means computation is yet to be done
- DONE=1 means a computation cycle has been encountered
- DONE=2 means computation is completed

Pseudocode (Generalized Sequence)

```
evalF(n)
  if(DONE(n)=2)
    Fib(n) ← b(n)
    return Fib(n)
  if(DONE(n)=1)
    "CYCLE ENDS"
  DONE(n)=1
  x ← g(n)
  y ← h(n)
  FIB(n) ← evalF(x) + evalF(y)
  DONE(n)=2
  return FIB(n)
```

The Fibonacci sequence can also be solved using a iterative method as shown below:

Pseudocode (Fibonacci Sequence)

```
FIB[0]=0
FIB[1]=1
evalFib(n)
  for i=2 to n
    FIB[i-1]+FIB[i-2]
```