FADML Week:02

Coin Exchange Problem

Let us consider we have a set **C** of **n coins**:

$$C = \{C_1, C_2, C_3, ..., C_n\}$$

Now, we are faced with the following problem:
Given a value **V** we need to return the minimum number of coins **i** of set **U** from the given set **C** i.e.

$\sum_{i \in U} C_i = V$, where **i** is the minimum possible number.

Solution 1:

We write the following pseudocode for the problem:

Coins(U={list of used coins},**P**={list of not used coins},**u**={sum of **U**}, **p**=**V-u**,**n**={no.of coins used}**)**

{

INITIAL VARIABLES

$U_{min} = NULL$

$m = \infty$

BASE

$if\ (p == 0)\ return\ (U, n)$

$if\ (p < 0)\ return\ (Null, \infty)$

$if\ (P == Null)\ return\ (Null, \infty)$

RECURSIVE SOL.

$for\ i = 1\ to\ n$:

{

$U' = U + \{C_i\}$ (1)

$P' = P - \{C_i\}$ (2)

$< s', m' >= Coins(U', P', u + C_{i'}, p - C_{i'}, n + 1)$ (3)

$if\ (m' < m)$ { (4)

$m = m'$ (5)

$U_{min} = U'$} (6)

}

$return\ (U_{min'}, m)$

Time Complexity Calculation

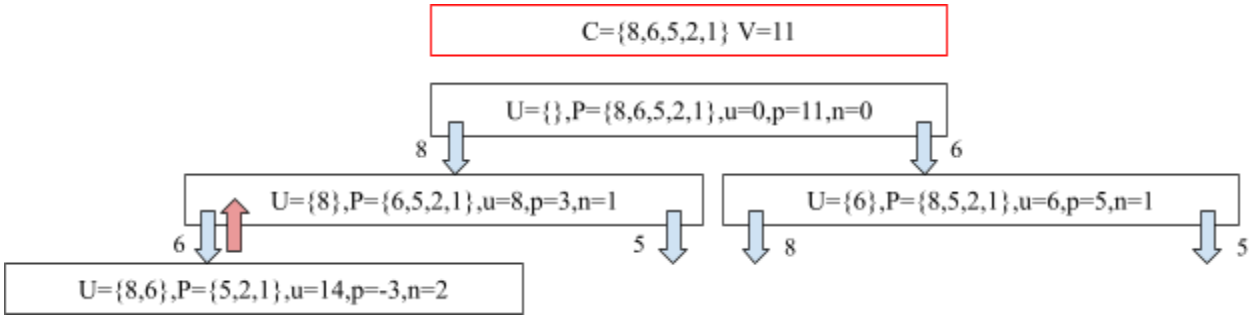
$$T(n) = nT(n-1) + O(n)$$

$nT(n-1)$: For steps 1,2,3
$O(n)$: For steps 4,5,6

Now, solving the above equation we will get:

$$T(n) = O(n!)$$

Recursive Structure of Solution



Solution 2 (Alternative Sol.):

We write the following pseudocode for the problem:

Coins(U={list of used coins},**P**={list of not used coins},**u**={sum of **U**}, **p=V-u**,**n**={no.of coins used}**)**

{

INITIAL VARIABLES

$U_{min} = NULL$

$m = \infty$

BASE

$if\ (p == 0)\ return\ (U, n)$

$if\ (p < 0)\ return\ (Null, \infty)$

$if\ (P == Null)\ return\ (Null, \infty)$

RECURSIVE SOL.

$for\ i = 1\ to\ n$:

{

$< s_0, m_0 >= Coins(U', P' - \{C_i\}, u + C_i, p - C_i, n + 1)$ (1)

$< s_1, m_1 >= Coins(U', P' - \{C_i\}, u, p, n)$ (2)

$if\ (m_1 < m_0)\{$ (3)

$m = m_1$ (4)

$U_{min} = U + \{C_i\}\ \}$ (5)

$else\{$

$U_{min} = U$ (6)

$m = m_0$ $\}$ (7)

$\}$

$return\ (U_{min}, m)$

Time Complexity Calculation

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

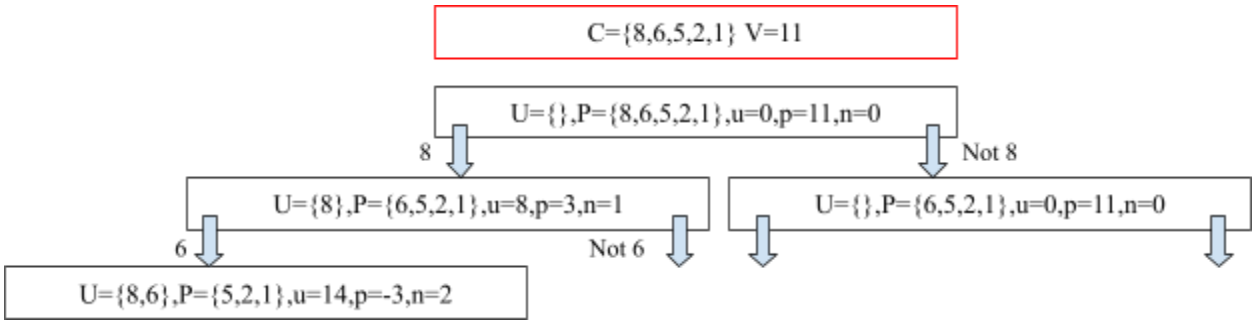
$2T(n-1)$: For steps 1,2

$O(1)$: For steps 3,4,5,6,7

Now, solving the above equation we will get:

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

Recursive Structure of Solution



Observation and Solution Refinement

If in a particular branch the value of **n** is greater than the best solution we have obtained till that time then we can completely ignore that branch and save computation. **This is known as Pruning the Recursive Tree and is part of the Branch and Bound Algorithm.**

Updated Solution 1:

Coins(U={list of used coins},**P**={list of not used coins},**u**={sum of **U**}, **p=V-u**,**n**={no.of coins used}**)**

{

INITIAL VARIABLES

$U_{min} = NULL$

$CB = m = \infty$

BASE

$if\ (p == 0)\ return\ (U, n)$

$if\ (p < 0)\ return\ (Null, \infty)$

$if\ (P == Null)\ return\ (Null, \infty)$
$if\ (CB <= n)\ return\ (Null, \infty)$

RECURSIVE SOL.
$for\ i = 1\ to\ n$:
{
$U' = U + \{C_i\}$
$P' = P - \{C_i\}$
$< s', m' >= Coins(U', P', u + C_i, p - C_i, n + 1)$
$if\ (m' < m)$ {
$m = m'$
$U_{min} = U'$}

$if(m < CB)\ \{CB = m\}$
}
$return\ (U_{min}, m)$

Updated Solution 2:

Coins(U={list of used coins},**P**={list of not used coins},**u**={sum of **U**}, **p**=**V-u**,**n**={no.of coins used}**)**
{
INITIAL VARIABLES
$U_{min} = NULL$
$CB = m = \infty$

BASE
$if\ (p == 0)\ return\ (U, n)$
$if\ (p < 0)\ return\ (Null, \infty)$
$if\ (P == Null)\ return\ (Null, \infty)$
$if\ (CB <= n)\ return\ (Null, \infty)$
RECURSIVE SOL.
$for\ i = 1\ to\ n$:
{
$< s_0, m_0 >= Coins(U', P' - \{C_i\}, u + C_i, p - C_i, n + 1)$
$< s_1, m_1 >= Coins(U', P' - \{C_i\}, u, p, n)$
$if\ (m_1 < m_0)$ {
$m = m_1$
$U_{min} = U + \{C_i\}$ }
$else$ {

$U_{min} = U$
$m = m_0$ }
$if(m < CB)\{CB = m\}$
}
$return\ (U_{min}, m)$

Memoization for Solution Refinement

We can also use a memoization table as shown below to further refine our solution:

	0	1	2	3	4	5	6	7	8	9	10	11
Φ	Φ,0	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞
1	Φ,0	{1},1	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞
2	Φ,0	{1},1	{2},1	{1,2},2	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞	Null,∞
5	Φ,0	{1},1	{2},1	{1,2},2	Null,∞	{5},1	{1,5},2	{2,5},2	{1,2,5},3	Null,∞	Null,∞	Null,∞
6	Φ,0	{1},1	{2},1	{1,2},2	Null,∞	{5},1	{6},1	{1,6},2	{6,2},2	{1,2,6},3	Null,∞	{5,6},2
8	Φ,0	{1},1	{2},1	{1,2},2	Null,∞	{5},1	{6},1	{1,6},2	{8},1	{1,8},2	{2,8},2	{5,6},2

Time and Space Complexity

Space Complexity: $O(kV)$
where k is the number of rows in memory and V is the number of values (in this case from 0 to 11 i.e. 12)

Time Complexity: $O(nlog_2 n) + O(kV)$

$O(nlog_2 n)$: required to sort the given set **C**

Now, the term $O(kV)$ will dominate therefore,
Worst-Case Space Complexity: $O(nV)$
Worst-Case Time Complexity: $O(nlog_2 n) + O(nV) = O(nV)$

Best-Case Space Complexity: $O(V)$
Best-Case Time Complexity: $O(nlog_2 n) + O(V) = O(V)$

Food for Thought

1. If the coins in set **C** had duplicates
2. If the set **C** had ∞ length

Algorithm Design Paradigm

1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithms
4. Branch and Bound
5. Combinatorial Exploration

Divide and Conquer Paradigm

1. Base Case(s): B(z)
2. Decomposition: $<x_1, x_2, \ldots x_n> \leftarrow$D(x)
3. Subroutine Calls: $Y_1 \leftarrow f(x_1), \ldots, Y_k \leftarrow f(x_k)$
4. Recomposition: $Y \leftarrow R(Y_1, Y_2, \ldots, Y_k)$

Recursive

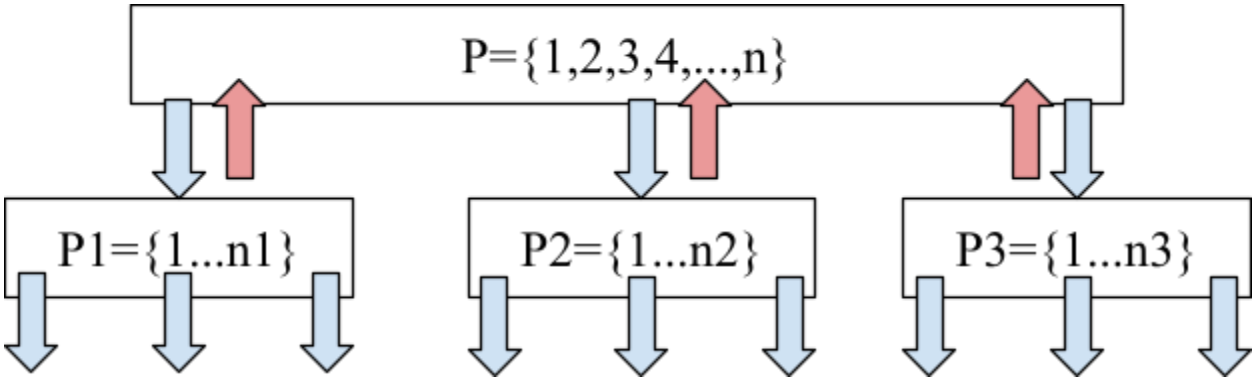
Time Complexity: $T(n) = \sum_{i=1}^{k} T(n_i) + D(n) + R(n)$

$T(n_i)$: Time complexity of subroutine calls

$D(n)$:Time complexity of Decomposition

$R(n)$:Time complexity of Recomposition

Recursive Structure of Divide and Conquer



General Time Complexity Derivation:

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

$T(n) = aT(\frac{n}{b}) + f(n)$, where $D(n) + R(n) = f(n)$

$T(n) = a^2 T(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n)$

.

.

$$T(n) = a^k T(\frac{n}{b^k}) + \sum_{i=0}^{k-1} a^i f(\frac{n}{b^i})$$

Now, $T(\frac{n}{b^k})$ is base case.

Therefore, $b^k = n$

$or,\ k = log_b n$

Hence, the time complexity equation becomes:

$$T(n) = a^{log_b n} T(1) + \sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$$

Now,$T(1) = C$(constant)

Therefore,

$$T(n) = C.n^{log_b a} + \sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$$

Note:

$Cn^{log_b a}$: Time complexity component due to solving the sub-problems

$\sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$: Time complexity component due to decomposition and recombinations

Now, we will introduce **Master's Theorem** where we will compare these 2 components to decide the final time complexity of Divide and Conquer algorithm.

Master's Theorem

Case 1: $f(n) = O(n^{log_b a - \epsilon}),\ \epsilon > 0$

Here, $Cn^{log_b a}$ dominates $\sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$ and the final time complexity results in:

$$T(n) = \theta(n^{log_b a})$$

Case 2: $f(n) = \theta(n^{log_b a})$

Here, $Cn^{log_b a}$ and $\sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$ have the same order of time complexity, therefore:

$$T(n) = O(n^{log_b a} log_b n)$$

Case 3: $f(n) = \Omega(n^{log_b a + \epsilon}),\ \epsilon > 0$

Here, $Cn^{log_b a}$ is dominated by $\sum_{i=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$ and the final time complexity results in:

$$T(n) = \theta(f(n))$$

Searching Problems

Given a set **L**$\{x_1, x_2, ..., x_n\}$ we want to find whether **x** is present in it.

1. **Searching in Unordered Sets**

   Here **L**$\{x_1, x_2, ..., x_n\}$ is unordered.

   We write the following pseudocode for the problem:

Search_U(L,x){

BASE:
$if\ (|L| == 0)$ return **False**
$if\ (|L| == 1)$
{ $if(\mathbf{x} == x_1)$ return **True**
$else$ return **False**
}

DECOMPOSITION:
Split **L** into L_1, L_2 (non-empty)

RECOMPOSITION:
if **Search_U(L_1,x)** return **True**
$elif$ **Search_U(L_2,x)** return **True**
$else$ return **False**

Time Complexity:

$T(n) = T(k) + T(n-k) + O(1)$ $\qquad\qquad$ $D(n) + R(n) = O(1)$

On Solving:
For any k, $T(n) = O(n)$

2. **Searching in Ordered Sets**

   Here **L**$\{x_1, x_2, ..., x_n\}$ is unordered.

   We write the following pseudocode for the problem:

Search_O(L,x){

BASE:
$if\ (|L| == 0)$ return **False**

$if(x_i > \mathbf{x})$ **DECOMPOSITION**

{

if **Search_O(L-**$\{x_i,\ldots,x_n\}$**,x):** return **True** **RECOMPOSITION**

else return **False** **RECOMPOSITION**

}

else **DECOMPOSITION**

{

if **Search_O(L-**$\{x_1,\ldots,x_i\}$**,x):** return **True** **RECOMPOSITION**

else return **False** **RECOMPOSITION**

}

Time Complexity:

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

$If\ i == 1$ or $any\ constant\ k$:

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

$if\ i == n/2$:

$T(n) = T(\frac{n}{2}) + O(1) = O(log_2 n)$

$if\ i == a$: $where,\ a \in (0,1)\ and\ a \neq 0.5$

$T(n) = O(log_{1/a} n),\ if\ a > 0.5$

$T(n) = O(log_{1/(1-a)} n),\ if\ a < 0.5$

Therefore, T(n) minimises at i=2. Hence, it is called BINARY SEARCH.
(the argument still holds even if we tried multi-probing with different values.**)**

FOOD FOR THOUGHT:

L=ordered **S**=ordered	**L**=ordered **S**=unordered
L=unordered **S**=ordered	**L**=unordered **S**=unordered

Where, **L** is the set to be searched in and **S** is the set of elements for which search is made.

Sorting Problems

Given an unordered set **L**$\{x_1, x_2, ..., x_n\}$ we want to sort it.

1. **Selection Sort**

   The logic of the algorithm is given as follows:

   a. Find the max in **L**(let it be $x_{max,1}$)

   b. Find the max in **L-x**$_{max,1}$(let it be $x_{max,2}$)

   c. Repeat steps a and b until **L={}**

   We write the following pseudocode:

   sort_1(L){

   BASE:

   $if(|L| <= 1)$return **L**

   DECOMPOSITION

   $x_i = findMAX(L)$

   $L' = L - \{x_i\}$

   RECURSIVE CALL

   M =**sort_1(**L'**)**

   RECOMPOSITION

   return $(x_i||M)$

   }

   Time Complexity:

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

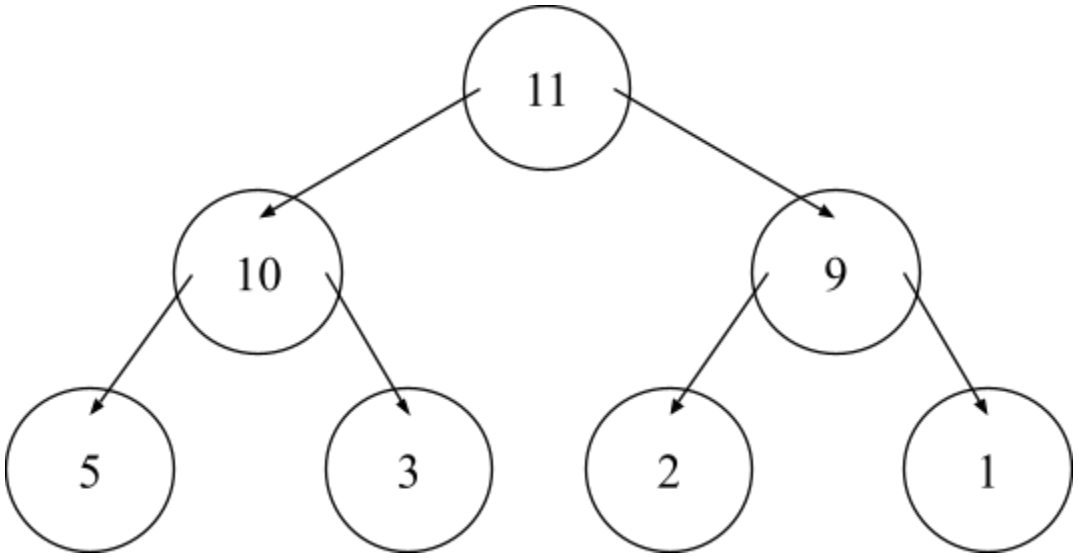
   $T(n) = O(n^2)$

2. **Heap Sort**

   In the selection sort we were repeatedly finding the max from the remaining elements resulting in the $O(n)$ term in (*) equation which ultimately results in $T(n) = O(n^2)$. In Heap Sort we try to improve this max finding by utilising the information already available from the previous max finding exercise, thereby resulting in the $O(n)$ in eq (*) to be $O(log_2 n)$ and finally giving us $T(n) = O(nlog_2 n)$. To achieve our objective we will first use the Max Heap DataStructure followed by adjusting the branches of the Tree everytime the root node is popped out. We will illustrate this with an example:

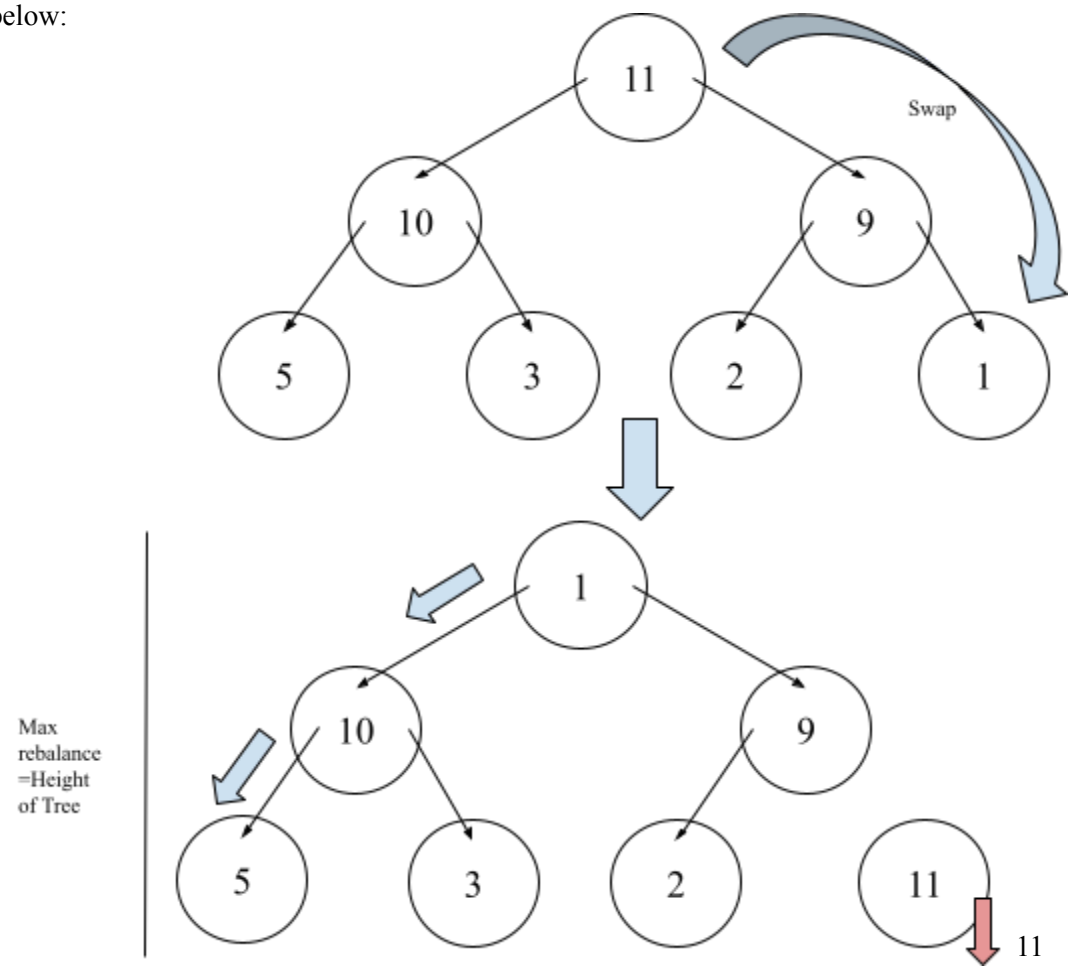
   Let **L**={5, 1, 2, 11, 3, 9, 10}

   Now, we will create the Max Heap (or Priority Queue) for this:



Now, creating this Heap Structure from list **L** takes $O(n)$ time.(**done only once.**)
Finally, the Heap Sort Algorithm can be written as follows:

1. Given **U={}.** We append to **U** the root node of the Heap and pop it out from the Heap structure. (**Note:** to have a balanced tree we swap with the rightmost leaf node and sift down as shown in the diagram.)
2. After the original root node has been removed we will balance the Heap Tree as shown below:



The time complexity of rebalancing the tree is: $O(H) = O(log_2 n)$

3. Repeat Steps 1 and 2 until the Heap is empty.

The Time Complexity for the Heap Sort is:
$T(n) = T(n-1) + O(log_2 n)$
On solving we will arrive at:
$T(n) = O(nlog_2 n)$

3. Insertion Sort

Insertion Sort is a simple sorting algorithm that works by building a sorted portion of the list one element at a time. It picks an element from the unsorted part and places it in the correct position within the sorted part.

We write the following pseudocode:

sort_2(L){

BASE:
$if(|L| <= 1)$return **L**

DECOMPOSITION
M **= sort_2(L-{x**$_n$**})**

RECOMPOSITION
M' **=insert(M,x**$_n$**)**

return M'
}

Time Complexity:
$T(n) = T(n-1) + "Insert"$
Using Arrays we get $O(n)$ for insertion, therefore:
$T(n) = O(n^2)$

Using BSTs or AVLs we get $O(log_2 n)$ for insertion, therefore:
$T(n) = O(nlog_2 n)$

4. Merge Sort

Merge Sort splits the given data into 2 non-empty lists recursively and then builds back the original data by **"merging"** the lists two at a time in such a way that the **"merged list"** is sorted in position. The essence of the algorithm lies in 2 things:

1. The recursive splitting should be done as a fraction of a list (generally 1/2) resulting in a recursive structure taking $O(log_2 n)$ to traverse.

2. The merging step of the algorithm taking $O(n)$ time.

We write the following pseudocode:

sort_3(L){

BASE:
$if(|L| <= 1)$return **L**

DECOMPOSITION
M_1 = **sort_3(**L_1**)**
M_2 = **sort_3(**L_2**)**
RECOMPOSITION
M =**merge(**M_1,M_2**)**
return M
}

Time Complexity:
$T(n) = T(k) + T(n-k) + "Merge"$

We write the following pseudocode for the Merging algorithm:

merge(M_1,M_2**){**
$if(|M_1| == 0)$:return M_2
$if(|M_2| == 0)$:return M_1

$if\ (x_1 < y_1)$:
{M' = **merge(**$M_1 - \{x_1\}, M_2$**)**
return $(x_1||M')$**}**

$else$:
{M' = **merge(**M_1,$M_2 - \{y_1\}$**)**
return $(y_1||M')$**}**
}

Time Complexity for Merging:
$T(n) = T(n-1) + O(1)$
$T(n) = O(n)$

Final Time Complexity:
$T(n) = T(k) + T(n-k) + O(n)$

If $k = const$:

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

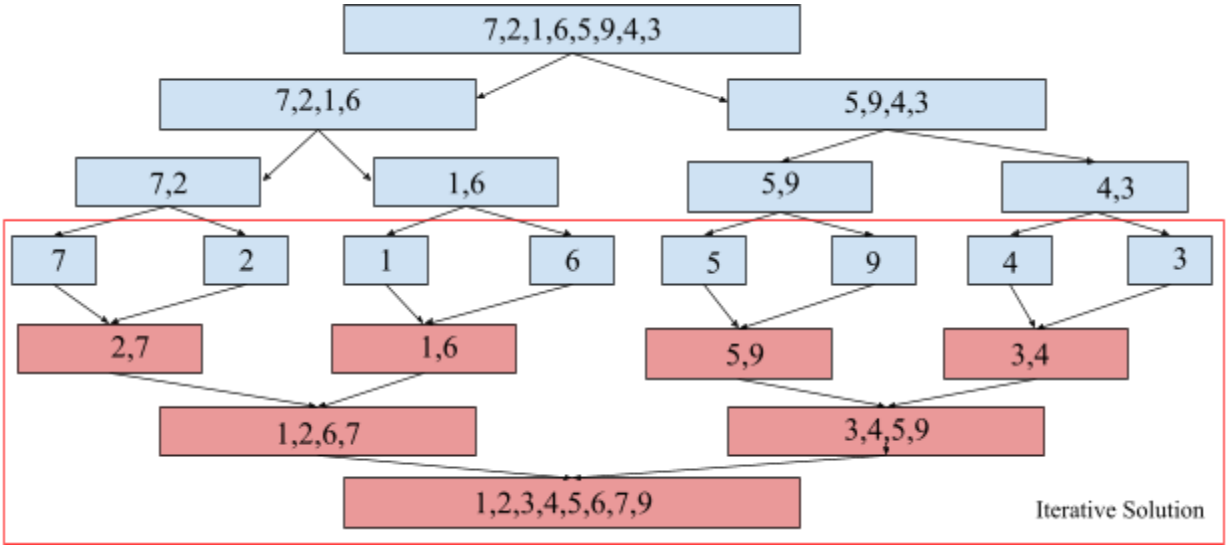
If $k = n/2$:

$T(n) = O(nlog_2 n)$

Iterative Merge Sort

Iterative Merge Sort is a variation of the Merge Sort algorithm that employs a bottom-up, non-recursive approach to sorting. Instead of recursively dividing the array into smaller subarrays, it begins by treating each element as an individual sorted subarray. These subarrays are then merged in pairs to form larger sorted subarrays. This process continues iteratively, doubling the size of the subarrays in each pass, until the entire array is merged into a single sorted sequence. By systematically merging sorted segments, Iterative Merge Sort efficiently organizes the data while maintaining the same time complexity as its recursive counterpart i.e. $O(nlog_2 n)$

Observe the following recursive structure to understand the iterative solution (**red box**)



5. **Quick Sort**

Our experience from merge sort allows us to devise another strategy. What if we could split the lists in such a way that we do not have to write a complicated merging algorithm? → In quick sort we divide the list into three parts: one part containing the pivot element and elements equal to the pivot element, one part containing everything less than the pivot element, and one part containing everything greater than the pivot element. **Pivot Element** is an arbitrary element from the list or in sophisticated procedures there are ways to choose the pivot element (example: median finding algorithm) as the performance of the algorithm depends on the pivot element we get in each stage.

We write the following pseudocode:

sort_4(L){

BASE:
$if(|L| <= 1)$return **L**

DECOMPOSITION
choose random x_i from **L**
$L_1 = L(x < x_i)$
$L_2 = L(x == x_i)$
$L_3 = L(x > x_i)$

RECURSIVE CALL
M_1 =**sort_4(**L_1**)**
M_3 =**sort_4(**L_3**)**

RECOMPOSITION
$M' = \{M_1||L_2||M_3\}$
return M'
}

Time Complexity:
$T(n) = T(k) + T(n-k) + "partition"$
$"partition" = O(n)$
$T(n) = T(k) + T(n-k) + O(n)$

If we get approximately good pivots such that $|L_1| \simeq |L_3|$ then:
$T(n) = O(nlog_2 n)$

If we get extremely bad pivots such that $|L_1|$ and $|L_3|$ are very lop-sided then:
$T(n) = O(n^2)$

Multiplication of 2 n-bit numbers:

Suppose we have multiply the following two bit numbers:

$$\begin{array}{r} 10110 \\ \times\ 10101 \\ \hline 101101 \end{array}$$

The naive approach is to go through all the combination of numbers resulting in a Time Complexity of $O(n^2)$

Solution Refinement:
We can refine our solution by recursively splitting the two numbers:

X=	X1	X2
Y=	Y1	Y2

Therefore,

$$X = 2^{\frac{n}{2}}X_1 + X_2$$

$$Y = 2^{\frac{n}{2}}Y_1 + Y_2$$

Hence,

$$X * Y = 2^n X_1 Y_1 + 2^{\frac{n}{2}}(X_1 Y_2 + Y_1 X_2) + X_2 Y_2$$

The Time Complexity can be written as:

$$T(n) = 4T(\tfrac{n}{2}) + O(n)$$

$4T(\frac{n}{2})$: For the multiplication part

$O(n)$: For the Additions

Using Master's Theorem:

$$n^{log_2 4} > n$$

Therefore, the final time complexity becomes:

$$T(n) = O(n^{log_2 4}) = O(n^2)$$

Further Refinement:
The above multiplication can be rewritten as:

$$X * Y = 2^n X_1 Y_1 + (X_1 + X_2)(Y_1 + Y_2) - X_1 Y_1$$

Note: $X_2 Y_2$ term gets cancelled out.

The Time Complexity becomes:

$$T(n) = 3T(\tfrac{n}{2}) + O(n)$$

$3T(\frac{n}{2})$: For the multiplication part

$O(n)$: For the Additions

Using Master's Theorem, the final time complexity becomes: $O(n^{log_2 3})$

FOOD FOR THOUGHT: Optimise Multiplication and find Time Complexity of a^n