Disjoint Set Operations

Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as S1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.
For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:
S1={1,7,8,9}　　　S2={2,5,10}
S1 U S2={1,2,5,7,8,9,10}

Find:

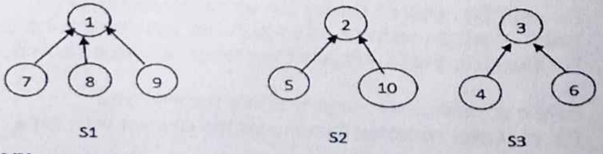
Given the element I, find the set containing i.

Example:
S1={1,7,8,9}　　　S2={2,5,10}　　　s3={3,4,6}
Then,
Find(4)= S3　　　Find(5)=S2　　　Find97)=S1

Set Representation:

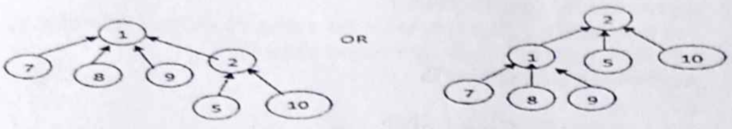
The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:
S1={1,7,8,9}　　　S2={2,5,10}　　　s3={3,4,6}
Then these sets can be represented as



Disjoint Union:

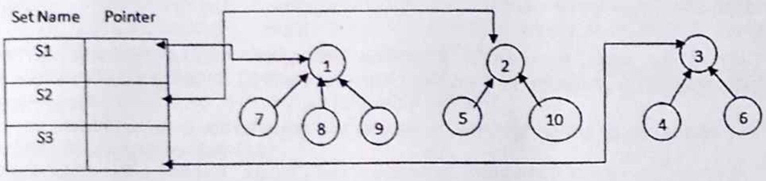
To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.



S1 U S2

Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.



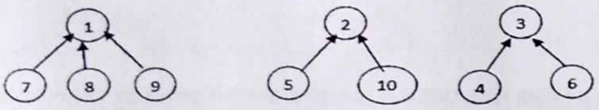
S1 S2 S3

Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



S1 S2 S3

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

```
Algorithm SimpleUnion(i,j)
{
        P[i]:=j;
}
```

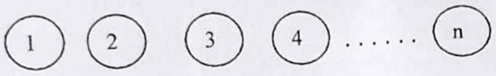
Algorithm for find operation:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

```
Algorithms SimpleFind(i)
{
        while( P[i]≥0) do i:=P[i];
        return i;
}
```

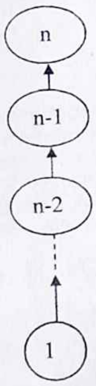
Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3)....... Union(n-1,n) and sequence of Find(1), Find(2)......... Find(n).

The sequence of Union operations results the degenerate tree as below.



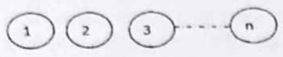
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take time complexity of O ($\sum_{i=1}^{n} i$) = O(n^2).

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

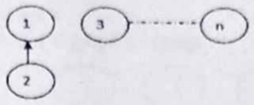
Weighting rule for Union:

If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.

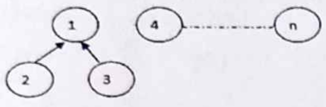
Consider Set



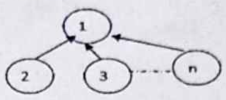
Union(1,2)



Union (1,3)



Union(1,n)



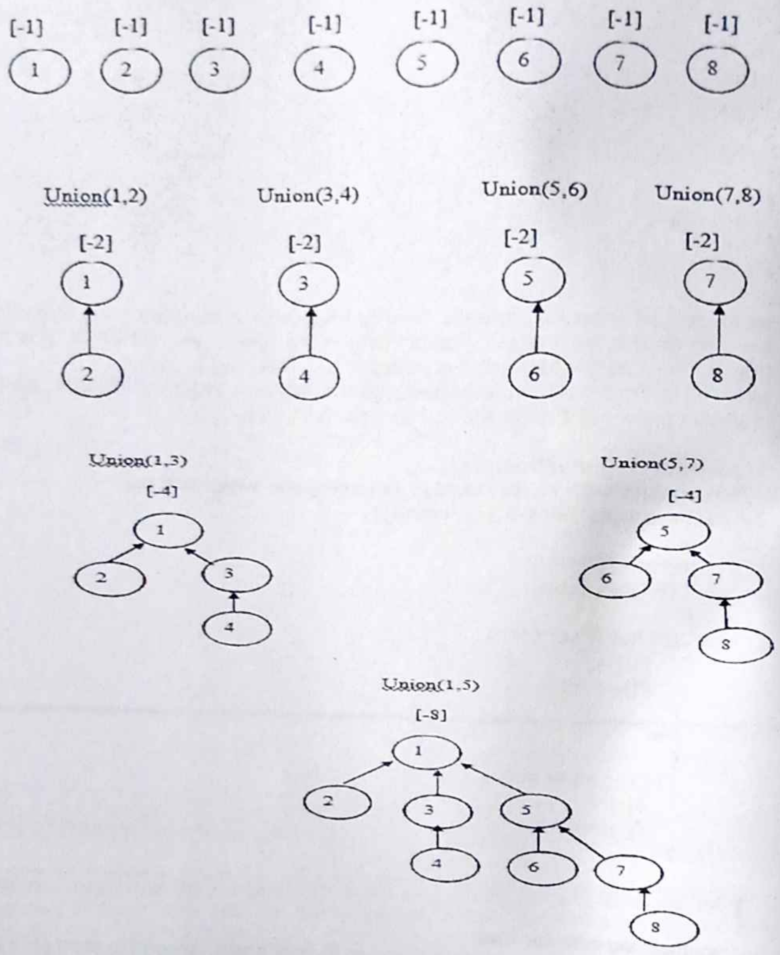
To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i.
Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i  and j , i≠j using the weighted rule
// P[i]=-count[i] and p[j]=-count[j]
{
        temp:= P[i]+P[j];
        if (P[i]>P[j]) then
        {
          // i has fewer nodes
              P[i]:=j;
              P[j]:=temp;
        }
        else
        {
          // j has fewer nodes
              P[j]:=i;
              P[i]:=temp;
        }

}
```

Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i]. Consider the tree created by WeightedUnion() on the sequence of 1≤i≤8. Union(1,2), Union(3,4), Union(5,6) and Union(7,8)



Now process the following eight find operations

Find(8), Find(8)..........................Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .
When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
        r:=i;
        while(P[r]>0) do r:=P[r]; //Find root
         while(i≠r)
         {
                 //reset the parent node from element i to the root
                 s:=P[i];
                 P[i]:=r;
                 i:=s;
         }
}
```

Recurrence Relations

Recurrence Relation for a sequence of numbers S is a formula that relates all but a finite number of terms of S to previous terms of the sequence, namely, $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer. Recurrence relations are also called as difference equations.

Sequences are often most easily defined with a recurrence relation; however the calculation of terms by directly applying a recurrence relation can be time consuming. The process of determining a closed form expression for the terms of a sequence from its recurrence relation is called solving the relation. Some guess and check with respect to solving recurrence relation are as follows:

- Make simplifying assumptions about inputs
- Tabulate the first few values of the recurrence
- Look for patterns, guess a solution
- Generalize the result to remove the assumptions

Examples: Factorial, Fibonnaci, Quick sort, Binary search etc.

Recurrence relation is an equation, which is defined in terms of itself. There is no single technique or algorithm that can be used to solve all recurrence relations. In fact, some recurrence relations cannot be solved. Most of the recurrence relations that we encounter are linear recurrence relations with constant coefficients.

Several techniques like substitution, induction, characteristic roots and generating function are available to solve recurrence relations.

The Iterative Substitution Method:

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. This is a "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we than substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge sort recurrence equation yields the equation.

$$\begin{aligned} T(n) &= 2\,(2\,T(n/2^2) + b\,(n/2)) + b\,n \\ &= 2^2\,T(n/2^2) + 2\,b\,n \end{aligned}$$

Plugging the general equation for T again yields the equation.

$$\begin{aligned} T(n) &= 2^2\,(2\,T(n/2^3) + b\,(n/2^2)) + 2\,b\,n \\ &= 2^3\,T(n/2^3) + 3\,b\,n \end{aligned}$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of merge-sort recurrence equation, the general form is:

$$T(n) = 2^i T(n/2^i) + i\, b\, n$$

Note that the general form of this equation shifts to the base case, T(n) = b, where $n = 2^i$, that is, when i = log n, which implies:

$$T(n) = b\, n + b\, n \log n.$$

In other words, T(n) is O(n log n). In a general application of the iterative substitution technique, we hope that we can determine a general pattern for T(n) and that we can also figure out when the general form of T(n) shifts to the base case.

The Recursion Tree:

Another way of characterizing recurrence equations is to use the recursion tree method. Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.

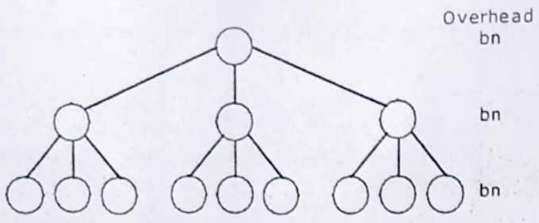
In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function T (n) associated with it. In addition, we associate an overhead with each node v in R, defined as the value of the non-recursive part of the recurrence equation for v.

For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence equation is then solved by summing the overheads associated with all the nodes of R. This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R.

For example, consider the following recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + bn & \text{if } n \geq 3 \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge sort algorithm so that we divide an unsorted sequence into three equal – sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the sub-problem solutions produced by v's children. We illustrate the tree R as follows:



The overheads of the nodes of each level, sum to bn. Thus, observing that the depth of R is $\log_3 n$, we have that T(n) is O(n log n).

The Guess-and-Test Method:

Another method for solving recurrence equations is the guess-and-test technique. This technique involves first making a educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying the guesses, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily", then it is possible that we need to use a slower-growing function. However, using this technique requires case careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess".

Example 2.10.1: Consider the following recurrence equation:

$T(n) = 2T(n/2) + b\, n \log n$. (assuming the base case $T(n) = b$ for $n < 2$)

This looks very similar to the recurrence equation for the merge sort routine, so we might make the following as our first guess:

First guess: $T(n) < c\, n \log n$.

for some constant c>0. We can certainly choose c large enough to make this true for the base case, so consider the case when n > 2. If we assume our first guesses an inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + b\, n \log n \\ &\le 2(c(n/2)\log(n/2)) + b\, n \log n \\ &\le c\, n(\log n - \log 2) + b\, n \log n \\ &\le c\, n \log n - c\, n + b\, n \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to c n log n for $n \ge 2$. Thus, this first guess was not sufficient. Let us therefore try:

Better guess: $T(n) \le c\, n \log^2 n$.

for some constant c > 0. We can again choose c large enough to make this true for the base case, so consider the case when $n \ge 2$. If we assume this guess as an

inductive hypothesis that is true for input sizes smaller than n, then we have
inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + b\,n \log n \\ &\le 2(c(n/2)\log^2(n/2)) + b\,n\log n \\ &\le c\,n(\log^2 n - 2\log n + 1) + b\,n\log n \\ &\le c\,n\log^2 n - 2c\,n\log n + c\,n + b\,n\log n \\ &\le c\,n\log^2 n \end{aligned}$$

Provided $c \ge b$. Thus, we have shown that T (n) is indeed $O(n \log^2 n)$ in this case. We must take care in using this method. Just because one inductive hypothesis for T (n) does not work, that does not necessarily imply that another one proportional to this one will not work.

Example 2.10.2: Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$): $T(n) = 2T(n/2) + \log n$

This recurrence is the running time for the bottom-up heap construction. Which is O(n). Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

First guess: $T(n) \le c\,n$.

For some constant $c > 0$. We can choose c large enough to make this true for the base case, so consider the case when $n \ge 2$. If we assume this guess as an inductive hypothesis that is true of input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\le 2(c(n/2)) + \log n \\ &= c\,n + \log n \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for n > 2. Thus, this first guess was not sufficient, even though T (n) is indeed O (n). Still, we can show this fact is true by using:

Better guess: $T(n) \le c(n - \log n)$

For some constant $c > 0$. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time $n < 8$. So consider the case when $n \ge 8$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\le 2c((n/2) - \log(n/2)) + \log n \\ &= c\,n - 2c\log n + 2c + \log n \\ &= c(n - \log n) - c\log n + 2c + \log n \\ &\le c(n - \log n) \end{aligned}$$

Provided $c \ge 3$ and $n \ge 8$. Thus, we have shown that T (n) is indeed O (n) in this case.

The Master Theorem Method:

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the master method. The master method is a "cook-book" method for determining the asymptotic characterization of a wide variety of recurrence equations. It is used for recurrence equations of the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ a\,T(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Where d > 1 is an integer constant, a > 0, c > 0, and b > 1 are real constants, and f (n) is a function that is positive for $n \geq d$.

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing f (n) to the special function $n^{\log_b a}$ (we will show later why this special function is so important.

The master theorem: Let f (n) and T (n) be defined as above.

1. If there is a small constant $\varepsilon > 0$, such that f (n) is $O(n^{\log_b a - \varepsilon})$, then T (n) is $\Theta\left(n^{\log_b a}\right)$.
2. If there is a constant $K \geq 0$, such that f (n) is $\Theta\left(n^{\log_b a} \log^k n\right)$, then T (n) is $\Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
3. If there are small constant $\varepsilon > 0$ and $\delta < 1$, such that f (n) is $\Omega\left(n^{\log_b a + \varepsilon}\right)$ and $af(n/b) < \delta f(n)$, for $n \geq d$, then T (n) is $\Theta(f(n))$.

Case 1 characterizes the situation where f (n) is polynomial smaller than the special function, $n^{\log_b a}$.

Case 2 characterizes the situation when f (n) is asymptotically close to the special function, and

Case 3 characterizes the situation when f (n) is polynomially larger than the special function.

We illustrate the usage of the master method with a few examples (with each taking the assumption that T (n) = c for n < d, for constants c > 1 and d > 1).

Example 2.11.1: Consider the recurrence T (n) = 4 T (n/2) + n

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in case 1, for f (n) is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$. This means that T (n) is $\Theta(n^2)$ by the master.

Example 2.11.2: Consider the recurrenceT (n) = 2 T (n/2) + n log n

In this case, $n^{\log_b a} = n^{\log_2 2} = n$. Thus, we are in case 2, with k=1, for f (n) is Θ (n log n). This means that T (n) is $\Theta(n \log^2 n)$ by the master method.

Example 2.11.3: consider the recurrenceT (n) = T (n/3) + n

In this case $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for f (n) is $\Omega(n^{1+\varepsilon})$, for $\varepsilon = 1$, and af(n/b) = n/3 = (1/3) f(n). This means that T (n) is Θ (n) by the master method.

Example 2.11.4: Consider the recurrence T (n) = 9 T (n/3) + $n^{2.5}$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. Thus, we are in Case 3, since f (n) is $\Omega(n^{2+\varepsilon})$ (for $\varepsilon=1/2$) and af(n/b) = 9 $(n/3)^{2.5}$ = $(1/3)^{1/2}$ f(n). This means that T (n) is Θ ($n^{2.5}$) by the master method.

Example 2.11.5: Finally, consider the recurrence T (n) = 2T ($n^{1/2}$) + log n

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable k = log n, which lets us write:

$$T(n) = T(2^k) = 2\,T(2^{k/2}) + k$$

Substituting into this the equation S (k) = T (2^k), we get that

$$S(k) = 2\,S(k/2) + k$$

Now, this recurrence equation allows us to use master method, which specifies that S (k) is O (k log k). Substituting back for T (n) implies T (n) is O (log n log log n).

CLOSED FROM EXPRESSION

There exists a closed form expression for many summations

Sum of first n natural numbers (Arithmetic progression)

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots\ldots + n = \frac{n(n+1)}{2}$$

Sum of squares of first n natural numbers

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots\ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of cubes

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \ldots\ldots + n^3 = \frac{n^2(n+1)^2}{4}$$

Geometric progression

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \ldots\ldots + 2^n = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$$

$$\sum_{i=1}^{n} r^i = \frac{r^n-1}{r-1}$$

SOLVING RECURRENCE RELATIONS

Example 2.13.1. Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 7 & , n > 1 \end{cases}$$

Solution: We first start by labeling the main part of the relation as Step 1:

Step 1: Assume n > 1 then,

$$T(n) = 2.T\left(\frac{n}{2}\right) + 7$$

Step 2: Figure out what $T\left(\frac{n}{2}\right)$ is; everywhere you see n, replace it with $\frac{n}{2}$.

$$T\left(\frac{n}{2}\right) = 2.T\left(\frac{n}{2^2}\right) + 7$$

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2.\left[2.T\left(\frac{n}{2^2}\right) + 7\right] + 7$$
$$= 2^2.T\left(\frac{n}{2^2}\right) + 3.7$$

Step 3: let's expand the recursive call, this time, it's $T\left(\frac{n}{2^2}\right)$:

$$T\left(\frac{n}{2^2}\right) = 2.T\left(\frac{n}{2^3}\right) + 7$$

Now substitute this back into the last T(n) definition (last line from step 2):

$$T(n) = 2^2.\left[2.T\left(\frac{n}{2^3}\right) + 7\right] + 7$$

$$2^3 . T\left(\frac{n}{2^3}\right) + 7.7$$

From this, first, we notice that the power of 2 will always match i, current. Second, we notice that the multiples of 7 match the relation $2^i - 1$: 1, 3, 7, 15. So, we can write a general solution describing the state of T (n).

Step 4: Do the ith substitution.

$$T(n) = 2^i . T\left(\frac{n}{2^i}\right) + (2^i - 1) . 7$$

However, how many times could we take these "steps"? Indefinitely? No... it would stop when n has been cut in half so many times that it is effectively 1. Then, the original definition of the recurrence relation would give us the terminating condition T (1) = 1 and us restrict size $n = 2^i$

$$\text{When, } 1 = \frac{n}{2^i}$$

$$\Rightarrow 2^i = n$$

$$\Rightarrow \log_2 2^i = \log_2 n$$

$$\Rightarrow i . \log_2 2 = \log_2 n$$

$$\Rightarrow i = \log_2 n$$

Now we can substitute this "last value for i" back into a our general Step 4 equation:

$$\begin{aligned} T\ (n) &= 2^i . T\left(\frac{n}{2^i}\right) + (2^i - 1) . 7 \\ &= 2^{\log_2 n} . T\left(\frac{n}{2^{\log_2 n}}\right) + (2^{\log_2 n} - 1) . 7 \\ &= n . T(1) + (n - 1) . 7 \\ &= n . 2 + (n - 1) . 7 \\ &= 9 . n - 7 \end{aligned}$$

This implies that T (n) is **O (n).**

Example 2.13.2. Imagine that you have a recursive program whose run time is described by the following recurrence relation:

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 4. n & , n > 1 \end{cases}$$

Solve the relation with iterated substitution and use your solution to determine a tight big-oh bound.

Solution:

Step 1: Assume n > 1 then,

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 4 \cdot n$$

Step 2: figure out what $T\left(\frac{n}{2}\right)$ is:

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2^2}\right) + 4 \cdot \frac{n}{2}$$

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2 \cdot \left[2 \cdot T\left(\frac{n}{2^2}\right) + 4\,\frac{n}{2}\right] + 4 \cdot n$$

$$= 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2 \cdot 4 \cdot n$$

Step 3: figure out what $T\left(\frac{n}{2^2}\right)$ is:

$$T\left(\frac{n}{2^2}\right) = 2 \cdot T\left(\frac{n}{2^3}\right) + 4 \cdot \frac{n}{2^2}$$

Now substitute this back into the last T (n) definition (last time from step 2):

$$T(n) = 2^2 \cdot \left[2 \cdot T\left(\frac{n}{2^3}\right) + 4 \cdot \frac{n}{2^2}\right] + 2 \cdot 4 \cdot n$$

$$= 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3 \cdot 4 \cdot n$$

Step 4: Do the ith substitution.

$$T(n) = 2^i \cdot T\left(\frac{n}{2^i}\right) + i \cdot 4 \cdot n$$

The parameter to the recursive call in the last line will equal 1 when i = $\log_2 n$ (use the same analysis as the previous example). In which case T (1) = 1 by the original definition of the recurrence relation.

Now we can substitute this back into a our general Step 4 equation:

$$T(n) = 2^i \cdot T\left(\frac{n}{2^i}\right) + i \cdot 4 \cdot n$$

$$= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot 4 \cdot n$$

$$= n \cdot T(1) + 4 \cdot n \cdot \log_2 n$$

$$= n + 4 \cdot n \cdot \log_2 n$$

This implies that T (n) is **O(n log n).**

Example 2.13.3. Write a recurrence relation and solve the recurrence relation for the following fragment of program:

Write T(n) in terms of T for fractions of n, for example, T(n/k) or T(n - k).

```
int findmax (int a[ ], int start, int end)
{
        int mid, max1, max2;

        if  (start == end)                          // easy case (2 steps)
                return (a [start]);

        mid = (start + end)  / 2;                   // pre-processing (3)
        max1 = findmax (a,  start, mid);            // recursive call: T(n/2) + 1
        max2 = findmax (a,  mid+1, end);            // recursive call: T(n/2) + 1
        if  (max1 >= max2)                          //post-processing (2)
                return (max1);
        else
                return (max2);
}
```

Solution:

The Recurrence Relation is:

$$T(n) = \begin{cases} 2, & \text{if } n = 1 \\ 2\,T(n/2) + 8, & \text{if } n > 1 \end{cases}$$

Solving the Recurrence Relation by **iteration method:**

Assume n > 1 then T(n) = 2 T (n/2) + 8

Step 1: Substituting n = n/2 , n/4 . . . . for n in the relation:

Since, T (n/2) = 2 T ((n/2) /2) + 8, and T (n/4) = 2T((n/4) /2) + 8,

T (n) = 2 [T(n/2)] + 8 (1)

$$T(n) = 2\ [\ 2\ \ T(n/4) \quad + 8\ ] \quad + 8 \qquad (2)$$

$$T(n) = 2\ [\ 2\ [\ 2\ T(n/8) + 8] + 8\ ] \quad + 8 \qquad (3)$$

$$= 2x2x2\ T(n/2x2x2) + 2x2x8 + 2x8 + 8,$$

Step 2: Do the ith substitution:

$$T(n) = 2^i\ T(n/2^i) + 8\ (2^{i-1} + \ldots + 2^2 + 2 + 1)$$

$$= 2^i\ T(n/2^i) + 8\ (\sum_{k=0}^{i-1} 2^k)$$

$$= 2^i\ T(n/2^i) + 8\ (2^i - 1) \text{ (the formula for geometric series)}$$

Step 3: Define i in terms of n:

If $n = 2^i$, then $i = \log n$ so, $T(n/2^i) = T(1) = 2$

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + 8\ (2^i - 1) \\ &= n\ T(1) + 8\ (n-1) \\ &= 2n + 8n - 8 = 10n - 8 \end{aligned}$$

Step 4: Representing in big-O function:

$T(n) = 10n - 8$ is **O(n)**

Example 2.13.4. If K is a non negative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & , n = 1 \\ 3.T\left(\dfrac{n}{2}\right) + k\ .\ n & , n > 1 \end{cases}$$

has the following solution (for n a power of 2)

$$T(n) = 3\ k\ .\ n^{\log_2 3} - 2k\ .n$$

Solution:

Assuming $n > 1$, we have $T(n) = 3T\left(\dfrac{n}{2}\right) + K\ .\ n$ (1)

Substituting $n = \dfrac{n}{2}$ for n in equation (1), we get

$$T\left(\frac{n}{2}\right) = 3T\left(\frac{n}{4}\right) + k\ \frac{n}{2} \qquad (2)$$

Substituting equation (2) for $T\left(\dfrac{n}{2}\right)$ in equation (1)

$$T(n) = 3\left(3T\left(\frac{n}{4}\right) + k\frac{n}{2}\right) + k\,.\,n$$

$$T(n) = 3^2\,T\left(\frac{n}{4}\right) + \frac{3}{2}\,k\,.\,n + k\,.\,n \qquad (3)$$

Substituting $n = \frac{n}{4}$ for n equation (1)

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{8}\right) + k\frac{n}{4} \qquad (4)$$

Substituting equation (4) for $T = \frac{n}{4}$ in equation (3)

$$T(n) = 3^2\left(3\,T\left(\frac{n}{8}\right) + k\,\frac{n}{4}\right) + \frac{3}{2}\,k\,.\,n + k\,.\,n$$

$$3^3\,T\left(\frac{n}{8}\right) + \frac{9}{4}\,k.n + \frac{3}{2}\,k.n + k.n \qquad (5)$$

Continuing in this manner and substituting n = 2^i, we obtain

$$T(n) = 3^i\,T\left(\frac{n}{2^i}\right) + \frac{3^{i-1}}{2^{i-1}}\,k.n + \ldots\ldots. + \frac{9kn}{4} + \frac{3}{2}\,k.n + k.n$$

$$T(n) = 3^i\,T\left(\frac{n}{2^i}\right) + \left(\frac{\left(\frac{3}{2}\right)^i - 1}{\frac{3}{2} - 1}\right) k\,.\,n \qquad \text{as} \quad \sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1}$$

$$= 3^i\,T\left(\frac{n}{2^i}\right) + 2\,.\left(\frac{3}{2}\right)^i k\,.\,n - 2\,k\,n \qquad (6)$$

As n = 2^i, then i = $\log_2 n$ and by definition as T (1) = k

$$T(n) = 3^i\ k + 2.\ \frac{3i}{n}\,.\,kn - 2kn$$

$$= 3^i\ (3k) - 2kn$$

$$= 3\,k\,.\,3^{\log_2 n} - 2\,k\,n$$

$$= 3k\,n^{\log_2 3} - 2k\,n$$

Example 2.13.5. Towers of Hanoi

The Towers of Hanoi is a game played with a set of donut shaped disks stacked on one of three posts. The disks are graduated in size with the largest on the bottom. The objective of the game is to transfer all the disks from post B to post A moving one disk at a time without placing a larger disk on top of a smaller one. What is the minimum number of moves required when there are n disks?

In order to visualize the most efficient procedure for winning the game consider the following:

1. Move the first n-1 disks, as per the rules in the most efficient manner possible, from post B to post C.

2. Move the remaining, largest disk from post B to post A.

3. Move the n - 1 disks, as per the rules in the most efficient manner possible, from post C to post A.

Let m_n be the number of moves required to transfer n disks as per the rules in the most efficient manner possible from one post to another. Step 1 requires moving n - 1 disks or m_{n-1} moves. Step 2 requires 1 move. Step 3 requires m_{n-1} moves again. We have then,

$$M_n = m_{n-1} + 1 + m_{n-1}, \text{ for } n > 2 = 2m_{n-1} + 1$$

Because only one move is required to win the game with only one disk, the initial condition for this sequence is $m_1 = 1$. Now we can determine the number of moves required to win the game, in the most efficient manner possible, for any number of disks.

$$m_1 = 1$$
$$m_2 = 2(1) + 1 = 3$$
$$m_3 = 2(3) + 1 = 7$$
$$m_4 = 2(7) + 1 = 15$$
$$m_5 = 2(15) + 1 = 31$$
$$m_6 = 2(31) + 1 = 63$$
$$m_7 = 2(63) + 1 = 127$$

Unfortunately, the recursive method of determining the number of moves required is exhausting if we wanted to solve say a game with 69 disks. We have to calculate each previous term before we can determine the next.

Lets look for a pattern that may be useful in determining an explicit formula for m_n. In looking for patterns it is often useful to forego simplification of terms. $m_n = 2m_{n-1} + 1, m_1 = 1$

$$m_1 = 1$$
$$m_2 = 2(1) + 1 \qquad = 2+1$$
$$m_3 = 2(2+1) + 1 \qquad = 2^2+2+1$$
$$m_4 = 2(2^2+2+1) + 1 \qquad = 2^3+2^2+2+1$$
$$m_5 = 2(2^3+2^2+2+1) + 1 \qquad = 2^4+2^3+2^2+2+1$$

$$m_6 = 2(2^4+2^3+2^2+2+1) + 1 = 2^5+2^4+2^3+2^2+2+1$$
$$m_7 = 2(2^5+2^4+2^3+2^2+2+1) + 1 = 2^6+2^5+2^4+2^3+2^2+2+1$$

So we guess an explicit formula:

$$M_k = 2^{k-1} + 2^{k-2} + \ldots\ldots + 2^2 + 2 + 1$$

By sum of a Geometric Sequence, for any real number r except 1, and any integer n ≥ 0.

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r\ \ 1}$$

our formula is then

$$m_k = \sum_{k=0}^{n-1} 2^k = \frac{2^n-1}{2-1} = 2^n - 1$$

This is nothing but **O (2^n)**

Thus providing an explicit formula for the number of steps required to win the Towers of Hanoi Game.

Example 2.13.6 Solve the recurrence relation T (n) = 5 T (n/5) + n/log n

Using iteration method,

T (n) = 25 T (n/25) + n/log (n/5) + n/log n

=125 T (n/125) + n/ log (n/25) + n/log (n/5) + n/log n

= 5^k T (n/5^k) + n/log (n/5^{k-1}) + n/log (n/5^{k-2}) . . . . + n/log n

When n = 5^k

= n * T(1) + n/log 5 + n/log 25 + n/log 125 . . . . . + n/log n

= c n + n (1/log 5 + 1/log 25 + 1/log 125 + ... 1/log n)

= c n + n $\log_5 2$ (1/1 + 1/2 + 1/3 + ... 1/k)

= c n + $\log_5 2$ n log (k)

= c n + $\log_5 2$ n log (log n)

= θ (n log (log n))

Example 2.13.7. Solve the recurrence relation T (n) = 3 T (n/3 + 5) + n/2

Use the substitution method, guess that the solution is T (n) = O (n log n)

We need to prove $T(n) <= c\, n \log n$ for some c and all $n > n_0$.

Substitute $c(n/3 + 5)\log(n/3 + 5)$ for $T(n/3 + 5)$ in the recurrence

$T(n) <= 3 * c(n/3 + 5) * (\log(n/3 + 5)) + n/2$

If we take n_0 as 15, then for all $n > n_0$, $n/3+5 <= 2n/3$, so

$T(n) <= (c\,n + 15\,c) * (\log n/3 + \log 2) + n/2$

$<= c\,n \log n - c\,n \log 3 + 15\,c \log n - 15\,c \log 3 + c\,n + 15\,c + n/2$

$<= c\,n \log n - (c\,n(\log 3-1) + n/2 - 15\,c \log n - 15\,c(\log 3 - 1)$

$<= c\,n \log n$, by choosing an appropriately large constant c

Which implies $T(n) = O(n \log n)$

Similarly, by guessing $T(n) >= c_1 n \log n$ for some c_1 and all $n > n_0$ and substituting in the recurrence, we get.

$T(n) >= c_1 n \log n - c_1 n \log 3 + 15\,c_1 \log n - 15\,c_1 \log 3 + c_1 n + 15\,c_1 + n/2$

$>= c_1 n \log n + n/2 + c_1 n(1 - \log 3) + 15\,c_1 + 15\,c_1(\log n - \log 3)$

$>= c_1 n \log n + n(0.5 + c_1(1 - \log 3) + 15\,c_1 + 15\,c_1(\log n - \log 3)$

by choosing an appropriately small constant c_1 (say $c_1 = 0.01$) and for all $n > 3$

$T(n) >= c1\, n \log n$

Which implies $T(n) = \Omega(n \log n)$

Thus, $T(n) = \theta(n \log n)$

Example 2.13.8. Solve the recurrence relation $T(n) = 2\,T(n/2) + n/\log n$.

Using iteration method,

$T(n) = 4\,T(n/4) + n/\log(n/2) + n/\log n$

$= 8\,T(n/8) + n/\log(n/4) + n/\log(n/2) + n/\log n$

$= 2^k T(n/2^k) + n/\log(n/2^{k-1}) + n/\log(n/2^{k-2}) \ldots\ldots + n/\log n$

When $n = 2^k$

$= n * T(1) + n/\log 2 + n/\log 4 + n/\log 8 \ldots\ldots + n/\log n$

$= c\,n + n(1/\log 2 + 1/\log 4 + 1/\log 8 + \ldots\ldots 1/\log n)$

$= c\,n + n(1/1 + 1/2 + 1/3 + \ldots\ldots 1/k)$

$= c\,n + n \log(k)$

$= c\,n + n \log(\log n)$

$= \theta(n \log(\log n))$

Example 2.13.9: Solve the recurrence relation:

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

Use the substitution method, guess that the solution is T (n) = O (n log n).

We need to prove T (n) <= c n log n for some c and all n > no.

Substituting the guess in the given recurrence, we get

$T(n) <= c(n/2)\log(n/2) + c(n/4)\log(n/4) + c(n/8)\log(n/8) + n$

$<= c(n/2)(\log n - 1) + c(n/4)(\log n - 2) + c(n/8)(\log n - 3) + n$

$<= c\,n\log n\,(1/2 + 1/4 + 1/8) - c\,n/2 - c\,n/2 - 3\,c\,n/8 + n$

$<= c\,n\log n\,(7/8) - (11\,c\,n - 8\,n)/8$

$<= c\,n\log n\,(7/8)$ (if c = 1)

$<= c\,n\log n$

From the recurrence equation, it can be inferred that T (n) > = n. So, T (n) = Ω (n)

Example 2.13.10: Solve the recurrence relation: $T(n) = 28\,T(n/3) + cn^3$

Let us consider : $n^{\log_b^a} \Rightarrow \frac{f(n)}{n^{\log^a{}_b}} = \frac{cn^3}{n^{\log_3^2}}$

According to the law of indices: $\frac{a^x}{a^y}$, we can write a^{x-y}

$$\frac{cn^3}{n^{\log_3^{28}}} = c\,n^{3-\log_3^{28}} = c\ n^r \qquad \text{where } r = 3 - \log_3{}^{28} < 0$$

It can be written as: $h(n) = O(n^r)$ where $r < 0$

f (n) for these from the table can be taken O (1)

$$T(n) = n\log_3^{28}\,[T(1) + h(n)]$$

$$= n\log_3^{28}\,[T(1) + O(1)] = \theta\left(n^{\log_3^{28}}\right)$$

Example 2.13.12: Solve the recurrence relation $T(n) = T(\sqrt{n}) + c$. $n > 4$

$$T(n) = T(n^{1/2}) + C$$
$$T(n)^{1/2} = T(n^{1/2})^{1/2} + C$$
$$T(n^{1/2}) = T(n^{1/4}) + C + C$$
$$T(n) = T(n^{1/4}) + 2c$$
$$T(n) = T(n^{1/2}) + 3c$$
$$= T(n^{1/2i}) + i\,c$$
$$= T(n^{1/n}) + C\log_2 n$$
$$T\left(\sqrt[n]{n}\right) + C\log_2 n = \theta\,(\log n)$$

Example 2.13.13: Solve the recurrence relation

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

$$T(n) = 2\left[2T\,(n^{1/2})^{1/2} + \log\sqrt{n}\right] + \log n$$
$$= 4\,T(n^{1/4}) + 2\log n^{1/2} + \log n$$
$$T(n^{1/4}) = 2\,T(n^{1/2})^{1/4} + \log n^{1/4}$$
$$T(n) = 4\left[2T\left(n^{1/2}\right)^{1/4} + \log n^{1/4}\right] + 2\log n^{1/4} + \log n$$
$$= 8\,T(n^{1/8}) + 4\log n^{1/4} + 2\log n^{1/2} + \log n$$
$$T(n^{1/8}) = 2\,T(n^{1/2})^{1/8} + \log n^{1/8}$$
$$T(n) = 8\left[2T(n^{1/16}) + \log n^{1/8}\right] + {}^{4}\log n^{1/4} + 2\log n^{1/2} + \log n$$
$$= 16\,T(n^{1/6}) + 8\log n^{1/8} + 4\log n^{1/4} + 2\log n^{1/2} + \log n$$
$$= 2^i\,T(n^{1/2^i}) + 2^{i-1}\log n^{1/2^{i-1}} + 2^{i-2}\log n^{1/2^{i-2}} + 2^{i-3}\log n^{1/2^{i-3}} + 2^{i-4}\log n^{1/2^{i-4}}$$
$$= 2^i\,T\left(n^{1/2^i}\right) + \sum_{k=1}^{n} 2^{i-k}\log n^{1/2^{i-k}}$$
$$= nT\left(n^{1/n}\right) + \Sigma\frac{n}{2^k}\log n^{k/n}$$
$$= \theta\,(\log n)$$