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.

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

Plugging the general equation for T again yields the equation.

$$T(n) = 2^2\,(2\,T(n/2^3) + b\,(n/2^2)) + 2\,b\,n$$
$$= 2^3\,T(n/2^3) + 3\,b\,n$$

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.

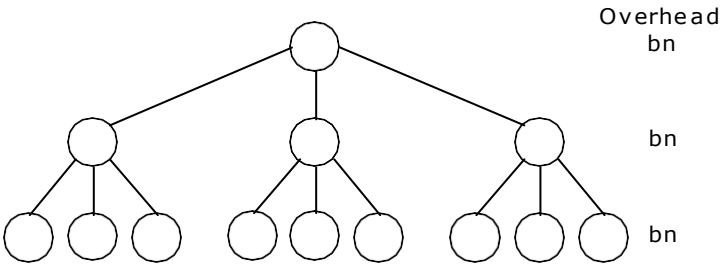
In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function T (n) associated with it. In addition, we associate an overhead with each node v in R, defined as the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

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

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:

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

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

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

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 $n \geq 2$. 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:

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

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:

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

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 $\epsilon > 0$, 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 $\Theta(n \log n)$. This means that T (n) is $\Theta(n \log^2 n)$ by the master method.

Example 2.11.3: consider the recurrence T (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 $\Omega(n^{0+\varepsilon})$, for ε = 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 ε=1/2) and af(n/b) = 9 $(n/3)^{2.5} = (1/3)^{1/2}$ f(n). This means that T (n) is $\Theta(n^{2.5})$ 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)

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

Sum of squares of first n natural numbers

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

Sum of cubes

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

Geometric progression

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

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

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

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 i^{th} substitution.

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

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 $n = 2^i$

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:

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

$$= 2^{\log_2 n} . T\left(\frac{n}{2^{\log_2 n}}\right) + \left(2^{\log_2 n} - 1\right) . 7$$

$$= n . T(1) + (n - 1) . 7$$

$$= n . 2 + (n - 1) . 7$$

$$= 9 . n - 7$$

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,

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

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

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

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

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

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

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

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

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

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

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

Step 4: Do the i^{th} substitution.

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

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)

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

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

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

Step 2: Do the i^{th} 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, $T\ (n/2^i) = T(1) = 2$

$$T(n) = 2^i\ T\ (n/2^i) + 8\ (2^i - 1)$$
$$= n\ T\ (1) + 8\ (n-1)$$
$$= 2n + 8n - 8 = 10n - 8$$

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

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

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 $T(n) = 3T\left(\frac{n}{2}\right) + K\ .\ n$ (1)

Substituting $n = \frac{n}{2}$ 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 $T\left(\frac{n}{2}\right)$ in equation (1)

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

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

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)

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

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

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

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

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

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

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

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

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

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

$$= 3kn^{\log_2 3} - 2kn$$

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. $m_n = 2m_{n-1} + 1$, $m_1 = 1$

$m_1 = 1$

$m_2 = 2(1) + 1 = 2+1$

$m_3 = 2(2+1) + 1 = 2^2+2+1$

$m_4 = 2(2^2+2+1) + 1 = 2^3+2^2+2+1$

$m_5 = 2(2^3+2^2+2+1) + 1 = 2^4+2^3+2^2+2+1$

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

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

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_o.

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 log3 + 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) = Ω (n log n)

Thus, T (n) = θ (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}) . . . . . + n/log n

When n = 2^k

= n * T(1) + n/log 2 + n/log 4 + n/log 8 . . . . . + n/log n

= c n + n (1/log 2 + 1/log 4 + 1/log 8 + ... 1/log n)

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

= c n + n log (k)

= c n + n log (log n)

= θ (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 : $n^{\log_b^a} \Rightarrow \frac{f(n)}{n^{\log_b^a}} = \frac{cn^3}{n^{\log_3^2}}$

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

$\frac{c\,n^3}{n^{\log_3^{28}}} = c\,n^{3-\log_3^{28}} = c\ n^r$ 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)

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

$= n^{\log_3^{28}} [T(1) + 0(1)] = \theta\left(n^{\log_3^{28}}\right)$

Example 2.13.12: Solve the recurrence relation T (n) = T (√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

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

$T(n) = 2\left[2T(n^{1/2})^{1/2} + \log\sqrt{n}\right] + \log n$

$= 4T(n^{1/4}) + 2\log n^{1/2} + \log n$

$T(n^{1/4}) = 2T(n^{1/2})^{1/4} + \log n^{1/4}$

$T(n) = 4\left[2T\left(n^{1/2}\right)^{1/4} + \log n^{1/4}\right] + 2\log n^{1/4} + \log n$

$= 8T(n^{1/8}) + 4\log n^{1/4} + 2\log n^{1/2} + \log n$

$T(n^{1/8}) = 2T(n^{1/2})^{1/8} + \log n^{1/8}$

$T(n) = 8\left[2T(n^{1/16}) + \log n^{1/8}\right] + 4\log n^{1/4} + 2\log n^{1/2} + \log n$

$= 16T(n^{1/6}) + 8\log n^{1/8} + 4\log n^{1/4} + 2\log n^{1/2} + \log n$

$= 2^i T(n^{1/2^i}) + 2^{i-1}\log n^{1/2^{i-1}} + 2^{i-2}\log n^{1/2^{i-2}} + 2^{i-3}\log n^{1/2^{i-3}} + 2^{i-4}\log n^{1/2^{i-4}}$

$= 2^i T\left(n^{1/2^i}\right) + \sum_{k=1}^{n} 2^{i-k}\log n^{1/2^{i-k}}$

$= nT\left(n^{1/n}\right) + \sum \frac{n}{2^k}\log n^{k/n}$

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

Chapter 3

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.
Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances p1, p2, …. Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

```
BINSRCH (a, n, x)
//      array a(1 : n) of elements in increasing order, n ≥ 0,
//      determine whether 'x' is present, and if so, set j such that x = a(j)
//      else return j

{
        low :=1 ; high :=n ;
        while (low ≤ high) do
        {
                mid :=|(low + high)/2|
                if (x < a [mid]) then high:=mid – 1;
                else if (x > a [mid]) then low:= mid + 1
                        else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches			**un-successful searches**
Θ(1),	Θ(log n),	Θ(log n)	Θ(log n)
Best	average	worst	best, average and worst

Analysis for worst case

Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

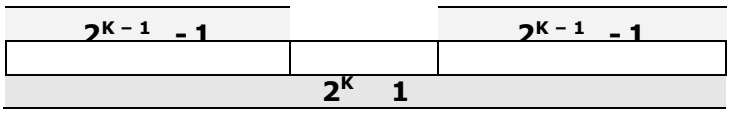
$T(0) = 0$

$T(n) = 1$ if x = a [mid]

$= 1 + T([(n + 1) / 2] - 1)$ if x < a [mid]

$= 1 + T(n - [(n + 1)/2])$ if x > a [mid]

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{2^K - 1 + 1}{2} \right\rceil = 2^{K-1}$ for $K \geq 1$

Giving,

$T(0) = 0$

$T(2^k - 1) = 1$ if x = a [mid]

$= 1 + T(2^{K-1} - 1)$ if x < a [mid]

$= 1 + T(2^{k-1} - 1)$ if x > a [mid]

In the worst case the test x = a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$$
\begin{aligned}
w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\
&= 1 + [1 + w(2^{k-2} - 1)] \\
&= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\
&= \ldots\ldots\ldots \\
&= \ldots\ldots\ldots \\
&= i + w(2^{k-i} - 1)
\end{aligned}
$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

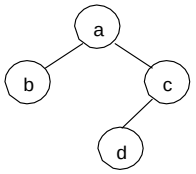
for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

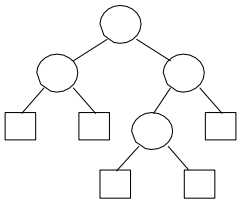
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n–1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

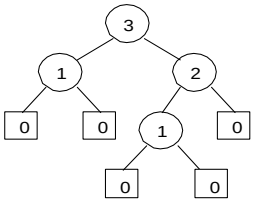


The tree given above in which the empty sub trees appear as square nodes is as follows:

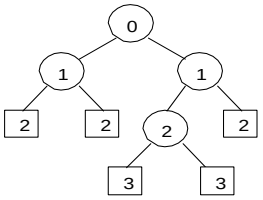


The square nodes are called as external nodes E(T). The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes I(T). A binary tree with n internal nodes has n+1 external nodes.

The height h(x) of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth d(x) of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length I(T) is the sum of the depths of the internal nodes of 'T':

$$I(T) = \sum_{x \in I(T)} d(x)$$

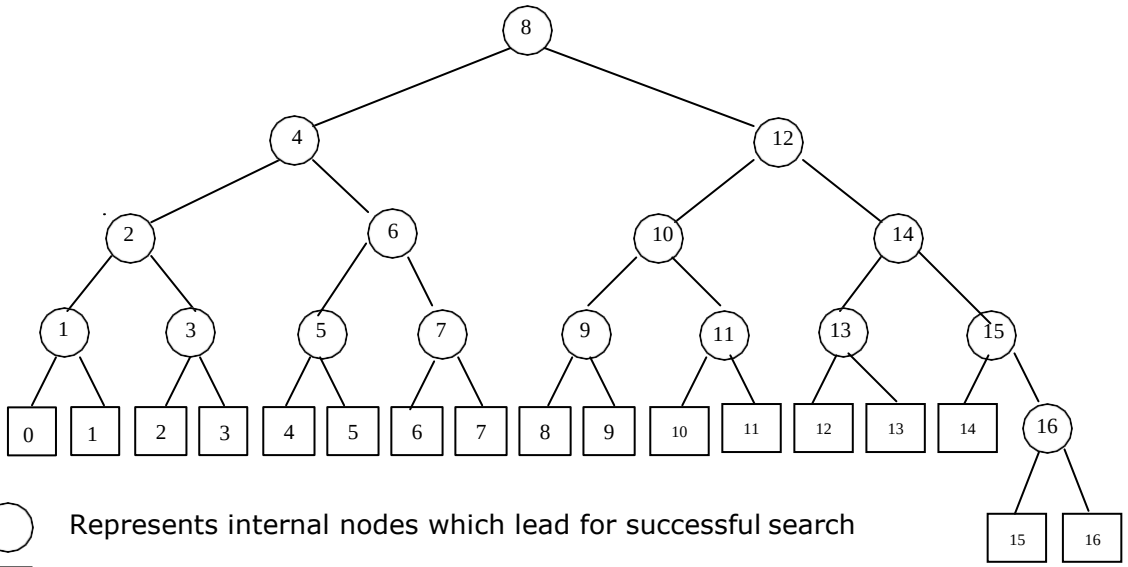
The external path length E(T) is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has I(T) = 4 and E(T) = 12.

A binary tree T with 'n' internal nodes, will have I(T) + 2n = E(T) external nodes.

A binary tree corresponding to binary search when n = 16 is



Represents internal nodes which lead for successful search

External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N+1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
h ptr			

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i ptr							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i ptr						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i ptr					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i ptr				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24			
				i ptr			

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
			h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26		
					i ptr		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	*2*	*3*	*4*
1	13	24	26

h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26	27	28

i ptr

Algorithm

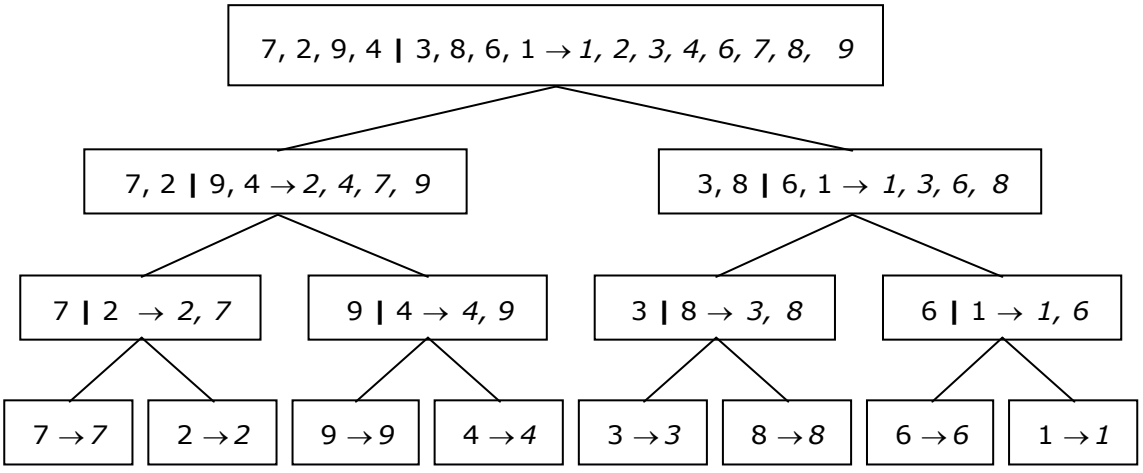
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := ⌊(low + high)/2⌋   //finds where to split the set
        MERGESORT(low, mid)   //sort one subset
        MERGESORT(mid+1, high) //sort the other subset
        MERGE(low, mid, high)   // combine the results
    }
}

Algorithm MERGE (low, mid, high)
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.

```
{
        h :=low; i := low; j:= mid + 1;
        while ((h ≤ mid) and (J ≤ high)) do
        {
                if (a[h] ≤ a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                else
                {
                        b[i] :=a[j]; j := j + 1;
                }
                i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
                }
        else
                for k := h to mid do
                {
                        b[i]  := a[K];  i := i + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

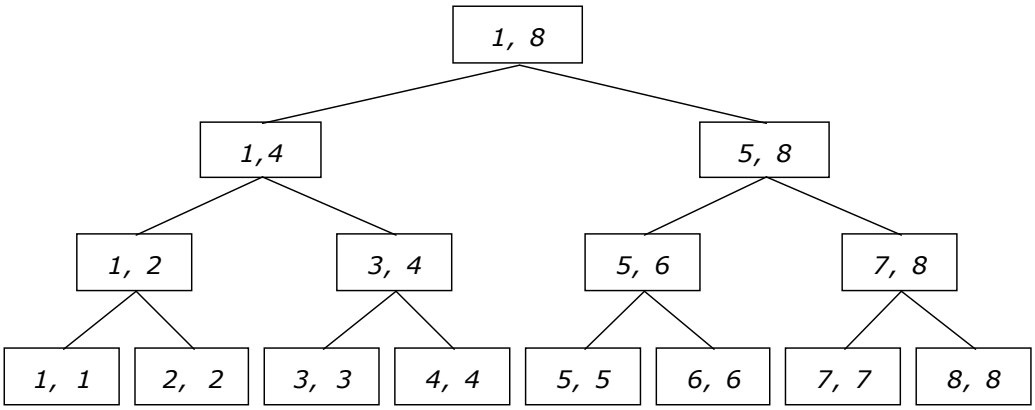
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



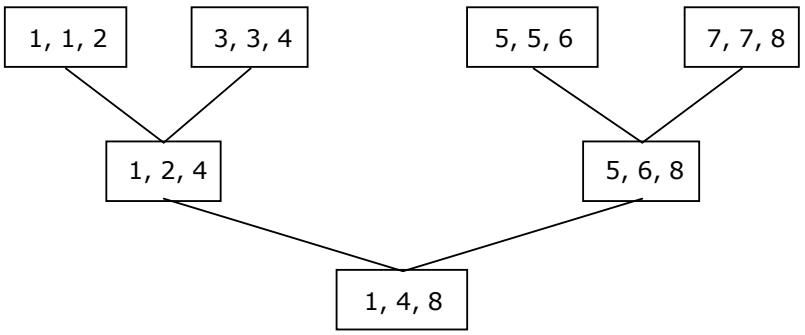
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

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

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

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

We have,

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

Again, by substituting n/4 into the main equation, we see that

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

So we have,

$T(n/4) = 2\ T(n/8) + n$
$T(n) = 8\ T(n/8) + 3n$

Continuing in this manner, we obtain:

$T(n) = 2^k\ T(n/2^k) + K.\ n$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

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

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

Representing this in O notation:

$T(n) =$ **O(n log n)**

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$

$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$

$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$

$C_{22} = A_{21} . B_{12} + A_{22} . B_{22}$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$T(1) = 1$
$T(n) = 8\ T(n/2)$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22}) (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} – B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} – A_{11}) (B_{11} + B_{12})$

$V = (A_{12} – A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S – T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$T(1) = 1$
$T(n) = 7\ T(n/2)$

Solving this for the case of $n = 2^k$ is easy:

$T(2^k) = 7\ T(2^{k-1})$

$= 7^2\ T(2^{k-2})$

$= - - - - - -$
$= - - - - - -$

$= 7^i\ T(2^{k-i})$

Put i = k

$= 7^k\ T(1)$

$= 7^k$

That is, $T(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$= O(n^{\log_2 7}) = O(n^{2.81})$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]

- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].

- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm Algorithm

```
QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 :
n) into ascending order a (n + 1) is considered to be defined and must be greater
than all elements in a(1 : n); A(n + 1) = + ∝ */
{
        if low < high then
        {
                j := PARTITION(a, low, high+1);
                                        // J is the position of the partitioning element
                QUICKSORT(low, j – 1);
                QUICKSORT(j + 1 , high);
        }
}

Algorithm PARTITION(a, m, p)
{
        V ← a(m); i ← m; j ← p;                 // A (m) is the partition element
        do
        {
                loop  i  := i  + 1  until  a(i) ≥ v          // i moves left to right
                loop  j  := j  – 1  until  a(j) ≤ v          // p moves right to left
                if (i < j) then INTERCHANGE(a, i, j)
         }  while (i ≥ j);
        a[m]  :=a[j]; a[j]  :=V;  // the partition element belongs at position P
        return j;
}
```

Algorithm INTERCHANGE(a, i, j)
{
P:=a[i];
a[i] := a[j];
a[j] := p;
}

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
											pivot, j	i	swap pivot & j
											70		
												79 pivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take T (0) = T (1) = 1, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + C\,n \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + C\,n \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

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

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

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

$$= \mathbf{O\ (n^2)} \qquad - \qquad (3)$$

Best Case Analysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

$$T(n) = 2\,T(n/2) + C\,n \qquad - \qquad (4)$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \qquad - \qquad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \qquad - \qquad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \qquad - \qquad (7)$$

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \qquad - \qquad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \qquad - \qquad (9)$$

Which yields, T (n) = C n log n + n = **O(n log n)** - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) + T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)] \

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)
nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n
T(n-1)/n = 2/n + T(n-2)/(n-1)
T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)
. .
. .
T(3)/4 = 2/4 + T(2)/3
T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)
Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 +T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2] + T(0)+
[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+----------+1/n+1/(n+1)]

$$T(n) = (n+1)2[\sum_{2\le k\le n+1} 1/k$$

=2(n+1) [–]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper

place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted	412	71	81	59	14	273	87	
	Sorted	27							
Iteration 1:	unsorted	412	71	81	59	14	273	87	
	Sorted	27	412						
Iteration 2:	unsorted	71	81	59	14	273	87		
	Sorted	27	71	412					
Iteration 3:	unsorted	81	39	14	273	87			
	Sorted	27	71	81	412				
Iteration 4:	unsorted	59	14	273	87				
	Sorted	274	59	71	81	412			
Iteration 5:	unsorted	14	273	87					
	Sorted	14	27	59	71	81	412		
Iteration 6:	unsorted	273	87						
	Sorted	14	27	59	71	81	273	412	
Iteration 7:	unsorted	87							
	Sorted	14	27	59	71	81	87	273	412

Chapter 4

Greedy Method

GENERAL METHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm.*

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
        solution  :=  ϕ;                  // initialize the solution to empty
        for i:=1 to n do
        {
                x := select (a);
                if  feasible (solution, x) then
                        solution := Union (Solution, x);
        }
        return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of p_i x_i is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i \; x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i \quad x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
        for i := 1 to n do x[i]  := 0.0                  // initialize x
        U := m;
        for i := 1 to n do
        {
                if  (w(i) > U) then break;
                x [i] := 1.0; U := U – w[i];
        }
        if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time.

Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i\ x_i$	$\sum p_i\ x_i$
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x_1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x_2 = 2/15

x_1	x_2	x_3	$\sum w_i\ x_i$	$\sum p_i\ x_i$
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

3. Considering the objects in the order of non-decreasing weights w_i.

x_1	x_2	x_3	$\sum w_i\ x_i$	$\sum p_i\ x_i$
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio p_i / w_i .

$\mathbf{p_1/w_1}$	$\mathbf{p_2/w_2}$	$\mathbf{p_3/w_3}$
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / x_i. Select the object with the maximum p_i / x_i ratio, so, x_2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / x_i ratio, so x_3 = ½ and the profit earned is 7.5.

x_1	x_2	x_3	$\sum w_i\ x_i$	$\sum p_i\ x_i$
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

This solution is the optimal solution.

4.4. OPTIMAL STORAGE ON TAPES

There are 'n' programs that are to be stored on a computer tape of length 'L'. Each program 'i' is of length l_i, $1 \le i \le n$. All the programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L'.

We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $i = i_1, i_2, \ldots, i_n$, the time t_J needed to retrieve program i_J is proportional to

$$\sum_{1 \le k \le j} l_{i_k}$$

If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is:

$$\frac{1}{n} \cdot \sum_{1 \le J \le n} t_j$$

For the optimal storage on tape problem, we are required to find the permutation for the 'n' programs so that when they are stored on the tape in this order the MRT is minimized.

$$d(I) = \sum_{J=1}^{n} \sum_{K=1}^{J} l_{i_k}$$

Example

Let n = 3, (l_1, l_2, l_3) = (5, 10, 3). Then find the optimal ordering?

Solution:

There are n! = 6 possible orderings. They are:

Ordering I	d(I)		
1, 2, 3	5 + (5 +10) +(5 + 10 + 3)	=	38
1, 3, 2	5 + (5 + 3) + (5 + 3 + 10)	=	31
2, 1, 3	10 + (10 + 5) + (10 + 5 + 3)	=	43
2, 3, 1	10 + (10 + 3) + (10 + 3 + 5)	=	41
3, 1, 2	3 + (3 + 5) + (3 + 5 + 10)	=	29
3, 2, 1	3 + (3 + 10) + (3 + 10 + 5)	=	34

From the above, it simply requires to store the programs in non-decreasing order (increasing order) of their lengths. This can be carried out by using a efficient sorting algorithm (Heap sort). This ordering can be carried out in O (n log n) time using heap sort algorithm.

The tape storage problem can be extended to several tapes. If there are m > 1 tapes, T_o, ... ,T_{m-1}, then the programs are to be distributed over these tapes.

The total retrieval time (RT) is $\sum_{J=0}^{m-1} d(I_J)$

The objective is to store the programs in such a way as to minimize RT.

The programs are to be sorted in non decreasing order of their lengths l_i's, $l_1 \le l_2 \le$ .. . . l_n.

The first 'm' programs will be assigned to tapes T_o, ... ,T_{m-1} respectively. The next 'm' programs will be assigned to T_0, ... ,T_{m-1} respectively. The general rule is that program i is stored on tape T_i mod m.

Algorithm:

The algorithm for assigning programs to tapes is as follows:

Algorithm Store (n, m)

```
// n is the number of programs and m the number of tapes
{
        j := 0;                         // next tape to store on
        for i :=1 to n do
        {
                Print ('append program', i, 'to permutation for tape', j);
                j := (j + 1) mod m;
        }
}
```

On any given tape, the programs are stored in non-decreasing order of their lengths.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \leq r \leq k$ are the jobs in 'j' and $d (j [1]) \leq d (j[2]) \leq \ldots \leq d (j[k])$. To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d [J[r]] \leq r$, $1 \leq r \leq k+1$.

Example:

Let n = 4, $(P_1, P_2, P_3, P_4,) = (100, 10, 15, 27)$ and $(d_1\ d_2\ d_3\ d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines**.**

Algorithm GreedyJob (d, J, n)

// J is a set of jobs that can be completed by their deadlines.

```
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

OPTIMAL MERGE PATERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
        treenode * lchild;
        treenode * rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{
        for i := 1  to n – 1 do
        {
                pt  ← new treenode
                (pt → lchild) ←  least (list);          //  merge  two  trees  with  smallest
        lengths
                (pt → rchild) ← least  (list);
                (pt → weight) ← ((pt → lchild) → weight) + ((pt → rchild) → weight);
                 insert (list, pt);
        }
        return  least  (list);                          // The tree left  in list is the      merge
tree
}
```

GRAPH ALGORITHMS

Basic Definitions:

- **Graph G** is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote n := |V|, m := |E|.
- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If (u, v) $\in$ E, then vertices u, and v are adjacent.
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge e $\in$ E. The graph which has such function assigned is called **weighted**.
- **Degree** of a vertex v is the number of vertices u for which (u, v) $\in$ E (denote deg(v)). The number of **incoming edges** to a vertex v is called **in–degree** of the vertex (denote indeg(v)). The number of **outgoing edges** from a vertex is called **out-degree** (denote outdeg(v)).

Representation of Graphs:

Consider graph G = (V, E), where V= $\{v_1, v_2,\ldots,v_n\}$.

Adjacency matrix represents the graph as an n x n matrix A = $(a_{i,j})$, where

$$a_{i,\,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise} \end{cases}$$

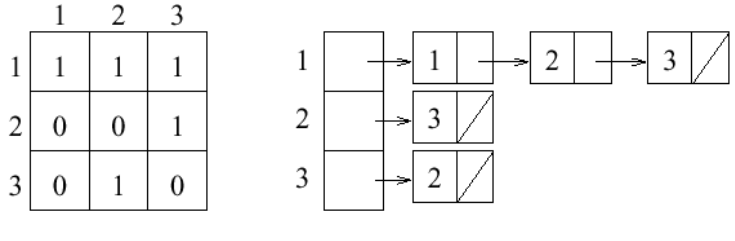
The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

$$a_{i,\,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ default, & otherwise, \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be ∞, meaning value larger than any other value).

Adjacency List: An array Adj [1 . . . . . n] of pointers where for 1 $\leq$ v $\leq$ n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



Adjacency matrix Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) ε E. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices (v_1, v_2, ... , v_k, $v_{k+1} = v_1$), where for all i, (v_i, v_{i+1}) ε E and all vertices in the cycle are distinct except pair v_1, v_{k+1}.

Subgraphs and Spanning Trees:

Subgraphs: A graph G′ = (V′, E′) is a subgraph of graph G = (V, E) iff V′ $\subseteq$ V and E′ $\subseteq$ E.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v. If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

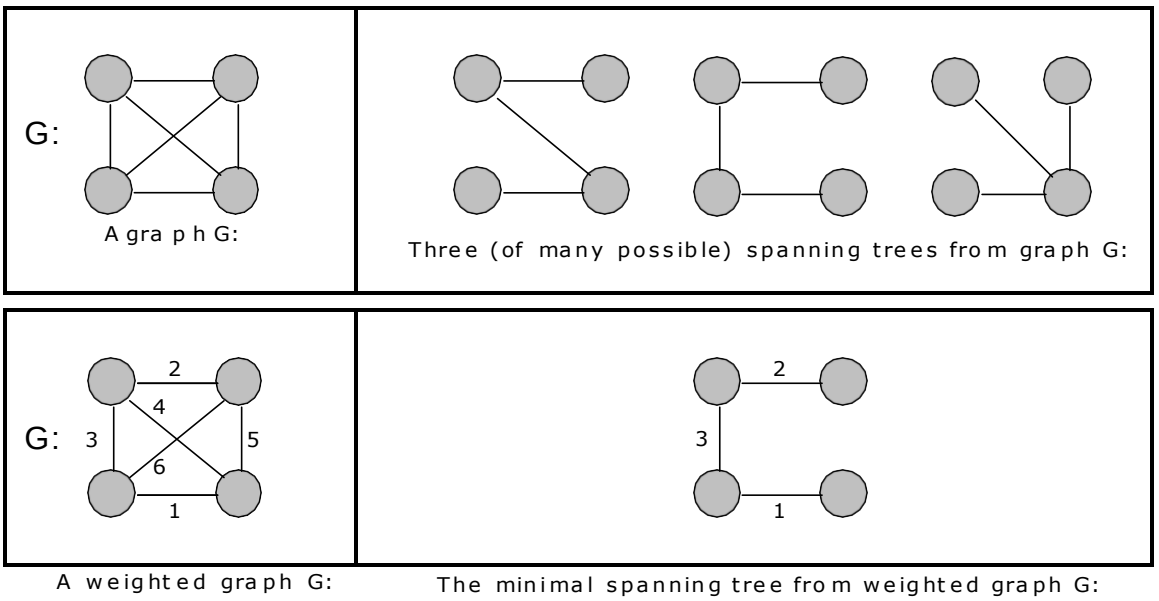
Lemma 1: *Let T be a spanning tree of a graph G. Then*

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w (T) is the sum of weights of all edges in T. The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

Algorithm Kruskal (E, cost, n, t)

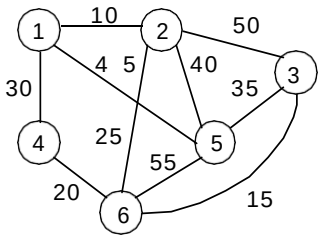
```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify;
        for  i := 1 to n do parent [i] := -1;
                                                // Each vertex is in a different set.
        i := 0; mincost := 0.0;
        while ((i < n -1) and (heap not empty)) do
        {
                Delete a minimum cost edge (u, v) from the heap and
                re-heapify using Adjust;
                j := Find (u); k := Find (v);
                if  (j ≠ k) then
                {
                        i := i + 1;
                        t [i, 1] := u; t [i, 2] := v;
                        mincost :=mincost + cost [u, v];
                        Union (j, k);
                }
        }
        if (i ≠ n-1) then write ("no spanning tree");
        else return mincost;
}
```

Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity

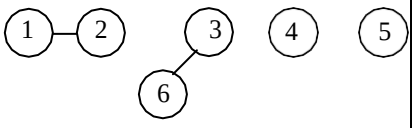
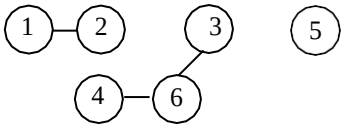
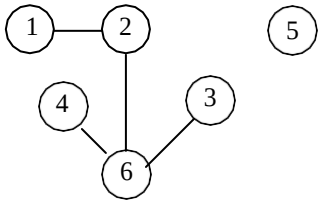
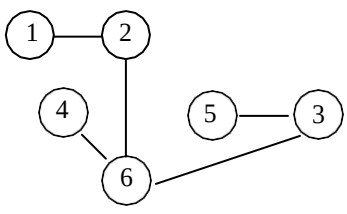
Example 1:



Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Find**s on the vertex sets suffice. When an edge is included in T, two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			{1}, {2}, {3}, {4}, {5}, {6}	
(1, 2)	10		{1, 2}, {3}, {4}, {5}, {6}	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		{1, 2}, {3, 6}, {4}, {5}	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		{1, 2}, {3, 4, 6}, {5}	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		{1, 2, 3, 4, 6}, {5}	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		{1, 2, 3, 4, 5, 6}	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorithm Algorithm Prim

```
(E, cost, n, t)
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or ∝ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for   i :=1 to n do                             // Initialize near
                if  (cost [i, l] < cost [i, k]) then near [i] := l;
                else near [i] := k;
        near [k] :=near [l] := 0;
        for  i:=2 to n -  1 do                          // Find n - 2 additional edges for t.
        {
                Let j be an index such that near [j] ≠ 0 and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j];
                mincost := mincost + cost [j, near [j]];
                near [j] := 0
                for   k:= 1 to n do                             // Update near[].
                        if ((near [k] ≠ 0) and (cost [k, near [k]] > cost [k, j]))
                                then near [k] := j;
        }
        return mincost;
}
```

Running time:

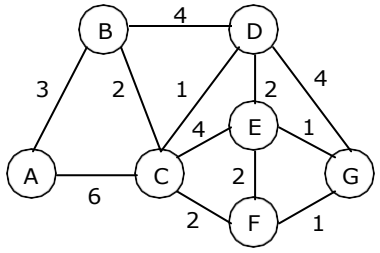
We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n^2) time when we implement dist with array, O (n + $|E|$ log n) when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in θ (1 + deg (u)) time. Therefore the running time is:

$$\Theta\left(\sum_{v \in V} 1+\deg(v)\right) = \Theta\left(n + \sum_{v \in V} \deg(v)\right) = \Theta(n+m)$$

EXAMPLE 1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.

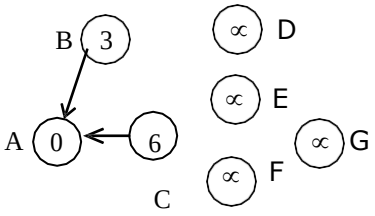


SOLUTION:

The cost adjacency matrix is $\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$

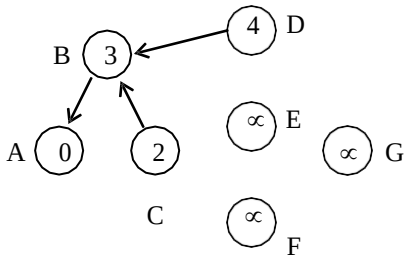
The stepwise progress of the prim's algorithm is as follows:

Step 1:



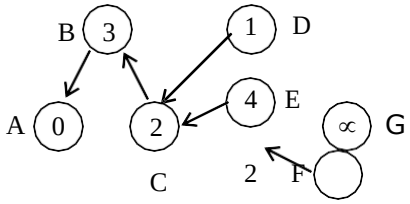
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



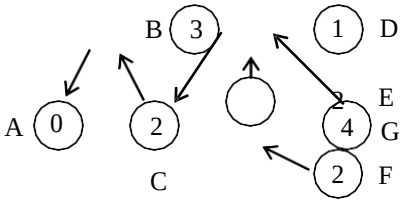
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



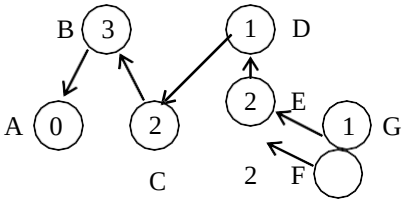
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step 4:



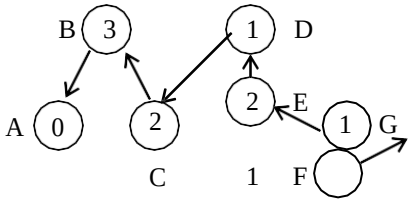
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



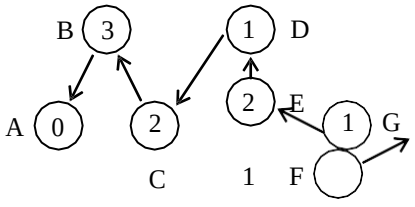
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

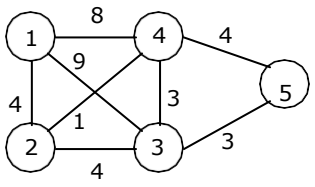
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

EXAMPLE 2:

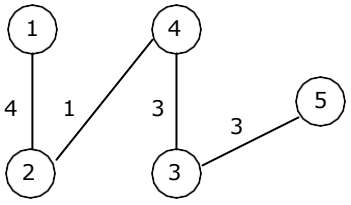
Considering the following graph, find the minimal spanning tree using prim's algorithm.



The cost adjacent matrix is $\begin{pmatrix} \infty & 4 & 9 & 8 & \infty \\ 4 & \infty & 4 & 1 & \infty \\ 9 & 4 & \infty & 3 & 3 \\ 8 & 1 & 3 & \infty & 4 \\ \infty & \infty & 3 & 4 & \infty \end{pmatrix}$

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

<u>The steps as per the algorithm are as follows:</u>

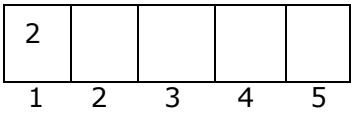
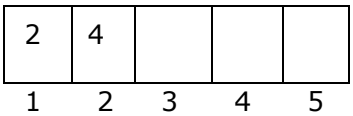
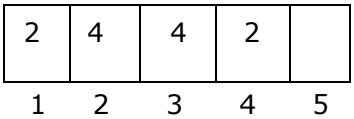
Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4
Min cost = cost (2, 4) = 1

T [1, 1] = 2

T [1, 2] = 4

for i = 1 to 5 Begin i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2 i = 2 is cost (2, 4) < cost (2, 2) 1 < ∝, Yes So near [2] = 4 i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4 i = 4 is cost (4, 4) < cost (4, 2) ∝ < 1, no So near [4] = 2 i = 5 is cost (5, 4) < cost (5, 2) 4 < ∞, yes So near [5] = 4 end near [k] = near [l] = 0 near [2] = near[4] = 0	Near matrix       	Edges added to min spanning tree: T [1, 1] = 2 T [1, 2] = 4
for i = 2 to n-1 (4) do **i = 2** for j = 1 to 5 j = 1 near(1)≠0 and cost(1, near(1)) 2 ≠ 0 and cost (1, 2) = 4 j = 2 near (2) = 0 j = 3 is near (3) ≠ 0 4 ≠ 0 and cost (3, 4) = 3		

j = 4
near (4) = 0

J = 5
Is near (5) $\neq$ 0
4 $\neq$ 0 and cost (4, 5) = 4

select the min cost from the above obtained costs, which is 3 and corresponding J = 3

min cost = 1 + cost(3, 4)
= 1 + 3 = 4

T (2, 1) = 3
T (2, 2) = 4

T (2, 1) = 3
T (2, 2) = 4

2	0	0	0	4
1	2	3	4	5

Near [j] = 0
i.e. near (3) =0

<u>for (k = 1 to n)</u>

K = 1
is near (1) $\neq$ 0, yes
2 $\neq$ 0
and cost (1,2) > cost(1, 3)
4 > 9, No

K = 2
Is near (2) $\neq$0, No

K = 3
Is near (3) $\neq$ 0, No

K = 4
Is near (4) $\neq$ 0, No

2	0	0	0	3
1	2	3	4	5

K = 5
Is near (5) $\neq$ 0
4 $\neq$ 0, yes
and is cost (5, 4) > cost (5, 3)
4 > 3, yes
than near (5) = 3

<u>i = 3</u>

<u>for (j = 1 to 5)</u>
J = 1
is near (1) $\neq$0
2 $\neq$ 0
cost (1, 2) = 4

J = 2
Is near (2) $\neq$0, No

J = 3
Is near (3) ≠ 0, no
Near (3) = 0

J = 4
Is near (4) ≠ 0, no
Near (4) = 0

J = 5
Is near (5) ≠ 0
Near (5) = 3 ➔ 3 ≠ 0, yes
And cost (5, 3) = 3

Choosing the min cost from the above obtaining costs which is 3 and corresponding J = 5

Min cost = 4 + cost (5, 3)
= 4 + 3 = 7

T (3, 1) = 5
T (3, 2) = 3

Near (J) = 0 ➔ near (5) = 0

<u>for (k=1 to 5)</u>

k = 1
is near (1) ≠ 0, yes
and cost(1,2) > cost(1,5)
4 > ∝, No

K = 2
Is near (2) ≠ 0 no

K = 3
Is near (3) ≠ 0 no

K = 4
Is near (4) ≠ 0 no

K = 5
Is near (5) ≠ 0 no

<u>i = 4</u>

<u>for J = 1 to 5</u>
J = 1
Is near (1) ≠ 0
2 ≠ 0, yes
cost (1, 2) = 4

j = 2
is near (2) ≠ 0, No

2	0	0	0	0
1	2	3	4	5

T (3, 1) = 5
T (3, 2) = 3

J = 3 Is near (3) ≠ 0, No Near (3) = 0 J = 4 Is near (4) ≠ 0, No Near (4) = 0 J = 5 Is near (5) ≠ 0, No Near (5) = 0 Choosing min cost from the above it is only '4' and corresponding J = 1 Min cost = 7 + cost (1,2) = 7+4 = 11 T (4, 1) = 1 T (4, 2) = 2 Near (J) = 0 ➔ Near (1) = 0 <u>for (k = 1 to 5)</u> K = 1 Is near (1) ≠ 0, No K = 2 Is near (2) ≠ 0, No K = 3 Is near (3) ≠ 0, No K = 4 Is near (4) ≠ 0, No K = 5 Is near (5) ≠ 0, No End.	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	T (4, 1) = 1 T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

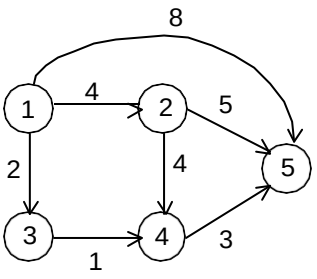
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the

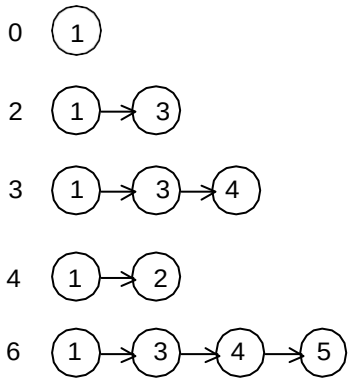
shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Graph



Shortest Paths

Algorithm:

Algorithm Shortest-Paths (v, cost, dist, n)

```
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
        for  i :=1 to n do
        {
                S  [i] :=  false;                       // Initialize S.
                dist [i] :=cost [v, i];
        }
        S[v] := true; dist[v]  := 0.0;                  // Put v in S.
        for num := 2 to n – 1 do
        {
                Determine n - 1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u]  :=  true;                                 // Put u is S.
                for (each w adjacent to u with S [w] = false) do
                        if (dist [w] > (dist [u] + cost [u,   w]) then       // Update distances
                                dist [w] := dist [u] + cost [u, w];
        }
}
```

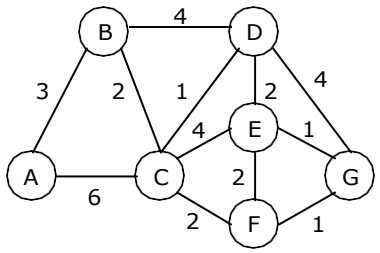
Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements A
- at most m = |E| times decrease the value of an item mB
- 'n' times select the smallest value nC
- For array A = O (n); B = O (1); C = O (n) which gives O (n^2) total.
- For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

Example 1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

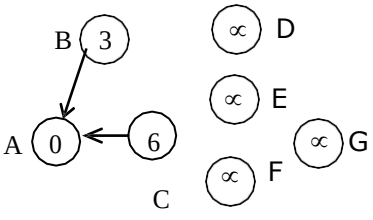
The cost adjacency matrix is $\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$

The problem is solved by considering the following information:

- Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.
- Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.
- Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

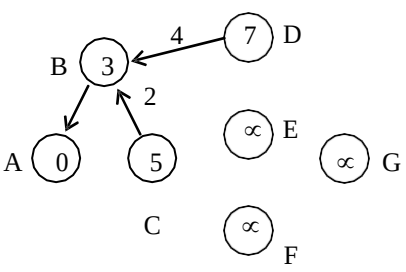
The progress of Dijkstra's algorithm on the graph shown above is as follows:

Step 1:



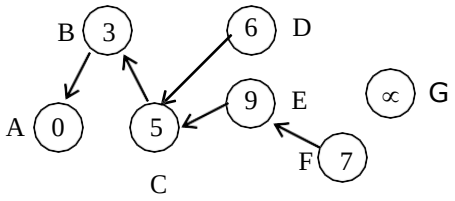
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



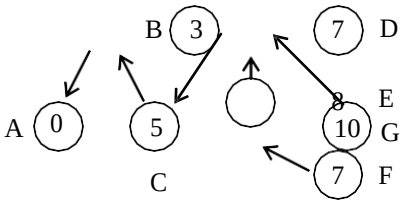
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



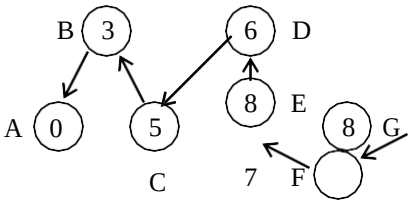
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∝
Next	*	A	B	C	C	C	A

Step 4:



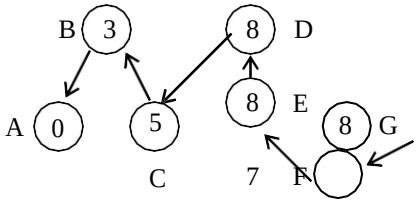
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



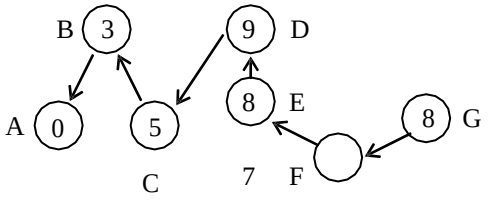
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F