S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.

Department of Computer Science & Engineering

Course: Design And Analysis of Algorithms (18CS42)

Module 2: Divide And Conquer,

Prof. A. A. Daptardar
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Definition

- It is a top-down technique for designing algorithms that consists of dividing the problem into smaller sub-problems hoping that the solutions of the sub-problems are easier to find & then combine the partial solutions onto the solutions of the original problems.

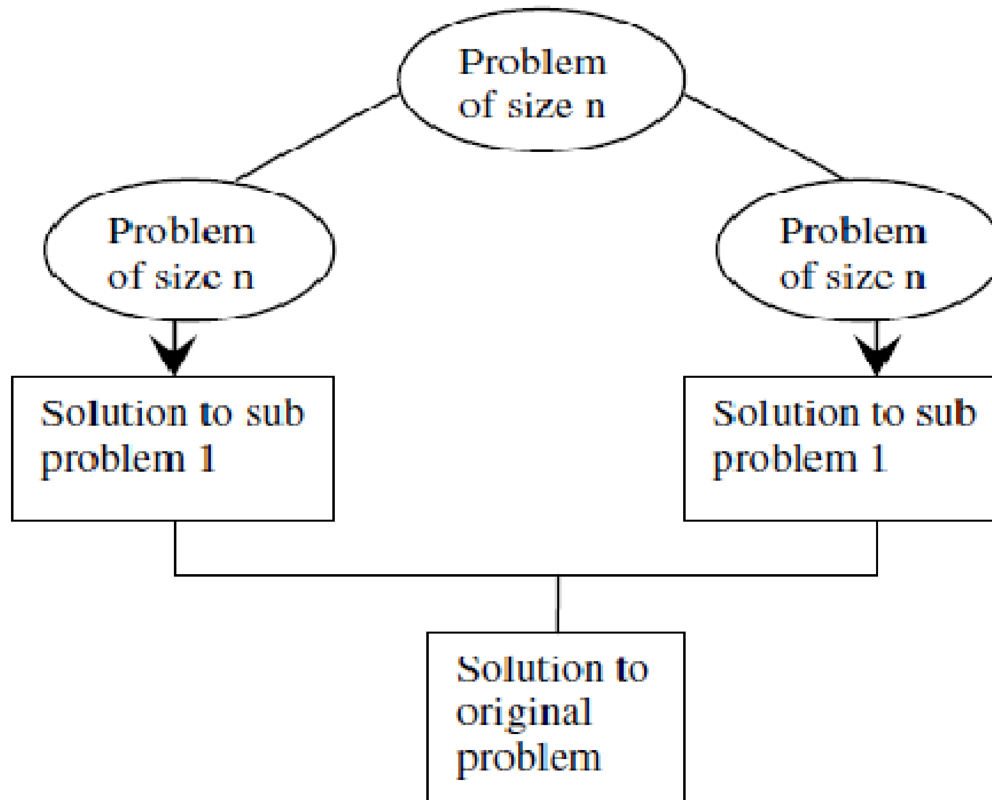
General Method

- The general method in solving a given problem is shown below:
  - An instance of a given problem is divided into a number of smaller instances of same type & equal size.
  - All the smaller instances of the problems are solved recursively.
  - The solutions of all the smaller instances are combined together to get a solution to the original problem.

General method

- Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k sub-problems
- These sub-problems must be solved, and then a method must be found to combine sub-solutions into a solution of the whole
- If the sub-problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied
- Often the sub-problems resulting from a divide-and-conquer design are of the same type as the original problem
- Similarly, smaller and smaller sub-problems of the same kind are generated until no more splitting of sub-problem is possible

Divide and Conquer Technique



- The general algorithm for divide and conquer (DAC) method is as follows:

//Purpose : Solve the problem of a given instance(P) by dividing into various smaller instances such as p1,p2,p3 ......pn.

//Input: The instances of a problem are (P)

//output : The solution S to the input instances

if small (P)

return G(P)

else

Divide P into p1, p2,.......,pk

S ← DAC(p1) + DAC(p2)+........+DAC(pk)

return S

end if

Divide and Conquer

Advantages

1. Solving difficult problems – D&C is a powerful method for solving difficult problems by breaking a problem into sub problems, solving sub problems and combining results of sub problem to get solution of original problem
2. Parallelism – D&C allows us to solve the sub problem independently, they allow execution in multi-processor machines, different sub problems can be executed on different processors
3. Memory Access – D&C algorithm makes efficient use of memory caches. Sub problems are small so all sub problems can be solved within cache, without accessing much slower main memory

Disadvantages

1. Recursion is slow – because of overhead of the repeated sub problem call
2. For some problem, D&C technique become more complicated than an iterative technique – For Example, to add n numbers in Array

Time Complexity

- The time complexity of DAC can be obtained as shown below:
  - An instance of size n can be divided into several instances say a of size n/b & the time complexity can be obtained using the recurrence relation :

    $T(n) = a\,T(n/b) + f(n)$
  - where a & b are positive constants such that $b > 1$ & f(n) is a function which is the time spent on dividing the problem into smaller instances & combining them to get a single solution.

Master Theorem

- The time complexity can be easily calculated using the following relation (Master Theorem):

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- where d is the power of n in f(n).

Ex: The recurrence relation is given by

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2) + T(n/2) + 1 & \text{otherwise} \end{cases}$$

Solve this using Backward Substitution & Master Theorem.

1) Backward Substitution :

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

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

replace n by n/2

$$= 2[2T(n/4)+1]+1$$

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

$$= 2^3T(n/2^3)+2^2+2+1$$

....

$$= 2^kT(n/2^k)+2^{k-1} + 2^{k-2}+.........+ 2^2+2+1$$

put $2^k = n$

$$= 2^kT(n/n)+2^{k-1} + 2^{k-2}+.........+ 2^2+2+1$$

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

$$= n*0 + a(r^n-1)/(r-1)$$

$$= 1(2^k-1)/(2-1) = 2^k = n$$

Time Complexity of given recurrence relation is $T(n) = \Theta(n)$

\

2) Using Master Theorem :

Here: $a = 2$

$b = 2$

$f(n) = \Theta(1)$

$d = 0$

Therefore:

$a > b^d$ i.e., $2 > 2^0$

Case 3 of master theorem holds good. Therefore:

$T(n) \in \Theta(n^{\log_b a})$

$\in \Theta(n^{\log_2 2})$

$\in \Theta(n)$

Binary search

- Binary search is one of the techniques used while searching for an item.
- But this technique is applied only if the items to be compared are in either ascending order or descending order.
- It inspects the middle element of the sorted list.
- If equal to the sought value, then the position has been found.
- Otherwise, if the key is less than the middle element, do a binary search on the first half, else on the second half.
- Two ways are used to perform binary searching are-
    1. Recursive Binary Search
    2. Iterative Binary Search

Recursive Binary Search

```
Algorithm Binary_Search(A, key, low, high)
//Purpose :  Search for an item in the list identified by A
// Input : A – list of elements
            low & high – lower bound & upper bound of the list
            key – element to be searched
//Output : position is returned if search is successful otherwise -1 is returned
if  low > high
      return -1
      mid ← (low+high)/2
      if key = A[mid]
         return mid
  else if key < A[mid]
         return Binary_Search(A, key, low, mid-1)
  else
         return Binary_Search(A, key, mid+1, high)
end
```

Iterative Binary Search

Algorithm Binary_Search(A, key, low, high)
//Purpose : Search for an item in the list identified by A
// Input : A – list of elements
        low & high – lower bound & upper bound of the list
        key – element to be searched
//Output : position is returned if search is successful otherwise -1 is returned
While low <= high do
{
mid ← (low+high)/2
if key < A[mid] then
high =mid-1
Else if key > A[mid] then
Low = mid+1
Else
Return mid;
end

Example of Binary Search

- Let us select the 14 entries as shown below-

(-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151)

- Place them in a[1 : 14]; and simulate the steps that BinSearch goes through as it searches for different values of x
- Only the variables low, high and mid need to be traced as we simulate the algorithm
- We try the following values for x: 151, -14 and 9 for two successful searches and one unsuccessful search

Tracing of Binary Search

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

(-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151)

values for x: 151, -14 and 9

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

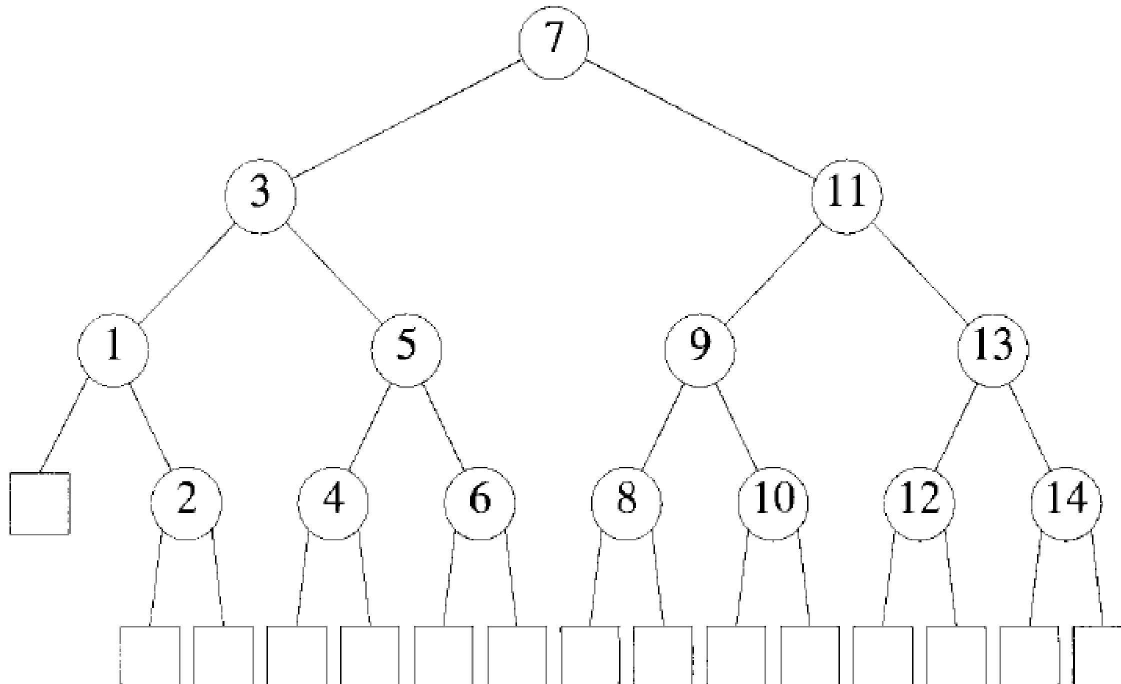
$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Table 3.2 Three examples of binary search on 14 elements

Computing Time of Binary Search

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4



3.1 Binary decision tree for binary search, $n = 14$

- Let us find the number of key comparisons in the worst case $C_{worst}(n)$.
- The worst-case inputs include all arrays that do not contain a given search key (and, in fact, some cases of successful searches as well).
- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \; C_{worst}(1) = 1.$$

- assume that n = 2^k *and solve the resulting recurrence* by backward substitutions or another method.

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

- Let us verify by substitution that indeed satisfies equation $C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ (4.2)

for any positive even number n. If n is positive and even, n = 2i where i > 0. The left-hand side of equation (4.2) for n = 2i is

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2 2i \rfloor + 1 = \lfloor \log_2 2 + \log_2 i \rfloor + 1$$
$$= (1 + \lfloor \log_2 i \rfloor) + 1 = \lfloor \log_2 i \rfloor + 2.$$

- The right hand side of the equation (4.2) for n = 2i is

$$C_{worst}(\lfloor n/2 \rfloor) + 1 = C_{worst}(\lfloor 2i/2 \rfloor) + 1 = C_{worst}(i) + 1$$
$$= (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2.$$

- Since both expressions are the same, we proved the assertion.

- What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

Finding Maximum And Minimum

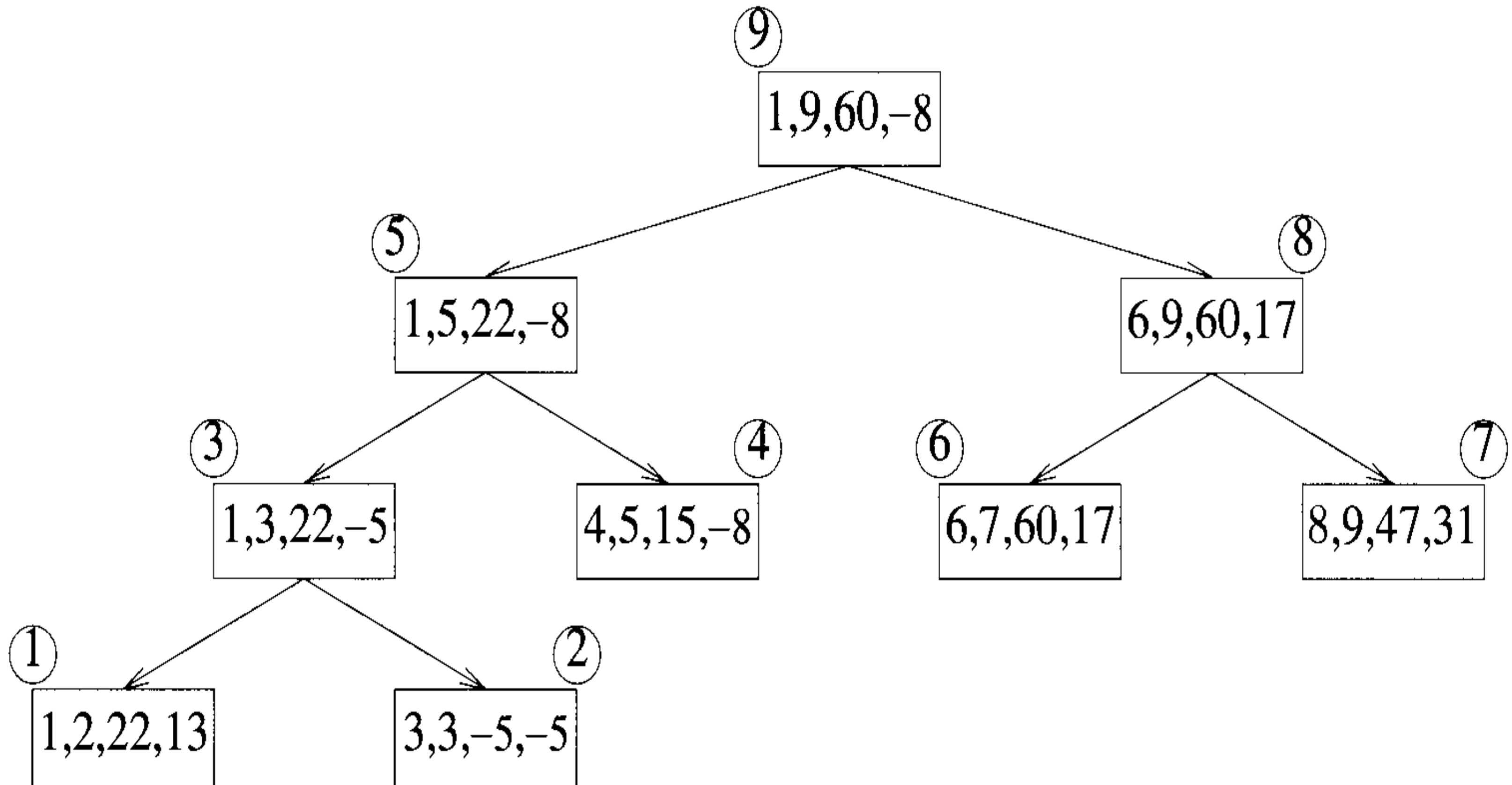
- Let us consider another simple problem that can be solved by the divide-and-conquer technique
- The problem is to find the maximum and minimum items in a set of n elements

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

- This algorithm requires 2(n-1) element comparisons in the best, worst and average cases.
- A divide-and-conquer algorithm for this problem would proceed as follows:

```
1    Algorithm MaxMin(i, j, max, min)
2    // a[1 : n] is a global array. Parameters i and j are integers,
3    // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4    // largest and smallest values in a[i : j], respectively.
5    {
6         if (i = j) then max := min := a[i]; // Small(P)
7         else if (i = j − 1) then  // Another case of Small(P)
8              {
9                   if (a[i] < a[j]) then
10                  {
11                       max := a[j]; min := a[i];
12                  }
13                  else
14                  {
15                       max := a[i]; min := a[j];
16                  }
17             }
18             else
19             {    // If P is not small, divide P into subproblems.
20                  // Find where to split the set.
21                       mid := ⌊(i + j)/2⌋;
22                  // Solve the subproblems.
23                       MaxMin(i, mid, max, min);
24                       MaxMin(mid + 1, j, max1, min1);
25                  // Combine the solutions.
26                       if (max < max1) then max := max1;
27                       if (min > min1) then min := min1;
28             }
29   }
```

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47



Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

- The number of element comparisons needed for MaxMin is represented by T(n), then the resulting Recurrence Relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

- When n is a power of 2, n = 2^k for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{1 \leq i \leq k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned}$$

- Let us see the count is when element comparisons have the same cost as comparisons between i and j. Let C(n) be this number.

$$\textbf{if } (i \geq j - 1) \ \{ \ // \ \text{Small}(P)$$

- A single comparison between i and j-1 is adequate to implement the modified if statement. Assuming $n = 2^k$ for some positive integer k, we get

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

Solving this equation, we obtain

$$\begin{aligned} C(n) &= 2C(n/2) + 3 \\ &= 4C(n/4) + 6 + 3 \\ &\vdots \\ &= 2^{k-1}C(2) + 3 \sum_0^{k-2} 2^i \\ &= 2^k + 3 * 2^{k-1} - 3 \\ &= 5n/2 - 3 \end{aligned}$$

Points about Algorithm

- If comparisons among the elements of a[] are much more costly than the comparisons of integer variables, then the divide-and-conquer technique has yielded a more efficient algorithm.
- It is sometimes necessary to work out the constants associated with the computing time bound for an algorithm.
- Both MaxMin and StraightMaxMin are $\Theta(n)$, so the use of asymptotic notation is not enough of a discriminator in this situation.

- Apply the MaxMin algorithm to on the following elements

  56, 40, 3, 68, 36, 89, 27, 8,13,55,72

Merge Sort

Introduction

- Given a sequence of elements a[1]…………a[n].
- The general idea here is to split them into two sets a[1]………a[n/2] and a[n/2+1]……….a[n].
- Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Working

- Divide the array into equal parts.
- Sort the left part of the array recursively.
- Sort the right part of the array recursively.
- Merge the left part & right part by comparing the elements & placing the lowest elements first in the resultant array.

- **Algorithm MergeSort(low, high)**

// Purpose: Sort the elements of the array between the lower bound & upper bound

// Input : low & high as lower bound & upper bound of the global array A[1:n]

//Output: A is sorted vector

```
if(low<high)
mid ← (low+high)/2
MergeSort(low, mid)
MergeSort(mid+1, high)
Merge(low, mid, high)
end if
```

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

- Consider the array of ten elements a[1:10]
- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- 310,285,179,652,351,423,861,254,450,520

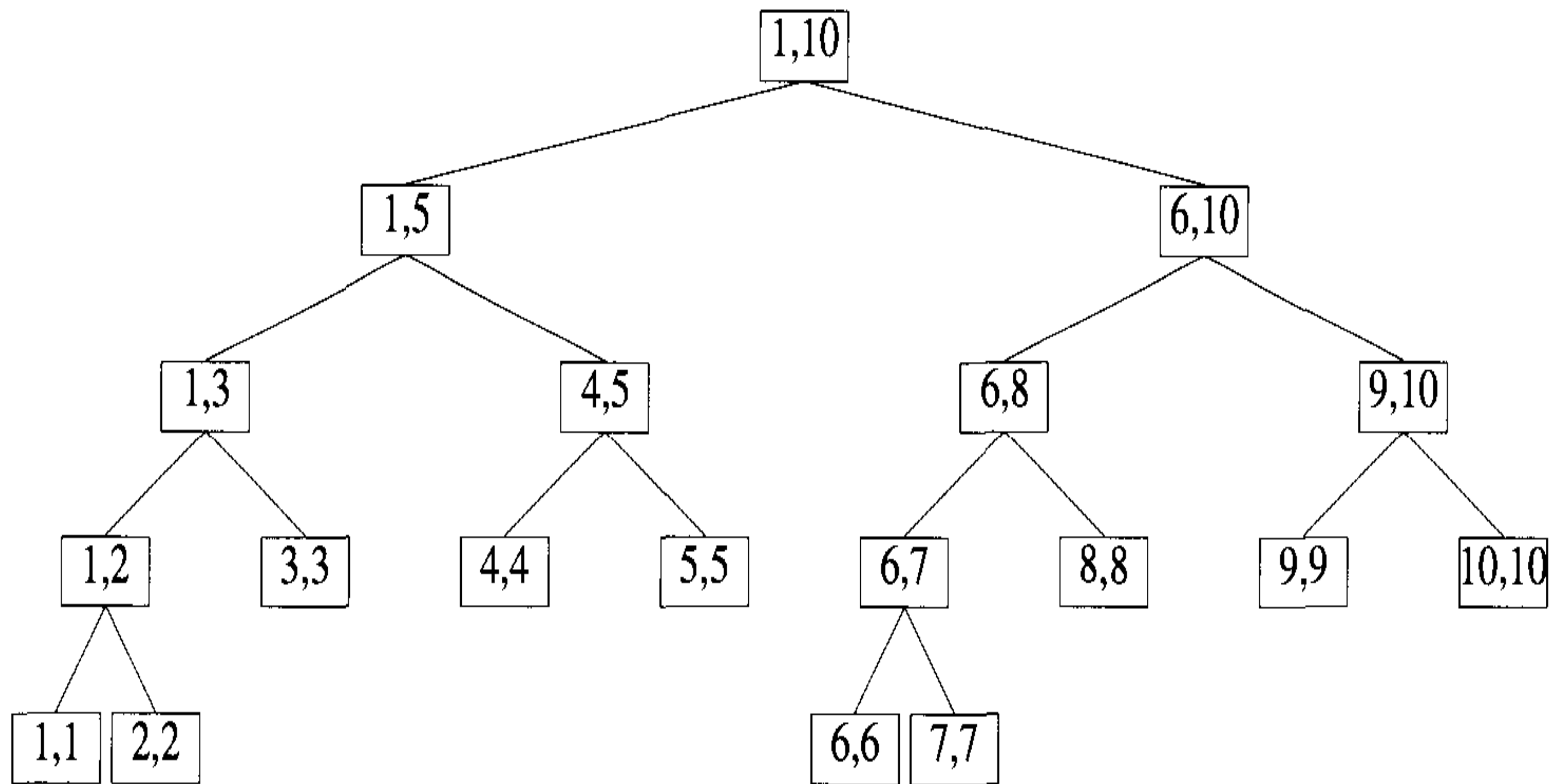


Figure 3.3 Tree of calls of MergeSort$(1, 10)$

$$(310 \mid 285 \mid 179 \mid 652,\ 351 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

$$(285,\ 310 \mid 179 \mid 652,\ 351 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

$$(179,\ 285,\ 310 \mid 652,\ 351 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

$$(179,\ 285,\ 310 \mid 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423 \mid 861 \mid 254 \mid 450,\ 520)$$

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 861 \mid 450,\ 520)$$

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 450,\ 520,\ 861)$$

$$(179,\ 254,\ 285,\ 310,\ 351,\ 423,\ 450,\ 520,\ 652,\ 861)$$

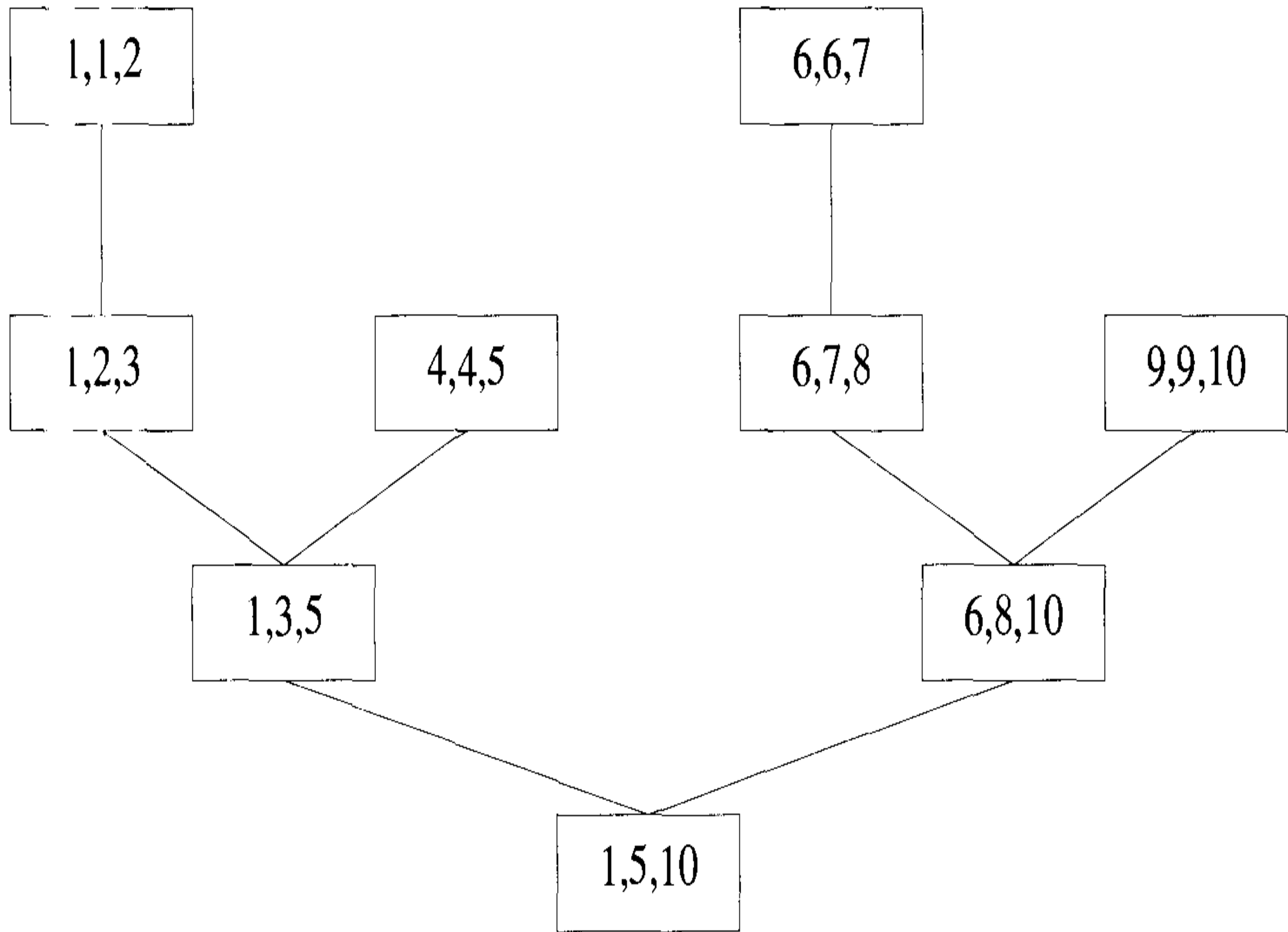


Figure 3.4 Tree of calls of Merge

Time Complexity

- If the time for merging operation is proportional to n, then the computing time for merge sort is described the recurrence relation,

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

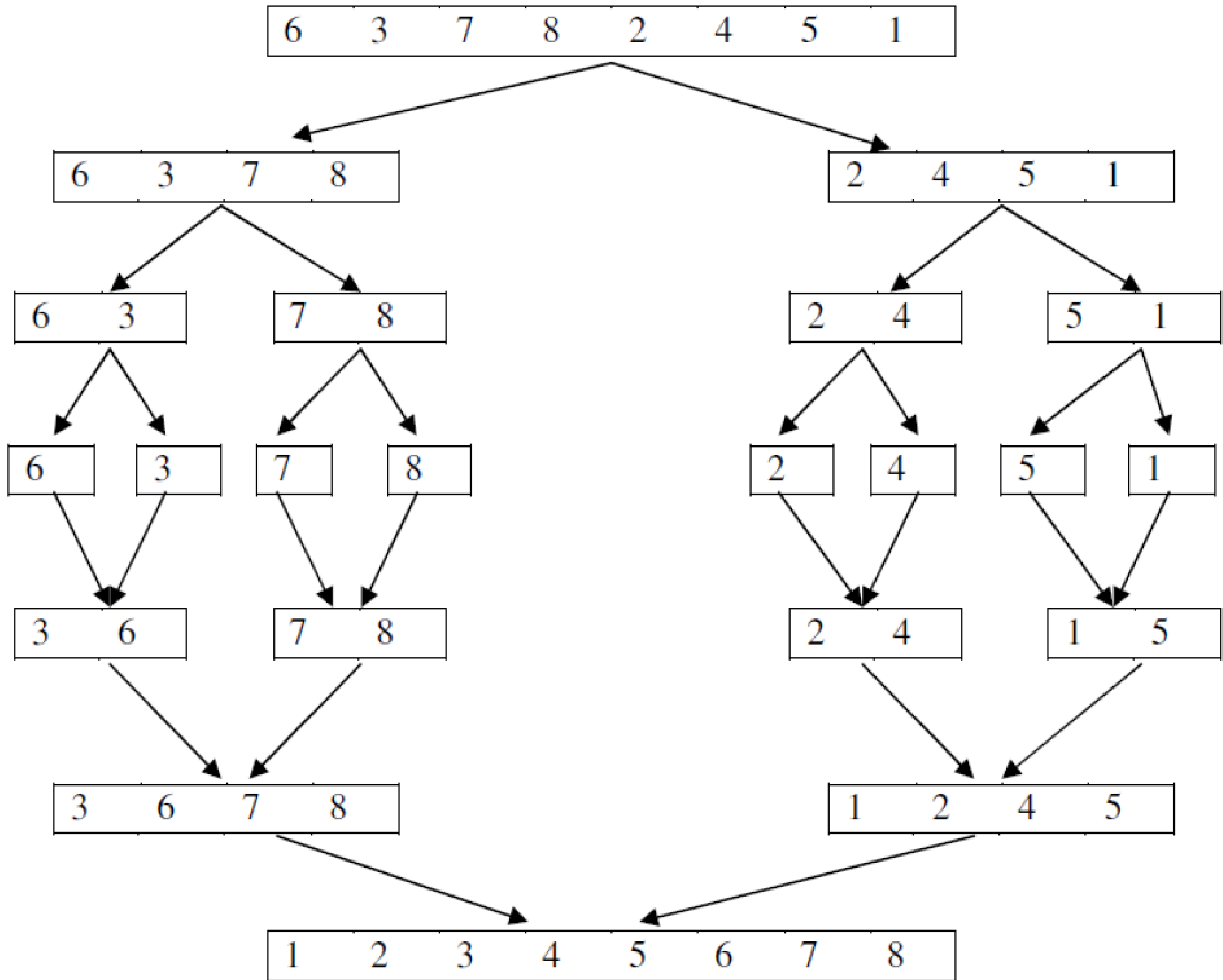
It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

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

Advantages and Limitations:

- **Advantages :**
  - Number of comparisons performed is nearly optimal.
  - Mergesort will never degrade to $O(n^2)$.
  - It can be applied to files of any size.
- **Limitations:**
  - Uses $O(n)$ additional memory.

- **Example:**
- Apply merge sort for the following list of elements : 6, 3, 7, 8, 2, 4, 5, 1



6 3 7 8 2 4 5 1
6 3 7 8
2 4 5 1
6 3
7 8
2 4
5 1
6
3
7
8
2
4
5
1
3 6
7 8
2 4
1 5
3 6 7 8
1 2 4 5
1 2 3 4 5 6 7 8

Quick Sort

(Partition Exchange Sort)

Steps

- Pick an element called pivot from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and all elements greater than pivot come after it.
- After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and sub-list of greater elements.

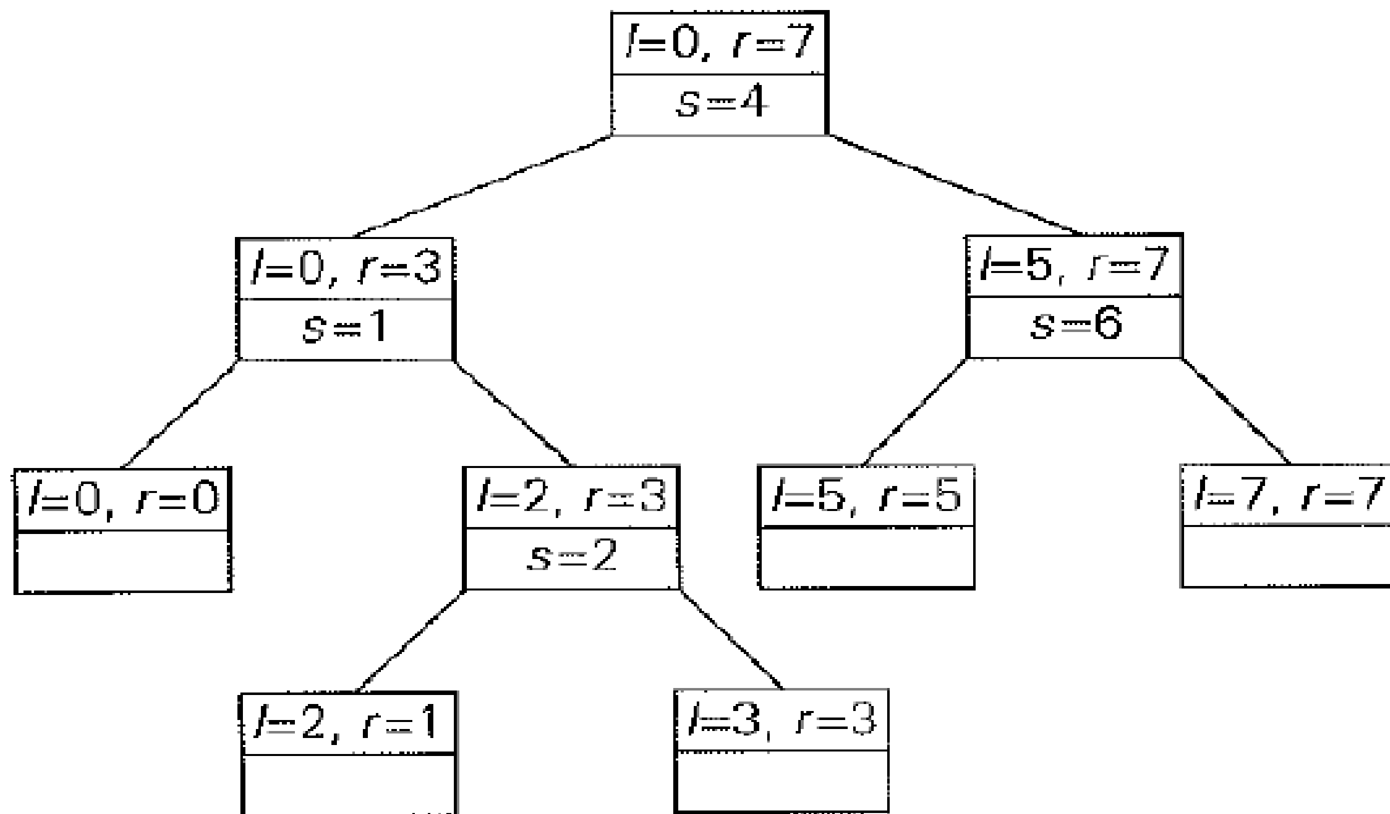
```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p],...,a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

```
1    Algorithm Partition(a, m, p)
2    // Within a[m], a[m + 1], . . . , a[p − 1] the elements are
3    // rearranged in such a manner that if initially t = a[m],
4    // then after completion a[q] = t for some q between m
5    // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6    // for q < k < p. q is returned. Set a[p] = ∞.
7    {
8        v := a[m]; i := m; j := p;
9        repeat
10       {
11           repeat
12               i := i + 1;
13           until (a[i] ≥ v);

14           repeat
15               j := j − 1;
16           until (a[j] ≤ v);

17           if (i < j) then Interchange(a, i, j);

18       } until (i ≥ j);

19       a[m] := a[j]; a[j] := v; return j;
20   }
```

```
1    Algorithm Interchange(a, i, j)
2    // Exchange a[i] with a[j].
3    {
4        p := a[i];
5        a[i] := a[j]; a[j] := p;
6    }
```

- Example :
- 0, 1, 2, 3, 4, 5, 6, 7
- 5, 3, 1, 9, 8, 2, 4, 7

0	1	2	3	4	5	6	7
5	3 (i)	1	9	8	2	4	7 (j)
5	3	1	9 (i)	8	2	4 (j)	7
5	3	1	4 (i)	8	2	9 (j)	7
5	3	1	4	8 (i)	2 (j)	9	7
5	3	1	4	2 (i)	8 (j)	9	7
5	3	1	4	2 (j)	8 (i)	9	7
2	3	1	4	**5**	8	9	7
2	3 (i)	1	4 (j)				
2	3 (i)	1 (j)	4				
2	1 (i)	3 (j)	4				
2	1 (j)	3 (i)	4				
1	**2**	3	4				
1							
		3	4 (i j)				
		3 (j)	4 (i)				
			4				
					8	9 (i)	7 (j)
					8	7 (i)	9 (j)
					8	7 (j)	9 (i)
					7	**8**	9
					7		


l=0, r=7
s=4
l=0, r=3
s=1
l=5, r=7
s=6
l=0, r=0
l=2, r=3
s=2
l=5, r=5
l=7, r=7
l=2, r=1
l=3, r=3

Analysis

First, let us obtain the worst-case value $C_w(n)$ of $C(n)$. The number of element comparisons in each call of Partition is at most $p - m + 1$. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition$(a, 1, n+1)$, is made and $r = n$; at level two at most two calls are made and $r = n - 1$; and so on. At each level of recursion, $O(r)$ element comparisons are made by Partition. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n, or $O(n^2)$. Exercise 7 examines input data on which QuickSort uses $\Omega(n^2)$ comparisons.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v has an equal probability of being the ith-smallest element, $1 \leq i \leq p-m$, in $a[m : p-1]$. Hence the two subarrays remaining to be sorted are $a[m : j]$ and $a[j+1 : p-1]$ with probability $1/(p-m), m \leq j < p$. From this we obtain the recurrence

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} [C_A(k-1)) + C_A(n-k)] \qquad (3.5)$$

The number of element comparisons required by Partition on its first call is $n+1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.5) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \qquad (3.6)$$

Replacing n by $n-1$ in (3.6) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting this from (3.6), we get

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

or

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots,$ we get

$$
\begin{aligned}
\frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= \frac{C_A(1)}{2} + 2\textstyle\sum_{3\leq k\leq n+1} \frac{1}{k} \\
&= 2\textstyle\sum_{3\leq k\leq n+1} \frac{1}{k}
\end{aligned}
\tag{3.7}
$$

Since

$$\sum_{3\leq k\leq n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

(3.7) yields

$$C_A(n) \leq 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Even though the worst-case time is $O(n^2)$, the average time is only $O(n \log n)$. Let us now look at the stack space needed by the recursion. In the worst case the maximum depth of recursion may be $n-1$. This happens, for example, when the partition element on each call to Partition is the smallest value in $a[m : p-1]$. The amount of stack space needed can be reduced to $O(\log n)$ by using an iterative version of quicksort in which the smaller of the two subarrays $a[p : j-1]$ and $a[j+1 : q]$ is always sorted first. Also, the second recursive call can be replaced by some assignment statements and a jump to the beginning of the algorithm. With these changes, QuickSort takes the form of Algorithm 3.14.

We can now verify that the maximum stack space needed is $O(\log n)$. Let $S(n)$ be the maximum stack space needed. Then it follows that

$$S(n) \leq \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

which is less than $2 \log n$.

Strassen's matrix multiplication

Strassen's matrix multiplication

- Let A and B be two n-by-n matrices. The product matrix C=AB is also an n-by-n matrix whose i, jth element is formed by taking the elements in the ith row of A and jth column of B and multiplying them to get

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

- For all i and j between 1 and n.
- The time using conventional method is $\Theta(n^3)$

- Imagine that A and B are each partitioned into four square sub-matrices, each submatrix having dimensions n/2 X n/2. Then the product of AB can be computed using above formula for the product of 2 X 2 matrices. If AB is

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

then

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

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U, and V as in (3.13). Then the C_{ij}'s are computed using the formulas in (3.14). As can be seen, P, Q, R, S, T, U, and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

$$
\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned} \tag{3.13}
$$

$$
\begin{aligned}
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned} \tag{3.14}
$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{3.15}$$

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n},\ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

- Apply the strassen's matrix multiplication to the following matrices.

$$A = \begin{pmatrix} 2 & 5 \\ 7 & 9 \end{pmatrix} \qquad B = \begin{pmatrix} 4 & 8 \\ 1 & 6 \end{pmatrix}$$

- $P = (A11+A22)(B11+B22) = (2 + 9)(4+6) = 11X10=110$
- $Q = (A21+A22)B11 = (7 + 9)\ 4= 16X4 = 64$
- $R = A11(B12-B22) = 2\ (8 – 6) = 2X\ 2 = 4$
- $S = A22(B21-B11) = 9\ (1 – 4) = 9X-3 = -27$
- $T = (A11+A12)B22 = (2+5)\ 6 = 7X6 = 42$
- $U = (A21-A11)(B11+B12) = (7-2)(4+8)= 5X12=60$
- $V = (A12-A22)(B21+B22) = (5-9)(1+6)= -4X7 = -28$

- C11 = P + S – T + V = 110 + (-27) – 42 + (-28) = 13
- C12 = R + T = 4 + 42 = 46
- C21 = Q + S = 64 + (-27) = 37
- C22 = P + R – Q + U = 110 + 4 – 64 + 60 =110

$$\underset{A}{\begin{pmatrix} 2 & 5 \\ 7 & 9 \end{pmatrix}} \times \underset{B}{\begin{pmatrix} 4 & 8 \\ 1 & 6 \end{pmatrix}} = \underset{C}{\begin{pmatrix} 13 & 46 \\ 37 & 110 \end{pmatrix}}$$

Decrease and Conquer
Topological Sorting

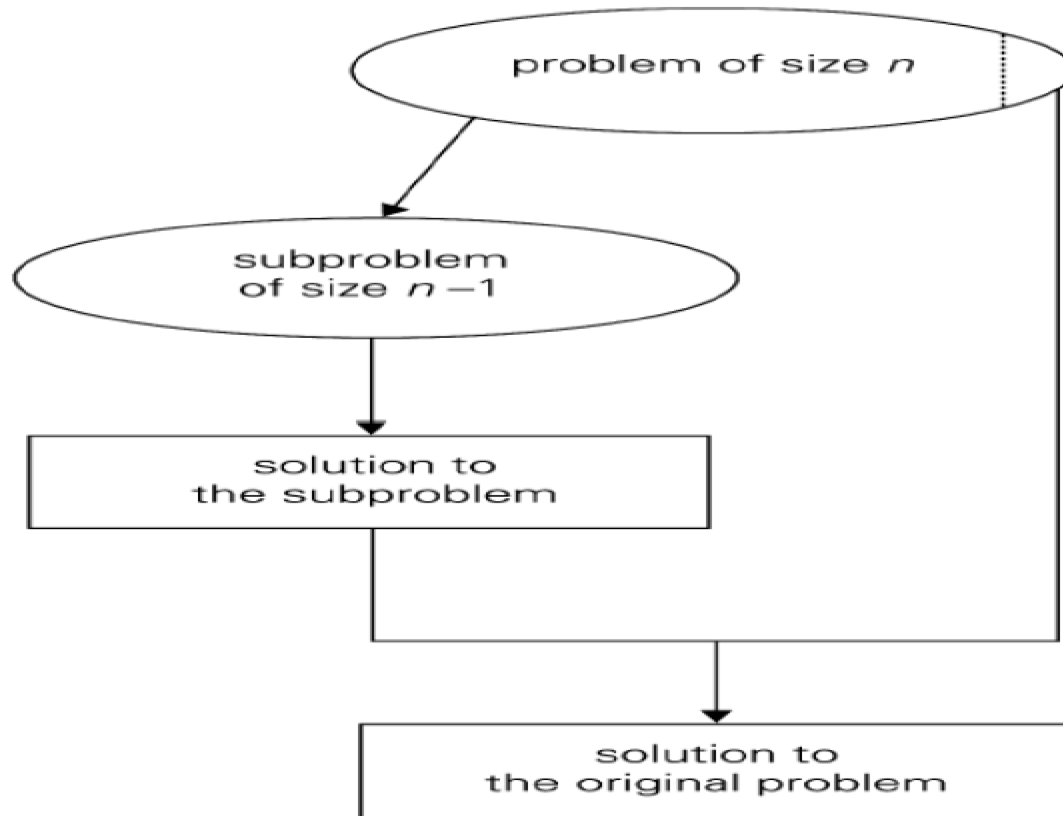
General Method

- Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

- The major variations of decrease and conquer are
- 1. Decrease by a constant :(usually by 1):
  - a. insertion sort
  - b. graph traversal algorithms (DFS and BFS)
  - c. topological sorting
  - d. algorithms for generating permutations, subsets
- 2. Decrease by a constant factor (usually by half)
  - a. binary search and bisection method
- 3. Variable size decrease
  - a. Euclid's algorithm

Decrease by Constant :

- The problem Size is usually decremented by one in each iteration of the loop.



Topological Sorting

- The topological sort of a directed acyclic graph G = (V, E) is a linear ordering of all the vertices such that for every edge (u, v) in graph G, the vertex u appears before the vertex v in the ordering.
- NOTE:
  - There is no solution for topological sorting if there is a cycle in the digraph .

- Topological sorting problem can be solved by using
  - DFS method
  - Source removal method

DFS Method

- Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited.
- On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.
- This process continues until a dead end-a vertex with no adjacent unvisited vertices-is encountered.
- At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.
- By then, all the vertices in the same connected component as the starting vertex have been visited.
- If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

- **ALGORITHM *DFS(G)***

//Implements a depth-first search traversal of a given graph

//Input: Graph G = *(V, E)*

/*Output: Graph G with its vertices marked with consecutive integers in the order they've been first encountered by the DFS traversal */

mark each vertex in *V with 0 as a mark of being "unvisited"*

Count $\longleftarrow$ *0*

for each vertex *v in V do*

if *v is marked with 0*

dfs(v)

dfs(v)

//visits recursively all the unvisited vertices connected to vertex *v hy a path*

//and numbers them in the order they are encountered via global variable *count*

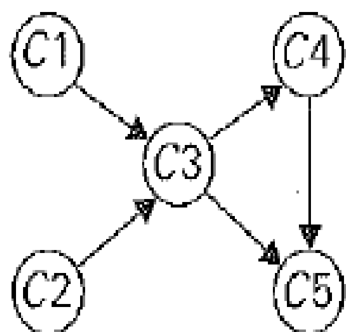
count ← *count + 1;*

mark v with count

for each vertex *w in V adjacent to v do*

if *w is marked with 0*

dfs(w)



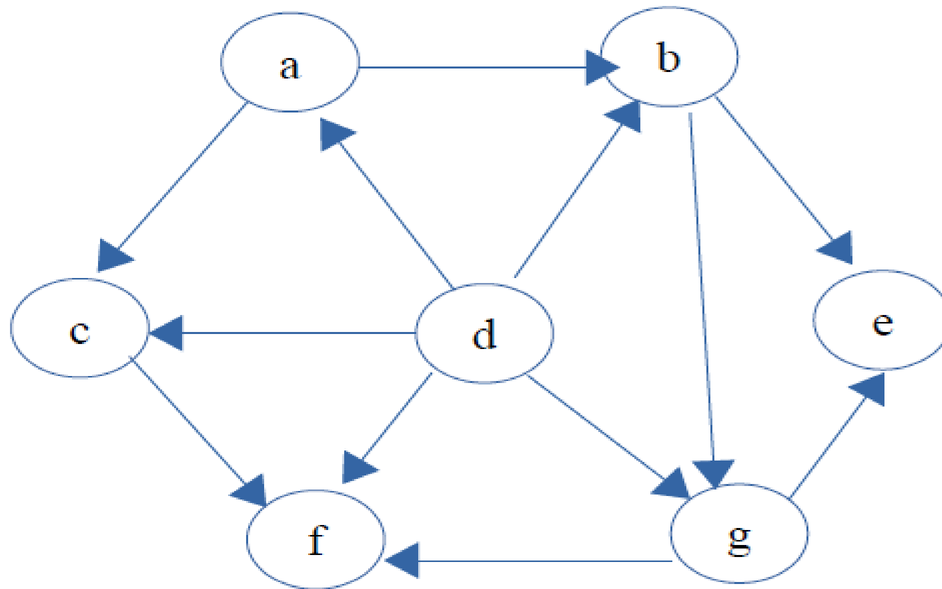
$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

The popping-off order:
$C5$, $C4$, $C3$, $C1$, $C2$
The topologically sorted list:
$C2 \quad C1 \rightarrow C3 \rightarrow C4 \rightarrow C5$

(a) (b) (c)

FIGURE 5.10 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

- Apply the DFS based algorithm to solve the topological sorting problem for the following graph :

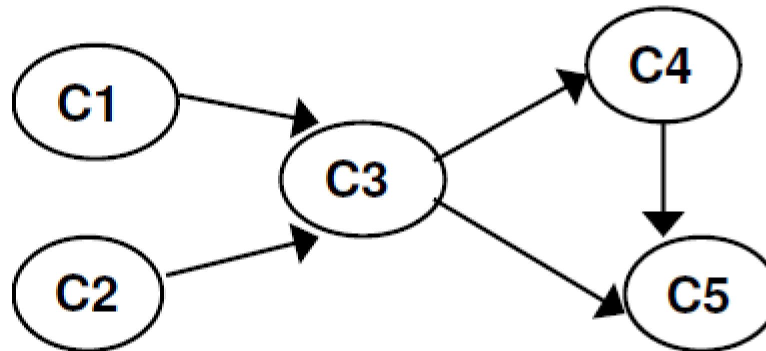


a -> b -> e -> g -> f -> c -> d

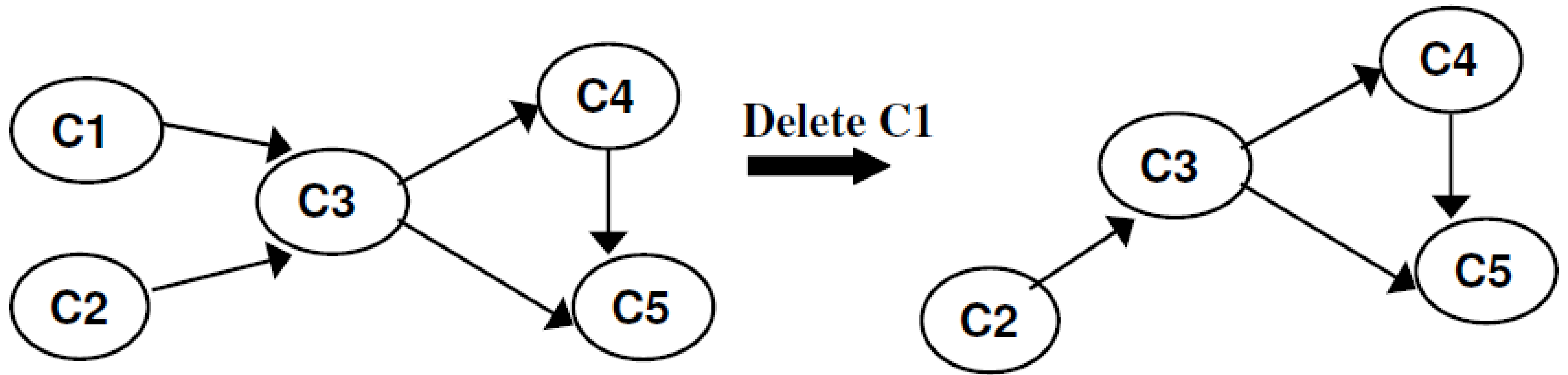
Source Removal Method

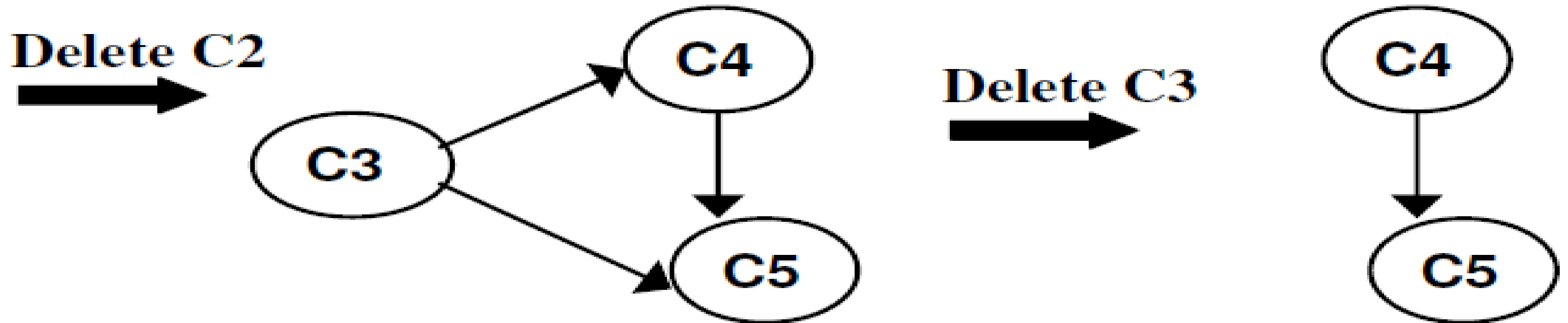
- Purely based on decrease & conquer.
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges.
- Delete it along with all the edges outgoing from it.

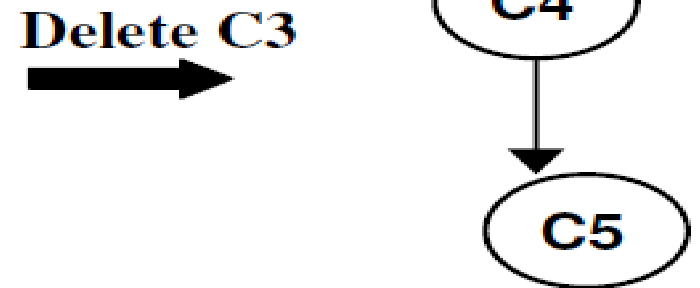
- Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:

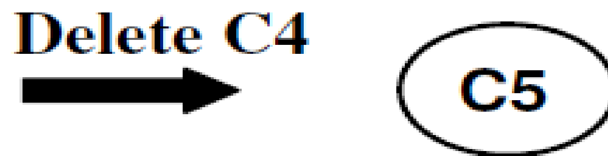


Solution:

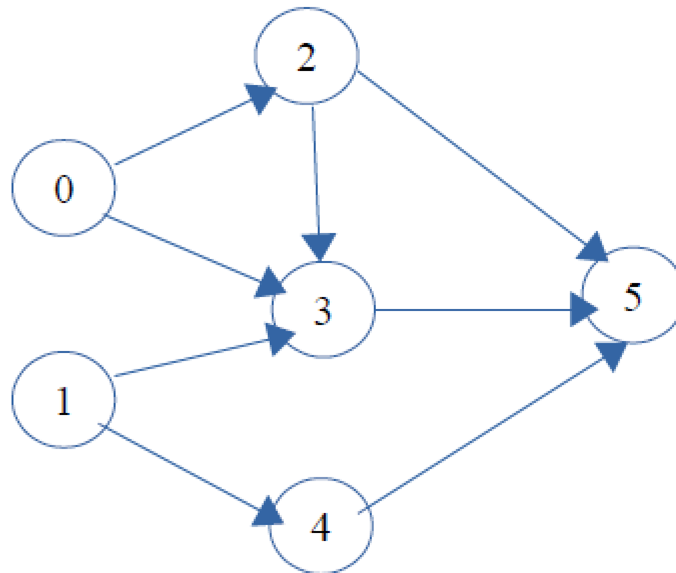








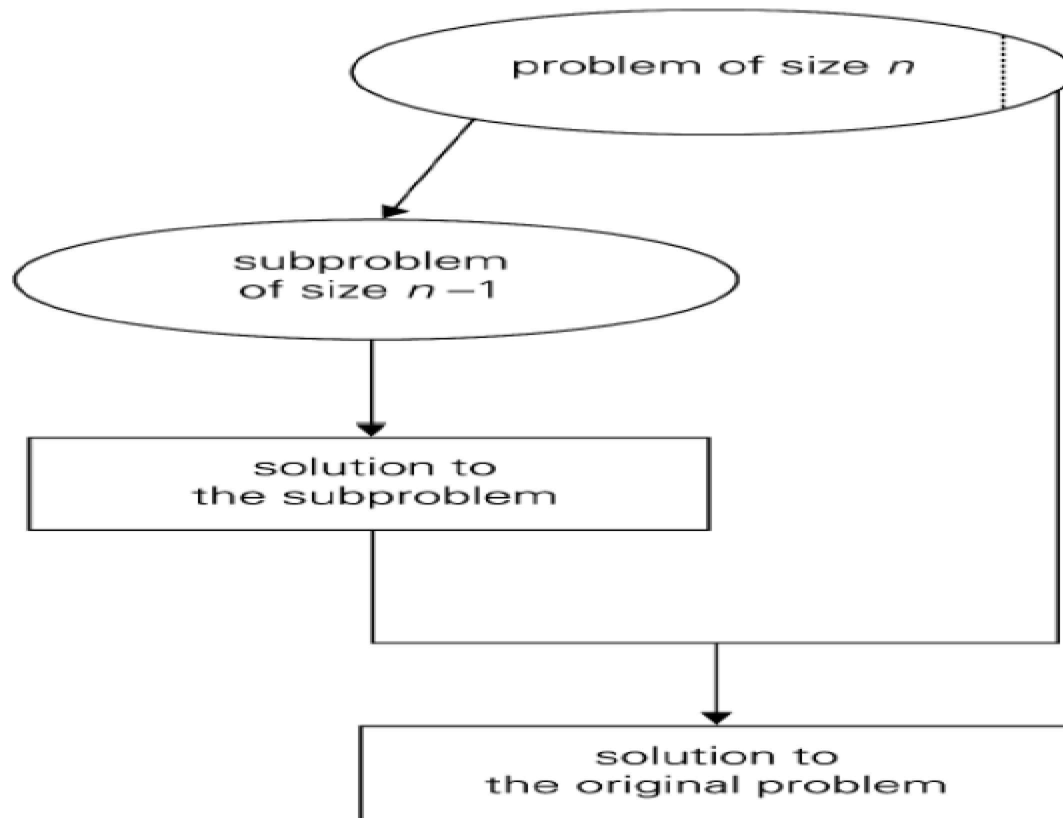
- Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:



- The major variations of decrease and conquer are
- 1. Decrease by a constant :(usually by 1):
  - a. insertion sort
  - b. graph traversal algorithms (DFS and BFS)
  - c. topological sorting
  - d. algorithms for generating permutations, subsets
- 2. Decrease by a constant factor (usually by half)
  - a. binary search and bisection method
- 3. Variable size decrease
  - a. Euclid’s algorithm

Decrease by Constant :

- The problem Size is usually decremented by one in each iteration of the loop.



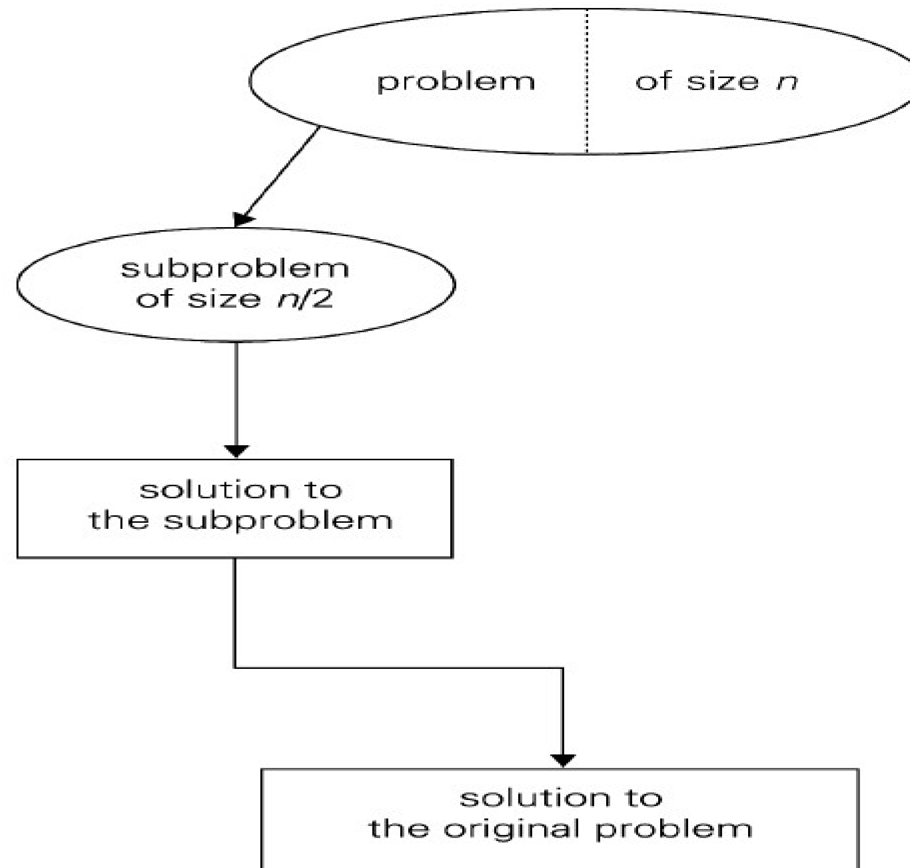
Consider, as an example, the exponentiation problem of computing a^n for positive integer exponents. The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the obvious formula: $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases} \qquad \textbf{(5.1)}$$

or "bottom up" by multiplying a by itself $n-1$ times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.)

Decrease by a constant Factor

- The problem size is reduced by same constant factor(usually by 2) on each iteration of the algorithm.



For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size will be to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances of the exponentiation problem with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \\ a & \text{if } n = 1. \end{cases} \quad (5.2)$$

If we compute a^n recursively according to formula (5.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $O(\log n)$ because, on each iteration, the size is reduced by at least one half at the expense of no more than two multiplications.

Variable – Size - Decrease

- In this, in each iteration of the loop, the size reduction pattern varies from one iteration of the algorithm to another iteration.
- Finding GCD of two numbers using Euclid's Algorithm

$$GCD(m, n) = \begin{cases} m & \text{if } n = 0 \\ GCD(n, m \bmod n) & \text{otherwise} \end{cases}$$

- Though the arguments on the right-hand side are always smaller than those on the left-hand side (at least starting with the second iteration of the algorithm), they are smaller neither by a constant nor by a constant factor.