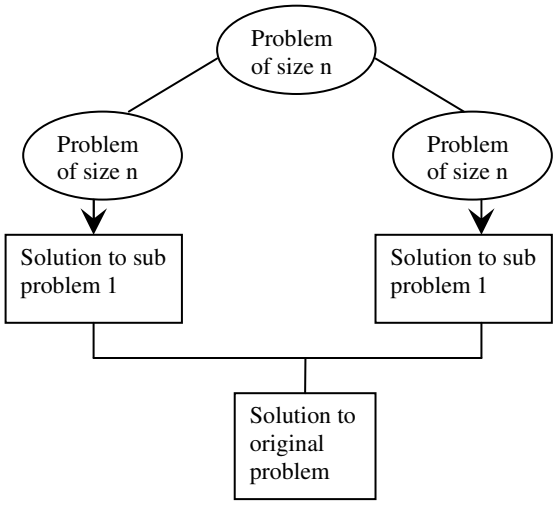
DIVIDE & CONQUER

Definition:
Divide & conquer is a general algorithm design strategy with a general plan as follows:

1. DIVIDE:
   A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. RECUR:
   Solve the sub-problem recursively.
3. CONQUER:
   If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Diagram 1 shows the general divide & conquer plan



NOTE:
The base case for the recursion is sub-problem of constant size.

Advantages of Divide & Conquer technique:

- For solving conceptually difficult problems like Tower Of Hanoi, divide & conquer is a powerful tool
- Results in efficient algorithms
- Divide & Conquer algorithms are adapted foe execution in multi-processor machines
- Results in algorithms that use memory cache efficiently.

Limitations of divide & conquer technique:

- Recursion is slow
- Very simple problem may be more complicated than an iterative approach. Example: adding n numbers etc

General divide & conquer recurrence:

An instance of size n can be divided into b instances of size n/b, with "a" of them needing to be solved. [$a \geq 1$, $b > 1$].
Assume size n is a power of b. The recurrence for the running time T(n) is as follows:

$$T(n) = aT(n/b) + f(n)$$

where:

f(n) – a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions

Therefore, the order of growth of T(n) depends on the values of the constants a & b and the order of growth of the function f(n).

Master theorem

Theorem: If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation

$$T(n) = aT(n/b) + f(n),$$

then

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

Example:

Let $T(n) = 2T(n/2) + 1$, solve using master theorem.

Solution:

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:

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

Merge Sort

Definition:
Merge sort is a sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final sorted sequence.

Features:

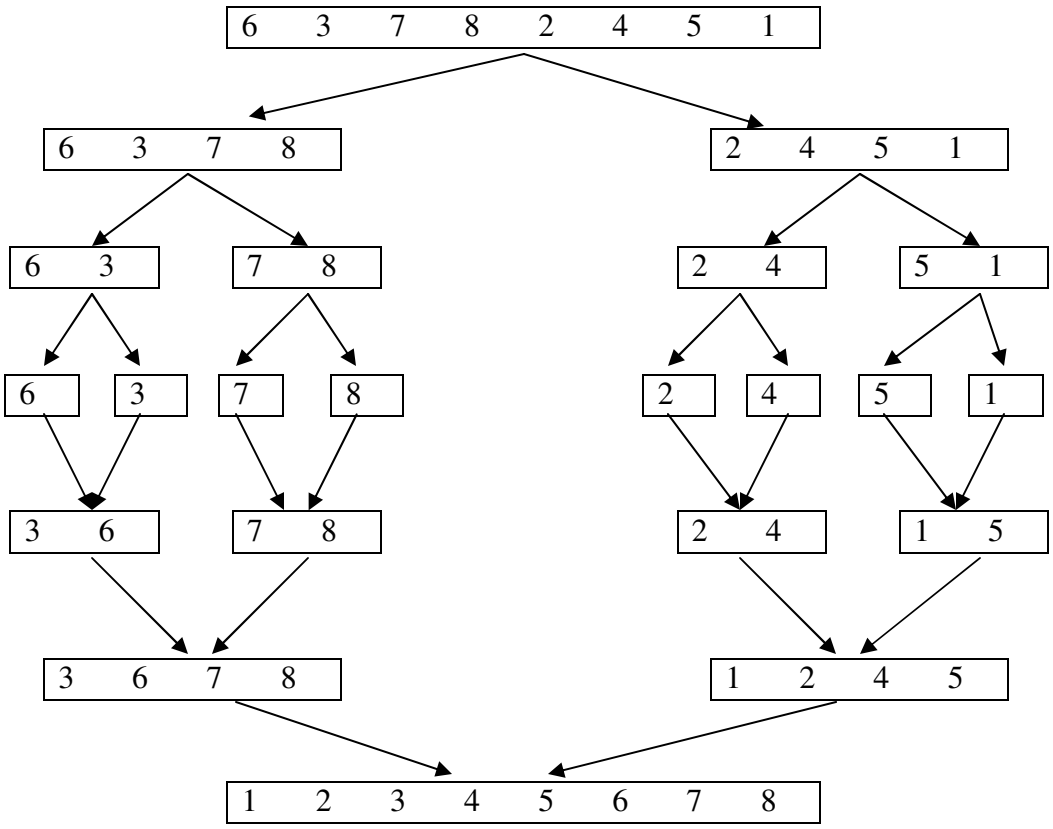
- Is a comparison based algorithm
- Is a stable algorithm
- Is a perfect example of divide & conquer algorithm design strategy
- It was invented by John Von Neumann

Algorithm:

```
ALGORITHM Mergesort ( A[0… n-1] )
//sorts array A by recursive mergesort
//i/p: array A
//o/p: sorted array A in ascending order

if n > 1
        copy A[0… (n/2 -1)] to B[0… (n/2 -1)]
        copy A[n/2… n -1)] to C[0… (n/2 -1)]
        Mergesort ( B[0… (n/2 -1)] )
        Mergesort ( C[0… (n/2 -1)] )
        Merge ( B, C, A )

ALGORITHM Merge ( B[0… p-1], C[0… q-1], A[0… p+q-1] )
//merges two sorted arrays into one sorted array
//i/p: arrays B, C, both sorted
//o/p: Sorted array A of elements from B & C

i ← 0
j ← 0
k ← 0
while i < p and j < q do
        if B[i] ≤ C[j]
                A[k] ← B[i]
                i ← i + 1
        else
                A[k] ← C[j]
                j ← j + 1
        k ← k + 1
if i == p
        copy C [ j… q-1 ] to A [ k… (p+q-1) ]
else
        copy B [ i… p-1 ] to A [ k… (p+q-1) ]
```

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

Solution:



Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists:
  Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element.
- Let C(n) denotes the number of times basic operation is executed. Then
  $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$
  $C(1) = 0$
  where, $C_{merge}(n)$ is the number of key comparison made during the merging stage.
  In the worst case:
  $C_{merge}(n) = 2\ C_{merge}(n/2) + n-1$ for $n > 1$
  $C_{merge}(1) = 0$
- Solving the recurrence equation using master theorem:
  $C(n) = 2C(n/2) + n-1$ for $n > 1$
  $C(1) = 0$

Here a = 2
b = 2
f(n) = n; d = 1
Therefore $2 = 2^1$, case 2 holds
$C(n) = \Theta(n^d \log n)$
$= \Theta(n^1 \log n)$
$= \Theta(n \log n)$

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.

Quick Sort

(Also known as "partition-exchange sort")

Definition:
Quick sort is a well –known sorting algorithm, based on divide & conquer approach. The steps are:

1. Pick an element called pivot from the list
2. 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
3. Recursively sort the sub-list of lesser elements and sub-list of greater elements.

Features:

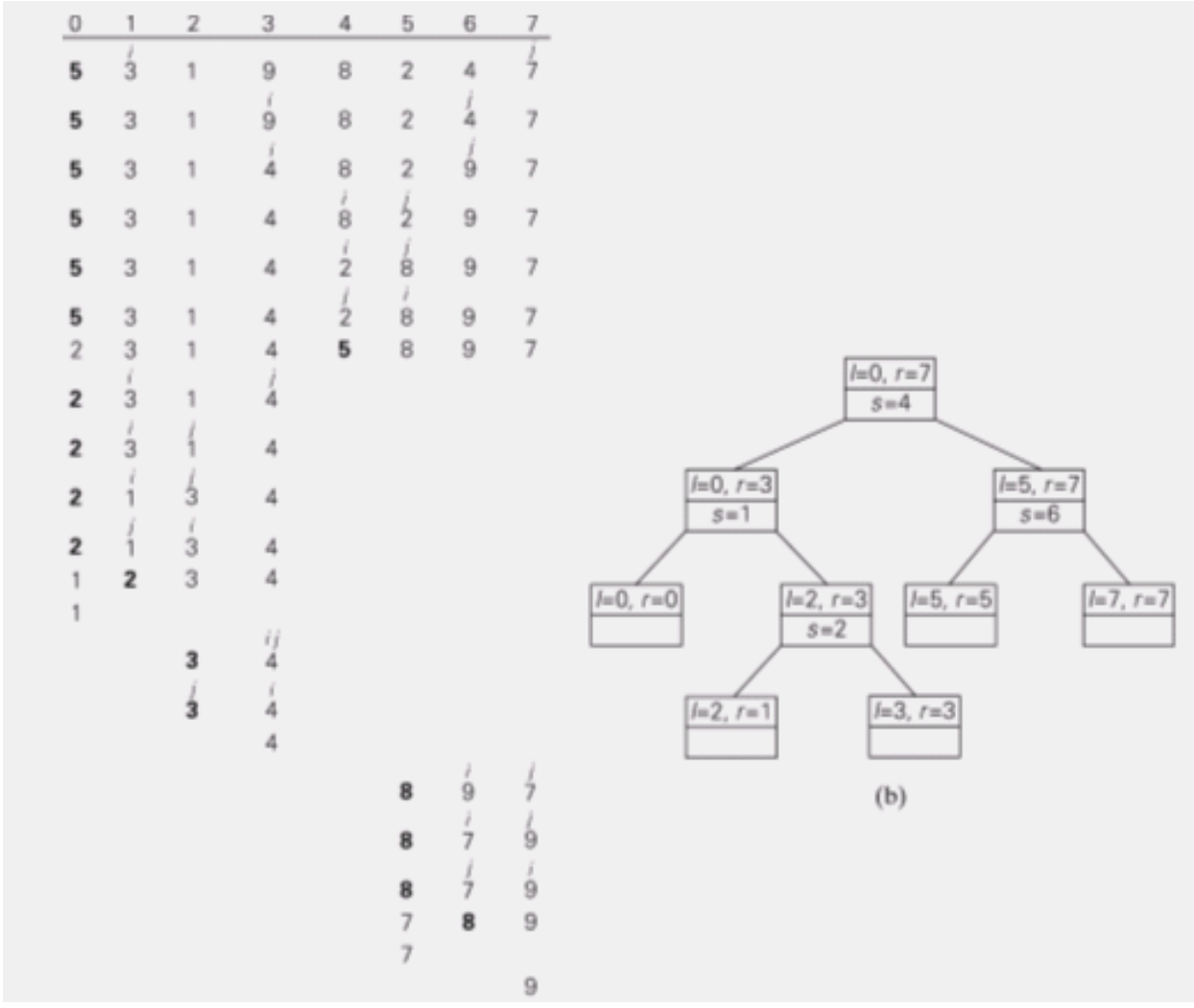
- Developed by C.A.R. Hoare
- Efficient algorithm
- NOT stable sort
- Significantly faster in practice, than other algorithms

Algorithm

ALGORITHM Quicksort (A[l …r])
//sorts by quick sort
//i/p: A sub-array A[l..r] of A[0..n-1],defined by its left and right indices l and r
//o/p: The sub-array A[l..r], sorted in ascending order
if l < r
s ← Partition (A[l..r]) // s is a split position
Quicksort(A[l..s-1])
Quicksort(A[s+1..r]

ALGORITHM Partition (A[l ..r])
//Partitions a sub-array by using its first element as a pivot
//i/p: A sub-array A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r)
//o/p: A partition of A[l..r], with the split position returned as this function's value
p ←A[l]
i ←l
j ← r + 1;
Repeat
repeat i ← i + 1 until A[i] >=p //left-right scan
repeat j ←j – 1 until A[j] < p //right-left scan
if (i < j) //need to continue with the scan
swap(A[i], a[j])
until i >= j //no need to scan
swap(A[l], A[j])
return j

Example: Sort by quick sort the following list: 5, 3, 1, 9, 8, 2, 4, 7, show recursion tree.



Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists:
  Best case: when partition happens in the middle of the array each time.
  Worst case: When input is already sorted. During key comparison, one half is empty, while remaining n-1 elements are on the other partition.
- Let C(n) denotes the number of times basic operation is executed in worst case: Then
  $C(n) = C(n-1) + (n+1)$ for $n > 1$ (2 sub-problems of size 0 and n-1 respectively)
  $C(1) = 1$

  Best case:
  $C(n) = 2C(n/2) + \Theta(n)$ (2 sub-problems of size n/2 each)

- Solving the recurrence equation using backward substitution/ master theorem, we have:
  $C(n) = C(n-1) + (n+1)$ for $n > 1$; $C(1) = 1$
  $\mathbf{C(n) = \Theta(n^2)}$

  $C(n) = 2C(n/2) + \Theta(n)$.
  $= \Theta(n^1 \log n)$
  $\mathbf{= \Theta(n \log n)}$

NOTE:
The quick sort efficiency in average case is $\Theta(n \log n)$ on random input.

Binary Search

Description:

Binary tree is a dichotomic divide and conquer search algorithm. Ti 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.

Algorithm:

Algorithm can be implemented as recursive or non-recursive algorithm.

ALGORITHM BinSrch (A[0 … n-1], key)

```
//implements non-recursive binary search
//i/p: Array A in ascending order, key k
//o/p: Returns position of the key matched else -1

l ← 0
r ← n-1

while l ≤ r do
        m ← ( l + r) / 2
        if key = = A[m]
                return m
        else
                if key < A[m]
                        r ← m-1
                else
                        l ← m+1
return -1
```

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- **Depend on**
  Best – key matched with mid element
  Worst – key not found or key sometimes in the list
- Let C(n) denotes the number of times basic operation is executed. Then
  $C_{worst}(n)$ = Worst case efficiency. Since after each comparison the algorithm divides the problem into half the size, we have
  $C_{worst}(n) = C_{worst}(n/2) + 1$ for $n > 1$
  $C(1) = 1$
- Solving the recurrence equation using master theorem, to give the number of times the search key is compared with an element in the array, we have:
  $C(n) = C(n/2) + 1$
  $a = 1$

b = 2
$f(n) = n^0$; d = 0
case 2 holds:

$C(n) = \Theta(n^d \log n)$
$= \Theta(n^0 \log n)$
$= \Theta(\log n)$

Applications of binary search:

- Number guessing game
- Word lists/search dictionary etc

Advantages:

- Efficient on very big list
- Can be implemented iteratively/recursively

Limitations:

- Interacts poorly with the memory hierarchy
- Requires given list to be sorted
- Due to random access of list element, needs arrays instead of linked list.

Binary tree traversals
and related properties binary tree

Binary Tree:

Definition of binary tree itself divides the tree into two sub-trees. Many problems about binary trees can be solved by applying the divide and conquer technique

Example 1:

Write an algorithm to find the height of a given binary tree.
Solution:

ALGORITHM BinTreeHeight (T)
//computes recursively the height of a binary tree
//i/p: A binary tree T
//o/p: Height of T

```
if T = = Ø
        return -1
else
        return ( max { BinTreeHeight ( T_L ), BinTreeHeight ( T_R ) } + 1 )
```

Analysis:

- **Input size:** number of nodes
- **Basic operation:**
  - Number of comparison made to compute the maximum of two numbers
  - Number of additions made
- No best, worst, average case
- Let n(T) be the number of nodes in the given binary tree. Since comparison & additions takes place equally; considering only number of additions, we have the recurrence:

  $A(n(T)) = A(n(T_L)), A(n(T_R)) + 1$ for $n(T) > 0$

  Solving the recurrence we have $A(n) = n$

Multiplication of large integers
using Divide & Conquer technique:

Description:
Large integers with over 100 decimal digits long are too long to fit in a single word of a modern computer, hence require special algorithms to treat them.

ALGORITHM using Divide & Conquer method:
Let A & B be two n-digits integers where n is a positive even number.
Let

a1 - first half of a's digits
a0 – second half of a's digits

Similarly

b1 - first half of a's digits
b0 – second half of a's digits

$a = a1a0$
$a = a1\ 10^{n/2} + a0$
similarly $b = b1\ 10^{n/2} + b0$
Therfore:

$$\begin{aligned} c &= a * b \\ &= (a1\ 10^{n/2} + a0\ ) * (b1\ 10^{n/2} + b0) \\ &= (a1 * b1)\ 10^{n} + (a1* b0 + a0 * b1)\ 10^{n/2} + (a0 * b0) \\ &= C2\ 10^{n} + C1\ 10^{n/2} + C0 \end{aligned}$$

Where:
C2 = a1 * b1 (product of the first halves)
C0 = a0 * b0 (product of the second halves)
C1 = (a1* b0) + (a0 * b1) – (C2 + C0) is the product of the sum of the a's halves and the sum of the b's halves minus the sum of C2 & C0

Analysis:

- **Input size: n -** number of digits
- **Basic operation:**
  - Multiplication
  - Addition
  - subtraction
- No best, worst, average case
- Let M(n) be the number of multiplication's recurrence:

  $M(n) = 3\,M(n/2)$ for $n > 1$

  $M(1) = 1$

  Since $n = 2^k$

  $M(2^k) = 3\,M(2^{k-1})$

  $= 3\,[3\,M(2^{k-2})]$

  $= 3^2\,M(2^{k-2})$

  …

  $= 3^i\,M(2^{k-i})$

  When i=k

  $= 3^k\,M(2^{k-k})$

  $= 3^k$

  $K = \log_2 n$, we have

  $M(n) = 3^{\log_2 n}$

  $= n^{\log_2 3}$

  $\approx n^{1.5}$

Strassen's matrix multiplication
using Divide & Conquer technique:

Description :

Strassen's algorithm is used for matrix multiplication. It is asymptotically faster than the standard matrix multiplication algorithm

ALGORITHM using Divide & Conquer method:

Let A & B be two square matrices.

C= A * B

We have,

$$\begin{vmatrix} C00 & C01 \\ C10 & C11 \end{vmatrix} = \begin{vmatrix} A00 & A01 \\ A10 & A11 \end{vmatrix} * \begin{vmatrix} B00 & B01 \\ B10 & B11 \end{vmatrix}$$

=

$$\begin{vmatrix} M1 + M4 - M5 + M7 & M8 + M5 \\ M2 + M4 & M1 + M3 - M2 + M6 \end{vmatrix}$$

Where:
M1 = (A00 + A11) * (B00 + B11)
M2 = (A10 + A11) * B00
M3 = A00 * (B01 – B11)
M4 = A11 * (B10 – B00)
M5 = (A00 + A01) * B11
M6 = (A10 – A00) * (B00 + B01)
M7 = (A01 – A11) * (B10 + B11)

Analysis:

- **Input size: n –** order of square matrix.
- **Basic operation:**
  - Multiplication (7)
  - Addition (18)
  - Subtraction (4)
- No best, worst, average case
- Let M(n) be the number of multiplication's made by the algorithm, Therefore we have:
  M (n) = 7 M(n/2) for n > 1
  M (1) = 1
  Assume $n = 2^k$

$$M(2^k) = 7\,M(2^{k-1})$$
$$= 7\,[7\,M(2^{k-2})]$$
$$= 7^2\,M(2^{k-2})$$
$$\cdots$$
$$= 7^i\,M(2^{k-i})$$

When i=k

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

$K = \log_2 n$, we have

$$M(n) = 7^{\log_2 n}$$
$$= n^{\log_2 7}$$
$$\approx n^{2.807}$$