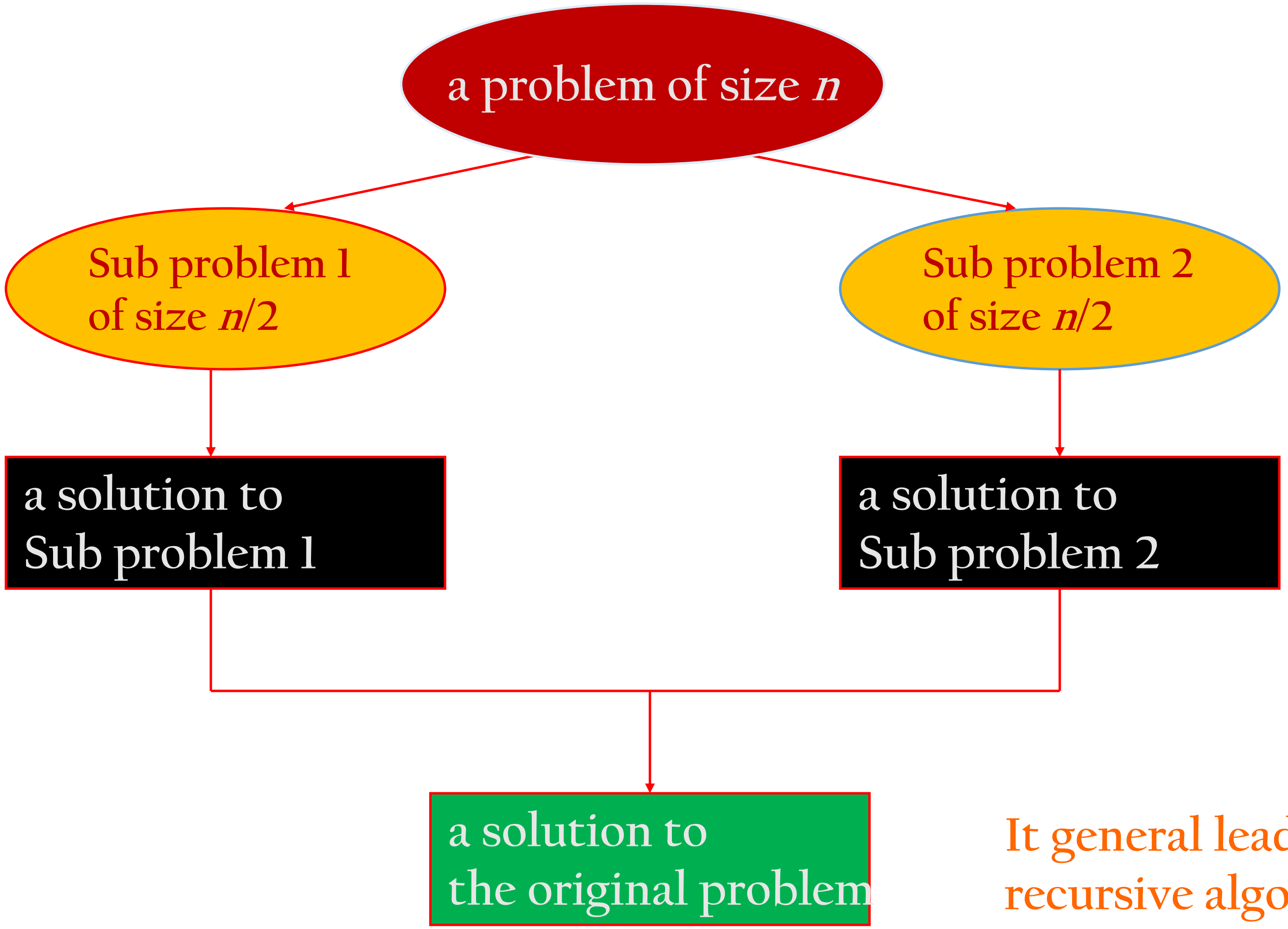
Chapter - 2
Divide and Conquer Algorithms

Essence of Divide and Conquer

- In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.
- When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide-and-Conquer Technique



It general leads to a recursive algorithm!

divide-and-conquer steps

- Broadly, we can understand *divide-and-conquer* approach in a three-step process.

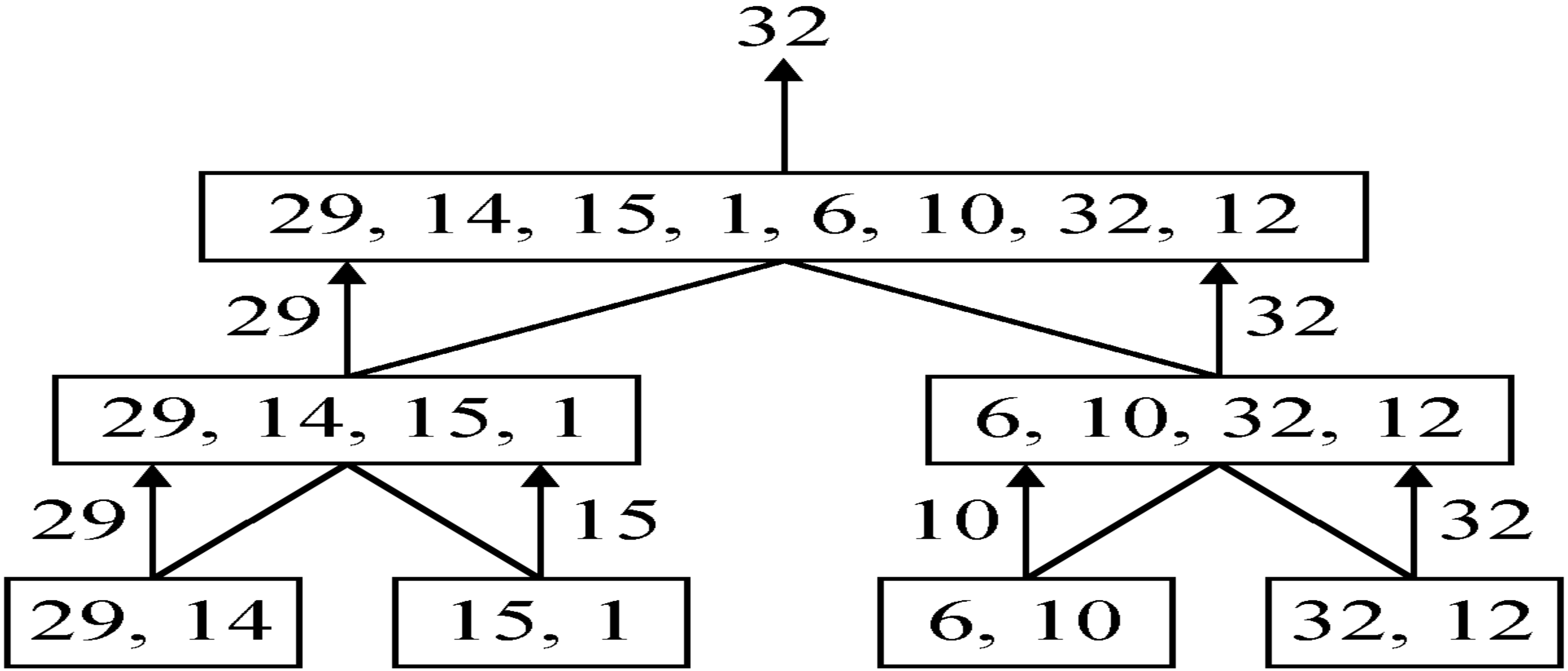
1) *Divide:* a problem to be solved is broken into a number of sub problems. Normally, the sub problems are similar to the original

2) *Conquer:* the sub problems are then solved independently, usually recursively.
3) *Combine:* finally, the solutions to the sub problems are combined to provide the answer to the original problem. The solutions to get a solution to the sub problems And finally a solution to the original problem.

- *Divide and Conquer* algorithms are normally recursive.

A simple example

- finding the maximum of a set S of n numbers



A general divide-and-conquer algorithm

Step 1: If the problem size is small, solve this problem directly; otherwise, *split* the original problem into 2 sub-problems with equal sizes.

Step 2: Recursively solve these 2 sub-problems by applying this algorithm.

Step 3: Merge the solutions of the 2 sub-problems into a solution of the original problem.

```
DAC(problem P)
{
        if Small(P) return S(P);
        else {

          divide P into smaller instances P1, P2, …, Pk, k≥1;

          Apply DAC to each of these subproblems;

          return Combine(DAC(P1), DAC(P2),…,DAC (Pk));
        }
}
```

Application of **divide-and-conquer**

- The following computer algorithms are based on **divide-and-conquer** programming approach −

  1. Finding Max and Min
  2. binary Search
  3. Merge Sort
  4. Quick Sort
  5. Strassen's Matrix Multiplication

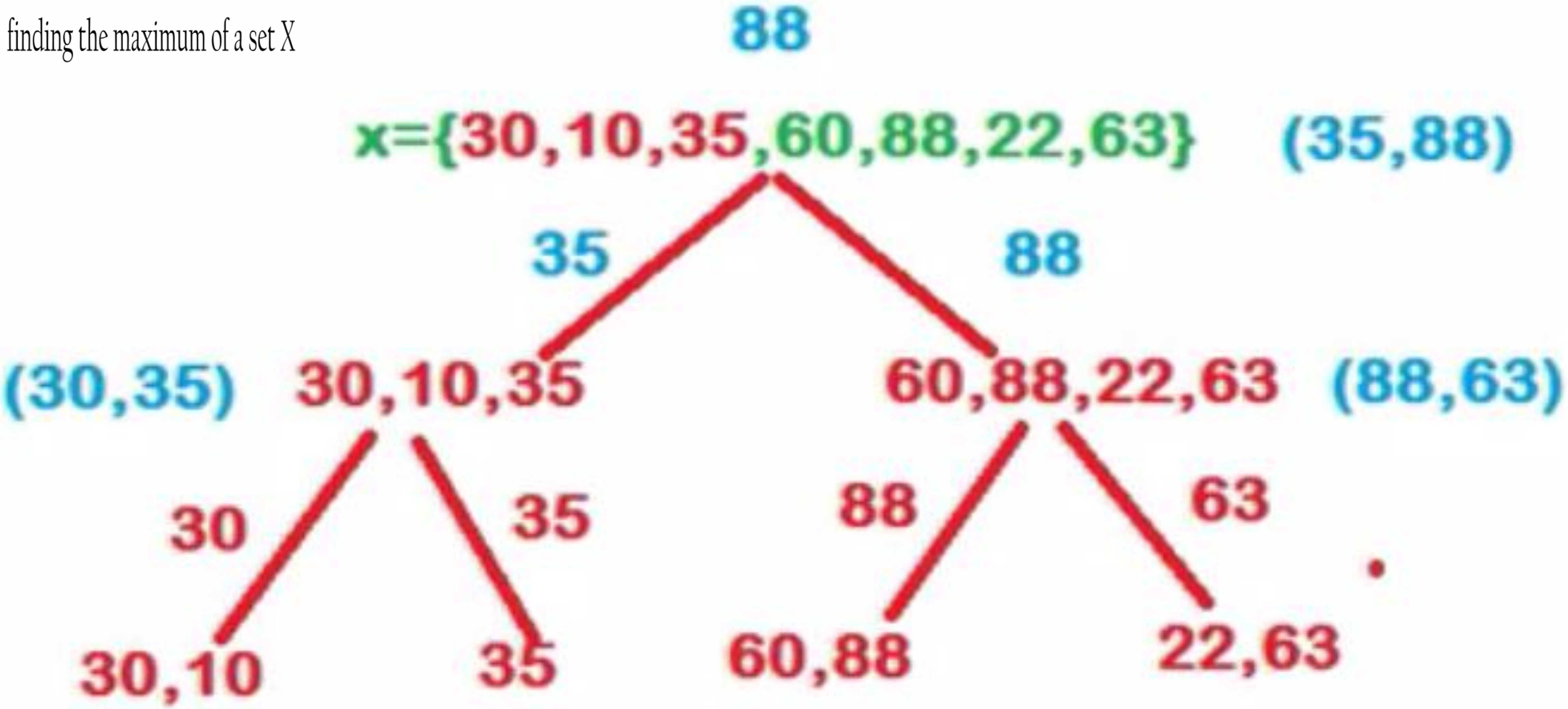
Note: There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Cont….

- *sorting*: ordering a list of values
- *searching*: finding the position of a value within a list
- Algorithm analysis should begin with a clear statement of the task to be performed.
- This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.
- Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:
- *time complexity*: how the number of steps required depends on the size of the input
- *space complexity*: how the amount of extra memory or storage required depends on the size of the input

1. Finding Max and Min

finding the maximum of a set X



Algorithm for maximum and minimum using divide-and-conquer

```
MaxMin(i, j, max, min)
// a[1:n] is a global array. Parameters i and j are integers,   // 1≤i≤j≤n. The effect is to set max and min to the largest and  // smallest values in a[i:j].
{
    if (i=j) then max := min := a[i]; //Small(P)
    else if (i=j-1) then // Another case of Small(P)
        {
            if (a[i] < a[j]) then max := a[j]; min := a[i];
            else max := a[i]; min := a[j];
        }
    else
    {
        // if P is not small, divide P into sub-problems.
        // Find where to split the set.
        mid := ( i + j )/2;
        // Solve the sub-problems.
        MaxMin( i, mid, max, min );
        MaxMin( mid+1, j, max1, min1 );
        // Combine the solutions.
        if (max < max1) then max := max1;
        if (min > min1) then min := min1;
    }}
```

Complexity:

Now what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

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

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

2. *binary search*

- divide sequence into two halves by comparing search key to midpoint
- recursively search in one of the two halves
- combine step is empty

Algorithm binary-search

Input: A sorted sequence of n elements stored in an array.

Output: The position of x (to be searched).

Step 1: If only one element remains in the array, solve it directly.

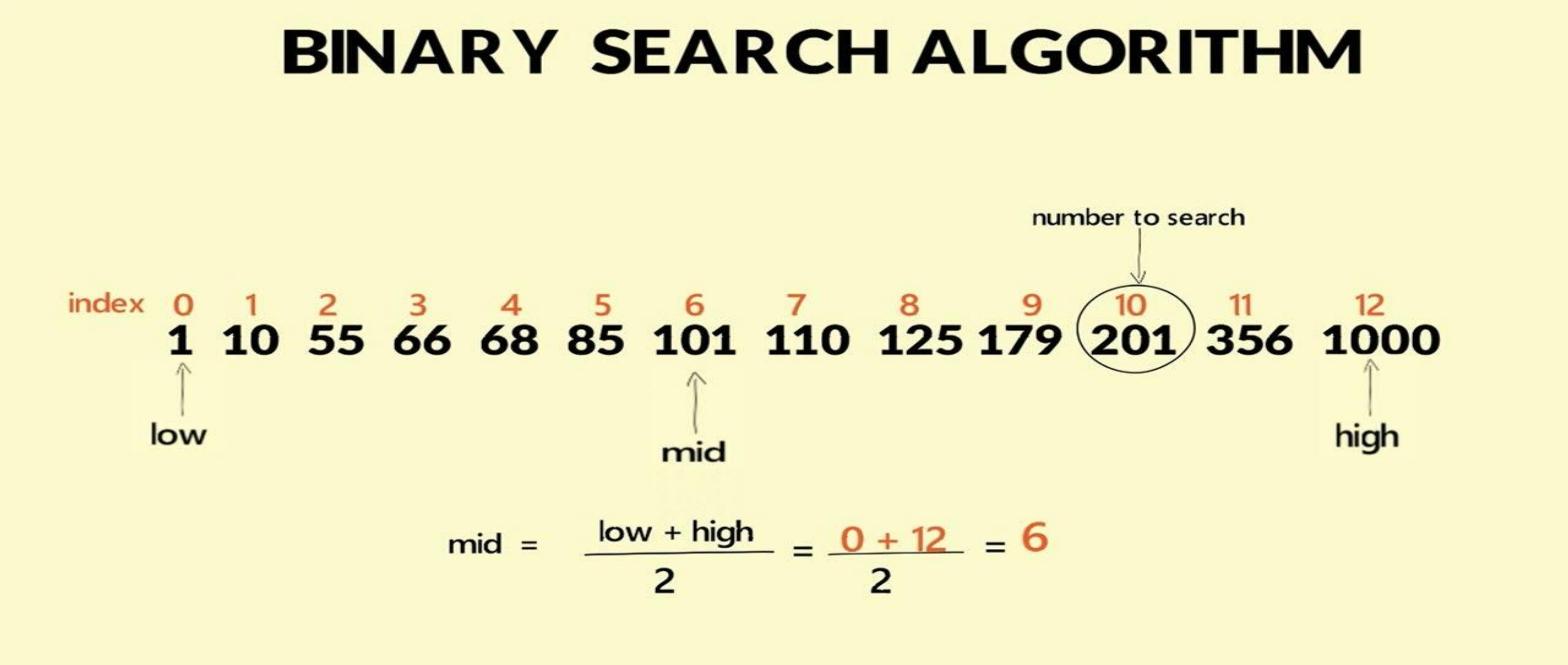
Step 2: Compare x with the middle element of the array.

Step 2.1: If x = middle element, then output it and stop.

Step 2.2: If x < middle element, then recursively solve the problem with x and the left half array.

Step 2.3: If x > middle element, then recursively solve the problem with x and the right half array.

Example of BinSearch



Algorithm BinSearch(a, low, high, x)

```
// a[]: sorted sequence in nondecreasing order
// low, high: the bounds for searching in a []
// x: the element to be searched
// If x = a[j], for some j, then return j else return –1
  if (low > high) then return –1          // invalid range
  if (low = high) then                    // if small P
    if (x == a[i]) then return i
    else return -1
  else            // divide P into two smaller subproblems
    mid = (low + high) / 2
    if (x == a[mid]) then return mid
    else if (x < a[mid]) then
        return BinSearch(a, low, mid-1, x)
    else return BinSearch(a, mid+1, high, x)
```

Binary Search

Precondition: S is a sorted list

```
index binsearch(number n, index low, index high,
          const keytype S[], keytype x)
      if low ≤ high then
          mid = (low + high) / 2
          if x = S[mid] then
              return mid
          elsif x < s[mid] then
              return binsearch(n, low, mid-1, S, x)
          else
              return binsearch(n, mid+1, high, S, x)
      else
          return 0
   end binsearch
```

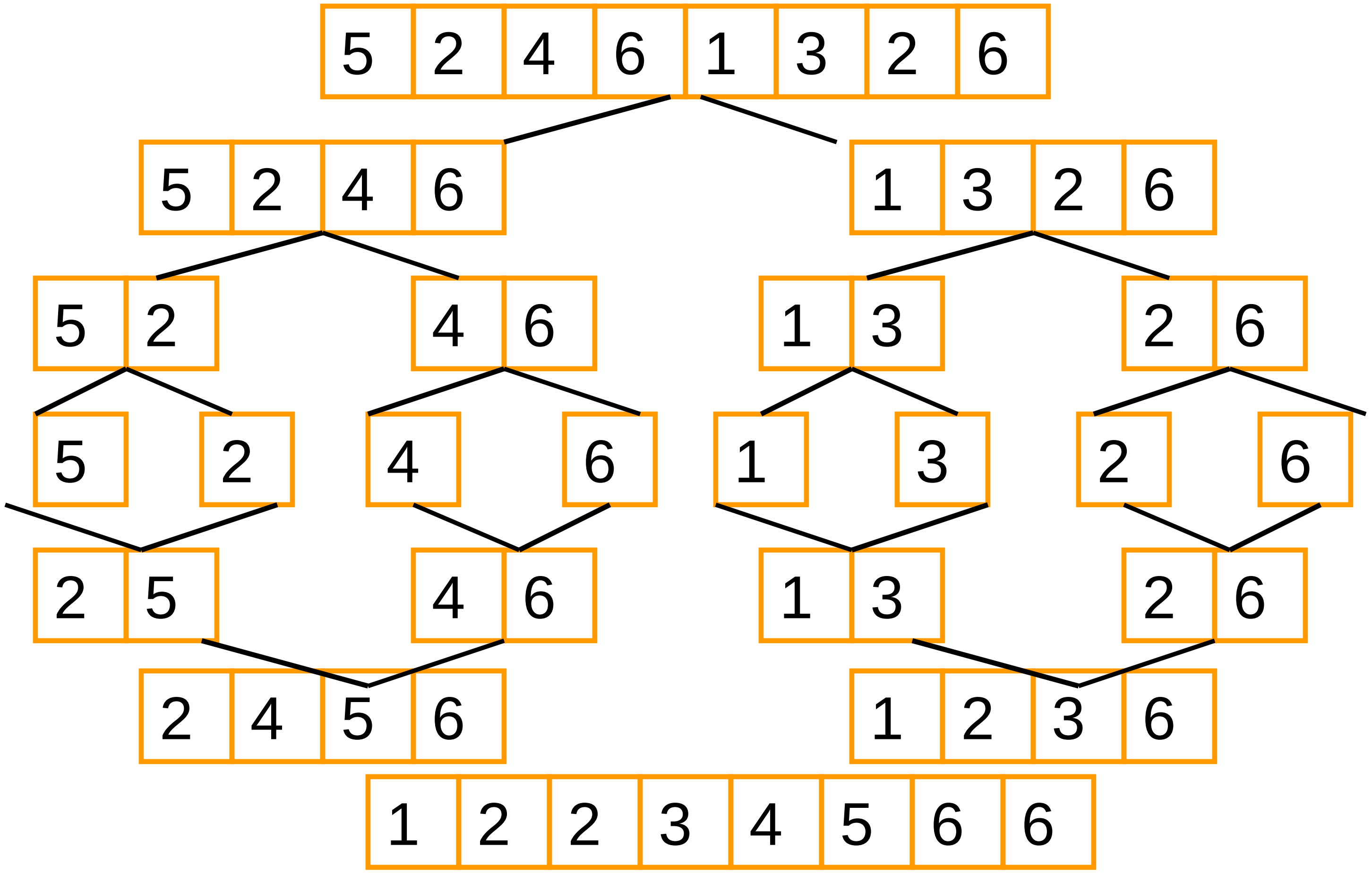
Analysis of Binary Search

- **Time efficiency T(N) = T(N/2) + 1**
  - Worst case: *O(log(n))*
  - Best case: O(1)
- **Optimal for searching a sorted array**
- **Limitations: must be a sorted array (not linked list)**

3. Merge sort

- DIVIDE the input sequence in half
- RECURSIVELY sort the two halves. Basis of the recursion is sequence with 1 key
- COMBINE the two sorted subsequences by merging them

Merge sort Example



Merge sort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Merge Sort algorithm

- The Merge Sort function keeps on splitting an array into two halves until a condition is met where we try to perform Merge Sort on a subarray of size 1, i.e., **p == r**.
- And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

1.ALGORITHM-MERGE SORT

2.1. If p**<r**

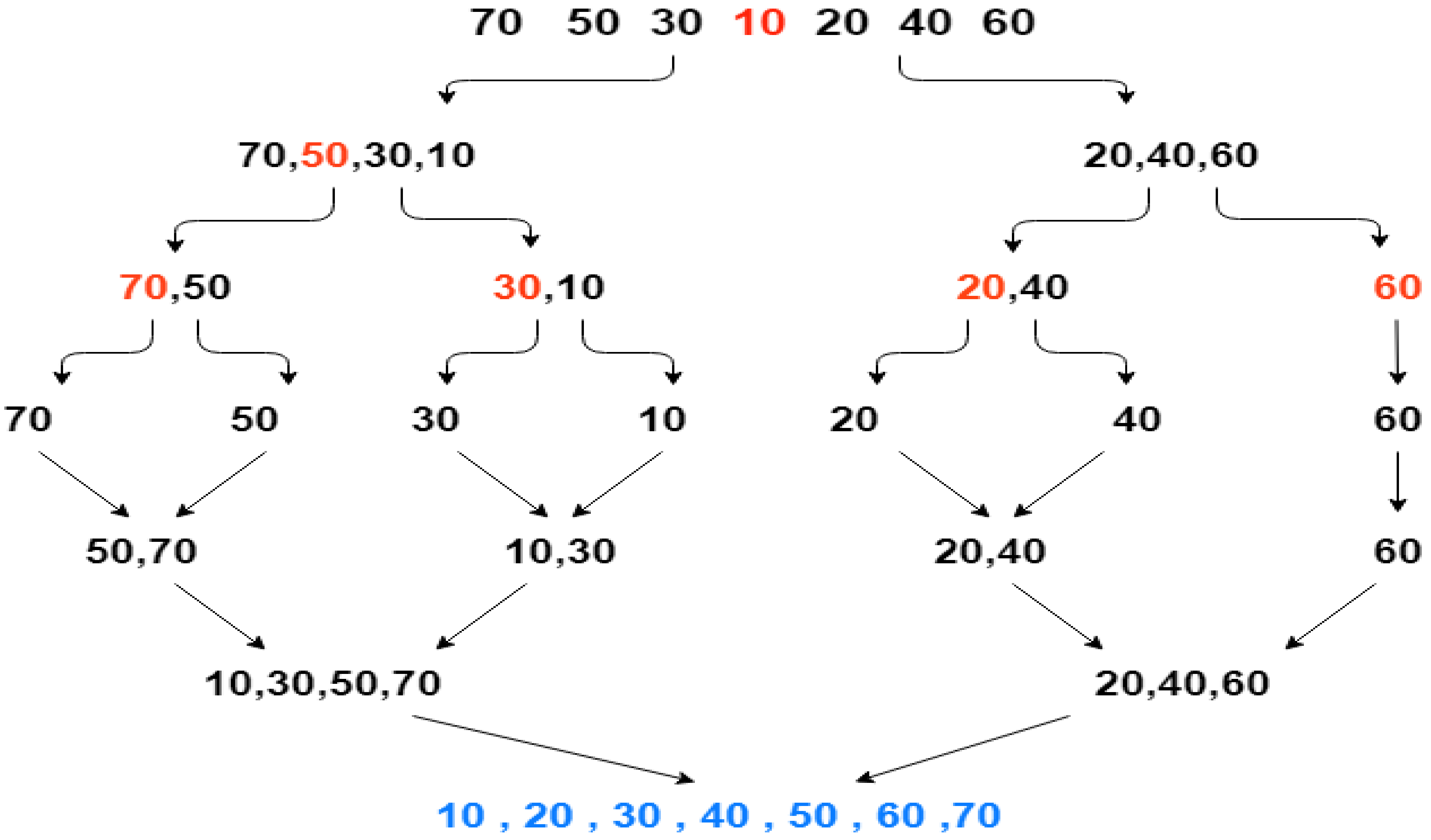
3.2. Then q → (p+ r)/2

4.3. MERGE-SORT (A, p, q)

5.4. MERGE-SORT (A, q+1,r)

6.5. MERGE (A, p, q, r)

- Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.
- As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.
- The following figure illustrates the dividing (splitting) procedure.


70 50 30 10 20 40 60
70,50,30,10
20,40,60
70,50
30,10
20,40
60
70
50
30
10
20
40
60
50,70
10,30
20,40
60
10,30,50,70
20,40,60
10 , 20 , 30 , 40 , 50 , 60 ,70

Algorithm Merge-Sort

Input: A set S of n elements.

Output: The sorted sequence of the inputs in non decreasing order.

Step 1: If $|S| \le 2$, solve it directly.

Step 2: Recursively apply this algorithm to solve the left half part and right half part of S, and the results are stored in S_1 and S_2, respectively.

Step 3: Perform the two-way merge scheme on S_1 and S_2.

Method of Solving Recurrences

1. Substitution method
2. Iteration method
3. Master method

3. Master Theorem (Simplified)

Theorem: (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + n^k,$$

defined for $n \geq 0$. (As usual let us assume that n is a power of b. The basis case, $T(1)$ can be any constant value.) Then

Case 1: if $a > b^k$ then $T(n) \in \Theta(n^{\log_b a})$.

Case 2: if $a = b^k$ then $T(n) \in \Theta(n^k \log n)$.

Case 3: if $a < b^k$ then $T(n) \in \Theta(n^k)$.

master method:

- a is the number of subproblems that are solved recursively; i.e. the number of recursive calls.
- b is the size of each subproblem relative to n; n/b is the size of the input to the recursive call.
- f(n) is the cost of dividing and recombining the subproblems.

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

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

Master Theorem: Pitfalls

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(x)
  - f(n) is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
  - b cannot be expressed as a constant, e.g. $T(n) = T(\sqrt{n})$
- The three cases of the master theorem do not cover all possibilities.
- The master theorem provides a general method for solving recurrences

Master Theorem: Example 1

- Let $T(n) = T(n/2) + ½\, n^2 + n$. What are the parameters?

  $a =$ 1

  $b =$ 2

  $d =$ 2

  Therefore, which condition applies?

  $1 < 2^2$, case 1 applies

- We conclude that

$$T(n) \in \Theta(n^d) = \Theta\ (n^2)$$

Master Theorem: Example 2

- Let $T(n) = 2\,T(n/4) + \sqrt{n} + 42$. What are the parameters?

  a = 2

  b = 4

  d = 1/2

  Therefore, which condition applies?

  $2 = 4^{1/2}$, case 2 applies

- We conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

Master Theorem: Example 3

- Let $T(n)= 3\,T(n/2) + 3/4n + 1$. What are the parameters?

  a = 3

  b = 2

  d = 1

  Therefore, which condition applies?

  $3 > 2^1$, case 3 applies

- We conclude that

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

- Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$

  No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

Recurrence Relation for Merge sort

- Let $T(n)$ be worst case time on a sequence of n keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2\,T(n/2) + \Theta(n)$
  - two sub problems of size $n/2$ each that are solved recursively
  - $\Theta(n)$ time to do the merge

Merge sort

- Sort into nondecreasing order

```
        [25] [57] [48] [37] [12] [92] [86] [33]
pass 1
        [25   57] [37   48] [12   92] [33   86]
pass 2
        [25   37   48   57] [12   33   86   92]
pass 3
        [12   25   33   37   48   57   86   92]
```

- $\log_2 n$ passes are required.
- time complexity: O(nlogn)

Time complexity

- Time complexity:

  T(n): # of comparisons

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

- Calculation of T(n):

  Assume $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \\ &\quad\vdots \\ &=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1 \\ &=2^{k-1}+2^{k-2}+\ldots+4+2+1 \\ &=2^k-1 = n-1 \end{aligned}$$

Time complexity of the general algorithm

- Time complexity:

$$T(n)=\begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where S(n) : time for splitting

M(n) : time for merging

b : a constant

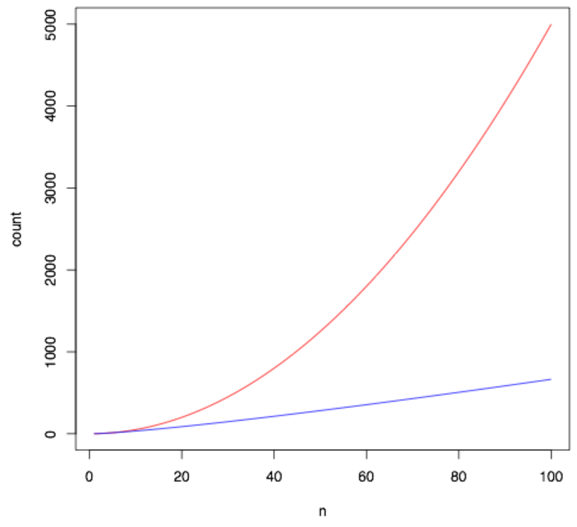
c : a constant

Best case: split in the middle — $\Theta(n \log n)$
Worst case: sorted array! — $\Theta(n^2)$
Average case: random arrays — $\Theta(n \log n)$

Note: Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Recap: Divide and Conquer Algorithms

- The divide and conquer strategy often reduces the number of iterations of the main loop from *n* to $\log_2 n$
  - binary search: $\mathcal{O}(\log_2 n)$
  - merge sort: $\mathcal{O}(n \times \log_2 n)$
  - Quick Sort: $\mathcal{O}(n \times \log_2 n)$
- It may not look like much, but the reduction in the number of iterations is significant for larger problems




Quick
Sort
Merge
Sort

5. Strassen's Matrix Multiplication

- First we will discuss the general method of matrix multiplication and later we will discuss Strassen's matrix multiplication algorithm

Problem Statement

- Let us consider two matrices A and B. We want to calculate the resultant matrix C by multiplying A and B.

Method

- First, we will discuss naïve method and its complexity. Here, we are calculating $\boldsymbol{C = AXB}$. Using Naïve method, two matrices ($\boldsymbol{A}$ and $\boldsymbol{B}$) can be multiplied if the order of these matrices are $\boldsymbol{p} \times \boldsymbol{q}$ and $\boldsymbol{q} \times \boldsymbol{r}$. Following is the algorithm

Algorithm: Matrix-Multiplication (X, Y, Z)

```
for i = 1 to p do
   for j = 1 to r do
      Z[i,j] := 0
      for k = 1 to q do
         Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

Complexity

- Here, we assume that integer operations take ***O(1)*** time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes ***O(n^3)*** time to execute.

Strassen's Matrix Multiplication Algorithm

- In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Matrix Multiplication

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

Multiplications: 8

Additions: 4

Addition of 2 matrices takes $O(N^2)$ time.

$$T(N) = 8T(N/2) + O(N^2)$$

Time Complexity of the above method is $O(N^3)$.

Faster Matrix Multiplication

- Clever idea due to Strassen
- Start with a recursive version of the straight forward algorithm
  - divide *A*, *B* and *C* into 4 quadrants
  - compute the 4 quadrants of *C* by doing 8 matrix multiplications on the quadrants of *A* and *B*, plus $\Theta(n^2)$ scalar operations

Strassen's Matrix Multiplication

- There is a way to get all the required information with only 7 matrix multiplications, instead of 8.
- Recurrence for new algorithm is
  - $T(\mathrm{n}) = 7T(n/2) + \Theta(n^2)$

You just need to remember 4 Rules :

- AHED (Learn it as 'Ahead')
- Diagonal
- Last CR
- First CR

Also, consider X as (Row +) and Y as (Column -) matrix

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{X} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

X , Y and **C** are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

-AHED
-Diagonals
-Last CR
-First CR

Check for Row (+)

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Check for Column (-)

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

-AHED
-Diagonals
-Last CR
-First CR

Check for Row (+)

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Check for Column (-)

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

-AHED
-Diagonals
-Last CR
-First CR

Idea: Reduce the number of recursive calls to 7.

Formula:

p1 = a(f-h)
p2 = (a+b)h
p3 = (c+d)e
p4 = d(g-e)

p5 = (a+d)(e+h)
p6 = (b-d)(g+h)
p7 = (a-c)(e+f)

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Complexity in Strassen matrix

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

Multiplications: 7

Addition & Subtraction of 2 matrices takes **$O(N^2)$** time.

$$T(N) = 7T(N/2) + O(N^2)$$

Time Complexity of the above method is **$O(N^{\log 7})$**

example

$$\begin{bmatrix} 1 & 0 \\ -3 & 2 \end{bmatrix} \cdot \begin{bmatrix} -1 & 4 \\ 3 & 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ -3 & 2 \end{bmatrix} \cdot \begin{bmatrix} -1 & 4 \\ 3 & 5 \end{bmatrix}$$

$$\begin{array}{c} \text{1st Row} \\ \text{Times} \\ \text{1st Column} \\ \\ \\ \\ \text{2nd Row} \\ \text{Times} \\ \text{1st Column} \end{array} \left[\begin{array}{c|c} (1)(-1)+(0)(3) & (1)(4)+(0)(5) \\ -1+0 & 4+0 \\ -1 & 4 \\ \hline (-3)(-1)+(2)(3) & (-3)(4)+(2)(5) \\ 3+6 & -12+10 \\ 9 & -2 \end{array}\right] \begin{array}{c} \text{1st Row} \\ \text{Times} \\ \text{2nd Column} \\ \\ \\ \\ \text{2nd Row} \\ \text{Times} \\ \text{2nd Column} \end{array}$$

$$\begin{bmatrix} 1 & 0 \\ -3 & 2 \end{bmatrix} \cdot \begin{bmatrix} -1 & 4 \\ 3 & 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ -3 & 2 \end{bmatrix} \cdot \begin{bmatrix} -1 & 4 \\ 3 & 5 \end{bmatrix}$$

Final Answer: $\begin{bmatrix} -1 & 4 \\ 9 & -2 \end{bmatrix}$

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem:
If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n^3)$

$T(n) = 7T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^{\log_2 7}) = \Theta(n^{2.1})$

(Strassen's Algorithm for Matrix Multiplication)


Learning is key to success