1. What is Algorithm? (Neat)

An algorithm is a sequence of unambiguous instructio for solving problem i.e for obtaining a required o/p for any legitimate input in finite amount of time.

2) Euclid's algorithm for computing gcd (m,n)

```
| compute gcd (m,n)
// I/P :- 2 non -ve, not both zero integer m & n
// o/p :- Greatest common devisor of m & n
~~when~~ while n ≠ 0 do
        r ← m mod n                r ≠ 0 do
        m ← n                      r ← m mod r
        n ← r                      m ← n
    return m                       n ← r
                                   return m
```

Step 1 :- If n=0, return m
Step 2 :- m ÷ n , remainder $\overset{\frown}{=}$ as $\boxed{r}$
Step 3 :- Assign value of n to m, r to n return m

3) Consecutive Integer Checking for gcd (m,n)
Step 1 :- Assign value of min {m, n} to t
Step 2 :- devide m by t , if r = 0 goto sec 3 or else goto step 4.
Step 3 :- n ÷ t , r = 0 return t
Step 4 :- decrease value of t by 1 . Goto step 2

```
1) t ← min (m,n)
2) if ~~m~~ n%t = 0   goto 3
                else goto 4
3) if m%t = 0   return t
                else goto 4
4) t ← t-1
5) goto 2
```

4) Middle School procedure :-
→ 1) Find prime factor of m
2) Find prime factor of n
3) Identify all the common factor in two prime expansion in ① & ②
4) Compute the product of all common factor & returns it.

Ex :- 60 , 24

60 = 2 × 2 × 3 × 5

24 = 2 × 2 × 2 × 3

2 × 2 × 3 = 12

60 = 2 × 30

2 × 3 × 10

2 × 2 × 3 × 5 ✓

5) sieve (n)

```
// Implements the sieve of ~~Erath~~ Eratosthenes
// I/p :- +ve integer n>1
// o/p :- Array L of all prime number less than / equal to
```

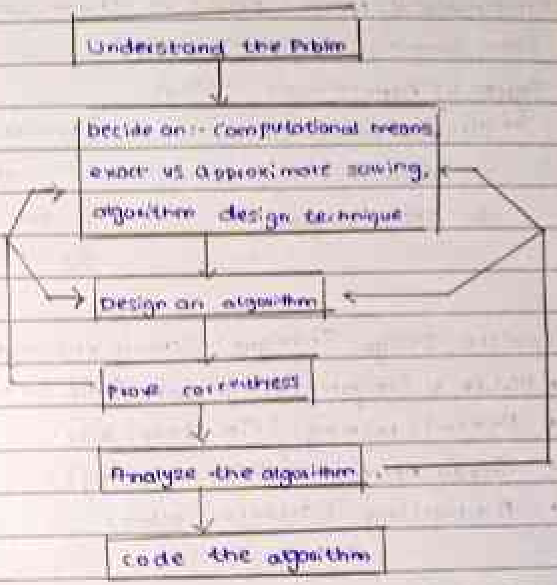
```
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0
        j ← p*p
    while j ≤ n do
        A[j] ← 0
        j ← j+p

i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i+1
return L
```

Ex:- 2, 3, ④, 5, ⑥, 7, ⑧, 9, ⑩, 11, ⑫, 13, ⑭, 15,

2, 3, x, 5, x, 7, x, ⑨, x, 11, x, 13, x, ⑮

2, 3, , 5, , 7, , , , 11, , 13,

c) Algorithm design & Analysis process.



① Understanding the pblm.

→ Before designing an algorithm, it's essential to understand the pblm clearly.

→ This includes indetifying the i/p, o/p, constraints & expected behavior of the soln.

→ It helps in selecting an efficient approach

② Decide on Computational Means & Design Techniques.

→ Computational Means:- Determine the resources available (CPU, memory, parallel computing)

→ Exact vs Approximate solving:-

↳ give the precise ans → provide near-optimal soln when exact soln is too slow.

Ex:- Dijkstra's algorithm for shortest path.

Eg: np-hard pblm.

→ Algorithm Design Technique:- choose the right method.

- Divide & Conquer (Merge sort, Quick sort)
- Dynamic prgming (Fib. knapsack)
- Greedy Approach (Huffman coding)
- Backtracking (sudoku solver)

③ Design & Algorithm

→ Develop a step-by-step procedure to solve the pblm

→ Consider factor like efficiency, simplicity & feasibility

→ Represent algorithm using pseudocode / flowchart before implementation

④ Prove Correctness.

→ Ensure that the algorithm always produce the correct o/p

→ Common technique for correctness proof:-

- Mathematical Induction
- loop Invariant
- Contradiction method

(5) Analyze the Algorithm

→ Time Complexity :- Measure how the running time ↑ses with I/p size (Big O notation)

Ex :- O(1) → constant time

$O(\log n)$ → Logarithmic (Binary search)

$O(n)$ → linear (Linear sort)

$O(n^2)$ → Quadratic (Bubble sort)

$O(2^n)$ → Exponential (Brute-force Soln)

→ Space Complexity :- Evaluate how many memory the algorithm uses.

(6) Code the Algorithm

→ Convert the designed algorithm into an actual prgm lang (C, python, Java, etc...)

→ Optimize for better readability, maintainability & efficiency

→ Perform testing using sample I/p

7) Sequential Search | Linear search (A[0..n-1], K)

```
// Search for the given value in a given array by SS.
// I/p :- A[0..n-1] and search key K.
// O/p :- The index 1st element in A that matches K
   // or -1 no matching elements
  i ← 0
 while i<n and A[i] ≠ K do
        i ← i+1
 if i<n return i     else return -1
```

Worst case:-

→ It happens when the element is at the last position / not in the list at all.

→ Time complexity $C_{worst}(n) = n$

Best case

→ when found at the 1st position

→ $C(n) = 1$

Average case

→ found somewhere in middle.

$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$

case	no of comparision	Time complexity.
Best	1	O(1)
worst	n	O(n)
Avg	n/2 ↓ $\frac{p(n+1)}{2} + n(1-p)$	O(n)

8) Asymphotic Notation & Basic Efficiency classes.

→ Ans:- It is $\overset{a}{\wedge}$ way of comparing func that ignores constants factors & small i/p size.

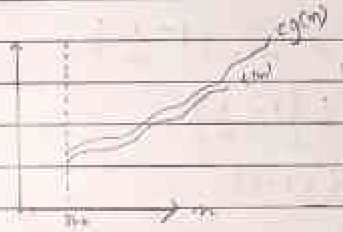
1)

$100n + 5 \in O(n^2)$ ✓

$n^2 \in O(n)$ ✗

1) Big Oh Notation (O):-

① A function $t(n)$ is said to be $O(g(n))$, denoted $t(n) \in O(g(n))$, ② if $t(n)$ is bounded above by some constant multiple of $g(n)$ $\forall$ n. ③ i.e if there exist some +ve constant c & some non -ve integer n_0 such that $t(n) \le c \cdot g(n)$ $\forall$ $n \ge n_0$



$t(n) = 100n + 5$

$\Rightarrow t(n) \le c \cdot g(n)$ $\forall$ $n \ge n_0$

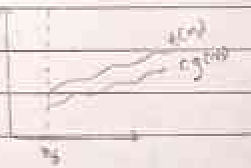
$100n + 5 \le 101n$ $\forall$ $n \ge 5$

$\Rightarrow c = 101$ $g(n) = n$ $n_0 = 5$

$\therefore t(n) \in O(g(n))$

2) Big Omega Notation (Ω):-

①, ② — below ③ $t(n) \ge c \cdot g(n)$ $\forall$ $n \ge n_0$



$t(n) = 10n^3 + 5$

$\Rightarrow t(n) \ge c \cdot g(n)$ $\forall$ $n \ge n_0$

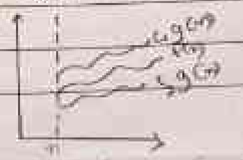
$10n^3 + 5 \ge 10n^3$ $\forall$ $n \ge 0$

$\Rightarrow c = 10,$ $g(n) = n^3$ $n_0 = 0$

$\therefore t(n) \in \Omega(g(n^3))$

3) Big Theta Notation (θ)

①, ② both above, below ③ $c_2 g(n) \le t(n) \le c_1 g(n)$



9) Pblm

a) Compare order of growth of $\frac{1}{2}n(n-1)$ & n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2}$$

$$= \frac{1}{2}\lim_{n\to\infty} \frac{n^2}{n^2} - \frac{n}{n^2}$$

$$= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)$$

$$= \frac{1}{2}\left\{1-\frac{1}{\infty}\right\}$$

$$= \frac{1}{2}\{1-0\}$$

$$= \frac{1}{2}$$ two alg have same growth n^2

$$= \frac{1}{2}n(n-1) \in \Theta(n^2)$$

b) $\log_2 n$ & $\sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\ln n / \ln 2}{\sqrt{n}}$$

$\log_2 n = \frac{\ln n}{\ln 2}$

3) $n!$ & 2^n

$$\lim_{n\to\infty} \frac{n!}{2^n} =$$

10) General plan for Analysis of Non-Recursive Algorithms.

→ a) Decide on a parameter indicating an i/p size.

b) Identify the algorithm's basic operation

c) Check whether the numb of times the basic operation executed depends only on size of i/p. If it also depends on some additional property - worst, best, avg case & if necessary, best case have to be investigated separately.

d) Set up sum expressing the num of times the algorithm's basic operation is executed.

e) Using standard formulas & rules of sum manipulation, either find a closed form formula for count / at very least, establish its order of growth

SF | Pos →?

(i) largest element in a list / MaxElement $A[0 .. n-1]$

→ // Determines the value of the largest element in array
// I/p :- $A[0 .. n-1]$ of real num
// O/p :- value of largest element

```
maxval ← A[0]
 for i ← 1 to n-1 do
     if A[i] > maxval
         maxval ← A[i]
 return maxval
```

Step 1 :- i/p size n
2 :- Basic operation :- Comparision

$$C(n) = \sum_{i=1}^{n-1} 1$$

$$C(n) = n-1-1+1$$
$$= n-1 \in \Theta(n)$$

(ii) Element Uniqueness prblm
// all element distinct
// I/P:- $A[0 .. n-1]$
// O/P :- Return true if all element in A distinct else false

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] = A[j] return false
return true.
```

Step 1:- i/p size - n

2:- B.O → Comparision.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n-1-i-\cancel{1}+\cancel{1}$$

$(n-2-0+1)(n-1-i+1)$ $(n-2)-1-i$

$\underline{(n-1)(n-2)}$ $= n-2\,2$

(iii) Matrix Multiplication.

// Multiplies 2 square matrices of order n by

// i/p :- $n \times n$ $[A]\ [B] = [C]$

// $C = AB \rightarrow$ o/p 2×3 3×4 $2,4$

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0.0
        for k ← 0 to n-1 do
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C.
```

$2,3 \times (3,4)$

$(2,4)$

Step 1 :- i/p size - n

2:- B.O. multiplication

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n-\cancel{1}+\cancel{1}$$

$$= \underline{\underline{n^3}} \in \Theta(n^3)$$

Date / /20

(9u) find the numb of binary digits.

```
// I/P → +ve decimal no. n
// O/P → binary numb

count ← 1
while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

step 1:- i/p size = n

B.O - addtion

$(n) =$

11>

a)

b)

c) check whether the no. of times the basic operation ~~d)~~ is executed can vary on different i/p of same size

d) set up a recurrence relation, with an appropriate initial condition, for the number of time the basic operation is executed

e) Solve the recurrence / at least, ascertain the order of growth of its soln

(i) factorial function $F(n) = n!$ for non -ve integer n.

if $n=0$ return 1

else return $F(n-1) * n$

~~$M(n)$~~ = i/p size $= n$

B.O = Multiplication

$M(n) = M(n-1) + 1$ ✓ for $n>0$

$M(0) = 0$ for $n=0$ ✓

$M(n) = M(n-1) + 1$

$= [M(n-2) + 1] + 1]$

$= M(n-2) + 2$

$M(n) = M(n-3) + 3$

*

$M(n) = M(n-i) + i$ ✓

Substitate $i=n$

$M(n) = m(n-n) + n$

$= M(0) + n$

$= \underline{\underline{n}} \in \Theta(n)$

(ii) TOH

Step 1:- i/p size $= n$

2:- B.O :- no. of mover

$M(n) = M(n-1) + 1 + m(n-1)$

$= 2M(n-1) + 1$ for $n>0$

$M(1) = 1$ for $n=1$

$① \Rightarrow M(n) = ~~2 [2M (n-2) + 1]~~ + 1$

$2 \cdot M(n-1) + 1$

$= 2 [2M(n-2) + 1] + 1$

$= 2^2 M(n-2) + 2 + 1$

$= 2^3 [M(n-3) + 2^2 + 2 + 1]$

$= 2^4 [M(n-4)] + 2^3 + 2^2 + 2 + 1]$

$= 2^i M(n-i) + 2^{i-1} + 2^{i-2} \dots 2+1$

$= 2^i M(n-i) + 2^i - 1$ $\quad \because \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

$= 2^{n-1} M(n(n-1)) 2^{n-1} - 1$

$= 2^{n-1} M(1) + 2^{n-1} - 1$ $\quad \sum_{i=0}^{i-1} \quad 2^{i-1+1} - 1$

$= 2^n - 1 \in \Theta(2^n)$

(iii) BinRec (n)

```
if n=1 return 1
else return (⌊n/2⌋)+1
```

Step 1 :- i/p size n

Step 2 :- B.O :- Addition

$A(n) = A(\lfloor n/2 \rfloor) + 1$

$A(1) = 0$

$n = 2^k$

$A(2^k) = A\left(\frac{2^k}{2}\right) + 1$

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

$\theta(2^0) = 0 ~~\theta(2^{-1})~~$

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

$= A[2^{k-2}] + 2$

$= A(2^{k-3}) + 3$

$= A(2^{k-i}) + i$

Substitute $i = k$

$= A(2^{k-k}) + k$

$= \underline{\underline{k}}$

$n = 2^k$

$k = \log_2 n$

$\underline{\underline{A(n) = \log_2 n \in (\log n)}}$

Brute force is a straight-forward approach to solving a problm, usually directly based on the problm statement & defination of the concepts involved.

(i) Selection Sort

```
for i ← 0 to n-2 do
    ~~for i~~ min ← i
    for j ← ~~i~~ i+1 to n-1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = n-1-\cancel{i}\cancel{-1}\cancel{+1}$ $\boxed{n^2}$

$(n-i-1)$

```
89   45   68   90   29   34   17
17 | 45   68   90   29   30   89
17   29 | 68   90   ~~89~~   30   89
17   29   30 | 90   45   68   89
17   29   30   45 | 90   68   89
17   29   30   45   68 | 90   89
17   29   30   45   68   89 | 90
```

$S(n)$ = key swap

$$S(n) = \sum_{i=0}^{n-2} 1 = n-2+1$$
$$= n-1 \in \Theta(n)$$

(ii) Bubble sort

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j] swap A[j] and A[j+1]
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{0}^{n-2-i} 1 = \sum_{i=0}^{n-2} n-2-i+1$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{(n-1)n}{2} \in (n^2)$$

$$S(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} = n^2$$

Sequential Search and Brute-Force String Matching

```
ALGORITHM  SequentialSearch2(A[0..n], K)
    //Implements sequential search with a search key as a sentinel
    //Input: An array A of n elements and a search key K
    //Output: The index of the first element in A[0..n − 1] whose value is
    //        equal to K or −1 if no such element is found
    A[n] ← K
    i ← 0
    while A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

```
ALGORITHM  BruteForceStringMatch($T[0..n-1]$, $P[0..m-1]$)
    //Implements brute-force string matching
    //Input: An array $T[0..n-1]$ of $n$ characters representing a text and
    //        an array $P[0..m-1]$ of $m$ characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //        matching substring or $-1$ if the search is unsuccessful
    for $i \leftarrow 0$ to $n - m$ do
        $j \leftarrow 0$
        while $j < m$ and $P[j] = T[i + j]$ do
            $j \leftarrow j + 1$
        if $j = m$ return $i$
    return $-1$
```

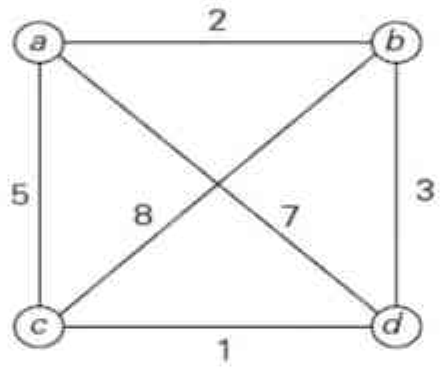
```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

- Exhaustive search is simply a brute-force approach to combinatorial problems

- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element

Traveling Salesman Problem

- The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances
- Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph



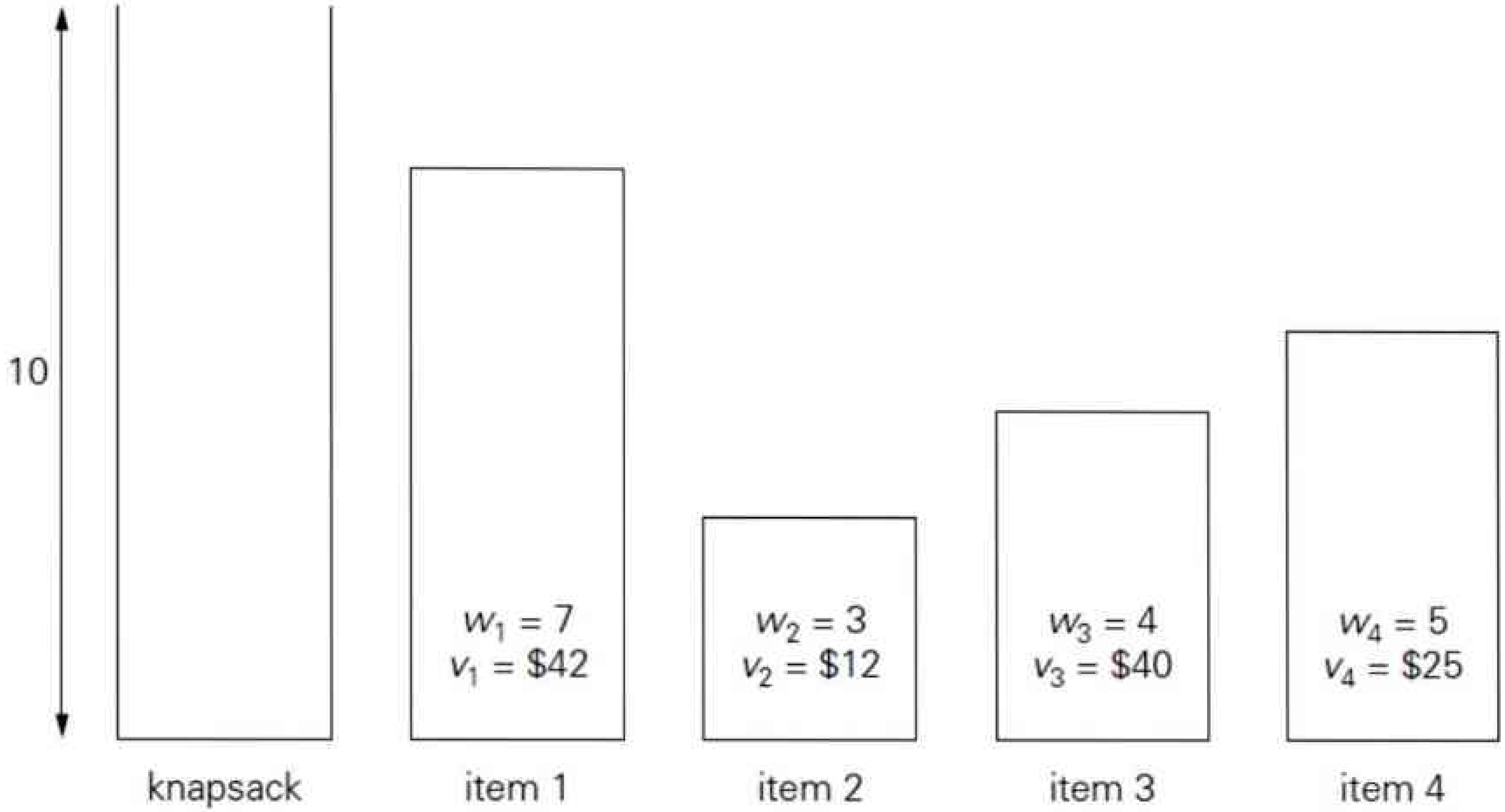
Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

We can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them

Knapsack Problem

Given n items of known weights w1, w2, . . . , wn and values v1, v2, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack

Since the number of subsets of an n-element set is 2n, the exhaustive search leads to a $O(2^n)$ algorithm, no matter how efficiently individual subsets are generated



Sunday, February 16, 2025

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Assignment Problem

- There are n people who need to be assigned to execute n jobs, one person per job.

- That is, each person is assigned to exactly one job and each job is assigned to exactly one person

- The cost that would accrue if the ith person is assigned to the jth job is a known quantity C[i, j] for each pair i, j = 1, 2, . . . , n

- The problem is to find an assignment with the minimum total cost

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18	
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30	
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24	
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26	etc.
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33	
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23	

FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search.

- Divide-and-conquer algorithms work according to the following general plan:
  - A problem is divided into several subproblems of the same type, ideally of about equal size
  - The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough)
  - If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

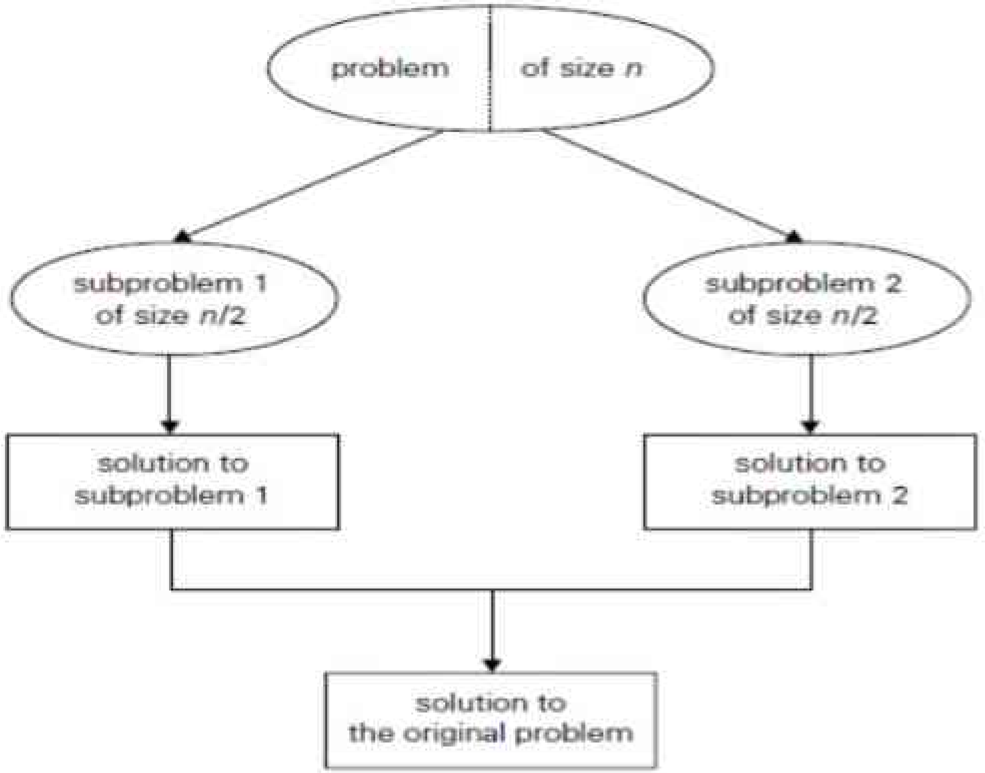

problem
of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

FIGURE 5.1 Divide-and-conquer technique (typical case).

Master method

$$T(n) = aT(n/b) + f(n), \tag{5.1}$$

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

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

Analogous results hold for the O and Ω notations, too.

- Mergesort is a perfect example of a successful application of the divide-and conquer technique

- It sorts a given array $A[0..n - 1]$ by dividing it into two halves $A[0..n/2 - 1]$ and $A[n/2..n - 1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one

```
ALGORITHM    Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing 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)    //see below
```

```
ALGORITHM  $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

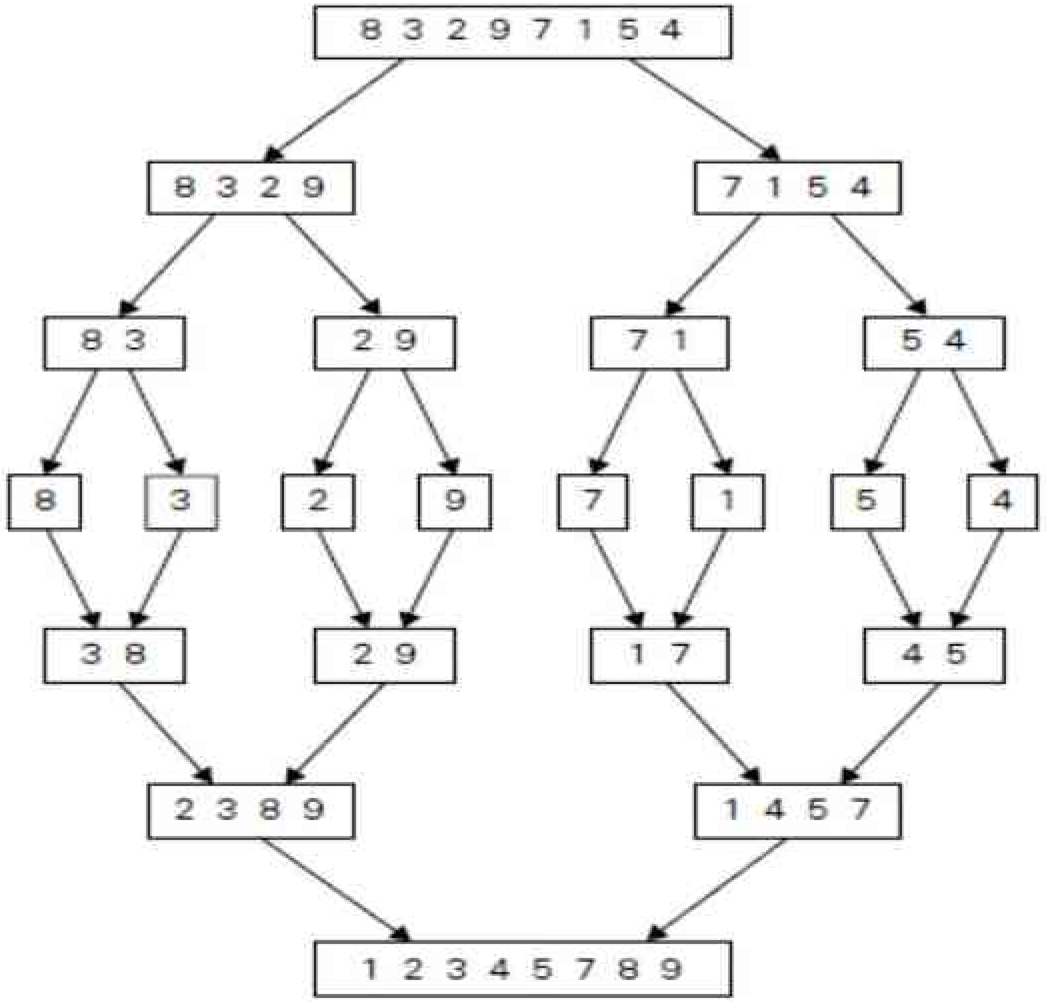


FIGURE 5.2 Example of mergesort operation.

- Hence, according to the Master Theorem

$$C_{worst}(n) \in \Theta(n \log n)$$

Quicksort

- Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value

- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \ge A[s]}$$

```
ALGORITHM Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //       indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

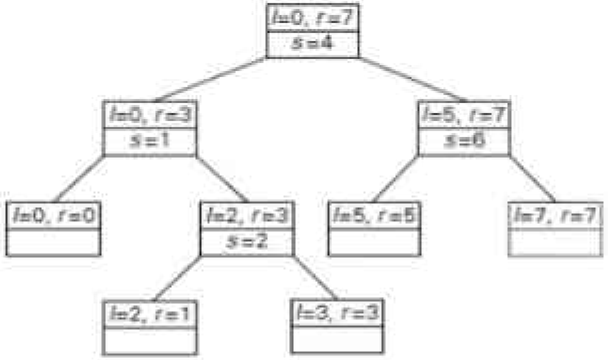
- As before, we start by selecting a pivot—an element with respect to whose value we are going to divide the subarray

- There are several different strategies for selecting a pivot

- We use the simplest strategy of selecting the subarray's first element: $p = A[l]$

```
ALGORITHM  HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: 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
        repeat j ← j − 1 until A[j] ≤ p
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])   //undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
```

0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{ij}{4}$				
		3	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9

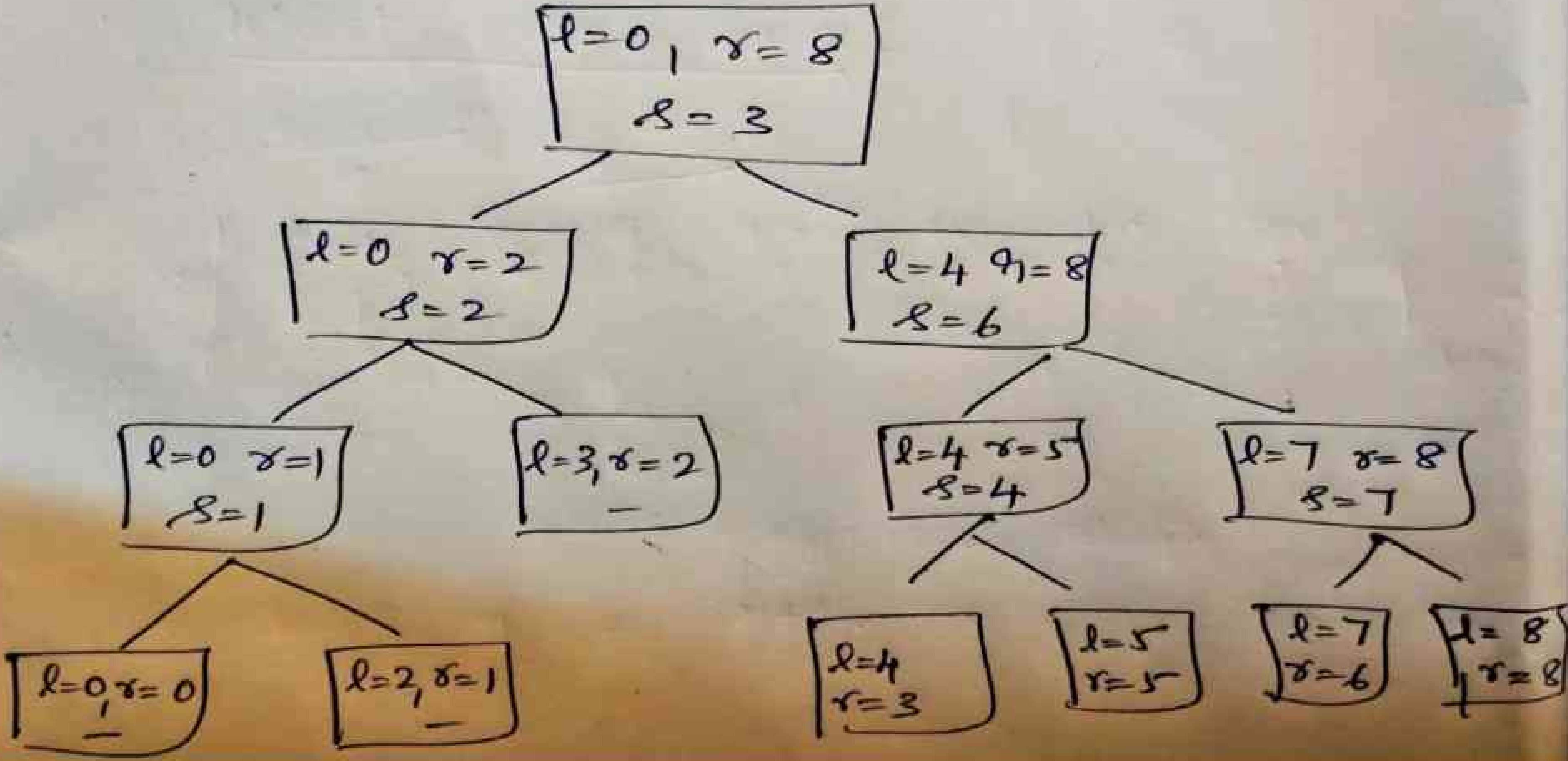


(b)

- The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

M, E, R, G, E, S, O, R, T



l=0, r=8
s=3
l=0 r=2
s=2
l=4 r=8
s=6
l=0 r=1
s=1
l=3, r=2
l=4 r=5
s=4
l=7 r=8
s=7
l=0, r=0
l=2, r=1
l=4
r=3
l=5
r=5
l=7
r=6
l=8
r=8

Binary Search

- Binary search is a remarkably efficient algorithm for searching in a sorted array

- It works by comparing a search key K with the array's middle element $A[m]$

- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\;\overset{\overset{K}{\updownarrow}}{A[m]}\;\underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}.$$

As an example, let us apply binary search to searching for $K = 70$ in the array

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

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

```
ALGORITHM  BinarySearch($A[0..n-1]$, $K$)
    //Implements nonrecursive binary search
    //Input: An array $A[0..n-1]$ sorted in ascending order and
    //       a search key $K$
    //Output: An index of the array's element that is equal to $K$
    //        or $-1$ if there is no such element
    $l \leftarrow 0$;   $r \leftarrow n-1$
    while $l \leq r$ do
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        if $K = A[m]$ return $m$
        else if $K < A[m]$ $r \leftarrow m-1$
        else $l \leftarrow m+1$
    return $-1$
```

- The worst-case time efficiency of binary search is $\Theta(\log n)$

Refer BinarySearch.c

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=0$, we can use the so-called ***little-oh notation***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.) ■

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n!\in\Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■