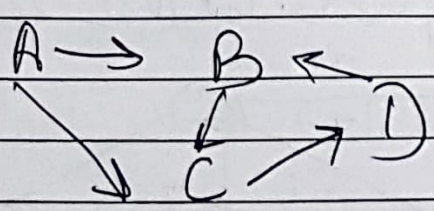
PAGE No.
DATE / /

Adjacency Matrix :- Better for dense graph
Adjacency List :- Better for sparse graph

→ Directed Graph :-
Only one direction edge

Ex 

BFS

9. HASHING
Storing data by dynamic DS called Hash table
Table organization and searching technique in which Insert/Delete/search operations done in O(1)
Uses HASH FUNCTION to put data in Hash table.
Enables to store & retrieve data quickly. Can be implemented as an array

Every entry in hash table is associated w/ some KEY.
Hash Function transforms key to a/an ~~for~~ table index.

Key (k) → [Hash Function] → Address h(k)

Ex:-

Name	key (k)	h(k) key%13
DLDA	985926	6
EDLC	970876	10
DSF	986047	11
COA	970428	5

0	1	2	3	4	5	6	7	8	9	10	11	12
					COA	D2DA				EDL	DSF	

Characteristics of good hash fnctn
→ Minimize collision i.e. distribute keys uniformly across the table
→ compute h(k) efficiently
→ Use all info in the key.

→ Collision:- Diff keys when hash to same locations.
Colliding keys are known as Synonyms.

Popular Hash Fnct^ns

- Division Remainder Method (Modulo Division Method)
- Mid Square Method
- Folding Method
- Multiplication Method

① Division Remainder Method :-

$h(k) = k \bmod n$ OR $k \bmod (n+1)$

Decide a n.

ID HUFFMAN ALGORITHM

Used for text compression

Lossy compression - Quality degraded

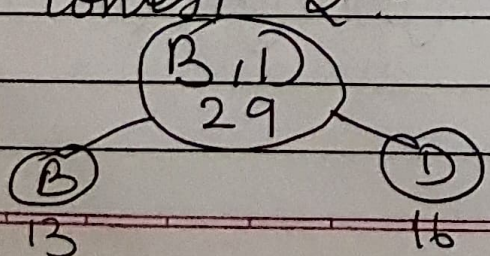
Ex:-

Character →	A	B	C	D	E
Frequency →	22	13	18	16	31

Sort ace to frequency

B	D	C	A	E
13	16	18	22	31

Combine lowest 2 :-

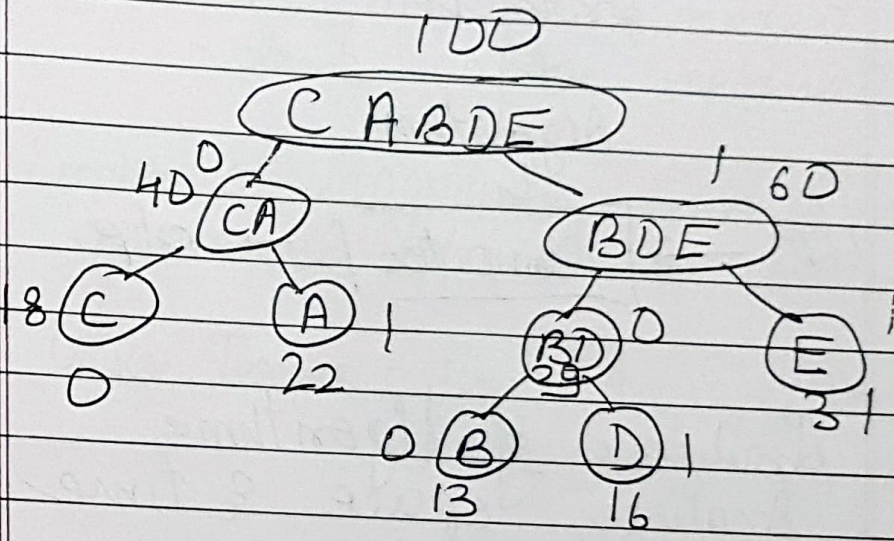


Reorder

C	A	B,D	E
18	22	29	31

B,D	E	C,A
29	31	40

CA	BDE
40	60



Code for each alphabet

A	0 1	22 x 2
B	1 0 0	13 x 3
C	0 0	18 x 2
D	1 0 1	16 x 3
E	1 1	31 x 2
		229 bits

Most frequently occuring character has lesser number of bits.

Before compression (100 + 40 + 60 + 29 + 31 + 18 + 22 + 31 + 13 + 16) = 8 000 bits

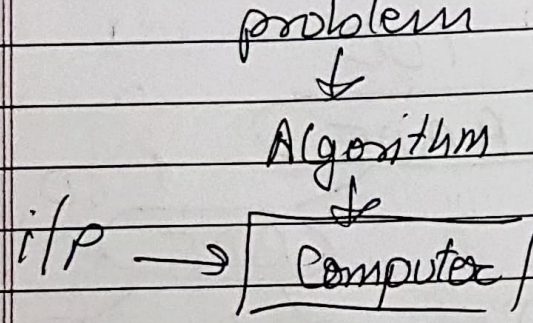
After compression = 229 bits

PAGE No.
DATE / /

Analysis of Algorithms

1 Introduction

Algorithm - Well defined computational procedure that takes some value or set of values as input & produces value(s) as output



→ Analysis of Algorithms
Analyze space & time complexity

Algorithms can be specified in natural language or pseudo code

Time Complexity :- Amt of computer time an algo needs to execute & get the result

PAGE No.
DATE / /

Space Complexity - Amt of memory needed to run an algo.

⇒ Priori Analysis
- Machine independent,
- done before implementing

⇒ Posteriori Analysis
- Target m/c dependent
- done after implementing

⇒ Order of Magnitude - Sum of no. of occurences of stmts contained in it.
Other Terms :- Best / Average / Worst case Running time

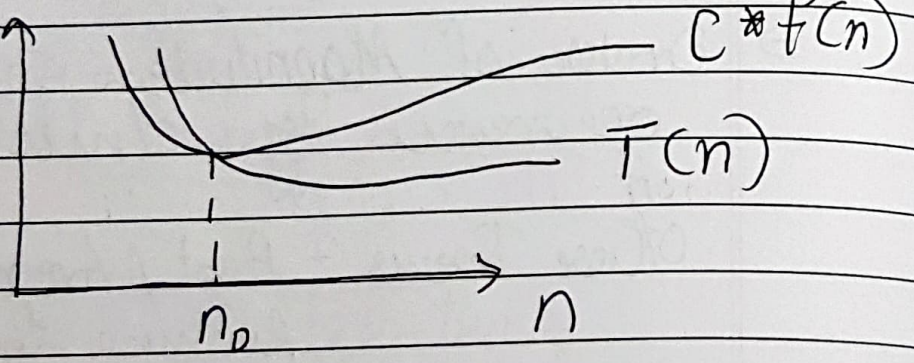
⇒ Asymptotic Notations - How running time of an algo increases w/ size of input (bounded)

i O Notation : Worst Case running time.

For non negative functions $T(n)$ and $f(n)$

$T(n) = O(f(n))$ if there are positive constants c, n_0 such that

$T(n) \leq c * f(n)$ for all n, $n \geq n_0$ ($c > 0$ $n_0 \geq 1$)

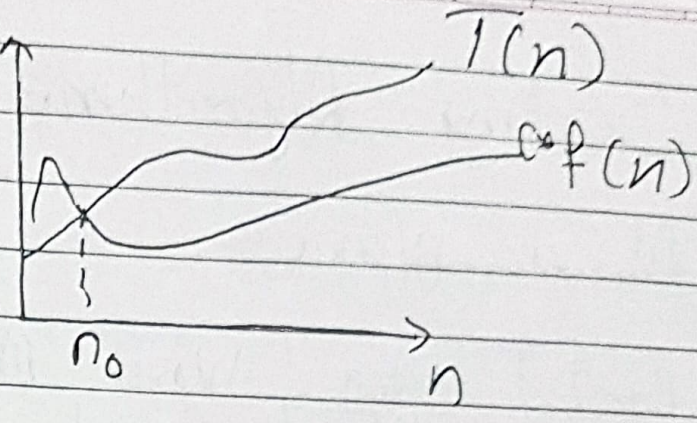


$f(n)$ is the uppere bound

ii Ω Notation :- Best Case

$T(n) = \Omega(f(n))$ if there is c and n_0 such that $T(n) \geq c * f(n)$ for all n, $n \geq n_0$ ($c > 0$, $n_0 \geq 1$)

$f(n)$ is the lower bound



(iii) Θ notation - Average Case
$T(n) = \Theta(g(n))$ if there is ~~some~~ c_1, c_2, n_0 such that

$$c_1 * g(n) \leq T(n) \leq c_2 * g(n)$$

for all n, $n \geq n_0$
$(c_1, c_2 > 0, n_0 \geq 1)$

→ Algorithm of two types
① Iterative - Count no. of times instructions are executed

② Recursive - Recurrence equation

2. Sorting Algorithms

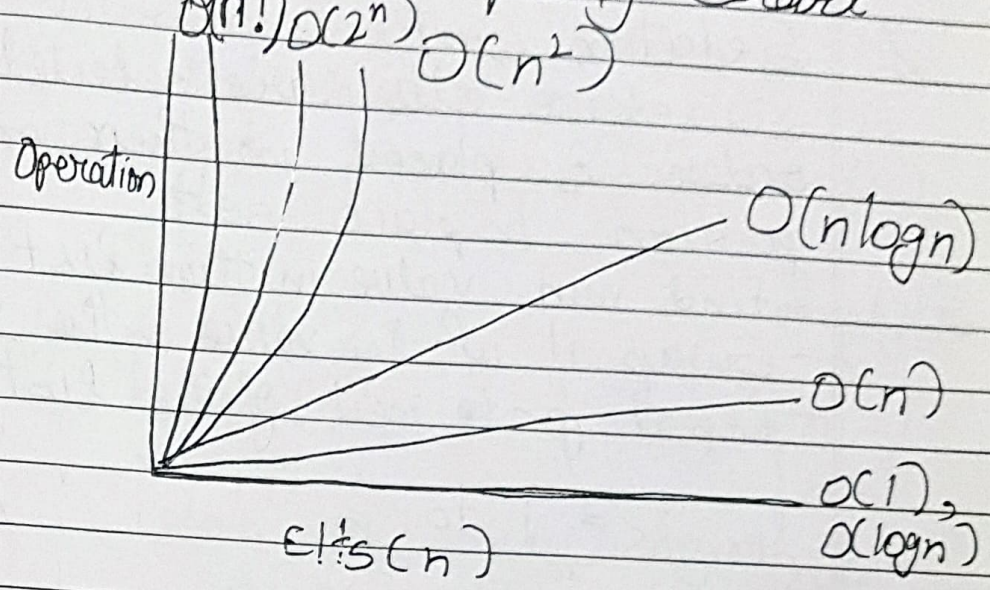
At a glance

Name (Sort)	Best Case	Avg Case	Worst Case	Memory	Method	Stable
Quick	$n \log n$	$n \log n$	n^2	$\log n$	Partition	N
Merge	$n \log n$	$n \log n$	$n \log n$	n	Merge	Y
Heap	$n \log n$	$n \log n$	$n \log n$	1	Selection	N
Insertion	n	n^2	n^2	1	Insertion	Y
Selection	n^2	n^2	n^2	1	Selection	N
Bubble	n	n^2	n^2	1	Exchange	Y
Shell	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	Insertion	N
Tree	$n \log n$	$n \log n$	$n \log n$	n	Insertion	Y
Cycle	n^2	n^2	1.2^2	1	Selection	N
TimSort ↑ Used by Python	n	$n \log n$	$n \log n$	n	Merge+ Insertion	Y

TimSort is hybrid of merge + insertion sort

PAGE NO.
DATE / /

Big O Complexity Chart



1. Bubble Sort

Simplest algo. Repeatedly swaps adjacent elts if they are in wrong order.

Not suitable for large datasets.

```
for(i = 0; i < n-1; i++)
    for(j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(arr[j], arr[j+1])
```

Useful for almost sorted datasets.

PAGE No.
DATE / /

2. Selection Sort

Successive elts are selected in order & placed in their propere position. In place sort.

- Find min. value in the list
- Swap it w/ the value in the 1st posn
- Repeat for rest of the list

	Step cost
for i = 1 to n	; n+1
j = i	;
for k = i+1 to n	; n(n+1)/2
if (A[k] < A[j]) then	; n(n+1)/2 - 1
j = k	; n(n+1)/2 - 1
~~swap (A[i], A[j])~~	;
swap (A[i], A[j])	; n

Use when list is small or memory is limited.

PAGE No.
DATE / /

3. Insertion Sort :- Places an unsorted element and place in its right position. (Like cards)

Code	Analysis
for $j = 2$ to n	n
key = A[j]	$n-1$
$i = j-1$	$n-1$
while $i > 0$ and $A[i] > key$	$\sum_{j=2}^{n} t_i$
$A[i+1] = A[i]$	$n-1$
$i = i-1$	$n-1$
$A[i+1] = key$	$n-1$

When list is sorted (nearly sorted.
Space complexity of O(1)

3. Merge Sort (Divide & Conquer)
Divides input array in half repeatedly till only one elt left, and merges the two sorted halves.

```
Merge_Sort (A, low, high)
  if (low < high)
      mid = (low + high)/2
      Merge_Sort (A, low, mid)
      Merge_Sort (A, mid+1, high)
      Merge (A, low, mid, high)
```

PAGE No.
DATE / /

```
Merge (A, low, mid, high)
  h = low, i = low, j = mid+1
  Auxiliary array B[]
  while (h ≤ mid & j ≤ high)
      if (A[h] ≤ A[j])
          B[i] = A[h]
          h = h+1
      else
          B[i] = A[j]
          j = j+1
      i = i+1
  if (h > mid)
      for k = j to high
          B[i] = A[k]
          i = i+1
  else
      for k = h to mid
          B[i] = A[k]
          i = i+1
  for k = low to high
      A[k] = B[k]
```

Uses:- When data structure doesn't allow random access. (linked list ex)
Used in external sorting.
Sequential access

5 Quicle Sort (Divide & Conquer)
Fastest known Sorting algo
Aka Partition Exchange Sort.
No merge step involved.

```
Quicksort (A, lb, ub)
  if (lb < ub)
      pivot = Partition (A, lb, ub)
      Quicksort (A, lb, pivot-1)
      Quicksort (A, pivot+1, ub)

Partition (A, lb, ub)
   x = A[lb]
   up = ub
   down = lb
   while (down < up)
       while (A[down] <= x & down < ub)
           down = down+1
       while( A[up] > x)
           up = up - 1
       if (down < up)
           Swap( A[down, A[up])
   A[lb] = A[up]
   A[up] = x
   Return up
```

x = 40

Ex:- 40 | 20 | 10 | 80 | 60

down ↓ up ↓

40 | 20 | 10 | 80 | 60

down up

40 | 20 | 10 | 80 | 60

down up

40 | 20 | 10 | 80 | 60

up up

After while partition :-

40 | 20 | 10 | 80 | 60

up down

A[lb] = A[up], A[up] = x

10	20	40	80	60

Partition will returns 3 as o/p

When to use :-

More effective when data fits in memory. It is an in-place sort.

Worst performance on sorted array.

Solution :- Randomized quicksort.

- Choose pivot randomly.
- Make random shuffling.

5. Searching

1. Linear Search

```
Linear-search (se3, A)
    for (i = i to n)
        if (A[i] = x)
            return i
    return null
```

Time Complexity :- O(n) [Avg, Worst]
Best :- O(1)

2. Binary Search (Divide & Conquer)

```
Bin_search (A, n, target)
    left = 1
    right = n
    while (left ≤ right)
        mid = floor((left + right)/2)
        if (target = A[mid])
            return mid
        else if (x > A[mid])
            low = mid + 1
        else
            high = mid - 1
```

Time Complexity Avg/Worse = O(logn)

PAGE No.
DATE / /

5. Approximating Time Complexity

⇒ Iterative :- Count number of times instructions are executed

Ex:- for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      ...

Time Complexity = $O(n^3)$

for i = 1 to n n
...
for i = 1 to n n

Total = n + n = 2n
Time Complexity = O(n)

⇒ Recursive Algorithms :-
Three methods :-
(i) Recursion Relation / Substitution
Ex:- A(n)
  if (n>1)
    return A(n/2) + A(n/2)

Time $T(n) = c + 2T(n/2)$

If return was n-1, it would be
$$T(n) = c + 2T(n-1)$$

* Some Common Recurrences

Recurrence	Solution
$T(n)=T(n/2)+d$	$T(n)=O(\log_2 n)$
$T(n/2)+n$	$O(n)$
$2T(n/2)+d$	$O(n)$
$2T(n/2)+n$	$O(n\log n)$
$T(n-1)+d$	$O(n)$

(ii) Master's Theorem :- For divide & conquer algo;
if problem of size n is divided into 'a' problems of size n/b
Let cost of each stage to divide & conquer be $f(n)$
Then, according to Master's theorem
If $T(n) = aT(n/b) + f(n)$
$a \geq 1$ $b \geq 1$ then,

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}) \\ & \text{and } af(n/b) < cf(n) \end{cases}$$

$\epsilon > 0$, $c < 1$

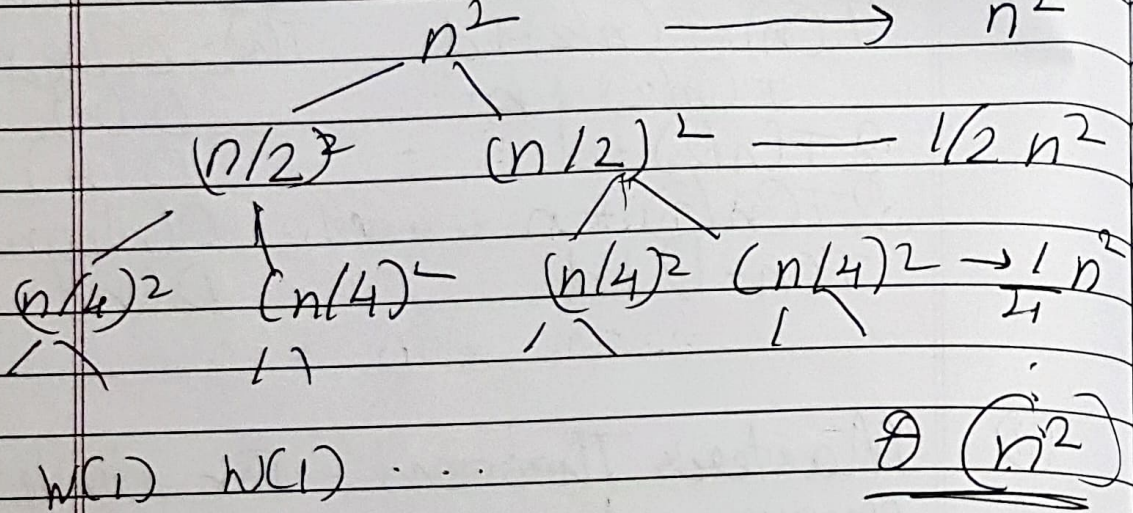
(iii) Recurrence Tree

Node - Cost at various levels

Sum up cost of all levels.

Ex :-

$W(n) = 2W(n/2) + n^2$



$\Theta(n^2)$

PAGE No.
DATE / /

5. DIVIDE & CONQUER

Divide problem to subproblems
Solve Subproblem - Conquer
Combine to get solution

⇒ Previous examples :-
① - Merge Sort
② - Quick Sort
③ - Binary Search

④ Strassen's Matrix Multiplication

Standard Matrix Multiplication :- $O(n^3)$
Strassens - " - : $O(n^{\log 7})$

where n is from $n \times n$ matrix

```
MatMul(A, B, C, n)
  if n = 1
      C = C + A * B
  else
      MatMul(A, B, C, n/4)
      MatMul(A, (B+n/4), (C+n/4), n/4)
      MatMul(A+2x(n/4), B, C+2xn/4, n/4)
      MatMul(A+2x(n/4), B+n/4, C+3xn/4, n/4)
      MatMul(A+(n/4), B+2x(n/4), C, n/4)
      MatMul(A+(n/4), B+3x(n/4), C+(n/4), n/4)
      MatMul(A+3x(n/4), B+2x(n/4), C+2x(n/4), n/4)
      MatMul(A+3x(n/4), B+3x(n/4), C+3x(n/4), n/4)
```

There are 18 $n/2 \times n/2$ matrix add^n / sub^n $O(n^2)$
7 $n/2 \times n/2$ multiplications
$T(n) = 7T(n/2) + O(n^2)$
$= O(n^{2.81})$
$T(n) = O(n^{\log 7})$

(5) Max & Min in a list
- Divide problems into $n/2$ subproblems & find min and max recursively.
Time Complexity = $O(n)$

```
MaxMin(i, y, max, min)
if i = y
   max = min = A[i]
else if (i = y - 1)
     if (A[i] < A[y])
        min = A[i]; max = A[y]
     else
        min = A[y]; max = A[i]
else
   mid = floor((i+y)/2)
   MaxMin(i, mid, max, min)
   MaxMin(mid+1, y, max1, min1)
   if (max < max1)
        max = max1
   if (min > min1)
        min = min1
```

PAGE No.
DATE / /

6. DYNAMIC PROGRAMMING

Method used when problem to be solved is a sequence of decisions. Final solution by combining solution to sequence of subproblems.

Steps :-
- Characterize structure of optimal sol^n
- Recursively define value of optimal sol^n
- Compute value of optimal sol^n, typically in a bottom-up fashion.
- Construct optimal sol^n from computed info.

(1) All Pairs Shortest Path - Floyd-Warshall Algorithm

A graph G(V, E), find shortest path between every pair of vertices

w(i,j) = 0 if i=j
w(i,j) = ∞ if no connecting edge
w(i,j) = wt of edge.

No of iterations = no of vertices

Algorithm
for
$n \leftarrow$ rows(W) ← No of vertices
$D(0) \leftarrow W$ # Input
for $k \leftarrow 1$ to n
  for $i \leftarrow 1$ to n
    for $j \leftarrow 1$ to n
      $D_{ij}^{k} \leftarrow \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$

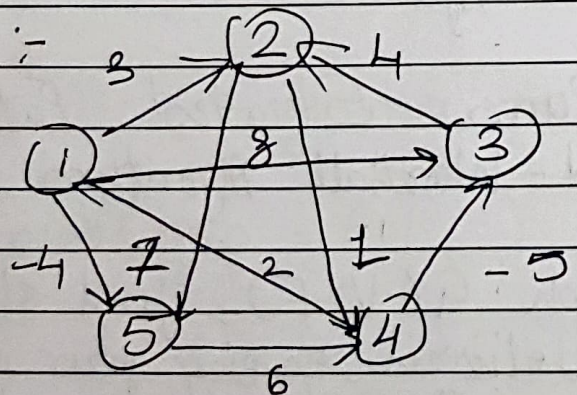
return D

Ex:-

$D_0 = W_0$

Time complexity $= O(n^3)$

Ex:-



Input- $D_0 = W_0$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

PAGE No.
DATE 1 1

Iteration 1 :- D_1 → Distance from to shortest all nodes via node 1

$$D^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Expl:-

i = 4, j = 2 To go from node 4 to 2 via 1, total cost
= min(∞, 2 + 3 = 5) = 5

Similarly i = 4, j = 3
Wt = min(−5, 2+8=10) = −5

Iteration k = 2

$$D^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$