UNIT-II

Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of n characters called the ***text*** and a string of m characters $(m \leq n)$ called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that t_i

$= p_0, \ldots, t_{i+j}$

$= p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1}$ text T

– – –

$p_0 \ldots p_j \ldots p_{m-1}$ pattern P

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

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

Thus, in the worst case, the algorithm makes $m(n - m + 1)$ character comparisons, which puts it in the$O(nm)$ class

Write the algorithm to perform Binary Search and compute its time complexity. Or Explain binary search algorithm with an example

BINARY SEARCH ALGORITHM

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop [successful search];
otherwise, continue searching by the same
method in A[0..m-1] if K < A[m] and in
A[m+1..n-1] if K > A[m]
//Input: An Array A[0…n-1] sorted in ascending orger 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 = 0; r=□ n-1
while l < r do
m =l+r]/2
if K = A[m] return m
else if K < A[m] r □=m-1
else l = m+1
return -1

Time complexity:

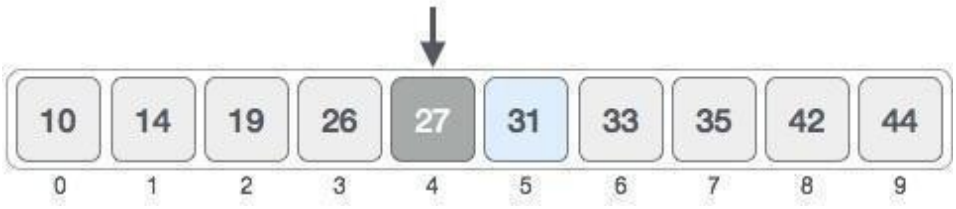
CWorst[n]=1,Cavg[n]=log2n,Cbest[n]=log2n+1

For Example

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + [high - low] / 2
Here it is, 0 + [9 - 0] / 2 = 4 [integer value of 4.5]. So, 4 is the mid of the array.

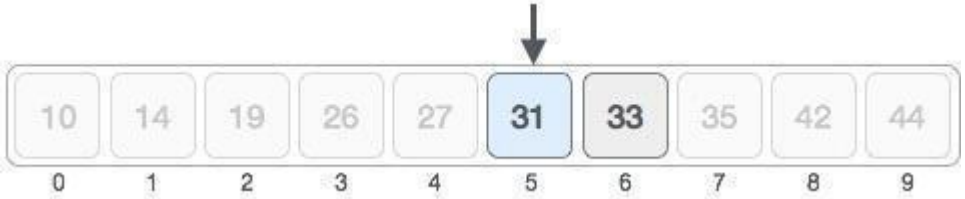


Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array,

to change our low to mid + 1 and find the new mid value again. low = mid + 1, mid = low + [high - low] / 2 Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Write down the algorithm to construct a convex hull based on divide and conquer strategy. Or Explain the convex hull problem and the solution involved behind it

CONVEX HULL OR QUICK HULL PROBLEM

Convex hull: smallest convex set that includes given points. An O[n^3] brute force time. Assume points are sorted by *x*-coordinate values

- Identify *extreme points* *P*1 and *P*2 [leftmost and rightmost]
- Compute *upper hull* recursively:
    1. find point *P*max that is farthest away from line *P*1*P*2
    2. compute the upper hull of the points to the left of line *P*1*P*max
    3. compute the upper hull of the points to the left of line *P*max*P*2
- Compute *lower hull* in a similar manner
- Finding point farthest away from line *P*1*P*2 can be done in linear time
- Time efficiency: T[n] = T[x] + T[y] + T[z] + T[v] + O[n], where x + y + z +v <= n.

  worst case: Θ[*n2*] T[n] = T[n-1] + O[n]

  average case: Θ[*n*]

If points are not initially sorted by *x*-coordinate value, this can be accomplished in O[*n* log *n*] time.

Several O[*n* log *n*] algorithms for convex hull are known.

CONVEX HULL THEOREM The convex hull of any set S of n>2 points not all on the same line is a convex polygon with the vertices at some of the points of S. [If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of S.]

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points.

To solve it, we need to find the points that will serve as the vertices of the polygon in question.

Mathematicians call the vertices of such a polygon —extreme points.‖

CLOSEST PAIR PROBLEM

1. Sort the points by x [list one] and then by y [list two].
2. Divide the points given into two subsets S1 and S2 by a vertical line $x = c$ so that

half the points lie
to the left or on the line and half the points lie to the right or on the line.

3. Find recursively the closest pairs for the left and right subsets.
4. Set $d = \min\{d1, d2\}$, We can limit our attention to the points in the symmetric vertical strip of width $2d$
   as possible closest pair. Let C1and C2 be the subsets of points in the left subset S1 and of the right subset S2, respectively, that lie in this vertical strip. The points in C1 and C2 are stored in increasing order of their y coordinates, taken from the second list.
5. For every point $P[x,y]$ in C1, we inspect points in C2 that may be closer to P than d. There can be no more than 6 such points [because $d \leq d2$]! Running time of the algorithm [without sorting] is: $T[n] = 2T[n/2] + M[n]$, where $M[n] \square \Theta[n]$ By the Master Theorem [with $a = 2$, $b = 2$, $d = 1$] $T[n] \square \Theta[n \log n]$ So the total time is $\Theta[n \log n]$.

CLOSEST-PAIR Problem

Find the two closest points in a set of n points [in the two-dimensional Cartesian plane]. Brute-force algorithm

- Compute the distance between every pair of distinct points
- And return the indexes of the points for which the distance

is the smallest. **ALGORITHM** BruteForceClosestPair[P]
//Finds distance between two closest points in the plane
by brute force //Input: A list P of n [n ≥ 2] points p1[x1,
y1], . . . , pn[xn, yn] //Output: The distance between the
closest pair of points
d←∞
for i ←1 to n −
1 do for j ←i +
1 to n do
d ←min[d, sqrt[[xi− xj]2 + [yi− yj]2]] //sqrt
is square root return d

Develop a pseudo code for divide and conquer algorithm for merge two sorted arrays into a single sorted one – explain with example. or Write down the algorithm for merge sorting. Explain how the following elements get sorted [310,285,179,652,351,423,861,254,450,520] or Sort the following set of elements using

merge sort:12,24,8,71,4,23,6,89,56. Or State and Explain Merge sort algorithm and give the recurrence relation and efficiency.

MERGE SORT

Merge sort 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.

Procedure:

Merge sort 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 then merging the two smaller sorted arrays into a single sorted one.

- Divide Step: If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.
- Recursion Step: Recursively sort array A1 and A2.
- Conquer Step: Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

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]

The non-recursive version of Mergesort starts from merging single elements into sorted pairs. **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←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]

Analysis of Merge sort algorithm

The recurrence relation for the number of key comparisons

C[n] is C[n] = 2C[n/2] + Cmerge[n] for n > 1, C[1] = 0.

In the worst case, Cmerge[n] = n − 1, and we have the

recurrence C worst[n] = 2Cworst[n/2] + n − 1 for n > 1,

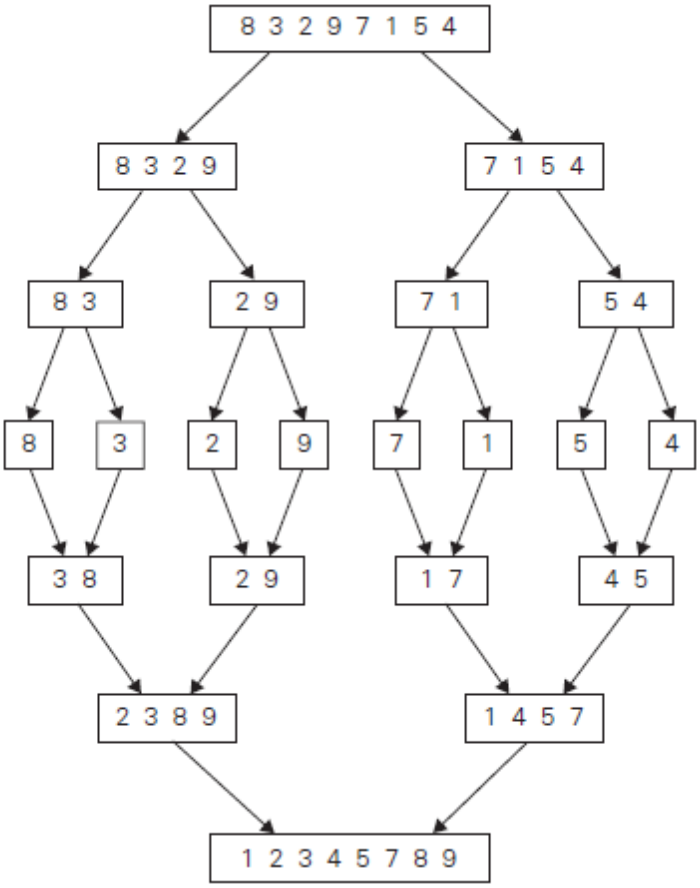
Cworst[1] = 0.

By Master Theorem, Cworst[n] ∈ Θ[n log n] the exact solution to the worst-case recurrence for n = 2 k Cworst[n] = n log2 n − n + 1.

For large n, the number of comparisons made by this algorithm in the average case turns out to be

about 0.25n less and hence is also in Θ [n log n].

For example



QUICK SORT

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. quicksort divides input elements 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:
Sort the two subarrays to the left and to the right of A[s] independently.
No work required to combine the solutions to the subproblems.
Here is pseudocode of quicksort: call Quicksort[A[0..n − 1]] where As a partition algorithm use the HoarePartition

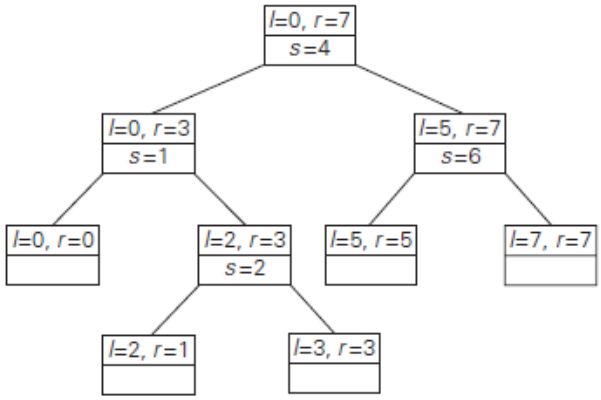
ALGORITHM Quicksort[A[l..r]]
//Sorts a subarray by quicksort

//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]]

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \cdot p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

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



(b)

Time Efficiency analysis

Best case: split in the middle — Θ[*n* log *n*]
Worst case: sorted array! — Θ[*n2*]
Average case: random arrays — Θ[*n* log *n*]

Explain the method used for performing Multiplication of two large integers. Explain how divide and conquer method can be used to solve the same.

Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern

computer, they require special treatment.In the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each ofthe n digits of the first number is multiplied by each of the n digits of the second number for thetotal of n2 digit multiplications.
The divide-and-conquer method does the above multiplication in less than n2 digit multiplications. For any pair of two-digit numbers $a = a1a0$ and $b = b1b0$, their product c can be

computed by the formula $c = a * b = c2102 + c1101 + c0$, where $c2 = a1 * b1$ is the product of their first digits,
$c0 = a0 * b0$ is the product of their second digits,
$c1 = [a1 + a0] * [b1 + b0] - [c2 + c0]$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of $c2$ and $c0$.

$c = a * b = [a110n/2 + a0] * [b110n/2 + b0] = [a1 * b1]10n + [a1 * b0 + a0 * b1]10n/2 + [a0 * b0]$

$= c210n + c110n/2 + c0$, where

$c2 = a1 * b1$ is the product of their first halves,
$c0 = a0 * b0$ is the product of their second halves,
$c1 = [a1 + a0] * [b1 + b0] - [c2 + c0]$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of $c2$ and $c0$.

Analysis of Time Complexity: By using Master Theorem, we obtain $A[n] \in \Theta[n\log 23]$,

Example: For instance: a = 2345, b = 6137, i.e., n=4.

Then C = a * b = [23*102+45]*[61*102+37]

$C = a * b = [a110n/2 + a0] * [b110n/2 + b0]$

$= [a1 * b1]10n + [a1 * b0 + a0 * b1]10n/2 + [a0 * b0]$

= [23 * 61]104 + [23 * 37 + 45 * 61]102 + [45 * 37]

= 1403•104 + 3596•102 + 1665

= 14391265

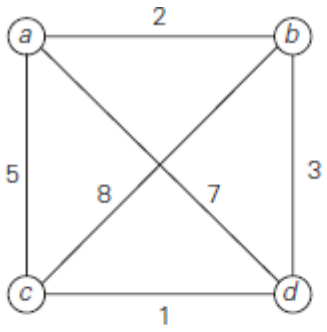
Find all the solution to the traveling salesman problem [cities and distance shown below] by exhaustive search. Give the optimal solutions. Or Explain exhaustive searching techniques with example. Or Find the optimal solution to the fractional knapsack problem with example. Or Solve the given knapsack problem un=3,m=20,[p1,p2,p3]=[25,24,15],[w1,w2,w3]=[18,15,10][M-15][N-14][M-14][N-15][M-16]

TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem [TSP]*** is one of the combinatorial problems. 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 bestated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.
[A Hamiltoniancircuit is defined as a cycle that passes through all the vertices of the graph exactly once]. A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex.

Figure presents a small instance of the problem and its solution by this method.

For example,



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

FIGURE Solution to a small instance of the traveling salesman problem by exhaustive search. **Time Complexity of TSP:O[n-1!]**

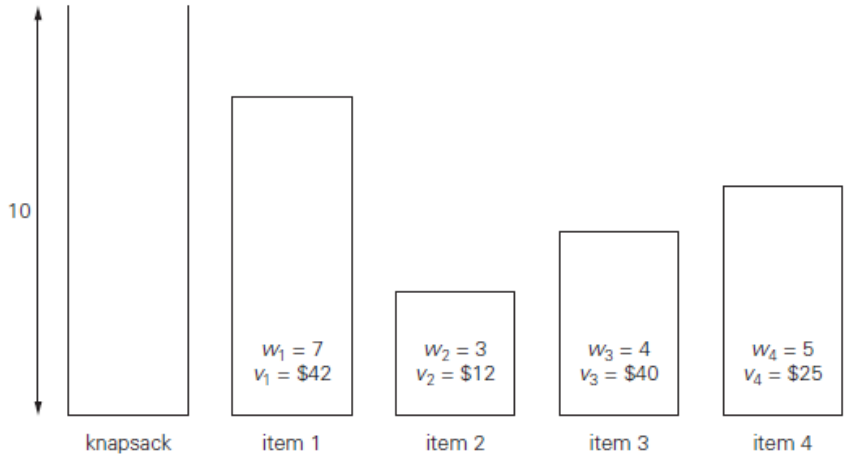
KNAPSACK PROBLEM

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

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without
  exceeding the plane‘s capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets [i.e., the ones with the total weight not exceeding the knapsack capacity], and finding a subset of the largest value among them.

- weights: $w1$ $w2$ ... wn
- values: $v1$ $v2$ ... vn
- a knapsack of capacity W



(a)

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

(b)

IGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

ASSIGNMENT PROBLEM

There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is C[i,j]. Find an assignment that minimizes the total cost. For Example,

A small instance of this problem follows, with the table entries representing he assignment costs $C[i, j]$:

	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

We can describe feasible solutions to the assignment problem as *n*-tuples _*j*1, . . . , *jn* in which the *i*th component, $i = 1, \ldots, n$, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person). For example, for the cost matrix above, _2, 3, 4, 1_ indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

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

<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
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.