Name – Vaishnavi Desai
Roll no.- 322008
Gr no.– 22010870
Div – B
Batch: B1
Subject: DAA

Assignment No.1

Aim: Implement Quick Sort (and Merge Sort) using divide and conquer strategy. And discuss their analysis with respect to Time and Space Complexity.

Objectives:

To study and implement Merge sort algorithm.
To study and implement Quick sort algorithm.
To analyze time and step count in both algorithms.

Theory:

Quick Sort: As the name suggests, this algorithm is quick or fast in terms of speed. This algorithm is based on the divide and conquer approach of programming. The divide and conquer approach involve dividing a task into small atomic sub-tasks and then solving those subtasks. The results of those subtasks are then combined and evaluated to get the final result. In the quick sort algorithm, we select a pivot element and we position the pivot in such a manner that all the elements smaller than the pivot element are to its left and all the elements greater than the pivot are to its right. The elements to the left and right of the pivot form two separate arrays. Now the left and right subarrays are sorted using this same approach. This process continues until each subarray consists of a single element. When this situation occurs, the array is now sorted and we can merge the elements to get the required array. There are 4 common ways of selecting a pivot. One can use any one of the following methods:

1. Pick the first element.
2. Pick the last element.
3. Pick a random element.
4. Pick the median element.

Merge Sort: The merge sort algorithm is an algorithm based on the divide and conquers strategy of solving problems. In this, we divide a problem into several sub-problems, each of which is solved individually, and after that, all of them are combined to give the final solution

Algorithm:
Let's first look at the approach we will follow, and then we will jump right into the code.

- We are given an array arr of size n.
- Check, if n == 0, i.e., we have an empty array, then we have no element to sort.
- If n == 1, ie. there is only one element in the array, then the array is already sorted
- If n > 1, then initialize variables left and right as left = 0 and right = n-1.
- Find middle = (left + right)/2.
- Call mergeSort(arr, left, middle) to recursively sort the first half of the array.
- Similarly, call mergeSort(arr, middle+1, right) to recursively sort the other half of the array.
- Finally call merge(arr, left, middle, right) to merge the obtained sorted arrays.
- Now, the given array is sorted.

Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
int stcnt = 0;
int stcnt1 = 0;

void swap(int arr[] , int pos1, int pos2){
	int temp;
	temp = arr[pos1];
	arr[pos1] = arr[pos2];
	arr[pos2] = temp;
stcnt+=4;

}

int partition(int arr[], int low, int high, int pivot){
	int i = low;
stcnt++;
	int j = low;
stcnt++;
	while( i <= high){
		if(arr[i] > pivot){
			i++;
stcnt++;
		}
		else{
			swap(arr,i,j);
			i++;
			j++;
stcnt+=2;
		}
```

```
        }
        return j-1;
}

void quickSort(int arr[], int low, int high){
        if(low < high){
        int pivot = arr[high];
stcnt++;
        int pos = partition(arr, low, high, pivot);
stcnt+=2;

        quickSort(arr, low, pos-1);
        quickSort(arr, pos+1, high);
        }
}
void merge(int arr[], int l, int m, int r) {
   int i, j, k, nl, nr;
   nl = m-l+1; nr = r-m;
stcnt1++;
   int larr[nl], rarr[nr];
stcnt1++;
   for(i = 0; i<nl; i++)
      larr[i] = arr[l+i];
stcnt1++;
   for(j = 0; j<nr; j++)
      rarr[j] = arr[m+1+j];
stcnt1++;
   i = 0; j = 0; k = l;
   while(i < nl && j<nr) {
      if(larr[i] <= rarr[j]) {
         arr[k] = larr[i];
         i++;
stcnt1++;
      }e1lse{
         arr[k] = rarr[j];
         j++;
stcnt1++;
      }
      k++;
stcnt1++
   }
   while(i<nl) {
      arr[k] = larr[i];
      i++; k++;
stcnt1++;
   }
   while(j<nr) {
      arr[k] = rarr[j];
      j++; k++;
stcnt+=2;
   }
}
void mergeSort(int arr[], int l, int r) {
```

```
   int m;
   if(l < r) {
      int m = l+(r-l)/2;

      mergeSort(arr, l, m);
      mergeSort(arr, m+1, r);
      merge(arr, l, m, r);
   }
}


int main()
{
int n;

  cout << "Enter the size" << endl;

  cin >> n;


int arr[n];

  int arr1[n];

for (int i = 0; i < n; i++)

    {

      arr[i] = rand () % 10;

}

for (int i = 0; i < n; i++)

    {

      arr1[i] = arr[i];

}

cout << "\nQuick Sort : " << endl;

  cout << "Random Array : " << endl;

for (int i = 0; i < n; i++)

    {

      cout << arr[i] << " ";

}

auto st = chrono::high_resolution_clock::now ();

quickSort (arr, 0, n - 1);

auto end = chrono::high_resolution_clock::now ();
```

```
  cout << "\nTime Average Case  : " << endl;
  cout << chrono::duration_cast < chrono::microseconds >
    (end - st).count () << " microseconds" << endl;

auto st1 = chrono::high_resolution_clock::now ();
quickSort (arr, 0, n - 1);
auto end1 = chrono::high_resolution_clock::now ();
  cout << "\nQuick Sort Time Worst Case  : " << endl;
  cout << chrono::duration_cast < chrono::microseconds >
    (end1 - st1).count () <<
" microseconds" << endl;
  cout << "\nStep count : " << ct << endl;
  cout << "Array after sorting" << endl;
  for (int i = 0; i < n; i++)
    {
      cout << arr[i] << " ";
}
cout << "\n\n\n\n\nMerge Sort : " << endl;
  cout << "Random Array : " << endl;
  for (int i = 0; i < n; i++)
    {
      cout << arr1[i] << " ";
} auto st2 = chrono::high_resolution_clock::now ();
mergeSort (arr1, 0, n - 1);
auto end2 = chrono::high_resolution_clock::now ();
  cout << "\nAverage Case  : " << endl;

cout << chrono::duration_cast < chrono::microseconds >
    (end2 - st2).count () <<
" microseconds" << endl;
auto st3 = chrono::high_resolution_clock::now ();
```

```
mergeSort (arr1, 0, n - 1);
  auto end3 = chrono::high_resolution_clock::now ();
  cout << "\nTime worst Case  : " << endl;
  cout << chrono::duration_cast < chrono::microseconds >
    (end3 - st3).count () << " microseconds" << endl;
cout << "\nStep count Merge Sort  : " << ct1 << endl;
  cout << "\nArray after sorting : " << endl;
  for (int i = 0; i < n; i++)
    {
      cout << arr1[i] << " ";
}
return 0;
}
}
```

Output :

```
Enter the size
500
Quick Sort :
Random Array :
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 9 2 2 8 9 7 3 6 1 2 9 3
    1 9 4 7 8 4 5 0 3 6 1 0 6 3 2 0 6 1 5 5 4 7 6 5 6 9 3 7 4 5 2 5 4 7 4 4 3 0 7 8
    6 8 8 4 3 1 4 9 2 0 6 8 9 2 6 6 4 9 5 0 4 8 7 1 7 2 7 2 2 6 1 0 6 1 5 9 4 9 0 9
    1 7 7 1 1 5 9 7 7 6 7 3 6 5 6 3 9 4 8 1 2 9 3 9 0 8 8 5 0 9 6 3 8 5 6 1 1 5 9 8
    4 8 1 0 3 0 4 4 4 4 7 6 3 1 7 5 9 6 2 1 7 8 5 7 4 1 8 5 9 7 5 3 8 8 3 1 8 9 6 4
    3 3 3 8 6 0 4 8 8 8 9 7 7 6 4 3 0 3 0 9 2 5 4 0 5 9 4 6 9 2 2 4 7 7 5 4 8 1 2 8
    9 3 6 8 0 2 1 0 5 1 1 0 8 5 0 6 4 6 2 5 8 6 2 8 4 7 2 4 0 6 2 9 9 0 8 1 3 1 1 0
    3 4 0 3 9 1 9 6 9 3 3 8 0 5 6 6 4 0 0 4 6 2 6 7 5 6 9 8 7 2 8 2 9 9 6 0 2 7 6 1
    3 2 1 5 9 9 1 4 9 1 0 7 5 8 7 0 4 8 0 4 2 9 6 1 0 4 2 2 2 0 5 5 2 9 0 2 8 3 8 0
    4 0 9 1 9 6 2 5 4 4 9 9 3 6 0 5 0 2 9 4 3 5 1 7 4 3 1 4 6 9 4 2 2 6 4 1 2 8 8 9
    2 8 8 8 6 8 3 8 3 3 3 8 0 4 7 6 8 9 0 6 8 7 9 0 3 3 3 7 3 2 6 5 2 6 5 8 7 9 6 0
    4 1 0 4 8 7 0 8 6 2 4 7 9 3 9 2 8 3 0 1 7 8 9 1 5 4 9 2 5 7 4 9 9 4 5 9 3 5 7 0
    8 1 9 9 7 8 2 5 3 4 9 0 2 0 1 9 6 2
Time Average Case  :
45 microseconds
```

```
Quick Sort Time Worst Case  :
51 microseconds

Step count : 52207
Array after sorting
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
    2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
    3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4
    4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
    4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
    5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
    6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
    7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
    8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

```
Merge Sort :
Random Array :
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 9 2 2 8 9 7 3 6 1 2 9 3
    1 9 4 7 8 4 5 0 3 6 1 0 6 3 2 0 6 1 5 5 4 7 6 5 6 9 3 7 4 5 2 5 4 7 4 4 3 0 7 8
    6 8 8 4 3 1 4 9 2 0 6 8 9 2 6 6 4 9 5 0 4 8 7 1 7 2 7 2 2 6 1 0 6 1 5 9 4 9 0 9
    1 7 7 1 1 5 9 7 7 6 7 3 6 5 6 3 9 4 8 1 2 9 3 9 0 8 8 5 0 9 6 3 8 5 6 1 1 5 9 8
    4 8 1 0 3 0 4 4 4 4 7 6 3 1 7 5 9 6 2 1 7 8 5 7 4 1 8 5 9 7 5 3 8 8 3 1 8 9 6 4
    3 3 3 8 6 0 4 8 8 8 9 7 7 6 4 3 0 3 0 9 2 5 4 0 5 9 4 6 9 2 2 4 7 7 5 4 8 1 2 8
    9 3 6 8 0 2 1 0 5 1 1 0 8 5 0 6 4 6 2 5 8 6 2 8 4 7 2 4 0 6 2 9 9 0 8 1 3 1 1 0
    3 4 0 3 9 1 9 6 9 3 3 8 0 5 6 6 4 0 0 4 6 2 6 7 5 6 9 8 7 2 8 2 9 9 6 0 2 7 6 1
    3 2 1 5 9 9 1 4 9 1 0 7 5 8 7 0 4 8 0 4 2 9 6 1 0 4 2 2 2 0 5 5 2 9 0 2 8 3 8 0
    4 0 9 1 9 6 2 5 4 4 9 9 3 6 0 5 0 2 9 4 3 5 1 7 4 3 1 4 6 9 4 2 2 6 4 1 2 8 8 9
    2 8 8 8 6 8 3 8 3 3 3 8 0 4 7 6 8 9 0 6 8 7 9 0 3 3 3 7 3 2 6 5 2 6 5 8 7 9 6 0
    4 1 0 4 8 7 0 8 6 2 4 7 9 3 9 2 8 3 0 1 7 8 9 1 5 4 9 2 5 7 4 9 9 4 5 9 3 5 7 0
    8 1 9 9 7 8 2 5 3 4 9 0 2 0 1 9 6 2
Average Case  :
81 microseconds
```

```
Time worst Case  :
47 microseconds

Step count Merge Sort  : 40900

Array after sorting :
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
    2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
    3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4
    4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
    4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
    5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
    6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
    7 7 7 7 7 7 7 7 77 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
    8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

Discussion:

Sorting Method

- **Merge sort** is an external sorting method in which the data that is to be sorted can be stored outside the memory and is loaded in small chunks into the memory for sorting.
- **Quicksort** is an internal sorting method, where the data that is to be sorted needs to be stored in the main memory throughout the sorting process.

Worst-Case Complexity

- **Merge sort** performs the same number of comparison and assignment operations for an array of a particular size. Therefore, its worst-case time complexity is the same as best-case and average-case time complexity, that is, O(n log n).
- In **quicksort**, as we've already discussed, the choice of the pivot plays an important role. Let's suppose we always choose the rightmost element of a list to be the pivot and the input array is reverse-sorted. The partitions created will be highly unbalanced (of sizes 0 and (n - 1) for a list of size n); that is, the sizes of the partitions differ a lot. This results in the worst-case time complexity of O(n2).

Speed

- **Merge sort** generally performs fewer comparisons than quicksort both in the worst-case and on average. If performing a comparison is costly, merge sort will have the upper hand in terms of speed.
- **Quicksort** is generally believed to be faster in common real-life settings. This is mainly due to its lower memory consumption which usually affects time performance as well.

Space Requirement

- **Merge sort** requires the creation of two subarrays in addition to the original array. This is necessary for the recursive calls to work correctly. Consequently, the algorithm must create n elements in memory. Thus, the space complexity is O(n). Merge-sort can be made in place, but all such algorithms have a higher time complexity than O(n log n).
- **Quicksort** is an in-place sorting algorithm. Its memory complexity is O(1).

Stability

- **Merge sort** is a stable sorting algorithm, i.e., it maintains the relative order of two equal elements.
- **Quicksort** is an unstable sorting algorithm, i.e., it might change the relative order of two equal elements.

Conclusion:

In this assignment, we implemented quick sort and merge sort. When we compared and analyzed the time used and the total number of steps, we discovered that quick sort's time complexity is O(nlog(n)) on average and O(n2) in its worst case, whereas merge sort's time complexity is O(nlog(n) in all cases. Hence, we can draw the conclusion that, on average, the quick sort is faster than the merge sort because it requires less time.

Name : Vaishnavi Desai
Class: TY-B
Batch: B-1
Subject: DAA

Assignment 2

Aim: To implement 01 Knapsack

Objective: To implement 01 Knapsack and compare DP solution and recursion solution.

Theory:
We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Algorithm:

1. Using Recursion:
   The maximum value obtained from ‘N’ items is the max of the following two values.
   - Maximum value obtained by N-1 items and W weight (excluding nth item)
   - Value of nth item plus maximum value obtained by N-1 items and (W – weight of the Nth item) [including Nth item].
   - If the weight of the ‘Nth‘ item is greater than ‘W’, then the Nth item cannot be included and Case 1 is the only possibility.
2. Using DP:
   - In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and the elements that can be kept as rows.
   - The state DP[i][j] will denote the maximum value of ‘j-weight’ considering all values from ‘1 to ith‘. So if we consider ‘wi‘ (weight in ‘ith‘ row) we can fill it in all columns which have ‘weight values > wi‘. Now two possibilities can take place:
   - Fill ‘wi‘ in the given column.
   - Do not fill ‘wi‘ in the given column.

   Now we have to take a maximum of these two possibilities,

- Formally if we do not fill the ‘ith‘ weight in the ‘jth‘ column then the DP[i][j] state will be the same as DP[i-1][j]
- But if we fill the weight, DP[i][j] will be equal to the value of (‘wi‘+ value of the column weighing ‘j-wi’) in the previous row.
- So we take the maximum of these two possibilities to fill the current state.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b) {
    if(a > b)
    return a;
    else return b;
}
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   vector<vector<int> > K(n + 1, vector<int>(W + 1));
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1]
                          + K[i - 1][w - wt[i - 1]],
                      K[i - 1][w]);
         else
            K[i][w] = K[i - 1][w];
      }
   }
   return K[n][W];
}
int knapSackRec(int W, int wt[], int val[], int n)
{
   if (n == 0 || W == 0)
      return 0;
   if (wt[n - 1] > W)
      return knapSack(W, wt, val, n - 1);
   else
      return max(
         val[n - 1]
            + knapSack(W - wt[n - 1], wt, val, n - 1),
         knapSack(W, wt, val, n - 1));
```

```
}
int main()
{
    /*int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);*/
    int W;
    cout<<"Enter weight:";
    cin>>W;
     int n;
     cout<<"\nEnter number of items:";
    cin>>n;
    int val[n];
    cout<<"\nEnter elements in val: ";
    for(int i=0;i<n;i++){
        cin>>val[i];
    }
    int wt[n];
    cout<<"\nEnter elements in wt: ";
    for(int i=0;i<n;i++){
        cin>>wt[i];
    }
     auto st = chrono::high_resolution_clock::now ();
    cout << knapSack(W, wt, val, n)<<endl;
     auto end = chrono::high_resolution_clock::now ();
       cout << "\nTime of DP solution: " << endl;
  cout << chrono::duration_cast < chrono::microseconds >
    (end - st).count () << " microseconds" << endl;
     auto st1 = chrono::high_resolution_clock::now ();
    cout << knapSackRec(W, wt, val, n);
     auto end1 = chrono::high_resolution_clock::now ();
     cout << "\nTime of Recurrsive solution: " << endl;
 cout << chrono::duration_cast < chrono::microseconds >
    (end1 - st1).count () << " microseconds" << endl;
    return 0;
}
```

Output:

```
Enter weight:50
Enter number of items:3
Enter elements in val: 60 100 120
Enter elements in wt: 10 20 30
220

Time of DP solution:
52 microseconds
220
Time of Recurrsive solution:
22 microseconds
```

```
Enter weight:8
Enter number of items:4
Enter elements in val: 3 4 6 5
Enter elements in wt: 2 3 1 4
15

Time of DP solution:
46 microseconds
15
Time of Recurrsive solution:
6 microseconds
```

```
Enter weight:25
Enter number of items:4
Enter elements in val: 24 18 18 10
Enter elements in wt: 24 10 10 7
36

Time of DP solution:
40 microseconds
36
Time of Recurrsive solution:
10 microseconds
```

```
Enter weight:25
Enter number of items:5
Enter elements in val: 24 18 18 10 10
Enter elements in wt: 24 10 10 7 7
38

Time of DP solution:
49 microseconds
38
Time of Recurrsive solution:
11 microseconds
```

```
Enter weight:60
Enter number of items:6
Enter elements in val: 100 280 120 340 210 120
Enter elements in wt: 10 40 20 30 50 60 20
560

Time of DP solution:
44 microseconds
560
Time of Recurrsive solution:
17 microseconds
```

Discussion:
Time Complexity using recursion: O(2N)
Space Complexity using recursion: O(N), Stack space required for recursion
Time Complexity using DP: O(N * W). where ‘N’ is the number of elements and ‘W’ is capacity.
Space Complexity using DP: O(N * W). The use of a 2-D array of size ‘N*W’.
0/1 Knapsack cannot be solved using a greedy approach.Most optimal solution is obtained using Dynamic Programming.

Conclusion: Thus, we implemented 0/1 Knapsack using dynamic programming and recursion. It is observed that time complexity is more for recursion than dynamic programming. The most efficient technique to solve this problem is dynamic programming.