

ACTIVITY SELECTION PROBLEM

A MINI PROJECT REPORT

18CSC204J - Design and Analysis of Algorithms Laboratory

Submitted by

SAMARTH JAIN [Reg No:RA2111003010943]

VIKRAM KALLAKRINDA [Reg No: RA2111003010945]

Under the Guidance of

Dr. Kishore Anthuvan Sahayaraj K

Assistant Professor, Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603203

MAY 2023



COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

Chengalpattu District

BONAFIDE CERTIFICATE

Register Nos. **RA2111003010943, RA2111003010945** Certified to be the bonafide work done by **Samarth Jain** and **Vikram Kallakrinda** of II Year/IV Sem B.Tech Degree Course in the **18CSC204J - Design and Analysis of Algorithms Laboratory** in **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**, Kattankulathur during the academic year 2022 – 2023.

FACULTY INCHARGE

Dr. Kishore Anthuvan Sahayaraj K

Department of Computing Technologies

SRMIST – KTR

Head of the Department

Dr. M. PUSHPALATHA

Department of Computing Technologies

SRMIST – KTR

Contents

1. Contribution table
2. Problem Definition
3. Problem Explanation with diagram and example
4. Design Techniques used (Greedy algorithm & Dynamic Programming)
5. Algorithm for the problem
6. Explanation of algorithm with example
7. Complexity analysis
8. Conclusion
9. References

Contribution Table

Task	Contributor
Problem definition and explanation	Samarth Jain
Design techniques – Greedy solution	Vikram Kallakrinda
Design techniques – Dynamic approach	Samarth Jain
Algorithm for the problem	Samarth Jain
Complexity analysis	Vikram Kallakrinda

Activity Selection Problem

Problem:

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Explanation:

Given a set of activities with start and finish times, select the maximum number of non-overlapping activities that can be performed in a single time slot.

Activity	1	2	3	4	5	6
Start	5	1	0	3	5	8
Finish	9	2	6	4	7	9

List of activities to be performed.


Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9

Step1: Sort the activities in ascending order of Finish times.

Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9


Step2: Select the first activity in the sorted list.

Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9




Step3: Select the next activity in the sorted list its `start` time is greater than or equal to the `finish` time of the previously selected activity.


Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9




Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9



Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9



Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9



Hence, the person can perform 4 non-conflicting activities.

Examples:

Input: start[] = {10, 12, 20}, finish[] = {20, 25, 30}

Output: {10,20}, {20,30}

Solution

Algorithm:

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array and print it
3. Do the following for the remaining activities in the sorted array
4. If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it

Initially : $arr[] = \{\{ 5,9 \}, \{ 1,2 \}, \{ 3,4 \}, \{ 0,6 \}, \{ 5,7 \}, \{ 8,9 \}\}$

Step 1: Sort the array according to finish time
`arr[] = { { 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 } }`

Step 2: Print first activity and make $i = 0$
`print = ({ 1,2 })`

```

Step 3:      arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
               ↑
               j

Start[ j ] >= finish[ i ]. print( { 3,4 } )
make i = j , j++

```

Step 4:

```
arr[] = { { 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 } }
```

↑
j

Start[j] < finish[i]. j++

```

Step 5:      arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
                                     ↑
Start[ j ] >= finish[ i ]. print ( { 5,7 } )
make i = j, j++

```

Step 6:

```
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
```

↑
j

make i = j, j++

Step 6: `arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }`
 `Start[j] < finish[i].`

Greedy Approach:

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct Activity {
    int start, finish;
};
bool activityCompare(Activity s1, Activity s2)
{
    return (s1.finish < s2.finish);
}
void printMaxActivities(Activity arr[], int n)
{
    sort(arr, arr + n, activityCompare);
    cout << "Following activities are selected :\n";
    int i = 0;
    cout << "(" << arr[i].start << ", " << arr[i].finish
        << ")";
    for (int j = 1; j < n; j++) {
        if (arr[j].start >= arr[i].finish) {
            cout << ", (" << arr[j].start << ", "
                << arr[j].finish << ")";
            i = j;
        }
    }
}
int main()
{
    Activity arr[] = { { 5, 9 }, { 1, 2 }, { 3, 4 },
                      { 0, 6 }, { 5, 7 }, { 8, 9 } };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    printMaxActivities(arr, n);
    return 0;
}
```

Time Complexity: $O(N \log N)$, If input activities may not be sorted. It takes $O(n)$ time when it is given that input activities are always sorted.

Space Complexity: $O(1)$

Dynamic Programming Approach:

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

Code:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
struct Job
{
    int start;
    int finish;
};
void findNonConflictingJobs(vector<Job> jobs)
{
    sort(jobs.begin(), jobs.end(),
        [](Job &x, Job &y) {
            return x.start < y.start;
        });
    vector<vector<Job>> L(jobs.size());

    for (int i = 0; i < jobs.size(); i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (jobs[j].finish < jobs[i].start &&
                L[j].size() < L[i].size()) {
                L[i] = L[j];
            }
        }
        L[i].push_back(jobs[i]);
    }
    vector<Job> max;
    for (auto &pair: L)
    {
        if (max.size() < pair.size()) {
            max = pair;
        }
    }
    for (Job &pair: max) {
        cout << "{" << pair.start << ", " << pair.finish << "} ";
    }
}
```

```

    }
}

int main()
{
    vector<Job> jobs =
    {
        {1, 4}, {3, 5}, {0, 6}, {5, 7}, {3, 8}, {5, 9},
        {6, 10}, {8, 11}, {8, 12}, {2, 13}, {12, 14}
    };

    findNonConflictingJobs(jobs);
    return 0;
}

```

Time Complexity: $O(n^2)$

Space Complexity: $O(n)$

Conclusion:

Hence, we can conclude that Greedy approach is a better solution to the activity selection problem as it takes less Time complexity and less Auxiliary space.

References

1. <https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/>
2. <https://www.javatpoint.com/activity-selection-problem>
3. <https://www.educative.io/answers/what-is-the-activity-selection-problem>