Practical: Activity Selection Problem

Student: Dhruv Sachdeva, sachdevadhruv023@ce.du.ac.in

Introduction

The Activity Selection Problem is a classic example of a **Greedy Algorithm**. Given a set of activities with start and finish times, the goal is to select the maximum number of activities that can be performed by a single person or machine, assuming that only one activity can take place at a time.

- **Iterative Approach:** Select activities greedily after sorting by finish time.
- **Recursive Approach:** Use recursion to select the next non-overlapping activity.

Problem 1. *1(a) Implement the Activity Selection problem using an Iterative Greedy approach.*

1. Iterative Activity Selection (Greedy)

Activities are sorted based on their finish time, and the earliest finishing compatible activity is selected next.

Algorithm 1 Iterative Activity Selection

Require: n activities with start time s_i and finish time f_i
Ensure: Maximum number of non-overlapping activities

```
Sort activities in increasing order of f_i
Select the first activity and set i ← 1
for j ← 2 to n do
    if s_j ≥ f_i then
        Select activity j
        i ← j
    end if
end for
```

C Code:

```c
#include <stdio.h>

typedef struct {
    int start;
    int finish;
} Activity;

void sortActivities(Activity activities[], int n) {
    int i, j;
    Activity temp;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (activities[i].finish > activities[j].finish) {
                temp = activities[i];
                activities[i] = activities[j];
                activities[j] = temp;
            }
        }
    }
}

void asp(Activity activities[], int n) {
    int i, j;
    printf("Following activities are selected (Iterative):\n");
    i = 0;
    printf("Activity %d (Start: %d, Finish: %d)\n",
           i + 1, activities[i].start, activities[i].finish);

    for (j = 1; j < n; j++) {
```

```
            if (activities[j].start >= activities[i].finish) {
                printf("Activity %d (Start: %d, Finish: %d)\n",
                       j + 1, activities[j].start, activities[j].finish);
                i = j;
            }
        }
}

int main() {
    int n, i;
    printf("Enter the number of activities: ");
    scanf("%d", &n);
    Activity activities[n];
    printf("Enter start and finish times:\n");
    for (i = 0; i < n; i++) {
        scanf("%d%d", &activities[i].start, &activities[i].finish);
    }
    sortActivities(activities, n);
    asp(activities, n);
    return 0;
}
```

Analysis

$$T(n) = O(n^2) \text{ (due to bubble sort)} \quad \text{or} \quad O(n \log n) \text{ if better sort used.}$$

The greedy algorithm ensures an optimal set of non-overlapping activities.

Problem 2. *1(b) Implement the Activity Selection problem using Recursion.*

2. Recursive Activity Selection

In this version, after sorting, the function recursively selects the next activity that starts after the last chosen one finishes.

Algorithm 2 Recursive Activity Selection

Require: n sorted activities with start time s_i and finish time f_i
Ensure: Maximum number of non-overlapping activities

```
m ← i + 1
while m < n and s_m < f_i do
    m ← m + 1
end while
if m < n then
    Select activity m
    call RecursiveActivitySelection(activities, m, n)
end if
```

C Code:

```c
#include <stdio.h>

typedef struct {
    int start;
    int finish;
} Activity;

void sortActivities(Activity activities[], int n) {
    int i, j;
    Activity temp;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (activities[i].finish > activities[j].finish) {
                temp = activities[i];
                activities[i] = activities[j];
                activities[j] = temp;
            }
        }
    }
}

void rec_asp(Activity activities[], int i, int n) {
    int m = i + 1;
    while (m < n && activities[m].start < activities[i].finish)
        m++;
    if (m < n) {
        printf("Activity %d (Start: %d, Finish: %d)\n",
               m, activities[m].start, activities[m].finish);
        rec_asp(activities, m, n);
    }
```

```
}

int main() {
    int n, i;
    printf("Enter the number of activities: ");
    scanf("%d", &n);
    Activity activities[n + 1];
    activities[0].start = 0;
    activities[0].finish = 0;

    printf("Enter start and finish times:\n");
    for (i = 1; i <= n; i++) {
        scanf("%d%d", &activities[i].start, &activities[i].finish);
    }

    sortActivities(activities + 1, n);
    printf("Following activities are selected (Recursive):\n");
    rec_asp(activities, 0, n + 1);
    return 0;
}
```

Analysis

$T(n) = O(n^2)$ *(due to sorting + recursion),* $S(n) = O(n)$ *(recursion stack)*

Time Complexity Comparison

Iterative vs Recursive:

Approach	***Time Complexity***
Iterative	$O(n^2)$ *(or* $O(n \log n)$ *with efficient sort)*
Recursive	$O(n^2)$ *(same asymptotic bound)*

Conclusion

- *Both iterative and recursive approaches yield the same optimal result set.*
- *The iterative version is generally preferred due to lower space usage.*
- *The recursive version, however, provides a cleaner representation of the greedy choice property.*