Bangabandhu Sheikh Mujibur Rahman
Digital University

# PROJECT REPORT

Prepared by:
MOBASHIA MEHAJABIN ARPITA (08)
MEHRIN FARZANA (13)
SRABONEE PAUL TULI (15)

Date: 07/11/2023

## Project Name: School Management System.

## Table of Contents:

## Introduction:

The School Management System is a comprehensive program that allows school administrators to manage student information, academic queries, and course planning efficiently. This system provides various functionalities, including adding students, displaying student information, mapping courses and career paths, sorting students by ID, searching students by ID, storing course marks and calculating CGPA, taking academic queries during Town Hall Meetings, displaying a list of academic queries, optimizing courses with limited credits, and avoiding time-overlapping courses.

## Objectives:

The main objectives of this project are:

- To create a flexible and efficient system for managing student information.
- To provide tools for academic advisors to assist students in course planning.
- To handle academic queries during Town Hall Meetings.
- To implement data structures and algorithms for efficient data management.

## Project Description:

### Functions:

1. **knapsack01:**

   - **Description:** This function implements the 0-1 Knapsack algorithm to find the maximum credit you can earn by selecting courses with a limited total contact hour.

   - **Parameters:** n (number of courses), C (maximum contact hours), w (array of contact hours per course), v (array of credits per course).

2. **activity_selection:**

   - **Description:** This function uses the greedy approach to select a maximum number of non-overlapping activities or courses.

   - **Parameters:** n (number of activities), a (a 2D array representing start and end times for each activity).

3. **avoidTimeOverlappingCourses:**

   - **Description:** This function allows a user to input course timings and suggests a set of non-overlapping courses to take.

   - **Interaction:** Takes user input for course timings.

4. **merge and merge_sort:**

   - **Description:** These functions implement the Merge Sort algorithm to sort a list of students based on their IDs.

   - **Parameters:** arr (array of student records), l (left index), mid (middle index), and r (right index).

5. **binary_search:**

   - **Description:** Performs a binary search on the sorted student list to find and display student details based on their ID.

   - **Parameters:** arr (sorted array of student records) and n (number of students).

6. **addStudent:**

   - **Description:** Allows the user to add a new student to the system with their name, ID, department, courses, and session.
   - **Interaction:** Takes user input for student information.

7. **displayStudentsInfo:**

   - **Description:** Displays the details of all the students in the system.
   - **Parameters:** studentList (array of student records) and numStudents (number of students).

8. **storeCourseMarks:**

   - **Description:** Allows the user to input marks for each course a student is taking and calculates the student's CGPA.
   - **Interaction:** Takes user input for course marks.

9. **enQueue, deQueue, and display:**

   - **Description:** Implement a basic queue system to enqueue, dequeue, and display academic queries.
   - Functions for managing a list of academic queries.

10. **Academic_query:**

    - **Description:** Provides a user interface for enqueuing and dequeuing academic queries during a "Town Hall Meeting."
    - **Interaction:** Allows users to enqueue and dequeue queries interactively.

11. **maximizeCoursesWithLimitedCredits:**

    - **Description:** Helps a user determine the maximum number of courses that can be taken within a specified limit of contact hours.
    - **Interaction:** Takes user input for course credits and contact hours**.**

**12.floyd_warshall:**

- **Description:** Applies the Floyd-Warshall algorithm to optimize the graph of course-career paths.
- **Interaction:** Takes user input to build the adjacency matrix representing course-career relationships.

## Data Structures Used:

1. **Arrays:** Arrays are used to store course names, student names, student department, and various other data in this program. They provide a way to store and manage data in a structured manner.
2. **Struct:** The struct data structure is used to create a composite data type Student to store information about students. It allows grouping multiple data elements of different types into a single unit.
3. **Queue:** A queue data structure is implemented to manage academic queries in the "Town Hall Meeting Academic Query System" (enQueue, deQueue operations). It ensures queries are processed in a first-in, first-out (FIFO) order.
4. **Dynamic Array (realloc):** Dynamic arrays are used to store a list of students, and the realloc function is used to dynamically allocate and resize the memory for the student list as new students are added.
5. **2D Array:** A 2D array is used to represent an adjacency matrix for the graph of course-career mapping, allowing efficient storage and retrieval of information about the relationships between courses and career paths.
6. **Sorting Algorithms (Merge Sort):** Merge sort is used to sort the list of students by their student IDs, enabling efficient searching and retrieval of student data.
7. **Binary Search:** Binary search is used to search for a student by their ID in the sorted list of students, providing a faster retrieval mechanism for student records.
8. **Linked List (Queue):** We implemented a linked list here, the enQueue and deQueue operations mimic the behavior of a queue implemented using a linked list, ensuring efficient management of academic queries.
9. **Graph:** We used graph here in our code.

## Algorithms Covered:

1. **0-1 Knapsack Algorithm:**
   - Implemented in the knapsack01 function.
   - It is used to find the maximum credit obtained from a set of courses while considering their weights (credit hours) and values (marks).

2. **Activity Selection Algorithm:**
   - Implemented in the activity_selection function.
   - It selects a maximum set of non-overlapping activities from a given list.

3. **Greedy Method to Avoid Time-Overlapping Courses:**
   - Implemented in the avoidTimeOverlappingCourses function.
   - It allows the user to input course start and end times and uses a greedy method to select courses that don't overlap in time.

4. **Merge Sort Algorithm:**
   - Implemented in the merge_sort function.
   - It performs merge sort on an array of student records based on their IDs.

5. **Binary Search Algorithm:**
   - Implemented in the binary_search function.
   - It performs a binary search on an array of student records to find a student by their ID**.**

6. **Floyd-Warshall Algorithm:**
   - Implemented in the floyd_warshall function.
   - It finds the all-pairs shortest paths in a weighted graph using the Floyd-Warshall Algorithm**.**

## Screenshot of Code with Description:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include<stdbool.h>
#define sleep(x)Sleep(x*1000)
#define SIZE 5
#define MAX_COURSES 5
char queue[SIZE][256];
int front = -1, rear = -1;
```

## Explanation:

In this section, you include necessary header files and define macros. The sleep macro is used to pause the program, and SIZE and MAX_COURSES are constants. queue is used for handling academic queries.

```
//0 1 knapsack algo START
int _max(int a, int b){
    if(a>=b)
        return a;
    else
        return b;
}
void knapsack01(int n, int C, int *w, int *v){
    int p[n+1][C+1], o[n];
    int k=0;
    for(int i=0; i<=n; i++){
        for(int j=0; j<=C; j++){
            if(i==0 || j==0){
                p[i][j]=0;
            }
            else if(j-w[i-1]>=0){
                p[i][j]=_max(p[i-1][j], v[i-1]+p[i-1][j-w[i-1]]);
            }
            else if(j-w[i-1]<0){
                p[i][j]=_max(p[i-1][j],p[i][j-1]);
            }

        }

    }
    printf("Max credit: %d\n", p[n][C]);

    for(int i=n, j=C; i>0; i--){
        if(p[i][j]!=p[i-1][j]){
            o[k]=i;
            j-=w[i-1];
            k++;
        }
    }
    printf("Courses included: ");
    for(int i=k-1; i>=0; i--)
        printf("%d ", o[i]);
}
//0 1 knapsack algo STOP
```

## Explanation:

This is the beginning of the 0-1 knapsack algorithm. The _max function returns the maximum of two values. The knapsack01 function uses dynamic programming to solve the 0-1 knapsack problem, maximizing credits while staying within a credit limit.

```
//Activity selection START
void activity_selection(int n, int a[n][2]){
    int x;
    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-i-1; j++){
            if(a[j][1]>a[j+1][1]){
                int tmp = a[j][1];
                a[j][1] = a[j+1][1];
                a[j+1][1] = tmp;

                tmp = a[j][0];
                a[j][0] = a[j+1][0];
                a[j+1][0] = tmp;
            }
        }
    }
    printf("Following activities are selected:\n");
    printf("(%d, %d)\n", a[0][0], a[0][1]);
    x=a[0][1];
    for(int i=1; i<n; i++){
        if(a[i][0]>x){
            x=a[i][1];
            printf("(%d, %d)", a[i][0], a[i][1]);
            printf("\n");
        }
    }
}
//Actvity selection STOP
```

## Explanation:

This section implements the activity selection algorithm, which sorts and selects non-overlapping activities based on their end times.

```
// Greedy Method to avoid time-overlapping courses
void avoidTimeOverlappingCourses() {
    // Greedy method implementation
    int n;
    printf("Number of courses (from which we will determine maximum how many and which
courses you can take without time overlap): ");
    scanf("%d", &n);
    int a[n][2];
    for(int i=0; i<n; i++){
        printf("Give the Start & End time of course %d: ", i+1);
        scanf("%d%d", &a[i][0], &a[i][1]);
    }
    activity_selection(n, a);
}
```

## Explanation:

This function uses the activity selection algorithm to help students avoid time-overlapping courses.

```
// Structure to store student information
struct Student {
    char name[50];
    int id;
    char department[50];
    char courses[MAX_COURSES][50]; // Assuming a student can take up to 5 courses
    int marks[MAX_COURSES];
    char session[10];
    float cgpa;
};
```

## Explanation:

The Student structure stores information about students, including their name, ID, department, courses, marks, session, and CGPA.

```
//Merge sort START
void merge(struct Student *arr,int l,int mid,int r){
int i=0,j=0,k=l;
int n1 = mid-l+1;
int n2 = r-mid;
struct Student left_arr[n1];
struct Student right_arr[n2];
for(int i=0; i<n1; i++){
    left_arr[i]=arr[l+i];
}
for(int j=0; j<n2; j++){
    right_arr[j]=arr[mid+1+j];
```

```
}
while(i<n1 && j<n2){
    if(left_arr[i].id<=right_arr[j].id){
    arr[k]=left_arr[i];
    i++;}
    else{
      arr[k]=right_arr[j];
    j++;
    }
    k++;

}
while(i<n1){
   arr[k]=left_arr[i];
    i++;
    k++;
}
while(j<n2){
   arr[k]=right_arr[j];
    j++;
    k++;
}
}
void merge_sort(struct Student *arr,int l,int r){
if(l<r){
    int mid = (l+r)/2;
    merge_sort(arr,l,mid);
    merge_sort(arr,mid+1,r);
    merge(arr,l,mid,r);
}
}
//Merge sort END
//Binary search START
void binary_search(struct Student *arr,int n){
  int search_id,left=0,right=n-1,flag=0;
  printf("Enter ID : ");
  scanf("%d",&search_id);
  while(left <= right){
    int mid=(left+right)/2;
    if(search_id < arr[mid].id){
    right = mid-1;
  }
   else if(search_id > arr[mid].id){
    left = mid+1;
  }
  else if(search_id == arr[mid].id){
        printf("\nName: %s\n", arr[mid].name);
        printf("ID: %d\n", arr[mid].id);
        printf("Department: %s\n", arr[mid].department);

        printf("Courses:\n");
        for (int j = 0; j < 5; j++) {
            if (strcmp(arr[mid].courses[j], "END") == 0) {
                break;
            }
            printf("Courses %d :%s \n",j+1, arr[mid].courses[j]);
        }

        printf("Session: %s\n", arr[mid].session);
        printf("\n");
    flag=1;
    break;
  }
}
  if(flag != 1){
  printf("NOT FOUND!!");
  printf("\n");
  }
  }
  //Binary search END
```

## Explanation:

These functions implement the merge sort algorithm to sort an array of **Student** structures by their IDs.

```
// Function to add a new student to the system
void addStudent(struct Student **studentList, int *numStudents) {
    // Prompt the user for student information
    struct Student newStudent;
    printf("\nEnter student name: ");
    scanf(" %[^\n]s", newStudent.name);
    printf("Enter student ID: ");
    scanf("%d", &newStudent.id);
    printf("Enter student department: ");
    scanf("%s", newStudent.department);

    // Assuming up to 5 courses can be entered
    printf("Enter up to 5 courses (one per line, 'END' to finish):\n");
    for (int i = 0; i < 5; i++) {
        printf("Course %d: ", i + 1);
        scanf("%s", newStudent.courses[i]);
        if (strcmp(newStudent.courses[i], "END") == 0) {
            break;
        }
    }

    printf("Enter student session: ");
    scanf("%s", newStudent.session);
    // Increase the size of the student list
    (*numStudents)++;
    *studentList = realloc(*studentList, sizeof(struct Student) * (*numStudents));

    // Add the new student to the list
    (*studentList)[(*numStudents) - 1] = newStudent;
    printf("\nStudent added successfully!\n\n");
}
```

## Explanation:

This function allows you to add a new student to the system. It prompts the user for student information, such as name, ID, department, courses, and session, and adds the new student to the list.

```
// Function to display all students
void displayStudentsInfo(struct Student *studentList, int numStudents) {
    if (numStudents == 0) {
        printf("\nNo students found.\n");
        return;
    }

    printf("List of Students:\n");

    for (int i = 0; i < numStudents; i++) {
        printf("\n\nStudent %d:\n", i + 1);
        printf("Name: %s\n", studentList[i].name);
        printf("ID: %d\n", studentList[i].id);
        printf("Department: %s\n", studentList[i].department);
        printf("Session: %s\n", studentList[i].session);}
}
```

## Explanation:

This function displays information about all the students currently in the system, including their names, IDs, departments, and sessions.

```
void storeCourseMarks(struct Student *studentList, int numStudents){
     if (numStudents == 0) {
        printf("No students found.\n");
        return;
    }

    printf("List of Students:\n");
    for (int i = 0; i < numStudents; i++) {
        float cg = 0;
        int total_course=0;
        printf("Student %d:\n", i + 1);
        printf("Name: %s\n", studentList[i].name);
        printf("ID: %d\n", studentList[i].id);
        printf("Department: %s\n", studentList[i].department);
        printf("Session: %s\n", studentList[i].session);
        printf("Courses and Marks:\n");
        for (int j = 0; j < MAX_COURSES; j++) {
            if (strcmp(studentList[i].courses[j], "END") == 0) {
                break;
            }
           // printf("  %s: \n", studentList[i].courses[j]);
             printf("Enter marks for %s: ",  studentList[i].courses[j]);
             scanf("%d", &studentList[i].marks[j]);
             total_course++;

             if (studentList[i].marks[j] >= 80) {
             cg = 4.00 + cg;
            } else if (studentList[i].marks[j] >= 75) {
             cg = 3.75 + cg;
             } else if (studentList[i].marks[j] >= 70) {
             cg = 3.50 + cg;
            } else if (studentList[i].marks[j] >= 65) {
             cg = 3.24 + cg;
             } else if (studentList[i].marks[j] >= 60) {
              cg = 3.00 + cg;}
            else if (studentList[i].marks[j] >= 55) {
           cg = 2.75 + cg;
          } else if (studentList[i].marks[j] >= 50) {
            cg = 2.50 + cg;
            } else if (studentList[i].marks[j] >= 45) {
              cg = 2.25 + cg;
            } else if (studentList[i].marks[j] >= 40) {
            cg = 2.00 + cg;}
            else
         {//printf("Grade: F\n");
          cg = 0.00;
         break;}
        }
        studentList[i].cgpa=cg/total_course;
        printf("\nCGPA: %.2f\n", studentList[i].cgpa);

        printf("\n");
    }
}
```

## Explanation:

This function allows you to input course marks for each student and calculates their CGPA based on the marks. It then displays this information.

```
void enQueue(char *query) {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1))
    printf("\nList of Academic Queries is full!! \n");
  else {
    if (front == -1)
      front = 0;

    rear = (rear + 1) % SIZE;
    strcpy(queue[rear], query);
    printf("\nInserted Query:  %s\n", queue[rear]);
  }
}
```

## Explanation :

This function is part of an academic query system. It adds a new academic query to a queue.

```
void deQueue() {
  char *query;
  if (front == -1) {
    printf("\nThere is no academic query!! \n");
  } else {
    query = queue[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    } else {
      front = (front + 1) % SIZE;
    }
    printf("\nDeleted Query: %s \n", query);
  }
}
```

## Explanation :

This function is part of the academic query system and removes the oldest query from the queue for display.

```
// Display the queue
void display() {
  char *query;
  int i;
 if (front == -1)
    printf(" \nThere is no academic query!!\n");
  else {
    int j =1;
    printf("\nAcademic Query list:  \n");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d.%s \n", j,queue[i]);
     j++;
    }
    printf("%d.%s\n",j, queue[i]);
  }
}
```

## Explanation :

This function is used to display all academic queries currently in the queue.

```
void Academic_query(){
 int op;
while(op !=3){
printf("\nTown Hall Meeting Academic Query System:\n");
printf("1. Enqueue Query\n");
printf("2. Dequeue Query\n");
printf("3. Exit\n");
printf("Enter your choice: ");
scanf("%d", &op);

switch (op) {
case 1: {
char query[256];
 printf("Enter query: ");
 scanf(" %[^\n]s", query);
 enQueue(query);
 break;
 }
 case 2:
 deQueue();
 break;
 case 3:
 break;
 default:
 printf("Invalid choice. Please try again.\n");}
 } ;
}
```

## Explanation:

This function manages academic queries within an interactive menu, allowing users to enqueue and dequeue queries.

```
// Dynamic Programming to maximize courses with limited credits
void maximizeCoursesWithLimitedCredits() {
    printf("Your maximum capacity of contact hour is 8h\n");
    printf("Enter number of courses (from which to determine maximum how many courses can
you take with max credit): ");
    int n, C=8;
    scanf("%d", &n);
    int w[n], v[n];
    for(int i=0; i<n; i++){
        printf("Enter %dth courses credit and contact hour: ", i+1);
        scanf("%d%d", &v[i],&w[i]);
    }
    knapsack01(n, C, w, v);
}
```

## Explanation:

This function helps determine the maximum number of courses that can be taken within a specified credit limit.

```
//Floyd warshal algorithm START
void floyd_warshall(){
  int n;
  printf("Enter number of vatices: ");
  scanf("%d", &n);
  int a[n][n];
  printf("Enter graph of course-career map as adjecency matrix(>=100 for Infinity): \n");
  for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
      scanf("%d", &a[i][j]);
    }
  }
  for(int k=0; k<n; k++){
    for(int i=0; i<n; i++){
      for(int j=0; j<n; j++){
        if(a[i][j]>(a[i][k]+a[k][j])){
          a[i][j]=a[i][k]+a[k][j];
        }
      }
    }
  }
  printf("Optimized graph: All-pairs least cost path:\n");
  for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
      printf("%d\t", a[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}
//Floyd warshal algorithm END
```

## Explanation:

This function implements the Floyd-Warshall algorithm, used to optimize course and career path mapping.

```
int main() {




    struct Student *studentList = NULL;
    int numStudents = 0;

    int choice;
    do {
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), FOREGROUND_RED |
FOREGROUND_INTENSITY);
        printf("\t\t**School Management System**\n\n");
        sleep(1);
```

```c
        printf("1. Add Student\n");
        printf("2. Display Students\n");
        printf("3. Map Courses and Career Paths\n");
        printf("4. Sort student by ID\n");
        printf("5. Search student by ID\n");
        printf("6. Store Course marks & Show result\n");
        printf("7. Take Academic Queries on Town Hall Meeting.\n");
        printf("8. List of Academic Queries on Town Hall Meeting.\n");
        printf("9. Maximum obtainable credit.\n");
        printf("10. Avoid taking time overlapping courses.\n");
        printf("0. Exit\n");
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),  FOREGROUND_RED |
FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY);
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                addStudent(&studentList, &numStudents);
                break;
            case 2:
                displayStudentsInfo(studentList, numStudents);
                break;
            case 3:
                floyd_warshall();
                break;
            case 4:
               merge_sort(studentList, 0, numStudents - 1);
                break;
            case 5:
                merge_sort(studentList, 0, numStudents - 1);
                binary_search(studentList,numStudents);
                break;
            case 6:
                storeCourseMarks(studentList, numStudents);
                break;
            case 7:
                Academic_query();
                break;
            case 8:
                display();
                break;
            case 9:
                maximizeCoursesWithLimitedCredits();
                break;
            case 10:
                avoidTimeOverlappingCourses();
            case 0:
                free(studentList);
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while(1);
    return 0;
    }
```

## Explanation:

The **main** function is the entry point of the program. It provides an interactive menu for various functions, including adding students, displaying student information, and more.

## Output:

```
  **School Management System**

1. Add Student
2. Display Students
3. Map Courses and Career Paths
4. Sort student by ID
5. Search student by ID
6. Store Course marks & Show result
7. Take Academic Queries on Town Hall Meeting.
8. List of Academic Queries on Town Hall Meeting.
9. Maximum obtainable credit.
10. Avoid taking time overlapping courses.
0. Exit

Enter your choice:
```

## Conclusion:

The School Management System is a comprehensive and efficient tool for managing student information and assisting students with their course planning. The use of data structures and algorithms ensures efficient data management and optimization. The project addresses the objectives of creating a flexible and user-friendly system for school administrators and academic advisors. It provides valuable tools for academic planning and query handling.
In conclusion, the School Management System project successfully fulfills its objectives and can be a valuable tool for educational institutions.