**BLOOM LEVEL SETTING**

###### **by**

##### **TEAM TITANS**

*A report for the CS204:Design and Analysis of Algorithm project*



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SRM UNIVERSITY AP::AMARAVATI**

Table of Contents

* Introduction
* Project Modules:
* Architecture Diagram
* Module Description
* Programming Details
* Field/table details
* Algorithm Details

(i)Sorting

(ii)Searching

(ii) Storing the details in a text file

* Source Code
* Comparison of Sorting Algorithms
* Comparison of Searching Algorithms
* Screen Shots

Introduction

Our University (herewith considered as SRM-AP) is going to implement OBE(Outcome Based Education) in their university and you assigned in the project to develop an application with any programming Language you are well versed and you were supposed to do searching and sorting using learned algorithms,comparing your sorting algorithm with any one of existing algorithm,displaying the time complexity of both algorithms and explaining advantages and disadvantages of the algorithm.

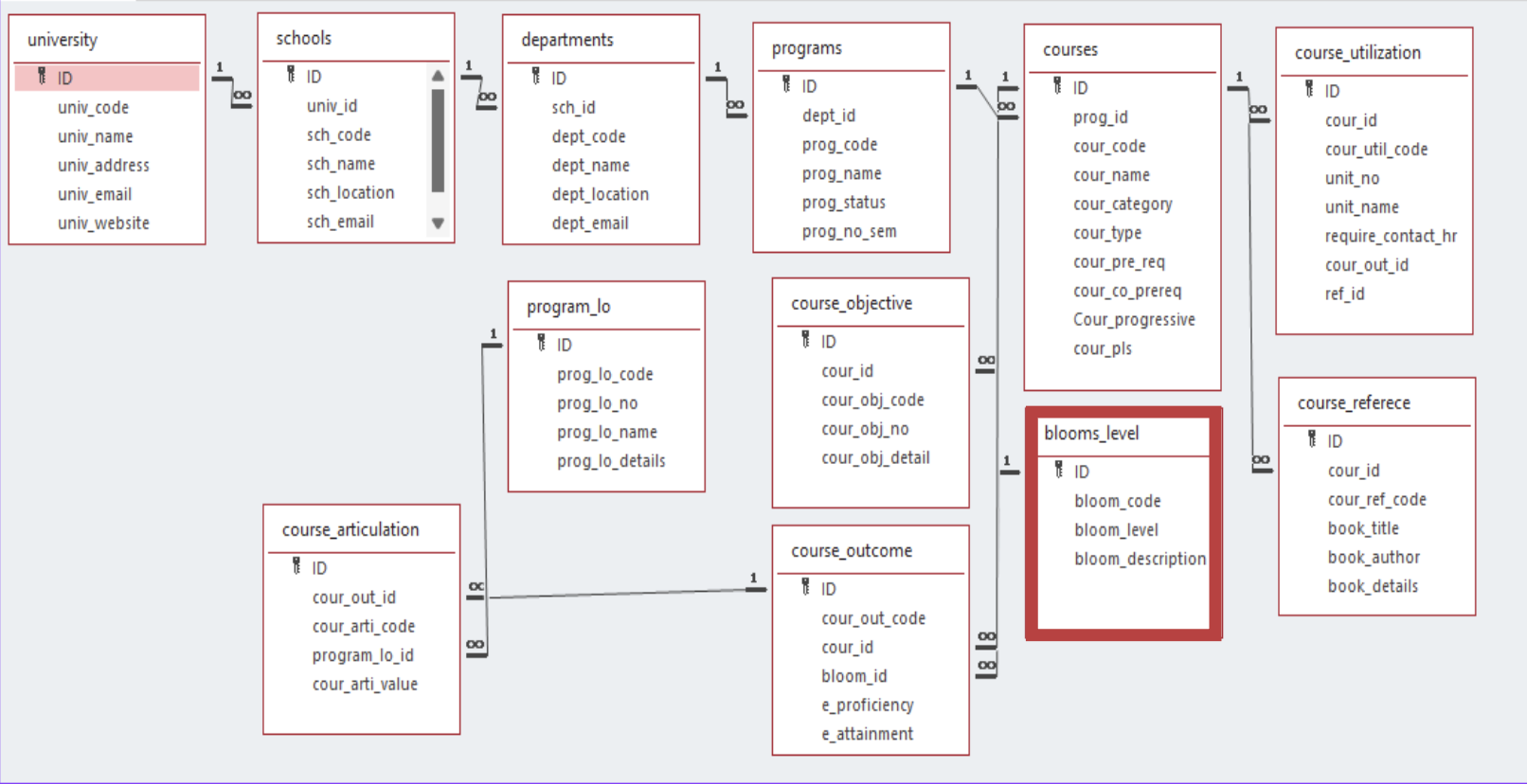
Project Modules:

1. Bloom Level setting

2. Search and Sorting Functions

3. Text File Handling

Architecture Diagram

****

Module Name: Bloom Level Setting

Module Description:

This module is used to create,Update,Retrieve,Delete(hereafter known as CURD) details of the module and storing the details in the text file.you have to provide option for any of the operations of fields mentioned below according to algorithms given for you

Programming Details naming conventions to be used:

File name: bloom\_setting.txt

Function Naming Conventions:

- Create: titans\_bloom\_create

- Update: titans\_bloom\_update

- Retrieve: titans\_bloom\_retrieve

- Delete: titans\_bloom\_delete

- Sorting: titans\_compare\_sort\_mergesort (Merge Sort) and titans\_quick\_sort (Quick Sort)

- Searching: titans\_linear\_search (Linear Search) and titans\_compare\_search\_binarysearch (Binary Search)

- File Operations: titans\_bloom\_storing (for storing data)

Bloom Level Setting:Field/table details

| Field name | Data type |
| --- | --- |
| id | int |
| bloom\_code | String |
| bloom\_level | String |
| bloom\_des | String |

Algorithm Details:

(i) Sorting

Sorting algorithms used:

- Quick Sort: Used for sorting Bloom entries by 'code'. Complexity: O(n^2) worst case, O(n log n) average case.

- Merge Sort: Used for sorting Bloom entries by 'code'. Complexity: O(n log n).

(ii) Searching

Searching algorithms used:

- Linear Search: Simple search with O(n) complexity.

- Binary Search: Searches after sorting, with O(log n) complexity.

Source Code (Full Implementation):

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

typedef struct {

int id;

char code[50];

char level[100];

char des[100];

} bloom;

bloom b[MAX];

int c = 0;

const char\* FILE\_NAME = "bloom\_setting.txt";

void titans\_load\_from\_file();

void titans\_bloom\_storing();

void titans\_bloom\_create();

void titans\_bloom\_update();

void titans\_bloom\_retrieve();

void titans\_bloom\_delete();

void titans\_quick\_sort(int low, int high);

int titans\_partition(int low, int high);

void titans\_compare\_sort\_mergesort(int left, int right);

void titans\_merge(int left, int mid, int right);

int titans\_linear\_search(char \*code);

int titans\_compare\_search\_binarysearch(char \*code);

void titans\_bloom\_search();

void titans\_complexity\_searching();

void titans\_complexity\_sorting();

void titans\_load\_from\_file() {

FILE \*file = fopen(FILE\_NAME, "r");

if (file == NULL) {

printf("File not found or unable to open: %s\n", FILE\_NAME);

return;

}

c = 0;

while (fscanf(file, "%d %49s %99s %99[^\n]", &b[c].id, b[c].code, b[c].level, b[c].des) == 4) {

c++;

}

fclose(file);

}

void titans\_bloom\_storing() {

FILE \*file = fopen(FILE\_NAME, "w");

if (file == NULL) {

printf("Error opening file for writing: %s\n", FILE\_NAME);

return;

}

for (int i = 0; i < c; i++) {

fprintf(file, "%d %s %s %s\n", b[i].id, b[i].code, b[i].level, b[i].des);

}

fclose(file);

}

void titans\_bloom\_create() {

if (c >= MAX) {

printf("Bloom list is full!\n");

return;

}

bloom bl;

printf("Enter bloom ID: ");

if (scanf("%d", &bl.id) != 1) {

printf("Invalid input!\n");

return;

}

printf("Enter bloom Code: ");

scanf("%49s", bl.code);

printf("Enter bloom Level: ");

scanf(" %99[^\n]", bl.level);

printf("Enter bloom Description: ");

scanf(" %99[^\n]", bl.des);

b[c++] = bl;

titans\_bloom\_storing();

printf("Bloom created successfully!\n");

}

void titans\_bloom\_update() {

int id;

printf("Enter bloom ID to update: ");

if (scanf("%d", &id) != 1) {

printf("Invalid input!\n");

return;

}

for (int i = 0; i < c; i++) {

if (b[i].id == id) {

printf("Enter new bloom Code: ");

scanf("%49s", b[i].code);

printf("Enter new bloom Level: ");

scanf(" %99[^\n]", b[i].level);

printf("Enter new bloom Description: ");

scanf(" %99[^\n]", b[i].des);

titans\_bloom\_storing();

printf("Bloom updated successfully!\n");

return;

}

}

printf("Bloom with ID %d not found.\n", id);

}

void titans\_bloom\_retrieve() {

if (c == 0) {

printf("No blooms available.\n");

return;

}

printf("\nList of Blooms:\n");

for (int i = 0; i < c; i++) {

printf("ID: %d\nCode: %s\nLevel: %s\nDescription: %s\n\n",

b[i].id, b[i].code, b[i].level, b[i].des);

}

}

void titans\_bloom\_delete() {

int id;

printf("Enter bloom ID to delete: ");

if (scanf("%d", &id) != 1) {

printf("Invalid input!\n");

return;

}

for (int i = 0; i < c; i++) {

if (b[i].id == id) {

for (int j = i; j < c - 1; j++) {

b[j] = b[j + 1];

}

c--;

titans\_bloom\_storing();

printf("Bloom deleted successfully!\n");

return;

}

}

printf("Bloom with ID %d not found.\n", id);

}

int titans\_partition(int low, int high) {

char pivot[50];

strcpy(pivot, b[high].code);

int i = low - 1;

for (int j = low; j < high; j++) {

if (strcmp(b[j].code, pivot) < 0) {

i++;

bloom temp = b[i];

b[i] = b[j];

b[j] = temp;

}

}

bloom temp = b[i + 1];

b[i + 1] = b[high];

b[high] = temp;

return i + 1;

}

void titans\_quick\_sort(int low, int high) {

if (low < high) {

int pivot = titans\_partition(low, high);

titans\_quick\_sort(low, pivot - 1);

titans\_quick\_sort(pivot + 1, high);

}

}

void titans\_merge(int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

bloom L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = b[left + i];

for (int i = 0; i < n2; i++)

R[i] = b[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (strcmp(L[i].code, R[j].code) <= 0) {

b[k] = L[i];

i++;

} else {

b[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

b[k] = L[i];

i++;

k++;

}

while (j < n2) {

b[k] = R[j];

j++;

k++;

}

}

void titans\_compare\_sort\_mergesort(int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

titans\_compare\_sort\_mergesort(left, mid);

titans\_compare\_sort\_mergesort(mid + 1, right);

titans\_merge(left, mid, right);

}

}

int titans\_linear\_search(char \*code) {

for (int i = 0; i < c; i++) {

if (strcmp(b[i].code, code) == 0) {

return i;

}

}

return -1;

}

int titans\_compare\_search\_binarysearch(char \*code) {

titans\_compare\_sort\_mergesort(0, c - 1);

int left = 0;

int right = c - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int cmp = strcmp(b[mid].code, code);

if (cmp == 0) {

return mid;

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

void titans\_complexity\_searching()

{

printf("Linear Search: time complexity is O(n).\n");

printf("Binary Search: time complexity is O(log n).\n");

}

void titans\_complexity\_sorting()

{

printf("Quick Sort: time complexity is O(n^2) for worst case[when last element is chosen as pivot as the elements are already in sorted order] and O(n log n) for the average case.\n");

printf("Merge Sort: time complexity is O(n log n).\n");

}

int main() {

titans\_load\_from\_file();

int choice;

while (1) {

printf("\n1. Create Bloom\n2. Update Bloom\n3. Retrieve Blooms\n4. Delete Bloom\n5. Compare Search Algorithm(binary search)\n6. Compare Sort

Algorithm(merge sort)\n7. Sort by Code (Quick Sort)\n8. Search by Code (Linear Search)\n9. Complexity search\n10. Complexity Sort\n11. Exit\n");

printf("Enter your choice: ");

if (scanf("%d", &choice) != 1) {

printf("Invalid input!\n");

while (getchar() != '\n');

continue;

}

switch (choice) {

case 1:

titans\_bloom\_create();

break;

case 2:

titans\_bloom\_update();

break;

case 3:

titans\_bloom\_retrieve();

break;

case 4:

titans\_bloom\_delete();

break;

case 5:

{

char code[50];

printf("Enter bloom code to search using Binary Search: ");

scanf("%49s", code);

int index = titans\_compare\_search\_binarysearch(code);

if (index != -1) {

printf("Bloom found using Binary Search:\nID: %d\nCode: %s\nLevel: %s\nDescription: %s\n",

b[index].id, b[index].code, b[index].level, b[index].des);

} else {

printf("Bloom with code %s not found using Binary Search.\n", code);

}

break;

}

case 6:

titans\_compare\_sort\_mergesort(0, c - 1);

printf("Merge Sort complete. Displaying results:\n");

titans\_bloom\_retrieve();

break;

case 7:

titans\_quick\_sort(0, c - 1);

printf("Quick Sort complete. Displaying results:\n");

titans\_bloom\_retrieve();

break;

case 8:

{

char code[50];

printf("Enter bloom code to search using Linear Search: ");

scanf("%49s", code);

int index = titans\_linear\_search(code);

if (index != -1) {

printf("Bloom found using Linear Search:\nID: %d\nCode: %s\nLevel: %s\nDescription: %s\n",

b[index].id, b[index].code, b[index].level, b[index].des);

} else {

printf("Bloom with code %s not found using Linear Search.\n", code);

}

break;

}

case 9:

titans\_complexity\_searching();

break;

case 10:

titans\_complexity\_sorting();

break;

case 11:

printf("Exiting...\n");

return 0;

default:

printf("Invalid choice. Please try again.\n");

break;

}

}

}

Sorting Algorithms

1. Quick Sort:

Algorithm

QuickSort(arr[], low, high)

if low < high

pivot = Partition(arr[], low, high)

QuickSort(arr[], low, pivot - 1)

QuickSort(arr[], pivot + 1, high)

Partition(arr[], low, high)

pivot = arr[high]

i = low - 1

for j = low to high - 1

if arr[j] is less than pivot based on the sorting criteria

swap arr[i] with arr[j]

i++

swap arr[i + 1] with arr[high]

return i + 1

Time complexity

**Best Case**: O(n log n) (when the pivot divides the array into two equal parts)

**Average Case**: O(n log n)

**Worst Case**: O(n^2) (when the pivot is always the smallest or largest element)

- Quick Sort is generally faster on average with O(n log n) complexity but may degrade to O(n^2) if the pivot selection is poor.

2. Merge Sort:

Algorithm

MergeSort(arr[], left, right)

if left < right

mid = (left + right) / 2

MergeSort(arr[], left, mid)

MergeSort(arr[], mid + 1, right)

Merge(arr[], left, mid, right)

Merge(arr[], left, mid, right)

create left\_subarray, right\_subarray

copy elements to left\_subarray and right\_subarray

merge the two subarrays back into arr[] based on the sorting criteria

Time Complexity

**Best Case**:O(n log n)

**Average Case**: O(n log n)

**Worst Case**: O(n log n) (merge sort always divides the array in half and merges, regardless of the order of elements)

- Merge Sort has a consistent time complexity of O(n log n), making it reliable for larger datasets or when stability is required.

Searching Algorithms

1. Linear Search:

Algorithm

LinearSearch(arr[], target\_code, target\_name, target\_email)

for each element in arr[]

if element's code == target\_code

and element's name == target\_name

and element's email == target\_email

return index

return -1

Time complexity

- Linear Search is straightforward with O(n) complexity. It doesn't require sorting but is inefficient for larger datasets.

2. Binary Search:

Algorithm

BinarySearch(arr[], target\_code, target\_name, target\_email)

left = 0

right = arr.length - 1

while left <= right

mid = (left + right) / 2

if arr[mid] matches target based on code, name, and email

return mid

else if arr[mid] is less than target based on sorting criteria

left = mid + 1

else

right = mid - 1

return -1

Time Complexity

- Binary Search is efficient with O(log n) complexity but requires a sorted list. It's well-suited for static datasets where quick lookups are needed.

# Comparison of Sorting Algorithms

Quick Sort:

Time Complexity: Average case \(O(n \log n)\); worst case \(O(n^2)\) if the data is already sorted or highly unbalanced when using the last element as a pivot.

Stability: Not a stable sort; may reorder elements with equal keys.

Performance: Fast on average, especially for large, randomized data.

Use Case: Preferred when memory is limited and average case efficiency is prioritized.

Algorithm: Divides data using a pivot element and recursively sorts each partition.

Merge Sort:

Data Requirement: Requires extra memory for temporary arrays but provides stable sorting.

Time Complexity: Consistently \(O(n \log n)\) for all cases, making it reliable.

Stability: Stable sort; maintains order of elements with equal keys.

Performance: Consistent performance across all datasets, ideal for large or pre-sorted data.

Use Case: Ideal when consistent performance and stability are needed, even for sorted or partially sorted data.

Algorithm: Divides data into halves, recursively sorts, and merges them in order.

# Comparison of Searching Algorithms

Binary Search:

Data Requirement Requires blooms to be sorted by `bloom code`, `bloom name`, or other relevant fields before searching.

Time Complexity:O(log n), making it efficient for larger datasets.

Performance: Faster for large, sorted datasets due to logarithmic complexity.

-Use Case: Suitable for quickly finding a specific `bloom code` or other sorted fields in large datasets.

Algorithm Divides the search space in half with each step (binary approach).

Linear Search:

Data Requirement: Works on unsorted data, no pre-sorting needed.

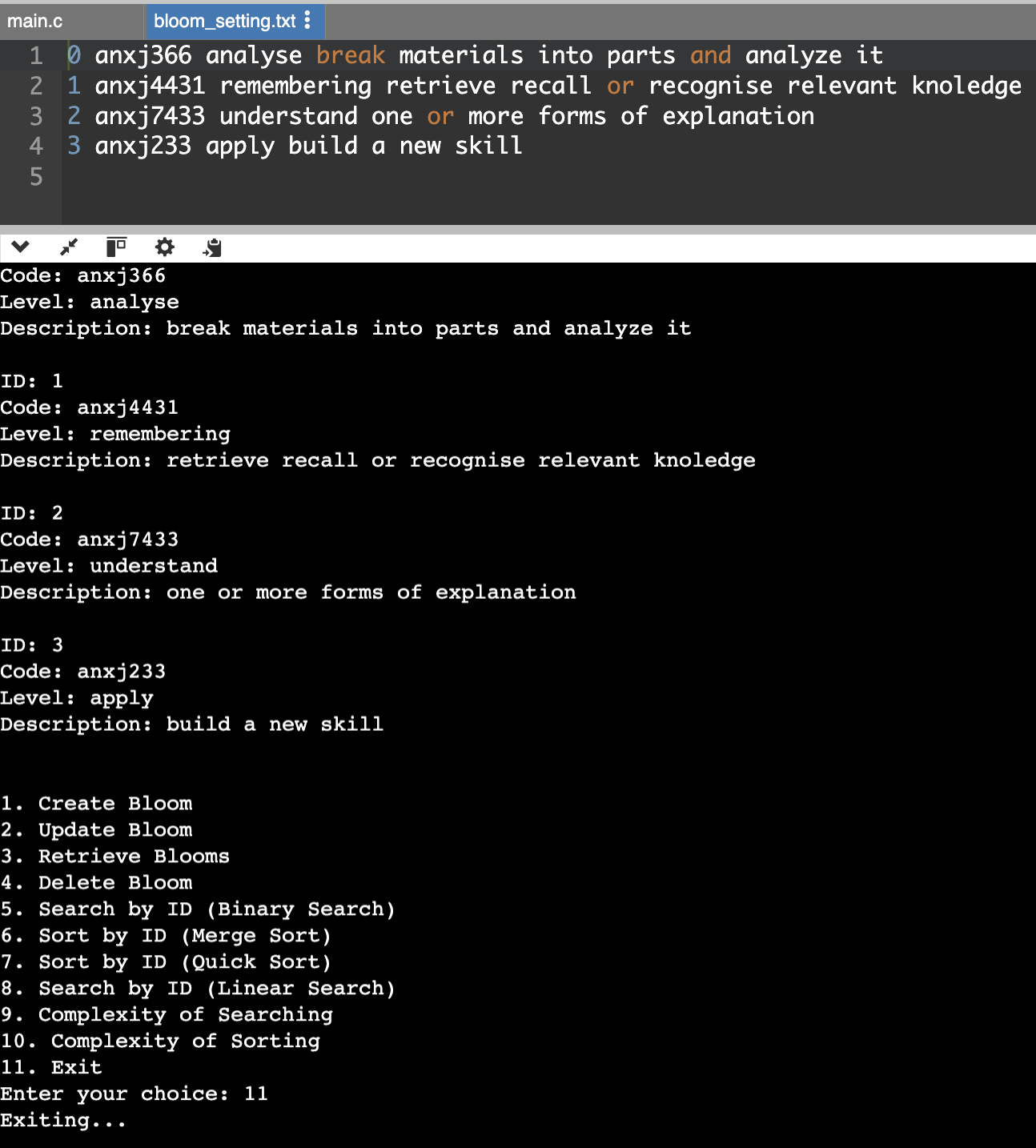
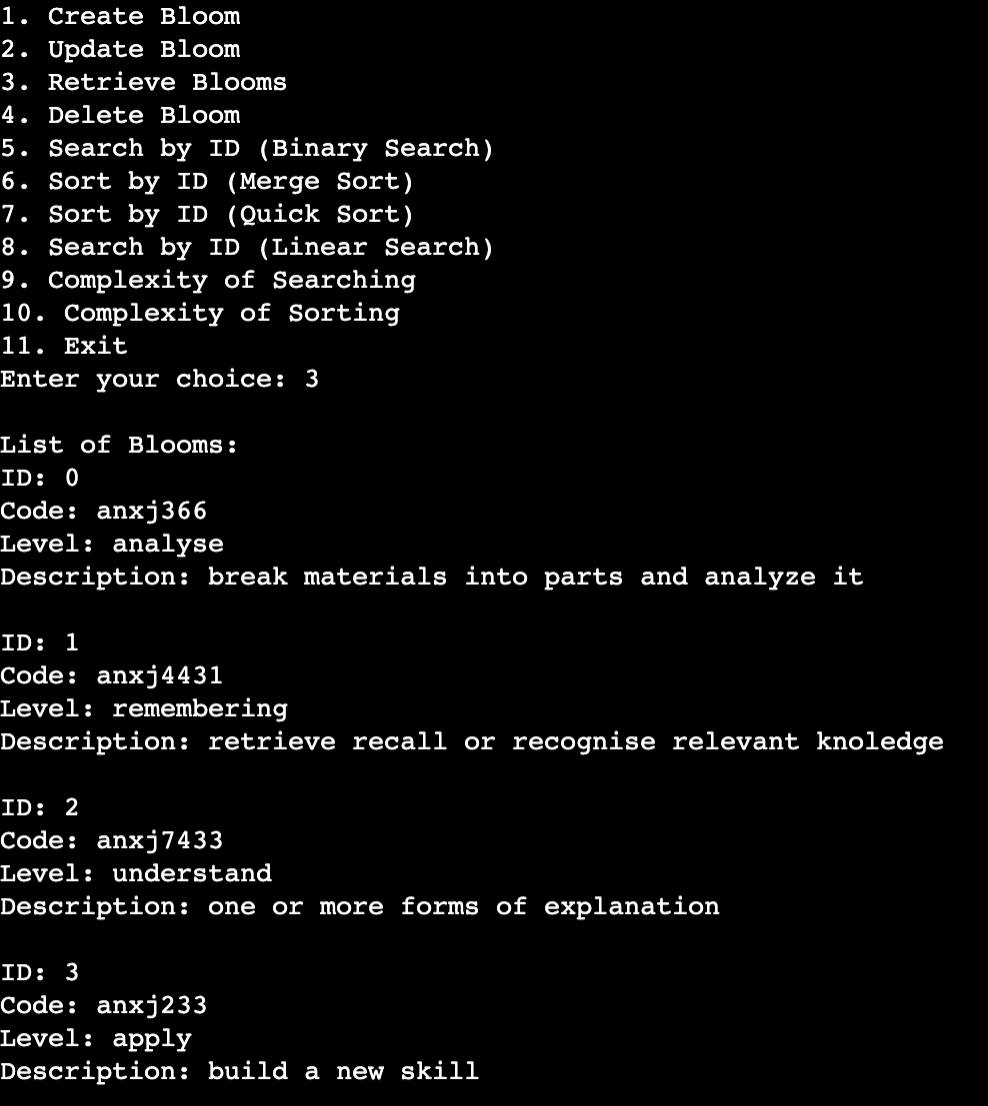
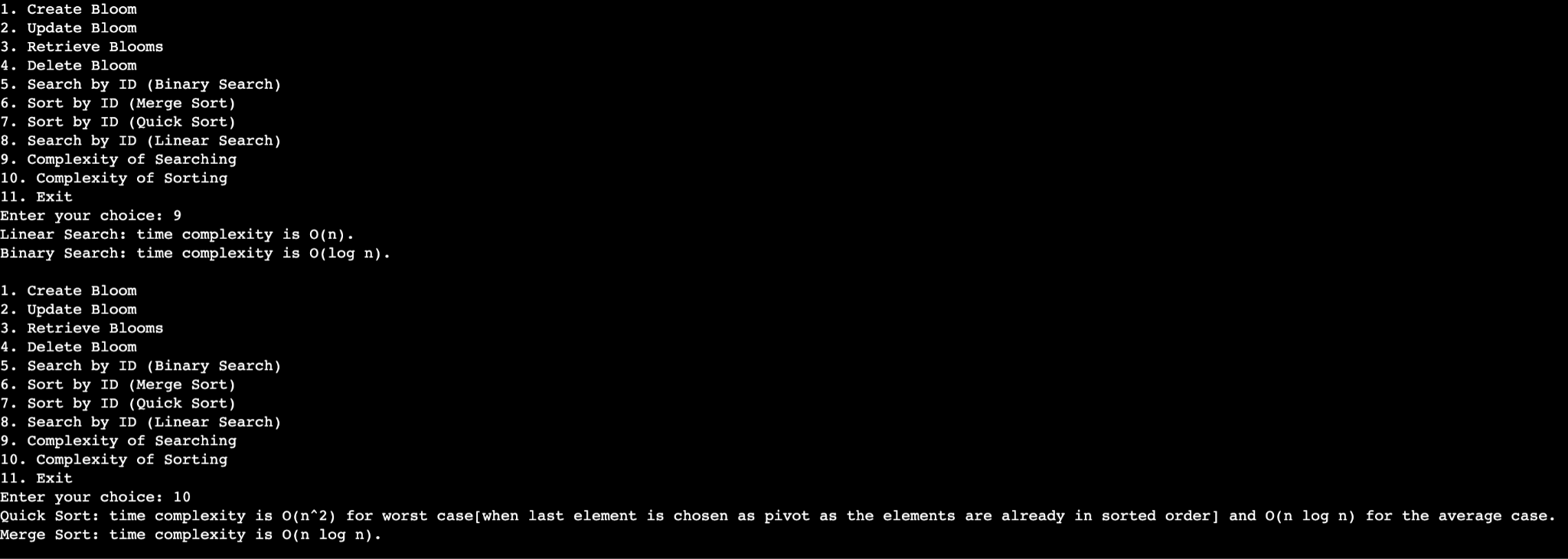
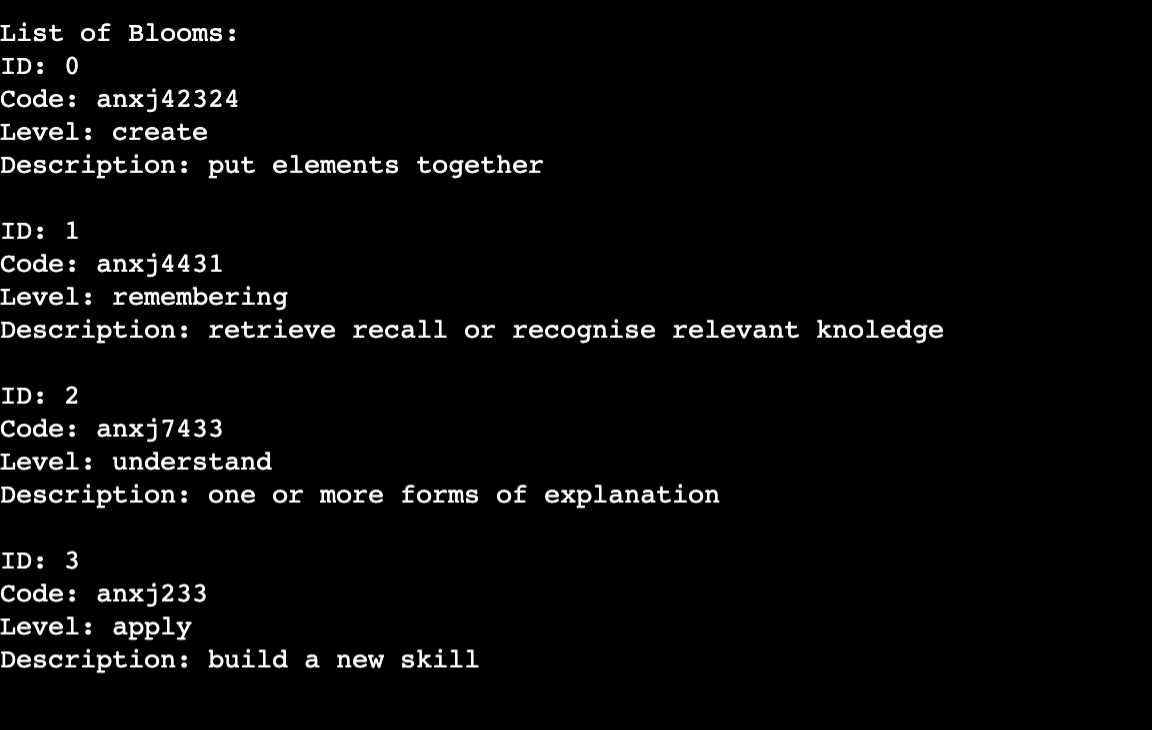
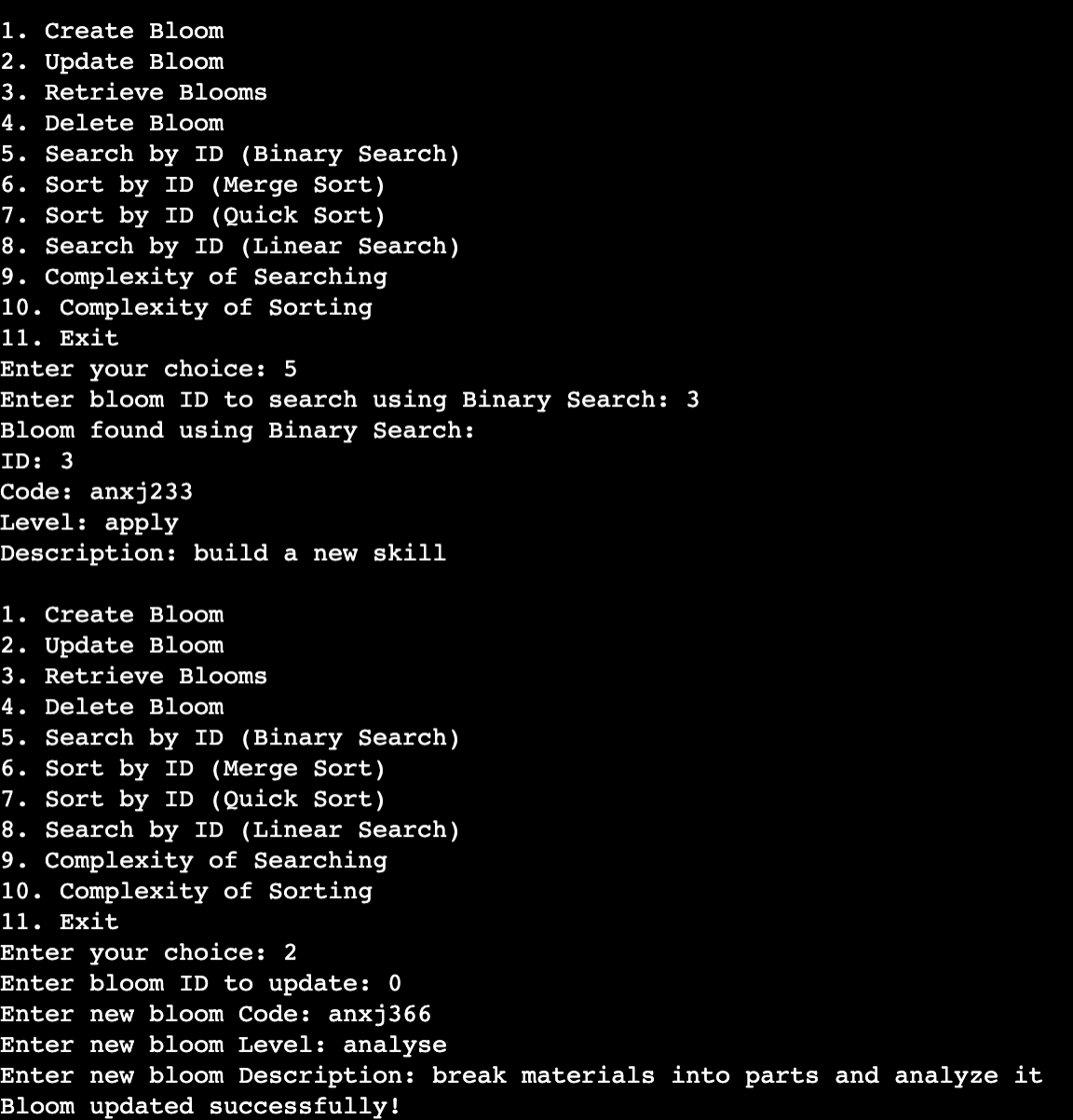
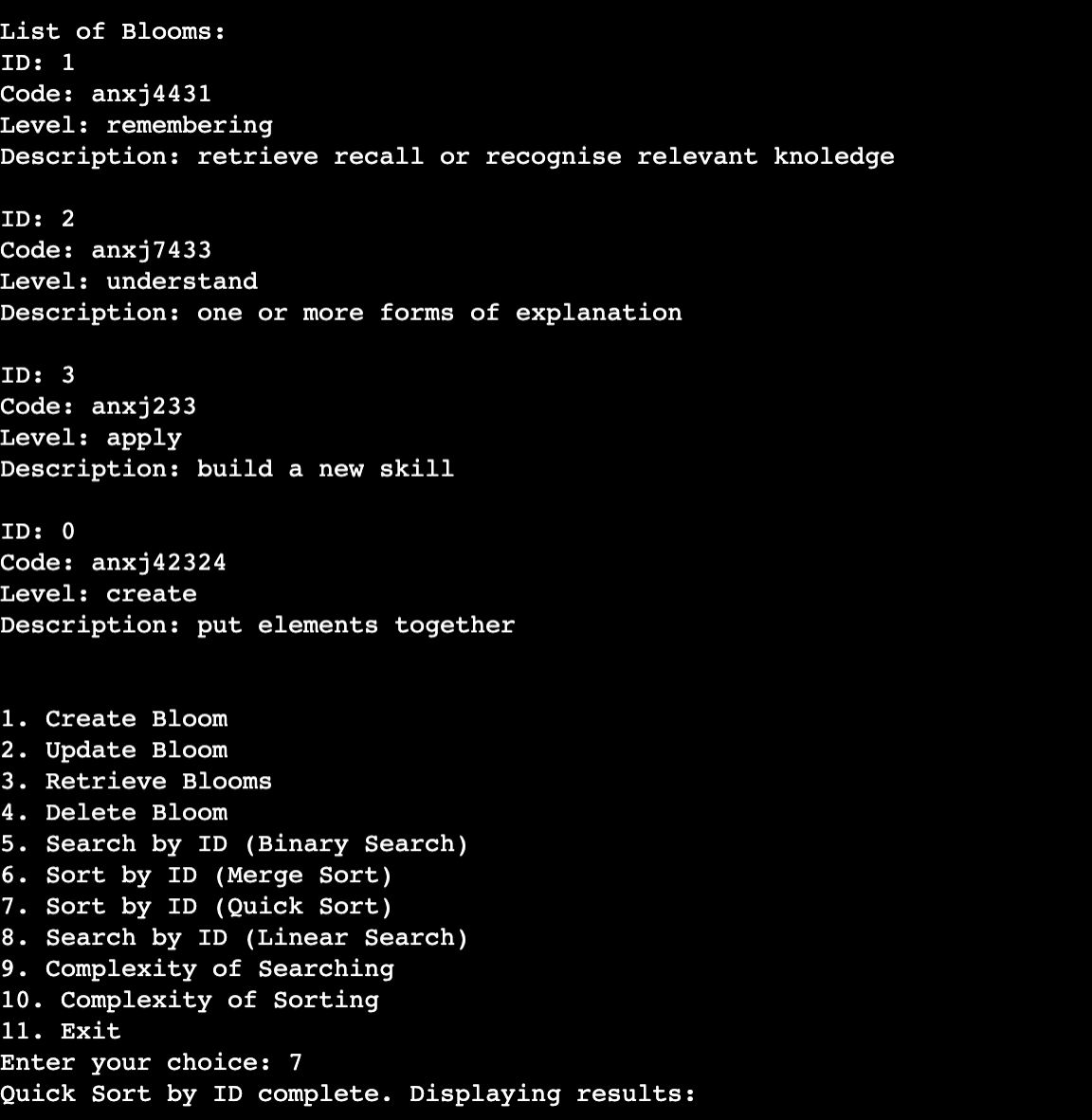
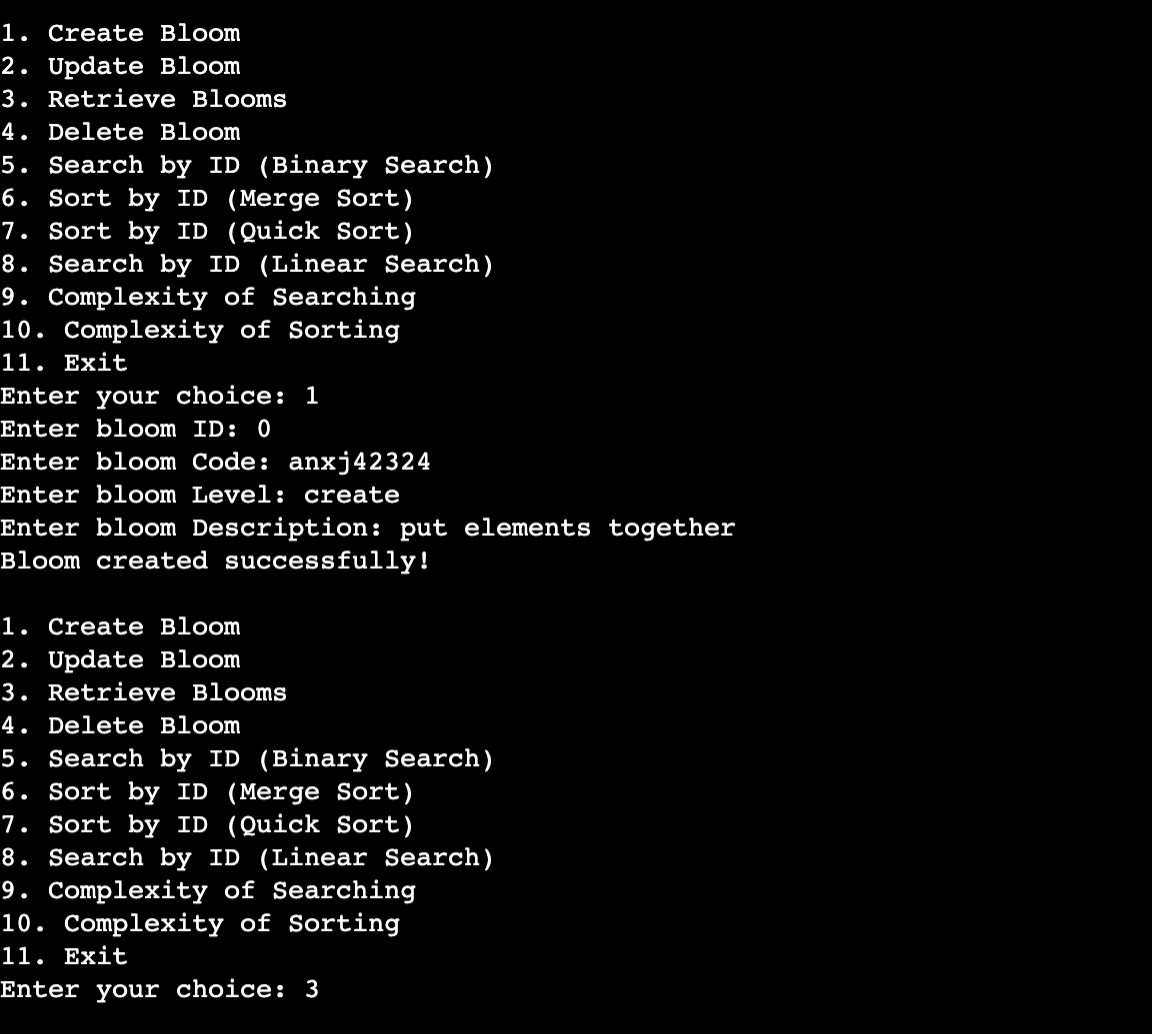
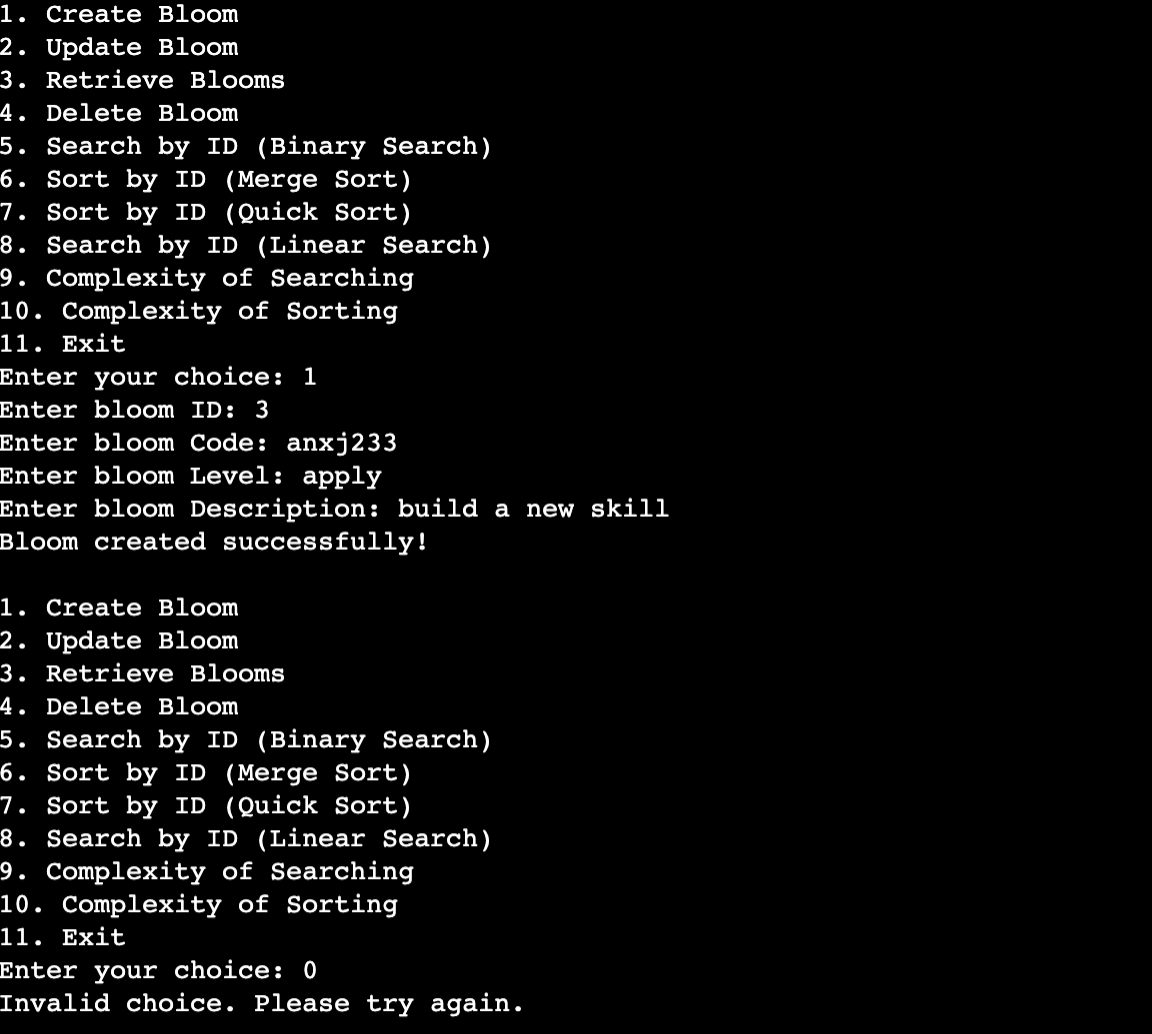
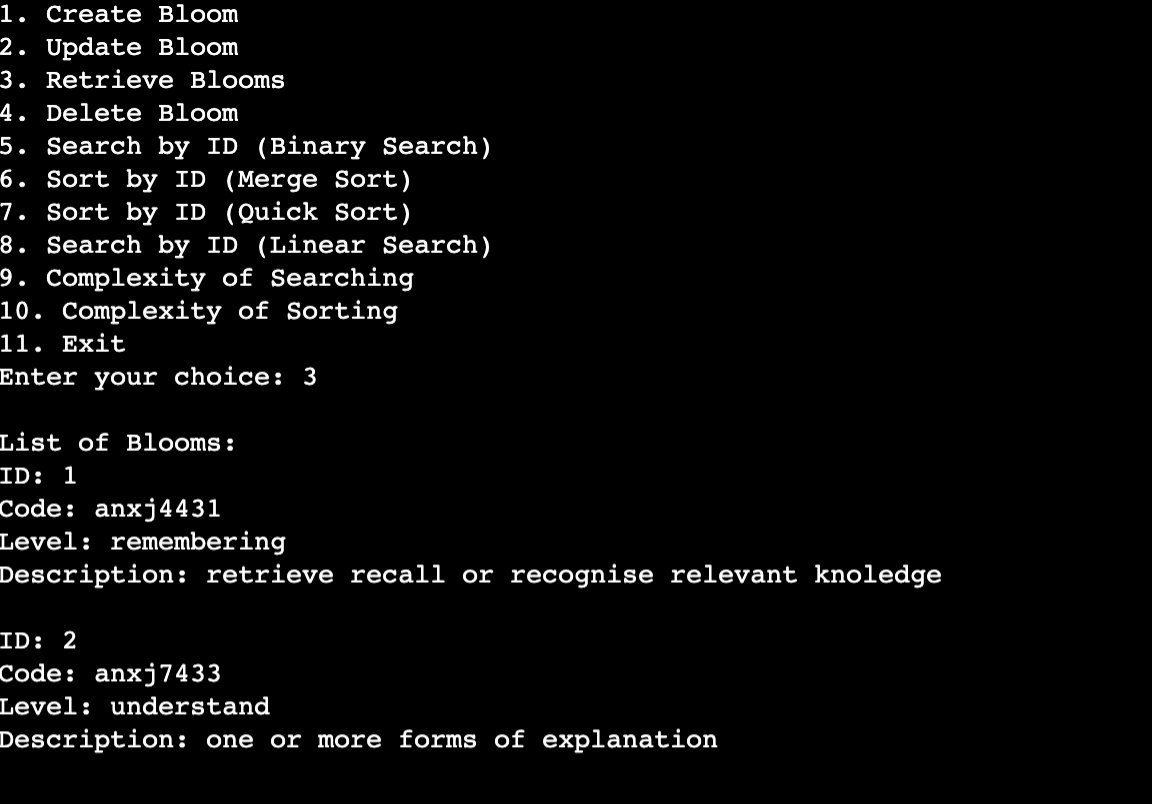
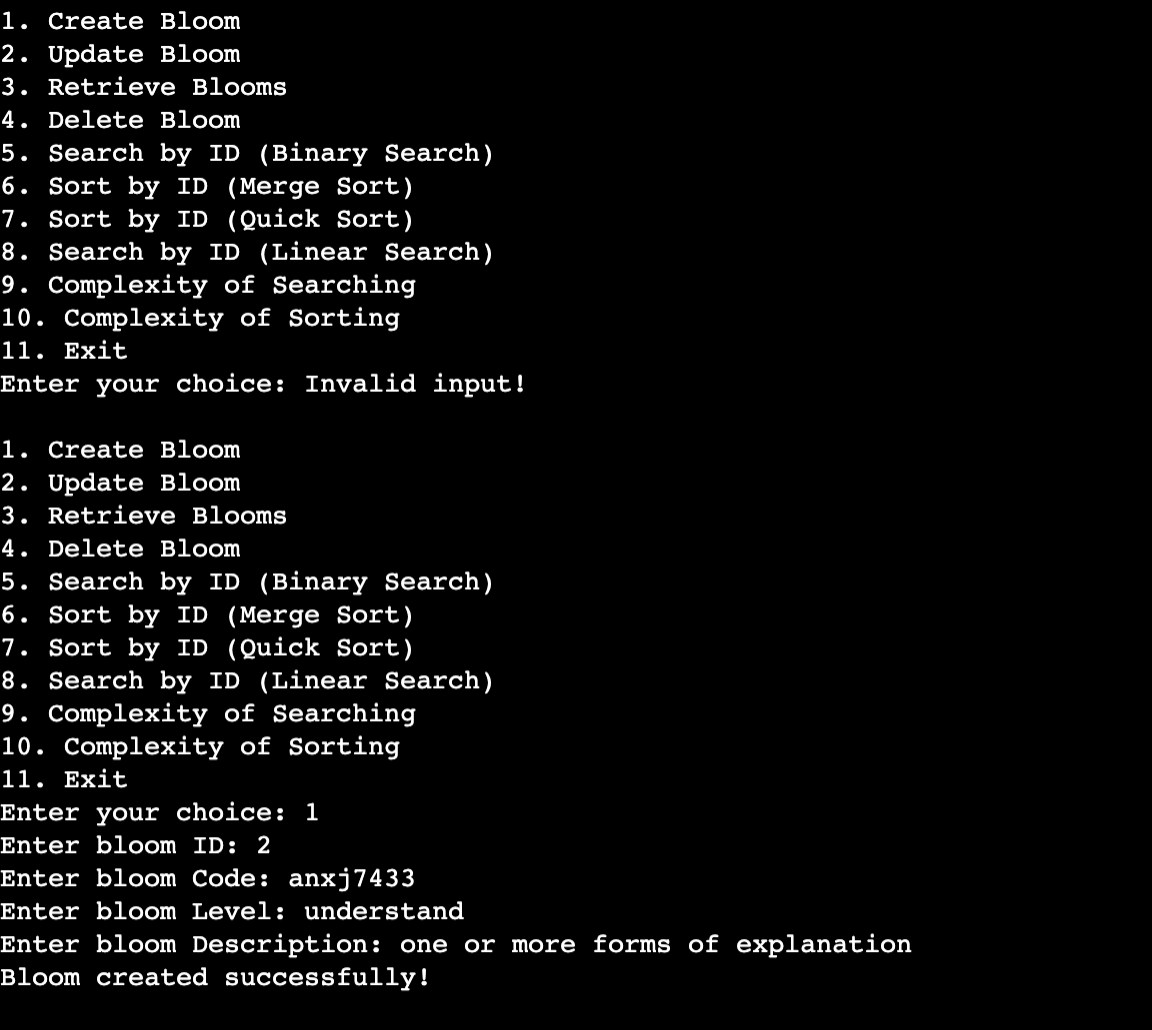
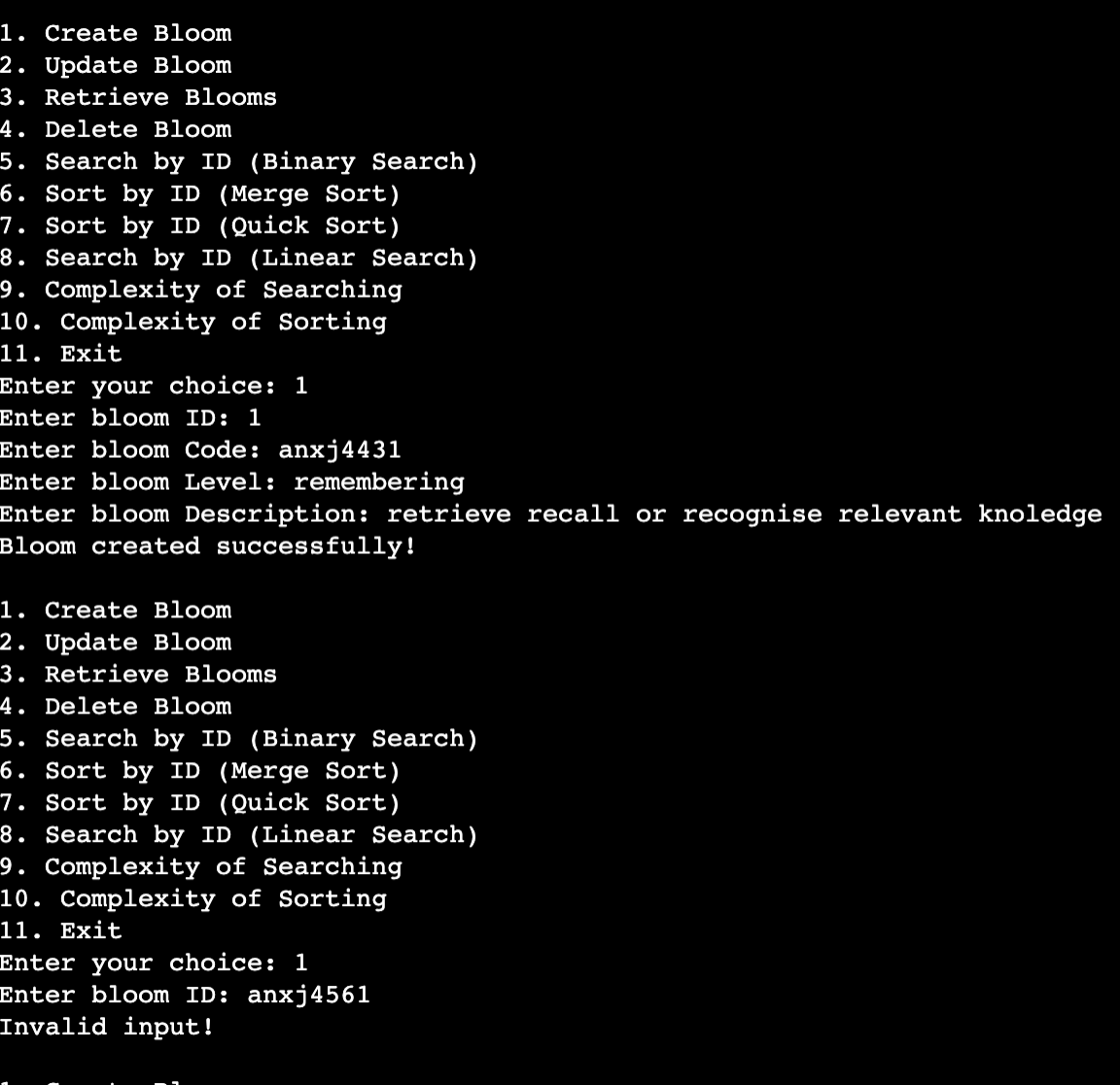
Time Complexity: (O(n), as each entry is checked sequentially.

Performance: Slower for large datasets, especially if not sorted.

Use Case: Practical for small or unsorted datasets.

- Algorithm: Checks each bloom entry one-by-one.

SCREEN SHOTS(OUTPUTS)



Conclusion

The Bloom module, as implemented, provides a comprehensive solution for managing Bloom taxonomy data effectively. The module covers the essential CRUD operations (Create, Retrieve, Update, Delete) while integrating efficient sorting and searching algorithms. By employing both Quick Sort and Merge Sort, the system offers flexibility: Quick Sort is generally faster with average cases, whereas Merge Sort ensures consistent performance and stability.

In addition, the Linear and Binary Search functionalities provide options for both simplicity and efficiency in data retrieval. Linear Search is straightforward and effective for unsorted data, while Binary Search, combined with sorting, enables rapid lookup in sorted datasets. The use of a text file for data storage ensures data persistence and easy retrieval.

Overall, this Bloom module is designed to be robust, efficient, and scalable, making it well-suited for use in educational settings where taxonomy levels and objectives need to be managed systematically. This project demonstrates a balance between simplicity in design and efficiency in execution, meeting the requirements for dynamic data handling and providing a framework adaptable for future enhancements.