# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 07 : Asymptotic Complexity of Algorithms  
&   
Sorting Algorithms**

**Date: 16th November, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# 

# Lab 07 : Asymptotic Complexity Analysis & Sorting Algortihms

**Introduction**

This lab is based on the analysis of different algorithms and sorting algorithms.

**Objectives**

Objective of this lab is to make students analyze different algorithms and their asymptotic complexities.

**Tools/Software Requirement**

Visual Studio 2012 or gcc or g++

**Tasks**

**Part A : Asymptotic Complexity Analysis**

**Task 1:**

You have already implemented a function that prints all elements of a list of size n, where n>=0. What is the Big-O complexity of that operation?

Answer: O(n), the list will iterate for n times. The worst and the best condition are O(n).

**Task 2:**

In assignment 1, you were asked to implement a function that prints all elements of a singly linked list in the **reverse order.** Your task is to answer the following questions:

1. Suppose the elements of a singly linked list are printed using an iterative approach with the help of two nested loops. What is the Big-O time complexity of printing n values in the reverse order? What is the Big-O **space complexity**?

Answer: The inner loop is used to store all element in stack or any data structure and then using the outer loop print the all the element in stack by popping them. The worst case would be O(n2) as two loops are used and the space complexity is O(n) as linear spacing, the size of list increases with n inputs.

1. Suppose the elements are printed using a recursive function given below.   
   Void RecursivePrint( node \*temp){

If (temp!=NULL){

RecursivePrint(temp->next);

cout<<temp->data;

}// end of if.

}

What is the Big-O time complexity of this function? What is the Big-O space complexity? Hint: stack, function calls!

Answer: The time complexity of this function using recursion will have the worst case of O(n). The space complexity will be O(n), increase will be linear as the size of list depends on n and the size of the Ram. The list elements will be stored in stack(ram) on depends on ram size and the ram has a limit.

**Task 3:**

Suppose you have an **array-based list** of size **n**. Implement a function takes a position number **pos** as input from the user, and returns the value stored at that position. What is the Big-O time complexity of this function?

Answer: The time complexity is O(1). It requires a single operation to access the linear array list, using the index number.

#include<iostream>

using namespace std;

#define size 10

int main()

{

int num = 0;

int list[size] = { 1,2,3,4,5,6,7,8,9,10 };

cout << " Enter the Postion: ";

cin >> num;

cout << " The data at index is " << list[num - 1] << endl;

return 0;

}

Text

Description automatically generated

**Task 4:**

What is best-case and worst-case time complexity to destroy a linked list of size n?

Answer: the best and worst-case to destroy a list of size n is O(n). you access each node of the list from the start and delete each node one by one. So, it will run for n times as equal to the size of the list.

**Task 5:**

What is best-case and worst-case time complexity to destroy an array-based list of size n?

Answer: A dynamic array-based list has the time complexity O(1). It requires one operation to deallocate the memory allocated, by the delete operation.

**Task 6:**

Your task is to reverse the order of all n elements of a singly linked list using a stack. Declare a stack of pointers to a class node (It should store the address to an object of the class node). Traverse the linked list and push the address of every node onto a stack. Pop the elements and update the links by reversing the order of nodes in a list. Update the start and last pointers.

What are the best case and Big-O time complexities to reverse a linked list using this approach? What is its Big-O space complexity?

O(n) and Ω(n) are the time complexities. The list is put in a stack for n times and then again n times for poping all elements. Therefore, the best and worst time complexities are the same.

O(n) from space complexities. The space will depend on the size n of the linked list. The larger the (n) size the larger the space of stack required.

Code:

#include<iostream>

using namespace std;

#define size 10

class node

{

public:

int data;

node\* next;

};

node\* stack = new node[size];

int head = -1;

// basic functions for operating stack

bool IsEmpty()

{

return head == -1;

}

bool IsFull()

{

return head == size - 1;

}

void Push(node \*temp)

{

if (!IsFull()) stack[++head] = \*temp;

}

node Pop()

{

if (!IsEmpty()) return stack[head--];

}

class linkedlist

{

public:

node\* start;

node\* last;

int length = 10;

linkedlist() //creating a linkedlist (pre-set)

{

node\* newnode1 = new node;

node\* newnode2 = new node;

node\* newnode3 = new node;

node\* newnode4 = new node;

node\* newnode5 = new node;

node\* newnode6 = new node;

node\* newnode7 = new node;

node\* newnode8 = new node;

node\* newnode9 = new node;

node\* newnode10 = new node;

node\* newnode = new node;

newnode1->data = 1;

newnode2->data = 2;

newnode3->data = 3;

newnode4->data = 4;

newnode5->data = 5;

newnode6->data = 6;

newnode7->data = 7;

newnode8->data = 8;

newnode9->data = 9;

newnode10->data = 10;

newnode1->next = newnode2;

newnode2->next = newnode3;

newnode3->next = newnode4;

newnode4->next = newnode5;

newnode5->next = newnode6;

newnode6->next = newnode7;

newnode7->next = newnode8;

newnode8->next = newnode9;

newnode9->next = newnode10;

newnode10->next = newnode;

start = newnode1;

last = newnode10;

}

void PutinStack()

{

node\* temp = new node; //temperary node

node\* temp\_ = new node;

temp\_->data = 1000;

node t;

temp = start; //assigning start node to temp

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

{

Push(temp); //pushing the addresses of the nodes to the stack

temp = temp->next;

}

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

{

if (i == 0) //the top node of stack becomes the first node of the list

{

t = Pop();

temp = &t;

start = temp;

cout << temp->data << endl;

}

else

{

if (i != length) //if not the last node until then keep poping element in the list

{

t = Pop();

temp\_ = &t;

temp->next = temp\_;

temp = temp->next;

cout << temp->data << endl;

}

if (i == length - 1) //the bottom node of stack becomes the last node of the list

last = temp;

}

}

}

void print() //member function to output the list using temp starting from start ptr

{

node\* temp = new node;

temp = start;

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

{

cout << temp->data << endl;

temp = temp->next;

}

}

};

int main()

{

linkedlist obj;

cout << " Original: " << endl; //calling the orginal list

obj.print();

cout << " Reverse: " << endl; //calling the reversed list

obj.PutinStack();

}

Screenshot:

Text

Description automatically generated with medium confidence

**Task 7:**

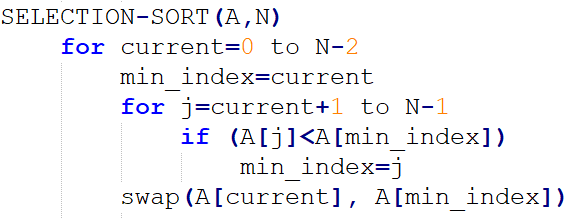
In the below given table,

|  |  |  |
| --- | --- | --- |
| **Operation** | **Big-O Complexity** | **Best-case Complexity** |
| Insert an element at the front of a singly linked list of size n | O(1) | Ω(1) |
| Insert an element at the tail end of a singly linked list of size n. **plast** points to last node. | O(1) | Ω(1) |
| Delete the last node of a singly linked list of size n. **plast** points to its last node. | O(n) | Ω(1) |
| Insertion at the front of an array list of size n | O(n) | Ω(1) |
| Insertion at the tail end of an array list of size n | O(1) | Ω(1) |
| Enqueue in a queue of length n. | O(1) | Ω(1) |
| Dequeue in a queue of length n. | O(1) | Ω(1) |
| Converting an expression of length n from infix to postfix form using stack | O(n) | Ω(n) |
| Finding an element via Binary Search algorithm in a sorted array-list of size n. | O(log(n)) | Ω(log(n)) |
| Finding an element via Binary Search algorithm in an **unsorted** array-list of size n. Think about it! | O(nlog(n)) | Ω(nlog(n)) |

**Part B : Sorting Algorithms**

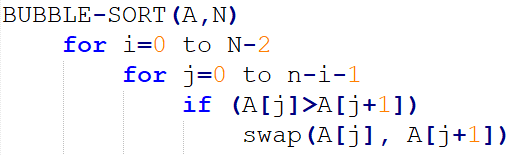
**Selection Sort:**

Selection sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



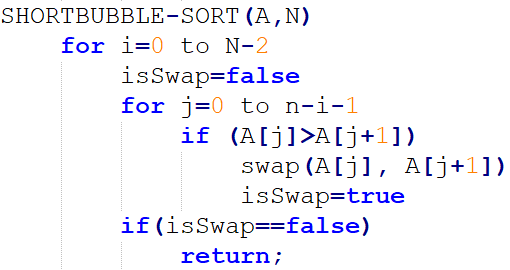
**Bubble Sort:**

Insertion sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Short Bubble Sort:**

It is a variant of Bubbles sort the best-case complexity of which is Ω (n). Note that the best-case and worst-case complexities of Bubble sort are of order n2.



**Task 1:**

Implement Selection sort, Bubble sort, Short Bubble sort algorithms in C++.  
  
Declare an array of large size e.g. 1000 or more, randomly store values in all the indexes in the range e.g. 1 to 3000. Then call the sort functions one by one. Do multiple runs of the program and note both clock time and the no of iterations.

Code:  
#include<iostream>

using namespace std;

#define size 10

#include<time.h>

int Bubble\_Sort(int list[])

{

int count = 0;

int temp = 0;

for (int i = 0; i < size; i++) // the algo runs for n times

{

for (int j = 0; j < size - 1; j++) //compares each element of the array so n-1 operations

{

if (list[j] > list[j + 1]) //if the element is more then the next element then swapping occurs

{

temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

}

count++; // counting for number of iterations

}

count++; // counting for number of iterations

}

return count;

}

int Short\_Bubble\_Sort(int list[])

{

int temp = 0;

int flag = 1;

int count = 0;

int n = size - 1;

while (n > 0 && flag != 0) // the algo runs until no more swapping occurs in list

{

flag = 0;

for (int j = 0; j < n ; j++) //compares each element of the array so n operations

{

if (list[j] > list[j + 1]) //if the element is more then the next element then swapping occurs

{

temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

flag = 1; // flag is set to one to indicate swap

}

count++;// counting for number of iterations

}

count++;// counting for number of iterations

n--; // as list starts sorting the number of comparing decrements

}

return count;

}

int Selection\_Sort(int list[])

{

int count = 0;

int min\_location = -1;

int temp = 0;

int smallest = 5000;

int current = 0;

int flag = 0;

int n = 0;

for (int i = 0; i < size - 1; i++)

{

min\_location = i; //storing the first index on min\_location

for (int j = i + 1; j < size; j++)

{

if (list[j] < list[min\_location]) //comparing with all the following values in list

min\_location = j; //storing the index of the smallest value

count++; // counting for number of iterations

}

temp = list[i]; //swapping the elements of the smallest and cuurent compare index

list[i] = list[min\_location];

list[min\_location] = temp;

}

return count;

}

int main()

{

int\* list = new int[size];

int\* list2 = new int[size];

int\* list3 = new int[size];

srand(time(0));

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

{

list[i] = 1 + (rand() % 3000); //random values from 1 to 3000

list2[i] = list[i]; //storing same list values in other for comparing time and space

list3[i] = list[i];

}

cout << " Orginal List." << endl;

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

cout << list[i] << endl;

cout << "\n\n";

cout << "Number of Iterations: " << Bubble\_Sort(list) << endl; //output number of iterations

cout << endl << " Sorted List." << endl; //displaying the sorted list

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

cout << list[i] << endl;

cout << "\n\n";

cout << endl;

cout << "Number of Iterations: " << Short\_Bubble\_Sort(list2) << endl;//output number of iterations

cout << endl << " Sorted List." << endl;//displaying the sorted list

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

cout << list2[i] << endl;

cout << "\n\n";

cout << endl;

cout << "Number of Iterations: " << Selection\_Sort(list3) << endl;//output number of iterations

cout << endl << " Sorted List." << endl;//displaying the sorted list

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

cout << list3[i] << endl;

cout << endl;

return 0;

}

ScreenShots:  
Shape

Description automatically generated with medium confidenceShape

Description automatically generated with medium confidence

For size 10000

Text

Description automatically generated

For Size 20000

Text

Description automatically generated

For Size 30000

Text

Description automatically generated

As the size of the list increases, the selection sort algo becomes better in terms of iterations. But, the number of iterations of selection sort and short-bubble sort are close as compared to normal bubble sort.