[CENG 315 All Sections] Algorithms

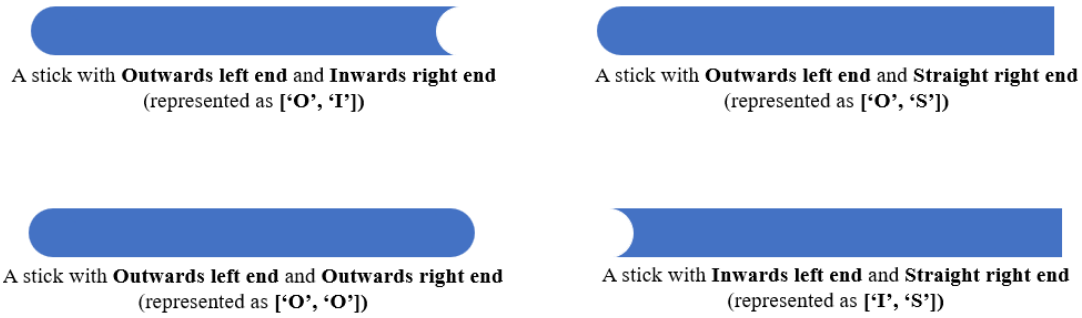
Description Submission view

THE3

Available from: Friday, November 11, 2022, 11:59 AM
Due date: Sunday, November 13, 2022, 11:59 AM
Requested files: the3.cpp, test.cpp, the3_solution.cpp (Download)
Type of work: Individual work

Problem:

In this exam, you are given an array of sticks with two end points where each end point can be any of the following 3 types: Inwards End, Outwards End and Straight End. An illustration for some possible stick instances are given in the figure below.



Each stick has also "size" property. The size differs from 1 to 10. The size of each stick is specified in a different input array. Your task is to build the longest path by combining the sticks end to end. The rules of combination are given as follows:

- An Inwards end can be combined with an Outwards end only.
- An Outwards end can be combined with an Inwards end only.
- A Straight end can be combined with another Straight end only.
- The path can be started with any type of end. Similarly, it can be finished with any type of end.
- While building the path, you should **preserve the ordering of the sticks given in the input array.** That is; if stick A comes before stick B in the input array, then stick A can not come after stick B in the resulting path.
- You do not have to use all the sticks given in the input array.
- You **should not reverse the sticks**. That is, left and right ends of the stick should not be swapped.
- In order to obtain the same results with the answer key, please obey the rules given in "Implementation" part.

An example for stick combination

Please examine the examples below. Note that, each stick is defined with its left and right end types . "I" represents Inwards end, "O" represents Outwards end and "S" represents Straight end. For instance, ['I', 'S'] represents a stick starting with Inwards end and ending with Straight end.

Example IO:

1) Given array arr = { {'I', 'S'}, {'O', 'I'}, {'S', 'O'} } and len = {1, 1, 1}:

- the longest path is {'I', 'S} + {'S', 'O'}.
- return value (i.e. max length) is 2 for each of three functions.
- number of recursive calls is 4.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 1],
  [1, 0, 1],
  [1, 2, 1]] .

2) Given array arr = { {'I', 'S'}, {'O', 'I'}, {'S', 'O'} } and len = {1, 5, 2}:

- the longest path is {'O', 'I'}.
- return value (i.e. max length) is 5 for each of three functions.
- number of recursive calls is 4.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 1],
  [5, 0, 1],
  [5, 3, 1]] .

3) Given array arr = { {'I', 'S'}, {'S', 'S'}, {'O', 'I'}, {'S', 'O'}, {'O', 'O'}, {'I', 'O'}, {'S', 'O'} } and len = {1, 1, 1, 1, 1, 1, 1}:

- the longest path is {'I', 'S'} + {'S', 'S'} + {'S', 'O'} + {'I', 'O'}.
- return value (i.e. max length) is 4 for each of three functions.
- number of recursive calls is 32.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 1],
  [0, 0, 2],
  [1, 0, 2],
  [1, 3, 2],
  [1, 3, 2],
  [1, 4, 2],
  [1, 4, 2]] .

4) Given array arr = { {'I', 'S'}, {'S', 'S'}, {'O', 'I'}, {'S', 'O'}, {'O', 'O'}, {'I', 'O'}, {'S', 'O'} } and len = {5, 3, 3, 1, 8, 5, 3}:

- the longest path is {'O', 'I'} + {'O', 'O'} + {'I', 'O'}.
- return value (i.e. max length) is 16 for each of three functions.
- number of recursive calls is 32.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 5],
  [0, 0, 8],
  [3, 0, 8],
  [3, 9, 8],
  [3, 11, 8],
  [3, 16, 8],
  [3, 16, 8]] .

5) Given array arr = { {'I', 'S'}, {'S', 'I'}, {'O', 'I'}, {'S', 'O'}, {'O', 'O'}, {'I', 'I'}, {'I', 'O'}, {'S', 'O'}, {'O', 'S'} } and len = {1, 1, 1, 1, 1, 1, 1, 1, 1}:

- the longest path is {'I', 'S'} + {'S', 'I'} + {'O', 'I'} + {'O', 'O'} + {'I', 'I'} + {'O', 'S'}.
- return value (i.e. max length) is 6 for each of three functions.
- number of recursive calls is 83.

- at memoization and dynamic programming, final mem array is:

  [[0, 0, 1],
  [2, 0, 1],
  [3, 0, 1],
  [3, 2, 1],
  [3, 4, 1],
  [5, 4, 1],
  [5, 5, 1],
  [5, 5, 1],
  [5, 5, 6]] .

6) Given array arr = { {'I', 'S'}, {'S', 'I'}, {'O', 'I'}, {'S', 'O'}, {'O', 'O'}, {'I', 'O'}, {'I', 'S'} } and len = {1, 1, 1, 1, 1, 1, 1}:

- the longest path is:

  --> {'I', 'S'} + {'S', 'I'} + {'O', 'I'} + {'O', 'O'} + {'I', 'O'} + {'I', 'S'}.

- return value (i.e. max length) is 6 for each of three functions.
- number of recursive calls is 53.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 1],
  [2, 0, 1],
  [3, 0, 1],
  [3, 2, 1],
  [3, 4, 1],
  [3, 5, 1],
  [3, 5, 6]] .
  .

7) Given array arr = { {'I', 'S'}, {'S', 'I'}, {'O', 'I'}, {'S', 'O'}, {'O', 'O'}, {'I', 'O'}, {'S', 'O'}, {'O', 'S'} } and len = {9, 9, 7, 8, 7, 10, 10, 5}:

- the longest path is {'I', 'S'} + {'S', 'I'} + {'O', 'I'} + {'O', 'O'} + {'I', 'O'}.
- return value (i.e. max length) is 42 for each of three functions.
- number of recursive calls is 60.
- at memoization and dynamic programming, final mem array is:

  [[0, 0, 9],
  [18, 0, 9],
  [25, 0, 9],
  [25, 17, 9],
  [25, 32, 9],
  [25, 42, 9],
  [25, 42, 9],
  [25, 42, 30]] .

8) Given array arr = { {'I', 'S'}, {'I', 'I'}, {'O', 'I'}, {'S', 'O'}, {'S', 'I'}, {'O', 'O'}, {'I', 'S'}, {'S', 'O'}, {'S', 'S'}} and len = {1, 1, 1, 1, 1, 1, 1, 1, 1}:

- there are 4 longest paths:

  --> {'I', 'S'} + {'S', 'I'} + {'O', 'O'} + {'I', 'S'} + {'S', 'O'} and

  --> {'I', 'S'} + {'S', 'I'} + {'O', 'O'} + {'I', 'S'} + {'S', 'S'} and

  --> {'I', 'I'} + {'O', 'I'} + {'O', 'O'} + {'I', 'S'} + {'S', 'O'} and

  --> {'I', 'I'} + {'O', 'I'} + {'O', 'O'} + {'I', 'S'} + {'S', 'S'}.

- return value (i.e. max length) is 5 for each of three functions.
- number of recursive calls is 60.
- at memoization and dynamic programming, final mem array is:

[[0, 0, 1],
[1, 0, 1],
[2, 0, 1],
[2, 2, 1],
[2, 2, 1],
[2, 3, 1],
[2, 3, 4],
[2, 5, 4],
[2, 5, 5]] .

Implementation:

You will implement three different functions for three different solutions of that problem:

- Direct recursive implementation in ***recursive_sln()***
- Recursion with memoization in ***memoization_sln()***
- Dynamic programming in ***dp_sln()***

All three functions are expected to **return** the answer to the given problem which is **the length of the longest path.** Return **only** the max length value and nothing more.

The number of recursive calls that your recursive function makes should be counted. That number should be stored using the ***int &number_of_calls*** variable**,** which is the last parameter at the definition of the *recursive_sln()*. Basically, the value of that variable should be incremented by one at each execution of the *recursive_sln()* function. In order to accomplish that, the increment operation may be done at the first line of the function implementation, as already done in the function template given to you. So, **do not change the first line of the *recursive_sln()* function and do not manipulate the *number_of_calls* variable at anywhere else**. Do **not return** that variable. Since it is passed by reference, its final value will be available for testing/grading without returning it. **IMPORTANT:** In order to obtain the same number_of_calls with the answer key, please use the following recurrence relation:

IF N == *size*-1
M(N) = max{ **M(n) where n < N**, M(i)+len(N) IF start(N) MATCHES end(i) where i < N}
ELSE
M(N) = max{ M(j) IF end(N) equals to end(j), M(i)+len(N) IF start(N) MATCHES end(i) }
where
i <= N-1 && i > t FOR ALL t start(N) matches end(t)
j <= N-1 && j > t FOR ALL t end(N) equals to end(t)
start(x) MATCHES end(y) IFF {{start(x) =='I' && end(y) =='O} OR {start(x)=='O' && end(y)=='I'} OR {start(x)=='S' && end(y) =='S'}}
size is the length of the initial input array, not the length of the current partial array passed to the function.

CAUTION: Please read this recurrence relation carefully. Put ***break*** statement(s) into the necessary places of your code to satisfy the above relation exactly. Also, use recurrence upto the last step which is the stopping case to end the recursion, that is: ***IF ... THEN return len[0]***.

The ***char**& arr*** variable is the parameter which passes the input array of sticks to your functions. **Do not modify that array!** Note that it is a 2D array where each element of it is an another array of size 2 representing a stick with 2 ends. That is, each inner array is in the form of **[<left end type>, <right end type>]** where the <left end type> and <right end type> are char variables ('I', 'O', or 'S') representing the left and right ends of the stick, respectively.

The ***int*& len*** variable is the parameter which passes the sizes of sticks defined in *arr* array to your functions. **Do not modify that array too!** The size of the i^{th} stick in the *arr* array is specified in the i^{th} element of len array. Size is an integer value between 1 and 10.

At *recursive_sln()* and *memoization_sln()*, ***int i*** is intended to represent and pass indices of arr. While testing and grading, it will be initialized to **sizeof(arr)-1** (i.e. the last index of the array) . At *dp_sln()*, instead of such a variable, directly the **size of the arr** is given via ***int size*** parameter.

For memoization and dynamic programming, you should use ***int**& mem*** variable (i.e. array), which is the last parameter at definitions of those functions, as **the array of memoized values**. For both *memoization_sln()* and *dp_sln()* functions, final values in the *mem* variable will be

considered for grading. Note that it is a 2D array. Each inner array is structered as an array of size 3 representing the stick combination ending with an Inwards end, Outwards end and Straight end, respectively. While testing and grading, all the inner arrays of *mem* array will be initialized to all -1's. So, while implementing your functions, **you can assume that *mem* is an array of array of -1's. Do not return that variable/array.**

The difference between *memoization_sln()* and *dp_sln()* functions is that the first one consists of top-down approach (recursive) and the other one includes bottom-up (iterative) approach.

Implement the functions in most efficient way.

Constraints:

- Maximum array size will be **1000**.

Evaluation:

- After your exam, black box evaluation will be carried out. You will get full points if

  1. your all three functions return the correct max length
  2. your recursive_sln() function makes the correct number of recursive calls
  3. and you fill the ***mem*** array correctly, as stated.
  4. you did not change the input arrays (the array of sticks and the length array).

Specifications:

- There are **3 tasks** to be solved in **12 hours** in this take home exam.
- You will implement your solutions in **the3.cpp** file.
- Do **not** change the first line of **the3.cpp**, which is **#include "the3.h"**
- *<iostream>, <climits>* , *<cmath>* , *<cstdlib>* are included in "the3.h" for your convenience.
- Do **not** change the arguments and return **types** of the functions **recursive_sln(), memoization_sln()** and **dp_sln()** in the file **the3.cpp.** (You should change return **values**, on the other hand.)
- Do **not** include any other library or write include anywhere in your **the3.cpp** file (not even in comments).
- Do **not** write any helper method.

Compilation:

- You are given **test.cpp** file to **test** your work on **ODTÜClass** or your **locale**. You can and you are encouraged to modify this file to add different test cases.
- If you want to **test** your work and see your outputs you can **compile and run** your work on your locale as:

```
>g++ test.cpp the3.cpp -Wall -std=c++11 -o test

> ./test
```

- You can test your **the3.cpp** on virtual lab environment. If you click **run**, your function will be compiled and executed with **test.cpp**. If you click **evaluate**, you will get a feedback for your current work and your work will be **temporarily** graded for **limited** number of inputs.
- The grade you see in lab is **not** your final grade, your code will be re-evaluated with **completely different** inputs after the exam.

The system has the following limits:

- a maximum execution time of 32 seconds
- a 192 MB maximum memory limit
- an execution file size of 1M.
- Solutions with longer running times will not be graded.
- If you are sure that your solution works in the expected complexity constrains but your evaluation fails due to limits in the lab environment, the constant factors may be the problem.

```
int recursive_sln(int i, char**& arr, int*& len, int &number_of_calls);
int memoization_sln(int i, char**& arr, int*& len, int**& mem);
int dp_sln(int size, char**& arr, int*& len, int**& mem);
```

Requested files

the3.cpp

```cpp
#include "the3.h"



int recursive_sln(int i, char**& arr, int*& len, int &number_of_calls) { //direct recursive
    number_of_calls+=1;

    //your code here

    return 0; // this is a dummy return value. YOU SHOULD CHANGE THIS!
}



int memoization_sln(int i, char**& arr, int*& len, int**& mem) { //memoization

    //your code here

    return 0; // this is a dummy return value. YOU SHOULD CHANGE THIS!
}



int dp_sln(int size, char**& arr, int*& len, int**& mem) { //dynamic programming

    //your code here

    return 0; // this is a dummy return value. YOU SHOULD CHANGE THIS!
}

```

test.cpp

```cpp
// this file is for you for testing purposes, it won't be included in evaluation.

#include <iostream>
#include <random>
#include <ctime>
#include <cstdlib>
#include "the3.h"

char getRandomEnd(){
    int r = rand()%3;
    if (r == 0)
        return 'I';
    if (r == 1)
        return 'O';
    return 'S';
}

void randomArray(char**& array, int*& len, int size)
{
    array = new char* [size];
    len = new int[size];
    for (int i = 0; i < size; i++)
    {
        char* stick = new char[2];
        char left = getRandomEnd();
        stick[0] = left;
        char right = getRandomEnd();
        stick[1] = right;
        array[i] = stick;
        len[i] = rand() % 10 + 1;
    }
}


void printArrayInLine(char** arr, int* len, int arraySize){
    std::cout << "[";
    for(int i = 0; i < arraySize; i++){
        std::cout << "[" << arr[i][0] << ", " << arr[i][1] << "]";
        if (i == arraySize - 1){
            continue;
        }
        else{
            std::cout << ", \n";
        }
    }
    std::cout << "]" << std::endl;

    std::cout << "{";
    for(int i = 0; i < arraySize; i++) {
        std::cout << len[i] << ", ";
        if (i == arraySize - 1){
            continue;
        }
        else{
            std::cout << ", \n";
        }
    }
    std::cout << "}" << std::endl;
}


void printMemInLine(int** arr, int arraySize){
    std::cout << "[";
    for(int i = 0; i < arraySize; i++){
        std::cout << "[" << arr[i][0] << ", " << arr[i][1] << ", " << arr[i][2] << "]";
        if (i == arraySize - 1){
            continue;
        }
        else{
            std::cout << ", \n";
        }
    }
    std::cout << "]" << std::endl;
}



void test(){
    clock_t begin, end;
    double duration;
    int max_length_rec;
    int max_length_mem;
    int max_length_dp;


    int size = 10;      // max 1000
    char** arr;
    int* len;
    randomArray(arr, len, size);
    std::cout << "Array:" << std::endl;
    printArrayInLine(arr, size);



    std::cout << "_____________RECURSIVE IMPLEMENTATION:_____________" << std::endl;

    int number_of_calls_rec = 0;

    if ((begin = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    max_length_rec = recursive_sln(size-1, arr, len, number_of_calls_rec);

    if ((end = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    std::cout << "Duration: " << duration << " seconds." << std::endl;

```

```
    std::cout << "Max length: " << max_length_rec << std::endl;
    std::cout << "Number of recursive calls: " << number_of_calls_rec << std::endl;

    std::cout << "--------------------------------------------------";
    std::cout << "\n" << std::endl;



    int** mem = new int*[size];


    std::cout << "_____________________MEMOIZATION:_________________" << std::endl;

    for(int i = 0; i < size; i++) {
        mem[i] = new int[3];
        for (int j = 0; j < 3; j++)
            mem[i][j] = -1;
    }


    if ((begin = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    max_length_mem = memoization_sln(size-1, arr, len, mem);
    if ((end = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    std::cout << "Duration: " << duration << " seconds." << std::endl;

    std::cout << "Max length: " << max_length_mem << std::endl;
    std::cout << "Final mem: " << std::endl;
    printMemInLine(mem, size);

    std::cout << "--------------------------------------------------";
    std::cout << "\n" << std::endl;



    std::cout << "_________________DYNAMIC PROGRAMMING:_____________" << std::endl;

    for(int i = 0; i < size; i++)
        for (int j = 0; j < 3; j++)
            mem[i][j] = -1;


    if ((begin = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    max_length_dp = dp_sln(size, arr, len, mem);

    if ((end = clock() ) ==-1)
        std::cerr << "clock error" << std::endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    std::cout << "Duration: " << duration << " seconds." << std::endl;

    std::cout << "Max length: " << max_length_dp << std::endl;
    std::cout << "Final mem: " << std::endl;
    printMemInLine(mem, size);

    std::cout << "--------------------------------------------------";
    std::cout << "\n" << std::endl;

}

int main()
{
    srandom(time(0));
    test();
    return 0;
}
```

the3_solution.cpp

```cpp
#include "the3.h"



int recursive_sln(int i, char**& arr, int*& len, int &number_of_calls){ //direct recursive
    number_of_calls+=1;

    if (i == 0)
        return len[0];

    int x = 0, y = 0;
    int current_call = number_of_calls;


    // definitely not use arr[i]
    for (int j = i-1; j >= 0; j--) {
         if (current_call == 1) {
            int temp = recursive_sln(j, arr, len, number_of_calls);
            x = temp > x ? temp : x;
         }
         else if (arr[j][1] == arr[i][1]) {
            x = recursive_sln(j, arr, len, number_of_calls);
            break;
         }
    }

    // definitely use arr[i]
    for (int j = i-1; j >= 0; j--)
        if ((arr[i][0] == 'I' && arr[j][1] == 'O') ||
            (arr[i][0] == 'O' && arr[j][1] == 'I') ||
            (arr[i][0] == 'S' && arr[j][1] == 'S')) {
            y = recursive_sln(j, arr, len, number_of_calls);
            break;
        }
    y += len[i];

    if (x > y)
        return x;
    return y;

}



int memoization_sln(int i, char**& arr, int*& len, int**& mem){ //memoization

    // mem[i][0] represents terminating by 'I'
    // mem[i][1] represents terminating by 'O'
    // mem[i][2] represents terminating by 'S'

    if (i == 0) {
        if (arr[0][1] == 'I') {
            mem[0][0] = len[0];
            mem[0][1] = 0;
            mem[0][2] = 0;
        }
        else if (arr[0][1] == 'O') {
            mem[0][1] = len[0];
            mem[0][0] = 0;
            mem[0][2] = 0;
        }
        else {
            mem[0][2] = len[0];
            mem[0][0] = 0;
            mem[0][1] = 0;
        }
    }
    else {
        if (mem[i - 1][0] == -1 || mem[i - 1][1] == -1 || mem[i - 1][2] == -1)
            memoization_sln(i - 1, arr, len, mem);
        mem[i][0] = mem[i-1][0];
        mem[i][1] = mem[i-1][1];
        mem[i][2] = mem[i-1][2];

        // starts with what?
        int update;
        if (arr[i][0] == 'I')
            update = mem[i-1][1] + len[i];
        else if (arr[i][0] == 'O')
            update = mem[i-1][0] + len[i];
        else
            update = mem[i-1][2] + len[i];

        // ends with what?
        if (arr[i][1] == 'I')
            mem[i][0] = mem[i][0] > update ? mem[i][0] : update;
        else if (arr[i][1] == 'O')
            mem[i][1] = mem[i][1] > update ? mem[i][1] : update;
        else
            mem[i][2] = mem[i][2] > update ? mem[i][2] : update;
    }


    int max = mem[i][0] > mem[i][1] ? mem[i][0] : mem[i][1];
    max = max > mem[i][2] ? max : mem[i][2];
    return max;
}



int dp_sln(int size, char**& arr, int*& len, int**& mem){ //memoization

    // mem[i][0] represents terminating by 'I'
    // mem[i][1] represents terminating by 'O'
    // mem[i][2] represents terminating by 'S'

    mem[0][0] = 0;
    mem[0][1] = 0;
    mem[0][2] = 0;
```

```
110
111     if (arr[0][1] == 'I')
112         mem[0][0] = len[0];
113     else if (arr[0][1] == 'O')
114         mem[0][1] = len[0];
115     else
116         mem[0][2] = len[0];
117
118     for (int i = 1; i < size; i++){
119         // first initialize with the previous ones
120         mem[i][0] = mem[i-1][0];
121         mem[i][1] = mem[i-1][1];
122         mem[i][2] = mem[i-1][2];
123
124         // starts with what?
125         int update;
126         if (arr[i][0] == 'I')
127             update = mem[i-1][1] + len[i];
128         else if (arr[i][0] == 'O')
129             update = mem[i-1][0] + len[i];
130         else
131             update = mem[i-1][2] + len[i];
132
133         // ends with what?
134         if (arr[i][1] == 'I')
135             mem[i][0] = mem[i][0] > update ? mem[i][0] : update;
136         else if (arr[i][1] == 'O')
137             mem[i][1] = mem[i][1] > update ? mem[i][1] : update;
138         else
139             mem[i][2] = mem[i][2] > update ? mem[i][2] : update;
140     }
141
142     int max = mem[size-1][0] > mem[size-1][1] ? mem[size-1][0] : mem[size-1][1];
143     max = max > mem[size-1][2] ? max : mem[size-1][2];
144     return max;
145
146 }
147
148
149
150
```

VPL