FINAL EXAM A

SECOND SEMESTER OF ACADEMIC YEAR 2022 - 2023

STUDENT ID: __________ NAME: __________ MACHINE NO: __________

Problem	1	2	3	4	5	6	TOTAL
	LinkedList	String or Sorting	Big-O	ADT-1& Recursion	ADT-2& Recursion	Trees& Recursion	
Score	17	15	18	15	17	18	100

Instructions:

- The time for this exam is ***2.5 hours***.
- Use of anything other than a pencil, pen, notes, the official textbook, and the computer provided by Educational Technology Center is prohibited. In particular, ***no computers or digital devices of any kind you carry with are permitted.***
- Answer submission guides:
  1) Be sure to create an answer folder on the ***D drive.*** The folder is named after ***your student ID + your name***. For example, if your student ID is 2022000123 and your name is "李明", you need to create a folder "***2022000123李明***". Please be sure to save the corresponding answer (cpp file) to this folder after each problem is completed.
  2) Please check the completed **1.cpp, 2.cpp, 3.cpp, 4.cpp, 5.cpp, 6.cpp** in your folder on the D drive. As soon as the exam time is up, follow the teacher's instructions to submit your folder.

1. LinkedList (17pts)

Write a function:

bool separable(ListNode* ***list***);

that given a pointer to a singly-linked list(单向链表) that stores integers, returns whether it can be cut into three segments with equal sums of internal elements.

Clarifications:

- A segment is a continuous sublist of the original list.
- The input list is guaranteed to be nonempty.
- All elements are positive.

Examples:

- separable({1 -> 2 -> 3 -> 4 -> 2 -> 1 -> 5}) should return true, since it can be cut into three segments {1 -> 2 -> 3}, {4 -> 2}, and {1 -> 5} with equal element sum (6).

- separable({1 -> 2 -> 3 -> 4 -> 2 -> 6}) should return true, since it can be cut into three segments {1 -> 2 -> 3}, {4 -> 2}, and {6} with equal element sum (6).

separable({1 -> 2 -> 3 -> 4 -> 2 -> 1 -> 2}) should return false, since it cannot be cut into three segments with equal element sum. Note that {1 -> 4}, {2 -> 3}, {2 -> 1 -> 2} is not a valid cut, since {1 -> 4} is not a continuous sublist of the original list.

Hints for separable function:

- First, traverse the linked list once to calculate the sum of all nodes values, denoted as ***total_sum***. If ***total_sum*** is not divisible by 3, return false. If it is divisible, calculate ***segment_sum*** = ***total_sum*/3**.
- Then, traverse the linked list again to determine whether there are three segments, where the sum of node values in each segment is equal to ***segment_sum***.
- This problem does **not require** the use of recursion; it can be solved by iterating through the linked list using loop.

2. String or Sorting(15pts)

Write a function:

void wordAccount(string *line*, Map <string, int> & *wordAcc*);

which calculates the number of different (English) words in a string, stores the results in the Map, and the calculation is not case sensitive, i.e., That is, if only the letter case of two English words is different, they will be considered as the same word. Here, it is required to standardize the capitalization of words, which means changing the first letter of the word to uppercase and all other letters to lowercase.

Example:

Input string ***line*** is "Dog apple App App dog".
Result should be {<"Apple", 3>,<"Dog", 2>} in the Map ***wordAcc.***

```
// The function wordAccount recognizes the words in the string line one by one
// Then calls the Captalize function (which has been given) to convert each word into standard format
// Finally writes each word and its occurrence times into wordAcc.
void wordAccount(string line, Map <string, int> & wordAcc)
{
    string word;
   // Fill your code here

}
```

3. Big-O (18pts)

Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N (**the variable N appears in the code** so use that value). As a reminder, when doing Big-O analysis, we write a simple expression that gives only a power of N, such as O(N^2) or O(log N), *not* an exact calculation. It may be helpful to remember this math identity: $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$.

Three Questions (6pts each)	

(a)	`int addInteger(int N){`
	`    int sum = 0;`
	`    for (int i = 1; i <= N + 2; i++) {`
	`        sum++;`
	`    }`
	`    for (int j = 1; j <= N *5; j++) {`
	`        sum++;`
	`    }`
	`    return sum;`
	`}`

(b)	`int showString(string  str){`
	`    int N = str.size();`
	`    string sumString;`
	`    for(int i = 1; i < str.size(); i *= 3) {`
	`          cout << str[i] << endl;`
	`          sumString += str[i];`
	`    }`
	`    return sumString.size();`
	`}`

(c)	`int myfunction(Vector<int> vec){`
	`    int N = vec.size();`
	`    int sum = 0;`
	`    for(int i = 0; i < vec.size(); i += (vec.size() / 6)) {`
	`          cout << vec[i] << endl;`
	`          sum += vec[i];`
	`    }`
	`    return sum;`
	`}`

Note: Similar to the previous one, this problem does not require programming. Please evaluate the runtime complexity of the ***3*** code fragments in the table above, and modify the definition of ***BigO string array*** in ***3.cpp*** with your solutions.

4. ADT-1&Recursion (15pts)

In this problem, you are asked to write a function that, given *n* positive integers $a_1, \dots, a_n$ and another positive integer *W*, decides whether there is a subset of $a_1, \dots, a_n$ whose sum is equal to *W*.

The signature of your function should be:

bool subsetSum(const Vector<int> & *a*, int *W*);

The input vector *a* contains the integers $a_1, \dots, a_n$

Example:

On the input *a* = {1, 3, 5, 9, 20} and *W* = 28, your function should return ***true***, because 3 + 5 + 20 = 28.

On the input *a* = {1, 3, 5, 9, 20} and *W* = 7, your function should return ***false***.

- You should use a **recursion** to solve the problem.
- You are free to write helper functions.

5. ADT-2&Recursion (17pts)

The goal of this problem is to write a function:

Set<Vector<string>> splitsOf(const string& *str*);

that takes as input a string, then returns all ways of splitting that string into a sequence of nonempty strings called pieces. For example, given the string "RUBY", you'd return a ***Set*** containing these **Vector<string>s**; notice that all letters are in the same relative order as in the original string:

{"R", "U", "B", "Y"}
{"R", "U", "BY"}
{"R", "UB", "Y"}
{"R", "UBY"}
{"RU", "B", "Y"}
{"RU", "BY"}
{"RUB", "Y"}
{"RUBY"}

If you take any one of these **Vectors** and glue the pieces together from left to right, you'll get back the original string "RUBY". Moreover, every possible way of splitting "RUBY" into pieces is included here. Each character from the original string will end up in exactly one piece, and no pieces are empty.

Hints: The decision tree for listing subsets is found by repeatedly considering answers to questions of the form "should I include or exclude this element?" The decision tree for listing permutations is found by repeatedly considering answers to questions of the form "which element should I choose next?" In this problem, the decision tree is found by repeatedly considering answers to this question: ***How many characters from the front of the string should I include in the next piece***? Based on this insight, draw the decision tree for listing all splits of the string "JET" along the lines of the decision trees we drew in class. At a minimum, please be sure to do the following:

- Label each entry in the decision tree with the arguments to the **recursive call** it corresponds to.
- Label each arrow in the decision tree with what choice it corresponds to. Now, implement the ***splitsOf*** function. For full credit, your implementation must be **recursive** and match the decision tree of this problem. **You are free to write helper or wrapping functions**.

For starters, here's the decision tree for all splits of "JET":

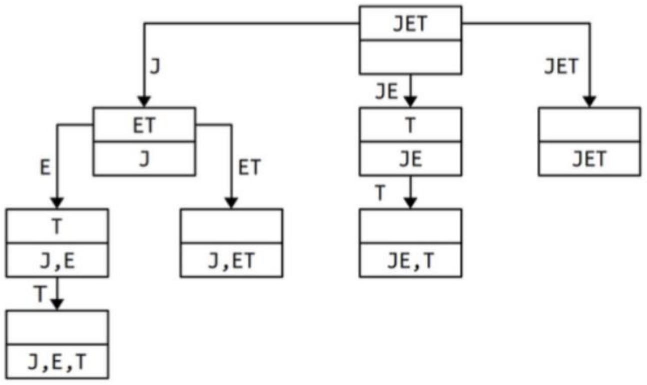


Fig.1 Decision Tree for splits of "JET"

Here, each decision is of the form "how many characters are we taking off the front of the string?" and we pass down through the recursion both the characters we haven't used yet and the split we've built up so far.

6. Trees & Recursion (18pts)

A binary tree is a tree where each node has up to two children. A node for this tree is defined as:

```
struct Node {
   int value;
   Node * left;
   Node * right;
};
```

Two binary trees are said to be isomorphic(同构) if they have the same structure and have the same values at every pair of corresponding nodes. For example, the two trees are isomorphic in **Fig.2** on the right side.

The following two trees in **Fig.3(a)** are not isomorphic because they do not have the same structures. In **Fig.3(b)**, the two trees are not isomorphic because the left children of the roots do not have the same values.

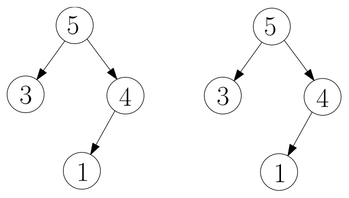
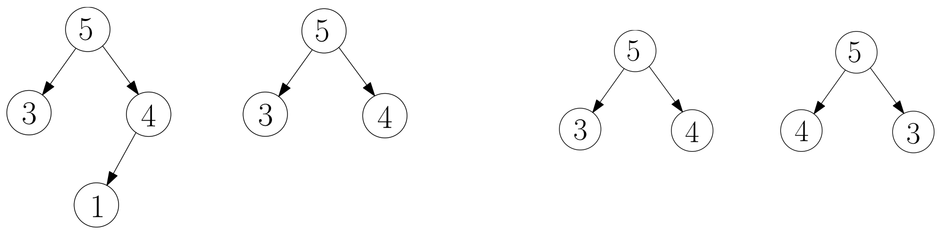


Fig.2 Isomorphic Trees

(So, our definition of isomorphism does not allow renaming left and right children.)



(a) With different structures **(b)** With different values of children

Fig.3 Non-isomorphic Trees

In this problem, you are asked to write a function that, given two trees A and B, decides whether tree B is isomorphic to any subtree of tree A. Your function signature should be:

bool containsSubtree(Node * rootA, Node * rootB);

- You should solve this problem using **recursion**.
- Unlike Binary Search Trees, no ordering rule governs the values in the trees.

Example:

In **Fig.3**, two instances where your function should return ***false***.

An instance for which your function should return ***true***,

see **Fig.4** on the right side.

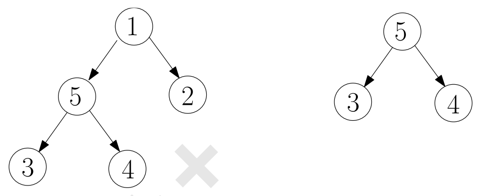


Fig.4 Isomorphic Trees

Summary of Relevant Functions and Data Types

We tried to include the most relevant member functions for the exam, but not all member functions are listed. You are free to use ones not listed here that you know exist.

```
int isalpha(int ch); // Check if the given character is an alphabetic character
int isspace(int c); // Check if the given character is an white space such as Space, '\t', '\n',etc.

class string {
  bool empty() const; // O(1)
  int size() const; // O(1)
  int find(char ch) const; // O(N)
  int find(char ch, int start) const; // O(N)
  string substr(int start) const; // O(N)
  string substr(int start, int length) const; // O(N)
  char& operator[](int index);  // O(1)
  const char& operator[](int index) const; // O(1)
};
string toUpperCase(string str);
string toLowerCase(string str);

class Vector {
  bool isEmpty() const; // O(1)
  int size() const; // O(1)
  void add(const Type& elem); // operator+= used similarly - O(1)
  void insert(int pos, const Type& elem); // O(N)
  void remove(int pos); // O(N)
  Type& operator[](int pos); // O(1)
};

class Grid {
  int numRows() const; // O(1)
  int numCols() const; // O(1)
  bool inBounds(int row, int col) const; // O(1)
  Type get(int row, int col) const; // or operator [][] also works -  O(1)
  void set(int row, int col, const Type& elem); // O(1)
};

class Stack {
  bool isEmpty() const; // O(1)
  void push(const Type& elem); // O(1)
  Type pop(); // O(1)
};

class Queue {
 bool isEmpty() const; // O(1)
 void enqueue(const Type& elem); // O(1)
 Type dequeue(); // O(1)
};

class Map {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void put(const Key& key, const Value& value); // O(logN)
 bool containsKey(const Key& key) const; // O(logN)
 Value get(const Key& key) const; // O(logN)
 Value& operator[](const Key& key); // O(logN)
};
```
Example for Loop: `for (Key key : mymap){…}`

```
class HashMap {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void put(const Key& key, const Value& value); // O(1)
 bool containsKey(const Key& key) const; // O(1)
 Value get(const Key& key) const; // O(1)
 Value& operator[](const Key& key); // O(1)
};
```
Example for Loop: `for (Key key : mymap){…}`

```
class Set {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void add(const Type& elem); // operator+= also adds elements – O(logN)
 bool contains(const Type& elem) const; // O(logN)
 void remove(ValueType value); // O(logN)
};
```
Example for Loop: `for (Type elem : mymap){…}`

```
class Lexicon {
  int size() const; // O(1)
  bool isEmpty() const; // O(1)
  void clear(); // O(N)
  void add(string word); // O(W) where W is word.length()
  bool contains(string word) const; // O(W) where W is word.length()
  bool containsPrefix(string pre) const; // O(W) where W is pre.length()
};
```

Example for Loop: `for (string str : english){…}`