Basic

Problem 2, 3, 4 - Sorting (2.5%)

For Problem 2, Problem 3 and Problem 4, you need to create a header file (*.h) for your functions. Functions implementations should be in another separated cpp file.

Note: Those who fail to provide a header file and another separated cpp file will be penalized with negative scores.

Preliminaries

What will you be given?

For Problem 2, in the zip file you download from Ceiba, you will find **EIGHT** template files:

1. `basic.h`
2. `basic.cpp`
3. `quickSort.cpp`
4. `quickSort.h`
5. `mergeSort.cpp`
6. `mergeSort.h`
7. `mergeSortInplace.cpp`
8. `mergeSortInplace.h`

What you should submit?

You should submit ONE zip file, which consisting SIX files:

1. `basic.cpp`: implementation of
    1. `createArray`
    2. `swap`
    3. `shuffle`
    4. `printArray`
2. `basic.h`: declaration of
    1. `createArray`
    2. `swap`

    3. `shuffle`
    4. `printArray`
3. `quickSort.cpp`: implementation of
    1. `partition`
    2. `quickSort`
4. `quickSort.h`: declaration of
    1. `partition`
    2. `quickSort`
5. `mergeSort.cpp`: implementation of
    1. `merge`
    2. `mergeSort`
6. `mergeSort.h`: declaration of
    1. `merge`
    2. `mergeSort`
7. `mergeSortInplace.cpp`: implementation of
    1. `mergeInplace`
    2. `mergeSortInplace`: declaration of
8. `mergeSortInplace.h`
    1. `mergeInplace`
    2. `mergeSortInplace`

All modifications must be done inside the `/*your code here*/` segment, any modification outside the allowed area is **strictly forbidden** and will result in **negative scores**.

Note: For all problems in Problem 2, please turn in your code **without** modifying the filename. **Those who modify the filename will be penalized with negative scores.**

How your code will be score?

There are THREE main `.cpp` files writen by TAs which will be used to score your code. Please see section 7（7. 作業繳交規格）in the auxilary file.

What is sorting?

In computer science, a sorting algorithm is an algorithm that puts n distinct integers in increasing order. For example, given the sequence $3, 1, 4, 2, 5$, the sorted sequence is $1, 2, 3, 4, 5$.

How to sort?

A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation that determines which of two elements should occur first in the final sorted list. If we sort a list of numbers by pairwise comparison, then we say such comparison method is comparison-based. In this problem, we will implement two well-known comparison sorting method: *mergesort* and *quicksort*.

A step-by-step guide to sorting

In this subsection, we will implement four functions

- `createArray` - to create an array from 0 to $n-1$;
- `shuffle` - to shuffle the array you just created;
- `swap` - to swap two elements in an array;
- `printArray` - to print out your array.

All four functions are foundations to implement quicksort and mergesort (both noninplace and inplace). We use `createArray` to create an array in order, and then use `shuffle` to shuffle the array randomly. By doing so, we can create our own testing data. Since we are doing comparison sorting, the `swap` function is needed. Finally, to see the state of the array, we need the `printArray` function. Let's examine these four functions one by one. First of all, let's create our own array of numbers.

Create your array

Specification

- **Function**: `int* createArray(int* A, int size)`
- **Discription**: create an array `A[0,...,size-1]`
- **Arguments**:
  - `int* A`: a pointer pointing to a array `A`
  - `int size`: an integer indicating the size of the array `A`
- **Return**: the pointer pointing to the array `A`

Discription

This function create an array consisting of numbers ranging from `0` to `size-1`. For example, the following code snippet creates an array called `A` consisting of numbers from 0 to 5:

```
// define the size of the array
int size = 5;

// allocate a memory space for creating a new array and name the array A
int* A = new int[size];

// create an array
A = createArray(A, size);

// release the memory space occupied by the array
delete [] A;
```

After the statement `A = createArray(A, size);`, an array `A[0, 1, ..., 4]` is created. Notice that this array starts from 0 and ends with 4, and the size of the array is 5 as required. Don't worry too much about the statement `int* A = new int[size];` and `delete [] A;`. They are operations on *pointers* and we'll be taught in great detail in further classes. For now, just view them as standard statements to create a dynamic array, i.e., an array that has no fixed size. All you have to do is finish the `createArray` function. The template of this function is given as below. Make sure you only modifiy the `/* your code here */` part; otherwise your code will not be graded.

```
int* createArray(int* A, int size){
    /*
    your code here
    */
    return A;
}
```

After creating your own array of size `size`, let's see how we can swap two elements of the array. The `swap` function is very useful and easy to implement, and can be used in lots of senerios. Besides, there are several ways to implement such function. Let's continue our journey to implement the `swap` function.

Swap

Specification

- **Function**: `void swap(int* A, int i, int j)`
- **Discription**: shuffle the array `A`
- **Arguments**:
  - `int* A`: a pointer pointing to a array `A`
  - `int i`: the index `i` of the array
  - `int j`: the index `j` of the array

- **Return**: `void`, that is, nothing is returned

Discription

Given an array `A`, the function `swap(int* A, int i, int j)` swaps the value of `A[i]` and `A[j]`. For example, if `A[0]==10` and `A[4]==20`, then after doing `swap(A, 0, 2)`, we have `A[0]==20` and `A[4]==10`. Eventhough there are numerous ways to implement the swap functions, you are required to abide by the above specification and finish the `/* your code here */` part of the below code snippet.

```
void swap(int* A, int i, int j){
    /*
    your code here
    */
}
```

Now, we know how to create an array and swap two element of the array. Next, we will implement a shuffle function, which shuffles an array randomly. By doing so, we create our own testing data for our mergesort algorithm and quicksort algorithm.

Shuffle your array

Specification

- **Function**: `void shuffle(int* A, int size)`
- **Discription**: shuffle the array `A`
- **Arguments**:
  - `int* A`: a pointer pointing to a array `A`
  - `int size`: an integer indicating the size of the array `A`
- **Return**: `void`

Description

To alleviate your brain loading, you are given a pseudocode to shuffle an array:

```
shuffle(A, size):
    For i = size-1 down to 1:
        set j to be a random number in {0, ..., i - 1}
            swap A[i] and A[j]
```

A pseudocode is a step-by-step recipe using programming-like language to describe algorithms. Show your best to translate the above pseudocode in C++ language. Again, only write your code in the `/* your code here */` part of the below code. Note that nothing is to be returned.

```
void shuffle(int* A, int size){
    /*
    your code here
    */
}
```

Here is some hint for implementing the `shuffle` function. Use the `rand()` function to set a random number, and pay attention to the range of the random numbers you're setting, which should be in $\{0, ..., i-1\}$. Also, DO NOT set a random seed in your code. A random seed will be given by TAs when testing your function. If you implement the code exactly following the given pesudocode given, the result should be unique with respect to a given random seed.

We've implement `createArray`, `swap`, and `shuffle`. But, how do we know that our implementation is valid? A straight forward way is to print our the array! For this reason, let's implement a print function to print out the result of an arrar.

Print your array

Specification

- **Function**: `void printArray(int* A, int size)`
- **Discription**: print the array `A`
- **Arguments**:
  - `int* A`: a pointer pointing to a array `A`
  - `int size`: an integer indicating the size of the array `A`
- **Return**: `void`

Description

The `printArray` function takes in an array and the size of the array as arguments, and print out the content of the array sequentially. Seperate each printed number with a whitespace. For example, if `A[4] = {0, 1, 2, 3}`, then `printArray(A, 4)` will output `0 1 2 3`. The templete of this function is given as follow.

```
void printArray(int* A, int size){
    /*
    your code here
    */
}
```

Remark

Congratulations! We've just implemented four basic functions for further implementation of the mergesort and quicksort. Now, you have equipped with four basic weapons to conquer the merge sort algorithm and the quick sort algorithm. Both sorting methods apply the divide-and-conquer technique, and you will be guide with care to tackle these problem using "divide-and-conquer" thanks to TAs. Have faith, have fun, and here we go!