The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated..

PSet 03 (Part 2) – Template

Table of Contents

Getting Started – files provided

recursion.cpp – a skeleton code

recursionDriver.cpp – to test recursion.cpp

binsearch.cpp – a skeleton code

mergesort.cpp – a skeleton code

recursionx, recursionx.exe – an executable on Mac and Windows

binsearchx, binsearchx.exe – an executable on Mac and Windows

binsearchTx, binsearchTx.exe – an executable on Mac and Windows

mergesortx, mergesortx.exe – an executable on Mac and Windows

mergesortTx, mergesortTx.exe – an executable on Mac and Windows

C++ Function Template

We've learned how to write a few sort functions and function pointers that help make programs easier to write, safer, and more maintainable. While sort functions and function pointers are powerful and flexible tools for effective programming, in certain cases they can also be somewhat limiting because of C++'s requirement that you specify the type of all parameters.

For example, let's suppose that you already developed a bubble sort program to sort integer numbers.

```
void bubbleSort(int *list, int n) {
```

```
   for (int i = 0; i < n - 1; i++) {
          for (int j = 0; j < n - i - 1; j++)
             if (list[j + 1] < list[j])
                swap(list[j + 1], list[j]);
   }
}
```

This function would work great -- for integers. What happens later when you realize your bubbleSort() function needs to work with doubles? Traditionally, the answer would be to overload the bubbleSort() function and create a new version that works with doubles:

```
void bubbleSort(double *list, int n) {
   for (int i = 0; i < n - 1; i++) {
          for (int j = 0; j < n - i - 1; j++)
             if (list[j + 1] < list[j])
                swap(list[j + 1], list[j]);
   }
}
```

Note that the code for the implementation of the double version of bubbleSort() is exactly the same as for the int version of bubbleSort()! What happens if you want to sort characters? You're stuck writing one function for each type you wish to use.

Wouldn't it be nice if we could write one version of bubbleSort() that was able to work with parameters of ANY type?

Welcome to the world of templates.

What is a function template?

In C++, function templates are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**.

Once we have created a function using these placeholder types, the compiler can create multiple "flavors" of a function from one template!

Creating function templates in C++

Let's take a look at an int version of max() function:

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}
```

Note that there are 3 places where specific types are used: parameters x, y, and the return value all specify that they must be integers. To create a function template, we're going to replace these specific types with placeholder types. In this case, because we have only one type that needs replacing (int), we only need one template type parameter.

You can name your placeholder types almost anything you want, so long as it's not a reserved word. However, in C++, it's customary to name your template types the letter T (short for "Type").

Here's our new function with a placeholder type:

```
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

This is a good start -- however, it won't compile because the compiler doesn't know what "T" is!

In order to make this work, we need to tell the compiler two things: First, that this is a template definition, and second, that T is a placeholder type. We can do both of those things in one line, using what is called **a template parameter declaration:**

```
template <typename T>      // this is the template parameter declaration
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

Believe it or not, that's all we need. This will compile!

If the template function uses multiple template type parameter, they can be separated by commas:

```
template <typename T1, typename T2>
// template function here
```

One final note: Because the function argument passed in for type T could be a class type, and it's generally not a good idea to pass classes by value, it would be better to make the parameters and return types of our templated function const references:

```
template <typename T>
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}
```

Using function templates

Using a function template is extremely straightforward -- you can use it just like any other function. Here's a full program using our template function:

Example 1:

```
#include <iostream>

template <typename T>
T max(const T x, T y) {
    return (x > y) ? x : y;
}

int main() {
    int i = max(3, 9);            // returns 9
    std::cout << i << '\n';

    double d = max(2.34, 5.67);   // returns 5.67
    std::cout << d << '\n';
```

```
    char ch = max('a', 'z');      // returns 'z'
    std::cout << ch << '\n';
}
```

Example 2:

```
#include <iostream>

template <typename T>
void bubbleSort(T *list, int n) {
  for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) // last i elements are already in place
           if (list[j + 1] < list[j])
               swap(list[j + 1], list[j]);
  }
}

template <typename T>
void print_list(T *list, int n) {
  for (int i = 0; i < n; i++) cout << list[i] << " ";  cout << endl;
}

int main() {
  // int list[] = { 3, 4, 1, 7, 0, 9, 6, 5, 2, 8};
  char list[] = {'b', 'u', 'b', 'b', 'l', 'e', 's', 'o', 'r', 't'};
  int N = sizeof(list) / sizeof(list[0]);

  cout << "UNSORTED: " << endl;
  print_list(list, N);

  bubbleSort(list, N);

  cout << "BUBBLE SORTED: " << endl;
  print_list(list, N);
  cout << "Happy Coding~~\n";
}
```

About binary search

The binary search algorithm is a method of searching a sorted array for a single element by cutting the array in half with each recursive pass. The trick is to pick a midpoint near the center of the array, compare the value at that point with the key being searched and then responding to one of three possible conditions: the key is found at the midpoint, the data at the midpoint is greater than the data being searched for, or the data at the midpoint is less than the key being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically, the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

Step 1: Implement binsearch.cpp

Use a skeleton code binsearch.cpp provided first to implement the recursive binary search.

In this code, a user may want to simply call the function with two parameters such as **`binary_search(list, size)`** at user's convenience. The **`binary_search(list, size)`** calls binary_search(data, key, lo, hi) "size" number of times while generating a new random key at every

call of the function, and displays the results. If the key is found in the list, it displays its index in the list. If the key is not found, it displays where it is supposed to be appeared if there is one. The range of random key values are between min and max of the list.

As you know, **binary_search(list, key, lo, hi)** is a recursive function. As you notice that all the recursive operations will be done in **binary_search(list, key, lo, hi)** with four parameters, not in **binary_search(list, size)**.

Check the following code and implement it in **binsearch.cpp** skeleton code provided. You may test this functionality first before you proceed the Step 1.

Code:

```
/*
   This implements a binary search recursive algorithm.
   INPUT: list is an array of integers SORTED in ASCENDING order,
          key is the integer to search for,
          lo is the minimum array index,
          hi is the maximum array index
   OUTPUT: an array index of the key found in the list
        if not found, return a modified index where it could be found.
*/

int binary_search(int *list, int key, int lo, int hi) {
  DPRINT(cout << "key=" << key << " lo=" << lo << " hi=" << hi << endl;);

  cout << "your code here \n";

   return 0;
}

// randomly generate a key to search between list[0] and list[size-1].
int get_a_key(int *list, int size) {
  int key = rand() % (list[size - 1] + 1 - list[0]) + list[0];
  return key;
}

// calls binary_search(data, key, lo, hi) "size" number of times
// while generating a new random key at every call of the function.
// and also displays the results. If the key is found in the list,
// it displays its index in the list. If the key is not found, it
// displays where it is supposed to be appeared if there is one.
void binary_search(int *list, int size) {
  DPRINT(cout << ">binary_search: size=" << size << endl;)
  int key = get_a_key(list, size);
  int idx = binary_search(list, key, 0, size);

  cout << "your code here \n";

  DPRINT(cout << "<binary_search\n";)
}

#if 1
int main(int argc, char *argv[]) {
  int list[] = { 0, 1, 4, 6 };
  // int list[] = { 3, 5, 6, 8, 9, 11 };

  int size = sizeof(list) / sizeof(list[0]);
  srand((unsigned)time(nullptr));    // turn off this line during debugging

  cout << "  list: ";
```

```
  for (auto x: list) cout << x << " ";    cout << endl;
  for (auto x: list) binary_search(list, size);
}
#endif
```

The **main()** function is provided to test two **binary_search()** functions.

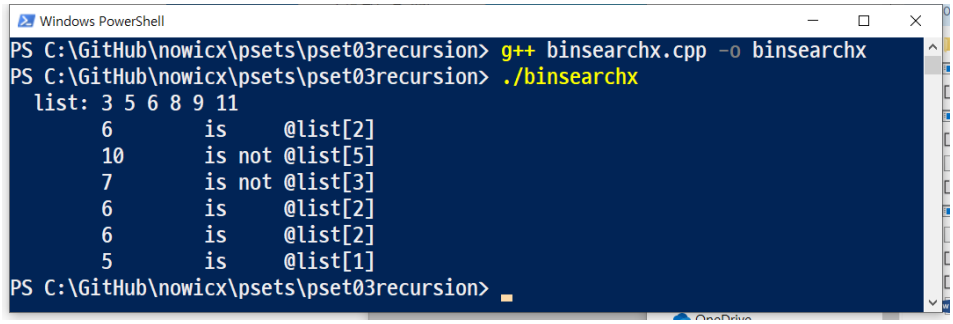
- If you want to find how to use **rand()**, **srand()** and a random number between a and b, refer to this web site. In Korean.

How to test the algorithm:

We want to generate random numbers or keys to test **binary_search().**

- The values of key should be between the first and last values of the list or or [3 .. 11] in this case.
- Repeat the test by the number of elements in the list.
- Display the result as shown below:

Sample Run:



- The first line shows that 6 is found at the list[2]. The second line in this result shows that 10 is not found. If it is found, it should be found at the list[5] and so on.

Take-away: How long does the `binary_search(list, key, lo, hi)` take with n items?

In one call to **binary_search()**, we eliminate at least half the elements from consideration. Hence, it takes $log_2 n$ (the base 2 logarithm of n) **binary_search()** calls to pare down the possibilities to one. Therefore **binary_search()** takes time proportional to $log_2 n$.

Step 2: binsearchT.cpp – a template version of binsearch.cpp

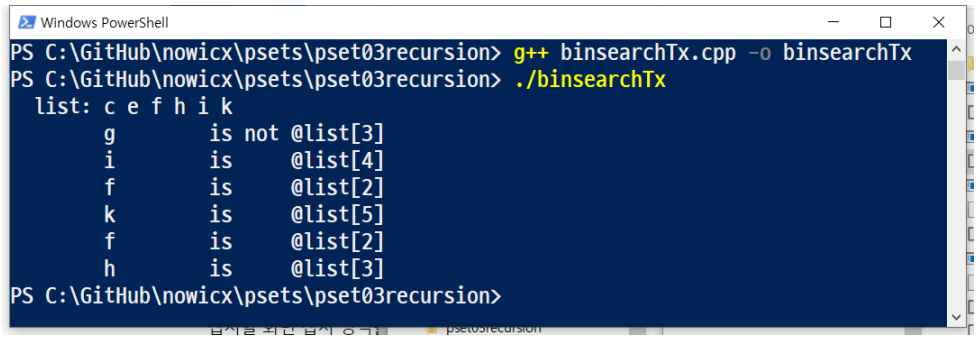
Now we want to implement a template version of binsearch.cpp that works with integer numbers as well as character data as well. This new version, **binsearchT.cpp** should work for either char type **list** or int type **list** without modification of the binary search functions at all except the **list** data type in the main().

The main() function shown below does not change except the input list data in either char or int type. Without changing anything in binary search functions in the file should work with the following main() even though the user uses either char list[] or int list[].

```
#if 1
int main(int argc, char *argv[]) {
  // char list[] = { 'a', 'b', 'e', 'g' };
  char list[] = { 'c', 'e', 'f', 'h', 'i', 'k' };
  // int list[] = { 0, 1, 4, 6 };
  // int list[] = { 3, 5, 6, 8, 9, 11 };

  int size = sizeof(list) / sizeof(list[0]);
  srand((unsigned)time(nullptr));    // turn off this line during debugging
```

```
  cout << "  list: ";
  for (auto x: list) cout << x << " ";    cout << endl;
  for (auto x: list) binary_search(list, size);
}
#endif
```

Sample Run:

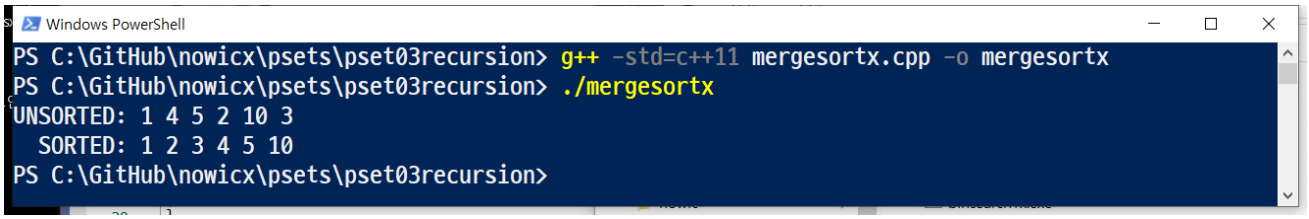


- The first line in this result shows that 'g' is not found. If it were found, it should be found at the list[3] The second line shows that 'i' is found at the list[4] and so on.

Step 3: mergesort.cpp – a template version of mergesort algorithm

Using a skeleton code provided with mergesort.cpp, complete a merge sort algorithm. Most of code for merge sort algorithm has been provided during the lecture. Don't change the function signatures. You may make it work with int list data type.

Sample Run:



Step 4: mergesortT.cpp – a template version of mergesort.cpp

Once you complete the coding of mergesort.cpp, make a copy of mergesort.cpp into mergesortT.cpp. Then modify it such that it works with char data types as well.

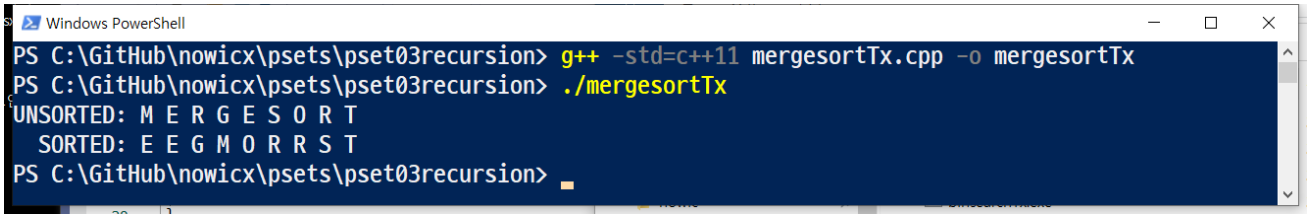
Complete the **mergesortT.cpp** such that it works the following main(). Use the function template such that it works both for `char list[]` or `int list[]` in the `main()`. The main() function should not change except the input `list` data in either char or int type. Without changing anything in merge sort functions should work with `main()` even though the user set either `char list[]` or `int list[]`.

mergesortT.cpp

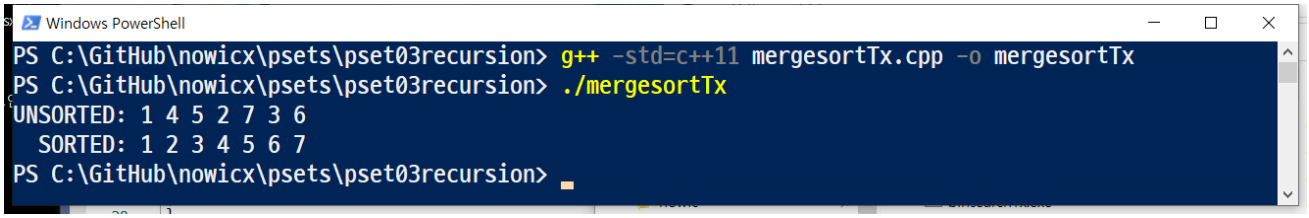
```
#include <iostream>
using namespace std;

#include <iostream>
#include <cassert>
using namespace std;

template <typename T>
int isSorted(T *a, int i, int j) { return a[i] <= a[j]; }
```

```
// your code here for the following three functions using template
// void merge(T *a, T *aux, int lo, int mi, int hi)
// void mergeSort(T *a, T *aux, int N, int lo, int hi)
// void mergeSort(T *a, int N)


int main() {
    char list[] = {'M','E','R','G','E','S','O','R','T'};
    // int list[] = { 1, 4, 5, 2, 7, 3, 6 };
    int N = sizeof(list) / sizeof(list[0]);

    cout << "UNSORTED: ";
    for (auto x: list) cout << x << " "; cout << endl;

    mergeSort(list, N);

    cout << "  SORTED: ";
    for (auto x: list) cout << x << " "; cout << endl;
}
```

Sample Run: with char list[], but without changing anything else.



Sample Run: with int list[], but without changing anything else.



Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____ Section: _____ Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the problem sets partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again if it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit and Grade

Submit the following files.

- **binsearch.cpp – 1 point**
- **binsearchT.cpp – 1 point**
- **mergesort.cpp – 1 point**
- **mergesortT.cpp – 1 point**

Upload your file **pset3** folder in Piazza.

Due

- Due: 11:55 pm