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 03b: Binary Search and More

Table of Contents

Getting Started

Assume that you have implemented a recursive binary search with two functions in binsearch.cpp as shown below:

```
// binsearch.cpp implements the following functions.

int binary_search(list, key, size);
int _binary_search(list, key, lo, hi);
```

Now, we want to implement an application program to test the algorithm while learning about std::vector class object, recursion, **streaming and piping**.

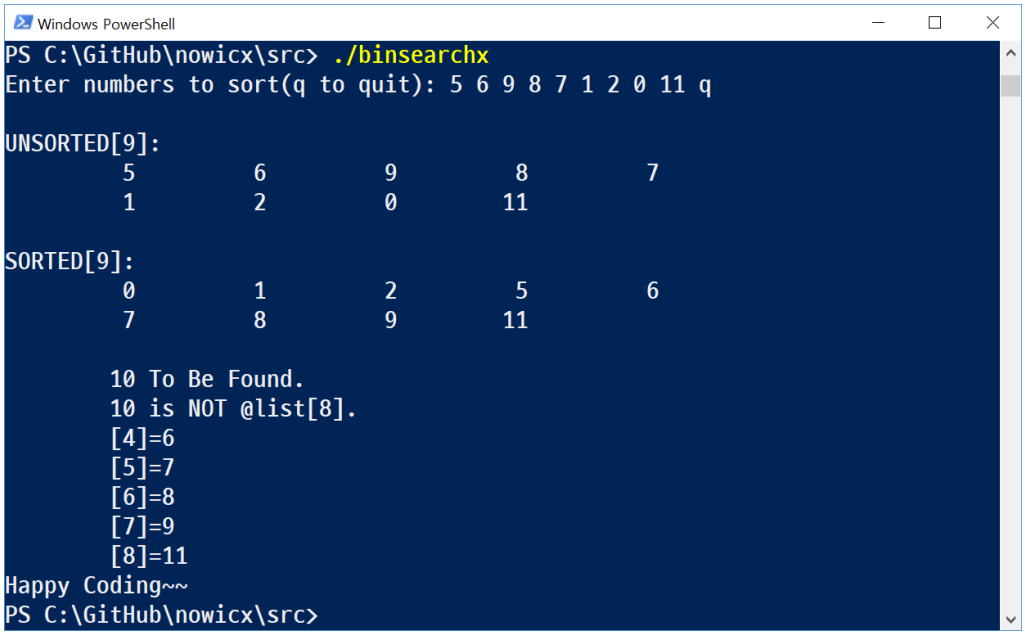
Files Provided:

- pset04binsearch.pdf – this file
- binsearchDriver.cpp – a skeleton code
- binsearch.cpp – a skeleton code
- binsearchx.exe, randomN.exe – a complete implementation of this problem sets
- printList.cpp – a complete code provided to compile with
- randomN.cpp, quicksort.cpp – you need to prepare by yourself.

Step 1: Read, Store and Sort

Run the sample program provided.

Sample Run:
To finish entering numbers, enter non-numeric character. For example, `q` for quit.



Read the program and understand what the code does. This program **binSearchDriver.cpp** will read input values from a console and store them in a list dynamically.

Step 2: Add a line

Read about std::vector class in

- [C-Tutorial] (https://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4027/C-Tutorial-A-Beginners-Guide-to-stdvector-Part-1.htm)
- [cplusplus](http://www.cplusplus.com/reference/vector/vector/?kw=vector).

Add **one line** which inserts user's input to the list.

Step 3: Recursive Binary Search

(1) Now we want to generate a random number and find out whether or not it is in the list. Use **binsearch.cpp** provided with you to complete two functions. The users usually calls **binary_search()** which invokes **_binary_search()**. Most of recursive work will be done in **_binary_search()**. Declare the following prototype in binSearchDriver5.cpp.

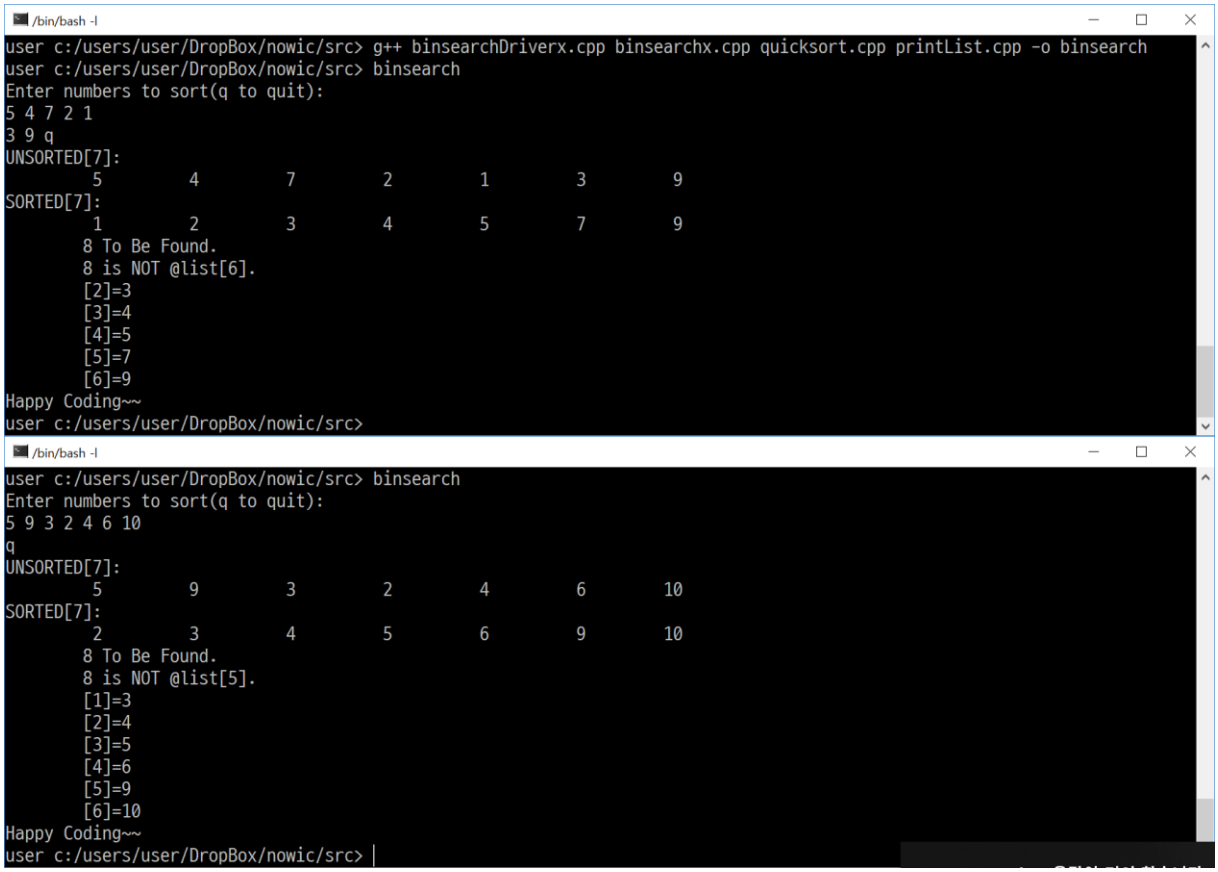
```
int binary_search(int *list, int toFind, int size);
int _binary_search(int *list, int toFind, int start, int end);
```

When you generate a random number to search in the sorted list, the range of the number should from 0 to the maximum number in the list.

(2) Print a few (3 ~ 4) elements of the list around the number you **searched** or **missed** as shown below. To complete this functionality, you need to modify **_binary_search()** such that it returns something useful, instead of -1, when it could not find the key.

(3) Remove some of debugging lines if necessary.

Here are some sample runs.



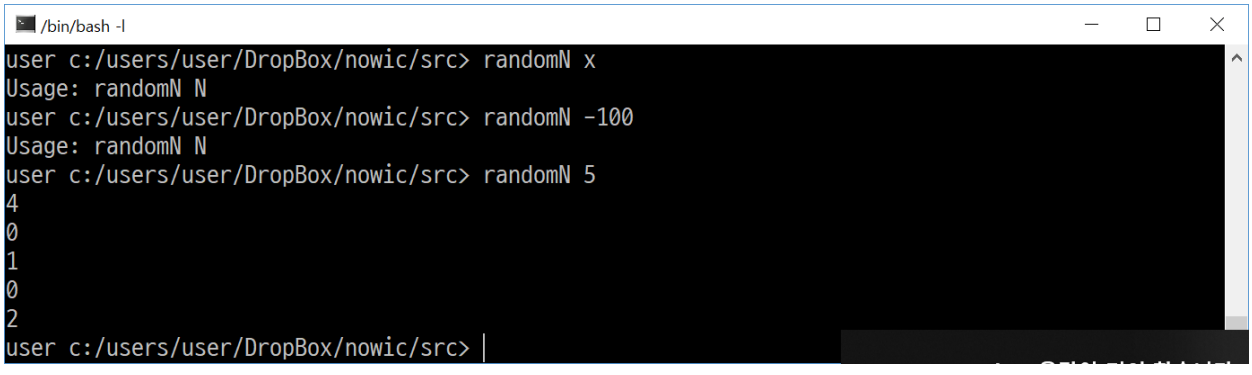
Step 4: RandomN

To prepare a binary search testing, we implement a utility program called **randomN**. It takes a command-line argument N and prints N random numbers in [0, N), line by line. Implement this functionality in **randomN.cpp**, a skeleton code provided with you.

Notice: If you use **rand()** to generate random numbers, it may good enough because of its range of the value it produces. **rand()** returns a random number between 0 and **RAND_MAX. RAND_MAX** is surprisingly small (but guaranteed at least 32767), not 1 Million nor 1 Billion) and depends on the implementation. You may expand it by using the following algorithm:

1) Take two random numbers `a, b`
2) Calculate `a * (RAND_MAX + 1) + b`
This will generate random values in the range `[0, (RAND_MAX+1)*(RAND_MAX+1)]`.

Sample Run: ran



Step 5: Using pipe or redirection

In this step, you may not have nothing to code if you have worked the problem set properly so far. You simply use the programs we have implemented so far.

If you use Windows, do Step 7 at the console created by clicking C:\MinGW\msys\1.0\msys.bat. It is recommended that you use mintty and bash, neither cmd nor PowerShell of Windows.

Based on the Unix or Unix like systems such as Linux or OSX:

- **Redirection:** The shell interprets the symbols `<`,`>`, and `>>` as instructions to reroute a **command's** input or output to or from a **file.**
- **Pipes:** To connect the `STDOUT` of one **command** to the `STDIN` of **another** use the pipe`|` symbol, commonly known as a pipe.

Every command in the Unix-like system that you start from the shell gets three **channels** assigned:

- **stdin** (channel 0): Where your command draws the input from. If you don't specify anything special this will be your keyboard input.
- **stdout** (channel 1): Where your command's output is sent to. If you don't specify anything special the output is displayed in your shell.
- **stderr** (channel 2): If anything wrong happens the command will send error message here. By default, the output is also displayed in your shell.

For example, the following sequences commands at a console which runs a shell:

```
cat <Enter>                 # nothing seems to happen, but it is waiting input from stdin.
Hello World <Enter>         # On <Enter>, it reads input from stdin
     …                      # It outputs to the stdout
Hello John <Enter>          # On <Enter>, it reads input from stdin
       …
<Ctrl-D>                    # stdin sends an EOF signal by pressing Ctrl-D on an empty line
```

Summary:

>	Save output to a file.
>>	Append output to a file.
<	Read input from a file.
2>	Redirect error messages.
|	Send the output from one program as input to another program.

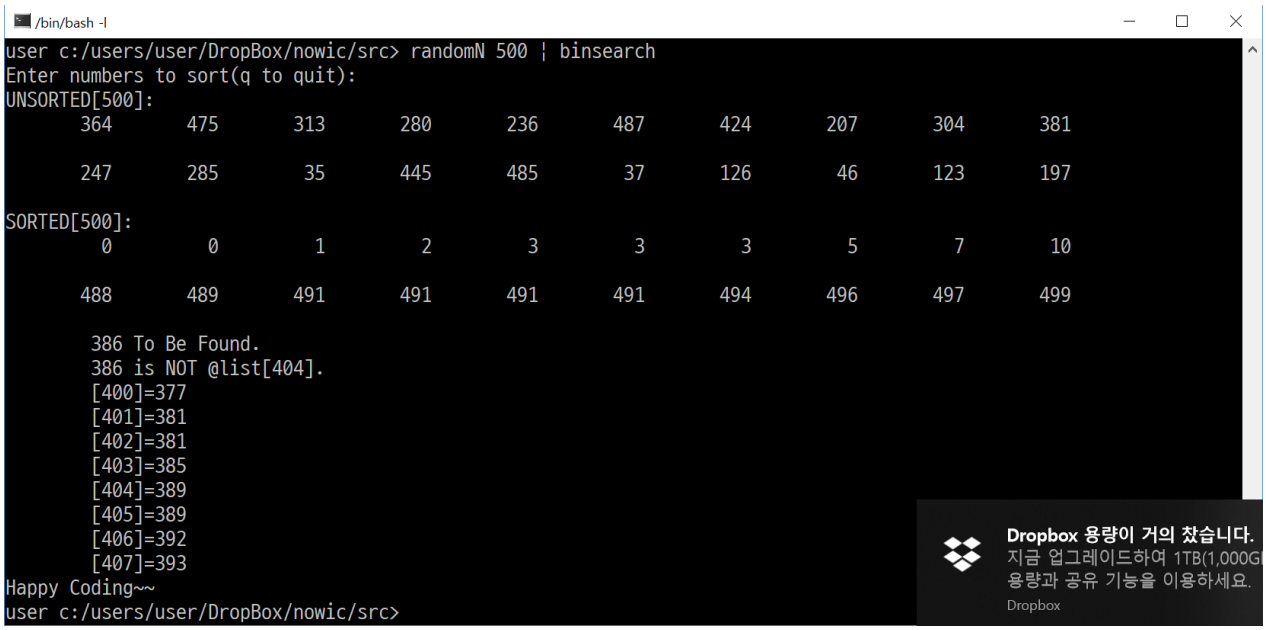
Now you are ready to try the redirection or pipe. Do the following commands at the console. `wc` is a Linux command that counts the number of words and `wc -l` (a lowercase L) counts the number of line.

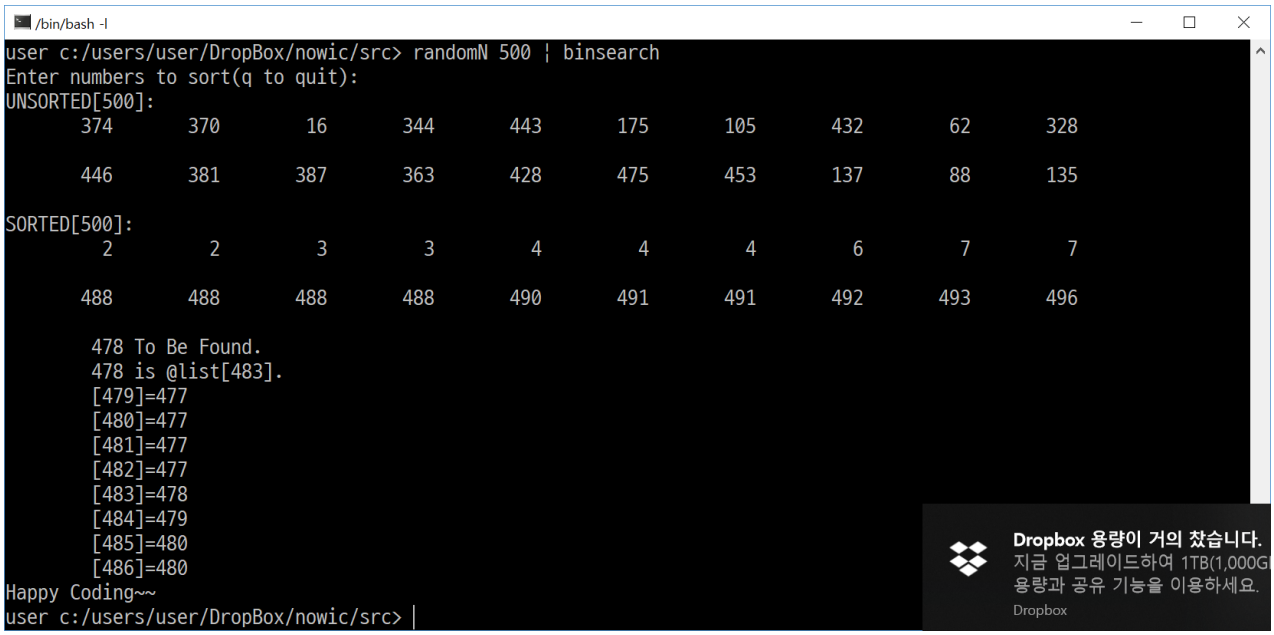
```
randomN 50                          # print 50 random numbers to stdout
randomN 30 | wc -l                 # print 30 random numbers to stdout
# read from stdin and count number of lines

randomN 500 > random500.txt         # print 500 random numbers to stdout, redirect stdout to a file
cat random500.txt                   # print the file to stdout
binsearch < random500.txt           # using redirection
```

You may pipe for the same thing above.

```
randomN 500 | binsearch             # print 500 random numbers to stdout,
                                     # all the output becomes input to stdin
```

You may see something like this:





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 as long as 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. You will submit at most **three** files.

- Part 1: Step 1 ~ 3 (2 Points)
  - **`binsearchDriver.cpp`**
  - **`binsearch.cpp`**
- Part 2: Step 4 ~ 5 (1 Point)
  - **`randomN.cpp`**

Upload your file **hw4** folder in Piazza.
NOTE: Don't forget that you have good comments on your source code.

Due

- Due: 11:55 pm, March **27**, 2019