6th Recitation

Sequential Search, Binary Search, Sorting Algorithms

Instructions:

- Download the file `aed2324_p06.zip` from the Moodle page and unzip it (it contains the folder `lib`; the `Tests` folder with the files `funSortProblem.h`, `funWithSearch.h`, `piece.h`, `product.h`, `funSortProblem.cpp`, `funwithSearch.cpp`, `piece.cpp`, `product.cpp`, `tests.cpp`; and the `CMakeLists` and `main.cpp` files).
- On CLion, create a project by selecting the folder containing the extracted files above.

1. Elementary Search (the goal of this exercise is to test a basic implementation of sequential search, binary search and check their execution times).

1.1 Sequential Search

Function to implement:

```
int FunWithSearch::searchLinear(const vector<int> & v, int key)
```

Expected time complexity: O(*n*).

This function should return the position (index) of the `key` in the vector `v` or -1 if the `key` does not exist in the vector. You can assume that the vector is **sorted incrementally** and that **there are no repeated numbers**.

Example call and expected output:

```
cout << FunWithSearch::search({2,3,5,7,8}, 2) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 8) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 7) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 1) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 10) << endl;
cout << FunWithSearch::search({2,3,5,7,8}, 6) << endl;
```

```
0
4
3
-1
-1
-1
```

Suggestion: (a first solution in time O(*n*))**:** Start by implementing a simple sequential search that goes through all the numbers in the vector trying to find the `key` in linear time. Copy the following code into the function, compile and run it to see how long it takes for each test.

```
for (unsigned i=0; i<v.size(); i++)
  if (v[i] == key)
     return i; // key found at index i
return -1; // key not found
```

1.2 Binary Search (improving to O(log *n*))

Function to implement:

```
int FunWithSearch::searchBinary(const vector<int> & v, int key)
```

Expected time complexity: Θ(log *n*).

The fact that the vector is sorted is essential, as it enables the use of binary search.

```
int low = 0, high = (int)v.size() - 1;
while (low <= high) {
  int middle = low + (high - low) / 2;
  if (key < v[middle]) high = middle - 1;
  else if (key > v[middle]) low = middle + 1;
  else return middle; // key found at index middle
}
return -1; // key not found
```

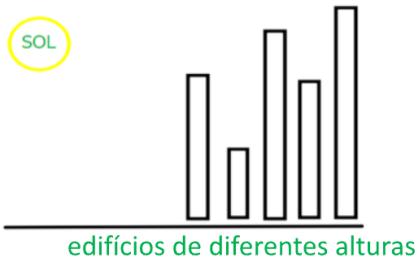
Implement this new solution and see what happens in all the tests. How long has it taken now?

2. Facing Sun

Implement this function using only linear data structures:

```
int FunWithSearch::facingSun(const vector<int> & values)
```

Expected time complexity: O(*n*)



The `values` vector represents the height of buildings in a given street, arranged in the vector by door number. Assuming that the sun rises on the side corresponding to the beginning of the street (see figure), the function returns the number of buildings that will see the sunrise (i.e. the number of buildings in which all those to their left are lower).

3. Square Root

Implement this function using binary search and only linear data structures:

```
int FunWithSearch::squareR(int num)
```

This function returns the square root of `num`. If `num` is not a perfect square, then it returns the largest integer less than the square root of `num` (i.e. the integer part of the number that represents the square root of `num`).

4. Minimum Difference

Implement this function using only linear data structures and some sorting algorithm(s).

```
int FunSortProblem::minDifference (const vector<unsigned> &values, unsigned nc)
```

Expected time complexity: O($n \times log\ n$)

The `values` vector represents several packets of chocolate, where each integer represents the number of chocolates in one of the packets. The chocolate packets will be distributed to `nc` children such that:

- Each child receives exactly one packet;
- The difference between the highest and lowest number of chocolates given to a child is minimal. The function returns the minimum difference between the highest and lowest number of chocolates given to a child. If the number of children is greater than the number of chocolate packets, the function returns -1.

Example call and expected output:

```
cout << FunSortProblem::minDifference ([3,4,1,9,56,7,9,12], 5) << endl;
```

```
6
```

Explanation: Delivered chocolates are 3, 4, 7, 9, 9

Extra Exercises

5. MinPages

Implement this function using binary search and using only linear data structures:

```
int FunWithSearch::minPages(const vector<int> & values, int numSt)
```

The vector `values` represents the number of pages in a set of books arranged on a shelf. The books will be distributed among a number of students (`numSt`) to help them complete a school assignment. Only contiguous books on the shelf can be allocated to a student. Each student must have at least one book allocated to them.

The function must find the distribution in which the maximum number of pages allocated to a student is minimum, and return this minimum value. If there is no valid distribution, the function returns -1.

6. Number of Inversions

Implement this function:

```
unsigned FunSortProblem::numInversions(const vector<int> & v)
```

Expected time complexity: Θ($n \times log\ n$)

The number of inversions in a vector indicates how far the vector is from being ordered. Note that in a sorted vector (in the desired order), the number of inversions is zero. The function to implement must find out how many inversions there are in the vector `v`.

The function to implement must find out how many inversions there are in the vector `v`. Two elements `v[i]` and `v[j]` form an inversion if `v[i] > v[j]` and `i < j`.

Example call and expected output:

```
cout << FunSortProblem::numInversions([10,50,20,40,30]) << endl;
```

```
40
```

Explanation: Inversions found were (50,20), (50,40), (50,30), (40,30).

Suggestion: Count the number of inversions when sorting the vector using the **Merge Sort** algorithm. In this algorithm, when the two sorted halves of the vector are joined, the existing inversions can be found (How?).

7. Nuts and Bolts

Implement this function using only linear data structures:

```
void FunSortProblem::nutsBolts(vector<Piece> &nuts, vector<Piece> &bolts)
```

Expected time complexity: $\Theta(n \times log\ n)$

The `nuts` vector represents a set of nuts (objects of the `Piece` class) identified by an `id` and `diameter`. The `bolts` vector represents a set of bolts (objects of the `Piece` class) identified by an id and diameter. Each nut corresponds exactly to a bolt and each bolt corresponds exactly to a nut. A nut and a bolt are matched if and only if they have the same diameter.

It is possible to compare a nut with a bolt, but it is not possible to directly compare two nuts or two bolts. The function must update the `nuts` and `bolts` vectors, which must contain the corresponding nut and bolt at the same index.

Suggestion: Use the partitioning concept from the **QuickSort** algorithm. Choose a random bolt, compare it with all the nuts and find the corresponding nut. Compare the corresponding nut now found with all the bolts, thus dividing the problem into two problems: one consisting of nuts and bolts smaller than the pair found and the other consisting of nuts and bolts larger than the pair found.