University of Waterloo
CS240E, Winter 2025
Programming Question 2

Due Date: Tuesday, March 11, 2025 at 5pm
Grace period: Submissions made before 11:59PM on Mar. 11 will be accepted without penalty.

Question 1 [40 marks]

a) [9+9+9+9=36 marks] Implement four methods of search in a sorted array that we saw in class (pseudo-codes are given at the end of this assignment):

1. Method 1 is *binary-search* as it was done in CS136.
2. Method 2 is *binary-search-improved* that uses at most $\log(2n+1)$ key-comparisons. (The version on the slides is not enough to find the index of the successor; please use the version from the course notes that is repeated below.)
3. Method 3 is *interpolation-search* that probes only once per round.
4. Method 4 is *interpolation-search-improved* that probes repeatedly per round to ensure that the sub-array has size $O(\sqrt{n})$.
   Note that you will have to fill in a symmetric case yourself.

You will maintain a single dynamic array in your program to perform these methods on.

b) [4(+10) marks] Create your own search-routine (call it "method 5") with the goal of achieving a small total number of key-comparisons over an (unknown) set of test cases.

- 2 marks will be given for any algorithm that beats Method 4 on some instances, and never is worse than Method 1.
- 2 marks will be given for any algorithm that beats Method 2 on some instances, and never is worse than Method 3.
- The 10 bonus-marks will be awarded as a competition; the submission that uses the fewest key-comparisons over our test set will receive 10 bonus-marks, the next-best will receive 9 bonus-marks, etc. The leaderboard will be updated regularly at `https://student.cs.uwaterloo.ca/~cs240e/w25/leaderboard2.phtml`. Your most recent correct submission made before the due date is your entry on the leaderboard.
- Only correct solutions (i.e., solutions that return the correct pair of numbers for nearly all tests) will receive marks.

There is no specific "best" algorithm for this, but your algorithm will have to integrate some aspects of binary search and some aspects of interpolation search. You are encouraged to do experiments with your implementation to figure out how many probes to use per round, and what spacing between them works best.

Submit: A C++ program `searchComparer.cpp` that provides a main function that accepts the following commands from stdin:

- `a` $n\ i_0\ i_1\ i_2 \ldots i_{n-1}$, where each i_j is an integer and $n \geq 1$ could be arbitrarily large. This resets the internally stored array A, making it such that it has length n and $A[j]$ stores the key with ID-number i_j. You do *not* have access to the actual key-values, instead (as in PQ1) keys are specified via ID-numbers and you should do key-comparisons via the helper-routines in class `Key` that we provide to you.

- `s` $m\ k$, where m, k are integers and $1 \leq m \leq 5$. This should use method m to search for the key with ID-number k in the internally stored array A.

  The output of the method should be a pair $b\ s$ of integers, printed space-separated on a single line, followed by a line break. Here b states whether the given key exists in A. Formally, if we write $k.value$ to denote the actual value of the key with ID-number k, then b should be 1 if $k.value = A[i].value$ for some index $0 \leq i < n$, and 0 otherwise. The integer s should be the index of the successor of the key with ID-number k, i.e., we have $A[s-1].value \leq k.value < A[s].value$. (The index of the successor is 0 if $k.value < A[0].value$ and n if $A[n-1].value \leq k.value$.)

- `l` prints the name you want to use for the leaderboard. If you do not want your method 5 to be on the leaderboard, you can print a blank line.

- `x` terminates the program.

Accessing and comparing keys: There is no particular relationship between an index i of the array and the ID-number of the key that is stored there. Also, just as in PQ1, there is no particular relationship between the ID-number of a key and the value that the key has; keys with different ID-numbers may have equal value. To aid debugging, for the public tests the key with ID-number i has value i. This is not necessarily the case for secret tests.

Class `Key` provides overloaded operators `<`, `==`, `<=` to do key-comparisons; you should compare two keys *only* via these operators.

Class `Key` also provides you with a `ratio` function that computes the fraction that you need in the formula for interpolation search. We will be keeping track of how often you call this function; it should not be called at all for Methods 1 and 2, and it should not be called more often than you do key-comparisons for the other methods. Failure to satisfy this means that you will fail the test; in particular do *not* abuse this function to do key-comparisons.

Further specifications and assumptions:

- We will do no illegal operations, i.e., parameters will be as specified above, and the (unknown) array is sorted in non-decreasing order. Also, the first operation will always be of type `a` (so there always is an array specified).
- "Printing on one line" means that the line must end with a newline. Trailing whitespace at the end of your output lines will be ignored by our test scripts.
- Your implementation of methods 1-4 must follow the given pseudocodes exactly as far as the number of key-comparisons is concerned. In particular the number of key-comparisons is unique (with some leeway for the missing case of Algorithm 4) and failure to use this many key-comparisons means failing the test even if the output is correct.
- You are *allowed* to use `std::vector`, `std::sqrt`, `std::floor`, `std::ceil` and `std::pair` (as well as standard library functions/classes that have nothing to do with CS240 material, see the assignment guidelines).
- You are *not allowed* to use the `<random>` header or otherwise implement a nondeterministic algorithm for b).
- You are *not allowed* to directly manipulate the private static variables of class `Key` or otherwise abuse the way the compiler lays out classes in memory. (A certain student relayed to me that this was "not explicitly disallowed" in the previous PQ, so now it is.)

Pseudo-codes

Algorithm 1: *binary-search*(A, n, k)

```
1 ℓ ← 0, r ← n − 1;
2 while ℓ ≤ r do
3 |   m ← ⌊(ℓ+r)/2⌋;
4 |   if A[m] = k then return "found at index m";
5 |   else if A[m] < k then ℓ ← m + 1;
6 |   else r ← m − 1;
7 return "not found, but would be between indices ℓ−1 and ℓ"
```

Algorithm 2: *binary-search-improved*(A, n, k) `// improved version`

```
1 ℓ ← 0, r ← n−1, χ ← 0;
2 while ℓ < r do
3 |   m ← ⌈(ℓ+r)/2⌉;
4 |   if (A[m] ≤ k) then ℓ ← m, χ ← 1;
5 |   else r ← m − 1;
6 if k > A[ℓ] then return "not found, but would be between indices ℓ and ℓ+1";
7 else if χ = 1 then return "found at index ℓ";
8 else if k ≥ A[ℓ] then return "found at index ℓ";
9 else return "not found, but would be between indices ℓ−1 and ℓ";
```

Algorithm 3: *interpolation-search*(A, n, k)

```
1 ℓ ← 0, r ← n − 1 ;
2 while ℓ ≤ r do
3 |   if k < A[ℓ] then return "not found, would be between indices ℓ − 1 and ℓ";
4 |   if k > A[r] then return "not found, would be between indices r and r + 1";
5 |   if k = A[r] then return "found at index r";
6 |   m ← ℓ + ⌈((k−A[ℓ])/(A[r]−A[ℓ])) · (r − ℓ − 1)⌉ ;
7 |   if A[m] = k then return "found at index m";
8 |   else if A[m] < k then ℓ ← m + 1;
9 |   else r ← m − 1;
  // Can argue that we never reach this point.
```

Algorithm 4: *interpolation-search-improved*(A, n, k)

```
1  if k < A[0] then return "not found, would be left of index 0";
2  if k > A[n − 1] then return "not found, would be right of index n−1";
3  if k = A[n − 1] then return "found at index n−1";
4  ℓ ← 0, r ← n − 1 ;
5  while r ≥ ℓ + 2 do                                   // inv:  A[ℓ] ≤ k < A[r]
6      N ← r − ℓ − 1,   p ← (k−A[ℓ])/(A[r]−A[ℓ]),   m ← ℓ + ⌈pN⌉;
7      if A[m] ≤ k then   // search rightward for a sub-array of size ≈ √n
8          for d = 1, 2, ... do
9              ℓ ← m + (d−1)⌈√N⌉, r′ ← m + d · ⌈√N⌉;
10             if r′ ≥ r then break out of for-loop;                  // out of bounds
11             if A[r′] > k then                                          // a probe
12                 r ← r′, break out of for-loop
13     else ...;                                  // proceed symmetrically going leftward
   // r ≤ ℓ + 1 and A[ℓ] ≤ k < A[r], so r = ℓ + 1 and k can only be A[ℓ]
14 if k = A[ℓ] then return "found at index ℓ";
15 else return "not found, would be between index ℓ and ℓ + 1";
```