Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada – DIMAp

Programming Language I • DIM0120
◁ Searching and Algorithm Complexity Exercises ▷

1 Introduction

The exercises in this list were chosen to help you develop your *problem solving skills*, with special focus on searching related problems[1]. The exercises often impose some time complexity restriction for the solutions, to encourage you to figure out the best time performance solutions possible.

In some of the exercises you will find *hints* that might guide your solution design by either recommending the direct use of classic algorithms that you have already learned (e.g. binary search), or by suggesting that you should drawn some inspiration from those same algorithms to create a new solution.

2 Writing Your Answers

The answers to the exercises should be preferably coded in C++. In case you do not have access to a computer and a compiler, you may write your answers in Portugol pseudocode.

You will notice that most of the exercises requires you to write a function to code the solution. In many cases it is necessary to pass an array or matrix to the function. So, it is important to recall how to pass arrays to functions. In C++ there are three possible strategies for that, described next:

1. **Regular array and its length.** This is the easiest way of passing an array as argument to a function: we provide the array name (pointer) to the function along with a variable with the array's size. The downside of this mechanism is that the function that receives the array must process the entire array. Of course, you may limit the portion of the array the function will process by passing on a smaller size; in this case the function will consider the subset starting at the beginning of the array until the defined size. Regardless of the size provided, the important point is that the subset always starts from the beginning of the array.
2. **A regular array and two indices.** In this approach we still provide the array name (pointer) but instead of the array's size, we send to the function a pair of indices defining a *closed-open* interval over the original array: [`A[first]`, `A[last]`). This gives us control over both the beginning and the end of the array's subset we wish to operate on.

[1] Most of the exercises were selected from this reference [1]

3. **A pair of pointer/iterators.** This approach is an improvement over the previous one: we only need to pass two pointers/iterators to the array. These two pointers/iterators also define a *closed-open* interval over the original array: [`first`, `last`). At first, the absence of the array name in the function's argument list might seem a bit strange, but remember that the array name in itself is just a fixed (const) pointer that holds the address of the beginning of the array. So, instead of passing a fixed pointer (the array name) plus an offset (the indices), we just pass two pointers to some memory location inside the array's assigned memory segment.
   This method offers several advantages: (1) we have total control over the interval of interest, since we can define the beginning and the end of that interval; (2) we only need to pass two parameters — the pointers; whereas in the previous method we need to provide three parameters — the array name and two indices, and ; (3) this is the same approach adopted by the STL library, which is a widely accepted standard.

See examples for each of these parameter-passing strategies in Code 1.

In the following exercises whenever we mention an array, feel free to choose to work with regular C++ arrays or with STL's `std::vector`. Often when a function fails to find a solution it should return an *invalid location*; if you are working with regular arrays and sizes (first approach) this means to return -1, for all the other approaches, this means to return `last`, a pointer/iterator to the location just after the last valid element in the working interval, if we consider [`first`, `last`).

3 Exercises

1. **Local minimum in an array.** Write a function that, given an array `a[]` of n distinct integers, finds and returns the location a *local minimum*. A local minimum in an array is an element `a[i]` such that both `a[i] < a[i-1]` and `a[i] < a[i+1]` (assuming the neighbor entry is inside the array). Note that an element that does not have one (or both) of its two neighbors can still be a local minimum. Your program should use $\sim 2\lg(n)$ compares in the worst case.

   Hint: Examine the middle value `a[n/2]` and its two neighbors `a[n/2 - 1]` and `a[n/2 + 1]`. If `a[n/2]` is a local minimum, stop; otherwise search in the half with the smaller neighbor.

2. **Local minimum in a matrix.** Given an n-by-n matrix `a[][]` of n^2 distinct integers, design an algorithm that runs in time proportional to $n\log(n)$ to find a local minimum: a pair of indices `i` and `j` such that `a[i][j] < a[i+1][j]`, `a[i][j] < a[i][j+1]`, `a[i][j] < a[i-1][j]`, and `a[i][j] < a[i][j-1]` (assuming `a[i][j]` has both neighbors).

   Hint: Find the minimum entry in row $n/2$, say `a[n/2][j]`. If it's a local mini-

Code 1 Three different approaches to pass an array to a function.

```
1  /// Prints out the array, using pointer + size
2  void print_1( int A[], size_t length ) {
3      cout << "A [ ";
4      for ( size_t i{0} ; i < length ; ++i )
5          cout << A[i] << " ";
6      cout << "]\n";
7  }
8
9  /*! Prints out the array, using pointer + two indexes that define a
10  *  closed-open range over the array.
11  */
12 void print_2( int A[], size_t first, size_t last ) {
13     cout << "A [ ";
14     while ( first != last )
15         cout << A[first++] << " ";
16     cout << "]\n";
17 }
18
19 /*! Prints out the array, using two pointers that define a
20  *  closed-open range over the array.
21  */
22 void print_3( int *first, int *last ) {
23     cout << "A [ ";
24     while ( first != last )
25         cout << *first++ << " ";
26     cout << "]\n";
27 }
28
29 void main() {
30     int A[]{ 10, 20, 30, 40, 50, 60, 70 };
31     size_t A_sz { sizeof(A)/sizeof(A[0]) }; // Determine how many elements in A.
32
33     print_1( A, A_sz );      // Prints the entire array.
34     print_1( A, 4 );         // Prints only the first 4 elements of the array.
35     print_2( A, 0, A_sz );   // Prints the entire array.
36     print_2( A, 2, 5 );      // Prints `A [ 30 40 50 ]`. This CAN'T be done with `print_1()`
37     print_3( &A[0], &A[A_sz] );  // Prints the entire array.
38     print_3( &A[2], &A[5] ); // Prints `A [ 30 40 50 ]`. This CAN'T be done with `print_1()`
39     return 0;
40 }
```

mum, then return it. Otherwise, check it's two vertical neighbors `a[n/2-1][j]` and `a[n/2+1][j]`. Recur in the half with the smaller neighbor.

Extra challenge: Design an algorithm that takes times proportional to n.

3. **Bitonic search**. An array is bitonic if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Write a function that, given a bitonic array of n distinct `int` values, determines whether a given integer is in the array. Your program should use $\sim 3\log(n)$ compares in the worst case.

   Hint: Use a version of binary search to find the maximum (in $\sim 1\lg(n)$ compares); then

use binary search to search in each piece (in $\sim 1\lg(n)$ compares per piece).

4. **Binary search with duplicates**. Modify binary search function so that it always returns the smallest index of a key of an item matching the search key.

5. **Hot or cold**. Your goal is the guess a secret integer between 1 and N. You repeatedly guess integers between 1 and N. After each guess you learn if it equals the secret integer (and the game stops); otherwise (starting with the second guess), you learn if the guess is hotter (closer to) or colder (farther from) the secret number than your previous guess. Design an algorithm that finds the secret number in $\sim 2\lg(N)$ guesses.

   In terms of design, first create the main program that reads from the command line a value for N. Then generate a random secret integer $\in [1, N]$. After that, asks the user for guesses, providing the corresponding clues (hot or cold). In the example below, the user has defined $N = 50$.

```
$./hot_cold 50

==================================================
Welcome to the Hot-Cold Guess game, copyright 2020.
--------------------------------------------------

This are the game rules:
    * I'll choose a random number in [1,50]. Your job is to guess that number.
    * From the second guess onward, I'll tell you if your guess
      is hot (closer than the previous guess) or cold (farther from
      the previous guess).
    * If you wish to quit the game, just type in a negative number.

Good luck!
--------------------------------------------------

>>> Guess the number: 35
>>> Nop, try again: 18
>>> Nop, it's hot though, try again: 27
>>> Nop, it's getting cold, try again: 21
>>> Nop, it's hot though, try again: 22
>>> Yeah, correct guess!

Thanks for playing.
```

   After that, instead of asking the user for a guess, just ask your function to guess the number.

6. **Floor and ceiling**. Given a set of comparable elements, the ceiling of x is the smallest element in the set greater than or equal to x, and the floor is the largest element less than or equal to x. Suppose you have an array of N items in ascending order. Write an $O(\log N)$ algorithm (function) that receives an array, finds and returns the floor and ceiling of x.

7. **Identity**. Given an array a of N distinct integers (positive or negative) in ascending order. Write a function that receives an array and finds an index `i` such that `a[i] = i` if such an index exists.

   Hint: binary search.

8. **Find a duplicate**. Given an array of N elements in which each element is an integer between 1 and N, write function to determine if there are any duplicates. Your algorithm should run in linear time and use $O(1)$ extra space.

   Hint: you may destroy the array.

9. **Search in a sorted, rotated array**. Given a sorted array of n distinct integers that has been rotated an unknown number of positions, e.g., $15\ 36\ 1\ 7\ 12\ 13\ 14$, write a function to determine if a given integer is in the list. The order of growth of the running time of your algorithm should be $\lg(n)$.

10. **Find the jump in the array**. Given an array of n integers of the form $1, 2, 3, \ldots, k-1, k+j, k+j+1, \ldots, n+j$, where $1 \leq k \leq nandj > 0$, design a logarithmic time algorithm to find the integer k. That is, the array contains the integers 1 through n, except that at some point, all remaining values are increased by j.

11. **Longest row of 0s**. Given an N-by-N matrix of 0s and 1s such that in each row no 0 comes before a 1, write a function that receives the matrix, finds and returns the row with the most 0s in $O(N)$ time.

12. **Sum of two**. Given two arrays A and B of at most N integers each, write a function that receives these arrays and determines whether the sum of any two distinct integers in A equals an integer in B. The function should return the three indices, two from A and one from B, for which the sum is true. In case there are no such indexes, your function should return three invalid locations (see the last paragraph in Section 2).

13. **Sum of three**. Given three arrays A, B, and C of at most N integers each, determine whether there exists a triple a in A, b in B, and c in C such that $a + b + c = 0$.

    Hint: Sort B in increasing order; sort C in decreasing order; for each a in A, scan B and C for a pair that sums to $-a$ (when the sum is too small, advance in B, when the sum is too large, advance in C).

14. **Find the duplicate**. Given a sorted array of $N + 2$ integers between 0 and N with exactly one duplicate, design a logarithmic time algorithm to find the duplicate.

    Hint: binary search.

$\sim$ The End $\sim$

References

[1] Robert Sedgewick and Kevin Wayne. Algorithms, 4th edition., 2011. https://algs4.cs.princeton.edu/home/.