ECE 2400 Computer Systems Programming

Topic 5: C Arrays

http://www.csl.cornell.edu/courses/ece2400
School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-14-10-42

Please do not ask for solutions. Students should compare their solutions to solutions from their fellow students, discuss their solutions with the instructors during lab/office hours, and/or post their solutions on Ed for discussion.

List of Problems

Problem 1. Short Answer

Carefully plan your solution before starting to write your response. Please be brief and to the point; if at all possible, limit your answers to the space provided.

Part 1.A Out-of-Bound Array Accesses

The following out-of-bounds array access is legal according to the C programming language standard.

```
int main( void )
{
  int a[] = { 1, 2, 3, 4 };
  int b   = a[100]; // out-of-bounds access
  return 0;
}
```

Other programming languages (e.g., Python, MATLAB, Java) require that such out-of-bounds array accesses cause a run-time error. Why doesn't the C programming language check at runtime to ensure that array accesses are never out of bounds?

Problem 2. Count Matches in Two Arrays of Integers

In this problem, we will explore two algorithms to count the number of matches between two unsorted arrays of integers. For example, consider the following two arrays:

```
int x[] = { 81, 38, 86, 36, 63, 14, 66, 37 };
int y[] = { 60, 66,  2, 97, 86,  4, 82, 44 };
```

The number of matches between the two arrays is two (i.e., 66, 86). Note that if there are duplicates in either of the input arrays, then each duplicate can create a distinct match. For example, consider the following two arrays:

```
int x[] = { 81, 38, 86, 36, 63, 86, 66, 37 };
int y[] = { 60, 66,  2, 97, 86, 66, 82, 44 };
```

The number of matches between the two arrays is four: 86 is included twice in array `x`, and each of these values match with 86 in array `y`; 66 is included twice in array `y`, and each of these values match with 66 in array `x`.

Part 2.A Quadratic Match Algorithm

Develop an algorithm whose execution time is $O(N^2)$ to count the number of matches between two arrays of integers. Implement your algorithm in a `count_matches_v1` function which takes as input an array `x`, an array `y`, and the size of both arrays `n` (i.e., both arrays are the same size). The function returns the number of matches. Your implementation cannot modify either input array. Your implementation should be correct and efficient in terms of both execution time and space usage. While you are welcome to use pseudo-code to plan your approach, your final solution must be written using valid C syntax.

```
int count_matches_v1( int* x, int* y, int n ) {
```

Part 2.B Linear Match Algorithm

Now assume we know something about the distribution of the data stored in the array `x`. More specifically, assume we know that most of the values stored in array `x` are duplicates. Let K be the number of unique values in array `x`. For example, consider the following two arrays:

```
int x[] = { 70, 86, 86, 66, 70, 86, 66, 70 };
int y[] = { 60, 66,  2, 97, 86, 66, 82, 44 };
```

$K = 3$ since the array `x` only has four unique values (i.e., 70, 66, 86). The number of matches between the two arrays is seven: 86 is included twice in array `x`, and each of these values match with 86 in array `y`; 66 is included twice in array `x`, and each of these values match with the two copies of 66 in array `y`.

Develop an algorithm whose execution time is $O(N)$ to count the number of matches between two arrays of integers under the assumption that most of the values stored in array `x` are duplicates. Implement your algorithm in a `count_matches_v2` function which has the same interface as `count_matches_v1`. *You can assume K is a compile-time constant.* Your implementation cannot modify either input array. Your implementation should be correct and efficient in terms of both execution time and space usage. While you are welcome to use pseudo-code to plan your approach, your final solution must be written using valid C syntax.

```
int count_matches_v2( int* x, int* y, int n ) {
```

Part 2.C Comparing Match Algorithms

Note: This problem involves material on complexity analysis from Topic 8.

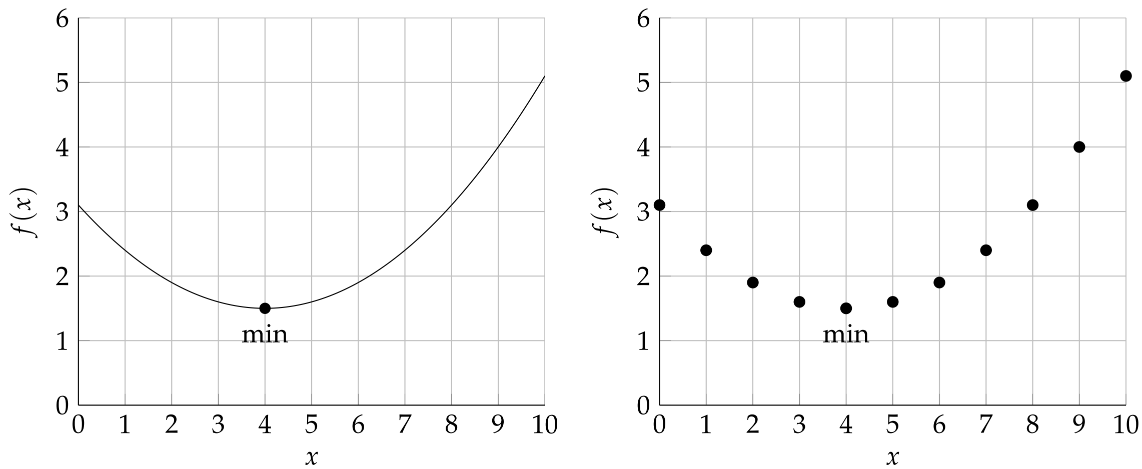
In this problem, you will be qualitatively comparing the two match algorithms. **Begin by filling in the following table.** The units for execution time should be the number of inner loop iterations. Your analysis should be for the worst case, but also generalized with respect to K (i.e., your equations for execution time should include K if appropriate).

	Worst Case Execution Time $T_K(N)$	Worst Case Time Complexity big-O
`count_matches_v1`		
`count_matches_v2`		

Use these results along with deeper insights to perform a comparative analysis of these two match algorithms, with the ultimate goal of **making a compelling argument for which algorithm will perform better across a large number of usage scenarios.** While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The **first paragraph** might discuss the performance of both algorithms using time complexity analysis. Justify the entries in the table. Remember that time complexity analysis is not the entire story; it is just the starting point for performance analysis. The **second paragraph** might discuss the stack or heap space usage of both algorithms using space complexity analysis. Remember that space complexity analysis is not the entire story; it is just the starting point for storage requirement analysis. The **third paragraph** might discuss other qualitative metrics such as generality, maintainability, and design complexity. The **final paragraph** can conclude by making a compelling argument for which algorithm will perform better in the general case, or if you cannot strongly argue for either implementation/algorithm explain why. Your answer will be assessed on how well you argue your position.

Problem 3. Finding the Minimum of a Convex Function

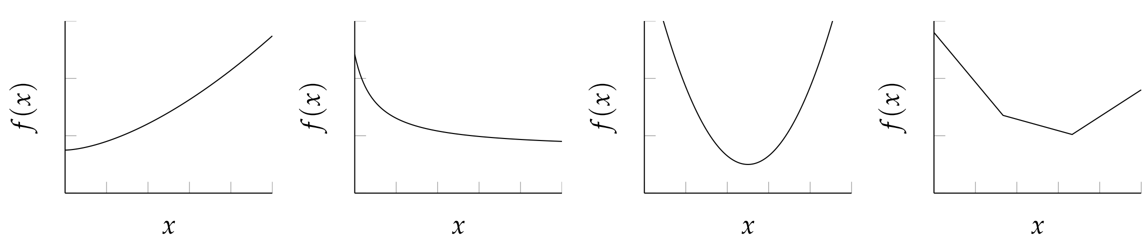
In this problem, we will explore two algorithms to find the minimum of a convex function. A convex function is a continuous function whose value at the midpoint of every interval in its domain does not exceed the arithmetic mean of its values at the ends of the interval. Technically, we will only be considering strictly convex functions in which there is one and only one minimum value. So there is only one local minimum and that local minimum is also a global minimum. The plot on the left shows a continuous convex function, while the plot on the right shows the corresponding discrete function sampled at integer values of x.



We can represent any discrete function as an array of doubles. The following shows an array corresponding to the above convex discrete function.

```
double y[] = { 3.1, 2.4, 1.9, 1.6, 1.5, 1.6, 1.9, 2.4, 3.1, 4.0, 5.1 };
```

Our goal is to develop algorithms to find the minimum of (strictly) convex functions. These algorithms will take as input an array of doubles and the size of that array as parameters, and they will return the minimum value of the convex function. You can assume the input array is indeed a valid (strictly) convex function. Here are some additional convex functions your algorithms should be able to analyze.



Part 3.A Iterative Search Algorithm

The first algorithm starts at the first element in the input array and does a linear search until it finds the minimum of the convex function. **Implement this iterative search algorithm as a C function. Your algorithm should not scan the entire array, but should instead stop as soon as it finds the minimum. You must handle any corner cases correctly.** While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C syntax. *You can assume the input array size is always greater than one.*

```
double find_min( double* y, int n ) {
  assert( n > 1 );
```

Part 3.B Recursive Search Algorithm

The second algorithm uses a recursive search to find the minimum. **Implement this recursive search algorithm as a C function. You must handle any corner cases correctly.** While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C syntax. *You can assume the input array size is always greater than one and is also even. You can assume the input array will always correspond to a convex function with exactly one minimum value.* We have provided you with the corresponding recursive helper function and the base case condition.

```
double find_min( double* y, int n ) {
  assert( (n > 1) && ((n % 2) == 0) );
  return find_min_h( y, 0, n-1 );
}

double find_min_h( double* y, int left, int right ) {
  // Condition to check for the base case
  if ( (right - left) == 1 ) {
```

Part 3.C Comparing Search Algorithms

Note: This problem involves material on complexity analysis from Topic 8.

In this problem, you will be qualitatively comparing the two search algorithms. **Begin by filling in the following table.**

	Wost Case Time Complexity	Worst Case Stack Space Complexity
Iterative Search		
Recursive Search		

Use these results along with deeper insights to perform a comparative analysis of these two search algorithms, with the ultimate goal of **making a compelling argument for which algorithm will perform better across a large number of usage scenarios.** While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The **first paragraph** might discuss the performance of both algorithms using time complexity analysis. Justify the entries in the table. Remember that time complexity analysis is not the entire story; it is just the starting point for performance analysis. The **second paragraph** might discuss the stack space usage of both algorithms using space complexity analysis. Justify the entries in the table. Remember that space complexity analysis is not the entire story; it is just the starting point for storage requirement analysis. The **third paragraph** might discuss other qualitative metrics such as generality, maintainability, and design complexity. The **final paragraph** can conclude by making a compelling argument for which algorithm will perform better in the general case, or if you cannot strongly argue for either algorithm explain why. Your answer will be assessed on how well you argue your position.