CS 477: Homework #3

Due on September 27, 2016 at 2:30pm

Monica Nicolescu

Matthew J. Berger

Problem 1

(U & G-required)[40 points]

(a) [20 points] Implement in C/C++ a divide and conquer algorithm for finding a position of the largest element in an array of n numbers. Show how your algorithm runs on the input:
A = [1 4 9 3 4 9 5 6 9 3 7].

(b) [10 points] What will be your algorithm's output for arrays with several elements of the largest value? Indicate the answer on the input given above.

(c) [10 points] Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

Solution:

(a) The algorithm I've created for this problem is.:

```
int GetPositionOfMaxInteger(int* arr, int start, int end)
{
    if (start == end) return start;

    int midpoint = floor(((start + end) / 2.0) + 0.5);
    int maxPosLeft = GetPositionOfMaxInteger(arr, start, midpoint - 1);
    int maxPosRight = GetPositionOfMaxInteger(arr, midpoint, end);

    return (arr[maxPosLeft] >= arr[maxPosRight]) ? maxPosLeft : maxPosRight;
}
```

The program to run the test input against the algorithm is:

```
#include <iostream>
#include <cmath>
int GetPositionOfMaxInteger(int* arr, int start, int end)
{
    if (start == end) return start;

    int midpoint = floor(((start + end) / 2.0) + 0.5);
    int maxPosLeft = GetPositionOfMaxInteger(arr, start, midpoint - 1);
    int maxPosRight = GetPositionOfMaxInteger(arr, midpoint, end);

    return (arr[maxPosLeft] >= arr[maxPosRight]) ? maxPosLeft : maxPosRight;
}
int main(int argc, char** argv)
{
    int arr[] = { 1, 4, 9, 3, 4, 9, 5, 6, 9, 3, 7 };
    int size = 10;
    int pos = GetPositionOfMaxInteger(arr, 0, size);
    std::cout << "Requested array: ";
    for(int i = 0; i < size; i++)
    {
        std::cout << arr[i];
        if (i < size - 1) std::cout << ",";
        else std::cout << std::endl;
    }
    std::cout << "Position of Leftmost Largest Integer: " << pos << std::endl;
    return 0;
}
```

Problem 1 continued on next page...

The output for this program is:

```
Requested array: 1,4,9,3,4,9,5,6,9,3
Position of Leftmost Largest Integer: 2
```

(b) The algorithm's output for arrays with several elements of the largest value will be the position of the leftmost largest value in the array. This is the first occurrence of the largest value when the array is scanned iteratively from left to right.

(c) The recurrence for the number of key comparisons made by my algorithm is:

$$C(n) = C(n/2) + C(n/2) + 1 \; for \; n > 1, C(1) = 0$$

Solving with backwards substitutions for $n = 2^k$ produces the following:

$$\begin{aligned} C(2^k) &= 2C(2^{k-1}) + 1 \\ &= 2[2C(2^{k-2} + 1] + 1 = 2^2C(2^{k-2}) + 2 + 1 \\ &= ... \\ &= 2^iC(2^{k-i}) + 2^{i-1} + 2^{i-2} + ... + 1 \\ &= 2^kC(2^{k-k} + 2^{k-1} + 2^{k-2} + ... + 1 = 2^k - 1 = n - 1 \end{aligned}$$

$C(n) = n - 1$ satisfies the recurrence for each value of $n > 1$ by substitution into the recurrence equation and looking at only the even or odd cases.

Let $n = 2i$ where $i > 0$. The left hand side of the recurrence equation is $n - 1 = 2i - 1$. The right hand side is:

$$\begin{aligned} C(n/2) + C(n/2) + 1 &= C(2i/2) + C((2i/2 + 1 \\ &= 2C(i) + 1 = 2(i - 1) + 1 = 2i - 1 \end{aligned}$$

This is equivalent to the left hand side.

Let $n = 2i + 1$ where $i > 0$. The left hand side of the recurrence equation is $n - 1 = 2i$. The right hand side is:

$$\begin{aligned} C(n/2) + C(n/2) + 1 &= C((2i + 1)/2) + C((2i + 1)/2 + 1 \\ &= C(i + 1) + C(i) + 1 = (i + 1 - 1) + (i - 1) + 1 = 2i \end{aligned}$$

This is equivalent to the left hand side as well.

Problem 2

(U & G-required)[40 points]

a) [20 points] Is Mergesort a stable sorting algorithm? Give a justification for your answer or provide a counter example.

b) [20 points] Is Quicksort a stable sorting algorithm? Give a justification for your answer or provide a counterexample.

Solution:

a) Most implementations of Mergesort **are** in fact stable sort implementations. This means they "preserve the input order of equal elements in the sorted output". Depending on how the merge is performed, the algorithm can become stable or unstable. For instance, if comparing two elements and using the $\leq$ sign (a stable merge operation) you can make the merge operation unstable by using a $<$ sign instead. This would prevent the input order from being preserved. In most cases however, the merge operation is implemented so that if you encounter equal elements you output the one that came from the "lower" half of the two halves being merged, thus preserving input order.

b) The most efficient implementations of Quicksort **are not** stable sort implementations. For instance, if comparing two equal items on either side of a pivot, most efficient implementations will actually reverse the two items. This does not preserve the input order and does not qualify as a stable sort.

Problem 3

Implement in C/C++ an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all of its positive elements. Your algorithm should be both time-efficient and space-efficient. Show the output of your algorithm on the input array A = [4 -3 9 8 7 -4 -2 -1 0 6 -5].
Solution:

(a) The algorithm I've created for this problem is.:

```
void RearrangeArray(int* arr, int size)
{
    int firstPositivePosition, currentIndex;
    for (currentIndex = 0, firstPositivePosition = -1; currentIndex < size; ↩
        currentIndex++)
    {
        if (arr[currentIndex] < 0)
        {
            int tmp = arr[++firstPositivePosition];
            arr[firstPositivePosition] = arr[currentIndex];
            arr[currentIndex] = tmp;
        }
    }
}
```

The program to run the test input against the algorithm is:

```
#include <iostream>
void RearrangeArray(int* arr, int size)
{
    int firstPositivePosition, currentIndex;
    for (currentIndex = 0, firstPositivePosition = -1; currentIndex < size; ↩
        currentIndex++)
    {
        if (arr[currentIndex] < 0)
        {
            int tmp = arr[++firstPositivePosition];
            arr[firstPositivePosition] = arr[currentIndex];
            arr[currentIndex] = tmp;
        }
    }
}
int main(int argc, char** argv){
    int arr[] = { 4, -3, 9, 8, 7, -4, -2, -1, 0, 6, -5 };
    int size = 11;
    std::cout << "Requested array: ";
    for(int i = 0; i < size; i++){
        std::cout << arr[i];
        if (i < size - 1) std::cout << ",";
        else std::cout << std::endl;
    }
    RearrangeArray(arr, size);
    std::cout << "Negative values moved to beginning of array: ";
    for(int i = 0; i < size; i++){
        std::cout << arr[i];
        if (i < size - 1) std::cout << ",";
        else std::cout << std::endl;
    }
}
```

The output for this program is:

```
Requested array: 4,-3,9,8,7,-4,-2,-1,0,6,-5
Negative values moved to beginning of array: -3,-4,-2,-1,-5,4,9,8,0,6,7
```

Problem 4

Extra Credit (I'm an undergraduate)

[20 points] Estimate how many times faster an average successful search will be in a sorted array of 100,000 elements if it is done by binary search versus sequential search.

Solution

For a sequential search, the complexity is $\mathcal{O}(N)$. For a binary search, the complexity is $\mathcal{O}\log_2 N$. A sequential search would iterate over 100,000 elements at worst (if the desired item in the sorted array was at the very last position). The binary search at worst would iterate over $log_2(100,000) \approx 17$ elements. An average successful binary search is faster in this case by a factor of at least $100,000/17 \approx \mathbf{5882.35}$.