Homework 2

CIS-675 DESIGN AND ANALYSIS OF ALGORITHMS

PROF. IMANI PALMER

4/21/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Question 1:

We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

The worst and average-case runtime of the insertion sort algorithm is $O(N^2)$. If the algorithm is completing faster than expected, the input of the sorting algorithm must be an array of numbers that is already sorted or partially sorted. In this case, the code inside the while loop will be executed less often. The best-case runtime for insertion sort is $O(N)$ and occurs when the input is already sorted. The worst-case occurs when the input is sorted in reverse order.

Question 2:

Give a 5 integer array such that it elicits the worst case running time for insertion sort.

A 5 integer array that elicits the worst case running time for insertion sort is: [9, 6, 5, 3, 2]

The array must be sorted in reverse order to elicit a worst-case runtime.

Question 3:

You are given an unsorted array of integers which may contain negative numbers.

Example: [2, -3, 0, **5, -2, -5**, -1]

Devise a divide and conquer algorithm that finds the largest sequential subsequence in the array. The largest subsequence in the above example is bold. For clarification, the algorithm should be the largest subsequence in the set of both increasing and decreasing subsequences.

The problem statement is asking for the subsequence in the array with the highest range that is in ascending or descending order. The following approach was devised and implemented:

1. Starting on the left side, find the first sequential ascending or descending subsequence in the array and insert it into a temporary array called "array1".
2. Find the second sequential ascending or descending subsequence in the array and insert it into a temporary array called "array2".
3. Find the total range covered by each subsequence by subtracting the last element from the first (or vice-versa).
4. Keep the temporary array with the higher range and use the other to store the next subsequence for comparison. Continue scanning where left off and finish when scan of original array is complete.

The worst-case runtime of this approach is $O(N)$. The worst-case occurs when there are no subsequences of length greater than 2. In this case, each element (excluding the first and last) are visited twice.

Example: [2, -3, 0, 5, -2, -5, -1].

Array1 = [2, -3]. Range = 2-(-3) = 5.

Array2 = [-3, 0, 5]. Range = 5-(-3) = 8. Range of array2 is greater than array1, so keep array2.

Array1 = [5, -2, -5]. Range = 5-(-5) = 10. Range of array1 is greater than array2, so keep array1.

Array2 = [-5, -1]. Range = -1-(-5) = 4. Range of array2 is not greater than array1. Keep array1.

Scan complete. Return array1.

The following algorithm was implemented in C.

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int* algorithm(int* array, int n){
   if(n < 2){
      return(array);
   }

   int i;
   int flag = 1;             // used to tell which temporary array holds the subs
equence with the highest range.
                             // (1 for array1 ; 2 for array2)
   int index = 0;            // used to parse original array
   int second_index = 0;     // used to parse subsequences
   int array1_length = 0;    // length of subsequence
   int array2_length = 0;
   int array1_value = 0;     // range of subsequence
   int array2_value = 0;
   int* array1 = (int*)malloc(n*sizeof(int)); // stores subsequence
   int* array2 = (int*)malloc(n*sizeof(int));

   if(index == n-2){
      return(array); // original array is only 2 elements, so return it.
   }

   // first iteration fills array1 and array2

   // if the first two elements are equal, put them in array1
   if(array[index] == array[index+1]){
      while((index < n-1)&&(array[index] == array[index+1])){
         array1[index] = array[index];
         array1_length++;
         index++;
      }
   }

   // if all elements in the original array are equal, return it.
   if(index >= n-1){
      return(array);
   }

   // Not all elements are equal. Load array1 and array2.
```

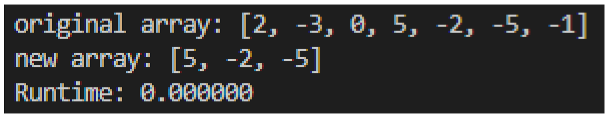
```
  // If sequence is decreasing, enter here.
  if(array[index] >= array[index+1]){
    while((index < n-1)&&(array[index] >= array[index+1])){
      array1[index] = array[index];
      array1_length++;
      index++;
    }
    array1[index] = array[index];
    array1_length++;

    array1_value = array1[0] - array1[array1_length-1];

    if(index >= n-1){
      return(array); // if no more elements in original array, return.
    }

    // if there is a second sequence, it will be increasing. Use array2
    while((index < n-1)&&(array[index] <= array[index+1])){
      array2[second_index] = array[index];
      array2_length++;
      index++;
      second_index++;
    }
    array2[second_index] = array[index];
    array2_length++;

    array2_value = array2[array2_length-1] - array2[0];

    if(array1_value > array2_value){
      if(index >= n-
1){ // if there are no more unscanned elements in original array,
                  // and array1 has greater range than array2, free arra
y2 and return array1.
        free(array2);
        return(array1);
      }
      flag = 1; // array1 is holding higher range. Do not change array1.
    }
    else{
      if(index >= n-1){
        free(array1);
        return(array2);
      }
      flag = 2; // array2 is holding higher range. Do not change array2.
    }
}

// Not all elements are equal. Load array1 and array2.
// If sequence is increasing, enter here.
else{
   while((index < n-1)&&(array[index] <= array[index+1])){
      array1[index] = array[index];
      array1_length++;
      index++;
   }
   array1[index] = array[index];
   array1_length++;

   array1_value = array1[array1_length-1] - array1[0];

   if(index >= n-1){
      return(array); // handles [5, 5, 3]
   }

   while((index < n-1)&&(array[index] >= array[index+1])){
      array2[second_index] = array[index];
      array2_length++;
      index++;
      second_index++;
   }
   array2[second_index] = array[index];
   array2_length++;

   array2_value = array2[0] - array2[array2_length-1];

   if(array1_value > array2_value){
      if(index >= n-1){
         free(array2);
         return(array1);
      }
      flag = 1; // array1 is holding higher range. Do not change array1.
   }
   else{
      if(index >= n-1){
         free(array1);
         return(array2);
      }
      flag = 2; // array2 is holding higher range. Do not change array2.
   }
}
```

```
while(index < n-1){
   // do not edit array1.
   if(flag == 1){
      // if the sequence is increasing, continue in that direction.
      if(array[index] <= array[index+1]){
         second_index = 0;
         array2_length = 0;
         while((index < n-1)&&(array[index] <= array[index+1])){
            array2[second_index] = array[index];
            array2_length++;
            index++;
            second_index++;
         }
         array2[second_index] = array[index];
         array2_length++;
         array2_value = array2[array2_length-1] - array2[0];
      }
      // if the sequence is decreasing, continue in that direction.
      else{
         second_index = 0;
         array2_length = 0;
         while((index < n-1)&&(array[index] >= array[index+1])){
            array2[second_index] = array[index];
            array2_length++;
            index++;
            second_index++;
         }
         array2[second_index] = array[index];
         array2_length++;
         array2_value = array2[0] - array2[array2_length-1];
      }
      // if array2 has greater range, set flag to 2.
      if(array2_value > array1_value){
         flag = 2;
      }
   }
   // do not edit array2.
   else{
      // if the sequence is increasing, continue in that direction.
      if(array[index] <= array[index+1]){
         second_index = 0;
         array1_length = 0;
         while((index < n-1)&&(array[index] <= array[index+1])){
            array1[second_index] = array[index];
```

```c
                array1_length++;
                index++;
                second_index++;
            }
            array1[second_index] = array[index];
            array1_length++;
            array1_value = array1[array1_length-1] - array1[0];
        }
        // if the sequence is decreasing, continue in that direction.
        else{
            second_index = 0;
            array1_length = 0;
            while((index < n-1)&&(array[index] >= array[index+1])){
                array1[second_index] = array[index];
                array1_length++;
                index++;
                second_index++;
            }
            array1[second_index] = array[index];
            array1_length++;

            array1_value = array1[0] - array1[array1_length-1];
        }
        // if array1 has greater range, set flag to 1.
        if(array1_value > array2_value){
            flag = 1;
        }
      }
   }

   if(flag == 1){
      free(array2);
      return(array1);
   }
   else{
      free(array1);
      return(array2);
   }
}

int main(){
   clock_t beginning, end;
   int i;
   int array[7] = {2, -3, 0, 5, -2, -5, -1};
   int n = 7;
```

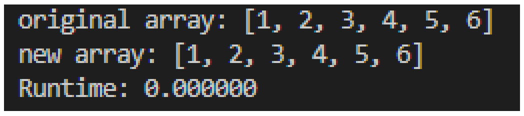
```
    printf("original array: [");
    for(i = 0; i < n-1; i++){
        printf("%d, ", array[i]);
    }
    printf("%d]\n", array[n-1]);

    beginning = clock();
    int* new_array = algorithm(array, n);
    end = clock();

    double runtime = ((double)(end - beginning))/CLOCKS_PER_SEC;

    int new_n = 3;
    printf("new array: [");
    for(i = 0; i < new_n-1; i++){
        printf("%d, ", new_array[i]);
    }
    printf("%d]\n", new_array[new_n-1]);
    printf("Runtime: %f\n", runtime);
}
```

The output of the code is the following:

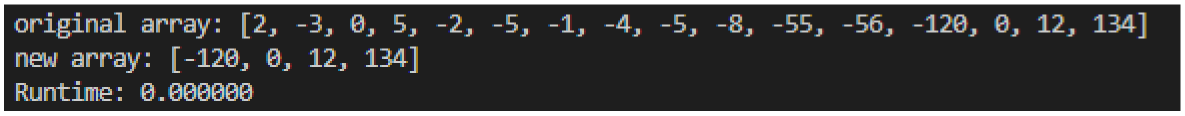


```
original array: [2, -3, 0, 5, -2, -5, -1]
new array: [5, -2, -5]
Runtime: 0.000000
```

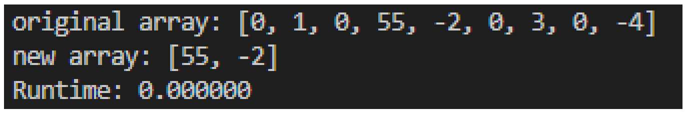
Other inputs were tested.



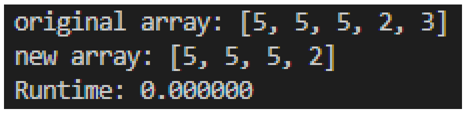
```
original array: [1, 2, 3, 4, 5, 6]
new array: [1, 2, 3, 4, 5, 6]
Runtime: 0.000000
```



```
original array: [2, -3, 0, 5, -2, -5, -1, -4, -5, -8, -55, -56, -120, 0, 12, 134]
new array: [-120, 0, 12, 134]
Runtime: 0.000000
```



```
original array: [0, 1, 0, 55, -2, 0, 3, 0, -4]
new array: [55, -2]
Runtime: 0.000000
```



```
original array: [5, 5, 5, 2, 3]
new array: [5, 5, 5, 2]
Runtime: 0.000000
```

<u>Question 4:</u>

Give some reasons as to why someone would use mergesort over quicksort.

For all three cases (best-case, worst-case, and average case), the mergesort algorithm yields $O(Nlog(N))$ time complexity, making the runtime predictable. It will never do better or worse than $O(Nlog(N))$. However, it requires a temporary array used for storage of size $O(N)$.

The best-case and average-case time complexity for quicksort is $O(Nlog(N))$. However, the worst-case time complexity is $O(N^2)$. It does not require any extra storage space.

Mergesort is ideal for systems that cannot allow a worst-case time complexity of $O(N^2)$ and do not care about the extra space required. Such systems would choose mergesort over quicksort, as quicksort may take $O(N^2)$ as its worst-case runtime.

A linked list does not have to be contiguous in memory. Therefore, mergesort does not require the temporary array it would need to sort an array. Therefore, Mergesort is ideal when sorting a linked list because it does not require the extra space and has a better worst-case runtime than quicksort.

<u>Question 5:</u>

Describe the ideal sorting algorithm.

The ideal sorting algorithm would have a worst-case and average-case time complexity of $O(nlog(n))$. Also, as with Insertion Sort, it should be able to identify inputs that are already or partially sorted making the best-case runtime $O(N)$. It would also be stable, meaning that it preserves the relative order of equal elements.

One ideal sorting algorithm is a hybrid algorithm called “TimSort”. TimSort is a combination of Insertion Sort and Merge Sort. The algorithm sorts smaller pieces of data using Insertion Sort, then combines them together using Merge Sort. Because Insertion Sort and Merge Sort are stable algorithms, TimSort is also stable. TimSort worst and average-case time complexity is $O(nlog(n))$ and is used in Python’s sorted() and sort() functions as well as Java’s Array.sort() function.