CS 677: Assignment 3

Terence Henriod

March 3, 2014

Abstract

In this assignment, divide and conquer algorithms are considered. One such method for finding the max of a given set of data is considered, and various sorting divide and conquer sorting methods are considered, including mergesort and a 3 partition quicksort.

1. Implement a Max algorithm (in C/C++) that uses a divide and conquer strategy to find the maximum among N items stored in an array A[0], . . . , A[n-1]. To illustrate the sequence of function calls made by your algorithm, each call to the algorithm should print the following tuple: <Return-Value, "Max(left-idx, right-idx)" >where Return-Value is the maximum value returned by that call and left-idx and right-idx are the indices of the array with which that particular recursive call is made.

   (a) Submit your code for the algorithm.

   Solution:

```
template <typename ItemType>
ItemType Max( ItemType* A, unsigned int left_idx, unsigned int right_idx )
{
  // variables
  ItemType Return_Value;
  ItemType left_temp;
  ItemType right_temp;
  int current_array_size = ( right_idx - left_idx + 1 );
  int left_upper_bound = 0;
  int right_lower_bound = 0;

  // base case: the current sub_array size is less than or equal to 2
  if( current_array_size == 1 )
  {
    // the only item is the greatest item
    Return_Value = A[left_idx];
  }
  // case: the current sub_array size is greater than 2
  else
  {
    // find the boundary indices of the two sub-arrays
    left_upper_bound = ( left_idx + right_idx ) / 2;
    right_lower_bound = ( ( left_idx + right_idx ) / 2 ) + 1;

    // find the max of each sub_array
    left_temp = Max( A, left_idx, left_upper_bound );
    right_temp = Max( A, right_lower_bound, right_idx );

    // find the greater of the two values
    Return_Value = ( left_temp < right_temp ? right_temp : left_temp );
  }

  // state the effect of this action for homework proposes
  cout << "<" << Return_Value << ", Max(" << left_idx << ", " << right_idx
       << ")>" << endl;

  // return the maximum value found for this array/sub-array
  return Return_Value;
}
```

(b) Submit the analysis of the running time of your algorithm.

Solution: The running time for this algorithm can be summarized by the following table:

Step Number	Action	Times Executed
1.	Variable initialization and computation	1
2.	Compare the array size	1
3.	In the base case, make an assignment	1
4.	In the other case, compute the sub-array bounds	$\Theta(1)$
5.	Make 2 recursive calls to find the max of each sub-array	$2 * M(\frac{n}{2})$
6.	Compare/assign the max of the two sub-arrays	1
7.	Return the maximum value found	1

So it would be reasonable to say that in general this algorithm is representedby the recurrence:

$$M(n) = 2 * M(\frac{n}{2}) + \Theta(1)$$

and even more accurately by:

$$M(x) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ M(n) = 2 * M(\frac{n}{2}) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Using the "Recursion Tree" method, we can see that the problem size at each node will be $\frac{n}{2^i}$, with a constant cost at each level; $1 = \frac{n}{2^i}$ gives $i = \lg n$, where i is the number of recursive calls made. Note that at each level, the number of nodes at level i is 2^i for this algortihm. So, summing the cost of all levels but the last one, and then adding the cost of the lowest level/leaves gives:

$$\begin{aligned} M(n) &= \sum_{i=0}^{\lg n} 1 + 2^{\lg n} M(1) \\ &= \lg n + n * M(1) \\ &= \Theta(\lg n) + n \\ &= \Theta(\lg n) + \Theta(n) \\ M(n) &= \Theta(n) \end{aligned}$$

(c) Submit the output of your algorithm for the following input:
A = [T I N Y E X A M P L E]

Solution:

```
Testing Problem 1 code by outputting
<Return_Value, Max(left_idx, right_idx)> tuples
=======================================================
<T, Max(0, 0)>
<I, Max(1, 1)>
<T, Max(0, 1)>
<N, Max(2, 2)>
<T, Max(0, 2)>
<Y, Max(3, 3)>
<E, Max(4, 4)>
<Y, Max(3, 4)>
<X, Max(5, 5)>
<Y, Max(3, 5)>
<Y, Max(0, 5)>
<A, Max(6, 6)>
<M, Max(7, 7)>
<M, Max(6, 7)>
<P, Max(8, 8)>
<P, Max(6, 8)>
<L, Max(9, 9)>
<E, Max(10, 10)>
<L, Max(9, 10)>
<P, Max(6, 10)>
<Y, Max(0, 10)>
```

2. Implement a bottom-up Mergesort algorithm (in C/C++) that works by making a sequence of passes over the entire array doing m-by-m merges, doubling m on each pass. For example, the algorithm first scans through the input performing 1-by-1 merges ot produce ordered sub-arrays of size 2; then, it scans through the input again performing 2-by-2 merges to produce ordered sub-arrays of size 4, and so on until the entire array is sorted. At each call to Merge, print out the input array that is being processed by that function call. Note: the final Merge may be an m-by-x merge, for some x less than or equal to m (if the array size is a multiple of m).

   (a) Submit your code for the algorithm.

   Solution: (Note: *Merge* code adapted from lecture notes/textbook)

```
template <typename ItemType>
void bottomUpMergeSort( ItemType* A, int n )
{
  // variables
  int m = 1; // the sub-array size to merge into size 2m
  int right_start = 0;
  int right_end = 0;
  int left_end = 0;

  // scan array, performing merges until two sub-arrays have been merged
  // to the original size
  while( m < n )
  {
    // perform the merge-scanning for the current sub-array size
    for( right_start = 0; right_start < n; right_start += ( 2 * m ) )
    {
      // find the end of the right sub-array
      right_end = right_start + m - 1;

      // find the end of the sub_arrays combined
      left_end = right_end + m;
      if( left_end >= n )
      {
        // prevent the end marker for the sub-arrays from overreaching the array
        left_end = n - 1;
      }

      // merge each pair of sub-arrays
      Merge( A, right_start, right_end, left_end );
    }

    // move up to the next sub-array size
    m *= 2;
  }

  // no return - void
}
```

```
template <typename ItemType>
void Merge( ItemType* A, int p, int q, int r )
{
  // variables
  int main_ndx = p;
  int left_ndx = 0;
  int right_ndx = 0;
  int left_size = q - p + 1;
  int right_size  = r - q;
  ItemType temp;
  ItemType left_array[left_size + 1];
    left_array[left_size] = kSentinel;
  ItemType right_array[right_size + 1];
    right_array[right_size] = kSentinel;

  // print the array being processed for hw purposes
  cout << "p: " << p << ", q: " << q << ", r: " << r << endl;
  cout << "Before: ";
  for( main_ndx = p; main_ndx <= r; main_ndx++ )
  {
    cout << A[main_ndx] << ' ';
  }
  cout << endl;

  // load up the sub-arrays
  for( main_ndx = p, left_ndx = 0; main_ndx <= q; left_ndx++, main_ndx++ )
  {
    // copy the element over
    left_array[left_ndx] = A[main_ndx];
  }
  for( right_ndx = 0; main_ndx <= r; right_ndx++, main_ndx++ )
  {
    // copy the element over
    right_array[right_ndx] = A[main_ndx];
  }


  // perform the merging
  for( main_ndx = p, right_ndx = 0, left_ndx = 0; main_ndx <= r; main_ndx++ )
  {
    // case: the element of the left sub-array is less or equal
    if( left_array[left_ndx] <= right_array[right_ndx] )
    {
      // store the element in the original array
      A[main_ndx] = left_array[left_ndx];

      // move on to the next element of the right sub-array
      left_ndx++;
    }
    // case: the item in the right array is the lesser of the two elements
```

```
    else
    {
      // store the element in the original array
      A[main_ndx] = right_array[right_ndx];

      // move on to the next element of the right sub-array
      right_ndx++;
    }
  }

  // print the merge result for hw purposes
  cout << "After:  ";
  for( main_ndx = p; main_ndx <= r; main_ndx++ )
  {
    cout << A[main_ndx] << ' ';
  }
  cout << endl << endl;

  // no return - void
}
```

(b) Submit the output of your algorithm for the following input:
A = [A S O R T I N G E X A M P L E]

Solution:

```
Testing Problem 2 code by outputting
the sub-arrays to be processed
=======================================================
p: 0, q: 0, r: 1
Before: A S
After:  A S

p: 2, q: 2, r: 3
Before: O R
After:  O R

p: 4, q: 4, r: 5
Before: T I
After:  I T

p: 6, q: 6, r: 7
Before: N G
After:  G N

p: 8, q: 8, r: 9
Before: E X
After:  E X

p: 10, q: 10, r: 11
```

```
Before: A M
After:  A M

p: 12, q: 12, r: 13
Before: P L
After:  L P

p: 14, q: 14, r: 14
Before: E
After:  E

p: 0, q: 1, r: 3
Before: A S O R
After:  A O R S

p: 4, q: 5, r: 7
Before: I T G N
After:  G I N T

p: 8, q: 9, r: 11
Before: E X A M
After:  A E M X

p: 12, q: 13, r: 14
Before: L P E
After:  E L P

p: 0, q: 3, r: 7
Before: A O R S G I N T
After:  A G I N O R S T

p: 8, q: 11, r: 14
Before: A E M X E L P
After:  A E E L M P X

p: 0, q: 7, r: 14
Before: A G I N O R S T A E E L M P X
After:  A A E E G I L M N O P R S T X
```

3. Does the Merge procedure produce proper output if and only if the two input sub-arrays are in sorted order? Prove your answer, or provide a counterexample.

   Solution: In order to prove this, we first need to prove that Merge works if the sub-arrays are sorted, and will not work if they are not properly sorted.

   Merge works if sub-arrays are sorted.

   Invariant: The *output array* will always remain sorted; any elements remaining in the *sub-arrays* will be sorted and greater than any elements in the output array.

   Initialization: When the two sub-arrays are untouched and the *output array* is empty, then the output array is sorted because there are no contents to be disordered. Similarly, if only one element has been taken from either of the sub-arrays (the least element of the two) and placed in the output array, we consider this one element to be sorted; of course the sub-array is sorted.

   Maintenance: If items are taken from the sub-arrays one at a time, taking the least next remaining element of the two, and adding it to the output array, then the output array remains sorted. Any element still remaining in the sub-arrays must be greater than any element that was taken before it (due to our assumption that the sub-arrays are sorted), along with only taking the least available element ensures that the ouput array will not have been disordered. Taking only the next remaining element (which will be the least remaining) cannot disorder a sub-array.

   Termination: Once both sub-arrays are empty, they are still "sorted." The output array must be sorted (see the Maintenance Condition), an so upon termination, the output is both complete and sorted.

   Merge does not work if sub arrays are not sorted.

   Assume that Merge still works with unsorted sub-arrays. Consider a case where the sub-arrays' first available element is not sorted, but their second available element is their least element. The remaining elements can be in any order. As items are transferred to the output array, the first items are taken, but then once the second/least elements are reached and placed in the output array, the output array is instantly unsorted. A constradiction has arisen, so we know that only having sorted sub-arrays enables Merge success.

   Thus, the Merge procedure produces proper output if and only if the sub-arrays are sorted.

4. For Quicksort, in situations where there are large numbers of duplicate keys in the unput file, there is potential for significant improvement of the algorithm. One solution is to partition the file into **three** parts, one each for keys *smaller than*, *equal to* and *larger than* the partitioning element. In this approach the keys equal to the partitioning element that are encountered in the left partition are kept at the partition's left end and the keys equal to the partitioning element that are encountered in the right partition are kept at the partition's right end. **Implement** in C/C++ this partitioning strategy as follows: choose the pivot to be the last element of the array. Scan the file from the left to find an element that is not smaller than the partitioning element and from the right to find an element that is not larger than the partitioning element, then exchange them. If the element on the left (after the exchange) is equal to the partitioning element, exchange it with the one at the left end of the partition (similarly on the right). When the pointers cross, put the partitioning element between the two partitions, then exchange all the keys equal to it into position on either side of it (the figure on the right (not shown) illustrates this process. During partitioning the algorithm maintains the following situation:

equal	less	XXX	greater	equal	v

(indices/pointers not shown)
Illustrate the behavior of your algorithm on the input in the above figure by printing, after each iteration (left & right scan and eventual changes), the elements in the partitions as in the example above (not shown).

(a) Submit your code for the algorithm.

Solution:

```
template <typename ItemType>
void threePartQuicksort( ItemType* A, int partition_start, int partition_end )
{
  // variables
  int pivot_ndx = partition_end - 1;
  int left_ndx = partition_start;
  int right_ndx = pivot_ndx - 1;
  int left_equal_ndx = left_ndx;
  int right_equal_ndx = right_ndx;
  ItemType pivot = A[pivot_ndx];

  // output the array for hw purposes
  cout << "At the start of a new call:" << endl;
  arrayPrint( A, partition_start, left_equal_ndx, left_ndx,
              right_ndx, right_equal_ndx, pivot_ndx );

  // case: the partition size is larger than 1
  if( partition_start < pivot_ndx )
  {
    // perform scanning and swapping to create the left and right partitions
    while( left_ndx < right_ndx )
    {
      // swap the two items to correctly partition them
      swap( A[left_ndx], A[right_ndx] );
```

```
  // case: the item at the left index is equal to the pivot
  //       and there is a not-equal element to swap it with
  if( ( A[left_ndx] == pivot ) && ( left_equal_ndx < left_ndx ) )
  {
    // make the swap
    swap( A[left_ndx], A[left_equal_ndx] );

    // update the left-equal sub-partition and the left partition
    left_equal_ndx++;
    left_ndx++;
  }

  // case: the item at the right index is equal to the pivot
  //       and there is a-not equal element to swap it with
  if( ( A[right_ndx] == pivot ) && ( right_ndx < right_equal_ndx ) )
  {
    // make the swap
    swap( A[right_ndx], A[right_equal_ndx] );

    // update the right-equal sub-partition and the right partition
    right_equal_ndx--;
    right_ndx--;
  }

  // scan from the left for an element that is not less than the pivot
  while( ( A[left_ndx] < pivot ) && ( left_ndx < pivot_ndx ) )
  {
    // advance the left index
    left_ndx++;
  }

  // scan from the right for an element that is not greater than the pivot
  while( ( A[right_ndx] > pivot ) && ( right_ndx > partition_start ) )
  {
    // advance the left index
    right_ndx--;
  }

  // output the array for hw purposes
  cout << "After an iteration of scan/swapping:" << endl;
  arrayPrint( A, partition_start, left_equal_ndx, left_ndx,
              right_ndx, right_equal_ndx, pivot_ndx );
}

// case: the pivot is greater than all other elements in the partition
//       (no left partition can/should be made)
if( left_ndx < pivot_ndx )
{
  // move the pivot to create 3 partitions: right, middle, left
```

```
      swap( A[left_ndx], A[pivot_ndx] );
      right_ndx = left_ndx - 1;
      left_ndx++;

      // bump everything from the left-equal sub-partition to the middle
      while( left_equal_ndx > partition_start )
      {
        left_equal_ndx--;
        swap( A[left_equal_ndx], A[right_ndx] );
        right_ndx--;
      }

      // bump everything from the right-equal sub-partition to the middle
      while( right_equal_ndx < pivot_ndx )
      {
        right_equal_ndx++;
        swap( A[right_equal_ndx], A[left_ndx] );
        left_ndx++;
      }
    }

    // display the effects of this function call for homework purposes
    cout << "After pivot swapping and creating the middle partition:" << endl;
    arrayPrint( A, partition_start, left_equal_ndx, left_ndx,
                right_ndx, right_equal_ndx, pivot_ndx );

    // sort the left partition
    threePartQuicksort( A, partition_start, right_ndx + 1 );

    // sort the right partition
    threePartQuicksort( A, left_ndx - 1, partition_end );
  }
  // base case: the partition was of size 1 or less
    // do nothing

  // display the effects of this function call for homework purposes
  cout << "At the end of a call:" << endl;
  arrayPrint( A, partition_start, left_equal_ndx, left_ndx,
              right_ndx, right_equal_ndx, pivot_ndx );

  // no return - void
}
```

(b) Submit the output of your algorithm for the following input:
A = [A B R A C A C A B R A B C D C]

Solution:

```
Testing Problem 4 code by outputting
the array with various indices shown
=======================================================
At the start of a new call:
ABRACACABRABCDC
^             ^^
l             jr

After an iteration of scan/swapping:
DBRACACABRABCAC
^             ^^
l             jr

After an iteration of scan/swapping:
ABRACACABRABCDC
^ ^          ^^^
l i          jqr

After an iteration of scan/swapping:
CBAACACABRABRDC
^^  ^       ^ ^^
lp  i       j qr

After an iteration of scan/swapping:
CBAABACABRADRCC
^^    ^    ^ ^ ^
lp    i    j q r

After an iteration of scan/swapping:
CBAABAAABRRDCCC
^^       ^^ ^  ^
lp       ji q  r

After pivot swapping and creating the middle partition:
BBAABAAACCCCRDR
^       ^     ^^
l       j     ir

At the start of a new call:
BBAABAAA
^     ^^
l     jr

After an iteration of scan/swapping:
ABAABABA
```

```
^    ^^^
l    jqr

After an iteration of scan/swapping:
ABAABBAA
^  ^ ^ ^
l  j q r

After an iteration of scan/swapping:
ABABBAAA
^ ^ ^  ^
l j q  r

After an iteration of scan/swapping:
ABBBAAAA
^  ^   ^
l  q   r

After pivot swapping and creating the middle partition:
AAAAABBB
^    ^ ^
l    i r

At the start of a new call:




At the end of a call:




At the start of a new call:
    ABBB
    ^ ^^
    l jr

After an iteration of scan/swapping:
    BBAB
    ^ ^^
    l jr

After an iteration of scan/swapping:
    ABBB
    ^^^^
    lijr

After an iteration of scan/swapping:
    BABB
```

```
    ^^^^
    lpir

After pivot swapping and creating the middle partition:
    ABBB
    ^  ^
    l  r

At the start of a new call:
    A
    ^
    r

At the end of a call:
    A
    ^
    r

At the start of a new call:
       B
       ^
       r

At the end of a call:
       B
       ^
       r

At the end of a call:
    ABBB
    ^  ^
    l  r

At the end of a call:
AAAAABBB
^    ^ ^
l    i r

At the start of a new call:
            RDR
            ^^^
            ljr

After an iteration of scan/swapping:
            DRR
            ^^^
            lir

After pivot swapping and creating the middle partition:
            DRR
```

```
            ^ ^
            l r

At the start of a new call:
            D
            ^
            r

At the end of a call:
            D
            ^
            r

At the start of a new call:
              R
              ^
              r

At the end of a call:
              R
              ^
              r

At the end of a call:
            DRR
            ^ ^
            l r

At the end of a call:
AAAAABBBCCCCDRR
^       ^     ^^
l       j     ir
```

5. Problem 7-2 (page 186), parts (a), (c), and (d). For (c) and (d) assume that you have the procedure at point (b) available.
Quicksort with equal element values
The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

(a) Suppose that all the element values are equal. What would be randomized quicksort's running time in this case?

Solution: In this case, the partitioning would not really occur. Since no elements would be greater than any pivot, no left partition could ever be formed. With no left partition ever forming, the problem size could only decrease by one element. In determining that only one element could be reduced from the problem, all other elements had to be visited. Thus, we get:

$$\begin{aligned} Quicksort(n) &= \sum_{i=0}^{n-1} n - i \\ &= \frac{n^2 + n}{2} \\ Quicksort(n) &= \Theta\big(n^2\big) \end{aligned}$$

(b) **Not assigned.**
The PARTITION procedure returns an index q such that each element of A$[p \ldots q-1]$ is less than or equal to A$[q]$ and each element of A$[q+1 \ldots r]$ is greater than q. Modify the PARTITION procedure to produce a procedure PARTITION'[A, p, r], which permutes the elements of A$[p \ldots r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of A$[q \ldots t]$ are equal,
- each element of A$[p \ldots q-1]$ is less than A$[q]$, and
- each element of A$[t+1 \ldots r]$ is greater than A$[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta\big(r-p\big)$ time.

(c) Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION', and call the new procedure RANDOMIZED-QUICKSORT'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'(p, r) that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

Solution:

Algorithm *RANDOMIZED-QUICKSORT'*$(A(0 \ldots n-1), p, r)$
(∗ // Input: an array A[0 . . . n-1, 0 . . . n-1] of integer numbers ∗)
1. **if** p < r
2. q, t = RANDOMIZED-PARTITION'(A, p , r)
(∗ Two returns ∗)
3. RANDOMIZED-QUICKSORT'(A, p, q-1)
4. RANDOMIZED-QUICKSORT'(A, t+1, r)
5.

Algorithm *QUICKSORT'*$(A(0 \ldots n-1), p, r)$
(∗ // Input: an array A[0 . . . n-1, 0 . . . n-1] of integer numbers ∗)
1. **if** p < r
2. q, t = RANDOMIZED-PARTITION'(A, p , r)
(∗ Two returns ∗)
3. RANDOMIZED-QUICKSORT'(A, p, q-1)
4. RANDOMIZED-QUICKSORT'(A, t+1, r)
5.

(d) Using QUICKSORT', how would you adjust the analysis in 7.4.2 to avoid the assumption that all elements are distinct?

Solution: Because the PARTITION' procedure groups all of the elements with the same value of the pivot, the problem size is decreased by faster than typical quicksort. We can avoid the assumption that all elements are distinct by simply not assuming that all elements are distinct. Then, in the case that some elements are the same, our problem size is decreased by more than 1 with each recursive call. There is still a problem size perhaps less than n at each level of the tree, but there is still $\Theta(n)$ work at each level. The height of this tree will be $\Theta(\log_3 n)$, which is essentially $\Theta(\lg n)$. While the asymptotic notation may not show it, the new procedure will have a better time complexity in what would be the original QUICKSORT's worst case, but a worse time complexity if all of the elements are distinct.