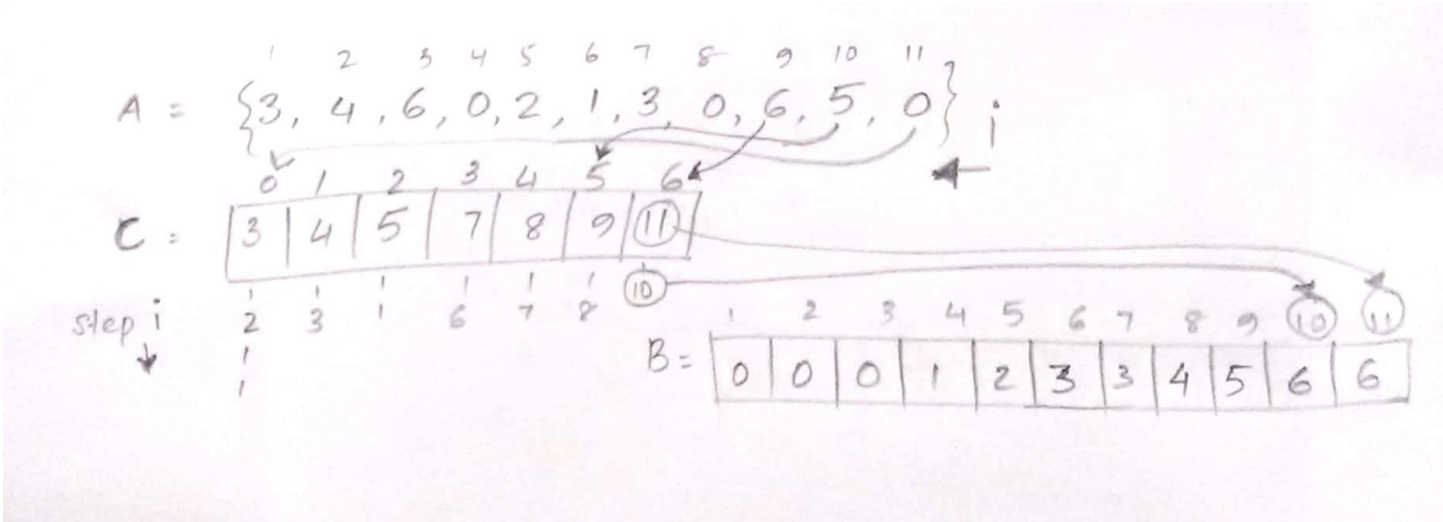
CSE 310 Assignment #3

akshayms@asu.edu

1. [5 pts] Write pseudo codes for the function *Change*(*A*, *heapsize, i, newValue*) that accesses an element at the index *i* in the max-heap *A* (Note that the index of this heap starts from 1), and changes its value to the value of the parameter *newValue*, and also maintains its max-heap property. Note that *newValue* can be larger or smaller than the value of *A*[*i*]. You can use the heap functions/algorithms we discussed in class to implement this function. Also if the value of *i* is out of the range, then it should not change anything in the heap.

```
void Change(A, heapsize, i, newValue){
if(i>heapsize)
        print ”heap out of bound”;
else
        A[i] = newValue;
        MAX-HEAPIFY(A,1);
        // calling MAX HEAPIFY function to make the tree MAX HEAP from the root
}
```

2. [6 pts] Illustrate the operation of COUNTING-SORT (*A, B, k*) on the array A = {3, 4, 6, 0, 2, 1, 3, 0, 6, 5, 0}. Note: you need to clearly mark array A, array B and counter array C's contents in each step (see pp. 195 Figure 8.2 as one example)



3. [6 pts] Suppose we use RANDOMZIED-SELECT(*A, p, r, i*) algorithm (pp. 215 Section 9.2) to select the minimum element of the array *A* = {3, 2, 9, 0, 7, 5, 4, 8, 6, 1}. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

A = { 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 } (WORST CASE)

randomly choose pivot as 9

{3, 2, 1, 0, 7, 5, 4, 8, 6, 9}

Quick SORT (A) with pivot 9 ⇒ {2, 1, 0, 7, 5, 4, 8, 6, 3, 9}

Left < 9 Right > 9

PARTITION LEFT ARRAY, and apply random sort

randomly select '8' as pivot

with pivot 8 ⇒ {2, 1, 0, 7, 5, 4, 3, 6, 8}.

⇒ {1, 0, 7, 5, 4, 3, 6, 2, 8}

left

with pivot 7 ⇒ {0, 2, 5, 4, 3, 6, 1, 7} with pivot 6 ⇒ {2, 5, 4, 3, 1, 0, 6}

left

with pivot 5 ⇒ {0, 4, 3, 1, 2, 5} with pivot 4 ⇒ {2, 3, 1, 0, 4}

with pivot 3 ⇒ {0, 1, 2, 3} with pivot 2 ⇒ {1, 0, 2}

with pivot 1 ⇒ {0, 1}

minimum element of A

4. [6 pts] In the algorithm we learned in class SELECT(*A, i*), the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

When we divide the input elements into sub lists of 7 elements, instead of 5, half of the sub lists has at least 4 elements greater than the pivot. We can ignore the sub list containing the pivot and the last one that can contain at most 7 elements.
Thus, the number of elements greater than pivot = $4[(1/2)(n/7)-2] = (2n/7)-8$
For similar reason the number of elements smaller than pivot is at least:$(2n/7)-8$
The recurrence part of the algorithm works on a problem of size at most, $n-((2n/7)-8) = (5n/7)+8$
∴ Running time $T(n) \leq T'((5n/7)+8) + T(n/7) + O(n)$
$T'((5n/7)+8)$ => applies Case#2 and Running time is $\Theta(n\log n)$ which is less than linear time.

Therefore, the algorithm's (T(n)) total running time is linear.

When we are dividing the input elements as a sub list of 3,
the subproblem size at the recurrence is at most $n-(n/3-4)=(2n/3+4)$
and the number of elements smaller or greater than the pivot is $2(1/2\times\lceil n/3\rceil-2) \geq (n3-4)$
we get the running time $T(n) \leq T(\lceil n/3\rceil) + T'((2n/3)+4)) + O(n)$

Here T(n) also looks linear for me, but according to the internet, it shouldn't be linear.
If it's not linear I need help solving this question.

5.

Execution time	*n =100*	*n =10000*	*n =1000000*	*n =100000000*
sumWithLoop	0. 000001	0.000037	0.003646	0.077308
sumWithLoop_OMP	0.000167	0.000159	0.005880	0.072525
sumRec	0.000002	0.000127	0.022748	0.287315
SumRec_OMP	0.000029	0.000154	0.043302	0.217181

```
int sumWithLoop(int * A, int n)
{
//this function will return sum calculate with loop function
        int sum=0;
        for(int i = 0; i < n; i++)
  {     //printf("%d ", A[i]);
    sum+= A[i];
  }

  return sum;
}

int sumWithLoop_OMP(int * A, int n)
{
// this function will calculate the sum with 4 threads and return it
        int sum=0,i;
        #pragma omp parallel shared(sum) private(i) //num_threads(4)
```

```
	{	//printf ( " Number of threads = %d", omp_get_max_threads ( ));
		#pragma omp for
		for(i = 0; i < n; i++ )
			{	//printf("%d ", A[i]);
				sum+= A[i];
			}

	}
	return sum;
}

int sumRec(int * A, int n, int start, int end)
{	int mid,lsum,rsum,rsize;
// this will return the sum calculated recursively
	 if (n == 0)
   {
      return 0;
   }
   else if (n == 1)
   {
      return A[0];
   }
   //printf("%p , %p ", A, A+mid);

   mid = n / 2;
   //mid = (start+end)/2;
   rsize = n - mid;
   lsum = sumRec(A, mid,start,end);
   rsum = sumRec(A+ mid,rsize,start,end);

   return lsum + rsum;

}

int sumRec_OMP(int * A, int n, int start, int end)
{
	int mid,lsum,rsum,rsize;
// this will return the parallelly calculated sum calculated recursively
	 if (n == 0)
   {
      return 0;
   }
   else if (n == 1)
   {
      return A[0];
   }
   //printf("%p , %p ", A, A+mid);

   mid = n / 2;
```

```
    //mid = (start+end)/2;
    rsize = n - mid;
    #pragma omp parallel num_threads(4)
    {
            #pragma omp sections
    {
            #pragma omp section
                        lsum = sumRec(A, mid,start,end);
                    #pragma omp section
                        rsum = sumRec(A+ mid,rsize,start,end);
    }
            }

    return lsum + rsum;

}
```