Name: Tuan-anh Vo
Date: 1/22/2013
Section: CSE331AC
Assignment: HW#2

1)

```
{ pre: size > 0 }
int k = 0;
int max = a[0];
{ inv: max is largest in a[0]...a[k]}
while ( k != size - 1) {
     { inv: max is largest in a[0]...a[k] }
     if (a[k+1] > max) {
          max = a[k+1];
        { max is largest in a[0]...a[k + 1] }
     } else {
          // do nothing
          { max is largest in a[0]...a[k + 1] }
     }
     k = k + 1;
     { inv: max is largest in a[0]...a[k] }
}
{ inv: max is largest in a[0].... a[k] & k == size - 1 --> max is largest in a[0]...a[size-1] }
{ post: max is largest in a[0]...a[size-1] }
```

At first, the invariant seemed to make k one larger from k-1 to k... maybe shifted the algorithm? We should not think of k as being the same, however, and that everything is shifted over. Instead, these are two different k but with a one-to-one mapping of every value of k-1 to a new value k in the new code. The choice of K was arbitrary and we end up looking at the exact same array indices either way. While we initialize k to 0 instead of k=1, we look at a[k+1] instead of a[k] respectively.... which correspond to exactly the same indices... THE ALGORITHM IS UNCHANGED. Since k is one larger than before, we had to change size to size-1, and a[k] to a[k+1]. This example shows that the CHOICE OF INVARIANT AFFECTS HOW WE SETUP OUT CODE, and the impact of the invariant indexing simply changes our choice of indexing for the code, but DIFFERENT INDEXING CHOICES MAY UTILIZE THE SAME ALGORITHM. It's not a matter of which method is easier or harder, it depends on a persons preference for 0 based indexing vs starting at 1.

---

2)

*The condition that x>=0 in the invariant is necessary because if the algorithm were to some how generate a negative value for x at the end, then x^0 would be a negative number. The precondition that x=x_pre>=0 at the beginning is not sufficient to guarantee this, we had to show that x never goes negative throughout the algorithm.
* I use the keyword new and old to refer to values of x and y at the start of each if/else clause, and the updated x & y values respectively. It would be confusing to refer to both old and new x with the same variable x. Moreover, using the word pre/post would conflict with the x_pre, y_pre of the function.

```
int expt(int x, int y) {
     {pre: (x == x_pre) & (y == y_pre) & (x_pre >= 0) & (y_pre >= 0) }
     int z = 1;
     { {x_pre^y_pre == 1*(x^y)) & (z == 1) & (x==xpre>=0) }
     { ( inv: x_pre^y_pre == z*(x^y) & x>=0) }
     while (y > 0) {
          { (inv: x_pre^y_pre == z*(x^y) & x>=0) & (y > 0)}
          if (y % 2 == 0) {
               { (inv: x_pre^y_pre == z*(x^y) & x>=0) & (y % 2 == 0) & (y > 0)}
               y = y/2;
               x = x*x;
               {(x_new == x_old^2) & (y_new = y_old/2) & (y_new > 0) & (x_new>=0)}
               { x_pre^y_pre == z*(x_new^y_new) == z*((x_old^2)^(y_old/2)) }
               {(inv: x_pre^y_pre == z*(x^y) & x>=0) }
          } else {
               { (inv: x_pre^y_pre == z*(x^y) & x>=0) & (y % 2 == 1) & y > 0}
               z = z*x;
               y = y-1;
               { (z_new == z_old * x) & (y_new == y_old-1) & (y_new >= 0) & (x>=0)}
               { x_pre^y_pre == z_new*(x^y_new) == z_old*x*(x^(y_old - 1)) }
               {(inv: x_pre^y_pre == z*(x^y) & x>=0)}
          }
          {inv: x_pre^y_pre == z*(x^y) & x>=0)
     }
     { ((inv: x_pre^y_pre == z*(x^y) & x>=0) & (y == 0)) -> (x_pre^y_pre == z*(x^0) & x >=0 ) }
     { x_pre^y_pre == z*1 }
     { post: x_pre^y_pre == z }
     return z;
}
```

---

3)

In the 3rd else clause, I decided not to do both L++ and R--. While incrementing just one of them is inefficient, it is easier to prove than having to worry about the case where if L + 1 == R, then L++ and R-- means L > R. Either ways, the running time is O(N). You might also notice that I went with the condition (L < R) in the while loop as opposed to (L != R). The problem is if we have an array of length 1, then L==1 & R==0, then we will enter the loop accessing a[L] where L >= a.length which causes index out of bound. The problem with the condition (L < R) now becomes !(L<R) == (L >= R). Knowing L >= R didn't allow me to conclude L==R, so I added a check that a.length > 1, this way we can add the invariant L<=R, which combined with L>=R gives us L==R. This code holds a[0] in the same position until all the other elements partitioned, then swaps a[0] into position L.

```
{Pre: A.length > 0}
if (a.length == 1) {
     return A;
     {POST: A is partitioned}
} else {
     {A.length > 1)
     int L = 1;
     int R = A.length - 1;
     {R==A.length-1 & R >=1 & L <= R}
     {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
     while(L < R) {
          {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
          if (A[L] <= A[0]) {
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
               {(A[L] <= A[0]) & (A[i] <= A[0] where ( 1 <= i <= L ))}
               L++;
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
          } else if (A[0] < A[R]) {
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
               {(A[0] < A[R]) & (A[0] < A[j] where ( R <= j <= A.length - 1 )) }
               R--;
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
          } else {
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
               {(A[0] < A[L] && A[R] <= A[0])} // from the other if/elseif being false
               swap(A[L], A[R]);
               {(A[L] <= A[0]) & (A[0] < A[R]) }
               {(A[i] <= A[0] where ( 1 <= i <= L )) & (A[0] < A[j] where ( R <= j <= A.length - 1 )) }
               L++;
               {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
          }
          {inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)}
     }
     {(inv: (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & (L <= R)) & (L >= R))
          -> (A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( R < j <= A.length - 1 )) & L==R }

     {(A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( L < j <= A.length - 1 ))}
     if (A[L] > A[0]) {
          {(A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( L < j <= A.length - 1 )) & (A[L] > A[0])}
          {(A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( L <= j <= A.length - 1 ))}
          L--;
          {(A[i] <= A[0] where ( 1 <= i <= L )) & (A[0] < A[j] where ( L < j <= A.length - 1 ))}
     } else {
          {(A[i] <= A[0] where ( 1 <= i < L )) & (A[0] < A[j] where ( L < j <= A.length - 1 )) & (A[L] <= A[0])}
          // do nothing
          {(A[i] <= A[0] where ( 1 <= i <= L )) & (A[0] < A[j] where ( L < j <= A.length - 1 ))}
     }
     {(A[i] <= A[0] where ( 1 <= i <= L )) & (A[0] < A[j] where ( L < j <= A.length - 1 ))}
     swap(A[L], A[0]);
     return A;
     {POST: A is partitioned... (A[i] <= A[0]_pre where ( 0 <= i <= L ))
          & (A[0]_pre < A[j] where ( L < j <= A.length - 1 )) & (A[L] == A[0]_PRE) & (0<=L<=A.length-1) }
}
```

---

4)

Note: In java, swap(a[j], min) will not work. Even if we write the 3 line switch for two values. Min is just a variable, rather than the array element itself.
We'd have to keep track of a minIndex everytime we update min such that a[minIndex] = min.

```
{pre: size > 0}
int j = 0
{inv: a[i] <= a[i+1] where 0 <= i < j & (a[i] <= a[k] where 0<=i<j & j<=k<size)}
while(j != size-1) {
     {inv: a[i] <= a[i+1] where 0 <= i < j}
     int minIndex = findMin(j, size-1);

     // Start findMin(j, size-1)
     int m = j + 1;
     min = a[j];
     {inv2: min = smallest in a[j..m-1]}
     while (m != size) {
          {inv2: min = smallest in a[j..m-1]}
          if (min > a[m]) {
               {min = smallest in a[j..m-1] & min < a[m]}
               min = a[m];
               {min = smallest in a[j..m]}
          else {
               // nothing to do
               {min = smallest in a[j..m]}
          }
          {min = smallest in a[j..m]}
          m = m+1;
          {inv2: min = smallest in a[j..m-1]}
     }
     {(inv2: min = smallest in a[j..m-1]) & (m = size)} => (min = smallest in a[j..size-1])}
     {min = smallest in a[j..size-1]}
     {(min <= a[k] where j <= k < size)}
     // End findMin(j, size-1) portion

     {(a[i] <= a[i+1] where 0 <= i < j) & (a[i] <= a[k] where 0<=i<j & j<=k<size) & (min <= a[k] where j <= k < size)}
     swap(a[j], min);
     {(a[i] <= a[i+1] where 0 <= i < j) & (a[i] <= a[k] where 0<=i<j & j<=k<size) & (a[j] <= a[k] where j <= k < size)}
     {(a[i] <= a[i+1] where 0 <= i <= j) & (a[i] <= a[k] where 0<=i<=j & j<=k<size)}
     {(a[i] <= a[i+1] where 0 <= i <= j) & (a[i] <= a[k] where 0<=i<=j & j<k<size)}
          // set of k values where j < k < size is a subset of j <= k < size from previous assertion.
     j++;
     {(a[i] <= a[i+1] where 0 <= i < j) & (a[i] <= a[k] where 0<=i<j & j<=k<size)}
}
{ (((inv: a[i] <= a[i+1] where 0 <= i < j) & (a[i] <= a[j] where 0<=i<j)) & j == size -1)
     --> (a[i] <= a[i+1] where 0 <= i < size-1)}
{post: a[i] <= a[i+1] where 0 <= i < size-1)
```