MC140: lecture #21

today's topic:

insertion sort

blort sort

insertion sort.

- insertion sort uses an auxiliary array
- it copies elements, one at a time, from the array being sorted into the auxiliary array
- the auxiliary array is always kept in sorted order, so as each entry is inserted, its existing entries are rearranged to make room for the new entry in its right place

insertion sort, 2.

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

/* define constant */
#define NUM_DICE 10

/* function prototypes */
int roll_die();
void roll_dice( int dice[],int size );
void print_dice( int dice[],int size );
void sort_dice( int *dice,int size );
void swap( int *a,int *b );
```

```
int main( void ) {

   int i, j;
   int dice[NUM_DICE];

   /* initialize random seed */
   srand( time ( NULL ));

   /* fill the dice array with
      random numbers */
   roll_dice( dice,NUM_DICE );

   /* print the dice */
   printf( "messy dice: " );
   print_dice( dice,NUM_DICE );

   /* sort the dice and print
      them again */
   sort_dice( dice,NUM_DICE );
   printf( "nice dice: " );
   print_dice( dice,NUM_DICE );

   return( 0 );

} /* end of main() */
```


insertion sort, 3.

```
int roll_die() {
  return(( rand() % 6 ) + 1 );
} /* end of roll_die() */

void roll_dice( int dice[], int size ) {
  int i;
  for ( i=0; i<size; i++ ) {
    dice[i] = roll_die();
  } /* end for i */
} /* end of roll_dice() */

void print_dice( int dice[], int size ) {
  int i;
  for ( i=0; i<size; i++ ) {
    printf( "%d ",dice[i] );
  } /* end for i */
  printf( "\n" );
} /* end of print_dice() */
```


insertion sort, 2.

```
void sort_dice( int dice[], int size ) {
   int aux[NUM_DICE], auxsize = 0;
   int i, j, k;
   for ( i=0; i<size; i++ ) {
        j = 0;
        while (( j < auxsize ) && ( dice[i] > aux[j] )) {
                j++;
        } /* end while */
        for ( k=auxsize; k>j; k-- ) {
                aux[k] = aux[k-1];
        } /* end for k */
        aux[j] = dice[i];
        auxsize++;
   } /* end for j */
   for ( i=0; i<size; i++ ) {
        dice[i] = aux[i];
   } /* end for i */
} /* end of sort_dice() */
```


sample run of insertion sort.

- *first, we insert 6 into aux.*

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 6 | | | | |

- *next, we insert 4 into aux; since 4 < 6, we shift 6 in aux.*

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 4 | 6 | | | |

- *next, we insert 5 into aux; since 5 < 6, we shift 6 in aux.*

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 4 | 5 | 6 | | |

- *next, we insert 2 into aux; since 2 < 4,5,6, we shift all of aux.*

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 2 | 4 | 5 | 6 | |

- *last, we insert 3 into aux; since 3 < 4,5,6, we shift those entries.*

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 2 | 3 | 4 | 5 | 6 |

sample run of insertion sort,2.

(copied from end of last slide)

dice = | 6 | 4 | 5 | 2 | 3 | aux = | 2 | 3 | 4 | 5 | 6 |

• *fiinally, we copy the contents of aux back to dice.*

dice = | 2 | 3 | 4 | 5 | 6 | aux = | 2 | 3 | 4 | 5 | 6 |

blort sort.

- blort sort is rather a silly sort:
- (1) the array being sorted is checked to see if it is in sorted order
- (2) if it is, then blort sort is done
- (3) if it isn't, then the elements in the array are *permuted* (randomly rearranged, or shuffled as in a deck of cards)
- then blort sort iterates, going back to (1)
- blort sort can take a long time to run!

blort sort, 2.

```
/* returns true if the dice array is sorted;
   false otherwise */
int isSorted( int dice[], int size ) {
  int i = 0, anyErrors = 0;
  while (( ! anyErrors ) && ( i < size-1 )) {
    if ( dice[i] > dice[i+1] ) {
      anyErrors = 1;
    }
    else {
      i++;
    }
  } /* end while */
  return( ! anyErrors );
} /* end of isSorted() */
```


blort sort, 3.

```
/* this function rearranges the elements in the dice
   array in random order */
void permute( int dice[], int size ) {
  int i, r;
  for ( i=0; i<size-1; i++ ) {
    r = (rand() % (size-i)) + i;
    swap( &dice[i],&dice[r] );
  } /* end for i */
} /* end of permute() */

/* this is the blort sort! */
void blortSort( int dice[],int size ) {
   while ( ! isSorted( dice,size )) {
       permute( dice,size );
   } /* end of while */
} /* end of blortSort() */
```


blort sort: sample run.

```
6 2 6 2 4
6 2 2 4 6
6 2 6 2 4
6 2 4 6 2
6 2 2 4 6
4 6 6 2 2
6 4 2 2 2
2 2 4 6 6      finally, it's sorted!
```


reading.

- the material covered today is not in the text book
- EXAM #3 will be on WED 11 APRIL