Week 09 Tutorial

Sorting

1. Consider the following simple table of enrolments, sorted by course code:

COMP1927	Jane	3970
COMP1927	John	3978
COMP1927	Pete	3978
MATH1231	John	3978
MATH1231	Adam	3970
PSYC1011	Adam	3970
PSYC1011	Jane	3970

   Now we wish to sort it by student name, so that we can easily see what courses each student is studying. Show an example of what the final array would look like if

   a. we used a *stable* sorting algorithm
   b. we used a *non-stable* sorting algorithm

2. How many comparisons would be required to sort this array

```
int a[5] = {6,8,3,4,2}
```

   for each of the following sorting algorithms

   a. selection sort
   b. adaptive bubble sort (ends when a pass mkes no swaps)

3. In lectures, we looked at a simple function to check whether an array of integers was sorted. A trickier problem is to determine whether the sort algorithm that produced a sorted array is stable. Assume that the array is composed of `Item`s with two fields, as in:

```
typedef struct { int a; int b; } Item;
```

   Assume also that the array has been sorted on the `Item.a` field.

   Given the final sorted array alone, you cannot determine whether the sort was stable. We also assume that the sorting client keeps a copy of the original array, which is available for checking.

   Given the above, write a function that takes the original array, the sorted array, and determines whether the sort was *stable*. The function is defined as follows:

```
int isStableSort(int original[], int sorted[], int lo, int hi) { ... }
```

4. Write a version of selection sort that builds a *new* sorted list from an original unsorted linked list. The original list should not be modified during the sorting. Use the definition for linked lists from above. The function is defined as:

```
List selectSort(List ls) { ... }
```

5. Implement an insertion sort algorithm on linked lists. Use the `List` data structure from the previous question.

   The insertion sort is encapsulated in a function called `sortList()` which is defined as follows:

```
List sortList(List L);
```

   `List L` is destroyed in the process of sorting.

   You can assume that you have a function `insertList()` which inserts a new item into a sorted list, retaining the order. The `insertList()` function is defined as follows:

```
List insertList(List L, Item it);
```

6. Sorting algorithms generally use comparison on the key to determine ordering. Ordering of items with equal keys has been left to the stability of the sorting algorithm used. Sometimes, it is important that items with the same key be ordered on a secondary key.

   Write a function that takes two items and determines an order based on the primary key `k` and then on the value of a secondary key `j`. The function has the following signature:

```
typedef struct item { int k; int j; /* ... other fields ...*/ } Item;

int itemCmp(Item a, Item b) { ... }
```

7. Write a program that reads data in the form:

```
5059413 Daisy  3762  15
3461045 Dotty  3648  42
3474881 Daisy  8543  16
5061020 Dave   3970  3
```

   from standard input and then ...

   - stores it in an array of `Student` structs `{zid,name,prog,magic}`
   - uses `qsort()` to sort it, making use of the `stuCmp()` function (see below)
   - order is initially by name, and then by zID if the names are the same
   - writes out sorted array, one `Student` per line
   - use the following format for printing students `"%7d %-20s %4d %d\n"`

   You can assume that all of the input lines contain valid student entries, and that the `MAX`*xxx* constants are defined, and large enough to deal with any input data. You do not need to implement the `stuCmp()` function; just assume it exists.

Use the following template for the program:

```
typedef struct _student {
   int  zid;            // 7-digit number
   char name[MAXNAME];  // names are stored *within* the struct
   int  prog;           // 4-digit number
   int  magic;          // random number?
} Student;

// return -ve if a < b, +ve if a > b, 0 if a == b
int stuCmp(Student a, Student b);

int main(int argc, char **argv)
{
   Student students[MAXSTUDES];
   int nstudes = 0;
   char line[MAXLINE];

   // read stdin line-by-line into students[]

   // sort the students[] array

   // print the contents of the students[] array

   return;
}
```

8. The code for `mergesort()` requires finding the midpoint of the list and splitting the list there:

```
 1. List mergesort(List c)
 2. {
 3.    List a, b;
 4.    if (c == NULL || c-&gt;next == NULL) return c;
 5.    a = c;  b = c->next;
 6.    while (b != NULL && b->next != NULL)
 7.       { c = c->next; b = b->next->next; }
 8.    b = c->next; c->next = NULL; // split list
 9.    return merge(mergesort(a), mergesort(b));
10. }
```

How would this code change if `Lists` were defined as

```
1. typedef struct _listNode {
2.    Item  item;
3.    Node *next;
4. } Node;
5. typedef struct _listRep {
6.    List *nodes;
```

```
7.    int   length;
8. } ListRep;
9. typedef ListRep *List;
```

9. Write a program that reads two sorted files and merges them onto it standard output. The names of the files are provided as command-line arguments.

```
$ ./merge  File1  File2
```

If either file is not readable, exit with the error message:

```
Invalid input file(s).
```

10. Consider the following shellSort algorithm (based on Sedgewick's implementation):

```
1. void shellSort(int a[], int lo, int hi)
2. {
3.    int i, j, h, val;
4.    for (h = 1; h <= (hi-lo)/9; h = 3*h+1) ;
5.    for ( ; h > 0; h /= 3) {
6.       for (i = lo+h; i <= hi; i++) {
7.          j = i;  val = a[i];
8.          while (j >= lo+h && val < a[j-h]) {
9.             a[j] = a[j-h];
10.            j -= h;
11.         }
12.         a[j] = val;
13.      }
14.   }
15. }
```

Describe how this would sort an array containing the first 100 positive integers sorted in descending order (i.e. {100,99,98,97,96,95,...1}).
