Computer Programming: Skills and Concepts
Tutorial 6(week 8): November 8 - 11, 2010

To accompany this tutorial sheet there is an electronic file, tutw8.tar, with some c-programs inside. Please download (and unpack, by typing `tar -xvf tutw8.tar`) this file from the course webpage:
`http://www.inf.ed.ac.uk/teaching/courses/cp1/`

Pointer revision

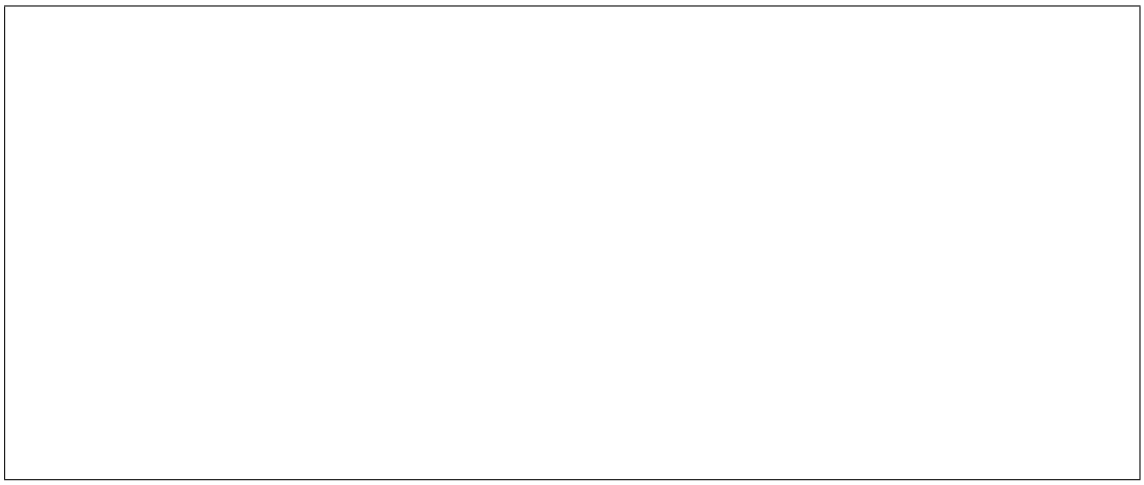
Consider the following functions with the following `main`. Describe the relationships between integer-values and pointer-values from the various `printf` statements (including those called from within the functions).

What is the difference between variables `n` and `m`?
How does it manifest itself during the running of the code?

```
void addressOnly(int n) {
  printf("addressOnly: address of parameter-var n is %p, val %d.\n", &n, n);
}

void valueOfPoint(int *p) {
  printf("valueOfPoint: value of pointer p is %p, points to val %d.\n", p, *p);
}
int m=8;

int main(void) {
  int n=5;
  printf("main: val of n is %d, address of n is %p.\n", n, &n);
  addressOnly(n);
  printf("main: Passing &n into valueOfPoint.\n");
  valueOfPoint(&n);
  printf("\n");
  printf("main: val of m is %d, address of m is %p.\n", m, &m);
  addressOnly(m);
  printf("main: Passing &m into valueOfPoint.\n");
  valueOfPoint(&m);
  return EXIT_SUCCESS;
}
```



Running-time of BubbleSort/MergeSort

This is a *programming task*. Please make an effort to try this out before you meet your tutor!!!

In the `sort.tar` file that was been available for lecture 19 there are three files inside - `bubblesort.c`, `mergesort.c` and `mergesmall.c`. The implementation `mergesort.c` contains code for the MergeSort algorithm. Its '`main`' function has code which asks the user for a positive integer, creates an array of that length, and *times* the execution of MergeSort on that array.

You have two tasks:

1. Adapt the 'main' function so that instead of asking the user for a particular length of array, instead create a (wrap-around) `for`-loop which will run the tests for lengths of 50000, 100000, 200000, 400000, ... (doubling at each step). The number of times you do this doubling will depend on the available-memory on the system where you run it (so a bit of trial-and-error may be needed)

2. Next add a similar section of code to the '`main`' of `bubblesort.c`. Run these experiments in order to find a practical comparison between BubleSort and MergeSort.

pre-Increment vs post-Increment

This is tricky.

In class on Thursday, 4th November (lecture slides 20), we discussed the difference between using `x++` and `++x` within another expression.

Consider the `merge` function (lecture slides 19, see below). Write a version `merge2` which uses `++i`, `++j`, `++k` within `a[++i]`, `b[++j]`, `c[++k]` (*instead* of using `i++`, `j++`, `k++`). You will need to make some changes to make sure that `merge2` performs the merging correctly.

```
void merge(int a[], int b[], int c[], int m, int n) {
  int i=0, j=0, k=0;
  while (i < m && j < n) {
    if (a[i] <= b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];
  }
  /* At this stage, done comparing.  All the rest of c will come
   * from either a, or else b, depending on which has stuff left. */
  while (i < m)
    c[k++] = a[i++];
  while (j < n)
    c[k++] = b[j++];
}
```