Computer Programming: Skills and Concepts
Solutions for week 8 Tutorial — November 16th-20th, 2015

Sorting

(i) Put the following words in lexicographical order using the bubble sort algorithm. How many swaps did you have to do?

hat, operate, but, not, the, other, bulb

Answer: Check the BubbleSort code in `bubblesort.c`. n for us is 7.

outside loop for $i = 6$:

j=0 no swap, a[1] $\leftrightarrow$ a[2], a[2] $\leftrightarrow$ a[3], no swap for j=3, a[4] $\leftrightarrow$ a[5], a[5] $\leftrightarrow$ a[6] $\Rightarrow$ 4 swaps.

array is then **hat, but, not, operate, other, bulb** | **the**.

outside loop for $i = 5$:

a[0] $\leftrightarrow$ a[1], j=1 no swap, j=2 no swap, j=3 no swap, a[4] $\leftrightarrow$ a[5] $\Rightarrow$ 2 swaps.

array is then **but, hat, not, operate, bulb** | **other, the**.

outside loop $i = 4$:

j=0, j=1, j=2 no swaps, a[3] $\leftrightarrow$ a[4] $\Rightarrow$ 1 swap

array is then **but, hat, not, bulb,** | **operate, other, the**.

outside loop $i = 3$:

j=0, j=1, no swaps, a[2] $\leftrightarrow$ a[3] $\Rightarrow$ 1 swap

array is then **but, hat, bulb,** | **not, operate, other, the**.

outside loop $i = 2$:

j=0, no swap, a[1] $\leftrightarrow$ a[2] $\Rightarrow$ 1 swap

array is then **but, bulb,** | **hat, not, operate, other, the**.

outside loop $i = 1$:

a[0] $\leftrightarrow$ a[1] $\Rightarrow$ 1 swap

array is then **bulb,** | **but, hat, not, operate, other, the**.

Total number of swaps: 4 + 2 + 1+ 1 + 1 + 1 = 10.

(ii) This time we will use MergeSort. In this case, count the number of times *any call of* `merge` places a value currently stored in `b` (the second array) before some of the `a` (first array) values, in creating the output array `c`.

Answer:

First call to mergesort: $n = 7$, so $n/2$ is 3.

$\Rightarrow$ two recursive calls on ArrL ={**hat, operate, but**}, and ArrR= {**not, the, other, bulb**} (will count rearrangements in the recursive calls later).

Top level: mergesort(ArrL) returns {**but, hat, operate**} and mergesort(ArrR) returns {**bulb, not, other, the**}.

merge({**but, hat, operate**}, {**bulb, not, other, the**}):
bulb goes first (+1), then **but, hat**, then **not** (+1), then **operate**, then **other, the**. So count 2 at top level.

ArrL ={**hat, operate, but**}: $n = 3$ so $n/2$ is 1.

⇒ two recursive calls on ArrLL ={**hat**} and ArrLR = {**operate, but**}. Within mergesort({**hat**}), no rearrangements. Within mergesort({**operate, but**}), just 1.

merge({**hat**}, {**but, operate**}): **but** goes first (+1), then **hat**, then **operate**.

So count of 1 + 1 = 2 for all levels of mergesort({**hat, operate, but**})

ArrR= {**not, the, other, bulb**}: $n = 4$ so $n/2$ is 2.

⇒ two recursive calls on ArrRL = {**not, the**} and ArrRR = {**other, bulb**}. mergesort({**not, the**}) has no rearrangments, mergesort({**other, bulb**}) 1 rearrangement to return {**bulb, other**}.

merge({**not, the**}, {**bulb, other**}): **bulb** goes first (+1), then **not**, then **other** (+1), finally **the**.

So count of 1 +2 = 3 over all levels of mergesort({**not, the, other, bulb**})

Overall we have 2 + 2+ 3 = 7.

(iii) Our implementation of BubbleSort was for arrays of type `int`. Describe how to alter the implementation to deal with arrays of strings (ie, arrays of `char*`).

Answer:

Comparisons: Comparison of strings cannot be done via `>`, `<`, `>=` or `<=`. We have to use the string function `strcmp` which belongs to the library `<string.h>`. Remember that `strcmp(s,t)` returns a -ve value if `s` is lexicographically less than `t`, 0 if the two strings are equal, and a +ve value if `s` is lexicographically greater than `t`.

In `BubbleSort` the tests are of the form `a[j] > a[j+1]`.

So in the string version, we would write this test as `strcmp(a[j], a[j+1]) > 0`.

Other changes:

These actually depend on whether the array of strings is of type `char[][SIZE]` for some fixed *string length* `SIZE`. Alternatively it could be of type `char*[]`, but then the individual strings in the array would need to have storage allocated for them via `malloc`.

The first approach is in `bubblestrA.c` - the function prototype [1] for BubbleSort is:

```
void BubbleSort(char a[][SIZE], int n)
```

Also, because each string `a[i]` is stored in a contiguous static block of `char` cells in memory, we have to use `strcpy` in performing the swaps within `BubbleSort`. So the test-and-swap code now looks like[2]:

```
if (strcmp(a[j],a[j+1]) > 0) {
  strcpy(temp, a[j]);
  strcpy(a[j], a[j+1]);
  strcpy(a[j+1], temp);
```

Also `temp` will have been declared as `char temp[SIZE]`.

[1] Other case has prototype `void BubbleSort(char *a[], int n)`.

[2] In other case, don't need `strcmp`: `a` holds non-static pointer values (declare `temp` as `char* temp;`). The code for this approach is in `bubblestrB.c`

Programming

1. The log* function (different from log itself - the * is important), which is applied to a number n, is defined to be the number of times one has to repeatedly apply log to n before the answer becomes less than or equal to 1. For example, assuming base 10 logs, we have that:

$\log^*(5) = 1$, $\log^*(50) = 2$, $\log^*(500) = 2$, $\log^*(5000000) = 2$,
$\log^*(10000000000) = 2$, $\log^*(10000000001) = 3$.

The "log-to-the-base-10" function in the `<math.h>` library is named `log10`. It has the function prototype `double log10 (double)`.

Your first task is to write a *recursive* `logstar` ("log*") function, with function prototype: `int logstar(double x)`

Answer:

```
int logstar(double x) {
  if (x <= 0)
    return -1;
  else if (x <= 1)
    return 0;
  else
    return (1+logstar(log10(x)));
}
```

2. Now write a new function `logstar2` which computes the same answer but using *iteration* (ie, a `for` or a `while`) rather than recursion.

Answer:

```
int logstar2(double x) {
  int d = 0;
  if (x <= 0)
    return -1;
  else {
    while (x > 1) {
      x = log10(x);
      d++;
    }
    return d;
  }
}
```