Supplementary Material For S3E7 Problem S3CS1 (Computer Science)

This file contains the supplementary material for S3E7 problem S3CS1. In particular, below we present the C code that sorts 3 and, separately, 4 integers creatively as explained in the corresponding episode.

---

3 Integers

Below we show the C code that implements the 3-compare-and-swap algorithm that we discussed in S3E7 and that can be readily compiled and tested.

Save the following in a disk file named `sort3ints.c`, for example:

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

void
swap( int *x, int *y )
{
     int       tmp = *x;

     *x = *y;
     *y = tmp;
}

size_t
sortA( int *a, int *b, int *c )
{
     size_t         nOfSwaps = 0;

     if ( *a > *b )
     {
          nOfSwaps++;
          swap( a, b );
     }

     if ( *a > *c )
     {
          nOfSwaps++;
          swap( a, c );
```

```
     }

     if ( *b > *c )
     {
          nOfSwaps++;
          swap( b, c );
     }

     return nOfSwaps;
}

extern int
main( int argc, char* argv[] )
{
     size_t         nOfSwaps = 0;
     int       a = atoi( argv[ 1 ] );
     int       b = atoi( argv[ 2 ] );
     int       c = atoi( argv[ 3 ] );

     printf( "Before the sort:\n\t %d %d %d\n",
          a, b, c );

     nOfSwaps = sortA( &a, &b, &c );

     printf( "After the %zu-swap(s) sort:\n\t %d %d %d\n",
          nOfSwaps, a, b, c );

     return 0;
}
```

Open a terminal window on any UNIX-like machine and type in the following command:

```
gcc -g -o sort3ints sort3ints.c
```

After the above compilation is over, give it a test:

```
./sort3ints 9 7 5
Before the sort:
      9 7 5
After the 3-swap(s) sort:
      5 7 9
```

The code shown above should be self-explanatory.

The `swap()` function does what it name implies, it swaps the values of the two integers passed into it.

The `sortA()` function creatively sorts the three integers passed into it in an **A**scending order and returns the number of value-swaps performed.

The `main()` function accepts the three integers of interest from the command line, prints their values before the sort, sorts these integers creatively in an ascending order and prints the sorted values out.

---

4 Integers

The 4-integer sort can be done in 5 tests. Below we show the only code that is different from the above 3-integer sorting code, the `sortA()` function:

```
void
sortA( int *a, int *b, int *c, int *d )
{
     if ( *a > *b )
          swap( a, b );

     if ( *a > *c )
          swap( a, c );

     if ( *b > *c )
          swap( b, c );

     if ( d > b )
     {
          if ( c > d )
               swap( c, d );
     }
     else
     {
          if ( a > d )
               swap( a, d );
     }
}
```

Save the above code in a disk file named `sort4ints.c` and compile and test it as was explained above:

```
gcc -g -o sort4ints sort4ints.c
```

and so on.

Going Wide. Extra For Experts

Below we show a language-specific way to implement custom sorting functions for objects of any type that can be sensibly compared in the C programming language.

The main tactical idea here is to define a custom comparator function and to operate in terms of "pointer to a pointer to" fashion.

Save the code shown below in a disk file named `sort3objs.c`, for example:

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

typedef int ( *larger_than_cmp_t )( void*, void* );

int
intLargerThan( void *a, void *b )
{
     int       *A = ( int* )a;
     int       *B = ( int* )b;

     if ( *A > *B )
          return 1;

     return 0;
}

void
swap( void **x, void **y )
{
     void      *tmp = *x;

     *x = *y;
     *y = tmp;
}

void
sortA( void **A, void **B, void **C, larger_than_cmp_t largerThan )
{
     if ( largerThan( *A, *B ) )
```

```
          swap( A, B );

     if ( largerThan( *A, *C ) )
          swap( A, C );

     if ( largerThan( *B, *C ) )
          swap( B, C );
}

extern int
main( int argc, char* argv[] )
{
     int       a = atoi( argv[ 1 ] );
     int       b = atoi( argv[ 2 ] );
     int       c = atoi( argv[ 3 ] );

     int       *pA = &a;
     int       *pB = &b;
     int       *pC = &c;

     printf( "Before the sort:\n\t %d %d %d\n",
          a, b, c );

     sortA( ( void** )&pA, ( void** )&pB, ( void** )&pC,
intLargerThan );

     printf( "After the sort:\n\t %d %d %d\n",
          *pA, *pB, *pC );

     return 0;
}
```

Compile that file like so:

```
gcc -g -o sort3objs sort3objs.c
```

and test it like so:

```
./sort3obj 9 7 5
Before the sort:
      9 7 5
After the sort:
      5 7 9
```

Feel free to modify the above code as you see fit. For example. If the input must be sorted in a descending order then the less-than comparator should be implemented instead:

```
typedef int ( *less_than_cmp_t )( void*, void* );
```

which for integers will looks as follows:

```
int
intLessThan( void *a, void *b )
{
     int       *A = ( int* )a;
     int       *B = ( int* )b;

     if ( *A < *B )
          return 1;

     return 0;
}
```

and so on.

If you have any questions then watch the corresponding episode and use its comments section.

This document is a companion for the Season 3 Episode 7.