CSC 407: Computer Systems II: 2019 Spring

Assignment #1

Purpose:

To go over:

- Compiler optimizations
- Program profiling (timing)
- Header files
- Linking and object file layout

Computing

Please ssh into one of the following:

- 140.192.36.184
- 140.192.36.185
- 140.192.36.186
- 140.192.36.187

or use your own Linux machine.

Please submit a .zip file (*not* .7z or any other non-standard compression!) file of your header file *and* a .txt/.pdf/.doc/.odt file containing your answer to the questions.

1. **Please copy-and-paste the following files (0 Points):**

   insertionSort.c

```
/*--------------------------------------------------------------
-------------*
 *----
  ----*
 *----           insertionSort.c
  ----*
 *----
  ----*
```

```
 *----       This file defines a function that implements
insertion----*
 *----   sort on a linked-list of integers.
  ----*
 *----
  ----*
 *----   ----   ----   ----   ----   ----   ----   ----
  ----   ----*
 *----
  ----*
 *----   Version 1a            2019 April 8          Joseph
Phillips ----*
 *----
  ----*
 *-------------------------------------------------------------
-------------*/

#include "headers.h"


// PURPOSE:  To sort the linked-list of nodes pointed to by
'nodePtr' using
//       the insertion sort algorithm.  Returns the first node
of the sorted
//       list.
struct Node*    insertionSort  (struct Node*  nodePtr
                                )
{

  struct Node*  startPtr       = NULL;
  struct Node*  endPtr         = NULL;
  struct Node*  lowestPtr;
  struct Node*  lowestPrevPtr;

  while  (nodePtr != NULL)
  {
    struct Node* prevPtr;
    struct Node* run;
```

```
  lowestPrevPtr        = NULL;
  lowestPtr            = nodePtr;

  for  (prevPtr = nodePtr, run  = nodePtr->nextPtr_;
                    run != NULL;
 prevPtr = run,     run = run->nextPtr_
)
  {
    if  (lowestPtr->value_ > run->value_)
    {
lowestPtr      = run;
lowestPrevPtr  = prevPtr;
    }
  }

  if  (lowestPrevPtr == NULL)
  {
    if  (startPtr == NULL)
    {
startPtr = endPtr      = lowestPtr;
    }
    else
    {
endPtr->nextPtr_       = lowestPtr;
endPtr                 = endPtr->nextPtr_;
    }

    nodePtr            = nodePtr->nextPtr_;
    endPtr->nextPtr_   = NULL;
  }
  else
  {
    if  (startPtr == NULL)
    {
startPtr = endPtr      = lowestPtr;
    }
    else
```

```
      {
  endPtr->nextPtr_       = lowestPtr;
  endPtr                 = endPtr->nextPtr_;
      }

      lowestPrevPtr->nextPtr_   = lowestPtr->nextPtr_;
      endPtr->nextPtr_          = NULL;
    }
  }

  print(startPtr);
  return(startPtr);
}
```

mergeSort.c

```
/*-------------------------------------------------------------
-------------*
 *----
  ----*
 *----           mergeSort.c
  ----*
 *----
  ----*
 *----       This file defines a function that implements
merge-sort on    ----*
 *----   a linked-list of integers.
  ----*
 *----
  ----*
 *----   ----   ----   ----   ----   ----   ----   ----
  ----   ----*
 *----
  ----*
 *----   Version 1a            2019 April 8          Joseph
Phillips ----*
 *----
  ----*
```

```
 *-------------------------------------------------------------
------------*/

#include "headers.h"


//  PURPOSE:  To sort the linked-list of nodes pointed to by
'nodePtr' using
//        the merge sort algorithm.  Returns the first node of
the sorted       list.
struct Node*     mergeSort      (struct Node*  nodePtr
                        )
{

  if  ( (nodePtr == NULL) || (nodePtr->nextPtr_ == NULL) )
  {
    return(nodePtr);
  }

  struct Node*   run;
  struct Node*   run2;
  struct Node*   lastPtr = NULL;

  for  ( run = run2 = nodePtr;
         (run2 != NULL) && (run2->nextPtr_ != NULL);
   lastPtr = run, run = run->nextPtr_,  run2 = run2->nextPtr_-
>nextPtr_
       );

  lastPtr->nextPtr_      = NULL;
  run2                   = mergeSort(run);
  run                    = mergeSort(nodePtr);

  nodePtr= NULL;
  lastPtr= NULL;

  while  ( (run != NULL) && (run2 != NULL) )
  {
```

```
  if  (run->value_ < run2->value_)
  {
    if  (nodePtr == NULL)
    {
      nodePtr = lastPtr       = run;
    }
    else
    {
lastPtr = lastPtr->nextPtr_    = run;
    }

    run= run->nextPtr_;
  }
  else
  {
    if  (nodePtr == NULL)
    {
      nodePtr = lastPtr       = run2;
    }
    else
    {
lastPtr = lastPtr->nextPtr_    = run2;
    }

    run2       = run2->nextPtr_;
  }
}

if  (run == NULL)
{
  if  (lastPtr == NULL)
  {
    nodePtr           = run2;
  }
  else
  {
    lastPtr->nextPtr_ = run2;
  }
```

```
  }
  else
  {
    if  (lastPtr == NULL)
    {
      nodePtr           = run;
    }
    else
    {
      lastPtr->nextPtr_ = run;
    }
  }

  return(nodePtr);
}

struct Node*     mergeSortWrapper(struct Node* nodePtr
                                  )
{
  nodePtr= mergeSort(nodePtr);

  print(nodePtr);
  return(nodePtr);
}
```

2. C programming (20 Points):

These two files need a `main()` to run their functions `insertionSort()` and `mergeSortWrapper()`. Then all three C files need a header file to inform them of what the others have that they need, including `Node.h` which defines the data-structure. Please finish both the `main.c` and `headers.h`

- Please make `print()` print the whole linked list.
- For `headers.h`, not everything needs to be shared.
  - `main()` needs `insertionSort()` and `mergeSortWrapper()`
  - Both `insertionSort()` and `mergeSortWrapper()` need `print()`.

Otherwise, it is best *not* to share too much, kind of like keeping methods and members `private` in C++ and Java.

headers.h

```
/*-------------------------------------------------------------
-------------*
 *----
  ----*
 *----           headers.h
  ----*
 *----
  ----*
 *----       This file declares common headers used through-
out the  ----*
 *----   the singly-linked list sorting program.
         ----*
 *----
  ----*
 *----   ----    ----    ----    ----    ----    ----    ----
  ----   ----*
 *----
  ----*
 *----   Version 1a             2019 April 8          Joseph
Phillips ----*
 *----
  ----*
 *-------------------------------------------------------------
-------------*/

#include <stdlib.h>
#include <stdio.h>
#include "Node.h"

//  YOUR CODE HERE
```

Node.h

```
/*-------------------------------------------------------------
-------------*
 *----
  ----*
 *----           Node.h
  ----*
 *----
  ----*
 *----       This file declares the struct that stores an
integer and      ----*
 *----   a next-pointer to implement a node in a singly-linked
list.    ----*
 *----
  ----*
 *----   ----   ----   ----   ----   ----   ----   ----
  ----   ----*
 *----
  ----*
 *----   Version 1a            2019 April 8          Joseph
Phillips ----*
 *----
  ----*
 *-------------------------------------------------------------
-------------*/

struct   Node
{
  int            value_;
  struct Node*   nextPtr_;
};
```

main.c

```
/*-------------------------------------------------------------
-------------*
 *----
  ----*
```

```
 *----           main.c
  ----*
 *----
  ----*
 *----       This file defines the main functions for a
program that     ----*
 *----   sorts a linked list of randomly-generated integers.
  ----*
 *----
  ----*
 *----   ----   ----   ----   ----   ----   ----   ----
  ----   ----*
 *----
  ----*
 *----   Version 1a            2019 April 8          Joseph
Phillips ----*
 *----
  ----*
 *---------------------------------------------------------
-------------*/

#include "headers.h"

#define          TEXT_LEN       256
#define          NUM_NUMBERS    65536

const int numNumbers     = NUM_NUMBERS;


//  PURPOSE:  To create and return the address of the first
node of a linked
//       list of 'length' struct Node instances, each with a
randomly-generated
//       integer in its 'value_' member variable.
struct Node*     createList     (int    length
                                 )
{
  if  (length == 0)
```

```
  {
    return(NULL);
  }

  struct Node*   startPtr       = (struct
Node*)malloc(sizeof(struct Node));
  struct Node*   endPtr         = startPtr;

  startPtr->value_              = rand() % 4096;
  startPtr->nextPtr_            = NULL;

  for  (length--;  length > 0;  length--)
  {
    endPtr->nextPtr_            = (struct
Node*)malloc(sizeof(struct Node));
    endPtr->nextPtr_->value_    = rand() % 4096;
    endPtr->nextPtr_->nextPtr_  = NULL;
    endPtr                      = endPtr->nextPtr_;
  }

  return(startPtr);
}


//  PURPOSE:  To print integer values in the linked list
pointed to by
//       'nodePtr'.  No return value.
void            print         (const struct Node*    nodePtr
                      )
{
  //  YOUR CODE HERE
}


//  PURPOSE:  To 'free()' the 'struct Node' instances of the
linked list
//       pointed to by 'nodePtr'.  No return value.
void            freeList      (struct Node*  nodePtr
```

```
                        )
{
  struct Node*   nextPtr;

  for  ( ;  nodePtr != NULL;  nodePtr = nextPtr)
  {
    nextPtr      = nodePtr->nextPtr_;
    free(nodePtr);
  }

}


//  PURPOSE:  To run this program.  Ignores command line
arguments.  Returns
//        'EXIT_SUCCESS' to OS.
int             main          ()
{
  int           choice;
  struct Node*  nodePtr = createList(numNumbers);

  print(nodePtr);

  do
  {
    char text[TEXT_LEN];

    printf
  ("How do you want to sort %d numbers?\n"
   "(1) Insertion sort\n"
   "(2) Merge sort\n"
   "Your choice (1 or 2)? ",
   NUM_NUMBERS
  );
    fgets(text,TEXT_LEN,stdin);
    choice = strtol(text,NULL,10);
  }
  while  ( (choice < 1) || (choice > 2) );
```

```
  switch  (choice)
  {
  case 1 :
    nodePtr      = insertionSort(nodePtr);
    break;
  case 2 :
    nodePtr      = mergeSortWrapper(nodePtr);
    break;
  }

  freeList(nodePtr);
  return(EXIT_SUCCESS);
}
```

Sample Initial Output:

```
$ ./assign1
...
53        1936    2909    151     65      2884    3534    3826
  1564    2806
1611      640     2004    751     3304    3327    1724    1759
  2947    1425
2399      1488    1365    2425    2998    2945    1864    392
  3813    3099
1013      3966    939     3923    21      1004    2711    3555
  734     180
2265      2346    820     173
How do you want to sort 65536 numbers?
(1) Insertion sort
(2) Merge sort
Your choice (1 or 2)? 2
...
4093      4093    4093    4093    4093    4093    4093    4093
  4093    4093
4093      4093    4093    4094    4094    4094    4094    4094
  4094    4094
4094      4094    4094    4094    4094    4094    4094    4095
  4095    4095
```

4095 4095 4095 4095 4095 4095 4095 4095
4095 4095
4095

3. **Timing: Part 1 (20 Points):**

   Compile and run the program without any extra optimizations, but with *profiling* for timing:

```
gcc -c -pg -O0 main.c
gcc -c -pg -O0 mergeSort.c
gcc -c -pg -O0 insertionSort.c
gcc main.o mergeSort.o insertionSort.o -pg -O0 -o assign1-0
```

   Run the program twice timing it both times, and answer the following:

   a. *How many **self seconds** did `insertionSort()` take?*
   b. *How many **self seconds** did `mergeSort()` take?*

4. **Timing: Part 2 (20 Points):**

   Compile and run the program *with* optimization, but with *profiling* for timing:

```
gcc -c -pg -O2 main.c
gcc -c -pg -O2 mergeSort.c
gcc -c -pg -O2 insertionSort.c
gcc main.o mergeSort.o insertionSort.o -pg -O2 -o assign1-2
```

   Run the program twice timing it both times, and answer the following:

   . *How many **self seconds** did `insertionSort()` take?*
   a. *How many **self seconds** did `mergeSort()` take?*

5. **Human vs. Compiler Optimization (10 Points):**

   Which is faster:

   - A bad algorithm and data-structure optimized with -O2
   - A good algorithm and data-structure optimized with -O0

6. Parts of an executable (Points 20):

Please find the following inside of `assign1-0` by using `objdump`.

- If it *can* be found then *both*
  a. Give the `objdump` command, and
  b. Show the `objdump` result
- If it *cannot* be found then tell why not. Where in the memory of the runtime process is it?

Look for:

b. The string constant in `main()`
c. Global integer `numNumbers` in `main.c`
d. The code for `freeList()`
e. The pointer argument `nodePtr` in `freeList()`

Question	Command	Result
(A)	_____	_____
(B)	_____	_____
(C)	_____	

		_____
(D)	_____	_____

7. Compiler optimizations (Points 10):

Look at the assembly code of `assign1-0` and `assign1-2`. *Find* and *show* at least **2** optimizations that the compiler did in either `assign1-2` or `assign1-0`.