Assignment 2
Sorting

Prof. Darrell Long
Department of Computer Engineering

Spring 2017

Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.

—Donald Knuth, Vol. III, *Sorting and Searching*

1 Introduction

Putting items into a sorted order is one of the most common tasks in Computer Science, and so as a result there are myriad library routines that will do this task for you—but that does not absolve you of the obligation of understanding how it is done, and in fact it behooves you to understand the various algorithms in order to make wise choices.

The best that can be accomplished (the *lower bound*) for sorting using *comparisons* is $\Omega(n \log n)$ where n is the number is elements to be sorted. If the universe of elements to be sorted is limited (small), then we can do better using a *Count Sort*, where is $O(n)$, where we count the number of occurrences of each element in an array, or a *Radix Sort*, which is also $O(n)$ with a constant proportional to the maximum number of digits in the numbers being sorted.

What is this O and Ω stuff? It's how we talk about the execution time (or space) of programs. We will discuss it in class, and you will see it again in CMPS 101.

1.1 min-Sort

Perhaps the simplest sorting method is to look for the smallest element (the minimum) and move it to the top. We could do this by making a copy of that element, and then shifting everything down by one, and then placing the copy in the top slot. But that seems silly, why move all of those elements? Let's just *exchange* (swap) the top element with the smallest element.

At this point what do we know? We know that the smallest element is at the top. Since that is true and it will not change (we call this an *invariant*), we can forget about it and move on. Let's consider the second element: Why not just do what we did the first time? If we do that, then what do we know? We now know that the top (first) element is the smallest, and the second element is the second smallest (or the same, if there are duplicates). We can repeat this for each element in the array, in succession, up to but not including the last element. By the method of *induction*, we can show that the array is sorted.

Why do we not need to concern ourselves with the last element? The answer is that if it were not the smaller element when we were at the last step, then it was exchanged with the penultimate element, and thus must necessarily be the largest (and consequently, last) element in the array.

To get you started, here is the code for `minSort`. Notice that it is composed of two functions: `minIndex` which finds the *location* of the smallest element, and `minSort` which actually performs the sorting.

```c
// minIndex: find the index of the least element.

uint32_t minIndex(uint32_t a[], uint32_t first, uint32_t last)
{
    uint32_t smallest = first; // Assume the first is the least
    for (uint32_t i = first; i < last; i += 1)
    {
        smallest = a[i] < a[smallest] ? i : smallest;
    }
    return smallest;
}

// minSort: sort by repeatedly finding the least element.

void minSort(uint32_t a[], uint32_t length)
{
    for (uint32_t i = 0; i < length - 1; i += 1)
    {
        uint32_t smallest = minIndex(a, i, length);
        if (smallest != i) // It's silly to swap with yourself!
        {
            SWAP(a[smallest], a[i]);
        }
    }
    return;
}
```

minSort (in **C**)

What is the time complexity of `minSort`? The key is to look at the `for` loop, and we see that the loop is executed `length` times. This would seem to indicate that the sort is $O(n)$, but we need to look further. In the `for` loop we see that a call is made to `minIndex`, and when we look there we find yet another `for` loop. This loop is executed `(last - first)` times, which is based on `length`. So we call `minIndex` approximately `length` times, each time requiring approximately `length` operations, thus the time complexity of `minSort` is $O(n^2)$.

It is often useful to define a macro when a task is done repetitively, and when we also may want to hide the details. One could write a function (an in-line function would be preferred), but you will recall that functions in **C** always pass their arguments *by value* which is inconvenient for the *swap* operation. A clever person might take advantage of the macro to do some instrumentation.

```c
# ifdef _INSTRUMENTED
# define SWAP(x,y) { uint32_t t = x; x = y; y = x; ; moveCount += 3; }
# else
# define SWAP(x,y) { uint32_t t = x; x = y; y = x; ; }
# endif
```

SWAP macro

1.2 Bubble Sort

Bubble Sort works by examining adjacent pairs of items. If the second item is smaller than the first, exchange them. If you can pass over the entire array and no pairs are out of order, then the array is sorted. You will have noticed that the largest element falls in a single pass to the bottom of the array. Since it is in fact the largest, we do not need to consider it again, and so the next pass must only consider $n-1$ pairs of items.

What then is the expected time complexity of this algorithm? The first pass requires n pairs to be examined; the second pass, $n-1$ pairs; the third pass $n-2$ pairs, and so forth. When Carl Friedrich Gauss was a child of 7 (in 1784), he is reported to have amazed his elementary school teacher being a pest he was given the task of summing the integers from 1 to 100. The precocious little Gauss produced the correct answer immediately, having quickly observed that the sum was actually 50 pairs of numbers, with each pair summing to 101, total 5,050. We can then easily see that:

$$n+(n-1)+(n-2)+\ldots+1=\frac{n(n+1)}{2},$$

and so the *worst case* time complexity) is $O(n^2)$ (it may be much better, if for example, the list is already sorted).

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure
```

bubbleSort (pseudocode)

1.3 Insertion Sort

Insertion Sort is another in-place sort, it functions by taking an item and then inserting it into its correct position in the array. It consumes one input element each repetition, and growing the sorted portion of the list. Each iteration, insertion sort removes one element from the input unsorted portion and finds its location in the sorted portion of the list.

```
procedure insertionSort( A : list of sortable items )
 for i = 1 to length(A)
    tmp = A[i]
    j = i - 1
    while j >= 0 and A[j] > tmp
        A[j + 1] = A[j]
        j = j - 1
    end while
    A[j + 1] = tmp
 end for
end procedure
```

insertionSort (pseudocode)

What is the expected time complexity of Insertion sort? We look and observe that there two two nested loops, each operating on approximately the length of the array (it grows shorter at each iteration, but as we learned from young Gauss, $\sum_{i=1}^{n}=\frac{n(n+1)}{2}$. Nested loops mean that we *multiply* the execution times, and so it is also $O(n^2)$.

1.4 QuickSort (recursive)

One of the most important skills that you will need to develop is the ability to read and understand, though perhaps not program in, languages with which you are unfamiliar. Below, you will find the code for QuickSort written in Python. It would be a questionable choice for you to simply try to translate this into **C**, but there are some interesting elements to it.

First, you see that the code partitions the array into three parts: (i) a part that is *lesser* in value than the pivot element; (ii) *equal* in value to the pivot point, and (iii) *greater* in value than the pivot point. The choice of pivot element is arbitrary.

Second, you will see that we first recursively call `quickSort` first on the left partition, and then on the right partition. We then join these together to form a sorted array.

Third, you may notice that aside from the arbitrarily chosen pivot element, there is no array indexing. One could, in principle, use this algorithm to sort a *linked list*.

You will want to write a helper function called `partition` that will divide an array into three parts, as described earlier. You should do this *in place*, in other words you do not need to create three arrays. Instead, you will *swap* elements so that those less than or equal to the pivot element are on the left, while those greater than it are on the right. Why was this acceptable in Python? Python is a very different language that has lists (which is can treat as arrays) as a native data type. Different language enable different choices, and writing this in Python leads to the (surprising to some) insight that QuickSort can be implemented on linked lists.

```python
def quickSort(a):
    if len(a) > 0:
        pivot = a[0]
        left  = []
        mid   = []
        right = []
        for x in a:
            if x == pivot:
                mid.append(x)
            elif x < pivot:
                left.append(x)
            else:
                right.append(x)
        return quickSort(left) + mid + quickSort(right)
    else:
        return []
```

quickSort.py

1.5 Merge Sort

Merge Sort is a different type of sort in that it works through the array sequentially. It is well-suited for the case when you *do not have* random access, such as when working with magnetic tapes, or linked lists.

There are two main varieties of Merge Sort: *binary merge sort* (which may be recursive) and *natural merge sort*. The recursive binary merge sort was originally discovered by John von Neumann, one of the great polymaths of the previous century.

A single item (or, if you prefer to start your induction with zero, and empty list) is by definition sorted. Now, consider the case where we have two unsorted lists: either (i) the item at the front of list 1 is the smallest, or (ii) the item at the front of list 2 is the smallest, but in either case *greater than or equal to* the item at the end of list 3. Move that item to the end of the the (initially empty) list 3. Repeat this process until both of list 1 or list 2 have elements at their head less than the element at the end of list 3 or are empty. What do we know at this point? We know that

(i) the third list contains a *sorted subsequence*, and (ii) that sequence is equal in length to the sum of the sorted subsequences that we pulled from lists 1 and 2. This sorted subsequence is called a *run*. We repeat this process until we have exhausted either list 1 or list 2, at which point we append the remaining list to list 3.

We now take list 3 and copy elements from it as long the elements continue to increase to list 1, when the first decreasing element occurs we switch to copying the run to list 2. We repeat this, switching between lists 1 and 2, until list 3 is depleted.

We return to merging elements from lists 1 and 2 to increasing length runs on list 3. We do this until there is a single run, at which point the list is sorted.

It is much easier to think about this if you do it *recursively*. Take the list, split it into two lists (`left` and `right`). Call `mergeSort(left)` and `mergeSort(right)` and merge the results. For the base case, the recursion will encounter a list of a single element, which by definition is sorted.

What is the execution time of this algorithm? Let's assume that we have runs of length 1 on tapes 1 and 2. We merge these onto list 3 and we now have (at worst) runs of length 2. We processed n elements during this pass. We now need to ask, How many times can we double starting with 1 until the value we get is greater than n? The answer is $\log n$. The worst case execution time of our algorithm is $O(n \log n)$.

```python
def mergeSort(items):
        if len(items) > 1:
        middle     = len(items) / 2  # Split at the mid-point
        leftList   = items[0:middle] # Left half
        rightList = items[middle:]  # Right half

                mergeSort(leftList)  # Sort the leftList list
                mergeSort(rightList) # Sort the RightList list

        # Merge the sorted lists
        l, r = 0, 0
                for i in range(len(items)):
            # Both lists hold elements
            if l < len(leftList) and r < len(rightList):
                # The left is smaller
                if leftList[l] < rightList[r]:
                    items[i] = leftList[l]
                    l += 1
                # The right is smaller
                else:
                    items[i] = rightList[r]
                    r += 1
            # Only the left has any elements
            elif l < len(leftList):
                items[i] = leftList[l]
                l += 1
            # Only the right has any elements
            else:
                items[i] = rightList[r]
                r += 1
```

mergeSort.py

2 Your Task

You task is to:

- Implement a testing harness for sorting algorithms. You will do this using `getopt`:

```
GETOPT(3)                    Linux Programmer's Manual
      GETOPT(3)



NAME
       getopt, getopt_long, getopt_long_only, optarg, optind,
   opterr, optopt -
       Parse command-line options

SYNOPSIS
       #include <unistd.h>

       int getopt(int argc, char * const argv[],
                  const char *optstring);

       extern char *optarg;
       extern int optind, opterr, optopt;

       #include <getopt.h>

       int getopt_long(int argc, char * const argv[],
                  const char *optstring,
                  const struct option *longopts, int *longindex);
```

getopt

- Implement *five* specified sorting algorithms.
- Gather statistics about their performance.

3 Specifics

You must use `getopt` to parse the command line arguments. To get you started, here is a hint.

```
    while ((c = getopt(argc, argv, "AmbiqMp:r:n:")) != -1)
```

- `-A` means employ all sorting algorithms.
- `-m` means enable `minSort`.
- `-b` means enable `bubbleSort`.
- `-i` means enable `insertionSort`.
- `-q` means enable `quickSort`.

- `-M` means enable `mergeSort`.
- `-p n` means print the first `n` elements of the array. The *default* value is `100`.
- `-r s` means set the random seed to `s`. The *default* is 8222022.
- `-n c` means set the array size to `c`. The *default* value is `100`.

It is important to read this carefully. None of these options is exclusive of any other (you may specify any number of them, including *zero*).

- Your random numbers should be *24 bits*, no larger ($2^{24} - 1 = 16777215$).
- You must use `rand()` and `srand()`, not because they are good (they are not), but because they are what is specified by the **C**99 standard.
- Your program *must* be able to sort any number of random integers *up to the memory limit of the computer*. That means that you will need to dynamically allocate the array using `calloc()`.
- Your program should have no *memory leaks*.

A large part of this assignment is understanding and comparing the performance of various sorting algorithms. Consequently, you *must* collect some simple statistics on each algorithm. In particular,

- The *size* of the array,
- The number of *moves* required (each time you transfer an element in the array, that counts), and
- The number of *comparisons* required (comparisons *only* count for elements, not for logic).

```
Valhalla:sorting darrell$ make
gcc -Wall -Werror -Wextra -pedantic -O3 -DMASK=0x00ffffff -c sorting.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c bv.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c bubblesort.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c minsort.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c insertionsort.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c quicksort.c
gcc -Wall -Werror -Wextra -pedantic -O3 -c mergesort.c
gcc -o sorting sorting.o bv.o bubblesort.o minsort.o insertionsort.o
quicksort.o mergesort.o
```

```
Valhalla:sorting darrell$ ./sorting
Valhalla:sorting darrell$
```

```
Valhalla:sorting darrell$ ./sorting -n 10 -r 1234 -i
Insertion Sort
10 elements
34 moves
25 compares
     667041    694558   3789884   3962622   6238630  10272683  10916661
   11017812  11838633  13274474
```

```
Valhalla:sorting darrell$ ./sorting -b -q -n 100000 -p 21
Bubble Sort
100000 elements
7489129170 moves
2496376390 compares
       176        685        805       1174       1217       1618       1643
      1861       1897       2192       2381       2437       2465       3023
      3058       3225       3285       3348       3425       3909       3928

Quick Sort
100000 elements
3539469 moves
1046295 compares
       176        685        805       1174       1217       1618       1643
      1861       1897       2192       2381       2437       2465       3023
      3058       3225       3285       3348       3425       3909       3928
```

```
Valhalla:sorting darrell$ ./sorting -A
Min Sort
100 elements
288 moves
5049 compares
     47320     272862     813325     931036    1300149    1451478    1482116
   1599339    1886666    1926530    1999678    2177992    2338129    2381540
   2504306    2752788    2996335    2998169    3171889    3590861    3829897
   3967508    4022167    4231676    4422014    4702258    4742819    4841493
   4915648    5160950    5607401    5792480    5812062    5967954    5968964
   6053032    6195346    6301066    6375315    6389795    6781764    6797856
   7260963    7261776    7443963    7614058    7685040    7685284    7739256
   7989314    8075396    8085360    8249909    8352629    8737396    9017069
   9214736    9256511    9320602    9414691    9489446    9526978    9657331
  10177720   10578757   10623131   10856398   10978990   11215204   11268563
  11339313   11464349   11556747   11680696   11697687   11774380   11989481
  12291309   12547422   12617960   12814415   12844019   12899314   12912456
  13059818   13226807   13494413   13782527   14314674   14404909   14506627
  14956880   15232419   15509585   15563419   15846433   15950040   16608657
  16703976   16712977
Bubble Sort
100 elements
7074 moves
2358 compares
     47320     272862     813325     931036    1300149    1451478    1482116
   1599339    1886666    1926530    1999678    2177992    2338129    2381540
   2504306    2752788    2996335    2998169    3171889    3590861    3829897
   3967508    4022167    4231676    4422014    4702258    4742819    4841493
   4915648    5160950    5607401    5792480    5812062    5967954    5968964
```

```
   6053032   6195346   6301066   6375315   6389795   6781764   6797856
   7260963   7261776   7443963   7614058   7685040   7685284   7739256
   7989314   8075396   8085360   8249909   8352629   8737396   9017069
   9214736   9256511   9320602   9414691   9489446   9526978   9657331
  10177720  10578757  10623131  10856398  10978990  11215204  11268563
  11339313  11464349  11556747  11680696  11697687  11774380  11989481
  12291309  12547422  12617960  12814415  12844019  12899314  12912456
  13059818  13226807  13494413  13782527  14314674  14404909  14506627
  14956880  15232419  15509585  15563419  15846433  15950040  16608657
  16703976  16712977
Insertion Sort
100 elements
2457 moves
2358 compares
     47320    272862    813325    931036   1300149   1451478   1482116
   1599339   1886666   1926530   1999678   2177992   2338129   2381540
   2504306   2752788   2996335   2998169   3171889   3590861   3829897
   3967508   4022167   4231676   4422014   4702258   4742819   4841493
   4915648   5160950   5607401   5792480   5812062   5967954   5968964
   6053032   6195346   6301066   6375315   6389795   6781764   6797856
   7260963   7261776   7443963   7614058   7685040   7685284   7739256
   7989314   8075396   8085360   8249909   8352629   8737396   9017069
   9214736   9256511   9320602   9414691   9489446   9526978   9657331
  10177720  10578757  10623131  10856398  10978990  11215204  11268563
  11339313  11464349  11556747  11680696  11697687  11774380  11989481
  12291309  12547422  12617960  12814415  12844019  12899314  12912456
  13059818  13226807  13494413  13782527  14314674  14404909  14506627
  14956880  15232419  15509585  15563419  15846433  15950040  16608657
  16703976  16712977
Quick Sort
100 elements
1542 moves
384 compares
     47320    272862    813325    931036   1300149   1451478   1482116
   1599339   1886666   1926530   1999678   2177992   2338129   2381540
   2504306   2752788   2996335   2998169   3171889   3590861   3829897
   3967508   4022167   4231676   4422014   4702258   4742819   4841493
   4915648   5160950   5607401   5792480   5812062   5967954   5968964
   6053032   6195346   6301066   6375315   6389795   6781764   6797856
   7260963   7261776   7443963   7614058   7685040   7685284   7739256
   7989314   8075396   8085360   8249909   8352629   8737396   9017069
   9214736   9256511   9320602   9414691   9489446   9526978   9657331
  10177720  10578757  10623131  10856398  10978990  11215204  11268563
  11339313  11464349  11556747  11680696  11697687  11774380  11989481
  12291309  12547422  12617960  12814415  12844019  12899314  12912456
  13059818  13226807  13494413  13782527  14314674  14404909  14506627
  14956880  15232419  15509585  15563419  15846433  15950040  16608657
  16703976  16712977
Merge Sort
100 elements
```

```
1344 moves
316 compares
     47320    272862    813325    931036   1300149   1451478   1482116
   1599339   1886666   1926530   1999678   2177992   2338129   2381540
   2504306   2752788   2996335   2998169   3171889   3590861   3829897
   3967508   4022167   4231676   4422014   4702258   4742819   4841493
   4915648   5160950   5607401   5792480   5812062   5967954   5968964
   6053032   6195346   6301066   6375315   6389795   6781764   6797856
   7260963   7261776   7443963   7614058   7685040   7685284   7739256
   7989314   8075396   8085360   8249909   8352629   8737396   9017069
   9214736   9256511   9320602   9414691   9489446   9526978   9657331
  10177720  10578757  10623131  10856398  10978990  11215204  11268563
  11339313  11464349  11556747  11680696  11697687  11774380  11989481
  12291309  12547422  12617960  12814415  12844019  12899314  12912456
  13059818  13226807  13494413  13782527  14314674  14404909  14506627
  14956880  15232419  15509585  15563419  15846433  15950040  16608657
  16703976  16712977
```

Submission

You *must* turn in your assignment in the following manner:

1. Have file called `Makefile` that when the grader types `make` will compile your program. At this point you will have learned about `make` and can create your own `Makefile`.
   - `CFLAGS=-Wall -Wextra -Werror -pedantic` must be included.
   - `CC=gcc` must be specified.
   - `make clean` must remove all files that are compiler generated.
   - `make` should build your program, as should `make all`.
2. You program *must* have the source and header files:
   - `minsort.h` specifies the interface to `minSort()`.
   - `minsort.c` implements `minSort()`.
   - Each sorting method will have its own pair of header file and source file.
   - `sorting.c` contains `main()` and *may* contain the other functions necessary to complete the assignment.
3. You may have other source and header files, but *do not try to be overly clever.*
4. A plain text file called `README` that describes how your program works.
5. The executable file produced by the compiler *must be called* `sorting`.
6. These files must be in the directory `assignment2`.
7. You must `commit` and `push` the directory and its contents using `git`.

Points will be assigned according to the difficulty of the sort involved.

- 5% – min-Sort
- 10% – Bubble Sort

- 15% – Insertion Sort
- 20% – Quick Sort (recursive)
- 30% – Merge Sort

A sort is not considered to be implemented if it does not sort *correctly every time.*

Additional criteria are:

- 10% – Code quality and correctness.
- 10% – Completeness: which includes things like the `Makefile`.