Assignment 5
Sorting

Prof. Max Dunne
CSE 13S

1 Introduction

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

The best execution time that can be accomplished, also referred to as 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 small, then we can do better using a *Count Sort* or a *Radix Sort* both of which have a time complexity of $O(n)$. The idea of *Count Sort* is to count the number of occurrences of each element in an array. For *Radix Sort*, a digit by digit sort is done by starting from the least significant digit to the most significant digit. It may also use *Count Sort* as a subroutine.

What is this O and Ω stuff? It's how we talk about the execution time (or space used) by a program. We will discuss it in class, and you will see it again in your Data Structures and Algorithms class.

The sorting algorithms that you are expected to implement are Bubble Sort, Shell Sort, Quick Sort and Binary Insertion Sort. The purpose of this assignment is to get you fully familiarized with each sorting algorithm. They are well-known sorts. You can use the Python pseudocode provided to you as guidelines. Do not get the code for the sorts from the Internet or you will be referred to for cheating.

1.1 Bubble Sort

Bubble sort works by examining adjacent pairs of items. If the second item is smaller than the first, swap them. As a result, the largest element falls to the bottom of the array in a single pass. Since it is in fact the largest, we do not need to consider it again. So in the next pass, we only need to consider $n-1$ pairs of items. The first pass requires n pairs to be examined; the second pass, $n-1$ pairs; the third pass $n-2$ pairs, and so forth. If you can pass over the entire array and no pairs are out of order, then the array is sorted.

Pre-lab Part 1

1. How many rounds of swapping do you think you will need to sort the numbers $8, 22, 7, 9, 31, 5, 13$ in ascending order using Bubble Sort?
2. How many comparisons can we expect to see in the worse case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.

In 1784, when Carl Friedrich Gauss was only 7 years old, he was reported to have amazed his elementary school teacher by how quickly he summed up the integers from 1 to 100. The precocious little Gauss produced the correct answer immediately after he quickly observed that the sum was actually 50 pairs of numbers, with each pair summing to 101 totaling to 5,050. We can then see that:

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

So the *worst case* time complexity is $O(n^2)$. However, it could be much better if the list is already sorted. If you haven't seen the inductive proof for this yet, you will in the applied discrete math class.

```
def Bubble_Sort(arr):
  for i in range(len(arr) - 1):
    j = len(arr) - 1
    while j > i:
      if arr[j] < arr[j - 1]:
        arr[j], arr[j - 1] = arr[j - 1], arr[j]
      j -= 1
  return
```

Bubble Sort (pseudocode)

1.2 Shell Sort

Shell Sort is a variation of insertion sort, which sorts pairs of elements which are far apart from each other. The interval (or *gap*) between the compared items being sorted is continuously reduced. Shell Sort starts with distant elements and moves out-of-place elements into position faster than a simple nearest neighbor exchange. In the following code, an array of intervals is created by using `gap(n)` for an unsorted list of n elements. For example, for $n = 20$ unsorted elements, the set of gaps is $\{9, 4, 1\}$.

What is the expected time complexity of Shell Sort? All this depends upon the gap sequence. The number of elements in the gap sequence and their respective size scales with the number of elements n being sorted. The first loop is executed `len(s)-step` times and that number decreases as the gap size decreases.

The following is the pseudocode for Shell Sort. Given the length of array n, the function $gap(n)$ produces an array of gaps. The rules is that if $n \leq 2$, $n = 1$, else $n = 5 * n//11$, in which // dumps the digits after the decimal. The array will be ranked from large to small. In the $Shell\_Sort(n)$, for each step in the array of gaps, it compares all the pairs that are away from each other by $step$ in index and switches the elements in the pair if they are not sorted.

```
def gap(n):
  while n > 1:
    n = 1 if n <= 2 else 5 * n // 11
    yield n
```

gap (pseudocode)

```
def Shell_Sort(arr):
  for step in gap(len(arr)):
    for i in range(step, len(arr)):
```

```
    for j in range(i, step - 1, -step):
      if arr[j] < arr[j - step]:
        arr[j], arr[j - step] = arr[j - step], arr[j]
  return
```

Shell Sort (pseudocode)

Pre-lab Part 2

1. The worst time complexity for Shell sort depends on the size of the gap. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap size? Cite any sources you used.
2. How would you improve the runtime of this sort without changing the gapp size?

1.3 Quicksort

Quicksort is a divide-and-conquer algorithm. It partitions arrays into two subarrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left subarray, and elements in the array that are greater than or equal to the pivot go to the right subarray. Note that Quicksort is an *in-place* algorithm, meaning it doesn't allocate additional memory for subarrays to hold partitioned elements. Instead, Quicksort utilizes a subroutine called `Partition()` that places elements less than the pivot into the left side of the array and elements greater than or equal to the pivot into the right side and returns the index that indicates the division between the partitioned parts of the array. Quicksort is then run recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element.

```
def Partition(arr, left, right):
  pivot = arr[left]
  lo = left + 1
  hi = right

  while True:
    while lo <= hi and arr[hi] >= pivot:
      hi -= 1

    while lo <= hi and arr[lo] <= pivot:
      lo += 1

    if lo <= hi:
      arr[lo], arr[hi] = arr[hi], arr[lo]
    else:
      break

  arr[left], arr[hi] = arr[hi], arr[left]
  return hi
```



```

def Quick_Sort(arr, left, right):
  if left < right:
    index = Partition(arr, left, right)
    Quick_Sort(arr, left, index - 1)
    Quick_Sort(arr, index + 1, right)
  return
```

Quicksort (pseudocode)

Pre-lab Part 3

1. Quicksort, with a worse case time complexity of $O(n^2)$, doesn't seem to live up to its name. Investigate and explain why Quicksort isn't doomed by its worst case scenario. Make sure to cite any sources you use.

1.4 Binary Insertion Sort

Binary Insertion Sort is a special type of insertion sort which uses the binary search algorithm to find the correct position of an inserted element in an array. Insertion sort works by finding the correct position of the element in the array and then inserting it into its correct position. Searching for an element using binary search is much like searching for a book on a shelf that is sorted alphabetically. First, identify the book sitting approximately at the midpoint between either end of the shelf. If it's the book you're looking for, then great! If the book you're looking for has a name that precedes the current book alphabetically, you only need to consider the left half of the shelf. Else, you only need to consider the right half of the shelf. Thus, it's clear that we are *halving* the search space each time we do a comparison, hence the name, binary search. Binary Insertion Sort uses binary search in order to determine where each element should go, reducing the number of comparisons between array elements we would ordinarily need for Insertion sort. For each element in the array, simply run a binary search through the elements to the left of the current element in order to find the index in which it should go.

```
def Binary_Insertion_Sort(arr):
  for i in range(1, len(arr)):
    value = arr[i]
    left = 0
    right = i

    while left < right:
      mid = left + ((right - left) // 2)

      if value >= arr[mid]:
        left = mid + 1
      else:
        right = mid

    for j in range(i, left, -1):
```

```
        arr[j - 1], arr[j] = arr[j], arr[j - 1]

    return
```

Binary Insertion Sort (pseudocode)

Each round in insertion sort involves picking a single element from the input array and finding a location in the sorted array where it can be placed. In the Binary Insertion Sort algorithm, this location is found using the binary search algorithm.

Pre-lab Part 4

1. Can you figure out what effect the binary search algorithm has on the complexity when it is combined with the insertion sort algorithm?

2 Your Task

For this assignment you have 3 tasks:

- **Task 1:** Implement a testing harness for sorting algorithms. You will do this using `getopt`.
- **Task 2:** Implement the four sorting algorithms Bubble Sort, Shell Sort, Quicksort and Binary Insertion Sort, whose pseudocode have been provided in the above section.
- **Task 3:** Gather statistics about each sort and its performance such as the *size* of the array, the number of moves required, and the number of *comparisons* required (comparisons for *elements*, not for the logic).

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, "Absqip:r:n:")) != -1)
```

- `-A` means employ *all* sorting algorithms.
- `-b` means enable `Bubble Sort`.
- `-s` means enable `Shell Sort`.
- `-q` means enable `QuickSort`.
- `-i` means enable `Binary Insertion Sort`.
- `-p n` means print the first `n` elements of the array. However if the `-p n` flag is not specified, your program should print the first 100 elements. The *default* `n` value is `100`.
- `-r s` means set the random seed to `s`. The *default* `s` value is 8222022.

- `-n c` means set the array size to `c`. The *default* `c` value is `100`.

It is important to read this *carefully*. None of these options are *exclusive* of any other (you may specify any number of them, including *zero*). The most natural data structure for this problem is a *set*.

- Your random numbers should be *30 bits*, no larger ($2^{30} - 1 = 1\,073\,741\,823$). (*Hint*: bit masking will help you here.)
- You must use `rand()` and `srand()`.
- 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*. Make sure you free before exiting. Valgrind should build without any errors.
- Your program must pass `infer` cleanly. Fix or explain any complaints by `infer` in your `README`.
- The executable file produced by the compiler *must be called* `sorting`.
- Your algorithms *must* correctly sort. If it does not sort, then for that sort you receive a *zero*.

A large part of this assignment is understanding and comparing the performance of various sorting algorithms. You essentially conducting an experiment. 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).

Pre-lab Part 5

1. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

4 Deliverables

You will need to turn in:

1. Your program *must* have the followingsource and header files:
    - Each sorting method will have its own pair of header file and source file.
        - `bubble.h` specifies the interface to `bubble.c`.
        - `bubble.c` implements Bubble Sort.
        - `shell.h` specifies the interface to `shell.c`.
        - `shell.c` implements Shell Sort.

    - `quick.h` specifies the interface to `quick.c`.
    - `quick.c` implements Quicksort.
    - `binary.h` specifies the interface to `binary.c`.
    - `binary.c` implements Binary Insertion Sort.
  - `sorting.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.

  You will likely have other source and header files, but *do not try to be overly clever.*

2. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. Typing `make` must build your program and `./sorting` alone as well as flags must run your program.

  - `CFLAGS=-Wall -Wextra -Werror -Wpedantic -std=c99` must be included.
  - `CC=clang` must be specified.
  - `make clean` must remove all files that are compiler generated.
  - `make valgrind` must build your program to check for memory mismanagement errors.
  - `make infer` must build and run `infer` on your program, passing without errors. Again, any errors that you cannot fix should be documented in your `README`.
  - `make` should build your program, as should `make all`.
  - Your program executable *must* be named `sorting`.

3. `README.md`: This *must* be in *markdown*. This must describe how to use your program and `Makefile`.

4. `DESIGN.pdf`: This *must* be a PDF. The design document should contain answers to the pre-lab questions at the beginning and describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudo-code.

  You *must* push the `DESIGN.pdf` before you push *any* code.

5. `WRITEUP.pdf`: This document *must* be a PDF. The writeup must include the following:

  - Identify the respective time complexity for each sort and include what you have to say about the constant.
  - What you learned from the different sorting algorithms.
  - How you experimented with the sorts.

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

- 10% – Bubble sort
- 15% – Shell Sort
- 20% – Quick Sort
- 20% – Binary Insertion Sort

A sort is not considered to be implemented if it does not sort *correctly every time*. If it does not sort correctly then that sort receives a zero. Additional criteria are:

- 10% – Code quality: this includes passing `infer` and consistent style.
- 10% – Completeness: which includes things like the `Makefile`.
- 15% – Supporting Documents: This includes your `WRITEUP.pdf`, `DESIGN.pdf`, and `README.md`.

5 Submission

To submit your assignment, refer back to `assignment0` for the steps on how to submit your assignment through `git`. Remember: *add, commit,* and *push*!

Your assignment is turned in *only* after you have pushed *and* submitted the commit ID on Canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. “I forgot to push” is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

6 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 1 §1.10
  - Chapter 4 §4.10–4.11
  - Chapter 5 §5.1–5.3

7 Examples

```
Bubble Sort
300 elements, 65430 moves, 44030 compares
     4879690     8565726    10082911    18153700    20428990    22843242    23697734
    29553441    31041143    32837107    33192435    38052897    41357431    44478931
    48950417    54899008    58259291    59582969    60278728    63074888    67038132
    83652098    88074691    91368359    93000463   100143045   104568041   104802123
   107339740   109656373   111508243   119396281   119606591   122505356   122988398
   126846790   127291023   128482584   129421256   129711536   133174074   133525456
   136807261   143592767   148048941   150580622   152392365   157404040   162744176
   167486952   167915664   169460827   177653006   178959793   180536272   182130793
   185587419   192299791   193566584   199531040   203647439   203707059   208694149
   211543367   221470759   221799112   223806700   223917017   230126174   232250665
   238063897   239108879   239238521   244735802   245913732   246216715   248191987
   248592856   251882760   253688452   253836378   253878856   254307866   256878688
   266129371   268113532   269123502   272757118   274591444   276603325   292239113
   294173389   301761273   309590988   315670423   323333023   331064360   340194372
   342925915   344180216
```

./sorting -b -n 300

```
Binary Insertion Sort
1000 elements, 769875 moves, 8595 compares
     2416949     5156682     6072641     6939507     7432408     7684930     8936987
    10901063    11614769    11671338
Quick Sort
1000 elements, 7011 moves, 14334 compares
     2416949     5156682     6072641     6939507     7432408     7684930     8936987
    10901063    11614769    11671338
Shell Sort
1000 elements, 19971 moves, 772356 compares
     2416949     5156682     6072641     6939507     7432408     7684930     8936987
    10901063    11614769    11671338
Bubble Sort
```

```
1000 elements, 769875 moves, 499094 compares
     2416949      5156682      6072641      6939507      7432408      7684930      8936987
    10901063     11614769     11671338
```

./sorting -n 1000 -p 10 -r 1 -A