# Assignment 3
# Sorting: Putting your affairs in order

Prof. Darrell Long
CSE 13S – Fall 2021

Due: October 17th at 11:59 pm

## 1 Introduction

> *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*

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.

What is this O and $\Omega$ stuff? It's how we talk about the execution time (or space used) by a program. We will discuss it in lecture and in section, and you will see it again in your Data Structures and Algorithms class, now named CSE 101.

The sorting algorithms that you are expected to implement are Insertion Sort, Shell Sort, Heap Sort, and recursive Quicksort. The purpose of this assignment is to get you fully familiarized with each sorting algorithm and for you to get a *feel* for computational complexity. They are well-known sorts. You can use the Python pseudocode provided to you as guides. Do not get the code for the sorts from the Internet or you will be referred to for cheating. We will be running plagiarism checkers.

## 2 Insertion Sort

Insertion Sort is a sorting algorithm that considers elements one at a time, placing them in their correct, ordered position. It is so simple and so ancient that we do not know who invented it. Assume an array of size $n$. For each $k$ in increasing value from $1 \leq k \leq n$ (using 1-based indexing), Insertion Sort compares the $k$-th element with each of the preceding elements in descending order until its position is found.

Assume we're sorting an array $A$ in increasing order. We start from and check if $A[k]$ is in the correct order by comparing it the element $A[k-1]$. There are two possibilities at this point:

1. $A[k]$ **is in the right place.** This means that $A[k]$ is greater or equal to $A[k-1]$, and thus we can move onto sorting the next element.

2. $A[k]$ **is in the wrong place.** Thus, $A[k-1]$ is shifted up to $A[k]$, and the original value of $A[k]$ is further compared to $A[k-2]$. Intuitively, Insertion Sort simply shifts elements back until the element to place in sorted order is slotted in.

**Insertion Sort in Python**

```python
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

## 3 Shell Sort

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> —C.A.R. Hoare

Donald L. Shell (March 1, 1924–November 2, 2015) was an American computer scientist who designed the Shell sort sorting algorithm. He earned his Ph.D. in Mathematics from the University of Cincinnati in 1959, and published the Shell Sort algorithm in the Communications of the ACM in July that same year.

Shell Sort is a variation of Insertion Sort and first sorts pairs of elements which are far apart from each other. The distance between these pairs of elements is called a *gap*. Each iteration of Shell Sort decreases the gap until a gap of 1 is used. At this point, the elements are sorted. What is the expected time complexity of Shell Sort? It depends entirely upon the gap sequence.

Any gap sequence that ends with a gap of 1 will produce a sorted array. That said, a gap sequences with either too many or too few gaps should be avoided as they either slow down passes through the array, or incur additional overhead. Perhaps one of the best known gap sequences is the sequence proposed by Vaughan Pratt in 1971 that is $O(n \log^2 n)$. American computer scientist Donald Knuth later proposed his own gap sequence based off Pratt's in 1973 that is $O(n^{\frac{3}{2}})$. Robert Sedgewick eventually proposed his own gap sequence in 1982. His gap sequence is $O(n^{\frac{4}{3}})$.

The following is the pseudocode for Shell Sort using Knuth's gap sequence. The sequence is produced using a *generator* in **Python**. You are expected to implement exactly this gap sequence. Note that **C** *does not* have generators, so you will need to find a way around this issue. For each gap in the gap sequence, the sort compares all the pairs in array `A` that are gap indices away from each other, shifting back array elements that are in the wrong order.

**Shell Sort in Python**

```python
from math import log

def gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i - 1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
            temp = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

From the definition of the gap sequence generator, you should see that gaps are computed as $\lfloor \frac{3^k-1}{2} \rfloor$ and the largest $k$ is $\lfloor \frac{\log(2n+3)}{\log(3)} \rfloor$, where $n$ is the length of the array to sort. You be iterating over the gap sequence starting with the largest $k$ first, making your way down to a gap size of 1.

## 4 Heapsort

> *Increasingly, people seem to misinterpret complexity as sophistication, which is baffling – the incomprehensible should cause suspicion rather than admiration.*
>
> —Niklaus Wirth

Heapsort, along with the heap data structure, was invented in 1964 by J. W. J. Williams. The heap data structure is typically implemented as a specialized binary tree. There are two kinds of heaps: *max* heaps and *min* heaps. In a max heap, any parent node *must* have a value that is greater than or equal to the values of its children. For a min heap, any parent node *must* have a value that is less than or equal to the values of its children. The heap is typically represented as an *array*, in which for any index $k$, the index of its left child is $2k$ and the index of its right child is $2k+1$. It's easy to see then that the parent index of any index $k$ should be $\lfloor \frac{k}{2} \rfloor$.

Heapsort, as you may imagine, utilizes a heap to sort elements. Heapsort sorts its elements using two routines that 1) build a heap, 2) fix a heap.

1. **Building a heap.** The first routine is taking the array to sort and building a heap from it. This means ordering the array elements such that they obey the constraints of a max or min heap. For our purposes, the constructed heap will be a *max heap*. This means that the largest element, the root of the heap, is the first element of the array from which the heap is built.

2. **Fixing a heap.** The second routine is needed as we sort the array. The gist of Heapsort is that the largest array elements are repeatedly removed from the top of the heap and placed at the end of the sorted array, if the array is to be sorted in increasing order. After removing the largest element from the heap, the heap needs to be *fixed* so that it once again obeys the constraints of a heap.

In the following Python code for Heapsort, you will notice a lot of indices are shifted down by 1. Why is this? Recall how indices of children are computed. The formula of the left child of $k$ being $2k$ only works assuming 1-based indexing. We, in Computer Science, especially in **C**, use 0-based indexing. So, we will run the algorithm assuming 1-based indexing for the Heapsort algorithm itself, subtracting 1 on each array index access to account for 0-based indexing.

**Heap maintenance in Python**

```python
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

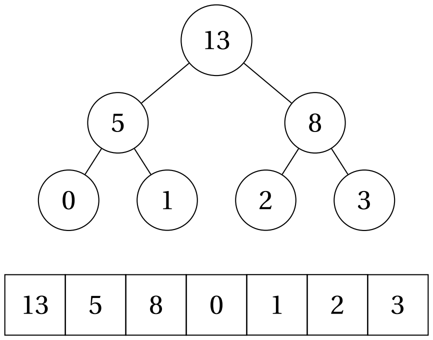


Figure 1: A max heap and its array representation.

**Heapsort in Python**

```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

## 5 Quicksort

> *If debugging is the process of removing software bugs, then programming must be the process of putting them in.*
>
> —Edsger Dijkstra

Quicksort (sometimes called partition-exchange sort) was developed by British computer scientist C.A.R. “Tony” Hoare in 1959 and published in 1961. It is perhaps the most commonly used algorithm for sorting (by competent programmers). When implemented well, it is the fastest known algorithm that sorts using *comparisons*. It is usually two or three times faster than its main competitors, Merge Sort and Heapsort. It does, though, have a worst case performance of $O(n^2)$ while its competitors are strictly $O(n \log n)$ in their worst case.

Quicksort is a divide-and-conquer algorithm. It partitions arrays into two sub-arrays 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 sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array.

Note that Quicksort is an *in-place* algorithm, meaning it doesn’t allocate additional memory for sub-arrays 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 applied recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element. Like with the Heapsort algorithm, the provided Quicksort pseudocode operates on 1-based indexing, subtracting one to account for 0-based indexing whenever array elements are accessed.

**Partition in Python**

```python
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

**Recursive Quicksort in Python**

```python
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

## 6 Your Task

> *Find out the reason that commands you to write; see whether it has spread its roots into the very depth of your heart; confess to yourself you would have to die if you were forbidden to write.*
>
> —Rainer Maria Rilke

Your task for this assignment is as follows:

1. Implement Insertion Sort, Shell Sort, Heapsort, and recursive Quicksort based on the provided Python pseudocode in **C**. The interface for these sorts will be given as the header files `insert.h`, `shell.h`, `heap.h`, and `quick.h`. You are not allowed to modify these files for any reason.

2. Implement a test harness for your implemented sorting algorithms. In your test harness, you will creating an array of pseudorandom elements and testing each of the sorts. Your test harness *must* be in the file `sorting.c`.

3. Gather statistics about each sort and its performance. The statistics you will gather are the *size* of the array, the number of *moves* required, and the number of *comparisons* required. Note: a comparison is counted only when two array elements are compared.

Your test harness must support any combination of the following command-line options:

- `-a` : Employs *all* sorting algorithms.
- `-e` : Enables Heap Sort.
- `-i` : Enables Insertion Sort.

- `-s` : Enables Shell Sort.
- `-q` : Enables Quicksort.
- `-r seed` : Set the random seed to `seed`. The *default* seed should be 13371453.
- `-n size` : Set the array size to `size`. The *default* size should be 100.
- `-p elements` : Print out `elements` number of elements from the array. The *default* number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.
- `-h` : Prints out program usage. See reference program for example of what to print.

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*.

## 7 Sets

For this assignment, you are required to use a *set* to track which command-line options are specified when your program is run. The code for sets is given in the resources repository in `set.h`. You may not modify this file. Make sure to take the time to go through and understand the code.

For manipulating the bits in a set, we use bit-wise operators. These operators, as the name suggests, will perform an operation on every bit in a number. The following are the six bit-wise operators specified in **C**:

| | | |
|---|---|---|
| & | bit-wise AND | Performs the AND operation on every bit of two numbers. |
| \| | bit-wise OR | Performs the OR operation on every bit of two numbers. |
| ~ | bit-wise NOT | Inverts all bits in the given number. |
| ^ | bit-wise XOR | Performs the exclusive-OR operation on every bit of two numbers. |
| << | left shift | Shifts bits in a number to the left by a specified number of bits. |
| >> | right shift | Shifts bits in a number to the right by a specified number of bits. |

Recall that the basic set operations are: *membership*, *union*, *intersection* and *negation*. The functions implementing these set operations are implemented for you. Using these functions, you will set (make the bit 1) or clear (make the bit 0) bits in the `Set` depending on the command-line options read by `getopt()`. You can then check the states of all the bits (the members) of the `Set` using a single `for` loop and execute the corresponding sort. Note: you most likely won't use all the functions, but you *must* use sets to track which command-line options are specified when running your program.

`Set empty_set(void)`

This function is used to return an empty set. In this context, an empty set would be a set in which all bits are equal to 0.

`bool member_set(uint32_t x, Set s)`

$$x \in s \iff x \text{is a member of set} s$$

This function returns a `bool` indicating the presence of the given value `x` in the set `s`. The bit-wise AND operator is used to determine set membership. The first operand for the AND operation is the set `s`. The second operand is the value obtained by left shifting 1 `x` number of times. If the result of the AND operation is a non-zero value, then `x` is a member of `s` and `true` is returned to indicate this. `false` is returned if the result of the AND operation is 0.

`Set insert_set(uint32_t x, Set s)`

This function inserts `x` into `s`. That is, it returns set `s` with the bit corresponding to `x` set to 1. Here, the bit is set using the bit-wise OR operator. The first operand for the OR operation is the set `s`. The second operand is value obtained by left shifting 1 by `x` number of bits.

`Set delete_set(uint32_t x, Set s)`

This function deletes (removes) `x` from `s`. That is, it returns set `s` with the bit corresponding to `x` cleared to 0. Here, the bit is cleared using the bit-wise AND operator. The first operand for the AND operation is the set `s`. The second operand is a negation of the number 1 left shifted to the same position that `x` would occupy in the set. This means that the bits of the second operand are all 1s except for the bit at `x`'s position. The function returns set `s` after removing `x`.

`Set union_set(Set s, Set t)`

$$s \cup t = \{x | x \in s \lor x \in t\}$$

The union of two sets is a collection of all elements in both sets. Here, to calculate the union of the two sets `s` and `t`, we need to use the OR operator. Only the bits corresponding to members that are equal to 1 in either `s` or `t` are in the new set returned by the function.

`Set intersect_set(Set s, Set t)`

$$s \cap t = \{x | x \in s \land x \in t\}$$

The intersection of two sets is a collection of elements that are common to both sets. Here, to calculate the intersection of the two sets `s` and `t`, we need to use the AND operator. Only the bits corresponding to members that are equal to 1 in both `s` and `t` are in the new set returned by the function.

`Set difference_set(Set s, Set t)`

The difference of two sets refers to the elements of set `s` which are not in set `t`. In other words, it refers to the members of set `s` that are unique to set `s`. The difference is calculated using the AND operator where the two operands are set `s` and the negation of set `t`. The function then returns the set of elements in `s` that are not in `t`.

This function can be used to find the complement of a given set as well, in which case the first operand would be the universal set $\mathbb{U}$ and the second operand would be the set you want to complement as shown below.

$$\overline{s} = \{x | x \notin s\} = \mathbb{U} - s$$

`Set complement_set(Set s)`

This function is used to return the complement of a given set. By complement we mean that all bits in the set are flipped using the NOT operator. Thus, the set that is returned contains all the elements of the universal set $\mathbb{U}$ that are not in `s` and contains none of the elements that are present in `s`.

## 8 Testing

- You will test each of the sorts specified by command-line option by sorting an array of pseudorandom numbers generated with `random()`. Each of your sorts should sort the *same* pseudorandom array. Hint: make use of `srandom()`.
- The pseudorandom numbers generated by `random()` should be *bit-masked* to fit in *30* bits. Hint: use bit-wise AND.
- Your test harness *must* be able to test your sorts with array sizes *up to the memory limit of the computer*. That means that you will need to dynamically allocate the array.
- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options.
- Your algorithms *must* correctly sort. Any algorithm that does not sort correctly will receive a *zero*.

A large part of this assignment is understanding and comparing the performance of various sorting algorithms. You essentially conducting an experiment. As stated in §6, you *must* collect the following statistics on each algorithm:

- 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).

## 9 Output

> *Creative output, you know, is just pain. I'm going to be cliche for a minute and say that great art comes from pain.*
>
> —Kanye West

The output your test harness produces *must* be formatted like in the following examples:

```
$ ./sorting -q -n 1000 -p 0
Quick Sort, 1000 elements, 18642 moves, 10531 compares
$ ./sorting -ie -n 15 -p 0
Heap Sort, 15 elements, 144 moves, 70 compares
Insertion Sort, 15 elements, 82 moves, 65 compares
$ ./sorting -a -n 15
Heap Sort, 15 elements, 144 moves, 70 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Shell Sort, 15 elements, 90 moves, 59 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Insertion Sort, 15 elements, 82 moves, 65 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Quick Sort, 15 elements, 135 moves, 51 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
```

For each sort that was specified, print its name, the statistics for the run, then the specified number of array elements to print. The array elements should be printed out in a table with 5 columns. Each array element should be printed with a width of 13. You should make use of the following `printf()` statement:

```c
printf("%13" PRIu32); // Include <inttypes.h> for PRIu32.
```

## 10 Statistics

> *There are three types of lies—lies, damn lies, and statistics.*
>
> —Benjamin Disraeli

To facilitate the gathering of statistics, you will be given, *and must use*, a small statistics module. The module itself revolves around the following struct:

```c
typedef struct {
  uint64_t moves;
  uint64_t comparisons;
} Stats;
```

The module also includes functions to *compare*, *swap*, and *move* elements.

`int cmp(Stats *stats, uint32_t x, uint32_t y)`

Compares `x` and `y` and increments the `comparisons` field in `stats`. Returns `-1` if `x` is less than `y`, `0` if `x` is equal to `y`, and `1` if `x` is greater than `y`.

`uint32_t move(Stats *stats, uint32_t x)`

"Moves" `x` by incrementing the `moves` field in `stats` and returning `x`. This is intended for use in Insertion Sort and Shell Sort, where array elements aren't swapped, but instead moved and stored in a temporary variable.

`void swap(Stats *stats, uint32_t *x, uint32_t *y)`

Swaps the elements pointed to by pointers `x` and `y`, incrementing the `moves` field in `stats` by 3 to reflect a swap using a temporary variable.

`void reset(Stats *stats)`

Resets `stats`, setting the `moves` field and `comparisons` field to 0. It is possible that you don't end up using this specific function, depending on your usage of the `Stats struct`.

## 11 Deliverables

> *Dr. Long, put down the Rilke and step away from the computer.*
>
> —Michael Olson

You will need to turn in the following source code and header files:

1. Your program *must* have the following source and header files:
    - `insert.c` implements Insertion Sort.
    - `insert.h` specifies the interface to `insert.c`.
    - `heap.c` implements Heap Sort.
    - `heap.h` specifies the interface to `heap.c`.
    - `quick.c` implements recursive Quicksort.
    - `quick.h` specifies the interface to `quick.c`.
    - `set.h` implements and specifies the interface for the set ADT.
    - `stats.c` implements the statistics module.
    - `stats.h` specifies the interface to the statistics module.
    - `shell.c` implements Shell Sort.
    - `shell.h` specifies the interface to `shell.c`.
    - `sorting.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.

You can have other source and header files, but *do not try to be overly clever*. The header files for each of the sorts are provided to you and may not be modified. Each sort function takes a pointer to a `Stats struct`, the array of `uint32_ts` to sort as the first parameter, and the length of the array as the second parameter. You will also need to turn in the following:

1. `Makefile`:

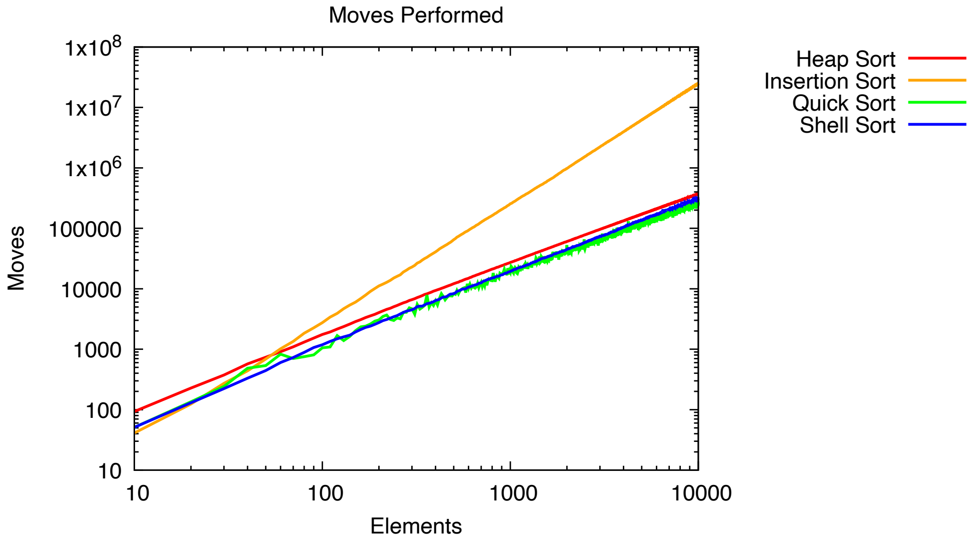
    - `CC = clang` must be specified.
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
    - `make` must build the `sorting` executable, as should `make all` and `make sorting`.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all your source code, including the header files.

2. `README.md`: This must use proper Markdown syntax. It must describe how to use your program and `Makefile`. It should also list and explain any command-line options that your program accepts. Any false positives reported by `scan-build` should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such 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 pseudocode.

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

    - What you learned from the different sorting algorithms. Under what conditions do sorts perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings?
    - Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. Use a UNIX tool — not some website — to produce these graphs. `gnuplot` is recommended. Attend section for examples of using `gnuplot` and other UNIX tools.
    - Analysis of the graphs you produce. Here is an example graph produced by `gnuplot` for reference (axes are log-scaled):



You should look carefully at any graphs that you produce. Are all of the lines smooth? For example, in this graph there are some features of interest. What could be causing features that appear in your own graphs?

## 12 Submission

Refer back assignment 0 for the instructions on how to properly 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 you want graded on Canvas. “I forgot to push” and “I forgot to submit my commit ID” are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

## 13 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 1 §1.10
  - Chapter 3 §3.5
  - Chapter 4 §4.10–4.11
  - Chapter 5 §5.1–5.3 & 5.10
- *C in a Nutshell* by T. Crawford & P. Prinz.
  - Chapter 6 – Example 6.5
  - Chapter 7 – Recursive Functions
  - Chapter 8 – Arrays as Arguments of Functions
  - Chapter 9 – Pointers to Arrays

*Paggimmnorr in C is eikl ggiinv a ekmnoy a aachinsw.*