Modeling Sorting Algorithms in Different Programming Languages

Hamza Rehioui

April 7, 2022

Abstract

As a part of my final project for the Introduction to Mathematical Modeling class with Dr. Van Nguyen, I decided to study how two sorting algorithms - bubble sort and merge sort - behave across three different programming languages. My choice of programming languages was based on having different paradigms: C as the imperative paradigm, Python as the object oriented paradigm, and finally, Racket for the functional paradigm. This would in turn give me an idea on how time complexities behave and how actual durations fare in front of expected ones.

Contents

1 Introduction **5**
- 1.1 Programming Languages . . . 5
  - 1.1.1 C Language . . . 5
  - 1.1.2 Python . . . 5
  - 1.1.3 Racket . . . 5
  - 1.1.4 R Language . . . 6
- 1.2 Algorithms . . . 6
  - 1.2.1 Bubble Sort . . . 6
  - 1.2.2 Merge Sort . . . 6

2 Data Collection **7**
- 2.1 Algorithm Codes . . . 7
  - 2.1.1 C Language . . . 7
  - 2.1.2 Python . . . 9
  - 2.1.3 Racket . . . 9
- 2.2 Testing Codes . . . 10
  - 2.2.1 C Language . . . 10
  - 2.2.2 Python . . . 12
  - 2.2.3 Racket . . . 12
  - 2.2.4 Bash Script . . . 12
- 2.3 Test Results . . . 14
  - 2.3.1 C Language . . . 14
  - 2.3.2 Python . . . 15
  - 2.3.3 Racket . . . 16

3 Visualization and Analysis **18**
- 3.1 R Visualization . . . 18
  - 3.1.1 C Language . . . 21
  - 3.1.2 Python . . . 22
  - 3.1.3 Racket . . . 23

4 Mathematical Modeling **24**
- 4.1 C Language . . . 24
  - 4.1.1 Bubble Sort . . . 24

4.1.2 Merge Sort 26
4.2 Python 27
4.2.1 Bubble Sort 27
4.2.2 Merge Sort 28
4.3 Racket 28
4.3.1 Bubble Sort 28
4.3.2 Merge Sort 29

5 Conclusion **31**
5.1 Recapitulation 31
5.2 Speed Differences and Reasons 32

List of Figures

3.1 Bubble Sort in C 21
3.2 Merge Sort in C 21
3.3 Bubble Sort in Python 22
3.4 Merge Sort in Python 22
3.5 Bubble Sort in Racket 23
3.6 Merge Sort in Racket 23

5.1 Comparison of Bubble Sort in different languages 31
5.2 Comparison of Merge Sort in different languages 32
5.3 Comparison of Merge Sort and Bubble Sort 32

List of Tables

2.1 Bubble Sort in C 14
2.2 Merge Sort in C 15
2.3 Bubble Sort in Python 15
2.4 Merge Sort in Python 16
2.5 Bubble Sort in Racket 16
2.6 Merge Sort in Racket 17

Chapter 1

Introduction

1.1 Programming Languages

In this research project, I will be using multiple programming languages to conduct my study. More specifically, I will be using C, Python, and Racket for the comparison, and R is the statistical tool used as a means of comparison and analysis.

1.1.1 C Language

C is a high-level and general-purpose programming language that is ideal for developing firmware or portable applications. Originally intended for writing system software, C was developed at Bell Labs by Dennis Ritchie for the Unix Operating System in the early 1970s. [5]

1.1.2 Python

Python is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis. Python is a general-purpose object-oriented language, meaning it can be used to create a variety of different programs and isnt specialized for any specific problems. This versatility, along with its beginner-friendliness, has made it one of the most-used programming languages today. [2]

1.1.3 Racket

Racket falls into a broad category of programming languages often called functional programming languages. Functional refers to a focus on functions in the mathematical sense. Unfortunately, the term can be a bit confusing, since many other languages (such as the widely-known C language) use constructs called functions, even though they do not follow a functional style. Functional languages are built around the evaluation of expressions and the application of

functions for their results, rather than the execution of sequences of commands for their effects on stored data. In some sense, functional languages are a little closer to expressing what to compute, while imperative (command-, statement-, and assignment-focused) languages are closer to expressing how to compute it. Functional languages are closer to math; imperative languages are closer to the model presented by most modern computer hardware systems. [**?**]

1.1.4 R Language

R is a programming language for statistical computing and graphics supported by the R Core Team and the R Foundation for Statistical Computing. Created by statisticians Ross Ihaka and Robert Gentleman, R is used among data miners and statisticians for data analysis and developing statistical software. Users have created packages to augment the functions of the R language. [4]

1.2 Algorithms

In this study, I have chosen two algorithms with different time complexities and varying difficulty: Bubble Sort and Merge Sort.

1.2.1 Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order. Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort. [1]

1.2.2 Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details. [3]

Chapter 2

Data Collection

In order to conduct my data collection phase, I had to write out a merge sort and a bubble sort in all three languages.

2.1 Algorithm Codes

2.1.1 C Language

Bubble Sort

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

void testWithArray(int n);
void swap(int *xp, int *yp);
void bubbleSort(int arr[], int n);

void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(&arr[j], &arr[j+1]);
           swapped = true;
        }
     }
     if (swapped == false)
        break;
```

```
    }
}
```

Merge Sort

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

void testWithArray(int n);
void merge(int arr[], int l, int m, int r);
void mergeSort(int arr[], int l, int r);

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

2.1.2 Python

Bubble Sort

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Merge Sort

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

2.1.3 Racket

Bubble Sort

```
#lang racket

(define (bsort-inner lst)
  (let loop ((lst lst) (res null))
    (let ((ca1 (car lst)) (cd1 (cdr lst)))
      (if (null? cd1)
          (values res ca1)
          (let ((ca2 (car cd1)) (cd2 (cdr cd1)))
            (if (<= ca1 ca2)
                (loop cd1 (cons ca1 res))
                (loop (cons ca1 cd2) (cons ca2 res))))))))

(define (bsort lst)
  (let loop ((lst lst) (res null))
    (if (null? lst)
        res
        (let-values (((ls mx) (bsort-inner lst)))
          (loop ls (cons mx res))))))
```

Merge Sort

```
#lang racket

(define (merge as bs)
  (match* (as bs)
    [((list) bs)  bs]
    [(as (list))  as]
    [((list a as ...) (list b bs ...))
     (if (< a b)
         (cons a (merge as (cons b bs)))
         (cons b (merge (cons a as) bs)))]))

(define (merge-sort vs)
  (match vs
    [(list)  vs]
    [(list a)  vs]
    [_  (define-values (lvs rvs)
          (split-at vs (quotient (length vs) 2)))
        (merge (merge-sort lvs) (merge-sort rvs))]))
```

2.2 Testing Codes

While now we know how the algorithms are laid out in all three languages, we still need the testing code that would generate arrays of custom length of random numbers and benchmark the time taken to run one sorting instance on that array. In this section, I will run display all testing codes.

2.2.1 C Language

Bubble Sort

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

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

void testWithArray(int n){

   int array[n];

   srand(time(NULL));

   for(int i=0;i < n ;i++){
      array[i]=(rand()%1000000000);
   }

   float startTime = (float)clock()/CLOCKS_PER_SEC;

   bubbleSort(array, n);

   float endTime = (float)clock()/CLOCKS_PER_SEC;

   float timeElapsed = endTime - startTime;

   printf("(size␣=␣%d):␣%lf\n", n, timeElapsed);

}
```

Merge Sort

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

void testWithArray(int n){

   int array[n];

   srand(time(NULL));

   for(int i=0;i < n ;i++){
      array[i]=(rand()%1000000000);
   }

   float startTime = (float)clock()/CLOCKS_PER_SEC;

   mergeSort(array, 0, n);

   float endTime = (float)clock()/CLOCKS_PER_SEC;

   float timeElapsed = endTime - startTime;

   printf("(size␣=␣%d):␣%lf\n", n, timeElapsed);

}
```

2.2.2 Python

Bubble Sort

```
import random
import time

def testWithArray(n):
    arr = [random.randint(0,1000000000) for i in range(n)]
    start = time.time()
    bubbleSort(arr)
    end = time.time()
    timeelapsed = end - start
    print("(size␣=␣", n ,"):␣" , timeelapsed)
```

Merge Sort

```
import random
import time

def testWithArray(n):
    arr = [random.randint(0,1000000000) for i in range(n)]
    start = time.time()
    mergeSort(arr)
    end = time.time()
    timeelapsed = end - start
    print("(size␣=␣", n ,"):␣" , timeelapsed)
```

2.2.3 Racket

Bubble Sort

```
#lang racket

(require benchmark-ips)
(define (listn x) (build-list x (thunk* (random 100000000))))
(define listnn (listn n))
(benchmark/ips "(size␣=␣n)" (bsort listnn))
```

Merge Sort

```
#lang racket

(require benchmark-ips)

(define (listn x) (build-list x (thunk* (random 100000000))))

(define listnn (listn n))

(benchmark/ips "(size␣=␣n)" (merge-sort listnn))
```

2.2.4 Bash Script

Finally, to run all of these tests, I have set up a BASH script to run on my computer and run four trials for each and every code.

```
gcc -o mcodeinc mcodeinc\ copy.c
touch mcodeinc.txt
./mcodeinc >> mcodeinc.txt
echo trial 1 C merge
./mcodeinc >> mcodeinc.txt
echo trial 2 C merge
./mcodeinc >> mcodeinc.txt
echo trial 3 C merge
./mcodeinc >> mcodeinc.txt
echo trial 4 C merge

touch mcodeinpy.txt
python mcodeinpy\ copy.py >> mcodeinpy.txt
echo trial 1 py merge
python mcodeinpy\ copy.py >> mcodeinpy.txt
echo trial 2 py merge
python mcodeinpy\ copy.py >> mcodeinpy.txt
echo trial 3 py merge
python mcodeinpy\ copy.py >> mcodeinpy.txt
echo trial 4 py merge

touch mcodeinrkt.txt
racket mcodeinrkt\ copy.rkt >> mcodeinrkt.txt
echo trial 1 rkt merge
racket mcodeinrkt\ copy.rkt >> mcodeinrkt.txt
echo trial 2 rkt merge
racket mcodeinrkt\ copy.rkt >> mcodeinrkt.txt
echo trial 3 rkt merge
racket mcodeinrkt\ copy.rkt >> mcodeinrkt.txt
echo trial 4 rkt merge

gcc -o codeinc codeinc\ copy.c
touch codeinc.txt
./codeinc >> codeinc.txt
echo trial 1 C bubble
./codeinc >> codeinc.txt
echo trial 2 C bubble
./codeinc >> codeinc.txt
echo trial 3 C bubble
./codeinc >> codeinc.txt
echo trial 4 C bubble

touch codeinpy.txt
python codeinpy\ copy.py >> codeinpy.txt
echo trial 1 py bubble
python codeinpy\ copy.py >> codeinpy.txt
echo trial 2 py bubble
python codeinpy\ copy.py >> codeinpy.txt
echo trial 3 py bubble
python codeinpy\ copy.py >> codeinpy.txt
echo trial 4 py bubble


touch codeinrkt.txt
racket codeinrkt\ copy.rkt >> codeinrkt.txt
```

```
echo trial 1 rkt bubble
racket codeinrkt\ copy.rkt >> codeinrkt.txt
echo trial 2 rkt bubble
racket codeinrkt\ copy.rkt >> codeinrkt.txt
echo trial 3 rkt bubble
racket codeinrkt\ copy.rkt >> codeinrkt.txt
echo trial 4 rkt bubble
```

2.3 Test Results

Once I finished testing both the algorithms and the testing code, I had to move on the actual data collection. In this section, I will run 4 trials on all three languages and average them out to find an average running time for each algorithm in each language. Later on in my paper, I will graph this dataset using R tools.

2.3.1 C Language

Bubble Sort

Table 2.1: Bubble Sort in C

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
100	0.000030	0.000030	0.000028	0.000033	0.000030
1000	0.002364	0.002238	0.002207	0.003026	0.002459
10000	0.314041	0.314055	0.315845	0.386813	0.332689
12500	0.501716	0.491243	0.493394	0.621091	0.526861
15000	0.741801	0.714311	0.716607	0.925301	0.774505
20000	1.293619	1.281955	1.291895	1.638969	1.376610
30000	2.946970	2.941916	2.963792	3.636154	3.122208
40000	5.290429	5.275138	5.821433	5.374234	5.440309
50000	8.305266	8.303268	8.482247	8.306714	8.349374
60000	11.950539	12.098505	13.396408	12.014277	12.364932

Merge Sort

Table 2.2: Merge Sort in C

	Trial 1	Trial 2	Trial 3	Trial 4	Average
100	0.000013	0.000014	0.000013	0.000013	0.000013
1000	0.000137	0.000136	0.000141	0.000200	0.000154
10000	0.001786	0.001721	0.001792	0.001778	0.001769
100000	0.021886	0.021719	0.022862	0.020841	0.021827
150000	0.031828	0.034832	0.033666	0.032922	0.033312
200000	0.044329	0.045112	0.043959	0.043971	0.044343
300000	0.068700	0.069779	0.067295	0.069056	0.068708
400000	0.093967	0.093550	0.091818	0.093572	0.093227
500000	0.121026	0.115452	0.118177	0.116026	0.117670
600000	0.153136	0.141397	0.142094	0.140003	0.144158

2.3.2 Python

Bubble Sort

Table 2.3: Bubble Sort in Python

	Trial 1	Trial 2	Trial 3	Trial 4	Average
100	0.000545	0.000563	0.000913	0.000627	0.000662
1000	0.052419	0.053385	0.060523	0.063075	0.057351
10000	5.987370	5.178716	5.230582	5.207895	5.401141
12500	9.667572	8.073384	8.180103	8.130587	8.512912
15000	11.987362	11.709624	11.807110	11.727474	11.807893
20000	21.391259	20.754657	20.906825	20.946968	20.999927
30000	50.913563	65.675317	47.738047	47.246690	52.893404
40000	101.911117	105.566518	96.915071	87.594773	97.996870
50000	137.184626	170.842736	140.369457	139.046607	146.860857
60000	201.889625	215.850203	210.537913	222.182034	212.614944

Merge Sort

Table 2.4: Merge Sort in Python

	Trial 1	Trial 2	Trial 3	Trial 4	Average
100	0.000221	0.000211	0.000214	0.000212	0.000215
1000	0.002788	0.002741	0.002908	0.002673	0.002778
10000	0.037074	0.034502	0.033595	0.034995	0.035042
100000	0.422744	0.420146	0.407640	0.406075	0.414151
150000	0.685619	0.645911	0.630336	0.640195	0.650515
200000	0.879264	0.941544	0.871177	0.875316	0.891825
300000	1.379623	1.412662	1.398469	1.398706	1.397365
400000	1.965367	1.966624	1.971832	1.964121	1.966986
500000	2.673376	2.579890	2.640890	2.621039	2.628799
600000	3.312979	3.345643	3.312136	3.667269	3.409507

2.3.3 Racket

Bubble Sort

Table 2.5: Bubble Sort in Racket

	Trial 1	Trial 2	Trial 3	Trial 4	Average
100	2.000018	2.000004	2.000028	2.000006	2.000014
1000	2.000730	2.001130	2.000785	2.000427	2.000768
10000	2.257822	2.073509	2.228522	2.245498	2.201338
12500	2.234341	2.120117	2.274963	2.215435	2.211214
15000	2.626533	2.649679	2.641777	2.604737	2.630682
20000	2.372068	2.687511	2.368035	2.370197	2.449453
30000	2.888484	3.706926	2.899261	2.843553	3.084556
40000	5.143040	5.840422	5.182659	5.138773	5.326224
50000	8.571410	8.297307	8.343139	8.397295	8.402288
60000	12.623135	12.819341	12.535683	12.496122	12.618570

Merge Sort

Table 2.6: Merge Sort in Racket

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
100	2.000021	2.000024	2.000015	2.000030	2.000023
1000	2.000362	2.000581	2.000393	2.000378	2.000429
10000	2.004199	2.002821	2.004174	2.011014	2.005552
100000	2.136685	2.122695	2.059807	2.145486	2.116168
150000	2.234657	2.210700	2.224171	2.163794	2.208331
200000	2.365633	2.043020	2.329901	2.044400	2.195739
300000	2.475769	2.529132	2.561504	2.512227	2.519658
400000	2.714498	2.544158	2.578192	2.799971	2.659205
500000	2.259050	2.288681	2.204578	2.180730	2.233260
600000	2.782474	2.824598	2.734272	2.960621	2.825491

Chapter 3

Visualization and Analysis

3.1 R Visualization

Having compiled all data needed to model and hypothesize, in this section, I will visualize, using the R language, both algorithms in each of the three languages. Here is the R code I used to come up with the graphs along with their most optimal models.

```
library(basicTrendline)

bubbleinrkt <- c(2.000014, 2.000768, 2.201338, 2.211214,
                 2.630682, 2.449453, 3.084556, 5.326224,
                 8.402288, 12.618570)

bubbleinpy <- c(0.000662, 0.057351, 5.401141, 8.512912,
                11.807893, 20.999927, 52.893404, 97.996870,
                146.860857, 212.614944)

bubbleinc <- c(0.000030, 0.002459, 0.332689, 0.526861,
               0.774505, 1.376610, 3.122208, 5.440309,
               8.349374, 12.364932)

n1 <- c(100, 1000, 10000, 12500, 15000,
        20000, 30000, 40000, 50000, 60000)

bubbleC <- data.frame(x = n1, y = bubbleinc)
bubblePy <- data.frame(x = n1, y = bubbleinpy)
bubbleRkt <- data.frame(x = n1, y = bubbleinrkt)

n2 <- c(100, 1000, 10000, 100000, 150000,
        200000, 300000, 400000, 500000, 600000)

mergeinrkt  <- c(2.000023, 2.000429, 2.005552, 2.116168,
                 2.208331, 2.195739, 2.519658, 2.659205,
                 2.233260, 2.825491)

mergeinpy   <- c(0.000215, 0.002778, 0.035042, 0.414151,
```

```
                0.650515, 0.891825, 1.397365, 1.966986,
                2.628799, 3.409507)

mergeinc    <- c(0.000013, 0.000154, 0.001769, 0.021827,
                0.033312, 0.044343, 0.068708, 0.093227,
                0.117670, 0.144158)

mergecurvefit <- function(x,y, lang){
  title = paste("Merge␣Sort␣in␣",lang)
  plot(x, y, pch=20, xlab = "Size␣of␣Sorted␣Array",
      ylab = "Time␣Spent␣(s)", main = title)

  f <- function(x,a,b) {a * x * log(x) + b}
  fit <- nls(y ~ f(x,a,b), start = c(a=0.1, b=0.1))
  co <- coef(fit)

  eq <- paste0("y␣=␣", formatC(co[1],
              format = "e", digits = 3), "␣*␣x␣*␣log(x)␣+␣",
              formatC(co[2], format = "e", digits = 3))

  mtext(eq, 3, line=-2)
  curve(f(x, a=co[1], b=co[2]),
          add = TRUE, col="red", lwd=2)
}

bubblecurvefit <- function(x,y, lang){
  title = paste("Bubble␣Sort␣in␣",lang)
  plot(x, y, pch=20, xlab = "Size␣of␣Sorted␣Array",
       ylab = "Time␣Spent␣(s)", main = title)

  f <- function(x,a,b,d) {(a*x^2) + (b*x) + d}
  fit <- nls(y ~ f(x,a,b,d), start = c(a=0.1, b=0.1, d=0.1))
  co <- coef(fit)

  eq <- paste0("y␣=␣", formatC(co[1], format = "e",
               digits = 3),  "␣*␣x^2␣+␣", formatC(co[2],
               format = "e", digits = 3), "␣*␣x␣+␣",
               formatC(co[3], format = "e", digits = 3))

  mtext(eq, 3, line=-2)
  curve(f(x, a=co[1], b=co[2], d=co[3]),
          add = TRUE, col="red", lwd=2)
}

mergecurvefit(n2, mergeinc, "C")
mergecurvefit(n2, mergeinpy, "Python")
mergecurvefit(n2, mergeinrkt, "Racket")

bubblecurvefit(n1, bubbleinc, "C")
bubblecurvefit(n1, bubbleinpy, "Python")
bubblecurvefit(n1, bubbleinrkt, "Racket")

bubblealllangs <- function(){
  title = paste("Bubble␣Sort␣in␣C,␣Python␣and␣Racket")
  x=seq(100,1000000,10)
```

```
  y1=3.4304*10^(-9)*x^2 - 1.5075*10^(-6)*x + 0.01212
  y2=5.9396*10^(-8)*x^2 - 3.4482*10^(-7)*x - 0.5980
  y3=4.1680*10^(-9)*x^2 - 8.4024*10^(-5)*x + 2.3450
  plot(x,y1,type='l',col='red', xlab = "# of elements",
       ylab = "Time spent (s)", main = title)
  lines(x,y2,col='green')
  lines(x,y3,col='blue')
  legend('bottomright',inset=0.05,c("C","Python","Racket"),
         lty=1,col=c("red","green","blue"),title="Language")

}

bubblealllangs()

mergealllangs <- function(){
  title = paste("Merge Sort in C, Python and Racket")
  x=seq(100,1000000,10)

  plot(x,y3,type='l',col='blue', xlab = "# of elements",
       ylab = "Time spent (s)", main = title,
       xlim = c(0,1000000), ylim = c(0,3.2))
  lines(x,y1,col='red')
  lines(x,y2,col='green')

  legend('bottomright',inset=0.15,c("C","Python","Racket"),
         lty=1,col=c("red","green","blue"),title="Language")
}

mergealllangs()

alllangs <- function(){
  title = paste("Merge Sort vs. Bubble Sort")
  x=seq(100,1000000,10)
  y1=3.4304*10^(-9)*x^2 - 1.5075*10^(-6)*x + 0.01212
  y2=5.9396*10^(-8)*x^2 - 3.4482*10^(-7)*x - 0.5980
  y3=4.1680*10^(-9)*x^2 - 8.4024*10^(-5)*x + 2.3450

  y4= 1.7977*10^(-8)*x*log(x) + 4.4163*10^(-4)
  y5= 4.1333*10^(-7)*x*log(x) - 0.05757
  y6= 8.7035*10^(-8)*x*log(x) + 2.02427

  plot(x,y3,type='l',col='blue', xlab = "# of elements",
       ylab = "Time spent (s)", main = title,
       xlim = c(0,100000), ylim = c(0,10))
  lines(x,y1,col='blue')
  lines(x,y2,col='blue')

  lines(x,y4,col='red')
  lines(x,y5,col='red')
  lines(x,y6,col='red')

  legend('bottomright',inset=0.15,
         c("Merge Sort", "Bubble Sort"),
         lty=1,col=c("red","blue"),title="Algorithm")
}
```

```
alllangs()
```

3.1.1 C Language

Bubble Sort

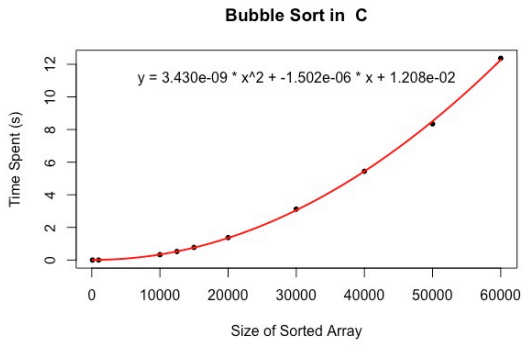

Figure 3.1: Bubble Sort in C

Merge Sort

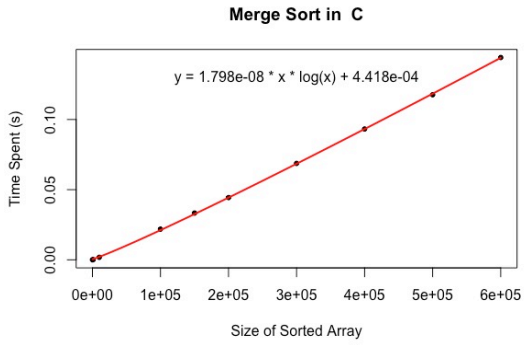

Figure 3.2: Merge Sort in C

3.1.2 Python

Bubble Sort

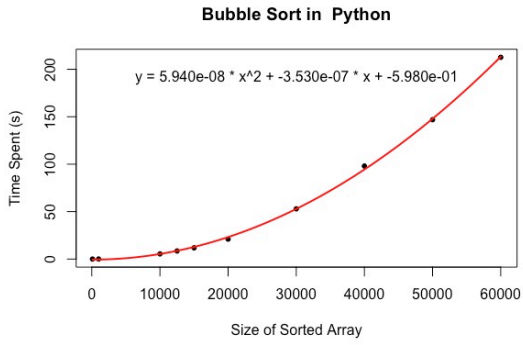

Figure 3.3: Bubble Sort in Python

Merge Sort

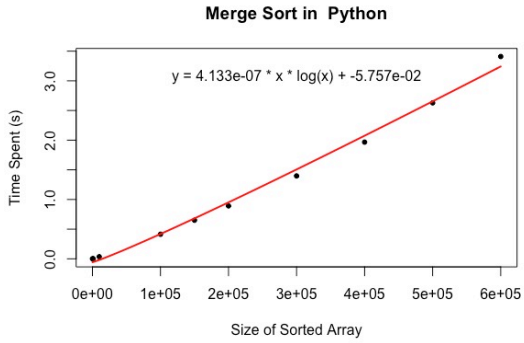

Figure 3.4: Merge Sort in Python

3.1.3 Racket

Bubble Sort

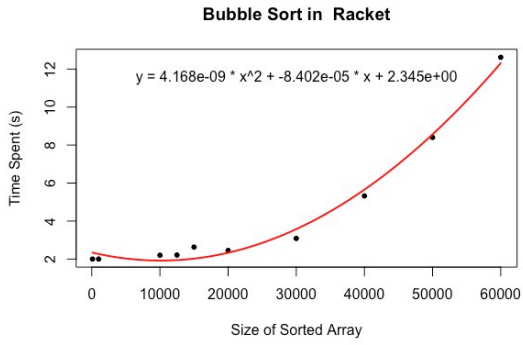

Figure 3.5: Bubble Sort in Racket

Merge Sort

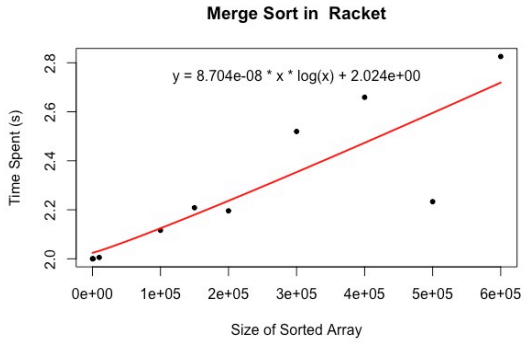

Figure 3.6: Merge Sort in Racket

Chapter 4

Mathematical Modeling

Having visualized and predicted the model for all six datasets using R intelligence tools, let's use the tools we have learned in the Mathematical Modeling class to find an adequate model for all six datasets. First, we must point out that it is a well known fact that a bubble sort has a time complexity of $\mathcal{O}(n^2)$. This is because the inner loop of a bubble sort does $\mathcal{O}(n)$ work on each iteration, and the outer loop runs for $\mathcal{O}(n)$ iterations, so the total work is $\mathcal{O}(n^2)$. Second, the time complexity of a merge sort is $\mathcal{O}(n * log(n))$ as the merge sort always divides an array into two halves and takes linear time to merge two halves.

4.1 C Language

4.1.1 Bubble Sort

We know that the model for the time complexity of a bubble sorting algorithm is $\mathcal{O}(n^2)$. This means that a possible equation for this model is a polynomial of the second degree, i.e.

$$y = ax^2 + bx + c \tag{4.1}$$

To find the values of a, b and c, we will be using the transformed Least-Squares fit method. Examples of the least-squares criterion appear easy to apply. However, it may be difficult for other types of functions. Let's apply the least-squares criterion to the model. Applying the least-squares criterion requires that we minimize:

$$S = \sum_{i=1}^{m} |y_i - f(x_i)|^2 = \sum_{i=1}^{m} |y_i - (ax_i^2 + bx_i + c)|^2 \tag{4.2}$$

$$= \sum_{i=1}^{m} |y_i - ax_i^2 - bx_i - c|^2 \tag{4.3}$$

A necessary condition to optimize is setting the partial derivatives $\frac{\delta S}{\delta a}$, $\frac{\delta S}{\delta b}$ and $\frac{\delta S}{\delta c}$ to zero:

$$\frac{\delta S}{\delta a} = -2 \sum_{i=1}^{m}(y_i - ax_i^2 - bx_i - c) \sum_{i=1}^{m} x_i^2 = 0 \tag{4.4}$$

$$\frac{\delta S}{\delta b} = -2 \sum_{i=1}^{m}(y_i - ax_i^2 - bx_i - c) \sum_{i=1}^{m} x_i = 0 \tag{4.5}$$

$$\frac{\delta S}{\delta c} = -2 \sum_{i=1}^{m}(y_i - ax_i^2 - bx_i - c) = 0 \tag{4.6}$$

Once we solve the equations above, we get the three following equalities:

$$\sum_{i=1}^{m} ax_i^4 + \sum_{i=1}^{m} bx_i^3 + \sum_{i=1}^{m} cx_i^2 = \sum_{i=1}^{m} x_i^2 y_i \tag{4.7}$$

$$\sum_{i=1}^{m} ax_i^3 + \sum_{i=1}^{m} bx_i^2 + \sum_{i=1}^{m} cx_i = \sum_{i=1}^{m} x_i y_i \tag{4.8}$$

$$\sum_{i=1}^{m} ax_i^2 + \sum_{i=1}^{m} bx_i + c\ m = \sum_{i=1}^{m} y_i \tag{4.9}$$

We can conclude from the equalities above the following matrix:

$$\begin{bmatrix} \sum x_i^4 & \sum x_i^3 & \sum x_i^2 \\ \sum x_i^3 & \sum x_i^2 & \sum x_i \\ \sum x_i^2 & \sum x_i & m \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x_i^2 y_i \\ \sum x_i y_i \\ \sum y_i \end{bmatrix} \tag{4.10}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.2825 \cdot 10^{+19} & 4.46329 \cdot 10^{+14} & 9482260000 \\ 4.46329 \cdot 10^{+14} & 9482260000 & 238600 \\ 9482260000 & 238600 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 77742172816 \\ 1519708.11 \\ 32.289977 \end{bmatrix} \tag{4.11}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 3.430453052632288 \cdot 10^{-9} \\ b &= -1.507590425645456 \cdot 10^{-6} \\ c &= 0.012124031270596 \end{aligned} \tag{4.12}$$

We, therefore conclude that a quadratic model for the bubble sort in the C language is as follows:

$$y = 3.4304 \cdot 10^{-9}x^2 - 1.5075 \cdot 10^{-6}x + 0.01212 \tag{4.13}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

4.1.2 Merge Sort

We know that the model for the time complexity of a bubble sorting algorithm is $\mathcal{O}(nlog(n))$. This means that a possible equation for this model is as follows, i.e.

$$y = a \cdot xlog(x) + b \tag{4.14}$$

To find the values of a, and b, we will be using the transformed Least-Squares fit method. The least-squares criterion's application seemed easy to apply. Let's apply the least-squares criterion to the model. Applying the least-squares criterion requires that we minimize:

$$S = \sum_{i=1}^{m} |y_i - f(x_i)|^2 = \sum_{i=1}^{m} |y_i - (a \cdot xlog(x) + b)|^2 \tag{4.15}$$

$$= \sum_{i=1}^{m} |y_i - a \cdot xlog(x) - b|^2 \tag{4.16}$$

A necessary condition to optimize is setting the partial derivatives $\frac{\delta S}{\delta a}$ and $\frac{\delta S}{\delta b}$ to zero:

$$\frac{\delta S}{\delta a} = -2 \sum_{i=1}^{m} (y_i - a \cdot xlog(x) - b) \sum_{i=1}^{m} xlog(x) = 0 \tag{4.17}$$

$$\frac{\delta S}{\delta b} = 2 \sum_{i=1}^{m} (y_i - a \cdot xlog(x) - b) = 0 \tag{4.18}$$

Once we solve the equations above, we get the three following equalities:

$$a \sum_{i=1}^{m} (xlog(x))^2 + b \sum_{i=1}^{m} xlog(x) = \sum_{i=1}^{m} xlog(x) y_i \tag{4.19}$$

$$a \sum_{i=1}^{m} xlog(x) + bm = \sum_{i=1}^{m} y_i \tag{4.20}$$

We can conclude from the equalities above the following matrix:

$$\begin{bmatrix} \sum (xlog(x))^2 & \sum xlog(x) \\ \sum xlog(x) & m \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum xlog(x) y_i \\ \sum y_i \end{bmatrix} \tag{4.21}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.98385 \cdot 10^{+13} & 12580155.81 \\ 12580155.81 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1240742.182 \\ 0.525181 \end{bmatrix} \tag{4.22}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 4.139572714805141 \cdot 10^{-8} \\ b &= 4.416302609266260 \cdot 10^{-4} \end{aligned} \tag{4.23}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models. model for the merge sort in the C language is as follows:

$$y = 4.1395 \cdot 10^{-8} x log(x) + 4.4163 \cdot 10^{-4} \tag{4.24}$$

And, in case we want to replicate the R result, we would just need to substitute $log(x) = \frac{ln(x)}{ln(10)}$ making the model also equal to:

$$y = 1.7977 \cdot 10^{-8} x ln(x) + 4.4163 \cdot 10^{-4} \tag{4.25}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

4.2 Python

4.2.1 Bubble Sort

Using our results from our analysis of the bubble sort in the C language, what we will reuse is the matrix equality to find a quadratic model for the bubble sort in Python:

$$\begin{bmatrix} \sum x_i^4 & \sum x_i^3 & \sum x_i^2 \\ \sum x_i^3 & \sum x_i^2 & \sum x_i \\ \sum x_i^2 & \sum x_i & m \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x_i^2 y_i \\ \sum x_i y_i \\ \sum y_i \end{bmatrix} \tag{4.26}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.2825 \cdot 10^{+19} & 4.46329 \cdot 10^{+14} & 9482260000 \\ 4.46329 \cdot 10^{+14} & 9482260000 & 238600 \\ 9482260000 & 238600 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1.34989 \cdot 10^{+12} \\ 26364213.57 \\ 557.145961 \end{bmatrix} \tag{4.27}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 5.939604847778412 \cdot 10^{-8} \\ b &= -3.448209508458306 \cdot 10^{-7} \\ c &= -0.598053936008141 \end{aligned} \tag{4.28}$$

We, therefore conclude that a quadratic model for the bubble sort in the Python language, is as follows:

$$y = 5.9396 \cdot 10^{-8} x^2 - 3.4482 \cdot 10^{-7} x - 0.5980 \tag{4.29}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

4.2.2 Merge Sort

Using our results from our analysis of the merge sort in the C language, what we will reuse is the matrix equality to find a quadratic model for the merge sort in Python:

$$\begin{bmatrix} \sum(xlog(x))^2 & \sum xlog(x) \\ \sum xlog(x) & m \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum xlog(x)y_i \\ \sum y_i \end{bmatrix} \tag{4.30}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.98385 \cdot 10^{+13} & 12580155.81 \\ 12580155.81 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 27673914.82 \\ 11.397183 \end{bmatrix} \tag{4.31}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 9.517296343438738 \cdot 10^{-7} \\ b &= -0.057572408904026 \end{aligned} \tag{4.32}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models. model for the merge sort in the Python language is as follows:

$$y = 9.5172 \cdot 10^{-7} xlog(x) - 0.05757 \tag{4.33}$$

And, in case we want to replicate the R result, we would just need to substitute $log(x) = \frac{ln(x)}{ln(10)}$ making the model also equal to:

$$y = 4.1333 \cdot 10^{-7} xln(x) - 0.05757 \tag{4.34}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

4.3 Racket

4.3.1 Bubble Sort

Using our results from our analysis of the bubble sort in the C language, what we will reuse is the matrix equality to find a quadratic model for the bubble sort in Racket:

$$\begin{bmatrix} \sum x_i^4 & \sum x_i^3 & \sum x_i^2 \\ \sum x_i^3 & \sum x_i^2 & \sum x_i \\ \sum x_i^2 & \sum x_i & m \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x_i^2 y_i \\ \sum x_i y_i \\ \sum y_i \end{bmatrix} \tag{4.35}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.2825 \cdot 10^{+19} & 4.46329 \cdot 10^{+14} & 9482260000 \\ 4.46329 \cdot 10^{+14} & 9482260000 & 238600 \\ 9482260000 & 238600 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 79869972206 \\ 1623117.854 \\ 42.925107 \end{bmatrix} \tag{4.36}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 4.168065707722193 \cdot 10^{-9} \\ b &= -8.402491569874184 \cdot 10^{-5} \\ c &= 2.345076914801395 \end{aligned} \tag{4.37}$$

We, therefore conclude that a quadratic model for the bubble sort in the Racket language, is as follows:

$$y = 4.1680 \cdot 10^{-9}x^2 - 8.4024 \cdot 10^{-5}x + 2.3450 \tag{4.38}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

4.3.2 Merge Sort

Using our results from our analysis of the merge sort in the C language, what we will reuse is the matrix equality to find a quadratic model for the merge sort in Racket:

$$\begin{bmatrix} \sum (xlog(x))^2 & \sum xlog(x) \\ \sum xlog(x) & m \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum xlog(x)y_i \\ \sum y_i \end{bmatrix} \tag{4.39}$$

Using computation tools (Excel and/or R), we find the values to fill the matrix, and the matrix becomes:

$$\begin{bmatrix} 2.98385 \cdot 10^{+13} & 12580155.81 \\ 12580155.81 & 10 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 31445483.03 \\ 22.763856 \end{bmatrix} \tag{4.40}$$

Using Matlab, I found that the results of this system of equations is:

$$\begin{aligned} a &= 2.004070590880563 \cdot 10^{-7} \\ b &= 2.024270397124837 \end{aligned} \tag{4.41}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models. model for the merge sort in the Racket language is as follows:

$$y = 2.0040705 \cdot 10^{-7}xlog(x) + 2.02427 \tag{4.42}$$

And, in case we want to replicate the R result, we would just need to substitute $log(x) = \frac{ln(x)}{ln(10)}$ making the model also equal to:

$$y = 8.7035 \cdot 10^{-8} x ln(x) + 2.02427 \tag{4.43}$$

We also notice that the model we found manually is almost equal to the one that R found, hence, we notice that R uses the least squares method in predicting models.

Chapter 5

Conclusion

5.1 Recapitulation

In this research paper, I have reported in detail all phases of my research, starting from the data collection itself, to plotting the data, to modeling it both manually and digitally, and the model results seems conclusive and correct. Now, in order to see how models compare to each other, here are two graphs with all models that will help us grasp graphically how each language compares to the other two languages for both sorting algorithms separately and unitedly.

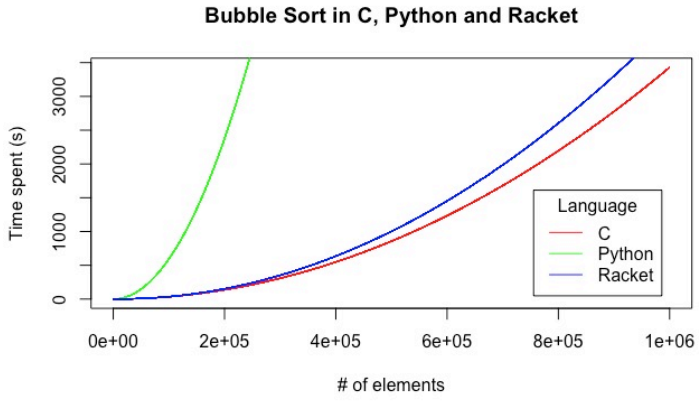

Figure 5.1: Comparison of Bubble Sort in different languages

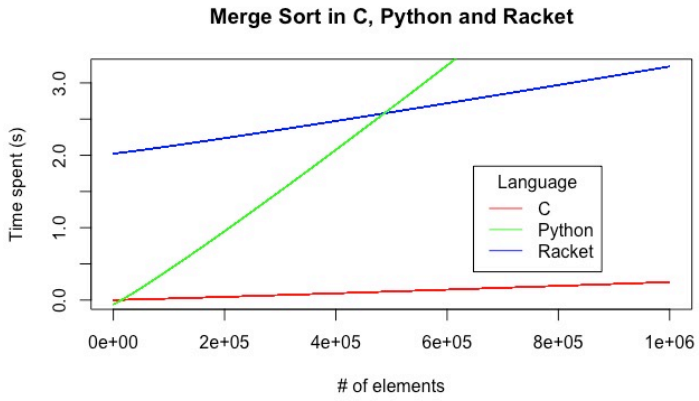

Figure 5.2: Comparison of Merge Sort in different languages

And finally, just to ensure graphically that a merge sorting algorithm is much much faster than a bubble sorting approach, here's a graph of all bubble sorts vs. all merge sorts.

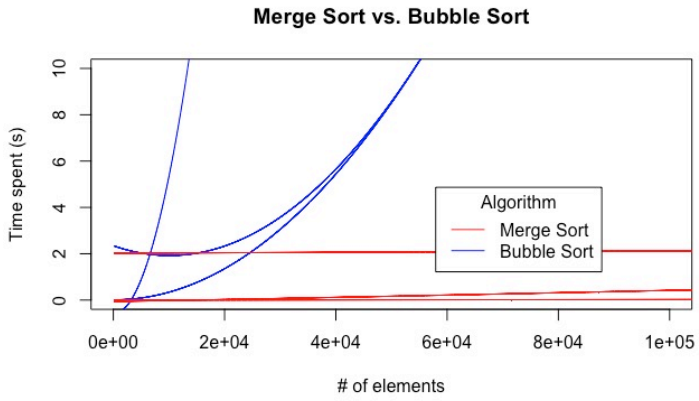

Figure 5.3: Comparison of Merge Sort and Bubble Sort

5.2 Speed Differences and Reasons

We can conclude from both the graphs and the numerical data that the fastest language by far in terms of sorting regardless of which algorithm is C. That is what I expected as C is a very low-level programming language that has a very low-level array definition that makes array mutable (in terms of their contents). This makes moving data around in one array very easy through indexing and

looping. The next fastest language is up to debate as both other languages, Python and Racket, are mostly interpreted (especially in our experimentation) which makes them inherently slower than compiled languages like C.
Additionally, the models and the experimental results show inconclusive data when it comes to second best, as Racket is faster than Python in terms of merge sorting, while it is slower than Python in bubble sorting.
The explanation I found for there being better performance in Racket than in Python is that merge sorting is subcomposed of many list constructions, which is what Racket and other LISP (List Processing) languages are designed to do best. This explanation also works for showing why a bubble sort is slower in Racket than it is in Python. And that is because Racket is functional. It was designed to deal with function, and process lists, making there no simple and fast way to access an element at a specific index. As a conclusion, it seems that all results and these time complexity based models are valid and adequate for predicting how these languages would run sorting algorithms in terms of the size of the sample.

Bibliography

[1] Bubble sort.

[2] What is python used for? a beginner's guide.

[3] Merge sort, Jan 2022.

[4] R (programming language), Mar 2022.

[5] Techopedia. What is c (programming language)? - definition from techopedia, May 2018.