**Team Project 2: Analyzing Sorting Algorithms**

**Apple 10 Team Report**

CS350-01, Fall 2021

Southeast Missouri State University

**11/29/2021**

|  |  |
| --- | --- |
| Name | Email |
| Blake Bleem | bhbleem1s@semo.edu |
| Austin Gray | agray2s@semo.edu |
| Sean Poston | sposton1s@semo.edu |

**Team Member Contributions**

Each team member’s contribution can be found in Table 1.

*Table 1 - Team member contributions*

|  |  |
| --- | --- |
| Team Member  Email | Contributions |
| Blake Bleem  bhbleem1s@semo.edu | * Participated in all three Lab times * Scheduled Lab times * Drafted abstract, introduction, and main contexts * Collected data and created graphs |
| Austin Gray  agray2s@semo.edu | * Drafted Requirement 5 of report * Drafted Discussion of report * Video edited the various parts of zoom recording into one video (issue caused by exchanging host during meeting) * Attended and fully participated in group zoom meetings |
| Sean Poston  sposton1s@semo.edu | * Sent initial email to get team together to begin work * Coordinated all zoom meetings * Drafted Requirement 1 of report * Drafted Requirement 2 of report * Drafted Introduction of report * Attended and fully participated in group zoom meetings |

Video Link to Team Presentation: <https://youtu.be/rm1xpwemXcA>

# Table of Contents

[**Table of Contents**](#_heading=h.r5ja3lkkxhlq) **2**

[**Abstract**](#_heading=h.2b31fzf0yto7) **3**

[**Introduction**](#_heading=h.ot618hnatw3v) **4**

[Radix Sort](#_heading=h.1x1kduvkt18h) 5

[Merge Sort](#_heading=h.od0ftaggg0tr) 7

[Selection Sort](#_heading=h.pfcelr9hew9l) 8

[Heapsort](#_heading=h.ic7ls8psxtxg) 9

[Quicksort](#_heading=h.l5k5wa5epd4p) 11

[Timsort](#_heading=h.ycgeuuedy0i0) 13

[**Main Context - Running the Algorithms**](#_heading=h.2et92p0) **13**

[Radix Sort](#_heading=h.3xtt1sedvc5t) 17

[Merge Sort](#_heading=h.ws1wq9jp0vaz) 18

[Selection Sort](#_heading=h.l26xs9frtw93) 18

[Heapsort](#_heading=h.syigqxir50ay) 19

[Quicksort](#_heading=h.63701vv4llls) 20

[Timsort](#_heading=h.hvi1x0qki6ur) 21

[**Discussion**](#_heading=h.upglbi) **21**

[**Conclusion**](#_heading=h.36ei31r) **24**

[**Acknowledge**](#_heading=h.1ljsd9k) **24**

[**References**](#_heading=h.t5gkxbt48pl2) **25**

[**Appendix - Source Codes**](#_heading=h.8uri9v52aeo6) **26**

[Driver Code (main.c)](#_heading=h.45jfvxd) 26

[Helper Code (array\_util.h)](#_heading=h.zdu1tsq2fcqm) 28

[Radix Sort (radixsort.h)](#_heading=h.vqikugdewpg0) 31

[Merge Sort (mergesort.h)](#_heading=h.t4spoyh2urxh) 32

[Selection Sort (selectionsort.h)](#_heading=h.r1p3abhzt4zu) 34

[Heap Sort (heapsort.h)](#_heading=h.onuol2llcg12) 35

[Quick Sort (quicksort.h)](#_heading=h.p1c5ostz56g2) 36

[Tim Sort (timsort.h)](#_heading=h.auvqxgycvq1t) 37

# 

# Abstract

In the world of Computer Science, one of the most prominent topics of interest is sorting algorithms. More specifically, the time complexities of sorting algorithms. This makes sense given the simpler computers of the past that needed the extra time efficiency, and even today, where even fast computers can struggle to sort large datasets. In this project, we went beyond the standard time complexity analysis, measuring also the power consumption and memory usage of these algorithms. By using a Raspberry Pi 3B, a common IoT device, we can explore how these algorithms really do affect devices in a real-world application. With these extra dimensions of complexity, we can form a better understanding of what makes a sorting algorithm good, and the conditions in which the algorithm works best. This knowledge could be useful for future applications, where power consumption and time could be crucial factors of an application or piece of hardware.

# 

# **Introduction**

For this project, our team wrote up a set of six sorting algorithms, each of them written in C. The algorithms used were radix sort, merge sort, selection sort, heapsort, quicksort, and timsort. Initially, we had these algorithms written in Python, but for the sake of getting a more machine-level comparison between the sorts we switched over to using pure C. After writing out these algorithms, we also wrote a driver code, that would construct three integers arrays of a given size, one already sorted, one sorted in reverse, and one randomly filled. The driver code would then take turns loading these arrays from memory, sorting them with the specified sorting algorithms, and then writing them back to memory. This code also found the time that was taken in performing these operations, using the system time.

By using the provided Raspberry Pi in the lab room, we were able to run this driver code on our six different algorithms, and compare their run times, power consumption, CPU usage, and memory usage. The power consumption was measured using a USB power meter connected to the Raspberry Pi, the times were measured by the driver code (using system time), and the CPU/memory usages were measured using the ‘top’ command.

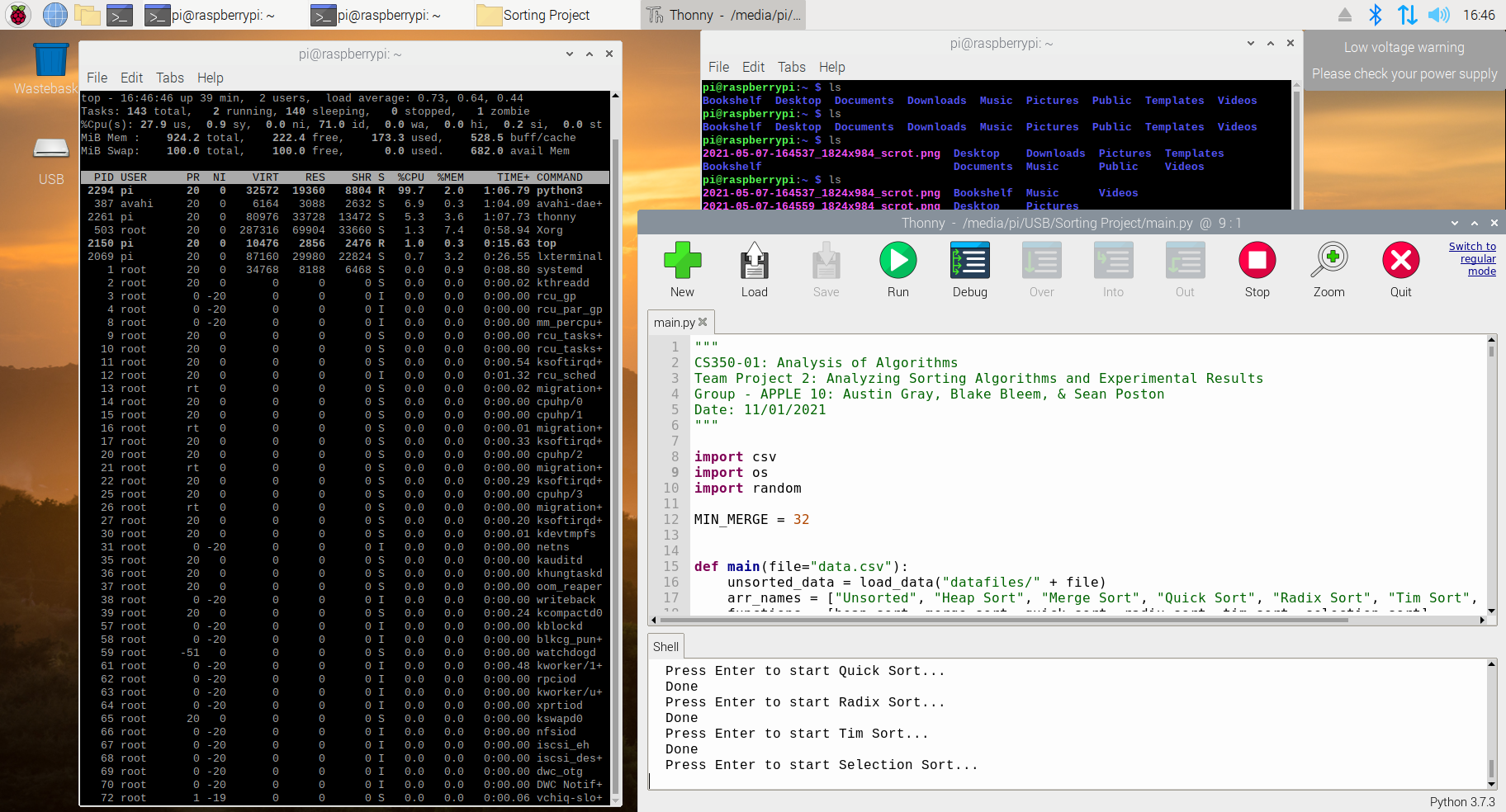
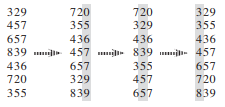


Fig 1: A sample of our original python code running on the Rasberry Pi, along with a view of the ‘top’ process running to the left

In our testing, we measured these statistics for array sizes varying from 10^4 elements long to 10^6 elements long. The larger array sizes became impractical for the testing environment, taking minutes to complete per sort (which overshot the allotted lab time). Below is a description of all of the sorting algorithms we used, as well as a measure of their theoretical time complexities.

## Radix Sort

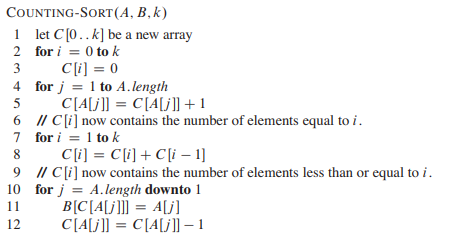
The first sorting algorithm that we tested is radix sort. This sort work by sorting the elements digit-by-digit, using a counting sort to place these numbers in the appropriate spots. By working your way through the digits, you can successfully sort all of the numbers in an array.



An example of how radix sort sorts numbers by digit

Radix sort can be broken down into the following pseudocode (counting sort is the stable sort):





When analyzing the complexity of these two algorithms, radix sort has a complexity of:



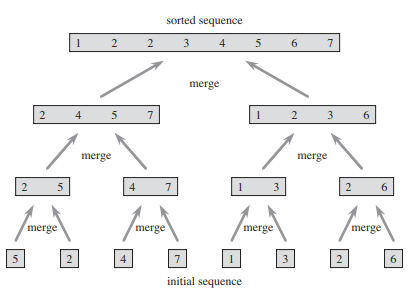
Where ‘d’ is the number of digits in the largest number, ‘k’ is 10, and ‘n’ is the length of the array. When simplifying this equation, we get the result:



Ignoring the constants, this equation simplifies into O(n) time complexity. It is also worth noting that this sort requires extra memory to store a duplicate of the array, where the values are copied into during each pass of the counting sort, along with a ten-element array for the digit counter, giving a total memory complexity of O(n).

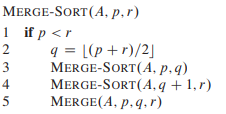
## Merge Sort

The second sort we did is merge sort. This sort works by breaking down and recursively sorting two halves of the original array. Through this divide and conquer method, merge sort can sort in place, meaning that it uses no extra memory (O(1) memory complexity).



An example of how merge sort breaks down then rebuilds the array

This is the pseudocode for merge sort:



Each pass of merge sort happens on an array of size n/2, and each individual merge takes O(n) time to complete, giving us a time complexity of:



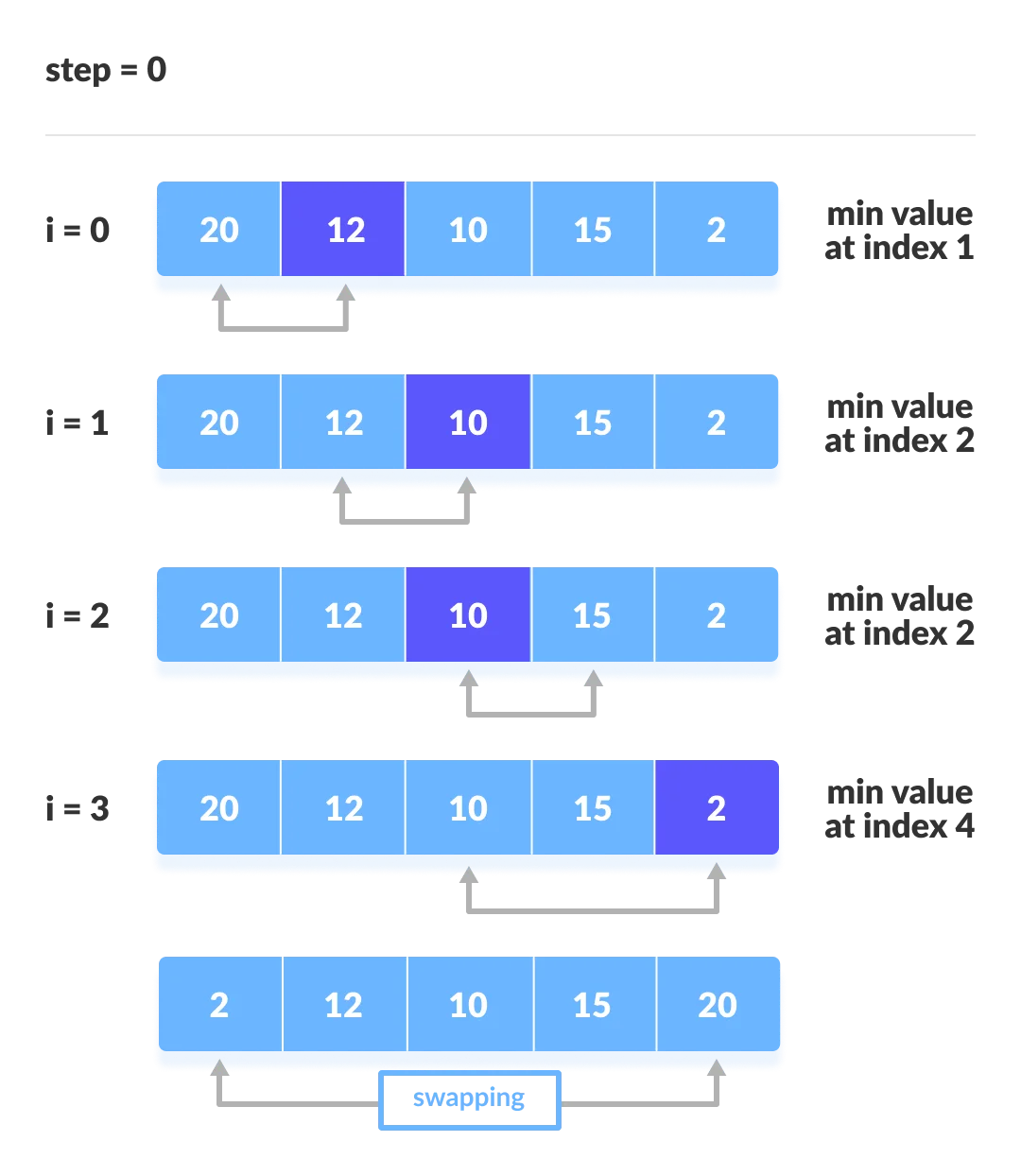
This can be simplified into:



When analyzing the time complexity of this algorithm, we take the largest term, which is O(n \* lg(n)). The one downside to this method is the number of recursive calls that must be made for arrays of extreme size, but in general, this isn’t a problem on modern computers.

## Selection Sort

Next we performed a selection sort. This sort works by managing two subarrays in any given array. The sorted subarray and the remaining subarray which is unsorted. In every iteration, the smallest element from the unsorted subarray is picked and moved to the sorted subarray.



Pseudocode for selection sort:

Selection Sort (arr, n)

1 **for** p=0 to n

2 min = p

3 **for** i = p+1 to n

4 **if** arr[i] < arr[min]

5 min = i

6 **swap**(arr, p, min)

Selection sort always iterates through the entirety of two nested loops on an array size n. This results in a time complexity of:



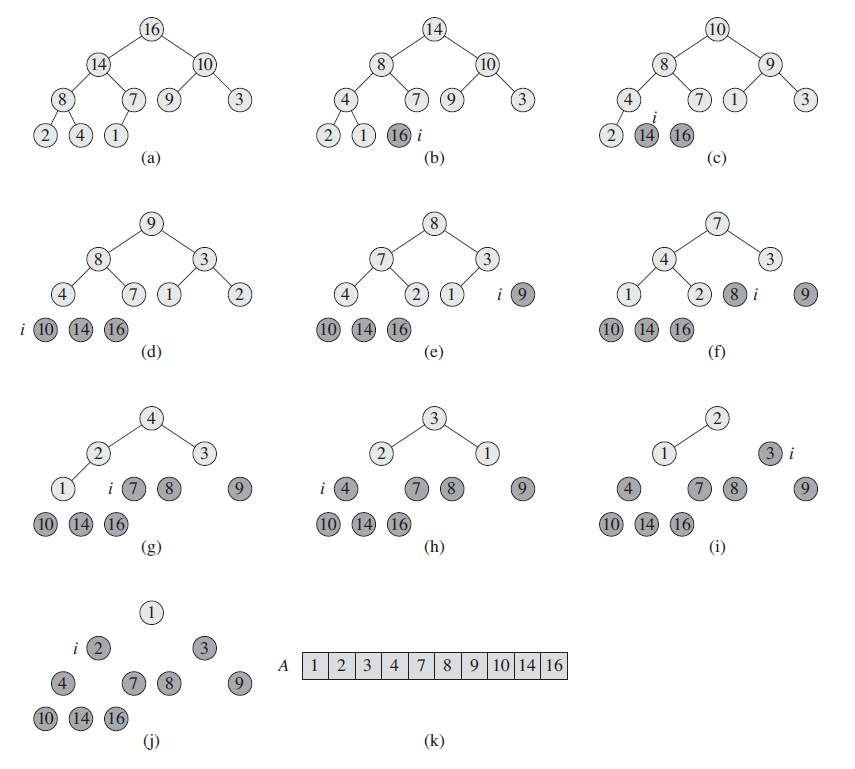
Which can be simplified into:

(n/2 + 1/2)n = n2/2 + n/2

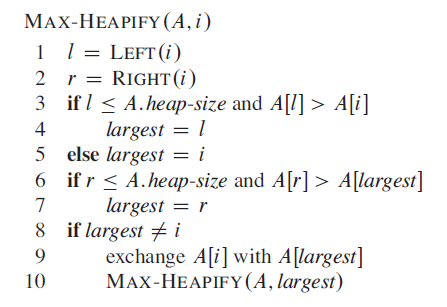
When analyzing the time complexity of this algorithm, we take the largest term, which is O(n2).

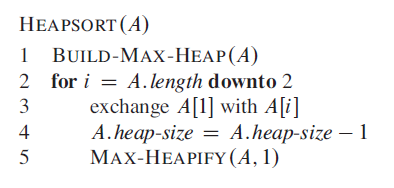
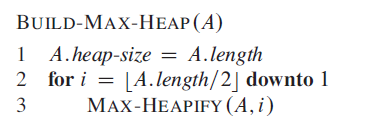
## Heapsort

Heapsort starts by using Build-Max-Heap to build a max-heap on the input array. Since the max of the array is stored at the root, we can put it into its correct final position at the end of the array. Then the algorithm decrements the heap-size and recursively calls max-heapify until the array is sorted.

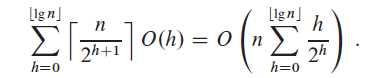


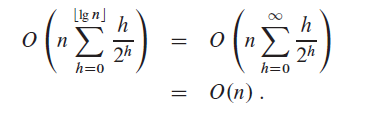
Heapsort can be broken down into the following pseudocode:





T

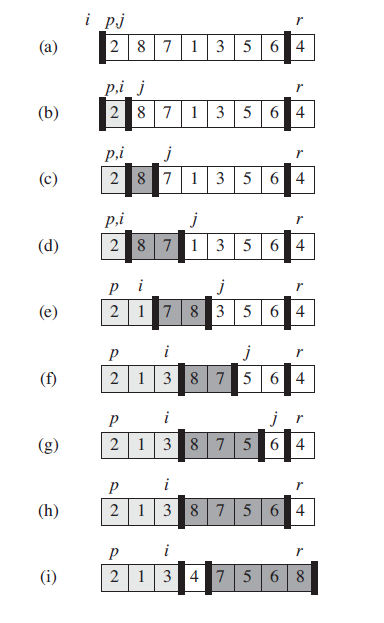




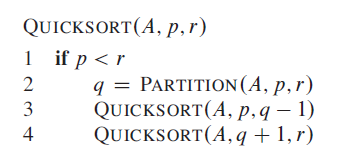
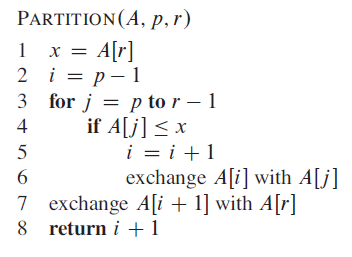
Here we see that building the Max-Heap occurs with a time-complexity of O(n). This results in an overall time-complexity of O(n lg n).

## Quicksort

Quicksort first partitions a given array into two subarrays such that each element of A[p..q-1] is less than or equal to A[q], which is less than or equal to each element of A[q+1..r]. Next it sorts the two subarrays by recursive calls to quicksort. Once the arrays are sorted, nothing else is needed to be done in order to combine them.



Pseudocode for quicksort :





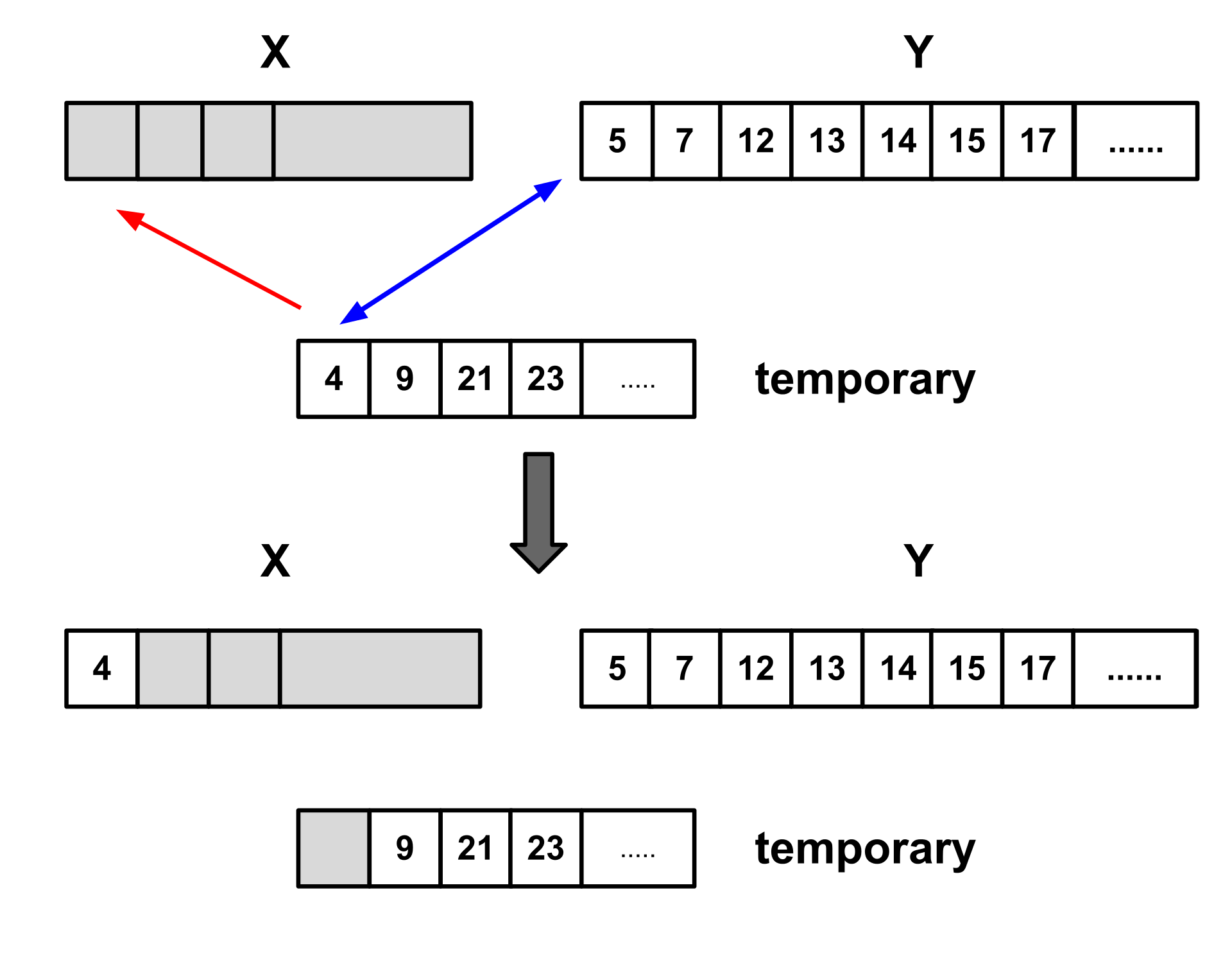
When analyzing the time complexity of this algorithm, we take the largest term, which is O(n2).

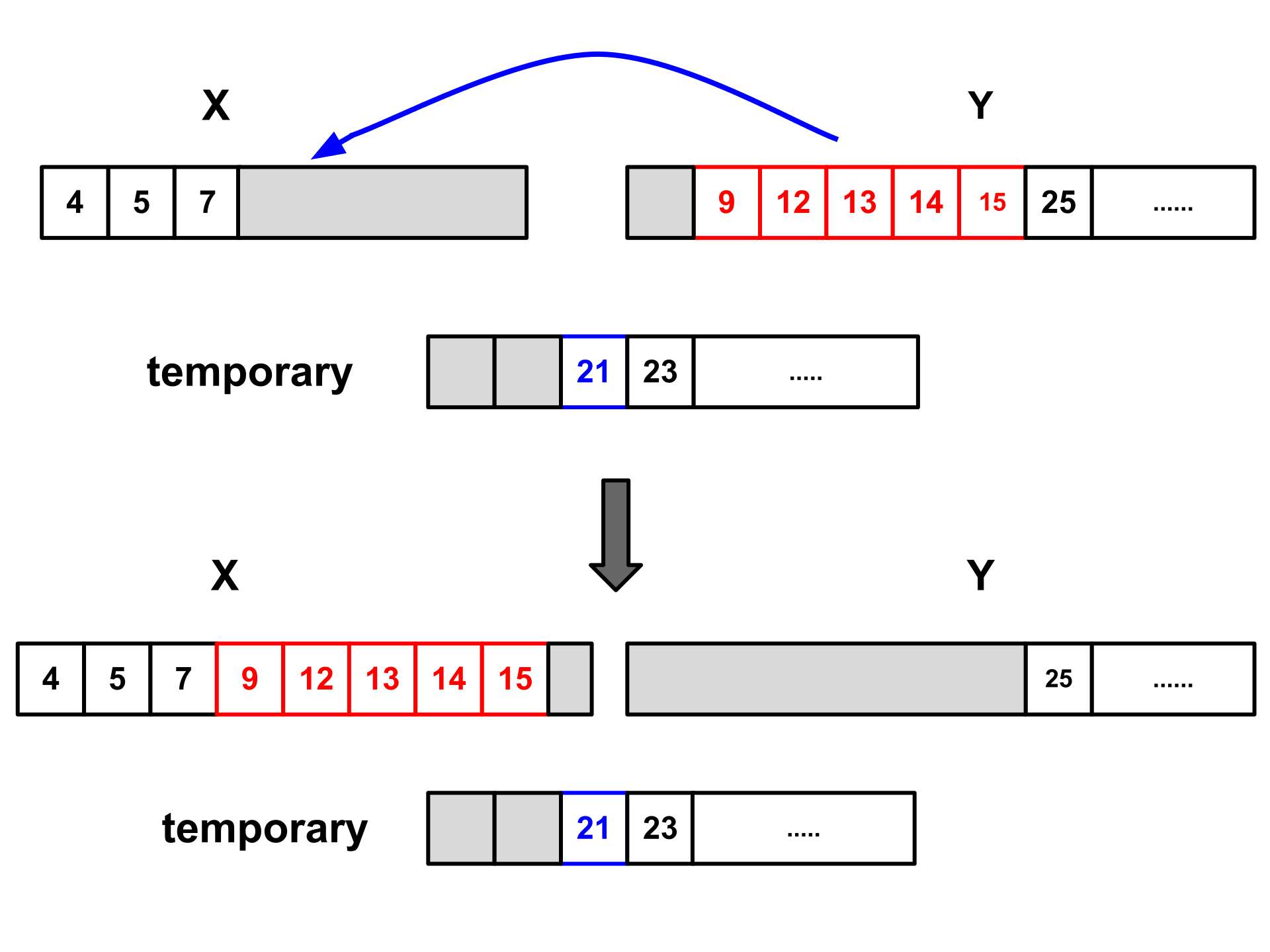


However, quicksort excels in its average case which has a time-complexity of O(n lg n).

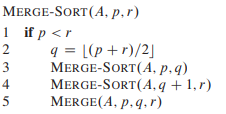
## Timsort

Timsort first divides a given array into blocks know as a “run.” These runs are sorted using insertion sort individually and then merged back together using the combine function used in merge sort.





Pseudocode for timsort:



Insertion-Sort(arr, start, end)

1 **for** i=start to end

2 temp = arr[i]

3 j=i-1

4 **while** j >= start **and** arr[j[ <temp

5 arr[j+1] = arr[j[

6 j++

7 arr[j+1] = temp

Timsort(arr, n)

1 **for** i=0 to n; i+= block\_size

2 j = **min**(i+block\_size, n)

3 **insertion-sort**(arr, i, j)

4 **for** i=block\_size to n; i\*=2

5 **for** l=0 to n; l \*= 2\*i

6 m = l+i-1

7 r = **min**(l+2\*i-1, n-1)

8 **if** m < r

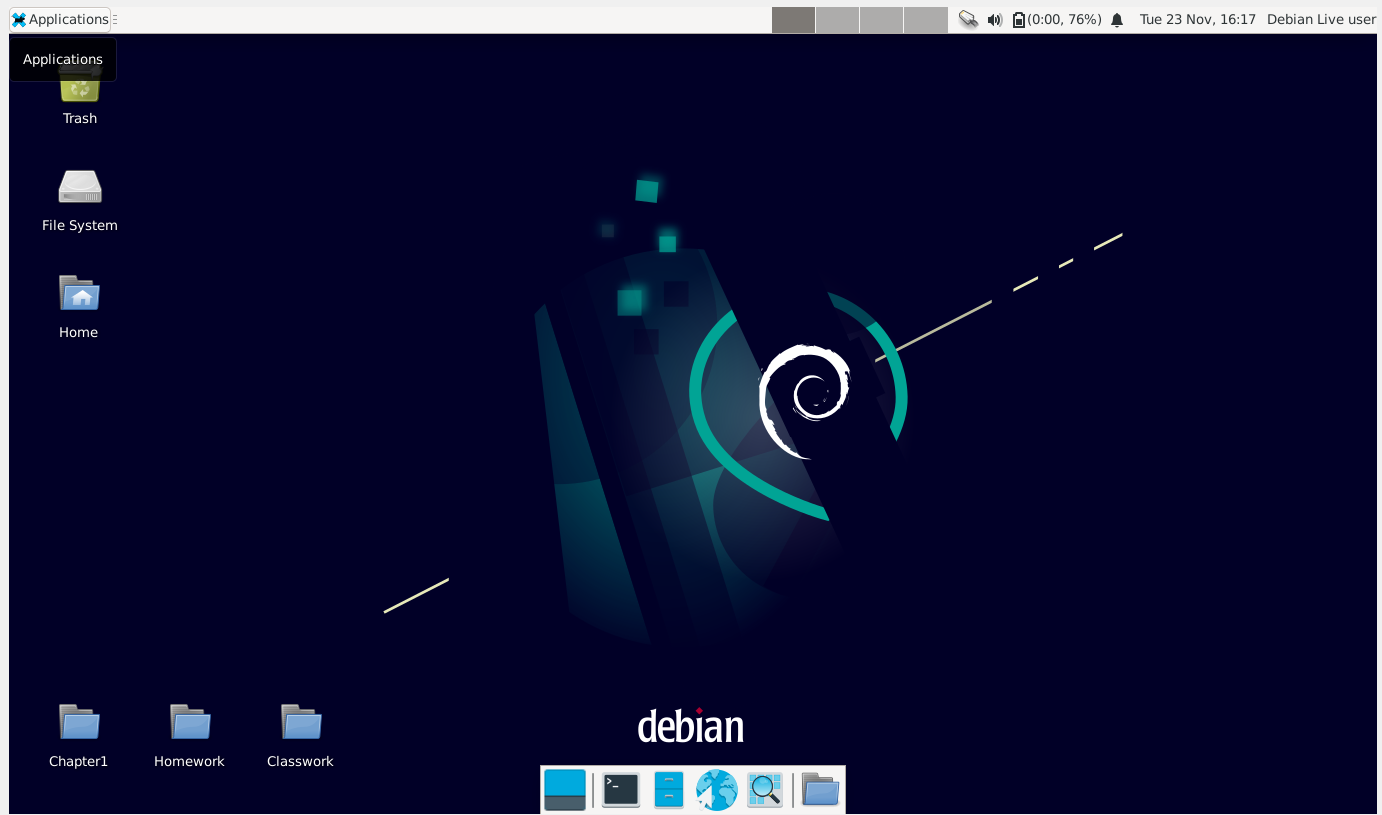
9 **merge**(arr, l, m, r)

We find with Timsort a best case time complexity of O(n), and an average case of O(n lg n).

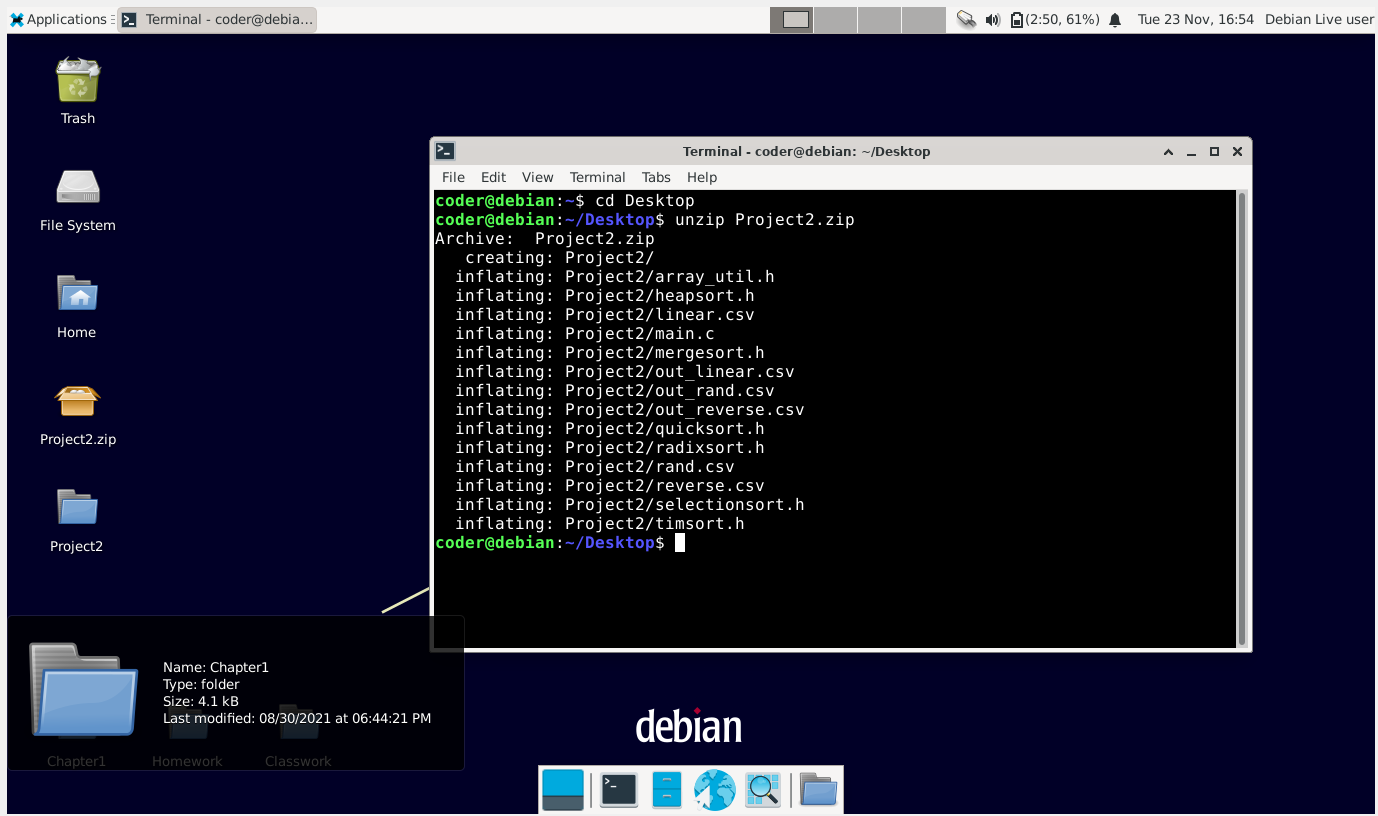
# **Main Context - Running the Algorithms**

To get our results, we put our code on a USB and took it to the lab to run it directly on the provided Raspberry Pi. The screenshots shown here follow the same procedure that we did in the lab, but were taken on a virtual machine that allowed screenshotting.

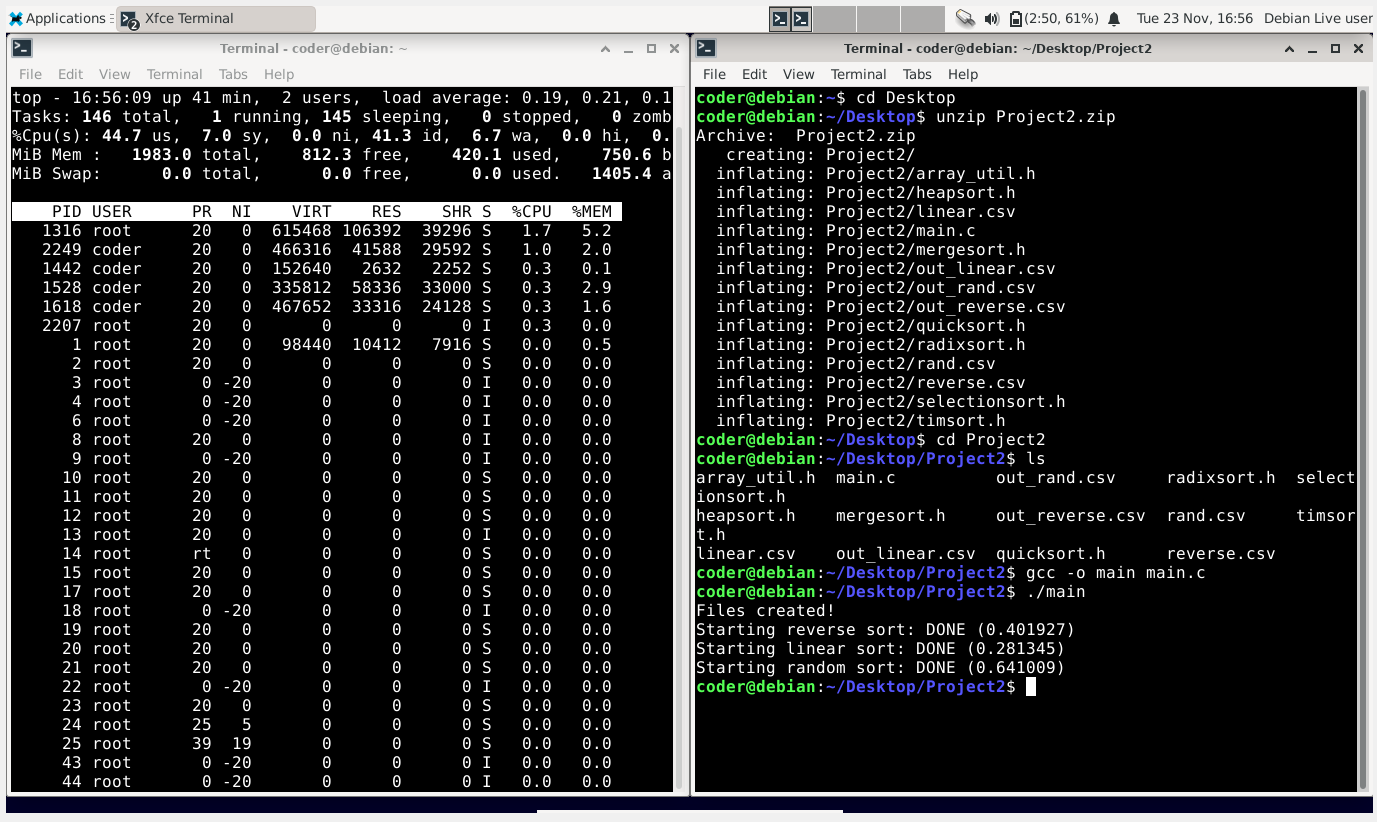
The first part of this lab was to start up the provided Linux environment.



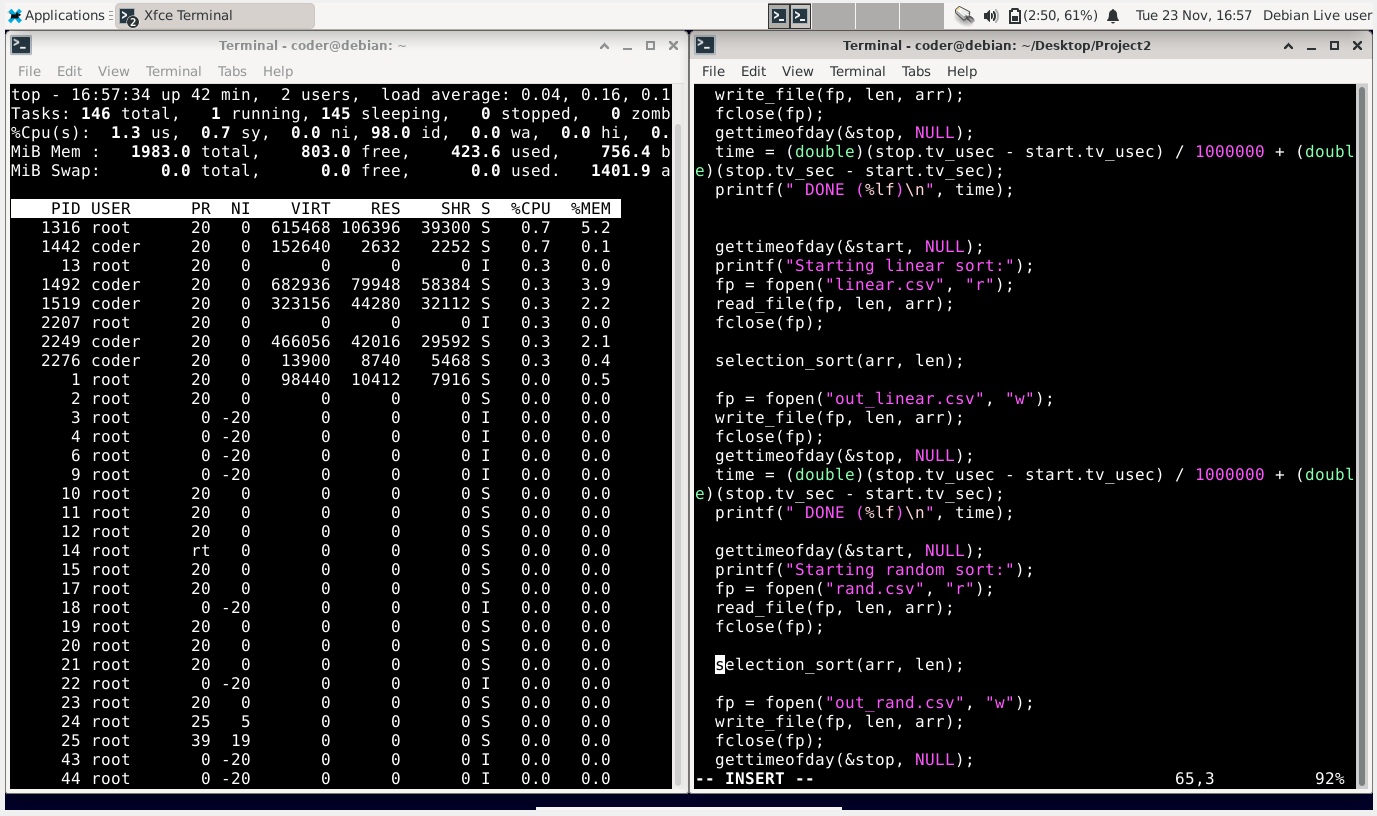
After opening the Linux environment, we could take the files from the USB, move them onto the Rasberry Pi, and extract our zip folder into runnable C files.



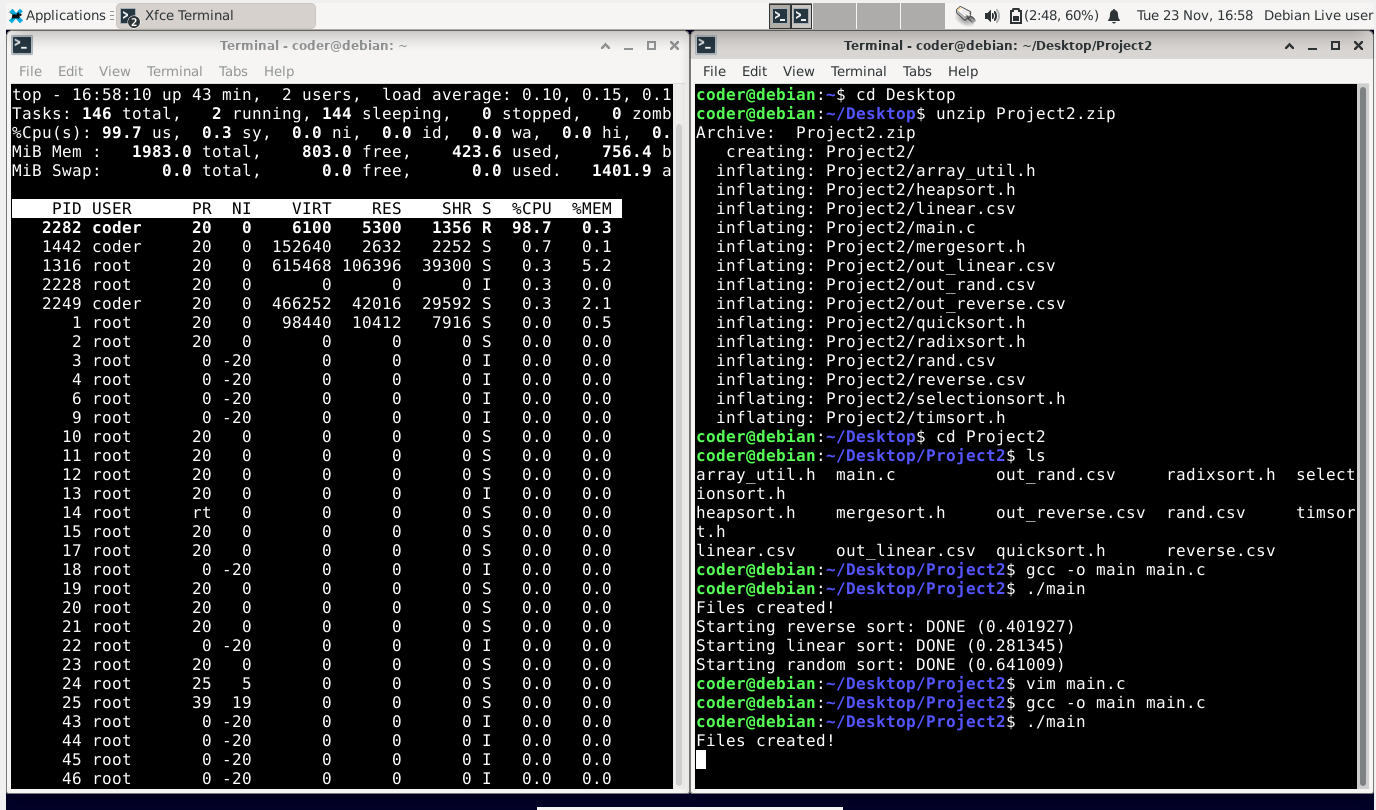
Inside of our project folder, we can compile and test our code to make sure it works. At the same time, we can open a second terminal (on the left) and run the ‘top’ command, which shows all active processes and their CPU/Memory usages.



Between each run of our code, we had to modify the driver code to change the sort types and the dataset sizes. The screenshot (where the cursor is highlighted) shows us changing the current sort into selection sort.



Recompiling the code, we can run it again to see the CPU/Memory usage of the current sort. This process is the first one listed on ‘top’, and we can see that it currently has 98.7% CPU usage and 0.3% memory usage. This process is then repeated for all of the sorting algorithms.



Below shows the individual algorithms, and all of the data that has been collected for each. Each sort has two individuals tables, one indicating the worst-case scenarios for power consumption, CPU percent usage (globally), and percent memory usage (globally). The other table contains information about the times taken to sort arrays of different sizes and layouts.

## Radix Sort

|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.484 | 0.019 | 25.9 | 0.9 |

From this table, we can see that radix sort used essentially the entirety of its CPU (99.9% of the CPU core). In terms of memory usage, from the other in-place sorts, we know that the standard size of the array is about 0.5% of memory. The 0.9% number, in this case, makes sense, since radix sort requires two extra arrays (one the size of n and one the size of k=10). Compared to the Idling amount, radix sort pulled a surprisingly low number of amps.

|  |  |  |  |
| --- | --- | --- | --- |
| RADIX Sort Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 0.051 | 0.550 | 5.954 |
| Already Sorted | 0.047 | 0.519 | 5.687 |
| Random | 0.089 | 0.866 | 8.512 |

When comparing the run times of our radix sort, we can safely conclude that the runtime truly is ~O(n) since when the dataset size magnitude increases by a factor of 10, the runtime increases by close to a factor of 10. One odd thing to note is that the runtime of the randomly created array is slightly larger than the ordered arrays. The main reason for this is probably due to the fact that the random integers could end up with more than six digits per number since they can take any integer value (up to ~2\*10^10).

## Merge Sort

|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.569 | 0.104 | 27.7 | 0.9 |

Like with Radix sort, this sort uses the full CPU core, and surprisingly uses the same amount of memory. This again is probably because merge sort requires a copy of the array that replaces the data in the original array. When comparing this sort to Radix, it uses significantly more power overall. We don’t know exactly why this value is higher, but we suspect it is due to the large number of context switches that happen because of recursion. Below shows the results of our runtimes. These numbers seem to be following the O(n) pattern when comparing their times, but do have a slightly higher value than expected, which supports the O(n \* lg(n)) assumption.

|  |  |  |  |
| --- | --- | --- | --- |
| MERGE Sort Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 0.045 | 0.480 | 5.127 |
| Already Sorted | 0.049 | 0.479 | 5.104 |
| Random | 0.055 | 0.559 | 5.198 |

## Selection Sort

|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.524 | 0.059 | 26.1 | 0 |

The power usage metrics measured in our experiments seemed consistent with its O(1) memory requirements. While it proved to be overall very inefficient in its sorting times and even incapable of sorting large data sets within a reasonable amount of time, it proved to be very memory efficient. Because we could not increase the size of the data in magnitudes of ten without the algorithm needing too much time, we cannot make any kind of claim as to how selection sort efficiency changes with regards to its n. However we can see the variance between the various array order cases. We found relatively low variance between those cases in which the arrays were already sorted versus randomly sorted (pre-sorted array requiring slightly less time). However, there was a noticeable increase in time required for the reverse-order array. These differences are not drastic and therefore reflective of the fact that selection sort is somewhat consistent between its cases. No matter what, the algorithm runs in O(n2) time. This is because of the nested loops which are always iterated through with no escape condition that could possibly shorten this time.

|  |  |  |  |
| --- | --- | --- | --- |
| SELECTION Sort Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 2.475 | too large | too large |
| Already Sorted | 2.376 | too large | too large |
| Random | 2.383 | too large | too large |

## Heapsort

|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.534 | 0.069 | 25.6 | 0.5 |

Heapsort times revealed a contrast to the results of our quicksort. Where quicksort’s efficiency excelled sorting randomly distributed arrays, the heapsort excelled in arrays which were presorted (both reverse and in-order). We also see in Heapsort that increasing the number of elements being sorted resulted in rapid growth of the time it takes to sort those elements. This is consistent with its O(n lg n) time-complexity.

|  |  |  |  |
| --- | --- | --- | --- |
| HEAPSORT Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 0.106 | 0.658 | 7.445 |
| Already Sorted | 0.108 | 0.685 | 7.702 |
| Random | 0.134 | 0.75 | 8.675 |

## Quicksort

|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.51 | 0.045 | 25.6 | 0.5 |

The results of the Quicksort tests show it to be a very interesting case. When looking at its power and memory usage we find it to be relatively low in its demands. Its time results reveal the algorithm to be efficient in the random case for the 10^4 array. However, the reverse-order array results in an extremely drastic increase, and then a drastic increase again to the already sorted array. This confirms the idea that Quicksort is a very context dependent algorithm. While the efficiency of it’s random case makes it a very practical algorithm in most cases. It’s worst-case scenarios reveal it to be less efficient than selection sort. Surprisingly, as we increased our n by factors of 10, the sort times for Quicksort grew too large to be measured.

|  |  |  |  |
| --- | --- | --- | --- |
| QUICKSORT Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 3.85 | too large | too large |
| Already Sorted | 5.98 | too large | too large |
| Random | 0.12 | 0.59 | 5.755 |

## Timsort

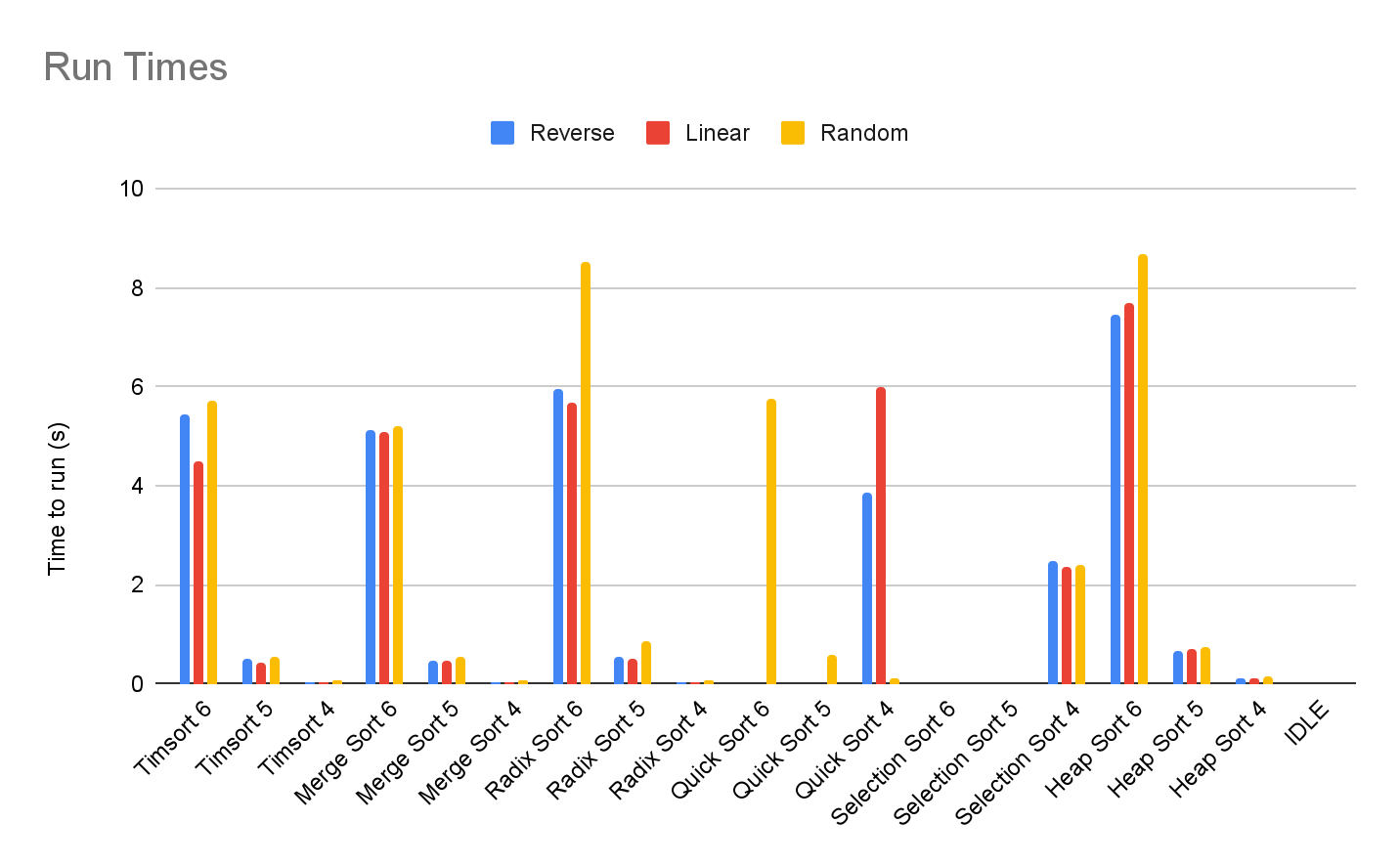
|  |  |  |  |
| --- | --- | --- | --- |
| Amps | Amps - IDLE Amps | CPU Usage (%) | Memory Usage (%) |
| 0.533 | 0.068 | 26.3 | 0.9 |

The results of the Timsort trials, like radix, showed that with each increase in magnitude for the number of elements, a nearly proportional increase in time by approximately the same magnitude could be found. While Timsort’s average and worst case have a time complexity of O(n lg n) its best case is O(n). We also see that it performed remarkably better in pre-sorted cases (both reversed and in-order) than randomly sorted arrays. At its best timsort proved capable of highly efficient results, but we see by its results with the random sorted array, that these benefits rely primarily on the current state of any array being sorted.

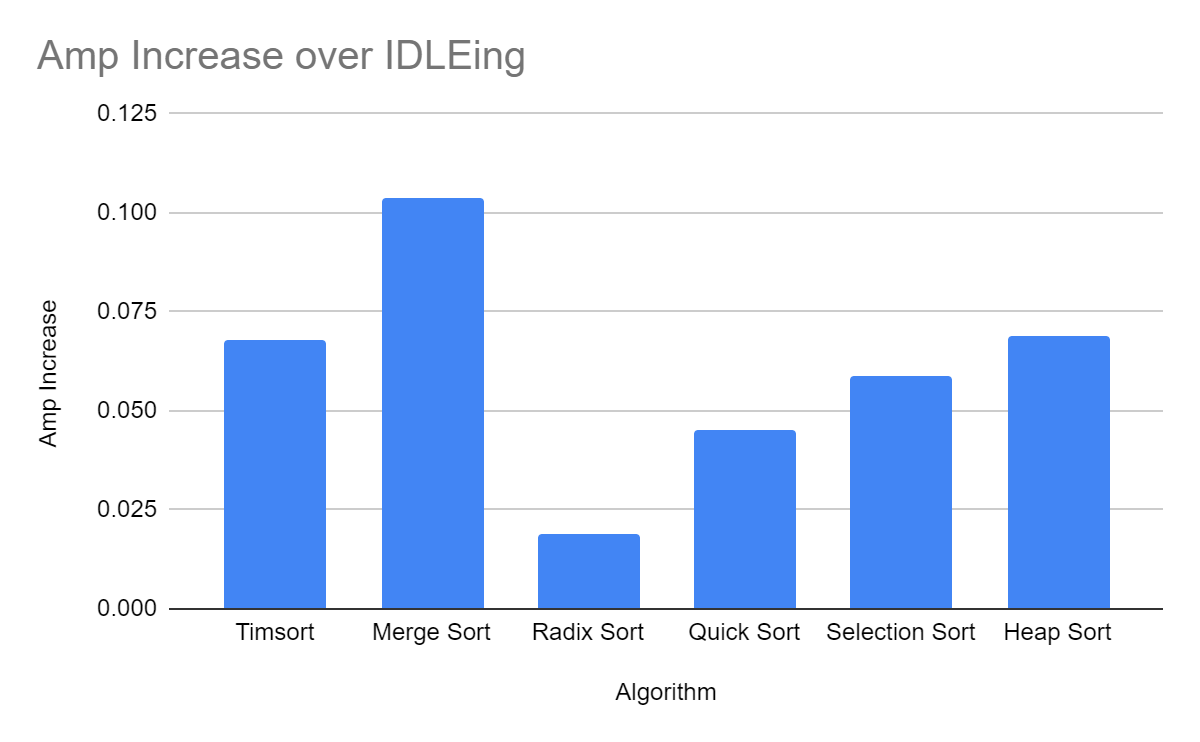
|  |  |  |  |
| --- | --- | --- | --- |
| TIMSORT Times | 10^4 elements | 10^5 elements | 10^6 elements |
| Reversed | 0.049 | 0.512 | 5.427 |
| Already Sorted | 0.041 | 0.421 | 4.507 |
| Random | 0.0553 | 0.546 | 5.73 |

# **Discussion**

Overall, with the data collected, we were able to obtain the following results:



This chart shows the basic runtime comparisons of each of the sorts, with the number being the size of the dataset (ex: Merge Sort 5 = merge sort on a dataset of size 10^5). As seen from this graph the best search (in the random case array) is just barely merge sort, followed by Tim sort and quicksort. Selection sort was by far the worst sort, taking too long to perform any sort of data longer than 10^4 elements. Tim sort performed the best when the array was already sorted (not having to make many comparisons at all), but merge sort still has the best reverse sort time. Overall, it would be safe to conclude that of the six algorithms here, merge sort was the best for the provided cases. It is also worth noting that Radix sort, which has a time complexity of O(n), took the third longest out of all of the sorts, most likely due to the excessive shuffling of data back and forth and between different arrays.



This chart shows the Amps that each sorting algorithm used when compared to the idle amp usage (measured amps - idle amps). Surprisingly, Radix sort (one of the slower algorithms) used the fewest amps while running, and merge sort (the overall fastest algorithm) used the most amps. We can also see this trend for Timsort, which was one of the fastest sorts, but used a higher number of amps. These results imply that there is a bit of a trade off between the amps used and the time taken to complete the sort.

Quick sort (which is on par with merge sort for randomized lists), has the second lowest power consumption overall, making it a bit of an exception to the trend. If you were to only focus on the time taken to sort random arrays (which is how most arrays are usually stored), then it would be fair to say that quicksort is the best of these six algorithms, since it has high speed without using too much power.

# **Conclusion**

This experiment did not bring anything new to the table, and most of the results were predictable. However, the proved information is an invaluable asset to any computer scientist. It’s important to know when to use specific sorting algorithms given a set of data because one’s given resources may vary. One organization may be pressed for time but can afford any amount of memory, but another organization may only have the budget for a certain amount of memory but not need the results as quickly. The electrical usage could also be a factor as well. While on this small scale the differing electrical usage is small, it’s an important consideration when talking on the scale of Amazon’s AWS or Microsoft’s Azure server warehouses. While the difference in our power consumption is negligible, these larger scale operations would have to consider all services being performed and the possible ramifications of not supplying enough power to each machine. All of the gathered information is important in the everyday operations of the world, whether we realize it or not, which is why this experiment is important.

# **Acknowledge**

We would like to acknowledge both Dr. David and Mr. Jiahao for their help throughout the project.

# 

# **References**

Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to algorithms, 3rd Edition*.

# 

# **Appendix - Source Codes**

## Driver Code (main.c)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <sys/time.h>

#include <time.h>

#include "array\_util.h"

#include "mergesort.h"

#include "heapsort.h"

#include "quicksort.h"

#include "radixsort.h"

#include "selectionsort.h"

#include "timsort.h"

// Max tested is 1000000 (10^6)

#define SIZE 1000000

int main(void) {

FILE \*fp;

int i, len;

struct timeval start, stop;

double time;

int arr[SIZE];

len = SIZE;

create\_files(len);

gettimeofday(&start, NULL);

printf("Starting reverse sort:");

fp = fopen("reverse.csv", "r");

read\_file(fp, len, arr);

fclose(fp);

tim\_sort(arr, len); // Change to appropriate sort method

fp = fopen("out\_reverse.csv", "w");

write\_file(fp, len, arr);

fclose(fp);

gettimeofday(&stop, NULL);

time = (double)(stop.tv\_usec - start.tv\_usec) / 1000000 + (double)(stop.tv\_sec - start.tv\_sec);

printf(" DONE (%lf)\n", time);

gettimeofday(&start, NULL);

printf("Starting linear sort:");

fp = fopen("linear.csv", "r");

read\_file(fp, len, arr);

fclose(fp);

tim\_sort(arr, len); // Change to appropriate sort method

fp = fopen("out\_linear.csv", "w");

write\_file(fp, len, arr);

fclose(fp);

gettimeofday(&stop, NULL);

time = (double)(stop.tv\_usec - start.tv\_usec) / 1000000 + (double)(stop.tv\_sec - start.tv\_sec);

printf(" DONE (%lf)\n", time);

gettimeofday(&start, NULL);

printf("Starting random sort:");

fp = fopen("rand.csv", "r");

read\_file(fp, len, arr);

fclose(fp);

tim\_sort(arr, len); // Change to appropriate sort method

fp = fopen("out\_rand.csv", "w");

write\_file(fp, len, arr);

fclose(fp);

gettimeofday(&stop, NULL);

time = (double)(stop.tv\_usec - start.tv\_usec) / 1000000 + (double)(stop.tv\_sec - start.tv\_sec);

printf(" DONE (%lf)\n", time);

}

## Helper Code (array\_util.h)

/\* Suffle the array with random values \*/

int \* shuffle\_arr(int \*arr, int length) {

int i, n;

for(i = 0; i < length; i++) {

arr[i] = (int) rand() % 1000;

}

return arr;

}

/\* Swap two elements in the array

\* this assumes that 'a' and 'b' are

\* both proper indexes in the array \*/

void swap(int \*arr, int a, int b) {

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

/\* Get the maximum value from an array \*/

int max(int \*arr, int length) {

int i, max = arr[0];

for(i = 1; i < length; i++) {

if(arr[i] > max) {

max = arr[i];

}

}

return max;

}

/\* Returns the digit at the 'pow' power,

\* where 'pow' is some multiple of 10

\* (1, 10, 100, etc.) \*/

int get\_digit(int num, int pow) {

return (num / pow) % 10;

}

/\* Prints out the array \*/

void print\_arr(int \*arr, int length) {

int i;

printf("[");

for(i = 0; i < length-1; i++) {

printf("%3d, ", arr[i]);

}

printf("%3d]\n", arr[length-1]);

}

/\* Create a singe file in linear order \*/

void create\_linear\_file(FILE \*fp, int length) {

int i;

for(i = 0; i < length; i++) {

fprintf(fp, "%d,\n", i);

}

}

/\* Create a singe file in reverse order \*/

void create\_reverse\_file(FILE \*fp, int length) {

int i;

for(i = 0; i < length; i++) {

fprintf(fp, "%d,\n", length - i);

}

}

/\* Create a single file and fill it randomly \*/

void create\_random\_file(FILE \*fp, int length) {

int i;

srand(time(NULL));

for(i = 0; i < length; i++) {

fprintf(fp, "%d,\n", rand());

}

}

/\* Makes the set of test files of a given size \*/

void create\_files(int data\_size) {

FILE \*file;

// Make random file

file = fopen("rand.csv", "w");

create\_random\_file(file, data\_size);

fclose(file);

// Make linear file

file = fopen("linear.csv", "w");

create\_linear\_file(file, data\_size);

fclose(file);

// Make reversed file

file = fopen("reverse.csv", "w");

create\_reverse\_file(file, data\_size);

fclose(file);

printf("Files created!\n");

}

/\* Read in a file to a given array/buffer \*/

void read\_file(FILE \*fp, int length, int \*buffer) {

int i, n;

for(i = 0; i < length; i++) {

fscanf(fp, "%d", &n);

while(!feof(fp) && fgetc(fp) != '\n');

buffer[i] = n;

}

}

/\* Write an array to file, in csv format(ish) \*/

void write\_file(FILE \*fp, int length, int \*buffer) {

int i;

// Write the array

for(i = 0; i < length; i++) {

fprintf(fp, "%d,\n", buffer[i]);

}

}

## Radix Sort (radixsort.h)

/\* Counting sort will put elements into buckets based

\* on the value at a specific decimal place (1s, 10s, etc)

\* it will then use these buckets to sort into a second array

\* and copy these elements back into the original array.

\* The result should contain an array sorted by the values

\* in the 'power' decimal place \*/

void counting\_sort(int \*arr, int length, int power) {

// Create empty array and a set of buckets

int temp[length];

int buckets[10] = {0};

int i, j;

// Count how many numbers have a digit in the 'power' place

for(i = 0; i < length; i++) {

buckets[get\_digit(arr[i], power)]++;

}

// Get the running sum of all the buckets

for(i = 1; i < 10; i++) {

buckets[i] += buckets[i-1];

}

// Do the counting sort (working backwards)

for(i = length-1; i >= 0; i--) {

// Get the current digit

j = get\_digit(arr[i], power);

// Decrement the bucket value

buckets[j]--;

// Insert the array element at the bucket value (in the temp array)

temp[buckets[j]] = arr[i];

}

// Copy the temp array to the original array

for(i = 0; i < length; i++) {

arr[i] = temp[i];

}

}

/\* Radix sort groups all elements into

\* buckets based on their decimal position.

\* these buckets are then sorted and and

\* then reconstructed back into the array \*/

void radix\_sort(int \*arr, int length) {

// Create temp variable

int power;

// Get the maximum element (to find the max number of digits)

int mx = max(arr, length);

// Do a counting sort for each digit (1s, 10s, 100s, etc.)

for(power = 1; mx / power > 0; power \*= 10) {

counting\_sort(arr, length, power);

}

}

## Merge Sort (mergesort.h)

/\* Helper method to merge the two arrays \*/

void merge(int \*arr, int l, int m, int r) {

// Return if length is 1 or less

if(r - l == 0) {

return;

}

// Create a new temp array, and three pointers (left, right, and temp array)

int temp[r-l+1];

int lp = l, rp = m+1, tp = 0;

// While both sides have elements, shuffle them together

while(lp <= m && rp <= r) {

if(arr[lp] > arr[rp]) {

temp[tp++] = arr[rp++];

}

else {

temp[tp++] = arr[lp++];

}

}

// Empty out the left array

while(lp <= m) {

temp[tp++] = arr[lp++];

}

// Empty out the right array

while(rp <= r) {

temp[tp++] = arr[rp++];

}

// Copy the temp array back into the original

tp = 0;

while(l <= r) {

arr[l++] = temp[tp++];

}

}

/\* Merge sort recursively breaks the array into two

\* smaller arrays, then sorts those individually

\* each smaller array is then shuffled together into

\* a sorted larger array \*/

void merge\_sort(int \*arr, int start, int end) {

// If array is size 1 or 0, just return it

if(end - start < 1) {

return;

}

// Find the mid point

int mid = start + ((end - start) / 2);

merge\_sort(arr, start, mid);

merge\_sort(arr, mid+1, end);

merge(arr, start, mid, end);

}

## Selection Sort (selectionsort.h)

/\* Selection sort loops through the

\* array and puts the smallest element

\* at the front \*/

void selection\_sort(int \*arr, int length) {

// Create two pointers and a minimum

int i, p, min;

for(p = 0; p < length; p++) {

min = p;

// Find the smallest index

for(i = p+1; i < length; i++) {

if(arr[i] < arr[min]) {

min = i;

}

}

// Swap the min and the p values

swap(arr, p, min);

}

}

## Heap Sort (heapsort.h)

/\* The heapify helper method, that recursively

\* makes a max heap from the sub tree \*/

void heapify(int \*arr, int length, int node) {

// Find left and right

int left = 2 \* node + 1;

int right = 2 \* node + 2;

// Create holder for largest node

int largest = node;

// See if left node is larger

if(left < length && arr[left] > arr[largest]) {

largest = left;

}

// See if right node is larger

if(right < length && arr[right] > arr[largest]) {

largest = right;

}

// Swap if the largest isn't aleary on top

if(largest != node) {

swap(arr, largest, node);

// Reheapify the child tree

heapify(arr, length, largest);

}

}

/\* Heap sort turns the array into a binary tree

\* and then sorts that tree \*/

void heap\_sort(int \*arr, int length) {

// Create temp variable

int i;

// Build the heap tree

for(i = length / 2 - 1; i >= 0; i--) {

heapify(arr, length, i);

}

// Slowly take elements from the heap and sort them

for(i = length - 1; i > 0; i--) {

// Move the root to the end

swap(arr, 0, i);

// Heapify the new heap

heapify(arr, i, 0);

}

}

## Quick Sort (quicksort.h)

/\* The partition helper function uses the last

\* element as a pivot, and puts all elements

\* smaller on the left, and all elements greater

\* on the right, it then returns the index of the

\* pivot \*/

int partition(int \*arr, int start, int end) {

// Get the pivot and create a pointer

int pivot = arr[end];

int i = start - 1;

/\* Loop through the sub array and move all small elements

\* to the left side of the pivot \*/

int j;

for(j = start; j < end; j++) {

if(arr[j] < pivot) {

swap(arr, ++i, j);

}

}

// Move the pivot to the center

swap(arr, i+1, end);

// Return the index of the pivot

return i+1;

}

/\* Quick sort is very similar to merge sort

\* except it uses partitions instead of cutting the

\* array in half \*/

void quick\_sort(int \*arr, int start, int end) {

// Break early if the array size is 1

if(end - start <= 0) {

return;

}

int partition\_index = partition(arr, start, end);

quick\_sort(arr, start, partition\_index-1);

quick\_sort(arr, partition\_index+1, end);

}

## Tim Sort (timsort.h)

/\* The block size is the size of each block

\* of data that gets insertion sorted \*/

#define BLOCK\_SIZE 32

/\* Find the minimum of two integers \*/

int min(int a, int b) {

return a < b ? a : b;

}

/\* TimSort also used an insertion sort \*/

void insertion\_sort(int \*arr, int start, int end) {

// Create temp variables

int i, j, temp;

// Loop through array sorting from left to right

for(i = start + 1; i < end; i++) {

/\* Get the current element and set a

\* second pointer to start backtracking

\* the array \*/

temp = arr[i];

j = i-1;

/\* Scan backwards and push the elements

\* forward one index \*/

while(j >= start && arr[j] > temp) {

arr[j+1] = arr[j];

j--;

}

/\* Put the current element back into the array \*/

arr[j+1] = temp;

}

}

/\* Tim Sort breaks the array into

\* blocks of a certain size, and then

\* merges these blocks together \*/

void tim\_sort(int \*arr, int length) {

// Create temp variables

int i, j, l, m, r;

// Firstly, sort each block with insertion sort

for(i = 0; i < length; i+= BLOCK\_SIZE) {

// j is either the end of the array or the end of the block

j = min(i + BLOCK\_SIZE, length);

insertion\_sort(arr, i, j);

}

// Start merging the blocks (each time the block size doubles)

for(i = BLOCK\_SIZE; i < length; i \*= 2) {

// Merge all individual blocks

for(l = 0; l < length; l += 2\*i) {

// Find mid and right size for merge

m = l + i - 1;

r = min((l + 2\*i - 1), (length - 1));

// Merge the two blocks (if necessary)

if(m < r) {

merge(arr, l, m, r);

}

}

}

}