University of Rhode Island ~ CSC 212:

Sorting Algorithms

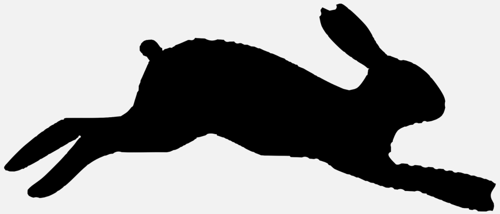
By: Joseph Yanez, Matthew Boekamp, Sahil Chadha, & Richard Buckley

Sorting Matters?

- Yes, sorting matters. Advanced sorting algorithms are detrimental to the field of computer science, as they allow us to process data, quickly and efficiently.
- What kind of services require efficient sorting? This one is easy **databases.** Databases are arguably one of the most important data storage solution in computer science.
- The time it takes for a user to find a particular product, can create or discourage a sale. For example, Walmart found that for every 1 second improvement in page load time, conversions increased by 2% (Cloudflare).

Goals of Sorting Algorithms

FAST & EFFICIENT



1. Space Complexity - algorithm should be efficient with storage
2. Stability - algorithm should not modify order of elements with the same keys
3. Simplicity - algorithm should be easy to understand & implement
4. Scalability - algorithm should work for both large and small data sets
5. Adaptability - algorithm should be efficient for all scenarios

Comparing the basic sorts

	Best Case	Worst Case	Space
Selection	●●○○○ $O(n^2)$	●●○○○ $O(n^2)$	●●●●●
Insertion	●●●●● $O(n)$	●●○○○ $O(n^2)$	●●●●●
Bubble	●●○○○ $O(n^2)$	●●○○○ $O(n^2)$	●●●●●
	In a fully sorted array, insertion sort performs the best	Compared to a worst case of n(log n), these are not as performant	These are all in-place sorting algorithms

Sorting Strategy

Strategy

01 **Comparison**

Compares values of neighbors

02 **Non-Comparison**

No comparisons needed

Sorting Stability

Stable?

TRUE

Keys will always remain in the same order if their calculated value is equivalent…the order of the inputs will not change unless they are deemed out of order.

FALSE

Keys will not always remain in the same order. If you had an array with three 1s and each had a unique id, the inputs and outputs of the ids may not remain the same.

Our Sorts

For this project, we implemented 4 different sorting algorithms, and used, both a visualizer, as well as a timing program to analyze how these sorting algorithms functioned under different input conditions.

INSERTION SORT

A basic sorting algorithm that is easy to implement, but does not have the best performance.

QUICK SORT

A more advanced sorting algorithm that is inplace, and has an average time complexity of O(n log n). This is not stable.

MERGE SORT

A more advanced sorting algorithm that is inplace, and has an average time complexity of O(n log n). This algorithm is stable, and has a better worst case time complexity.

RADIX SORT

Unlike other algorithms here, in Radix sort, no comparisons are made between elements. Radix sort is not stable by nature, but can be made stable.

Our Sorts: Insertion Sort

Insertion Sort

Best Case

- $O(n)$, this case only occurs when the array is fully sorted.

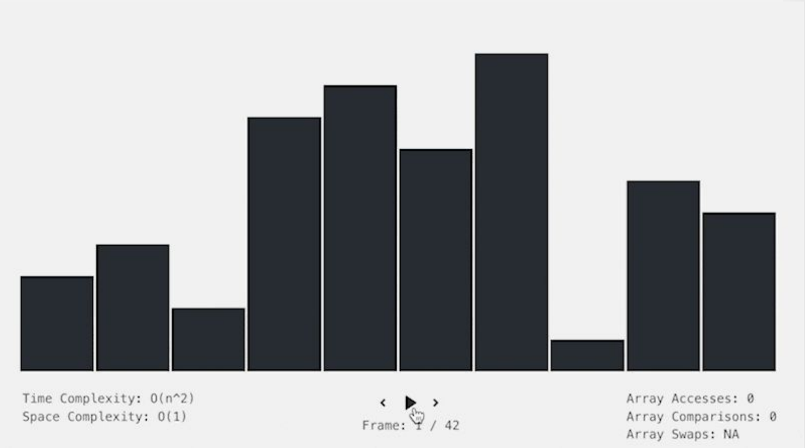
Average Case

- $O(n^2)$, to fully sort the array, insertion sort compares every element.

Worst Case

- $O(n^2)$, this is generally too slow for a sorting algorithm, and as such, undesired.

Our Sorts: Insertion Sort



Insertion Sort

```
void insertionSort(std::vector <int> &array, int sizeN){
    // define our pointers
    int i, j, k = 0;

    // loop through array
    for (i = 1 ; i < sizeN ; i++){
        j = i;

        // store temp
        k = array[i];

        // move element until it is in order
        while (j > 0 and k < arr[j - 1]){
            array[j] = array[j - 1];
            j--;
        }

        // restore temp
        array[j] = k;
    }
}
```

Our Sorts: Quick Sort

Quick Sort

Best Case

- O(n log n), the best any comparison based algorithm can do.

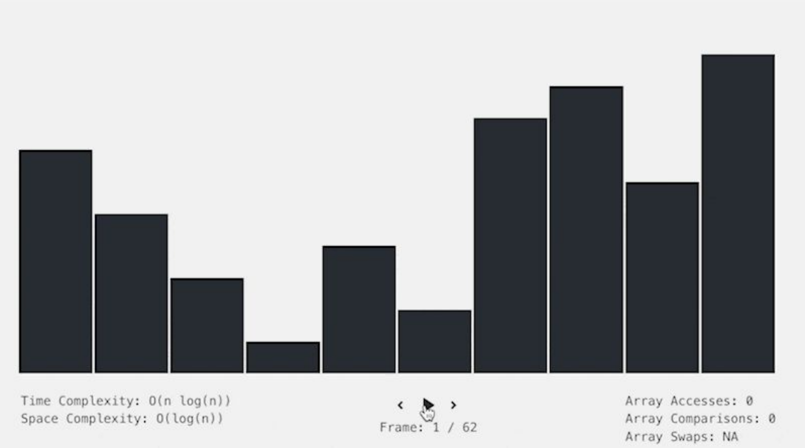
Average Case

- O(n log n), generally QuickSort finishes in O(n log n) time.

Worst Case

- O(n^2), this occurs when the array is already sorted. Shuffle to counteract

Our Sorts: Quick Sort



Quick Sort

```
int quickSort(std::vector <int> *quick, int left, int right) {
        int i = left;
        int j = right + 1;

        while (i < j) {

            // while quick[i] < pivot, increase i
            while (quick[++i] < quick[left])
                if (i == right) break;

            // while quick[i] > pivot, decrease j
            while (quick[left] < quick[--j])

                if (j == left) break;

            // if i and j cross exit theloop
            if(i >= j) break;

            // swap quick[i], quick[j]
            std::swap(quick[i], quick[j]);
        }

        // swap the pivot with quick[j]
        std::swap(quick[left], quick[j]);

        //return pivot's position
        return j;
}

// recursively calls quicksort
void r_quicksort(std::vector<int> *quick, int left, int right){
    if (right <= left) return;

    int pivot = quickSort(quick, left, right);

    r_quicksort(quick, left, pivot - 1);
    r_quicksort(quick, pivot + 1, right);
}
```

Our Sorts: Merge Sort

Merge Sort

Best Case

- O(n log n), the best any comparison based algorithm can do.

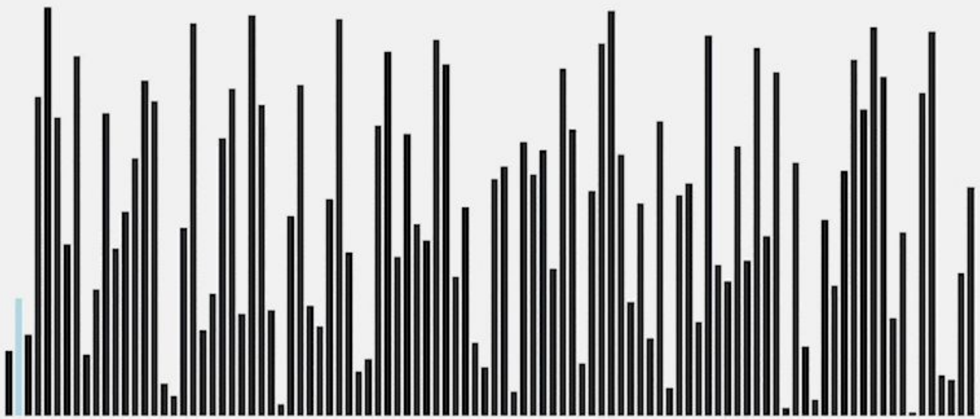
Average Case

- O(n log n), the best any comparison based algorithm can do.

Worst Case

- O(n log n), the best any comparison based algorithm can do.

Our Sorts: Merge Sort



Time Complexity: O(n log(n))
Space Complexity: O(n)

‹ ›
Frame: 1 / 873

Array Accesses: 3
Array Comparisons: 1
Array Swaps: NA

Quick Sort

```
void Sorts::merge(std::vector<int>& arr, int left, int mid, int right) {
    int i = left;       // Index for left subarray
    int j = mid + 1;    // Index for right subarray
    int k = 0;          // Index for temporary array

    std::vector<int> temp(right - left + 1); // Temporary array

    // Merge the two subarrays into the temporary array
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    // Copy the remaining elements of the left subarray, if any
    while (i <= mid)
        temp[k++] = arr[i++];

    // Copy the remaining elements of the right subarray, if any
    while (j <= right)
        temp[k++] = arr[j++];

    // Copy the merged elements back to the original array
    for (i = left, k = 0; i <= right; i++, k++)
        arr[i] = temp[k];
}

 void mergeSort(std::vector<int>& arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2; // Calculate the mid-point

            // Recursively divide the array into two halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }
```

Our Sorts: Radix Sort

Radix Sort

Best Case

O(kn), where k is the number of bytes of the largest element in the array

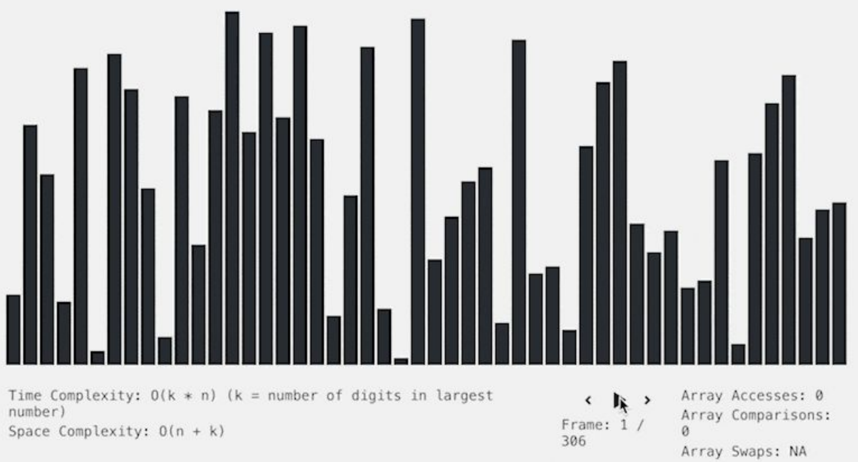
Average Case

O(kn), where k is the number of bytes of the largest element in the array

Worst Case

O(kn), where k is the number of bytes of the largest element in the array

Our Sorts: Radix Sort



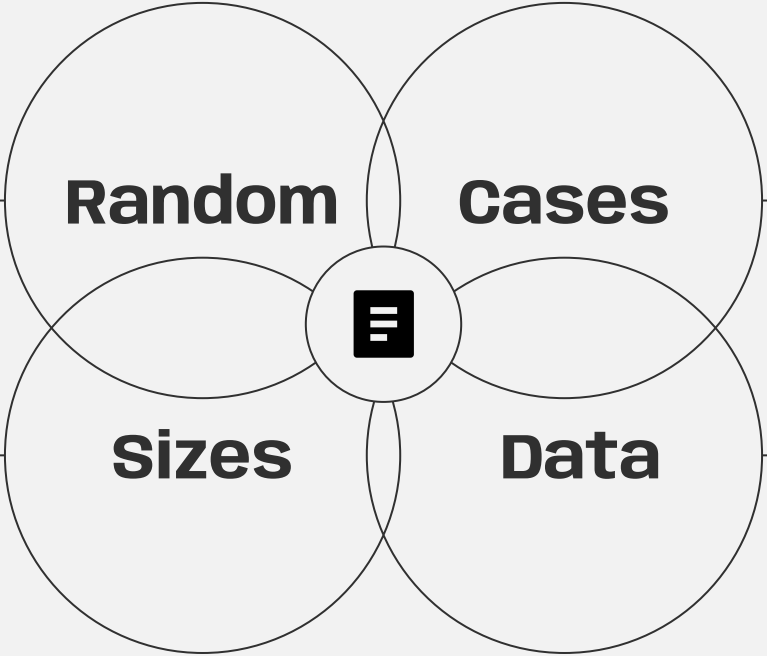
Quick Sort

```
void radixSort(std::vector<int> &array, int numDigits) {

    for (int i = 0 ; i < numDigits ; i++) {

        // create empty containers that we will organize into
        std::vector< std::vector<int> > containers(10);

        // for num in arr
        for (int v : array)
            containers[(v / (int)std::pow(10, i)) % 10].push_back(v);

        // clear out arr
        array.clear();

        // Rinse & Repeat
        for (const std::vector<int> &c : containers)
            array.insert(arr.end(), c.begin(), c.end());

    }

}
```

Timing Criteria.

Arrays of randomly generated elements

Both large and small arrays



Test all special cases that slow down algorithm

We want to test different distributions (partially sorted, fully sorted)

Timing (How we did it).

How we did it.

- First of all, we would like to credit the timing logic to Stack Overflow – they have allowed for us to be as precise as possible.
- We ran each sorting algorithm x times, with the results linked on this google spreadsheet.
- We used a Macbook Pro, using a python script to call the c++ program, with a given text file, and then we sent the results up to a google sheet.

```
#include <chrono>
#include <iostream>

int main() {
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    auto t1 = high_resolution_clock::now();
    long_operation();
    auto t2 = high_resolution_clock::now();

    /* Getting number of milliseconds as an integer. */
    auto ms_int = duration_cast<milliseconds>(t2 - t1);

    /* Getting number of milliseconds as a double. */
    duration<double, std::milli> ms_double = t2 - t1;

    std::cout << ms_int.count() << "ms\n";
    std::cout << ms_double.count() << "ms\n";
    return 0;
}
```

Timing (How we did it).

How we did it.

- Now that we have our resultant `.csv` files, we can proceed with scoring each of the algorithms.
- With over 1000 tests per edge case, we must find a way to efficiently score each algorithm, so we decided to use z-scores.
- Z-scores are a statistical measure of how far something deviates from the mean (which would have a z-score of 0), and in this case, we are looking for the lowest z-score possible.

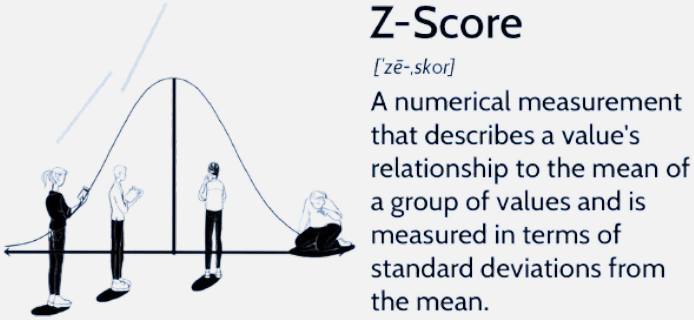


Image from investopedia

Timing (How we did it).

5%

Python

Python generates test cases, and stores them in .csv

90 %

TimeIt

We call our C++ timeit program to time each of the arrays being sorted.

100%

Pandas

We use pandas & google sheets to statistically analyze data

Timing Results.

Ranking our Sorting Algorithms on speed.

Algorithm	Avg. Speed
Radix Sort	285,328 ms
Quick Sort	511,278 ms
Merge Sort	2,097,358 ms
Insertion Sort	27,679,716 ms

These results are from randomly generated samples. See results here.

Timing Results.

How we did it.

- We did this math in a Google Sheet with formulas, and we were able to find some surprising outcomes.
- We discovered that radix sort was the fastest, even when we added outliers to ensure that it was slowed down.
- The fastest sort that we had tested was the radix sort. Next came merge, then quick, and finally insertion sort.

sort_name	z-score (avg)
radix_sort	-0.59
merge_sort	-0.18
quick_sort	-0.12
insertion_sort	0.89

Thank You

Thank you for your time throughout our presentation.

Thank you to SlidesGo! for this amazing presentation template.

Any Questions? 👏

A quick note: please don’t dock points from us for making an additional visualizer with react, though that seems like a lot more work (it was), I mainly did it for my portfolio, as I want to get an internship next year (fingers crossed), and it was a cool project
- Rich