Sorting Algorithms

SEDDIK Mohamed Taki Eddine

Batna 2 UNIVERSITY
Faculty of Mathematics and Computer Science
Department of Computer Science

November 9, 2023

Sorting Algorithms Outline I

Introduction to Sorting Algorithms

What is Sorting?

Sorting refers to the process of arranging data (often numbers or words) in a particular sequence or order, either in ascending or descending form.

Why is it Important?

- It's a fundamental operation in computer science.
- Used in numerous applications, like database algorithms, for better search performance.
- Provides a foundation to understand algorithm efficiency and design.

Types of Sorting Algorithms

Various Algorithms

There are numerous ways to sort data, each with its advantages and challenges:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- ... and many more.

Types of Sorting Algorithms

Factors to Consider

- Time complexity
- Space complexity
- Stability
- Internal vs External sorting

Goals for this Chapter

What we aim to achieve

- Understand the mechanics of each sorting algorithm.
- Compare and contrast the performance of different algorithms.
- Choose the right algorithm for the right job.
- Implement some of the primary sorting algorithms.

Selection Sort

Steps to Perform Selection Sort

1. Start with the first element as the minimum.
2. Traverse through the list to find the minimum element.
3. Swap the first element with the found minimum.
4. Take the next element as minimum and repeat.

Example

Sorting an Array

$$
\begin{aligned}
\text{Original Array:} \quad & arr = [64, 34, 25, 12, 22, 11, 90] \\
\text{1st pass:} \quad & arr = [11, 34, 25, 12, 22, 64, 90] \\
\text{2nd pass:} \quad & arr = [11, 12, 25, 34, 22, 64, 90] \\
\text{3rd pass:} \quad & arr = [11, 12, 25, 22, 34, 64, 90] \\
\text{Sorted Array:} \quad & arr = [11, 12, 25, 22, 34, 64, 90]
\end{aligned}
$$

Time Complexity

Understanding the Complexity

For every element, we compare:

- 1st element with $n-1$ elements.
- 2nd element with $n-2$ comparisons.
- And so on...

This forms an arithmetic series:

$$(n-1)+(n-2)+\ldots+3+2+1$$

Time Complexity (cont'd)

Calculating the Sum

- The sum is:

$$\frac{(n-1)(n)}{2} = \frac{n^2 - n}{2}$$

- Time complexity is dominated by the highest order term: $O(n^2)$

Algorithm in C

```c
#include<stdio.h>

void selectionSort(int arr[], int n) {
    for(int i = 0; i < n-1; i++) {
        int min_idx = i;
        for(int j = i+1; j < n; j++) {
            if(arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    selectionSort(arr, n);
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Conclusion

Summary

- Selection Sort has a time complexity of $O(n^2)$.
- Simple, but not always efficient.
- Suitable for smaller lists or specific use-cases.

Bubble Sort Algorithm

Steps to Perform Bubble Sort

1. Start from the first element.
2. Compare the current element with the next element.
3. If current element > next element, swap them.
4. Move to the next element and repeat until the end of the array.
5. Repeat the process for each element.

Example

Sorting an Array

$$
\begin{aligned}
\text{Original Array:} \quad & arr = [64, 34, 25, 12, 22, 11, 90] \\
\text{1st pass:} \quad & arr = [34, 25, 12, 22, 11, 64, 90] \\
\text{2nd pass:} \quad & arr = [25, 12, 22, 11, 34, 64, 90] \\
\text{3rd pass:} \quad & arr = [12, 22, 11, 25, 34, 64, 90] \\
\text{Sorted Array:} \quad & arr = [11, 12, 22, 25, 34, 64, 90]
\end{aligned}
$$

Time Complexity

Understanding the Complexity

For every element, we compare:

- 1st pass requires $n - 1$ comparisons.
- 2nd pass requires $n - 2$ comparisons.
- And so on...

This forms an arithmetic series:

$$(n-1) + (n-2) + \ldots + 3 + 2 + 1$$

Time Complexity (cont'd)

Calculating the Sum

- The sum is:

$$\frac{(n-1)(n)}{2} = \frac{n^2 - n}{2}$$

- Time complexity is dominated by the highest order term: $O(n^2)$

Algorithm in C

```c
#include<stdio.h>

void bubbleSort(int arr[], int n) {
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Conclusion

Summary

- Bubble Sort has a time complexity of $O(n^2)$.
- It's intuitive but often less efficient than other sorting algorithms.
- Suitable for educational purposes and smaller lists.

Introduction to Insertion Sort

What is Insertion Sort?

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is similar to the way we sort playing cards in our hands.

Characteristics of Insertion Sort

- Stable sort
- Adaptive: becomes faster when the list is partially sorted.
- Time complexity: $O(n^2)$ in the worst case, but can be much faster for nearly sorted data.
- Space complexity: $O(1)$ additional memory.

How Insertion Sort Works

Steps of Insertion Sort

1. Start from the second element (assume the first element is sorted).
2. Compare the current element with the previous elements. If the current element is smaller than the previous element, compare it with the elements before the previous one. Continue this process until you reach a position where the current element is greater, or until you reach the beginning of the array.
3. Insert the current element at the found position.
4. Repeat for all elements.

Example

Sorting a List Using Insertion Sort

$$
\begin{aligned}
\text{Original List:} &\quad [8, 4, 23, 42, 16, 15] \\
\text{Insert 4:} &\quad [4, 8, 23, 42, 16, 15] \\
\text{Insert 23:} &\quad [4, 8, 23, 42, 16, 15] \\
\text{Insert 42:} &\quad [4, 8, 23, 42, 16, 15] \\
\text{Insert 16:} &\quad [4, 8, 16, 23, 42, 15] \\
\text{Insert 15:} &\quad [4, 8, 15, 16, 23, 42]
\end{aligned}
$$

Algorithm in C

```c
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Conclusion

Summing Up

- Insertion Sort is intuitive and simple, making it suitable for small lists or partially sorted data.
- It's an in-place, stable sort.
- While it has a quadratic worst-case time complexity, it often performs well for small datasets or nearly sorted data.

Introduction to Merge Sort

What is Merge Sort?

Merge Sort is a Divide and Conquer algorithm. It works by recursively splitting a list in half. Once you have single-element lists, you merge them back together in a sorted manner.

Features of Merge Sort

- Stable sort
- External and Internal sorting
- Time complexity: $O(n \log n)$
- Space complexity: $O(n)$ (due to the merging process)

How Does It Work?

Steps of Merge Sort

1. Divide the unsorted list into n sublists, each containing one element.
2. Repeatedly merge sublists to produce newly sorted sublists until there is only one sublist remaining.

Example

Sorting a List

Original List:	$[38, 27, 43, 3, 9, 82, 10]$
Splitting:	$[38, 27]$ $[43, 3]$ $[9, 82]$ $[10]$
Merging:	$[27, 38]$ $[3, 43]$ $[9, 82]$
Merge Again:	$[3, 27, 38, 43]$ $[9, 10, 82]$
Final Merge:	$[3, 9, 10, 27, 38, 43, 82]$

Algorithm in C

```c
#include <stdio.h>

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

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, n-1);
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

(Note: This is a partial code; full merge and mergeSort functions not shown for brevity)

Conclusion

Summary

- Merge Sort is efficient with a time complexity of $O(n \log n)$.
- It's a stable sort and works well for large datasets.
- The primary drawback is the additional space requirement of $O(n)$.

Introduction to Quick Sort

What is Quick Sort?

Quick Sort is a Divide and Conquer algorithm. It picks an element as a "pivot" and partitions the array so that all smaller elements come before the pivot and all larger elements come after it. This process is recursively applied to the sub-arrays.

Key Features of Quick Sort

- Not stable by default
- In-place sorting
- Time complexity: $O(n^2)$ worst case, but $O(n \log n)$ on average
- Space complexity: $O(\log n)$ due to recursive call stack

Working Principle

Steps of Quick Sort

1. Choose a 'pivot' element from the array.
2. Rearrange the elements using the pivot, such that elements smaller than pivot are on the left, and elements greater than pivot are on the right.
3. Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with greater values.

Example

Sorting a List

$$\begin{aligned} \text{Original List:} &\quad [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] \\ \text{Choosing pivot (e.g., last element):} &\quad [9, 7, 5, 11, 12, 2, 14, 3, 10, \underline{6}] \\ \text{After 1st partition:} &\quad [5, 2, 3, \underline{6}, 12, 7, 14, 9, 10, 11] \end{aligned}$$

(Note: This process is recursively applied to sub-arrays)

Algorithm in C

```
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);}}
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(
        arr[0]);
    printf("Given array is \n");
    for (int i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);}
    printf("\n");
    mergeSort(arr, 0, arr_size - 1);
    printf("Sorted array is \n");
    for (int i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);}
    printf("\n");
    return 0;}
    }
```

Conclusion

In Summary

- Quick Sort is generally faster in practice compared to other $O(n \log n)$ algorithms, such as Merge Sort.
- It's an in-place sort (requires minimal additional memory).
- The primary concern is its worst-case scenario, which can be mitigated using randomized or median-of-three pivoting.

Introduction to Quick Sort

What is Quick Sort?

Quick Sort is a Divide and Conquer algorithm. It picks an element as a "pivot" and partitions the array so that all smaller elements come before the pivot and all larger elements come after it. This process is recursively applied to the sub-arrays.

Key Features of Quick Sort

- Not stable by default
- In-place sorting
- Time complexity: $O(n^2)$ worst case, but $O(n \log n)$ on average
- Space complexity: $O(\log n)$ due to recursive call stack

Working Principle

Steps of Quick Sort

1. Choose a 'pivot' element from the array.
2. Rearrange the elements using the pivot, such that elements smaller than pivot are on the left, and elements greater than pivot are on the right.
3. Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with greater values.

Example

Sorting a List

$$\begin{aligned} \text{Original List:} &\quad [9, 7, 5, 11, 12, 2, 14, 3, 10, 6] \\ \text{Choosing pivot (e.g., last element):} &\quad [9, 7, 5, 11, 12, 2, 14, 3, 10, \underline{6}] \\ \text{After 1st partition:} &\quad [5, 2, 3, \underline{6}, 12, 7, 14, 9, 10, 11] \end{aligned}$$

(Note: This process is recursively applied to sub-arrays)

Algorithm in C

```
#include<stdio.h>
void swap(int* a, int* b) {
    int t = *a; *a = *b; *b = t;}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);}}
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);}}
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    for(int i=0; i<n; i++) printf("%d␣", arr[i]);
    return 0;}}
```

Conclusion

In Summary

- Quick Sort is generally faster in practice compared to other $O(n \log n)$ algorithms, such as Merge Sort.
- It's an in-place sort (requires minimal additional memory).
- The primary concern is its worst-case scenario, which can be mitigated using randomized or median-of-three pivoting.

Conclusion: Sorting Algorithms

Selection Sort
In-place
$O(n^2)$

Bubble Sort
Adaptive
$O(n^2)$

Merge Sort
Divide & Conquer
$O(n \log n)$

Quick Sort
Partitioning
$O(n^2)$ worst

Insertion Sort
Adaptive
$O(n^2)$

Key Points

- Different algorithms suit different tasks and data.
- Stability and in-place characteristics can be important.
- Always consider the specific use-case when choosing an algorithm.