Analysis and Design of Algorithms

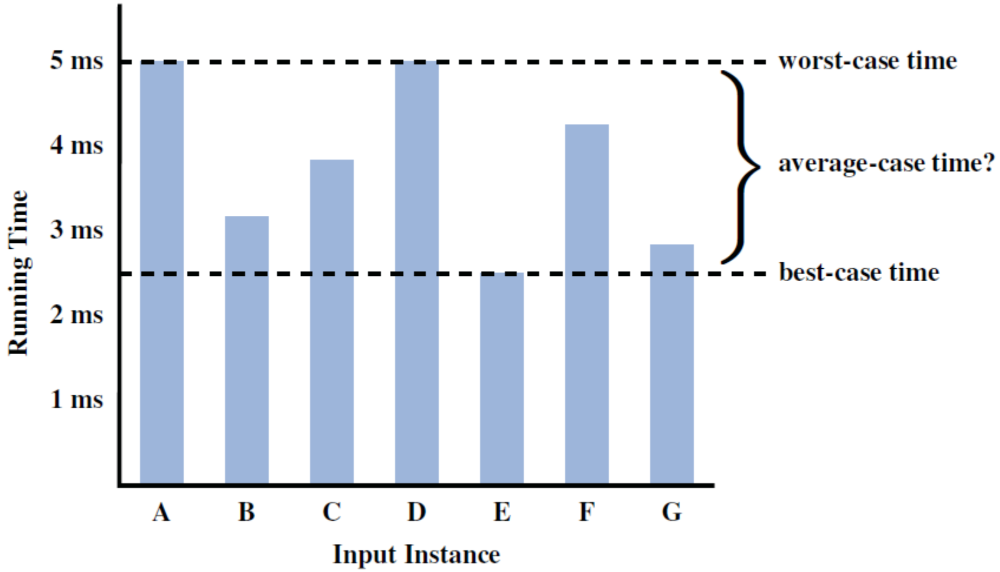
Session 2.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

Iterative Sorting Algorithms

- Why will we study them?
  - To practice the design and a priori and a posteriori analysis of iterative algorithms that involve best, worst, and average cases.
  - To understand algorithms that resemble others of greater interest.
- Which ones will we study?
  - Three with quadratic complexity.
  - Three with quasi-linear complexity.
  - One with linear complexity.



Sorting algorithms
with quadratic complexity

- Why will we study them?
  - To understand the terminology and basic mechanisms of sorting algorithms.
  - They can be extended to better general-purpose methods.
- When is it advisable to use them? (at least one condition is met)
  - When sorting a small amount of data.
  - When the data is nearly sorted.
  - When there are many repeated data points.

Sorting algorithms
with quadratic complexity

- How do we identify them?
  - The number of steps needed to sort N random elements is proportional to N^2.
    - They compare each element against all or almost all other elements.
- What advantages do they have over faster algorithms?
  - Ease of implementation.
  - Most of them are **stable**.
    - For example, if a list of students is sorted alphabetically and then sorted by GPA: students with the same GPA remain in alphabetical order.
  - They don't require extra memory in the order of N.

Sorting algorithms
with quadratic complexity

- Which quadratic sorting method will be better?
  - Selection
  - Insertion
  - Bubble
- Let's define quantitative efficiency criteria that are independent of the machine:
  - Number of comparisons between data elements.
  - Number of data element movements (reads or writes).

Sorting by Selection

- Finds the smallest element in the array and swaps it with the element in the first position.
- Finds the second smallest element (or the smallest of the remaining N – 1) and swaps it with the second element, and so on.

```
6 9 8 1 4 2 5 0 4 7
0 9 8 1 4 2 5 6 4 7
0 9 8 1 4 2 5 6 4 7
0 1 8 9 4 2 5 6 4 7
0 1 8 9 4 2 5 6 4 7
0 1 2 9 4 8 5 6 4 7
```

…

Selection Sort visualize | Algorithms | HackerEarth

Sorting by Selection

- How many **comparisons** does the Selection algorithm perform if the array is unordered?

```
def selectionSort(arr):
    for i in range(len(arr)-1):
        minV = i
        for j in range(i+1,len(arr)):
            if(arr[j]<arr[minV]):
                minV = j

        if(minV != i):
            arr[i], arr[minV] = arr[minV], arr[i]
```

Sorting by Selection

- The outer loop runs N – 1 times. The inner loop runs N – i times.
  - In the 1st iteration of the outer loop, N – 1 comparisons are made.
  - In the 2nd iteration of the outer loop, N – 2 comparisons are made.
  - In the (N – 1)-th iteration of the outer loop, 1 comparison is made.

```python
def selectionSort(arr):
    for i in range(len(arr)-1):
        minV = i
        for j in range(i+1,len(arr)):
            if(arr[j]<arr[minV]):
                minV = j

        if(minV != i):
            arr[i], arr[minV] = arr[minV], arr[i]
```

Sorting by **Selection**

- In total, the number of comparisons is: $1 + 2 + \ldots + (N - 2) + (N - 1)$.
- Recalling that: $1 + 2 + \ldots + N = \frac{1}{2} N(N+1)$
- $\therefore$ Comparisons $= \frac{1}{2}(N - 1)(N) = 0.5N2 - 0.5N \in O(N2)$
- What is the spatial complexity?
- And what if the array were sorted?
- And if the array were reversed?

Sorting by Selection

- How many **moves** does the Selection algorithm perform if the array is sorted, reversed, and unordered?

```
def selectionSort(arr):
    for i in range(len(arr)-1):
        minV = i
        for j in range(i+1,len(arr)):
            if(arr[j]<arr[minV]):
                minV = j

        if(minV != i):
            arr[i], arr[minV] = arr[minV], arr[i]
```

Sorting by Selection

- If the array is **sorted**, the index of the smallest element, min_index, will always be the same as the index of the outer loop i.
- It will never execute the swap operation.
- Hence, the number of swaps is: $0 \in O(K)$

```
def selectionSort(arr):
    for i in range(len(arr)-1):
        minV = i
        for j in range(i+1,len(arr)):
            if(arr[j]<arr[minV]):
                minV = j

        if(minV != i):
            arr[i], arr[minV] = arr[minV], arr[i]
```

Sorting by Selection

- If the array is **reversed**, in the first half of the outer loop, min_index will always be different from i: one swap per iteration.
- At the beginning of the second half, the array will already be sorted.
- Recall that the `swap` method performs three movements.
- Hence, the number of swaps is: $3(\frac{1}{2} N) = 1.5N \in O(N)$.
- What if the array is **unordered**?
  - We would expect an average of 0.75N swaps.
  - For this case, **a posteriori** analysis is suggested.

Sorting by Insertion

- It starts with the second element.
- It moves one element to the left until finding a smaller element.
- The elements on the left side are already sorted.

```
6 9 8 1 4 2 5 0 4 7  temp = 9
6 9 8 1 4 2 5 0 4 7  temp = 8
6 9 9 1 4 2 5 0 4 7
6 8 9 1 4 2 5 0 4 7  temp = 1
6 8 9 9 4 2 5 0 4 7
6 8 8 9 4 2 5 0 4 7
6 8 8 9 4 2 5 0 4 7
6 6 8 9 4 2 5 0 4 7
1 6 8 9 4 2 5 0 4 7  temp = 4
…
```

[Insertion Sort Animation by Y. Daniel Liang (pearsoncmg.com)]

Sorting by Bubble

- It traverses the list from left to right, swapping adjacent elements if necessary.
  - The largest element will end up at the final position.
- The traversal is repeated without comparing the last element from the previous pass.
  - If no swaps occurred in a pass, the array is sorted 👍

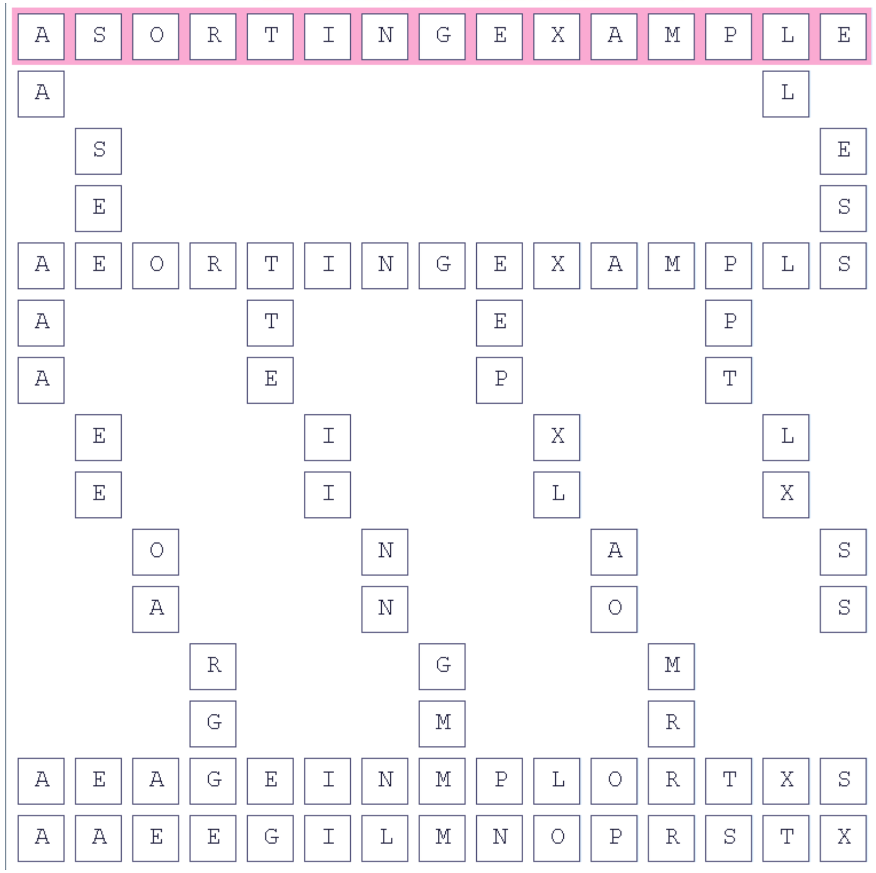
```
6 9 8 1 4 2 5 0 4 7
6 8 9 1 4 2 5 0 4 7
6 8 1 9 4 2 5 0 4 7
6 8 1 4 9 2 5 0 4 7
6 8 1 4 2 9 5 0 4 7
6 8 1 4 2 5 9 0 4 7
…
6 8 1 4 2 5 0 4 7 9
```

Bubble Sort Animation by Y. Daniel Liang (yongdanielliang.github.io)

Sorting by ShellSort

- The **Insertion sort** method is slow because it only swaps adjacent elements:
- If the smallest element is at the very end, it takes N steps to place it in its correct position.
- **ShellSort** is an extension of **Insertion sort** that speeds up the process by swapping elements that are far apart.
  - The hallmark of this algorithm is:
  - In a pass, the elements k, k+h, k+2h, k+3h,… should form an ordered array, for all $0 \le k < h$.
  - In each pass, the value of h is decreased until it reaches 1.
  - What is the initial value of h? How should it be decreased?

Sorting by ShellSort

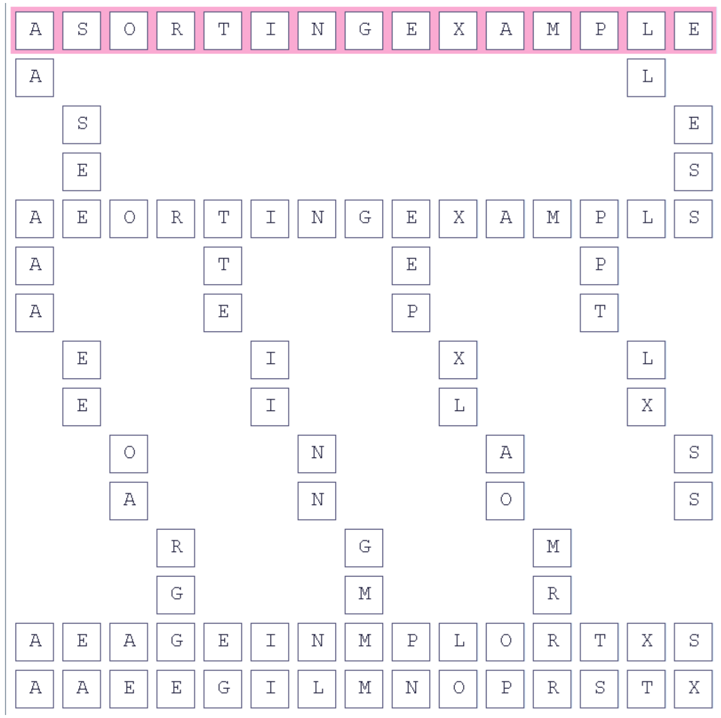


- Some authors suggest the sequence:
  - 1093, 364, 121, 40, 13, 4, 1
  - The right-to-left relationship is 3k + 1.

Sorting by ShellSort

- Some authors suggest the sequence:
  - 1093, 364, 121, 40, 13, 4, 1
  - The right-to-left relationship is 3k + 1.
  - Pairs and odds alternate.
  - The execution time is considerably reduced compared to the **Insertion** method.
  - A poor sequence is:

    64, 32, 16, 8, 4, 2, 1
    - It doesn't compare pairs with odds.
  - IT'S NOT STABLE!!! (Doesn't preserve the order of elements as they appear in the array)



Conclusions

- We analyzed and implemented some quadratic algorithms.
- We observed that they are easy to implement and help us understand how to analyze them.
- We recognized the importance of analyzing different cases: best, worst, and average.