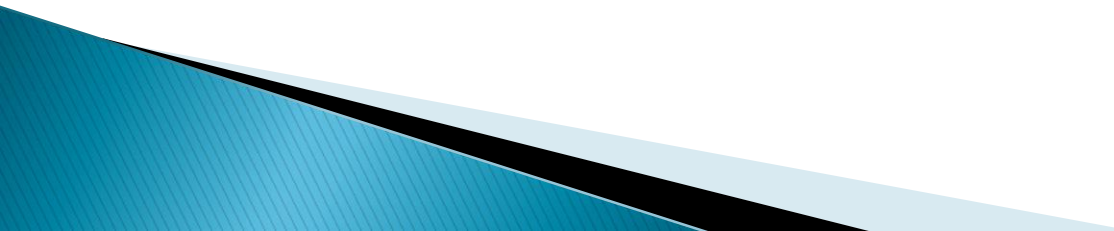🔨 **Build X: Algorithms**

NEW

Welcome!
Monday January 25 18:30 - 20:30

Topics to discuss:

- What is Complexity/Efficiency? "*Big O*"
  - Time Complexity
  - Memory Consumption
- How can we calculate them in theory?
- Examples based on Different Sorting Algorithms
  - Bubble Sort
  - Counting Sort
  - Insertion Sort
  - Selection Sort

What is an Algorithm?

"An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output."
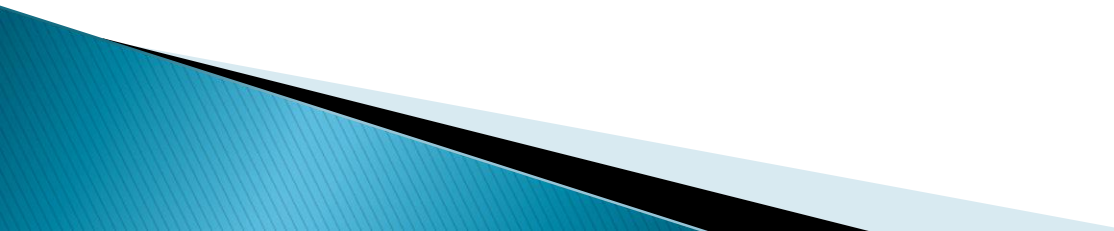
2 important aspects

- ▸ Time complexity

- ▸ Space Complexity

Big O notation

- **Big O notation** is used in Computer Science to describe the performance or complexity of an algorithm.

- It specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used by the algorithm.

Time Complexity

▸ Why do we need it ?
  ◦ Simple. We want our data as fast as possible.

  ◦ If an algorithm is too slow then we go back to the drawing board.

▸ Algorithms can depend on :
  ◦ The size of the input
  ◦ The type of the input
    • eg. *A sorting algorithm may run much faster when given a set of integers that are already sorted than it would when given the same set of integers in a random order.*

Time Complexity

- An algorithm is usually evaluated in a:
  - **Worst case runtime**
    - how long it would take for the algorithm to run if it were given the most insidious of all possible inputs.
  - **Average case runtime**
    - the average of how long it would take the algorithm to run if it were given all possible inputs.

- The worst-case is often easier to reason about, and therefore is more frequently used as a benchmark for a given algorithm.

Time Complexity

- Count the number of steps your algorithms takes to solve a problem.

Time Complexity

- General values of time complexities:
  - O (1) – constant
  - O (log(n)) – logarithmic
  - O (n) – linear
    - Eg. *for (i = 0; i < n; ++i) { do something}*
  - O (n*log(n))
  - O (n^2) – quadratic
  - O (2^n) – exponential

  - General case : O (n) = c * F(n)
    - c is a constant
    - F is a function that depends on N

Time Complexity

▸ General rules to get the value of O()

- ◦ Drop lower terms
- ◦ Drop constant factors
  - • Eg. $5*(n^4) + 20*(n^3) + 13$ is $O(n^4)$
- ◦ Use the simplest expression of the function
  - • Eg $3*n$ is $O(n)$ not $O(3*n)$

Examples

- f(n) = n –> O(n)
- f(n) = 25 * n –> O(n)
- f(n) = n ^ 4 –> O(n^4)
- f(n) = n ^ r –> O(n^r)
- f(n) = n ^ 3 + 2*(n^2) + 26 –> O(n^3)
- f(n) = (10^2) * n + 100 –> O(n)

Space Complexity

- **Space complexity** is a measure of the amount of working storage an **algorithm** needs.

- As with time complexity, we are concerned with the worst-case scenario.

Space Complexity

- How to calculate it
  - As in time complexity we drop the constants and lower terms and we focus on the data structures that we actually use.

  - Eg. Our program uses:
    - 1 ArrayList of length N;
    - 2 Vectors of length M;
    - 1 Matrix of AxB;
  - Memory consumption: $N*(b)+2*M*(b2)+A*B*(b3)$
    - Where b,b2,b3 represents the number of bits an individual element of the structure needs.

The most important one?

It's a mix between them.
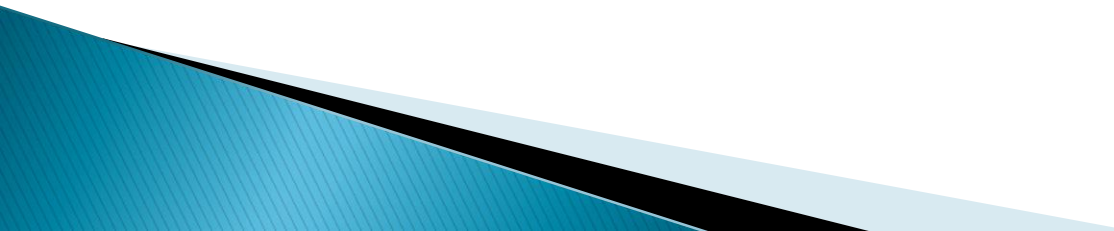
Examples

▸ Most basic example : Sorting Algorithms

◦ Numbers
◦ Strings
◦ Objects.
◦ Etc.

All are based on an algorithm

Bubble Sort

- The most basic sorting algorithm
- Easy to implement
- Easy to use
- Which means …. not that good in real life

Bubble Sort – Example

6 5 3 1 8 7 2 4

Bubble Sort – Step by step example

- Example "5 1 4 2 8". In each step, elements written in **bold** are being compared and swapped if needed. Three passes will be required.

- First Pass
- (**5 1** 4 2 8) ->(**1 5** 4 2 8
  (1 **5 4** 2 8) ->(1 **4 5** 2 8),
  (1 4 **5 2** 8) ->(1 4 **2 5** 8),
  (1 4 2 **5 8**) ->(1 4 2 **5 8**), since these elements are in order (8 > 5), algorithm does not swap them.

-
- Second Pass
- (**1 4** 2 5 8) ->(**1 4** 2 5 8),
- (1 **4 2** 5 8) ->(1 **2 4** 5 8),
- (1 2 **4 5** 8) ->(1 2 **4 5** 8),
- (1 2 4 **5 8**) ->(1 2 4 **5 8**)
  !!!Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.!!!
- Third Pass
- (**1 2** 4 5 8) ->(**1 2** 4 5 8)
  (1 **2 4** 5 8) ->(1 **2 4** 5 8)
  (1 2 **4 5** 8) ->(1 2 **4 5** 8)
  (1 2 4 **5 8**) ->(1 2 4 **5 8**)

Source Wikipedia

Bubble Sort –code

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
      swapped = false
      for i = 1 to n-1 inclusive do
        /* if this pair is out of order */
        if A[i-1] > A[i] then
          /* swap them and remember something changed */
          swap( A[i-1], A[i] )
          swapped = true
        end if
      end for
    until not swapped
end procedure
```

Source Wikipedia

Bubble Sort – Conclusion

▶ Time complexity :
  ◦ *Worst case* : $O(n^2)$
  ◦ Best case : Elements are already sorted $O(n)$

▶ Space Complexity :
  ◦ Extremely efficient in terms of memory usage because it does not use any extra memory. It only swaps elements around

Bubble Sort- Conclusions

▸ When can we use it?
  ◦ On small lists
  ◦ Check if the list is already sorted – O(n)

▸ Conclusion :

    It is not practical ☹

Counting Sort

- Basic sorting algorithm
- Easy to implement
- Easy to use

- Which means …. has its own disadvantages
  ◦ Not always !

- Description:
  ◦ It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence.

Counting Sort – Example

Input :

    1 5 2 1 0 9 8 2 4 5 2 3 5 1 1 1

Step 1:

    0 1 2 3 4 5 6 7 8 9

    1 5 3 1 1 3 0 0 1 1

In our initial input "0" appears once, "1" appears 5 times, "2" appears 3 times, etc.

Step 2 :

    0 1 1 1 1 1 2 2 2 3 4 5 5 5 8 9

Counting Sort

- ▸ Time complexity
  - ◦ O(n) – read and print the values
- ▸ Memory consumption
  - ◦ O (max−min) – keep another data structure for counting
    - Max = highest element
    - Min = minimum element
- ▸ When can we use it?
  - ◦ Good if our elements are in a specified interval with no high variations
  - ◦ Usually used as a sub-routine of other algorithms (Radix Sort)

Insertion Sort

▸ Advantages
- ◦ Simple implementation
- ◦ Efficient for small data sets
- ◦ More efficient(in practice not in theory) than "Bubble Sort" or "Selection Sort"
- ◦ Becomes more efficient if elements are closer to their "sorted positions"
- ◦ Memory consumption

▸ Disadvantages
- ◦ Consumes a lot of time

Insertion Sort – Description

- Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain

Insertion Sort – Example

6   5   3   1   8   7   2   4

Insertion Sort – Step by step example

Input 3 7 4 9 5 2 6 1

- **3** _7_ 4 9 5 2 6 1
- 3 **7** _4_ 9 5 2 6 1
- 3 **4** 7 _9_ 5 2 6 1
- 3 4 7 **9** _5_ 2 6 1
- 3 4 **5** 7 9 _2_ 6 1
- **2** 3 4 5 7 9 _6_ 1
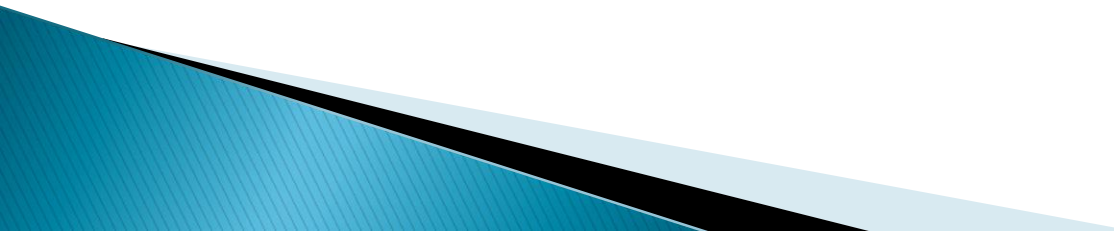- 2 3 4 5 **6** 7 9 _1_

Output **1** 2 3 4 5 6 7 9

Insertion Sort

- Time Complexity
  - O(n^2)
  - O(n) – values are already sorted
- Memory Consumption:
  - Does not require any additional memory

- When to use it?
  - As a sub-routine for other algorithms
    - Eg. Quick-Sort and Merge-Sort

Selection Sort

- Simple algorithm to use;
- In practice better than insertion sort and bubble sort

Steps:
- Select the lower element in the list
- Swap it with the current element
- Repeat for the rest of the elements

Selection Sort – Description

- The algorithm divides the input list into two parts:
  ◦ the sublist of items already sorted, which is built up from left to right at the front (left) of the list;
  ◦ the sublist of items remaining to be sorted that occupy the rest of the list.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Source Wikipedia

Selection Sort – Example

Input : 64 25 12 22 11

- 11 25 12 22 64
  - sorted sublist = {11}
- 11 12 25 22 64
  - sorted sublist = {11, 12}
- 11 12 22 25 64
  - sorted sublist = {11, 12, 22}
- 11 12 22 25 64
  - sorted sublist = {11, 12, 22, 25}
- 11 12 22 25 64
  - sorted sublist = {11, 12, 22, 25, 64}

Output : 11 12 22 25 64

Source Wikipedia

Selection Sort

8
5
2
6
9
3
1
4
0
7

Selection Sort

- ▶ Time complexity
  - ◦ O (n^2)

- ▶ Memory consumption
  - ◦ Does not require any additional memory

- ▶ Using heaps as data structures greatly improves optimality

Any questions?

Next week

- Insertion Sort with better complexity
- Merge Sort ("Divide et Impera")
- Quick Sort
- Radix Sort

- Real life examples