Basic Sorting Algorithms (2)

Instructor: Krishna Venkatasubramanian

CSC 212

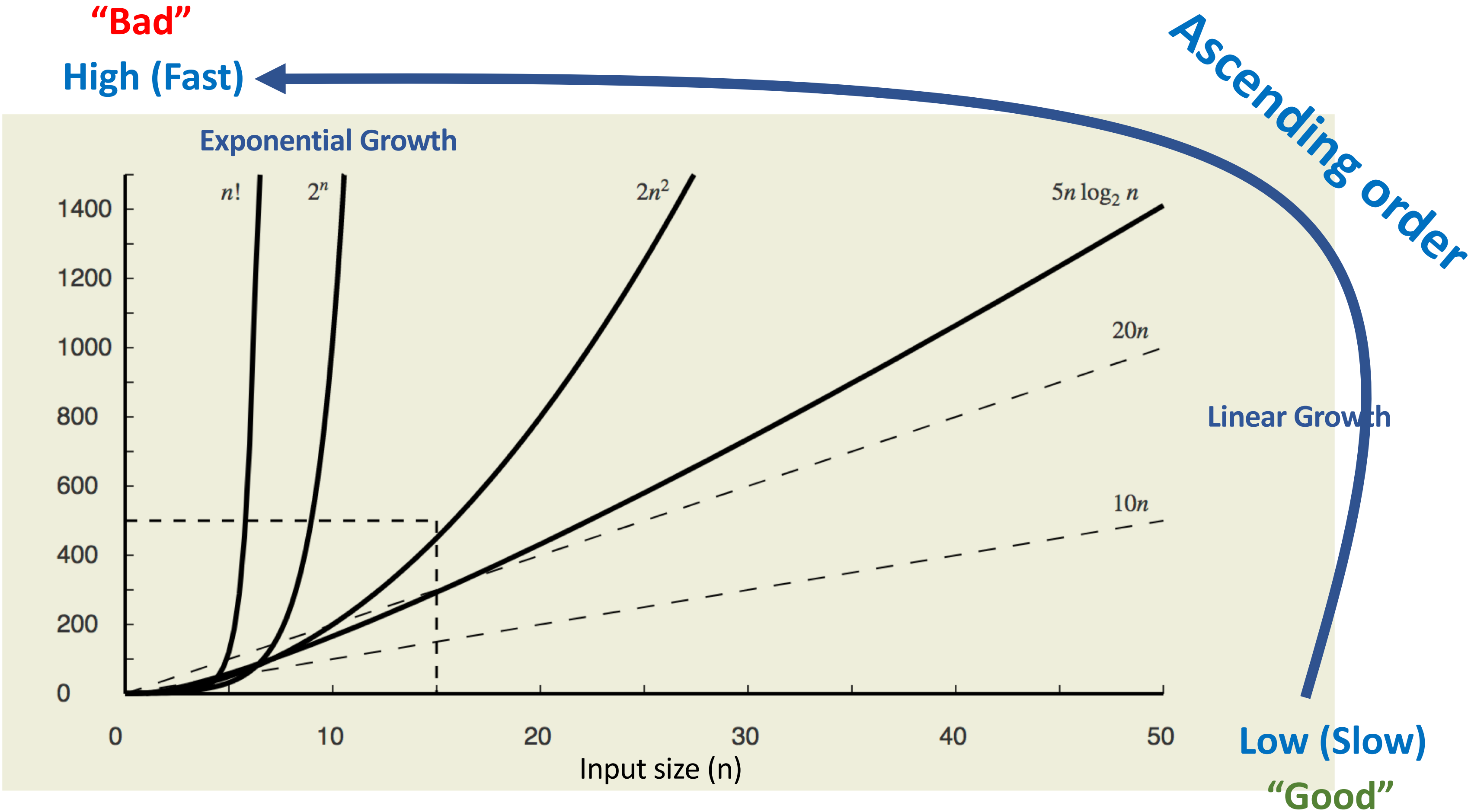
Announcement

- Don’t forget the office hours poll on Piazza.
  - Last day to do it today.
  - We will make our decision (to change or not to change) tomorrow.

- Who is Ming?

- Grade for Quiz 1 will be out soon.

Clarification: Asymptotic Growth Rates



Sorting: Problem Definition

- **Sorting takes an unordered collection and makes it an ordered one.**

Input:

1	2	3	4	5	6
77	42	35	12	101	5

Output:

1	2	3	4	5	6
5	12	35	42	77	101

Sorting Algorithms

- *Insertion Sort --- covered already*
- *Bubble Sort --- covered already*
- **Selection Sort**
- Heap Sort
- Merge Sort
- Quick Sort
- …

Selection Sort

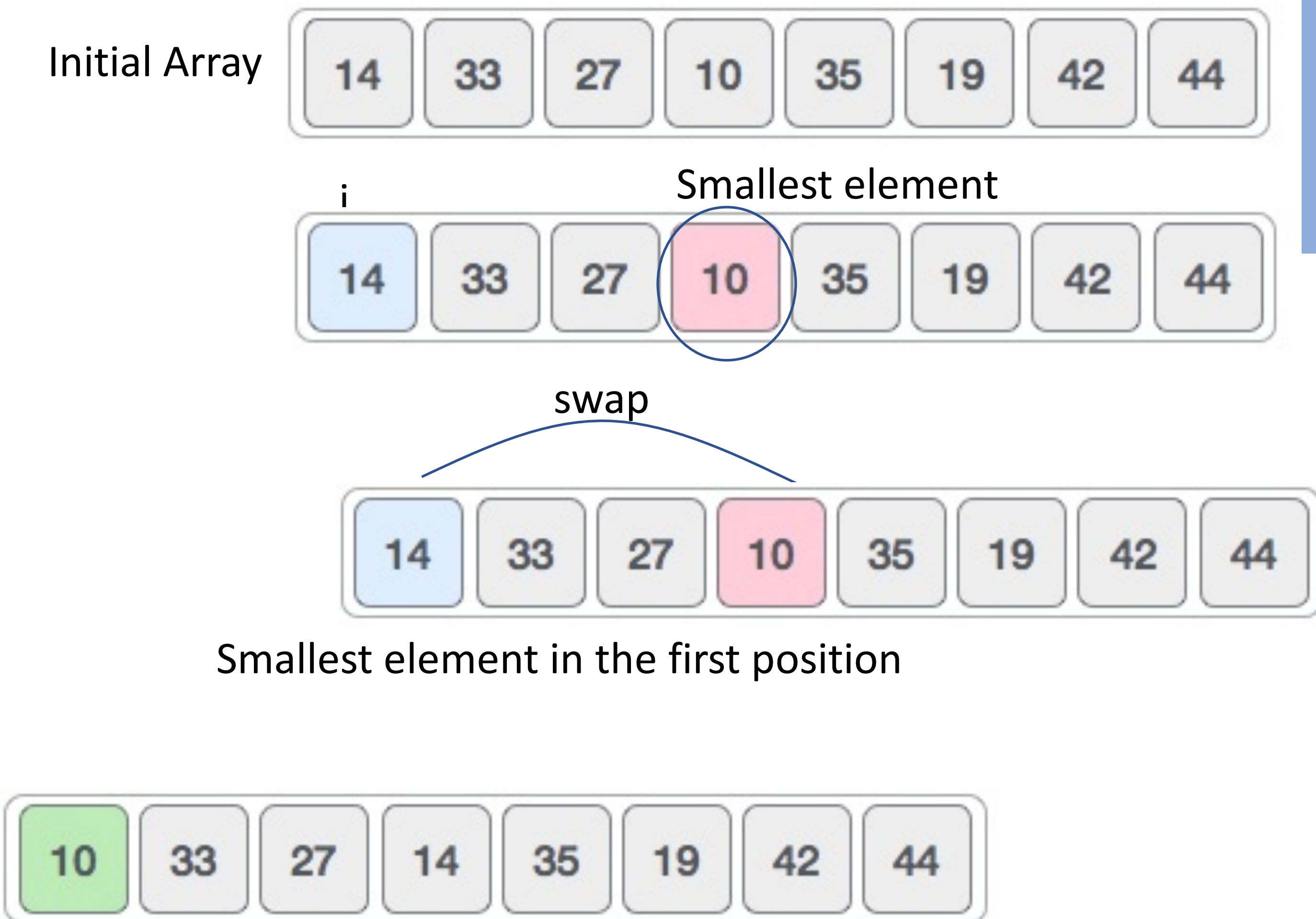
- Simple Algorithm --- has similarities to both Insertion-Sort and Bubble-Sort
  - Array divided into two parts – sorted and unsorted (like Insertion-Sort)
  - Stack the array from smallest to the largest (inverse of Bubble-Sort)
- **Invariants**
  - Elements in the sorted array are fixed
  - No element in the sorted array is greater than element in the unsorted array

1	2	4	5	18	9	10	7

SORTED ARRAY UNSORTED ARRAY

Example (of basic steps)

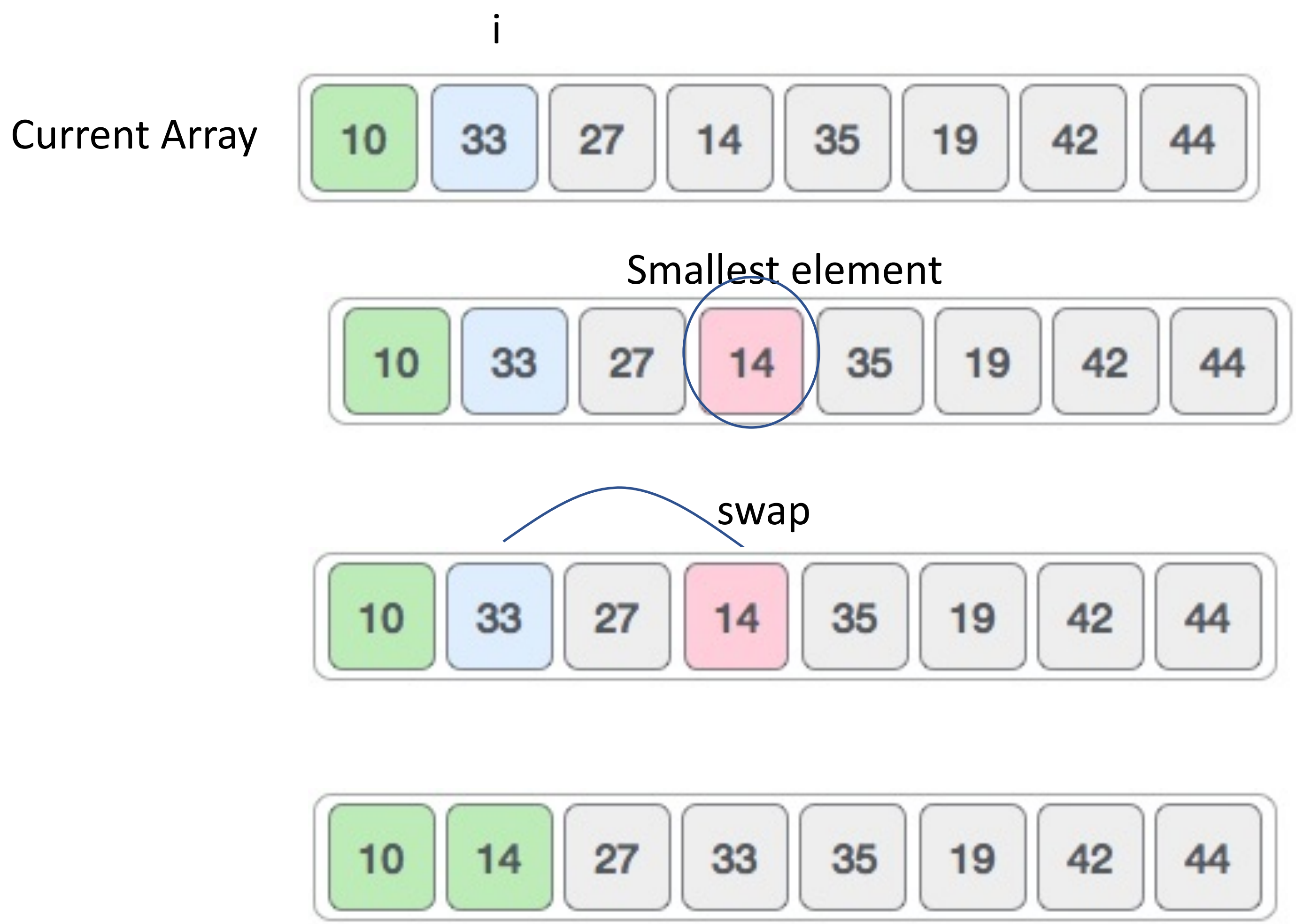
i = denotes position which is to be brought into the sorted array (initially the first element, as the entire array is unsorted)



The first element is in it’s correct spot

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

Next Step



The first two element is in it's correct spot

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

Continuing On…

Green = already sorted (numbers in correct position)

Blue = i^{th} position

Red = smallest element, which swaps with the i^{th} position element

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

The Selection Sort Algorithm

```
Selection Sort( A ):

 for i in range(len(A)-1):

    # Find the minimum element in remaining unsorted array
    min_id = i
    for j in range(i+1, len(A)):
        if A[min_id] > A[j]:
            min_id = j

    # Swap the found minimum element with the first element
    A[i], A[min_id] = A[min_id], A[i]
```

Why len(A) – 1?

Why len(A) here ?

Selection Sort Time Complexity

- **Best-Case Time Complexity**
  - **The scenario under which the algorithm will do the least amount of work (finish the fastest)**

- **Worst-Case Time Complexity**
  - **The scenario under which the algorithm will do the largest amount of work (finish the slowest)**

Selection Sort Time Complexity

- **Best-Case Time Complexity**
  - **Array is already sorted**
  - **Need N-1 iterations**
  - **(N-1) + (N-2) + …. + 1 = (N-1)* N / 2 comparisons**

Called Quadratic Time
$O(N^2)$
Order-of-N-square

Called Quadratic Time
$O(N^2)$
Order-of-N-square

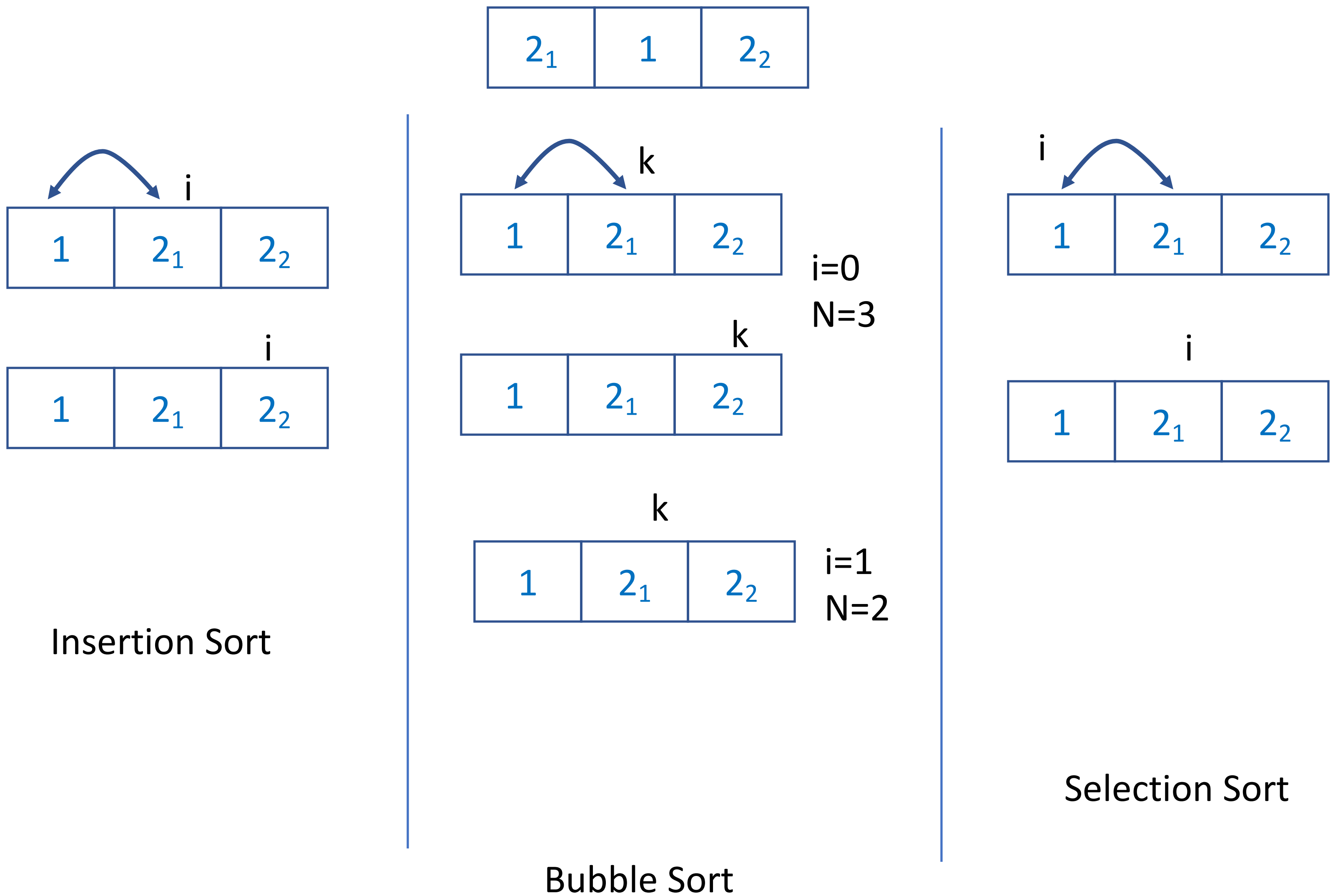
- **Worst-Case Time Complexity**
  - **Array reverse sorted**
  - **Still need N-1 iterations**
  - **(N-1) + (N-2) + …. + 1 = (N-1)* N / 2**

Sorting Algorithm Additional Properties

- Memory Requirement:
  - **Denotes the amount of auxiliary storage** needed beyond that used by the list itself, under the same assumption.
    - No extra storage required. **Sorting in place.**

- Stability:
  - **Stable sorting algorithms sort repeated elements in the same order that they appear** in the **input**.
    - When sorting some kinds of data, only part of the data is examined when determining the sort order.
    - If two items compare as equal if one came before the other in the input, it will also come before the other in the output.

Stability Example



Memory Use and Stability of Basic Sorting Algorithms

- Insertion Sort
  - Memory = O(1) *# in place sorting (small amount of auxiliary space allowed, like to temp swap a variable)*
  - Stability = Yes
  - Best-case Running Time = O(n)
  - Worst-case Running Time = $O(n^2)$
- Bubble Sort
  - Memory = O(1) *# in place sorting*
  - Stability = Yes
  - Best-case Running Time = $O(n^2)$
  - Worst-case Running Time = $O(n^2)$
- Selection Sort
  - Memory = O(1) *# in place sorting*
  - Stability = Yes
  - Best-case Running Time = $O(n^2)$
  - Worst-case Running Time = $O(n^2)$

Exchange Sorts

- All three sorting algorithms seen thus far are examples of **Exchange sorting** algorithms
  - They swap (exchange) elements to sort

- Other algorithms exist that are faster that these for typical conditions

- So why are these algorithms slow?

General Slowness of Exchange Sort

- The crucial bottleneck is that only *adjacent* records are compared.
  - Thus, comparisons and moves (for Insertion and Bubble Sort) are by single steps.

- The cost of any exchange sort can be *at best* **the total number of <u>inversions</u>**
  - The number of elements with key value greater than the current record's key value appears before it that need to be moved.

Quiz

- How many **inversions** are in the following array?

14	5	81	92	101	7

5 inversions

4	5	1	9	11	12

2 inversions

63	-51	-9	17	1	0

8 inversions

Worst Case

N elements

X_n	X_{n-1}	X_{n-2}	….	1

$1+2+3...+n-1 = n(n-1)/2$
$= O(n^2)$ inversions

$X_n > X_{n-1} > X_{n-2}.... > 1$

Basic Sorting algorithms

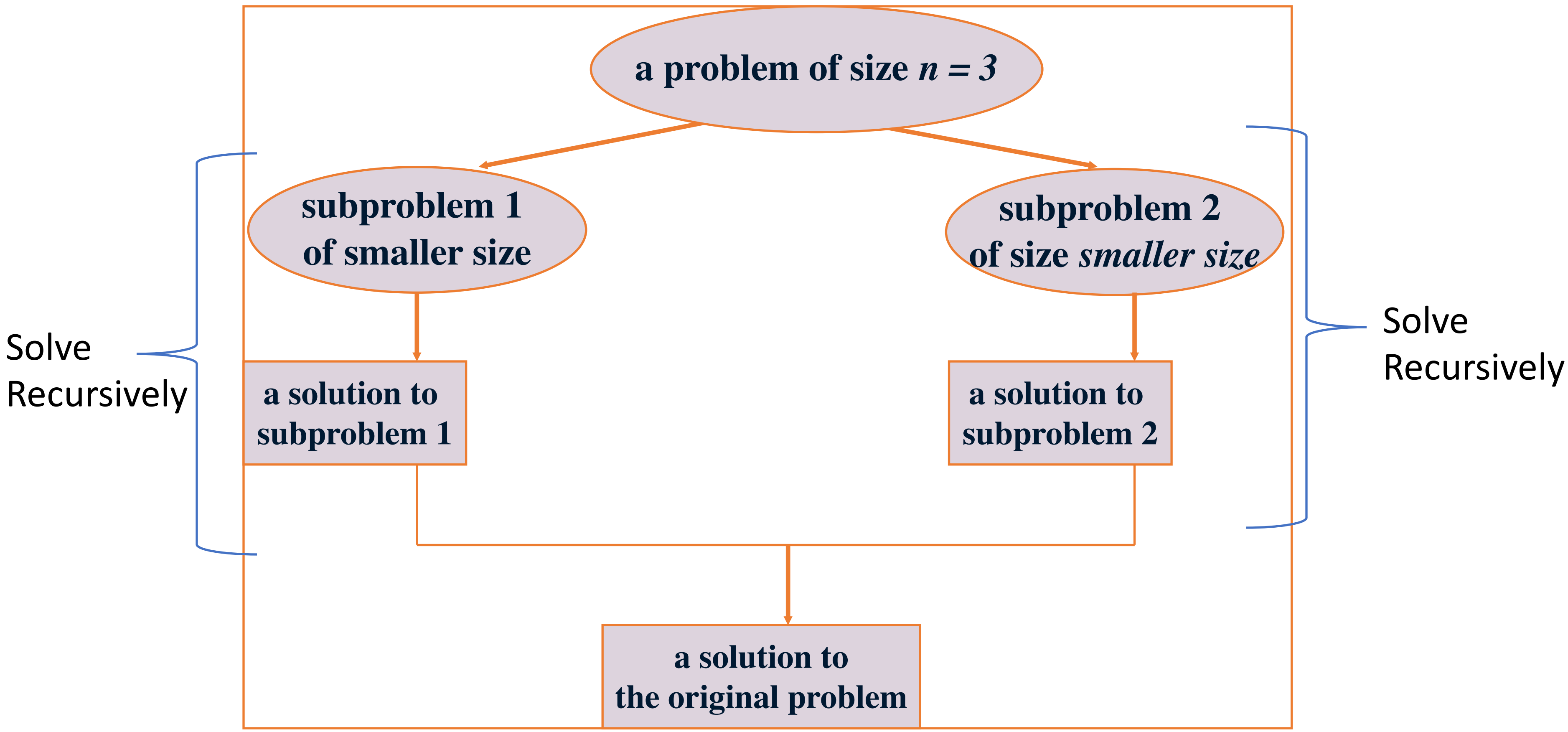
- Insertion, selection and bubble sort have quadratic worst-case performance
  - That is $O(n^2)$
- The faster comparison based algorithm ?
  - $O(n*\log_2 n)$ or $O(n\lg n)$ complexity
  - REMEMBER: $O(n\lg n)$ more efficient than $O(n^2)$
- Examples Mergesort and Quicksort

Divide and Conquer Algorithms

- Before going to the details it is useful to under the concept of **Divide and Conquer Algorithms** and **Recursion**

- Divide and Conquer Algorithms
  - **Divide:** Break the larger problem into sub-problems that are smaller instances of the same problem
  - **Conquer:** the sub-problems are solved ***recursively***
    - If the sub-problem is really small, then solve in a straight-forward manner
  - **Combine:** combine the solutions of the sub-problems to find the solution of the original problem!

Visually Speaking!



Expand Further

https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

Recursive Problem Solving

- What is Recursion?
  - a method of solving a problem where the solution depends on **solutions to smaller instances of the same problem**
- In effect
  - it is **basically a function calling itself** with a slightly smaller set of parameters
  - Can you see why Divide and Conquer Algorithms use it?
- **Recursive Algorithms MAY NOT be intuitive and need a lot of practice to work with!**

Example: Factorial

- Factorial of a positive number p (written as n!) is
  - p! = p*p-1*p-2*…*1
  - p! = 1, if p = 0
- Non-recursive Algorithm

```
def fact(n):
     for i in range(1,n+1):
          fact = fact * i
     return fact
```

- Recursive Algorithm

```
def fact(n):
     if n == 1:
        return 1
     else:
        return n*fact(n-1)
```

Example: Summing an Array

- Given an array A, sum of an array is a number which is a sum of all its values.

- Non-recursive Algorithm

```
def sum(A, size):
     sum = 0
     for i in range(size):
         sum = sum A[i]
     return sum
```

- Recursive Algorithm

```
def sum(A,size):
     if size == 1:
       return A[0]
     else:
       return size + sum(A,size-1)
```

Example: Power of a Number

- Power of a number b^n (where n >=0) is defined as
  - $b^n = b*b*b*b\ldots*b$ (multiplied n times with itself)
  - $b^n = 1$, if $n = 0$

- Non-recursive Algorithm

```
def power(b,n):
    power = ??
    for i in range(n):
        power = power * b
    return power
```

- Recursive Algorithm

```
def power(b,n):
    if n == 0:
      return 1
    else:
      return b*power(b,n-1)
```

The Selection Sort Algorithm

```
Selection_Sort( A ):

   for i in range(len(A)-1):

    min_id = i
    for j in range(i+1, len(A)):
        if A[min_id] > A[j]:
            min_id = j

    swap(A[i], A[min_id])
```

Non-recursive Algorithm

i is the index of the array at which we want to "correctly" populate. Initially ZERO

Recursive Algorithm

```
Selection_Sort_Rec( A, n, i ):
   if i == n:
     return
   else:
     k = find_min_index(A, n, i)
     if k != i:
       A[i], A[k] = A[k], A[i]

     Selection_Sort_Rec(A,n,i+1)
```

find_min_index is a function to find the min element in an Array A, starting at I and ending at n. It needs to be written separately


That's all Folks!
Any Question?