INSERTION SORT

Consider an example: arr[]: {12, 11, 13, 5, 6}

1) First Pass: Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6

2) Second Pass: Now, move to the next two elements and compare them

11	12	13	5	6

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

3) Third Pass: Now, two elements are present in the sorted sub-array which are 11 and 12

Moving forward to the next two elements which are 13 and 5

11	12	13	5	6

Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6

After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6

Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6

Here, 5 is at its correct position

4) Fourth Pass: Now, the elements which are present in the sorted sub-array are 5, 11 and 12

Moving to the next two elements 13 and 6

5	11	12	13	6

Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13

Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13

Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13

Finally, the array is completely sorted.

Best Case

- The "best case" is that the input array is already sorted
- Then we won't need to execute the inner while loop to insert the chosen element
- Let $t_j = 1$

Analysis of Efficiency (Best Case)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} 1$$

$$+ c_6 \sum_{j=2}^{n} 0 + c_7 \sum_{j=2}^{n} 0 + c_8(n-1)$$

Analysis of Efficiency (Best Case)

$$T(n) = an - b$$

where

$$a = c_1 + c_2 + c_4 + c_5 + c_8$$

$$b = c_2 + c_4 + c_5 + c_8$$

In the best case, the run time grows *linearly* with input size ($O(n)$)

- **When to Use**: Insertion Sort is efficient for small data sets or nearly sorted data.
- **Why**: It has a simple implementation and works well when the input data is mostly sorted because it makes minimal comparisons and swaps.

Insertion Sort

```
INSERTION-SORT(A)

for j = 2 to A.length
  key = A[j]

  // shift elements in sorted sequence A[1..j - 1]
  i = j - 1
  while i > 0 and A[i] > key
    A[i+1] = A[i]
    i = i - 1

  // insert A[j] into the sorted sequence
  A[i+1] = key
```

Worst Case

- In the worst case, the input array is reverse sorted order
- We will need to compare each chosen element to all of the currently-sorted elements
- Let $t_j = j$

Summation Identities

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

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

Analysis of Efficiency (Worst Case)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

Analysis of Efficiency (Worst Case)

$$T(n) = an^2 + bn + c$$

vhere

$$a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}$$

$$b = c_1 + c_2 + c_4 + + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8$$

$$c = -\left(c_2 + c_4 + c_5 + c_8\right)$$

ı the worst case, the run time grows *quadratically* with input size ($O(n^2)$)

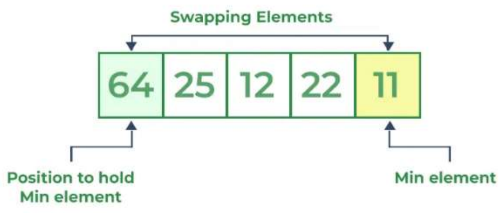
```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        key = lst[i]
        j = i - 1
        while j >= 0 and key < lst[j]:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = key
```

SELECTION SORT

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

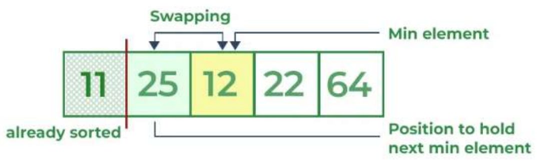
1) For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

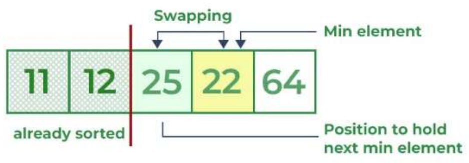


2) Second Pass: For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

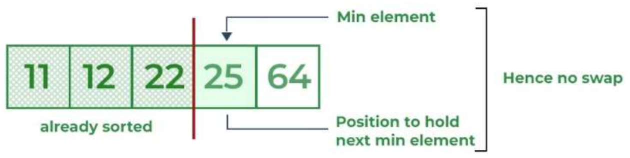
After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



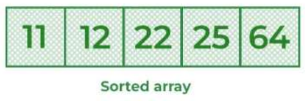
3) Third Pass: Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array. While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

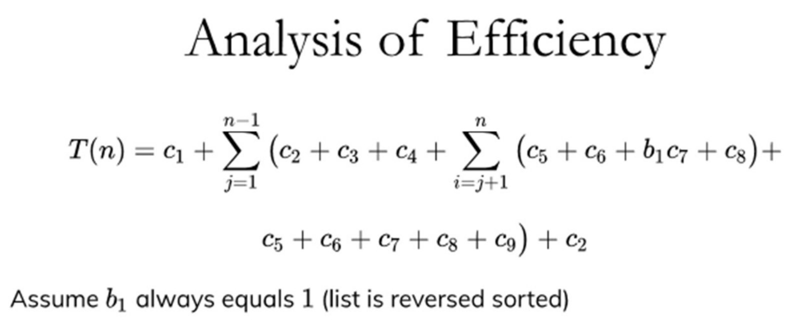


4) Fourth pass: Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array. As 25 is the 4th lowest value hence, it will place at the fourth position.



5) Fifth Pass: At last the largest value present in the array automatically get placed at the last position in the array. The resulted array is the sorted array.

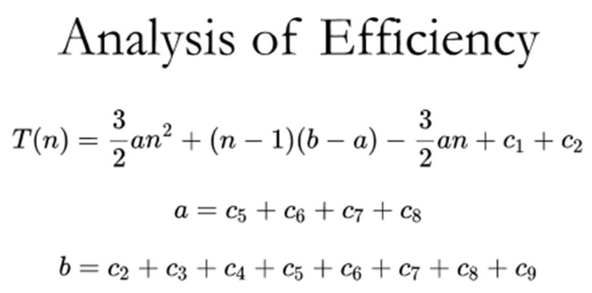




Analysis of Efficiency

$$T(n) = c_1 + \sum_{j=1}^{n-1} \left(c_2 + c_3 + c_4 + \sum_{i=j+1}^{n} (c_5 + c_6 + b_1 c_7 + c_8) + c_5 + c_6 + c_7 + c_8 + c_9\right) + c_2$$

Assume b_1 always equals 1 (list is reversed sorted)

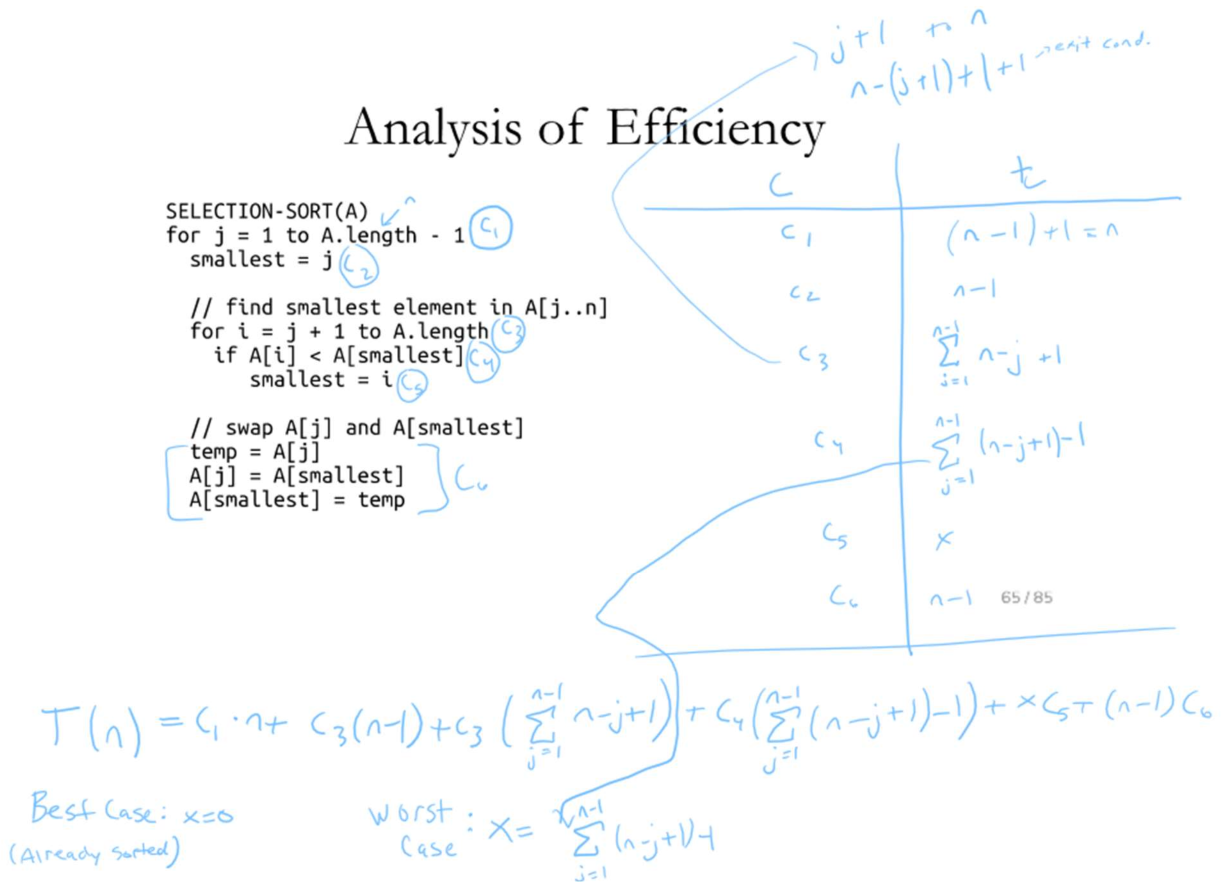


Analysis of Efficiency

$$T(n) = \frac{3}{2}an^2 + (n-1)(b-a) - \frac{3}{2}an + c_1 + c_2$$

$$a = c_5 + c_6 + c_7 + c_8$$

$$b = c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9$$



$j+1$ to n

$n-(j+1)+1+1 \to$ exit cond.

Analysis of Efficiency

```
SELECTION-SORT(A)
for j = 1 to A.length - 1        c1
  smallest = j                   c2

  // find smallest element in A[j..n]
  for i = j + 1 to A.length      c3
    if A[i] < A[smallest]        c4
       smallest = i              c5

  // swap A[j] and A[smallest]
  temp = A[j]                    
  A[j] = A[smallest]             c6
  A[smallest] = temp
```

c	t
c_1	$(n-1)+1=n$
c_2	$n-1$
c_3	$\sum_{j=1}^{n-1} n-j+1$
c_4	$\sum_{j=1}^{n-1} (n-j+1)-1$
c_5	x
c_6	$n-1$

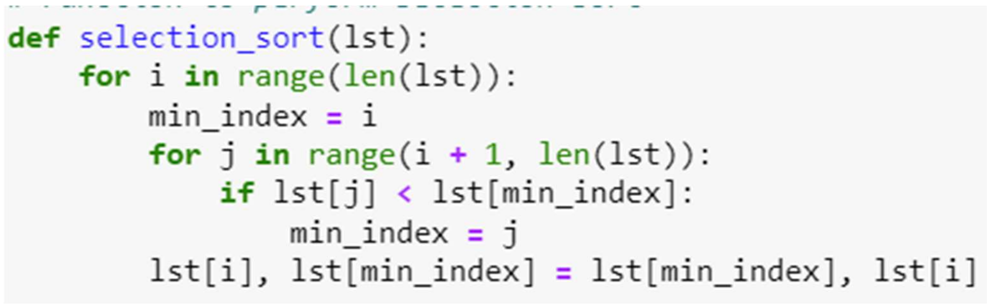
65 / 85

$$T(n) = c_1 \cdot n + c_3(n-1) + c_3\left(\sum_{j=1}^{n-1} n-j+1\right) + c_4\left(\sum_{j=1}^{n-1}(n-j+1)-1\right) + x c_5 + (n-1)c_6$$

Best Case: $x=0$
(Already Sorted)

Worst Case: $x = \sum_{j=1}^{n-1} (n-j+1)-1$

- **When to Use**: Selection Sort is suitable for small data sets and is generally more efficient than Bubble Sort.
- **Why**: It consistently performs the same number of swaps regardless of the initial order of the elements.

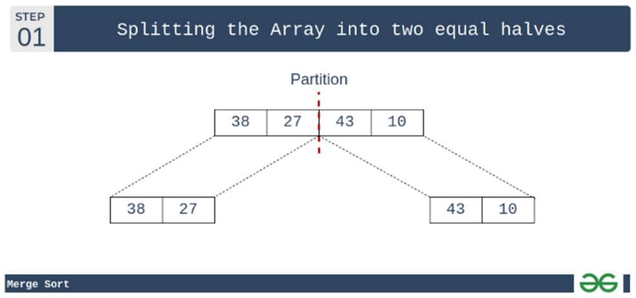


```python
def selection_sort(lst):
    for i in range(len(lst)):
        min_index = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[min_index]:
                min_index = j
        lst[i], lst[min_index] = lst[min_index], lst[i]
```

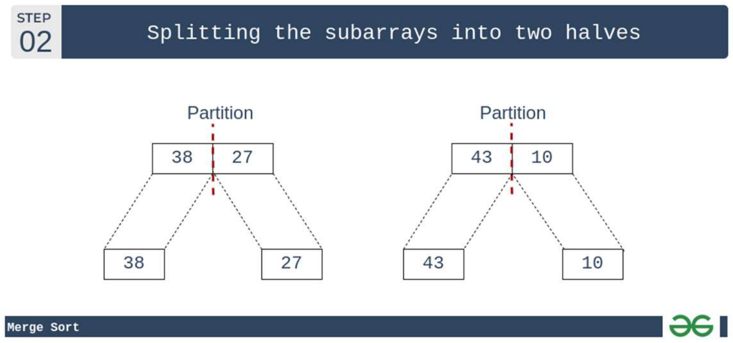
MERGE SORT

Lets consider an array arr[] = {38, 27, 43, 10}

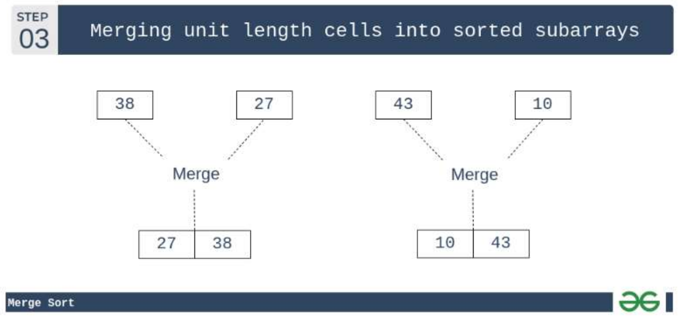
1) Initially divide the array into two equal halves:



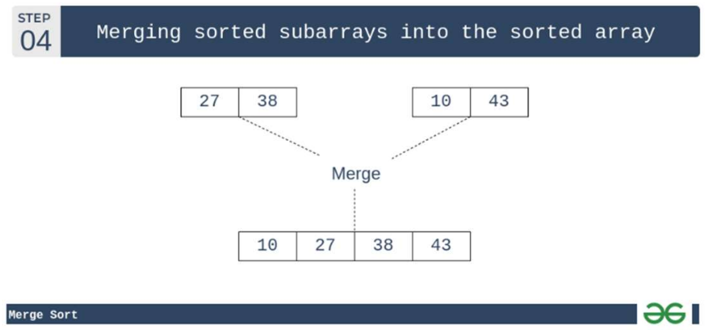
2) These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



3) These sorted subarrays are merged together, and we get bigger sorted subarrays.

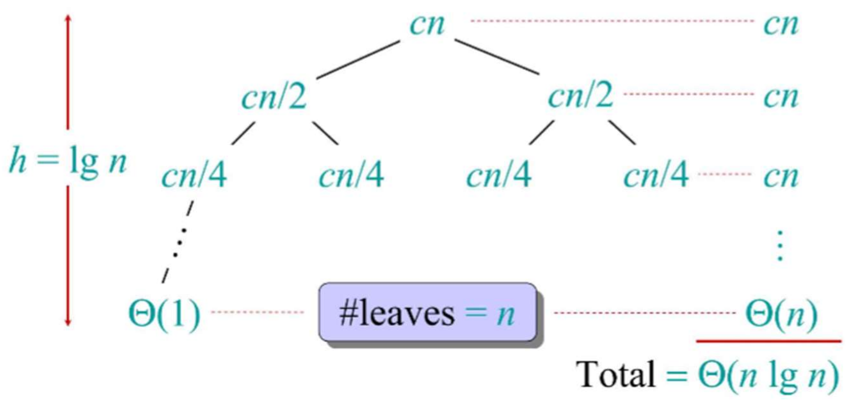


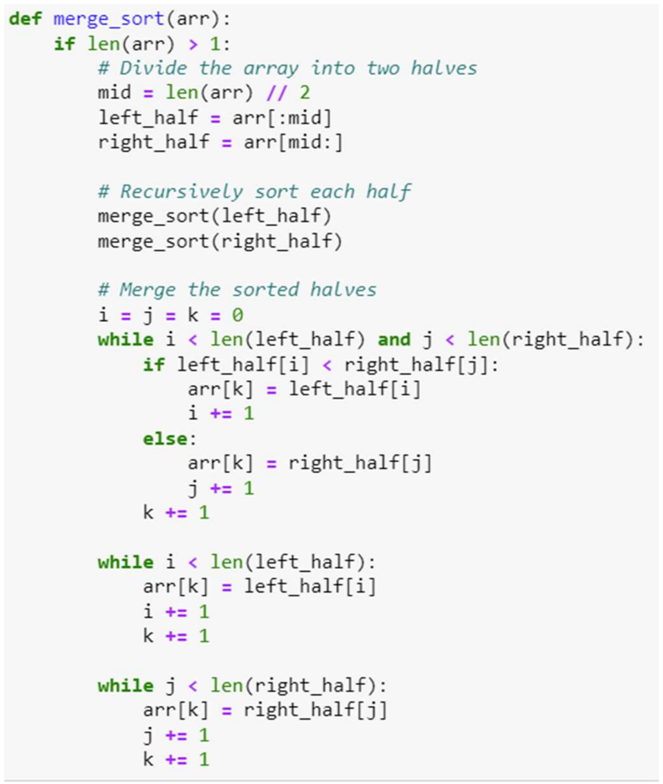
4) This merging process is continued until the sorted array is built from the smaller subarrays.



Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$





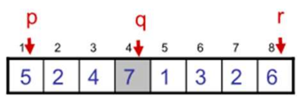
```python
def merge_sort(arr):
    if len(arr) > 1:
        # Divide the array into two halves
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort each half
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```



Alg.: MERGE-SORT(*A*, p, r)

```
if p < r                          ▷ Check for base case
  then q ← ⌊(p + r)/2⌋            ▷ Divide
       MERGE-SORT(A, p, q)        ▷ Conquer
       MERGE-SORT(A, q + 1, r)    ▷ Conquer
       MERGE(A, p, q, r)          ▷ Combine
```

- Initial call: MERGE-SORT(*A*, 1, n)

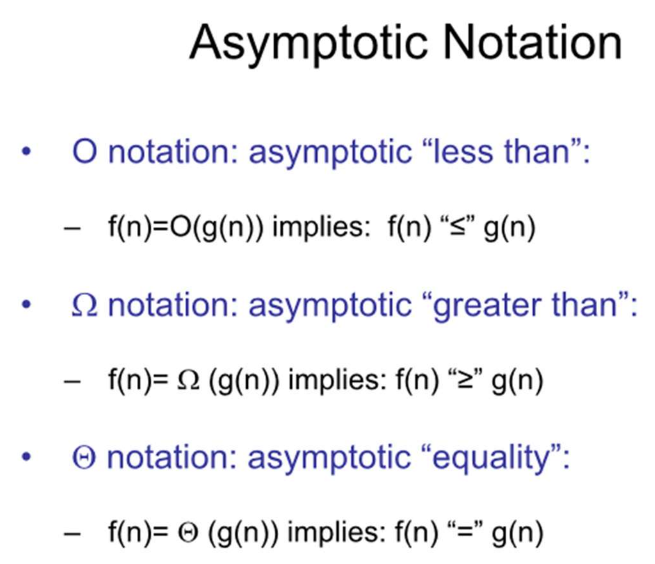
- **When to Use**: Merge Sort is efficient for large data sets and is a good choice for sorting linked lists.
- **Why**: It's a divide-and-conquer algorithm that guarantees a stable sort and has a consistent O(n log n) time complexity.

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: $\Theta(n\log n)$

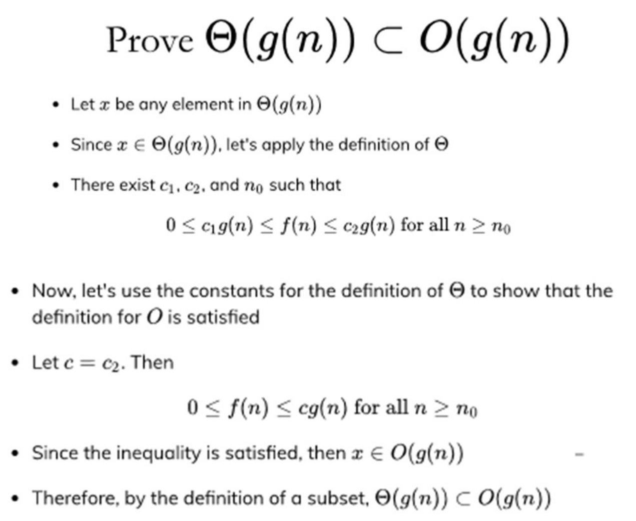
- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

BIG O Notation



Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)



$$\text{Prove } \Theta(g(n)) \subset O(g(n))$$

- Let x be any element in $\Theta(g(n))$
- Since $x \in \Theta(g(n))$, let's apply the definition of Θ
- There exist c_1, c_2, and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- Now, let's use the constants for the definition of Θ to show that the definition for O is satisfied
- Let $c = c_2$. Then

$$0 \le f(n) \le c g(n) \text{ for all } n \ge n_0$$

- Since the inequality is satisfied, then $x \in O(g(n))$
- Therefore, by the definition of a subset, $\Theta(g(n)) \subset O(g(n))$