CSE 221 Algorithms: Ch2

Instructor: Saraf Anika
CSE, SoSET, EDU
Spring 2022

Asymptotic Notations

- O-**notation:**

  When we have only an ***asymptotic upper bound***, we use O-notation. For a given function *g(n)*, we denote by *O(g(n))* (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

  $$O(g(n)) = \{\, f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c*g(n) \text{ for all } n \ge n_0 \}$$

- $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots\ldots < 2^n < 3^n \ldots\ldots < n^n$

  Lower Bound Avg. Bound Upper Bound

- ***Example:***

  $f(n) = 2n+3$ ……………………….[$f(n) = O(n)$]

  $2n+3 <= 2n^2+3n^2$

  $\underbrace{2n+3}_{f(n)} <= \underbrace{5}_{c}\underbrace{n^2}_{g(n)}$ ……………………….[$n>=1$]

  $\boldsymbol{f(n) = O(n^2)}$

Asymptotic Notations Cont.

- **Ω-notation:**
  When we have only an ***asymptotic lower bound***, we use Ω-notation. For a given function *g(n)*, we denote by *Ω(g(n))* (pronounced "big omega of g of n" or sometimes just "omega of g of n") the set of functions

  $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le c*g(n) \le f(n)$ for all $n \ge n_0\}$

- $\mathbf{1 < logn < \sqrt{n} < n < nlogn < n^2 < n^3 < \ldots\ldots\ldots < 2^n < 3^n \ldots\ldots < n^n}$

  Lower Bound Avg. Bound Upper Bound

- ***Example:***

  f(n) = 2n+3 ……………………….[f(n) = Ω(n)]

  $\underbrace{2n+3}_{f(n)} >= \underbrace{1}_{c}*\underbrace{logn}_{g(n)}$……………………[for all n >= 1]

  f(n) = Ω(logn)

Asymptotic Notations Cont.

- **Θ Notation:**

When we have only an ***asymptotic average bound***, we use Θ-notation. For a given function *g(n)*, we denote by *Θ(g(n))* (pronounced "big theta of g of n" or sometimes just "theta of g of n") the set of functions

$\Theta(g(n)) = \{ f(n)$: there exist positive constants $c1, c2$ and n_0 such that $0 \le c1*g(n) \le f(n) \le c2*g(n)$ for all $n \ge n_0\}$

- $\underbrace{1 < \log n < \sqrt{n} <}_{\text{Lower Bound}} \underbrace{n}_{\text{Avg. Bound}} \underbrace{< n\log n < n^2 < n^3 < \ldots\ldots\ldots\ldots < 2^n < 3^n \ldots\ldots < n^n}_{\text{Upper Bound}}$

- ***c1*g(n) <= f(n) <= c2*g(n)***
- ***Example:***

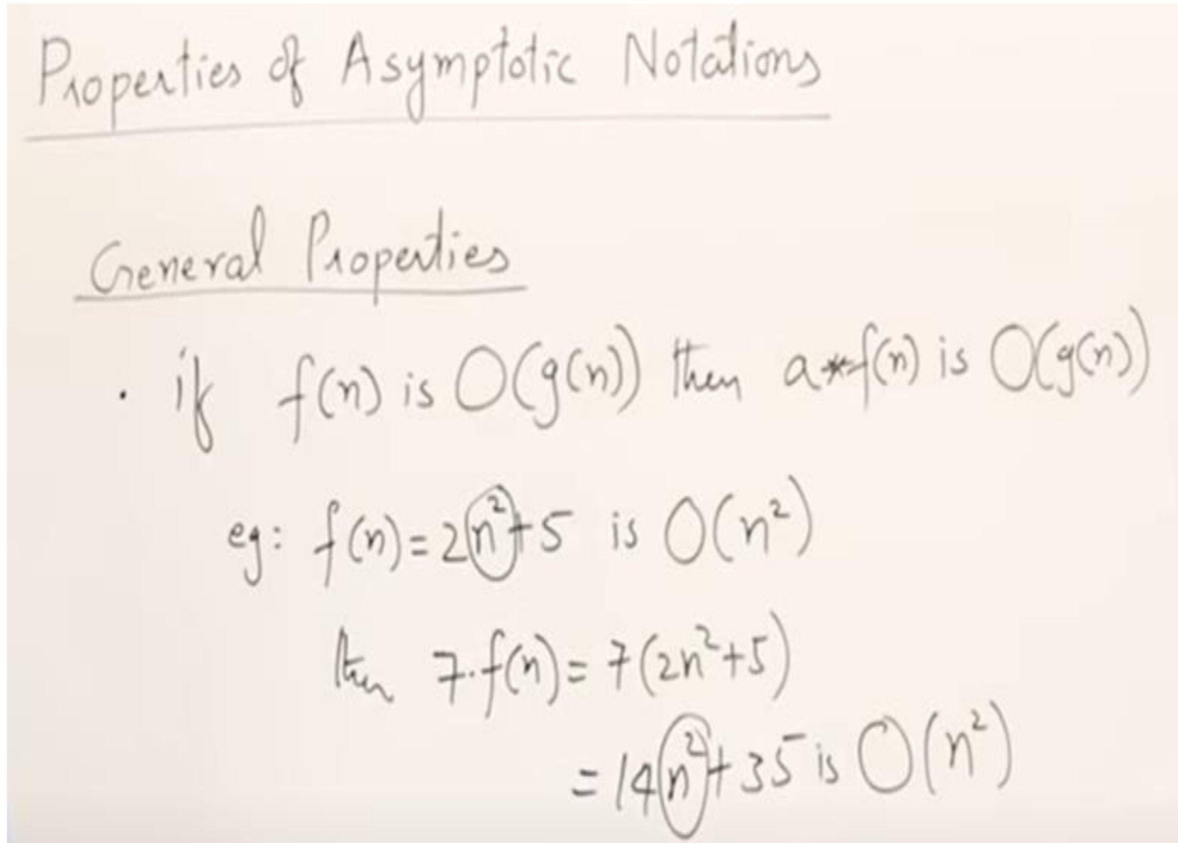
f(n) = 2n+3

*1*n <= 2n+3 <= 5*n*

f(n) = Θ(n)

Properties of Asymptotic Function

- General Properties:



Properties of Asymptotic Function Cont.

- Reflexive

Reflexive

if $f(n)$ is given then $f(n) = O(f(n))$

eg: $f(n) = n^2$ $\quad O(n^2)$

- Transitive

Transitive

if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$

then $f(n) = O(h(n))$

eg: $f(n) = n$ $\quad g(n) = n^2$ $\quad h(n) = n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$

then n is $O(n^3)$

Properties of Asymptotic Function Cont.

- Symmetric

Symmetric

if $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$

eg: $f(n) = n^2 \quad g(n) = n^2$

$f(n) = \Theta(n^2)$

$g(n) = \Theta(n^2)$

- Transpose Symmetric

Transpose Symmetric

if $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$

eg: $f(n) = n \quad g(n) = n^2$

then n is $O(n^2)$ and

n^2 is $\Omega(n)$

Properties of Asymptotic Function Cont.

- Example:

if $f(n) = O(g(n))$

and $f(n) = \Omega(g(n))$

$$g(n) \leq f(n) \leq g(n)$$

$$\therefore f(n) = \Theta(g(n))$$

Comparison of functions(Time/Space)

- First Method
  - Smaller/Greater/Equal

n	n^2	$<$	n^3
2	$2^2=4$		$2^3=8$
3	$3^2=9$		$3^3=27$
4	$4^2=16$		$4^3=64$

- Second Method
  - Using logarithm

$n^2 \qquad n^3$

Apply Log on Both sides

$\log n^2 \qquad \log n^3$

$2\log n < 3\log n$

Comparison of functions(Time/Space) Cont.

- Logarithm Laws : $a^{\log_c b} = b^{\log_c a}$

Logarithmic laws

Products:	$\log_b mn$	=	$\log_b m + \log_b n$
Ratios:	$\log_b \frac{m}{n}$	=	$\log_b m - \log_b n$
Powers:	$\log_b n^p$	=	$p \log_b n$
Roots:	$\log_b \sqrt[q]{n}$	=	$\frac{1}{q} \log_b n$
Change of bases:	$\log_b n$	=	$\log_a n \log_b a$

Exponent and Logarithm Rules

$(a > 0, a \neq 1)$

	Exponent	Logarithm $(m, n > 0)$
Product Rule	$a^m \cdot a^n = a^{m+n}$	$\log_a(m \cdot n) = \log_a m + \log_a n$
Quotient Rule	$\frac{a^m}{a^n} = a^{m-n}$	$\log_a\left(\frac{m}{n}\right) = \log_a m - \log_a n$
Power Rule	$(a^m)^n = a^{m \cdot n}$	$\log_a(m^n) = n \cdot \log_a m$

- Examples:

$f(n) = n^2 \log n > g(n) = n(\log n)^{10}$

Apply Log

$\log[n^2 \log n]$ $\quad$ $\log[n(\log n)^{10}]$

$\log n^2 + \log\log n$ $\quad$ $\log n + \log(\log n)^{10}$

$2\log n + \log\log n$ $\quad$ $\log n + 10\log\log n$

$f(n) = 3n^{\sqrt{n}}$ $\quad$ $g(n) = 2^{\sqrt{n}\log_2 n}$

$3n^{\sqrt{n}}$ $\quad$ $2^{\log_2(n^{\sqrt{n}})}$

$3n^{\sqrt{n}}$ $\quad$ $(n^{\sqrt{n}})^{\log_2 2}$

$3\underline{n}^{\sqrt{n}}$ $\quad$ $\underline{n}^{\sqrt{n}}$

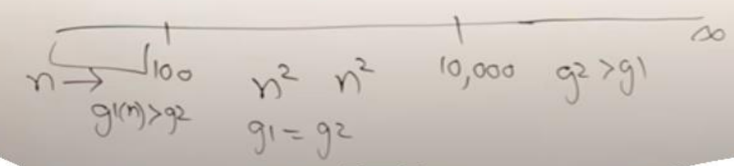
Comparison of functions(Time/Space) Cont.

- g2(n) is going to be always greater than g1(n) because of the last condition of n value.

$$g1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$$

$$g2(n) = \begin{cases} n^2 & n < 10{,}000 \\ n^3 & n \geq 10{,}000 \end{cases}$$

$g2 > g1$



- Find out the True/False.

True or False

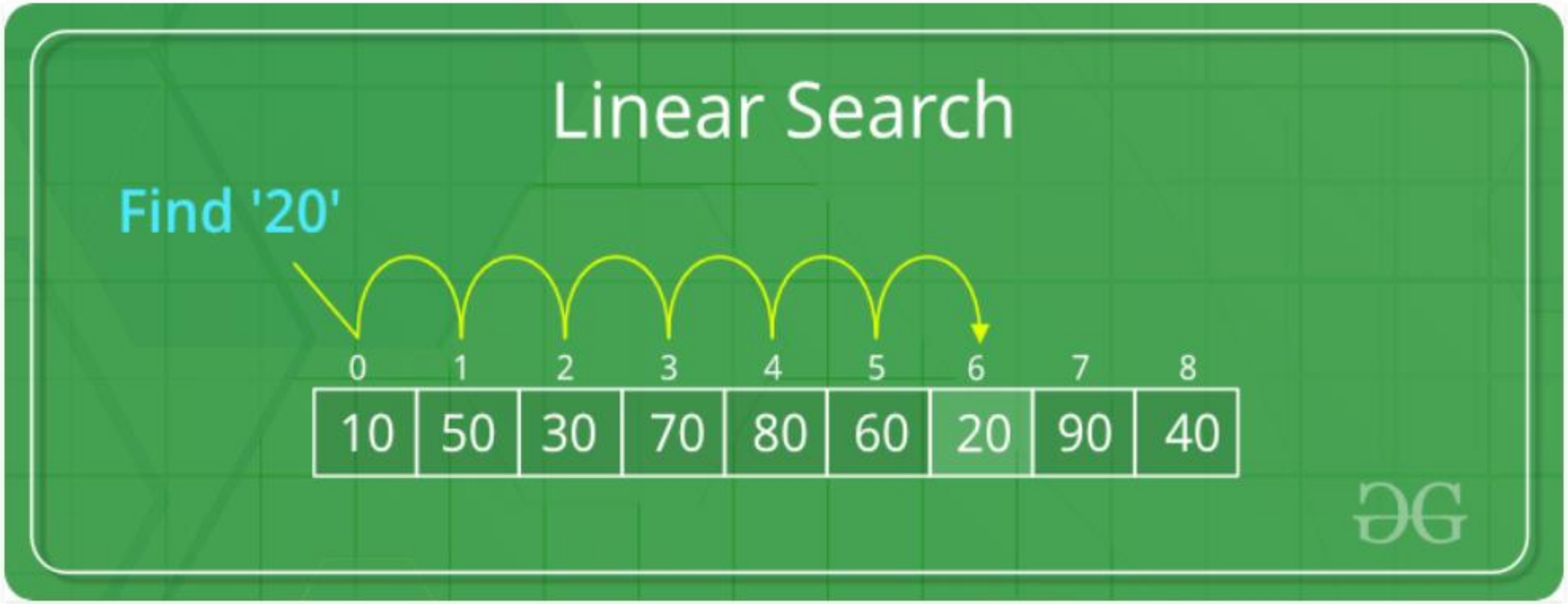
1. $(n+k)^m = \Theta(n^m)$
2. $2^{n+1} = O(2^n)$
3. $2^{2n} = O(2^n)$
4. $\sqrt{\log n} = O(\log \log n)$
5. $n^{\log n} = O(2^n)$

Best, Worst & Average Case

- Linear Search
- Binary Search Tree

Linear Search

- A simple approach to do a linear search is, i.e
  - Start from the leftmost element of ***arr[]*** and one by one compare ***x(key element)*** with each element of ***arr[]***
  - If ***x*** matches with an element, return the index.
  - If ***x*** doesn't match with any of elements, return ***-1***.
- Example:



Linear Search Cont.

- Best Case: Searching key element present at first index.
  - Best Case Time: will be constant. $B(n) = O(1)$

- Worst Case: Searching key element present at last index.
  - Worst Case Time: will be n. $W(n) = O(n)$

- Average Case: (All possible case time/number of cases). Rarely being used.
  - Average Case Time: $(1+2+3+....+n)/n = (n(n+1)/2)/n = (n+1)/2$. $A(n) = (n+1)/2$

Linear Search Cont.

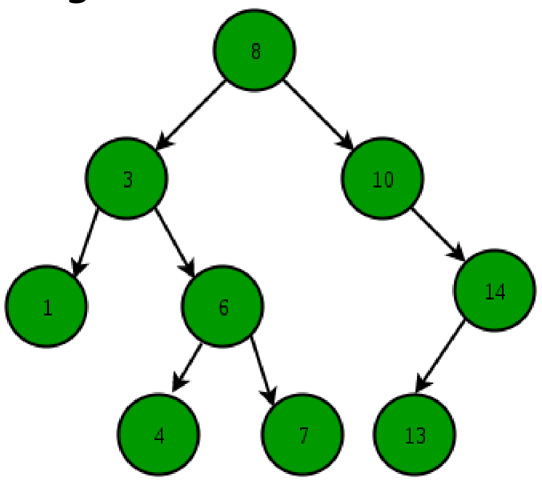
- Asymptotic Notations:

$B(n) = 1$
$B(n) = O(1)$
$B(n) = \Omega(1)$
$B(n) = \Theta(1)$

$\omega(n) = n$
$\omega(n) = O(n)$
$\omega(n) = \Omega(n)$
$\omega(n) = \Theta(n)$

Binary Search Tree

- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.
  - Height of the tree is ***h = logn.***



Binary Search Tree Cont.

- Best Case: Searching for the root element.
  - Best Case Time: will be constant. $B(n) = O(1)$

- Worst Case: Searching for element present at the leaf.
  - Worst Case Time: Depends on the height of the Binary Search Tree. So, $W(n) = h$.

- We can apply the same asymptotic notations here like Linear Search.

Sorting: Bubble Sort, Insertion Sort, Selection Sort

- **Selection sort**: repeatedly pick the smallest element to append to the result.
  - Divides the array into two parts: sorted (left) and unsorted (right) subarray.
  - Selects the smallest element from unsorted subarray and places in the first position of that subarray (ascending order). Search and swap.

```
SelectionSort (Arr, N) // Arr is an array of size N.
{
    For ( I:= 1 to (N-1) ) // N elements => (N-1) pass
    {
    // I=N is ignored, Arr[N] is already at proper place.
    // Arr[1:(I-1)] is sorted subarray, Arr[I:N] is undorted subarray
    // smallest among { Arr[I], Arr[I+1], Arr[I+2], ..., Arr[N] } is at place min_index

        min_index = I;
        For ( J:= I+1 to N ) // Search Unsorted Subarray (Right lalf)
        {
            If ( Arr [J] < Arr[min_index] )
                min_index = J; // Current minimum
        }
        // Swap I-th smallest element with current I-th place element
        If (min_Index != I)
              Swap ( Arr[I], Arr[min_index] );

    }
}
```

Selection Sort (Example)

- **Selection sort** is an in-place algorithm.

It performs all computation in the original array and no other array is used. Hence, the **space complexity** works out to be O(**1**).

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	$O(1)$

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Sorting:Cont.

- **Insertion sort**: repeatedly add new element to the sorted result. Shifting and sorting
  - Take the first element as a sorted sub-array.
  - Insert the second element into the sorted sub-array (shift elements if needed).
  - Insert the third element into the sorted sub-array.
  - Repeat until all elements are inserted.

```
InsertionSort (Arr, N) // Arr is an array of size N.
{
    For ( I:= 2 to N ) // N elements => (N-1) pass
    {
    // Pass 1 is trivially sorted, hence not considered
    // Subarray { Arr[1], Arr[2], ..., Arr[I-I] } is already sorted

        insert_at = I; // Find suitable position insert_at, for Arr[I]
        // Move subarray Arr [ insert_at: I-1 ] to one position right

        item = Arr[I]; J=I-1;
        While ( J ? 1 && item < Arr[J] )
        {
                Arr[J+1] = Arr[J]; // Move to right
                // insert_at = J;
                J--;
            }
            insert_at = J+1; // Insert at proper position
            Arr[insert_at] = item; // Arr[J+1] = item;
        }
    }
}
```

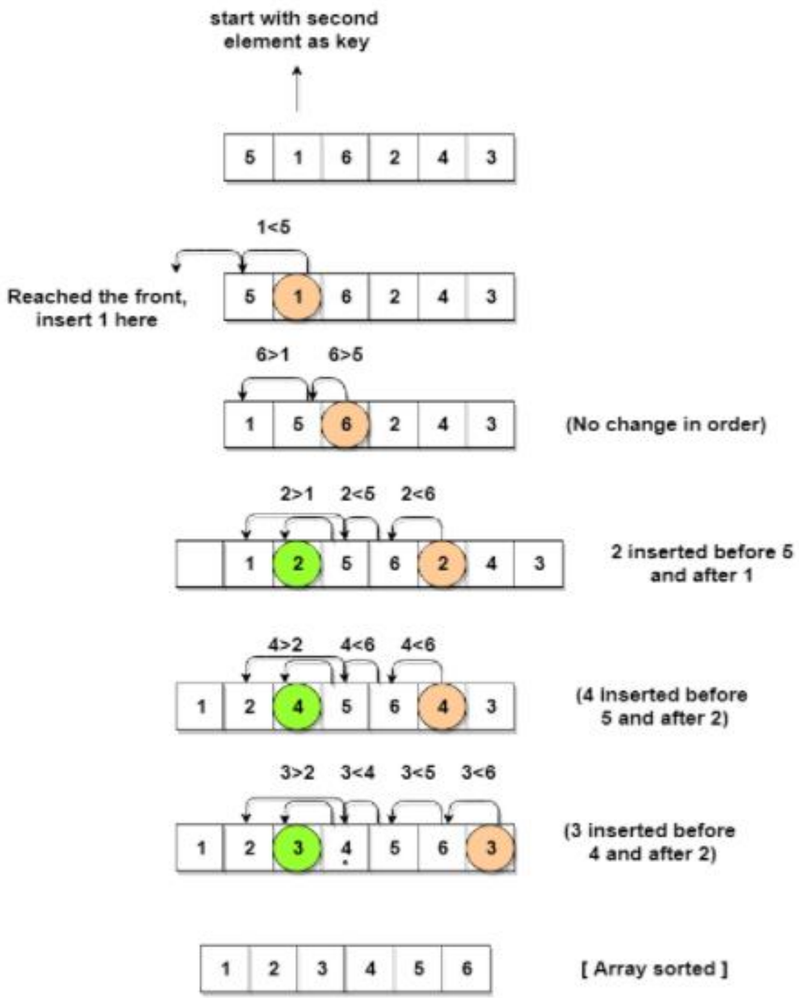
Insertion sort(example)

- **Insertion sort** is an in-place algorithm.

It performs all computation in the original array and no other array is used. Hence, the **space complexity** works out to be O(**1**).

- The Best Case Time Complexity will be O(n) instead of $O(n^2)$

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)



Sorting:Cont.

- **Bubble sort**: repeatedly compare neighbor pairs and swap if necessary.

```
BubbleSort (Arr, N) // Arr is an array of size N.
{
    For ( I:= 1 to (N-1) ) // N elements => (N-1) pass
    {
    // Swap adjacent elements of Arr[1:(N-I)]such that
    // largest among { Arr[1], Arr[2], ..., Arr[N-I] } reaches to Arr[N-I]
        noSwap = true; // Check occurrence of swapping in inner loop
        For ( J:= 1 to  (N-I) ) // Execute the pass
        {
            If ( Arr [J] > Arr[J+1] )
            {
                Swap( Arr[j], Arr[J+1] );
                noSwap = false;
            }
        }
        If (noSwap) // exit the loop
            break;
    }
}
```

Bubble sort(example)

Time Complexity	
Best	O(n)
Worst	O(n^2)
Average	O(n^2)
Space Complexity	O(1)