

**Data structures** are ways to store and organize data in a way that can be access and modify

**Technique** are used to help develop algorithms that are correct and efficient

**Hard problems** aka np complete problem does not exist an algorithm that is efficient yet. The best we can do is identify it is a np complete problem and produce a good but not most efficient algorithm for the problem

**Parallelism** is dividing each task into subtask that run the same time to complete the task faster. It should be used since a core is clocked at a certain speed and if the core is too high, the chip will melt. This is why chips have multiple cores. We can develop “multithreaded” algorithms to take advantage of multi core processors.

**Efficiency** is important when choosing or designing the correct algorithm for a specific task. For instance insertion sort vs merge sort. Insertion sort runs on  $n^2$  while merge sort runs on  $n \log n$ . Insertion sort will be faster for smaller inputs  $n$  but will be exponentially slower than merge sort as the values of  $n$  increases. This is demonstrated where two computers sort an input size of 10 million. Computer 1 at 10 billion operation per second which is 1000x faster than computer 2 is program by an expert programmer who codes in machine language insert sort. The run time is  $2 * n^2$ . Computer 2 is program by a novice on an outdated language with a garbage compiler. The run time is  $50 * n \log n$ . Computer 1 running input  $n$  would be  $2 * 10 \text{ mil}^2 / 10 \text{ bil} \approx 20,000 \text{ secs}$ . Computer 2 running input  $n$  would be  $50 * 10 \text{ mil} \log 10 \text{ mil} / 10 \text{ mil} \approx 1163 \text{ secs}$ . Computer 2 is 17 times faster despite being at a massive disadvantage. This illustrates that choosing the correct algorithm for the correct task matters.

### 1-1 Comparison of running times

For each function  $f(n)$  and time  $t$  in the following table, determine the largest size  $n$  of a problem that can be solved in time  $t$ , assuming that the algorithm to solve the problem takes  $f(n)$  microseconds.

	1 Second	1 minute	1 hour	1 day	1 month	1 year	1 century
<b>lg n</b>	$2^{10^6}$	$2^{6 \cdot 10^7}$	$2^{3.6 \cdot 10^9}$	$2^{8.64 \cdot 10^{10}}$	$2^{2.59 \cdot 10^{12}}$	$2^{3.15 \cdot 10^{13}}$	$2^{3.15 \cdot 10^{15}}$
<b><math>\sqrt{n}</math></b>	$10^{12}$	$3.6 \cdot 10^{15}$	$1.3 \cdot 10^{19}$	$7.46 \cdot 10^{21}$	$6.72 \cdot 10^{24}$	$9.95 \cdot 10^{26}$	$9.95 \cdot 10^{30}$
<b>n</b>	$10^6$	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.64 \cdot 10^{10}$	$2.59 \cdot 10^{12}$	$3.15 \cdot 10^{13}$	$3.15 \cdot 10^{15}$
<b>n lg n</b>	$6.24 * 10^4$	$2.8 \cdot 10^6$	$1.33 \cdot 10^8$	$2.76 \cdot 10^9$	$7.19 \cdot 10^{10}$	$7.98 \cdot 10^{11}$	$6.86 \cdot 10^{13}$
<b><math>n^2</math></b>	$10^3$	7745	60000	293938	1609968	5615692	56156922
<b><math>n^3</math></b>	$10^2$	391	1532	4420	13736	31593	146645

	1 Second	1 minute	1 hour	1 day	1 month	1 year	1 century
$2^n$	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

## Mathematic recap

**Summation:** When there is a loop such as a for or while loop, it is represented as a summation

$$\lim_{n \rightarrow \infty} \sum_{k=1}^{n=\infty} a_k$$

If the limit does not exist it diverges otherwise it converges

$$\lim_{n \rightarrow \infty} \sum_{k=1}^{n=\infty} |a_k|$$

This limit converges if some is finite

### Linearity:

1. You can use linearity to split summation into two

$$\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

2. You can also use linearity to bring out asymptotic notations

$$\sum_{k=1}^n \Theta(f(k)) = \Theta\left(\sum_{k=1}^n f(k)\right)$$

## Arithmetic Series

The summation

$$\sum_{k=1}^n k = 1 + 2 + \dots + n ,$$

is an **arithmetic series** and has the value

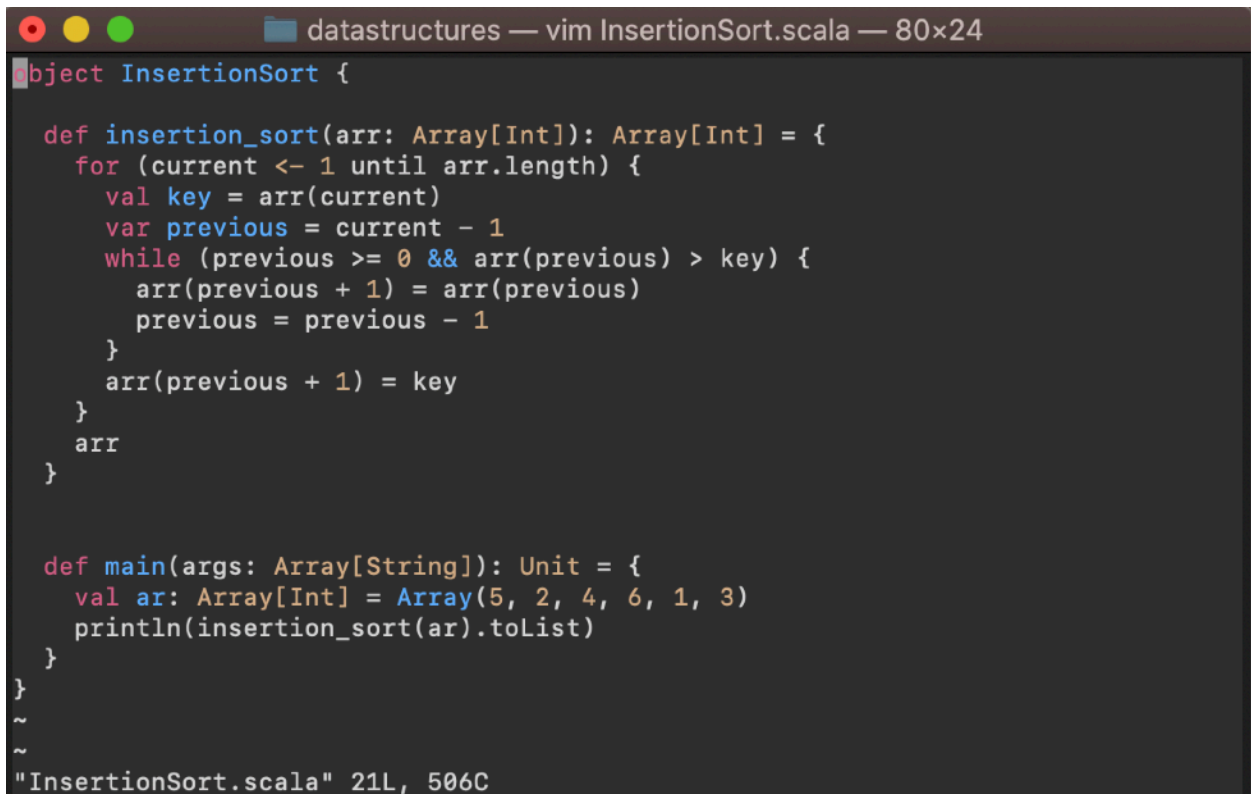
$$\begin{aligned} \sum_{k=1}^n k &= \frac{1}{2}n(n+1) \\ &= \Theta(n^2) . \end{aligned}$$

**Insertion Sort:** current value goes from left to right but insertion to sorted sequence goes from right to left. Keep decrementing previous index until current value is greater than previous value. Place the current value 1 index above previous value index where the value is greater.

### Pseudo code

1. Loop from 1 to the length of the array
2. Set the key = array[loop value]
3. Set a new value = loop value - 1
4. While loop check if new value is greater than 0 and array[new value] is greater than key value #2
5. Entering the loop set Array[new value index + 1] = new index value
6. Decrease new value by 1
7. Exit loop set Array[new value index + 1] = key #2

### Scala implementation

A screenshot of a vim editor window. The title bar shows a file icon, the text "datastructures — vim InsertionSort.scala", and the window size "80x24". The editor has a dark background with light-colored text. The code is written in Scala and implements the Insertion Sort algorithm. It defines an object "InsertionSort" with two methods: "insertion\_sort" and "main". The "insertion\_sort" method takes an array of integers and returns a sorted array. The "main" method creates an array [5, 2, 4, 6, 1, 3] and prints the result of "insertion\_sort". At the bottom of the window, the status line shows the file name and line/column number: "InsertionSort.scala" 21L, 506C.

```
object InsertionSort {  
  
  def insertion_sort(arr: Array[Int]): Array[Int] = {  
    for (current <- 1 until arr.length) {  
      val key = arr(current)  
      var previous = current - 1  
      while (previous >= 0 && arr(previous) > key) {  
        arr(previous + 1) = arr(previous)  
        previous = previous - 1  
      }  
      arr(previous + 1) = key  
    }  
    arr  
  }  
  
  def main(args: Array[String]): Unit = {  
    val ar: Array[Int] = Array(5, 2, 4, 6, 1, 3)  
    println(insertion_sort(ar).toList)  
  }  
}  
~  
~  
"InsertionSort.scala" 21L, 506C
```

## Example output

```
felixchen@Felixs-MacBook-Pro datastructures % scala InsertionSort  
List(1, 2, 3, 4, 5, 6)
```

### Determine correctness of algorithm:

1. Initialization: It is true prior to loop
2. Maintenance: if #1 is true, it remains true as it enters next iteration
3. Termination: When the loop ends, the invariant (never changing function) outputs a useful property to show the algorithm is correct

Note: initialization is like a base case, maintenance is like inductive step, and termination stops the infinite inductive steps

### Correctness of Insertion sort algorithm:

Initialization: since sequence begins with only 1 value, it is technically a sorted sequence

Maintenance: since we are inserting each current value into a sequential

Termination: The condition for the for loop to end is when it reaches to the length of the array.

Current value increments by 1 until it is equal to to length of array. The array return has the same 0...n indices but values are in sorted order. Thus the algorithm is correct

### Exercise 2.11

Arr = (31; 41; 59; 26; 41; 58)

Current starts at 41 and subarray starts at 31  
31

41 > 31  
31 41

59 > 41  
31 41 59

26 !> 59 -> 26 !> 41 -> 26 !> 31  
31 41 59 26 -> 31 41 26 59 -> 31 26 41 59 -> 26 31 41 59

58 !> 59  
26 31 41 59 58 -> 26 31 41 58 59

Returns sorted array  
26 31 41 58 59

## Exercise 2.12

### Scala implementation



```
object InsertionSortB {  
  def decrease_insertion_sort (arr: Array[Int]): Array[Int] = {  
    for (curr <- 1 until arr.length) {  
      val key = arr(curr)  
      var prev = curr - 1  
      while (prev >= 0 && arr(prev) < key) {  
        arr(prev + 1) = arr(prev)  
        prev -= 1  
      }  
      arr(prev + 1) = key  
    }  
    arr  
  }  
  
  def main(args: Array[String]): Unit = {  
    val ar: Array[Int] = Array(5, 2, 4, 6, 1, 3)  
    println(decrease_insertion_sort(ar).toList)  
  }  
}
```

### Example output

```
felixchen@Felixs-MacBook-Pro datastructures % scala InsertionSortB  
List(6, 5, 4, 3, 2, 1)
```

Analyzing Algorithms:

Some computers can calculate  $2^k$  by shift 1 bit to the left. Since this is done in constant time,  $2^k$  can be calculated in constant time.

Input size depends on the problem being studied. Sometimes there more than 1 input for instance a graph might have vertices and edges

Runtime: the number of steps it takes to execute a program

INSERTION-SORT( $A$ )	<i>cost</i>	<i>times</i>
1 <b>for</b> $j = 2$ <b>to</b> $A.length$	$c_1$	$n$
2 $key = A[j]$	$c_2$	$n - 1$
3     // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$ .	0	$n - 1$
4 $i = j - 1$	$c_4$	$n - 1$
5 <b>while</b> $i > 0$ and $A[i] > key$	$c_5$	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	$c_8$	$n - 1$

Note: entering the while loop everything is  $n - 1$ . The while loop is summation from 2 to  $n$  because of the for loop. Every instruction inside the for loop will be summed up. Each

$$\begin{aligned}
 T(n) = & c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\
 & + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1) .
 \end{aligned}$$

instruction inside will be sigma 2 to  $n$  ( $t - 1$ ).

Note: For best case scenario, everything is already sorted, thus it will not enter while loop since second condition can be false. Therefore everything inside the loop will have a cost of 0.  
 Linear function:  $an + b$

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Worst case cost if array is full sorted in reverse order

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

and

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

Quadratic function:  $an^2 + bn + c$

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Worst Case and average case analysis:

Note:

1. Worst case gives us a guarantee that the algorithm will never take any longer
2. Worst case occurs fairly often for some algorithms. For instance search a database with an absent value

### 3. Average case sometimes is just as bad as worst case

Average case runtime will be used in probabilistic analysis technique. This technique will be used to analyze randomized algorithm

### Order Of Growth

Usually convert all the costs into coefficient ex.  $an^2 + bn + c$ . We can go even further by ignoring the lower powers since they are rather insignificant so we have  $an^2$ . Finally we can ignore the coefficient all together and have  $n^2$ . The worst case for insertion sort is  $\Theta(n^2)$

Note: Usually higher order of growth functions run faster with smaller inputs. For instance  $\Theta(n^3)$  runs faster than  $\Theta(n^2)$  for smaller inputs but exponentially slower as inputs get larger.

Exercise 2.21

Express the function  $n^3/1000 - 100n^2 - 100n + 3$  in terms of  $\Theta$ -notation.

$\Theta(n^3)$

### Designing Algorithms:

For insertion sort we use **incremental** approach where we insert into a sorted array

### Divide and Conquer:

**Divide:** problem into subproblems that are smaller instances of the original problem

**Conquer:** the subproblems recursively. If the subproblems are small enough, solve it directly

**Combine:** the solutions to the subproblem into the solution for the original problem

### Designing Merge Sort:

**Divide:** Divide the  $n$ -element sequence to be sorted in two subsequences of  $n/2$  elements each

**Conquer:** Sort the two subsequences recursively using merge sort

**Combine:** Merge the two sorted subsequences to produce the sorted answer

\* In most cases there will be multiple two subsequences and multiple merges



Note:  $p, q, r$  are indices beginning, average, last

Merge: Since  $p \leq q < r$ , we can merge the two sorted subsequences  $A[p \dots q]$  with  $A[q + 1 \dots r]$  into  $A[p \dots r]$

\*merge takes  $\theta(n)$  time since  $n = r - p + 1$  is total number of elements being merged.

**Merge Sort:** In terms of card analogy, we have left hand cards and right hand cards that hold roughly half of the deck  $\pm 1$ . We take 1 card from the left and right hand and compare the value. Which ever is lower gets inserted into the new sorted deck of cards. Gradually the deck of cards gets bigger where the bottom half is sorted and top half is sorted until the entire deck is sorted

**Initialization:** Since  $k == p$  prior to loop the subarray is empty. And since  $i, j == 0$ , both left hand and right hand decks are the lowest values

**Maintenance:** Since the if else condition is  $L[i] \leq R[j]$ , only the smallest value compared from the  $L[i]$  and  $R[j]$  will be inserted into the subarray. The index  $i$  will be incremented whenever a card from  $L[i]$  is inserted to the subarray and index  $j$  will be incremented whenever a card from  $R[j]$  is inserted into subarray. This makes sure that we're inserting in a sorted order to the subarray as we go through the left hand and right hand cards.

**Termination:** The condition for loop to end is  $k = r + 1$ . All but two sentinel cards have been inserted back into array

```

datastructures — vim MergeSort.scala — 114x57
object MergeSort {

  def sort(A:Array[Int], l:Int, r:Int): Array[Int] = {
    if (l < r) {
      val m: Int = math.floor((l + r) / 2).toInt
      sort(A, l, m)
      sort(A, m + 1, r)
      merge(A, l, m, r)
    }
    A
  }

  def merge(A: Array[Int], l: Int, m: Int, r: Int): Array[Int] = {
    val n1:Int = m - 1 + 1
    val n2:Int = r - m
    val L :Array[Int] = new Array[Int](n1 + 1)
    val R: Array[Int] = new Array[Int](n2 + 1)
    for (i <- 0 until n1) {
      L(i) = A(l + i)
    }
    for (j <- 0 until n2) {
      R(j) = A(m + j + 1)
    }
    L(n1) = Int.MaxValue
    R(n2) = Int.MaxValue
    var i = 0
    var j = 0
    for (k <- 1 to r) {
      if (L(i) <= R(j)) {
        A(k) = L(i)
        i = i + 1
      }
      else {
        A(k) = R(j)
        j = j + 1
      }
    }
    A
  }

  def mergeSort(A: Array[Int]): Array[Int] = {
    sort(A, 0, A.length - 1)
  }

  def main(args: Array[String]): Unit = {
    val a: Array[Int] = Array(2,4,6,1,3,7)
    val b :Array[Int] = Array(2,5,7,1,6,8)
    val c:Array[Int] = Array(2,5,7,1,6,8,2,5,7,1,6,8)
    println(mergeSort(a).toList)
    println(mergeSort(b).toList)
    println(mergeSort(c).toList)
  }

}

-- INSERT --

```

Scala implementation:

Output:

```
felixchen@Felixs-MacBook-Pro datastructures % scala MergeSort
List(1, 2, 3, 4, 6, 7)
List(1, 2, 5, 6, 7, 8)
List(1, 1, 2, 2, 5, 5, 6, 6, 7, 7, 8, 8)
```

### Analysis of Merge Sort:

Divide: division, splitting the subarray into 2 takes constant time so  $D(n) = \Theta(1)$

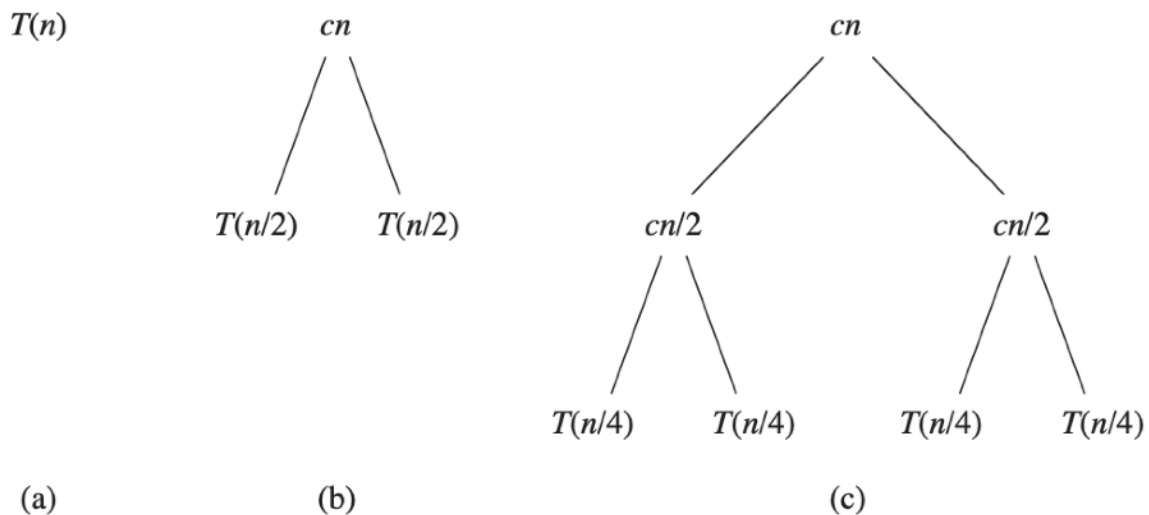
Conquer: Were recursively solving two subproblems, each half the array size  $n/2$  so  $2T(n/2)$

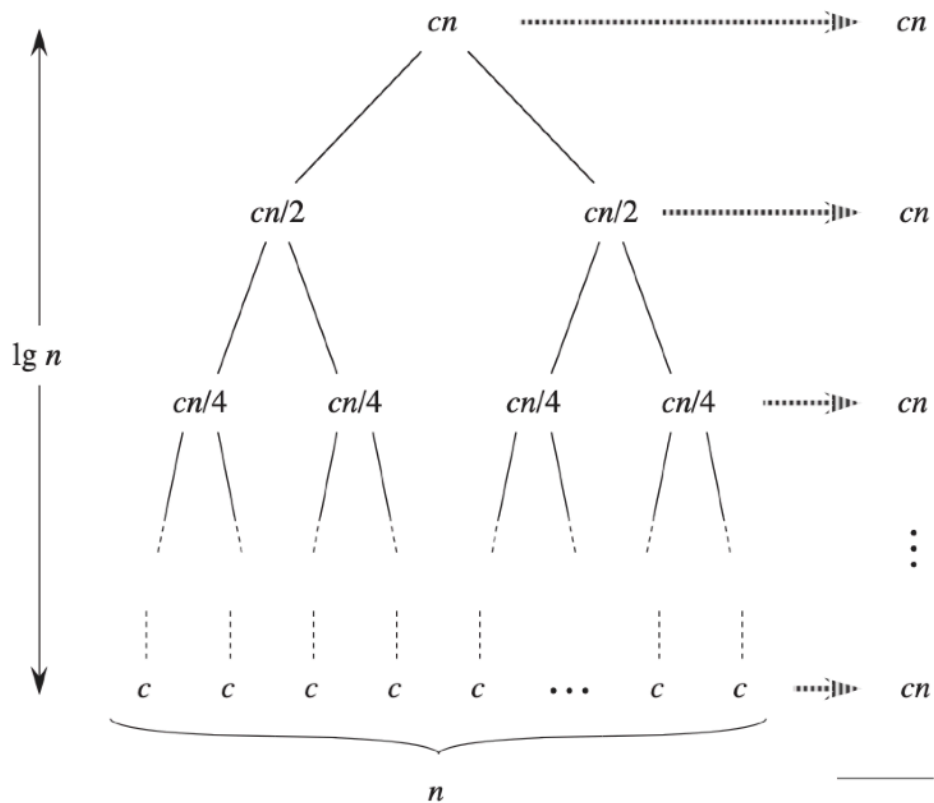
Combine: the loops in merge do at most  $n$  number of steps so  $C(n) = \Theta(n)$

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

\*\*chapter 4 “master theorem” will explain more on why  $T(n) = \Theta(n \lg n)$ \*\*

Recursion tree





(d)

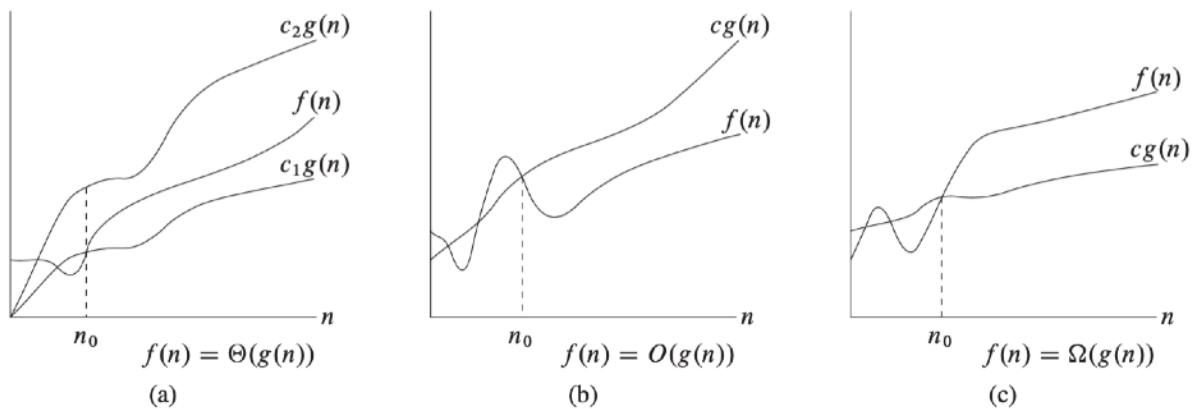
Total:  $cn \lg n + cn$

**Explanation:** The total number of levels is  $\lg n + 1$ . The each level(width/merge) is worth  $cn$ . So  $cn(\lg n + 1) = cn \lg n + cn$ .  $(i + 1) + 1 = \lg 2^{(i + 1)} + 1$

### Growth of Functions (43 -53)

**Asymptotic efficiency:** is when we only look at input size  $n$  that are large enough to make growth of runtime useful

\*\* The notation we use to describe asymptotic notation is in set of natural numbers =  $\{1, 2, 3, 4, 5, 6, \dots\}$  \*\*



**Theta notation:**  $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .^1$

**Explanation:** For theta notation,  $f(n)$  must be sandwich between  $c_1g(n)$  and  $c_2g(n)$  after  $n_0$ .

Instead of writing  $f(n)$  is in set  $\Theta(g(n))$  we can write  $f(n) = \Theta(g(n))$  to express the same thing. This is because after  $n_0$ ,  $f(n)$  is below or above  $c_1g(n)$  and  $c_2g(n)$  thus it is equal to  $g(n)$  to a constant factor. We can also say  $g(n)$  is **asymptotically tight bound** for  $f(n)$

\*\* $\Theta(g(n))$  by definition needs to **asymptotic nonnegative** aka **asymptotic positive** . This means that the function is positive for all sufficiently large inputs  $n$

//pg 46 need clarification

**O notation:**  $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} .$

**Explanation:** For  $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ .  
 $O$  notation,  $f(n)$  must be under the **asymptotic upper bound**  $cg(n)$  after  $n_0$ .

$f(n) = \theta(g(n))$  implies  $f(n) = O(g(n))$  since  $\theta$  is a stronger version of  $O$  since  $O(g(n))$  is a subset of  $\theta(g(n))$ .  $O(g(n))$  is just saying that for any value, no matter the size,  $f(n)$  will never exceed the upper bound.

**Omega notation:**

**Explanation:** For omega notation,  $f(n)$  must be above the **asymptotic lower bound**  $cg(n)$  after  $n_0$ .

**Theorem 3.1:** For any two functions  $f(n)$  and  $g(n)$ , we have  $f(n) = \theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  $f(n) = \omega(g(n))$ .

\*\*  $2n^2 + 3n + 1 = 2n^2 + \theta(n)$  means that  $2n^2 + 3n + 1 = 2n^2 + f(n)$ , where  $f(n)$  is some function in

$\theta(n)$ .

For this instance let 
$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

$f(n) = 3n +$

1, which is indeed  $\theta(n)$ .  $2n^2 + \theta(n)$  is also equal to  $\theta(n^2)$

**o notation:**

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$$

**Explanation:** The main

difference between little  $o$  and big  $O$  is that  $0 \leq f(n) \leq cg(n)$  where  $c > 0$  compared to big  $O$  where it is only some instances. Also,  $f(n)$  becomes rather insignificant compared to  $g(n)$  as input  $n$  approaches infinity, thus not asymptotically tight.

### **Little omega notation:**

**Explanation:** Little omega is to omega what little o is to big O.

//Study this topic more in depth. Skip for now

Three methods for solving recurrence: (83-97)

1. Substitution method
2. Recursion tree
3. Master method