Advanced Data Structures (Fall 2011) Istanbul Technical University Computer Eng. Dept.

Chapter 2: Getting Started

Course slides from
Leiserson's @MIT
Edmonds@York Un.
Ruan @UTSA
have been used in
preparation of these slides.

Last updated: Sept. 22, 2011

Outline

- Insertion Sort
  - Pseudocode Conventions
  - Analysis of Insertion Sort
  - Loop Invariants and Correctness
- Merge Sort
  - Divide and Conquer
  - Analysis of Merge Sort
- Growth of Functions
  - Asymptotic notation
  - Comparison of functions
  - Standard notations and common functions

Sorting Problem

Input: A sequence of *n* numbers

$\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering)

$\langle a_1', a_2', \ldots, a_n' \rangle$

such that

$a_1' <= a_2' <= \ldots <= a_n'$

Sorting

- Numbers to be sorted (a_i) also known as *keys*
  - for example, studentID, checkNumber, etc.
- In real implementations of sorting algorithms records need to also be moved with keys
- To save time, pointers may be copied instead of actual records
- Algorithm does not concern itself with these details, but it will be your job when you actually do the programming

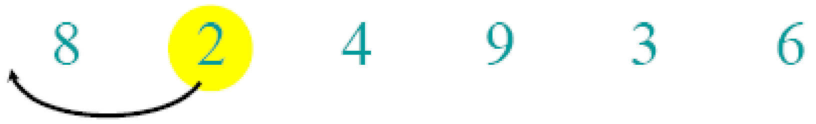
Insertion Sort

- Simple algorithm
- Basic idea:
  - Assume initial *j*-1 elements are sorted
  - Until you find place to insert *j*th element, move array elements to right
  - Copy *j*th element into its place
- Insertion Sort is an “in place” sorting algorithm. No extra storage is required.

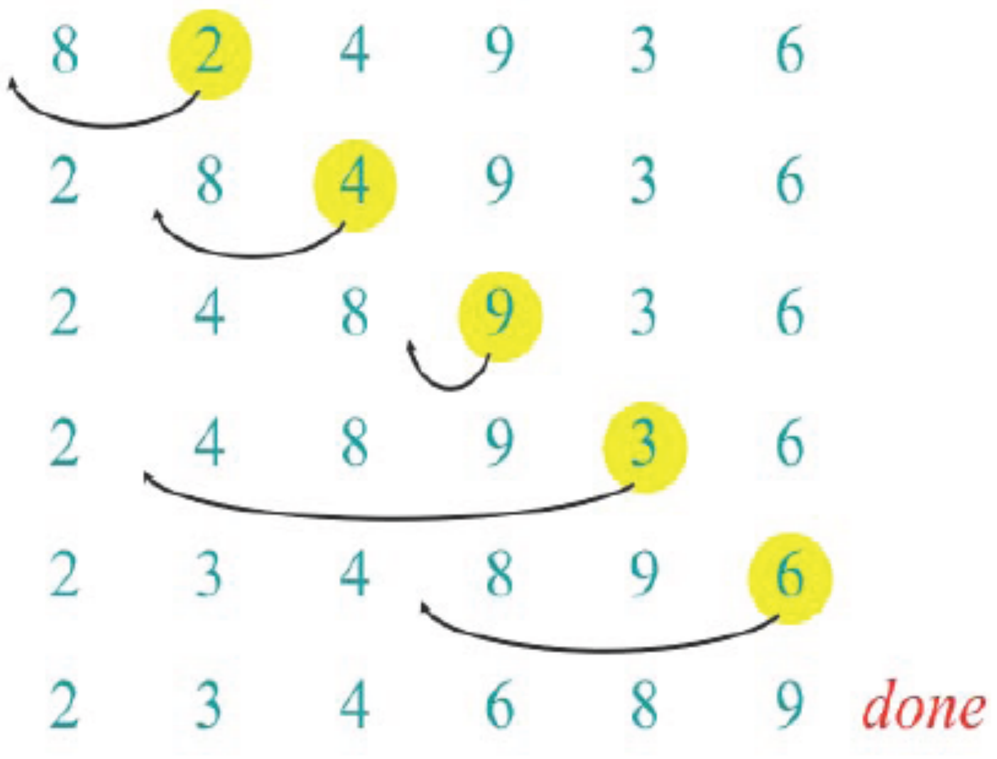
Insertion Sort Example

8 2 4 9 3 6

Insertion Sort Example



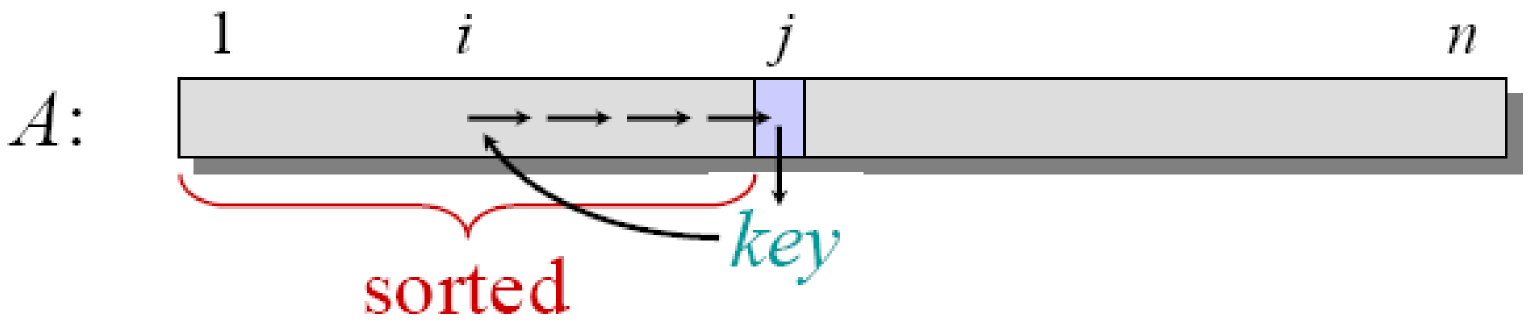
Insertion Sort Example



Insertion Sort

“pseudocode”

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



Pseudocode Conventions

- Indentation
  - indicates block structure
  - saves space and writing time
- Looping constructs (**while, for, repeat**) and conditional constructs (**if, then, else**)
  - like in C, C++, and Java
  - we assume that loop variable in a **for** loop is still defined when loop exits
- Symbol "▷" indicates that remainder of line is comment
- Multiple assignment $i \leftarrow j \leftarrow e$ assigns to both variables *i* and *j* value of *e* $(== j \leftarrow e,\ i \leftarrow j)$
- Variables are local, unless otherwise specified

Pseudocode Conventions

- Array elements are accessed by specifying array name followed by index in square brackets
  - A[i] indicates ith element of array A
  - Notation ".." is used to indicate a range of values within an array (A[i..j] = A[1], A[2],…, A[j])
- We often use **objects**, which have **attributes** (equivalently, **fields**)
  - For an attribute *attr* of object x, we write *attr*[*x*]
  - Equivalent of *x.attr* in Java or *x*-> *attr* in C++
- Objects are treated as references, like in Java
  - If *x* and *y* denote objects, then assignment *y* ← *x* makes *x* and *y* reference same object
  - It does not cause attributes of one object to be copied to another

Pseudocode Conventions

- Parameters are passed **by value**, as in Java and C (and the default mechanism in C++).
  - When an object is passed by value, it is actually a reference (or pointer) that is passed
  - Changes to the reference itself are not seen by caller, but changes to the object's attributes are
- Boolean operators "and" and "or" are **short-circuiting**
  - If after evaluating left-hand operand, we know result of expression, then we do not evaluate right-hand operand
  - If x is FALSE in "x and y", then we do not evaluate y
  - If x is TRUE in "x or y", then we do not evaluate y

Efficiency

- Correctness alone is not sufficient
- **Brute-force** algorithms exist for most problems
- To sort *n* numbers, we can enumerate all permutations of these numbers and test which permutation has the correct order
  - Why cannot we do this?
  - Too slow!
  - By what standard?

How to measure complexity?

- Accurate running time is not a good measure
- It depends on **input**
- It depends on the **machine** you used and who implemented the algorithm

- We would like to have an analysis that **does not depend** on those factors

Machine-independent

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each simple operation (e.g. +, -, =, *, if, for) takes 1 step.
    - Loops and subroutine calls are *not* simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits
    - No memory hierarch (caches, virtual mem) is modeled

Running Time

- **Running Time:T(n):** Number of primitive operations or steps executed for an input of size n.
- Running time depends on input
  - already sorted sequence is easier to sort
- Parameterize running time by size of input
  - short sequences are easier to sort than long ones
- Generally, we seek upper bounds on running time
  - everybody likes a guarantee

Kinds of Analysis

- **Worst-case:** (usually)
  - T(*n*) = maximum time of algorithm on any input of size *n*
- **Average-case:** (sometimes)
  - T(*n*) = expected time of algorithm over all inputs of size *n*
  - Need assumption about statistical distribution of inputs
- **Best-case:** (bogus)
  - Cheat with a slow algorithm that works fast on some input

Analysis of insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`      key = A[j]`	c_2	$(n-1)$
`      i = j - 1;`	c_3	$(n-1)$
`      while (i > 0) and (A[i] > key) {`	c_4	$\sum_{j=2..n} t_j$
`            A[i+1] = A[i]`	c_5	$\sum_{j=2..n} t_j -1$
`            i = i - 1`	c_6	$\sum_{j=2..n} t_j -1$
`      }`	0	
`      A[i+1] = key`	c_7	$(n-1)$
`  }`	0	
`}`		

t_j : number of times while loop test is executed for jth **`for`** loop iteration

Analysis of insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`    key = A[j]`	c_2	$(n-1)$
`    i = j - 1;`	c_3	$(n-1)$
`    while (i > 0) and (A[i] > key) {`	c_4	S
`      A[i+1] = A[i]`	c_5	$S – (n-1)$
`      i = i - 1`	c_6	$S – (n-1)$
`    }`	0	
`    A[i+1] = key`	c_7	$(n-1)$
`  }`	0	
`}`		

t_j : number of times while loop test is executed for jth **`for`** loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 S + c_5(S - (n-1)) + c_6(S - (n-1)) + c_7(n-1)$
  $= c_8 S + c_9 n + c_{10}$
- What can S be?
  - Best case -- inner loop body never executed
    - $t_j = 1 \Rightarrow S = n - 1$
    - $T(n) = an + b$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_j = j \Rightarrow S = 2 + 3 + \ldots + n = n(n+1)/2 - 1$
    - $T(n) = an^2 + bn + c$ is a quadratic function
  - Average case
    - Can assume that in average, we have to insert A[j] into the middle of A[1..j-1], so $t_j = j/2$
    - $S \approx n(n+1)/4$
    - T(n) is still a quadratic function

Insertion Sort Running Time

Theta Notation, see next week.

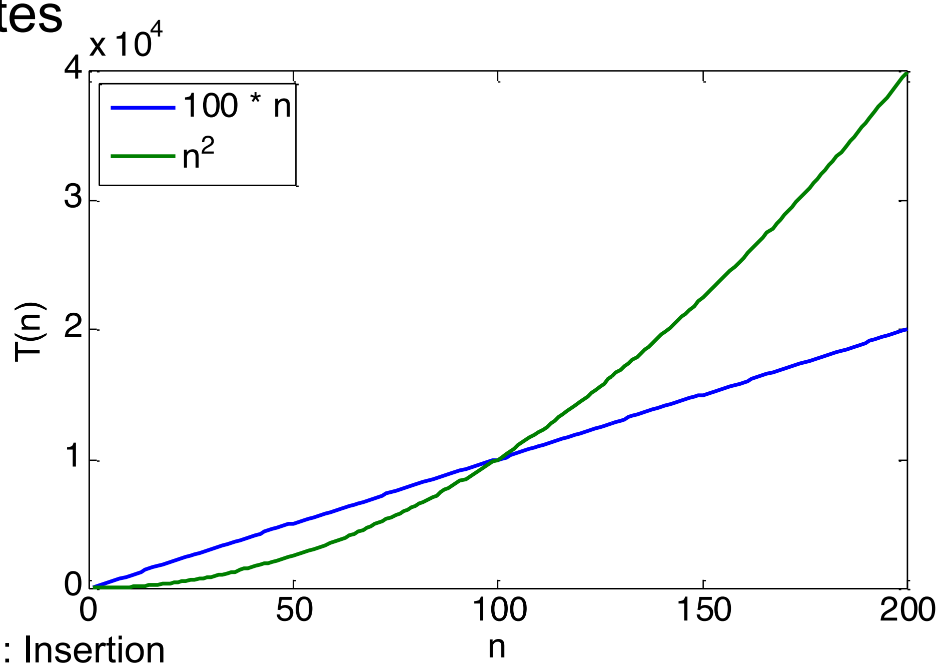
- **Best-case:**
  - $\Theta(n)$, inner loop not executed at all
- **Worst-case:** Input reverse sorted
  - $T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$ [Arithmetic series]
- **Average-case:** All permutations equally likely
  - $T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$

Is Insertion Sort a fast sorting algorithm?

- Moderately so, for small *n*
- Not at all, for large *n*

Asymptotic Analysis

- Ignore actual and abstract statement costs
- *Order of growth* is the interesting measure:
  - Highest-order term is what counts
    - As the input size grows larger it is the high order term that dominates



Loop invariants and correctness of insertion sort

- Claim: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A [1..j-1] but in sorted order.
- Proof: by induction

Review: Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Prove correctness using loop invariants

- Initialization (basis): the loop invariant is true prior to the first iteration of the loop
- Maintenance:
  - Assume that it is true before an iteration of the loop (Inductive hypothesis)
  - Show that it remains true before the next iteration (Step)
- Termination: show that when the loop terminates, the loop invariant gives us a useful property to show that the algorithm is correct

Prove correctness using loop invariants

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
     }
     A[i+1] = key
  }
}
```

Loop invariant: at the start of each iteration of the for loop, the subarray consists of the elements originally in A[1..j-1] but in sorted order.

Initialization

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷ Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
     }
     A[i+1] = key
  }
}
```

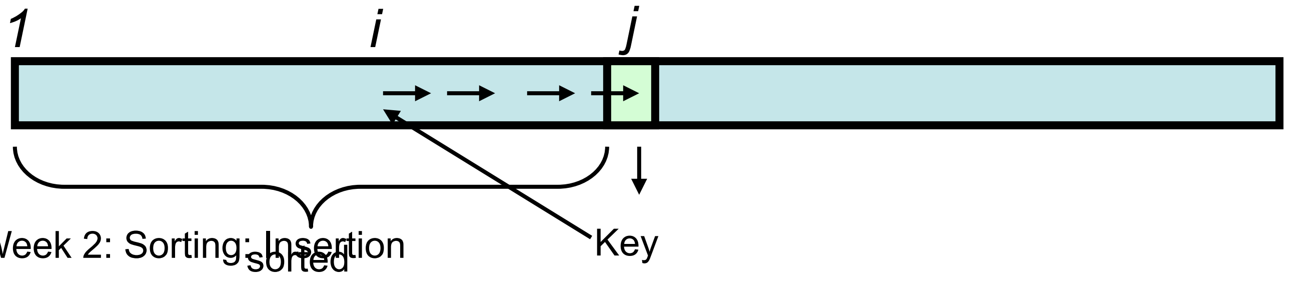
Subarray A[1] is sorted. So loop invariant is true before the loop starts.

Maintenance

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷ Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
     }
     A[i+1] = key
  }
}
```

Assume loop variant is true prior to iteration j

Loop variant will be true before iteration j+1

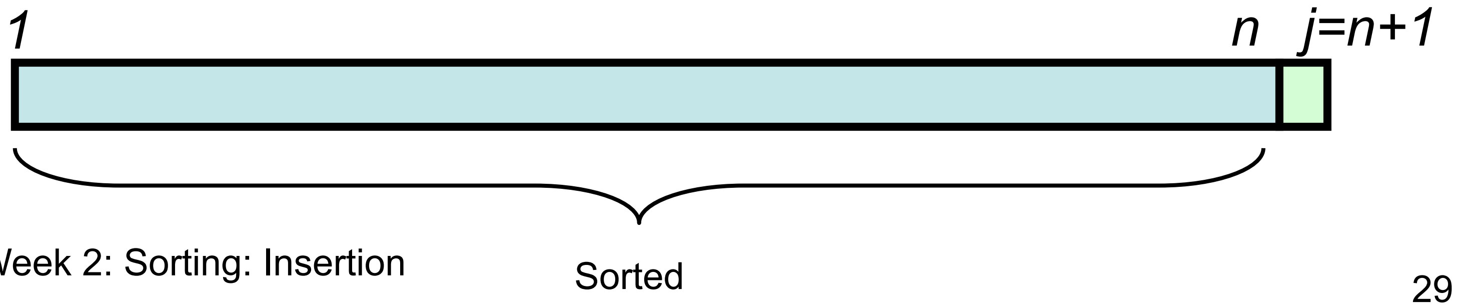


Termination

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j];
      i = j - 1;
      ▷ Insert A[j] into the sorted sequence A[1..j-1]
      while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
      }
      A[i+1] = key
  }
}
```

The algorithm is correct!

Upon termination, A[1..n] contains all the original elements of A in sorted order.



Merge Sort

- Insertion sort used an incremental approach to sorting: sort the smallest subarray (1 item), add one more item to the subarray, sort it, add one more item, sort it, etc.

- Let us think about how the merge sort works. Basically, it uses a *divide-and-conquer* approach, based on the concept of *recursion*.

Merge Sort

- *Divide-and-conquer*:
  - *Divide* the problem into several subproblems.
  - *Conquer* the subproblems by solving them recursively. If the subproblems are small enough, solve them directly.
  - *Combine* the solutions to the subproblems to get the solution for the original problem.

Merge Sort

- *Divide-and-conquer*:
  - *Divide* the n-element sequence to be sorted into two subsequences of n/2 each.
  - *Conquer* by sorting the subsequences recursively by calling merge sort again. If the subsequences are small enough (of length 1), solve them directly. (Arrays of length 1 are already sorted.)
  - *Combine* the two sorted subsequences by merging them to get a sorted sequence.

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3         Merge-Sort(A, p, q)
4         Merge-Sort(A, q+1, r)
5         Merge(A, p, q, r)}
```

- A is the (sub)array *when the procedure is called.*
- p, q, and r are indices numbering elements of the array such that $p \le q \le r$; p is the lowest index and r is the highest index.

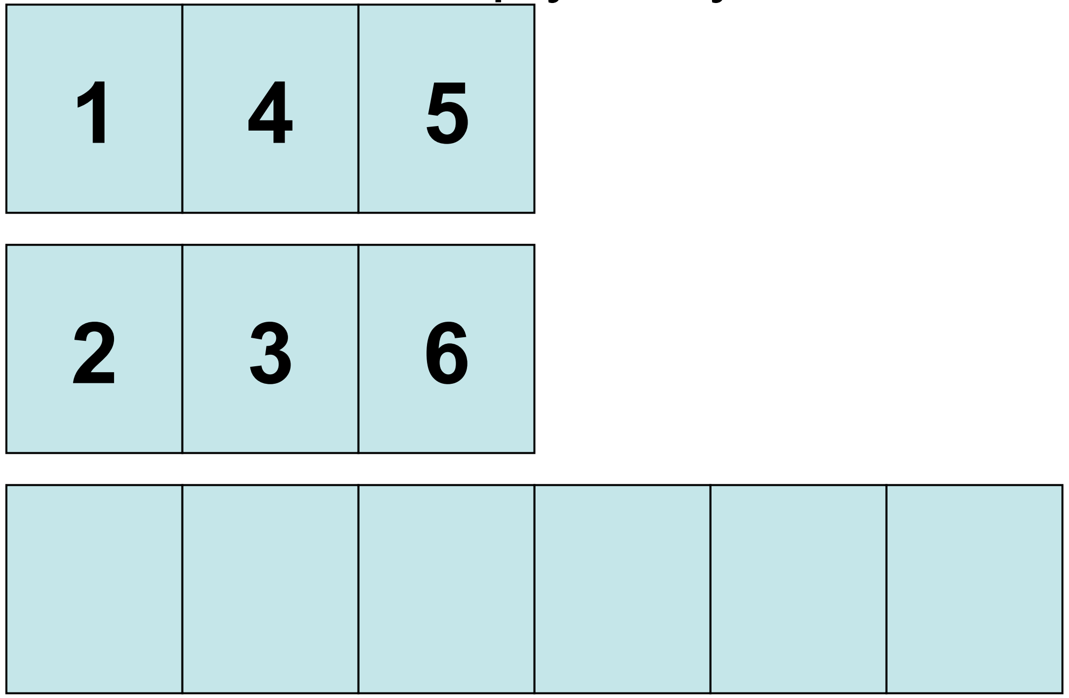
Merge Sort

- Note that the merge sort basically consists of recursive calls to itself. The base case (which stops the recursion) occurs when a subsequence has a size of 1.

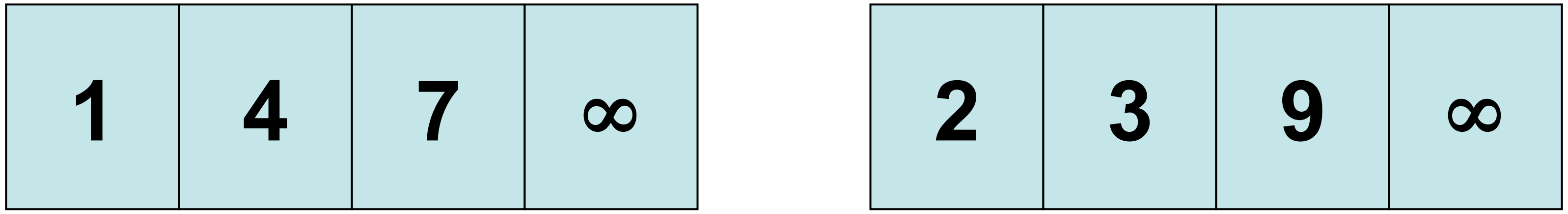
- The combine step is accomplished by a call to an algorithm called Merge.

Merge

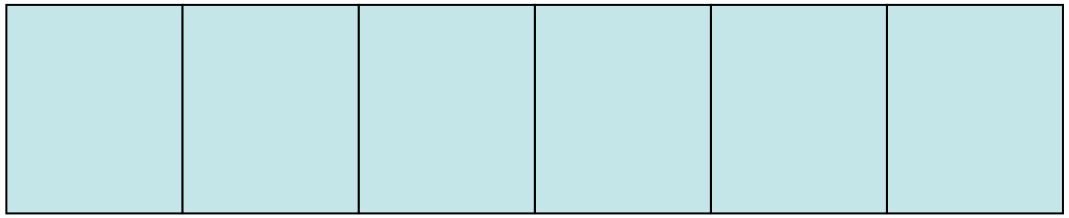
- Without going into detail about how Merge-Sort works yet, let us take a look at the Merge part. Merge works by assuming you have two already-sorted sublists and an empty array:



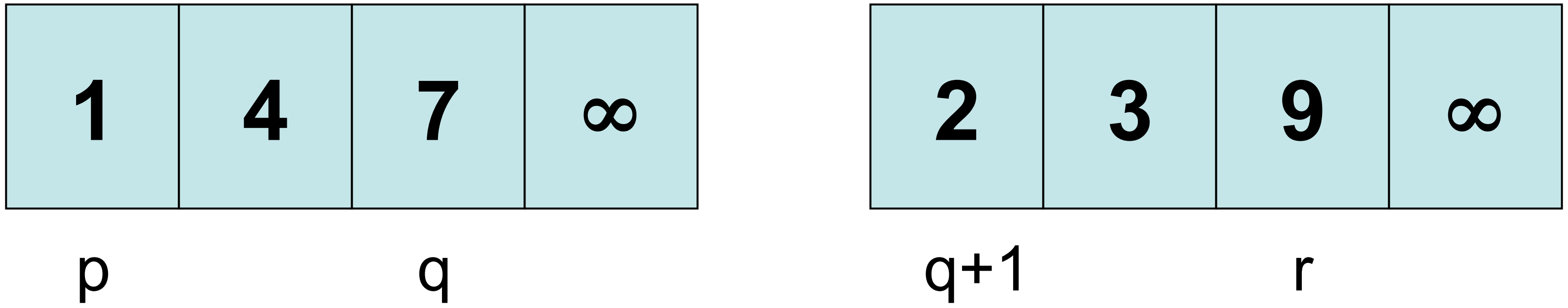
Merge



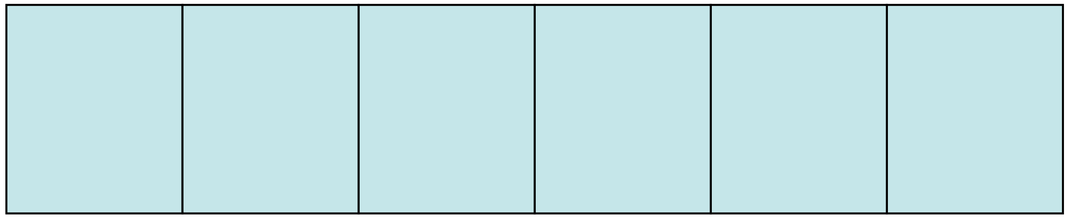
- Let us assume we have a *sentinel* (infinity, which is guaranteed to be larger than the last item) at the end of each sublist which lets us know when we have hit the end of the sublist.



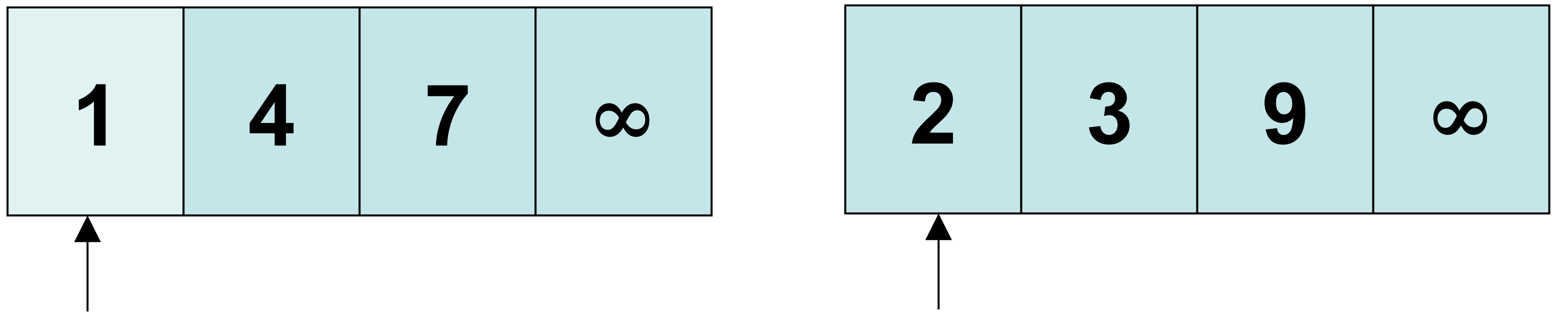
Merge



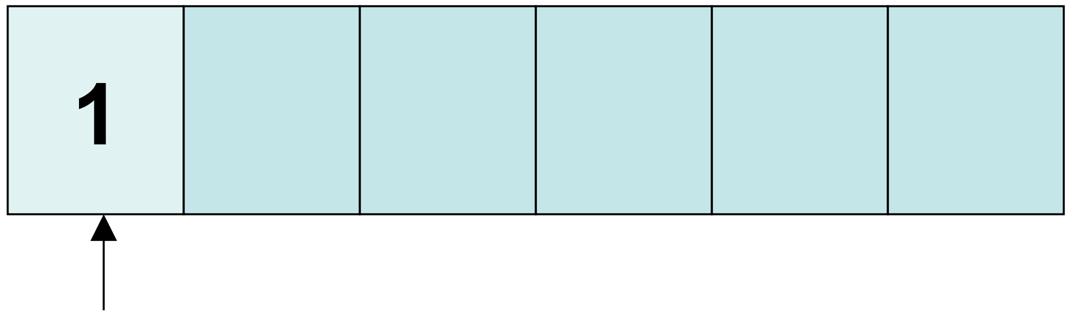
- The two sublists are indexed from p to q (for the first sublist) and from q+1 to r for the second sublist. There are (r – p) + 1 items in the two sublists combined, so we will need an output array of that size.



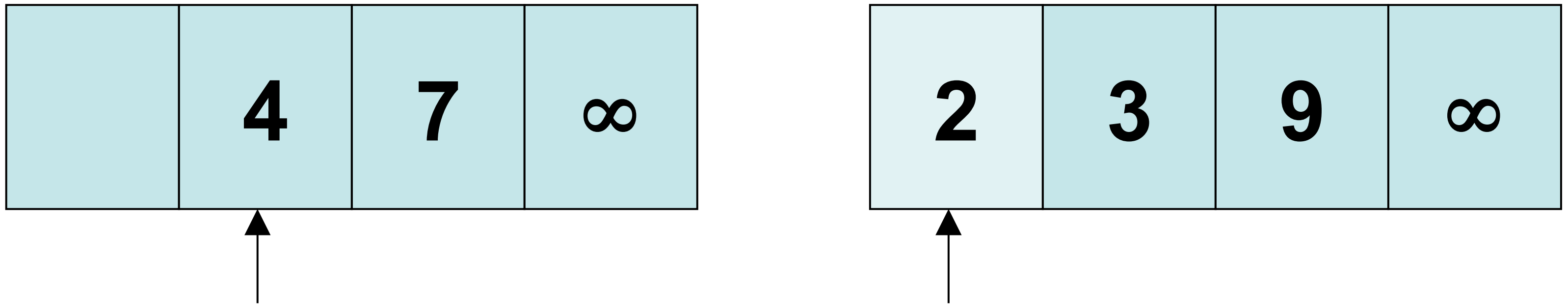
Merge



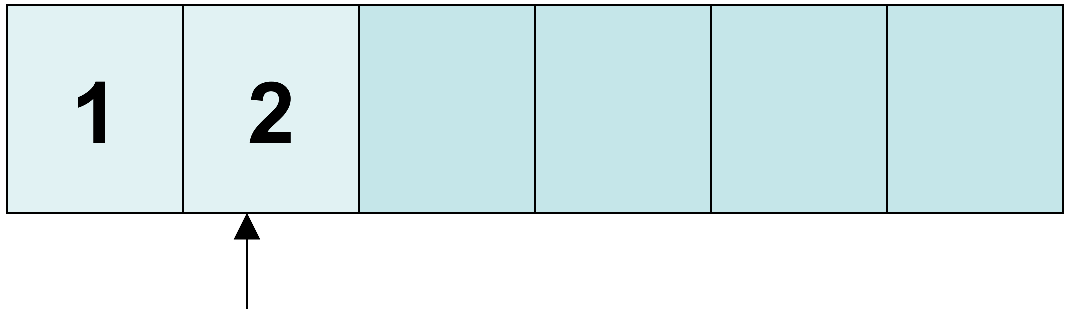
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



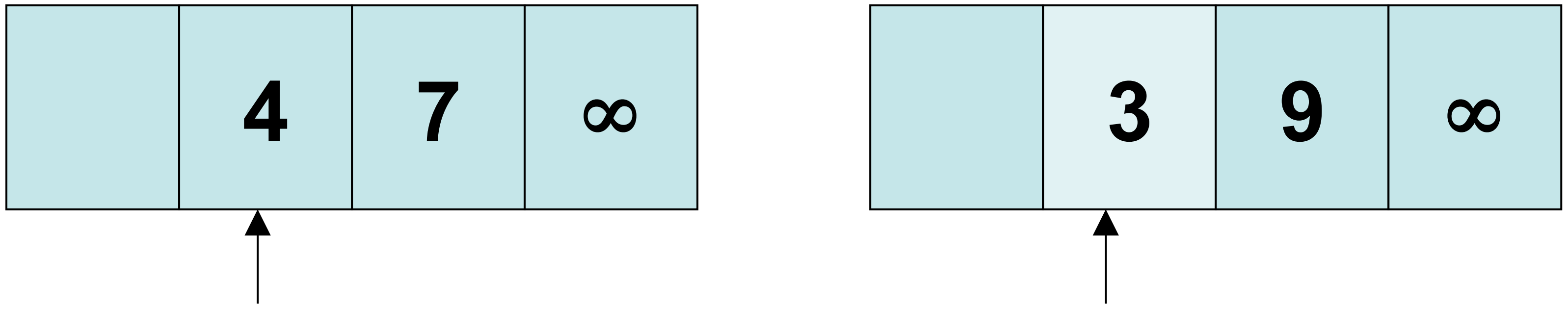
Merge



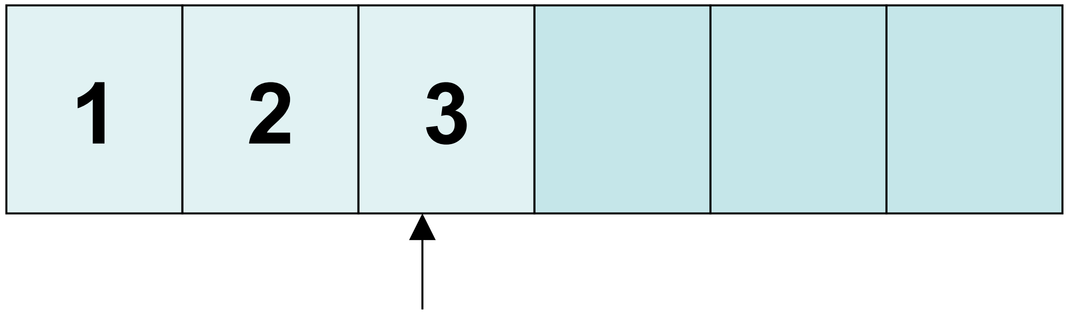
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



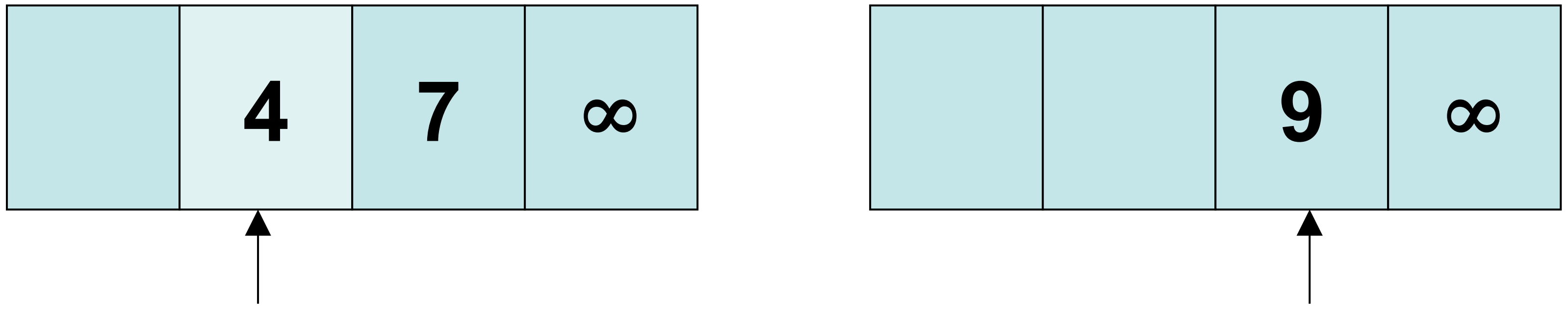
Merge



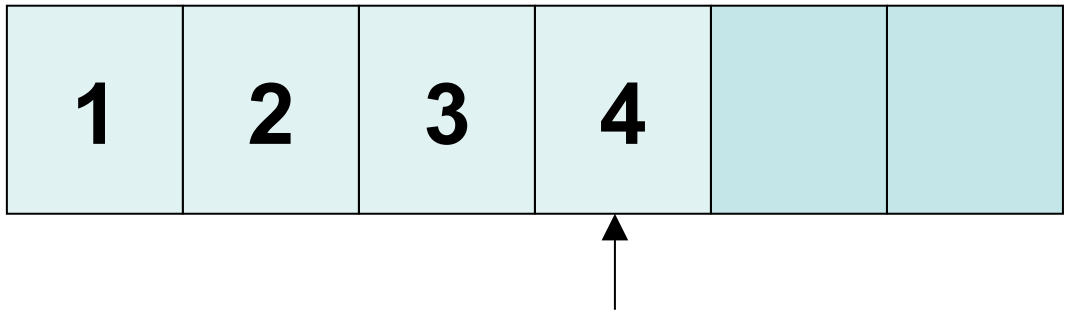
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



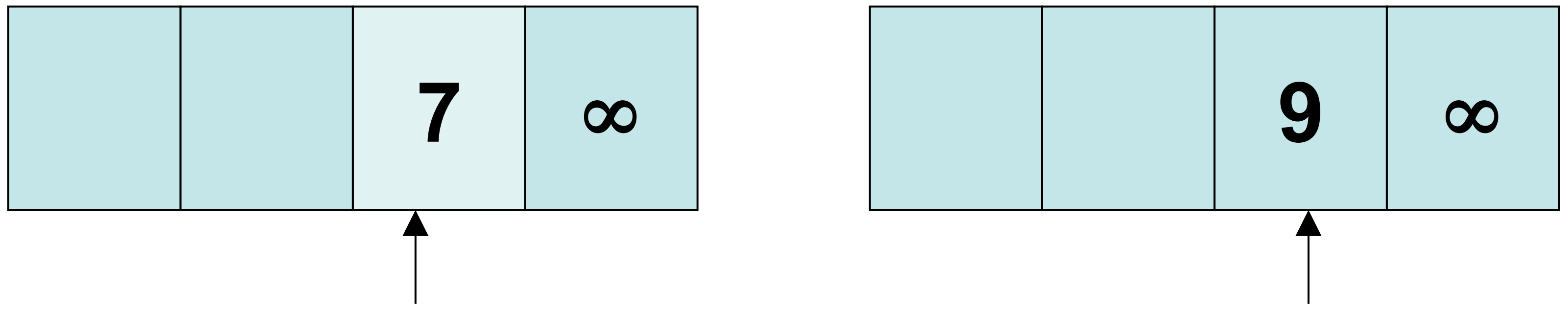
Merge



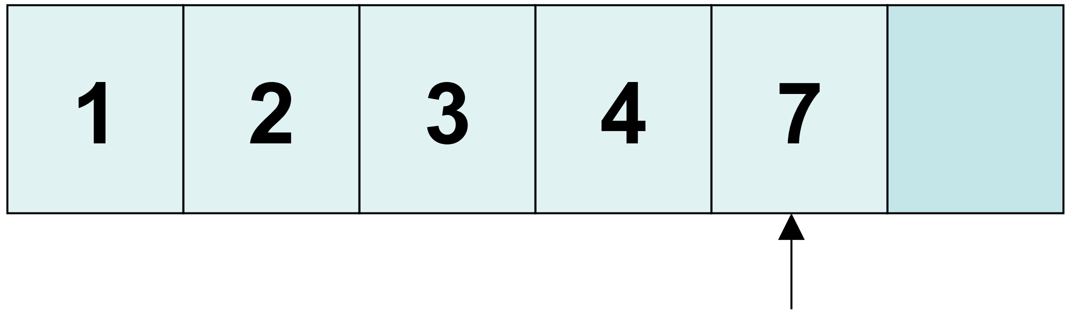
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



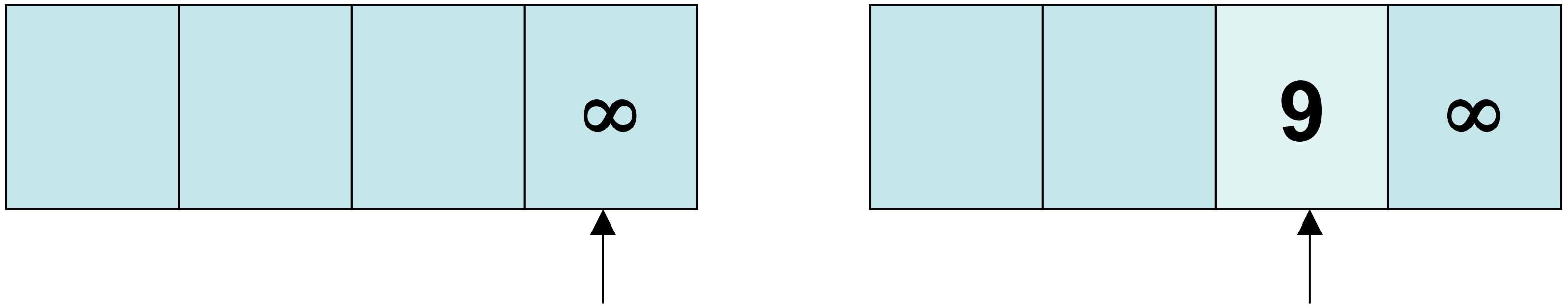
Merge



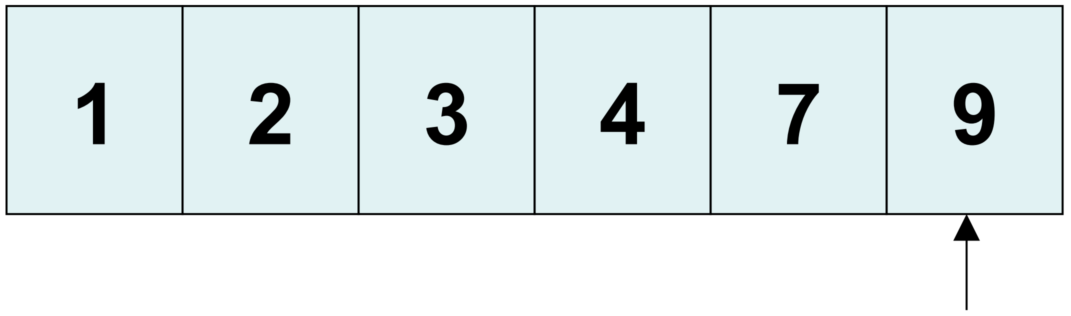
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



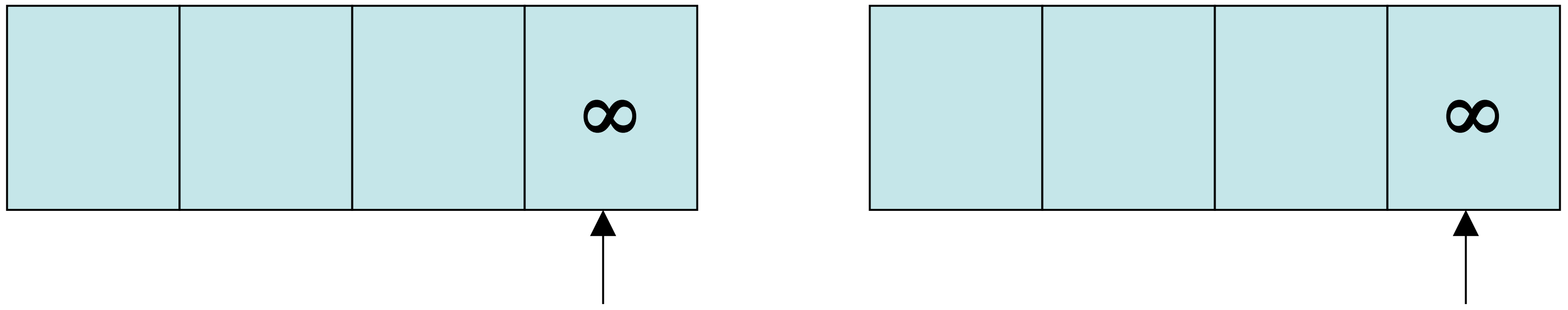
Merge



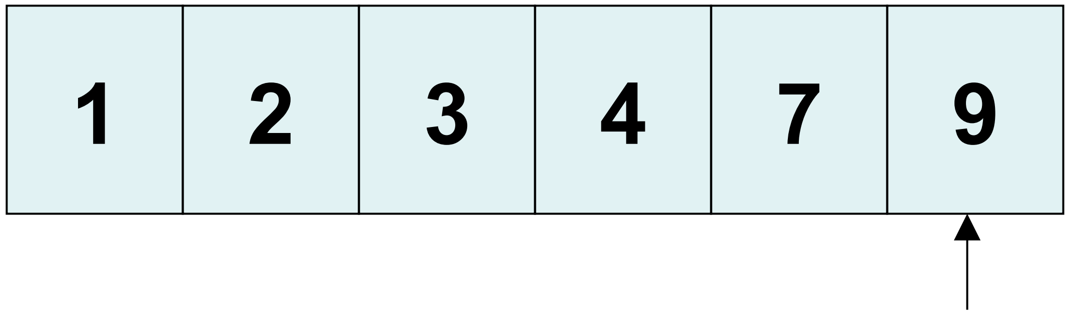
- Look at the first item in each subarray. Choose the smallest item.
- Move the chosen item to the output array.



Merge



- We know that we have only n = (r – p) + 1 items. So, we will make only (r – p) + 1 moves.
- Here r = 1 and p = 6, and (6 – 1) + 1 = 6, so when we have made our 6th move we are through.



Merge

- Assuming that the two sublists are in sorted order when they are passed to the Merge routine, is Merge guaranteed to output a sorted array?
- Yes. We can verify that each step of Merge preserves the sorted order that the two sublists already have.

Merge(A, p, q, r)

```
$n_1$ ← (q - p) + 1
$n_2$ ← (r - q)
create arrays L[1..$n_1$+1] and R[1..$n_2$+1]
for i ← 1 to $n_1$ do
  L[i] ← A[(p + i) -1]
for j ← 1 to $n_2$ do
  R[j] ← A[q + j]
L[$n_1$ + 1] ← ∞
R[$n_2$ + 1] ← ∞
i ← 1
j ← 1
for k ← p to r do
  if L[i] <= R[j]
       then A[k] ← L[i]
            i ←i + 1
       else A[k] ← R[j]
            j ← j + 1
```

Analysis of Merge

- Line 1 computes the length n_1 of the subarray A[p..q].
- Line 2 computes the length n_2 of the subarray A[q+1..r].
- We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3.
- The **for** loop of lines 4-5 copies the subarray A[p..q] into L[1.. n_1], and the **for** loop of lines 6-7 copies the subarray A[q+1..r] into R[1..n2].
- Lines 8-9 put the sentinels at the ends of the arrays L and R.
- Lines 10-17, illustrated in Figure 2.3, perform the r - p + 1 basic steps by maintaining the following loop invariant.

Analysis of Merge

- The loop in lines 12-17 of Merge is the heart of how Merge works. They maintain the loop invariant:
- At the start of each iteration of the **for** loop of lines 12-17, the subarray A[p..k-1] contains the k - p smallest elements of L[1..n_1+1] and R[1..n_2+1], in sorted order.
- Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Analysis of Merge

- To prove that Merge is a correct algorithm, we must show that:
- **Initialization:** the loop invariant holds prior to the first iteration of the for loop in lines 12-17
- **Maintenance:** each iteration of the loop maintains the invariant
- **Termination:** the invariant provides a useful property to show correctness when the loop terminates

Initialization

- As we enter the *for* loop, k is set equal to p.
- This means that subarray A[p..k-1] is empty.
- Since k - p = 0, the subarray is guaranteed to contain the k - p smallest elements of L and R.
- By lines 10 and 11, i = j = 1, so L[i] and R[j] are the smallest elements of their arrays that have not been copied into A.

Maintenance

- As we enter the loop, we know that A[p..k-1] contains the k - p smallest elements of L and R.
- Assume L[i] <= R[j]. Then:
  - L[i] is the smallest element not copied into A.
  - Line 14 will copy L[i] into A[k].
  - At this point the subarray A[p..k] will contain the k - p + 1 smallest elements.
  - Incrementing k (in line 12) and i (in line 15) reestablishes the loop invariant for the next iteration.
- Assume L[i] >= R[j]. Then:
  - Lines 16-17 maintain the loop invariant.

Termination

- The loop invariant states that subarray "A[p..k-1] contains the k - p smallest elements of L[1..n_1+1] and R[1..n_2+1], in sorted order."
- When we drop out of the loop, k = r + 1.
- So r = k – 1, and A[p..k-1] is actually A[p..r], which is the whole array.
- The arrays L and R together contain $n_1 + n_2 + 2$ elements. From lines 1 and 2 we know that $n_1 + n_2 = ((q – p) + 1) + ((r – q) = (r – p) + 1$, and this is the number of all of the elements in the array. The extra 2 is the two sentinel elements.

Merge Sort

- Now let us look at Merge-Sort again:

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← $\lfloor (p+r)/2 \rfloor$
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

- Line 1 is our base case; we drop out of the recursive sequence of calls when p >= r.

Merge Sort

- Given our Merge routine, we can now see how Merge-Sort works.
  - Assume a list of length = 2^m
  - Take an unsorted list as input.
  - Split it in half. Now you have two sublists.
  - Split those in half, and so on, until you have lists of length 1.
  - Merge those into sublists of length 2, then merge those into sublists of length 4, etc. Keep going until you have just one list left.
  - That list is now sorted.

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3            Merge-Sort(A, p, q)
4            Merge-Sort(A, q+1, r)
5            Merge(A, p, q, r)}
```

- Let us call Merge-Sort with an array of 4 elements: Merge-Sort(A, 1, 4), where p = 1 and r = 4.
- Line 1: $p < r$, so do the *then* part of the *if*
- Line 2: $q \leftarrow \lfloor (p+r)/2 \rfloor$, which is 2
- Line 3: we call Merge-Sort(A, 1, 2)
- WAIT HERE (let us call our place Z) until we return from this call

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

- Calling Merge-Sort(A, 1, 2)
- Line 1: $p < r$, so do the *then* part of the *if*
- Line 2: $q \leftarrow \lfloor (p+r)/2 \rfloor$, which is 1
- Line 3: we call Merge-Sort(A, 1, 1)
- WAIT HERE (let us call our place Y) until we return from this call

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

- Calling Merge-Sort(A, 1, 1)
- Line 1: p = r, so skip the *then* part of the *if*
- Return from this call to Y

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

- We called Merge-Sort(A, 1, 2)
- We have returned from our call in line 3
- Line 4: We call Merge-Sort(A, 2, 2)
- WAIT HERE (let us call our place X) until we return from this call

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

- Calling Merge-Sort(A, 2, 2)
- Line 1: p = r, so skip the *then* part of the *if*
- Return from this call to X

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3           Merge-Sort(A, p, q)
4           Merge-Sort(A, q+1, r)
5           Merge(A, p, q, r)}
```

- We called Merge-Sort(A, 2, 2)
- We have returned from our call in line 4
- Line 5: We call Merge(A, 1, 2, 2)
- What does Merge do?

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3          Merge-Sort(A, p, q)
4          Merge-Sort(A, q+1, r)
5          Merge(A, p, q, r)}
```

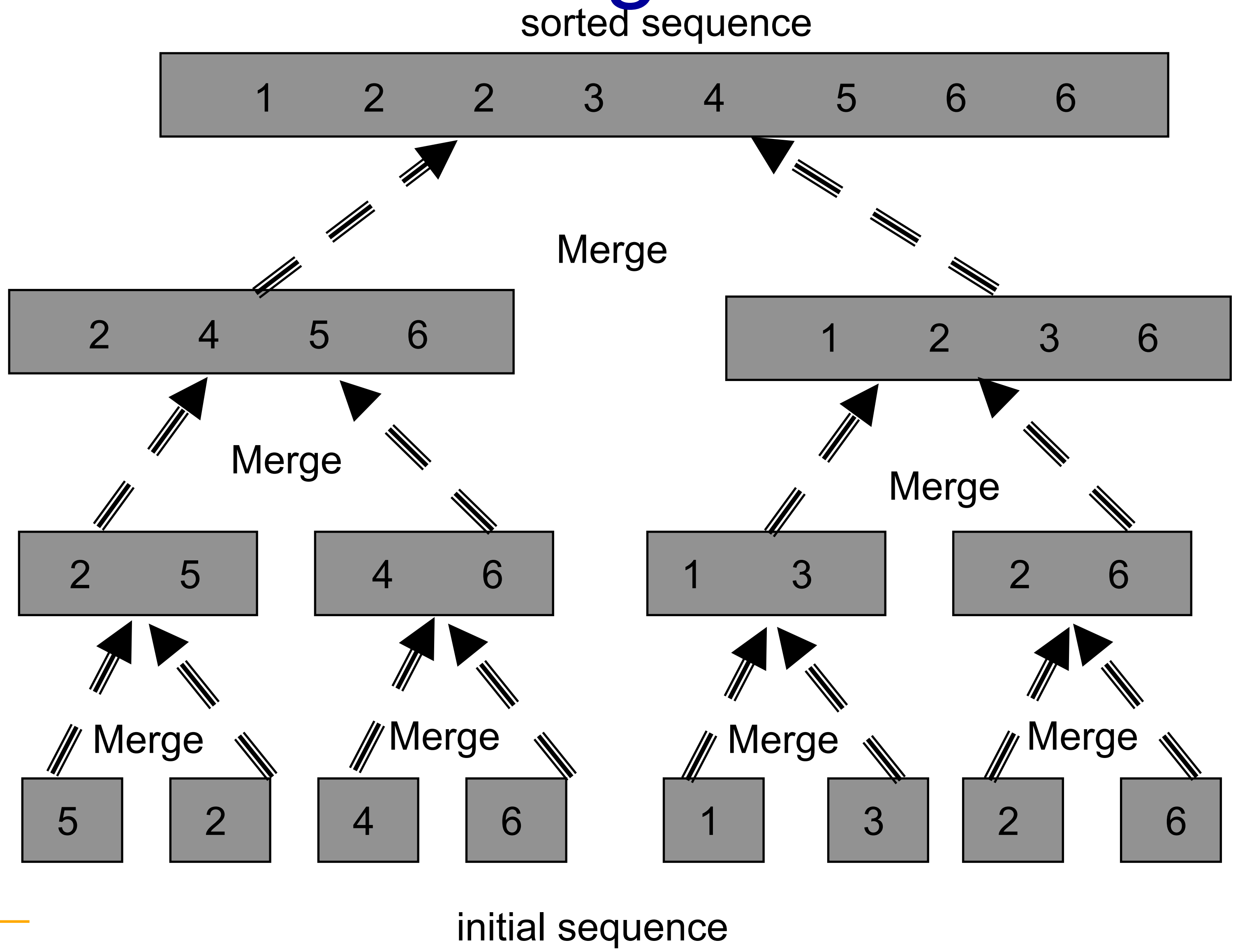
- Step 5: Merge(A, 1, 2, 2) :
- creates two temporary arrays of 1 element each
- copies A[1] and A[2] into these 2 arrays
- merges the elements in these two temporary arrays back into A[1..2] in sorted order
- returns from the call to Z

Merge Sort

```
Merge-Sort(A, p, r)
1  if p < r
2  then {q ← ⌊(p+r)/2⌋
3            Merge-Sort(A, p, q)
4            Merge-Sort(A, q+1, r)
5            Merge(A, p, q, r)}
```

- Return from call to Merge-Sort(A, 1, 2) in Line 3. At this point half of our original array, A[1..2], is in sorted order.
- Next we call Merge-Sort(A, 3, 4). It will put A[3..4] into sorted order.
- Line 5 will merge A[1..2] and A[3..4] into A[1..4] in sorted order.

Merge Sort


sorted sequence
1 2 2 3 4 5 6 6
Merge
2 4 5 6
1 2 3 6
Merge
Merge
2 5
4 6
1 3
2 6
Merge
Merge
Merge
Merge
5
2
4
6
1
3
2
6
initial sequence

Analysis of Divide-and-Conquer Algorithms

- The Merge-Sort algorithm contains a recursive call to itself. When an algorithm contains a recursive call to itself, its running time often can be described by a *recurrence equation*, or *recurrence*.
- The recurrence equation describes the running time on a problem of size n in terms of the running time on smaller inputs.
- We can use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Analysis of Divide-and-Conquer Algorithms

- A recurrence of a divide-and-conquer algorithm is based on its 3 parts: divide, conquer, and combine.
- Let T(n) be the running time on a problem of size n.
- If the problem is small enough, say n <= c, we can solve it in a straightforward manner, which takes constant time, which we write as $\Theta(1)$.
- If the problem is bigger, we solve it by dividing the problem to get *a* subproblems, each of which is *1/b* the size of the original. For Merge-Sort, both *a* and *b* are 2.

Analysis of Divide-and-Conquer Algorithms

- Assume it takes D(n) time to divide the problem into subproblems.
- Assume it takes C(n) time to combine the solutions to the subproblem into the solution for the original problem.
- We get the recurrence:

$$T(n) = \begin{cases} \Theta(1) & , \text{ if } n \le c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

Analysis of Merge Sort

- **Base case:** n = 1. Merge sort on an array of size 1 takes constant time, Θ(1).
- **Divide:** The Divide step of Merge-Sort just calculates the middle of the subarray. This takes constant time. So D(n) = Θ(1).
- **Conquer:** We make 2 calls to Merge-Sort. Each call handles ½ of the subarray that we pass as a parameter to the call. The total time required is 2T(n/2).
- **Combine:** Running Merge on an n-element subarray takes Θ(n), so C(n) = Θ(n).

Analysis of Merge Sort

- Here is what we get

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

- By inspection, we can see that we can ignore the Θ(1) factor, as it is irrelevant compared to Θ(n). We can rewrite this recurrence as:

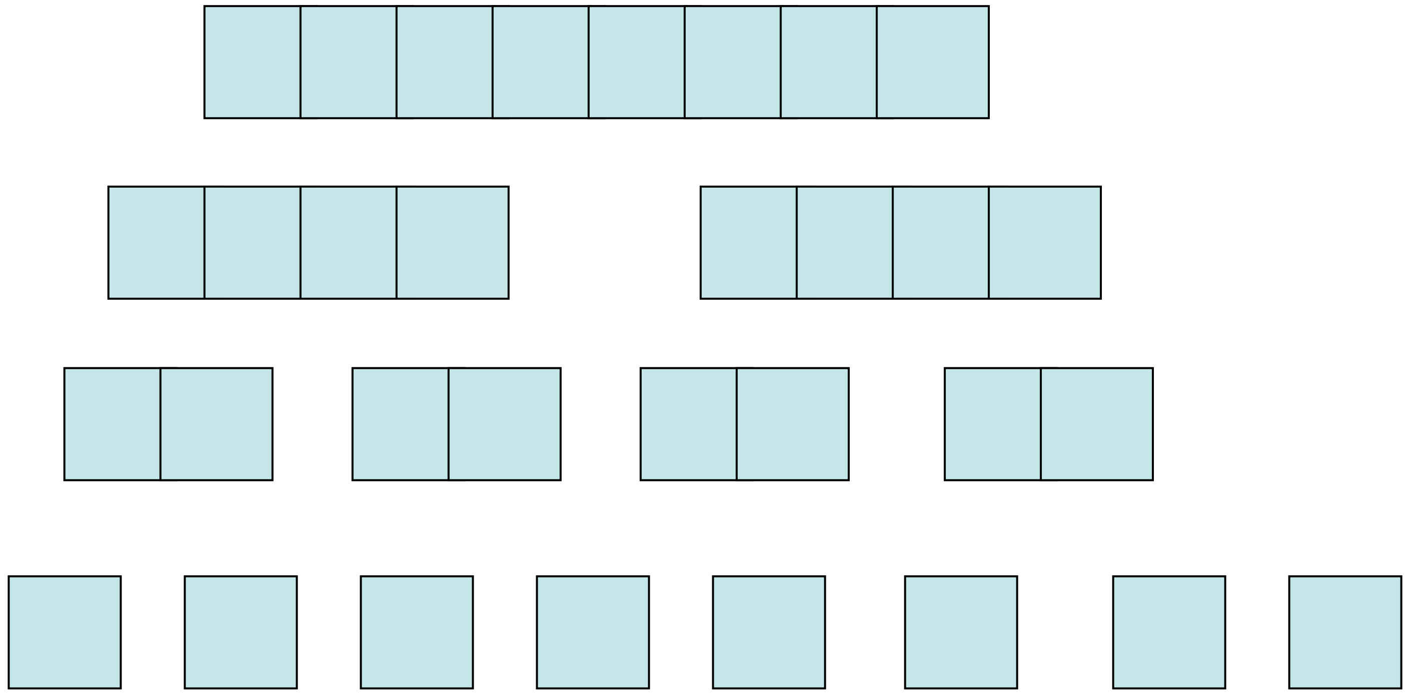
$$T(n) = \begin{cases} c & , \quad \text{if } n = 1 \\ 2T(n/2) + cn, & \quad \text{if } n > 1 \end{cases}$$

Analysis of Merge Sort

- How much time will it take for the Divide step?
- Let us assume that n is some power of 2.
- Then for an array of size n, it will take us $\log_2 n$ steps to recursively subdivide the array into subarrays of size 1.
- Example: $8 = 2^3$

Analysis of Merge Sort

- Example: $8 = 2^3$
- Step 0:
- Step 1:
- Step 2:
- Step 3:



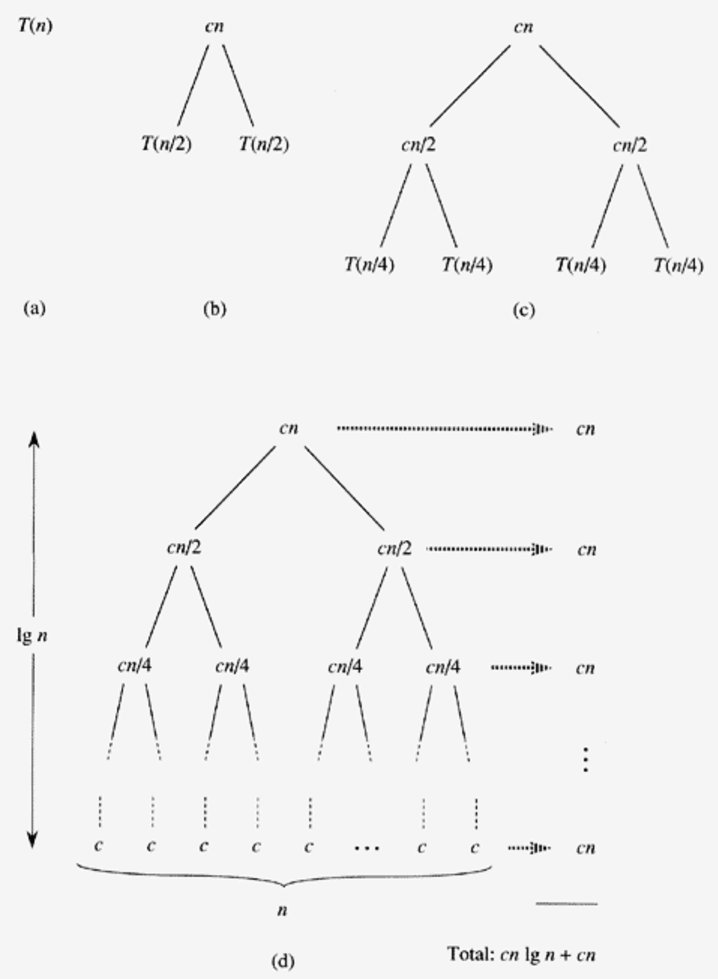


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Analysis of Merge Sort

- So, it took us $\log_2 n$ steps to divide the array all the way down into subarrays of size 1.
- As a result, we will have $\log_2 n + 1$ (sub)arrays to deal with. In our example, where $n = 8$ and $\log_2 n = 3$, we will have to deal with arrays of size 1, 2, 4, and 8.
- Every time we Merge the arrays, it takes us n steps, since we have to put each array item into its proper position within each array.

Analysis of Merge Sort

- Consequently, we will have $\log_2 n + 1$ recursive calls of the Merge-Sort function, and each time we call Merge-Sort the Merge function will cost us n steps, times a constant value.
- The total cost, then, can be expressed as:

  $cn(\log_2 n + 1)$
- Multiplying this out gives:

  $cn(\log_2 n) + cn$
- Ignoring the low-order term and the constant c gives:

  $\Theta(n \cdot \log_2 n)$

Asymptotic Notation

- What does asymptotic mean?
- Asymptotic describes behavior of function in the limit - for sufficiently large values of its parameter

Asymptotic Notation

- The order of growth of the running time of an algorithm is defined as the highest-order term (usually the leading term) of an expression that describes the running time of the algorithm
- We ignore the leading term's constant coefficient, as well as all of the lower order terms in the expression
- Example: The order of growth of an algorithm whose running time is described by the expression $an^2 + bn + c$ is simply n^2

Big O

- Let us say that we have some function that represents the sum total of all the running-time costs of an algorithm; call it f(n)
- For merge sort, the actual running time is:

$$f(n) = cn(\log_2 n) + cn$$

- We want to describe the running time of merge sort in terms of another function, g(n), so that we can say f(n) = O(g(n)), like this:

$$cn(\log_2 n) + cn = O(n\log_2 n)$$

Big O: Definition

- For a given function g(n), O(g(n)) is the set of functions

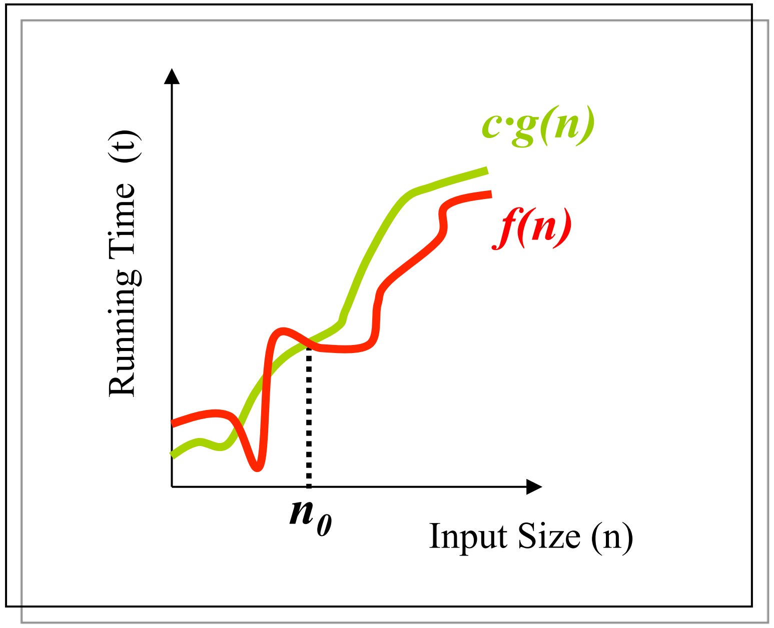
O(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \bullet g(n)$ for all $n \geq n_0$ }

c is the multiplicative constant

n_0 is the threshold

Big O

$f(n) \in O(g(n))$



Big O

- Big O is an upper bound on a function, to within a constant factor.
- O(g(n)) is a *set* of functions
- Commonly used notation

  $f(n) = O(g(n))$
- Correct notation

  $f(n) \in O(g(n))$

Big O

- Question:

  How do you demonstrate that $f(n) \in O(g(n))$?

- Answer:

  Show that you can find values for c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$

Big O

Example: Show that $7n - 2$ is $O(n)$

- Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $7n - 2 \leq cn$ for every integer $n \geq n_0$.
- Choose $c = 7$ and $n_0 = 1$.
- It is easy to see that $7n - 2 \leq 7n$ for every integer $n \geq 1$.
- $\therefore$ $7n - 2$ is $O(n)$

Big O

Example: Show that $20n^3 + 10n \log n + 5$ is $O(n^3)$

- Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $20n^3 + 10n \log n + 5 \leq cn^3$ for every integer $n \geq n_0$.
- How do we find c and n_0?
- Note that $10n^3 > 10 n \log n$, and that $5n^3 > 5$.
- So, $15n^3 > 10n \log n + 5$
- And $20n^3 + 15n^3 > 20n^3 + 10n \log n + 5$
- Therefore, $35n^3 > 20n^3 + 10n \log n + 5$

Big O

- So we choose c = 35 and $n_0 = 1$
- An algorithm that takes $20n^3 + 10n \log n + 5$ steps to run cannot possibly take any more than $35n^3$ steps, for every integer $n \geq 1$
- Therefore $20n^3 + 10n \log n + 5$ is $O(n^3)$

Big O

Example: Show that $\frac{1}{2} n^2 - 3n$ is $O(n^2)$

- Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\frac{1}{2} n^2 - 3n \leq cn^2$ for every integer $n \geq n_0$
- Choose $c = \frac{1}{2}$ and $n_0 = 1$
- Now $\frac{1}{2} n^2 - 3n \leq \frac{1}{2} n^2$ for every integer $n \geq 1$

Big O

Example: Show that $an(\log_2 n) + bn$ is $O(n \cdot \log n)$

- Find a real constant c > 0 and an integer constant $n_0 \geq 1$ such that

  $$an(\log_2 n) + bn \leq cn \cdot \log n$$

  for every integer $n \geq n_0$.
- Choose $c = a+b$ and $n_0 = 2$ (why 2?)
- Now $an(\log_2 n) + bn \leq cn \cdot \log n$ for every integer $n \geq 2$.

Big O

Example: Show that $an(\log_2 n) + bn$ is $O(n \cdot \log n)$

- Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

  $$an(\log_2 n) + bn \leq cn \cdot \log n$$

  for every integer $n \geq n_0$.
- Choose $c = a+b$ and $n_0 = 2$ (why 2?)
- $\log_2 1 = 0$, $a \cdot 1 \cdot (\log_2 1) + b \cdot 1 \leq c \cdot 1 \cdot \log 1$
- $0+b \leq 0$ => NOT TRUE!

Big O

- Question:

  Is $n = O(n^2)$?

- Answer:

  Yes. Remember that $f(n) \in O(g(n))$ if there exist positive constants *c* and n_0 such that

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

  If we set $c = 1$ and $n_0 = 1$, then it is obvious that $c \bullet n \le n^2$ for all $n \ge n_0$.

Big O

- What does this mean about Big-O?
- When we write f(n) = O(g(n)) we mean that some constant times g(n) is an asymptotic upper bound on f(n); we are not claiming that this is a *tight* upper bound.

Big O

- Big-O notation describes an upper bound
- Assume we use Big-O notation to bound the *worst case* running time of an algorithm
- Now we have a bound on the running time of the algorithm on *every input*

Big O

- Is it correct to say "the running time of insertion sort is $O(n^2)$"?
- Technically, the running time of insertion sort depends on the characteristics of its input.
  - If we have *n* items in our list, but they are already in sorted order, then the running time of insertion sort *on this particular input* is O(n).

Big O

- So what do we mean when we say that the running time of insertion sort is O(n^2)?
- What we normally mean is:

  the *worst case* running time of insertion sort is O(n^2)
- That is, if we say that "the running time of insertion sort is O(n^2)", we guarantee that under no circumstances will insertion sort perform worse than O(n^2).

Big Theta: Definition

- For a given function g(n), Θ(g(n)) is the set of functions:

Θ(g(n)) = {f(n): there exist positive constants c_1, c_2, and n_0 such that

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

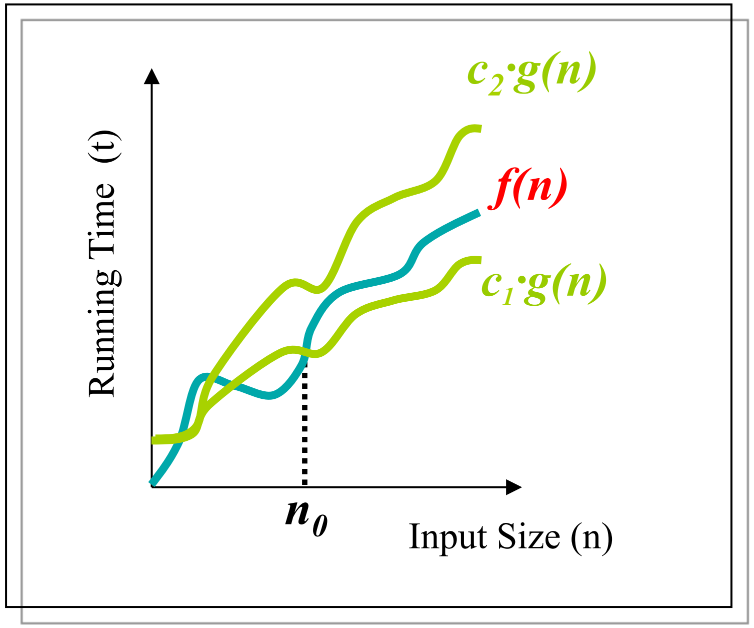
for all $n \geq n_0$ }

Big Theta

- What does this mean?
- When we use Big-Theta notation, we are saying that function *f(n)* can be "sandwiched" between some *small* constant times *g(n)* and some *larger* constant times *g(n)*.
- In other words, *f(n)* is equal to *g(n)* to within a constant factor.

Big Theta

$f(n) \in \Theta(g(n))$



Big Theta

- If $f(n) = \Theta(g(n))$, we can say that $g(n)$ is an *asymptotically tight bound* for $f(n)$.
- Basically, we are guaranteeing that f(n) never performs any better than c_1 g(n), but also never performs any worse than c_2 g(n).
- We can see this visually by noting that, after n_0, the curve for f(n) never goes below c_1 g(n) and never goes above c_2 g(n).

Big Theta

- Let us look at the performance of the merge sort.
- We said that the performance of merge sort was $cn(\log_2 n) + cn$
- Does this depend upon the characteristics of the input for merge sort? That is, does it make a difference if the list is already sorted, or reverse sorted, or in random order?
- No. Unlike insertion sort, merge sort behaves exactly the same way for any type of input.

Big Theta

- The running time of merge sort is:

  $cn(\log_2 n) + cn$

- So, using asymptotic notation, we can discard the “+ cn” part of this equation, giving:

  $cn(\log_2 n)$

- And we can disregard the constant multiplier, c, which gives us the running time of merge sort:

  $\Theta(n(\log_2 n))$

Big Theta

- Why would we prefer to express the running time of merge sort as $\Theta(n(\log_2 n))$ instead of $O(n(\log_2 n))$?
- Because Big-Theta more precise than Big-O
- If we say that the running time of merge sort is $O(n(\log_2 n))$, we are merely making a claim about merge sort's asymptotic upper bound, whereas if we say that the running time of merge sort is $\Theta(n(\log_2 n))$, we are making a claim about merge sort's asymptotic upper *and lower* bounds

Big Theta

- Would it be incorrect to say that the running time of merge sort is $O(n(\log_2 n))$?
- No, not at all.
- It is just that we are not giving all of the information that we have about the running time of merge sort.
- But sometimes all we need to know is the worst-case behavior of an algorithm. If that is so, then Big-O notation is fine.

Big Theta

- One final note: the definition of $\Theta(g(n))$ technically requires that every member f(n) $\in$ $\Theta(g(n))$ be asymptotically nonnegative – that is, f(n) must be nonnegative whenever n is sufficiently large.
- We assume that every function used within Θ notation (and the other notations used in your textbook's Chapter 3) is asymptotically nonnegative

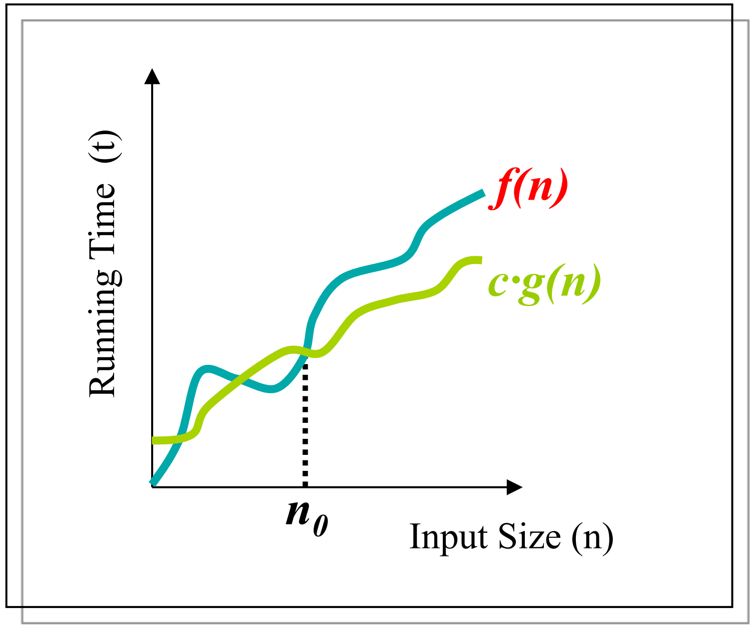
Big Omega: Definition

- For a given function g(n), Ω(g(n)) is the set of functions:

  Ω(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$ }

Big Omega

$f(n) \in \Omega(g(n))$



Big Omega

- We know that Big-O notation provides an asymptotic upper bound on a function.
- Big-Omega notation provides an *asymptotic lower bound* on a function.
- Basically, if we say that $f(n) = \Omega(g(n))$ then we are guaranteeing that, beyond n_0, $f(n)$ never performs any better than $c\ g(n)$.

Big Omega

- We usually use Big-Omega when we are talking about the *best case* performance of an algorithm.
- For example, the best case running time of insertion sort (on an already sorted list) is $\Omega(n)$.
- But this also means that insertion sort never performs any better than $\Omega(n)$ on any type of input.
- So the running time of insertion sort is $\Omega(n)$.

Big Omega

- Could we say that the running time of insertion sort is $\Omega(n^2)$?
- No. We know that if its input is already sorted, the curve for merge sort will dip below n^2 and approach the curve for n.
- Could we say that the *worst case* running time of insertion sort is $\Omega(n^2)$?
- Yes.

Big Omega

- It is interesting to note that, for any two functions f(n) and g(n), $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Little o: Definition

- For a given function g(n), o(g(n)) is the set of functions:

  o(g(n))= {f(n): for any positive constant *c*, there exists a constant n_0 such that $0 \le f(n) < c\, g(n)$ for all $n \ge n_0$ }

Little o

- Note the < instead of ≤ in the definition of Little-o:

  $0 \le f(n) < c\, g(n)$ for all $n \ge n_0$

- Contrast this to the definition used for Big-O:

  $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$

- Little-o notation denotes an *upper bound that is not asymptotically tight*. We might call this a *loose* upper bound.
- Examples: $2n \in o(n^2)$ but $2n^2 \notin o(n^2)$

Little o: Definition

- Given that $f(n) = o(g(n))$, we know that g grows *strictly faster* than f. This means that you can multiply g by a positive constant c and beyond n_0, g will always exceed f.
- No graph to demonstrate little-o, but here is an example:

  $n^2 = o(n^3)$ but

  $n^2 \neq o(n^2)$.

  Why? Because if c = 1, then $f(n) = c\, g(n)$, and the definition insists that $f(n)$ be <u>less than</u> $c\, g(n)$.

Little omega: Definition

- For a given function g(n), $\omega(g(n))$ is the set of functions:

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

Little omega: Definition

- Note the < instead of ≤ in the definition:

  $0 \le c\, g(n) < f(n)$

- Contrast this to the definition used for Big-Ω:

  $0 \le c\, g(n) \le f(n)$

- Little-omega notation denotes a *lower bound that is not asymptotically tight.* We might call this a *loose* lower bound.
- Examples:

  $n \notin \omega(n^2)$ $n \in \omega(\sqrt{n})$ $n \in \omega(\lg n)$

Little omega

- No graph to demonstrate little-omega, but here is an example:

  n^3 is $\omega(n^2)$ but

  $n^3 \neq \omega(n^3)$.

  Why? Because if c = 1, then *f(n)* = *c g(n)*, and the definition insists that *c g(n)* be strictly <u>less than</u> *f(n)*.

Comparison of Notations

$f(n) = o(g(n)) \approx a < b$

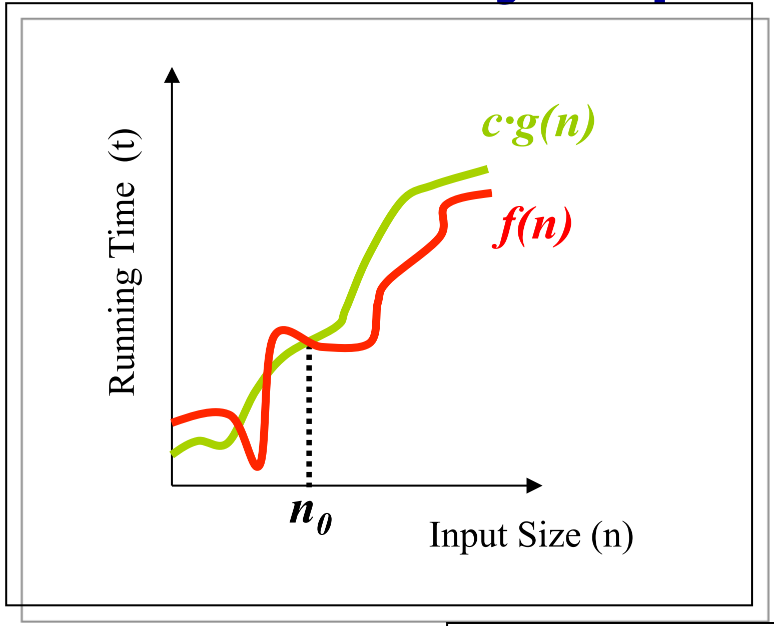
$f(n) = O(g(n)) \approx a \leq b$

$f(n) = \Theta(g(n)) \approx a = b$

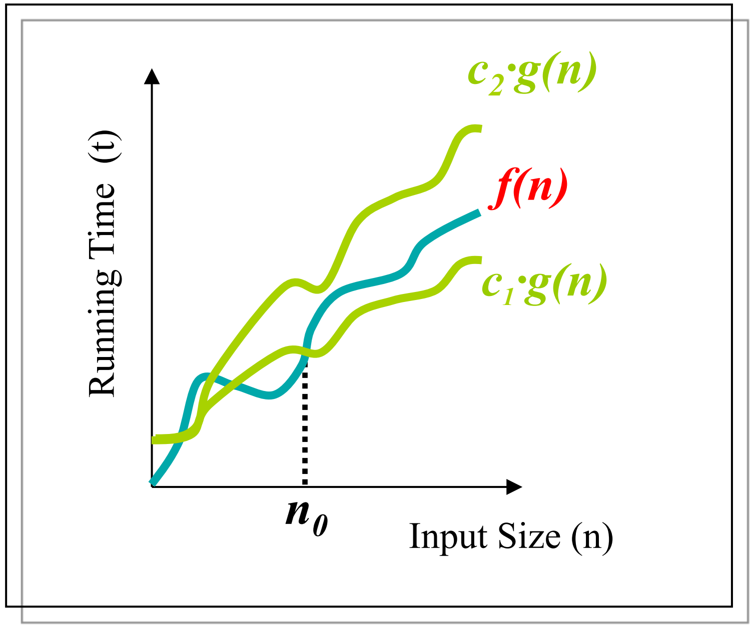
$f(n) = \Omega(g(n)) \approx a \geq b$

$f(n) = \omega(g(n)) \approx a > b$

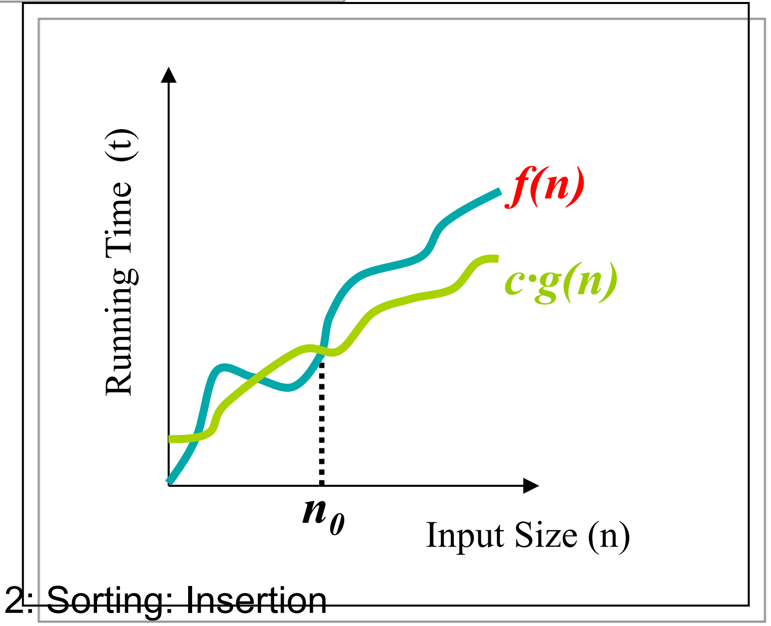
Asymptotic Notation



Big O



Big Theta



Big Omega

Asymptotic Notation in Equations and Inequalities

- When asymptotic notation stands alone on right-hand side of equation, '=' is used to mean '$\in$'.
- In general, we interpret asymptotic notation as standing for some anonymous function we do not care to name.
- Example: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for some $f(n) \in \Theta(n)$. (In this case, $f(n) = 3n + 1$, which is in $\Theta(n)$.)

Asymptotic Notation in Equations and Inequalities

- This use of asymptotic notation eliminates inessential detail in an equation (e.g., we do not have to specify lower-order terms; they are understood to be included in anonymous function).
- The number of anonymous functions in an expression is the number of times asymptotic notation appears
  - Example: $\sum_{i=1}^{n} O(i)$ is <u>one</u> anonymous function
  - not the same as O(1)+O(2)+…+O(n), which has n hidden constants

Asymptotic Notation in Equations and Inequalities

- Appearance of asymptotic notation on left-hand side of equation means, no matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.
- Example: $2n^2 + \Theta(n) = \Theta(n^2)$ means that for any function $f(n) \in \Theta(n)$ there is some function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n.

Comparison of Functions

- Transitivity:

  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$

  $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$

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

  $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$

  $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

Comparison of Functions

- Reflexivity:

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

  $f(n) = O(f(n))$

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

Comparison of Functions

- Symmetry:

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

Comparison of Functions

- Transpose symmetry:

  $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Comparison of Functions

- Analogies:

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \omega(g(n)) \approx a > b$$

Comparison of Functions

- Asymptotic relationships:

- f(n) is asymptotically smaller than g(n) if

  $f(n) = o(g(n))$
- f(n) is asymptotically larger than g(n) if

  $f(n) = \omega(g(n))$

Comparison of Functions

- Asymptotic relationships:
- Not all functions are asymptotically comparable.
- That is, it may be the case that neither f (n) = o(g(n)) nor f(n) = ω(g(n)) is true.

Using limit of ratio to show order of growth of a function

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0\Rightarrow f(n)=o(g(n))\Rightarrow f(n)=O(g(n))$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c,c>0,c<\infty\Rightarrow f(n)=\Theta(g(n))\Leftrightarrow$$

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

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty\Rightarrow f(n)=\omega(g(n))\Rightarrow f(n)=\Omega(g(n))$$

Standard Notation

- Pages 51 – 56 contain review material from your previous math courses
- Please read this section of your textbook and refresh your memory of these mathematical concepts
- The remaining slides in this section are for your aid in reviewing the material

Monotonicity

- A function f(n) is *monotonically increasing* if $m \le n$ implies $f(m) \le f(n)$.
- A function f(n) is *monotonically decreasing* if $m \le n$ implies $f(m) \ge f(n)$.
- A function f(n) is *strictly increasing* if $m < n$ implies $f(m) < f(n)$.
- A function f(n) is *strictly decreasing* if $m < n$ implies $f(m) > f(n)$.

Floor and Ceiling

- For any real number x, the floor of x is the greatest integer less than or equal to x.
- The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing.
- For any real number x, the ceiling of x is the least integer greater than or equal to x.
- The ceiling function $f(x) = \lceil x \rceil$ is monotonically increasing.

Modulo Arithmetic

- For any integer *a* and any positive integer *n*, the value of *a modulo n* (or *a mod n*) is the remainder we have after dividing *a* by *n*.
- a mod n = a - $\lfloor a/n \rfloor$ n
- if (a mod n) = (b mod n), then a ≡ b mod n (read as “a is equivalent to b mod n”)

Polynomials

- Given a nonnegative integer d, a polynomial in n of degree d is a function p(n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the constants a_0, a_1, ..., a_d are the coefficients of the polynomial and $a_d \neq 0$.

Polynomials

- A polynomial is asymptotically positive if and only if $a_d > 0$.
- If a polynomial $p(n)$ of degree d is asymptotically positive, then $p(n) = \Theta(n^d)$.
- For any real constant $a \geq 0$, n^a is monotonically increasing.
- For any real constant $a \leq 0$, n^a is monotonically decreasing.
- A function is polynomially bounded if f (n) = $O(n^k)$ for some constant k.

Exponentials

- For all n and a ≥ 1, the function a^n is monotonically increasing in n.
- For all real constants a and b such that a > 1,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

This means that $n^b = o(a^n)$, which means that any *exponential* function with a base strictly greater than 1 grows *faster* than any *polynomial* function.

Logarithms

- $\lg n = \log_2 n$ (binary logarithm)
- $\ln n = \log_e n$ (natural logarithm)
- $\lg^k n = (\lg n)^k$ (exponentiation)
- $\lg \lg n = \lg (\lg n)$ (composition)
- $\lg n + k$ means $(\lg n) + k$, not $\log (n + k)$
- If $b > 1$ and we hold b constant, then, for $n > 0$, the function $\log_b n$ is strictly increasing.
- Changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor.

Logarithms

- A function is *polylogarithmically bounded* if f(n) = $O(\lg^k n)$ for some constant k.
- $\lg^b n = o(n^a)$ for any constant $a > 0$
- This means that any positive polynomial function grows faster than any polylogarithmic function.

Factorials

- N factorial is defined for integers ≥ 0 as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \bullet (n-1)! & \text{if } n > 0 \end{cases}$$

- A weak upper bound on $n!$ is $n! \leq n^n$
  - $n! = o(n^n)$
  - $n! = \omega(2^n)$
  - $\lg(n!) = \Theta(n \lg n)$

Fibonacci Numbers

- The Fibonacci numbers are defined by the recurrence:

  $F_0 = 0$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2} \geq 2$
- Fibonacci numbers grow exponentially