CS590/CPE590

Analysis of Algorithms | Sorting | Computational Complexity

Kazi Lutful Kabir

Spring 2023

Define a Problem, and Solve It

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational steps that transform the input into the output
- **Data Structure:**
  - An organized method of storing and retrieving data
- **Our Task:**
  - Given a problem, design a ***correct*** and ***good*** algorithm that solves it.

Importance of Analysis of Algorithms

- To recognize **limitations of various algorithms** for solving a problem
- To understand **relationship between problem size and running time**
- To learn how to **analyze an algorithm's running time** without coding it
- To learn techniques for writing **more efficient code**
- To recognize bottlenecks in code as well as which parts of code are easiest to **optimize**

What do we Analyze?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (complexity)
  - Basic operations to do task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

Input Size

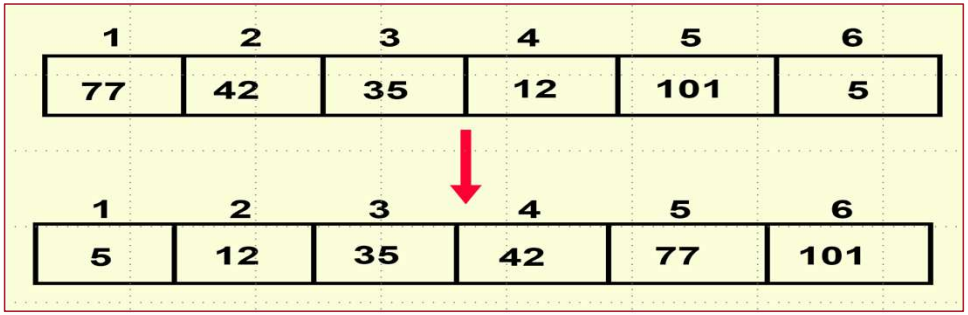
- Time and space complexity
  - This is generally a function of the input size
    - e.g., sorting, multiplication
  - How we characterize input size depends
    - Sorting: number of input items
    - Graph algorithms: number of nodes & edges
    - Multiplication of 2 Integers: total number of bits in the two integers

Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b

- The time taken by an algorithm depends on the input.
  -- Sorting 1000 numbers takes longer than sorting 10 numbers.
  -- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
  -- For example, we will see that insertion sort takes less time to sort *n* elements when they are already sorted than when they are in reverse sorted order.

The Sorting Problem



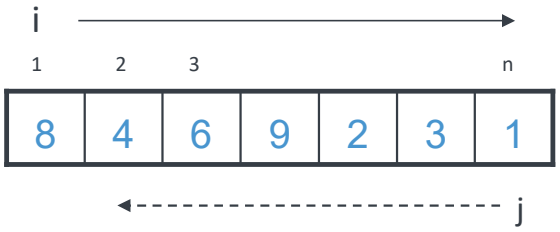
- Problem: Sort real numbers in ascending order
- Problem Statement:
  - **Input:** A sequence of n numbers $\langle a_1, \ldots, a_n \rangle$
  - **Output:** A permutation $\langle a_1', \ldots, a_n' \rangle$ s.t. $a_1' \leq a_2' \leq \ldots \leq a_n'$
- There are many sorting algorithms. How many can you list?

An Incomplete List of Sorting Algorithms

- Selection sort
- *Insertion* sort
- Library sort
- Shell sort
- Gnome sort
- Bubble sort
- Comb sort
- Flash sort
- Bucket sort
- Radix sort
- Counting sort
- Pigeonhole sort
- *Mergesort*
- Quicksort
- Heap sort
- Smooth sort
- Binary tree sort
- Topological sort

Bubble Sort

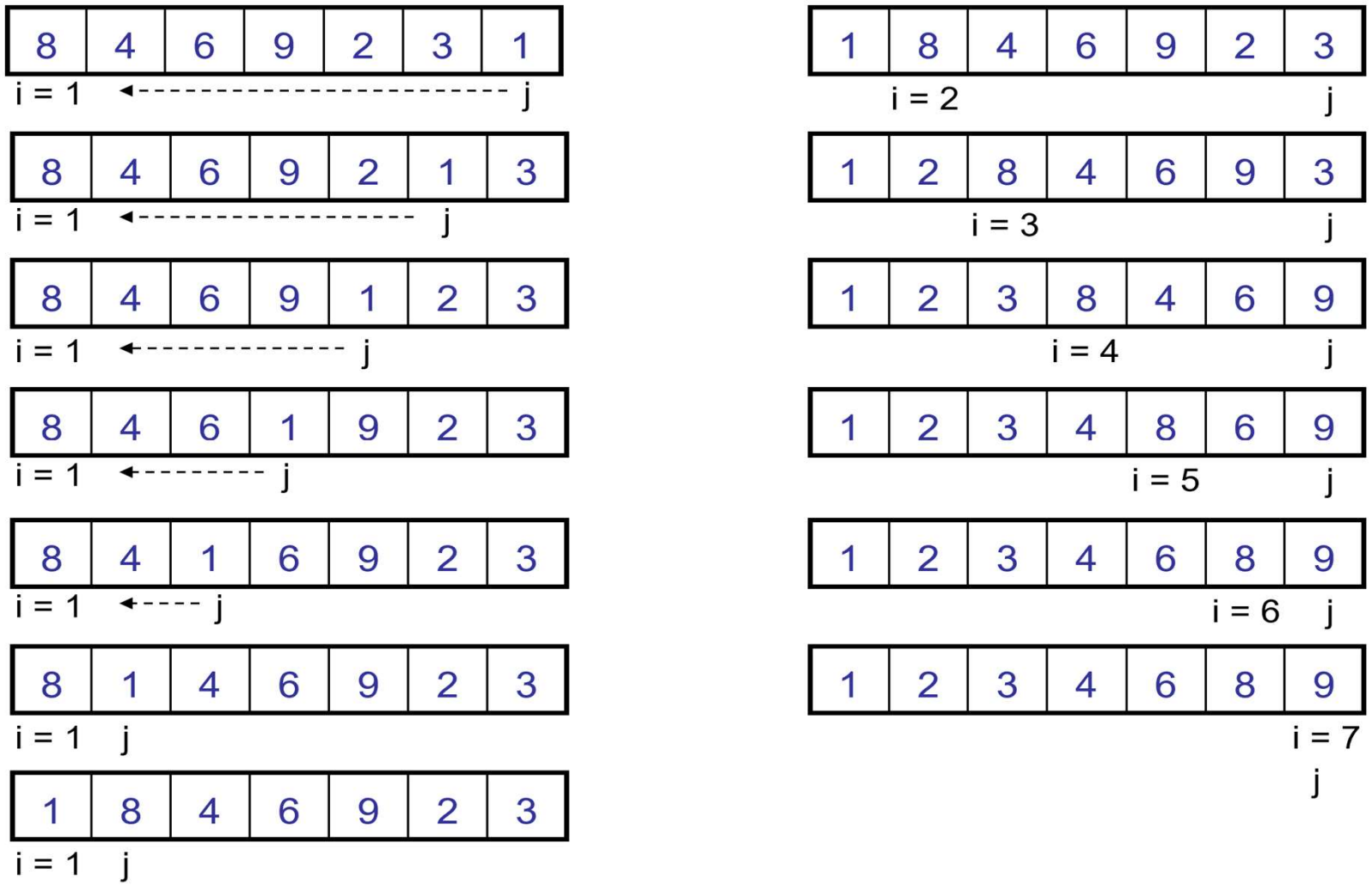
- **Idea:**
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slow

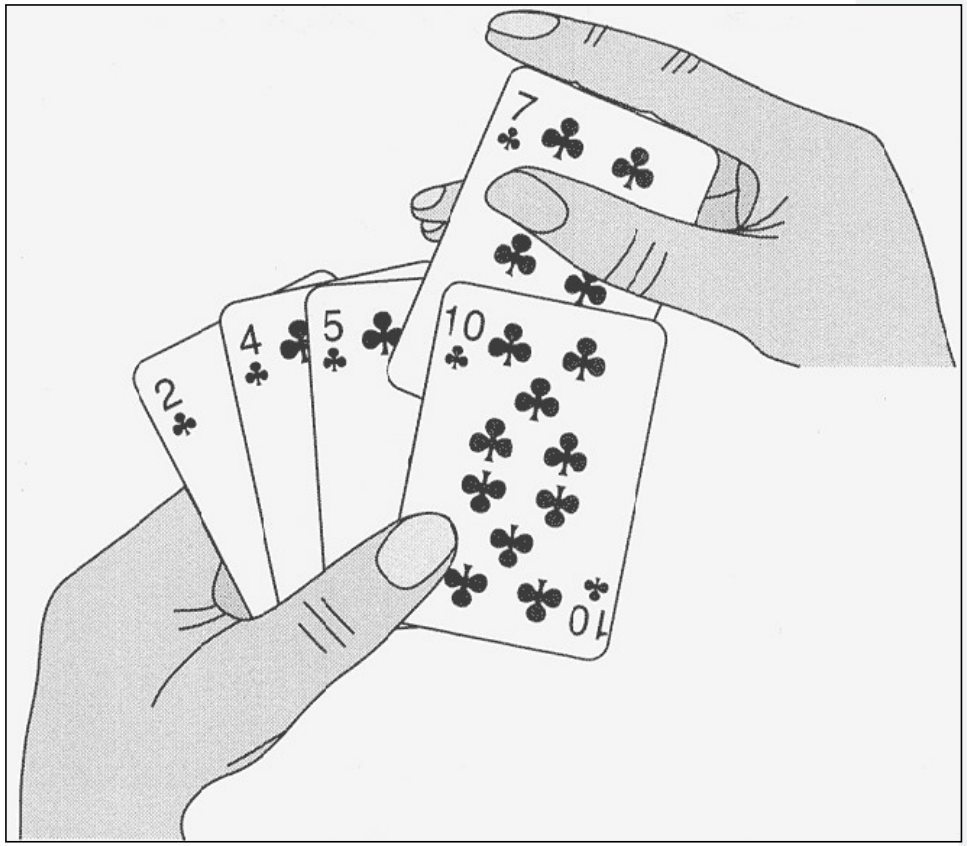
Bubble Sort

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```


8 4 6 9 2 3 1
i = 1
j
8 4 6 9 2 1 3
i = 1
j
8 4 6 9 1 2 3
i = 1
j
8 4 6 1 9 2 3
i = 1
j
8 4 1 6 9 2 3
i = 1
j
8 1 4 6 9 2 3
i = 1
j
1 8 4 6 9 2 3
i = 1
j
1 8 4 6 9 2 3
i = 2
j
1 2 8 4 6 9 3
i = 3
j
1 2 3 8 4 6 9
i = 4
j
1 2 3 4 8 6 9
i = 5
j
1 2 3 4 6 8 9
i = 6
j
1 2 3 4 6 8 9
i = 7
j

Another Example: Insertion Sort

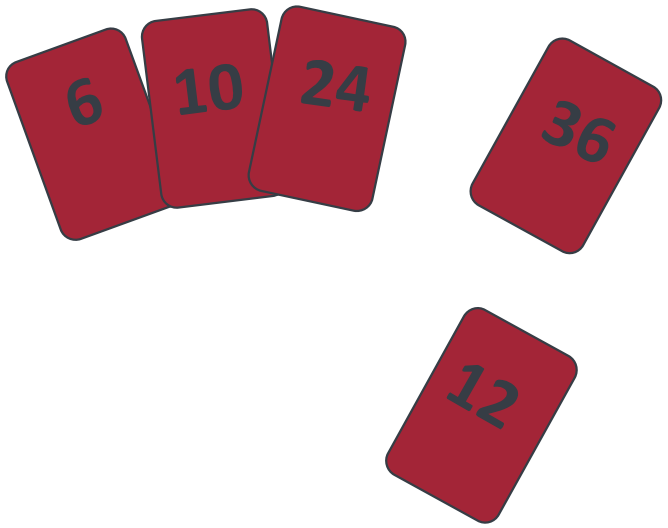
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table



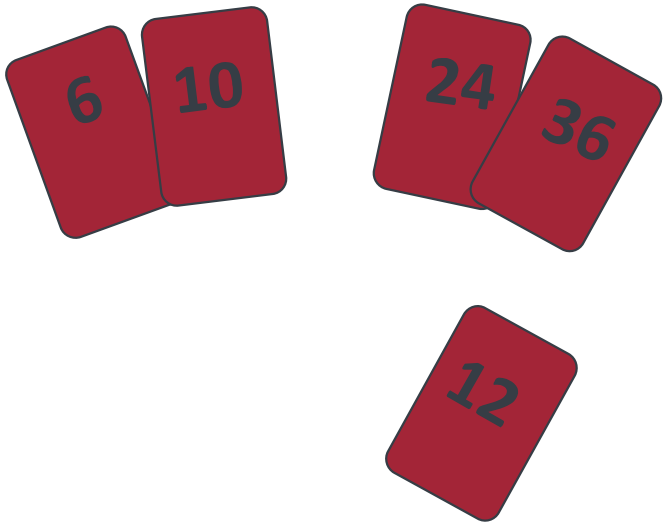
Insertion Sort

To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort

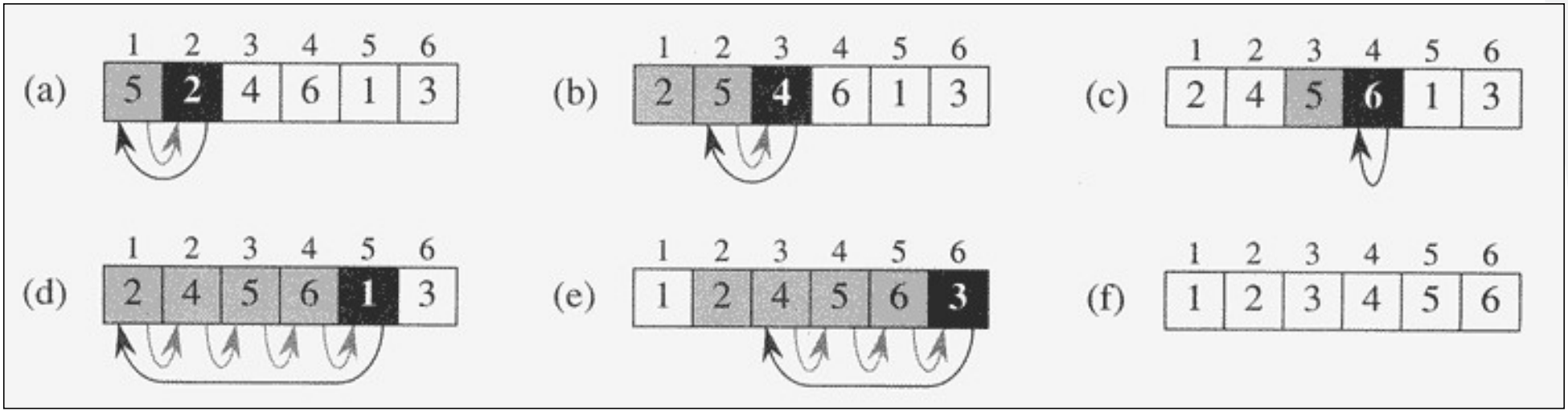


Insertion Sort

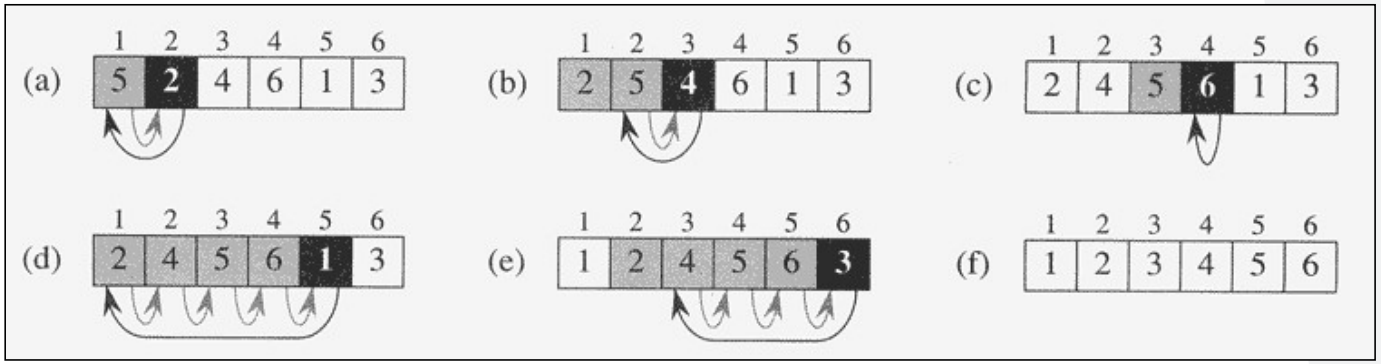


An Example: Insertion Sort

$A = \{5, 2, 4, 6, 1, 3\}$



An Example: Insertion Sort



INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Insertion Sort: Correctness

We often use a **loop invariant** to help us understand why an algorithm gives the correct answer.

Loop Invariant:

- At the start of each iteration of the "outer" for loop – the loop indexed by i.
- The subarray $A[1, \cdots, i-1]$ consists of the elements originally in $A[1, \cdots, i-1]$ but in sorted order.

To prove correctness, we must show three things about loop invariant:

- **Initialization** – It is true prior to the first iteration of the loop.
- **Maintenance** – If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination** – When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort

Initialization:

- Show the loop invariant holds before the 1st loop: $i = 2$
- The subarray $A[1, \cdots, i-1]$ consists the single element $A[1]$ which is the original element in $A[1]$.
- This subarray is sorted showing the loop invariant holds prior to the first iteration of the loop.

Maintenance:

- Shows each iteration maintains the loop invariant – for loop works by moving $A[i-1], A[i-2], A[i-3]$, ... by one position to the right until it finds the proper position for $A[i]$ at point it inserts the value of $A[i]$
- The subarray consists in sorted order.
- Increment i for the next iteration of the for loop then preserves the loop invariant.

Termination:

- The for loop terminates when $i > A.length = n$ and we have $i = n + 1$ at that time.
- We have subarray $A[1, \cdots, n]$ in sorted order.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

Analyzing Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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 **while** $i > 0$ 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	$n-1$

Analyzing Insertion Sort

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

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

Analyzing Insertion Sort

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

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

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

Analyzing Insertion Sort

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

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

and

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

Analyzing Insertion Sort

the worst case, the running time of INSERTION-SORT is

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

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Asymptotic Performance

- We care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

Asymptotic Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs
- Best case
  - Provides a lower bound on running time

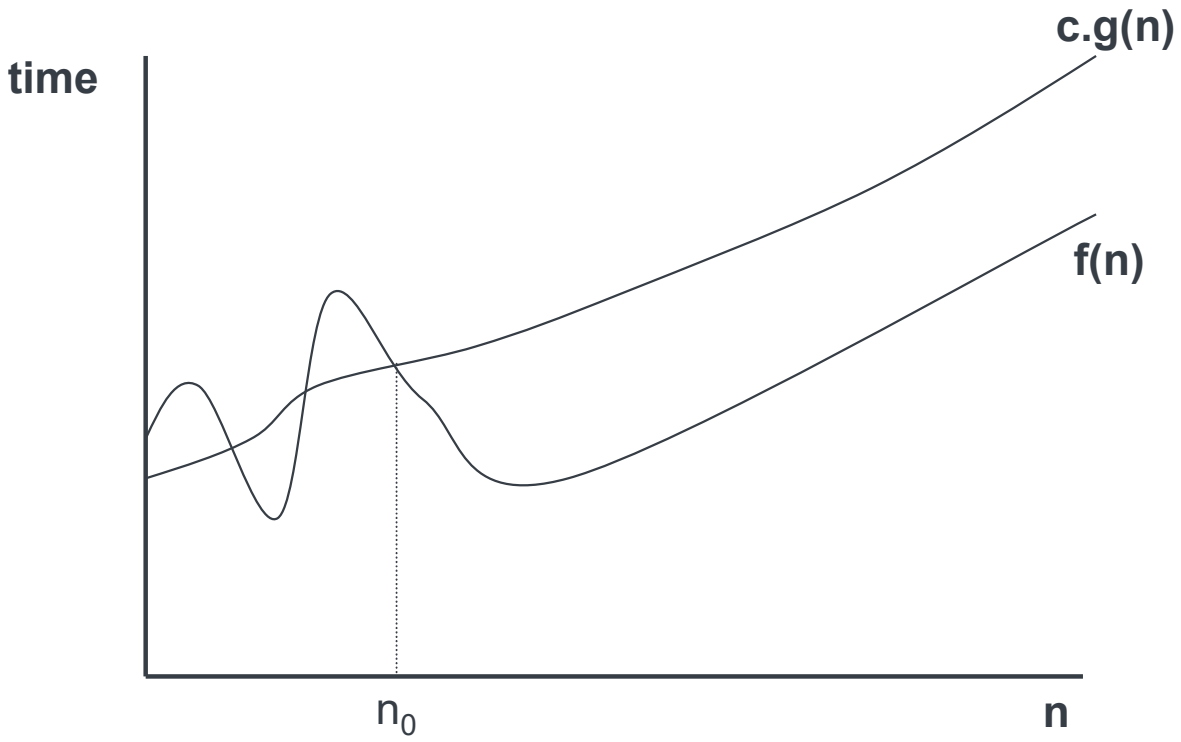
Analyzing Insertion Sort

- **Best case:** the array is sorted (inner loop body never executed)
  - $T(n) = an + b$, a **linear** function of n
- **Worst case:** the array is reverse sorted (inner loop body executed for all previous elements)
  - $T(n) = an^2 + bn + c$, a **quadratic** function of n

Upper Bound Notation

- We say Insertion Sort's run time is $\boldsymbol{O(n^2)}$
  - Properly we should say run time is *in* $\boldsymbol{O(n^2)}$
  - Read *O* as "Big-*O*" (you'll also hear it as "order")
- In general, a function
  - $\boldsymbol{f(n)}$ **is** $\boldsymbol{O(g(n))}$ if there exist positive constants c and n_0 such that $\boldsymbol{0 \leq f(n) \leq c \cdot g(n)}$ **for all** $\boldsymbol{n \geq n_0}$
- Formally
  - $O(g(n)) = \{ f(n)$: $\exists$ positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0 \}$

Upper Bound Notation



We say $g(n)$ is an asymptotic upper bound for $f(n)$

Insertion Sort is $O(n^2)$

- Proof
  - The run-time is **$an^2 + bn + c$**
    - If any of a, b, and c are less than 0, replace the constant with its absolute value
  - $an^2 + bn + c \leq (a + b + c)n^2 + (a + b + c)n + (a + b + c)$
    $\leq 3(a + b + c)n^2$ for $n \geq 1$

    Let $c' = 3(a + b + c)$ and let $n_0 = 1$. Then

    $an^2 + bn + c \leq c' n^2$ for $n \geq 1$

    Thus $an^2 + bn + c \Rightarrow O(n^2)$.
- Question
  - Is Insertion Sort $O(n^3)$?
  - Is Insertion Sort $O(n)$?

Is $100n + 5 \in O(n^2)$?

- To show that a function is O(g(n)), we need to give values for the positive constants c and n_0.
- **$f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$**
- $0 \leq 100n + 5 \leq 100n + n = 101n \leq 101n^2$
  $c = 101$
  $n_0 = 2$
  Note: c do not have to be integers.
- **Ans: Yes**

Is $100n + 5 \in \mathbf{O(n^2)}$?

- Alternative way:
  $0 \leq 100n + 5 \leq 100n + 5n \ (\forall n \geq 1) = 105n \leq 105n^2$
  $c = 105$
  $n_0 = 1$

- Note: in practice we usually want the upper bound to be as tight as possible by having the same order of growth as the bounded function.
- Prove that **$100n + 5 \in O(n)$** rather than $100n + 5 \in O(n^2)$) and with a constant c as small as possible.

Find an upper bound for f(n) = 3n + 8.

- Looking only at the highest order term 3n and disregarding the coefficient, we can guess that **f(n) ∈ O(n)**.
- Provide the smallest integer value of c that proves it.
  $0 \leq 3n + 8 \leq 3n + n \ (\forall n \geq 8) = 4n$
  c = 4
  n_0 = 8
- The constant c cannot be smaller if we require it to be an integer. (However, A non-integer constant c like 3.01 would work too to prove that f(n) ∈ O(n)).

Find an upper bound for $f(n) = n^2 + 1$.

- Looking only at the highest order term n^2 we guess that **$f(n) \in O(n^2)$**.

- Provide the smallest integer value of c that proves it.

  $0 \leq n^2 + 1 \leq 2n^2 \quad (\forall n \geq 1)$

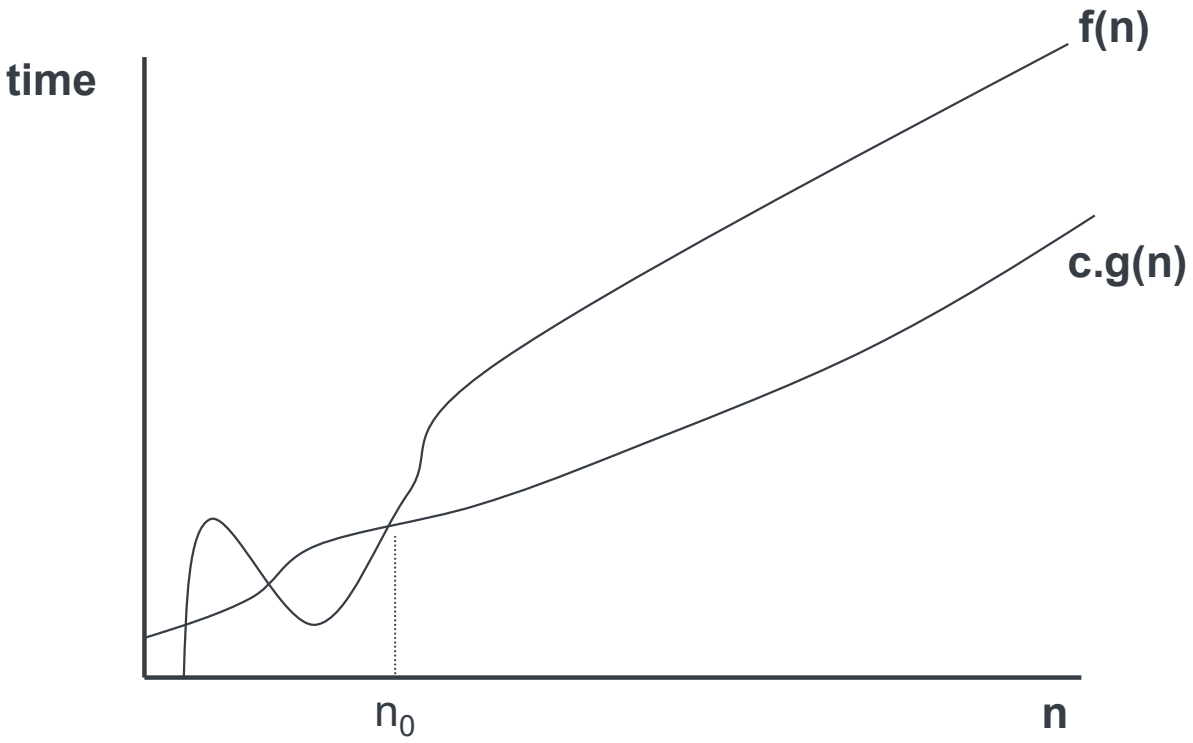
  $c = 2$

  $n_0 = 1$

Lower Bound Notation

- We say Insertion Sort's run time is **$\Omega(n)$**
- In general, a function
  - **$f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall\ n \geq n_0$**
- Proof:
  - Suppose run time is an + b
    - Assume a and b are positive
  - an $\leq$ an + b

Lower Bound Notation



We say $g(n)$ is an asymptotic lower bound for $f(n)$

Is $n^3 \in \Omega(n^2)$?

$0 \leq n^2 \leq n^3 \quad (\forall n \geq 1)$

$c = 1$

$n_0 = 1$

Ans: Yes

Find a lower bound for $f(n) = 5n^2$.

Looking only at the highest order term $5n^2$ and disregarding the coefficient, we guess that $f(n) \in \Omega(n^2)$.

Provide the biggest integer value of c that proves it.

$\mathbf{0 \leq 5n^2 \leq 5n^2 \ (\forall n)}$

$\mathbf{c = 5}$

$\mathbf{n_0 = 1}$

Is $f(n) = 100n + 5 \in \Omega(n^2)$? NO

- We need to find positive constants c and n_0 such that
  $0 \leq cn^2 \leq 100n + 5 \ (\forall n \geq n_0)$
  We have $100n + 5 \leq 100n + 5n \ (\forall n \geq 1) = 105n$
- So, by combining the two we must have:
  $cn^2 \leq 105n \ (\forall n \geq \max(n_0, 1))$
  $n(cn - 105) \leq 0 \ (\forall n \geq \max(n_0, 1)$, so $n > 0)$
  $cn - 105 \leq 0 \ (\forall n \geq \max(n_0, 1))$
  $n \leq 105/c \ (\forall n \geq \max(n_0, 1))$
- Since n can grow to infinity, it is impossible to find a positive constant c such that n is bounded above by the constant 105/c. Therefore, the c we need to find does not exist. Therefore $f(n) = 100n + 5 \notin \Omega(n^2)$.

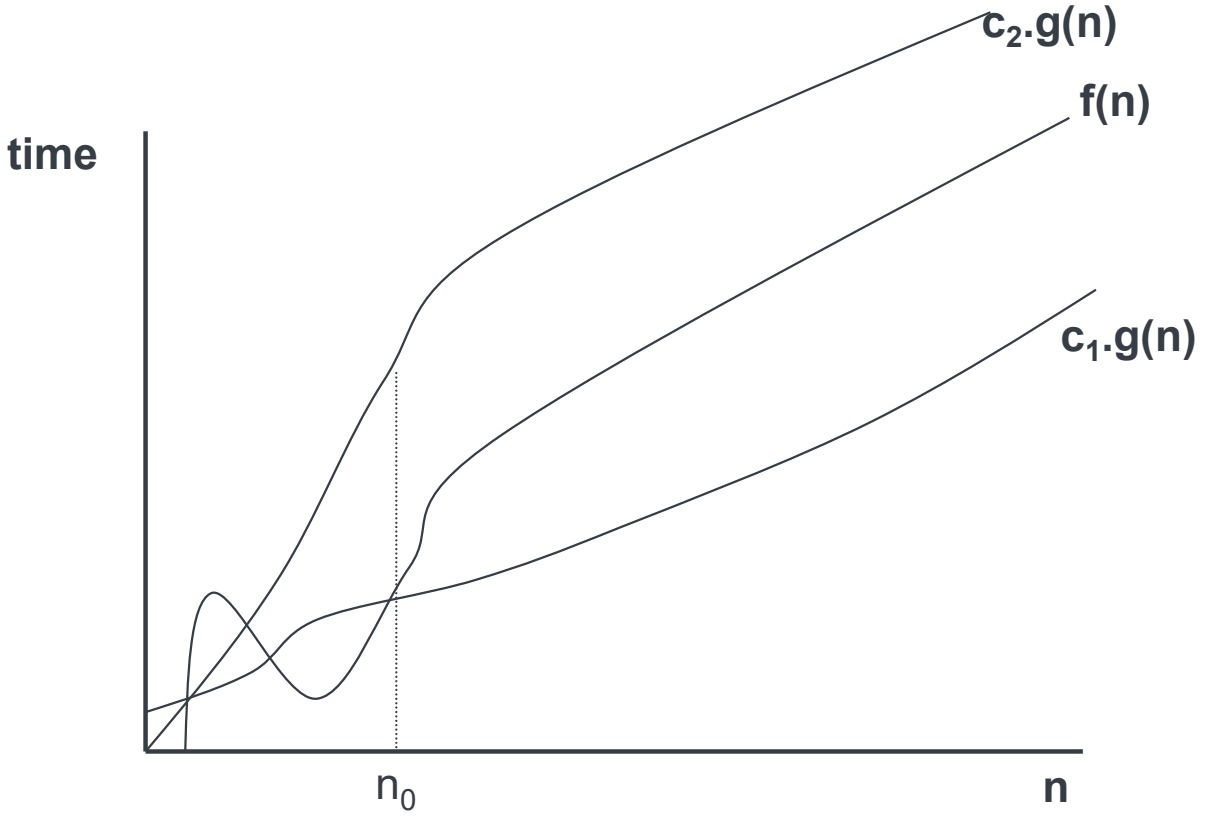
Asymptotic Tight Bound

- A function $f(n)$ is $\Theta(g(n))$ if $\exists$ positive constants c_1, c_2, and n_0 such that

$$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\ \forall\ n \geq n_0$$

- Theorem
  - **$f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$**

Asymptotic Tight Bound



We say $g(n)$ is an **asymptotic tight bound** for $f(n)$

Is $n(n-1)/2 \in \Theta(n^2)$? YES

- We prove the **upper bound** first:
  $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2 \ (\forall n \geq 1)$
- We prove the **lower bound** next:
  We have: $\mathbf{n(n-1)/2 = n^2/2 - n/2}$
  And $\forall n \geq 2$ we have: $1 \leq n/2$
  so, $-1 \geq -n/2$
  so, $-n/2 \geq -(n/2)(n/2)$
  so, $n^2/2 - n/2 \geq n^2/2 - (n/2)(n/2)$.
  By combining the two equations we then get:
  $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2)(n/2) = n^2/4 \ (\forall n \geq 2)$
  From these upper and lower bounds we get $\mathbf{c_1 = 1/4}$, $\mathbf{c_2 = 1/2}$, and $\mathbf{n_0 = 2}$(the maximum of $\forall n \geq 1$ for the upper bound and $\forall n \geq 2$ for the lower bound).
- Note: we are not trying to find integer constants c_1 and c_2 here. We could easily use $c_2 = 1$ but there is no integer c_1 that would work.

Is $n \in \Theta(n^2)$? NO

We need c_1 and c_2 and n_0 such that:

$c_1 n^2 \leq n \leq c_2 n^2 \quad (\forall n \geq n_0)$

The first part looks suspicious:

$c_1 n^2 \leq n \quad (\forall n \geq n_0$, with n_0 positive so n is positive too)

$c_1 n \leq 1 \quad (\forall n \geq n_0)$

$n \leq 1/c_1 \quad (\forall n \geq n_0)$

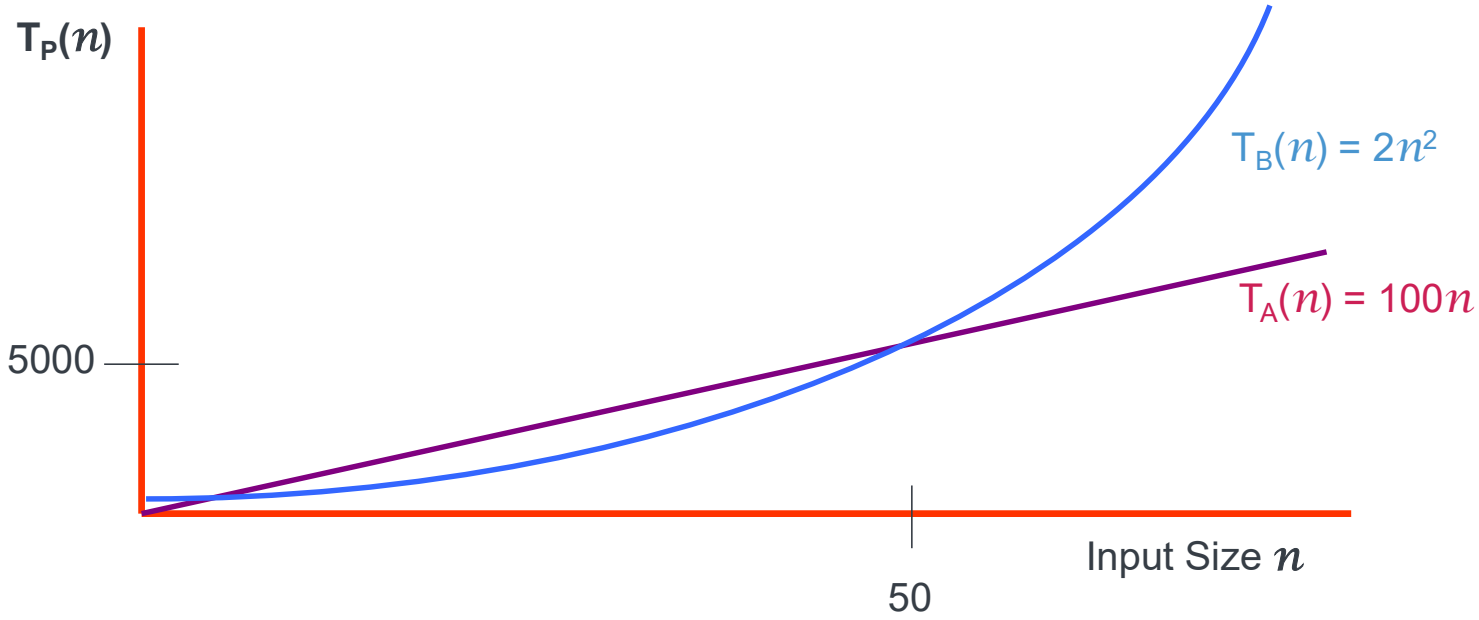
Contradiction: n cannot be smaller than a constant!

Practical Complexity

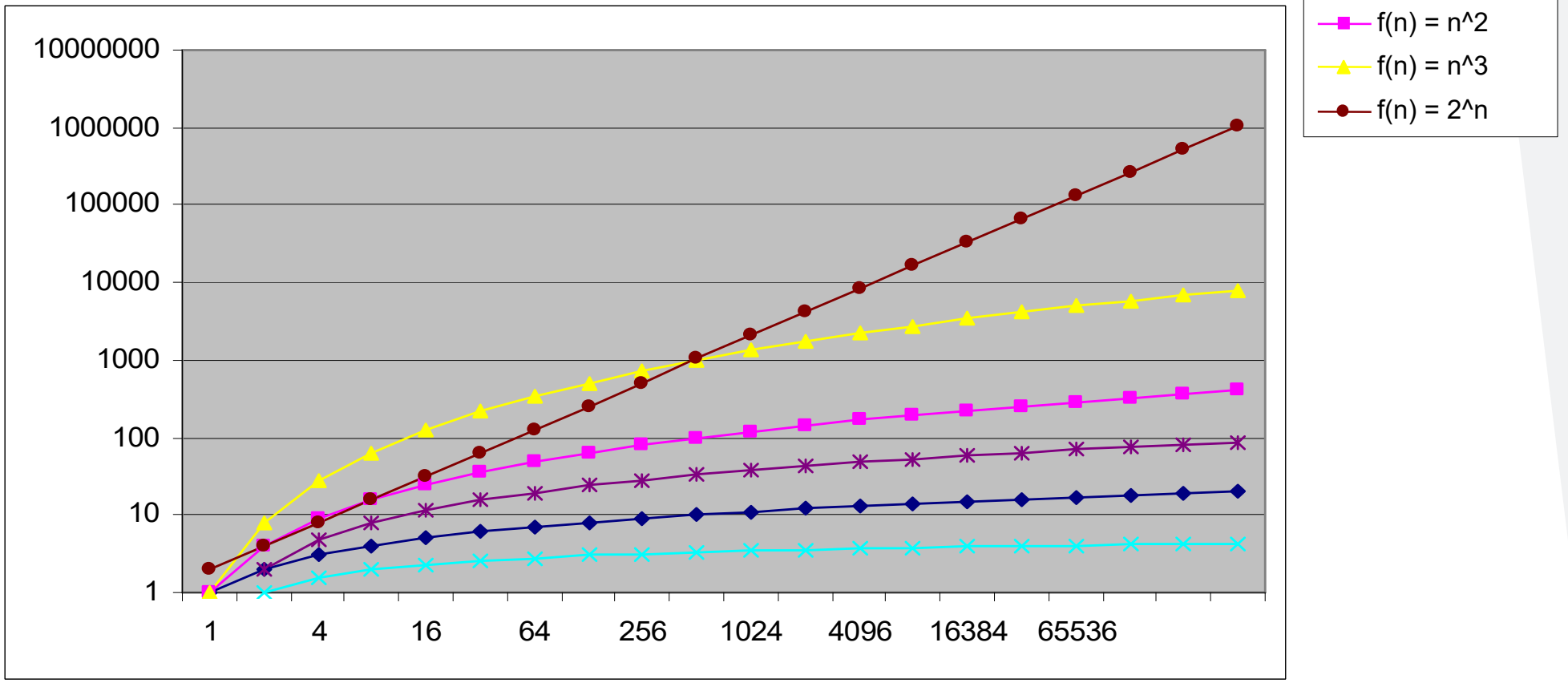
For large input sizes, constant terms are insignificant

Program *A* with running time $T_A(n) = 100n$

Program *B* with running time $T_B(n) = 2n^2$



Practical Complexity

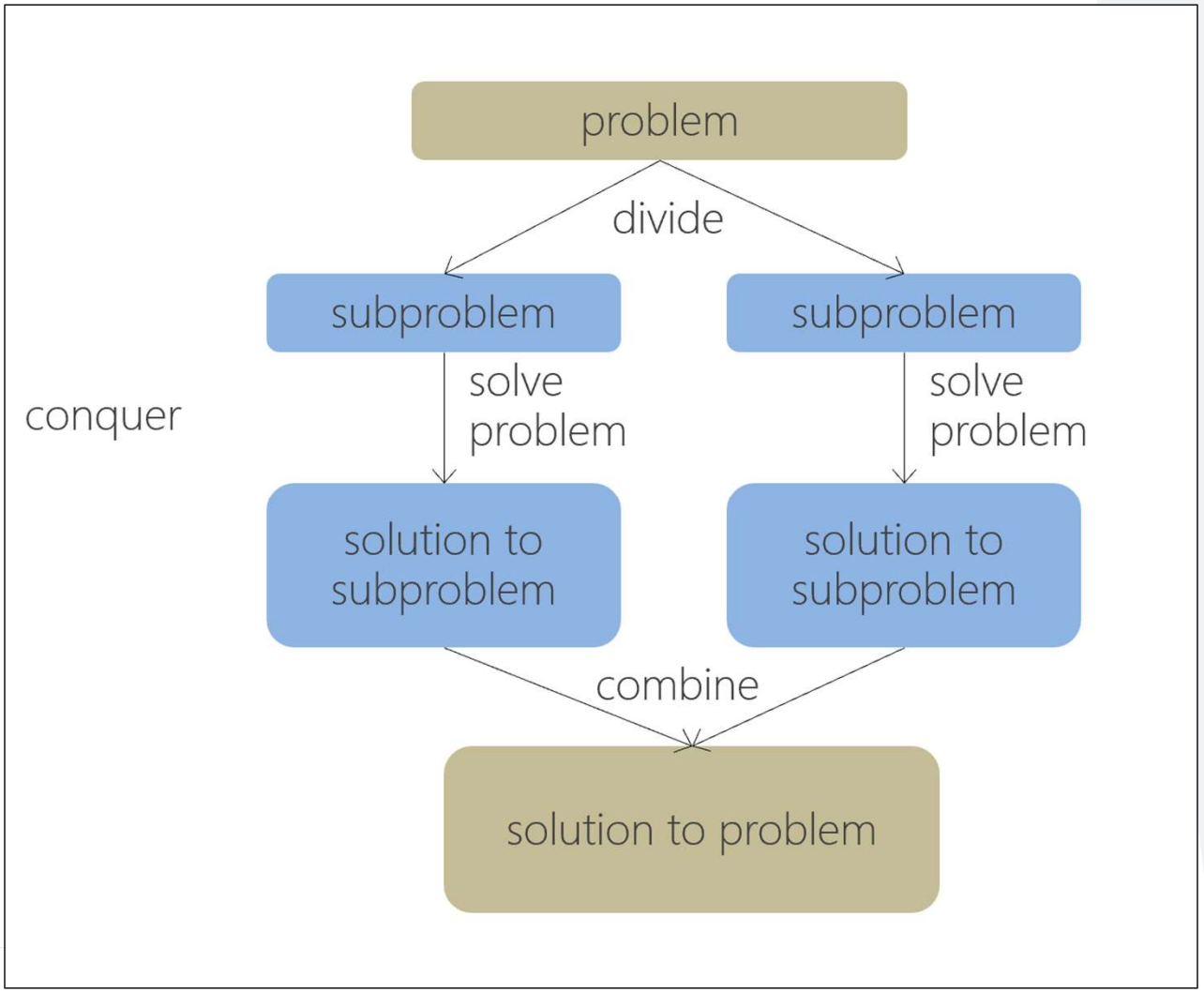


Practical Complexity

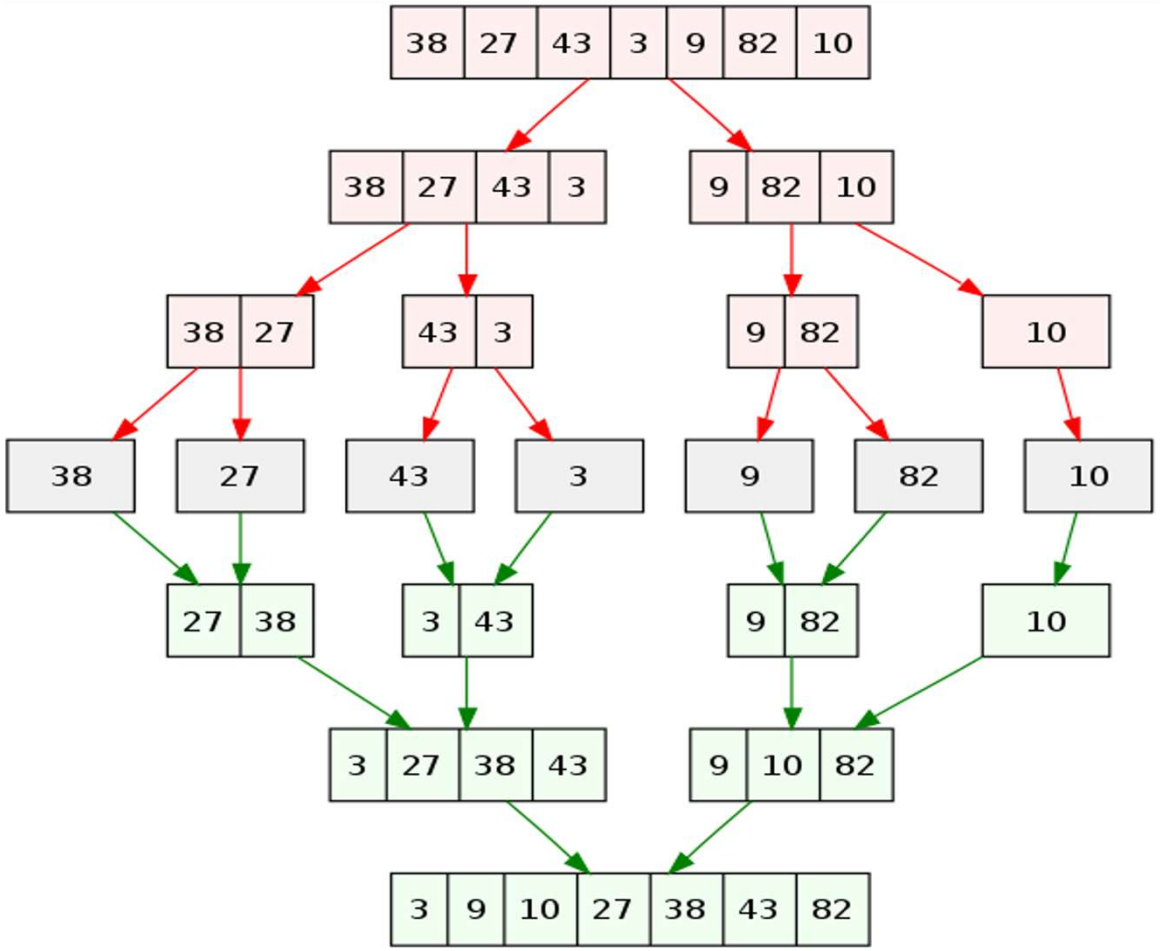
Function	Descriptor	Big-O
c	Constant	$O(1)$
$\log n$	Logarithmic	$O(\log n)$
n	Linear	$O(n)$
$n \log n$	$n \log n$	$O(n \log n)$
n^2	Quadratic	$O(n^2)$
n^3	Cubic	$O(n^3)$
n^k	Polynomial	$O(n^k)$
2^n	Exponential	$O(2^n)$
$n!$	Factorial	$O(n!)$

Merge Sort

- A divide-and-conquer algorithm
- Divide the array into two smaller arrays of about equal sizes
- Sort each smaller array ***recursively***
- Merge the two sorted arrays to get one sorted array



Merge Sort Example



Questions to Ponder

How do we divide the array?
How much time is needed?

How do we merge the two sorted arrays?
How much time is needed?

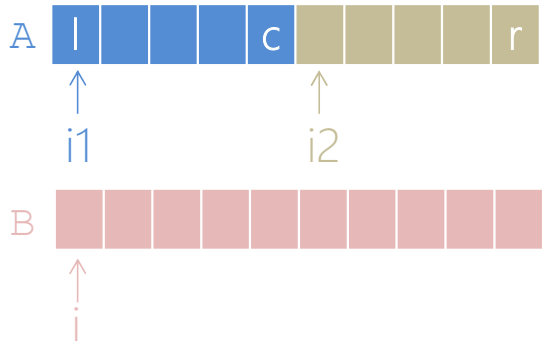
Mergesort

```
MERGESORT(A, B, lo, hi)
    IF lo < hi
        mid = lo + (hi-lo)/2
        MERGESORT(A, B, lo, mid)
        MERGESORT(A, B, mid+1, hi)
        MERGE(A, B, lo, mid, hi)
```

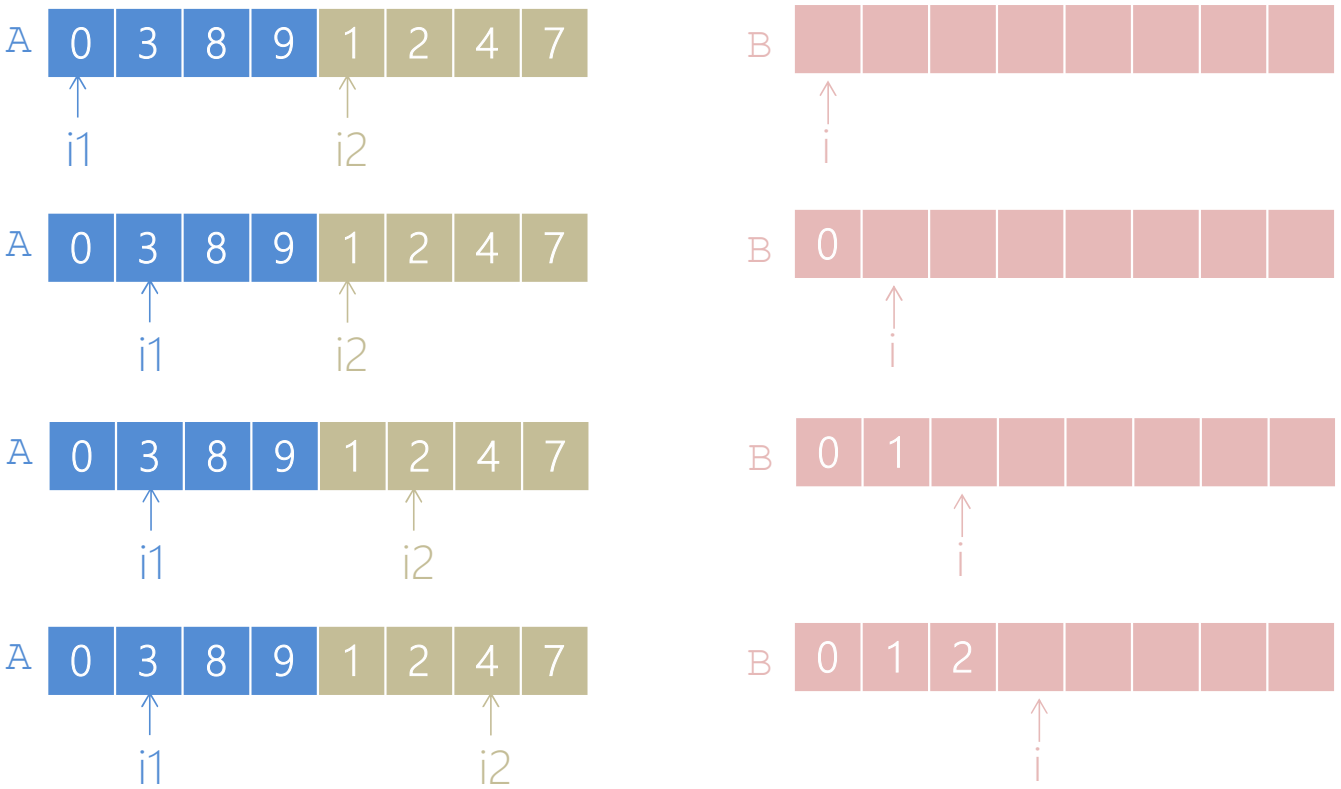
Merging

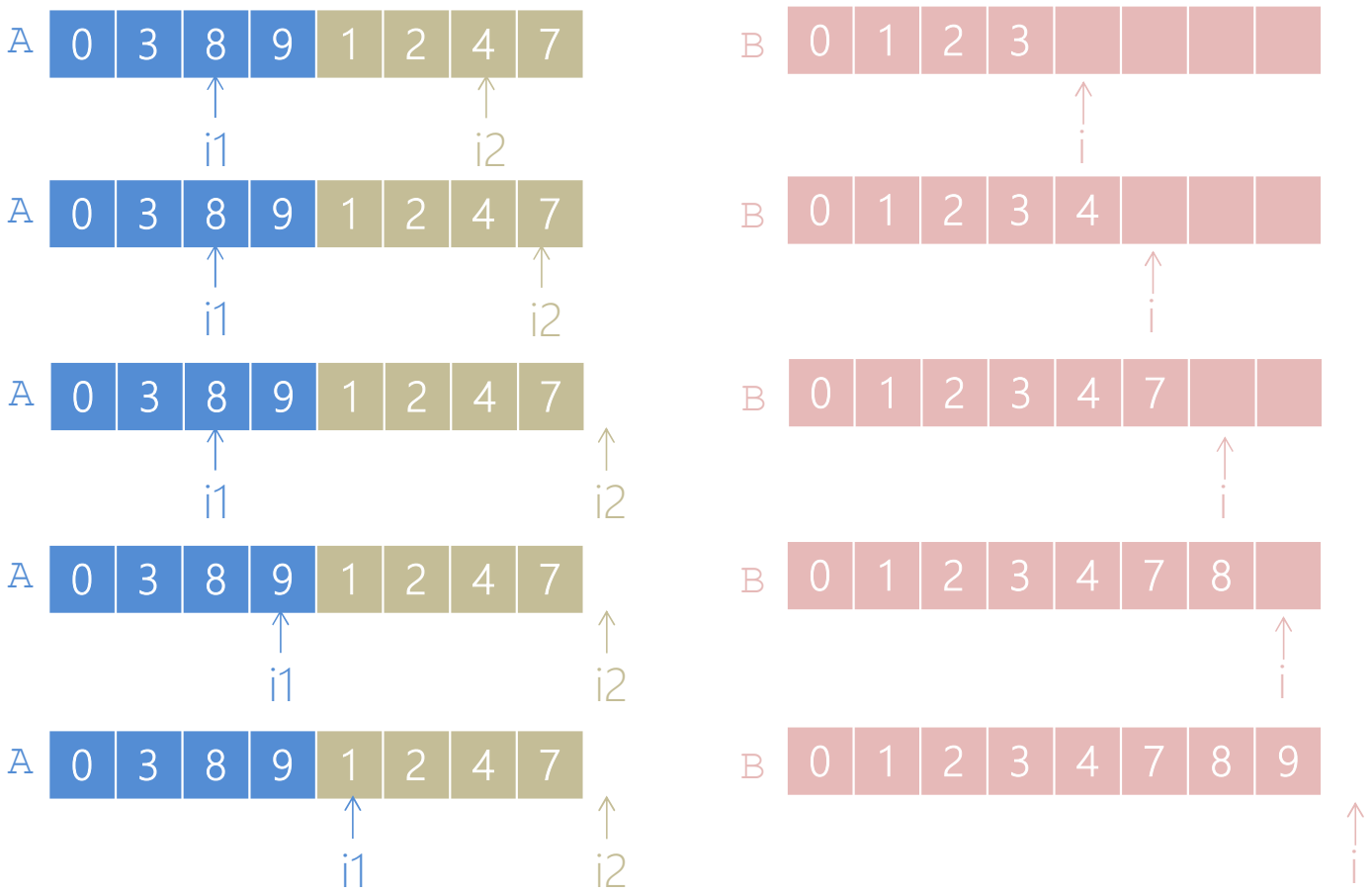
```
MERGE(A, B, lo, mid, hi)
    i1 = lo, i2 = mid+1, i=lo
    WHILE i1<=mid AND i2<=hi
        IF A[i1]<=A[i2]
            B[i++] = A[i1++]
        ELSE
            B[i++] = A[i2++]
    FOR i1 TO mid
        B[i++] = A[i1++]
    FOR i2 TO hi
        B[i++] = A[i2++]
    Copy B[lo..hi] back into A[lo..hi]
```

- Merge two sorted sub-arrays `A[lo..mid]` and `A[mid+1, hi]` into `A[lo..hi]`
- Use an extra array `B` with the same size as array `A`



Merge Example




A
0 3 8 9 1 2 4 7
i1
i2
A
0 3 8 9 1 2 4 7
i1
i2
A
0 3 8 9 1 2 4 7
i1
i2
A
0 3 8 9 1 2 4 7
i1
i2
A
0 3 8 9 1 2 4 7
i1
i2
B
0 1 2 3
i
B
0 1 2 3 4
i
B
0 1 2 3 4 7
i
B
0 1 2 3 4 7 8
i
B
0 1 2 3 4 7 8 9
i

Discussion on Merge

- Suppose that `A[lo..hi]` contains `n` elements
  - What is the worst-case running time?
  - What is the best-case running time?
  - What is the extra storage cost?

Analysis of Merge Sort

- Let `T(n)` denote the running time of MergeSort where `n` is the number of items to be sorted
- Assume that `n` is a power of 2.

Divide: $\Theta(1)$ time
Conquer: 2T(n/2) time
Combine: Θ(n) time

- Recurrence equation:

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

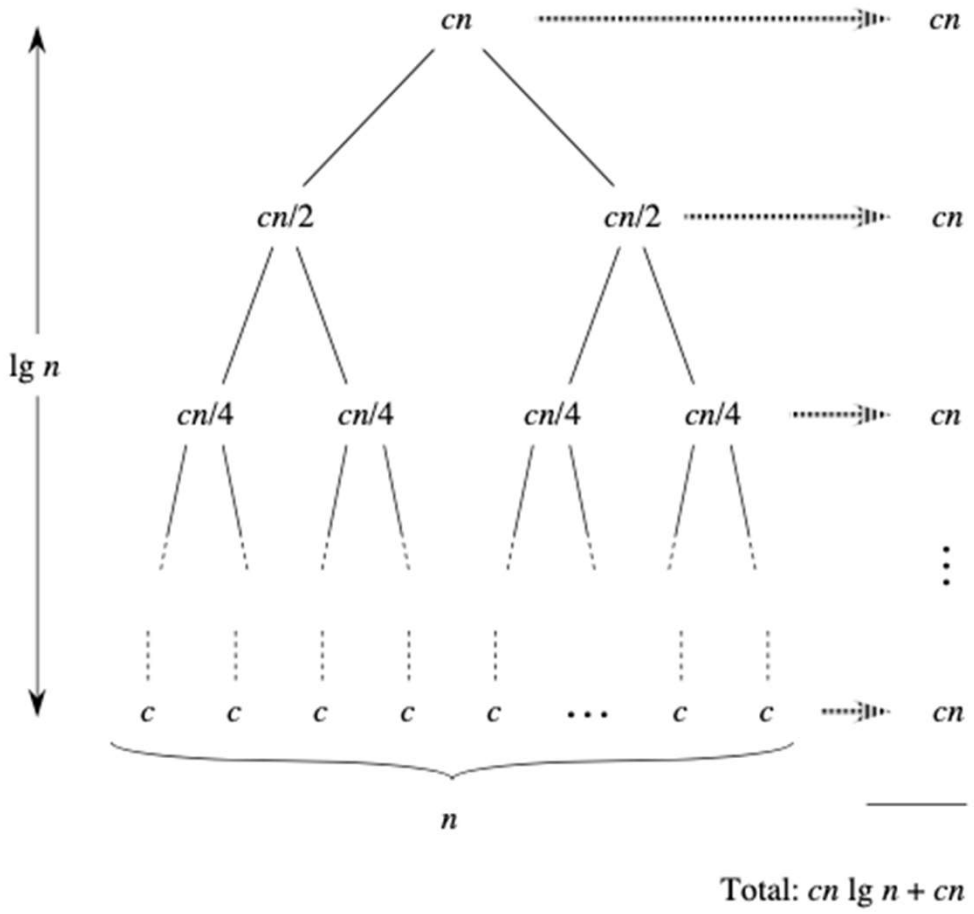
Analyzing Merge Sort

- Running time for merge-sort is $T(n) = \Theta(n \lg n)$ where $\lg n = \log_2 n$
- Rewrite recurrence as

$$T(n) = \begin{cases} c & if\ (n = 1), \\ 2T\left(\frac{n}{2}\right) + cn & if\ (n > 1). \end{cases}$$

- Let's Draw the recursion tree.

Analyzing Merge Sort



Each level has cost cn.

- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves ⇒ cost/level stays the same.

There are $\lg n + 1$ levels (height is $\lg n$).
Total cost is sum of costs at each level.
Total cost is $cn \lg n + cn \Rightarrow \Theta(n \lg n)$.

Analysis of Merge Sort

For Merge Sort:

$$T(n) = \Theta(n \lg n)$$

This is in fact the best, average, worst, and therefore overall running time.

Comments:

- We need an extra array B, so twice as much memory!

Contents of this presentation are partially adapted from
My CS385Q (Fall2022 Lectures)
and from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-3)
and are also based on
Book Chapter-2 & 3, Introduction to Algorithms by *Cormen, Leiserson, Rivest, & Stein*

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK **YOU**

Stevens Institute of Technology