香港科技大学（广州）
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Asymptotic Analysis

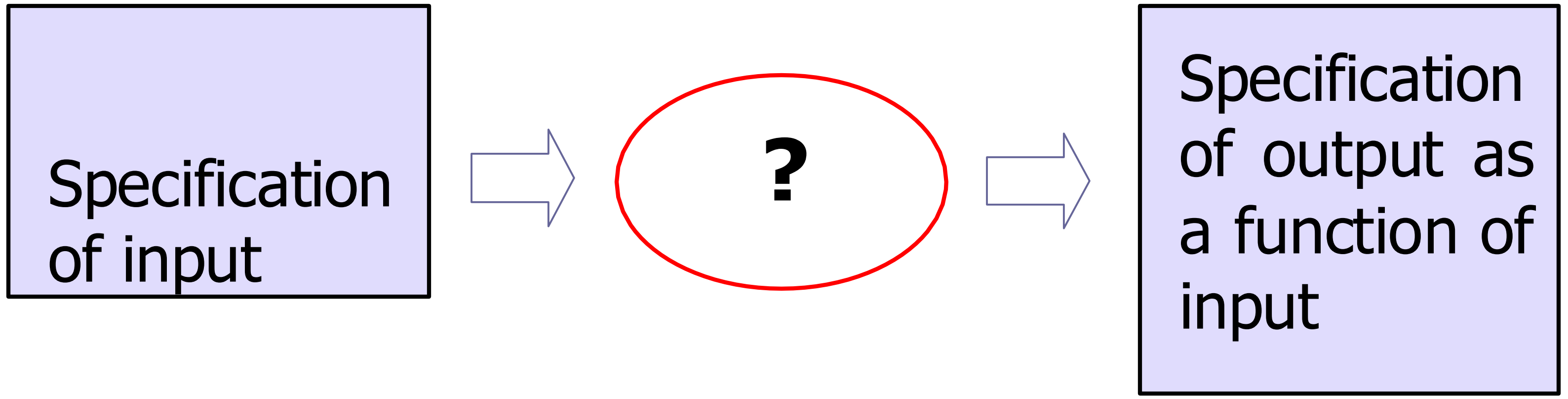
Data Structures and Algorithms

- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions

- Program: an implementation of an algorithm in some programming language

- Data structure: Organization of data needed to solve the problem

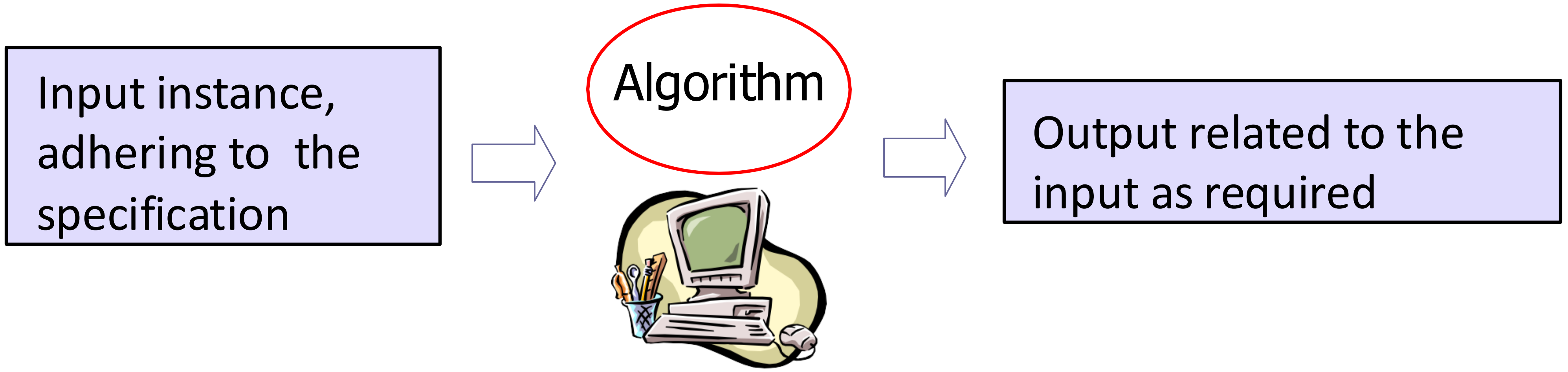
Algorithmic Problem



- Infinite number of input *instances* satisfying the specification.
  - E.g., a sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000
    - 3

Other boundary cases?

Algorithmic Solution


Input instance, adhering to the specification
Algorithm
Output related to the input as required

- Algorithm describes actions on the input instance
- Many correct algorithms for the same algorithmic problem

Open Question Points:0.5

设置

What is a good algorithm?

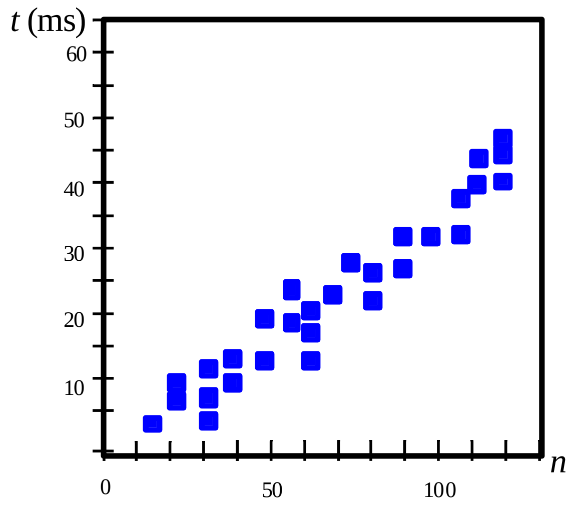
Answer

What is a Good Algorithm?

- Efficient:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - The number of bits in an input number
  - Number of data elements (numbers, points)

Measuring the Running Time

How should we measure the running time of an algorithm?



Experimental Study

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition
- Use a system call to get an accurate measure of the actual running time

Limitations of Experimental Studies

- Must implement and test the algorithm to determine its running time

- Experiments done only on a limited set of inputs
  - May not be indicative of the running time on other inputs not included in the experiment

- To compare two algorithms, the same hardware and software environments needed

Beyond Experimental Studies

We will develop a general methodology for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of testing one of its implementations
- Considers all possible inputs
- Evaluates the efficiency of any algorithm being independent of the hardware and software environment

Example

- **Algorithm** `arrayMax(A,n)`
- Input: An array A storing n integers
- Output: The maximum element in A

Pseudo-Code (Functional / Recursive)

```
algorithm arrayMax(A[0..n-1])
{
    A[0]                                      # if n=1
    max(arrayMax(A[0..n-2]),A[n-1])           # otherwise
}
```

Pseudo-Code (Imperative)

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm

- E.g., `algorithm arrayMax(A,n)`
  - Input: An array A storing n integers
  - Output: The maximum element in A

```
currentMax ← A[0]
   for i ← 1 to n-1 do
   if currentMax < A[i] then currentMax ← A[i]
return currentMax
```

Pseudo-Code

- It is more structured than usual prose but less formal than a programming language
- Expressions
  - use standard mathematical symbols to describe numeric and boolean expressions
  - use $\leftarrow$ for assignment (“=” in Python)
  - use = for equality relationship (“==” in Python)
- Method declarations
  - algorithm `name(param1,param2)`

Pseudo-Code

- Programming constructs
  - decision structures: **`if ... then ... [else ... ]`**
  - while-loops: **`while ... do`**
  - repeat-loops: **`repeat ... until ...`**
  - for-loop: **`for ... do`**
  - array indexing: **`A[i], A[i,j]`**
- Methods
  - calls: object `method(args)`
  - returns: **`return`** `value`

Analysis of Algorithms

- **Primitive Operation:** Low-level operation independent of programming language
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - Arithmetic and logical operations (e.g., addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

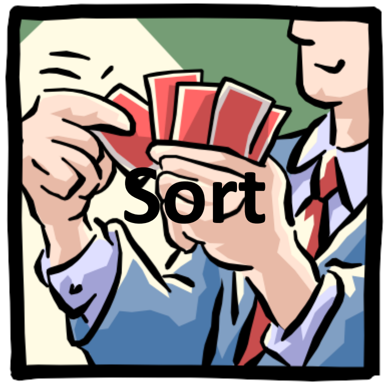
Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$ $\longrightarrow$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$\longrightarrow$ $b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

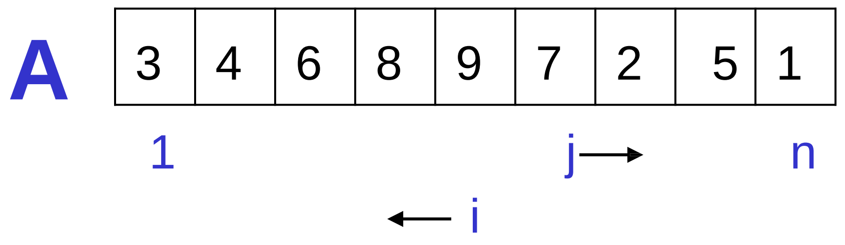
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sort



Strategy

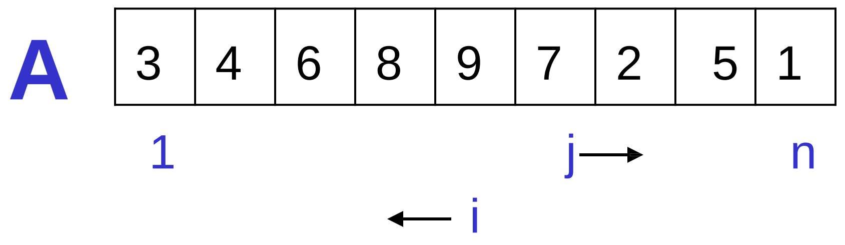
- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

INPUT: an array A[0..n-1] of integers
OUTPUT: a permutation of A such that A[0]≤ A[1]≤ …≤A[n-1]

Pseudo-Code (Functional/Recursive)

```
algorithm insertionSort(A[0..n-1])
{
  A[0]                                      # if n=1
  insert(insertionSort(A[0..n-2]), A[n-1])  # otherwise
}
algorithm insert(A[0..n-1], key)
{
  append(A[0..n-1], key)                    # if key>=A[n-1]
  append(newarray(key), A[0])               # if n=1&key<A[0]
  append(insert(A[0..n-2],key), A[n-1])     # otherwise
}
```

Insertion Sort



Strategy
- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

1. Try run it!
2. Understand why it is correct

INPUT: an array A[0..n-1] of integers
OUTPUT: a permutation of A such that A[0]≤ A[1]≤ …≤A[n-1]

```
for j ← 1 to n-1 do
    key ← A[j]
    # insert A[j] into the sorted
    sequence A[0..j-1]
    i ← j-1
    while i>=0 and A[i]>key do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

Analysis of Insertion Sort

```
for j ← 1 to n-1 do
 key ← A[j]
 # insert A[j] into the
 sorted sequence A[0..j-1]
 i ← j-1
 while i>=0 and A[i]>key do
  A[i+1] ← A[i]
  i--
 A[i+1] ←  key
```

cost	Times
c_1	n
c_2	$n-1$
0	$n-1$
c_3	$n-1$
c_4	$\sum_{j=1}^{n-1} t_j$
c_5	$\sum_{j=1}^{n-1} (t_j-1)$
c_6	$\sum_{j=1}^{n-1} (t_j-1)$
c_7	$n-1$

Total time = $n(c_1+c_2+c_3+c_7)+\sum_{j=1}^{n-1} t_j(c_4+c_5+c_6)-(c_2+c_3+c_5+c_6+c_7)$

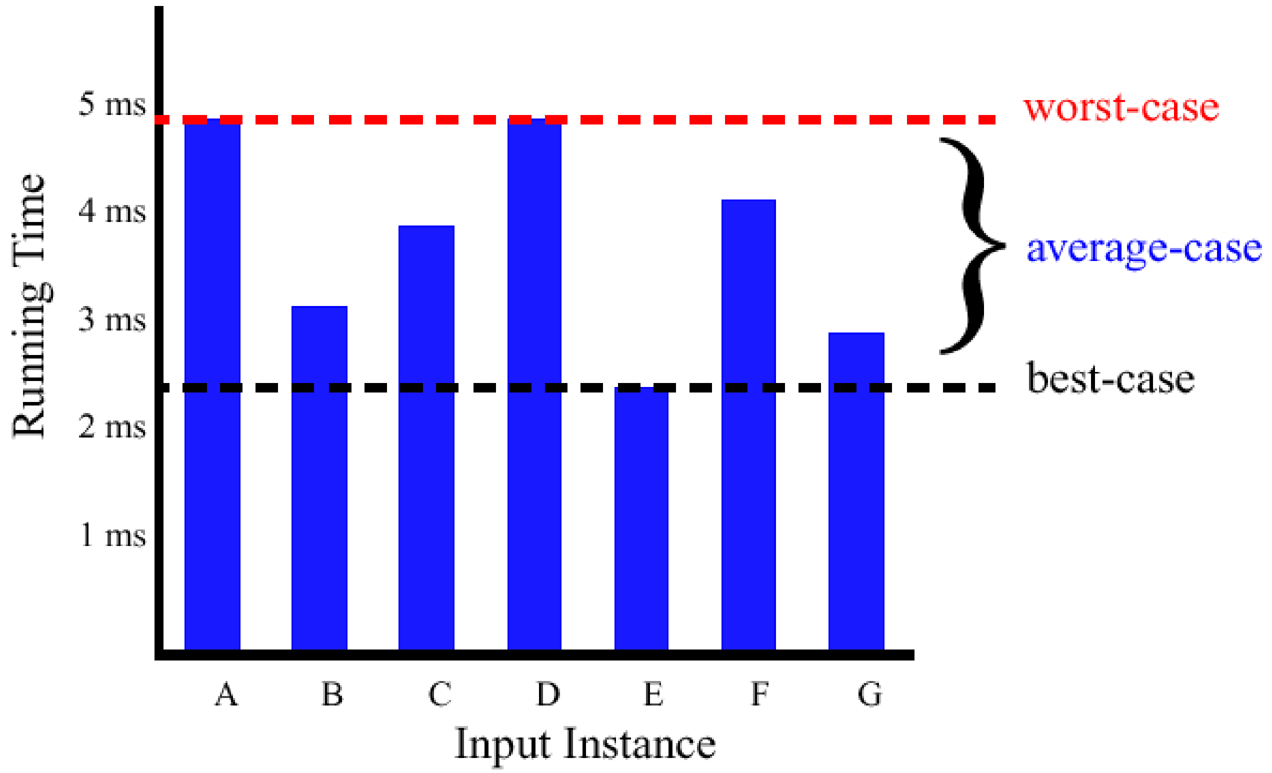
Best/Worst/Average Case

Total time = $n(c_1 + c_2 + c_3 + c_7) + \sum_{j=1}^{n-1} t_j(c_4 + c_5 + c_6) - (c_2 + c_3 + c_5 + c_6 + c_7)$

- **Best case**:
  - elements already sorted; t_j=1, running time = f(n), i.e., *linear* time
- **Worst case**:
  - elements are sorted in inverse order; t_j=j+1, running time = f(n^2), i.e., *quadratic* time
- **Average case**:
  - t_j=(j+1)/2, running time = f(n^2), i.e., *quadratic* time

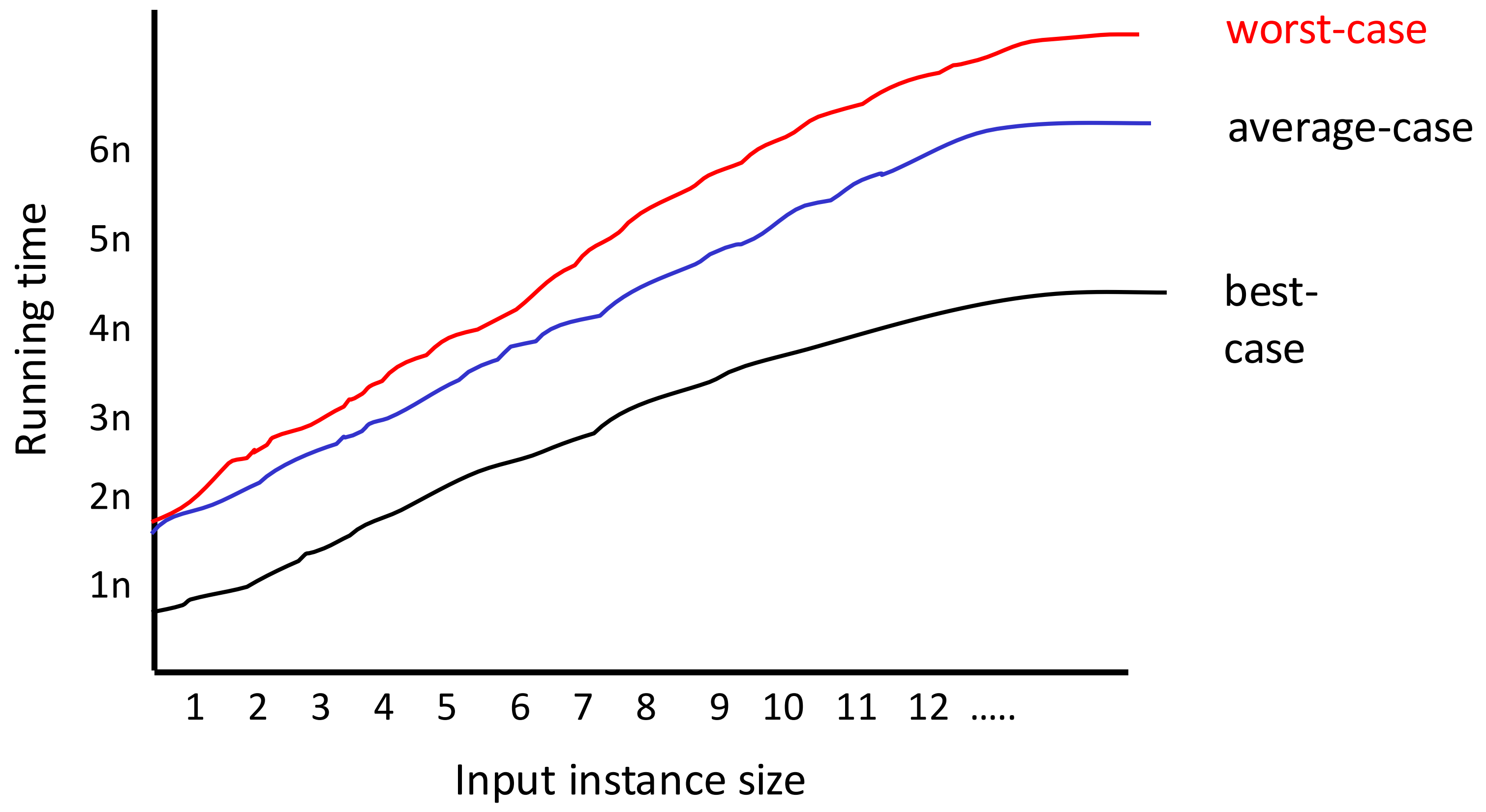
Best/Worst/Average Case (2)

- For a specific size of input n, investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Best/Worst/Average Case (4)

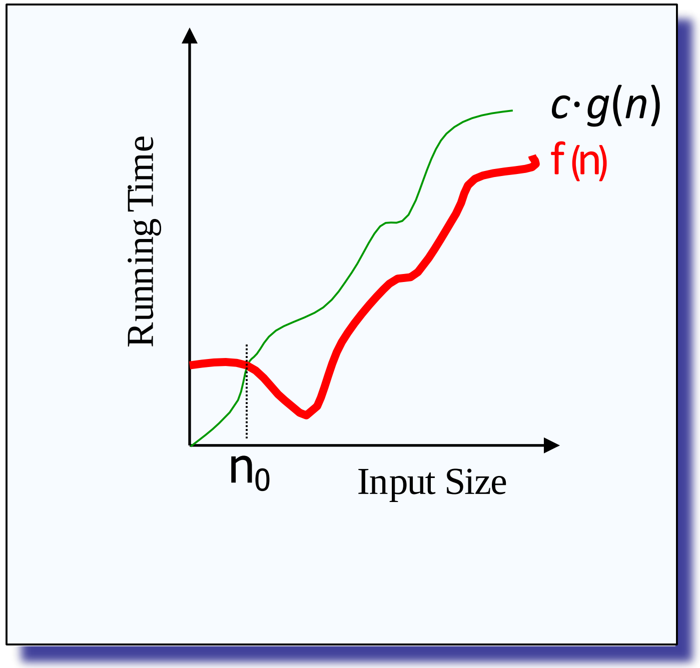
- **Worst case** is usually used: It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- **Average case** is often as bad as **worst case**
- Finding **average case** can be very difficult

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of “details”, which may be affected by specific implementation and hardware
  - like “rounding”: 1,000,001 ≈ 1,000,000
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input in the limit
  - Asymptotically more efficient algorithms are best for all but small inputs

Asymptotic Notation

- The “big-Oh” *O*-Notation
  - asymptotic upper bound
  - f(n) is O(g(n)), if there exists constants *c* and n_0, *s.t.* **f(n) ≤ c·g(n)** for all n ≥ n_0
  - f(n) and g(n) are functions over non-negative integers
    - We usually assume both f(n) and g(n) are non-negative too
- Used for *worst-case* analysis

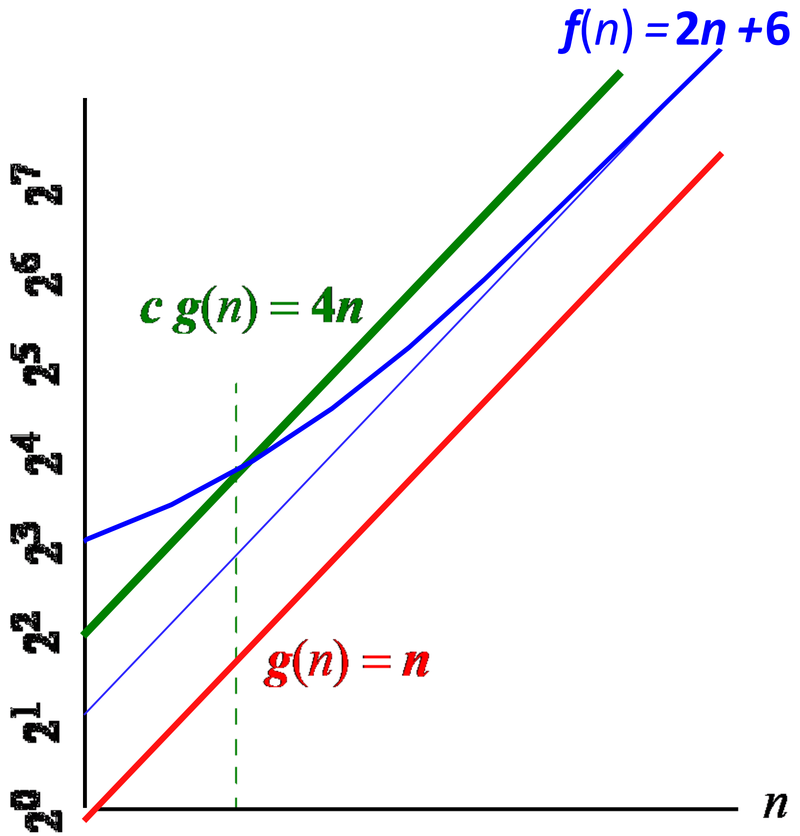


Example

For functions f(n) and g(n) there are positive constants c and n_0 such that: $f(n) \le c\, g(n)$ for $n \ge n_0$

conclusion:

2n+6 is O(n)



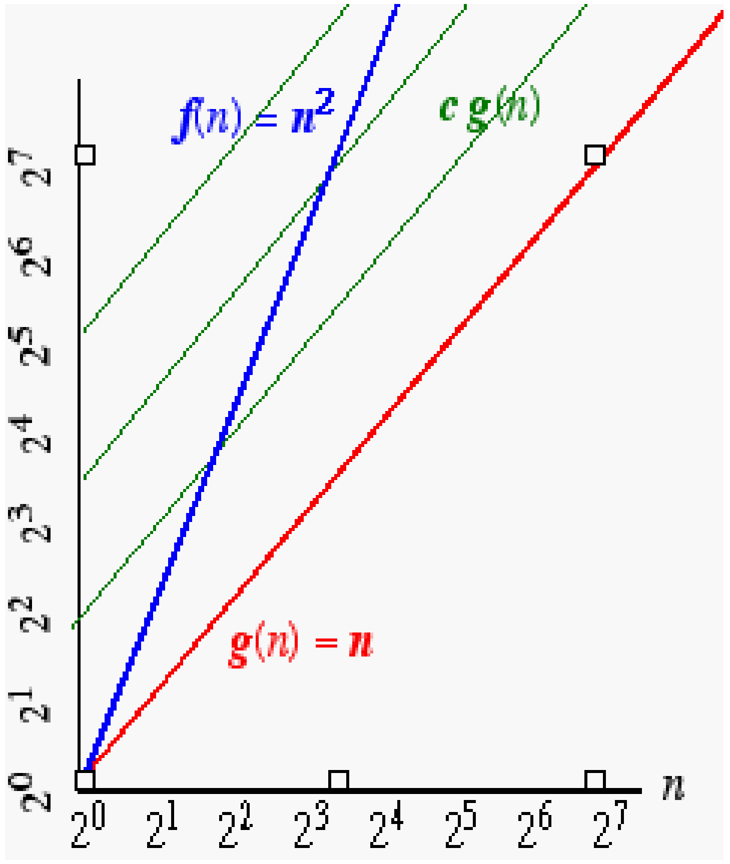
Another Example

On the other hand…

n^2 is not O(**n**) because there is no **c** and **n_0** such that:

$$n^2 \le cn \text{ for } n \ge n_0$$

The graph to the right illustrates that no matter how large a c is chosen there is an n big enough that **$n^2 > cn$**



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors
  - $50\ n \log n$ is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Note: Even though $(50\ n \log n)$ is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

Asymptotic Analysis of Running Time

- Use *O*-notation to express number of primitive operations executed as function of input size
- Comparing asymptotic running times
  - an algorithm that runs in *O*(n) time is better than one that runs in *O*(n^2) time
  - similarly, *O*(log n) is better than *O*(n)
  - hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still *O*(n) but might be less efficient than one running in time $2n^2$, which is *O*(n^2)

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]        ← 1 step
    A[i] ← a/(i+1)
return array A
```

i iterations with i=0,1,2...n-1

n iterations

Analysis: running time is $O(n^2)$

A Better Algorithm

```
**Algorithm** prefixAverages2(X):
Input: An $n$-element array X of numbers.
Output: An $n$-element array A of numbers such that A[$i$] is the average of elements X[0], ... , X[$i$].
s ← 0
**for** i ← 0 **to** n **do**
    s ← s + X[i] A[i] ← s/(i+1)
**return** array A
```

Analysis: Running time is O(n)

Asymptotic Notation *(terminology)*

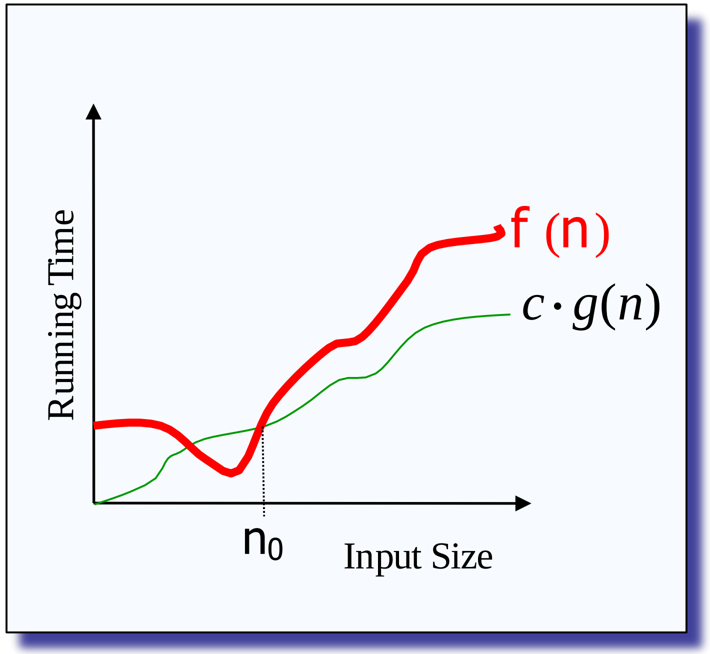
- Special classes of algorithms:
  - Logarithmic: $O(\log n)$
  - Linear: $O(n)$
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k)$, $k \geq 1$
  - Exponential: $O(a^n)$, $a > 1$
- “Relatives” of the Big-Oh
  - $\Omega(f(n))$: Big Omega -asymptotic lower bound
  - $\Theta(f(n))$: Big Theta -asymptotic tight bound

Asymptotic Notation

- The “big-Omega” Ω–Notation
  - asymptotic lower bound
  - *f(n)* is Ω*(g(n))* if there exists

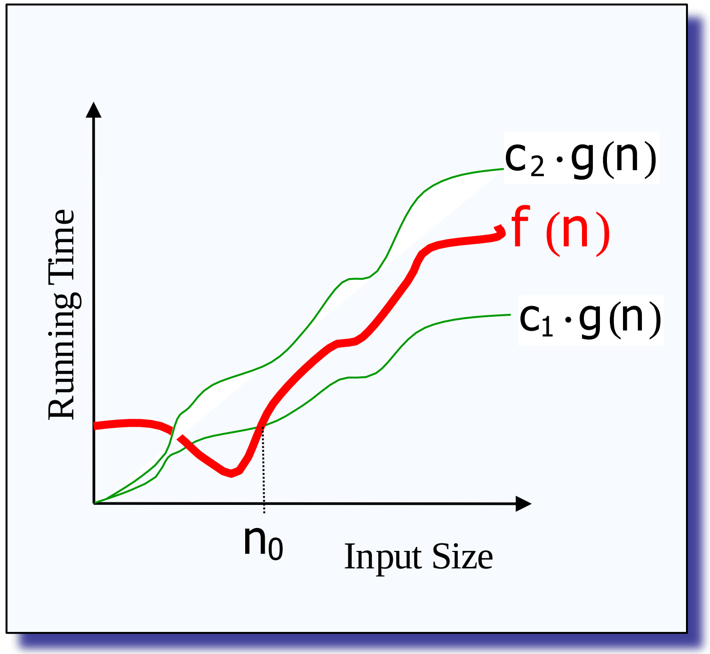
    constants *c* and n_0, *s.t.*

    c g(n) ≤f(n) for $n \geq n_0$
- Used to describe *best- case* running times or lower bounds for algorithmic problems
  - E.g., lower-bound for searching in an unsorted array is Ω*(n).*



Asymptotic Notation

- The "big-Theta" Θ–Notation
  - asymptotically tight bound
  - f(n) is Θ(g(n)) if there exists constants c_1, c_2, and n_0, s.t. **c_1 g(n) ≤f(n) ≤c_2 g(n)** for $n \geq n_0$



- f(n) is Θ(g(n)) if and only if f(n) is O(g(n)) and f(n) is Ω(g(n))
- O(f(n)) is often misused instead of Θ(f(n))

Asymptotic Notation

Two more asymptotic notations

- "Little-Oh" notation *f(n)* is *o(g(n))*
  non-tight analogue of Big-Oh
  - For every *c>0,* there should exist n_0, s.t.
    f(n) ≤c g(n) for $n \geq n_0$
  - Used for **comparisons** of running times.
    If *f(n) is o(g(n)),* it is said that *g(n) dominates f(n).*
- "Little-omega" notation *f(n)* is *ω(g(n))*
  non-tight analogue of Big-Omega

Asymptotic Notation

- Analogy with real numbers
  - $f(n)$ is $O(g(n))$ ≅ $f \leq g$
  - $f(n)$ is $\Omega(g(n))$ ≅ $f \geq g$
  - $f(n)$ is $\Theta(g(n))$ ≅ $f = g$
  - $f(n)$ is $o(g(n))$ ≅ $f < g$
  - $f(n)$ is $\omega(g(n))$ ≅ $f > g$
- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

Comparison of Running Times

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
400*n*	2500	150000	9000000
20*n* log *n*	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Sorting

The problem of sorting

- Input:
  - Sequence $< a_1, a_2, \ldots, a_n >$ of numbers.
- Output
  - Permutation $< a'_1, a'_2, \ldots, a'_n >$ such that

  a'1 ⍰ a'2 ⍰ … ⍰ a'n
- Example
  - Input: 8 2 4 9 3 6
  - Output: 2 3 4 6 8 9

Bubble Sort

- Bubble Sort

"Bubbling Up" the Largest Element

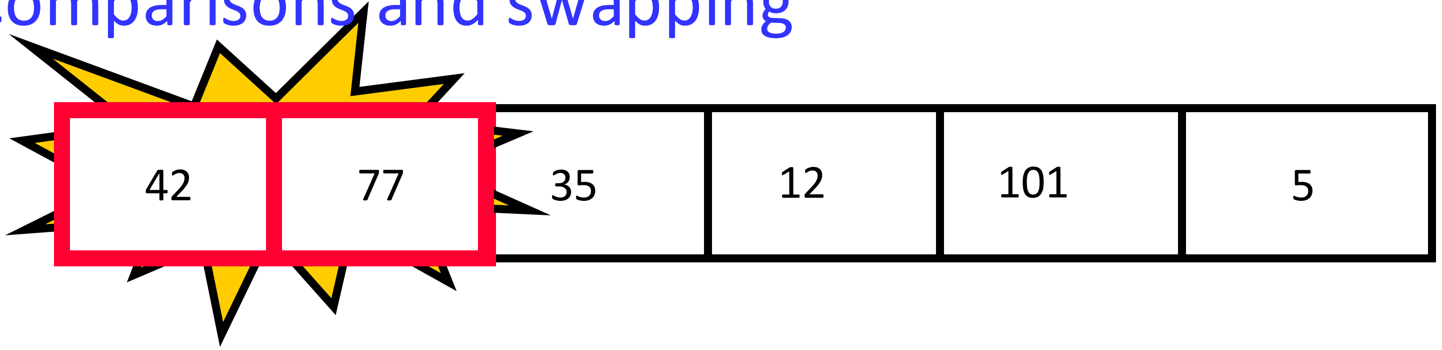
- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping

77	42	35	12	101	5

"Bubbling Up" the Largest Element

Traverse a collection of elements

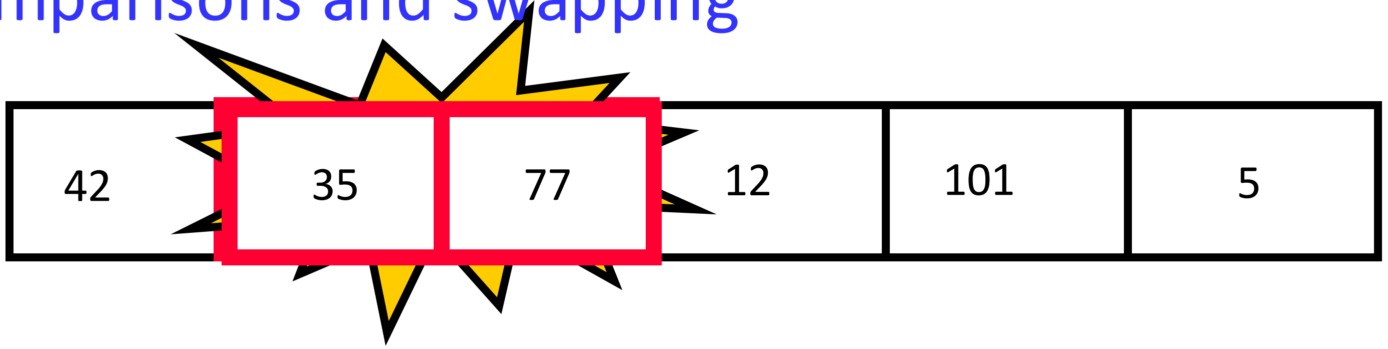
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

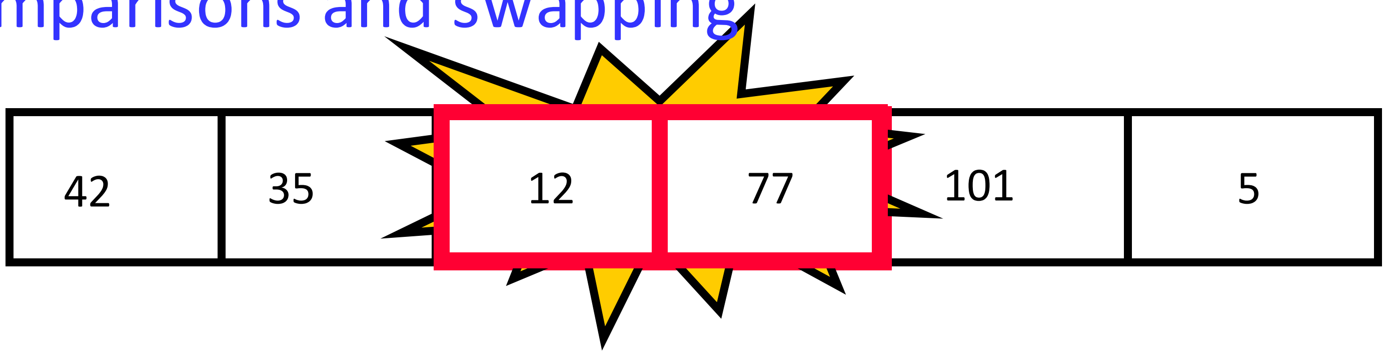
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

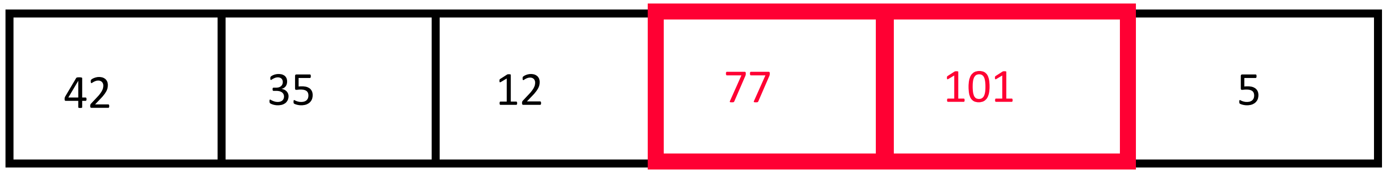
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping

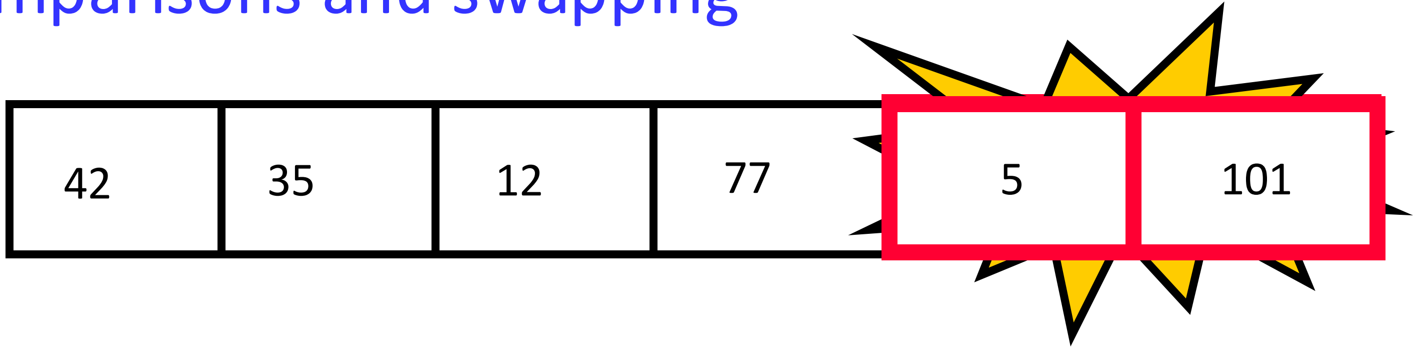


No need to swap

"Bubbling Up" the Largest Element

Traverse a collection of elements

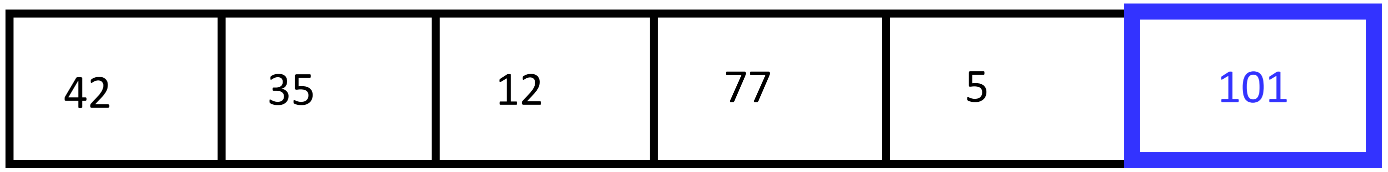
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



Largest value correctly placed

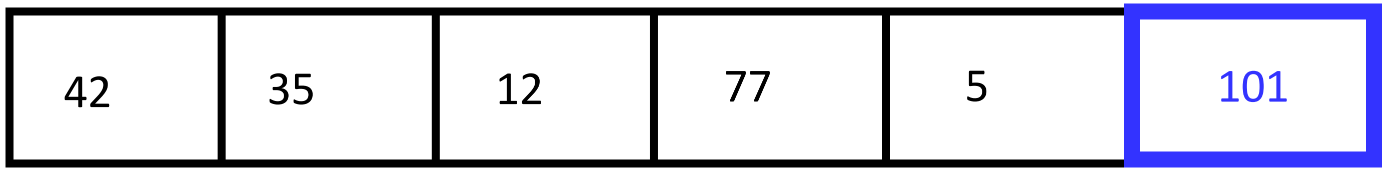
The “Bubble Up” Algorithm

```
index <- 1
last_compare_at <- n – 1

loop
  exitif(index > last_compare_at)
  if(A[index] > A[index + 1]) then
    Swap(A[index], A[index + 1])
  endif
  index <- index + 1
endloop
```

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

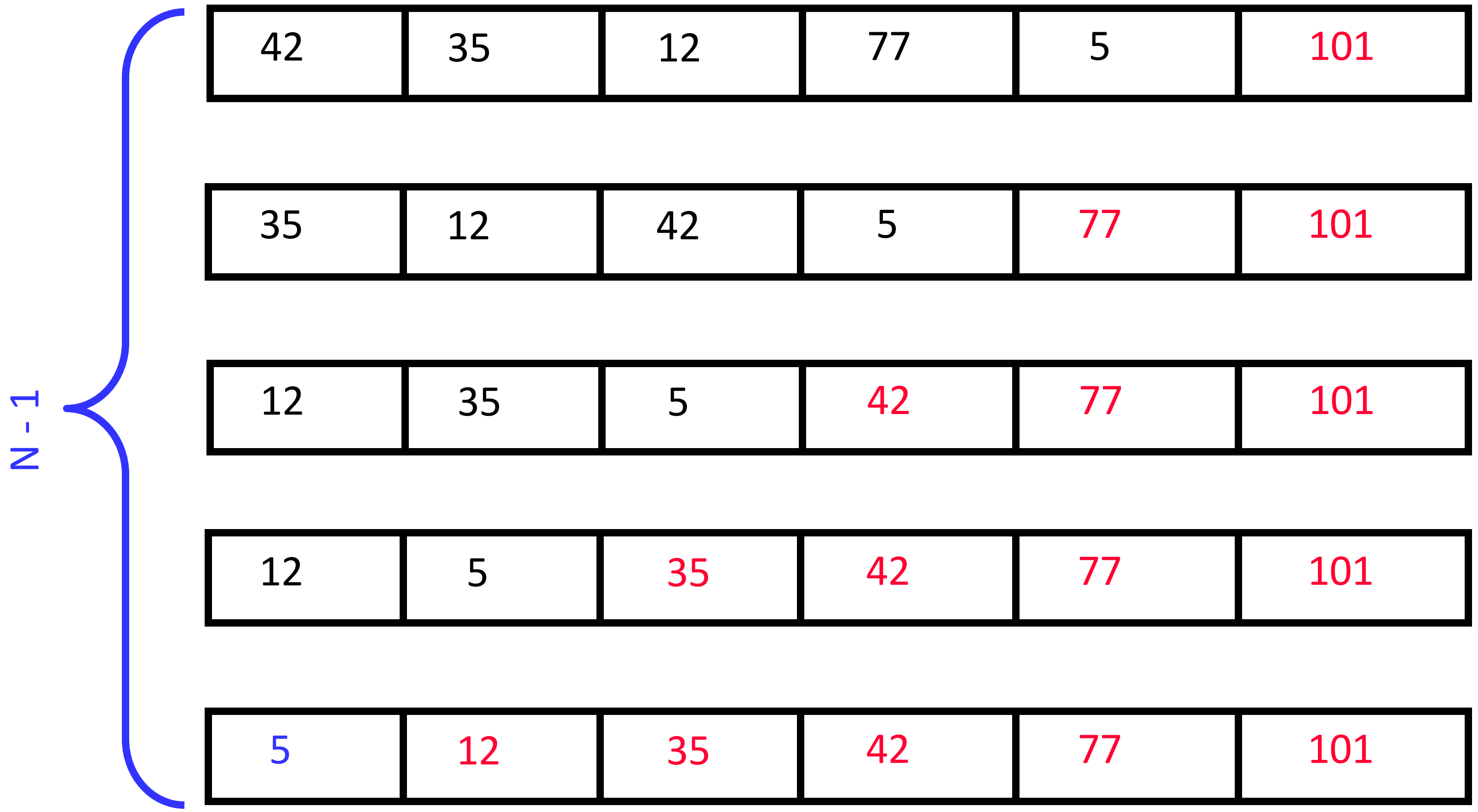


Largest value correctly placed

Repeat “Bubble Up” How Many Times?

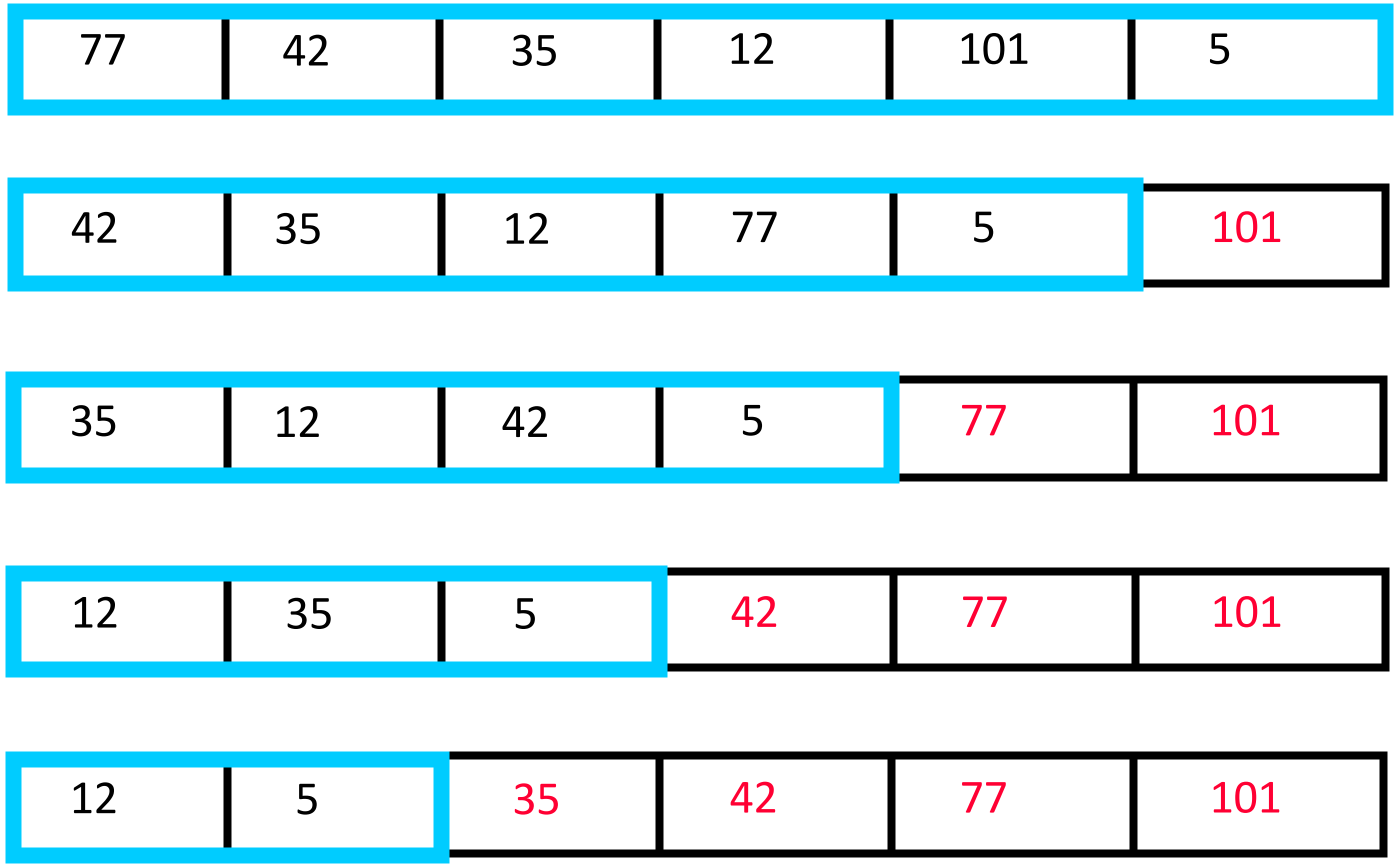
- If we have N elements…
- And if each time we bubble an element, we place it in its correct location…
- Then we repeat the “bubble up” process N – 1 times.
- This guarantees we’ll correctly place all N elements.

“Bubbling” All the Elements



42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101

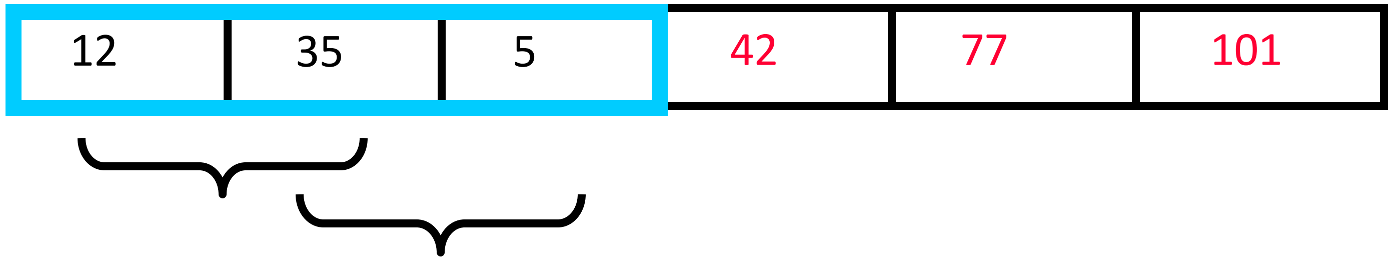
Reducing the Number of Comparisons



Reducing the Number of Comparisons

- On the Nth "bubble up", we only need to do MAX-N comparisons.

- For example:
- This is the 4th "bubble up"
- MAX is 6
- Thus we have 2 comparisons to do

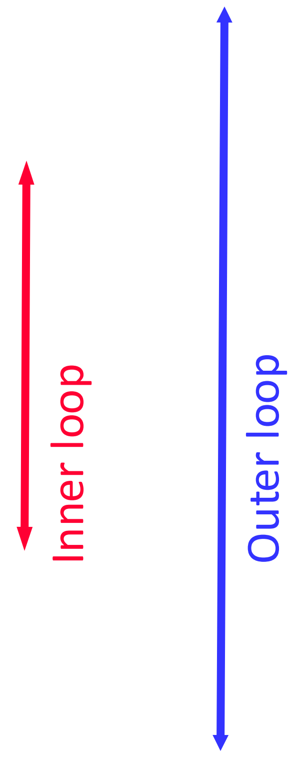


Putting It All Together

- Putting It All Together

```
procedure Bubblesort(A)
  to_do, index isoftype Num
  to_do <- N - 1

  loop
    exitif(to_do = 0)
    index <- 1
    loop
      exitif(index > to_do)
      if(A[index] > A[index + 1]) then
        Swap(A[index], A[index + 1])
      endif
      index <- index + 1
    endloop
    to_do <- to_do - 1
  endloop
endprocedure // Bubblesort
```



Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of “bubble ups,” the collection was sorted?
- We want to be able to detect this and “stop early”!

5	12	35	42	77	101

Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”
- If no swapping occurred, then we know that the collection is already sorted!
- This boolean “flag” needs to be reset after each “bubble up.”

```
did_swap: Boolean
did_swap <- true

loop
  exitif ((to_do = 0) OR NOT(did_swap))
  index <- 1
  did_swap <- false
  loop
    exitif(index > to_do)
    if(A[index] > A[index + 1]) then
      Swap(A[index], A[index + 1])
      did_swap <- true
    endif
    index <- index + 1
  endloop
  to_do <- to_do - 1
endloop
```

Running Time

- The running time depends on the input: an already sorted sequence is easier to sort.

- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of Analyses

- **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 of statistical distribution of inputs.
- **Best-case:** (bogus)
  - Cheat with a slow algorithm that works fast on *some* input.

Machine-Independent Time

What is bubble sort's worst-case time?

- *it* depends on the speed of our computer:
- relative speed (on the same machine),
- absolute speed (on different machines).

BIG IDEA:

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

Θ-notation

Math:

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ $\}$

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Other Notations

$\text{O}(g(n)) = \{\ f(n)$: there exist positive constants c_2, and n_0 such that $0 \le f(n) \le c_2\, g(n)$ for all $n \ge n_0$ $\}$

$\wedge(g(n)) = \{\ f(n)$: there exist positive constants c_1, and n_0 such that $0 \le c_2\, g(n) \le f(n)$ for all $n \ge n_0$ $\}$

$\text{o}(g(n)) = \{\ f(n)$: for any $\varepsilon \ge 0$, there exist n_0 such that $0 \le g(n) \le \varepsilon\, f(n)$ for all $n \ge n_0$ $\}$

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

Example

Sort the functions in increasing order of asymptotic (big-O) complexity:

$$\begin{aligned}
f_1(n) &= n^{0.999999} \log n \\
f_2(n) &= 10000000n \\
f_3(n) &= 1.000001^n \\
f_4(n) &= n^2
\end{aligned}$$

Example

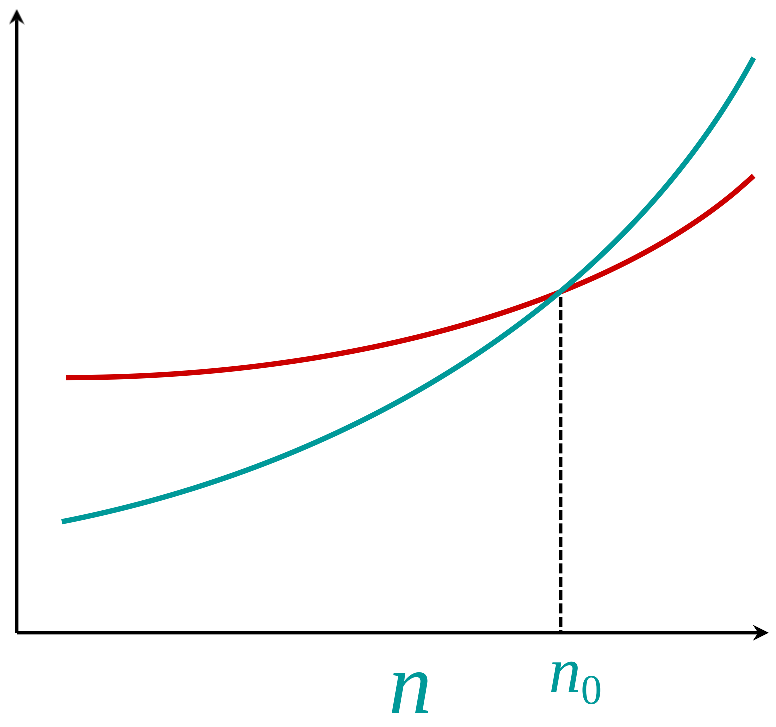
Solution: The correct order of these functions is $f_1(n), f_2(n), f_4(n), f_3(n)$. To see why $f_1(n)$ grows asymptotically slower than $f_2(n)$, recall that for any $c > 0$, $\log n$ is $O(n^c)$. Therefore we have:

$$f_1(n) = n^{0.999999} \log n = O(n^{0.999999} \cdot n^{0.000001}) = O(n) = O(f_2(n))$$

The function $f_2(n)$ is linear, while the function $f_4(n)$ is quadratic, so $f_2(n)$ is $O(f_4(n))$. Finally, we know that $f_3(n)$ is exponential, which grows much faster than quadratic, so $f_4(n)$ is $O(f_3(n))$.

Asymptotic Performance

When *n* gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Insertion Sort Analysis

Worst case: Input reverse sorted.

$$T(n) = \sum^{n} \Theta(j) = \Theta(n^2) \qquad \text{[arithmetic series]}$$

Properties of Bubble Sort

- Bubble n sort is a stable sorting algorithm.
- Bubble sort is an in-place sorting algorithm.
- Number of swaps in bubble sort = Number of inversion pairs present in the given array.
- Bubble sort is beneficial when array elements are less and the array is nearly sorted.

References

- Big-O Cheat Sheet: https://www.bigocheatsheet.com

The End