DTS103TC
Design and Analysis of Algorithms

Lecture 1: Complexity Analysis

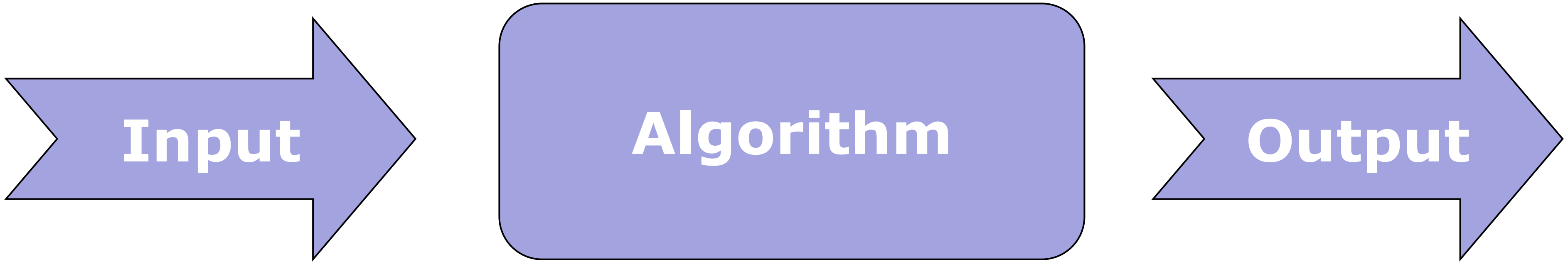
Dr. Pascal Lefevre
School of AI and Advanced Computing

Learning outcomes

- Algorithm definition
- Examples of algorithmic problems
- Insertion sort
- Analysis of algorithms
- Mathematical Induction
- Worst-case and average-case time complexity
- Space complexity
- Understand asymptotic complexity and notation
- Carry out simple asymptotic analysis of algorithms

What is an algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output.



- We can also view an algorithm as a tool for solving a well-specified computational problem.
- Daily life examples: cooking recipe

Algorithm vs. Program

- Algorithm
  - Design
  - Domain Knowledge
  - Any language
  - Hardware & OS
  - Analyze
- Program
  - Implementation
  - Programmer
  - Programming Language
  - Hardware & OS
  - Testing

Some Well-known Algorithms

- Sorting
  - Insertion sort
  - Merge sort
- Searching
- Graph algorithms
  - Minimum Spanning Trees
  - Shortest Path
- String matching
  - The Rabin-Karp algorithm
  - The Knuth-Morris-Pratt algorithm
- Number-Theoretic Algorithms
  - The RSA public-key cryptosystem

Sorting

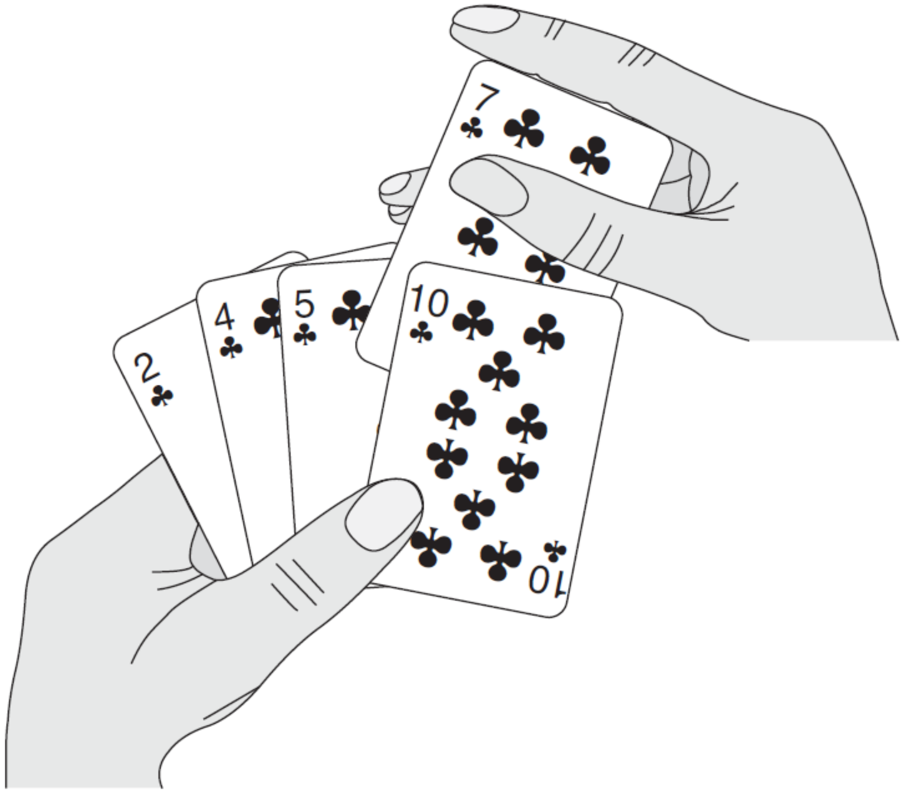
- Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

- Output: A reordering $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

- Example:
  - Input: <8,2,4,9,3,6>
  - Output: <2,3,4,6,8,9>

Insertion sort

Sorting a hand of cards using insertion sort



Insertion sort – cont’d

8 2 4 9 3 6

Insertion sort – cont'd

8 2 4 9 3 6

Insertion sort – cont’d

8 2 4 9 3 6

2 8 4 9 3 6

Insertion sort – cont’d

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Insertion sort – cont’d

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

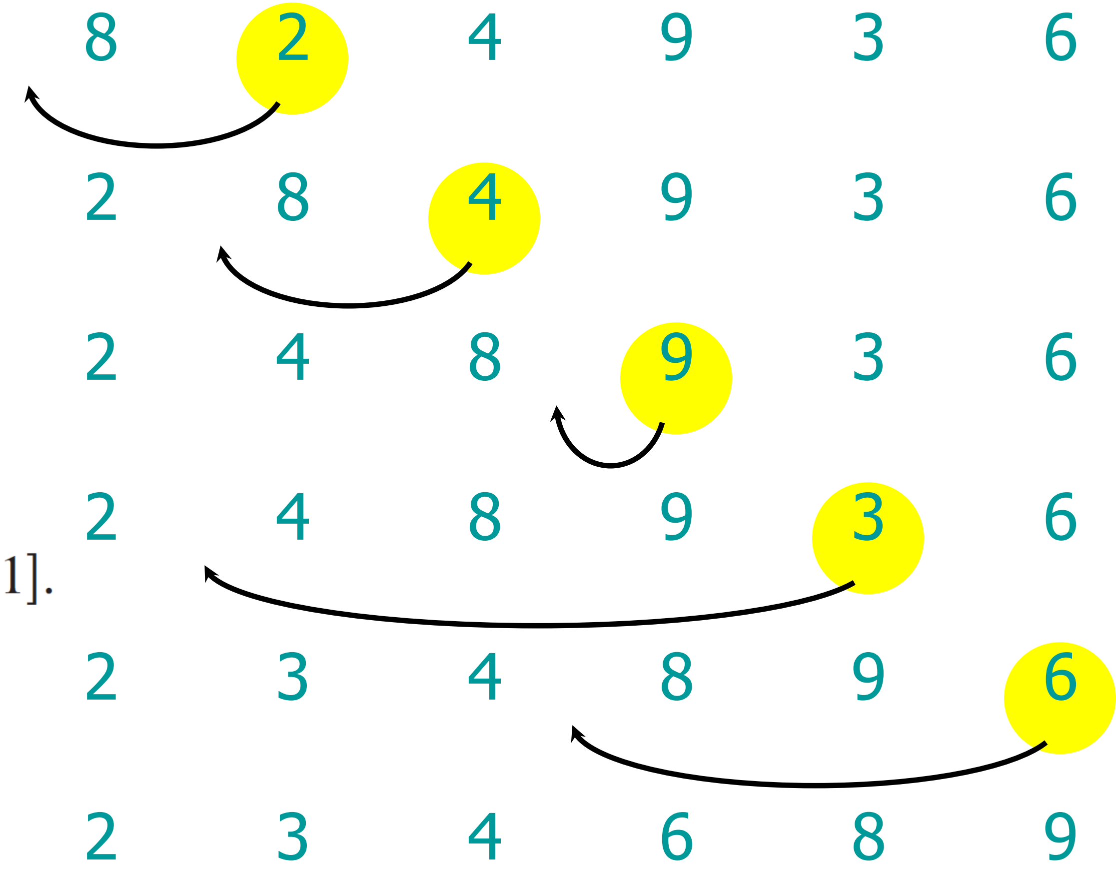
Insertion sort – cont'd

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Insertion sort – cont’d

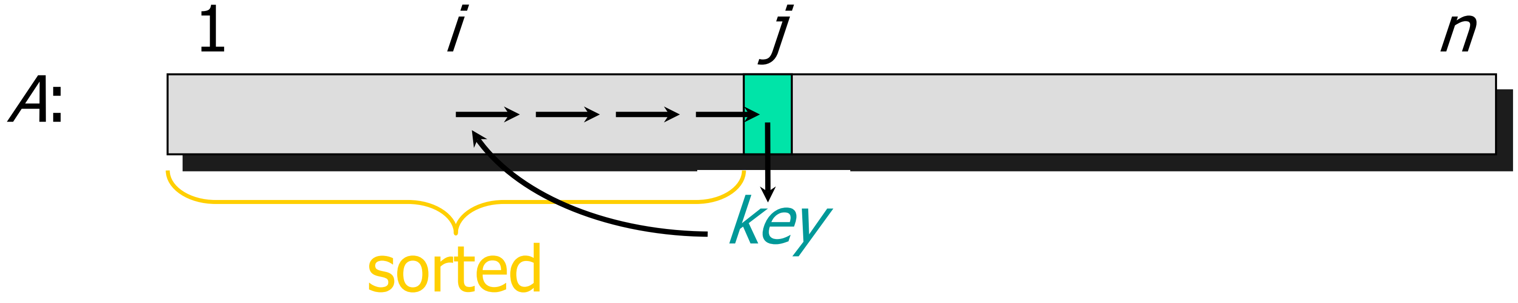
8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Insertion sort – cont’d



Pseudocode

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



Learning outcomes

- Algorithm definition
- Examples of algorithmic problems
- Insertion sort
- Analysis of algorithms
- Mathematical Induction
- Worst-case and average-case time complexity
- Space complexity
- Understand asymptotic complexity and notation
- Carry out simple asymptotic analysis of algorithms

Analysis of Algorithms

- Proof of correctness: show that the algorithm gives the desired result
- Time complexity analysis: find out how fast the algorithm runs
- Space complexity analysis: decide how much memory space the algorithm requires
- Look for improvement: can we improve it to run faster or use less memory?

Mathematical Induction

- Mathematical induction: a mathematical technique to prove that a statement holds for every natural number n=0,1,2,….

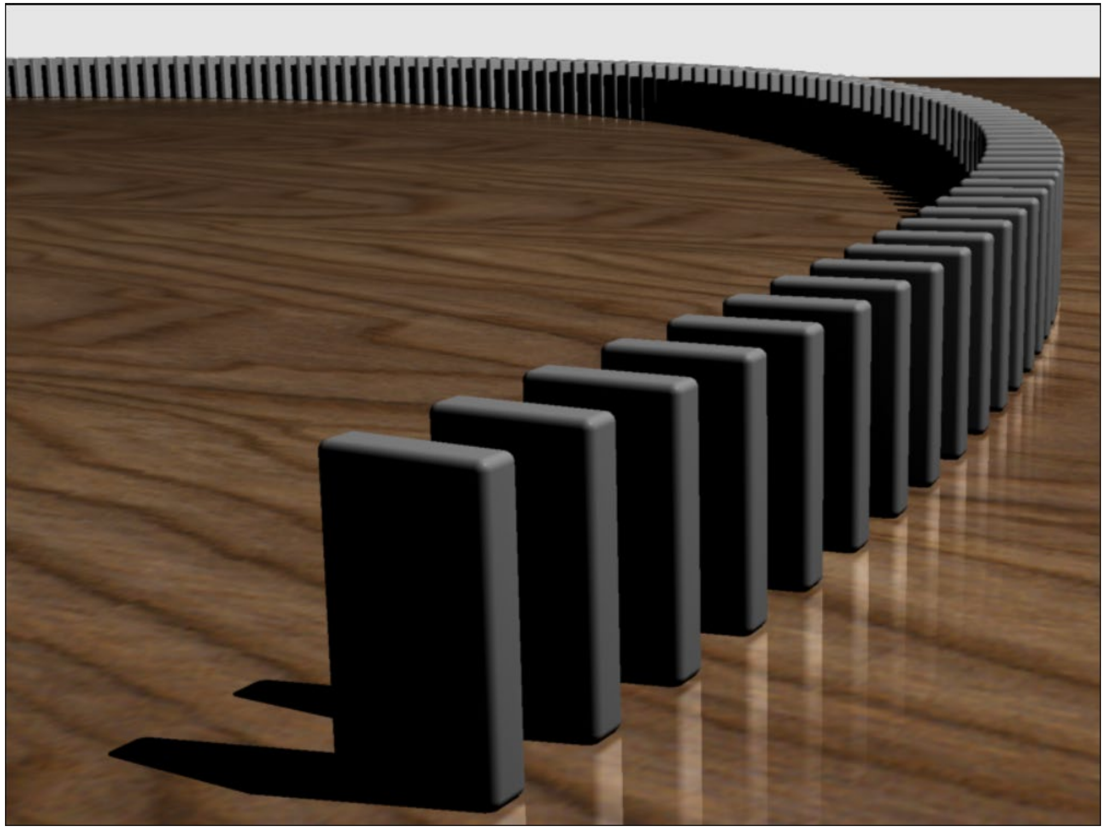
- For example, to prove $1+2+\ldots+n = n(n+1)/2$ $\forall$ integers $n \geq 1$
  - when n is 1, L.H.S = 1, R.H.S = 1*2/2 = 1 **OK!**
  - when n is 2, L.H.S = 1+2 = 3, R.H.S = 2*3/2 = 3 **OK!**
  - when n is 3, L.H.S = 1+2+3 = 6, R.H.S = 3*4/2 = 6 **OK!**

However, none of these constitute a proof and we cannot enumerate over all possible numbers.

=> Mathematical Induction

Mathematical induction – cont'd

- Mathematical induction can be informally illustrated by reference to the sequential effect of falling dominoes.
- If the first domino falls, then the second domino falls. If the second domino falls, then the third domino will fall too. And so on.
- Conclusion: If the first domino falls, then any n, nth domino falls.



Mathematical Induction Examples

- To prove: $1+2+\ldots+n = n(n+1)/2$ $\forall$ integers $n \geq 1$
- Base case: When $n=1$, L.H.S = 1, R.H.S = $1*2/2=1$. Therefore, the statement is true for $n=1$.
- Induction hypothesis: Assume that statement is true when $n=k$ for some integer $k \geq 1$.
  - i.e., assume that $1+2+\ldots+k = k(k+1)/2$
- Induction step: When $n=k+1$,
  - L.H.S = $1+2+\ldots+k+(k+1) = (k^2+3k+2)/2$
  - R.H.S = $(k+1)((k+1)+1)/2 = (k^2+3k+2)/2$ = L.H.S

Mathematical Induction Examples – Cont'd

- We have proved
  - statement true for n=1
  - If statement is true for n=k, then also true for n=k+1
- In other words,
  - true for n=1 implies true for n=2 (induction step)
  - true for n=2 implies true for n=3 (induction step)
  - true for n=3 implies true for n=4 (induction step)
  - and so on ......
- Conclusion: true for all integers n

Question

n! = n(n-1)(n-2) … *2*1

Use Mathematical Induction to prove $2^n < n!$ $\forall$ integers $n \geq 4$.

Mathematical Induction Examples – Cont'd

- To prove 2^n< n! ∀ integers n ≥ 4.
- Base case: When n=4, L.H.S = 16, R.H.S = 4! = 4*3*2*1 = 24, L.H.S < R.H.S. So, statement true for n=4
- Induction hypothesis: Assume that statement is true for some integer k ≥ 4, i.e., assume 2^k < k!
- Induction step: When n=k+1
  - L.H.S = 2^{k+1} = 2*2k < 2*k! <- by hypothesis
  - R.H.S = (k+1)! = (k+1)*k! > 2*k! > L.H.S <-because k+1>2
  - So, statement true for k+1
- Conclusion: statement true ∀ integers n ≥ 4.

Loop invariants and the correctness of insertion sort

- Back to Insertion Sort, We use loop invariants (Similar to Mathematical Induction) to help us understand why an algorithm is correct.

- 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.

Loop invariants and the correctness of insertion sort – cont'd

Initialization: When j = 2, the subarray A[1..j-1] consists of just the single element A[1], which shows that the loop invariant holds prior to the first iteration of the loop.

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

Loop invariants and the correctness of insertion sort – cont'd

Maintenance: The body of the for loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right until it finds the proper position for A[j] (lines 4–7), at which point it inserts the value of A[j] (line 8). The subarray A[1…j] then consists of the elements in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

```
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
```

Loop invariants and the correctness of insertion sort – cont'd

Termination: The condition causing the for loop to terminate is that =j =n+1. We have the entire array A[1…n] consists of the elements in sorted order. Hence, the algorithm is correct.

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

Complexity Measures

- Why we need to analyze algorithm complexity?
  - Computing time is a bounded resource, and so is space in memory.

- What we need to analyze?
  - Analyzing an algorithm has come to mean predicting the resources that the algorithm requires (computational time, space, power consumption, number of exchanged messages, and so on).

Running Time

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

- It is convenient to define the notion of step so that it is as machine-independent as possible.

- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - $T_A(n)$ = time of A on length n inputs

Running Time – cont'd

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes time c_i to execute and executes n times will contribute $c_i n$ to the total running time.

	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	$n-1$

Running Time – cont’d

The total running time T(n) for INSERTION-SORT is:

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

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

Running Time – cont'd

- The running time also depends on the input: an already sorted sequence (Best-case) is easier to sort.

- Generally, we seek upper bounds (Worst-case) on the running time, to have a guarantee of performance.

best-case for Insertion Sort

- The best case occurs if the array is already sorted, which means $t_j = 1$.
- $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
- It's a linear function of n.

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

Worst-case for Insertion Sort

- The worst case occurs if the array is reverse sorted.
- $T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$
- It is a quadratic function of n.

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

Worst-case and average-case analysis

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input.

- The average-case running time is the amount of time used by the algorithm, averaged over all possible inputs. The average-case is often roughly as bad as the worst case.

Order of growth

- For Insertion Sort, we expressed the worst-case running time as
    - $T(n) = an^2 + bn + c$
- We consider only the leading term of a formula (e.g., an^2).
- We write that insertion sort has a worst-case time complexity of
    - $O(n^2)$

Time complexity

- Time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm.
- We commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size.
- The time complexity is commonly expressed using big O notation
- Insertion sort has a time complexity of $O(n^2)$

Space complexity

- The space complexity of an algorithm is the amount of memory space required to solve an instance of the computational problem.

- Space complexity is often expressed in big O notation.

- Auxiliary space refers to space other than that consumed by the input

- We commonly considers the auxiliary space complexity

What is the space complexity of Insertion sort?

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

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6
2	3	4	6	8	9

Space complexity of Insertion sort

- In-place: An in-place algorithm updates its input sequence only through replacement or swapping of elements

- Only requires a constant amount $O(1)$ of additional memory space

- (Auxiliary) space complexity: $O(1)$

Selection sort

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

Selection sort

```
for i = 1 to n-1:
    min = i
    for j = i+1 to n do
        if a[j] < a[min]
            min = j
    swap a[i] and a[min]
```

Selection sort

- Worst-case Time complexity?
- Best-case time complexity?
- Average-case time complexity?
- (Auxiliary) space complexity?

Selection sort

- Worst-case Time complexity: $O(n^2)$
- Best-case time complexity: $O(n^2)$
- Average-case time complexity: $O(n^2)$
- (Auxiliary) space complexity: $O(1)$

Learning outcomes

- Algorithm definition
- Examples of algorithmic problems
- Insertion sort
- Analysis of algorithms
- Mathematical Induction
- Worst-case and average-case time complexity
- Space complexity
- Understand asymptotic complexity and notation
- Carry out simple asymptotic analysis of algorithms

Time Complexity Analysis

- How fast is the algorithm?
  - Depend on the speed of the computer
  - Waste time coding and testing if the algorithm is slow

- Identify some important operations/steps and count how many times these operations/steps needed to executed

- How to measure efficiency?
  - Number of operations usually expressed in terms of input size ***n***

Time Complexity Analysis

- Suppose:
  - an algorithm takes n^2 comparisons to sort n numbers
  - we need 1 sec to sort 5 numbers (25 comparisons)
- Now, if we can perform 2500 comparisons in 1 sec (100 times speedup), How many numbers we can sort?
  - 50 numbers (10 times more)

Time Complexity Analysis

- The time complexity of Insertion Sort is: $O(n^2)$
  - If we doubled the input size, how much longer would the algorithm take?
    - Roughly 4 times
  - If we trebled the input size, how much longer would it take?
    - Roughly 9 times

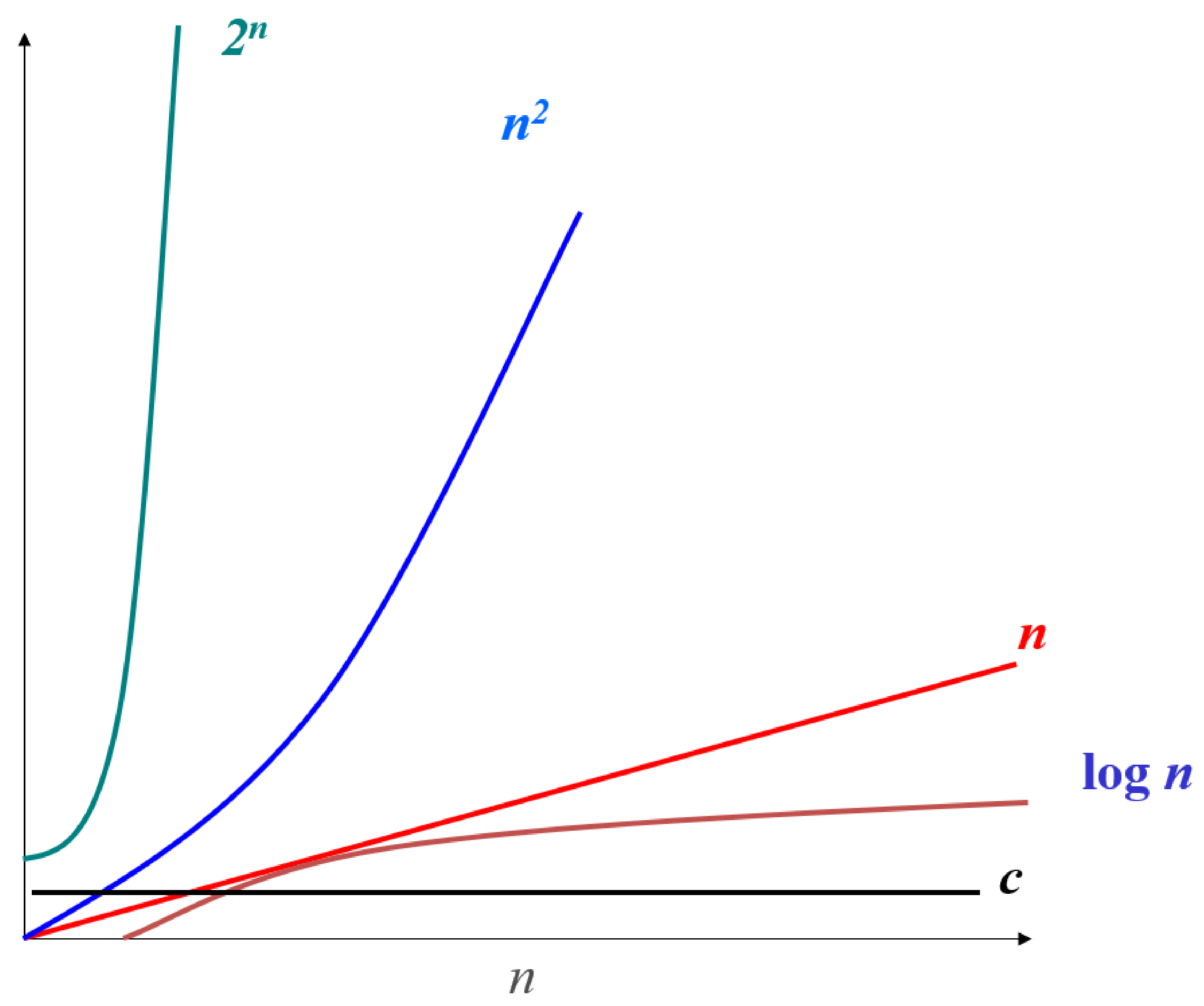
Time complexity
– Big O notation

Which algorithm is the fastest?

- Consider a problem that can be solved by 5 algorithms A_1, A_2, A_3, A_4, A_5 using different number of operations.

  - $f_1(n) = \log n$ $\quad (\log n \;\; stand\ for\ \log_2 n)$ $\quad (\log_2 2^x = x)$
  - $f_2(n) = c$ $\quad$ (constant)
  - $f_3(n) = n^2$
  - $f_4(n) = n$
  - $f_5(n) = 2^n$

Relative growth rate



Growth of functions

n	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	5.7	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.84×10^{19}
128	7	11.3	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	22.6	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	

Hierarchy of functions

- We can define a hierarchy of functions each having a **greater** order of magnitude than its predecessor:

1	$\log n$	$\underbrace{n \quad n^2 \; n^3 \ldots n^k \ldots}$	2^n
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
constant	logarithmic	polynomial	exponential

- As n increases, the values of the later functions increase more rapidly than the values of the earlier ones.

Hierarchy of functions

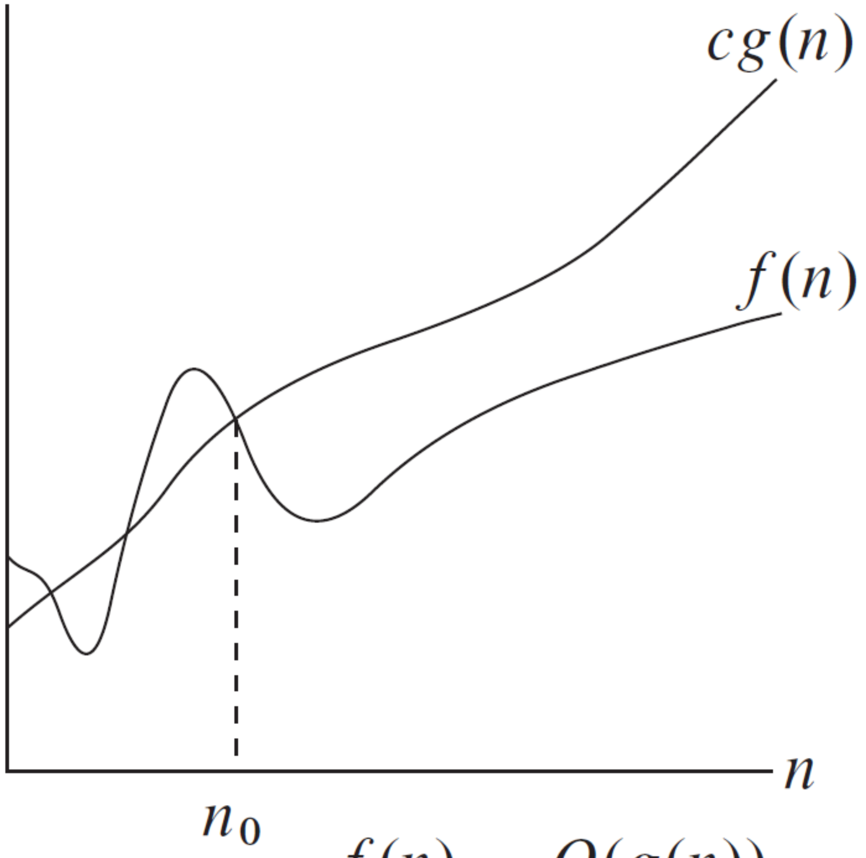
- When we have a function, we can assign the function to some function in the hierarchy:
  - For example, $\mathrm{f}(n) = an^2 + bn + c$
  - The term with the highest power is an^2.
    The growth rate of f(n) is dominated by n^2.

- This concept is captured by Big-O notation

Big-O notation

- f(n) = O(g(n)): There exists a constant c and n_0 such that
$$f(n) \le c \times g(n) \text{ for all } n \ge n_0$$

- O-notation provides an
- asymptotic upper bound
- on a function



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

Big-O notation

- Examples:
  - $2n^3 = O(n^3)$
  - $2n^3 + n^2 = O(n^3)$
  - $nlogn + n^2 = O(n^2)$

- function on L.H.S and function on R.H.S are said to have the same order of magnitude

Proof of order of magnitude

- Show that $2n^3 + n^2$ is $O(n^3)$
  - Since $n^2 < n^3$ for all n > 1,
  - we have $2n^3 + n^2 \leq 2n^3 + n^3 = 3n^3$ for all n > 1.
  - Therefore, by definition $2n^3 + n^2$ is $O(n^3)$. (c = 3, n_0 =1)

- Show that $nlogn + n^2$is $O(n^2)$
  - Since $logn < n$ for all n > 1,
  - we have$nlogn + n^2 \leq n^2 + n^2 = 2n^2$ for all n > 1.
  - Therefore, by definition$nlogn + n^2$is $O(n^2)$. (c = 2, n_0 =1)

Exercises

- Prove the order magnitude:
  - Show that $n^3 + 3n^2 + 3$ is $O(n^3)$
  - Show that $4n^2 \log n + n^3 + 5n^2 + n$ is $O(n^3)$

Exercises

- $n^3 + 3n^2 + 3$
  - $3n^2 \le n^3 \quad \forall n \ge 3$
  - $3 \le n^3 \quad \forall n \ge 2$
  - $\Longrightarrow n^3 + 3n^2 + 3 \le 3n^3 \quad \forall n \ge 3$

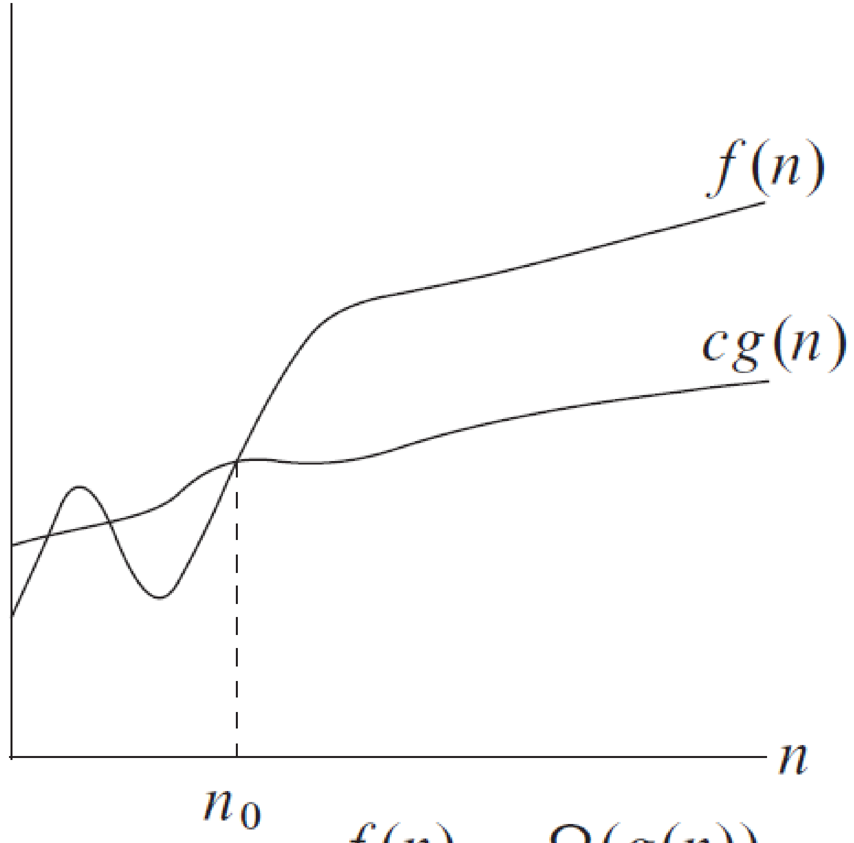
- $4n^2 \log n + n^3 + 5n^2 + n$
  - $4n^2 logn \le 4n^3 \quad \forall n \ge 1$
  - $5n^2 \le n^3 \quad \forall n \ge 5$
  - $\mathrm{n} \le n^3 \quad \forall n \ge 1$
  - $\Longrightarrow 4n^2 logn + n^3 + 5n^2 + n \le 7n^3 \quad \forall n \ge 5$

c and n_0 could be different when proving the order of magnitude

Ω-notation

- $f(n) = \Omega(g(n))$: There exists a constant c and n_0 such that $c \times g(n) \le f(n)$ for all $n \ge n_0$

- Ω-notation provides an
- asymptotic lower bound.

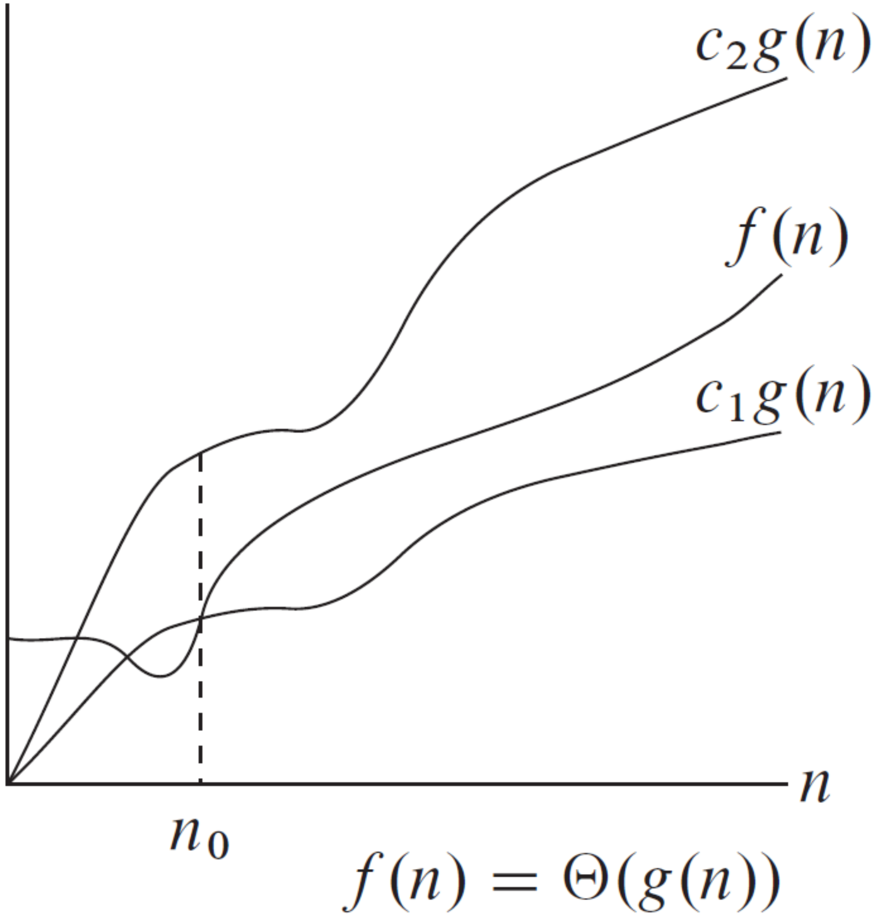


$f(n) = \Omega(g(n))$

Θ-notation

- f(n) = Θ(g(n)): There exists constant c_1, c_2 and n_0 such that $c_1 \times g(n) \leq \mathrm{f(n)} \leq c_2 \times g(n)$ for all $n \geq n_0$

- Θ notation provides an
- asymptotically tight bound



Asymptotic Notations

- Since O-notation describes an upper bound, we usually use it to bound the worst-case running time of an algorithm.
  - $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input.
  - The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, does not imply $\Theta(n^2)$ bound on the running time of insertion sort on every input. Best-case insertion sort runs in $\Theta(n)$ time.
  - $n = O(n^2)$, BUT O-notation informally describing asymptotically tight upper bounds

Exercises

- Write the computation complexity directly:
  - $n^3 + 3n^2 + 3$ $O(n^3)$
  - $4n^2 \log n + n^3 + 5n^2 + n$ $O(n^3)$
  - $2n^2 + n^2 \log n$ $O(n^2 \log n)$
  - $6n^2 + 2^n$ $O(2^n)$

Time complexity of this?

```
for (i=0;i<n;i++)
{
    stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=n;i>0;i--)
{
      stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=0;i<n;i=i+2)
{
    stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
    {
        stmt
    }
```

O(?)

O(n^2)

Time complexity of this?

```
for (i=0;i<n;i++)
{
      stmt
}
for (j=0;j<n;j++)
{
      stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=0;i<n;i++)
    for (j=0;j<i;j++)
    {
        stmt
    }
```

O(?)

$\mathbf{O}(n^2)$

Time complexity of this?

```
j=0
for (i=0;j<n;i++)
{
      j=j+i
}
```

O(?)

$$\mathbf{O}(\sqrt{n})$$

Time complexity of this?

```
for (i=1;i<n;i=i*2)
{
      stmt
}
```

O(?)

O(logn)

Time complexity of this?

```
k=0
for (i=1;i<n;i=i*2)
{
      k++;
}
for (j=1;j<k;j=j*2)
{
      stmt
}
```

O(?)

O(loglogn)

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Time complexity of this?

```
for (i=0;i<n;i++)
{
    for (j=1;j<n;j=j*2)
    {
        stmt
    }
}
```

O(?)

O(nlogn)

Some algorithms we learnt

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

O(?)

$$O(n^2)$$

Some algorithms we learnt

```
for i = 1 to n-1:
    min = i
    for j = i+1 to n do
        if a[j] < a[min]
            min = j
    swap a[i] and a[min]
```

O(?)

$O(n^2)$

Searching

- Input: n numbers $a_1, a_2, \ldots, a_n$ and a number X
- Output: determine if X is in the sequence or not

Sequential search

To find 7

- **12** 34 2 9 7 5
  7
- 12 **34** 2 9 7 5
  7
- 12 34 **2** 9 7 5
  7
- 12 34 2 **9** 7 5
  7
- 12 34 2 9 **7** 5
  7 found!

Sequential search

To find 10

- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10 not found!

Sequential search

```
i = 1
found = false
while (i<=n && found==false)
{
        if X == a[i] then
                found = true
        else
                i = i+1
}
```

Best case: X is 1st no.
$\Rightarrow$ 1 comparison $\Rightarrow$ O(1)

Worst case: X is last OR X is not found $\Rightarrow$ n comparisons $\Rightarrow$ O(n)

How to improve Searching?

- Time complexity of Sequential searching is $O(n)$.
- If a sorted array is given, can we improve the time complexity?

Binary search

- Input: a sequence of n **sorted** numbers $a_1, a_2, \ldots, a_n$ in ascending order and a number X

- Idea of algorithm:
  - compare X with number in the middle
  - then focus on only the first half or the second half (depend on whether X is smaller or greater than the middle number)
  - reduce the amount of numbers to be searched by half

Binary Search

To find 24

3 7 11 12 15 19 24 33 41 55
24

19 24 33 41 55
24

19 24
24

24
24

found!

Binary Search

To find 30

3 7 11 12 **15** 19 24 33 41 55

30

19 24 **33** 41 55

30

19 24

30

24

30

not found!

Binary Search – Pseudo Code

```
first = 1, last = n, found = false
while (first <= last && found == false)
{
        mid = ⌊(first+last)/2⌋
        if (X == a[mid])
             found = true
        else
             if (X < a[mid])
                   last = mid-1
           else
                  first = mid+1
}
if (found == true)
        report "Found"
else
        report "Not Found"
```

Best case: X is the number in the middle ⇒ 1 comparison ⇒ O(1)

Worst case: at most (logn+1) comparisons ⇒ O(logn)

Why?
Every comparison reduces the amount of numbers by at least half
E.g., 16 ⇒ 8 ⇒ 4 ⇒ 2 ⇒ 1

Learning outcomes

- Algorithm definition
- Examples of algorithmic problems
- Insertion sort
- Analysis of algorithms
- Mathematical Induction
- Worst-case and average-case time complexity
- Space complexity
- Understand asymptotic complexity and notation
- Carry out simple asymptotic analysis of algorithms