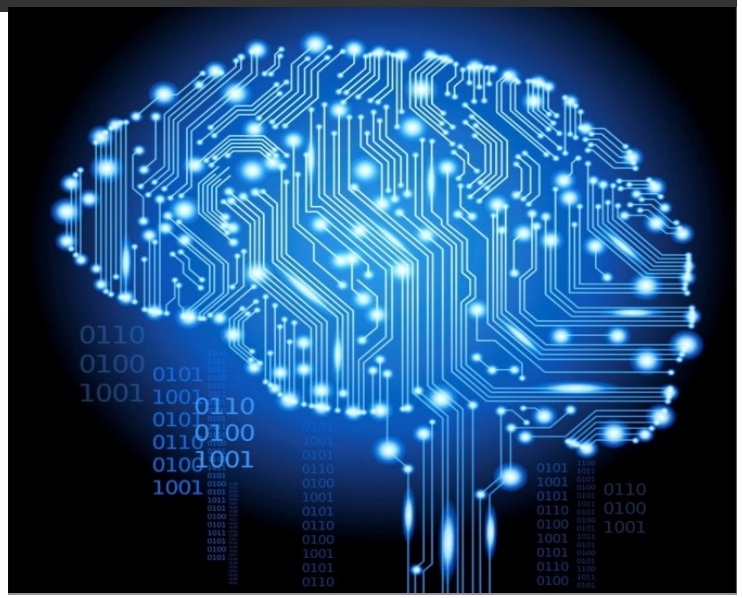
MONASH University

Information Technology

FIT1008/2085
Algorithms and Algorithmic Complexity

Prepared by:
Alexey Ignatiev



Objectives for this lecture

- **To understand the concept of an algorithm**
- **To understand algorithmic complexity**
  - See how complexity is defined (as dependent on problem size)
  - Understand the basics of asymptotics
  - Study the Big-O notation and its basic properties
- **Get familiar with common complexity "classes":**
  - $O(1)$, $O(n)$, $O(\log n)$, $O(n^2)$, etc.
- **In the following lessons, we will use this knowledge to understand the complexity of sorting algorithms**
  - Best-case complexity
  - Worst-case complexity

MONASH University

Algorithm

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist
- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist
- **Example problems:**
  - Given integers a and b, find their greatest common divisor
- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Example problems:**
  - Given integers a and b, find their greatest common divisor

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship

Example: `GCD(45, 30) -> 15`

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist
- **Example problems:**
  - Given integers a and b, find their greatest common divisor
- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship
- **Example algorithms:**
  - Euclid's algorithm for GCD

Example: `GCD(45, 30) -> 15`

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Example problems:**
  - Given integers a and b, find their greatest common divisor
  - Given nodes v and u in a graph G, find the shortest path from v to u

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship
- **Example algorithms:**
  - Euclid's algorithm for GCD
  - Dijkstra's algorithm for shortest path

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Example problems:**
  - Given integers a and b, find their greatest common divisor
  - Given nodes v and u in a graph G, find the shortest path from v to u
  - Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, produce a sorted list $\langle a'_1, \ldots, a'_n \rangle$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship
- **Example algorithms:**
  - Euclid's algorithm for GCD
  - Dijkstra's algorithm for shortest path

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Example problems:**
  - Given integers a and b, find their greatest common divisor
  - Given nodes v and u in a graph G, find the shortest path from v to u
  - Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, produce a sorted list $\langle a'_1, \ldots, a'_n \rangle$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship

- **Example algorithms:**
  - Euclid's algorithm for GCD
  - Dijkstra's algorithm for shortest path

Example: `SORT([4, 0, 3, -1]) -> [-1, 0, 3, 4]`

What is an algorithm and when is it needed?

- **General computational problem:**
  - Well-specified but this is *subjective*
  - Defines input-to-output relationship
  - Various problem instances exist

- **Example problems:**
  - Given integers a and b, find their greatest common divisor
  - Given nodes v and u in a graph G, find the shortest path from v to u
  - Given a sequence of numbers $\langle a_1, \ldots, a_n \rangle$, produce a sorted list $\langle a'_1, \ldots, a'_n \rangle$ s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- **Algorithm to solve it:**
  - A finite set of instructions
    - Well-specified but *subjective*
  - Correct:
    - *Halts* for any problem instance
    - Correct input-to-output relationship

- **Example algorithms:**
  - Euclid's algorithm for GCD
  - Dijkstra's algorithm for shortest path
  - Bubble Sort algorithm

Example: `SORT([4, 0, 3, -1]) -> [-1, 0, 3, 4]`

Running example 1 – problem formulation

- **Given a positive integer k, determine whether k is even**

Running example 1 – problem formulation

- **Given a positive integer k, determine whether k is even**
  - Examples:
    - `is_even(10) -> True`
    - `is_even(7)  -> False`

Running example 1 – problem formulation

- **Given a positive integer k, determine whether k is even**
  - Examples:
    - `is_even(10) -> True`
    - `is_even(7) -> False`
  - Assume binary representation of input k

Example:

```
bin(10) = 0b1010
 bin(7) = 0b111
bin(27) = 0b11011
```

Running example 1 – problem formulation

- **Given a positive integer k, determine whether k is even**
  - Examples:
    - `is_even(10) -> True`
    - `is_even(7)  -> False`
  - Assume binary representation of input k
  - How do we solve it?
    - There are at least 2 algorithms!

Example:

```
bin(10) = 0b1010
 bin(7) = 0b111
bin(27) = 0b11011
```

Running example 1 – problem formulation

- **Given a positive integer k, determine whether k is even**
  - Examples:
    - `is_even(10) -> True`
    - `is_even(7)  -> False`
  - Assume binary representation of input k
  - How do we solve it?
    - There are at least 2 algorithms!

Example:

```
bin(10) = 0b1010
 bin(7) = 0b111
bin(27) = 0b11011
```

Hint: *we need to check the right-most digit in the binary representation of k.*

Running example 1 - algorithms

- **Traverse the bits from left to right:**

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b101

res = None
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b101

res = None
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b01
b = 1
res = None
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b01
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b01
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b1
b = 0
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b1
b = 0
res = True
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b1
b = 0
res = True
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b0
b = 1
res = True
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b0
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b0
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b0
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

Looks hard! Can we do better?

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b101
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b10
b = 1
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b10
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

Let $k = 5$:

```
k = 0b10
b = 1
res = False
```

Running example 1 - algorithms

- **Traverse the bits from left to right:**
  - Assume that we have access only to the left-most bit of k
    - *Unrealistic assumption but…*

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**
  - Any language supports this
    - *Including Python*

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

Note: Both algorithms solve the same problem but one runs faster than the other!

Running example 2 – problem formulation

- Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.

Running example 2 – problem formulation

- **Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.**
  - The protocol we can follow:
    - We can iteratively make guesses $b \in \{0, \ldots, k\}$
    - Alice replies with -1, 0, or 1 if $b < a$, $b = a$, $b > a$, respectively
    - The problem is solved when our guess is correct, i.e. when $b = a$

Running example 2 – problem formulation

- **Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.**
  - The protocol we can follow:
    - We can iteratively make guesses $b \in \{0, \ldots, k\}$
    - Alice replies with -1, 0, or 1 if $b < a$, $b = a$, $b > a$, respectively
    - The problem is solved when our guess is correct, i.e. when $b = a$
  - Assume the following implementation for Alice's reply:

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Running example 2 – problem formulation

- **Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.**
  - The protocol we can follow:
    - We can iteratively make guesses $b \in \{0, \ldots, k\}$
    - Alice replies with -1, 0, or 1 if $b < a$, $b = a$, $b > a$, respectively
    - The problem is solved when our guess is correct, i.e. when $b = a$
  - Assume the following implementation for Alice's reply:

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Two conditional instructions.

Running example 2 – problem formulation

- **Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.**
  - The protocol we can follow:
    - We can iteratively make guesses $b \in \{0, \ldots, k\}$
    - Alice replies with -1, 0, or 1 if $b < a$, $b = a$, $b > a$, respectively
    - The problem is solved when our guess is correct, i.e. when $b = a$
  - Assume the following implementation for Alice's reply:

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Two conditional instructions.

Two `return` statements.

Running example 2 – problem formulation

- **Alice thinks of a positive number $0 \leq a < k$. The value of a is *hidden* from us. Given the known upper bound k, the problem is to find a.**
  - The protocol we can follow:
    - We can iteratively make guesses $b \in \{0, \ldots, k\}$
    - Alice replies with -1, 0, or 1 if $b < a$, $b = a$, $b > a$, respectively
    - The problem is solved when our guess is correct, i.e. when $b = a$
  - Assume the following implementation for Alice's reply:

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Two conditional instructions.

Two `return` statements.

`int(True) == 1` and `int(False) == 0`

Running example 2 - algorithms

- **Naïve approach:**
  - Test every value from 0 to $k-1$
  - Stop when the correct one is found

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b

def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Running example 2 - algorithms

- **Naïve approach:**
  - Test every value from 0 to $k-1$
  - Stop when the correct one is found

This algorithm makes k iterations in the worst case.

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b

def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Running example 2 - algorithms

- **Naïve approach:**
  - Test every value from 0 to $k-1$
  - Stop when the correct one is found

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b

def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**
  - Halve the search space at each step
    - *We will cover binary search later!*

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

Running example 2 - algorithms

- **Naïve approach:**
  - Test every value from 0 to $k-1$
  - Stop when the correct one is found

```
def naive(k):
    for b in range(k):
        if ali
            br
    return b

def alice_repl
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

Depending on the outcome, we either
(1) stop, or
(2) consider interval $[b+1, ub]$, or
(3) consider interval $[lb, b-1]$

- **A computer scientist's approach:**
  - Halve the search space at each step
    - *We will cover binary search later!*

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

Running example 2 - algorithms

- **Naïve approach:**
  - Test every value from 0 to $k-1$
  - Stop when the correct one is found
- **A computer scientist's approach:**
  - Halve the search space at each step
    - *We will cover binary search later!*

Two algorithms solve the same problem but run differently!

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b

def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

MONASH University

Algorithmic complexity

Algorithmic complexity

alternatively, **computational complexity**

Algorithmic complexity

alternatively, **computational complexity**

- **Basically, there are two types:**

  - **Time complexity**
    - How much time does an algorithm spend solving the corresponding problem?
      - *Usually measured as the number of "elementary operations" performed*

Algorithmic complexity

alternatively, **computational complexity**

- **Basically, there are two types:**

  - **Time complexity**
    - How much time does an algorithm spend solving the corresponding problem?
      - *Usually measured as the number of "elementary operations" performed*

  - **Space complexity**
    - How much space does an algorithm spend solving the corresponding problem?
      - Usually measured as the amount of "memory" occupied simultaneously

Algorithmic complexity

alternatively, **computational complexity**

- **Basically, there are two types:**
  - **Time complexity**
    - How much time does an algorithm spend solving the corresponding problem?
      - *Usually measured as the number of "elementary operations" performed*
  - **Space complexity**
    - How much space does an algorithm spend solving the corresponding problem?
      - Usually measured as the amount of "memory" occupied simultaneously
- **Both types of complexity are measured wrt. the input size!**

Algorithmic complexity

alternatively, **computational complexity**

- **Basically, there are two types:**
  - **Time complexity**
    - How much time does an algorithm spend solving the corresponding problem?
      - *Usually measured as the number of "elementary operations" performed*
  - **Space complexity**

Space complexity is out of scope of FIT1008!

    - How much space does an algorithm spend solving the corresponding problem?
      - Usually measured as the amount of "memory" occupied simultaneously
- **Both types of complexity are measured wrt. the input size!**

Time complexity as a function of size

- **Algorithm's time complexity can be measured as a function of input size**
  - Given an input of size n, we can calculate the value $T(n)$…
  - … as the number of *"elementary" steps* (instructions) performed by the algorithm on the input of size n
    - each elementary step contributes a 1 to the total value $T(n)$

Often they are *memory accesses*.

Time complexity as a function of size

- **Algorithm's time complexity can be measured as a function of input size**
  - Given an input of size n, we can calculate the value $T(n)$…
  - … as the number of *"elementary" steps* (instructions) performed by the algorithm on the input of size n
    - each elementary step contributes a 1 to the total value $T(n)$

Often they are *memory accesses*.

- **The main question: *"how does the algorithm perform when n is large?"***
  - In other words, *"how well does it scale?"*

Time complexity as a function of size

- **Algorithm's time complexity can be measured as a function of input size**
  - Given an input of size n, we can calculate the value $T(n)$…
  - … as the number of *"elementary" steps* (instructions) performed by the algorithm on the input of size n
    - each elementary step contributes a 1 to the total value $T(n)$

Often they are *memory accesses*.

- **The main question: *"how does the algorithm perform when n is large?"***
  - In other words, *"how well does it scale?"*
  - We are mostly interested in the "order of approximation" of $T(n)$

This is where the **Big-O notation** helps!

Time complexity as a function of size

- **Algorithm's time complexity can be measured as a function of input size**
  - Given an input of size n, we can calculate the value $T(n)$…
  - … as the number of *"elementary" steps* (instructions) performed by the algorithm on the input of size n
    - each elementary step contributes a 1 to the total value $T(n)$

Often they are *memory accesses*.

- **The main question: *"how does the algorithm perform when n is large?"***
  - In other words, *"how well does it scale?"*
  - We are mostly interested in the "order of approximation" of $T(n)$
  - i.e. when n approaches $+\infty$

This is where the **Big-O notation** helps!

We apply **asymptotic analysis** for this!

Time complexity as a function of size

- **Algorithm's time complexity can be measured as a function of input size**
  - Given an input of size n, we can calculate the value $T(n)$…
  - … as the number of *"elementary" steps* (instructions) performed by the algorithm on the input of size n
    - each elementary step contributes a 1 to the total value $T(n)$

Often they are *memory accesses*.

- **The main question: *"how does the algorithm perform when n is large?"***
  - In other words, *"how well does it scale?"*
  - We are mostly interested in the "order of approximation" of $T(n)$
  - i.e. when n approaches $+\infty$

This is where the **Big-O notation** helps!

We apply **asymptotic analysis** for this!

- **Another question: *"what is n?"***

Sometimes it is clear, sometimes not – ***let's see next!***

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
- **Example problem:**
  - *"is integer n even?"*

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation

- **Example problem:**
  - *"is integer n even?"*
  - `n = 10 = 0b1010`

Size equals 4

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation

- **Example problem:**
  - *"is integer n even?"*
  - `n = 10 = 0b1010`

Size equals 4

  - `n = 125 = 0b1111101`

Size equals 7

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation

- **Example problem:**
  - *"is integer n even?"*
  - `n = 10 = 0b1010`

Size equals 4

  - `n = 125 = 0b1111101`

Size equals 7

  - in general, size equals $\lceil \log_2(n+1) \rceil$

$\approx \log_2 n$

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection

- **Example problem:**
  - *"sort an input array"*
  - size of `[1, 2, 3, 5, 8]` is 5

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection
  - **String input:**
    - Number of characters

- **Example problem:**
  - *"is a given string a palindrome?"*
  - size of "`hello world!`" is 12

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection
  - **String input:**
    - Number of characters
  - **$n \times m$ matrix:**
    - Number of rows n
    - Number of columns m

- **Example problem:**
  - "multiply two matrices"

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection
  - **String input:**
    - Number of characters
  - **$n \times m$ matrix:**
    - Number of rows n
    - Number of columns m

- **Example problem:**
  - "multiply two matrices"

$$\begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}$$

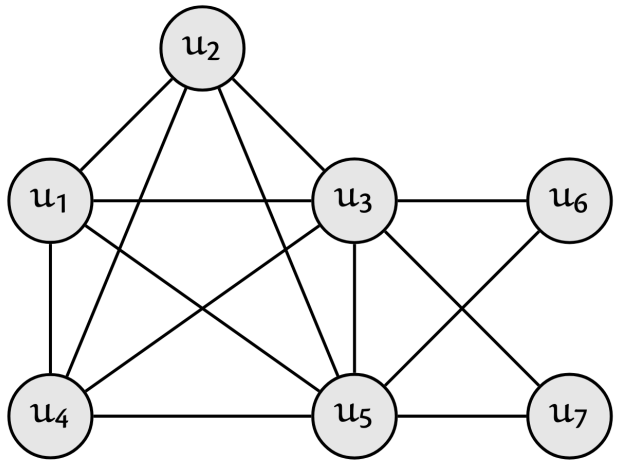
Number of columns equals m

Number of rows equals n

What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection
  - **String input:**
    - Number of characters
  - **$n \times m$ matrix:**
    - Number of rows n
    - Number of columns m
  - **Graph or tree:**
    - Number of nodes
  - **Etc.**

- **Example problem:**
  - *"find a maximum clique in a graph"*
  - The size of this graph equals 7:



What is input size?

- **It depends on the type of input:**
  - **Numeric input:**
    - Length of its binary representation
  - **Collection (array, list, set, stack):**
    - Number of elements in collection
  - **String input:**
    - Number of characters
  - **$n \times m$ matrix:**
    - Number of rows n
    - Number of columns m
  - **Graph or tree:**
    - Number of nodes
  - **Etc.**

In other cases, input size can be defined similarly.

It is normally clear from the context!

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input k**?

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input k**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

$T(n) = 1 +$

How many elementary steps given the **size of input k**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1

$n \approx \log k$

$T(n) = 1 + n\times$

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1

2

$n \approx \log k$

$T(n) = 1 + n \times (2$

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 (res = None)

2 (while k:)

$n \approx \log k$

2 (res = (b == 0))

1 (return res)

$T(n) = 1 + n \times (2 + 2)$

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 (`res = None`)

2 (`while k:`)

$n \approx \log k$

2 (`res = (b == 0)`)

1 (`return res`)

$T(n) = 1 + n \times (2 + 2) + 1$

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 (res = None)

2 (while k:)

$n \approx \log k$

2 (res = (b == 0))

1 (return res)

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 (res = None)

2 (while k:)

$n \approx \log k$

2 (b = pop_left(k))

2 (res = (b == 0))

1 (return res)

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

2 (def is_even2(k):)

$T(n) = 2 +$

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 (res = None)

2 (while k:)

$n \approx \log k$

2 (b = pop_left(k); res = (b == 0))

1 (return res)

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

2 (def is_even2(k):)

2 (res = (b == 0))

$T(n) = 2 + 2 +$

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 · 2 · 2 · 1 · $n \approx \log k$

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

2 · 2 · 1

$T(n) = 2 + 2 + 1$

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

1 · 2 · 2 · 1 · $n \approx \log k$

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

2 · 2 · 1

$T(n) = 2 + 2 + 1 = 5$

Constant function on the input size!

How many elementary steps given the **size of input n**?

Recall that size n of k equals $\log k$

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b



def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b



def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

4 elementary steps in total

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) =$

size of input!

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

$T(n) = 2^n \times$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4

$T(n) = 2^n \times (4$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

4 elementary steps in total

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4

1

$T(n) = 2^n \times (4 + 1$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

4 elementary steps in total

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4

1

1

$T(n) = 2^n \times (4 + 1 + 1)$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

4 elementary steps in total

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 1 1 1

$T(n) = 2^n \times (4 + 1 + 1) + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 1 1 1

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 1 1 1

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

3 1

$T(n) = 3 + 1 +$

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 1 1 1

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

3 3 $5 = 1 + 4$ 2 3 3 1

$T(n) = 3 + 1 + 16 \times$

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 1 1 1

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

3 n 3 $5 = 1 + 4$ 2 3 3 1

$T(n) = 3 + 1 + 16 \times n$

How many elementary steps here?

Recall that the size n of input is $\log k$.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 (for b in range(k):) 1 (if alice_replies(b) == 0:) 1 (break) 1 (return b)

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

How many elementary steps here?

Recall that the size n of input is $\log k$.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

3 (lb, ub = 0, k - 1) n (while lb <= ub:) 3 (b = (lb + ub) // 2) $5 = 1 + 4$ (reply = alice_replies(b)) 2 (return b) 3 (lb = b + 1) 3 (ub = b - 1) 1 (return lb)

$T(n) = 3 + 1 + 16 \times n$

Why at most n?

Initially, there are k possibilities.
Each iteration halves it,
thus decrementing $\log k$ by 1.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

at most $k = 2^{\log k} = 2^n$

4 (for the `for` line), 1 (for the `if` line), 1 (for the `break` line), 1 (for the `return` line)

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

4 elementary steps in total

How many elementary steps here?

Recall that the size n of input is $\log k$.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

3 (for `lb, ub = 0, k - 1`), n (for `while lb <= ub:`), 3 (for `b = (lb + ub) // 2`), $5 = 1 + 4$ (for `reply = alice_replies(b)`), 2 (for `return b`), 3 (for `lb = b + 1`), 3 (for `ub = b - 1`), 1 (for `return lb`)

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Why at most n?

Initially, there are k possibilities.
Each iteration halves it,
thus decrementing $\log k$ by 1.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

This is the number of elementary steps performed wrt. **input size $\log k$**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

Here we check all possibilities – this is known as "**brute force**".

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

This is the number of elementary steps performed wrt. **input size $\log k$**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

Here we check all possibilities – this is known as "**brute force**".

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

This is the number of elementary steps performed wrt. **input size $\log k$**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Linear on input size, i.e. on n.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Important: *the picture changes if we change the perspective.*

Consider the number of steps wrt. **input *k* itself**.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

Linear on k.

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1 = \mathbf{6} \times \boldsymbol{k} + \mathbf{1}$

Important: *the picture changes if we change the perspective.*

Consider the number of steps wrt. **input k itself**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

Linear on k.

$$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1 = \mathbf{6} \times \boldsymbol{k} + \mathbf{1}$$

Important: *the picture changes if we change the perspective.*

Consider the number of steps wrt. **input k itself**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4 = \mathbf{16} \times \mathbf{\log} \boldsymbol{k} + \mathbf{4}$$

Logarithmic on k.

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

Linear on k.

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1 = \mathbf{6} \times \boldsymbol{k} + \mathbf{1}$

Important: *the picture changes if we change the perspective.*

Consider the number of steps wrt. **input k itself**.

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4 = \mathbf{16} \times \mathbf{\log} \boldsymbol{k} + \mathbf{4}$

Logarithmic on k.

Always be clear about the input size when analyzing complexity!

MONASH University

Asymptotics and Big-O Notation

Rationale

- **In practice, we do not count the exact number of elementary steps**
  - It is difficult
  - It is time-consuming
  - It is *error-prone*
- **Instead, we determine the most "expensive" parts of the code**
  - You will master this skill with (lots of) practice!
- **How badly do they impact the "rate of growth" of algorithmic complexity?**

This is when the Big-O notation comes in handy!

A bit of history…

Sometimes called **Bachmann-Landau notation.**

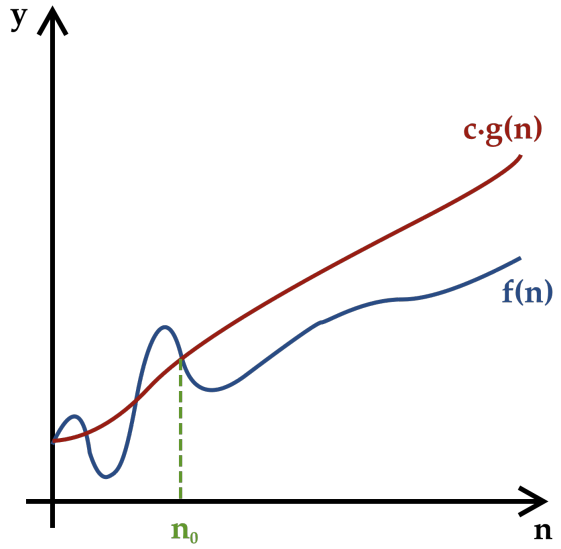
- **The original concept and the corresponding O-notation was introduced by German mathematician *Paul Bachmann* in 1894**
  - The name is determined by the German word *"ordnung"*
  - i.e. *"order of approximation"*
- **Popularised in mathematics by German mathematician *Edmund Landau***
  - *Mathematical analysis, differential equations, etc.*
- **Adapted by Donald Knuth for algorithmic complexity analysis in CS**
  - *This is what we are using!*

What is O()?

- **Assume we are given two functions $f(n)$ and $g(n)$ defined on $n \in \mathbb{N}$.**
- **$f(n) = O(g(n))$ if the following holds:**
  - there are two positive constants n_0 and c such that for all $n \geq n_0$

$$0 \leq f(n) \leq c \cdot g(n)$$

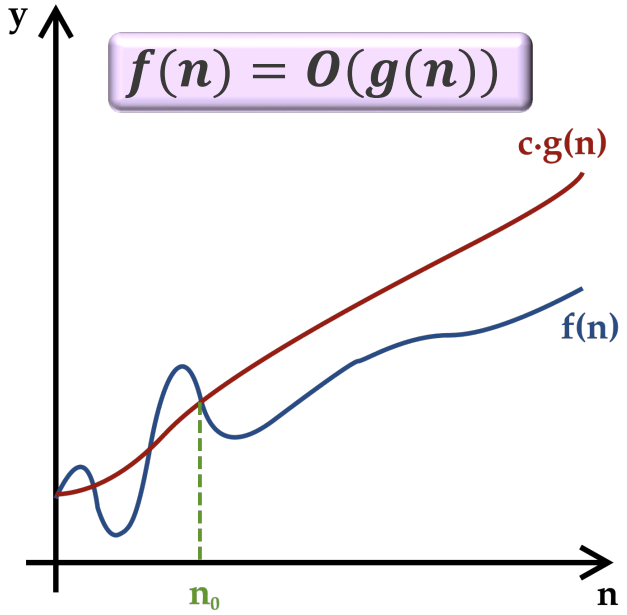
  - *$g(n)$ is referred to as asymptotic upper bound for $f(n)$, i.e. as n approaches $+\infty$*
- **Geometric interpretation:**



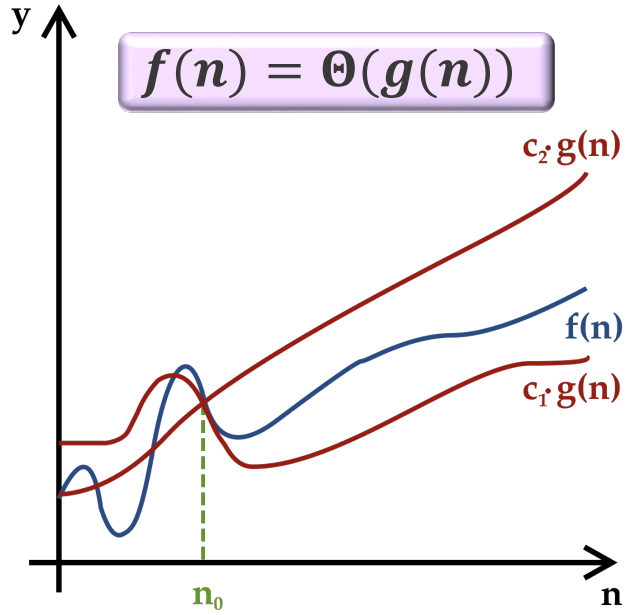
This is what asymptotic analysis means.

Big-Omega and Big-Theta – similar notation

Out of scope of FIT1008

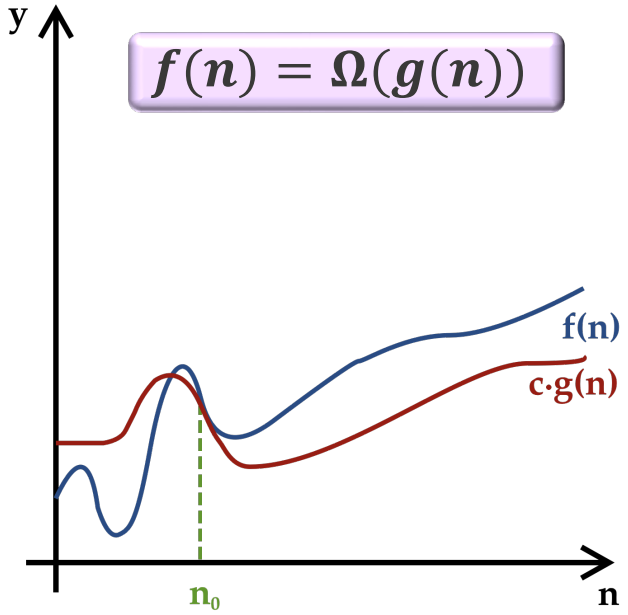


$g(n)$ - asymptotic upper bound for $f(n)$



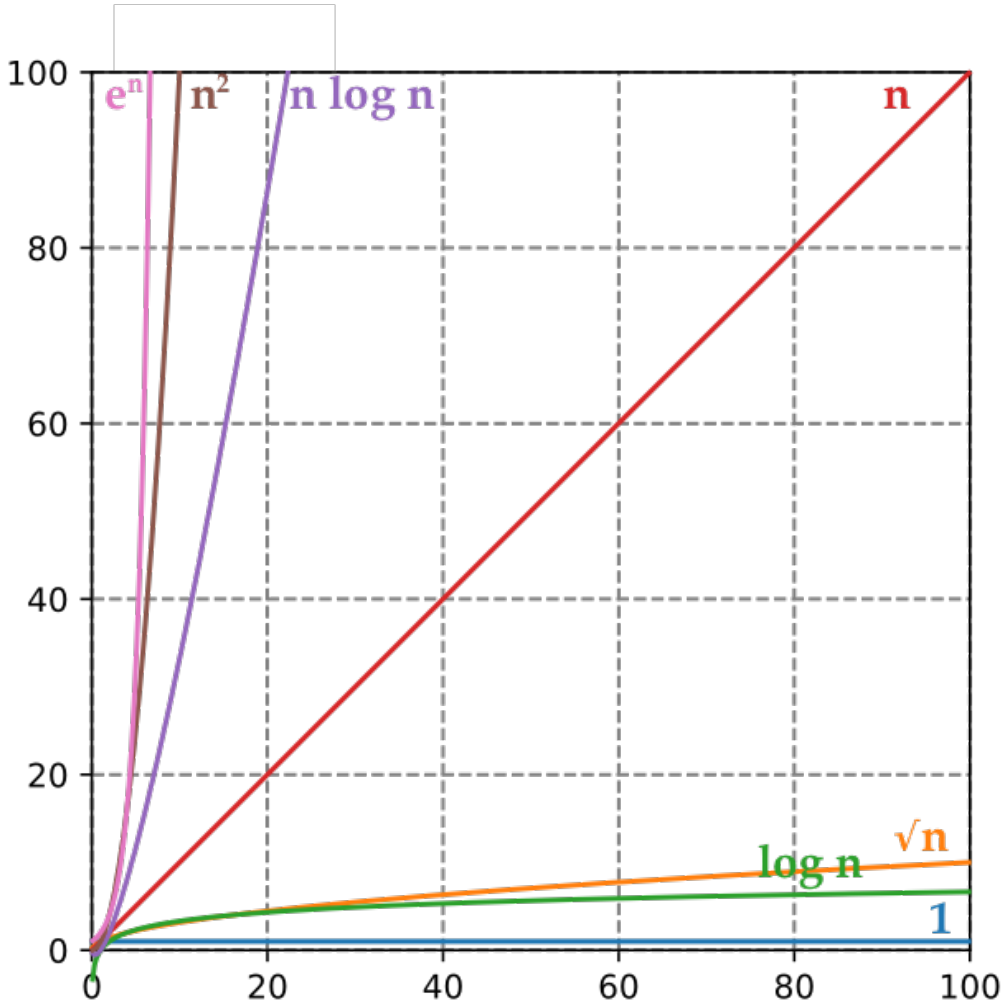
$g(n)$ - asymptotic tight bound for $f(n)$

both lower and upper!



$g(n)$ - asymptotic lower bound for $f(n)$

Common function growth rates



Big-O examples

- Is it true that $4n^2 + 1000n + 100 = O(n^2)$?

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**
  - No! It is $O(n^3)$!

Why? Let $c = 2$. Then $n^3 + n^2 \cdot \log^2 n \leq 2n^3$ for all $n \geq n_0 = 16$.

There are **no such constants c' and n_0'** to make the definition of Big-O hold for $O(n^2 \cdot \log^2 n)$!

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**
  - No! It is $O(n^3)$!

Why? Let $c = 2$. Then $n^3 + n^2 \cdot \log^2 n \leq 2n^3$ for all $n \geq n_0 = 16$.

There are **no such constants c' and n_0'** to make the definition of Big-O hold for $O(n^2 \cdot \log^2 n)$!

- **In general, it is hard to select constants c and n_0 every time**

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**
  - No! It is $O(n^3)$!

Why? Let $c = 2$. Then $n^3 + n^2 \cdot \log^2 n \leq 2n^3$ for all $n \geq n_0 = 16$.

There are **no such constants c' and n_0'** to make the definition of Big-O hold for $O(n^2 \cdot \log^2 n)$!

- **In general, it is hard to select constants c and n_0 every time**
- **Instead, we can focus on a single component growing faster than others!**
  - We say that such a component **dominates** all the others.

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**
  - No! It is $O(n^3)$!

Why? Let $c = 2$. Then $n^3 + n^2 \cdot \log^2 n \leq 2n^3$ for all $n \geq n_0 = 16$.

There are **no such constants c' and n_0'** to make the definition of Big-O hold for $O(n^2 \cdot \log^2 n)$!

- **In general, it is hard to select constants c and n_0 every time**
- **Instead, we can focus on a single component growing faster than others!**
  - We say that such a component **dominates** all the others.
  - For instance, $4n^2$ dominates $1000n + 100$ while n^3 dominates $n^2 \cdot \log^2 n$.

Big-O examples

- **Is it true that $4n^2 + 1000n + 100 = O(n^2)$?**
  - Yes!

Why? Let $c = 20$. Then $4n^2 + 1000n + 100 \leq 20n^2$ for all $n \geq n_0 = 63$.

- **Is it true that $n^3 + n^2 \cdot \log^2 n = O(n^2 \cdot \log^2 n)$?**
  - No! It is $O(n^3)$!

Why? Let $c = 2$. Then $n^3 + n^2 \cdot \log^2 n \leq 2n^3$ for all $n \geq n_0 = 16$.

There are **no such constants c' and n_0'** to make the definition of Big-O hold for $O(n^2 \cdot \log^2 n)$!

- **In general, it is hard to select constants c and n_0 every time**
- **Instead, we can focus on a single component growing faster than others!**
  - We say that such a component **dominates** all the others.
  - For instance, $4n^2$ dominates $1000n + 100$ while n^3 dominates $n^2 \cdot \log^2 n$.
  - See the previous slide for the growth rate of common functions.

The same applies to the analysis of algorithmic time complexity!

Basic properties of Big-O

- **Constant growth rate:**
  - If c is any positive constant then $c = O(1)$

The **dominating** one.

Recall: pick the fastest growing component!

- **Sum of O():**
  - If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
- **Product of O():**
  - If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
    - In particular, if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(1)$ then $f_1(n) \cdot f_2(n) = O(g_1(n))$

Thus, we ignore the constant!

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

$T(n) = 2 + 2 + 1 = 5$

Constant function on the input size!

What is the complexity in terms of $O()$?

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2 = \boldsymbol{O(n)}$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

$T(n) = 2 + 2 + 1 = 5$

Constant function on the input size!

What is the complexity in terms of $O()$?

Running example 1 – *checking if a number is even*

- **Traverse the bits from left to right:**

```
def is_even1(k):
    res = None
    while k:
        b = pop_left(k)
        res = (b == 0)
    return res
```

$T(n) = 1 + n \times (2 + 2) + 1 = 4n + 2 = \boldsymbol{O(n)}$

Linear function on the input size!

- **Get the value of right-most bit:**

```
def is_even2(k):
    b = pop_right(k)
    res = (b == 0)
    return res
```

$T(n) = 2 + 2 + 1 = 5 = \boldsymbol{O(1)}$

Constant function on the input size!

What is the complexity in terms of $O()$?

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

What is the complexity in terms of $O()$?

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1$

4 elementary steps in total = $\boldsymbol{O(1)}$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

What is the complexity in terms of $O()$?

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1 = \boldsymbol{O(2^n)}$

4 elementary steps in total = $\boldsymbol{O(1)}$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4$

What is the complexity in terms of $O()$?

Running example 2 – *finding a guessed number*

- **Naïve approach:**

```
def naive(k):
    for b in range(k):
        if alice_replies(b) == 0:
            break
    return b
```

$T(n) = 2^n \times (4 + 1 + 1) + 1 = 2^n \times 6 + 1 = \boldsymbol{O(2^n)}$

4 elementary steps in total = $\boldsymbol{O(1)}$

```
def alice_replies(b):
    if b < HIDDEN_VALUE_A:
        return -1
    else:
        return int(b > HIDDEN_VALUE_A)
```

- **A computer scientist's approach:**

```
def clever(k):
    lb, ub = 0, k - 1
    while lb <= ub:
        b = (lb + ub) // 2
        reply = alice_replies(b)
        if reply == 0:
            return b
        elif reply < 0:
            lb = b + 1
        elif reply > 0:
            ub = b - 1
    return lb
```

$T(n) = 3 + 1 + 16 \times n = 16 \times n + 4 = \boldsymbol{O(n)}$

What is the complexity in terms of $O()$?

A few more Big-O examples

- **What is the complexity of this?**
  - This function computes the sum of elements in a list:

```
def sum(input_list):
    res = 0
    for item in input_list:
        res += item
    return res
```

A few more Big-O examples

- **What is the complexity of this?**
  - This function computes the sum of elements in a list:

```
def sum(input_list):
    res = 0
    for item in input_list:
        res += item
    return res
```

  - Here, input size == number of items n
  - Loop iterates n times

Thus, depends on input size!

  - Complexity is $O(n)$

A few more Big-O examples

- **What is the complexity of this?**
  - This function computes the sum of elements in a list:

```
def sum(input_list):
    res = 0
    for item in input_list:
        res += item
    return res
```

  - Here, input size == number of items n
  - Loop iterates n times

Thus, depends on input size!

  - Complexity is $O(n)$

- **What is the complexity of this?**
  - This function computes the sum of first 10 elements in a list:

```
def sum(input_list):
    res = 0
    for i in range(min(len(input_list), 10)):
        res += input_list[i]
    return res
```

A few more Big-O examples

- **What is the complexity of this?**
  - This function computes the sum of elements in a list:

```
def sum(input_list):
    res = 0
    for item in input_list:
        res += item
    return res
```

  - Here, input size == number of items n
  - Loop iterates n times

Thus, depends on input size!

  - Complexity is $O(n)$

- **What is the complexity of this?**
  - This function computes the sum of first 10 elements in a list:

```
def sum(input_list):
    res = 0
    for i in range(min(len(input_list), 10)):
        res += input_list[i]
    return res
```

  - Again, input size == number of items n
  - Loop iterates at most 10 times – a constant!

Does not depend on input size!

  - Complexity is $O(1)$

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

A single operation!

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

A single operation!

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively
  - Complexity of `multiply` is $O(1)$

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

Iterates $l = 2^m$ times!

A single operation!

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively
  - Complexity of `multiply` is $O(1)$

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

Iterates $l = 2^m$ times!

A single operation!

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively
  - Complexity of `multiply` is $O(1)$
  - Complexity of `weird_func` is $O(2^m)$
    - Because it is $O(2^m) \times O(1)$

Yet another example

- **Analyse the complexity of function:**
  - Given 2 integers k and l, the function computes $k \times (-1)^l$

```
def weird_func(k, l):
    res = k
    for i in range(l):
        res = multiply(res)
    return res

def multiply(k):
    return k * (-1)
```

Iterates $l = 2^m$ times!

A single operation!

- **Some points to make:**
  - There are 2 inputs of size $n \approx \log k$ and $m \approx \log l$, respectively
  - Complexity of `multiply` is $O(1)$
  - Complexity of `weird_func` is $O(2^m)$
    - Because it is $O(2^m) \times O(1)$

But it is $O(l)$ if we are interested in complexity wrt. the input numbers k and l themselves.

Once again: it is crucial to understand and be clear about with respect to what the complexity is analysed!

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

Total complexity:

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n) \times \langle \ldots \rangle$

Total complexity: $O(n) \cdot (\ldots)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n) \times \langle \dots \rangle$
  - This part is $O(1)$

Total complexity: $O(n) \cdot (O(1) + \cdots)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\dots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\dots\rangle$

Total complexity: $\boldsymbol{O(n)\cdot(O(1)+O(1)\cdot(\dots)+)}$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$

Total complexity: $O(n)\cdot(O(1)+O(1)\cdot O(n)+\cdots)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Total complexity: $O(n)\cdot\left(O(1)+O(1)\cdot O(n)+O(2^n)\right)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Recall that this part becomes $O(n)$.

Total complexity: $O(n)\cdot\left(O(1)+O(1)\cdot O(n)+O(2^n)\right)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Total complexity: $\boldsymbol{O(n)\cdot\left(O(1)+O(n)+O(2^n)\right)}$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Inside the parentheses, $O(2^n)$ dominates.

Total complexity: $O(n)\cdot\left(O(1)+O(n)+O(2^n)\right)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Total complexity: $O(n)\cdot O(2^n)$.

Yet another example, sorry!

- **Analyse the complexity of `func0()`:**

```
def func0(input_of_size_n):
    for i in range(n):
        ... # constant time operations
        a = b + 2
        ... # constant time operations
        for j in range(100):
            res += func1(res)
        res -= func2(res)
    return res
def func1(something):
    ...
def func2(something):
    ...
```

Assume it is $O(n)$.

Assume it is $O(2^n)$.

- **Some points to make:**
  - The outer loop runs for n times
    - i.e. it should be $O(n)\times\langle\ldots\rangle$
  - This part is $O(1)$
  - The inner loop runs for 100 times
    - i.e. it is $O(1)\times\langle\ldots\rangle$
  - This is known to be $O(n)$
  - This is known to be $O(2^n)$

Total complexity: $O(n \cdot 2^n)$.

Summary

- **After these lesson you are now able to:**
  - Understand the concept of algorithm and how its time complexity is analysed
  - Understand the **Big-O notation** and its properties:
    - Basics of asymptotic analysis
    - Combination of Big-O's
  - Apply the Big-O notation when reasoning about algorithmic complexity
  - Understand basic algorithmic complexity classes and how they behave
    - Constant, linear, logarithmic, etc.