# Data Structures and Algorithms

## Lecture 2: Complexity and search

**Heikki Peura**

**h.peura@imperial.ac.uk**

# Last time

Algorithms and functions

- Recipes
- Abstraction

**Plan for today**:

- **Search** algorithms
- How to analyse **algorithm complexity**

Has anyone studied search algorithms before?

Has anyone studied search algorithms before?

Go to www.menti.com

- How many guesses will we need?

# What is the worst we could do?

If the answers were "yes" and "no"?

- Linear search - worst case: go through everything

If the answers are "too low" and "too high"?

- Each time, discard half of remaining numbers
- Binary search - worst case?

# Logarithms

**Exponentials**: $2^4 = 2 \times 2 \times 2 \times 2 = 16$

**Logarithm** is flipping the exponential:

- $\log_2 16 = 4$
- “How many 2s do we multiply to get 16?"
- “How many times do we divide 16 by 2 to get to 1?”

# Logarithms

**Exponentials**: $2^4 = 2 \times 2 \times 2 \times 2 = 16$

**Logarithm** is flipping the exponential:

- $\log_2 16 = 4$
- “How many 2s do we multiply to get 16?"
- “How many times do we divide 16 by 2 to get to 1?”

For numbers up to 128, we need to make at most $\log_2 128 = 7$ guesses (divide the range in half 7 times)

- What about for numbers up to 1024? 2048? 1,000,000?

# Logarithms

**Exponentials**: $2^4 = 2 \times 2 \times 2 \times 2 = 16$

**Logarithm** is flipping the exponential:

- $\log_2 16 = 4$
- “How many 2s do we multiply to get 16?"
- “How many times do we divide 16 by 2 to get to 1?”

For numbers up to 128, we need to make at most $\log_2 128 = 7$ guesses (divide the range in half 7 times)

- What about for numbers up to 1024? 2048? 1,000,000?

**Guess the number**:

- **Linear search** (guess in order): at most $n$ guesses
- **Binary search** (guess in middle): at most $\log_2(n)$ guesses

# Algorithm design: searching a list

Suppose we have a list `L`. We want to check whether it contains a number, eg `13`.

# What are computers good at?

1. Performing simple calculations
    - Arithmetic operations
    - Comparisons
    - Assignments
    - Accessing memory

2. Remembering the results

# Goals in designing algorithms

1. **Correctness** — finds the correct answer for any input
2. **Efficiency** — finds the answer quickly

# Goals in designing algorithms

1. **Correctness** — finds the correct answer for any input
2. **Efficiency** — finds the answer quickly

- It is important to understand both: think of airplane software, Uber or algorithmic trading...

# Goals in designing algorithms

1. **Correctness** — finds the correct answer for any input
2. **Efficiency** — finds the answer quickly

- It is important to understand both: think of airplane software, Uber or algorithmic trading...

**Efficiency**:

- How much **time** will our computation take?
- How much **memory** will it need?

# Example: linear search

Is `x` in list `A`?

```
def linear_search(A,x):
    for elem in A:
        if elem == x:
            return True
    return False

```

**Efficiency**:

- How much **time** will our computation take?
- How much **memory** will it need?

# How much time will it take?

**Simple**: run and time it? But time depends on

1. Speed of computer
2. Specifics of implementation
3. Value of input

# How much time will it take?

**Simple**: run and time it? But time depends on

1. Speed of computer
2. Specifics of implementation
3. Value of input

We can avoid 1 and 2 by measuring time in the **number of basic steps executed**

- Step: **constant-time computer operation**
  - Arithmetic operations
  - Comparisons
  - Assignments
  - Accessing memory

# How much time will it take?

**Simple**: run and time it? But time depends on

1. Speed of computer
2. Specifics of implementation
3. Value of input

We can avoid 1 and 2 by measuring time in the **number of basic steps executed**

- Step: **constant-time computer operation**
  - Arithmetic operations
  - Comparisons
  - Assignments
  - Accessing memory

For 3, **measure number of steps depending on the size of input**

# Complexity and input

Searching for an item in a list?

```
def linearSearch(A,x):
    # A is a list of length n
    for elem in A:
        if elem == x:
            return True
    return False
```

# Complexity and input

Searching for an item in a list?

```
def linearSearch(A,x):
    # A is a list of length n
    for elem in A:
        if elem == x:
            return True
    return False
```

- $x$ could be the first element of $A$
- $x$ could not be in $A$
- **How to give a general complexity measure?**

# Complexity cases

Cases for given input size (length of $A$):

- **Best case** — minimum time
- **Worst case** — maximum time
- **Average case** — average or expected time over all possible inputs

# Complexity cases

Cases for given input size (length of $A$):

- **Best case** — minimum time
- **Worst case** — maximum time
- **Average case** — average or expected time over all possible inputs

**Principle**: focus on worst-case analysis

- Upper bound on running time
- Bonus: usually easier to analyze

# Example

```python
def sum_up_to(n):
    result = 0                 # 1 step
    while n > 0:               # 1 step, n times
        result = result + n    # 2 steps, n times
        n = n - 1              # 2 steps, n times
    return result              # 1 step
```

# Example

```python
def sum_up_to(n):
    result = 0                  # 1 step
    while n > 0:                # 1 step, n times
        result = result + n     # 2 steps, n times
        n = n - 1               # 2 steps, n times
    return result               # 1 step
```

Total: $5n + 2$ steps

- As $n$ gets large, 2 is irrelevant
- Arguably, so is 5
  - It's the size of the problem that matters

# Example

```
def sum_up_to(n):
    result = 0                # 1 step
    while n > 0:              # 1 step, n times
        result = result + n   # 2 steps, n times
        n = n - 1             # 2 steps, n times
    return result             # 1 step
```

Total: $5n + 2$ steps

- As $n$ gets large, 2 is irrelevant
- Arguably, so is 5
  - It's the size of the problem that matters

**Principle 2**: ignore constant factors and lower-order terms

- These depend on computer and program implementation
- They do not matter for large inputs
- Simplifies comparisons

# Example

```python
def f(x):                    # x integer
    ans = 0                  # 1 step
    for i in range(100):
        ans += 1             # 200 steps
    for i in range(x):
        ans += 1             # 2*x
    for i in range(x):
        for j in range(x):
            ans -= 1         # 2*x^2
    return ans               # 1 step for return
```

# Example

```python
def f(x):                   # x integer
    ans = 0                 # 1 step
    for i in range(100):
        ans += 1            # 200 steps
    for i in range(x):
        ans += 1            # 2*x
    for i in range(x):
        for j in range(x):
            ans -= 1        # 2*x^2
    return ans              # 1 step for return
```

Steps: $202 + 2x + 2x^2$

- $x$ small -> first loop dominates ($x = 3$)

# Example

```python
def f(x):                    # x integer
    ans = 0                  # 1 step
    for i in range(100):
        ans += 1             # 200 steps
    for i in range(x):
        ans += 1             # 2*x
    for i in range(x):
        for j in range(x):
            ans -= 1         # 2*x^2
    return ans               # 1 step for return
```

Steps: $202 + 2x + 2x^2$

- $x$ small -> first loop dominates ($x = 3$)
- $x$ large -> last loop dominates ($x = 10^6$)

# Example

```python
def f(x):                    # x integer
    ans = 0                  # 1 step
    for i in range(100):
        ans += 1             # 200 steps
    for i in range(x):
        ans += 1             # 2*x
    for i in range(x):
        for j in range(x):
            ans -= 1         # 2*x^2
    return ans               # 1 step for return
```

Steps: $202 + 2x + 2x^2$

- $x$ small -> first loop dominates ($x = 3$)
- $x$ large -> last loop dominates ($x = 10^6$)
- Only need to consider last (nested) loop for large $x$

# Example

```
def f(x):                       # x integer
    ans = 0                     # 1 step
    for i in range(100):
        ans += 1                # 200 steps
    for i in range(x):
        ans += 1                # 2*x
    for i in range(x):
        for j in range(x):
            ans -= 1            # 2*x^2
    return ans                  # 1 step for return
```

Steps: $202 + 2x + 2x^2$

- $x$ small -> first loop dominates ($x = 3$)
- $x$ large -> last loop dominates ($x = 10^6$)
- Only need to consider last (nested) loop for large $x$
- Does the 2 in $2x^2$ matter? For large $x$, order of growth much more important

# Algorithm analysis

**Principle 0**: measure number of basic operations as function of input size

# Algorithm analysis

**Principle 0**: measure number of basic operations as function of input size

**Principle 1**: focus on worst-case analysis

# Algorithm analysis

**Principle 0**: measure number of basic operations as function of input size

**Principle 1**: focus on worst-case analysis

**Principle 2**: ignore constant factors and lower-order terms

# Algorithm analysis

**Principle 0**: measure number of basic operations as function of input size

**Principle 1**: focus on worst-case analysis

**Principle 2**: ignore constant factors and lower-order terms

**Principle 3**: only care about large inputs

- Only large problems are interesting
- What happens when size gets very large?

# Algorithm analysis

**Principle 0**: measure number of basic operations as function of input size

**Principle 1**: focus on worst-case analysis

**Principle 2**: ignore constant factors and lower-order terms

**Principle 3**: only care about large inputs

- Only large problems are interesting
- What happens when size gets very large?

**Formal way to describe this approach**:

- Big-O notation: upper bound on worst-case running time

# Big-O: bound on runtime growth rate

Let $T(n)$ be the number of steps taken for input size $n$:

- Example: $T(n) = 202 + 2n + 2n^2$

# Big-O: bound on runtime growth rate

Let $T(n)$ be the number of steps taken for input size $n$:

- Example: $T(n) = 202 + 2n + 2n^2$

**Definition**: $T(n)$ is $O(f(n))$ if for all sufficiently large $n$, $T(n)$ is bounded above by a constant multiple of $f(n)$

# Big-O: bound on runtime growth rate

Let $T(n)$ be the number of steps taken for input size $n$:

- Example: $T(n) = 202 + 2n + 2n^2$

**Definition**: $T(n)$ is $O(f(n))$ if for all sufficiently large $n$, $T(n)$ is bounded above by a constant multiple of $f(n)$

**Example**: $T(n)$ is $O(n^2)$ if for all large enough $n$, $T(n)$ is bounded above by a constant multiple of $f(n) = n^2$

- Gist: for high values of $n$, does $f(n)$ "grow at least as quickly"?
  - $cf(n) = cn^2$? (we can pick any number for $c$)
  - $T(n) = 202 + 2n + 2n^2$ ?

# Big-O: bound on runtime growth rate

Let $T(n)$ be the number of steps taken for input size $n$:

- Example: $T(n) = 202 + 2n + 2n^2$

**Definition**: $T(n)$ is $O(f(n))$ if for all sufficiently large $n$, $T(n)$ is bounded above by a constant multiple of $f(n)$

**Example**: $T(n)$ is $O(n^2)$ if for all large enough $n$, $T(n)$ is bounded above by a constant multiple of $f(n) = n^2$

- Gist: for high values of $n$, does $f(n)$ "grow at least as quickly"?
  - $cf(n) = cn^2$? (we can pick any number for $c$)
  - $T(n) = 202 + 2n + 2n^2$ ?
- What if $f(n) = n$, that is $O(n)$?

# Big O tells us how fast the algorithm is

**Fast algorithm**: worst-case running time grows slowly with input size

- $O(1)$: constant running time — primitive operations
- $O(\log n)$: logarithmic running time
- $O(n)$: linear running time — linear search
- $O(n \log n)$: log-linear time
- $O(n^c)$: polynomial running time
- $O(c^n)$: exponential running time
- $O(n!)$: factorial running time

# Binary search on sorted list

**Algorithm** for finding $x$ in sorted list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search on left half of L
  - Otherwise repeat search on right half

# Binary search on sorted list

**Algorithm** for finding $x$ in sorted list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search on left half of L
  - Otherwise repeat search on right half

Find number 24 in a list $L = [9, 24, 32, 56, 57, 59, 61, 99]$

# Binary search on sorted list

**Algorithm** for finding $x$ in sorted list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search on left half of L
  - Otherwise repeat search on right half

Find number 24 in a list $L = [9, 24, 32, 56, 57, 59, 61, 99]$

First iteration

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 24 | 32 | **56** | 57 | 59 | 61 | 99 |

| | | |
|---|---|---|
| 9 | 24 | 32 |

| | | | | |
|---|---|---|---|---|
| 56 | 57 | 59 | 61 | 99 |

$L[i] = 56 > 24$ —> discard right half and search left half

# Binary search on sorted list

**Algorithm** for finding $x$ in sorted list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search on left half of L
  - Otherwise repeat search on right half

Find number 24 in a list $L = [9, 24, 32, 56, 57, 59, 61, 99]$

First iteration

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 24 | 32 | **56** | 57 | 59 | 61 | 99 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | 24 | 32 | 56 | 57 | 59 | 61 | 99 |

$L[i] = 56 > 24$ —> discard right half and search left half

Second iteration

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | **24** | 32 | 56 | 57 | 59 | 61 | 99 |

$L[i] = 24$ —> return True

# Binary search complexity

**Algorithm** for finding $x$ in list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search in left half of L
  - Otherwise repeat search in right half

**Complexity** = **# of iterations** * **Constant time per iteration**

# Binary search complexity

**Algorithm** for finding $x$ in list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search in left half of L
  - Otherwise repeat search in right half

**Complexity** = **# of iterations** * **Constant time per iteration**

But **how many** iterations?

# Binary search complexity

**Algorithm** for finding $x$ in list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search in left half of L
  - Otherwise repeat search in right half

**Complexity** = **# of iterations** * **Constant time per iteration**

But **how many** iterations?

- How many times can you split $n$ items in half?

# Binary search complexity

**Algorithm** for finding $x$ in list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search in left half of L
  - Otherwise repeat search in right half

**Complexity** = **# of iterations** * **Constant time per iteration**

But **how many** iterations?

- How many times can you split $n$ items in half?
- $\log_2(n)$ (but base of logarithm does not matter for big-O)

# Binary search complexity

**Algorithm** for finding $x$ in list $L$:

- Pick an index $i$ roughly dividing $L$ in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, repeat search in left half of L
  - Otherwise repeat search in right half

**Complexity** = **# of iterations** * **Constant time per iteration**

But **how many** iterations?

- How many times can you split $n$ items in half?
- $\log_2(n)$ (but base of logarithm does not matter for big-O)
- Complexity $O(\log n)$!

# Complexity matters

You're planning a trip around the world visiting 10 cities.
What's the cheapest route?

Check all alternative routes?

# Complexity matters

You're planning a trip around the world visiting 10 cities. What's the cheapest route?

Check all alternative routes?

- There are $10 * 9 * 8 * ... * 2 * 1 = 3628800$ possible routes
- Factorial complexity $O(n!)$
- Travelling salesperson problem

# Review

Measuring algorithm time complexity:

- Number of basic steps taken
- Worst-case analysis
- Focus on large inputs

**Workshop after the break**

- Big O practice
- Play with search algorithms

# Workshop

## Workshop zip file on the Hub

- HTML instructions
- At some point, you'll need the `.py`-file with skeleton code (open in Spyder)