Yasin Ceran

A Gentle Introduction to Algorithms

HW
ICW
Quiz

Midterm
Final

Presented by Yasin Ceran

August 19, 2025

Motivation

This is an introductory class about algorithms and their complexity. I intend this class to be quite practical as it may be one of the most important classes for your future career. I would like to give three arguments to substantiate this claim.

1. Five years after their students have graduated, Stanford University regularly asks their former students to rank those classes that were the most useful for their professional career. Together with programming and databases, the class on algorithms consistently ranks highest.
2. On Quora, the answers to the question " What are the 5 most important CS courses that every computer science student must take?" consistently list the class algorithms and data structures among those courses that are most valuable for a professional career.
3. The practical importance of the topic of this class can also be seen by the availability of book titles like "Algorithms for Interviews" **?** or the Google job interview questions.

Overview I-Complexity

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

- When we discuss algorithms it is only natural that we also implement these algorithms. To this end, we need a programming language. I will use *Python* to implement the algorithms presented in this lecture.

- Complexity of algorithms

  In general, in order to solve a given problem it is not enough to develop an algorithm that implements a function f computing the value $f(x)$ for a given argument x. It is also important that the computation of $f(x)$ does not consume too much time, memory, or energy. Hence, we have to develop efficient algorithms. In order to be able to discuss the concept of efficiency we need to discuss the growth rate of functions. This notation is useful to abstract from unimportant details when discussing the runtime of algorithms.

Overview II-Recurrence

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

- Recurrence Relations

  The notion of a recurrence relation is the discrete analogue of the notion of a differential equation. For example, the equation

  $$a_{n+2} = a_{n+1} + a_n$$

  is a recurrence relation. Together with the initial values $a_0 = 0$ and $a_1 = 1$, this equation defines a sequence of natural numbers. The numbers a_n for $n = 1, \cdots, 8$ are given as follows:

  $a_0 = 0,\ a_1 = 1,\ a_2 = 1,\ a_3 = 2,\ a_4 = 3,\ a_5 = 5,\ a_6 = 8,\ a_7 = 13,\ a_8 = 21.$

  Recurrence relations occur naturally when analysing the runtime of algorithms.

Overview III-Sorting Algorithms

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

- Sorting algorithms

  Sorting algorithm are among those algorithms that are most frequently used in practice. Furthermore, these algorithms are both easy to understand and easy to analyse. Therefore, we start our discussion of algorithms and their complexity with these algorithms. In this lecture, we discuss the following sorting algorithms:

  1. insertion sort,
  2. merge sort,
  3. quick sort,
  4. treesort, and
  5. heapsort.

Overview IV-Dictionaries and Sets

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

- Dictionaries and sets

  A dictionary is a data structures that can be used to implement a function on a finite domain. For example, a telephone book can be viewed as a function mapping names to telephone numbers. Most high level programming languages provide dictionaries as basic data structures. We discuss various data structures that can be used to implement dictionaries efficiently. These data structures can also be used to implement sets.

Overview V-Priority Queues

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

- Priority queues

  A priority queue is a data structure that can best be described as a sorted list that remains sorted when elements are inserted or removed. Some graph theoretical algorithms use priority queues as one of their basic building blocks. Therefore, our discussion of graph theory is preceded by a chapter on priority queues.

Algorithms and Programs

- This is a lecture on algorithms, not on programming.
- An algorithm is an abstract concept to solve a given problem. In contrast, a program is a concrete implementation of an algorithm.
- In order to implement an algorithm as a program we have to cover every detail, be it trivial or not. On the other hand, to specify an algorithm it is often sufficient to describe just the interesting aspects.

Example: An algorithm to solve birthday matching

- Maintain a record of names and birthdays (initially empty)
- Interview each student in some order
    - If birthday exists in record, return found pair!
    - Else add name and birthday to record
- Return None if last student interviewed without success

Desirable Properties of Algorithms

Before we start with our discussion of algorithms we should think about our goals when designing algorithms.

a. Algorithms have to be correct.
b. Algorithms should be efficient with respect to both computing time, memory, and, last not least, energy consumption.
c. Algorithms should be simple.

Check Your Understanding

a. What reasons do I have to assume that this lecture is more important than other lectures that you will be attending in this semester.

b. What are the most important features of an algorithm?

c. What is the difference between an algorithm and a program?

Correctness and Induction

- Programs/algorithms have fixed size, so how to prove correct?
- For small inputs, can use case analysis
- For arbitrarily large inputs, algorithm must be recursive or loop in some way
- Must use induction (why recursion is such a key concept in computer science)
- Example: Proof of correctness of birthday matching algorithm
  - Induct on k: the number of students in record
  - **Hypothesis**: if first k contain match, returns match before interviewing student $k+1$
  - **Base case**: $k = 0$, first k contains no match
  - Assume for induction hypothesis holds for $k = k'$, and consider $k = k' + 1$
  - If first k' contains a match, already returned a match by induction
  - Else first k' do not have match, so if first $k' + 1$ has match, match contains $k' + 1$
  - Then algorithm checks directly whether birthday of student $k' + 1$ exists in first k'

An Approach to Measure Complexity

Sometimes it is necessary to have a precise understanding of the complexity of an algorithm. In order to obtain this understanding we could proceed as follows:

1. We implement the algorithm in assembly language.
2. We count how many additions, multiplications, assignments, etc. are needed for an input of a given size. Additionally, we have to count all storage accesses.
3. We look up the amount of time that is needed for the different operations in the processor handbook.
4. Using the information discovered in the previous two steps we predict the running time of our algorithm for given input.

An Analogy

Imagine the following scenario: You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to get the file to your friend as fast as possible. How should you send it?

- Most people's first thought would be email, FTP, or some other means of electronic transfer. That thought is reasonable, but only half correct. If it's a small file, you're certainly right, it would take 5-10 hours to get to an airport, hop on a flight, and then deliver it to your friend.
- But what if the file were really, really large? Is it possible that it's faster to physically deliver it via plane? Yes, actually it is, A one-terabyte file could take more than a day to transfer electronically. It would be much faster to just fly it across the country. If your file is that urgent (and cost isn't an issue), you might just want to do that.
- What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

Asymptotic Notation

- $\mathcal{O}$ Notation characterizes an upper bound on the asymptotic behavior of a function.

  Consider, for example, the function $7n^3 + 100n^2 - 20n + 6$. Its highest-order term is $7n^3$, and so we say that this function's rate of growth is n^3. Because this function grows no faster than n^3, we can write that it is $\mathcal{O}(n^3)$

- Time estimate below based on one operation per cycle on a 1 GHz single-core machine

- Particles in universe estimated $< 10^{100}$

Input	Constant	Logarithmic	Linear	Log-linear	Quadratic	Polynomial	Exponential
n	$\mathcal{O}(1)$	$\mathcal{O}(logn)$	$\mathcal{O}(n)$	$\mathcal{O}(nlogn)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^c)$	$2^{\mathcal{O}(n^c)}$
1000	1	≈ 10	1000	$\approx 10,000$	1,000,000	1000^c	$2^{1000} \approx 10^{301}$
Time	1 ns	10 ns	1 μs	10 μs	1ms	10^{3c-9}s	10^{281}millenia

Table: Asymptotic Time Table

Model of Computation

- Specification for what operations on the machine can be performed in $\mathcal{O}(1)$ time
- Model in this class is called the Word-RAM
- **Machine word**: block of w bits (w is word size of a w-bit Word-RAM)
- **Memory**: Addressable sequence of machine words
- **Processor** supports many **constant time** operations on a $\mathcal{O}(1)$ number of words (**integers**):
  - **integer** arithmetic: $(+, -, *, //, \%)$
  - **logical** operators: $(\&\&, ||, !, ==, >, <, <=, >=)$
  - **bitwise** arithmetic: $(\&, |,\rangle >, <<)$
  - Given a word a, can **read** word at address a, **write** word to address a
- Memory address must be able to access every place in memory
  - Requirement: $w \geq \#$ bits to represent largest memory address, i.e., $log_2 n$
  - 32-bit words $\rightarrow$ max $\approx$ 4 GB memory,
  - 64-bit words $\rightarrow$ max $\approx$ 16 exabytes of memory.

Data Structure

- A data structure is a way to store non-constant data, that supports a set of operations.
- A collection of operations is called an interface.
  - Sequence: Extrinsic order to items (first, last, nth).
  - Set: Intrinsic order to items (queries based on item keys).
- Data structures may implement the same interface with different performance.
- **Example: Static Array** - fixed width slots, fixed length, static sequence interface.
  - `StaticArray(n)`: allocate static array of size n initialized to 0 in $\mathcal{O}(n)$ time.
  - `StaticArray.get_at(i)`: return word stored at array index i in $\mathcal{O}(1)$ time.
  - `StaticArray.set_at(i, x)`: write word x to array index i in $\mathcal{O}(1)$ time.
- Stored word can hold the address of a larger object.
- Like Python `tuple` plus `set_at(i, x)`, Python `list` is a dynamic array.

```python
class StaticArray:
    def __init__(self, n):
        self.data = [None] * n
    def get_at(self, i):
        if not (0 <= i < len(self.data)): raise IndexError
        return self.data[i]
    def set_at(self, i, x):
        if not (0 <= i < len(self.data)): raise IndexError
        self.data[i] = x

def birthday_match(students):
    '''
    Find a pair of students with the same birthday
    Input:  tuple of student (name, bday) tuples
    Output: tuple of student names or None
    '''
    n = len(students)                                    # O(1)
    record = StaticArray(n)                              # O(n)
    for k in range(n):                                   # n
        (name1, bday1) = students[k]                     # O(1)
        for i in range(k):                               # k check if in record
            (name2, bday2) = record.get_at(i)            # O(1)
            if bday1 == bday2:                           # O(1)
                return (name1, name2)                    # O(1)
        record.set_at(k, (name1, bday1))                 # O(1)
    return None                                          # O(1)
```

- Two loops: outer $k \in \{0, \ldots, n-1\}$, inner is $i \in \{, \ldots, k\}$
- Running time is $\theta(n) + \sum_{k=0}^{n-1}(\theta(1) + k * \theta(1)) = \theta(n^2)$

Example 1: Single Loop

```python
for i in range(n):
    print(a)   # Freq: 1
```

- **Time**: Frequency $n \cdot 1 = n$, so $O(n)$.
- **Space**: i, a (O(1) auxiliary).

$O(n)$ $\quad$ $O(n)$

$O(\cancel{3}\underline{}\, n)$ $\quad$ $O(n^2)$

Example 2: Nested Loops

```python
for i in range(n):
    for j in range(n):
        print(a)   # Freq: 1
```

- **Time**: Frequency $n \cdot n \cdot 1 = n^2$, so $O(n^2)$.
- **Space**: i, j, a (O(1) auxiliary).

n n

3 n

$O(n^2)$

$\cancel{O(3)}$ $O(1)$

Example 3: Dependent Nested Loops

```python
for i in range(n):
    for j in range(i):
        print(a)   # Freq: 1
```

- **Time**: Frequency $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$, so $O(n^2)$.
- **Space**: i, j, a (O(1) auxiliary).

$O(n^2)$?

$i = n$

$i = 0 \quad j = 0$

$i = 1 \quad j = 1$

$i = 2 \quad 2$

$+ \; i = n \quad n$

$1 + 2 + 3 + \quad + n$

$1 + 2 + \quad + n$

$n + n-1 + \quad + 1$

$(n+1) + (n+1) \quad + (n+1)$

$\frac{n(n+1)}{2}$

$O\left(\frac{n^2+n}{2}\right) \sim O(n^2)$

$\frac{n^2}{2} \gg \frac{n}{2}$

$O(n^2)$

Example 4: Linear While Loop

```python
p = 0
i = 1
while p < n:
    p = p + i   # Freq: 1
    i = i + 1   # Freq: 1
```

- **Time**: Iterations $\approx \sqrt{2n}$, so $O(\sqrt{n})$.
- **Space**: p, i ($O(1)$ auxiliary).

	$p_0 = 0$	$i_0 = 1$
$j=1$	$p_1 = 0 + i_0$	$i_1 = 2$
$j=2$	$p_2 = i_0 + i_1$	$i_2 = 3$
$j=3$	$p_3 = i_0 + i_1 + i_2$	$i_3 = 4$
$j=k$	p_k $i_0 + i_1 + \ldots$	$+ i_k$

$p_k = \underbrace{1 + 2 + \ldots + k}_{} = n$

$\frac{k(k+1)}{2} = n$

$k^2 + \cancel{k} = 2n$

$k = \sqrt{2n}$

$k \sim \sqrt{n}$

Example 5: Logarithmic While Loop

```python
i = 1
while i < n:
    print(a)   # Freq: 1
    i = i * 2   # Freq: 1
```

- **Time**: Iterations $\lceil \log_2 n \rceil$, so $O(\log n)$.
- **Space**: i, a (O(1) auxiliary).

$i = 1$

$J=1 \quad i_1 = 2 \quad a$

$J=2 \quad i_2 = 2 \cdot 2 \quad a$

$J=3 \quad i_3 = 2 \cdot 2 \cdot 2 \quad a$

$J=k, \; i_k = 2 \cdot 2 \cdots 2 = 2^k$

$\log(2^k) = \log(n)$

$\boxed{k = \log n}$

Example 6: Reverse Logarithmic Loop

```python
i = n
while i > 1:
    print(a)   # Freq: 1
    i = i // 2   # Freq: 1
```

- **Time**: Iterations $\lfloor \log_2 n \rfloor + 1$, so $O(\log n)$.
- **Space**: i, a (O(1) auxiliary).

n

$n/2$

$n/4$

$n/2^k = 1$

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = k = \cancel{\log 2}$

Example 7: Square Root Loop

```python
i = 0
while i * i < n:
    print(a)  # Freq: 1
    i = i + 1  # Freq: 1
```

- **Time**: Iterations $\lfloor\sqrt{n}\rfloor$, so $O(\sqrt{n})$.
- **Space**: i, a (O(1) auxiliary).

$i=0$

$i=1$	a
$i=2$	a
$a=k$	a

n

$k^2 = n$

$k = \sqrt{n}$

Example 8: Logarithmic Assignment

```python
p = 0
i = 1
while i < n:
    p = p + 1   # Freq: 1
    i = i * 2   # Freq: 1
```

- **Time**: Iterations $\lceil \log_2 n \rceil$, so $O(\log n)$.
- **Space**: p, i (O(1) auxiliary).

$p_0 = 0$, $i_0 = 1$

$j = 1$ $p_1 = 1$ | $i_1 = 2 \, i_0$

$j = 2$ $p_2 = 2$ | $i_2 \; 2 \; 2 \; i_0$

$j = k$ $p_j = k$ | $i_j = 2^k \; i_0 = 2^k$

?

$i < n$

$2^k = n$

$k = \log_2 n$

Example 9: Logarithmic Inner Loop

```python
j = 1
while j < n:
    print(a)   # Freq: 1
    j = j * 2   # Freq: 1
```

400

- **Time**: Iterations $\lceil \log_2 n \rceil$, so $O(\log n)$.
- **Space**: j, a (O(1) auxiliary).

```
p=0
i=1
while i<n:
    p=p+1
    i=i*2
```

p=k

log n

log(log(n))

```
j=1
while j<p:
    print('a')
    j=j*2
```

log p

Example 10: Linear-Logarithmic Loop

```python
for i in range(n):
    j = 1
    while j < n:
        print(a)   # Freq: 1
        j = j * 2   # Freq: 1
```

$\log n$

$n \log n$

- **Time**: Frequency $n \cdot \log_2 n$, so $O(n \log n)$.
- **Space**: i, j, a (O(1) auxiliary).

$\log 1 + \log 2 + \log(n!)$

$2^k = n$

$k = \log n$

$\log a + \log b = \log(a b)$

$\log(1\ 2 \ldots n) + \log n$

$n \log n$ $\quad \log n + \log n + \log n + \log n$ $\quad \log(n!)$ $\quad \log 1 + \log 2 + \ldots + \log n$

Example 11: Matrix Multiplication

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]   # Freq: 1
```

- **Time**: Frequency n^3, so $O(n^3)$.
- **Space**: i, j, k (O(1) auxiliary); output array C (O(n^2) if considered).

Example 12: Bubble Sort

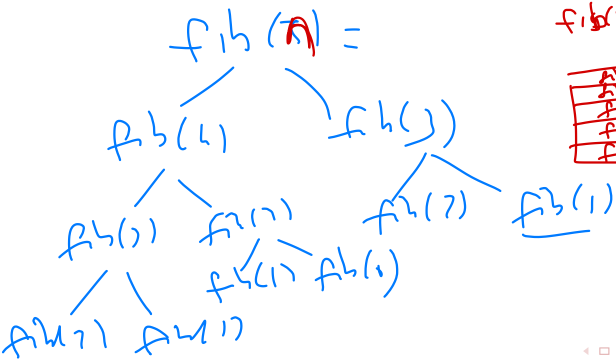
```python
for i in range(n-1):
    for j in range(n-i-1):
        if A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]  # Freq: 1
```

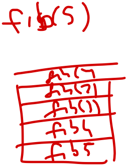
- **Time**: Frequency $\frac{n(n-1)}{2}$, so $O(n^2)$.
- **Space**: i, j (O(1) auxiliary); in-place sorting (no extra array).

Example 13: Recursive Fibonacci

```python
def fib(n):
    if n <= 1:
        return n   # Freq: 1
    return fib(n-1) + fib(n-2)   # 2 recursive calls
```

- **Time**: Calls $\approx 2^n$, so $O(2^n)$.
- **Space**: Recursion stack depth $\approx n$, so $O(n)$ auxiliary.





n

n-1

n 2

Example 14: Linear Search

```
def linear_search(A, x):
    for i in range(len(A)):
        if A[i] == x:   # Freq: 1
            return i
    return -1
```

- **Time**: Frequency n, so $O(n)$.
- **Space**: i, x (O(1) auxiliary).

Example 15: Binary Search

```
def binary_search(A, x):
    left, right = 0, len(A)-1
    while left <= right:
        mid = (left + right) // 2  # Freq: 1
        if A[mid] == x:
            return mid
        elif A[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

- **Time**: Iterations $\log_2 n$, so $O(\log n)$.
- **Space**: *left*, *right*, *mid*, *x* (O(1) auxiliary).

Example 16: Sum of Pairs

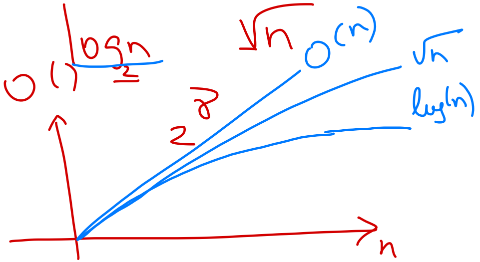
```python
def find_pair_sum(A, target):
    for i in range(len(A)):
        for j in range(i+1, len(A)):
            if A[i] + A[j] == target:  # Freq: 1
                return (i, j)
    return None
```

- **Time**: Frequency $\frac{n(n-1)}{2}$, so $O(n^2)$.
- **Space**: i, j, $target$ (O(1) auxiliary).

Asymptotic Notations

- **Big O ($\mathcal{O}$)**: Upper bound (worst-case). $f(n) \leq c \cdot g(n)$ for $n \geq k$, $c > 0$.
- **Omega (Ω)**: Lower bound (best-case).
- **Theta (Θ)**: Tight bound (average).
- Order: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$.

n^2 2^n



Complexity Comparison

Input	$O(1)$	$O(\log n)$	$O(\sqrt{n})$	$O(n)$	$O(n \log n)$	$O(n^2)$
$n = 1000$	1	≈ 10	≈ 32	1000	$\approx 10,000$	1,000,000
Time (1 GHz)	1 ns	10 ns	32 ns	1 μs	10 μs	1 ms

Table: Time Complexity for $n = 1000$

Any Questions?

Any Questions?

Thank you for your attention!
Please share your questions or feedback.