Lecture 1

Introduction to algorithms and data structures
Today’s plan

1. What is an algorithm
2. word RAM model of computation, complexity of an algorithm
3. Elementary data structures: array, list, queue, stack
4. Dynamic array
What is an algorithm

Muhammad ibn Musa al-Khwarizmi, c. 780-850
An algorithm is a sequence of elementary operations that transforms the input into the output.
Algorithms

Primality test

\[ f(n) = \begin{cases} 
1, & \text{if } n \text{ is prime} \\
0, & \text{otherwise} 
\end{cases} \]
Applications of prime numbers

The **RSA encryption system** is one of the most widely used methods of securely transmitting information.

It relies on large prime numbers, that are extremely hard to find.

**Open Problem 1: Mersenne prime**

Find a new prime of form $2^n - 1$ that has at most $100,000,000$ digits. You can **win** USD $3000!
Algorithms

• **Ideal algorithm:** simple to implement, fast, uses little memory.

• To design such algorithms, we need a simple, realistic **model of computation**. The model should not depend on programming languages.
word RAM model

Charles Babbage. The Analytical Engine.
word RAM model

Elementary operations: basic arithmetic and bitwise operations on registers, conditionals (if-then), goto, copying words between registers and main memory, malloc (add an extra memory word), halt

NB! $w \approx \log n$, where $n$ is the input size
Complexity of an algorithm

Two main resources: time and space

- $Time(n) = \text{the maximum number of elementary operations used for an input of size } n$
- $Space(n) = \text{the maximum number of memory words used for an input of size } n$
Primality test

for \( i \leftarrow 2 \) to \( n - 1 \) do

\[ \text{mult} \leftarrow i \]

while \( \text{mult} < n \) do

\[ \text{mult} \leftarrow \text{mult} + i \]

if \( \text{mult} = n \) then

return 0

return 1

\[ f(n) = \begin{cases} 
1, & \text{if } n \text{ is prime} \\
0, & \text{otherwise} 
\end{cases} \]

Space: \( c_1 \) (number of registers + 3)

Time: \( \leq c_2 \cdot \sum_{i=2}^{n-2} \frac{n}{i} \leq c_2 \cdot n \ln n \)
Complexity: asymptotic

- Often, it is hard to compute the constants exactly

- We will mainly study the asymptotic growth
\textbf{$O()$, $\Omega()$, $\Theta()$ notation}

Let $f(n), g(n) \in \mathbb{N} \rightarrow \mathbb{R}^+$

- We say that $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if
  \[
  \exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ : \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)
  \]

- We say that $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if
  \[
  \exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ : \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)
  \]

- We say that $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) if
  \[
  \exists n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_+ : \forall n \geq n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
  \]
Primality test

for $i \leftarrow 2$ to $n - 1$ do

    $mult \leftarrow i$

    while $mult < n$ do

        $mult \leftarrow mult + i$

    if $mult = n$ then

        return 0

    return 1

$f(n) = \begin{cases} 
1, & \text{if } n \text{ is prime} \\
0, & \text{otherwise} 
\end{cases}$

Space: $c_1 = O(1)$ (number of registers + 3)

Time: $\leq c_2 \cdot \sum_{i=2}^{n-2} \frac{n}{i} \leq c_2 \cdot n \ln n = O(n \log n)$
Asymptotic complexity vs real efficiency

- Asymptotic complexity **matters.** A $\Theta(n)$-time algorithm is slower than a $\Theta(\log n)$-time algorithm if $n$ is large enough.

- An algorithm with time complexity $\Theta(n^2)$ can be faster than an algorithm with time complexity $\Theta(n)$ for all reasonable values of $n$ if the hidden constant in $\Theta(n)$ is too large.

- Sometimes, the time (or the space) complexity is high, but the **inputs** on which this complexity is reached are very rare in practice.

- The **programming language** used has a real impact on performance.
Linear vs binary search in a sorted sequence

Given a sequence of integers $a_1 \leq a_2 \leq \ldots \leq a_n$ and an integer $x$, return 1 if $x = a_i$ for some $i$, and 0 otherwise.

- Linear search algorithm: $\Theta(n)$ time
- Binary search algorithm: $\Theta(\log n)$ time
Linear vs binary search in a sorted sequence
Linear search in a sorted sequence: Python vs C
Elementary data structures

Rivers and Tides: Andy Goldsworthy’s project by T. Riedelsheimer
Data structures

• Imagine that we have a collection of objects, i.e. a database of DNA sequences

• We must be able to quickly extract all sequences that have a certain property (i.e. contain a certain gene)

• Scanning the whole database each time is too expensive

• Solution: preprocess the database and store certain information about it in an organised form

• This form is called a data structure
Data structures

We care about:

• the space the data structure occupies
• construction time
• query time
• update time
Elementary data structures

- array (tableau)
- linked list (liste chaînée)
- queue (file)
- stack (pile)
Array, list, queue, stack

Data structures storing a sequence $e_0, e_2, \ldots, e_{n-1}$ of elements (e.g., integers, floating-point numbers, complex objects, etc.)

Diverse specifications of such a structure, allowing different operations, with different efficiency:

- **Random access**: given $i$, access $e_i$
- **Access** the first element ($e_0$), the last element ($e_{n-1}$)
- **Insertion** at the beginning (before $e_0$), at a random position (between $e_i$ and $e_{i+1}$), at the end (after $e_{n-1}$)
- **Deletion** of the first element ($e_0$), of a random element ($e_i$), of the last element ($e_{n-1}$)
Array

- Contiguous memory area of a fixed size, pre-allocated
- Random access in $O(1)$ time
- Insertion, deletion impossible
- Corresponds to **classic arrays** of programming languages:
  - bracketed arrays in C or Java
  - std::array in C++ 2011
  - numpy.array in Python
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

**Linear search**

```
for i ← 0 to n − 1 do
    if $e_i = x$ then
        return 1
    return 0
```

**Time** = $O(n)$

**Space** = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

Linear search

\[
\begin{array}{c}
\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
\quad \text{if } e_i = x \text{ then} \\
\qquad \text{return 1} \\
\text{return 0}
\end{array}
\]

**Time** = $O(n)$

**Space** = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

**Linear search**

for $i \leftarrow 0$ to $n - 1$ do
  if $e_i = x$ then
    return 1
  return 0

Time = $O(n)$
Space = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

Linear search

```
for i ← 0 to n − 1 do
  if $e_i = x$ then
    return 1
  return 0
```

Time = $O(n)$
Space = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

```
for i ← 0 to n − 1 do
  if $e_i = x$ then
    return 1
  return 0
```

Time = $O(n)$
Space = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

**Binary search**

\[
l \leftarrow 0, r \leftarrow n - 1
\]

\[
\text{while } l \leq r \text{ do}
\]

\[
m \leftarrow \left\lfloor \frac{l + r}{2} \right\rfloor
\]

\[
\text{if } e_m = x \text{ then}
\]

\[
\text{return 1}
\]

\[
\text{else if } e_m < x
\]

\[
l = m + 1
\]

\[
r = m - 1
\]

\[
\text{else if } e_m > x
\]

\[
\text{return 0}
\]

\[\sqrt{3}\]?

**Time** = $O(\log n)$

**Space** = $O(1)$

At each step, the search area shrinks by a factor of at least two.
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise

Binary search

\[
\begin{array}{c}
\text{Binary search} \\
\begin{array}{cccc}
1 & 2 & 2 & 5 & 5 \\
\end{array} \\
\begin{array}{c}
l \\
r \\
\end{array} \\
\end{array}
\]

\[\sqrt{3}?\]

At each step, the search area shrinks by at least two times.

**Time** = $O(\log n)$

**Space** = $O(1)$
Linear and binary search in a sorted array

Given an integer $x$, return 1 if $e_i = x$, and 0 otherwise.

**Binary search**

1. $l \leftarrow 0$, $r \leftarrow n - 1$
2. while $l \leq r$ do
   1. $m \leftarrow \lfloor \frac{l + r}{2} \rfloor$
   2. if $e_m = x$ then
      1. return 1
   3. else if $e_m < x$
      1. $l = m + 1$
   4. else if $e_m > x$
      1. $r = m - 1$
   5. return 0

**Time** = $O(\log n)$

**Space** = $O(1)$

At each step, the search area shrinks by at least two times.
Doubly linked list

- The head and the tail can be accessed in $O(1)$ time
- Random access in $O(n)$ time
- Insertion at head / deletion in $O(1)$ time
- In programming languages: std::list in C++
Doubly linked list

Insert($L$, $x$)

\[
\begin{align*}
  x\.next & \leftarrow L\.head \\
  \text{if } L\.head \neq NIL \text{ then} & \\
  L\.head\.prev & = x \\
  L\.head & \leftarrow x \\
  x\.prev & = NIL
\end{align*}
\]

Delete($L$, $x$)

\[
\begin{align*}
  & \text{if } x\.prev \neq NIL \text{ then} \\
  & x\.prev\.next \leftarrow x\.next \\
  \text{else } L\.head & \leftarrow x\.next \\
  & \text{if } x\.next \neq NIL \\
  & x\.next\.prev \leftarrow x\.prev
\end{align*}
\]
Stack (or LIFO for last-in-first-out)

- Abstract data structure that can be implemented in different ways, e.g. with a linked list
- Access to the top element (peek) in $O(1)$ time
- Insertion to the top (push) in $O(1)$ time
- Deletion of the top element (pop) in $O(1)$ time
- In programming languages:
  - `std::stack` in C++
  - not explicit in Python, but standard lists can be used
Stack (or LIFO for last-in-first-out)

Open problem 2: Optimum Stack Generation
Given a finite alphabet $\Sigma$ and a string $X \in \Sigma^n$. Find a shortest sequence of stack operations push, pop, peek that prints out $X$. You must start and finish with an empty stack.

The current best algorithm by Bringmann et al. solves the problem in $\tilde{O}(n^{2.8603})$ time. Can it be done faster?
Queue (or FIFO for first-in-first-out)

- Abstract data structure that can be implemented in different ways, e.g., with a doubly linked list
- Access to the elements in the back / front in $O(1)$ time
- Deletion from the front (dequeue) and insertion to the back (enqueue) in $O(1)$ time

**In programming languages:**
- std::queue in C++
- not explicit in Python, collections.deque can be used
Dynamic array
Dynamic array

- We saw that an array allows $O(1)$-time random access, but no insertions.

- A list allows insertions in $O(1)$ time, but random access can take $\Omega(n)$ time.

- We want to develop a data structure that allows both fast random access and insertions.
Dynamic array

- Pointer towards a classic array of capacity $c +$ the number of elements $n \leq c$ stored
- Random access in $O(1)$ time
- Insertion at the end in amortised $O(1)$ time (see further)
- **In programming languages:**
  - std::vector in C++
  - lists and array in standard Python, or numpy.ndarray
Dynamic array: insertion at the end

**Input:** array $A$ of capacity $c$ and size $n$, element $x$

**Output:** array $A$ of size $n + 1$ with $A[n] = x$

if $n = c$ then
   allocate new array $A'$ of size $\max\{2c, 1\}$
   for $i \leftarrow 0$ to $n - 1$ do
      $A'[i] \leftarrow A[i]$
   deallocate array pointed by $A$ and make $A$ point to $A'$
   $c \leftarrow \max\{2c, 1\}$
   $A[n] \leftarrow x$
   $n \leftarrow n + 1$
Dynamic array: insertion at the end

- In the worst case, we can spend $\Theta(n)$ time to insert an element (when $n = c$)
- However, any $n$ insertions take $O(n)$ time
- We say that each insertion takes $O(1)$ amortised time
- **NB!** We measure the average performance of each insertion in the worst case
Potential method: idea

Consider a sequence of \( n \) operations \( o_1, o_2, \ldots, o_n \) that take time \( t_1, t_2, \ldots, t_n \)

Let \( D_0 \) be the initial data structure and \( D_i \) be the data structure after applying the \( i \)-th operation \( o_i \) to \( D_{i-1} \)

Associate a potential \( \Phi(D) \) (real number) to a data structure \( D \)

Define \( \hat{t}_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) \)

Then \( \sum_{i=1}^{n} \hat{t}_i = \sum_{i=1}^{n} t_i + (\Phi(D_n) - \Phi(D_0)) \)

If \( \Phi(D_n) \geq \Phi(D_0) \), then \( \sum_{i=1}^{n} \hat{t}_i \geq \sum_{i=1}^{n} t_i \)
Potential method: Dynamic array

Time of the $i$-th insertion $t_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$

Potential $\Phi(D) = 2 \times (\text{number of elements}) - \text{capacity}$

$\hat{t}_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = 3$ (see blackboard)

Therefore, $\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} \hat{t}_i = 3n = O(n)$
Insertion in a std::vector in C++

![Graph showing cumulative time vs. number of inserted elements]

- Cumulated time (s) on the y-axis.
- Number of inserted elements on the x-axis, marked with powers of two (2^20, 2^21, 2^22, 2^23, 2^23 * 10^7).

The graph illustrates the time taken to insert elements into a std::vector as the number of elements increases.
Next lecture

- Divide and conquer
- Analysis of recursive algorithms
- Master theorem
- Fast multiplication of polynomials
- Discrete Fourier transform

(Thank you!)