Outline

- How can we measure and compare algorithms meaningfully?
  - an algorithm will run at different speeds on different computers
- $O$ notation.
- Complexity types.
  - Worst-case vs average-case
  - Real-time vs amortized-time

Selection sort algorithm

```c
void selection_sort(int array[], int size) {
    // Βρίσκουμε το μικρότερο στοιχείο του πίνακα, το τοποθετούμε στη θ
    // και συνεχίζουμε με τον ίδιο τρόπο στον υπόλοιπο πίνακα.
    for (int i = 0; i < size; i++) {
        int min_position = i;
        for (int j = i; j < size; j++)
            if (array[j] < array[min_position])
                min_position = j;
        // swap των στοιχείων i και min_position
        int temp = array[i];
        array[i] = array[min_position];
        a[min_position] = temp;
    }
}
```

Running Time

- Array of 2000 integers
- Computers A, B, …, E are progressively faster.
  - The algorithm runs faster on faster computers.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer A</td>
<td>51.915</td>
</tr>
<tr>
<td>Computer B</td>
<td>11.508</td>
</tr>
<tr>
<td>Computer C</td>
<td>2.382</td>
</tr>
<tr>
<td>Computer D</td>
<td>0.431</td>
</tr>
<tr>
<td>Computer E</td>
<td>0.087</td>
</tr>
</tbody>
</table>
**More Measurements**

- What about different programming languages?
- Or different compilers?
- Can we say whether algorithm A is better than B?

**A more meaningful criterion**

- Algorithms **consume resources**: e.g. time and space
- In some fashion that depends on the **size of the problem** solved
  - the bigger the size, the more resources an algorithm consumes
- We usually use \( n \) to denote the size of the problem
  - the **length of a list** that is searched
  - the **number of items** in an array that is sorted
  - etc

**selection_sort running time**

In msecs, on two types of computers

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Home Computer</th>
<th>Desktop Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>12.5</td>
<td>2.8</td>
</tr>
<tr>
<td>250</td>
<td>49.3</td>
<td>11.0</td>
</tr>
<tr>
<td>500</td>
<td>195.8</td>
<td>43.4</td>
</tr>
<tr>
<td>1000</td>
<td>780.3</td>
<td>172.9</td>
</tr>
<tr>
<td>2000</td>
<td>3114.9</td>
<td>690.5</td>
</tr>
</tbody>
</table>

**Curves of the running times**

If we plot these numbers, they lie on the following two curves:

- \( f_1(n) = 0.0007772n^2 + 0.00305n + 0.001 \)
- \( f_2(n) = 0.0001724n^2 + 0.00040n + 0.100 \)
**Discussion**

- The curves have the **quadratic** form \( f(n) = an^2 + bn + c \)
  - difference: they have **different constants** \( a, b, c \)
- Different computer / programming language / compiler:
  - the curve that we get will be of the same form!
- The exact numbers change, but the **shape of the curve** stays the same.

**Complexity classes, \( O \)-notation**

- We say that an algorithm belongs to a **complexity class**
- A class is denoted by \( O(g(n)) \)
  - \( g(n) \) gives the running time as a function of the size \( n \)
  - it describes the **shape** of the running time curve
- For **selection sort** the time complexity is \( O(n^2) \)
  - take the **dominant term** of the expression \( an^2 + bn + c \)
  - throw away the constant coefficient \( a \)

**Why only the dominant term?**

\[
f(n) = an^2 + bn + c
\]

with \( a = 0.0001724, b = 0.0004 \) and \( c = 0.1 \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( f(n) )</th>
<th>( an^2 )</th>
<th>( n^2 ) term as % of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>2.8</td>
<td>2.7</td>
<td>94.7</td>
</tr>
<tr>
<td>250</td>
<td>11.0</td>
<td>10.8</td>
<td>98.2</td>
</tr>
<tr>
<td>500</td>
<td>43.4</td>
<td>43.1</td>
<td>99.3</td>
</tr>
<tr>
<td>1000</td>
<td>172.9</td>
<td>172.4</td>
<td>99.7</td>
</tr>
<tr>
<td>2000</td>
<td>690.5</td>
<td>689.6</td>
<td>99.9</td>
</tr>
</tbody>
</table>

- The lesser term \( bn + c \) **contributes very little**
  - even though \( b, c \) are much larger than \( a \)
  - Thus we can **ignore this lesser term**
- Also: we **ignore the constant** \( a \) in \( an^2 \)
  - It can be thought of as the “time of a single step”
  - It depends on the computer / compiler / etc
  - We are only interested in the shape of the curve
Common complexity classes

<table>
<thead>
<tr>
<th>$O$-notation</th>
<th>Adjective Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Quasi-linear</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
</tr>
<tr>
<td>$O(10^n)$</td>
<td>Exponential</td>
</tr>
<tr>
<td>$O(2^{2^n})$</td>
<td>Doubly exponential</td>
</tr>
</tbody>
</table>

Sample running times for each class

Assume 1 step = 1 μsec.

<table>
<thead>
<tr>
<th>$g(n)$</th>
<th>$n = 2$</th>
<th>$n = 16$</th>
<th>$n = 256$</th>
<th>$n = 1024$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1 μsec</td>
<td>1 μsec</td>
<td>1 μsec</td>
<td>1 μsec</td>
</tr>
<tr>
<td>$\log n$</td>
<td>1 μsec</td>
<td>4 μsec</td>
<td>8 μsec</td>
<td>10 μsec</td>
</tr>
<tr>
<td>$n$</td>
<td>2 μsec</td>
<td>16 μsec</td>
<td>256 μsec</td>
<td>1.02 ms</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>2 μsec</td>
<td>64 μsec</td>
<td>2.05 ms</td>
<td>10.2 ms</td>
</tr>
<tr>
<td>$n^2$</td>
<td>4 μsec</td>
<td>25.6 μsec</td>
<td>65.5 ms</td>
<td>1.05</td>
</tr>
<tr>
<td>$n^3$</td>
<td>8 μsec</td>
<td>4.1 ms</td>
<td>16.8 ms</td>
<td>17.9 min</td>
</tr>
<tr>
<td>$2^n$</td>
<td>4 μsec</td>
<td>65.5 ms</td>
<td>$10^{63}$ years</td>
<td>$10^{297}$ years</td>
</tr>
</tbody>
</table>

The largest problem we can solve in time $T$

Assume 1 step = 1 μsec.

<table>
<thead>
<tr>
<th>$g(n)$</th>
<th>$T = 1$ min</th>
<th>$T = 1$ hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$6 \times 10^7$</td>
<td>$3.6 \times 10^9$</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$2.8 \times 10^6$</td>
<td>$1.3 \times 10^8$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$7.75 \times 10^3$</td>
<td>$6.0 \times 10^4$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$3.91 \times 10^2$</td>
<td>$1.53 \times 10^3$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>$10^n$</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Complexity of well-known algorithms

- Sequential searching of an array: $O(n)$
- Binary searching of a sorted array: $O(\log n)$
- Hashing (under certain conditions): $O(1)$
- Searching using binary search trees: $O(\log n)$
- Selection sort, Insertion sort: $O(n^2)$
- Quick sort, Heap sort, Merge sort: $O(n \log n)$
- Multiplying two square $\times$ matrices: $O(n^3)$
- Traveling salesman, graph coloring: $O(2^n)$
Formal definition of \( O \)-notation

\( f(n) \) is the function giving the actual time of the algorithm. We say that \( f(n) = O(g(n)) \) iff

- there exist two positive constants \( K \) and \( n_0 \)
- such that \( |f(n)| \leq K|g(n)| \quad \forall n \geq n_0 \).

We will not focus on the formal definition in this course.

Intuition

- An algorithm runs in time \( O(g(n)) \) iff it finishes in at most \( g(n) \) steps.
- A "step" is anything that takes constant time
  - a basic operation, eg \( a = b + 3 \)
  - a comparison, eg \( i f(a == 4) \)
  - etc
- Typical way to compute this
  - find an expression \( f(n) \) giving the exact number of steps (or an upper bound)
  - find \( g(n) \) by removing the lesser terms and coefficients (justified by the formal definition)

Example

- An algorithm takes \( f(n) \) number of steps, where
  - \( f(n) = 3 + 6 + 9 + \cdots + 3n \)
- We will show that the algorithm runs in \( O(n^2) \) steps.
- First find a closed form for \( f(n) \):
  - \( f(n) = 3(1 + 2 + \cdots + n) = 3 \cdot \frac{n(n+1)}{2} = \frac{3}{2} n^2 + \frac{3}{2} n \)
- Throw away
  - the lesser term \( \frac{3}{2} n \)
  - and the coefficient \( \frac{3}{2} \)
- We get \( O(n^2) \)

Scale of strength for \( O \)-notation

To determine the dominant term and the lesser terms:

\[ O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(10^n) \]

Example:

\[ O(6n^3 - 15n^2 + 3n \log n) = O(6n^3) = O(n^3) \]
Ignoring bases of logarithms

- When we use $O$-notation, we can **ignore the bases of logarithms**
  - assume that all logarithms are in base 2.
- Changing base involves multiplying by a **constant coefficient**
  - ignored by then $O$-notation
- For example, $\log_{10} n = \frac{\log n}{\log 10}$. Notice now that $\frac{1}{\log 10}$ is a constant.

$O(1)$

- It is easy to see why the $O(1)$ notation is the right one for constant time
- Constant time means that the algorithm finishes in $k$ steps
- $O(k)$ is the same as $O(1)$, constants are ignored

Caveat 1

- $O$-complexity talks about the behaviour for **large values** of $n$
  - this is why we ignore lesser terms!
- For small sizes a “bad” algorithm might be faster than a “good” one
- We can test the algorithms **experimentally** to choose the best one

Caveat 2

- $O(g(n))$ complexity is an **upper bound**
  - the algorithm finishes in **at most** $g(n)$ steps
- Comparing algorithms can be misleading!
  - item A costs **at most** 10 euros
  - item B costs **at most** 5000 euros
  - which one is cheaper?
- Programmers often say $O(g(n))$ but mean $\Theta(g(n))$
  - finishes in **exactly** $g(n)$ steps
  - we won't use $\Theta$ but keep this in mind
Types of complexities

- Depending on the **data**
  - Worst-case vs Average-case
- Depending on the **number of executions**
  - Real-time vs amortized-time

Worst-case vs Average-case

- Say we want to sort an array, **which values** are stored in the array?
- **Worst-case**: take the worst possible values
- **Average-case**: average wrt to all possible values
- Eg. quicksort
  - worst-case: $O(n^2)$ (when data are already sorted)
  - average-case: $O(n \log n)$

Real-time vs amortized-time

- **How many times** do we run the algorithm?
  - **Real-time**: just once
    - $n$ is the size of the problem
  - **Armortized-time**: multiple times
    - take the average wrt all execution (**not** wrt the values!)
    - $n$ is the number of executions
- Example: Dynamic array! (we will see it soon)

Some algorithms and their complexity

We will analyze the following algorithms

- Sequential search
- Selection sort
- Recursive selection sort
Sequential search

```c
int sequential_search(int target, int array[], int size) {
    for (int i = 0; i < size; i++) {
        if (array[i] == target)
            return i;
    }
    return -1;
}
```

- The steps to locate target depends on its position in array
  - if target is in array[0], then we need only one step
  - if target is in array[i-1], then we need i steps

Complexity analysis

Worst case
- This is when target is in array[size-1]
- The algorithm needs n steps
- So its complexity is $O(n)$

Average case
- Assume that we always search for a target that exists in array
- If target == array[i-1] then we need i steps
- Average wrt all possible positions i (all are equally likely)
  $$\text{Average} = \frac{1 + \ldots + n}{n} = \frac{n(n+1)}{2n} = \frac{n}{2} + \frac{1}{2}$$
- Therefore the average is $O(n)$
  - Same if we consider targets that don’t exist in array

Selection sort algorithm

```c
void selection_sort(int array[], int size) {
    // Βρίσκουμε το μικρότερο στοιχείο και το τοποθετούμε στη θέση του.
    // Και συνεχίζουμε με τον ίδιο τρόπο στον υπόλοιπο πίνακα.
    for (int i = 0; i < size; i++) {
        int min_position = i;
        for (int j = i; j < size; j++)
            if (array[j] < array[min_position])
                min_position = j;
        // swap των στοιχείων i και min_position
        int temp = array[i];
        array[i] = array[min_position];
        array[min_position] = temp;
    }
}
```
Complexity analysis of selection_sort

- Inner for:
  - its body is constant: 1 step
  - \( n - i \) repetitions (\( n = \text{size}, i = \text{current value of } i \))
  - so the whole loop takes \( n - i \) steps

- Outer for:
  - its body takes \( n - i \) steps
    - +1 for the constant swapping part (ignored compared to \( n - i \))
  - first execution: \( n \) steps, second: \( n - 1 \) steps, etc
  - Total: \( n + \ldots + 1 = \frac{n(n+1)}{2} \) steps

- So the time complexity of the algorithm is \( O(n^2) \)

Recursive selection_sort

Elegant recursive version of the algorithm

```c
// Ταξινομεί τον πίνακα array μεγέθους size
void selection_sort(int array[], int size) {
    // Με λιγότερα από 2 στοιχεία δεν έχουμε τίποτα να κάνουμε
    if (size < 2)
        return;
    // Τοποθετούμε το ελάχιστο στοιχείο στην αρχή
    swap(array, 0, find_min_position(array, size));
    // Ταξινομούμε τον υπόλοιπο πίνακα
    selection_sort(array + 1, size - 1);
}
```

Analysis of recursive selection_sort

- How many steps does selection_sort take?
  - Let \( g(n) \) denote that number
  - \( g(0) = g(1) = 1 \) (nothing to do)
  - For \( n > 1 \) selection_sort calls:
    - \( \text{find_min_position}: n \) steps
    - \( \text{swap}: 1 \) step (ignored compared to \( n \))
    - \( \text{selection_sort}: g(n-1) \) steps

So \( g(n) = \begin{cases} 
    n + g(n-1) & n > 1 \\
    1 & n \leq 1 
\end{cases} \)
Analysis of recursive selection_sort

This is a recurrence relation, we can solve it by unrolling:

\[ g(n) = n + g(n - 1) \]
\[ = n + (n - 1) + g(n - 2) \]
\[ = n + (n - 1) + (n - 2) + g(n - 3) \]
... 
\[ = n + ... + 1 \]
\[ = \frac{n(n + 1)}{2} \]

So again we get complexity \( O(n^2) \)

ADTList using Linked Lists

What is the worst case complexity of each operation?

- list_insert_next
- list_remove_next
- list_next
- list_last
- list_find

Readings

- Robert Sedgewick. Αλγόριθμοι σε C, Κεφ. 2.