Course Name: Analysis and Design of Algorithms

Topics to be covered

- Algorithms
  - What is an Algorithm?
  - Characteristics
  - Complexity

Algorithms

- What is an algorithm?

- An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

- This is a rather vague definition. You will get to know a more precise and mathematically useful definition when you attend CS420.

- But this one is good enough for now…

Algorithms

- Properties of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Algorithm Examples

- We will use a pseudocode to specify algorithms, which slightly reminds us of Basic and Pascal.
- Example: an algorithm that finds the maximum element in a finite sequence

- **procedure** max(a_1, a_2, …, a_n: integers)
- max := a_1
- **for** i := 2 **to** n
-         **if** max < a_i **then** max := a_i
- {max is the largest element}

Algorithm Examples

- Another example: a linear search algorithm, that is, an algorithm that linearly searches a sequence for a particular element.
- **procedure** linear_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
- $i := 1$
- **while** ($i \leq n$ and $x \neq a_i$)
- $i := i + 1$
- **if** $i \leq n$ **then** location := i
- **else** location := 0
- {location is the subscript of the term that equals x, or is zero if x is not found}

Algorithm Examples

- If the terms in a sequence are ordered, a binary search algorithm is more efficient than linear search.

- The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

found !

Algorithm Examples

- **procedure** binary_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
- i := 1 {i is left endpoint of search interval}
- j := n {j is right endpoint of search interval}
- **while** (i < j)
- **begin**
- $m := \lfloor (i + j)/2 \rfloor$
- **if** $x > a_m$ **then** i := m + 1
- **else** j := m
- **end**
- **if** $x = a_i$ **then** location := i
- **else** location := 0
- {location is the subscript of the term that equals x, or is zero if x is not found}

Complexity

- In general, we are not so much interested in the time and space complexity for small inputs.

- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with $n = 10$, it is gigantic for $n = 2^{30}$.

Complexity

- For example, let us assume two algorithms A and B that solve the same class of problems.
- The time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.
- For n = 10, A requires 50,000 steps, but B only 3, so B seems to be superior to A.
- For n = 1000, however, A requires 5,000,000 steps, while B requires $2.5 \cdot 10^{41}$ steps.

Complexity

- This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.

- So what is important is the **growth** of the complexity functions.

- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

Complexity

- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	$5{,}000n$	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Complexity

- This means that algorithm B cannot be used for large inputs, while running algorithm A is still feasible.

- So what is important is the **growth** of the complexity functions.

- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

The Growth of Functions

- The growth of functions is usually described using the **big-O notation**.

- **Definition:** Let f and g be functions from the integers or the real numbers to the real numbers.
- We say that f(x) is O(g(x)) if there are constants C and k such that

- $|f(x)| \leq C|g(x)|$

- whenever $x > k$.

The Growth of Functions

- When we analyze the growth of **complexity functions**, f(x) and g(x) are always positive.
- Therefore, we can simplify the big-O requirement to
- $f(x) \leq C \cdot g(x)$ whenever $x > k$.
- If we want to show that f(x) is O(g(x)), we only need to find **one** pair (C, k) (which is never unique).

The Growth of Functions

- The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(x) for large x.
- This boundary is specified by a function g(x) that is usually much **simpler** than f(x).
- We accept the constant C in the requirement
- $f(x) \leq C \cdot g(x)$ whenever $x > k$,
- because **C does not grow with x.**
- We are only interested in large x, so it is OK if $f(x) > C \cdot g(x)$ for $x \leq k$.

The Growth of Functions

- Example:
- Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.
- For $x > 1$ we have:
- $x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$
- $\Rightarrow x^2 + 2x + 1 \leq 4x^2$
- Therefore, for $C = 4$ and $k = 1$:
- $f(x) \leq Cx^2$ whenever $x > k$.
- $\Rightarrow f(x)$ is $O(x^2)$.

The Growth of Functions

- Question: If f(x) is $O(x^2)$, is it also $O(x^3)$?

- **Yes.** x^3 grows faster than x^2, so x^3 grows also faster than f(x).

- Therefore, we always have to find the **smallest** simple function g(x) for which f(x) is O(g(x)).

The Growth of Functions

- "Popular" functions $g(n)$ are
- $n \log n$, 1, 2^n, n^2, $n!$, n, n^3, $\log n$

- Listed from slowest to fastest growth:
- 1
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n
- $n!$

The Growth of Functions

- A problem that can be solved with polynomial worst-case complexity is called **tractable**.

- Problems of higher complexity are called **intractable.**

- Problems that no algorithm can solve are called **unsolvable**.

- You will find out more about this in CS420.

Useful Rules for Big-O

- For any **polynomial** $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers,
- $f(x)$ is $O(x^n)$.

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$

- If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, then $(f_1 + f_2)(x)$ is $O(g(x))$.

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 f_2)(x)$ is $O(g_1(x)\, g_2(x))$.

Complexity Examples

- What does the following algorithm compute?
- **procedure** who_knows($a_1, a_2, \ldots, a_n$: integers)
- m := 0
- **for** i := 1 to n-1
-         **for** j := i + 1 to n
-                 **if** $|a_i – a_j| > m$ **then** $m := |a_i – a_j|$
- {m is the maximum difference between any two numbers in the input sequence}
- Comparisons: n-1 + n-2 + n-3 + … + 1
-                 $= (n – 1)n/2 = 0.5n^2 – 0.5n$

- Time complexity is $O(n^2)$.

Complexity Examples

- Another algorithm solving the same problem:
- **procedure** max_diff($a_1, a_2, \ldots, a_n$: integers)
- min := a1
- max := a1
- **for** i := 2 to n
- **if** a_i < min **then** min := a_i
- **else** if a_i > max **then** max := a_i
- m := max - min
- Comparisons: 2n - 2
- Time complexity is O(n).