Sequences

Sequences represent ordered lists of elements. A sequence is defined as a function from a subset of \( \mathbb{N} \) to a set \( S \). We use the notation \( a_n \) to denote the image of the integer \( n \). We call \( a_n \) a term of the sequence.

Example:

<table>
<thead>
<tr>
<th>subset of ( \mathbb{N} ):</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S ):</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>…</td>
</tr>
</tbody>
</table>

We use the notation \( \{a_n\} \) to describe a sequence. Important: Do not confuse this with the \( \{} \) used in set notation.

It is convenient to describe a sequence with an equation.

For example, the sequence on the previous slide can be specified as \( \{a_n\} \), where \( a_n = 2n \).

The Equation Game

1, 3, 5, 7, 9, … \( a_n = 2n - 1 \)
-1, 1, -1, 1, -1, … \( a_n = (-1)^n \)
2, 5, 10, 17, 26, … \( a_n = n^2 + 1 \)
0.25, 0.5, 0.75, 1, 1.25 … \( a_n = 0.25n \)
3, 9, 27, 81, 243, … \( a_n = 3^n \)

What does \( \sum_{j=m}^{n} a_j \) stand for?

It represents the sum \( a_m + a_{m+1} + a_{m+2} + \ldots + a_n \).

The variable \( j \) is called the index of summation, running from its lower limit \( m \) to its upper limit \( n \). We could as well have used any other letter to denote this index.

Strings

Finite sequences are also called strings, denoted by \( a_1a_2a_3\ldots a_n \).

The length of a string \( S \) is the number of terms that it consists of.

The empty string contains no terms at all. It has length zero.
Summations
How can we express the sum of the first 1000 terms of the sequence \( a_n = n^2 \) for \( n = 1, 2, 3, \ldots \)?
We write it as \( \sum_{j=1}^{1000} j^2 \)?
What is the value of \( \sum_{j=1}^{1000} j \)?
It is \( 1 + 2 + 3 + 4 + 5 + 6 = 21 \).
What is the value of \( \sum_{j=1}^{1000} i \)?
It is so much work to calculate this…

It is said that Carl Friedrich Gauss came up with the following formula:
\[
\sum_{j=1}^{n} j = \frac{n(n+1)}{2}
\]
When you have such a formula, the result of any summation can be calculated much more easily, for example:
\[
\sum_{j=4}^{100} j = \frac{100(100+1)}{2} = \frac{10100}{2} = 5050
\]

Double Summations
Corresponding to nested loops in C or Java, there is also double (or triple etc.) summation:
Example:
\[
\sum_{i=1}^{5} \sum_{j=1}^{5} ij = \sum_{i=1}^{5} (i + 2i) = \sum_{i=1}^{5} 3i = 3 + 6 + 9 + 12 + 15 = 45
\]

Table 2 in 4th Edition: Section 1.7
5th Edition: Section 3.2
6th and 7th Edition: Section 2.4
contains some very useful formulas for calculating sums.

In the same Section, Exercises 15 and 17 (7th Edition: Exercises 31 and 33) make a nice homework.

Enough Mathematical Appetizers!
Let us look at something more interesting:

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 or CS620.

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(a1, a2, …, an: integers)
    max := a1
    for i := 2 to n
        if max < ai then
            max := ai
    {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; a1, a2, …, an: integers)
    i := 1
    while (i ≤ n and x ≠ ai)
        i := i + 1
    if i ≤ 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}
```

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

acdfghjlmpsvxz

center element

Algorithm Examples

binary search for the letter 'j'

search interval

acdfghjlmpsvxz

center element

Algorithm Examples

binary search for the letter 'j'

search interval

acdfghjlmpsvxz

center element

Algorithm Examples

procedure binary_search(x: integer; a1, a2, ..., an:
integers)
i := 1   {i is left endpoint of search interval}
j := n  {j is right endpoint of search interval}
while (i < j)
begin
m := (i + j)/2
if x > am then i := m + 1
else j := m
end
if x = ai 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

What is the time complexity of the linear search algorithm?
We will determine the worst-case number of comparisons as a function of the number n of terms in the sequence.
The worst case for the linear algorithm occurs when the element to be located is not included in the sequence.
In that case, every item in the sequence is compared to the element to be located.

Obviously, on sorted sequences, binary search is more efficient than linear search.
How can we analyze the efficiency of algorithms?
We can measure the
• time (number of elementary computations) and
• space (number of memory cells) that the algorithm requires.
These measures are called computational complexity and space complexity, respectively.
Algorithm Examples

Here is the linear search algorithm again:

```plaintext
procedure linear_search(x: integer; a1, a2, ..., an: integers)
    i := 1
    while (i ≤ n and x ≠ ai)
        i := i + 1
    if i ≤ 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}
```

Complexity

For n elements, the loop

```plaintext
while (i ≤ n and x ≠ ai)
    i := i + 1
```

is processed n times, requiring 2n comparisons. When it is entered for the (n+1)th time, only the comparison i ≤ n is executed and terminates the loop. Finally, the comparison if i ≤ n then location := i is executed, so all in all we have a worst-case time complexity of 2n + 2.

Reminder: Binary Search Algorithm

```plaintext
procedure binary_search(x: integer; a1, a2, ..., an: integers)
    i := 1 {i is left endpoint of search interval}
    j := n {j is right endpoint of search interval}
    while (i < j)
        begin
            m := ⌊(i + j)/2⌋
            if x > am then i := m + 1
            else j := m
        end
    if x = ai 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

What is the time complexity of the binary search algorithm?

Again, we will determine the worst-case number of comparisons as a function of the number n of terms in the sequence.

Let us assume there are n = 2^k elements in the list, which means that k = log n.

If n is not a power of 2, it can be considered part of a larger list, where 2^k < n < 2^(k+1).

In the first cycle of the loop

```plaintext
while (i < j)
    begin
        m := ⌊(i + j)/2⌋
        if x > am then i := m + 1
        else j := m
    end
```

the search interval is restricted to 2^(k-1) elements, using two comparisons.

In the second cycle, the search interval is restricted to 2^(k-2) elements, again using two comparisons.

This is repeated until there is only one (2^2) element left in the search interval.

At this point 2k comparisons have been conducted.
Complexity

Then, the comparison

while \( i < j \)

exits the loop, and a final comparison

if \( x = a \), then\ location := i

determines whether the element was found.

Therefore, the overall time complexity of the binary search algorithm is \( 2k + 2 = 2 \lceil \log n \rceil + 2 \).

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.

Comparison: time complexity of algorithms A and B

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm A</th>
<th>Algorithm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>5,000n</td>
<td>( \lceil 1.1^n \rceil )</td>
</tr>
<tr>
<td>10</td>
<td>50,000</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>500,000</td>
<td>13,781</td>
</tr>
<tr>
<td>1,000</td>
<td>5,000,000</td>
<td>2.5 \times 10^{41}</td>
</tr>
<tr>
<td>1,000,000</td>
<td>5 \times 10^{10}</td>
<td>4.8 \times 10^{12392}</td>
</tr>
</tbody>
</table>

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.