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…

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}
```

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}
```

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'

center element

Algorithm Examples

binary search for the letter 'j'

center element

Algorithm Examples

binary search for the letter 'j'

center element

Algorithm Examples

binary search for the letter 'j'

found!

Algorithm Examples

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

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.

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.

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 \( 2^k < n < 2^{k+1} \).

Reminder: Binary Search Algorithm

**procedure** binary_search(x: integer; a1, a2, …, an: integers)

\( i := 1 \)  \( \text{\{i is left endpoint of search interval} \)

\( j := n \)  \( \text{\{j is right endpoint of search interval} \)

**while** \( i < j \)

begin

\( m := \lceil (i + j)/2 \rceil \)

if \( x > a_m \) then \( i := m + 1 \)

else \( j := m \)

end

if \( x = a_i \) then \( \text{location := i} \)

else \( \text{location := 0} \)

(location is the subscript of the term that equals \( x \), or is zero if \( x \) is not found)
In the first cycle of the loop

\[ \text{while } (i < j) \]

begin

\[ m := \lfloor \frac{(i + j)}{2} \rfloor \]

if \( x > a_m \) 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 \( (2^0) \) element left in the search interval.

At this point \( 2k \) comparisons have been conducted.

Then, the comparison

\[ \text{while } (i < j) \]

exits the loop, and a final comparison

if \( x = a_i \) then \( \text{location} := i \)

determines whether the element was found.

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

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 \( n = 2^{30} \).

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm A ( 5,000n )</th>
<th>Algorithm B ( 1.1^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( 5,000 )</td>
<td>( 1.1^n )</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 ( 10^{41} )</td>
</tr>
<tr>
<td>1,000,000</td>
<td>( 5 \times 10^9 )</td>
<td>4.8 ( 10^{1398} )</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.

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

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
\[ f(x) \leq C \cdot g(x) \text{ whenever } x > k. \]

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
\[ f(x) \leq C \cdot g(x) \text{ whenever } x > k, \]
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:
\[ f(x) \leq Cx^2 \text{ whenever } x > k. \]
\[ \Rightarrow f(x) \text{ is } O(x^2). \]

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)).
“Popular” functions $g(n)$ are $n \log n$, $1$, $2^n$, $n^2$, $n^3$, $n!$, $n$, $\log n$

Listed from slowest to fastest growth:

- $1$
- $\log n$
- $n$
- $n \log n$
- $n^2$
- $n^3$
- $2^n$
- $n!$

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.