Algorithms and Data Structures

Complexity of Algorithms

Marcin Sydow

Desired Properties of a Good Algorithm

Any good algorithm should satisfy 2 obvious conditions:

1. compute **correct** (desired) output (for the given problem)
2. be **effective** ("fast")

ad. 1) correctness of algorithm

ad. 2) complexity of algorithm

Complexity of algorithm measures how "fast" is the algorithm (**time complexity**) and what "amount" of memory it uses (**space complexity**) - time and memory - 2 basic resources in computations

Example - the Search Problem

Problem of **searching** a key in an array

What does the amount of work of this algorithm depend on?

```
find(arr, len, key)
```

Specification:

input: arr - array of integers, len - it's length, key - integer to be found

output: integer $0 \leq i < len$ being the index in arr, under which the key is stored

Example - the Search Problem

Problem of **searching** a key in an array

What does the amount of work of this algorithm depend on?

```
find(arr, len, key)
```

Specification:

input: arr - array of integers, len - it's length, key - integer to be found

output: integer $0 \leq i < len$ being the index in arr, under which the key is stored (is it a complete/clear specification?)

Example - the Search Problem

Problem of **searching** a key in an array

What does the amount of work of this algorithm depend on?

`find(arr, len, key)`

Specification:

input: arr - array of integers, len - it's length, key - integer to be found

output: integer $0 \leq i < len$ being the index in arr, under which the key is stored (is it a complete/clear specification?) **or** the value of -1 when there is no specified key in (first len positions of) the array

Example - the Search Problem

Problem of **searching** a key in an array

What does the amount of work of this algorithm depend on?

`find(arr, len, key)`

Specification:

input: arr - array of integers, len - it's length, key - integer to be found

output: integer $0 \leq i < len$ being the index in arr, under which the key is stored (is it a complete/clear specification?) **or** the value of -1 when there is no specified key in (first len positions of) the array

code:

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

What does the amount of work of this algorithm depend on?

The "speed" of algorithm

How to measure how fast (or slow) an algorithm is?

There are 2 issues to be considered when designing such a measure:

1. independence on any programming language (and hardware/software platform)
2. maximum independence on particular input data

It should be an internal property of the algorithm itself

Any idea?

The "speed" of algorithm

How to measure how fast (or slow) an algorithm is?

There are 2 issues to be considered when designing such a measure:

1. independence on any programming language (and hardware/software platform)
2. maximum independence on particular input data

It should be an internal property of the algorithm itself

Any idea? Count basic **operations** of the algorithm

Dominating Operations

Simplification: it is not necessary to count all the operations - it is enough to count the "representative" ones

Before doing a complexity analysis **2 steps must be done**:

1. determine the dominating operation set
2. observe what (in input) influences the number of dominating operations (**data size**)

Dominating operations are those which cover the amount of work which is proportional to the whole amount of work of the algorithm (they well represent the whole)

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ? yes

comparison `arr[i] == key` ?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ? yes

comparison `arr[i] == key` ? yes

both the above?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ? yes

comparison `arr[i] == key` ? yes

both the above? yes

return statement `return i` ?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ? yes

comparison `arr[i] == key` ? yes

both the above? yes

return statement `return i` ? no

index increment `i++` ?

Example - determining operating operations

What can be the **dominating operation** set in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

assignment `i = 0` ? no

comparison `i < len` ? yes

comparison `arr[i] == key` ? yes

both the above? yes

return statement `return i` ? no

index increment `i++` ? yes

Example, cont. - determining the data size

What is the **data size** in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Example, cont. - determining the data size

What is the **data size** in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Data size: **length of array** arr

Example, cont. - determining the data size

What is the **data size** in the following algorithm?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Data size: **length of array** arr

Having determined the **dominating operation** and **data size** we can determine time complexity of the algorithm

Time Complexity of Algorithm

Definition

Time Complexity of Algorithm is the number of dominating operations executed by the algorithm **as the function** of data size.

Time complexity measures the "amount of work" done by the algorithm during solving the problem in the way which is **independent** on the implementation and particular input data.

The lower time complexity the "faster" algorithm

Example - time complexity of algorithm


```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Assume:

dominating operation: comparison `arr[i] == key`

data size: the length of array (denote by n)

Thus, the number of dominating operations executed by this algorithm ranges:

- from **1** (the key was found under the first index)
- to **n** (the key is absent or under the last index)

There is no single function which could express the dependence of the number of executed dominating operations on the data size for this algorithm.

Pessimistic Time Complexity

let's assume the following denotations:
n - data size
D_n - the set of all possible input datasets of size n
$t(d)$ - the number of dominating operations for dataset d (of size n)
$(d \in D_n)$

Definition

Pessimistic Time Complexity of algorithm:
$W(n) = sup\{t(d) : d \in D_n\}$

(W(n) - **W**orst)

Pessimistic Time Complexity expresses the number of dominating operations in the worst case of input data of size n

E.g. for our example the pessimistic time complexity is given by the formula:

Pessimistic Time Complexity

let's assume the following denotations:
n - data size
D_n - the set of all possible input datasets of size n
$t(d)$ - the number of dominating operations for dataset d (of size n)
$(d \in D_n)$

Definition

Pessimistic Time Complexity of algorithm:
$W(n) = sup\{t(d) : d \in D_n\}$

(W(n) - **W**orst)

Pessimistic Time Complexity expresses the number of dominating operations in the worst case of input data of size n

E.g. for our example the pessimistic time complexity is given by the formula:
$W(n) = n$

Average Time Complexity of Algorithm

let's assume the following denotations:
n - data size
D_n - the set of all possible input datasets of size n
$t(d)$ - the number of dominating operations for dataset d (of size n) ($d \in D_n$)
X_n - random variable, its value is t(d) for $d \in D_n$

p_{nk} - probability distribution of the random variable X_n (i.e. the probability that for input data of size n the algorithm will execute k dominating operations ($k \geq 0$))

Definition

Average Time Complexity of Algorithm:
$A(n) = \sum_{k \geq 0} p_{nk} \cdot k = \sum P(X_n = k) \cdot k$
(expected value of the random variable representing the number of dominating operations)

(A(n) **A**verage)

Example - Determining the Average Time Complexity

Let's determine the average time complexity for our exemplary algorithm (find)

First, we have to assume some **probabilistic model** of input data (i.e. the probabilistic distribution of possible input datasets)

Let's make a simplistic assumption: the key to be found occurs exactly once in array and with the same probability on each index (uniform distribution) ($\forall_{0 \leq k < n} P(X_n = k) = 1/n$)

Thus:
$A(n) = \sum_{k \geq 0} P(X_n = k) \cdot k = \sum_{0 \leq k < n} 1/n \cdot k = \frac{n+1}{2}$

Space Complexity of Algorithm

Definition

Space Complexity of Algorithm: **S(n)** is the **number of units of memory used** by algorithm **as a function of data size**

This characteristic is more dependent on particular platform than time complexity.

As a memory unit one can consider the machine word.

Note:

We will assume, that the memory used for keeping the **input data** is **not considered** because usually arrays (and other compound types) are passed as arguments to functions by reference, which does not involve much memory

In our example space complexity is **constant** - because it consumes memory only for a single variable (plus some fixed number of additional temporal variables), **independently** on the input data size: **S(n) = const**

Omitting Unimportant Details

The real time spent by an **implementation** of the algorithm may differ between **particular platforms** by a **constant multiplicative factor**. (e.g. CPU speed)

Thus, it would be very useful to have a **notation** allowing for expressing the complexity functions with **neglecting unimportant details** (as multiplicative or additive constant, for example)

E.g. for the following function:
$A(n) = 2.1 \cdot n - 1$
The most important information is that it is a **linear function** - it's rank of complexity is linear

Does such a notation exist?

Asymptotic Notation - "Big O"

The notation is called "asymptotic notation".
There are a couple of flavours. The most common is "big O":

Definition

The function g(n) is **the upper bound of rank of order** of the function f(n):

$f(n) = O(g(n)) \Leftrightarrow \exists_{c>0} \exists_{n_0} \forall_{n \geq n_0} f(n) \leq c \cdot g(n)$

The O() notation intuitively corresponds to the "$\leq$" symbol (in terms of ranks of orders of functions).

E.g. the fact that W(n) of our exemplary algorithm has an upper bound of the linear rank can be noted as:
$W(n) = \frac{n+1}{2} = O(n)$
The constant space complexity S(n) of that algorithm can be expressed with the following special notation:
$S(n) = O(1)$

Asymptotic Notation - "Big Theta"

Another important flavour of asymptotic notation is "big Theta":

Definition

The function f(n) has **the same rank of order** as the function g(n): $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge g(n) = O(f(n))$

The $\Theta()$ notation intuitively corresponds to the "=" symbol (in terms of ranks of orders of functions).

Notice, that $\Theta()$ is defined with the use of $O()$, similarly as "=" symbol can be defined with the use of "$\leq$" symbol.

E.g. the expression: $f(n) = n^2 + n - 3 = \Theta(n^2)$
reads as "the $n^2 + n - 3$ function" is of **square rank of order**.

Other Flavours of Asymptotic Notation

We have 5 relation symbols for comparing numbers: $= \leq \geq < >$

In total, we also have 5 analogous symbols for comparing ranks of functions:

1. Θ - "$=$"
2. O - "$\leq$"
3. Ω - "$\geq$"
4. o - "$<$"
5. ω - "$>$"

(in general, a capital letter denotes "non-sharp" inequality and lowercase denotes a "sharp" one)

E.g.:
W(n)=o(n) (lowercase o)
means: "the rank of function W(n) is lower than linear"

Some Remarks on Using the Asymptotic Notation

Notice: in expressions like "f(n)=O(g(n))" the "=" has a special meaning - it does not represent the "normal" equality. The expression has it's meaning only as a whole.

E.g. it does not make sense to use asymptotic notation as the first expression on the left-hand side of the "=" symbol.
E.g. expressions like "O(f(n)) = n" or "O(f(n)) = O(g(n))" **do not make any sense**

Besides the standard usage of the asymptotic notation on the right-hand side of the "=" symbol, it can be also used in the following way:
$f(n) = g(n) + O(h(n))$
Which means: $f(n) - g(n) = O(h(n))$
("the ranks of functions f and g differ at most by a rank of function h")

Remarks: Comparing Ranks of Functions

Sometimes the following technique is useful.

Ranks of some 2 functions f(n) and g(n) can be compared by computing the following limit:

$lim_{n \to \infty} \frac{f(n)}{g(n)}$

there are 3 possible cases for the limit:

1. ∞ - in that case f(n)=$\omega(g(n))$ (f has higher rank)
2. a positive constant - in that case f(n)=$\Theta(g(n))$ (the same ranks)
3. zero - in that case f(n)=o(g(n)) ("lowercase o") (g has higher rank)

The Most Common Ranks of Functions

- **constant** (e.g. $S(n) = 3 = \Theta(1)$)
- **logarithmic** (e.g. $W(n) = 2 + lg_2 n = \Theta(\log(n))$)
- **linear** (e.g. $A(n) = 2n + 1 = \Theta(n)$)
- **linear-logarithmic** (e.g. $A(n) = 1.44 \cdot n \log(n) = \Theta(n \log(n))$)
- **square** (e.g. $W(n) = n^2 + 4 = \Theta(n^2)$)
- **cubic** (e.g. $A(n) = \Theta(n^3)$)
- sub-exponential (e.g. $A(n) = \Theta(n^{\log(n)})$)
- exponential (e.g. $A(n) = \Theta(2^n)$)
- factorial (e.g. $W(n) = \Theta(n!)$)

In simplification: in practise, an over-polynomial rank of time complexity is considered as "unacceptably high"

In case of space complexity, even linear rank is considered as **very high**

Questions/Problems:

- How to measure the "speed of algorithm"
- What 2 things should be determined before starting assessing the time complexity of an algorithm
- What is a dominating operation
- Definition of Time Complexity of Algorithm
- Definition of Space Complexity of Algorithm
- Definition of Pessimistic Time Complexity
- Definition of Average Time Complexity
- Be able to determine time complexity for simple algorithms
- What is the purpose of the asymptotic notation
- Definition and interpretation of the O() notation
- Definitions (and interpretations) of the other types of asymptotic notations
- Ability to express rank of a given function with the asymptotic notation

Thank you for your attention