LECTURE 4:

Analyzing the Complexity of Algorithms (I)

Outline

- What is complexity analysis ?
- How can be time complexity measured ?
- Examples
- Best-case, worst-case and average-case analysis

What is complexity analysis ?

To establish the amount of computing resources needed to execute the algorithm

Usefulness: to compare algorithms and to estimate before algorithm is executed the expected amount of resources needed to obtain the result

What is complexity analysis ?

Computing resources:

- Running time = time needed to execute the the algorithm
- Memory (space) = space needed to store the data processed by the algorithm

Efficient algorithm: an algorithm which uses a “reasonable amount” of computing resources

If an algorithm uses less resources than another one then the first one is considered to be more efficient than the second one

What do we measure/compare ?

Is the goal of our measurement a number ? (Like 1.4s, or 1Gb ?)

NO ! Our algorithms are “general recipes”, designed to run on **many inputs**.

INSTANCE = “concrete case of a problem”

PROBLEM = set of all instances.

The amount of resources **depends on the size of the input data** = size of the instance.

The main aim of complexity analysis is to answer the question:

how does the running time and/or space of the algorithm depend on input size ?

Determining the input size of a given instance

… Usually, quite straightforward

Input size = dimension of the space needed to store all input data

It can be expressed in one of the following ways:

- the number of elements (real values, integer values, characters etc) belonging to the input data
- the number of bits necessary to represent the input data

How can we determine the input size for a given problem ?

Examples:

1. Find the minimum of an array x[1..n]

   Input size: n

1. Compute the value of a polynomial of order n

   Input size: n

Compute the sum of two m*n matrices

Input size: (m,n) or mn

1. Verify if a number n is prime or not

   Input size: n or $[\log_2 n]+1$

How can we determine the input size for a given problem ?

Complexity can be influenced by data representation

Example: 100x100 ”sparse” matrix - contains only 50 nonzero elements

Same data, multiple representations:

1. Classical (100*100=10000 values)
2. One-dimensional array of non-zero elements:

   for each non-zero element: (i,j,a[i,j]) (~150 values)

Second variant: more space-efficient

First variant: can be more efficient with respect to time

Tradeoff *(compromise) between space efficiency and time efficiency*

Input size: complexity tradeoffs

Complexity can be influenced by data representation

Example: 100x100 ”sparse” matrix - contains only 50 nonzero elements

Same data, multiple representations:

1. Classical (100*100=10000 values)
2. One-dimensional array of non-zero elements:

   for each non-zero element: (i,j,a[i,j]) (~150 values)

Give me examples of operation that are more efficient in the first representation than in the second one !

Outline

- What is complexity analysis ?
- How can be time complexity measured ?
- Examples
- Best-case, worst-case and average-case analysis

How can we measure time complexity ?

Computational **model**: random access machine (RAM)

Characteristics (simplifying assumptions):

All processing steps are executed sequentially
(no parallelism in execution)

- The time of executing the basic operations does not depend on the values of the operands
  (no time difference between computing 1+2 and computing 12433+4567)

- The time to access data does not depend on their address (no difference between processing the first element of an array and processing the last element)

How can we measure time complexity ?

Measuring unit = time needed to execute a basic operation

Basic operations:

- Assignment
- Arithmetical operations
- Comparisons
- Logical operations

Running time = number of basic operations

The running time expresses the dependence of the number of operations on the input size

Example 1

Preconditions: n>=1

Input size: n

Algorithm:

```
Sum(n)
1: S←0
2: i ← 0
3: WHILE i<n DO
4:       i ← i+1
5:       S ← S+i
6: ENDWHILE
7: RETURN S
```

Algorithm:

Operation	Cost	Iterations
1	c1	1
2	c2	1
3	c3	n+1
4	c4	n
5	c5	n

Running time:

$T(n)=(c3+c4+c5)n+(c1+c2+c3)=$

$= a*n +b$

Example 1

- Assuming that all basic operations have an unit cost:

$T(n)=3(n+1)$

- The values of the constants appearing in the expression of the running time are not very important. The important fact is that the running time depends linearly on the input size

- Since the algorithm is equivalent to:

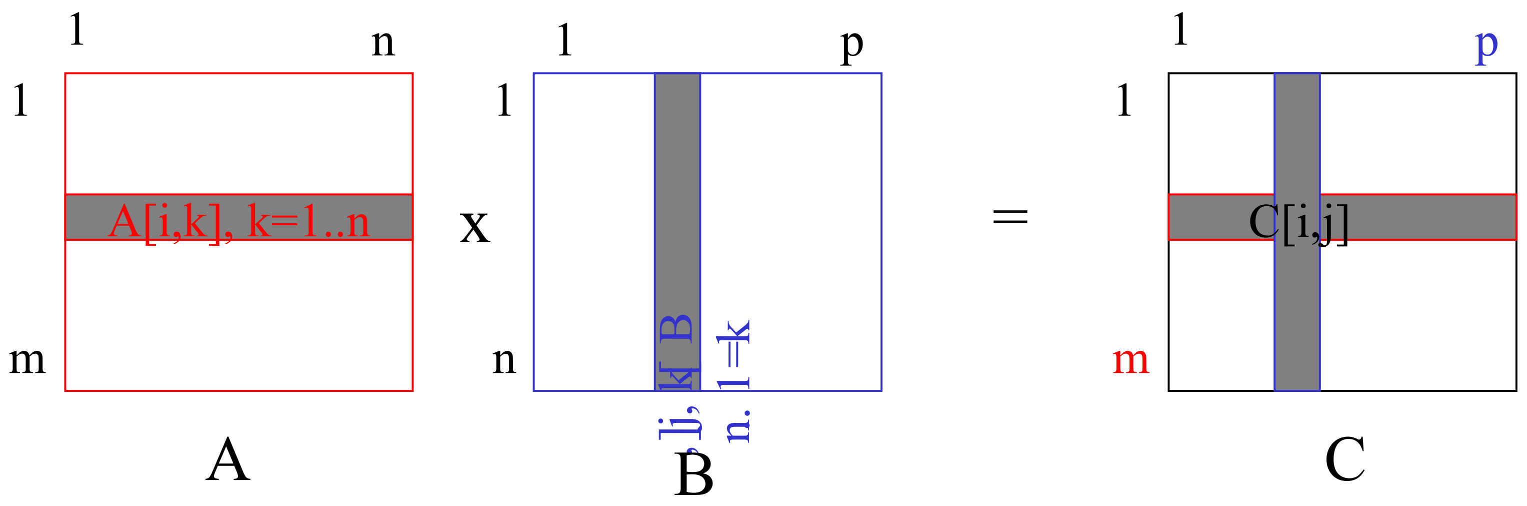
```
S:=0
FOR i:=1,n DO S:=S+i  ENDFOR
```

it is easy to see that the cost of updating the counting variable i is $2(n+1)$; the other $(n+1)$ operations correspond to the operations involving S (initialization and modification)

Example 2

Preconditions: A_{m*n}, B_{n*p} Outpur: C=A*B

Input size: (m,n,p)



$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

C[i,j]=A[i,1]*B[1,j]+A[i,2]*B[2,j]+…+A[i,n]*B[n,j],
i=1..m, j=1..p

Example 2

Basic idea: for each i=1..m and j=1..p compute the sum over all k

Algorithm:

```
Product(A[1..m,1..n],B[1..n,1..p])
1:  FOR i ← 1,m DO
2:   FOR j ← 1,p DO
3:       C[i,j] ← 0
4:       FOR k ← 1,n DO
5:           C[i,j] ← C[i,j]+A[i,k]*B[k,j]
6:       ENDFOR
7:   ENDFOR
8:  ENDFOR
9: RETURN C[1..m,1..p]
```

Costs table

Op.	Cost	Iter	Total
1	2(m+1)	1	2(m+1)
2	2(p+1)	m	2m(p+1)
3	1	mp	mp
4	2(n+1)	mp	2mp(n+1)
5	3	mpn	3mnp

T(m,n,p)=5mnp+5mp+4m+2

Example 2

no need to always do such a detailed analysis !

Sufficient: to identify the dominant operation

Dominant operation: most frequent/expensive operation

Algorithm:

```
Product(A[1..m,1..n],B[1..n,1..p])
1:  FOR i ← 1,m DO
2:    FOR j ← 1,p DO
3:       C[i,j] ← 0
4:       FOR k ← 1,n DO
5:          C[i,j] ← C[i,j]+A[i,k]*B[k,j]
6:       ENDFOR
7:    ENDFOR
8:  ENDFOR
RETURN C[1..m,1..p]
```

Analysis:

$T(m,n,p)=mnp$

Example 3

Preconditions: x[1..n], n>=1 Result: m=min(x[1..n])

Input size: n

Algorithm:

```
Minimum(x[1..n])
1:  m←x[1]
2:   FOR i ← 2,n DO
3:       IF x[i]<m THEN
4:                    m ← x[i]
5:       ENDIF
6:   ENDFOR
7:RETURN m
```

Table of costs:

Op.	Cost	Iter.	Total
1	1	1	1
2	2n	1	2n
3	1	n-1	n-1
4	1	t(n)	t(n)

$T(n)=3n+t(n)$

The running time depends not only on n but also on the properties of input data !

Example 3

When the running time depends also on the properties of input data we can analyze at least two cases:

- Best case (x[1]<=x[i], i=1..n): t(n)=0 => T(n)=3n
- Worst case (x[1]>x[2]>…>x[n]): t(n)=n-1=> T(n)=4n-1

Thus 3n<=T(n)<=4n-1

Both the lower and the upper bound depend linearly on the input size

Dominant operation:

comparison

T(n) =n-1

Algorithm:

```
Minimum(x[1..n])
1:  m ← x[1]
2:   FOR i ← 2,n DO
3:       IF x[i]<m THEN
4:                  m ← x[i]
5:       ENDIF
6:   ENDFOR
7: RETURN m
```

Example 4

Preconditions: x[1..n], n>=1, v a value

Result: the variable “found” contains the truth value of the statement “*the value v is in the array x[1..n]*”

Input size: n

Algorithm (sequential search):

```
search(x[1..n],v)
1: found ← False
2: i:=1
3: WHILE (found=False) AND (i<=n) DO
4:      IF x[i]=v                    //t1(n)
5:           THEN found ← True       //t2(n)
6:           ELSE  i ← i+1           //t3(n)
7:      ENDIF
8: ENDWHILE
9: RETURN found
```

Costs table

Op.	Cost
1	1
2	1
3	t1(n)+1
4	t1(n)
5	t2(n)
6	t3(n)

Example 4

The running time depends on the properties of the array.

Case 1: the value v is in the array (let k be the first position of v)

Case 2: the value v is not in the array

$$t1(n)=\begin{cases} k & \text{if v is in the array} \\ n & \text{if v is not in the array} \end{cases}$$

$$t2(n)=\begin{cases} 1 & \text{if v is in the array} \\ 0 & \text{if v is not in the array} \end{cases}$$

$$t3(n)=\begin{cases} k-1 & \text{if v is in the array} \\ n & \text{if v is not in the array} \end{cases}$$

Algorithm (sequential search):

```
search(x[1..n],v)
1:  found ← False
2:  i ← 1
3:  WHILE (found=False) AND (i<=n) DO
4:      IF x[i]=v                          // t1(n)
5:            THEN found ← True            // t2(n)
6:            ELSE  i ← i+1                // t3(n)
7:      ENDIF
8:  ENDWHILE
9: RETURN found
```

Example 4

Best case: x[1]=v

t1(n)=1, t2(n)=1, t3(n)=0

T(n)= 6

Worst case: v is not in the array

t1(n)=n, t2(n)=0, t3(n)=n

T(n)=3n+3

The lower and the upper bound:

6<= T(n) <= 3(n+1)

The lower bound is constant, the upper bound depends linearly on n

$$t1(n)=\begin{cases} k & \text{if v is in the array} \\ n & \text{if v is not in the array} \end{cases}$$

$$t2(n)=\begin{cases} 1 & \text{if v is in the array} \\ 0 & \text{if v is not in the array} \end{cases}$$

$$t3(n)=\begin{cases} k-1 & \text{if v is in the array} \\ n & \text{if v is not in the array} \end{cases}$$

Example 4

```
Search(x[1..n],v)
1:  i ← 1
2:  while x[i]<>v and i<n do
3:      i ← i+1
4:  endwhile
5: if x[i]=v then found ← true
6:           else found ← false
7: endif
8: return found
```

Best case:

$T(n)=4$

Worst case:

$T(n)=1+n+(n-1)+2=2n+2$

Example 4

For some problems the best case and the worst case are exceptional cases

Thus … the running time in the best case and in the worst case do not give us enough information

Another type of analysis … average case analysis

The aim of average case analysis is to give us information about the behavior of the algorithm for typical (random) input data

Outline

- What is efficiency analysis ?
- How can be time efficiency measured ?
- Examples
- Best-case, worst-case and average-case analysis

Best-case and worst-case analysis

Best case analysis:

- gives us a lower bound for the running time

- it can help us to identify inefficient algorithms (if an algorithm has a high cost in the best case)

Worst case analysis:

- gives us the largest running time with respect to all input data of size n (this is an upper bound of the running time)

- the upper bound of the running time is more important than the lower bound

Average-case analysis

This analysis is based on knowing the distribution probability of the input space.

This means to know which is the occurrence probability of each instance of input data (how frequently each instance appears)

The average running time is the mean value (in a statistical sense) of the running times corresponding to different instances of input data.

Average-case analysis

Example: sequential search (dominant operation: comparison)
Hypotheses concerning the probability distribution of input space:

- **Probability that the value v is in the array:** p
  - the value v appears with the same probability on any position
  - the probability that the value v is on position k, given that it appears, is 1/n

- **Probability that the value v is not in the array:** 1-p

$T_a(n)=p(1+2+\ldots+n)/n+(1-p)n=p(n+1)/2+(1-p)n=(1-p/2)n+p/2$

If $p=0.5$ one obtains $T_a(n)=3/4\ n+1/4$

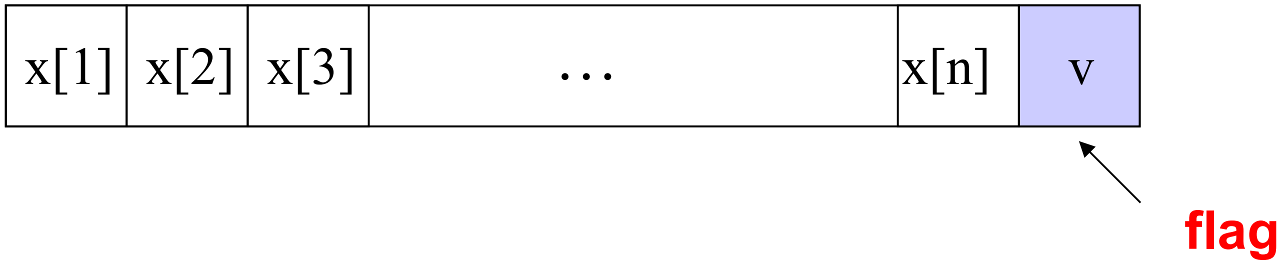
The average running time of sequential search is, as in the worst case, linear with respect to the input size

Average-case analysis

Example: sequential search (flag variant)

Basic idea:

- the array is extended with a position (n+1) and on this position we place v
- the extended array is searched until the value v is found (it will be found at least on position n+1 – in this case we can decide that the value is not in the initial array x[1..n])



Average-case analysis

Algorithm:

```
Search_flag(x[1..n],v)
i ← 1
WHILE x[i]<>v DO
      i ← i+1
ENDWHILE
RETURN i
```

Dominant operation: comparison

Assumption: probability that v is on position k is 1/(n+1)

Average running time:

$T_a(n)=(1+2+\ldots+(n+1))/(n+1)$

$=(n+2)/2$

Remark:

- by changing the hypothesis on the distribution probability of input space the value of average running time changed (however it is still linear)

The average running time is NOT (in general) the arithmetic mean of the running times corresponding to best and average cases

Summary: steps in estimating the running time

- Identify the input size
- Identify the dominant operation
- Count the number of executions of the dominant operation
- If this number depends on the properties of input data analyze:
  - Best case
  - Worst case
  - Average case

What is the order of growth?

In the expression of the running time one of the terms will become significantly larger than the other ones when n becomes large : this is the so-called dominant term

$T1(n)=an+b$	Dominant term: $a\,n$
$T2(n)=a \log n+b$	Dominant term: $a \log n$
$T3(n)=a\,n^2+bn+c$	Dominant term: $a\,n^2$
$T4(n)=a^n+b\,n+c$ $(a>1)$	Dominant term: a^n

What is the order of growth?

Let us analyze what happens with the dominant term when the input size is multiplied by k:

$T'_1(n)=an$	$T'_1(kn)= a\ kn=k\ T'_1(n)$
$T'_2(n)=a \log n$	$T'_2(kn)=a \log(kn)=T'_2(n)+a\log k$
$T'_3(n)=a\ n^2$	$T'_3(kn)=a\ (kn)^2=k^2\ T'_3(n)$
$T'_4(n)=a^n$	$T'_4(kn)=a^{kn}=(a^n)^k=T'_4(n)^k$

What is the order of growth?

The order of growth expresses how does the dominant term of the running time increase with input size

	Order of growth
$T'_1(kn)= a\ kn=k\ T'_1(n)$	Linear
$T'_2(kn)=a \log(kn)=T'_2(n)+a \log k$	Logarithmic
$T'_3(kn)=a\ (kn)^2=k^2\ T'_3(n)$	Quadratic
$T'_4(kn)=a^{kn}=(a^n)^k\ =(T'_4(n))^k$	Exponential

How can we interpret the order of growth?

Between two algorithms the one having a smaller order of growth is considered more efficient

However, this is true only for large enough input sizes

Example. Let us consider

$T1(n)=10n+10$ (linear order of growth)

$T2(n)=n^2$ (quadratic order of growth)

If $n<=10$ then $T1(n)>T2(n)$

In this case the order of growth is relevant only for $n>10$

A comparison of orders of growth

n	$\log_2 n$	$n\log_2 n$	n^2	2^n
10	3.3	33	100	1024
100	6.6	664	10000	10^{30}
1000	10	9965	1000000	10^{301}
10000	13	132877	100000000	10^{3010}