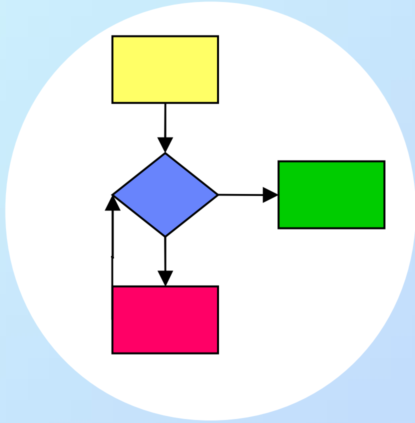
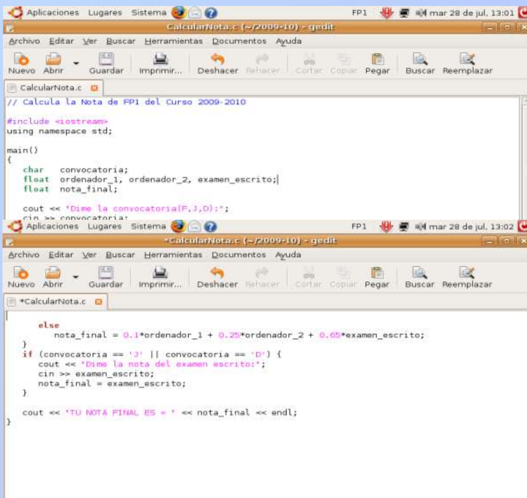
A.R.A.
GRUPOS DE ALTO RENDIMIENTO ACADÉMICO



Programming 1

Lecture 8

Evaluating the temporary cost of an algorithm



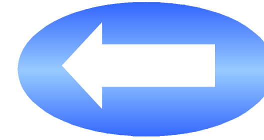
Dpnt. de Ciència de la Computació i Intel·ligència Artificial
Dpto. de Ciencia de la Computación e Inteligencia Artificial

Objectives

- Understand the concept of efficiency of an algorithm
- Learn how to analyse the temporary cost of an algorithm
- Learn how to use the temporary cost as a criterium to guide the design of an algorithmic solution

Topics

1. **Efficiency of an algorithm**



2. Methods to calculate the efficiency
3. Temporary cost of an algorithm
4. Cost analysis by step counting
5. Exercises

Efficiency of an algorithm

- Sometimes there are several algorithms to solve a given problem, which one must we use?
- A criterium to choose an algorithm can be its efficiency
- The **efficiency of an algorithm** is related to the amount of resources needed by the algorithm:
  - **Execution time**
  - Storing space

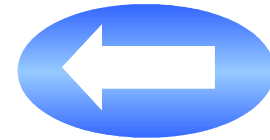
In Programming 1, only the efficiency from the point of view of the execution time will be studied

Execution time of an algorithm

- The execution time of an algorithm depends on:
  - The size of data to be processed
  - Computer speed
  - Quality of the code generated by the compiler or interpreter

Topics

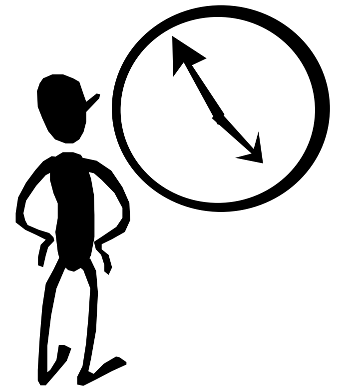
1. Efficiency of an algorithm
2. **Methods to calculate the efficiency**



3. Temporary cost of an algorithm
4. Cost analysis by step counting
5. Exercises

Empirical or a posteriori method

- It consists of programming the algorithms and testing them on a computer. The time and space are then measured
- Disadvantages
  - It does not allow to compare the algorithm in several supports
  - It requires the effort of programming each algorithm to choose the best one
  - The comparison can only be done for few problem sizes



Empirical method: example

- Empirical measurement of temporary efficiency for several sorting algorithms for arrays

Sorting algorithms

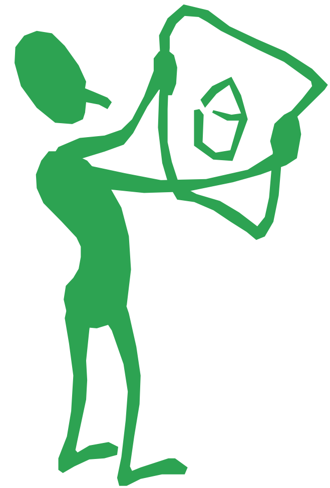
	10.000	50.000	100.000	500.000
Insertion	0,34	9,38	37,82	918
Selection	0,70	18,19	73,65	1740
Bubble	0,88	22,24	98,27	2068
Shell Sort	0,01	0,04	0,06	0,32
Quick Sort	0,01	0,02	0,04	0,23
Heap Sort	0,01	0,02	0,05	0,27

Array size

Execution time in seconds

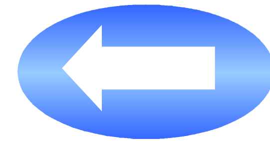
Analytical or a priori method

- It consists of mathematically calculating the amount of resources consumed by the algorithm according to the problem size
- Advantages
  - It consists of analysing the algorithm instead of the program → the results are independent from the machine or the language
  - The mathematical expression for the execution time depends on the problem size
  - The algorithms are not programmed



Topics

1. Efficiency of an algorithm
2. Methods to calculate the efficiency
3. **Temporary cost of an algorithm**
4. Cost analysis by step counting
5. Exercises



Problem size

- The problem size is any parameter from which the problem complexity can be expressed
  - It is generally related to the volume of input data to be managed
- Examples

Problem	Problem size
Array sorting	Amount of elements in the array
Searching an element in a two-dimensional array	Amount of elements in the array (=rows*columns)
Calculate the factorial of a number	The value of the number

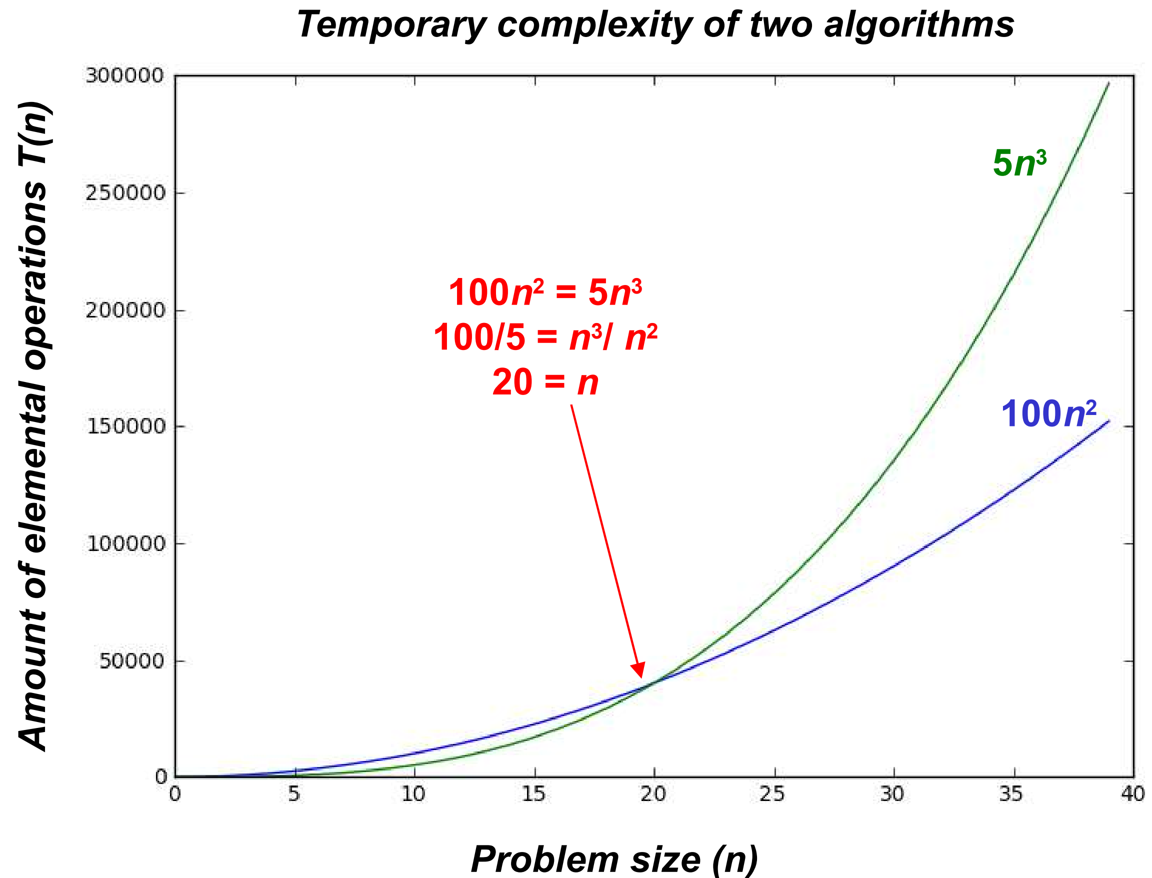
Calculating the temporal complexity of an algorithm

- **T(n)**: Function of the problem size. It returns the **execution time** of an implementation of the algorithm for a given problem size n
- T(n) is calculated counting the number of elemental operations or program steps done by the algorithm
- The result of T(n) is so independent from the specific machine

Example

- Which algorithm is more efficient for the same problem, if $T_1(n) = 100n^2$ and $T_2(n) = 5n^3$?

n	$100n^2$	$5n^3$
1	100	5
…	…	…
10	10000	5000
…	…	…
20	40000	40000
…	…	…
30	90000	135000
…	…	…
40	160000	320000
…	…	…



Asymptotic analysis

- It is the study of complexity for large problem sizes
- It allows the comparison of temporary cost of the algorithms according to their magnitude order of complexity
- The temporary complexity is analysed independently from the computer speed where their implementation is run
- The complexity functions that only differ from a constant factor are considered identical in terms of temporary cost (asymptotic criterium)

Complexity order of an algorithm

- An algorithm requires a **time of order *T(n)*** to be executed, if a positive constant *c* exists and there is an implementation of the algorithm so that it can solve all the cases of size *n* in a time ≤ *cT(n)*
- *T(n)* depends on the algorithm, *c* depends on the implementation
- Example
  - The execution time of an algorithm is

    $$T(n) = 32n^2 + 78n + 54$$

  - As $n \leq n^2$ and $1 \leq n^2$ for every value of $n \geq 1$

    $$T(n) = 32n^2 + 78n + 54 < 32n^2 + 78n^2 + 54n^2 = 164n^2$$

  - This algorithm has a **quadratic** execution time, that is, in n^2 order

Complexity orders and efficiency

- The most common complexity orders, sorted from higher to lower efficiency, are:

Order	*T(n)*
logarithmic	log(n)
linear	n
quasi-lineal	n log(n)
quadratic	n^2
polynomial (a > 2)	n^a
exponential (a ≥ 2)	a^n
factorial	n!

Exercise

- We can use a computer during 1000 seconds to solve a given problem. 4 different algorithms can be executed, in which execution times in seconds are $T(n)=100n$, $T(n)= 5n^2$, $T(n)=n^3/2$ and $T(n)=2^n$

Which is the maximum problem size that can be solved for each algorithm?

T(n)	n_1
100n	10
$5n^2$	14
$n^3/2$	12
2^n	10

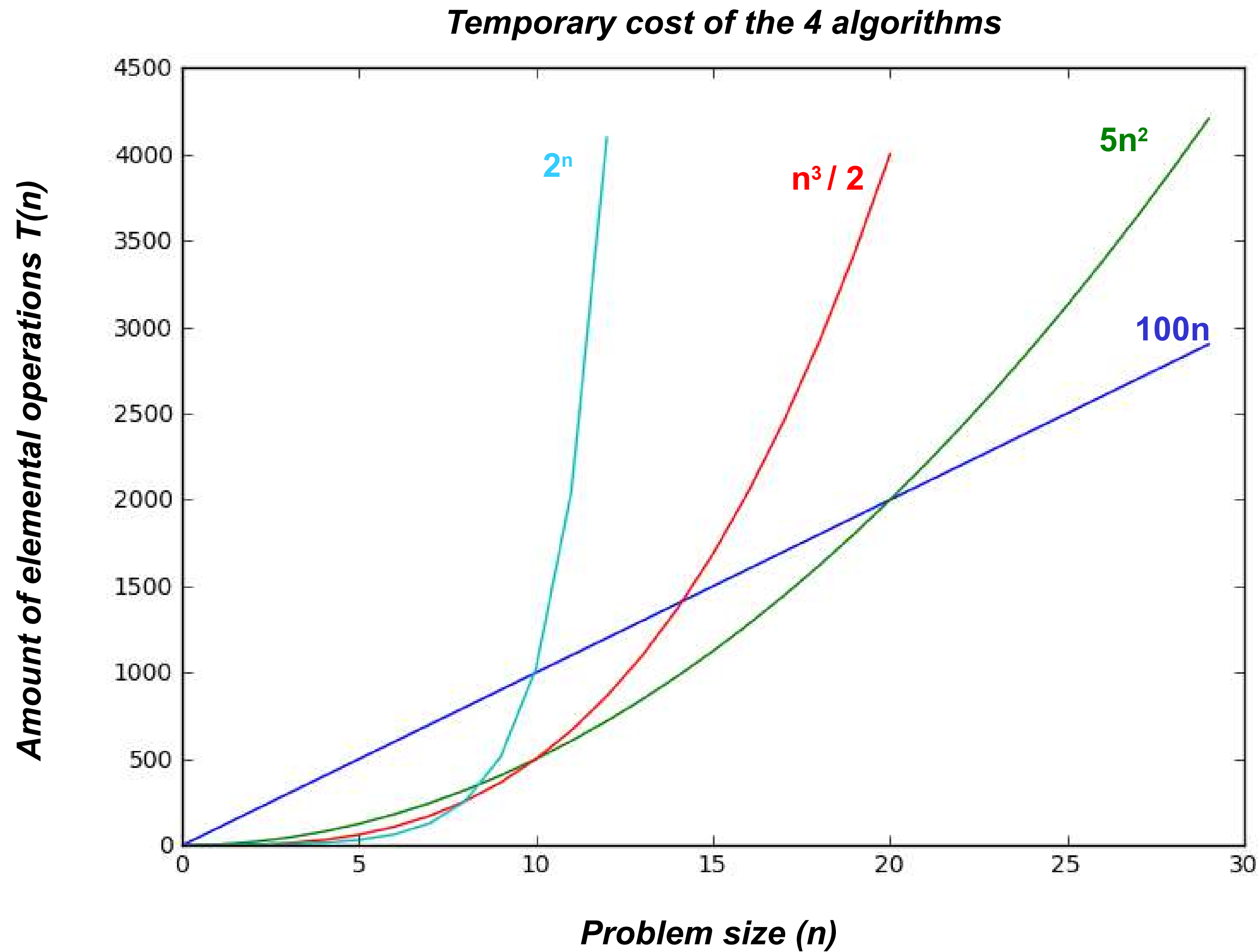
If the computer speed is increased 10 times, which are the sizes now?

T(n)	n_2
100n	100
$5n^2$	45
$n^3/2$	27
2^n	12

As long as the computer speed is increased, how do the problem sizes increase?

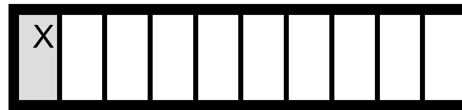
T(n)	n_2 / n_1
100n	10
$5n^2$	3,2
$n^3/2$	2,3
2^n	1,3

Temporary cost for several complexity orders

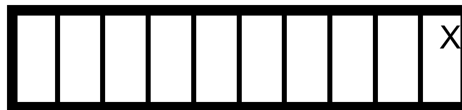


Worst case, best case and average case

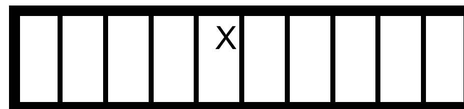
- **Worst case**: when the higher amount of elemental operations are needed
- **Best case**: when the lower amount of elemental operations are needed
- **Average case**: when the amount of elemental operations is the expected value (the probability distribution of the input data must be known)
- Example: sequential search of an element in an array
  - Best case: the element is in the first position (one comparison)



  - Worst case: the element is in the last position (n comparisons)



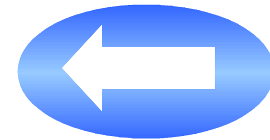
  - Average case: the element is in a middle position (n/2 comparisons)



Usually (we will do so in P1), in the analytical method for studying the complexity of an algorithm, the most important is the **worst case**. Then, the asymptotic criterium is used to obtain the temporary complexity, expressed in terms of the **mathematical notation O( )**

Topics

1. Efficiency of an algorithm
2. Methods to calculate the efficiency
3. Temporary cost of an algorithm
4. **Cost analysis by step counting**
5. Exercises



Program steps

- A program step is an elemental operation that is performed in an algorithm
- To analyse the temporary cost of an algorithm, the function T(n) is calculated **counting the amount of program steps** done by this algorithm

Program steps: elemental operations

- The following are the elemental operations and the amount of steps

Statement	Steps
Assignment statement ***S;***	Cost(S) = 1 step
Input statement ***S;***	Cost(S) = 1 step
Output statement ***S;***	Cost(S) = 1 step
Return statement in a function ***S;***	Cost(S) = 1 step
Logic expression ***E*** (not included in the previous statements)	Cost(E) = 1 step
Arithmetic expression ***E*** (not included in the previous statements)	Cost(E) = 1 step

Program steps: control statements

Statement	Steps
Sequence of statements ***{S_1; S_2}***	Cost(S_1) + Cost(S_2)
Selection statement ***if (condition) {S_1} else {S_2}***	Cost(condition) + MAXIMUM (Cost(S_1),Cost(S_2))
Loop with initial condition ***while (condition) {S}***	[Cost(condition) + Cost(S)] * num_iterations + Cost(condition)
Loop with final condition ***do {S} while (condition)***	[Cost(S) + Cost(condition)] * num_iterations
Counter-controlled loop ***for (init; cond; incr) {S}***	Cost(Init) + [Cost(cond) + Cost(S) + Cost(incr)] * num_iterations + Cost(cond)
Module call from an elemental operation ***module_name (...)***	Cost(module) + 1

Example 1: calculate the temporary cost

```
main() {
   int  a, n, c;

   cin >> n;              ---------> 1
   a = 1;                 ---------> 1
   while (a <= n) {
      C = Calculate(n);
      a = a+1;            ---------> 1
   }
}

int Calculate(int n) {
   int cal, i;

   cal = 1;               ---------> 1
   for (i=1; i <= n; i++) {
      cal := cal * n;     ---------> 1
   }
   return(cal);           ---------> 1
}
```

T(n) = Cost(main) = 2 + Cost(while) = $2+3n^2+7n+1 = 3n^2+7n+3 \rightarrow O(n^2)$

Cost(while) = [1+1+Cost(Calculate)+1]*n+1 = [1+1+3n+4+1]*n+1 = $3n^2+7n+1$

Cost(Calculate) = 2 + Cost(for) = 2+3n+2 = 3n+4

Cost(for) = 1+[1+1+1]*n +1 = 3n+2

Example 2: calculate the temporary cost

```
int Binary_search(int name_array[], int elem)  {
   int      pos_init, pos_end, pos_middle;
   bool  found;

   pos_init = 0;
   pos_end = SIZE_MAX -1;
   found = false;
   while (pos_init <= pos_end   &&   ! found) {
         pos_middle = (pos_init + pos_end) / 2;
         if  (elem == name_array[pos_middle])
               found = true;
         else if (elem > name_array[pos_middle] )
               pos_init = pos_middle +1;
         else
               pos_end = pos_middle -1;
   }
   if (! found)
        pos_middle = -1;

   return(pos_middle);
}
```

T(n) = Cost(Search) = 4 + Cost(while) + Coste(if) = 4+ 4log(n)+1 +2 = 4log(n) +7 → O(log(n))

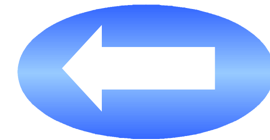
Cost(while) = [1+(1+Cost(if-elseif)]*log(n)+1 = [1+1+2]*log(n)+1 = 4log(n) +1

Cost(if-elseif) = maximum(2,2,2) = 2

Cost(if) = 2

Topics

1. Efficiency of an algorithm
2. Methods to calculate the efficiency
3. Temporary cost of an algorithm
4. Cost analysis by step counting
5. **Exercises**



Exercise 1: calculate the temporary cost

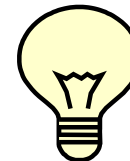
```
main() {
   int  n;

   do {
      cout << " Enter a number:";
      cin >> n;
      cout << "The result is " << Test(n);
   } while (n > 0);
}

int Test(int n) {
   int i, acu;

   acu = 0;
   i = 1;
   while (i <=n ) {
      if (n % i = = 0)
         acu = acu+1;
      i = i+2;
   }
   return(acu);
}
```

i=i+2 ➔ the loop performs n/2 iteractions



Exercise 2: calculate the temporary cost

```
main()
{
   int row, col, n, m;

   cout << "Enter the number of rows:";
   cin >> n;
   cout << “Enter the number of columns:”;
   cin >> m;
   for (row=1; row<=n; row++) {
       for (col=0; col<m; col++) {
           if (row = = 1  ||  row = = n  ||  col = = 0 ||  col = = m-1)
              cout << “@”;
           else
              cout << “*”;
       }
       cout << endl;
   }
}
```

Exercise 3: calculate the temporary cost

```
void Multiply_Matrices(int m1[N][N], int m2[N][N], int mres[N][N])
{
   int  i, j, k;

   for (i=0; i < N; i++) {
      for (j=0; j < N; j++) {
         mres[i][j] = 0;
         for (k=0; k < N; k++) {
            mres[i][j] = mres[i][j] + (m1[i][k] * m2[k][j]);
         }
      }
   }
}
```

Exercise 4: calculate the temporary cost

```
main() {
   int  a, b, c;

   cin >> c;
   do {
      c = c-1;
      cin >> a;
      b = Nd(a);
      cout << "answer = ", b;
   } while (c > 0);
}


int Nd(int m) {
   res = 0;
   n = 0;
   while (n <=m) {
      n = n+1;
      res = res + 10;
   }
   return(res);
}
```

Exercise 5: calculate the number of iterations of the loop

- Consider that n>0

Loop_1

```
i = 1;
do {
      i = i+1;
} while (i < n);
```

Loop_2

```
i = 0;
do {
      i = i+1;
} while (i < n);
```

Loop_3

```
i = n;
do {
      i = i/2;
} while (i > 0);
```

Loop_4

```
i = 2;
while (i <= 1) {
      i = i-1;
}
```

Loop_5

```
i = n-1;
while (i > 1) {
      i = i-1;
}
```

Loop_6

```
i = n;
while (i > 0) {
      i = i-2;
}
```

Loop_7

```
for (i=0; i <= n; i++) {
      cout << endl;
}
```

Loop_8

```
for (i=1; i < n; i++) {
      cout << endl;
}
```

Loop_9

```
for (i=1; i <= n; i--) {
      cout << endl;
}
```