Elementary Algorithmics

1. Introduction: Begin detailed study of algorithms.

2. Problems and Instances:
   - In the multiplication example (12, 26) is an instance. (-12, 26) is not, (12.5, 17) is not. Most interesting problems have an infinite collection of instances, some don't eg. “Perfect chess game.”

   - Algorithm can be rejected on the basis of a single wrong result. It is difficult to prove the correctness of an algorithm. To make it possible, specify the domain of definition, that is the set of instances to be considered.

   - Since any real computing device has a limit on the size of the instances it can handle (storage, numbers too large, ...) this limit cannot be attributed to the algorithm we choose to use. In this class we'll ignore the practical limitations, and we'll be content to prove that our algorithms in the abstract.

3. Efficiency of Algorithms:
   - When solving a problem there may be several suitable algorithms are available. Obviously choose the best. How to decide? If we had one or two instances (may be small too), simply choose the easiest to program or use the one for which a program exists. Otherwise, we have to choose very carefully.

   - The empirical (or a posteriori) approach to choosing an algorithm consists of programming the computing techniques and trying them on different instances with the help of a computer. The theoretical (or a priori) approach, which we favor consists of determining mathematically the quantity of resources needed by each algorithm as function of the size of the instances considered. The resources of most interest are: computing time (most critical) and storage space. Throughout the class when we speak of the efficiency of an algorithm, we shall mean how fast it runs. Occasionally, we'll also be interested in an algorithm's storage requirements or any other resources (like number of processors required by a parallel algorithm).

   - The size of an instance corresponds formally to the number of bits needed to represent the instance on a computer. In our analysis, we'll be less formal, the use of “size” to mean any integer that in some way measures the number of components in an instance.

     Examples:
     - In sorting size is the number of elements to be sorted.
     - In graphs size is the number of nodes or edges or both.

   - The advantage of the theoretical approach is that it doesn't depend on the computer used, nor the programming language, nor the skill of the programmer. There is the hybrid approach where efficiency is determined theoretically, then any required numerical parameters are determined empirically for a particular program and a machine.

- Bits are a natural way to measure storage. We want to be able to measure efficiency in terms of time it takes to arrive at an answer. There is no such obvious choice. An answer is given by the principle of invariance, which states that: "Two different implementations of the same algorithm will not differ in efficiency by more than a multiplicative constant." Different implementations mean different machines or programming language.

  If two implementations of the same algorithm take $t_1(n)$ and $t_2(n)$ *seconds, respectively, to solve an instance of size n, then there exists always positive constants c and d such that:*

  $t_1(n) \le c\,t_2(n)$ *and* $t_2(n) \le d\,t_1(n)$ *whenever n is sufficiently large.*

- In other words the running time of either implementation is bounded by a constant multiple of the running time of the other.

- Thus a change of machine may allow us to solve a problem 10 times faster or 100 times faster, giving an increase in speed by a contant factor. A chage of algorithm on the other hand may give us an improvement that gets more and more marked as the size of the instances increases.

- The principle allows us to decide that there will be no such unit. Instead, we only express the time taken by an algorithm to within a multiplicative constant. We say: *"An algorithm for some problem takes a time in the order of* $t(n)$, *for a given function t, if there exists a positive constant c, and an implementation of the algorithm capable of solving every instance of size n in not more than* $c\,t(n)$ *(seconds) arbitrary time units."*

Units are irrelevant $c\,t(n)$ seconds, $b\,t(n)$ years, ...
Certain orders are so frequent that we give them names:

cn : Algorithm in the order of n and called linear algorithm
cn^2 : Algorithm in the order of n^2 and called quadratic
n^k : Algorithm in the order of n^k and called polynomial
c^n : Algorithm in the order of c^n and called exponential

Don't ignore the multiplicative constant or sometimes called "hidden contant", and you may think linear is faster than polynomial, ..., etc. Consider the following example:

Given two algorithms to be run on the same machine. One takes n^2 *days, the other* n^3 *seconds for an instance n. On instances requiring 20 million years plus to solve that the quadratic outperforms the cubic. From a theoretical point of view we prefer (asymptotically) the quadratic, in practice it's the opposite. It is the hidden constant that made the quadratic unattractive.*

4. Average and Worst Case Analysis:

Time taken or storage used by an algorithm for 2 different instances of the same size can vary considerably. To illustrate this, consider the following 2 different cases:

```
procedure insert(T[1..n])
    for i = 2 to n do
        x = T[i]
        j = j – 1
        while j > 0 and x < T[j] do
            T[j + 1] = T[j]
            j = j -1
        T[j + 1] = x
```

```
9   6   8   10   7
6   9   8   10   7
6   8   9   10   7
6   8   9   10   7
6   7   8    9   10
```

```
procedure select(T[1..n])
    for i = 1 to n-1 do
        min j = i
        min x = T[i]
        for j = i + 1 to n do
            if T[j] < min x then
                min j = j
                min x = T[j]
        T[min j ] = T[i]
        T[i] = min x
```

```
9   6   8   10   7
6   9   8   10   7
6   7   8   10   9
6   7   8   10   9
6   7   8    9   10
```

Let U and V be 2 arrays of n elements, where U is sorted in ascending order and V in descending order.

V represents the worst case possible for these 2 algorithms. Nonetheless, **select** algorithm is not too sensitive to original order of array to be sorted: “if T[j] < min x” is executed the same number of times in every case the variation in execution time is only due to the number of times the assignments in the “then” part of this test are executed. When the test was conducted on a machine, results showed that no more than 15% variation from one case to another whatever the initial order of the elements to be sorted. We'll show that “**select** T” is quadratic regardless of T[n].

In **insert**ion the situation is different for both U and V, in U the condition controlling *while* is always false, in V the *while* is executed i -1 times for each i. The variation between the 2 is considerable, and increases with n.

> *Insert 5000 elements takes 1/5 seconds to sort an array already in ascending order and it takes 3.5 minutes for an array in descending order.*

For this reason we consider the worst case of the algorithm, and previously we said “an algorithm must solve every instance of size 'n' in no more than $c\,t(n)$ seconds for an appropriate c that depends on the implementations. If it is to run in a time in the order of $t(n)$: we implicitly had the worst case in mind.

Worst case analysis is appropriate for crucial controlling machine (power plant...). If the algorithm is to be used many times on many different instances it may be more important to talk about average execution time on instances of size n.

Insertion time varies from n to n^2 . There are $n!$ possible arrays of size n. If all are equally likely then average time is also in the order of n^2 . Later on we'll see how another sort algorithm, Quicksort takes $n \log n$ on average.

Average cases are harder to analyze than worst cases. Assumptions like arrays to be equally probable is unrealistic.

Example: update an array that is partially sorted

A useful analysis of the average requires some apriori knowledge of the distribution of the instances to be solved. This may be unrealistic and unpractical.

5. Elementary Operation:
An elementary operation is one whose execution time can be bounded above by a constant depending only on the implementation used – the machine, the programming language, ... and so on. Thus, the constant doesn't depend on the other parameters of the instance being considered. Since we're concerned with the execution times defined to within a multiplicative constant, it's only the number of elementary operations that matters in the analysis.

Example: an instance in an algorithm requires
a: addition, m: multiplication, s: assignment instructions
suppose t_a, t_m, t_s *micro seconds* is the time requirement for each operation. Total time required can be bounded by: $t \leq at_a + mt_m + st_s \leq max(t_a, t_m, t_s) * (a+m+s)$
since exact time is unimportant, we simplify by saying that elementary operations can be executed at unit cost.

A single line of a program may correspond to a number of elementary operations.

Example: $x \leftarrow min\{T[i] \mid i \leq i \leq n\}$ and this is an abbreviation of

$x \leftarrow T[i]$
for $i \leftarrow 2$ to n do
  if $T[i] < x$ then $x \leftarrow T[i]$

Some operations are too complex to be elementary, like: $n!$, check for divisibility... etc. Otherwise Wilson's theorem $(n \text{ divides } (n-1)!+1 \text{ iff } n \text{ is a prime})$ would let us check for primality with astonishing efficiency.

6. Why Efficiency?

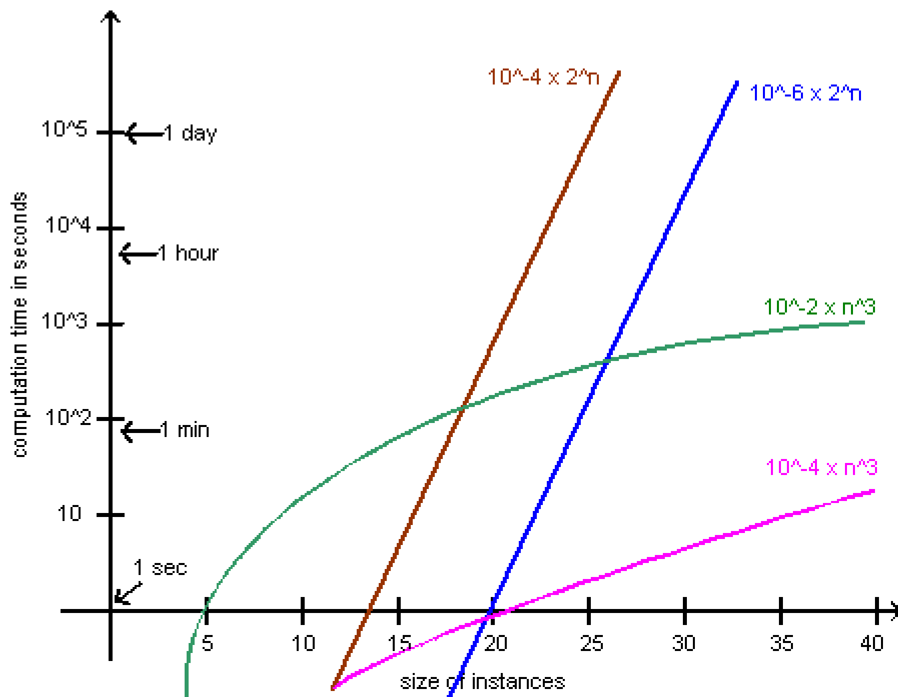
Why not simply buy better equipment for faster results. This is out of the question. Look at the following example: Assume you have an exponential algorithm that run instances of size *n* on a computer in $10^{-4} x 2^n$ seconds.

Size n = 10 solved in 1/10 second.
Size n = 20 solved in about 2 minutes.
Size n = 30 solved in about more than a day.
Size n = 38 solved in about one year (non-stop).

Assume you want to solve larger instances, and you invest money on a system 100 times faster. $10^{-6} x 2^n$ seconds. For one year you can solve instances of size 45. Previously solve *n* now $n + \log 100 \approx n + 7$

Suppose you decide to invest on algorithm and the new algorithm is $10^{-2} x n^3$ seconds.

$n = 10 \rightarrow$ time = 10 seconds
$n = 20 \rightarrow$ between 1 and 2 minutes
$n = 30 \rightarrow$ 4.5 minutes
$n = 200 \rightarrow$ one day
$n = 1500 \rightarrow$ one year



The new algorithm offers a factor $\sqrt[3]{100} \approx 4$ and 5

7. Examples:

- Calculating determinants: Recursive algorithm takes a time proportional to $n!$ for $n x n$ matrix (worse than exponential) but Gauss-Jordan method takes time proportional to n^3 .
  Recursive: $10x10$ takes 10 minutes, $20x20$ takes approximately 10 million years.
  Gauss-Jordan: $10x10$ takes $1/100$ seconds, $100x100$ almost 5.5 seconds

- Multiplication of large integers: in the order of $m \cdot n$ using divide and conquer approach $m \cdot n^{0.59}$

- Calculating greatest common divisor:

  function gcd(m, n)

  $i \leftarrow min(m, n)$

  $repeat\ i \leftarrow i-1\ until\ i$

  divides m and n exactly

  return i

  This algorithm is in the order of *n*. But Euclid's algorithm is more efficient and in the order of $\log n$.