Computing Fundamentals
Computational Complexity

Salvatore Filippone

salvatore.filippone@uniroma2.it

2014–2015

Computational Complexity

An obvious question in computer science is the following:

How much does it cost to solve a given problem?

This is the topic of *computational complexity*, and it translates into two main questions:

1. How long will a program run (until completion)?
2. How much memory will a program need to be able to run?

Accordingly we talk of *time* and *space* complexity.
Tipically, we consider various *instances* of a class of problems, each instance having a *size*; as an example, consider computing

$$z = [1, 2, 3] + [4, 5, 6];$$

which is a specific instance of the problem of summing two vectors of size 3, which in turn is a special case of the general problem of computing the sum of two vectors of arbitrary size.

Computational Complexity

To compute how long a program will run we need to

> *Figure out all the operations that will be executed by the program given a problem instance, and how long each of them will take*

In practice, when doing estimates, we often:

- Select and count only a subset of operations (that we consider relevant);
- Assume all such operations take the same amount of time;

These assumptions are only a first-order (sometimes rather crude) approximation, but they are sufficient for the purposes of this course. They also imply we are only examining the *algorithm* (i.e. the logical sequence of operations), while ignoring such details as the particular processor, interpreter, program etc.
In many cases we will count *arithmetic operations on floating-point data* (unless noted otherwise).

Computational Complexity

We will use an asymptotic notation:

> *Given an algorithm to solve a problem we will define its running time* $T(n)$ *to be* $O(f(n))$ *if there are constants* c *and* n_0 *such that*
>
> $$T(n) \leq c \cdot f(n), \qquad \text{for all } n \geq n_0$$
>
> *and the bound is valid for all problem instances of size* n

Note that the actual running time of an algorithm is not necessarily the same for all instances of size n: we have implicitly defined above the *worst case* complexity by asking the upper bound to be valid for all instances.
We may also define an *average case* complexity over all instances with the same size n.
As an example: summing two n-vectors always takes n arithmetic operations, but sorting an n-vector depends on the contents.

Computational Complexity

Special cases:

- $O(1)$: an algorithm taking constant time, independently of size;
- $O(\log(n))$: a logarithmic algorithm;
- $O(n)$: a linear algorithm;
- $O(n^k)$: a polynomial algorithm;
- $O(a^n)$: an exponential algorithm.

Polynomial algorithms are considered *tractable*.
By and large, we should prefer algorithms with a better (i.e. lower) asymptotic bound: An $O(n^2)$ algorithm will eventually overtake an $O(n^3)$ algorithm, even if the leading coefficient is larger.
Note: we are always looking for *tight* bounds.

Computational Complexity

n	$O(n)$	$O(n\log(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1.0000e+00	0.0000e+00	1.0000e+00	1.0000e+00	2.0000e+00
2	2.0000e+00	4.0000e+00	4.0000e+00	8.0000e+00	4.0000e+00
4	4.0000e+00	1.6000e+01	1.6000e+01	6.4000e+01	1.6000e+01
8	8.0000e+00	4.8000e+01	6.4000e+01	5.1200e+02	2.5600e+02
16	1.6000e+01	1.2800e+02	2.5600e+02	4.0960e+03	6.5536e+04
32	3.2000e+01	3.2000e+02	1.0240e+03	3.2768e+04	4.2950e+09
64	6.4000e+01	7.6800e+02	4.0960e+03	2.6214e+05	1.8447e+19
128	1.2800e+02	1.7920e+03	1.6384e+04	2.0972e+06	3.4028e+38
256	2.5600e+02	4.0960e+03	6.5536e+04	1.6777e+07	1.1579e+77
512	5.1200e+02	9.2160e+03	2.6214e+05	1.3422e+08	1.3408e+154
1024	1.0240e+03	2.0480e+04	1.0486e+06	1.0737e+09	Inf
2048	2.0480e+03	4.5056e+04	4.1943e+06	8.5899e+09	Inf
4096	4.0960e+03	9.8304e+04	1.6777e+07	6.8719e+10	Inf
8192	8.1920e+03	2.1299e+05	6.7109e+07	5.4976e+11	Inf
16384	1.6384e+04	4.5875e+05	2.6844e+08	4.3980e+12	Inf
32768	3.2768e+04	9.8304e+05	1.0737e+09	3.5184e+13	Inf
65536	6.5536e+04	2.0972e+06	4.2950e+09	2.8147e+14	Inf
131072	1.3107e+05	4.4564e+06	1.7180e+10	2.2518e+15	Inf
262144	2.6214e+05	9.4372e+06	6.8719e+10	1.8014e+16	Inf
524288	5.2429e+05	1.9923e+07	2.7488e+11	1.4412e+17	Inf
1048576	1.0486e+06	4.1943e+07	1.0995e+12	1.1529e+18	Inf

Computational Complexity

An example: the Discrete Fourier Transform (DFT)

$$F(j) = \sum_{k=0}^{N-1} f(k) e^{-\frac{kj}{N} 2\pi i}, \qquad j = 0, \ldots, N-1.$$

As defined it costs $O(N^2)$; in 1965 Cooley and Tukey discovered an $O(N \log(N))$ algorithm called the Fast Fourier Transform (FFT).
Things that would *NOT* exist without the FFT:

- CD;
- JPEG;
- DVD;
- Digital TV;
- Cell phones;
- Digital controls (ABS, ESP, Common-rail injection);
- ...

Computational Complexity

A few caveats:

- If we are handling small problem instances, maybe the $O(n^3)$ algorithm is better: $5n^3 < 100n^2$ for all n less than 20; some "optimal" algorithms are good only on astronomically large inputs, and are thus useless in practice;
- If a program is only going to be used once or twice, the time to write it becomes important: if the slow algorithm is easier to code, it may be better;
- In some cases the fastest algorithm takes too much space;
- In some cases an algorithm may be the best in the *average* case and at the same time very bad in the *worst* case (e.g. quicksort).

And finally: Never try to improve a program without actually knowing (and measuring) its performance

Premature optimization is the root of all evil *D. Knuth*

Computational Complexity

How to compute an instruction count:

- Simple scalar statements have a simple $O(1)$ cost;
- The cost of a sequence is the sum of the individual costs;
- The cost of a loop is the sum over all iterations of the cost of each iteration, possibly including the cost to test for termination;
- Array statements have a cost that can be understood by expanding them into equivalent loops;
- A conditional statement has a worst-case cost that is the largest of the `if` and `else` parts; to get the *average* cost we need to multiply the two branches by the probability that each is taken; all this plus the cost of the branching condition.

These simple rules can get us very far, but the devil (as usual) is in the details.

Statements involving scalar quantities.

```
a = 2.5;      % 0 or 1: Cost of assignment is often ignored;

b = a*a+1;    % Here we have 2 floating point operations;
c = b^3;      % b^3 is b*b*b, so again 2 operations;

for k=n1:n2 % This is executed (n2-n1+1) times
    c=a+b     % Cost here is 1 independent of K
end           % total cost: 1*(n2-n1+1)

if (mod(k,2) == 0) % If K is a random integer 50% prob.
    c=a*b+c;       % worst case is IF branch of cost 2
else               % average case costs 1.5
    b=b+1;         % plus 2 for evaluating (MOD()==0)
end
```

Computational Complexity

Loops: to figure out cost must compute

$$\sum_{i \in \mathcal{I}} C(i)$$

- i is an iteration;
- $\mathcal{I}$ is the set of all iterations;
- $C(i)$ is the cost of the i-th iteration.

Often (but not always) the cost per iteration is constant; determining $\mathcal{I}$ is easy for `for` loops, may be nontrivial for `while` loops. To nested loops there correspond multiple sums:

```
for i=1:n
  for j=1:k
   <statement>
  end
end
```

$$Opcnt = \sum_{i=1}^{n} \sum_{j=1}^{k} C(\text{statement}_{ij}).$$

Vector (scaled) sums of size n: `c = a + alpha*b;`

```
for i=1:n
  c(i) = a(i) + alpha*b(i);
end
```

Cost:

$$\sum_{i=1}^{n} 2 = 2 \sum_{i=1}^{n} 1 = 2n$$

Scalar product size n: `c = x*y;`

```
c = 0
for i=1:n
    c = c+x(i)*y(i);
end
```

Cost:

$$\sum_{i=1}^{n} 2 = 2 \sum_{i=1}^{n} 1 = 2n$$

Matrix-vector products (matrix $m \times n$): `y = y + A*x;`

```
for i=1:m
    for j=1:n
        y(i) = y(i) + A(i,j)*x(j);
    end
end
```

Cost:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} 2 = \sum_{i=1}^{m} 2 \sum_{j=1}^{n} 1 = \sum_{i=1}^{m} 2n = 2n \sum_{i=1}^{m} 1 = 2mn$$

Matrix-matrix products (matrix $m \times k \times n$): `C = C + A*B;`

```
for i=1:m
  for j=1:n
    for p=1:k
      C(i,j) = C(i,j) + A(i,p)*B(p,j);
    end
  end
end
```

Cost:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{p=1}^{k} 2 = 2mnk$$

Triangular linear systems

If a coefficient matrix is lower triangular it is easy to solve $Lx = b$ by forward substitution:

Triangular linear systems

If a coefficient matrix is lower triangular it is easy to solve $Lx = b$ by forward substitution:

```
1   n=size(L,1);
2   for i=1:n
3       x(i) = b(i) - L(i,1:i-1)*x(1:i-1);
4       x(i) = x(i) / L(i,i);
5   end
```

If the diagonal is unitary, the division steps can be skipped. The total number of operations executed is $\approx n^2$.

Triangular linear systems

What is the operation count?

- At each loop iteration $i = 1 \dots n$, we have a dot product of size $i - 1$ at step 3, plus one scalar add
- A dot product of size k costs $2k$ operations;
- At each loop iteration we have a division;

Note that each iteration has a different cost! Therefore:

$$\begin{aligned} \text{opcnt} = \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} (2(i-1)+1) &= (\sum_{i=1}^{n} 2) + 2(\sum_{i=1}^{n} (i-1)) \\ &= 2n + 2\sum_{i=0}^{n-1} i = 2n + 2\frac{(n-1)n}{2} = n^2 + n \end{aligned}$$

Useful tricks:

- $$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2} = O(n^2)$$
- $$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3} = O(n^3)$$
- Note that the leading term can be found with an integral

$$\sum_{i=0}^{n} i^2 \approx \frac{n^3}{3} = \int_0^n x^2 dx$$

- $$\sum_{i=n_1}^{n_2} 1 = n_2 - n_1 + 1$$

Searching & Sorting

Searching and Sorting are essential problems in computer science.

> *We have a collection of items (records) R_i each one with a key K_i, $i = 1, \ldots, n$, and we are given a key K: we need to find the index j of a record (if any) with $K_j = K$.*

This formulation of the problem lends immediately to the *sequential search* algorithm where we only need to check for equality $K_j = K$:

```
function res=search(key,v)
  % Sequential search
  k=1;
  found=false;
  res=-1;
  n=length(v);
  while ((k<=n)&&(~found))
    if (v(k)==key)
      found = true;
      res=k;
    end
    k=k+1;
  end
```

What is the runtime of sequential search? We must count the number of iterations through the loop.

- If the search is successful, we have performed exactly `res` iterations through the loop; if we don't know any better, we may assume a uniform distribution, which means on average $res = n/2$;
- If the search is unsuccessful, we have performed n iterations.

So, this algorithm is $O(n)$.
Can we do any better?

Suppose that the keys K admit an *order relation*:

1. Each pair of keys satisfies K_i, K_j satisfies exactly one of three relations (trichotomy): $K_i < K_j$ or $K_i = K_j$ or $K_j < K_i$;
2. If $K_i < K_q$ and $K_q < K_j$ then $K_i < K_j$ (transitivity).

Now we can ask: what happens if you happen to have a *sorted* set of records:

$$i < j \Rightarrow K_i < K_j$$

So:

> *Compare the search key K with the item in the middle: if the key is smaller, then an equal item can only be in the first half-vector, if it's larger it can only be in the second half-vector. An if it's equal you're done.*

(by application of transitivity).

```
function res=bsearch(k,v)
  % Binary search
  found=false;
  res=-1;
  n=length(v);
  first=1; last=n
  while ((first<=last)&&(~found))
    m = floor((first+last)/2)
    if (v(m)==key)
      found = true;
      res   = m
    elseif (key < v(m))
      last = m-1
    elseif (key > v(m))
      first = m+1
    end
  end
```

What is runtime of binary search?

- At each step we are searching a subvector `v(first:last)`, of size $last - first + 1$;
- In the worst case, we stop when $first > last$, i.e. empty vector;
- As we move from a step to the next, the length of the vector halves,

$$n \to \frac{n}{2} \to \frac{n}{2^2} \cdots$$

In other words: we are searching for the smallest i such that

$$\frac{n}{2^i} \leq 1,$$

or

$$i \geq \log_2(n).$$

Hence binary search is $O(\log(n))$. (But beware of argument copy!).

We see it's beneficial to have items sorted. How do we do it? First idea: insertion sorting. Take one item of the input at a time and put it in the right place in the output.

Searching & Sorting

We see it's beneficial to have items sorted. How do we do it? First idea: insertion sorting. Take one item of the input at a time and put it in the right place in the output.

```
function vs=insert(v)

  if (~((size(v,1)==1)||(size(v,2)==1)))
    error("V is not a vector");
  end
  trans=(size(v,2)~=1);
  if (trans)
    v=v';
  end
  vs=[];
  for x=v
    i=lsrch(x,vs);
    vs=[vs(1:i), x, vs(i+1:length(vs))];
  end
  if (trans)
    vs = vs';
  end
```

Hopelessly $O(n^2)$.

Searching & Sorting

We see it's beneficial to have items sorted. How do we do it? First idea: insertion sorting.

```
function vs=insert1(v)

  if (~((size(v,1)==1)||(size(v,2)==1)))
    error("V is not a vector");
  end

  n=length(v);
  for j=2:n
    i=j-1;
    xch=true;
    while ((i >= 1) && (xch))
      xch= (v(i) > v(i+1))
      if (xch)
        t=v(i);
        v(i)=v(i+1);
        v(i+1)=t;
      end
      i=i-1;
    end
  end
  vs=v;
```

Second strategy: merge sort.
Suppose you have two sorted sequences; then it is easy to see that a very simple pass looking at the first elements at each step will be enough to build a single sorted sequence containing all their elements. This is called *merge*
Then we have the *mergesort* algorithm:

- If the vector is of length 1 it's already sorted;
- Otherwise, call recursively on the first half, then on the second half, then merge the two sorted subvectors.

Main program

```
function b = mergesort(a)
% Listing 16-5 Merge sort
%  This function sorts a column array,
   if (~((size(a,1)==1)||(size(a,2)==1)))
     error("a is not a vector");
   end
   b=a;
   sz = length(b);
   if sz > 1
     szb2 = floor(sz / 2);
     first = mergesort(b(1 : szb2));
     second = mergesort(b(szb2+1 : sz));
     b = merge(first, second);
   end
   if ((size(a,2)==1)&&(size(b,2)>1))
      b=b';
   end
end
```

Second strategy: merge sort.

```
function b = merge(first, second)
  %   Merges two sorted arrays
  i1 = 1; i2 = 1;  out = 1;
  b=first;
% as long as neither i1 nor i2 past the end,
% move the smaller element into a
  while (i1 <= length(first)) && (i2 <= length(second))
    if lt(first(i1), second(i2))
      b(out) = first(i1); i1 = i1 + 1;
    else
      b(out) = second(i2); i2 = i2 + 1;
    end
    out = out + 1;
  end
% copy any remaining entries of the first array
  while i1 <= length(first)
    b(out) = first(i1); i1 = i1 + 1; out = out + 1;
  end
% copy any remaining entries of the second array
  while i2 <= length(second)
    b(out) = second(i2); i2 = i2 + 1; out = out + 1;
  end
end
```

Third strategy: quicksort

If all the items in the first half are less than all the items in the second half, then you can sort recursively the two halves and the result will be sorted.

Then we have the *quicksort* algorithm:

- If the vector is of length 1 it's already sorted;
- Otherwise, partition the vector into two halves such that all items in the first half are less than those in the second, then call recursively on both halves.

This is also called *partition-exchange* sorting.

Third strategy: quicksort

```
% Listing 16-3 - Quick sort
function a = quicksort(a)
%  This function sorts a column array,
%  using the quick sort algorithm
  if (~((size(a,1)==1)||(size(a,2)==1)))
    error("a is not a vector");
  end
  a=quicksorti(a,1,length(a));
end

function a = quicksorti(a, from, to)
  if (from < to)
    [a p] = partition(a, from, to);
%     from,p
%     p+1,to
    a = quicksorti(a, from, p);
    a = quicksorti(a, p + 1, to);
  end
end
```

Third strategy: quicksort

```
function [a lower] = partition(a, from, to)
%  This function partitions a column array
   pivot = a(from); i = from - 1; j = to + 1;
   while (i < j)
     i = i + 1;
     while lt(a(i), pivot)
       i = i + 1;
     end
     j = j - 1;
     while gt(a(j), pivot)
       j = j - 1;
     end
     if (i < j)
       temp = a(i); % swap
       a(i) = a(j); % a(i) with a(j)
       a(j) = temp;
     end
   end
   lower = j;
end
```

How do we count operations for recursive functions?
Let's model the behaviour:

> *Each instance of a recursive function is either a base instance, with a known cost, or it splits the problem of size n into (a) subproblems of size (n/b) whose solutions will be combined to build the overall solution*

How do we count operations for recursive functions?
Let's model the behaviour:

> *Each instance of a recursive function is either a base instance, with a known cost, or it splits the problem of size n into (a) subproblems of size (n/b) whose solutions will be combined to build the overall solution*

In formulae:

$$T(n) = \begin{cases} 1 & n = 1 \\ aT(\frac{n}{b}) + n^{\alpha} & n > 1 \end{cases}$$

The term n^{α} is assumed to measure the cost of splitting the problem and of combining the solutions of the subproblems.
Note: if we have $c \cdot n^{\alpha}$ we can easily define an auxiliary function $U = T/c$.

Unroll the recurrence for $n = b^k$:

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + n^\alpha \\
&= a\left(aT\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^\alpha\right) + n^\alpha = a^2 T\left(\frac{n}{b^2}\right) + a\left(\frac{n}{b}\right)^\alpha + n^\alpha \\
&= a^3 T\left(\frac{n}{b^3}\right) + a^2\left(\frac{n}{b^2}\right)^\alpha + a\left(\frac{n}{b}\right)^\alpha + n^\alpha \\
&= a^k T\left(\frac{n}{b^k}\right) + \sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j}\right)^\alpha \\
&= \sum_{j=0}^{k} a^j \left(\frac{n}{b^j}\right)^\alpha = n^\alpha \sum_{j=0}^{k} \left(\frac{a}{b^\alpha}\right)^j
\end{aligned}$$

having used:

$$T\left(\frac{n}{b^k}\right) = T(1) = 1 = 1^\alpha = \left(\frac{n}{b^k}\right)^\alpha$$

Case 1:

$$a > b^{\alpha};$$

in this case

$$\sum_{j=0}^{k} \left(\frac{a}{b^{\alpha}}\right)^{j} = \frac{\left(\frac{a}{b^{\alpha}}\right)^{k+1} - 1}{\frac{a}{b^{\alpha}} - 1};$$

asymptotically this can be approximated by

$$\sum_{j=0}^{k} \left(\frac{a}{b^{\alpha}}\right)^{j} \leq c \cdot \left(\frac{a}{b^{\alpha}}\right)^{k+1} = \frac{ac}{b^{\alpha}} \frac{a^{k}}{n^{\alpha}}$$

which gives

$$T(n) \leq n^{\alpha} \cdot \frac{ac}{b^{\alpha}} \frac{a^{k}}{n^{\alpha}} = \beta \cdot a^{k}$$

hence

$$T(n) = O(a^{k}) = O(a^{\log_b n}) = O(n^{\log_b a})$$

Case 2:

$$a < b^{\alpha};$$

in this case

$$\sum_{j=0}^{k} \left(\frac{a}{b^{\alpha}}\right)^{j} \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^{\alpha}}\right)^{j} = c < \infty,$$

hence

$$T(n) \leq n^{\alpha} \cdot c = O(n^{\alpha}).$$

Case 3:

$$a = b^{\alpha};$$

in this case we have

$$\sum_{j=0}^{k} 1 = k + 1 = O(k)$$

hence

$$T(n) = O(n^{\alpha} k) = O(n^{\alpha} \log_b n).$$

Example: merge-sort and quicksort. Both algorithms can be modeled by

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

Hence behaviour is

$$T(n) = O(n \log(n)).$$

Note: it is *essential* to split into equal halves; this is *guaranteed* for merge-sort, but only *statistically true* for quicksort. So: the *average* behaviour of quicksort is better, but the *worst case* behaviour is bad, whereas mergesort is always good.

Example: Matrix multiplication.
We know that $C = AB$ takes $2n^3$ operations, right?

Example: Matrix multiplication.
We know that $C = AB$ takes $2n^3$ operations, right?
Well, consider the problem of multiplying two 2×2 matrices

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The standard way to compute the product:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 multiplications and 4 additions.

Strassen (1968) discovered a formula later improved by Winograd

$$
\begin{aligned}
u &= (A_{21} - A_{11})(B_{21} - B_{22}) \\
v &= (A_{21} + A_{22})(B_{21} - B_{11}) \\
w &= A_{11}B_{11} + (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12}) \\
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= w + v + (A_{11} + A_{12} - A_{21} - A_{22})B_{22} \\
C_{21} &= w + v + A_{22}(B_{21} + B_{12} - B_{11} - B_{22}) \\
C_{22} &= u + v + w
\end{aligned}
$$

These are 7 multiplications and 15 additions; commutativity is not used, so the terms can be submatrices.

Hence matrix multiplication costs

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(\frac{n}{2}) + 15 * n^2 & n > 1 \end{cases}$$

but

$$7 > 4 = 2^2$$

hence

$$T(n) = O(n^{\log_2(7)}) = O(n^{2.8074})$$

Current record is $O(n^{2.376})$.