Lecture 1: Analyzing algorithms

A royal mathematical challenge (1202):

Suppose that rabbits take exactly one month to become fertile, after which they produce one child per month, forever. Starting with one rabbit, how many are there after n months?

Leonardo da Pisa, aka Fibonacci

The proliferation of rabbits

Rabbits take one month to become fertile, after which they produce one child per month, forever.

<table>
<thead>
<tr>
<th>Time</th>
<th>Fertile</th>
<th>Not fertile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td></td>
<td>🐇</td>
</tr>
<tr>
<td>One month</td>
<td>🐇</td>
<td></td>
</tr>
<tr>
<td>Two months</td>
<td>🐇</td>
<td>🐇</td>
</tr>
<tr>
<td>Three months</td>
<td>🐇</td>
<td>🐇</td>
</tr>
<tr>
<td>Four months</td>
<td>🐇</td>
<td>🐇</td>
</tr>
<tr>
<td>Five months</td>
<td>🐇</td>
<td>🐇</td>
</tr>
</tbody>
</table>
The Fibonacci sequence

\[ F_1 = 1, \quad F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \]

These numbers grow very fast: \( F_{30} > 10^6 \)!

In fact, \( F_n \approx 2^{0.694n} \approx 1.6^n \), exponential growth.

The Fibonacci sequence

\[ F_1 = 1, \quad F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \]

Can you see why \( F_n < 2^n \)?
Computing Fibonacci numbers

function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)

A recursive algorithm

Running time analysis

function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)

Exponential time... how bad is this?

Eg. Computing $F_{200}$ needs about $2^{140}$ operations. How long does this take on a fast computer?
IBM Summit

Can perform up to 200 quadrillion (\(= 200 \times 10^{15}\)) operations per second.

Is exponential time all that bad?

The Summit needs \(2^{82}\) seconds for \(F_{200}\).

<table>
<thead>
<tr>
<th>Time in seconds</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{10})</td>
<td>17 minutes</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>12 days</td>
</tr>
<tr>
<td>(2^{30})</td>
<td>32 years</td>
</tr>
<tr>
<td>(2^{40})</td>
<td>cave paintings</td>
</tr>
<tr>
<td>(2^{45})</td>
<td>\textit{homo erectus} discovers fire</td>
</tr>
<tr>
<td>(2^{51})</td>
<td>extinction of dinosaurs</td>
</tr>
<tr>
<td>(2^{57})</td>
<td>creation of Earth</td>
</tr>
<tr>
<td>(2^{60})</td>
<td>origin of universe</td>
</tr>
</tbody>
</table>
Post mortem

What takes so long?
Let’s unravel the recursion…

```
function Fib1(n)
    if n = 1 return 1
    if n = 2 return 1
    return Fib1(n-1) + Fib1(n-2)
```

```
F(n)
`/  \
/    \
F(n-1)   F(n-2)
`/  \
/    \
F(n-2)   F(n-3)   F(n-3)
`/  \
/    \
F(n-3)   F(n-4)   F(n-4)   F(n-4)
```

The same subproblems get solved over and over again!

A better algorithm

Subproblems: $F_1, F_2, \ldots, F_n$. Solve them in order and save their values!

```
function Fib2(n)
    Create an array fib[1..n]
    fib[1] = 1
    fib[2] = 1
    for i = 3 to n:
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

[1] Does it return the correct answer?
[2] How fast is it?
Polynomial vs. exponential

Polynomial running times:

Exponential running times:

To an excellent first approximation:
  polynomial is reasonable
  exponential is not reasonable

This is one of the most fundamental dichotomies in the analysis of algorithms.

A more careful analysis

```python
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

```python
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Problem: we cannot count these additions as single operations!

How many bits does \( F_n \) have?

Addition of \( n \)-bit numbers takes \( O(n) \) time.

\( \text{Fib1: } O(n 2^{0.7n}) \) time

\( \text{Fib2: } O(n^2) \) time
Addition

Adding two $n$-bit numbers takes $O(n)$ simple operations:

E.g. $22 + 13$:

```
[22]  1  0  1  1  0
[13]  1  1  0  1
```

Big-O notation

```
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Running time is proportional to $n^2$.

But what is the constant: is it $2n^2$ or $3n^2$ or what?

The constant depends upon:

The units of time – minutes, seconds, milliseconds,…

Specifics of the computer architecture.

It is much too hairy to figure out exactly. Moreover it is nowhere as important as the huge gulf between $n^2$ and $2^n$. So we simply say the running time is $O(n^2)$. 