Example:

input: road map, two locations A and B.
output: shortest path from A to B.

Our task: discover the algorithm.

Turing machine

definition:

algorithms

this course: "well-defined computational procedure that transforms an input into an output"
Focus in this course:

* correctness of algorithms
* does it terminate?
* efficient (= fast):
  - estimate running time
  - count the number of steps
    - what does this mean?
    - what is a "step"?
* limits of efficiency: some problems cannot be solved efficiently
* pseudocode, no programming
Example to illustrate this:
Fibonacci numbers

\[ F_0 = 0, \ F_1 = 1, \ \text{for } n \geq 2: \ F_n = F_{n-1} + F_{n-2} \]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

This definition leads to an algorithm that returns \( F_n \) for any given input \( n \geq 0 \):

```
Algorithm fib(n):
    if n ≤ 1 : return n
    else: return fib(n-1) + fib(n-2)
```

* correct? yes
* terminate? yes
efficient?

how does the running time depend on the input n?

Define \( T(n) = \# \text{ of steps when running} \ fib(n) \).

for \( n = 0 \) or \( n = 1 \):

- comparison "\( n \leq 1 \)" \( \{ \) 2 steps
- return value

for \( n \geq 2 \):

- comparison "\( n \leq 1 \)" : 1 step
- compute \( n-1 \) : 1 step
- call \( fib(n-1) \) : \( T(n-1) \) steps
- compute \( n-2 \) : 1 step
- call \( fib(n-2) \) : \( T(n-2) \) steps
- compute sum of 2 results : 1 step
\[ T(0) = 2 \]
\[ T(1) = 2 \]

for \( n \geq 2 \): \( T(n) = T(n-1) + T(n-2) + 4 \)

**Exercise**: Prove by induction that
\[ T(n) \geq F_n. \]

**: running time of algorithm \texttt{fib}(n) is**
\[ \geq F_n. \]

**But**: \( F_n \) is very large:

**Exercise**: Prove by induction that
\[ \forall_{n \geq 2} \quad F_n \geq 2 \quad \text{for} \quad n \geq 6 \]
\[ \star \quad F_n = \frac{1}{\sqrt{5}} \left( \frac{1+ \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1- \sqrt{5}}{2} \right)^n \quad \text{for} \quad n \geq 0 \]
\[ \text{fib}(n) \text{ takes at least exponential time} \]
\[ \text{fib}(200) \text{ will not terminate during our lifetime.} \]

Why is \text{fib}(n) so slow?

Recursion tree for \text{fib}(3):

```
      fib(3)
     /     \
   fib(2)   fib(1)
  /     \     /
fib(1)  fib(0) fib(1)
```

\[ \Rightarrow \text{fib}(1) \text{ is called twice} \]
Better algorithm:

Algorithm fib'(n):

if $n \leq 1$: return $n$

else: initialize array $f[0..n]$

$f[0] = 0$
$f[1] = 1$

for $i = 2$ to $n$: $f[i] = f[i-1] + f[i-2]$

return $f[n]$

correct? yes
terminate? yes

running time: \underline{linear} in $n$.

Conclusion:

fib(n): exponential (very slow)

fib'(n): linear (very fast)
But: is it realistic to say that fib'(n) takes a linear number of steps?

In our analysis: one step is:

\[
\{ \text{comparison, addition, subtraction} \}
\]

of very large numbers

In kindergarten, you learned:

two n-bit numbers can be added in a linear number of bit-operations

When running fib'(n):

\[ \approx n \text{ additions of numbers} \]

each of these numbers is \( \leq F_n \)

each of these numbers has \( \approx n \) bits

\[ \therefore \text{fib'}(n) \text{ makes a quadratic number of bit-operations.} \]
Exercise: Convince yourself that $\text{fib}(n)$ makes a number of bit-operations that is proportional to $n \cdot F_n$.

Asymptotic notation:

\[ f(n) = \Theta(g(n)) : \]
\[ \exists c \in \mathbb{R}, \forall n \geq n_0 : c \cdot g(n) \leq f(n) \leq c' \cdot g(n) \]

\[ f(n) = O(g(n)) : \]
\[ \exists c \in \mathbb{R}, \forall n \geq n_0 : f(n) \leq c \cdot g(n) \]

\[ f(n) = \Omega(g(n)) : \]
\[ \exists c \in \mathbb{R}, \forall n \geq n_0 : f(n) \geq c \cdot g(n) \]
0/ω/Θ:

* ignore constant factors
* focus on the dependency on n
* consider large values of n

Running time, in bit-operations, of

* \( \text{fib}(n) : \mathcal{O}(n \cdot F_n) \) : exponential
* \( \text{fib}'(n) : \mathcal{O}(n^2) \) : quadratic
I assume you know:

* induction
* $O, \Omega, \Theta$
* sorting algorithms:
  - insertion sort
  - selection sort
  - bubble sort
  \( O(n^2) \)
  - merge sort: \( O(n \log n) \)
  - quicksort:
    \( O(n^2) \) worst case,
    \( O(n \log n) \) "always"
* comparison-based sorting: \( \Omega(n \log n) \) for every algorithm
* balanced binary search trees (AVL, red-black, ...)
  - search, insert, delete: \( O(\log n) \)