universität
innsbruck

Discrete structures

Cezary Kaliszyk Raoul Schikora
Vincent van Oostrom

http://cl-informatik.uibk.ac.at

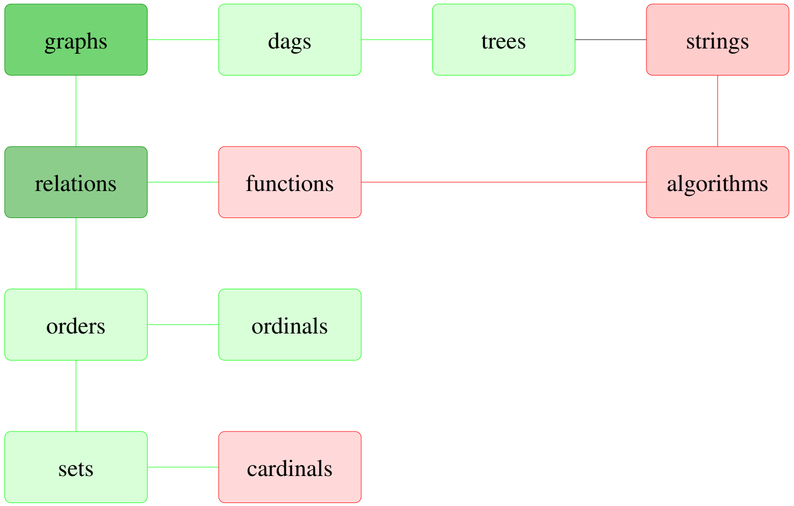
Summary last week

- ingredients of RSA continued (with proofs):
- Fermat's Little Theorem: $a^{p-1} \equiv 1 \pmod{p}$ if p prime, $p \nmid a$
- Euler's Theorem: $a^{(p-1)\cdot(q-1)} \equiv 1 \pmod{p \cdot q}$ if $\gcd(a, p \cdot q) = 1$
- Chinese remainder $\mathrm{crt} : x \mapsto (x \bmod p, x \bmod q)$ bijection if $\gcd(p, q) = 1$,
- 3 methods to compute inverse of crt given pair (a, b):
  - search $0 \leq x < p \cdot q$ mapped to (a, b) by crt (by bijection; brute force)
  - $x \equiv vqa + upb \pmod{pq}$ with u, v s.t. $up + vq = 1$ (by Bézout; for p, q)
  - $x \equiv a + p \cdot ((p' \cdot (b - a)) \bmod q) \pmod{p \cdot q}$ (by inverse modulo; $p \cdot p' \equiv 1 \pmod{q}$)
- recapitulation of motivation for (technical definition of) complexity; $O(n)$

Course themes

- directed and undirected graphs
- relations and functions
- orders and induction
- trees and dags
- finite and infinite counting
- elementary number theory
- Turing machines, algorithms, and complexity
- decidable and undecidable problem

Discrete structures



Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

$$\begin{aligned} T(n) &= 2\cdot T(\frac{n}{2}) + c\cdot n \\ &= 2\cdot(2\cdot T(\frac{n}{2^2}) + c\cdot\frac{n}{2}) + c\cdot n \\ &= 2^2\cdot T(\frac{n}{2^2}) + 2\cdot c\cdot n \\ &= 2^3\cdot T(\frac{n}{2^3}) + 3\cdot c\cdot n \\ &= \ldots \\ &= 2^k\cdot T(\frac{n}{2^k}) + k\cdot c\cdot n \end{aligned}$$

1. $T(n) = 2^k\cdot T(\frac{n}{2^k}) + k\cdot c\cdot n$ for $1 \leq k <? \log n$
2. base case $T(n) = c$ if $n = 2^k$, i.e. if $k = \log n$
3. set $k := \log n$. $T(n) = 2^{\log n}\cdot c + \log n\cdot c\cdot n = c\cdot n\cdot\log n + c\cdot n$; closed-form for $T(n)$
4. asymptotic complexity of solution: $T(n) \in O(n\cdot\log n)$

Verifying solutions/solving by guessing

Recall

- recurrence specifies unique function
- method: guess solution, verify solution by substitution/induction

Example

1. guess $f(n) = c\cdot n\cdot\log n + c\cdot n$ solves $T(n) = 2\cdot T(\frac{n}{2}) + c\cdot n$ if $n \geq 2$, c otherwise
2. verify by substituting guess f for T in recurrence: (may use induction)
   - case $n = 1$: $f(1) = c$ ✓
   - case $n > 1$:

$$\begin{aligned} T(n) = f(n) &= c\cdot n\cdot\log n + c\cdot n \\ &= 2\cdot(c\cdot\frac{n}{2}\cdot\log\frac{n}{2} + c\cdot\frac{n}{2}) + c\cdot n \\ &=_{\mathrm{IH}} 2\cdot T(\frac{n}{2}) + c\cdot n \quad ✓ \end{aligned}$$

using $\log(\frac{a}{b}) = (\log a) - (\log b)$, well-founded $<$-induction on n ($\frac{n}{2} < n$ if $n \geq 2$)

Lemma

Let $T\colon \mathbb{N} \to \mathbb{N}$ be defined by recurrence

$$T(n) = aT(\frac{n}{b}) + f(n)$$

with $a, b \in \mathbb{N}$ with $b > 1$, and such that $\exists k$ with $n = b^k$. Then

$$T(n) = a^kT(1) + \sum_{i=0}^{k-1} a^i f(\frac{n}{b^i}) \tag{1}$$

Proof.

by repeated self-substitution of the recurrence, we see that for all $\ell \geqslant 1$:

$$a^iT(\frac{n}{b^i}) = a^{i+1}T(\frac{n}{b^{i+1}}) + a^if(\frac{n}{b^i})$$

and therefore $T(n) = a^kT(1) + a^{k-1}f(\frac{n}{b^{k-1}}) + \cdots + af(\frac{n}{b}) + f(n)$ ■

Definition (Divide-and-conquer algorithms)

- the algorithm solves instances up to size m directly
- instances of size $n > m$ are split (divide) into a further instances of sizes $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$, solves these recursively; we then combine (conquer) their solutions

Definition

- let the time to split and combine be $f(n)$
- let the total time be $T(n)$, where we assume $T(n+1) \geqslant T(n)$
- We define

$$T^-(n) := \begin{cases} a\cdot T^-(\lfloor n/b \rfloor) + f(n) & \text{if } n > m \\ T(n) & \text{if } n \leqslant m \end{cases}$$

$$T^+(n) := \begin{cases} a\cdot T^+(\lceil n/b \rceil) + f(n) & \text{if } n > m \\ T(n) & \text{if } n \leqslant m \end{cases}$$

Example (Recall mergesort)

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
| (x <= y)  = x:(merge xs (y:ys))
| otherwise = y:(merge (x:xs) ys)

mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort (fsthalf xs)) (mergesort (sndhalf xs))
```

Question

Can we give a bound on the complexity of merge sort?

Definition (Recapitulation)

- the algorithm solves instances up to size m directly
- instances of size $n > m$ are split into a (divide) further instances of sizes $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$, solves these recursively, and then combines (conquer) their solutions

Observation

- Let $n = m \cdot b^k$
- algorithm splits k times, hence there are, for $r := \log_b a$:

$$a^k = (b^r)^k = (b^k)^r = \left(\frac{n}{m}\right)^r ,$$

  basic instances
- solving just the basic instances costs $\Theta(n^r)$
- r captures ratio of recursive calls a vs. decrease in size b:

Observation

- $a \cdot T(\lfloor n/b \rfloor) + f(n) \leqslant T(n) \leqslant a \cdot T(\lceil n/b \rceil) + f(n)$
- Taking splitting and combining into account, allows asymptotic analysis of $T^{\pm}(n)$

Theorem (master theorem)

Let $T(n)$ be an increasing function that satisfies the following recursive equations

$$T(n) = \begin{cases} c & n = 1 \\ aT(\frac{n}{b}) + f(n) & n = b^k,\ k = 1, 2, \dots \end{cases}$$

where $a \geqslant 1$, $b > 1$, $c > 0$. If $f \in \Theta(n^s)$ with $s \geqslant 0$, then

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^s \\ \Theta(n^s \log n) & \text{if } a = b^s \\ \Theta(n^s) & \text{if } a < b^s \end{cases}$$

Example (merge sort, continued)

for mergesort $a = b = 2$ and moreover $f \in \Theta(n^1)$, as splitting and combining is linear in n (hence $s = 1$). The master theorem yields the following bound on the runtime

$$T(n) \in \Theta(n \cdot \log n)$$

we have $a = b^s$, since $a = b = 2$ and $s = 1$ (second case)

Example

Consider the recurrence:

$$T(n) = 4T(\frac{n}{2}) + n^1$$

then $a = 4$, $b = 2$, $r = \log_b a = 2$ and $a > b^s$, hence by the first case of the theorem: $T(n) \in \Theta(n^2)$

Proof of the master theorem

Case $f \in \Theta(n^s)$ with $a = b^s$

- set $r := \log_b a$; then $r = s$
- we use properties of Θ, resp. properties of the exponential function to conclude:

$$a^i f(\frac{n}{b^i}) = \Theta(a^i \frac{n^r}{(b^i)^r}) = \Theta(a^i \frac{n^r}{(b^r)^i}) = \Theta(a^i \frac{n^r}{a^i}) = \Theta(n^r)$$

- from which we obtain (as $n = b^k$)

$$\sum_{i=0}^{k} a^i f(\frac{n}{b^i}) = \Theta(\sum_{i=0}^{k} n^r) = \Theta(kn^r) = \Theta(n^r \log n)$$

- moreover we already know that

$$a^k T(1) \in \Theta(n^r)$$

Proof (continued)

- recall equation (1)

$$T(n) = a^k T(1) + \sum_{i=0}^{k-1} a^i f(\frac{n}{b^i})$$

- its terms can be bounded as follows:

$$a^k T(1) \in \Theta(n^r)$$

$$\sum_{i=0}^{k-1} a^i f(\frac{n}{b^i}) \in \Theta(n^r \log n)$$

- and therefore

$$T(n) \in \Theta(n^r \log n)$$

∎

Example

$T(n) = 8 \cdot T(\frac{n}{2}) + n^2$

- $a = 8$, $b = 2$, $f(n) = n^2$,
- $\log_b a = 3$, $s = 2$, $8 > 2^2$ so by case 1 $T(n) \in \Theta(n^3)$

Example

$T(n) = 9 \cdot T(\frac{n}{3}) + n^3$

- $a = 9$, $b = 3$,$f(n) = n^3$,
- $\log_b a = 2$, $s = 3$, $9 < 3^3$ so by case 3 $T(n) \in \Theta(n^3)$

Example

$T(n) = T(\frac{n}{2}) + 1$ (binary search)

- $a = 1$, $b = 2$,$f(n) = 1$,
- $\log_b a = 0$, $s = 0$, $1 = 2^0$ so by case 2 $T(n) \in \Theta(\log n)$

Limitations of Master theorem

- split into non-equal-sized or non-fractional parts, e.g. Fibonacci (generating functions)
- $f(n)$ not of complexity $\Theta(n^s)$ for some s (can be relaxed)

Limitations of algorithms (recall from earlier lecture)

- There are more functions $f : \mathbb{N} \to \mathbb{N}$ than there are algorithms (programs, TMs); so some functions cannot be represented by algorithms;
- No algorithms for checking interesting properties of programs (TMs) themselves; termination (halting problem), reachability (unreachable code), ... No interesting property of programs can be programmed.
- No algorithm for checking whether a formula in first-order logic is universally valid (Entscheidungsproblem).
- No algorithm for checking whether Diophantine equations have a solution (Hilbert's 10th problem).
- ...

Remark

These limitations will be addressed in the last few weeks of course (i.e. now)

Function defined by a TM (recall from 3rd lecture)

Definition

a TM M

- accepts $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (t, y, n)$$

- rejects $x \in \Sigma^*$, if $\exists\, y, n$:

$$(s, \vdash x \sqcup^\infty, 0) \xrightarrow[M]{*} (r, y, n)$$

- halt on input x, if x is accepted or rejected
- does not halt on input x, if x is neither accepted nor rejected
- is total, if M halts on all inputs

Definition

A function $f : A \to B$ is defined by a TM M for every $x \in A$, M accepts input x with $f(y)$ on the tape (and does not halt or rejects on inputs $x \notin A$).

Computable functions

Idea of computability

$f : \mathbb{N} \to \mathbb{N}$ computable if there is an effective procedure to compute $f(n)$ for input n

Definition (computability via TM)

$f : \mathbb{N} \to \mathbb{N}$ computable if it can be defined by a TM

Examples of computable functions

remark

computability equivalently defined via models of computation: μ-recursive functions, λ-calculus, register machines, term rewriting, ...

Example

- any function programmable in some programming language
  square root, counting the number of 3s, compression, etc.
- effective $\neq$ efficient
  factorial, Ackermann function (complexity far worse than exponential)
- unbounded search functions
  the least number that has property P (need not exist)
- functions defined by finite cases
  $f(n) = n$ if n odd, otherwise n^2

Limits of computability

Lemma

there exist functions that are not computable (more functions than programs)

Proof.

- any program may be encoded by a finite bit-string
- $\Rightarrow$ there are countably many programs; (recall $\bigcup_i \{0,1\}^i$ is countable)
- there are uncountably many functions $\mathbb{N} \to \mathbb{N}$ (recall $\mathbb{N} \to \{0,1\}$ is uncountable)
- $\Rightarrow$ some function $\mathbb{N} \to \mathbb{N}$ is not computable ■

Theorem

concrete non-computable functions (diagonalise away from TM behaviours)

To do after Christmas: details of the above: coding, diagonalising way