Summary last week

- Multiplicative atomicity: prime $\iff$ indecomposable $\iff$ $|$-minimal (proof)
- Chinese remainder in 3 versions: bijection, Bézout, RSA (proofs)

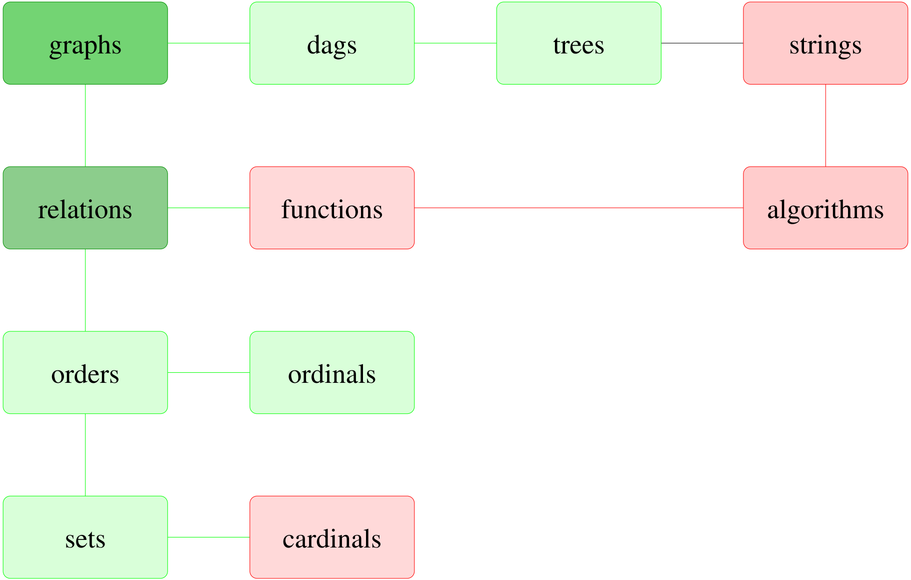
Summary last week

- Multiplicative atomicity: prime $\iff$ indecomposable $\iff$ $|$-minimal (proof)
- Chinese remainder in 3 versions: bijection, Bézout, RSA (proofs)
- comparing (complexity) functions asymptotically (using $\lim$, $\limsup$, $\liminf$)
- $O(f) = \{g \mid \exists m, c, \forall n \geq m, g(n) \leq c \cdot f(n)\}$; asymptotically bounded above by f
- $\Omega(f) = \{g \mid \exists m, c, \forall n \geq m, g(n) \geq c \cdot f(n)\}$; asymptotically bounded below by f
- $\Theta(f) = O(f) \cap \Omega(f)$; asymptotically same growth as f
- $o(f) = \{g \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\}$; asymptotically negligible w.r.t. f
- $\limsup$-characterisation of O: $f \in O(g) \Leftrightarrow \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
- $\liminf$-characterisation of Ω: $f \in \Omega(g) \Leftrightarrow \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$

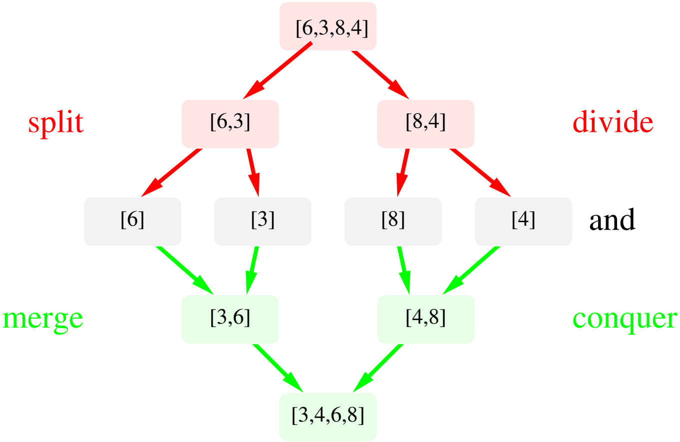
Course themes

- 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

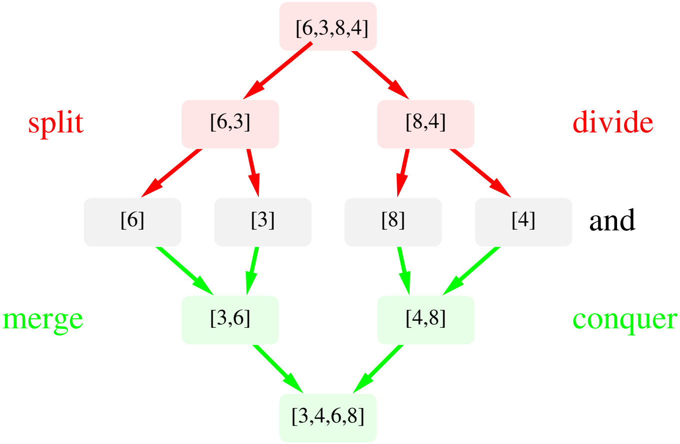
Discrete structures



Divide-and-conquer



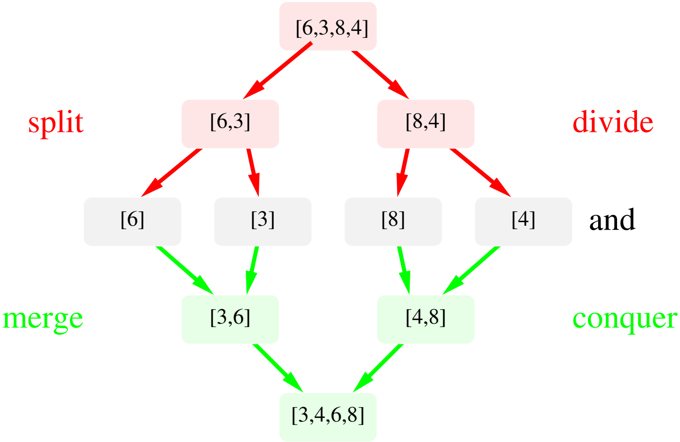
Divide-and-conquer



Complexity?

$O(n \cdot \log_2 n)$

Divide-and-conquer



Complexity?

$O(n \cdot \log n)$: each level $O(n)$ operations, $O(\log n)$ levels ($\log n$ splits, merges)

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

remarks

- problems of constant size as base cases

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equations of algorithm

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equations of algorithm
  - equations of shape $A(n) = \ldots a \cdot A(\frac{n}{b}) \ldots$; number a of smaller parts $\frac{n}{b}$

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \dots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \dots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \dots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equations of algorithm
  - equations of shape $A(n) = \dots a \cdot A(\frac{n}{b}) \dots$
  - input $I \Rightarrow$ length n of input

Divide-and-conquer

divide-and-conquer

recursive design paradigm for algorithms $\mathcal{A}$:

- divide: divide the input I into a number a of smaller parts $I_1, \ldots, I_a$
  solve the subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ recursively
- conquer: combine solutions of subproblems $\mathcal{A}(I_1), \ldots, \mathcal{A}(I_a)$ into solution for $\mathcal{A}(I)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equations of algorithm
  - equations of shape $A(n) = \ldots a \cdot A(\frac{n}{b}) \ldots$
  - input $I \Rightarrow$ length n of input
  - operation $\Rightarrow$ complexity of operation

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equation of mergesort

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equation of mergesort
  - equation $M(n) = 2 \cdot M(\frac{n}{2}) + n \cdot c$, if $n \geq 2$, otherwise c

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equation of mergesort
  - equation $M(n) = 2 \cdot M(\frac{n}{2}) + n \cdot c$, if $n \geq 2$, otherwise c
  - input $L \Rightarrow$ length n of list

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equation of mergesort
  - equation $M(n) = 2 \cdot M(\frac{n}{2}) + n \cdot c$, if $n \geq 2$, otherwise c
  - input $L \Rightarrow$ length n of list
  - complexity of merging $n \cdot c$; merging linear in sum n of sizes of sublists L_1,L_2

Divide-and-conquer for mergesort

divide-and-conquer for mergesort

recursive list L sorting algorithm $\mathcal{M}(L)$:

- divide: split input list L into 2 smaller sublists L_1, L_2
  solve sorting problems $\mathcal{M}(L_1), \mathcal{M}(L_2)$ recursively
- conquer: merge solutions of $\mathcal{M}(L_1), \mathcal{M}(L_2)$ into a single list solving $\mathcal{M}(L)$

remarks

- problems of constant size as base cases
- complexity analysis done using recurrence equation of mergesort
  - equation $M(n) = 2 \cdot M(\frac{n}{2}) + n \cdot c$, if $n \geq 2$, otherwise c
  - input $L \Rightarrow$ length n of list
  - complexity of merging $n \cdot c$
  - comparison,consing,... $\Rightarrow$ all some fixed complexity c

Algorithm and recurrence for complexity of mergesort

Mergesort in Haskell

```haskell
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))
```

Algorithm and recurrence for complexity of merge

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))
```

Complexity of merge in sum n of lengths of input lists

$E(n) = c + E(n-1)$ if neither input list is empty; c time of a comparison

$= c \cdot n$ otherwise

Algorithm and recurrence for complexity of merge

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))
```

Complexity of merge in sum n of lengths of input lists

$E(n) = c + E(n-1)$ if neither input list is empty

$= c \cdot n$ otherwise; $c \cdot n$ time for returning list

Algorithm and recurrence for complexity of merge

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))
```

Complexity of merge in sum n of lengths of input lists

$E(n) = c + E(n-1)$ if neither input list is empty

$\quad\;\; = c \cdot n$ otherwise

recurrence: specification of E contains itself

Algorithm and recurrence for complexity of merge

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))
```

Complexity of merge in sum n of lengths of input lists

$E(n) = c + E(n-1)$ if neither input list is empty

$\quad\;\; = c \cdot n$ otherwise

recurrence: closed-form solution $E(n) = c \cdot n$

Algorithm and recurrence for complexity of mergesort

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))
```

Complexity of mergesort in length n of input list

$$M(n) = 2 \cdot M(\tfrac{n}{2}) + E(n) \text{ if } n \geq 2$$
$$= c \text{ if } n \not\geq 2$$

$E(n)$ time for merging

Algorithm and recurrence for complexity of mergesort

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))
```

Complexity of mergesort in length n of input list

$M(n) = 2 \cdot M(\frac{n}{2}) + c \cdot n$ if $n \geq 2$

$\quad = c$ if $n \not\geq 2$ $\quad$ c time for base case

Algorithm and recurrence for complexity of mergesort

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))
```

Complexity of mergesort in length n of input list

$$
\begin{aligned}
M(n) &= 2 \cdot M(\tfrac{n}{2}) + c \cdot n && \text{if } n \geq 2 \\
&= c && \text{if } n \not\geq 2
\end{aligned}
$$

recurrence: specification of M contains itself

Algorithm and recurrence for complexity of mergesort

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))
```

Complexity of mergesort in length n of input list

$$
\begin{aligned}
M(n) &= 2 \cdot M(\tfrac{n}{2}) + c \cdot n && \text{if } n \geq 2 \\
&= c && \text{if } n \not\geq 2
\end{aligned}
$$

recurrence: closed-form solution $M(n) = c \cdot n \cdot \log n + c \cdot n$

Recurrences

Definition

- Recall: function is a set of (input,output) pairs; cannot be recursive

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification; cf. functional program

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

Example

the function $f\colon \mathbb{N} \to \mathbb{N}$, defined for $n \geqslant 1$ by:

$$g(n) = \begin{cases} 1 & n = 1 \\ 2 \cdot g(\frac{n}{2}) + n & n \geqslant 2 \end{cases}$$

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

Example

the Fibonacci numbers defined by

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geqslant 2 \end{cases}$$

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

Example

the Fibonacci numbers defined by

$$f_0 = 0 \qquad f_1 = 1 \qquad f_n = f_{n-1} + f_{n-2}$$

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

solving recurrence?

a closed-form solution: no recursive calls in right-hand side

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \rightarrow \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

solving recurrence?

a closed-form solution: no recursive calls in right-hand side

1. $g(n) = n \cdot \log n + n$; using master theorem (today)

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

solving recurrence?

a closed-form solution: no recursive calls in right-hand side

1. $g(n) = n \cdot \log n + n$
2. $f(n) = f_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$ where $\phi = \frac{1+\sqrt{5}}{2}$; using generating functions (not this course)

Recurrences

Definition

- Recall: function is a set of (input,output) pairs;
- recurrence is recursive equational specification;
- here: recurrences that specify functions $\mathbb{N} \to \mathbb{N}$;
- recurrences often written using indices: f_n instead of $f(n)$.

solving recurrence?

a closed-form solution: no recursive calls in right-hand side

1. $g(n) = n \cdot \log n + n$
2. $f(n) = f_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$ where $\phi = \frac{1+\sqrt{5}}{2}$

because recursive, solution unique so can be verified by substitution

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

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

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

1. $T(n) = 2^k \cdot T(\frac{n}{2^k}) + k \cdot c \cdot n$ for $1 \leq k < ?$

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

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 $k = \log n$

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

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 $T(n)$

Solving recurrences by self-substitution

self-substitution

repeatedly substitute recurrence into itself; look for pattern

Example

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 $k := \log n$. $T(n) = 2^{\log n} \cdot c + \log n \cdot c \cdot n = c \cdot n \cdot \log n + c \cdot n$
4. asymptotic complexity of solution: $T(n) \in O(n \cdot \log n)$

Verifying solutions/solving by guessing

Recall

- recurrence specifies unique function

Verifying solutions/solving by guessing

Recall

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

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

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)

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:
   - case $n = 1$: $f(1) = c$ ✓

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:
   - case $n = 1$: $f(1) = c$ ✓
   - case $n > 1$:

$$\begin{aligned} T(n) = f(n) &= c \cdot n \cdot \log n + c \cdot n \\ &= 2 \cdot (c \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c \cdot \frac{n}{2}) + c \cdot n \\ &=_{\text{IH}} 2 \cdot T(\frac{n}{2}) + c \cdot n \qquad \checkmark \end{aligned}$$

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:
   - case $n = 1$: $f(1) = c$ ✓
   - case $n > 1$:

$$\begin{aligned} T(n) = f(n) &= c \cdot n \cdot \log n + c \cdot n \\ &= 2 \cdot (c \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c \cdot \frac{n}{2}) + c \cdot n \\ &=_{\text{IH}} 2 \cdot T(\frac{n}{2}) + c \cdot n \qquad \checkmark \end{aligned}$$

using $\log(\frac{a}{b}) = (\log a) - (\log b)$

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:
   - 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 $T: \mathbb{N} \to \mathbb{N}$ *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*

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

Lemma

Let $T: \mathbb{N} \to \mathbb{N}$ *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*

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

Proof.

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

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

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

Lemma

Let $T: \mathbb{N} \to \mathbb{N}$ *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*

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

Proof.

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

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

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

Definition (Divide-and-conquer algorithms)

- the algorithm solves instances up to size m directly

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 (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))
```

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

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 $T^{\pm}(n)$

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

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

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 (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 $r = s$

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:

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

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 $n = b^k$)

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

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)

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

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:

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

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

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

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

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

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

- $a = 8$, $b = 2$, $f(n) = n^2$,
- $\log_b a = 3$, $s = 2$, $8 > 2^2$ 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

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

- $a = 8$, $b = 2$, $f(n) = n^2$,
- $\log_b a = 3$, $s = 2$, $8 > 2^2$ 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)