Introduction to Algorithm Analysis

Lecture #1 of Algorithms & Data Structures (Module 7)

Rom Langerak

E-mail: r.langerak@utwente.nl

Februari 4, 2019

Outline of the ADS part of module 7

Algorithms & Data Structures:

Introduction to algorithm analysis
1 lecture, 0.5 exercise course

Sorting and heaps
1 lecture, 0.5 exercise course

Trees
1 lecture, 1 exercise course

Dynamic programming
1 lecture, 1 exercise course

all in the coming two weeks.

Exercises

Bonus foints for ADS part:

- Each Tuesday exercise is placed on Blackboard
- should be handed in at next day (Friday)
- each exercise good for 0.125 bonus point added to exam score
- no feedback is given: you have to check your work against the model answer on Blackboard

The word “Algorithm”

Origin

Abu Ja’far Muhammad ibn Musa **Al-Khwarizmi**
(lived in Baghdad from about 780 to about 850)

Meaning

An **algorithm** is a **scheme** of **steps** to go
from **given data** (input)
to a **desired result** (output).

A step can be an arithmetic operation, or a yes-no decision, or a choice between alternatives ... (example: looking up a word in a dictionary)

Quality of algorithms

The quality of an algorithm is judged in a number of ways.

1. The steps always lead to a **correct result**.

2. The scheme wastes no steps, it is an **efficient way** to reach the result.

3. The scheme wastes no resources, e.g. it uses a reasonable amount of memory.

The word “Complexity”

By the complexity of an algorithm we **do not** mean that it is difficult to understand how the scheme of steps works

If we speak of the **complexity of an algorithm** we discuss the relation between
the size of the input for the algorithm and
the effort needed to reach a result
(maybe a lot of steps, maybe a lot of memory ...)

The complexity of algorithms

Effort needed to reach a result has always two aspects

- the number of steps taken — time complexity
- the amount of space used — space complexity

time complexity $\neq$ space complexity

The complexity of algorithms

Assess complexity independent of type of computer used, programming language, programming skills, etc.

- Technology improves things by a constant factor only
- Even a supercomputer cannot rescue a “bad” algorithm
  - a faster algorithm on a slower computer will *always* win for sufficiently large inputs

The time complexity of algorithms

Analysis is based on choice of basic operations such as:

- "comparing two numbers" for *sorting* an array of numbers
- "multiplying two real numbers" for *matrix multiplication*

This approach works, but be careful

- # basic operations should be good estimate of total # steps
- # basic operations constitutes basis for determining the rate of growth of the number of steps as the input gets larger

Time complexity in practice – Actual run times

Complexity	$33n$	$46n \log n$	$13n^2$	$3.4n^3$	2^n
n	*Solution time*				
10	.00033 sec	.0015 sec	.0013 sec	.0034 sec	.001 sec
10^2	.0033 sec	.03 sec	.13 sec	3.4 sec	$4 \cdot 10^{16}$ yr
10^3	.033 sec	.45 sec	13 sec	.94 hour	
10^4	.33 sec	6.1 sec	1300 sec	39 days	
10^5	3.3 sec	1.3 min	1.5 days	108 yr	

Note: impact of high constant factors diminishes if n grows

In practice — Maximum solvable input size

Complexity	$33n$	$46n \log n$	$13n^2$	$3.4n^3$	2^n
time allowed	*Maximum solvable input size*				
1 sec	30,000	2,000	280	67	20
1 min	1,800,000	82,000	2,170	260	26
1 hour	108,000,000	1,180,800	16,818	1,009	32

We cannot handle input 60 times larger if we increase time (or speed) by factor 60

In practice – The effect of faster computers

Let N be the maximum input size that can be handled in a fixed time

What happens to N if we take a computer that is K times faster?

# steps performed on input of size n	maximum feasible input size N_{fast}
$\log n$	N^K
n	$K \cdot N$
n^2	$\sqrt{K} \cdot N$
2^n	$N + \log K$

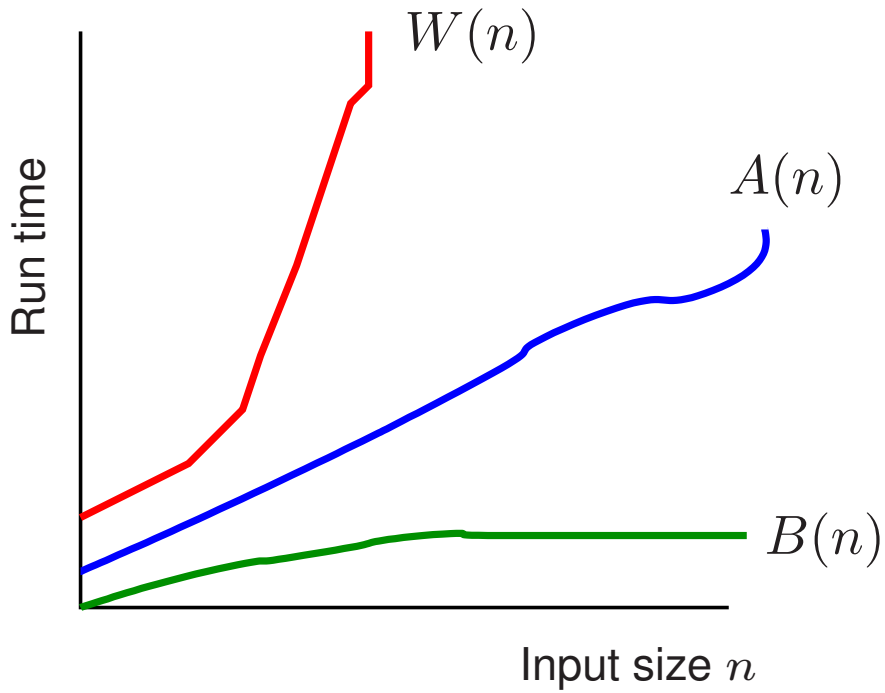
Average, best and worst case (time) complexity – Intuition

Consider a given algorithm A

- The worst case complexity of A is the *maximum* # basic operations performed by A on any input of a certain size
- The best case complexity of A is the *minimum* # basic operations performed by A on any input of a certain size (often not so interesting - why?)
- The average case complexity of A is the *average* # basic operations performed by A on any input of a certain size

Each of these complexities defines a function: number of operations (steps) versus input size

Average, best and worst case complexity – in a picture



Average, best and worst case (time) complexity

D_n	=	set of inputs of size n
$t(I)$	=	# basic operations needed for input I known by analysis the algorithm
$\Pr(I)$	=	the probability that input I occurs known by experience, or assumption, e.g., "all inputs occur equally frequent")

Now formally:

- The worst case complexity: $W(n) = \mathit{max}\{\, t(I) \mid I \in D_n \,\}$
- The best case complexity: $B(n) = \mathit{min}\{\, t(I) \mid I \in D_n \,\}$
- The average case complexity: $A(n) = \sum_{I \in D_n} \Pr(I) \cdot t(I)$

Linear search

Input: array E with n entries and item K to be looked up

Output: does K occur in the array E?

```
def linsearch(E, n, K):
    i = 0
    found = False
    while i<n and not found:
        found = (E[i]==K)
        i = i+1
    return found
```

Analyzing linear search

Basic operation = comparison of integer K with array element $E[\ldots]$

- $D_n =$ all permutations of n elements out of a set of $N > n$ elements
- $W(n) = n$, as in worst case K is last element in array, or is not found
- $B(n) = 1$, as in best case K is the first element in the array

?? $A(n) \approx \frac{1}{2}n$, as on average half of the array needs to be checked? **No**

Average case complexity for linear search – 1

There are two scenarios:

- K occurs in array E; this yields average complexity $A_{succ}(n)$
- K does not occur in array E; this yields average complexity $A_{fail}(n)$

$$A(n) = \Pr\{K \text{ in } E\} \cdot A_{succ}(n) + \Pr\{K \text{ not in } E\} \cdot A_{fail}(n)$$

$$\Pr\{K \text{ not in } E\} = 1 - \Pr\{K \text{ in } E\}$$

$$A_{fail}(n) = n$$

Average case complexity for linear search – 2

How about $A_{succ}(n)$?

- assume all elements in the array E are distinct
- note that if $E[i] == K$ the search takes i comparisons

Then $$A_{succ}(n) = \sum_{i=1}^{n} \Pr\{E[i] == K \mid K \text{ in } E\} \cdot i$$

$\equiv$ (* assume K can equally well be at any index if it occurs in E *)

$$A_{succ}(n) = \sum_{i=1}^{n} \frac{1}{n} \cdot (i) = \frac{1}{n} \cdot \sum_{i=1}^{n} i$$

$\equiv$ (* standard series *)

$$A_{succ}(n) = \frac{1}{n} \cdot \frac{(n+1)}{2} \cdot n = \frac{n+1}{2}$$

Average case complexity for linear search – 3

Putting the results together yields:

$$A(n) = \Pr\{K \text{ in } E\} \cdot \frac{n+1}{2} + (1 - \Pr\{K \text{ in } E\}) \cdot n$$

Note that if $\Pr\{K \text{ in } E\}$ equals

- 1, then $A(n) = \frac{n+1}{2} = A_{succ}(n)$ $\approx$ 50% of E checked
- 0, then $A(n) = n = W(n)$ E is entirely checked
- $\frac{1}{2}$, then $A(n) = \frac{3 \cdot n}{4} + \frac{1}{4}$ $\approx$ 75% of E checked

Asymptotic analysis

Exactly determining $A(n), B(n)$ and $W(n)$ is very hard, and not so useful

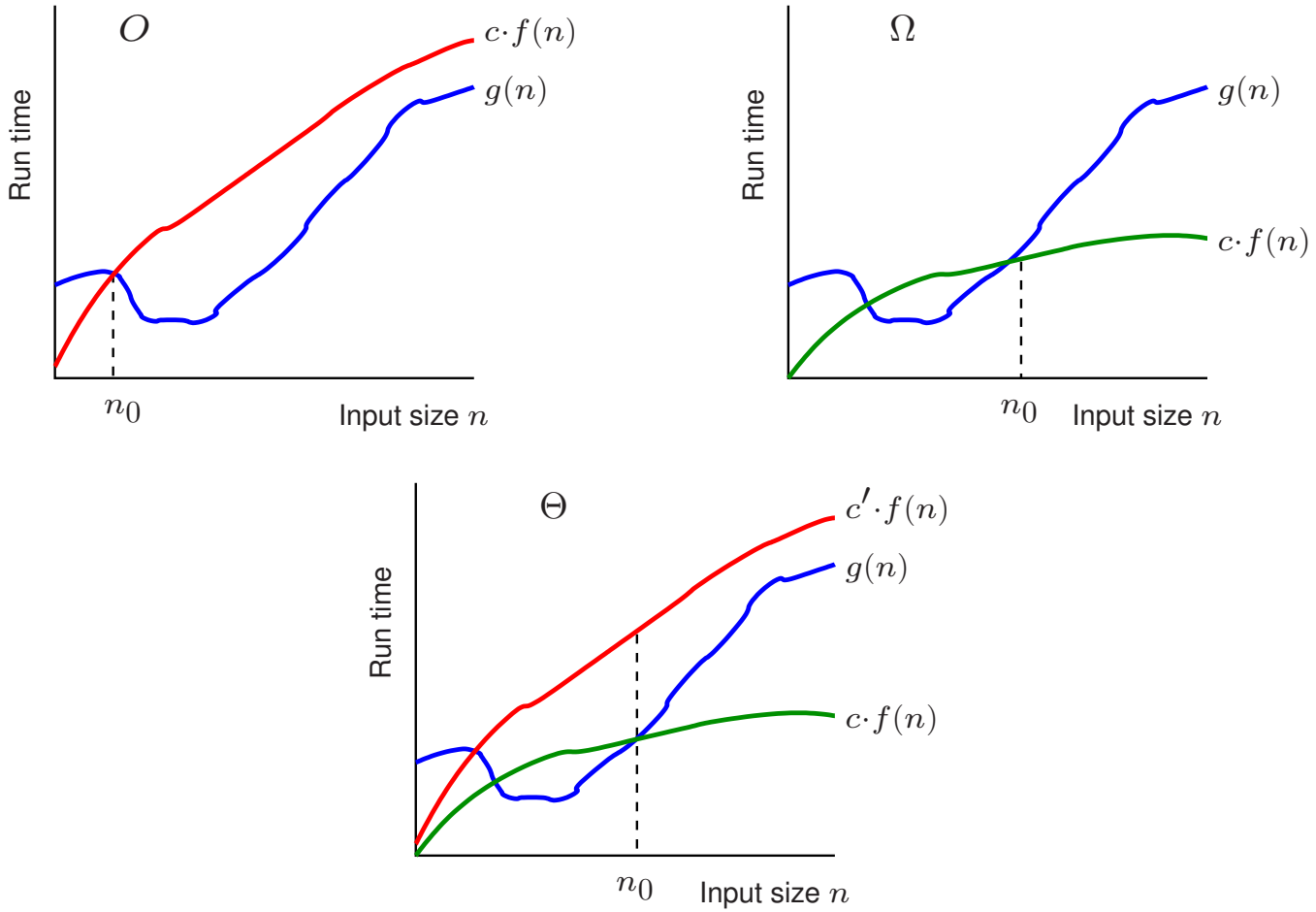
Typically no exact analysis, but asymptotic analysis

- look at growth of execution time for $n \to \infty$
- thus ignoring small inputs and constant factors
- intuition: drop lower order terms, e.g.,

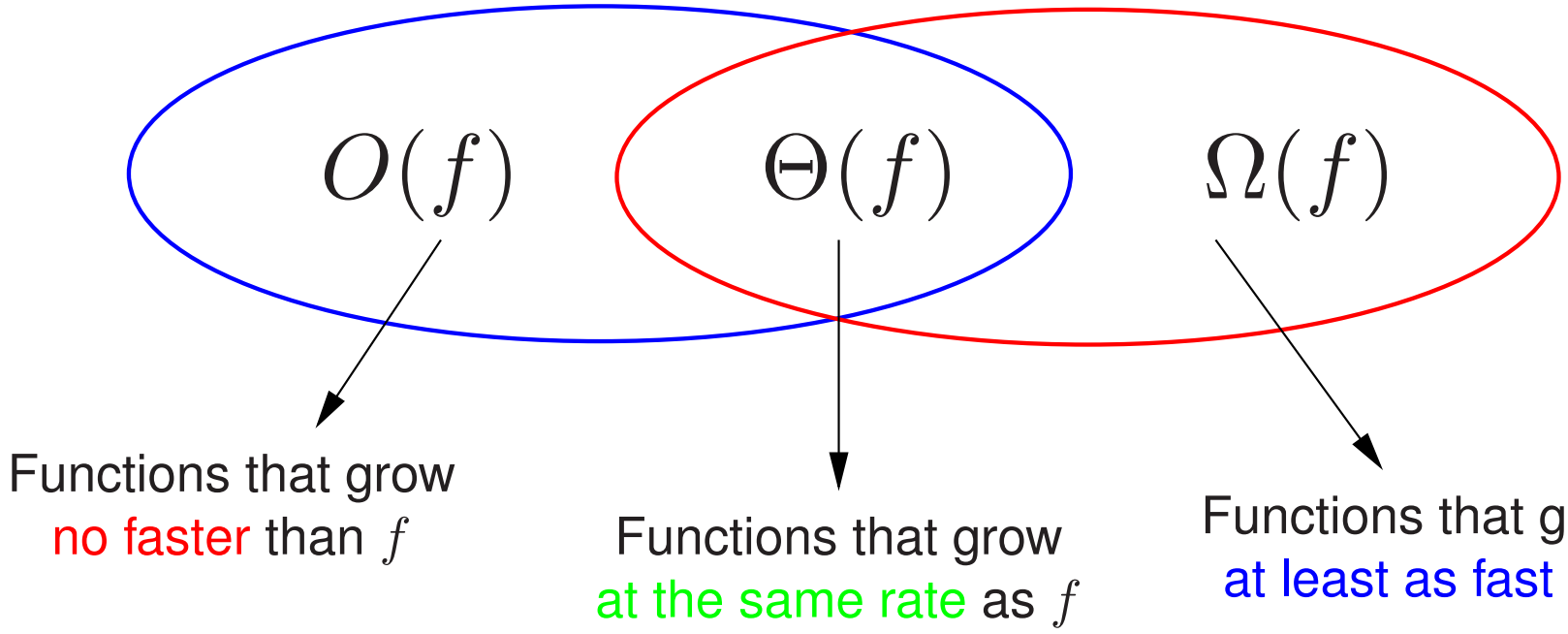
$$W(n) = 5n^4 + 3n^3 + 10 \text{ is like } n^4$$

- thus, we obtain lower/upper bounds on $A(n)$, $B(n)$ and $W(n)$ now!
- mathematical ingredient: asymptotic order of functions (classes O, Ω and Θ)

Classes O, Ω and Θ – 1



Classes O, Ω and Θ – 2



Classes O, Ω and Θ – 3

Let f and g be functions from $\mathbb{N}$ (input size) to $\mathbb{R}^{\geqslant 0}$ (step count), let $c > 0$, and $n_0 > 0$

- $O(f)$ is the set of functions that grow no faster than f
  - $g \in O(f)$ means $c \cdot f(n)$ is an upper bound on $g(n)$, $n > n_0$
- $\Omega(f)$ is the set of functions that grow at least as fast as f
  - $g \in \Omega(f)$ means $c \cdot f(n)$ is a lower bound on $g(n)$, $n > n_0$
- $\Theta(f)$ is the set of functions that grow at the same rate as f
  - $g \in \Theta(f)$ means $c' \cdot f(n)$ is an upper bound on $g(n)$, $n > n_0$ **and** $c \cdot f(n)$ is a lower bound on $g(n)$, $n > n_0$

The class big-oh

Formally, $g \in O(f)$ iff $\exists c > 0, n_0$ such that $\forall n \geqslant n_0 : 0 \leqslant g(n) \leqslant c \cdot f(n)$

Handy alternative:

$g \in O(f)$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \neq \infty$

Example: consider $g(n) = 3n^2 + 10n + 6$. We have:

- $g \notin O(n)$ since $\lim_{n\to\infty} g(n)/n = \infty$
- $g \in O(n^2)$ since $g(n) \leqslant 20n^2$ for $n \geqslant 1$
- $g \in O(n^3)$ since $g(n) \leqslant \frac{1}{10}n^3$ for n sufficiently large

Tightest upper bounds are of most use! $g \in O(n^2)$ says more than $g \in O(n^3)$

The class big-omega

Formally, $g \in \Omega(f)$ iff $\exists c > 0, n_0$ such that $\forall n \geqslant n_0 : c \cdot f(n) \leqslant g(n)$

Handy alternative:

$g \in \Omega(f)$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \neq 0$

Example: consider $g(n) = 3n^2 + 10n + 6$. We have:

- $g \in \Omega(n)$ since $\lim_{n\to\infty} g(n)/n = \infty > 0$
- $g \in \Omega(n^2)$ since $\lim_{n\to\infty} g(n)/n^2 = 3 > 0$
- $g \notin \Omega(n^3)$ since $\lim_{n\to\infty} g(n)/n^3 = 0$

Tightest lower bounds are of most use! $g \in \Omega(n^2)$ says more than $g \in \Omega(n)$

The class big-theta

$g \in \Theta(f)$ iff $\exists c, c' > 0, n_0$ such that $\forall n \geqslant n_0 : c \cdot f(n) \leqslant g(n) \leqslant c' \cdot f(n)$

Handy alternative: $g \in \Theta(f)$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some $0 < c < \infty$

- recall $g \in \Theta(f)$ if and only if $g \in O(f)$ and $g \in \Omega(f)$

Example: consider $g(n) = 3n^2 + 10n + 6$. We have:

- $g \notin \Theta(n)$ since $\lim_{n\to\infty} g(n)/n = \infty > 0$
- $g \in \Theta(n^2)$ since $\lim_{n\to\infty} g(n)/n^2 = 3$
- $g \notin \Theta(n^3)$ since $\lim_{n\to\infty} g(n)/n^3 = 0$

Help in finding limits

Note that

if f, g are differentiable and

both $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$

then $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{g'(n)}{f'(n)}$

if the last limit exists

This is a consequence of l'Hôpital's rule

Some elementary properties

- Reflexivity:
  - $f \in O(f)$
  - $f \in \Omega(f)$
  - $f \in \Theta(f)$
- Transitivity:
  - $f \in O(g)$ and $g \in O(h)$ imply $f \in O(h)$
  - $f \in \Omega(g)$ and $g \in \Omega(h)$ imply $f \in \Omega(h)$
  - $f \in \Theta(g)$ and $g \in \Theta(h)$ imply $f \in \Theta(h)$
- Symmetry:
  - $f \in \Theta(g)$ if and only if $g \in \Theta(f)$
- Relation between O and Ω:
  - $f \in O(g)$ if and only if $g \in \Omega(f)$

Fibonacci numbers

Consider the growth of a rabbit population, e.g.:

- suppose we have two rabbits, one of each sex
- rabbits have bunnies once a month after they are 2 months old
- they always give birth to twins, one of each sex
- they never die and never stop propagating

The # (pairs of) rabbits after n months is computed by:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n+2) &= Fib(n+1) + Fib(n) \text{ for } n \geqslant 0 \end{aligned}$$

We thus obtain the sequence:

n	0	1	2	3	4	5	6	7	8	9	...
$Fib(n)$	0	1	1	2	3	5	8	13	21	34	...

Fibonacci – A naive algorithm

```
def fibRec(n):
    if (n==0) or (n==1):
        return n
    else:
        return fibRec(n-2)+fibRec(n-1)
```

The # arithmetic steps $T_{fibRec}(n)$ needed to compute *fibRec*(*n*) is:

$$\begin{aligned} T_{fibRec}(0) &= 0 \\ T_{fibRec}(1) &= 0 \\ T_{fibRec}(n+2) &= T_{fibRec}(n+1) + T_{fibRec}(n) + 3 \text{ for } n \geqslant 0 \end{aligned}$$

This is a recurrence equation!

To decide the complexity class of *fibRec* we solve this equation

Fibonacci – Counting steps for the naive algorithm

$$\begin{array}{rcl} T_{\mathit{fibRec}}(0) & = & 0 \\ T_{\mathit{fibRec}}(1) & = & 0 \\ T_{\mathit{fibRec}}(n{+}2) & = & T_{\mathit{fibRec}}(n{+}1) + T_{\mathit{fibRec}}(n) + 3 \text{ for } n \geqslant 0 \end{array}$$

Using induction we can prove (check!) that $T_{\mathit{fibRec}}(n) = 3{\cdot}\mathit{Fib}(n{+}1) - 3$

How large is $\mathit{Fib}(n)$? Can we give upper/lower bounds?

It follows (by induction) that $2^{(n-2)/2} \leqslant \mathit{Fib}(n) \leqslant 2^{n-2}$ for $n \geqslant 2$

But, this means that the complexity of *fibRec* is exponential:

$$T_{\mathit{fibRec}}(n) \in \Omega((\sqrt{2})^n)$$

Fibonacci revisited – An iterative algorithm

```
def fibIter(n):
    if (n==0) or (n==1):
        return n
    else:
        a = 0
        b = 1
        for i in range(2, n+1):
            a, b = b, a+b
        return b
```

The # arithmetic steps $T_{\mathit{fibIter}}(n+2) = 2(n+1)$ for $n \geqslant 0$ and 0 otherwise

So, the complexity of *fibIter* is linear:

$$T_{\mathit{fibIter}}(n) \in \Theta(n)$$

General problem – Solving recurrence equations

For simple cases closed-form solutions exist, e.g.:

- $T(n) = c{\cdot}T(n-1)$ with $T(0) = k$ has unique solution $T(n) = k{\cdot}c^n$
- $T(n) = T(n-1) + f(n)$ has unique solution $T(n) = T(0) + \sum_{i=1}^{n} f(i)$

Sometimes solutions can be guessed (and then proven). For the general case no general solution does exist.

Typical case:

$$T(n) = a{\cdot}T\left(\tfrac{n}{b}\right) + f(n) \text{ with } a \geqslant 1 \text{ and } b > 1$$

- problem is divided into a similar problems of size $\frac{n}{b}$ each
- non-recursive cost $f(n)$ to split problem or combine solutions of sub-problems

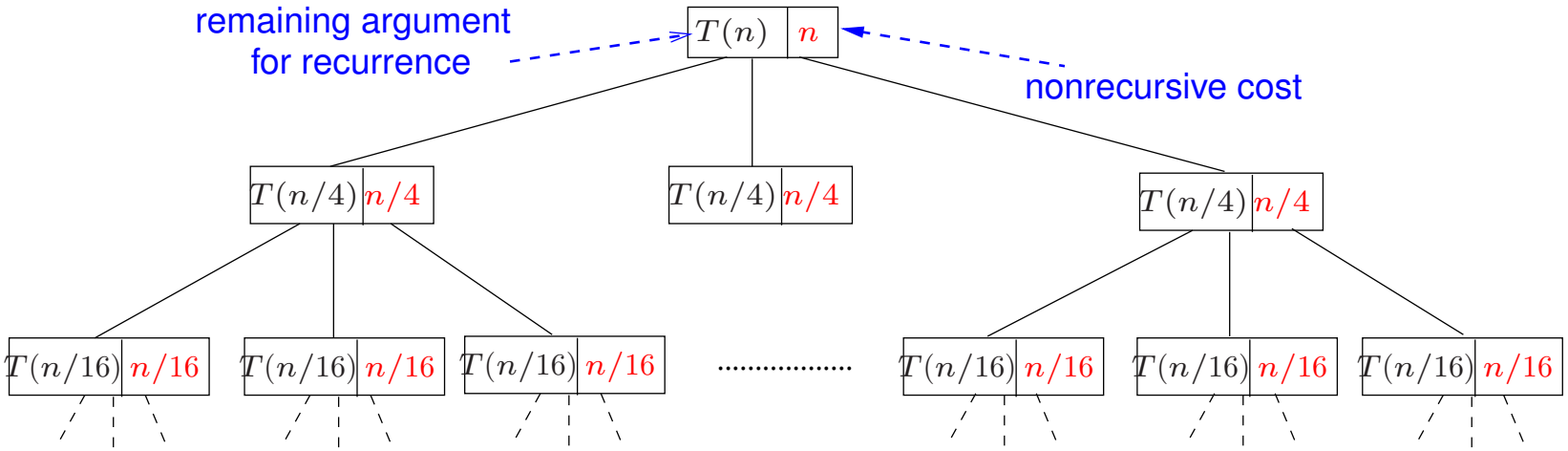
The recursion tree method – 1

Visualize the back-substitution process as a tree, keeping track of

- the size of the remaining arguments in the recurrence
- the nonrecursive costs

$\Rightarrow$ very useful to obtain a good guess for substitution method

The recurrence tree of $T(n) = 3 \cdot T(n/4) + n$ looks like:



Intermezzo: Recursion tree properties

$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $a \geqslant 1$ and $b > 1$

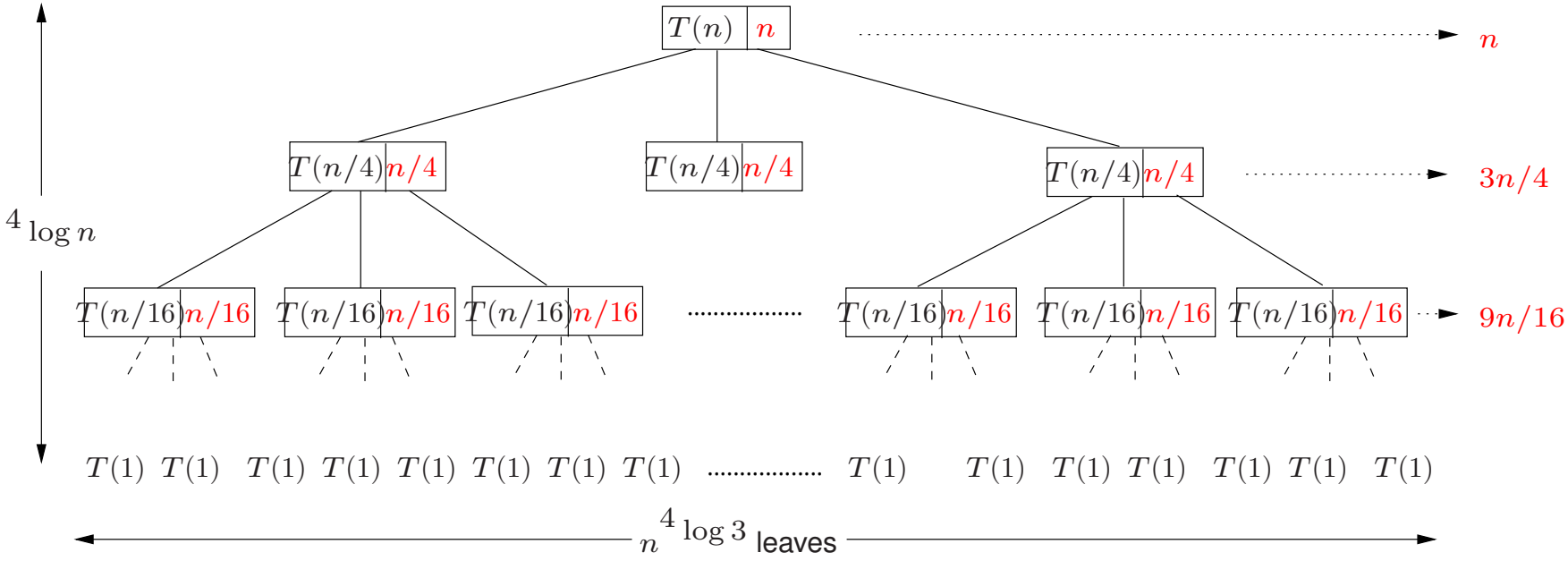
The level of the recursion tree for $T(n)$ is the least k such that $b^k \geqslant n$.
So the level of this recursion tree is $\lceil {}^b\log n \rceil = \lceil \log n / \log b \rceil$.

The number of nodes at level m in the recursion tree for $T(n)$ is a^m.

The number of leaves in the recursion tree for $T(n)$ is
$a^{(\log n / \log b)} = n^{(\log a / \log b)}$.

(Calculus: ${}^x\log y = {}^z\log y / {}^z\log x$ and $x^{{}^z\log y} = y^{{}^z\log x}$)

The recursion tree method – 2



$$T(n) = \underbrace{\sum_{i=0}^{4\log n-1}}_{\text{sum levels}} \underbrace{\left(\frac{3}{4}\right)^{i} \cdot n}_{\text{level cost}} + \underbrace{c \cdot n^{4\log 3}}_{\text{total cost leafs}}$$

The recursion tree method – 3

An upper bound on the complexity can now be obtained by:

$$T(n) = \sum_{i=0}^{4^{\log n - 1}} \left(\frac{3}{4}\right)^i \cdot n + c \cdot n^{4^{\log 3}}$$

$$\Rightarrow \quad \text{(* ignore the smaller terms *)}$$

$$T(n) < \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \cdot n + c \cdot n^{4^{\log 3}}$$

$$\equiv \quad \text{(* geometric series *)}$$

$$T(n) < \frac{1}{1-(3/4)} \cdot n + c \cdot n^{4^{\log 3}}$$

$$\equiv \quad \text{(* calculus *)}$$

$$T(n) < 4 \cdot n + c \cdot n^{4^{\log 3}}$$

$$\equiv \quad \text{(* obtain the order *)}$$

$$T(n) \in O(n)$$

The algorithm thus has a linear worst-case complexity

The Master theorem

Consider: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $a \geqslant 1$ and $b > 1$

- problem is divided into a similar problems of size $\frac{n}{b}$ each
- non-recursive cost $f(n)$ to split problem or combine solutions of sub-problems
- we saw that # leafs in the recursion tree is n^E with $E = \log a / \log b$

The Master theorem says:

	if	then
1.	$f(n) \in O(n^{E-\varepsilon})$ for some $\varepsilon > 0$	$T(n) \in \Theta(n^E)$
2.	$f(n) \in \Theta(n^E)$	$T(n) \in \Theta(f(n) \cdot \log n)$
3.	$f(n) \in \Omega(n^{E+\varepsilon})$ for some $\varepsilon > 0$ and $af(\frac{n}{b}) \leqslant cf(n)$ for some $c < 1$	$T(n) \in \Theta(f(n))$

If none of the cases apply, the Master theorem gives no clue!

Applying the Master theorem

$T(n) = 4 \cdot T(n/2) + n$

- so $a = 4$, $b = 2$ and $f(n) = n$; $E = \log 4 / \log 2 = 2$
- since $f(n) = n \in O(n^{2-\varepsilon})$, case 1 applies: $T(n) \in \Theta(n^2)$

$T(n) = 4 \cdot T(n/2) + n^2$

- so $a = 4$, $b = 2$ and $f(n) = n^2$; $E = \log 4 / \log 2 = 2$
- since $n^2 \notin O(n^{2-\varepsilon})$, case 1 does not apply
- but since $f(n) = n^2 \in \Theta(n^2)$, case 2 applies: $T(n) \in \Theta(n^2 \cdot \log n)$

$T(n) = 4 \cdot T(n/2) + n^3$

- so $a = 4$, $b = 2$ and $f(n) = n^3$; $E = \log 4 / \log 2 = 2$
- clearly, cases 1 and 2 do not apply since $E = 2$
- since $f(n) = n^3 \in \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$, case 3 *might* apply
- . . . and as $4(n/2)^3 = 1/2n^3 \leqslant cf(n)$ for $c = 1/2$, case 3 applies: $T(n) \in \Theta(n^3)$

The Master theorem does not always apply

$T(n) = 4 \cdot T(n/2) + \frac{n^2}{\log n}$

- we have $a = 4$, $b = 2$ and $f(n) = n^2/(\log n)$; $E = 2$
- $n^2/(\log n) \notin O(n^{2-\varepsilon})$ since $f(n)/n^2 = (\log n)^{-1} \notin O(n^{-\varepsilon})$

$\Rightarrow$ case 1 of the Master theorem does not apply

- $n^2/(\log n) \notin \Theta(n^2)$

$\Rightarrow$ case 2 of the Master theorem does not apply

- $f(n) \notin \Omega(n^{2+\varepsilon})$ since $f(n)/n^2 = (\log n)^{-1} \notin \Omega(n^{+\varepsilon})$

$\Rightarrow$ case 3 of the Master theorem does not apply

[$\Rightarrow$] The Master theorem does not apply to this case at all!

By substitution one obtains:

$$T(n) \in \Theta(n^2 \cdot \log(\log n))$$

Understanding the Master theorem – 1

Consider: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $a \geqslant 1$ and $b > 1$

Suppose $f(n)$ is small enough, i.e., $f(n) \ll n^E$

- to get a feeling let $f(n) = 0$, then
- only the costs at the leafs will count!
- there will be ${}^b\log n$ levels in the recursion tree
- the # nodes at level k equals a^k

$\Rightarrow\ T(n) = a^{{}^b\log n} = n^E$

This yields case 1 of the Master theorem

$f(n) < n^E$ is insufficient; $f(n)$ must be polynomially smaller than n^E

Understanding the Master theorem – 2

Consider: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $a \geqslant 1$ and $b > 1$

Suppose $f(n)$ is large enough, i.e., $f(n) \gg n^E$

- if $f(n)$ is large enough, it will exceed $a \cdot f(n/b)$
- for instance $f(n) = n^3 > (n/3)^3 + (n/3)^3 + (n/3)^3$ unless $a \geqslant 27$

This yields case 3 of the Master theorem

Understanding the Master theorem – 3

If $f(n)$ and n^E grow equally fast, the depth of the tree is important

- this perfect balance is what one wants to achieve with divide-and-conquer algorithms!

This yields case 2 of the Master theorem