APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Lectures 1-2

September 25-27th, 2019

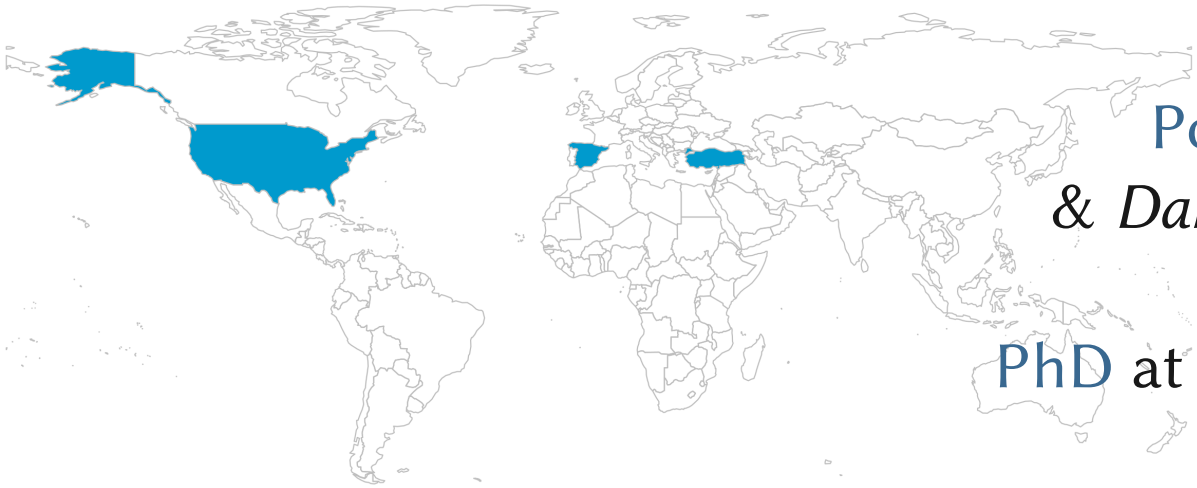
APA: Advanced Programming, Algorithms and Data Structures

- Instructor
  - Emre Guney (emre.guney@upf.edu)
- Course work
  - Practical class & assignment performance (20%)
  - Written exam (30%)
  - Project (50%)
- Relevant material
  - *Introduction to Algorithms* by T.H. Cormen, C.E.Leiserson, R.L. Rivest, and C. Stein, McGraw-Hill, 2002.
  - *Algorithm Design* by J. Kleinberg and É. Tardos, Addison-Wesley, 2006.
  - *An Introduction to Bioinformatics Algorithms* by N.C. Jones and P.A. Pevzner, The MIT Press Cambridge, 2004. (available online)
  - *Algorithms in Bioinformatics* by W. SUNG, CRC Press, 2009.

Who am I?

Researcher at *GRIB - IMIM / UPF*
& RD team lead at *(a biotech) start-up*

BdP fellow at *Inst. for Biomedical Research*
& Consultant at *DZZOM* 2017



Postdoc at *Northeastern Uni.*
& *Dana-Farber Cancer Inst.* 2015

PhD at *Uni. of Pompeu Fabra* 2012

MS at *Koc Uni.* 2007

BS at *Middle East Tech. Uni.* 2005

emreguney.net

Who are you?

- Previous degrees and relevant course work
- Favorite programming language(s) and confidence in coding
- Previous experience with code development in a team (git, GitHub, BitBucket, etc.)
- The motivation behind taking this class
- Anything else you would like to comment

What do you think?

- Good practices in teaching based on previous classes you took

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

Every strategy (algorithm) has its pros / cons

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

Every strategy (algorithm) has its pros / cons

- Computation time (on average, best and worst case depending on the input)

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

Every strategy (algorithm) has its pros / cons

- Computation time (on average, best and worst case depending on the input)
- Physical resources (i.e. memory, bandwith usage)

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

Every strategy (algorithm) has its pros / cons

- Computation time (on average, best and worst case depending on the input)
- Physical resources (i.e. memory, bandwith usage)
- Practical use / feasibility (i.e., ease of implementation, handling outliers, missing data)

What do you think?

- Good practices in teaching based on previous classes you took
- How to find a consensus?
  - Ranking
  - Voting
  - ...
- What if we have preference weights for each?

Every strategy (algorithm) has its pros / cons

- Computation time (on average, best and worst case depending on the input)
- Physical resources (i.e. memory, bandwith usage)
- Practical use / feasibility (i.e., ease of implementation, handling outliers, missing data)

All of the above also depends on the choice of
the data structure and the programming language

Algorithms, programming and data structures – Why do we care?

- We want to do sth
- We want to do **more** with limited (**less**) resources
- Time is valuable

It is all about (big) data in the end.

Algorithms, programming and data structures – Why do we care?

- We want to do sth
- We want to do **more** with limited (**less**) resources
- Time is valuable

It is all about (big) data in the end.

Functional complexity:	n	n^2	n^3
$n = 10$	<1 sec	<1 sec	<1 sec
$n = 100$	<1 sec	< 1 sec	1 sec
$n = 1,000$	<1 sec	1 sec	18 mins
$n = 10,000$	<1 sec	2 mins	12 days
$n = 100,000$	<1 sec	3 hours	32 years
$n = 1,000,000$	1 sec	12 days	31,710 years

(Processing times of inputs of increasing sizes for a processor performing a million high-level instructions per second.)

Adapted from: Algorithm Design, J. Kleinberg and É Tardos, Addison Wesley, 2006

Course objective

Gaining better understanding of algorithm design considerations as well as programming concepts and data structures underlying these considerations

Course objective

Gaining better understanding of algorithm design considerations as well as programming concepts and data structures underlying these considerations

We first study the problem of sorting permutations by transpositions only. We develop a novel 1.5-approximation algorithm, that is considerably simpler than the extant ones. Moreover, the analysis of the algorithm is significantly less involved, and provides a good starting point for studying related open problems. Next, we consider the problem of sorting permutations by transpositions, transreversals and revrevs (an operation reversing each of two consecutive segments). Our main result is a quadratic 1.5-approximation algorithm for sorting by these operations, improving over the extant best known approximation ratio of 1.75. We present an implementation of both algorithms that runs in time $O(n^{3/2}\sqrt{\log(n)})$, improving over the quadratic running time of previous algorithms. The improvement is achieved by exploiting an efficient data structure.

Source: Combinatorial Algorithms for Genome Rearrangements and DNA Oligonucleotide Arrays, T. Hartman, PhD thesis submitted to Feinberg Graduate School, Weizmann Institute of Science, 2004

Comparing computational complexity

$$2^n$$
$$e^n$$
$$n!$$
$$n^3$$
$$n\log(n)$$
$$n^2$$
$$n\sqrt{n}$$
$$n$$
$$\sqrt{n}$$
$$\ln(n)$$
$$\log(n)$$
$$1$$

Comparing computational complexity

$$2^n$$
$$e^n$$
$$n!$$
$$n^3$$
$$n\log(n)$$
$$n^2$$
$$n\sqrt{n}$$
$$n$$
$$\sqrt{n}$$
$$\ln(n)$$
$$\log(n)$$
$$1$$

Find the 3 functions violating the ranking (for large n)

Comparing computational complexity

2^n	*
e^n	
$n!$	*
n^3	
$n\log(n)$	*
n^2	
$n\sqrt{n}$	
n	
$\sqrt{n}$	
$\ln(n)$	
$\log(n)$	
1	

Find the 3 functions violating the ranking (for large n)

Comparing computational complexity

$$n!$$
$$e^n$$
$$2^n$$
$$n^3$$
$$n^2$$
$$n\sqrt{n}$$
$$n\log(n)$$
$$n$$
$$\sqrt{n}$$
$$\ln(n)$$
$$\log(n)$$
$$1$$

Comparing computational complexity

$$n!$$
$$e^n$$
$$2^n$$
$$n^3$$
$$n^2$$
$$n\sqrt{n}$$
$$n\log(n)$$
$$n$$
$$\sqrt{n}$$
$$\ln(n)$$
$$\log(n)$$
$$1$$

Given $\ln(n) > \log(n)$, what is the base of the log?

Names of common function families

$n!$	factorial
k^n	exponential
n^k	polynomial
n^3	cubic
n^2	quadratic
$n \log(n)$	quasi-linear
n	linear
$\sqrt{n}$	root-n
$\log(n)$	logarithmic
1	constant

Note that $k \in \mathbb{Z}$ above.

Algorithms, programming and data structures in Bioinformatics

Subject	4 Exhaustive Search	5 Greedy Algorithms	6 Dynamic Programming Algorithms	7 Divide-and-Conquer Algorithms	8 Graph Algorithms	9 Combinatorial Pattern Matching	10 Clustering and Trees	11 Hidden Markov Models	12 Randomized Algorithms
Mapping DNA	o								
Sequencing DNA					o				
Comparing Sequences			o	o		o			
Predicting Genes			o						
Finding Signals	o	o						o	o
Identifying Proteins					o				
Repeat Analysis						o			
DNA Arrays					o				
Genome Rearrangements		o							
Molecular Evolution							o		

Source: An Introduction to Bioinformatics Algorithms, N.C. Jones and P.A. Pevzner, The MIT Press Cambridge, 2004

Disclaimer on lecture notes

Some of the following material has been reproduced from existing lecture notes by

- Luigia Petre (with her permission), Åbo Akademi University (*Approximation and Randomized Algorithms* lecture notes)
- Ana Bell, Eric Grimson, John Guttag (MIT OpenCourseWare CC-BY-NC SA license), Massachusetts Technical University (*Introduction to Computer Science and Programming in Python* lecture notes)
- Erik D. Demaine, Srini Devadas (MIT OpenCourseWare CC-BY-NC SA license), Massachusetts Technical University (*Introduction to Algorithms* lecture notes)

Stable matching problem

- 1962
  - David Gale and Lloyd Shapley -> mathematical economists
  - Could one design a *job recruiting process* that is **self-enforcing**?
  - 1950s: National Resident Matching Program
- Given
  - Set of preferences among employers and applicants
- Can we assign applicants to employers so that for every employer E and every applicant A who is not scheduled to work for E, we have at least one of:
  - E prefers every one of its accepted applicants to A
  - A prefers the current situation over working at E

Courtesy of L. Petre

Matching Residents to Hospitals

Goal. Given a set of preferences among hospitals and medical school students, design a self-reinforcing admissions process.

Unstable pair: applicant x and hospital y make an unstable pair if:

- x prefers y to its assigned hospital.
- y prefers x to one of its admitted students.

Stable assignment. Assignment with no unstable pairs.

- Natural and desirable condition.
- Individual self-interest will prevent any applicant/hospital deal from being made.

Courtesy of L. Petre

Stable Matching Problem

Goal. Given n men and n women, find a "suitable" matching.

- Participants rate members of opposite sex.
- Each man lists women in order of preference from best to worst.
- Each woman lists men in order of preference from best to worst.

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

favorite ↓ least favorite ↓

	1st	2nd	3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Courtesy of L. Petre

Stable Matching Problem

Perfect matching: everyone is matched monogamously.

- Each man gets exactly one woman.
- Each woman gets exactly one man.

Stability: no incentive for some pair of participants to undermine assignment by joint action.

- In matching M, an unmatched pair m-w is unstable if man m *and* woman w prefer each other to current partners.
- Unstable pair m-w could each improve by eloping.

Stable matching: perfect matching with no unstable pairs.

Stable matching problem. Given the preference lists of n men and n women, find a stable matching if one exists.

Courtesy of L. Petre

Questions

1. Does there exist a stable matching for every set of preference lists?
2. Given a set of preference lists, can we *efficiently* construct a stable matching if one exists?

Courtesy of L. Petre

Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

favorite ↓ least favorite ↓

	1st	2nd	3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Courtesy of L. Petre

Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?

A. No. Bertha and Xavier will hook up.

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

favorite ↓ least favorite ↓

	1st	2nd	3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Courtesy of L. Petre

Stable Matching Problem

Q. Is assignment X-A, Y-B, Z-C stable?

A. Yes.

favorite ↓ least favorite ↓

	1st	2nd	3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

favorite ↓ least favorite ↓

	1st	2nd	3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Courtesy of L. Petre

Propose-And-Reject Algorithm

Propose-and-reject algorithm. [Gale-Shapley 1962] Intuitive method that guarantees to find a stable matching.

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every
woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet
proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
```

Courtesy of L. Petre

Proof of Correctness: Termination

Observation 1. Men propose to women in decreasing order of preference.

Observation 2. Once a woman is matched, she never becomes unmatched; she only "trades up."

Claim. Algorithm terminates after at most n^2 iterations of while loop.

Pf. Each time through the while loop a man proposes to a new woman. There are only n^2 possible proposals. ▪

	1st	2nd	3rd	4th	5th
Victor	A	B	C	D	E
Wyatt	B	C	D	A	E
Xavier	C	D	A	B	E
Yancey	D	A	B	C	E
Zeus	A	B	C	D	E

	1st	2nd	3rd	4th	5th
Amy	W	X	Y	Z	V
Bertha	X	Y	Z	V	W
Clare	Y	Z	V	W	X
Diane	Z	V	W	X	Y
Erika	V	W	X	Y	Z

n(n-1) + 1 proposals required

Courtesy of L. Petre

Proof of Correctness: Perfection

Claim. All men and women get matched.

Pf. (by contradiction)

- Suppose, for sake of contradiction, that Zeus is not matched upon termination of algorithm.
- Then some woman, say Amy, is not matched upon termination.
- By Observation 2, Amy was never proposed to.
- But, Zeus proposes to everyone, since he ends up unmatched. ▪

Courtesy of L. Petre

Proof of Correctness: Stability

Claim. No unstable pairs.

Pf. (by contradiction)

- Suppose A-Z is an unstable pair: each prefers each other to partner in Gale-Shapley matching S*.

men propose in decreasing order of preference

- Case 1: Z never proposed to A.
  - ⇒ Z prefers his GS partner to A.
  - ⇒ A-Z is stable.

- Case 2: Z proposed to A.
  - ⇒ A rejected Z (right away or later)
  - ⇒ A prefers her GS partner to Z.
  - ⇒ A-Z is stable.

women only trade up

S*

Amy-Yancey
Bertha-Zeus
. . .

- In either case A-Z is stable, a contradiction. ▪

Courtesy of L. Petre

Summary

Stable matching problem. Given n men and n women, and their preferences, find a stable matching if one exists.

Gale-Shapley algorithm. Guarantees to find a stable matching for any problem instance.

Q. If there are multiple stable matchings, which one does GS find?

Courtesy of L. Petre

More on stable matching algorithm

- Symmetry of the problem vs algorithm
  - Stable matching problem is symmetric w.r.t. to men and women
  - The G-S algorithm is asymmetric: If all men put different women as their first choice, they will end up with their first choice
  - The women's preferences disregarded, introducing unfairness to the algorithm

- Deterministic vs undeterministic algorithm

```
while (some man is free and hasn't proposed to every woman) {
        Choose such a man m
```

  - The algorithm does not specify which man should be choosen

Stable matching problem

- Enough precision to
  - ask concrete questions
  - start thinking about an algorithm to solve the problem
- Design algorithm for problem
- Analyze algorithm
  - Correctness
  - Bound on running time

- Fundamental design techniques

Courtesy of L. Petre

Algorithm analysis

- How do resource requirements change when input size increases?
  - Time, space
  - Notational machinery
- Problems of discrete nature
  - Implicit searching over large space of possibilities
  - Goal: efficiently find solution satisfying conditions
- Focus on running time

Courtesy of L. Petre

Algorithm efficiency

1. An algorithm is efficient if, when implemented, it runs quickly on real-input instances

Problems with this

- Where we implement an algorithm
- How well we implement an algorithm

BETTER definition

- Platform-independent
- Instance-independent
- Predictive value with respect to increasing input size

Courtesy of L. Petre

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Courtesy of L. Petre

Algorithm efficiency

1. An algorithm is efficient if, when implemented, it runs quickly on real-input instances
2. An algorithm is **efficient** if it achieves a better worst-case performance, at an *analytical* level, then brute-force search

- → Brute-force search provides no insight into the structure of the problem we are studying!
- → Definition somewhat vague
  - → "better performance"?

Courtesy of L. Petre

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n ! for stable matching with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c\,N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Courtesy of L. Petre

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method

Courtesy of L. Petre

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Courtesy of L. Petre

More on the definition of efficiency in terms of poly-time

- This definition is negatable: we can say *when there is no efficient algorithm for a particular problem*
- Previous definitions were subjective
  - First definition turned efficiency into a moving target
  - The poly-time definition is more absolute
- Promotes the idea that problems have an intrinsic level of computational tractability
  - Some admit efficient solutions, some do not

Courtesy of L. Petre

Asymptotic Order of Growth

Basic assumption: an algorithm's worst-case running time on inputs of size n grows at a rate at most proportional to some function f(n)

Upper bounds. T(n) is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have **$T(n) \leq c \cdot f(n)$**.

Lower bounds. T(n) is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have **$T(n) \geq c \cdot f(n)$**.

Tight bounds. T(n) is $\Theta(f(n))$ if T(n) is *both* **$O(f(n))$** and **$\Omega(f(n))$**.

Ex: $T(n) = 32n^2 + 17n + 32$.

- T(n) is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- T(n) is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Courtesy of L. Petre

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.

↑ can avoid specifying the base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

↑ log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

↑ every exponential grows faster than every polynomial

Courtesy of L. Petre

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Courtesy of L. Petre

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: `n` is `len(L)`
  - index
  - store
  - length
  - append
  - ==
  - remove
  - copy
  - reverse
  - iteration
  - in list
- Dictionaries: `n` is `len(d)`
- worst case
  - index
  - store
  - length
  - delete
  - iteration
- average cas
  - index
  - store
  - delete
  - iteration

Source: MIT OpenCourseWare

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

Source: MIT OpenCourseWare

See wiki.python.org/moin/TimeComplexity for more info on the (amortized) worst case performance

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
```


Source: MIT OpenCourseWare

LOGARITHMIC COMPLEXITY

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return result
```

only have to look at loop as no function calls

within while loop, constant number of steps

how many times through loop?

- how many times can one divide i by 10?
- *O(log(i))*

O() FOR ITERATIVE FACTORIAL

- complexity can depend on number of iterative calls

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

- overall ***O(n)*** – n times round loop, constant cost each time

O() FOR RECURSIVE FACTORIAL

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n – 1)
```

- computes factorial recursively
- if you time it, may notice that it runs a bit slower than iterative version due to function calls
- still ***O(n)*** because the number of function calls is linear in n, and constant effort to set up call
- **iterative and recursive factorial** implementations are the **same order of growth**

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant O(1)

constant O(1)

linear O(n)

constant O(1)

- Best case:

  O(1)

- Worst case:

  O(1) + O(n) + O(1) ➔ **O(n)**

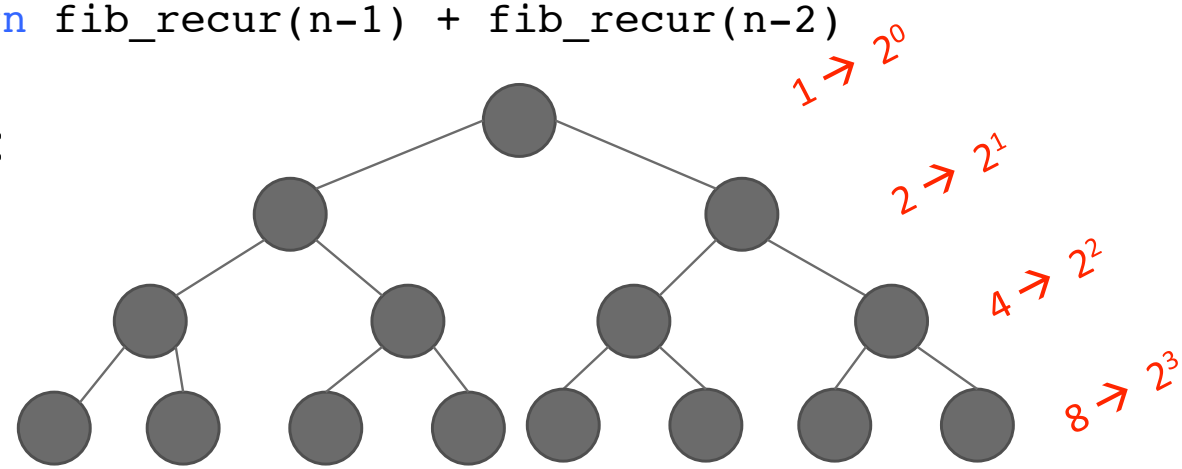
Source: MIT OpenCourseWare

COMPLEXITY OF RECURSIVE FIBONACCI

```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

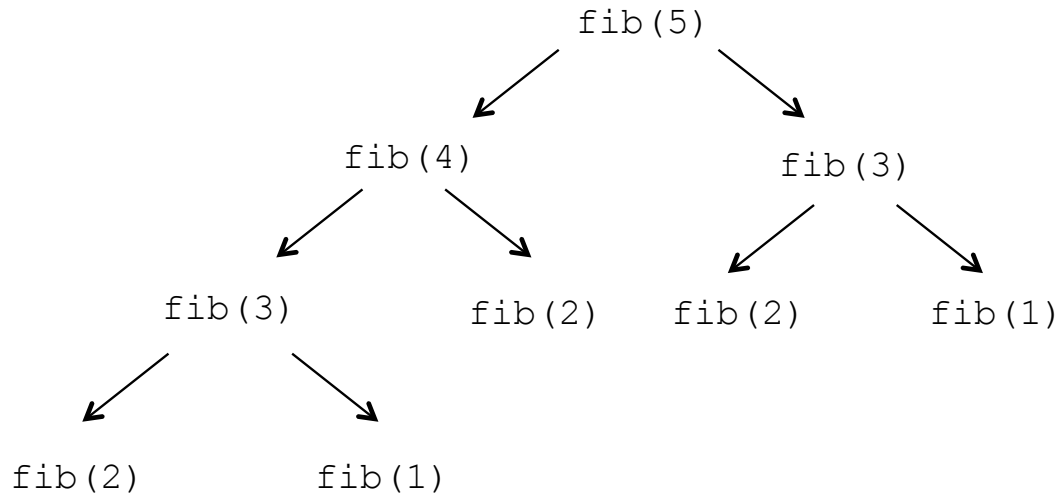
- Worst case:

  $O(2^n)$



Source: MIT OpenCourseWare

COMPLEXITY OF RECURSIVE FIBONACCI



- actually can do a bit better than 2^n since tree of cases thins out to right
- but complexity is still exponential

Source: MIT OpenCourseWare

Various algorithm design techniques

- Exhaustive Search
- Branch-and-Bound Algorithms
- Greedy Algorithms
- Dynamic Programming
- Divide-and-Conquer Algorithms
- Machine Learning
- Randomized Algorithms

Various algorithm design techniques

- Exhaustive Search
- Branch-and-Bound Algorithms
- Greedy Algorithms
- Dynamic Programming
- Divide-and-Conquer Algorithms
- Machine Learning
- Randomized Algorithms

Computational tractablity in relation to the choice of the algorithms and data structures

LINEAR SEARCH ON UNSORTED LIST: RECAP

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

speed up a little by returning True here, but speed up doesn't impact worst case

- must look through all elements to decide it's not there
- O(len(L)) for the loop * O(1) to test if e == L[i]
- overall complexity is **O(n) – where n is len(L)**

Assumes we can retrieve element of list in constant time

LINEAR SEARCH ON SORTED LIST: RECAP

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- O(len(L)) for the loop * O(1) to test if e == L[i]
- overall complexity is **O(n) – where n is len(L)**

SEARCHING A SORTED LIST -- n is len(L)

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
  - assumes the **list is sorted**!
- when does it make sense to **sort first then search**?
  - SORT + O(`log n`) < O(`n`) → SORT < O(`n`) – O(`log n`)
  - when sorting is less than O(`n`)
- **NEVER TRUE!**
  - **to sort a collection of n elements must look at each one at least once!**

AMORTIZED COST -- n is len(L)

- why bother sorting first?
- in some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- SORT + `K`*O(`log n`) < `K`*O(`n`)

→ for large `K`, **SORT time becomes irrelevant,** if cost of sorting is small enough

The problem of sorting

Input: array $A[1 \ldots n]$ of numbers.

Output: permutation $B[1 \ldots n]$ of A such that $B[1] \leq B[2] \leq \cdots \leq B[n]$.

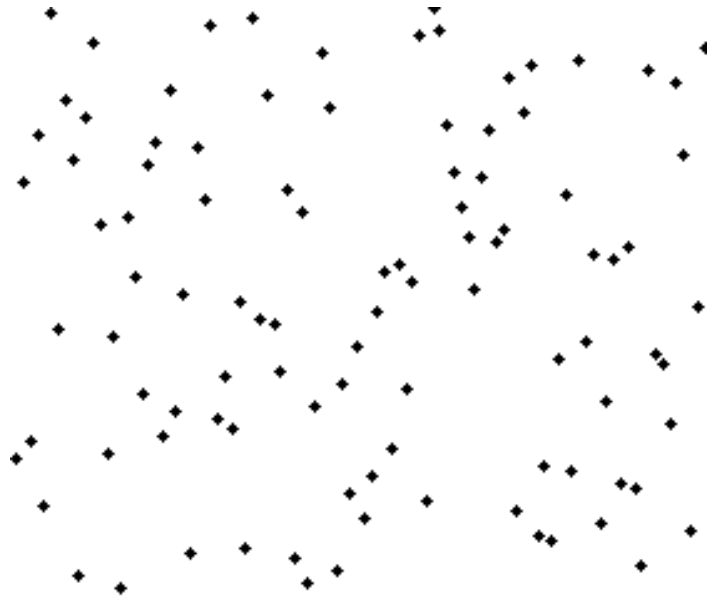
e.g. A = [7, 2, 5, 5, 9.6] → B = [2, 5, 5, 7, 9.6]

How can we do it efficiently ?

Source: MIT OpenCourseWare

BUBBLE SORT

- **compare consecutive pairs** of elements
- **swap elements** in pair such that smaller is first
- when reach end of list, **start over** again
- stop when **no more swaps** have been made
- largest unsorted element always at end after pass, so at most n passes



CC-BY Hydrargyrum
https://commons.wikimedia.org/wiki/File:Bubble_sort_animation.gif

Source: MIT OpenCourseWare

COMPLEXITY OF BUBBLE SORT

```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

O(len(L))

O(len(L))

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **$O(n^2)$ where n is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes

SELECTION SORT

- first step
  - extract **minimum element**
  - **swap it** with element at **index 0**
- subsequent step
  - in remaining sublist, extract **minimum element**
  - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
  - at i'th step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

ANALYZING SELECTION SORT

- loop invariant
  - given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix
    1. base case: prefix empty, suffix whole list – invariant true
    2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
    3. when exit, prefix is entire list, suffix empty, so sorted

COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

len(L) times → O(len(L))

len(L) – suffixSt times → O(len(L))

- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **O(n^2) where n is len(L)**

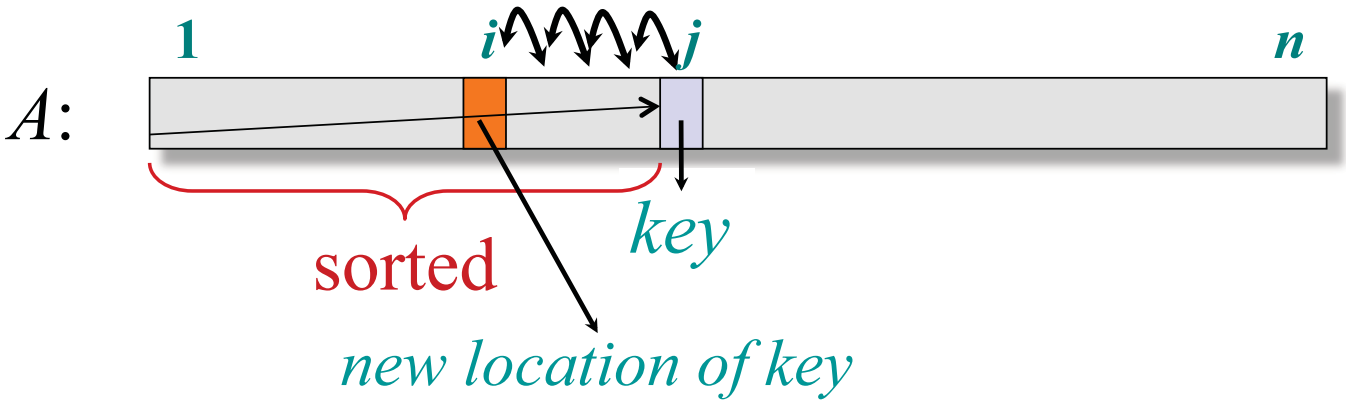
Insertion sort

INSERTION-SORT (A, n) $\triangleright$ $A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \,..\, j\text{-}1]$**.**
by pairwise key-swaps down to its right position

Illustration of iteration *j*



Source: MIT OpenCourseWare

Example of insertion sort

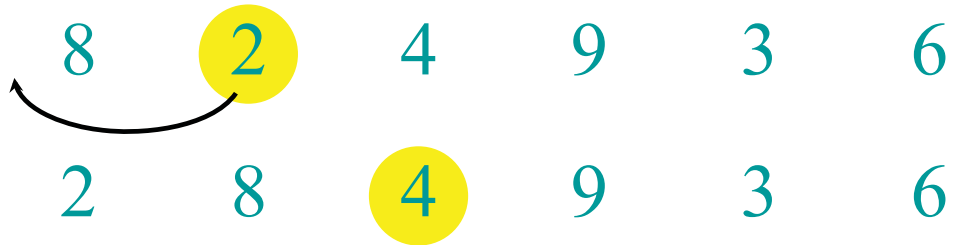
8 2 4 9 3 6

Source: MIT OpenCourseWare

Example of insertion sort

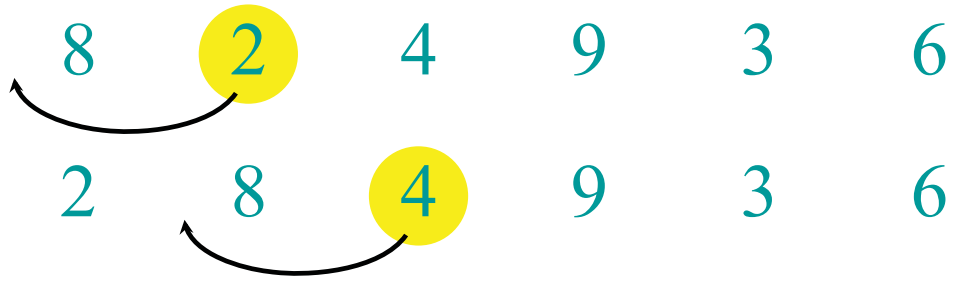
Source: MIT OpenCourseWare

Example of insertion sort



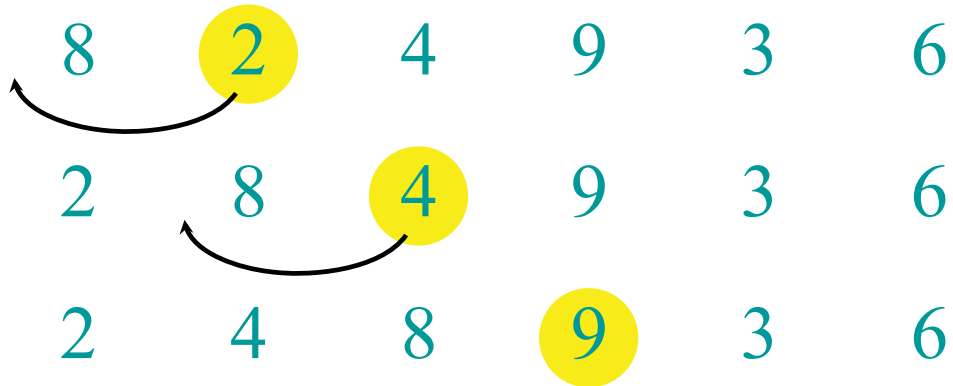
Source: MIT OpenCourseWare

Example of insertion sort



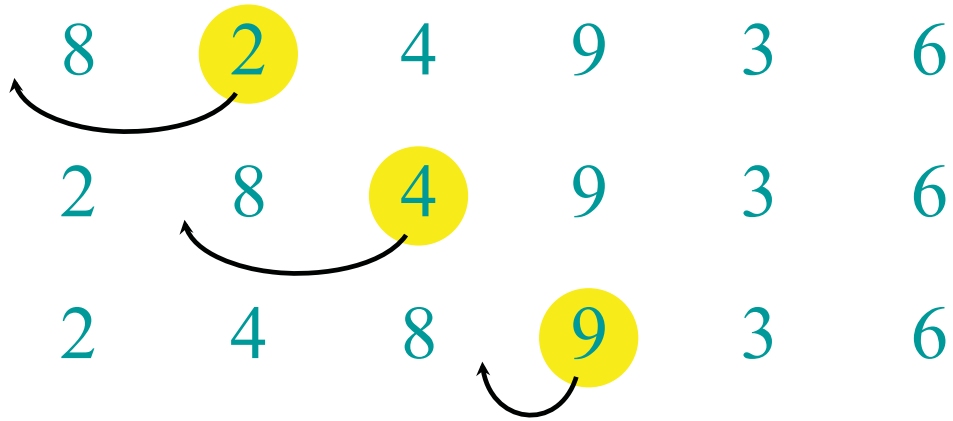
Source: MIT OpenCourseWare

Example of insertion sort



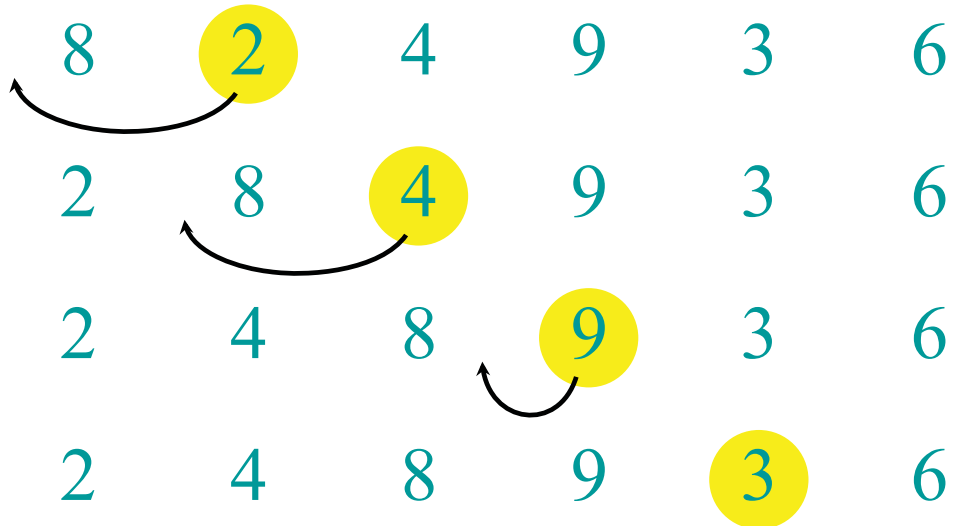
Source: MIT OpenCourseWare

Example of insertion sort



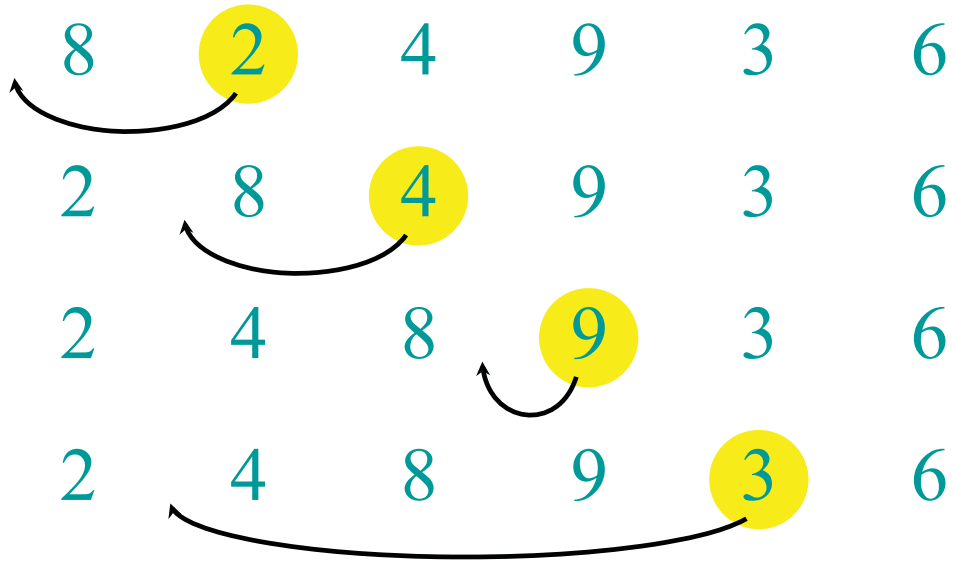
Source: MIT OpenCourseWare

Example of insertion sort



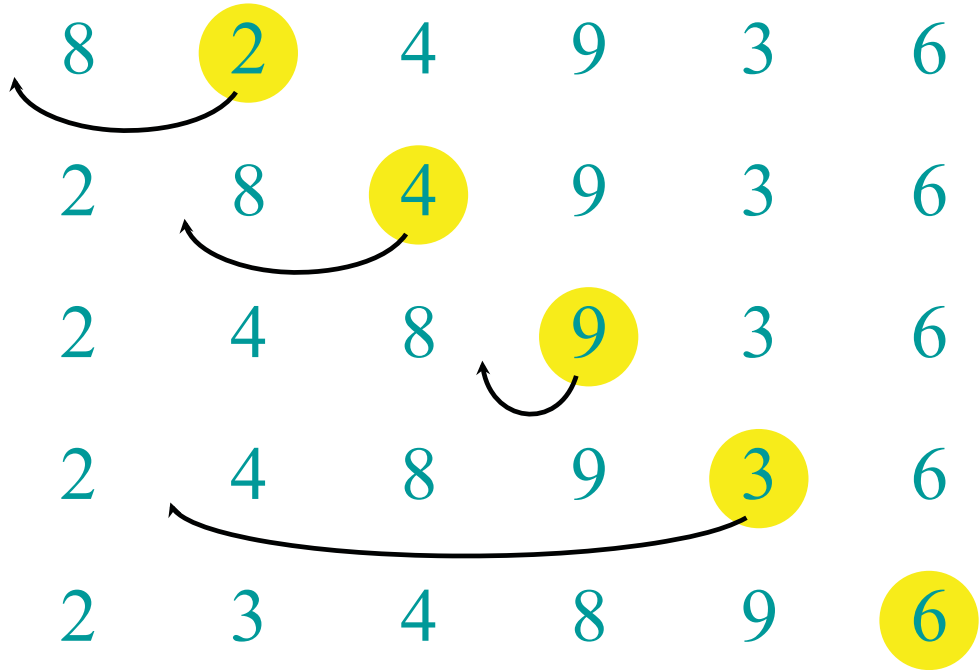
Source: MIT OpenCourseWare

Example of insertion sort



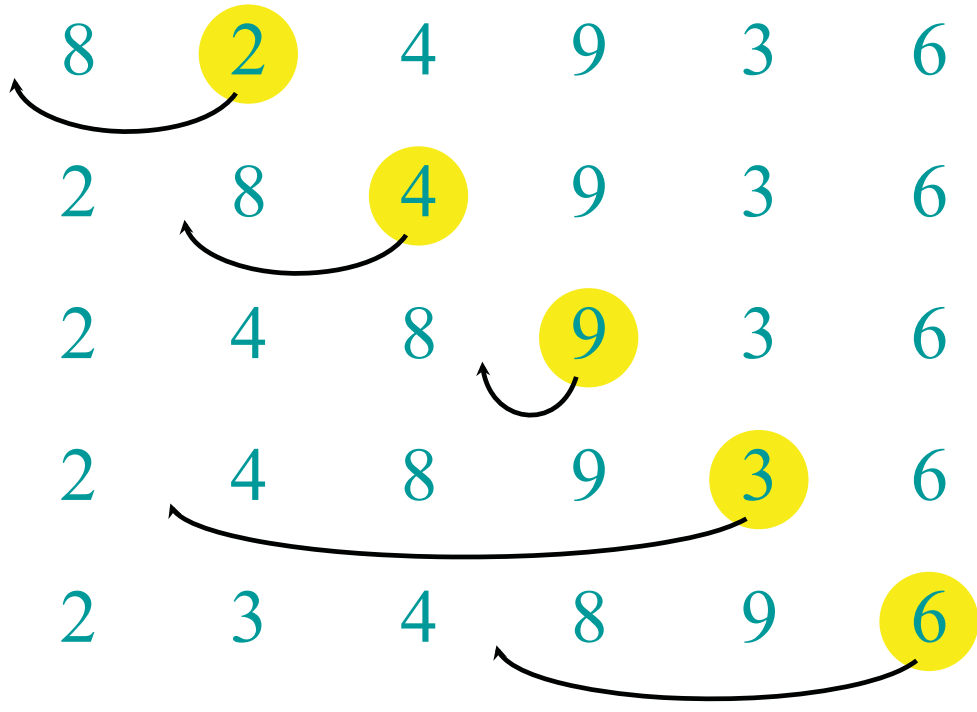
Source: MIT OpenCourseWare

Example of insertion sort



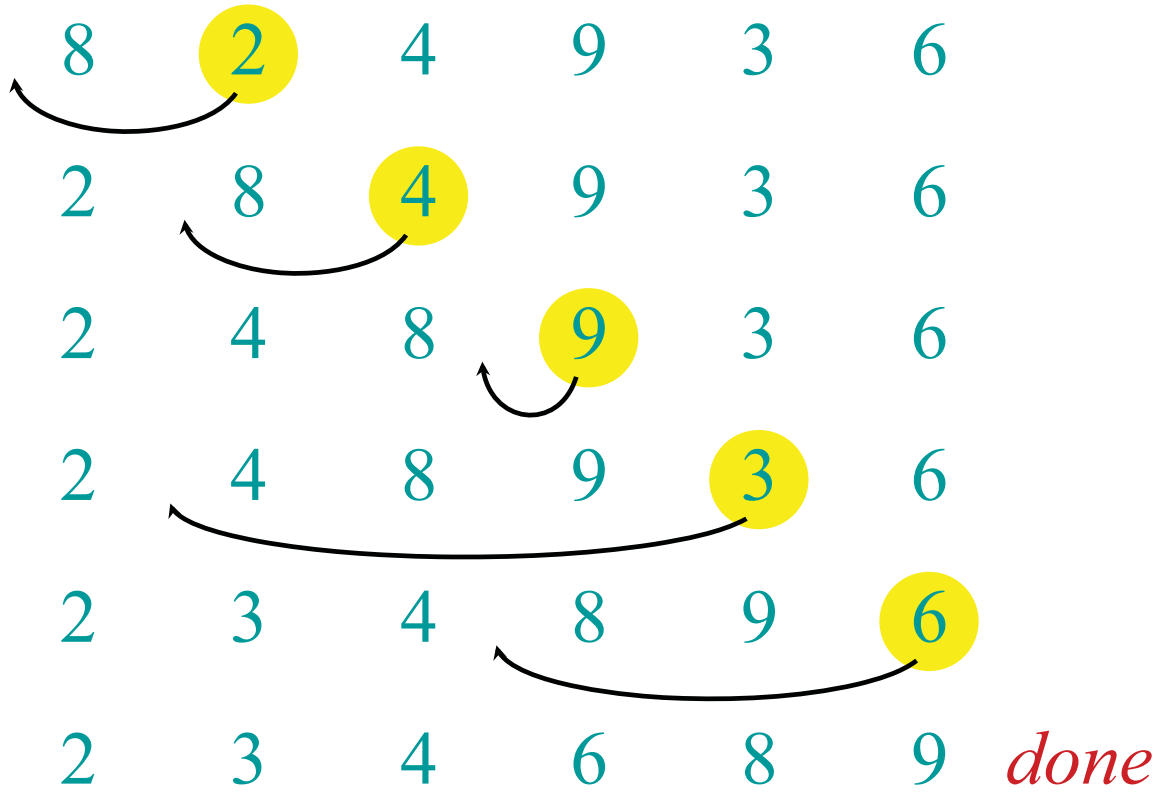
Source: MIT OpenCourseWare

Example of insertion sort



Source: MIT OpenCourseWare

Example of insertion sort



Running time? $\Theta(n^2)$ because $\Theta(n^2)$ compares and $\Theta(n^2)$ swaps

e.g. when input is A = [n, n − 1, n −

Source: MIT OpenCourseWare

16

Binary Insertion sort

BINARY-INSERTION-SORT (A, n) $\triangleright$ $A[1 \ldots n]$

for $j \leftarrow 2$ **to** n

insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \ldots j\text{-}1]$**.**

Use binary search to find the right position

Binary search with take $\Theta(\log n)$ time.
However, shifting the elements after insertion will still take $\Theta(n)$ time.

Complexity: $\Theta(n \log n)$ comparisons
(n^2) swaps

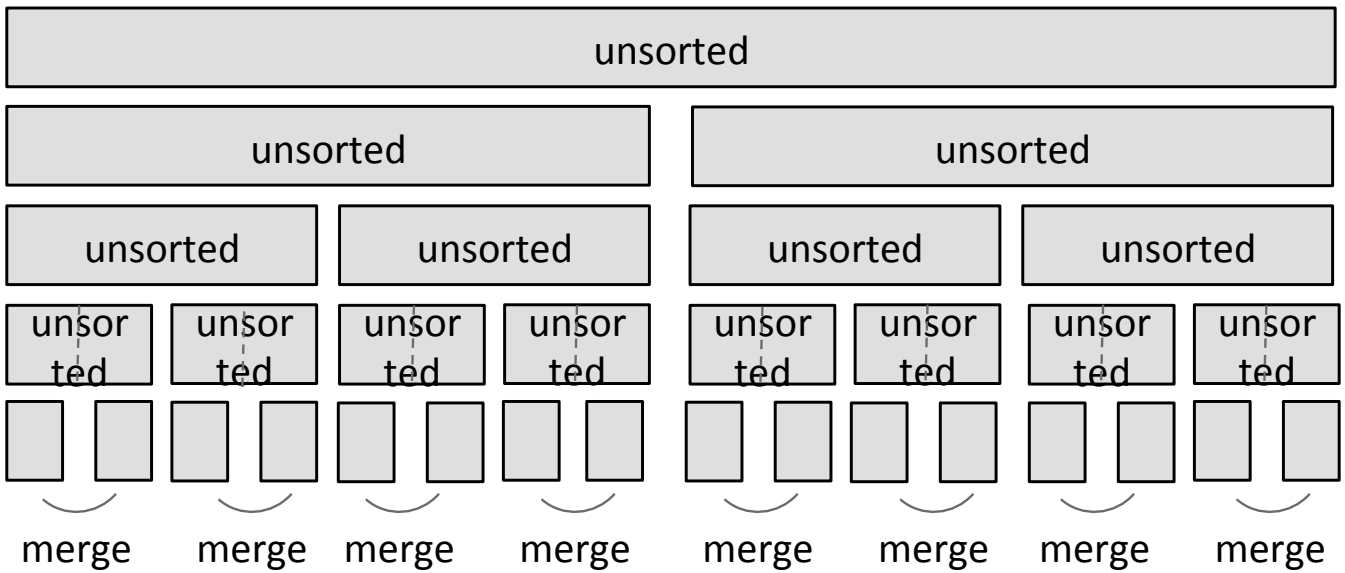
Source: MIT OpenCourseWare

MERGE SORT

- use a divide-and-conquer approach:
  1. if list is of length 0 or 1, already sorted
  2. if list has more than one element, split into two lists, and sort each
  3. merge sorted sublists
     1. look at first element of each, move smaller to end of the result
     2. when one list empty, just copy rest of other list

MERGE SORT

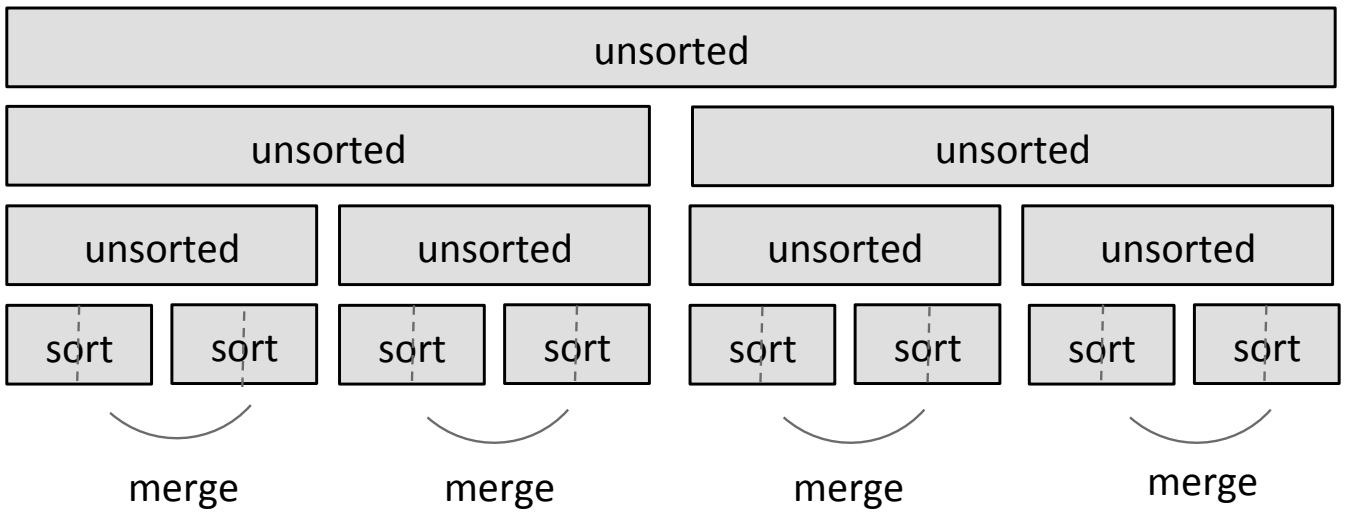
- divide and conquer



- **split list in half** until have sublists of only 1 element

MERGE SORT

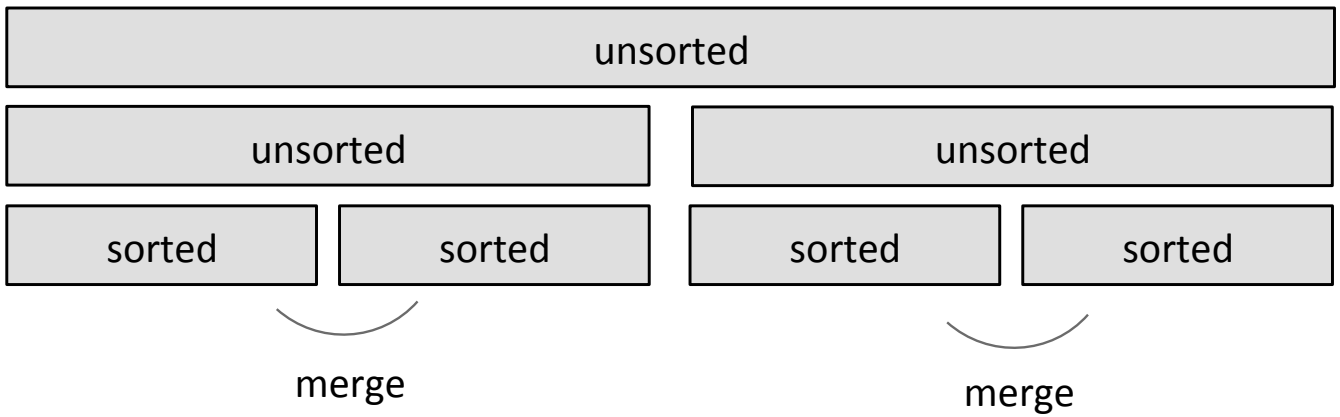
- divide and conquer



- merge such that **sublists will be sorted after merge**

MERGE SORT

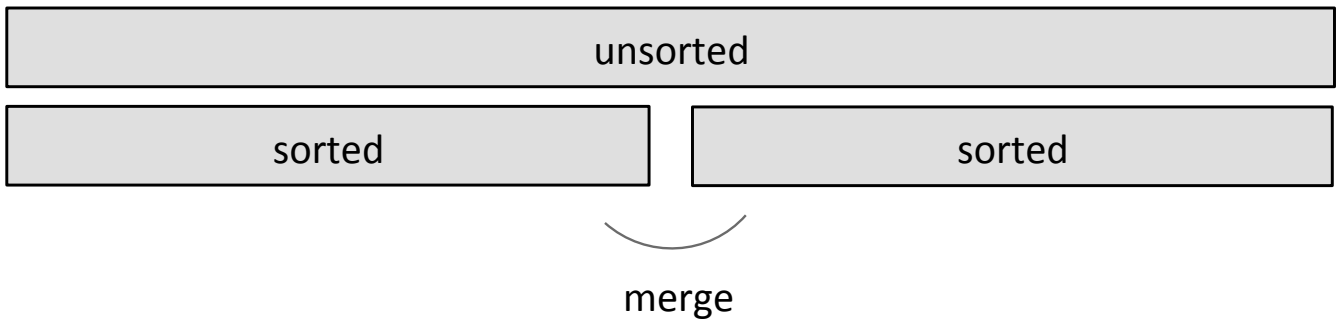
- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

- divide and conquer



- merge sorted sublists
- sublists will be sorted after merge

MERGE SORT

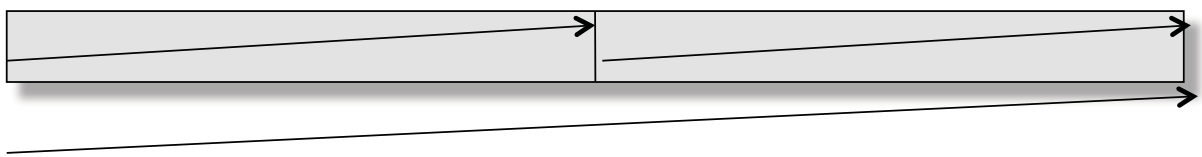
- divide and conquer – done!

Meet Merge Sort

divide and conquer

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[\,1 \ldots n/2\,]$ and $A[\,n/2+1 \ldots n\,]$.
3. "***Merge***" the two sorted sub-arrays.



Key subroutine: **MERGE**

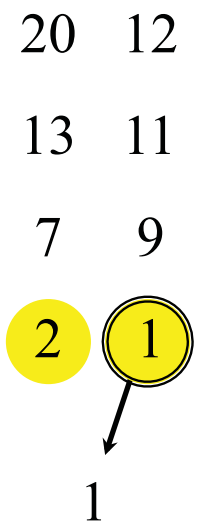
Source: MIT OpenCourseWare

Merging two sorted arrays

20	12
13	11
7	9
2	1

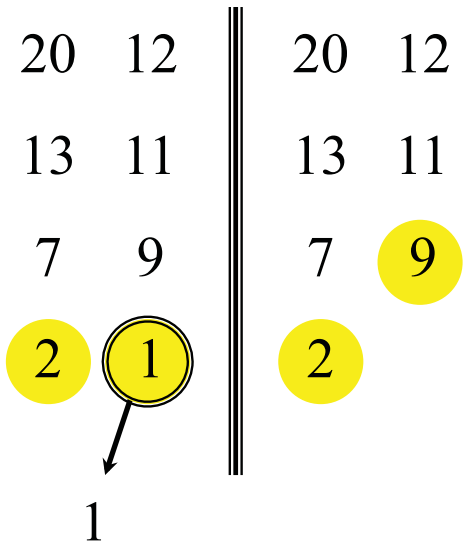
Source: MIT OpenCourseWare

Merging two sorted arrays



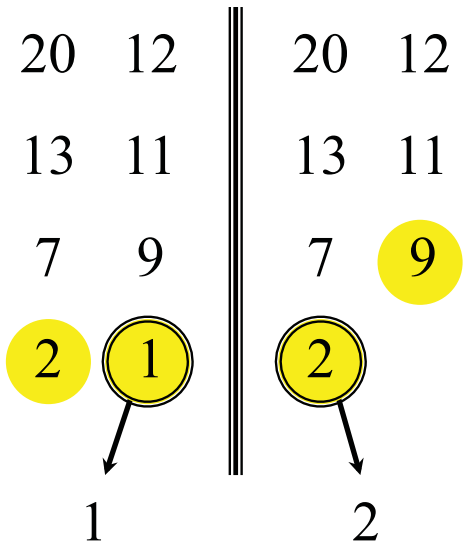
Source: MIT OpenCourseWare

Merging two sorted arrays



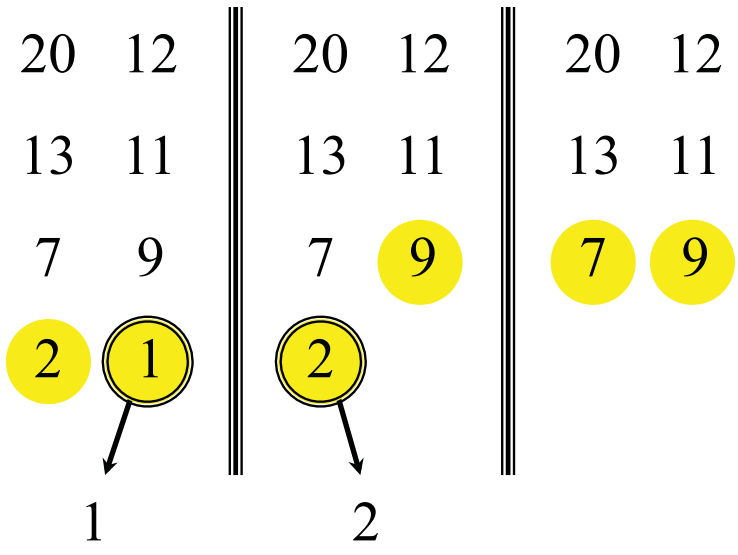
Source: MIT OpenCourseWare

Merging two sorted arrays



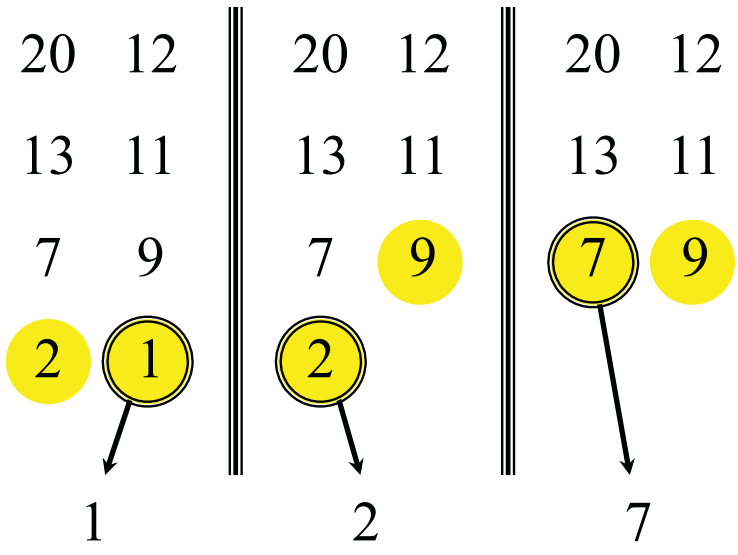
Source: MIT OpenCourseWare

Merging two sorted arrays



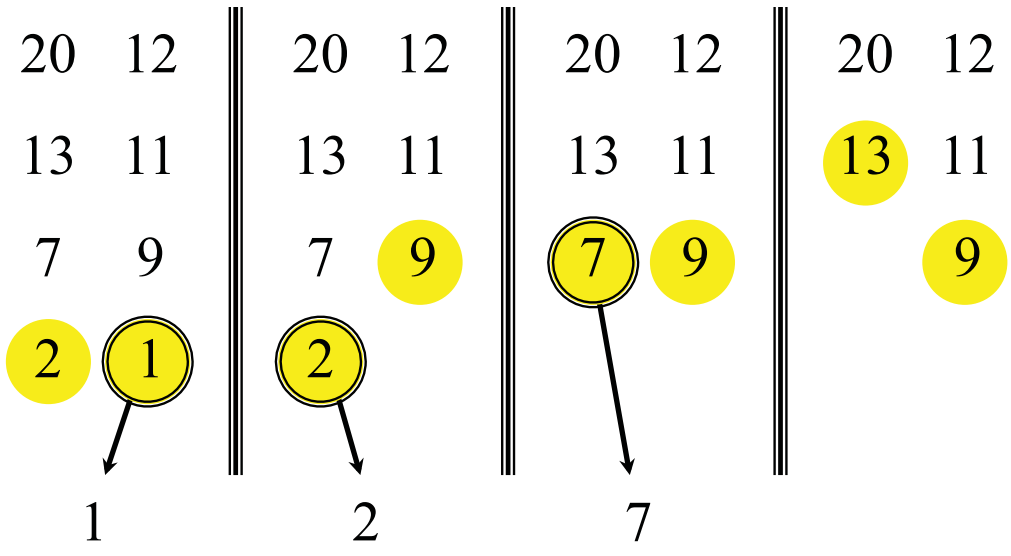
Source: MIT OpenCourseWare

Merging two sorted arrays



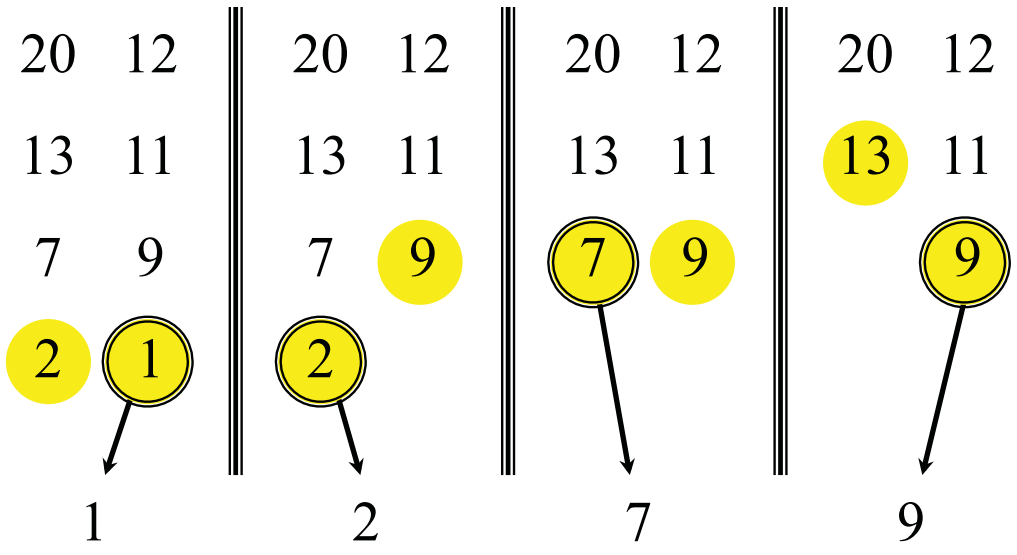
Source: MIT OpenCourseWare

Merging two sorted arrays



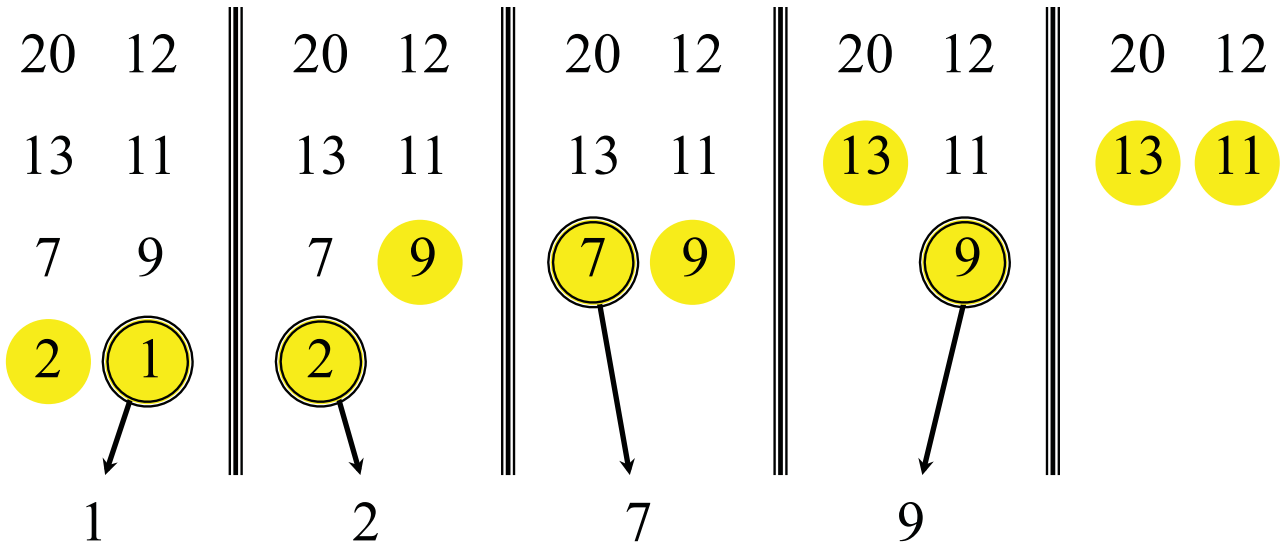
Source: MIT OpenCourseWare

Merging two sorted arrays



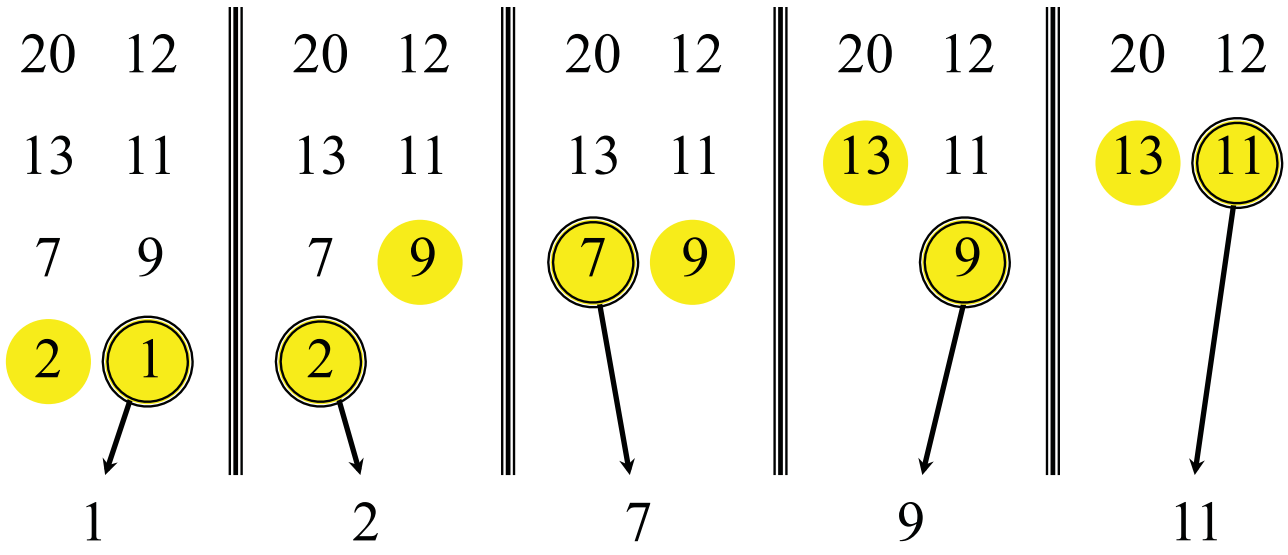
Source: MIT OpenCourseWare

Merging two sorted arrays



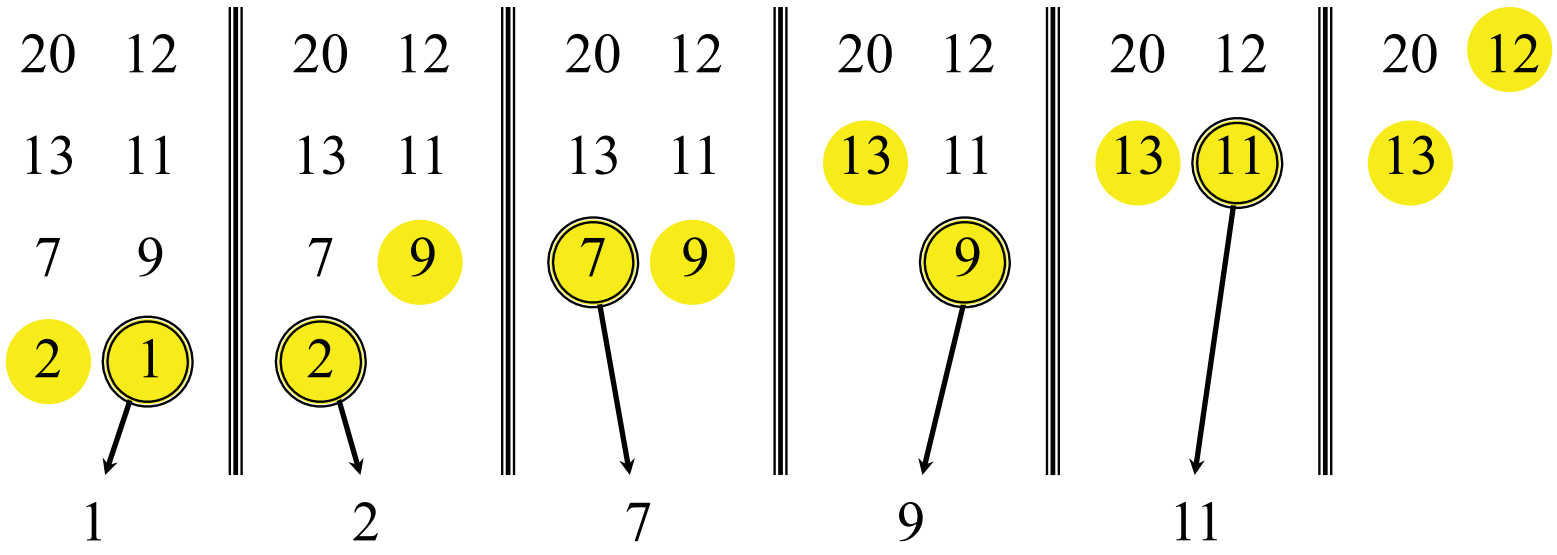
Source: MIT OpenCourseWare

Merging two sorted arrays



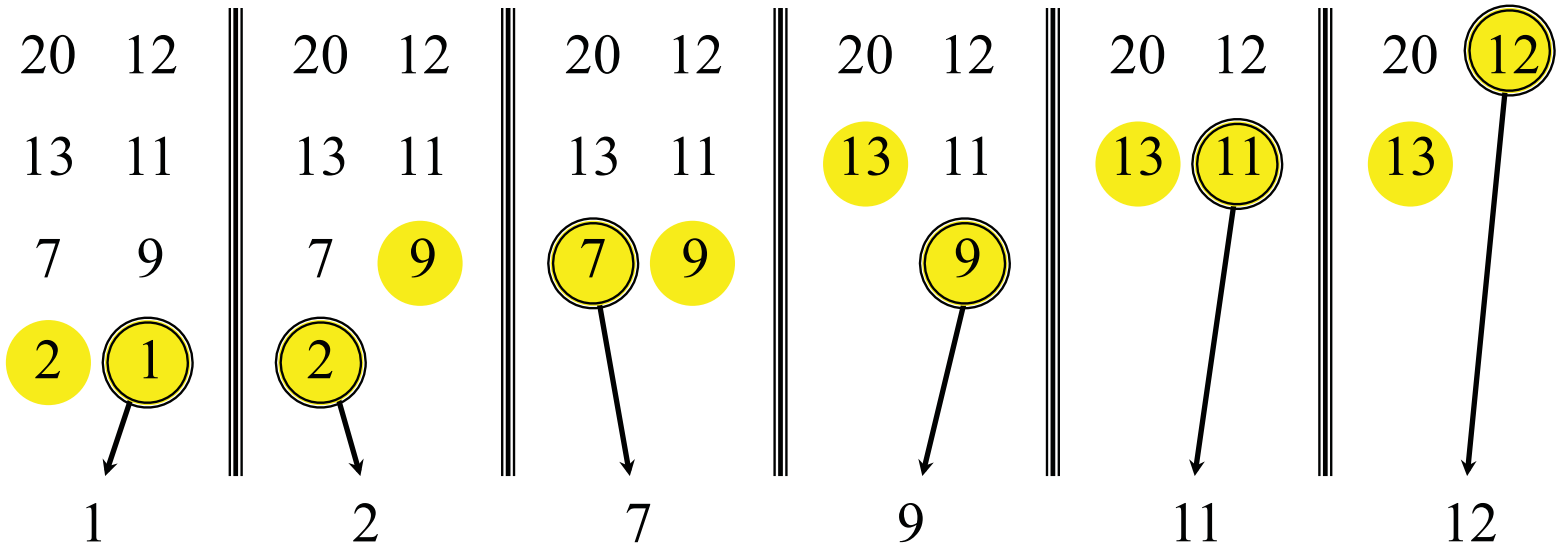
Source: MIT OpenCourseWare

Merging two sorted arrays



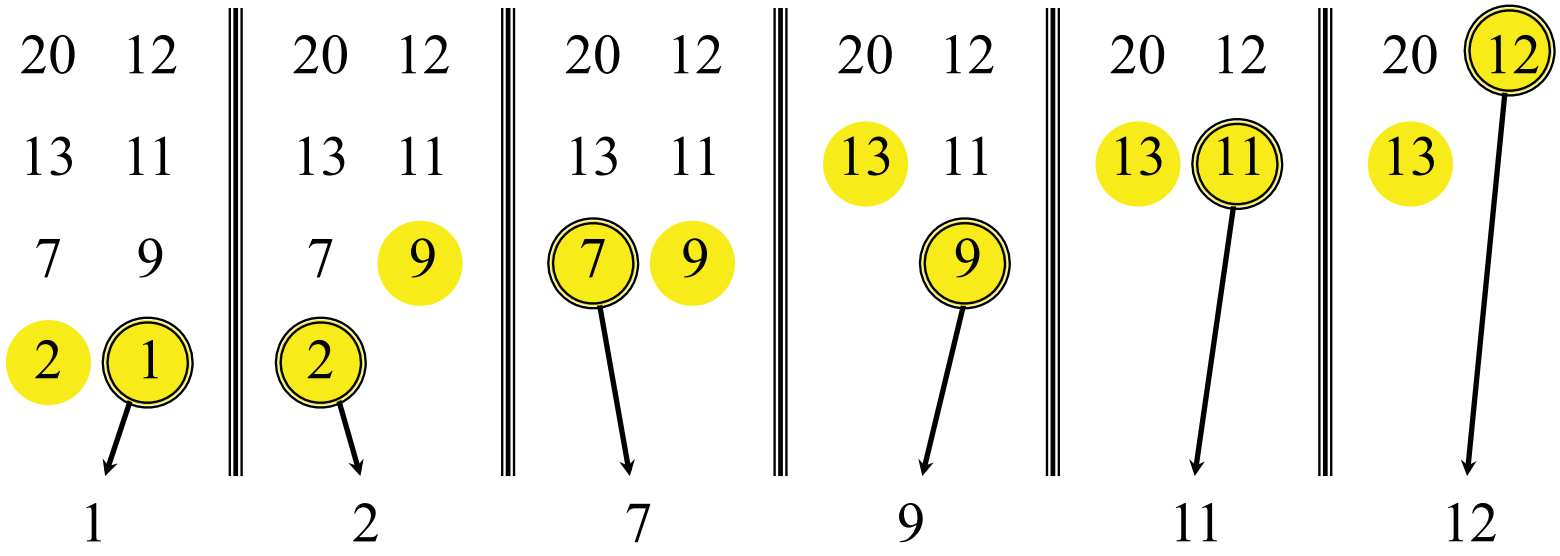
Source: MIT OpenCourseWare

Merging two sorted arrays



Source: MIT OpenCourseWare

Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Source: MIT OpenCourseWare

Analyzing merge sort

MERGE-SORT $A[1 \ldots n]$	$T(n)$
1. If $n = 1$, done.	$\Theta(1)$
2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.	$2T(n/2)$
3. "Merge" the two sorted lists	$\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$T(n) = ?$

Source: MIT OpenCourseWare

Recurrence solving

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Source: MIT OpenCourseWare

Recursion tree

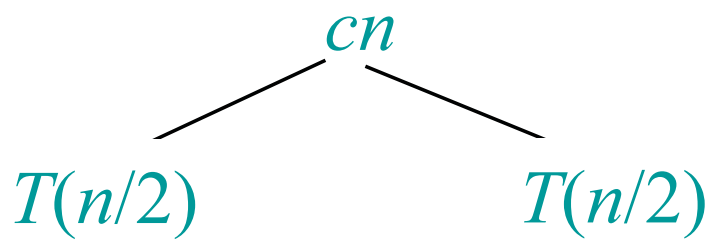
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

Source: MIT OpenCourseWare

Recursion tree

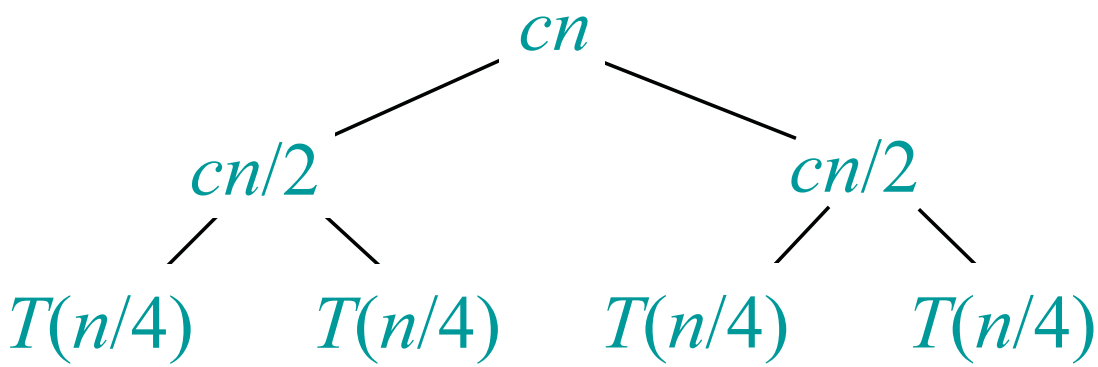
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

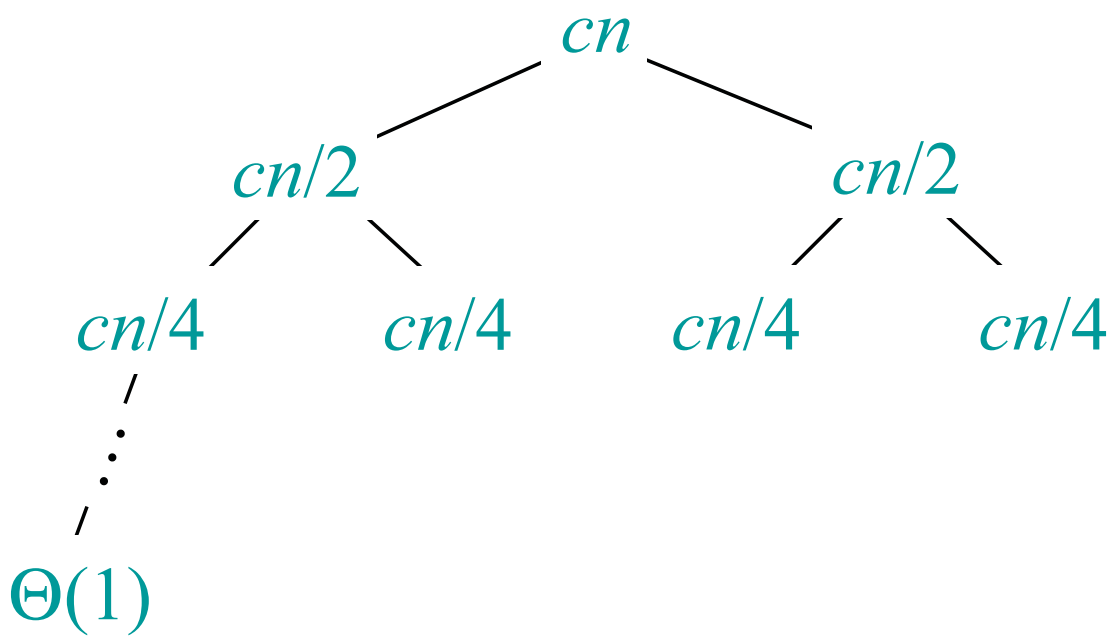
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

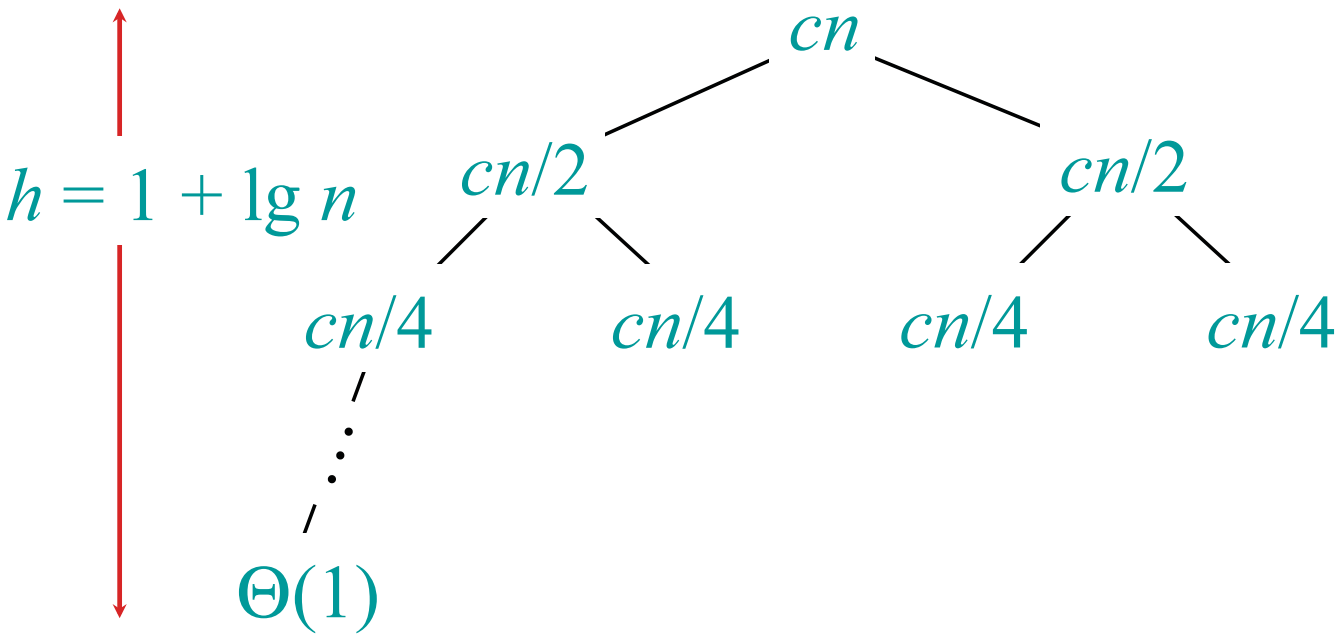
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

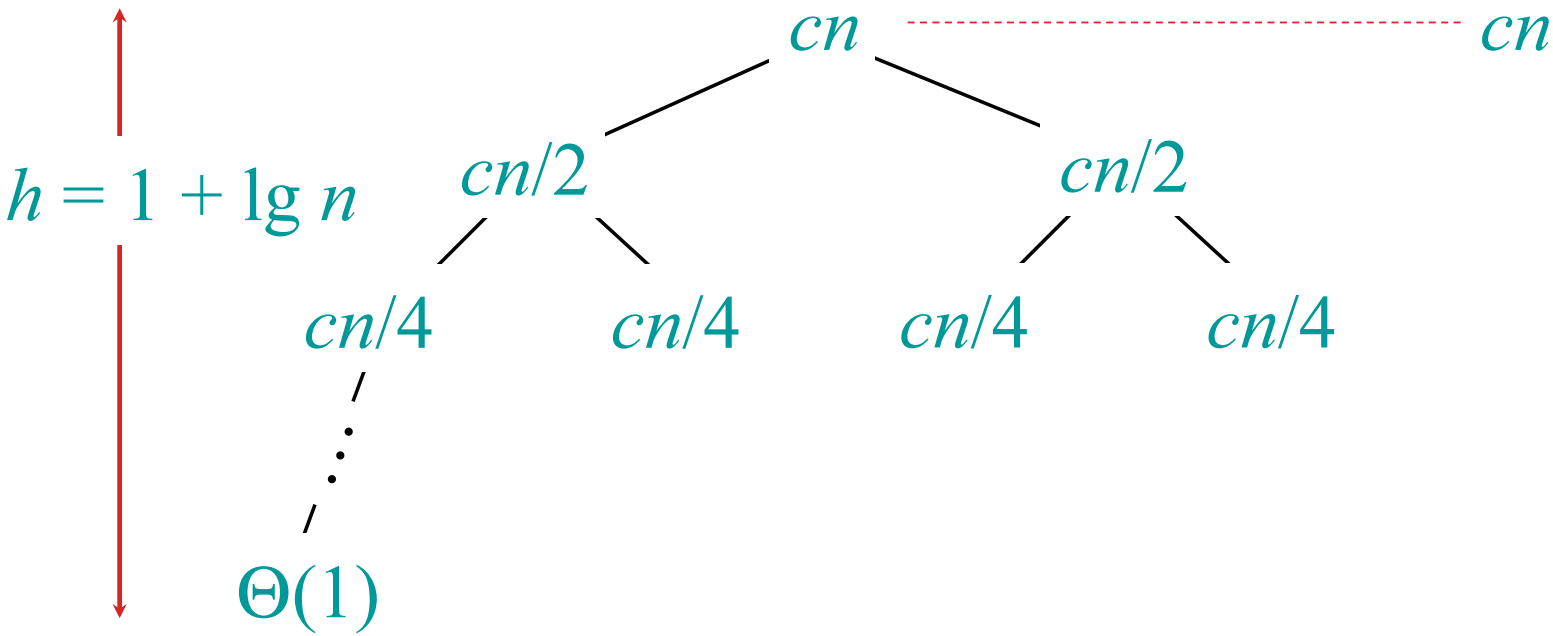
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

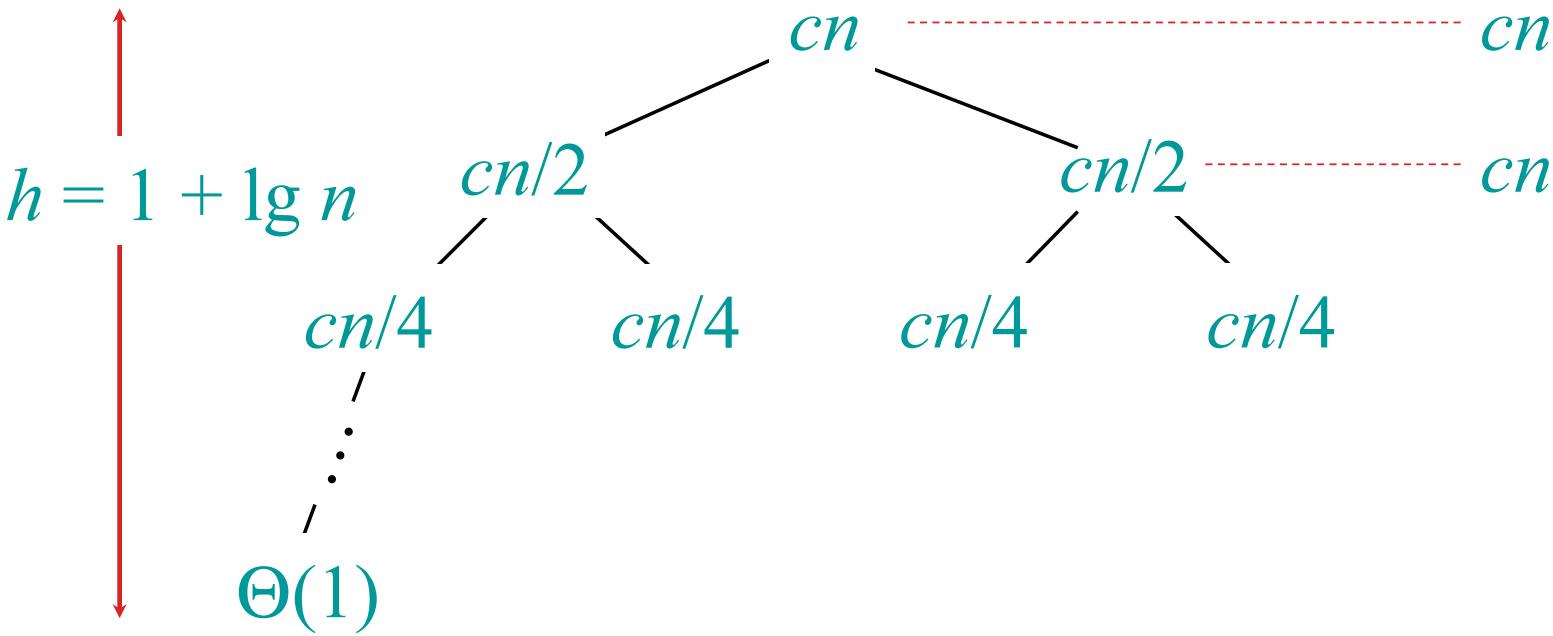
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

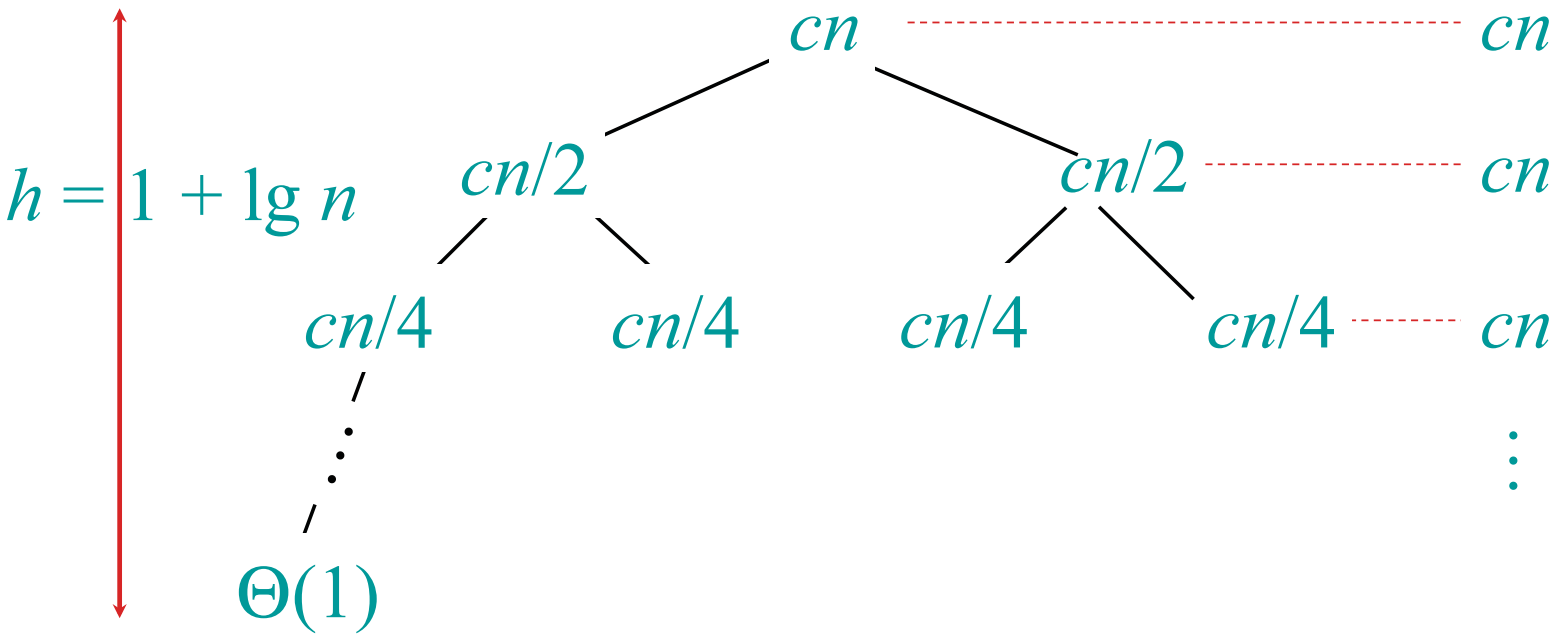
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

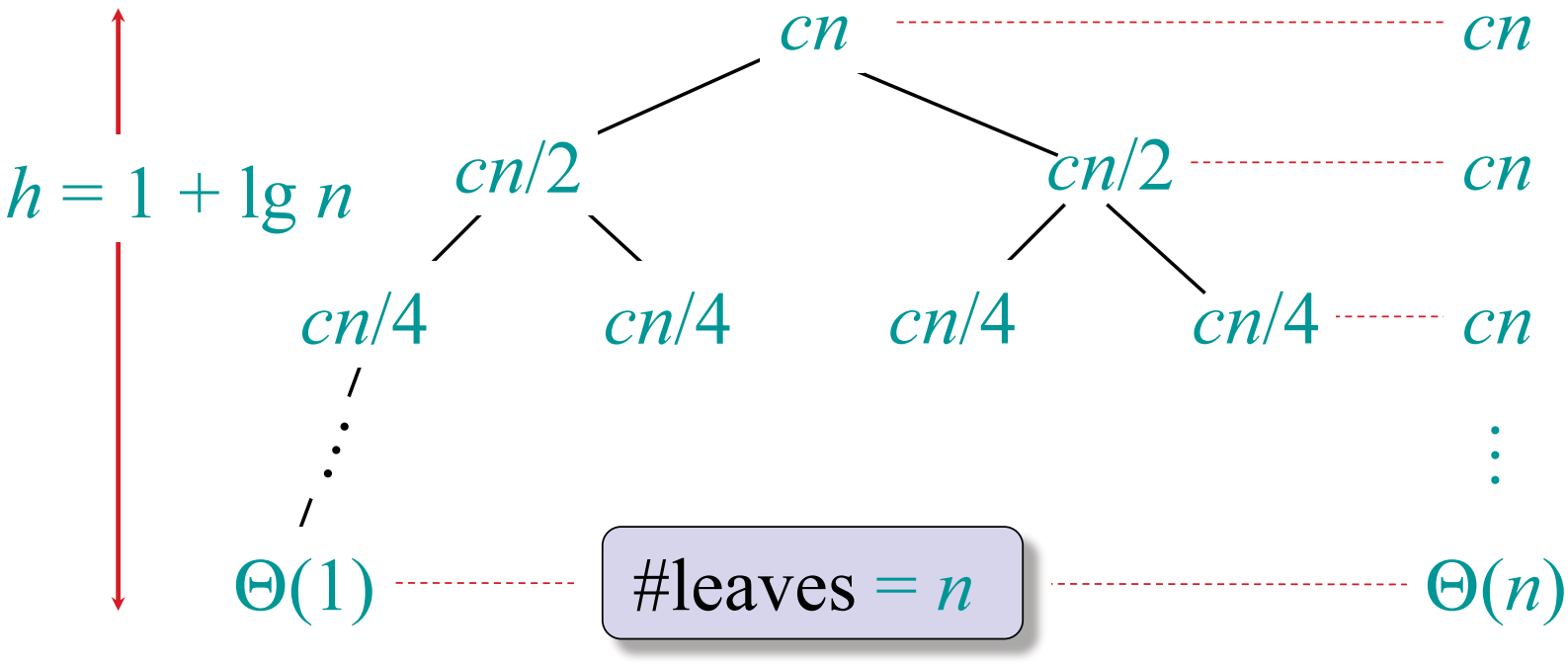
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

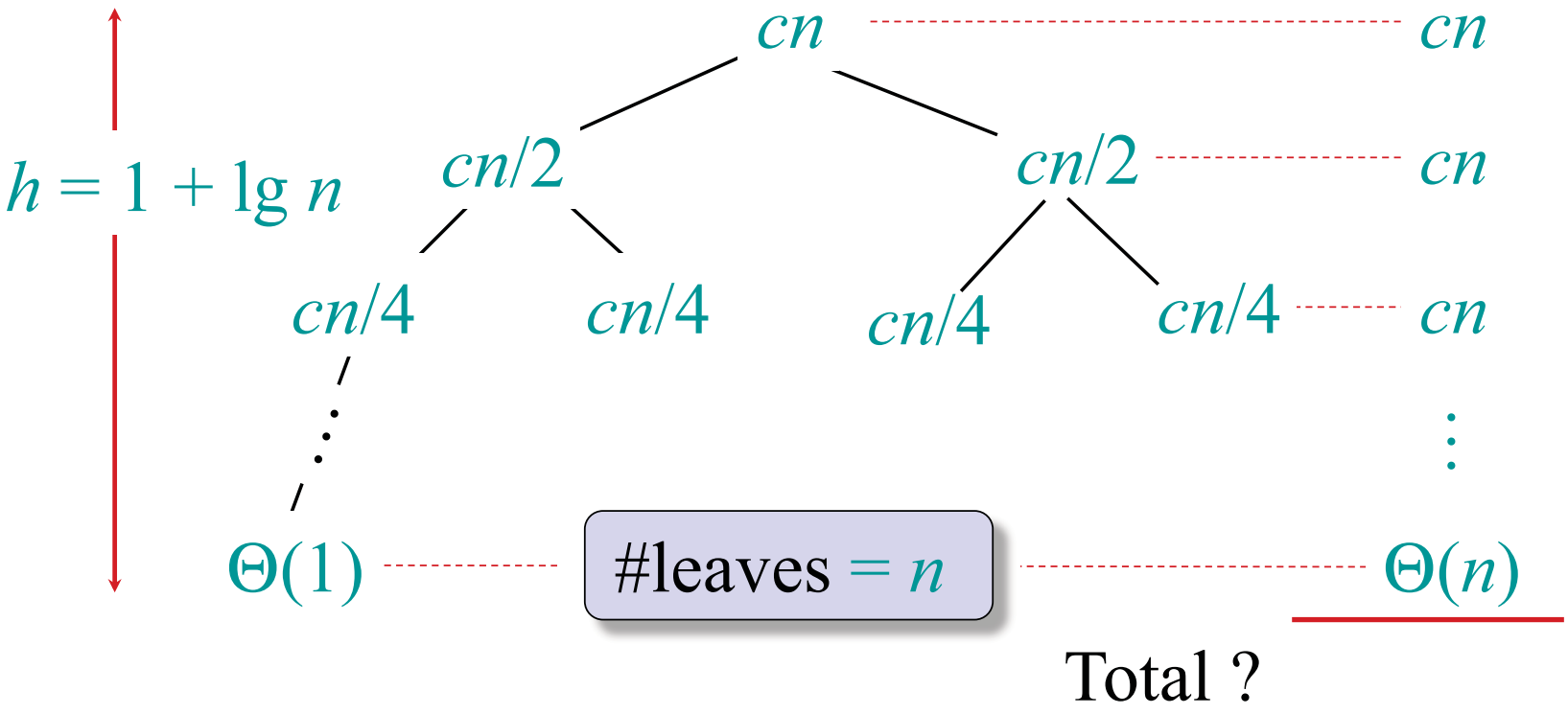
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

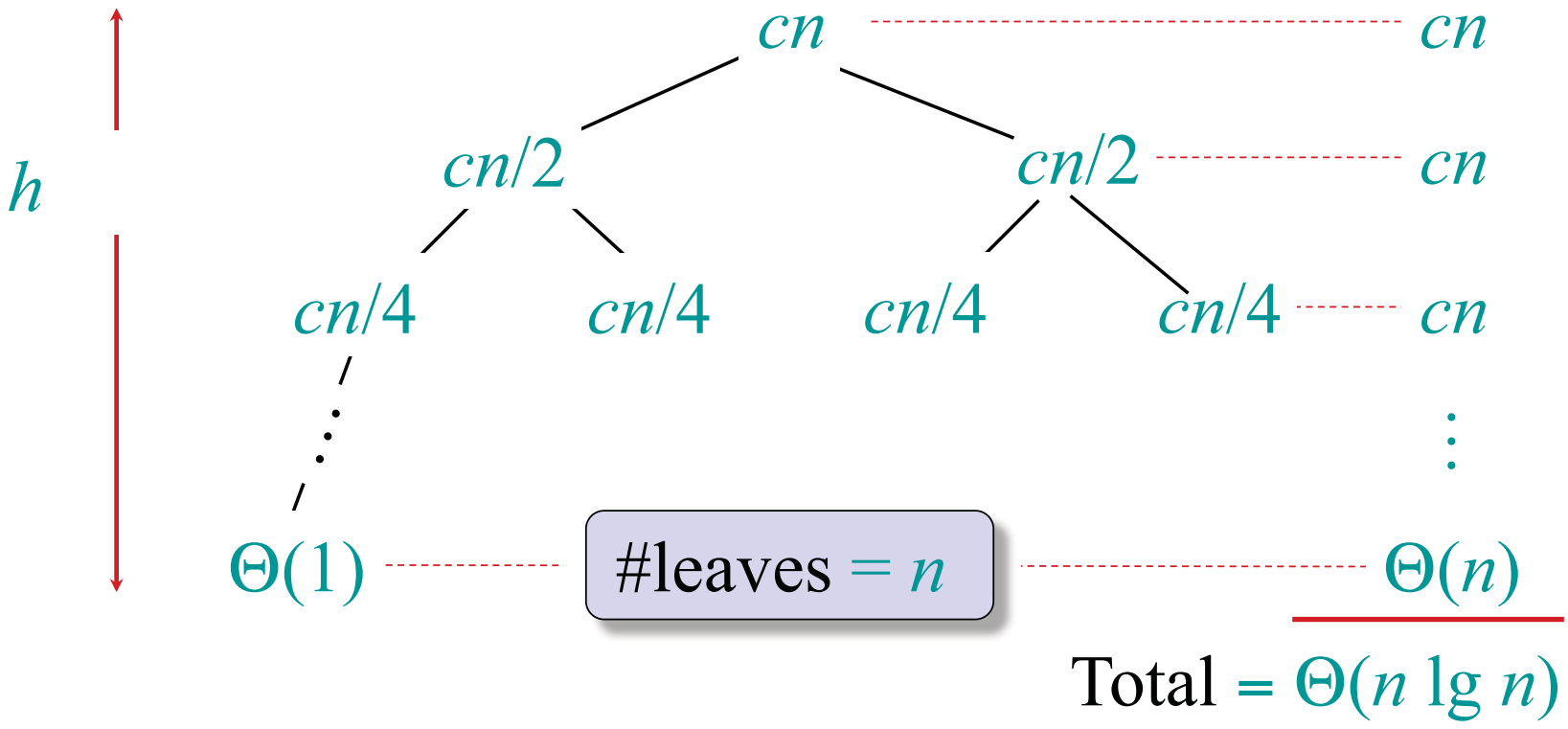
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Source: MIT OpenCourseWare

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

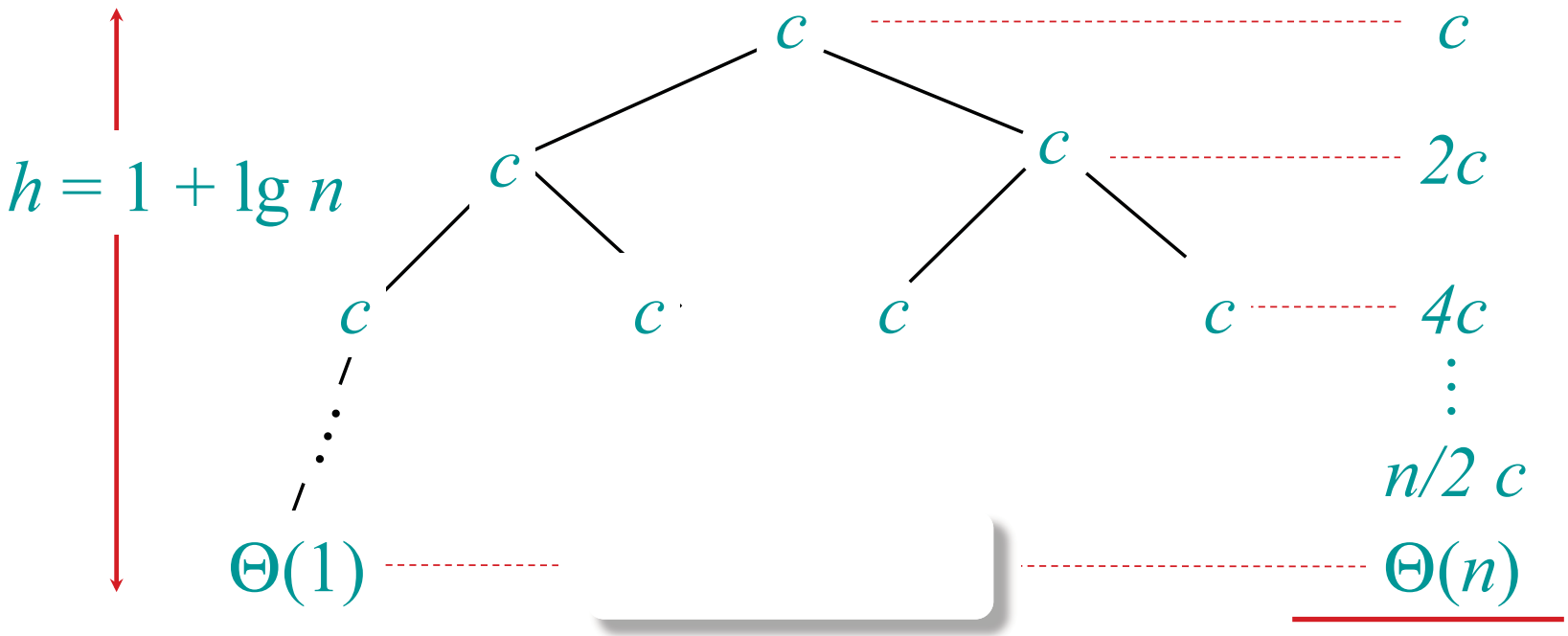


Equal amount of work done at each level

Source: MIT OpenCourseWare

Tree for different recurrence

Solve $T(n) = 2T(n/2) + c$, where $c > 0$ is constant.



Note that $1 + \frac{1}{2} + \frac{1}{4} + \ldots < 2$

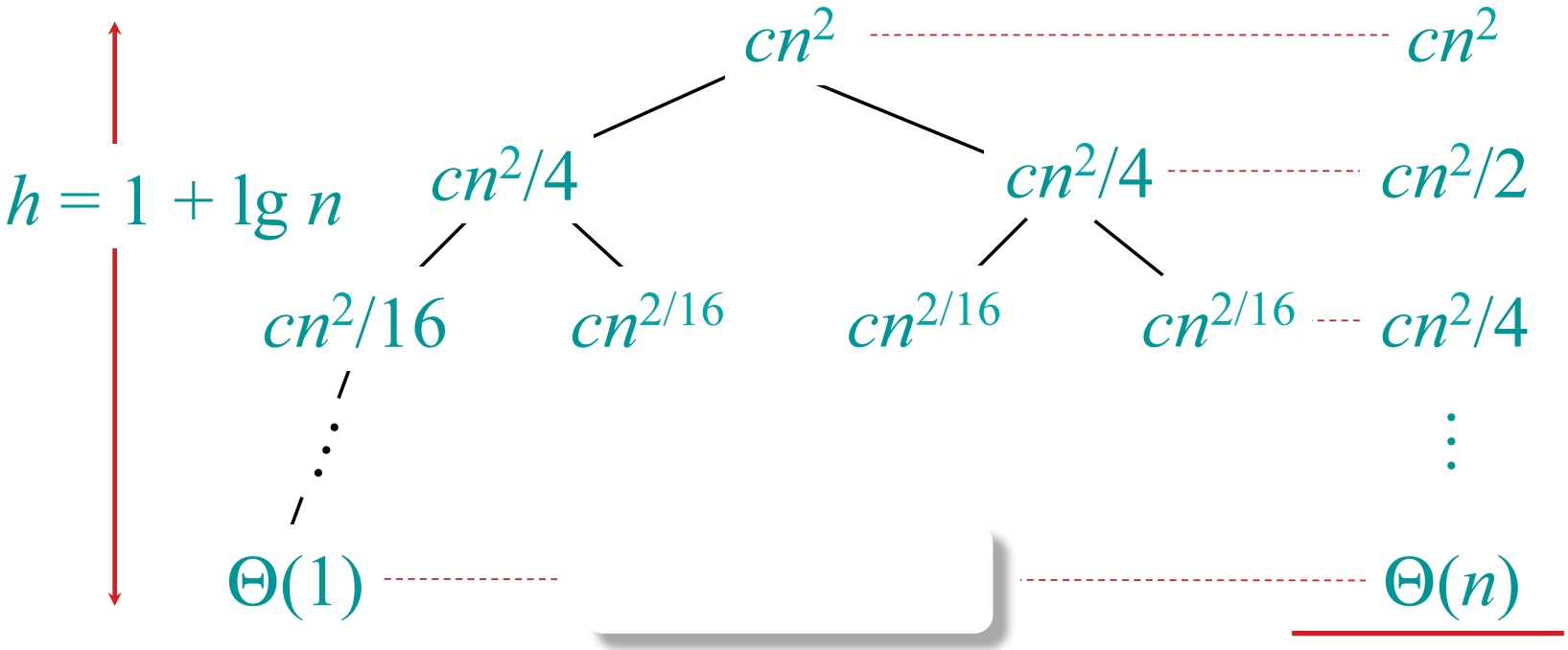
All the work done at the leaves

Total = $\Theta(n)$

Source: MIT OpenCourseWare

Tree for yet another recurrence

Solve $T(n) = 2T(n/2) + cn^2$, $c > 0$ is constant.



Note that $1 + ½ + ¼ + \ldots < 2$

All the work done at the root

Total = $\Theta(n^2)$

Source: MIT OpenCourseWare