Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 1

Prof. Charles E. Leiserson

Welcome to *Introduction to Algorithms*, Fall 2001

Handouts

1. Course Information
2. Calendar
3. Registration (MIT students only)
4. References
5. Objectives and Outcomes
6. Diagnostic Survey

Course information

1. Staff
2. Distance learning
3. Prerequisites
4. Lectures
5. Recitations
6. Handouts
7. Textbook (CLRS)
8. Website
9. Extra help
10. Registration (MIT only)
11. Problem sets
12. Describing algorithms
13. Grading policy
14. Collaboration policy

➢ Course information handout

Analysis of algorithms

The theoretical study of computer-program performance and resource usage.

What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

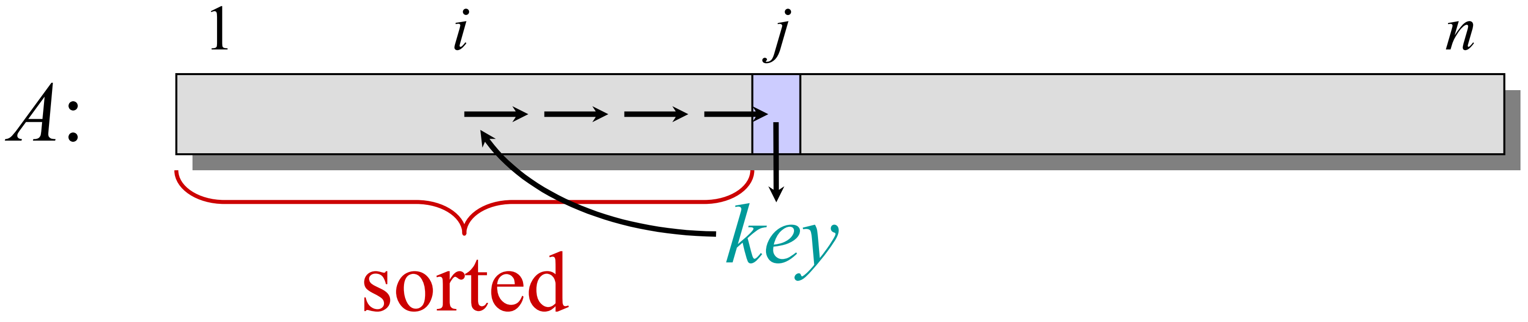
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

"pseudocode"

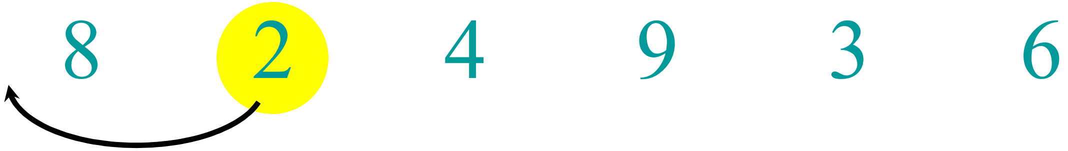
```
INSERTION-SORT $(A, n)$        ▷ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



Example of insertion sort

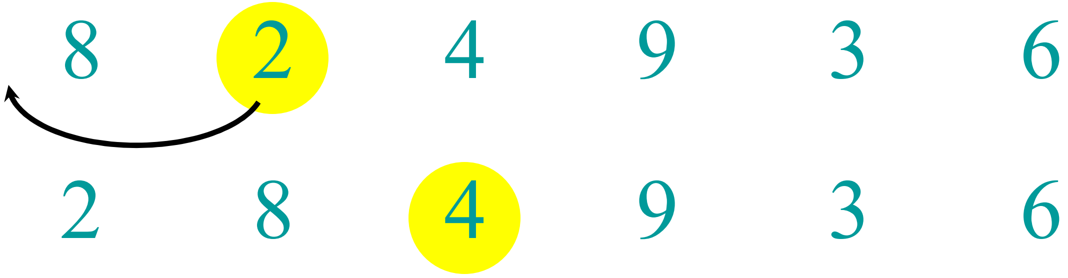
8 2 4 9 3 6

Example of insertion sort



8 2 4 9 3 6

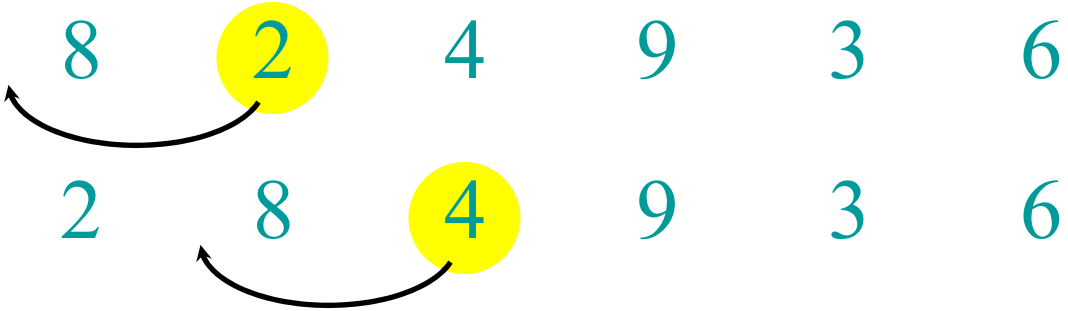
Example of insertion sort



8 2 4 9 3 6

2 8 4 9 3 6

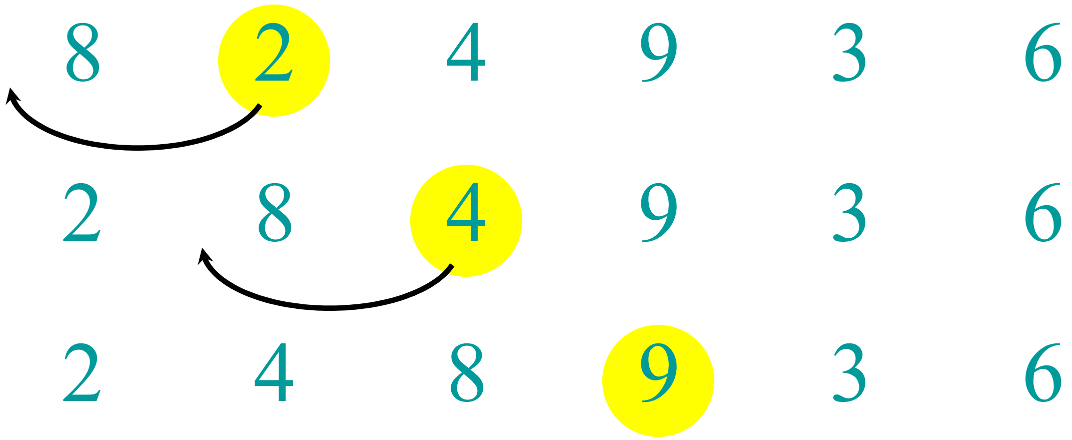
Example of insertion sort



8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

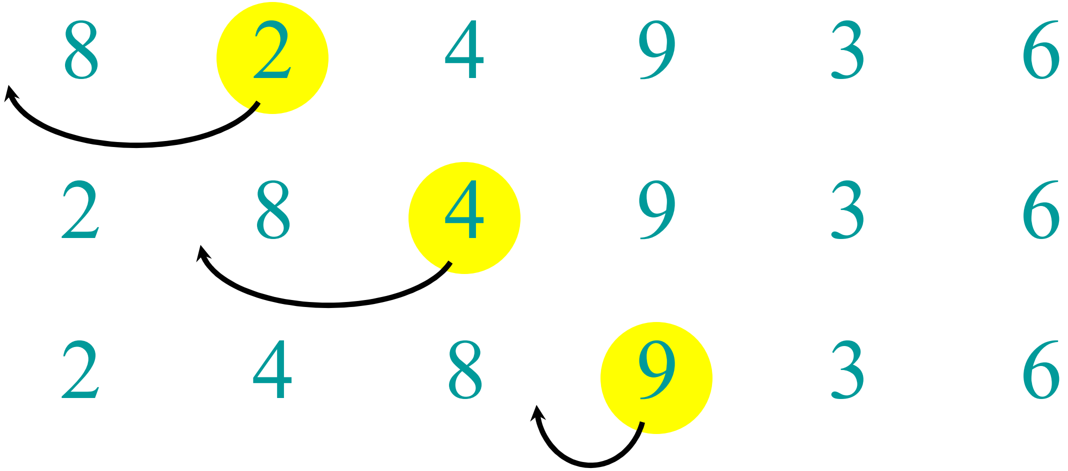


8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

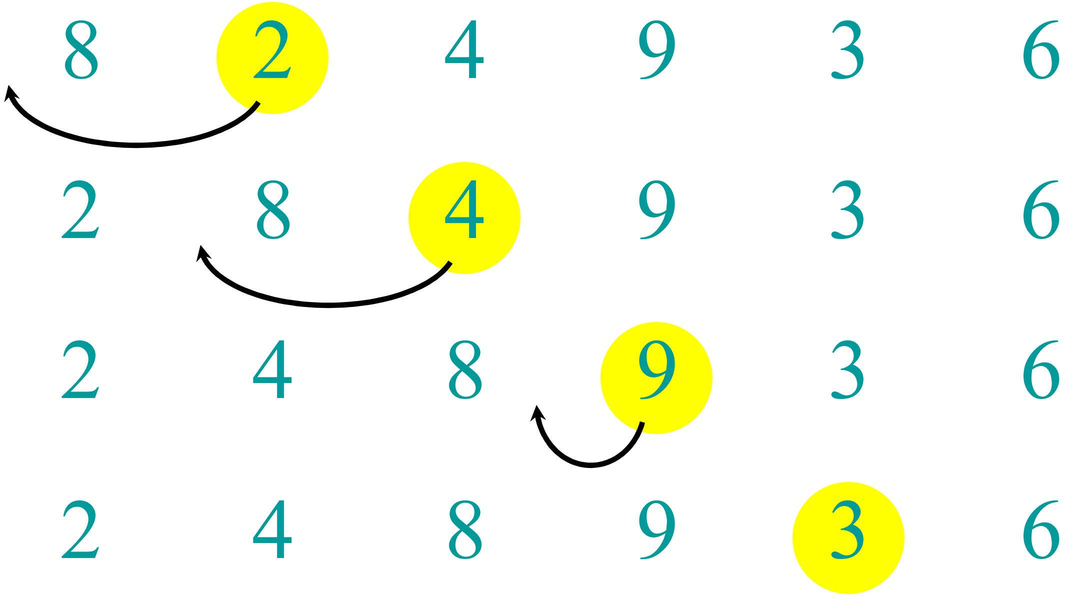


8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort



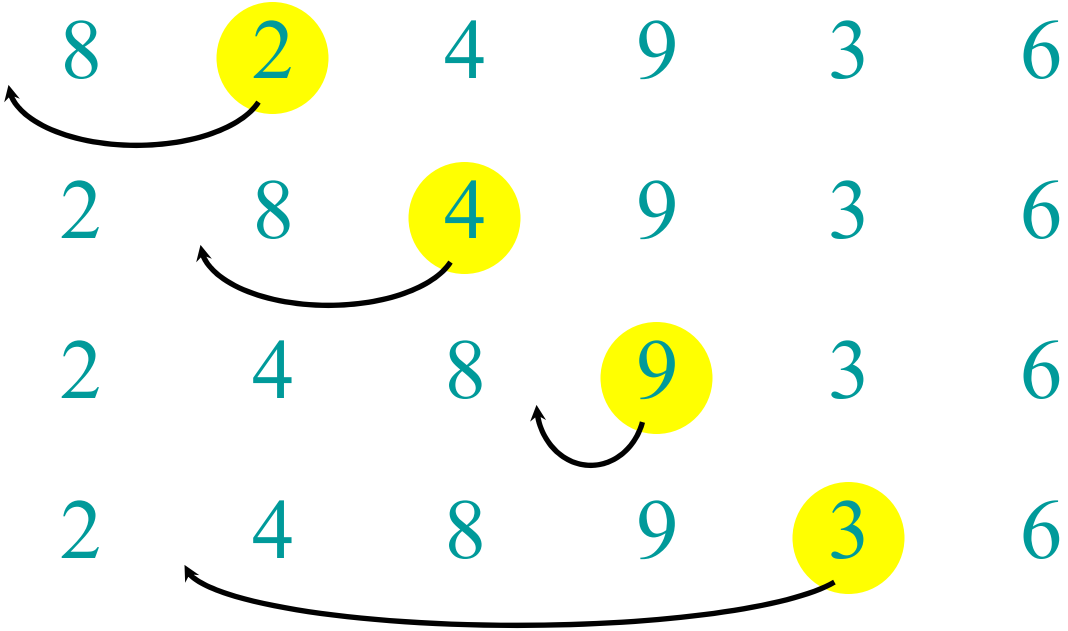
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort



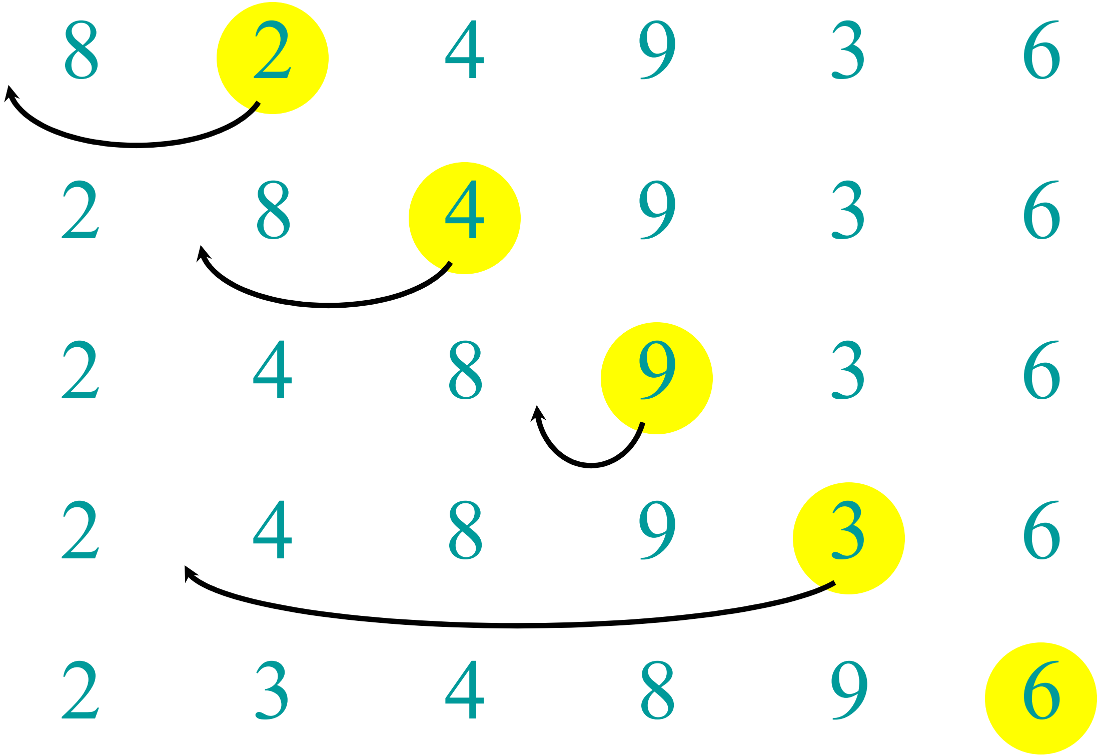
8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort



8 2 4 9 3 6

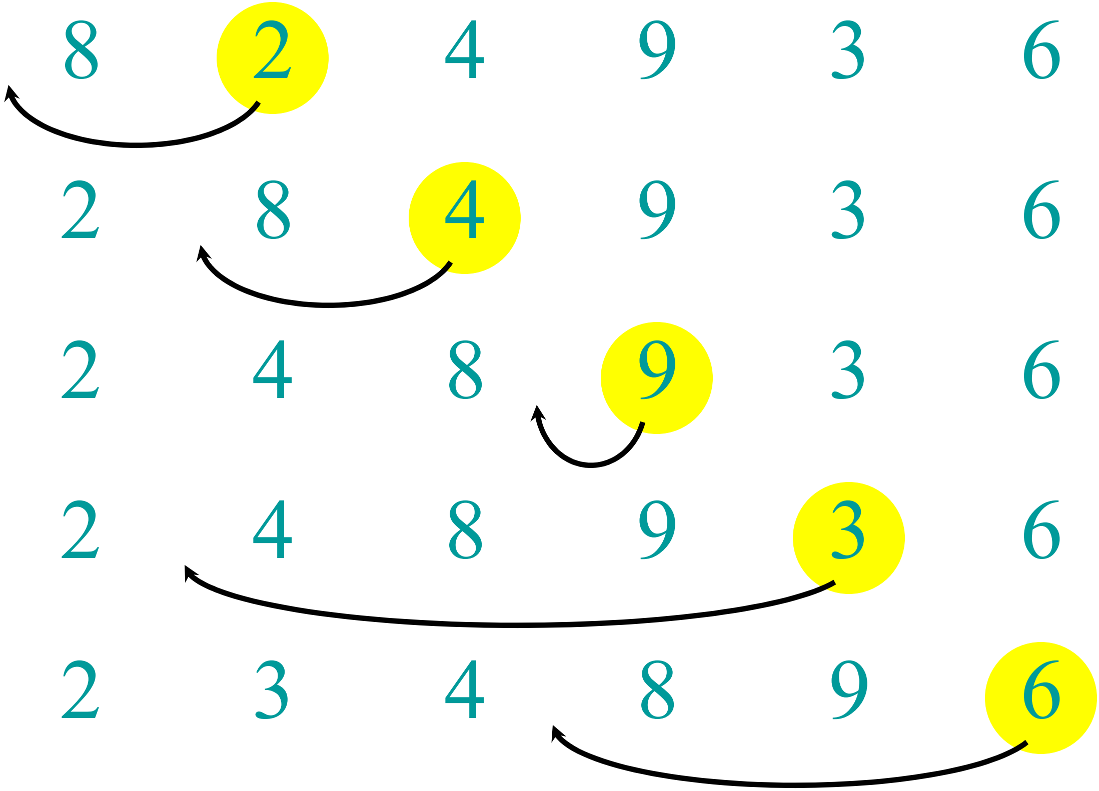
2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

Example of insertion sort



8 2 4 9 3 6

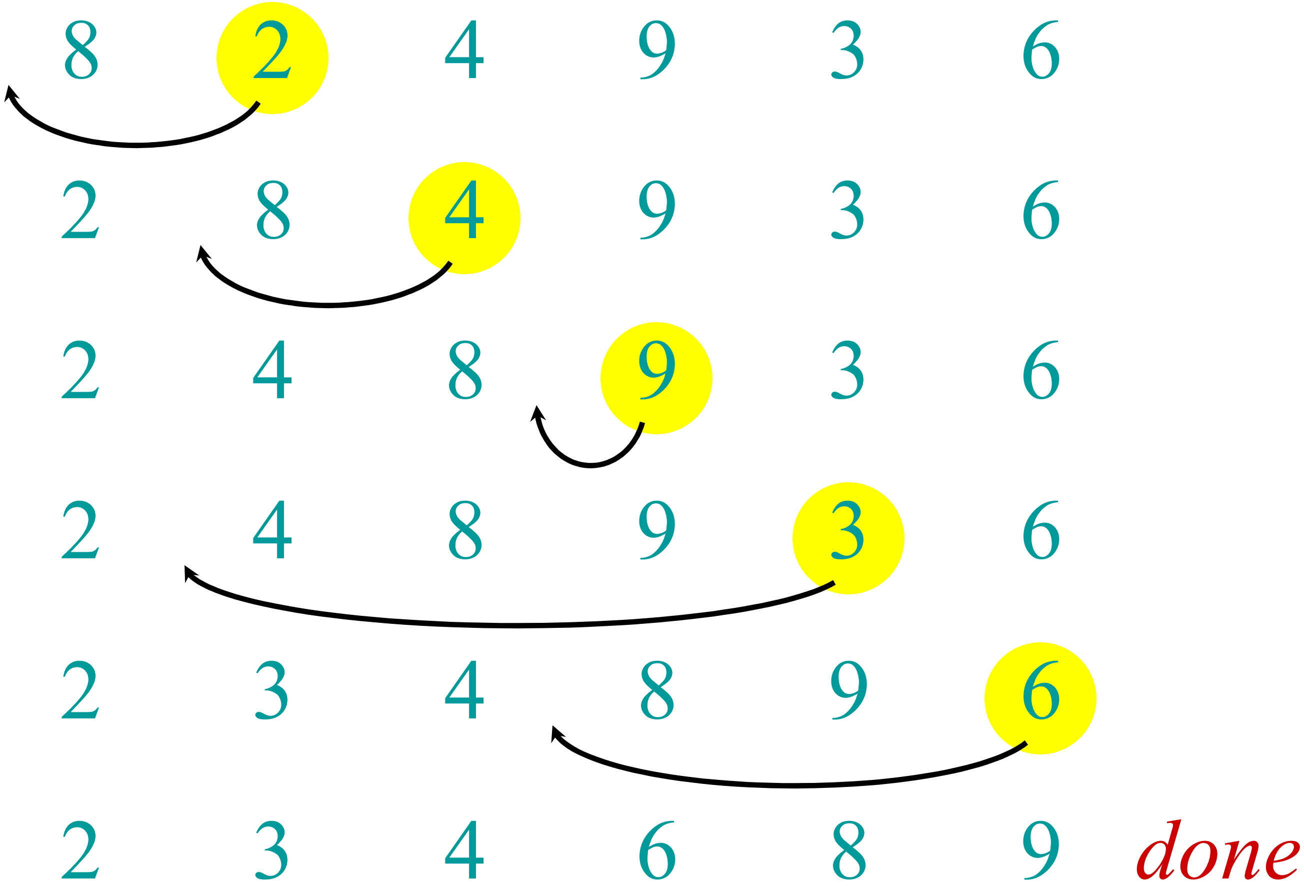
2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

Example of insertion sort



8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 *done*

Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.

Machine-independent time

What is insertion sort's worst-case time?

- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

BIG IDEA:

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

"Asymptotic Analysis"

Θ-notation

Math:

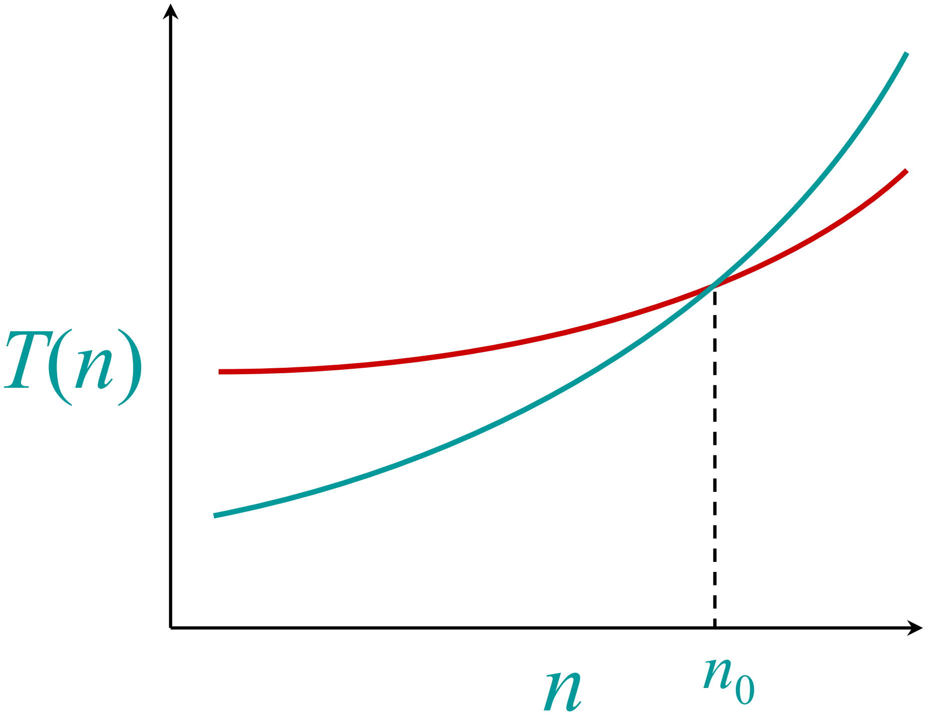
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ $\}$

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta\left(n^2\right)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

Merge sort

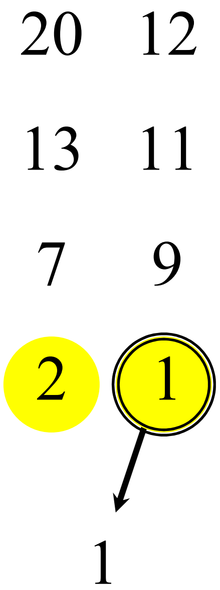
```
MERGE-SORT $A[1 \,..\, n]$
    1. If $n = 1$, done.
    2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$
       and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$ .
    3. "Merge" the 2 sorted lists.
```

Key subroutine: **MERGE**

Merging two sorted arrays

20	12
13	11
7	9
2	1

Merging two sorted arrays



20 12

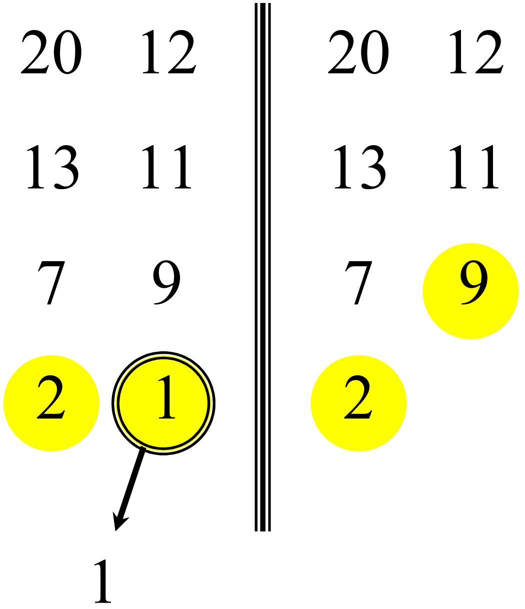
13 11

7 9

2 1

1

Merging two sorted arrays



20 12 ‖ 20 12

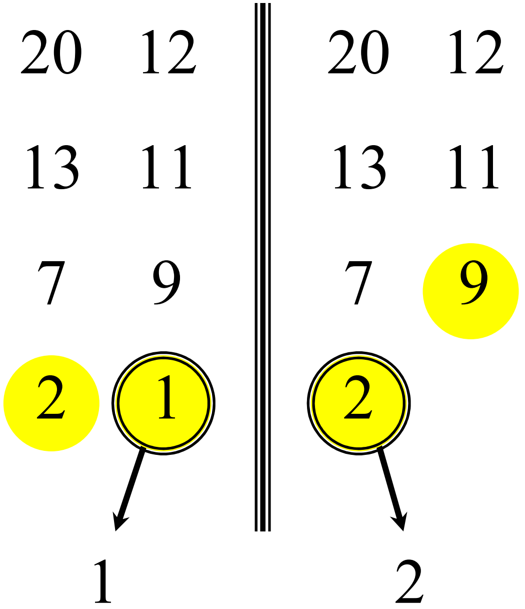
13 11 ‖ 13 11

7 9 ‖ 7 9

2 1 ‖ 2

1

Merging two sorted arrays



20 12 20 12

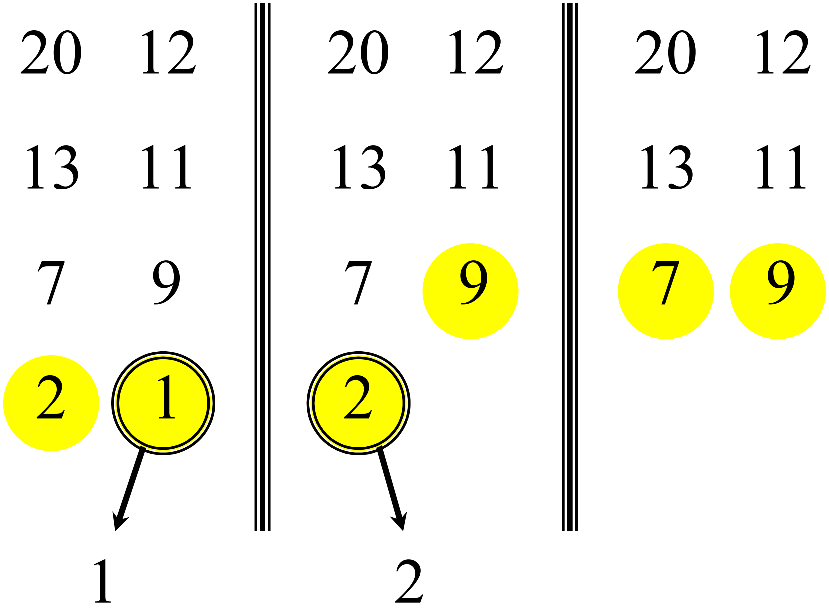
13 11 13 11

7 9 7 9

2 1 2

1 2

Merging two sorted arrays



20 12

13 11

7 9

2 1

1

20 12

13 11

7 9

2

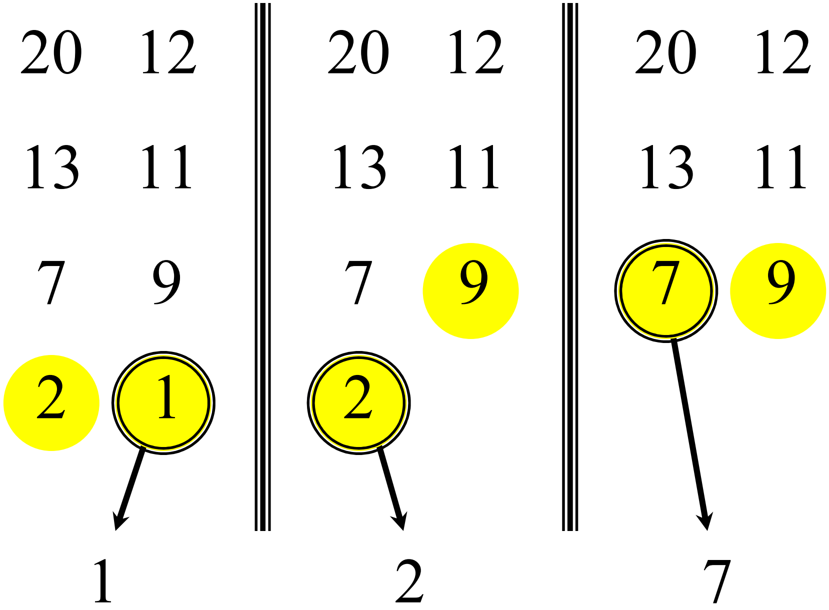
2

20 12

13 11

7 9

Merging two sorted arrays



20 12

13 11

7 9

2 1

1

20 12

13 11

7 9

2

2

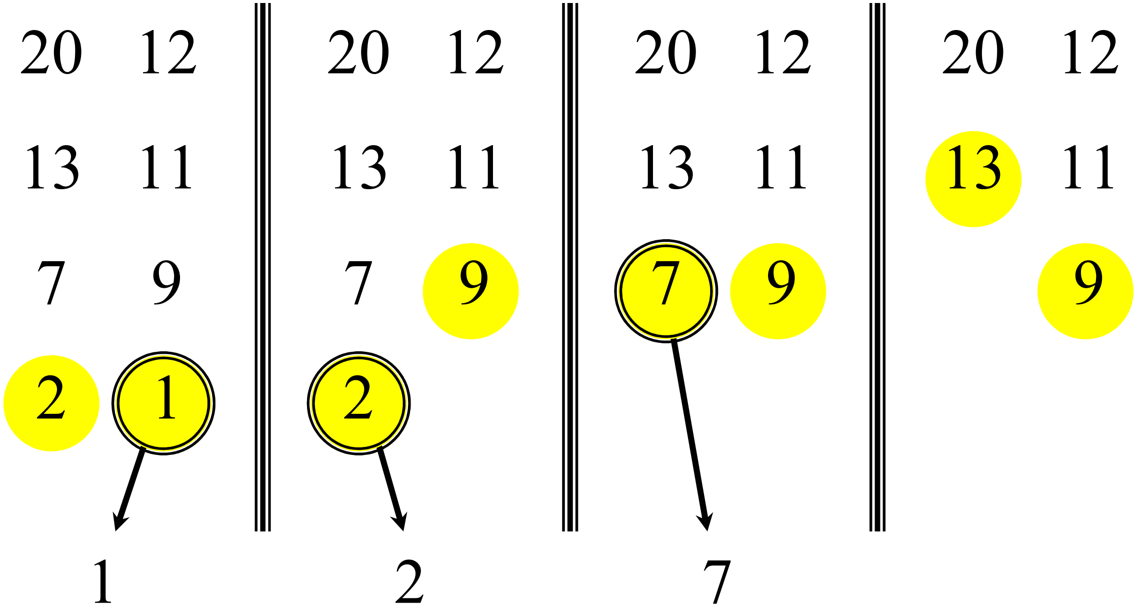
20 12

13 11

7 9

7

Merging two sorted arrays



20 12

13 11

7 9

2 1

1

20 12

13 11

7 9

2

2

20 12

13 11

7 9

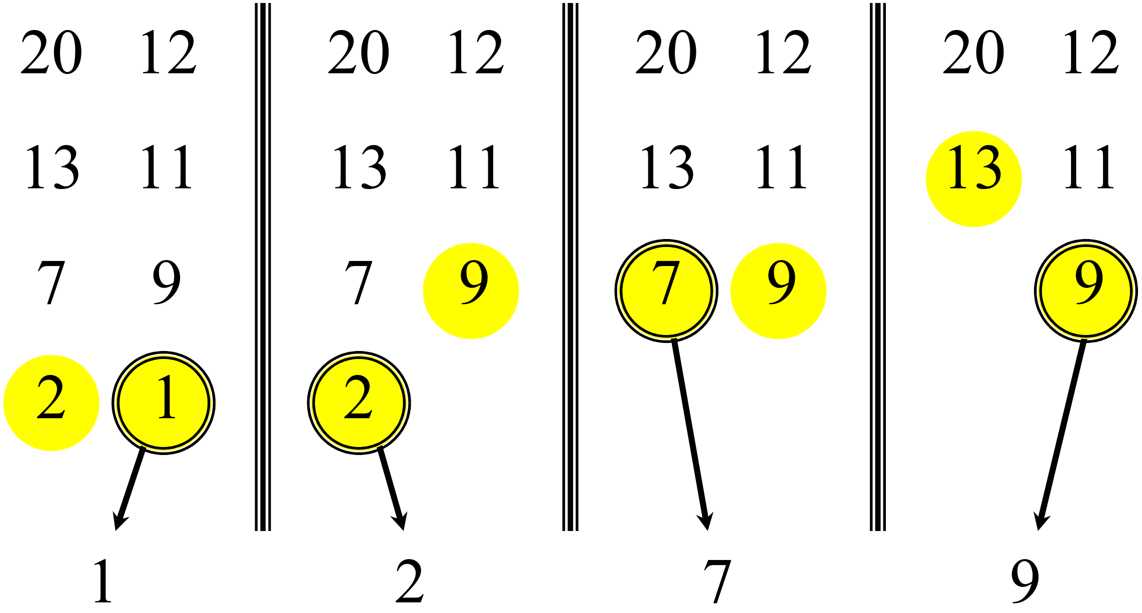
7

20 12

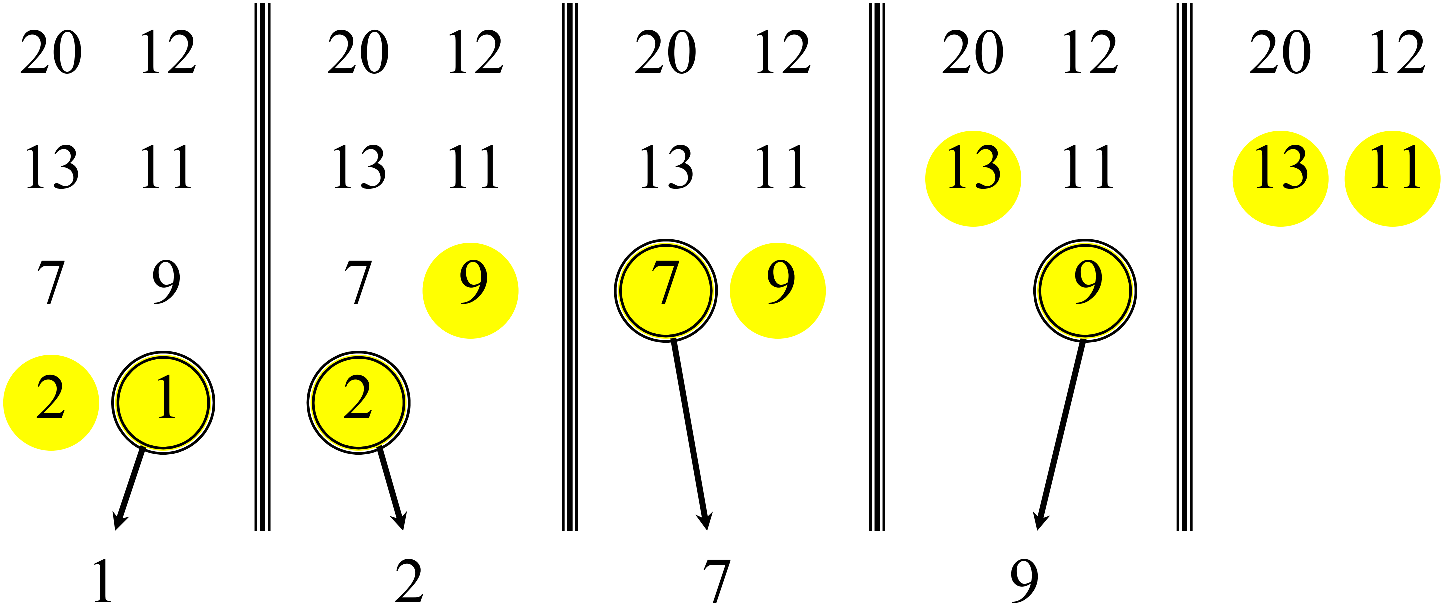
13 11

9

Merging two sorted arrays



Merging two sorted arrays



20 12
13 11
7 9
2 1
1

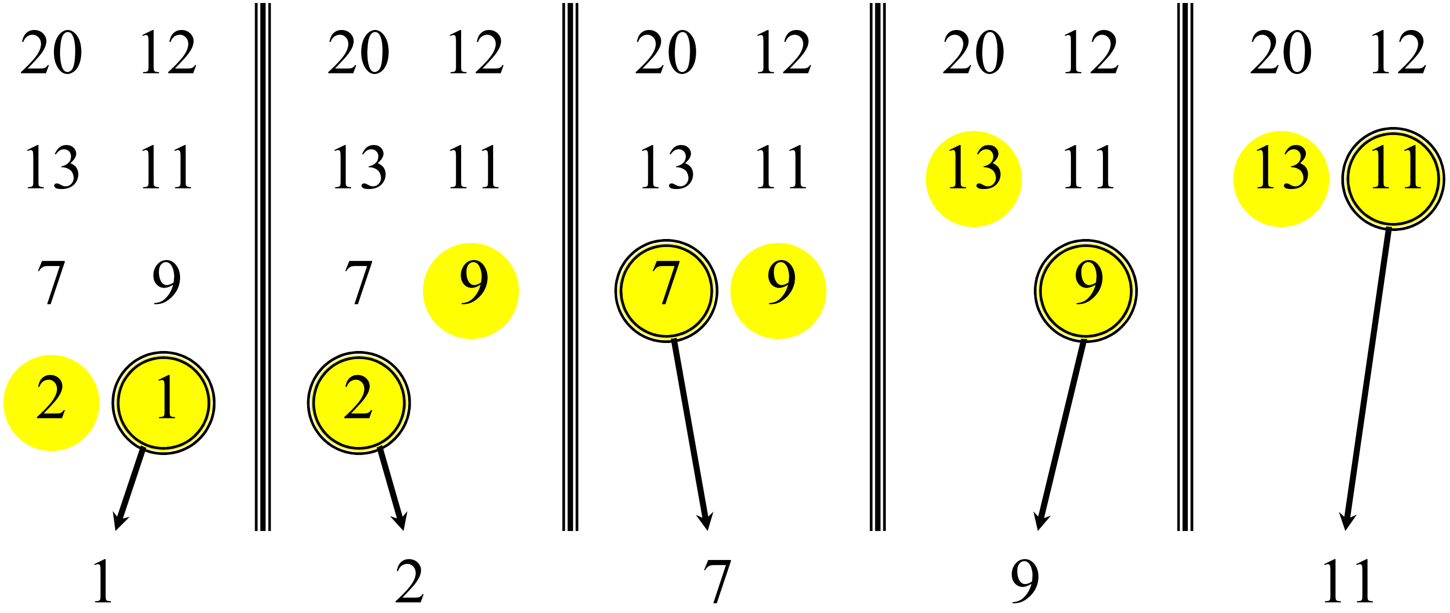
20 12
13 11
7 9
2
2

20 12
13 11
7 9
7

20 12
13 11
9
9

20 12
13 11

Merging two sorted arrays



20 12
13 11
7 9
2 1
1

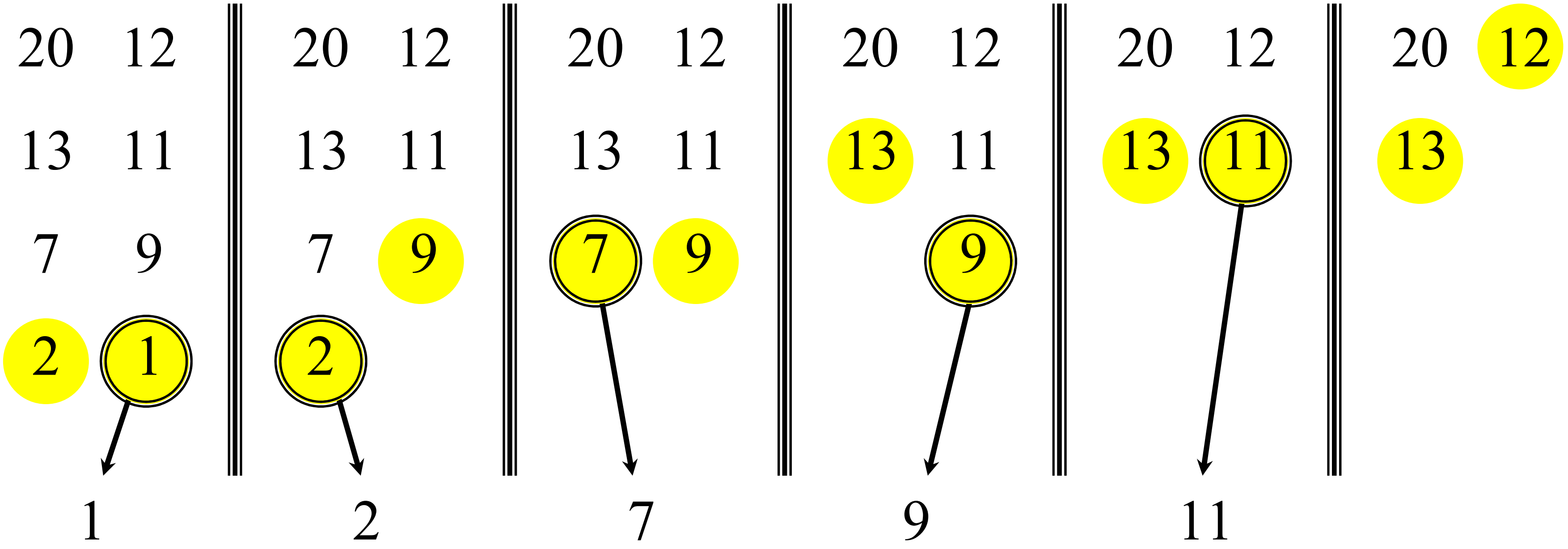
20 12
13 11
7 9
2
2

20 12
13 11
7 9
7

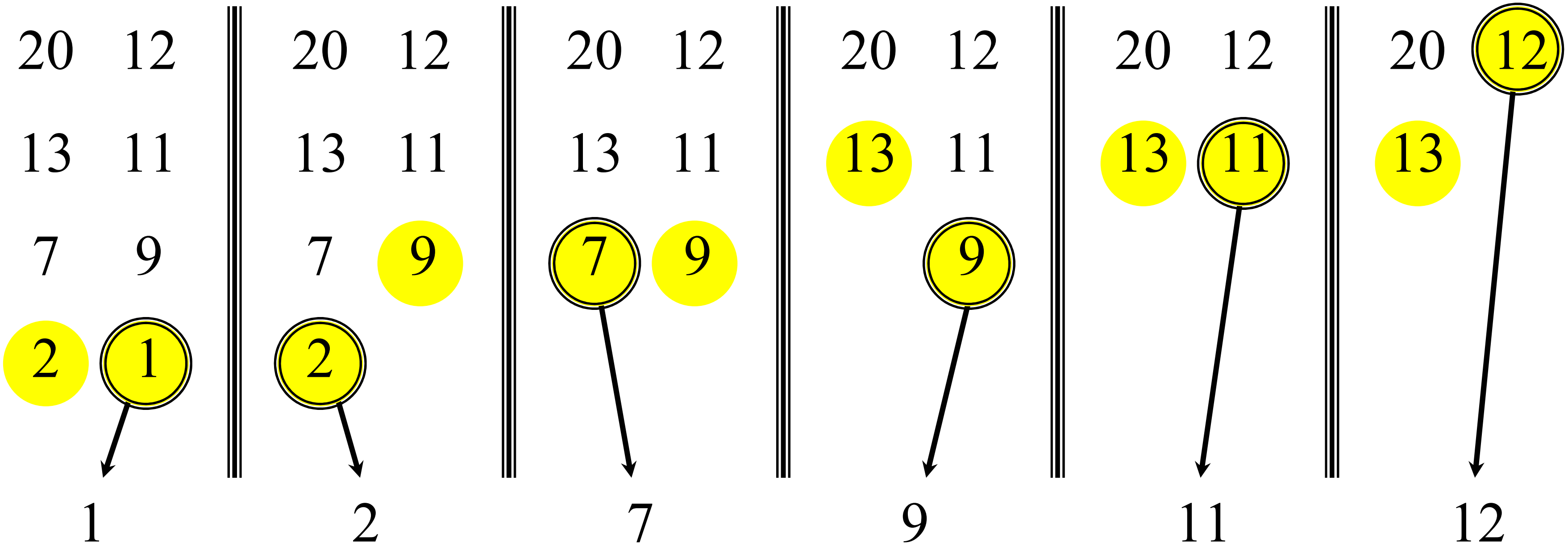
20 12
13 11
9
9

20 12
13 11
11

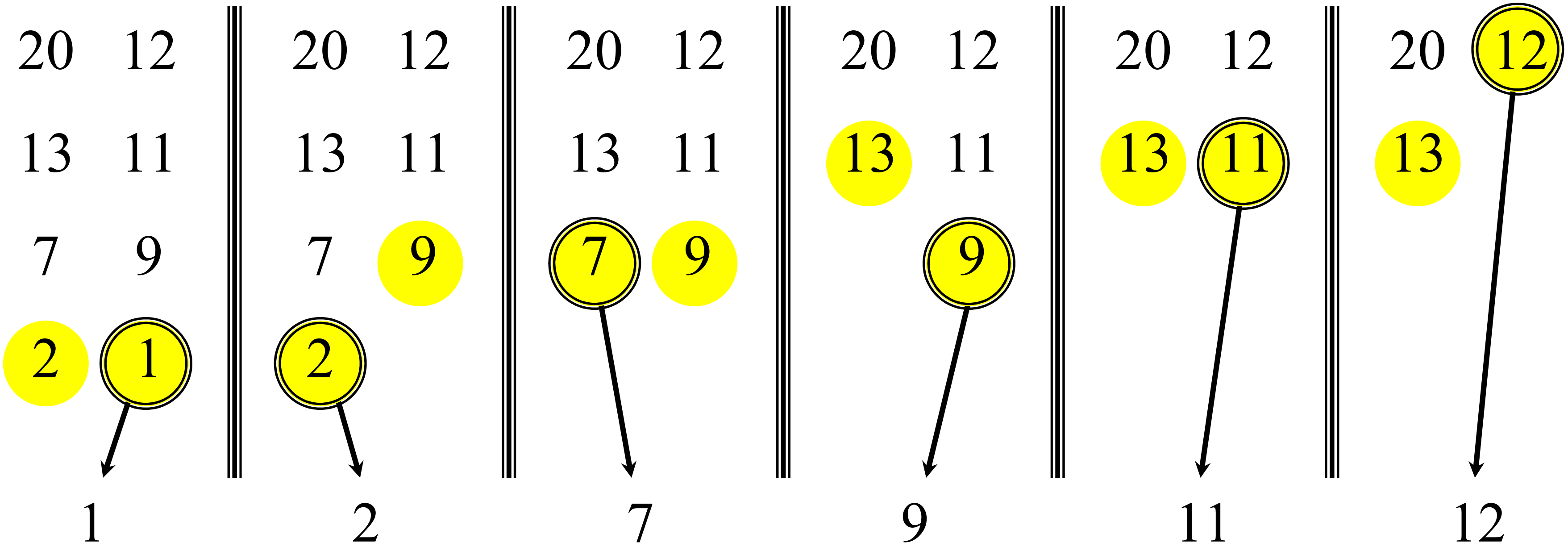
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



20 12
13 11
7 9
2 1
1

20 12
13 11
7 9
2
2

20 12
13 11
7 9
7

20 12
13 11
9
9

20 12
13 11
11

20 12
13
12

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

Analyzing merge sort

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

Abuse

Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

Recurrence for merge sort

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

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

Recursion tree

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

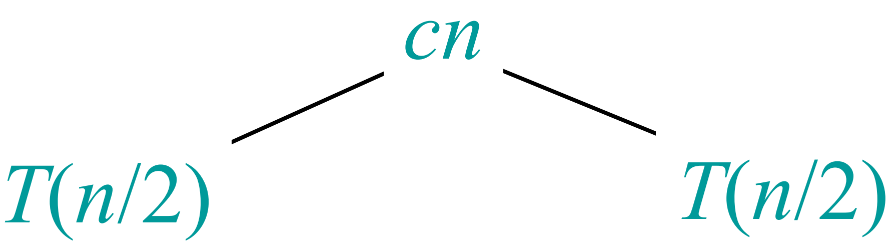
Recursion tree

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

$$T(n)$$

Recursion tree

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

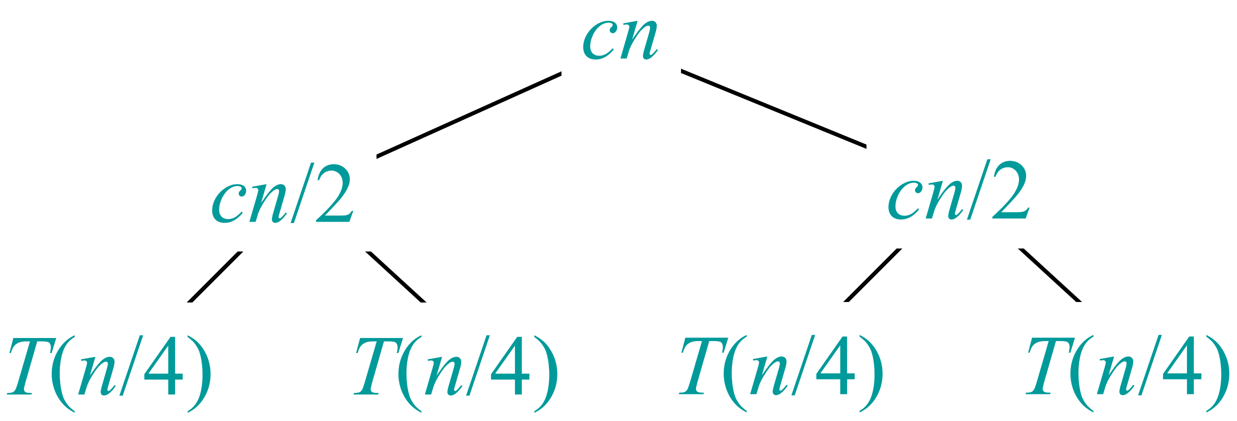


cn

$T(n/2)$ $T(n/2)$

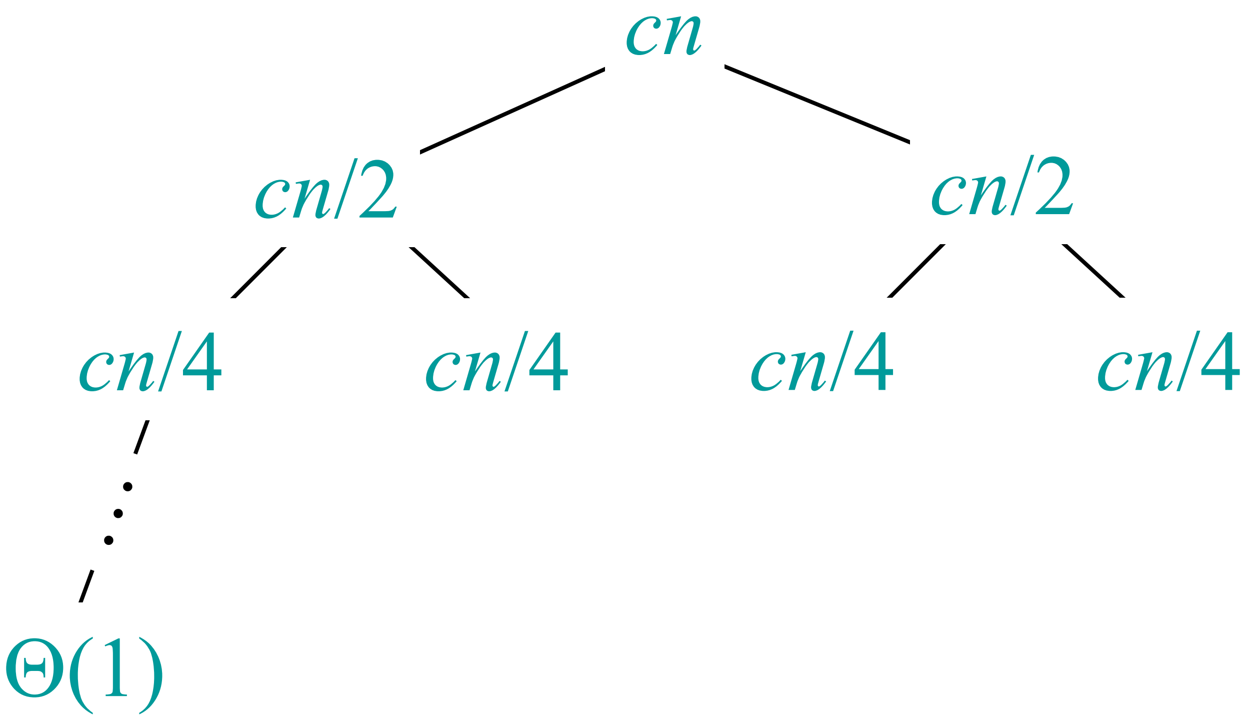
Recursion tree

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



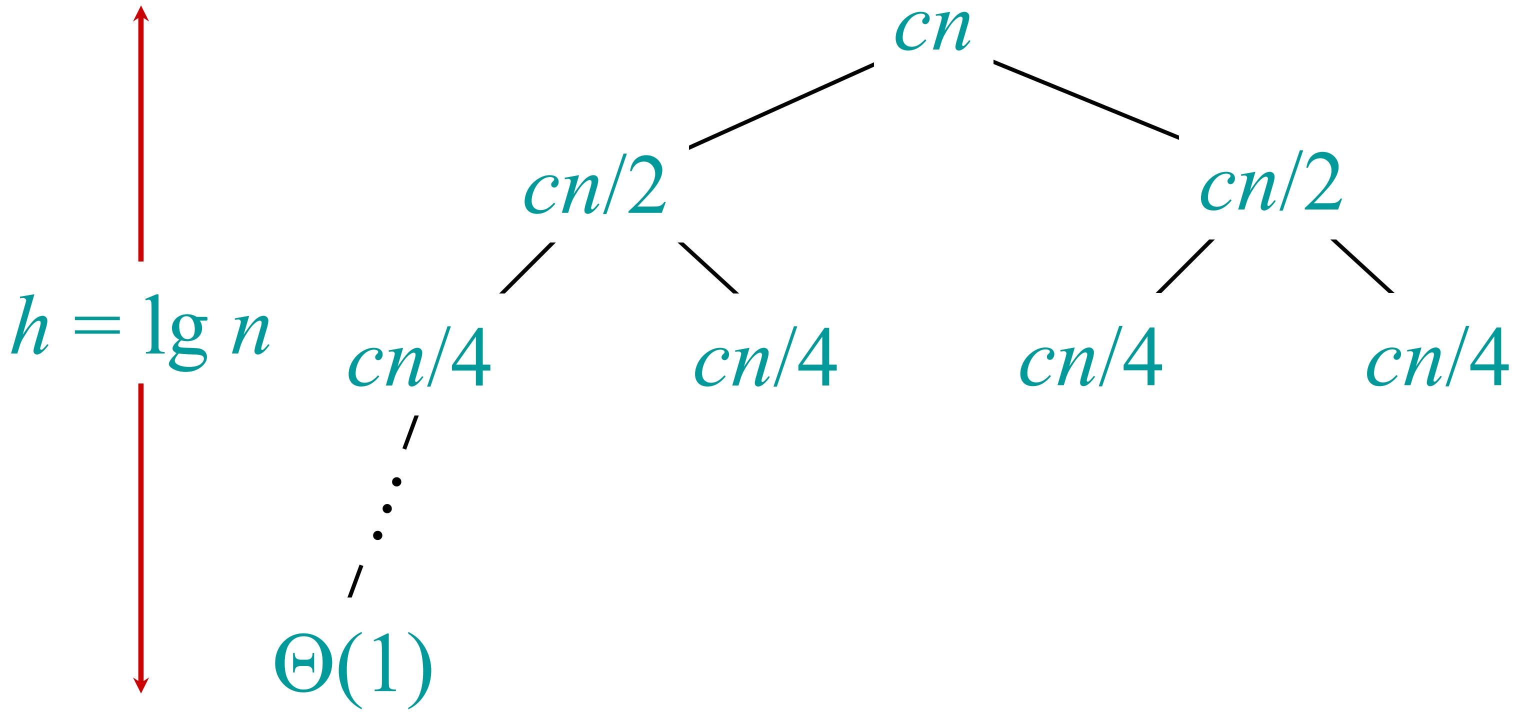
Recursion tree

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



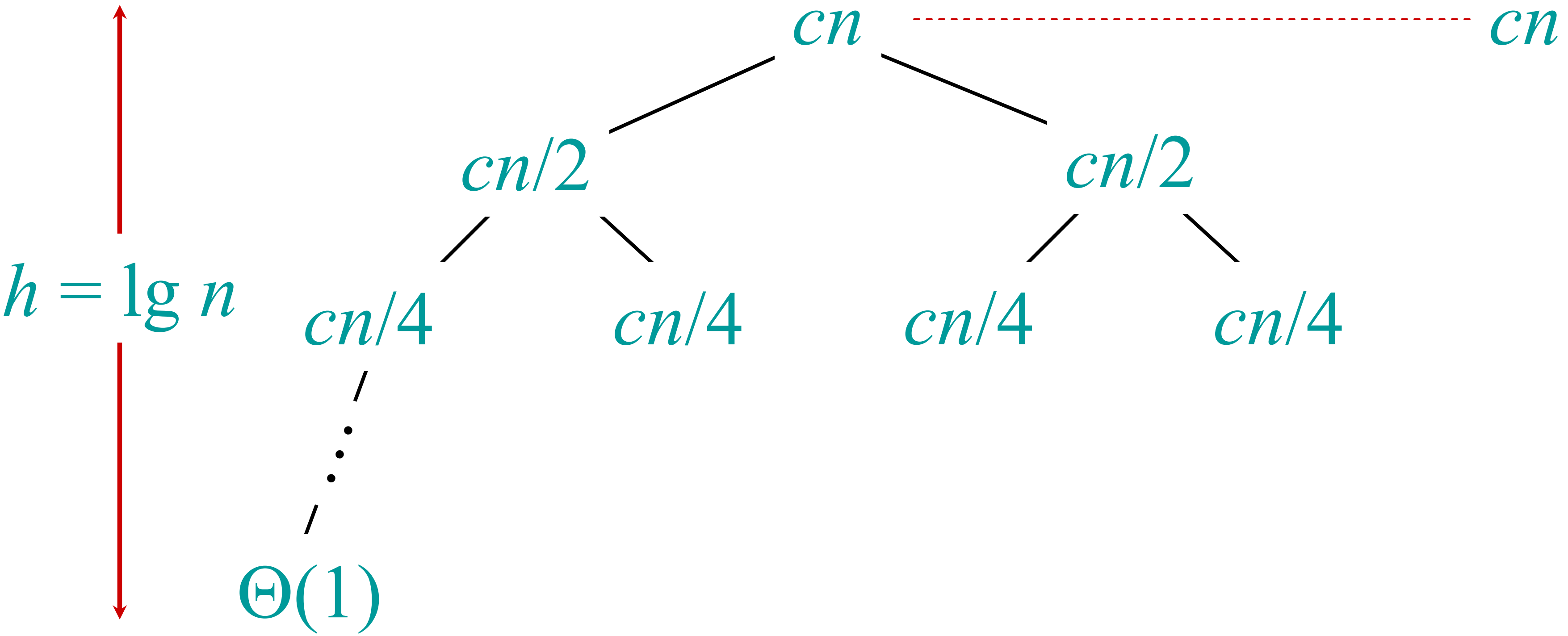
Recursion tree

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



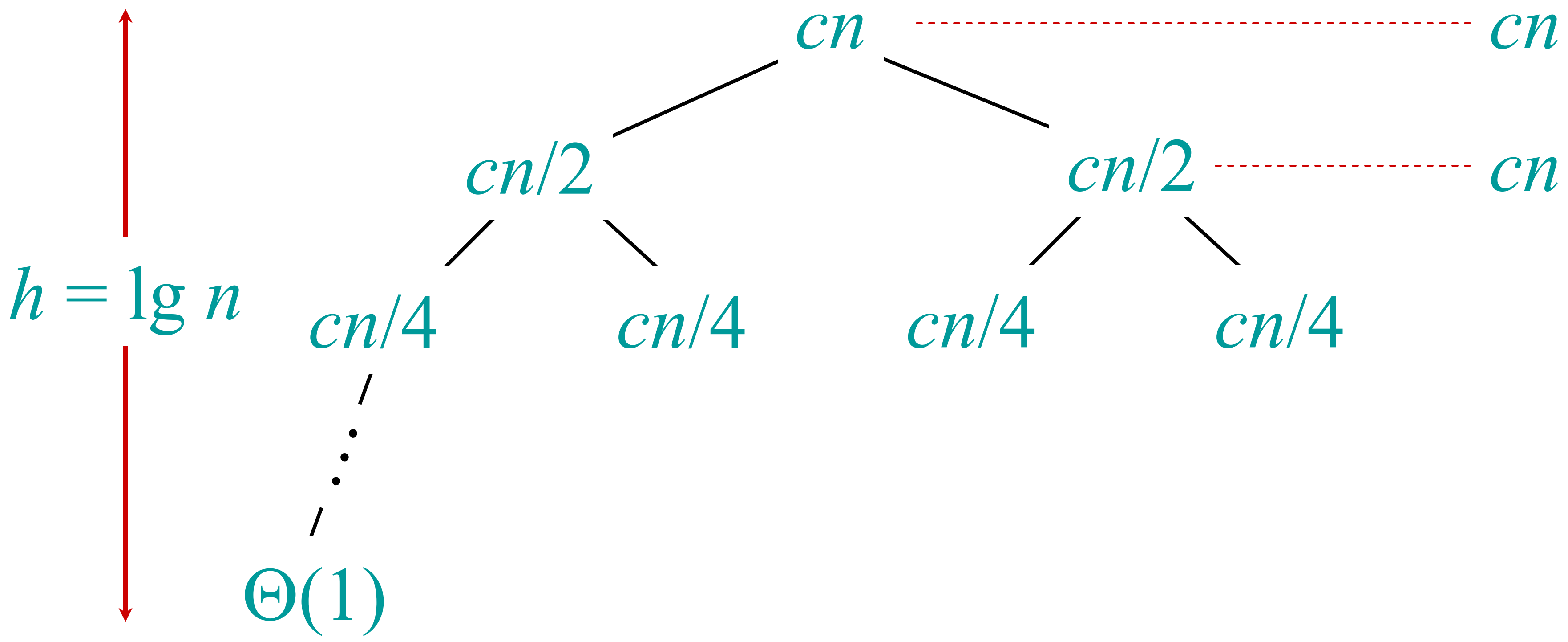
Recursion tree

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



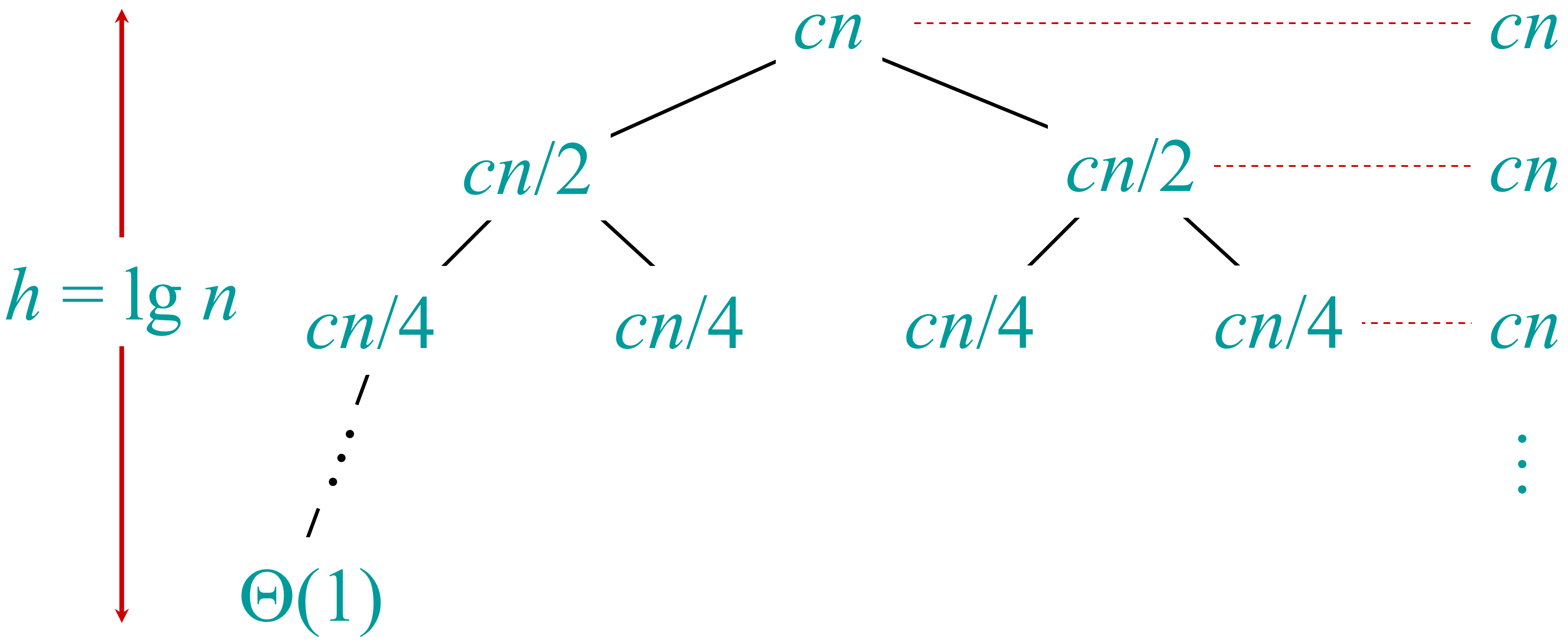
Recursion tree

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



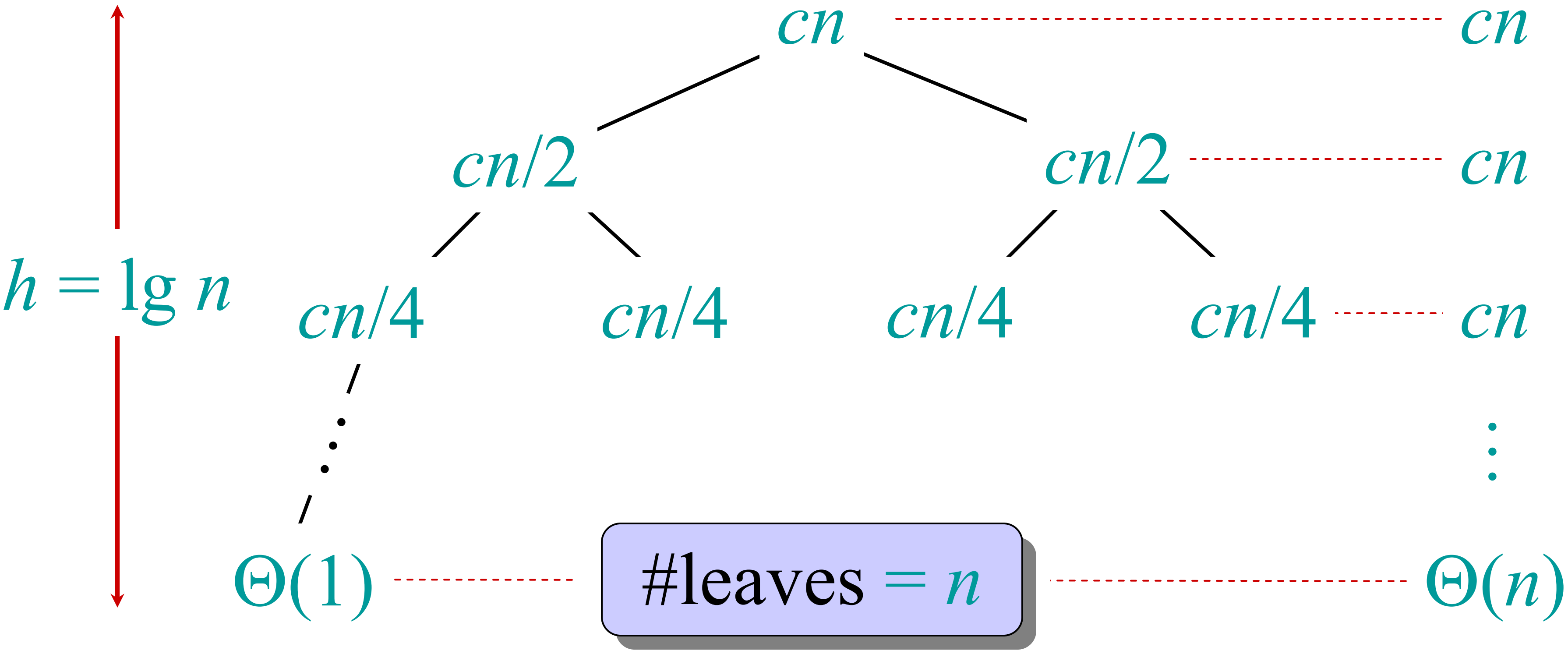
Recursion tree

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



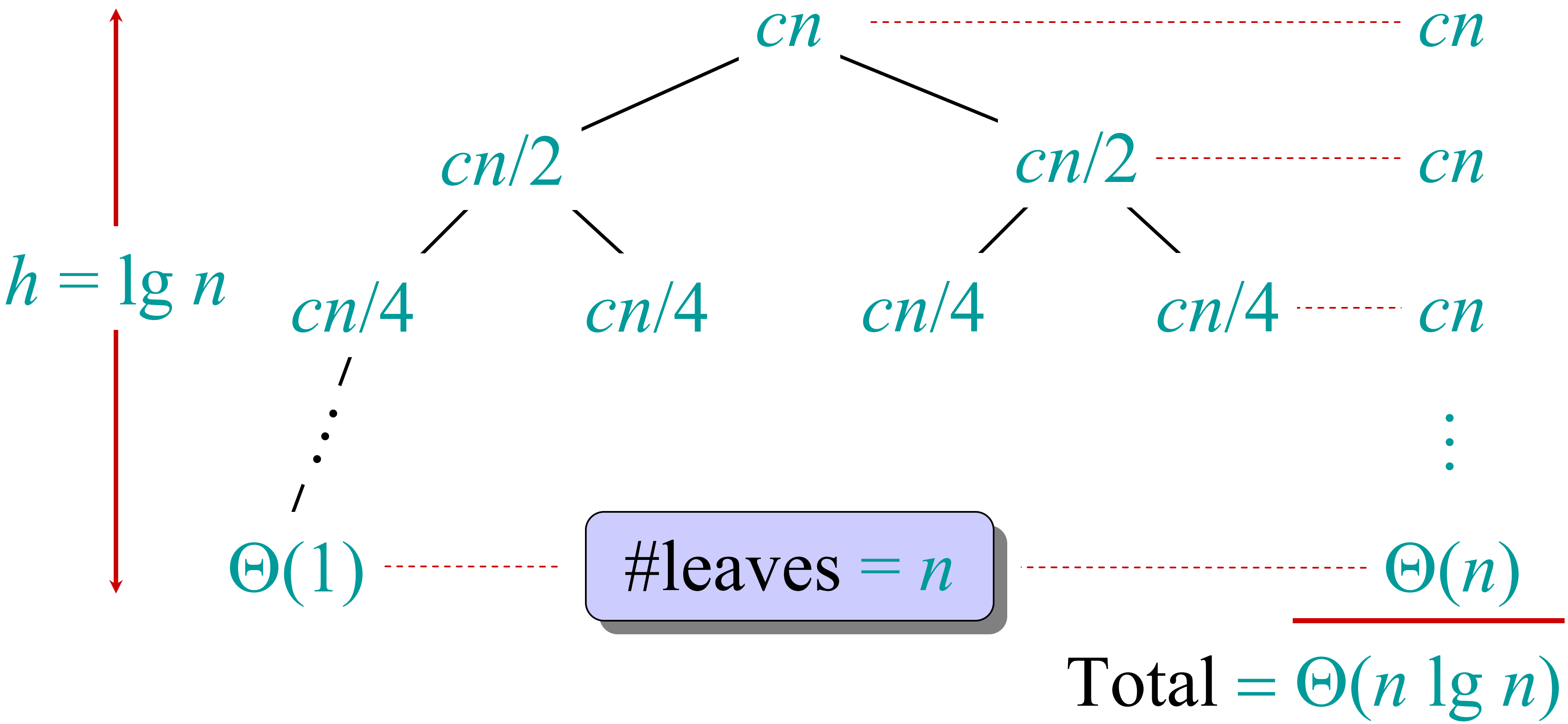
Recursion tree

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



Recursion tree

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



Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 2

Prof. Erik Demaine

Solving recurrences

- The analysis of merge sort from ***Lecture 1*** required us to solve a recurrence.
- Recurrences are like solving integrals, differential equations, etc.
    - Learn a few tricks.
- ***Lecture 3***: Applications of recurrences.

Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

Example: $T(n) = 4T(n/2) + n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \leq ck^3$ for $k < n$.
- Prove $T(n) \leq cn^3$ by induction.

Example of substitution

$$\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^3 + n \\
&= (c/2)n^3 + n \\
&= cn^3 - ((c/2)n^3 - n) \longleftarrow \textit{desired} - \textit{residual} \\
&\le cn^3 \longleftarrow \textit{desired}
\end{aligned}$$

whenever $(c/2)n^3 - n \ge 0$, for example, if $c \ge 2$ and $n \ge 1$.

residual

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!

A tighter upper bound?

We shall prove that $T(n) = O(n^2)$.

Assume that $T(k) \le ck^2$ for $k < n$:

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4cn^2 + n \\ &= \cancel{O(n)} \end{aligned}$$

Wrong! We must prove the I.H.

$$\begin{aligned} &= cn^2 - (-n) \quad [\ \text{desired} - \text{residual}\ ] \\ &\le cn^2 \end{aligned}$$

for ***no*** choice of $c > 0$. Lose!

A tighter upper bound!

Idea: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\le c_1 n^2 - c_2 n \quad \text{if } c_2 > 1. \end{aligned}$$

Pick c_1 big enough to handle the initial conditions.

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

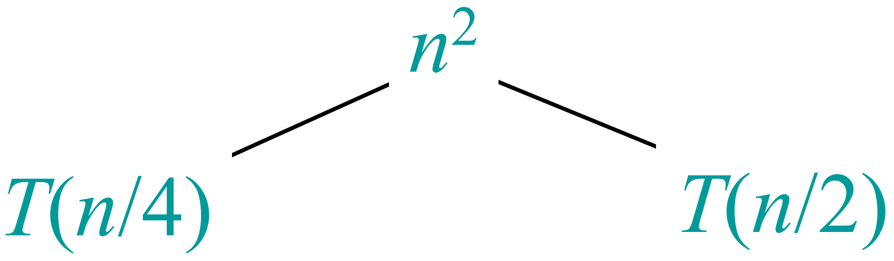
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$T(n)$$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

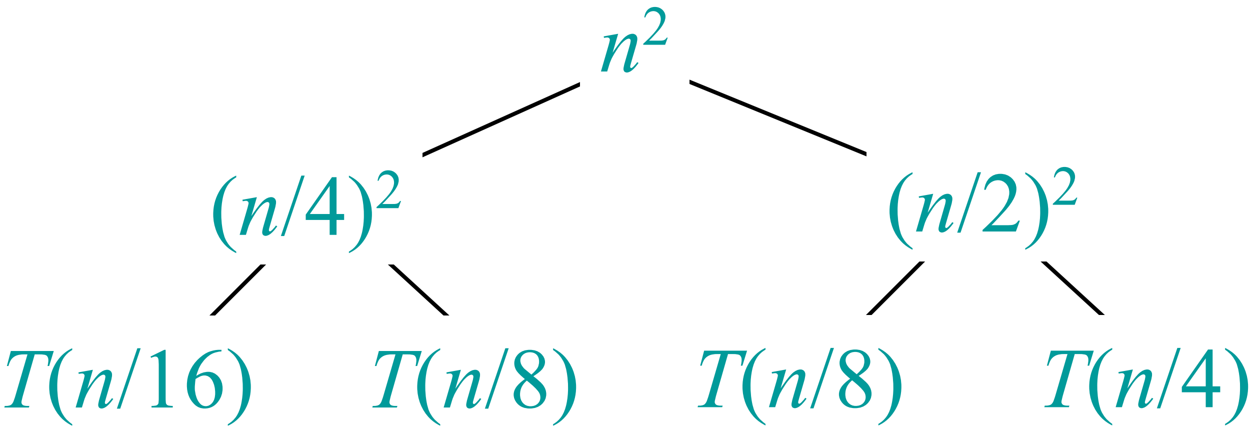


n^2

$T(n/4)$ $T(n/2)$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



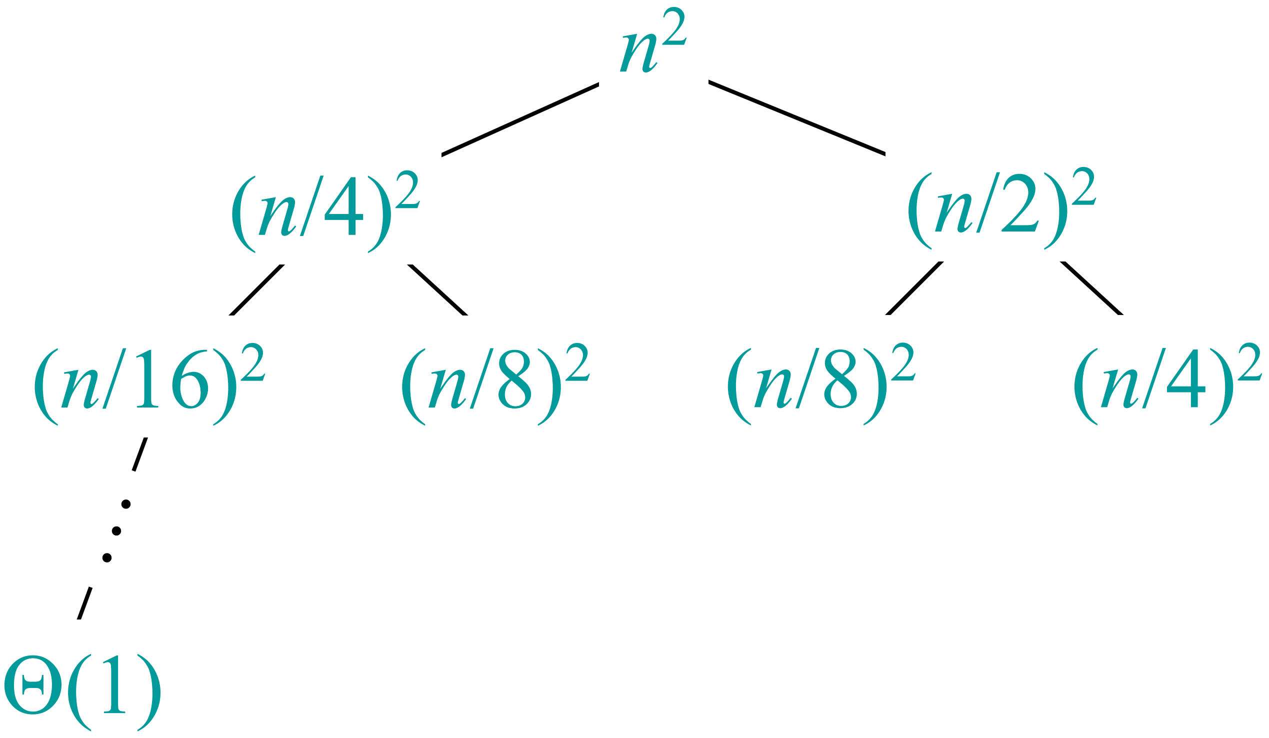
n^2

$(n/4)^2$ $(n/2)^2$

$T(n/16)$ $T(n/8)$ $T(n/8)$ $T(n/4)$

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



n^2

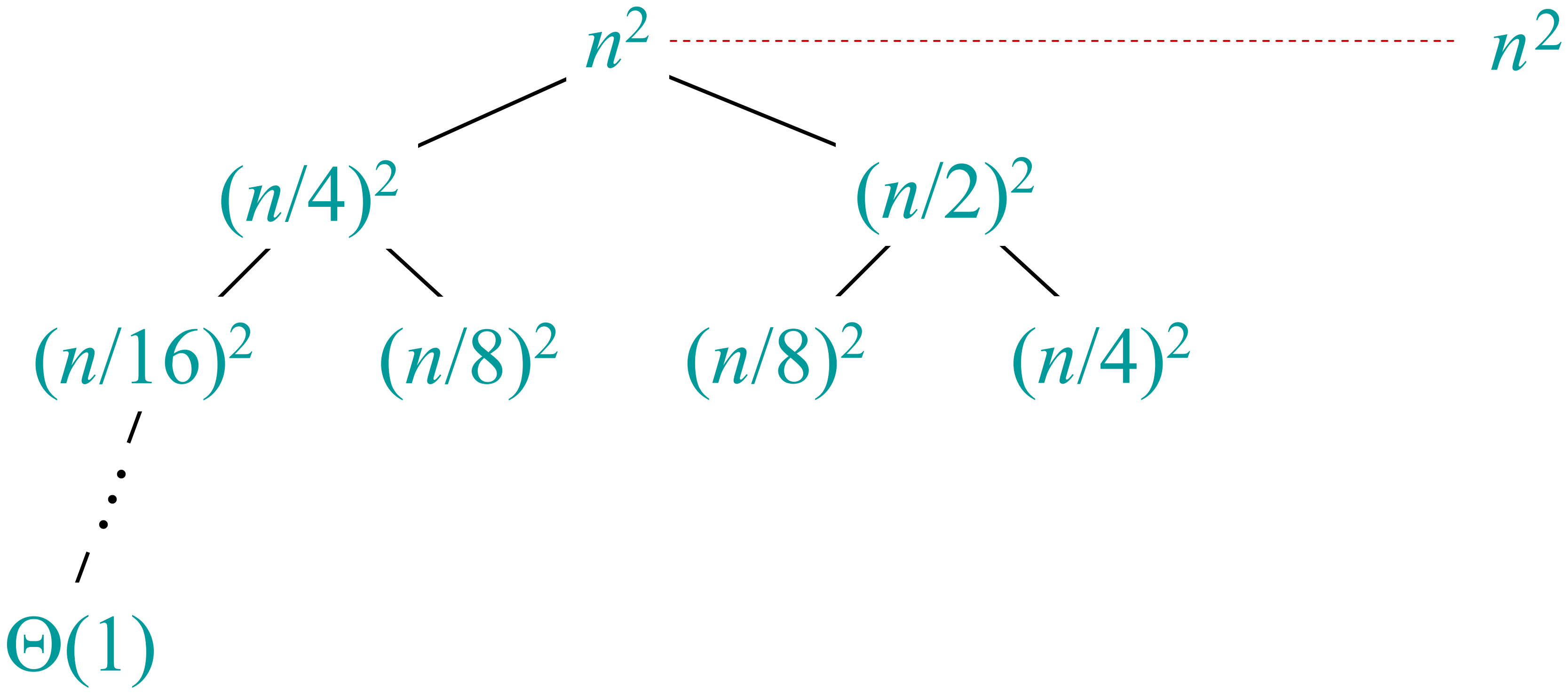
$(n/4)^2$ $(n/2)^2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$

$\Theta(1)$

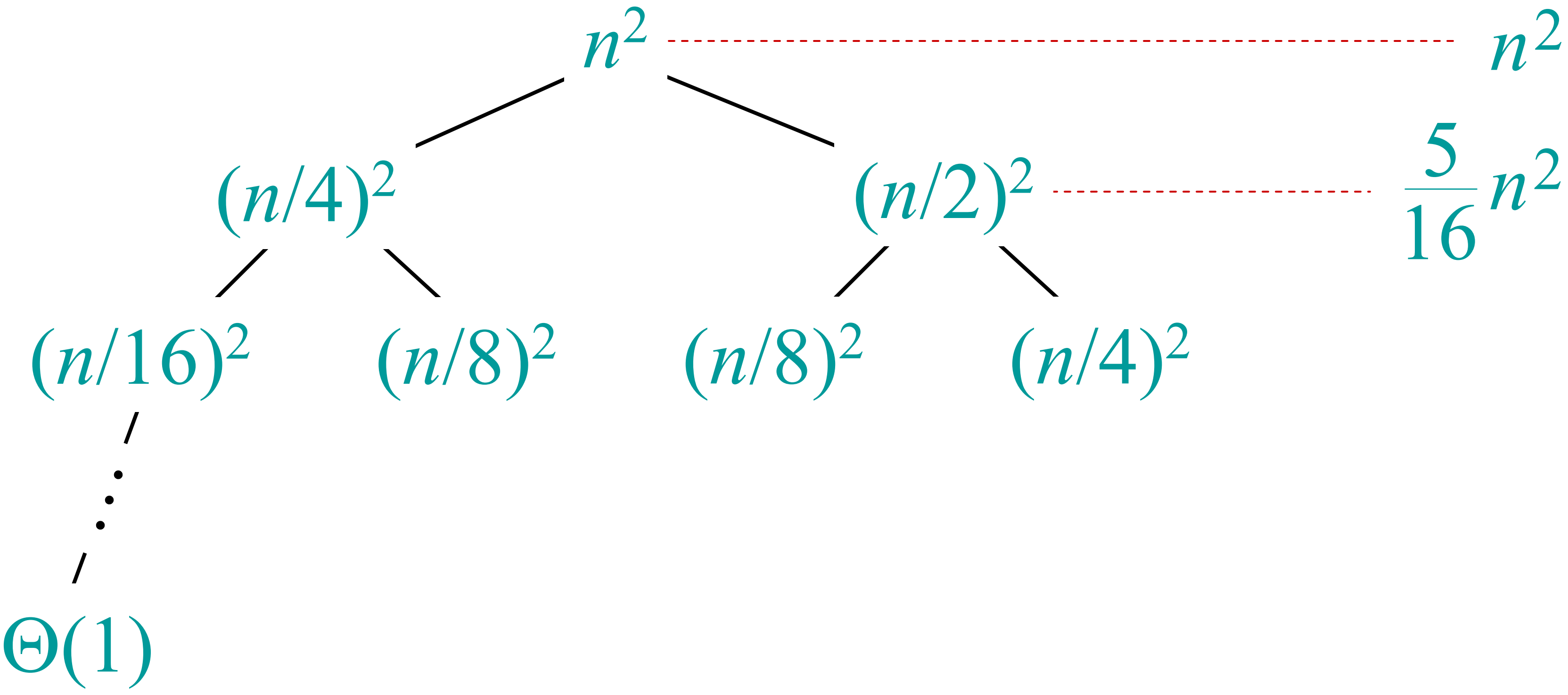
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



n^2 .... n^2

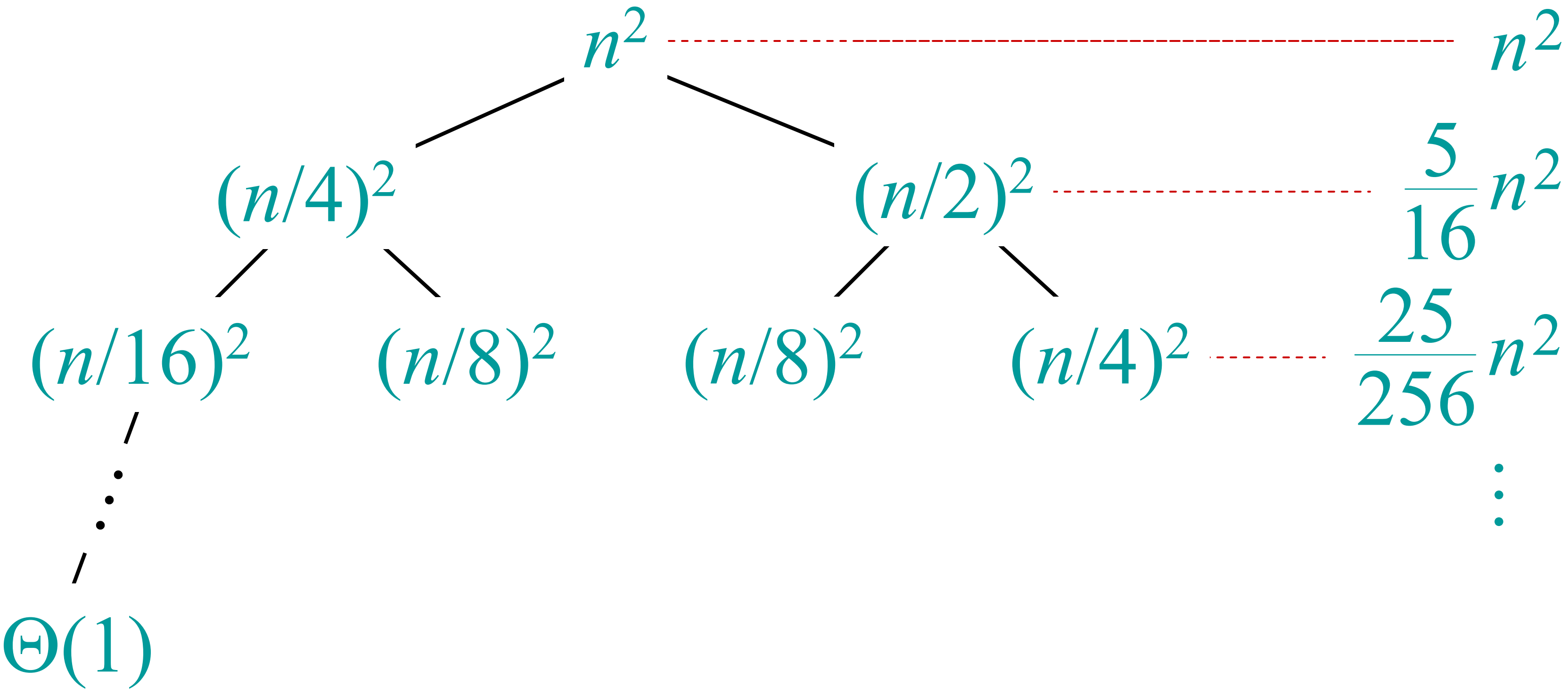
$(n/4)^2$ $\quad$ $(n/2)^2$ .... $\frac{5}{16}n^2$

$(n/16)^2$ $\quad$ $(n/8)^2$ $\quad$ $(n/8)^2$ $\quad$ $(n/4)^2$

$\Theta(1)$

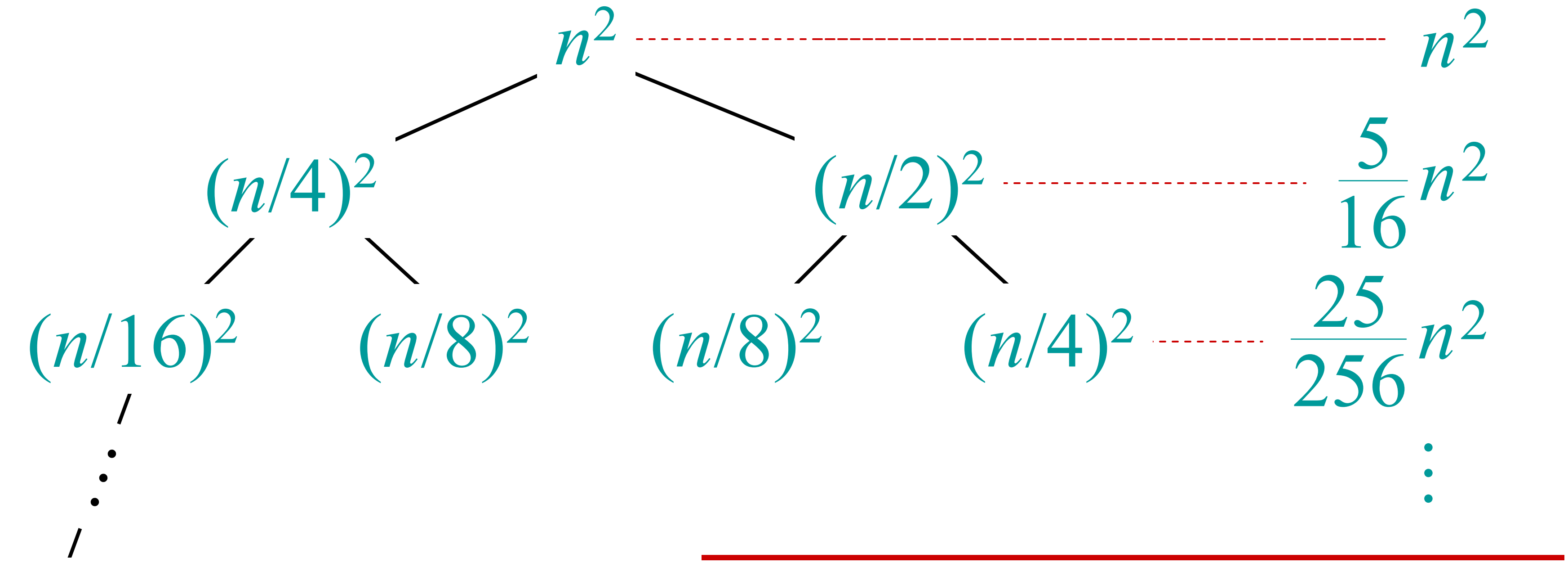
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



n^2 n^2

$(n/4)^2$ $(n/2)^2$ $\frac{5}{16}n^2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$ $\frac{25}{256}n^2$

$\Theta(1)$

$$\text{Total} = n^2\left(1+\frac{5}{16}+\left(\frac{5}{16}\right)^2+\left(\frac{5}{16}\right)^3+\cdots\right)$$

$$= \Theta(n^2)$$

geometric series

The master method

The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$

Case 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2).$

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

Case 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n).$

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

General method (Akra-Bazzi)

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n)$$

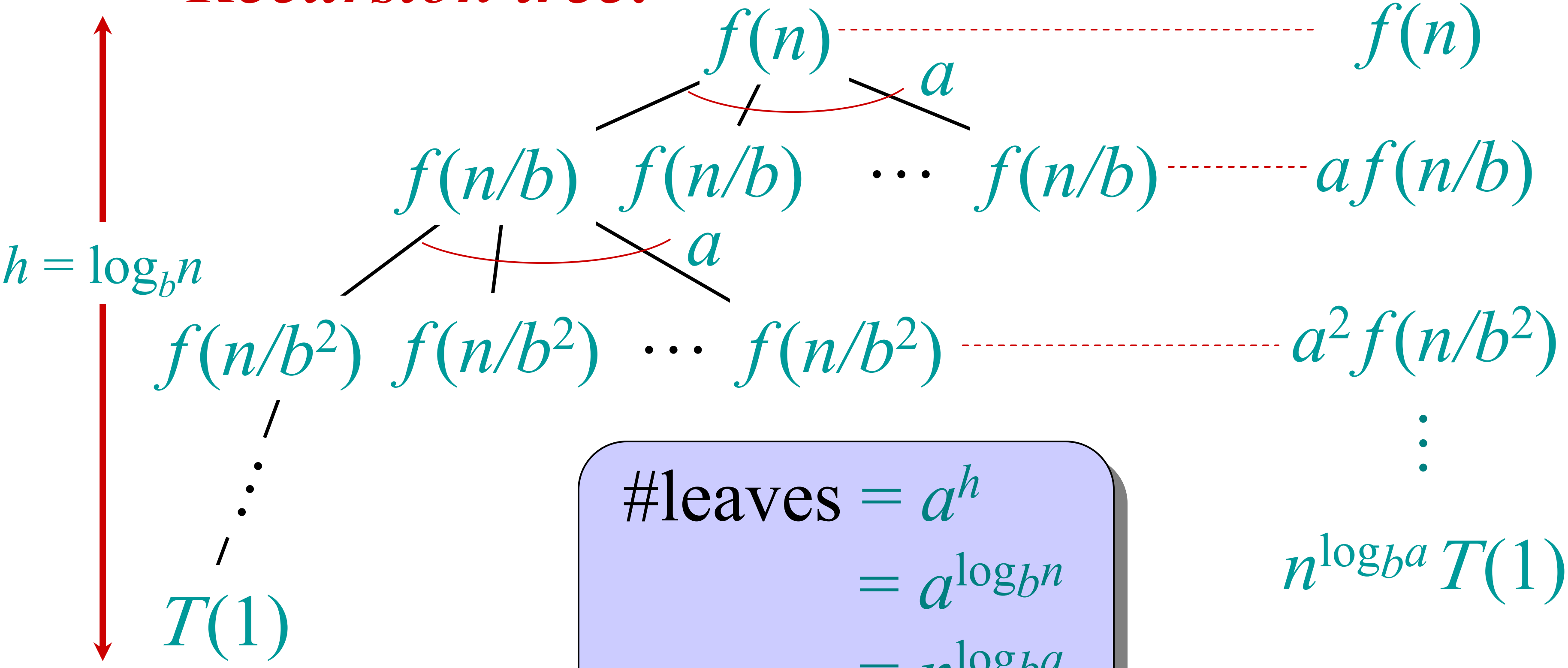
Let p be the unique solution to

$$\sum_{i=1}^{k} \left(a_i / b_i^p\right) = 1.$$

Then, the answers are the same as for the master method, but with n^p instead of $n^{\log_b a}$. (*Akra and Bazzi also prove an even more general result.*)

Idea of master theorem

Recursion tree:



$f(n)$

a

$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$

a

$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$

$T(1)$

$h = \log_b n$

$f(n)$

$a f(n/b)$

$a^2 f(n/b^2)$

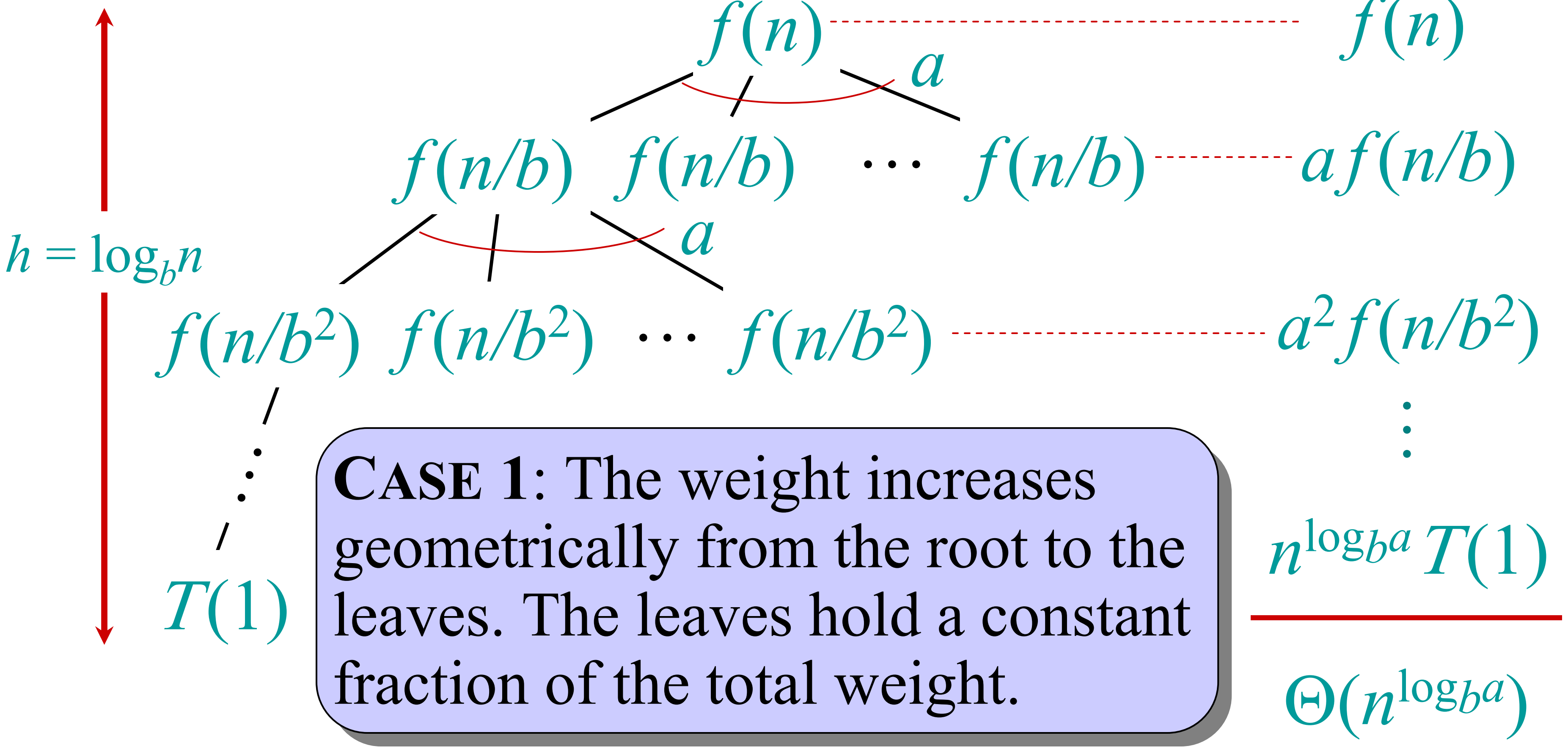
$\vdots$

$n^{\log_b a} T(1)$

$$\begin{aligned} \#\text{leaves} &= a^h \\ &= a^{\log_b n} \\ &= n^{\log_b a} \end{aligned}$$

Idea of master theorem

Recursion tree:



$h = \log_b n$

$f(n)$ — a — $f(n)$

$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$ — a — $af(n/b)$

$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$ — $a^2 f(n/b^2)$

$\vdots$

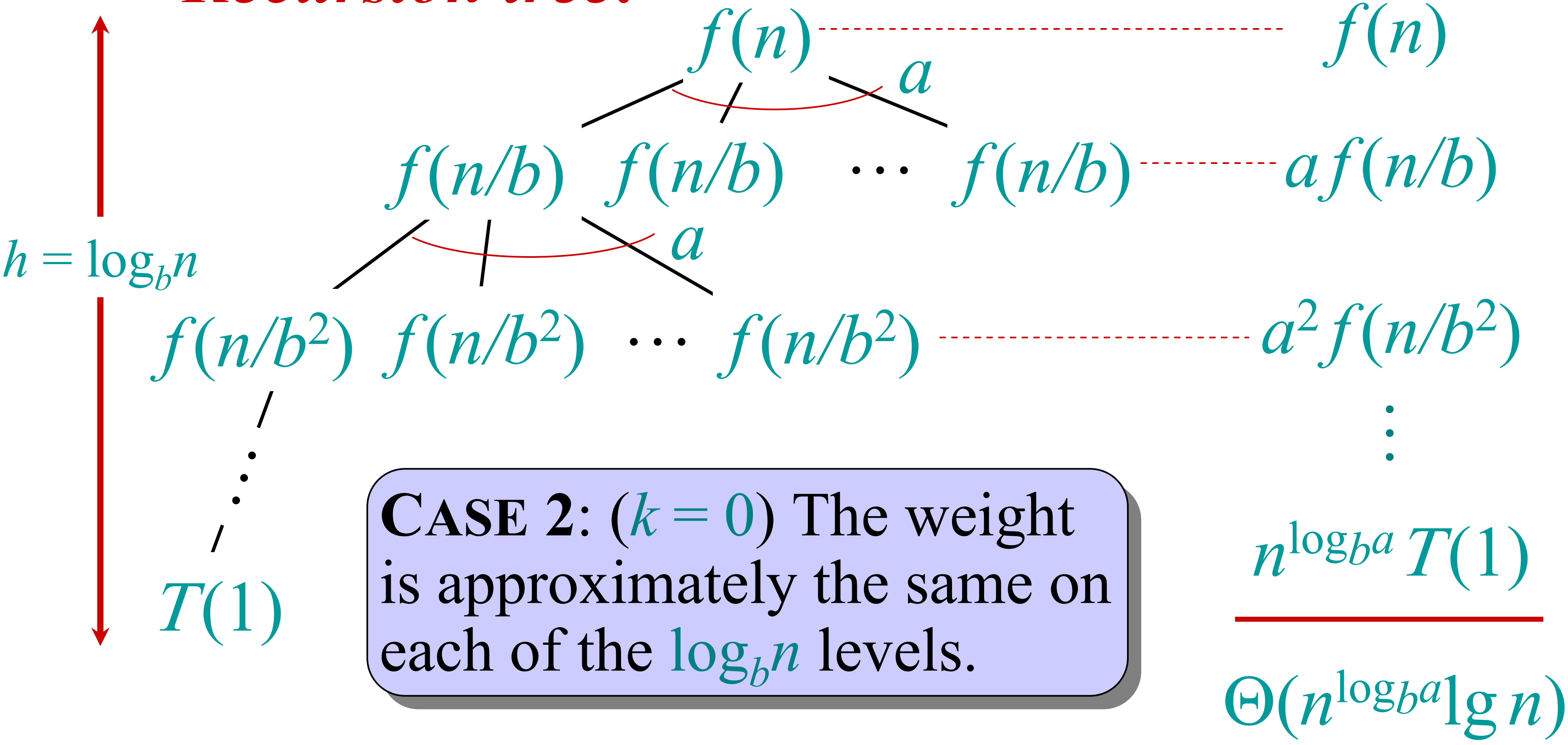
$T(1)$ — $n^{\log_b a}\, T(1)$

$\Theta(n^{\log_b a})$

Case 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Idea of master theorem

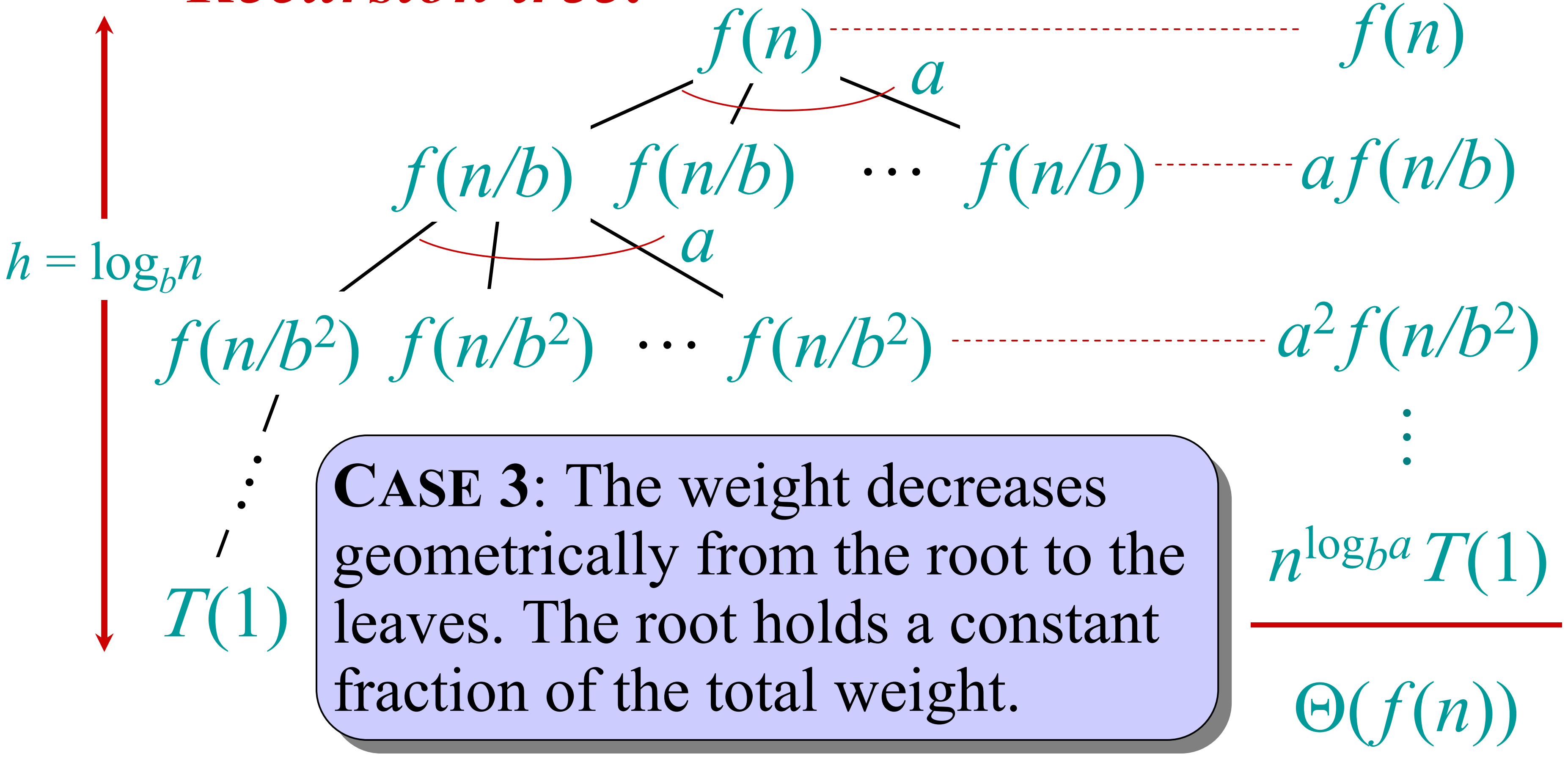
Recursion tree:


$f(n)$
a
$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$
a
$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$
$T(1)$
$h = \log_b n$
$f(n)$
$af(n/b)$
$a^2 f(n/b^2)$
$\vdots$
$n^{\log_b a} T(1)$
$\Theta(n^{\log_b a} \lg n)$

CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Idea of master theorem

Recursion tree:



$f(n)$ a

$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$ a

$h = \log_b n$

$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$

$T(1)$

$f(n)$

$af(n/b)$

$a^2 f(n/b^2)$

$\vdots$

$n^{\log_b a} T(1)$

$\Theta(f(n))$

CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Conclusion

- Next time: applying the master method.
- For proof of master theorem, see CLRS.

Appendix: geometric series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

Return to last
slide viewed.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 3

Prof. Erik Demaine

The divide-and-conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

Example: merge sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.

$$T(n) = 2\,T(n/2) + O(n)$$

# subproblems

subproblem size

work dividing and combining

Master theorem (reprise)

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \le c\,f(n)$

$\Rightarrow T(n) = \Theta(f(n))$.

Merge sort: $a = 2$, $b = 2 \;\Rightarrow\; n^{\log_b a} = n$

$\Rightarrow$ CASE 2 $(k = 0) \;\Rightarrow\; T(n) = \Theta(n \lg n)$.

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

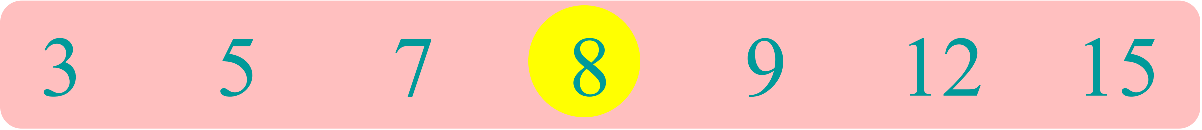
3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9



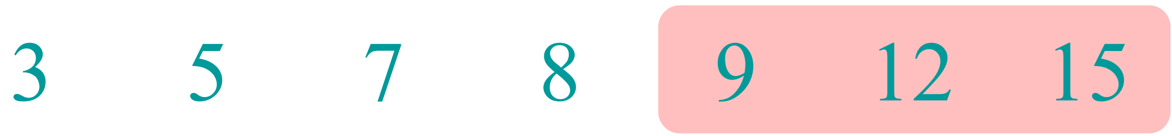
3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9



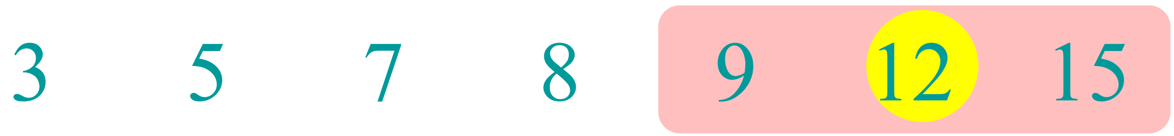
3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9



3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

# subproblems

subproblem size

work dividing and combining

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{Case 2 } (k = 0)$$

$$\Rightarrow \; T(n) = \Theta(\lg n)\,.$$

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \;\Rightarrow\; T(n) = \Theta(\lg n) .$$

Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$$

Naive recursive algorithm: $\Omega(\phi^n)$ (exponential time), where $\phi = (1+\sqrt{5})/2$ is the ***golden ratio***.

Computing Fibonacci numbers

Naive recursive squaring:

$F_n = \phi^n/\sqrt{5}$ rounded to the nearest integer.

- Recursive squaring: $\Theta(\lg n)$ time.
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_\mathrm{n}$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$.

Recursive squaring

Theorem: $$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n .$$

Algorithm: Recursive squaring.

Time = $\Theta(\lg n)$.

Proof of theorem. (Induction on n.)

Base ($n = 1$): $$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 .$$

Recursive squaring

Inductive step ($n \geq 2$):

$$
\begin{aligned}
\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}
&= \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n} \qquad \blacksquare
\end{aligned}
$$

Matrix multiplication

$$\left.\begin{array}{ll}\textbf{Input:} & A = [a_{ij}],\ B = [b_{ij}]. \\ \textbf{Output:} & C = [c_{ij}] = A \cdot B.\end{array}\right\}\ i, j = 1, 2, \ldots, n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Standard algorithm

```
for $i \leftarrow 1$ to $n$
    do for $j \leftarrow 1$ to $n$
        do $c_{ij} \leftarrow 0$
           for $k \leftarrow 1$ to $n$
               do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
```

Running time $= \Theta(n^3)$

Divide-and-conquer algorithm

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

$$C \quad = \quad A \quad \cdot \quad B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dg \end{aligned}\right\}$$

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Analysis of D&C algorithm

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

# submatrices

submatrix size

work adding submatrices

$$n^{\log_b a} = n^{\log_2 8} = n^3 \;\Rightarrow\; \text{C}\small{\text{ASE}}\ 1 \;\Rightarrow\; T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= (a + d)(e + h) \\
&\quad + d(g - e) - (a + b)h \\
&\quad + (b - d)(g + h) \\
&= ae + ah + de + dh \\
&\quad + dg - de - ah - bh \\
&\quad + bg + bh - dg - dh \\
&= ae + bg
\end{aligned}
$$

Strassen's algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using + and – .
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using + and – on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Analysis of Strassen

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

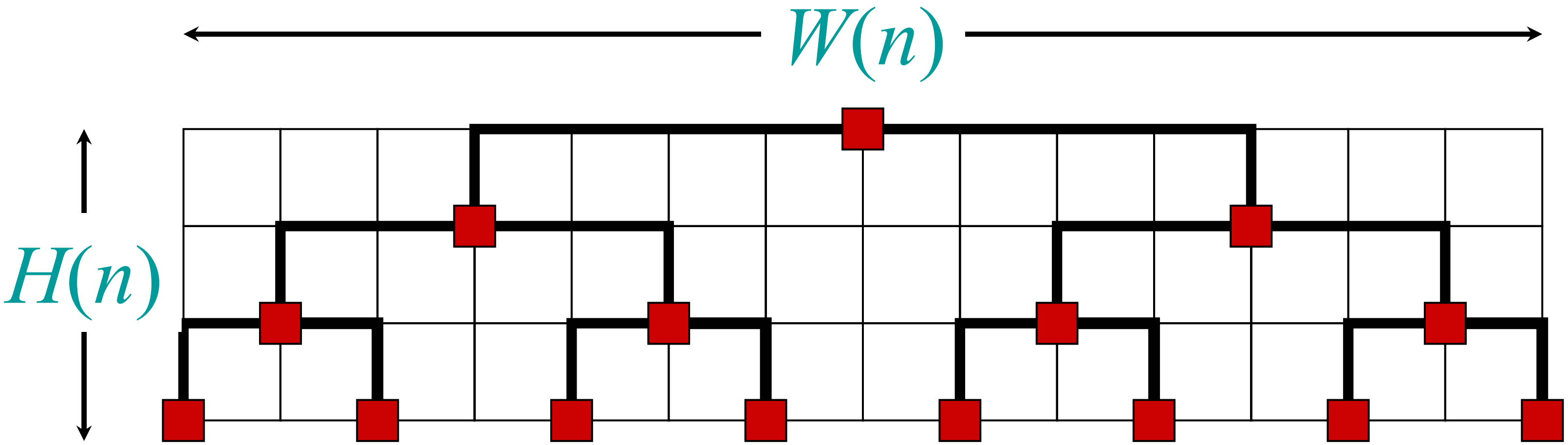
$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \;\Rightarrow\; \text{Case 1} \;\Rightarrow\; T(n) = \Theta(n^{\lg 7}).$$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Best to date (of theoretical interest only): $\Theta(n^{2.376\cdots})$.

VLSI layout

Problem: Embed a complete binary tree with n leaves in a grid using minimal area.

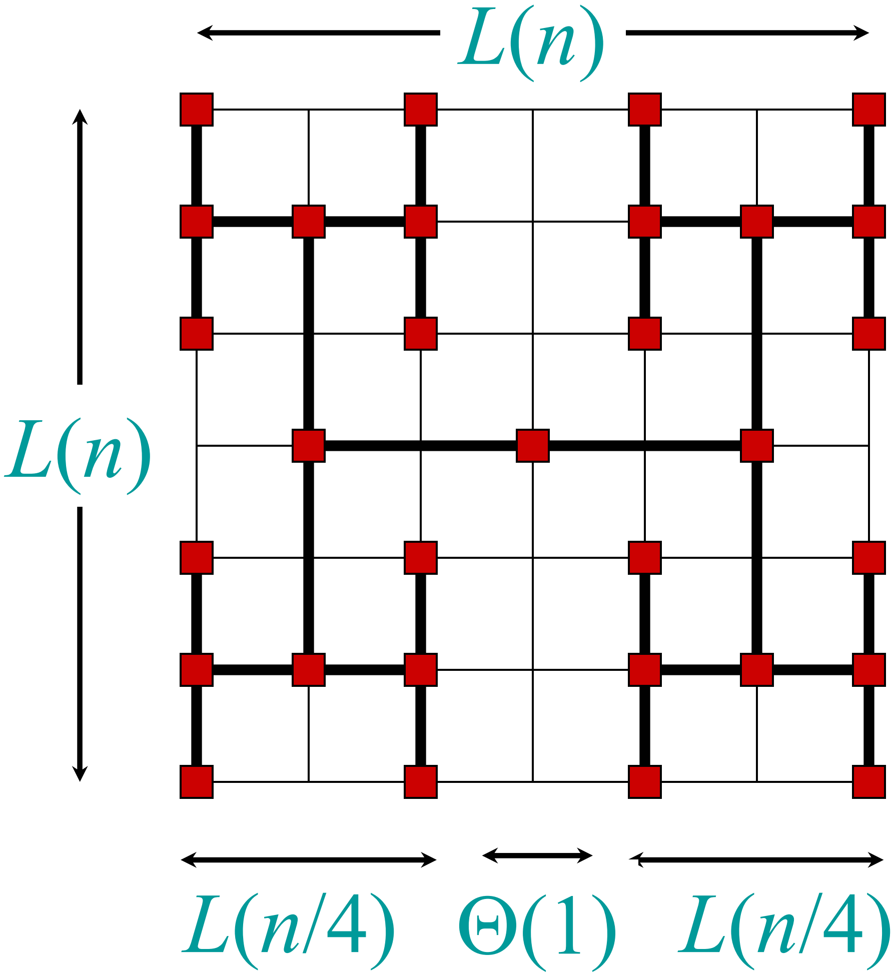


$$H(n) = H(n/2) + \Theta(1) = \Theta(\lg n)$$

$$W(n) = 2\,W(n/2) + \Theta(1) = \Theta(n)$$

$$\text{Area} = \Theta(n \lg n)$$

H-tree embedding



$$\begin{aligned} L(n) &= 2L(n/4) + \Theta(1) \\ &= \Theta(\sqrt{n}) \end{aligned}$$

Area $= \Theta(n)$

Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- Can lead to more efficient algorithms

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 4

Prof. Charles E. Leiserson

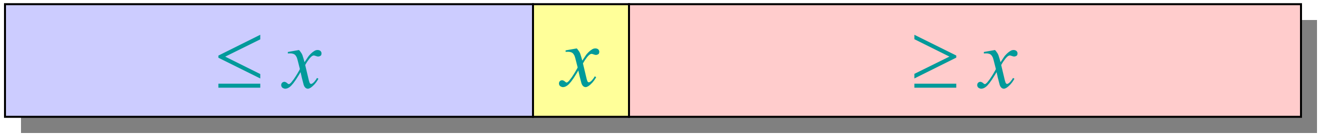
Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).

Divide and conquer

Quicksort an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\le x \le$ elements in upper subarray.



$\le x$ | x | $\ge x$

2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

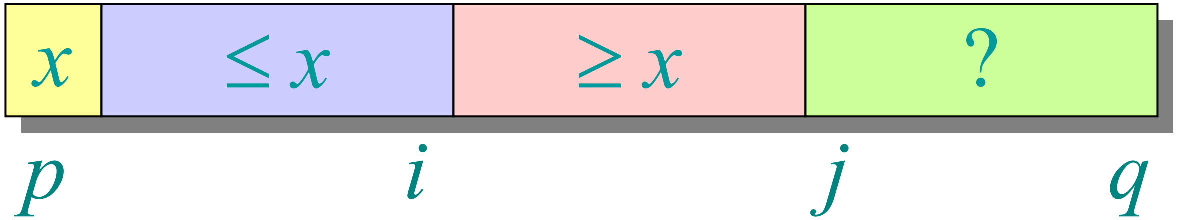
Key: *Linear-time partitioning subroutine.*

Partitioning subroutine

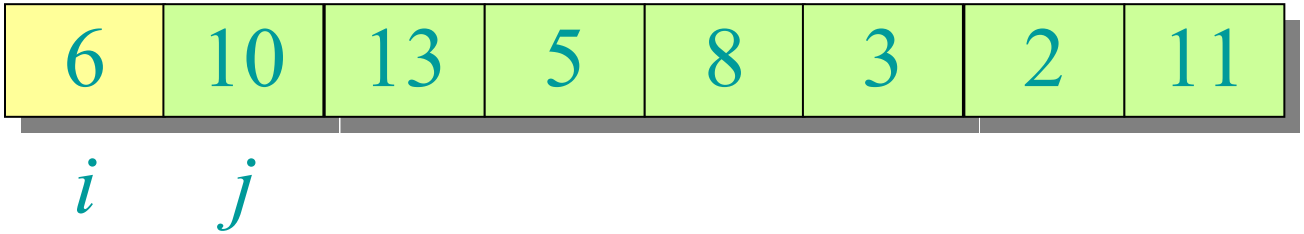
```
PARTITION($A, p, q$)  ▷ $A[p\,..\,q]$
    $x \leftarrow A[p]$         ▷ pivot $= A[p]$
    $i \leftarrow p$
    for $j \leftarrow p + 1$ to $q$
        do if $A[j] \leq x$
              then  $i \leftarrow i + 1$
                    exchange $A[i] \leftrightarrow A[j]$
    exchange $A[p] \leftrightarrow A[i]$
    return $i$
```

Running time $= O(n)$ for n elements.

Invariant:

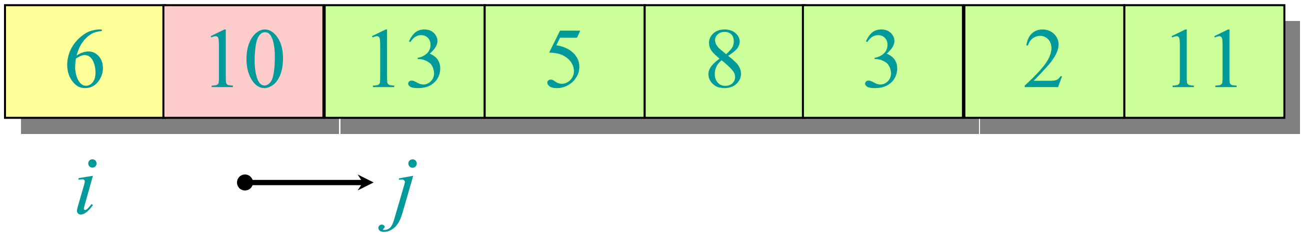


Example of partitioning



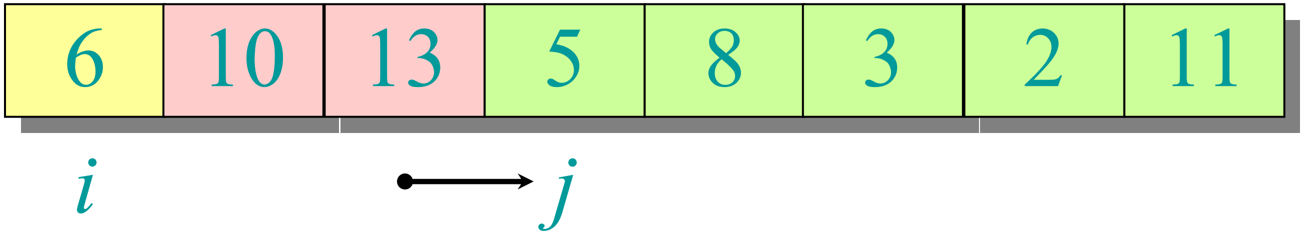
6	10	13	5	8	3	2	11
i	*j*						

Example of partitioning



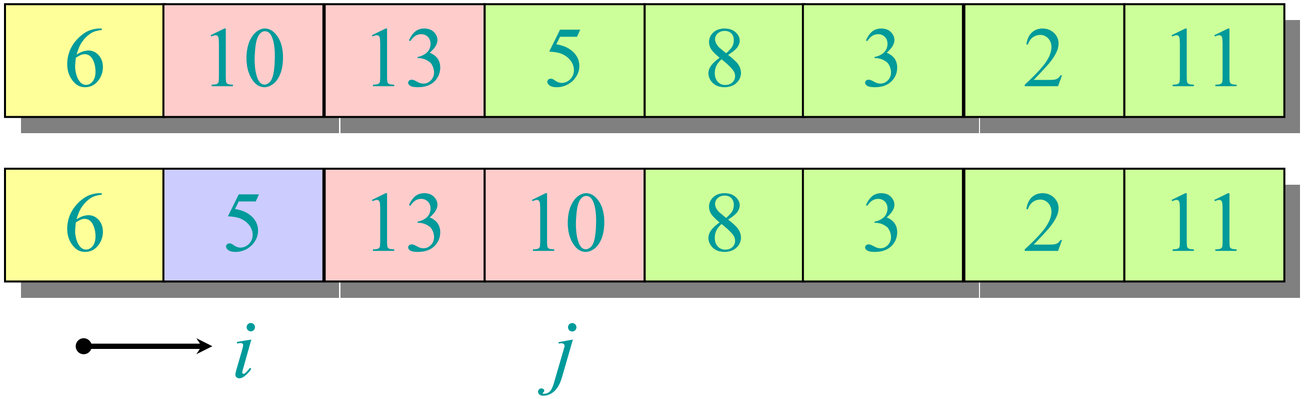
6	10	13	5	8	3	2	11
i		*j*					

Example of partitioning



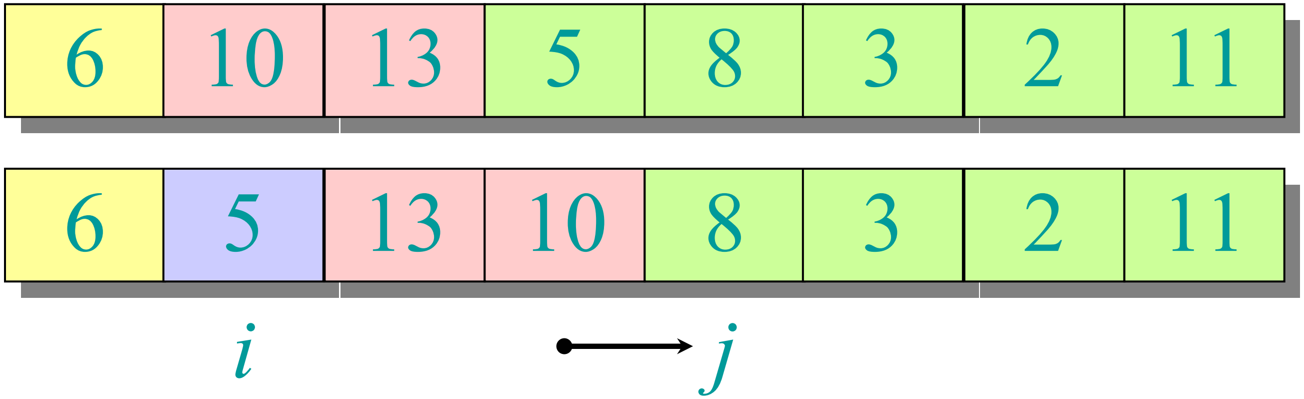
6 10 13 5 8 3 2 11

Example of partitioning



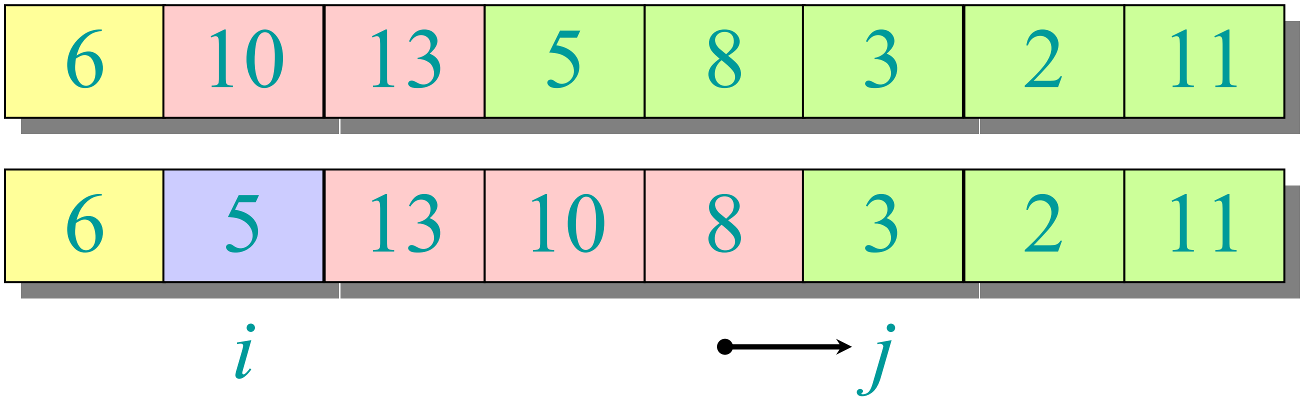
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11

Example of partitioning

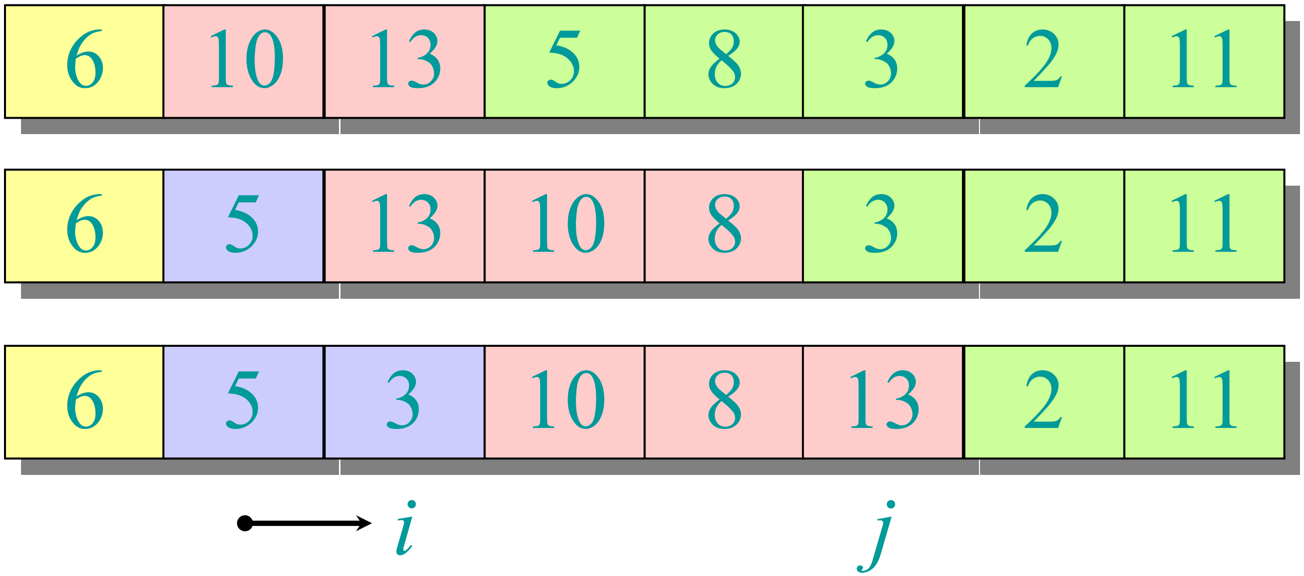


6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11

Example of partitioning

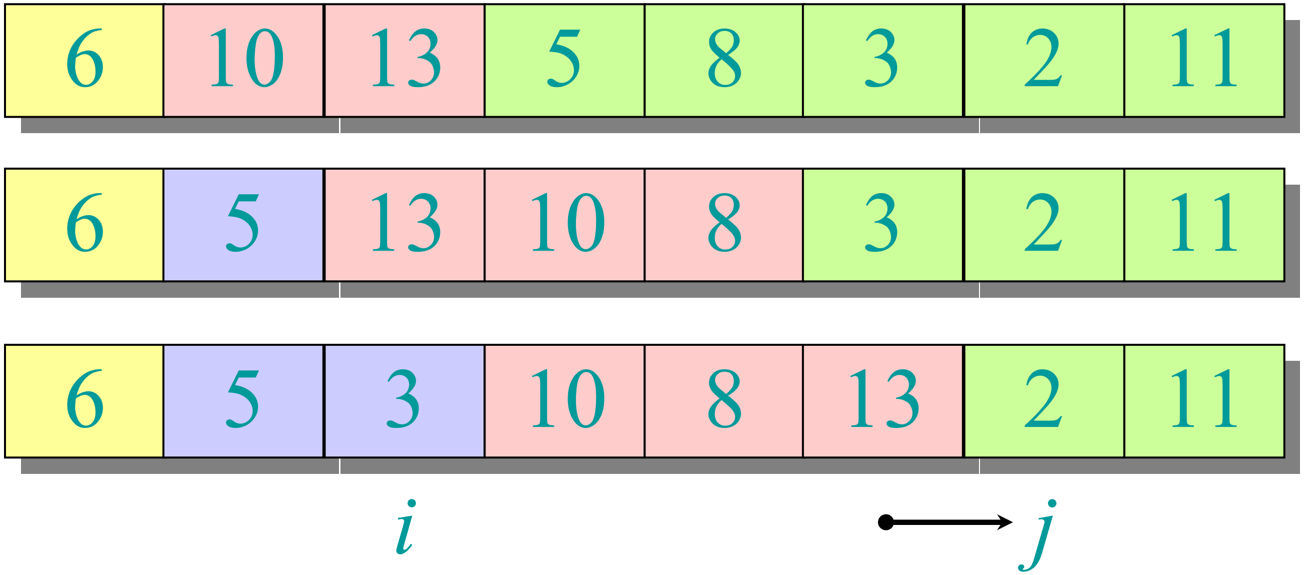


Example of partitioning



6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11

Example of partitioning

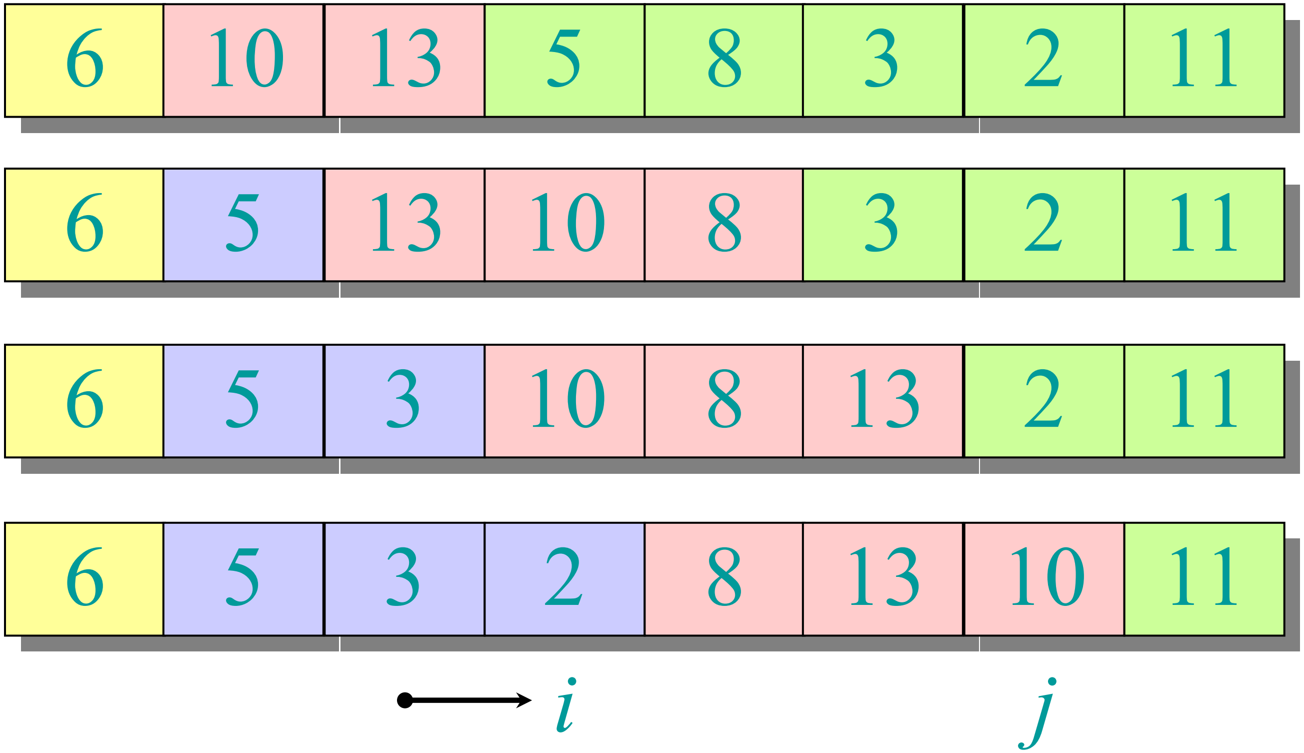


6 10 13 5 8 3 2 11

6 5 13 10 8 3 2 11

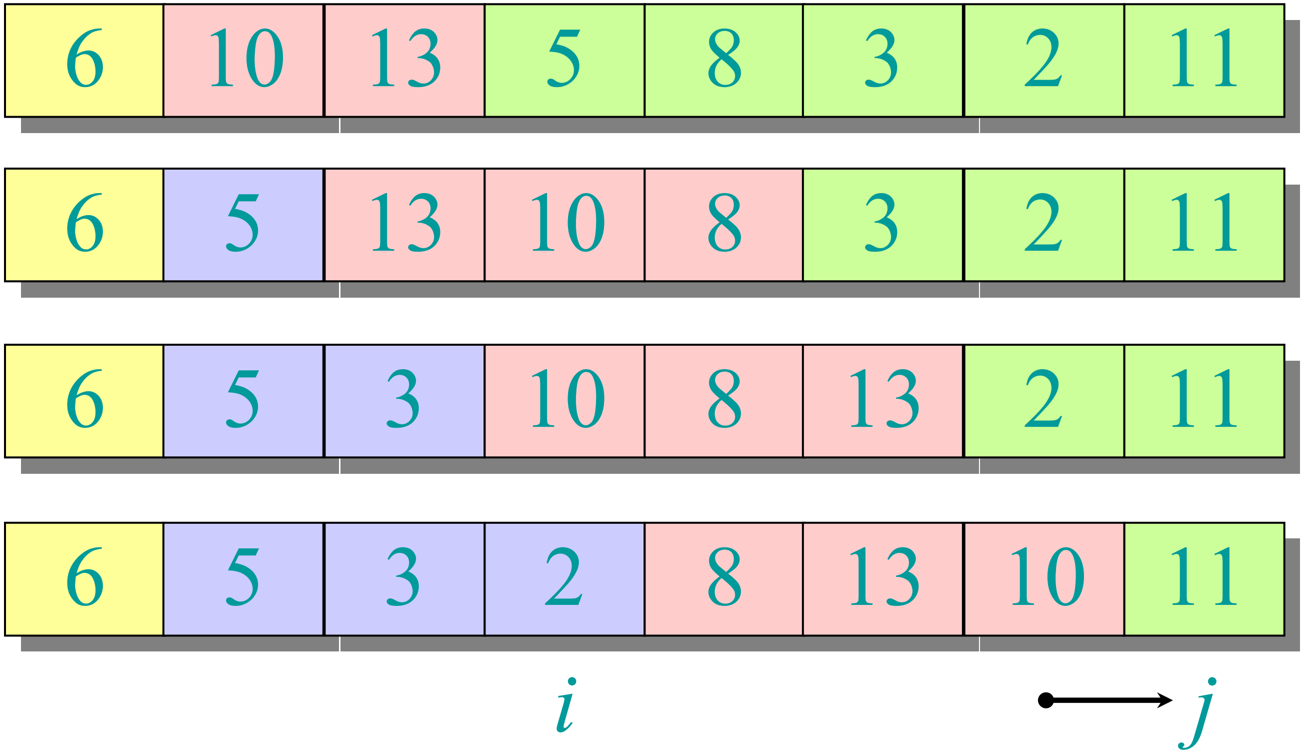
6 5 3 10 8 13 2 11

Example of partitioning



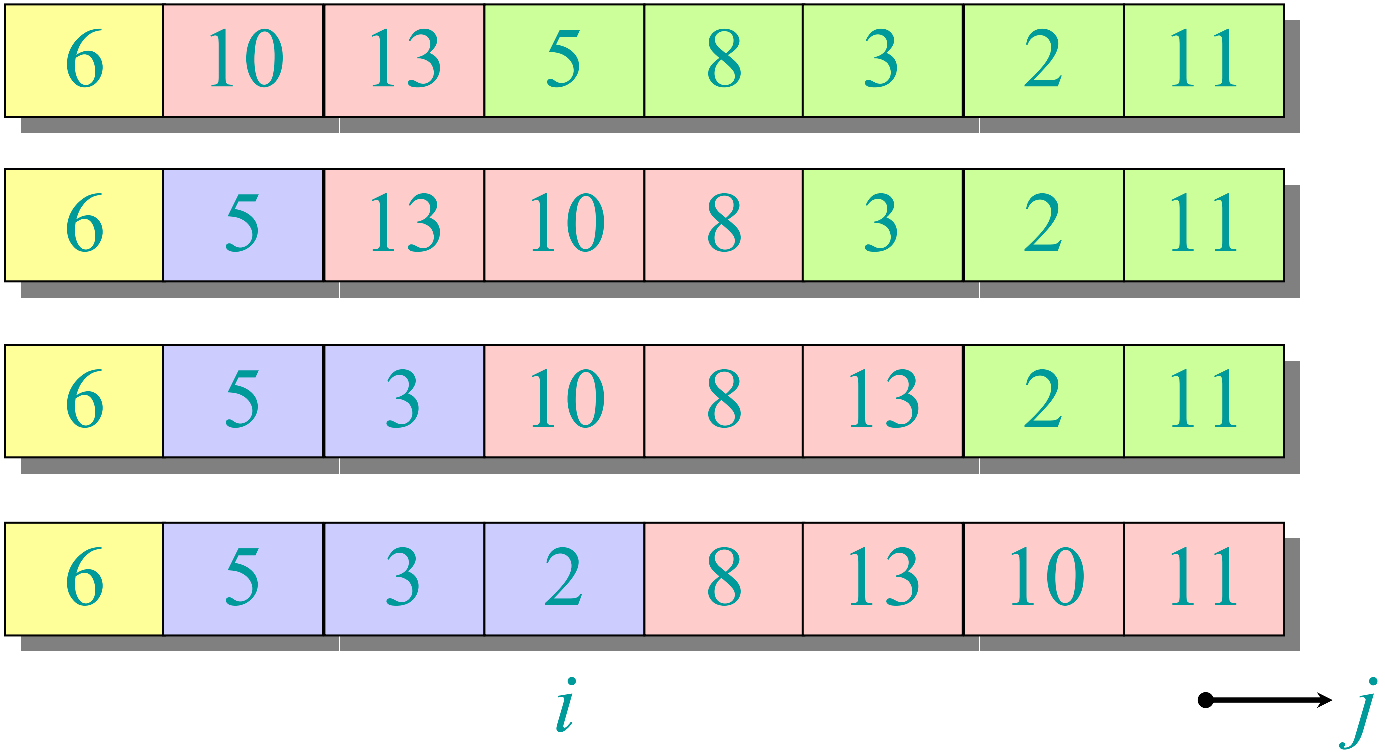
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

Example of partitioning



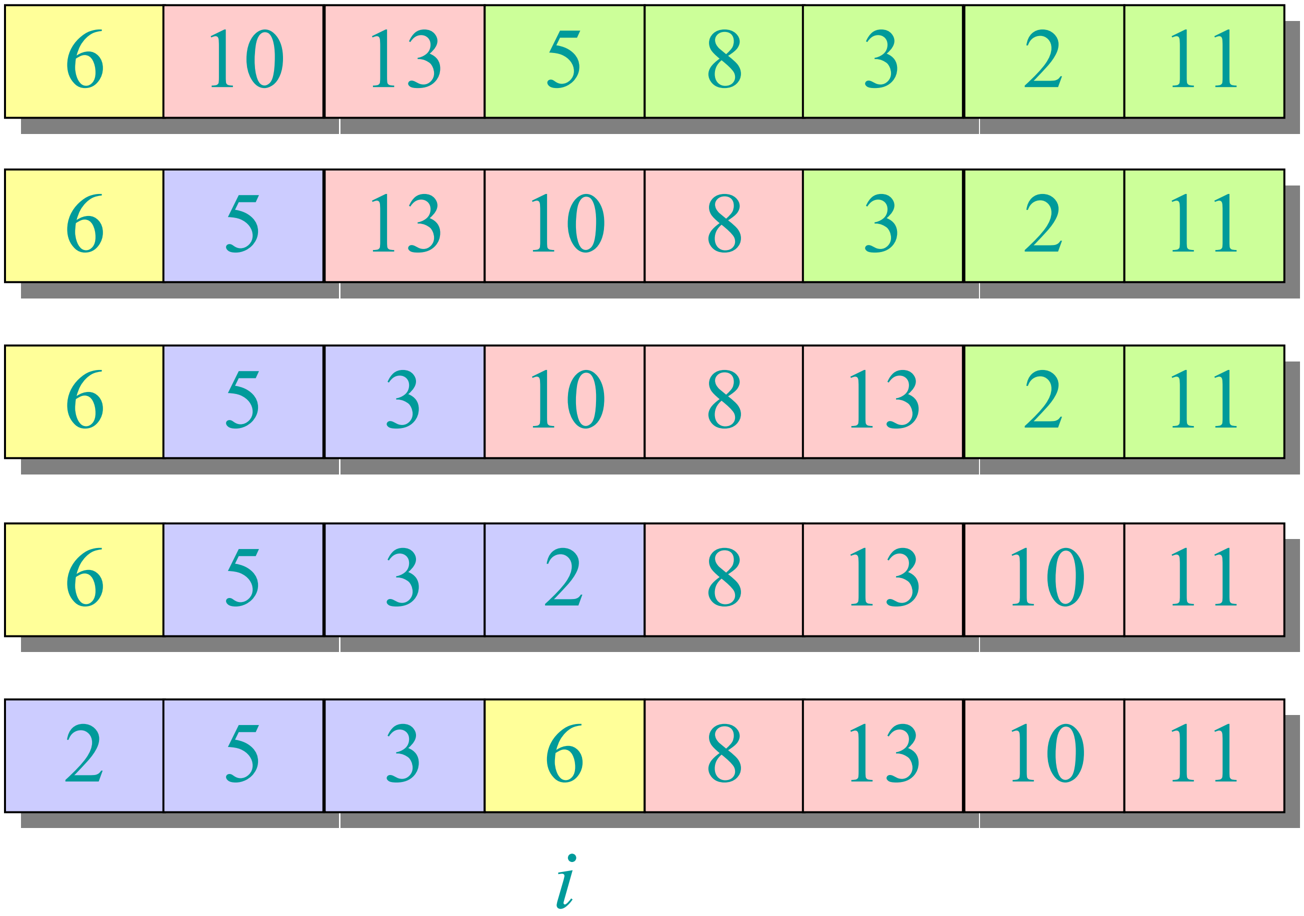
6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

Example of partitioning



6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

Example of partitioning



6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11

Pseudocode for quicksort

```
QUICKSORT$(A, p, r)$
    if $p < r$
        then $q \leftarrow$ PARTITION$(A, p, r)$
            QUICKSORT$(A, p, q-1)$
            QUICKSORT$(A, q+1, r)$
```

Initial call: QUICKSORT$(A, 1, n)$

Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.

Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \qquad \textit{\textbf{(arithmetic series)}} \end{aligned}$$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

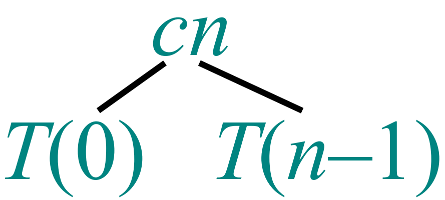
Worst-case recursion tree

$$T(n) = T(0) + T(n{-}1) + cn$$

$T(n)$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

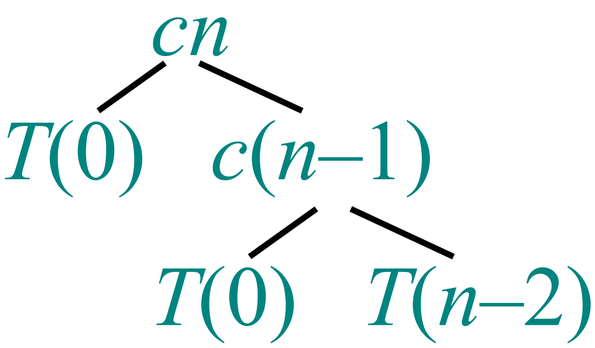


cn

$T(0)$ $T(n-1)$

Worst-case recursion tree

$$T(n) = T(0) + T(n\text{–}1) + cn$$



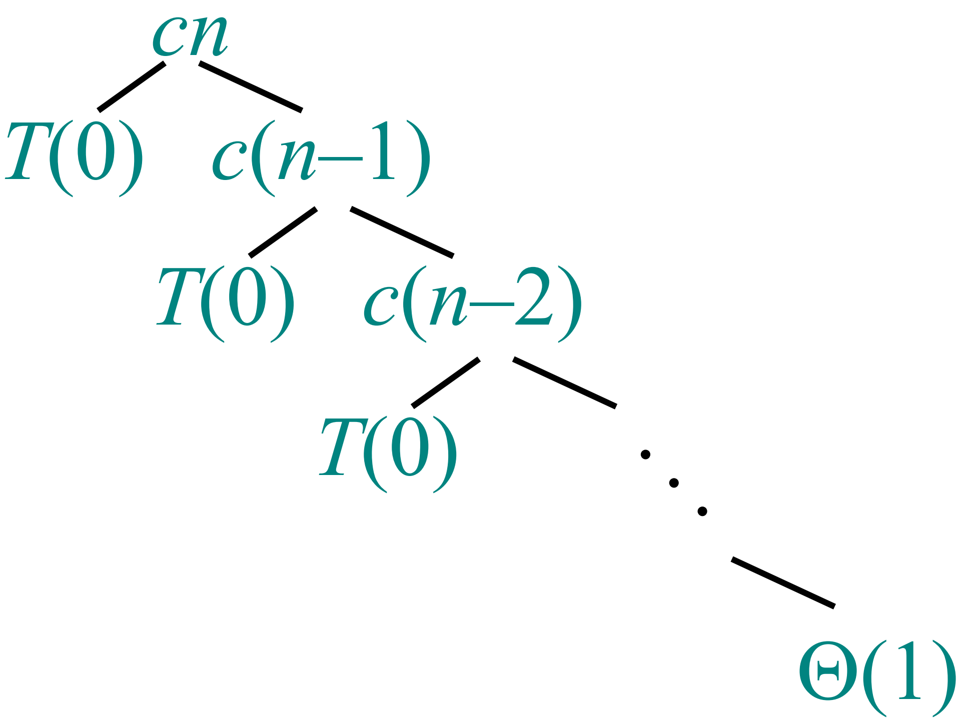
cn

$T(0)$ $c(n\text{–}1)$

$T(0)$ $T(n\text{–}2)$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



cn

$T(0)$ $c(n-1)$

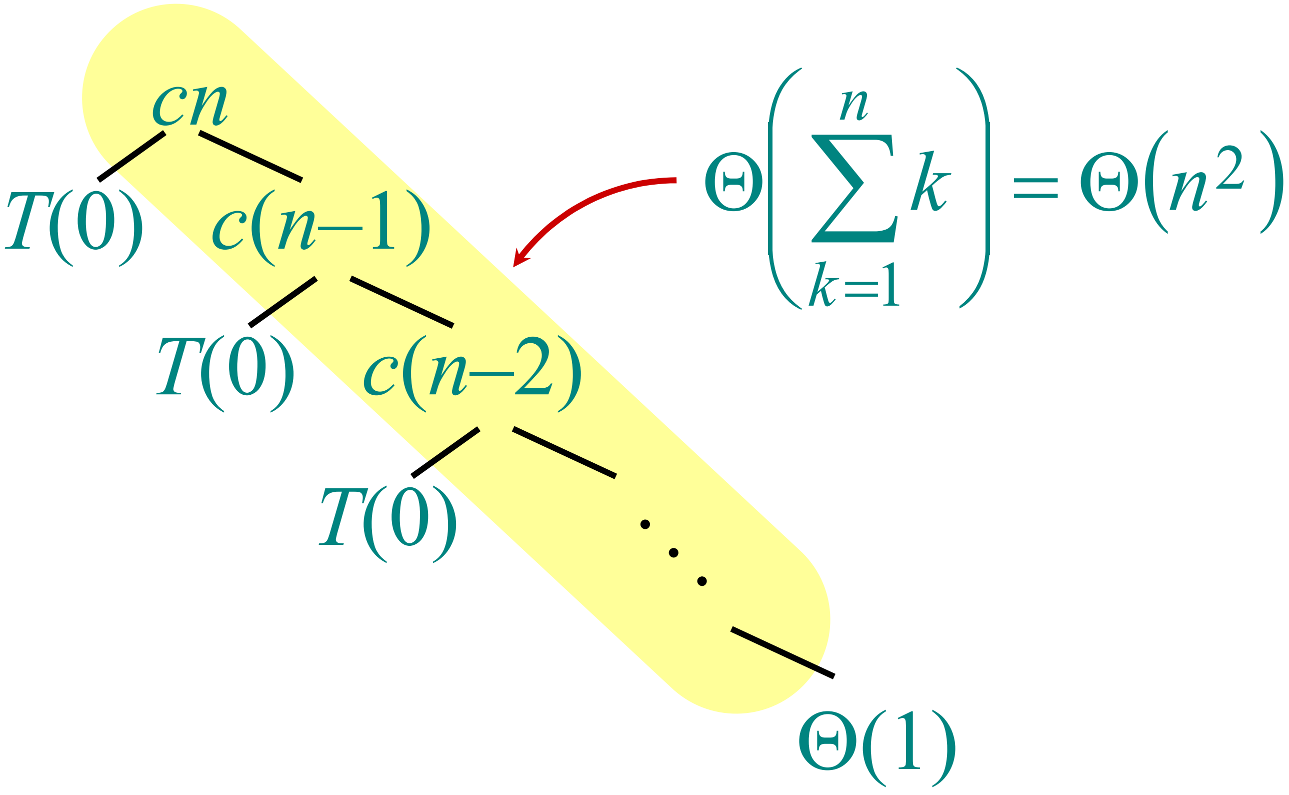
$T(0)$ $c(n-2)$

$T(0)$

$\Theta(1)$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



cn

$T(0)$ $c(n-1)$

$T(0)$ $c(n-2)$

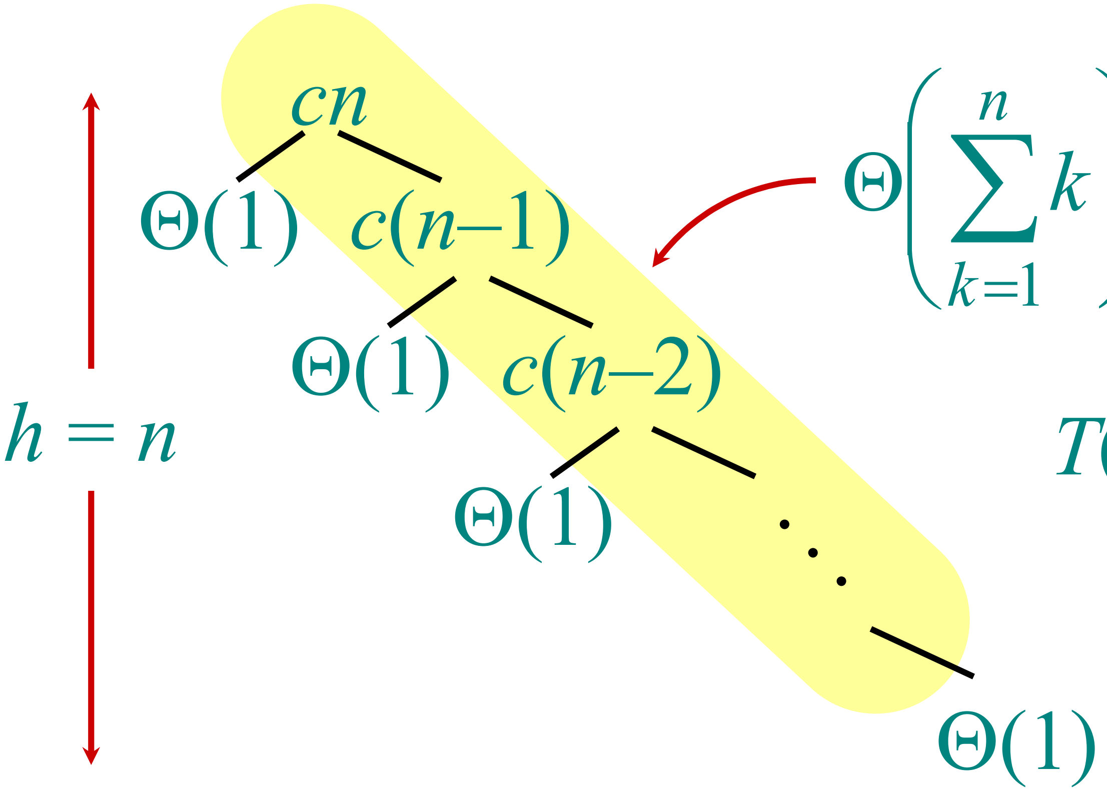
$T(0)$ …

$\Theta(1)$

$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Best-case analysis

(For intuition only!)

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

(same as merge sort)

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

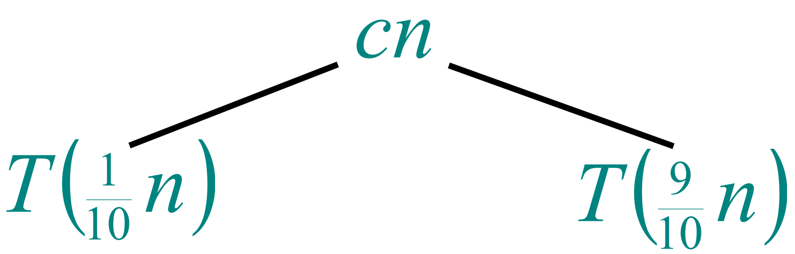
$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

Analysis of “almost-best” case

$$T(n)$$

Analysis of “almost-best” case

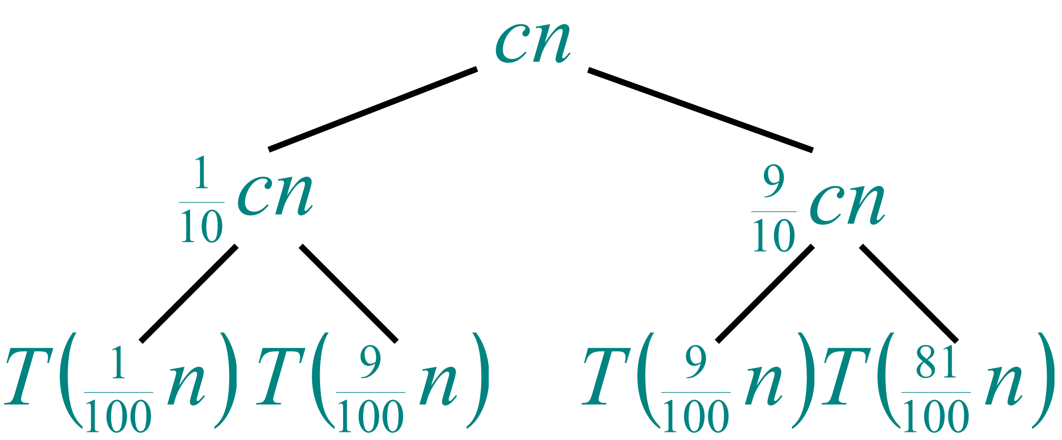


cn

$T\left(\frac{1}{10}n\right)$

$T\left(\frac{9}{10}n\right)$

Analysis of “almost-best” case

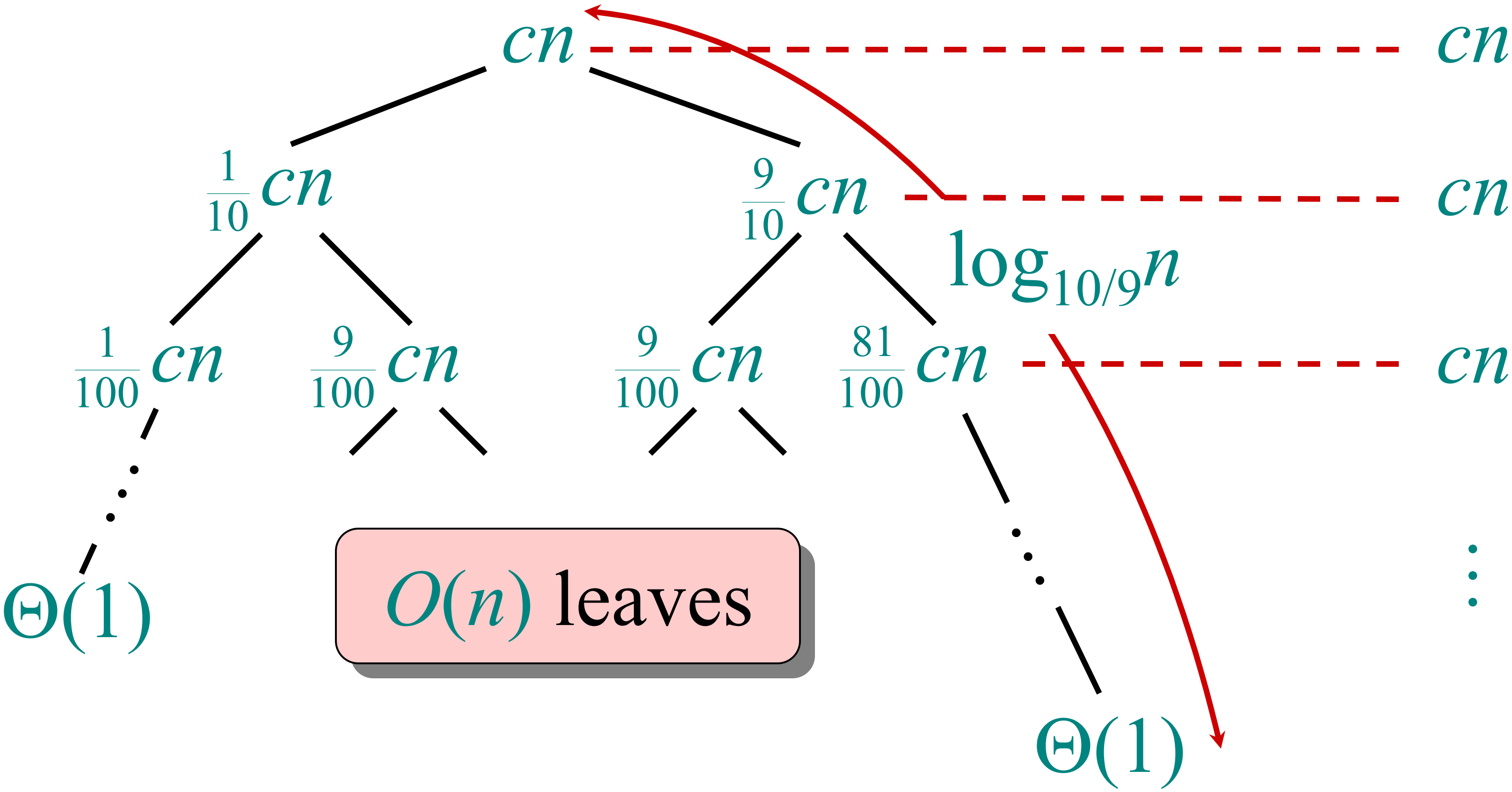


cn

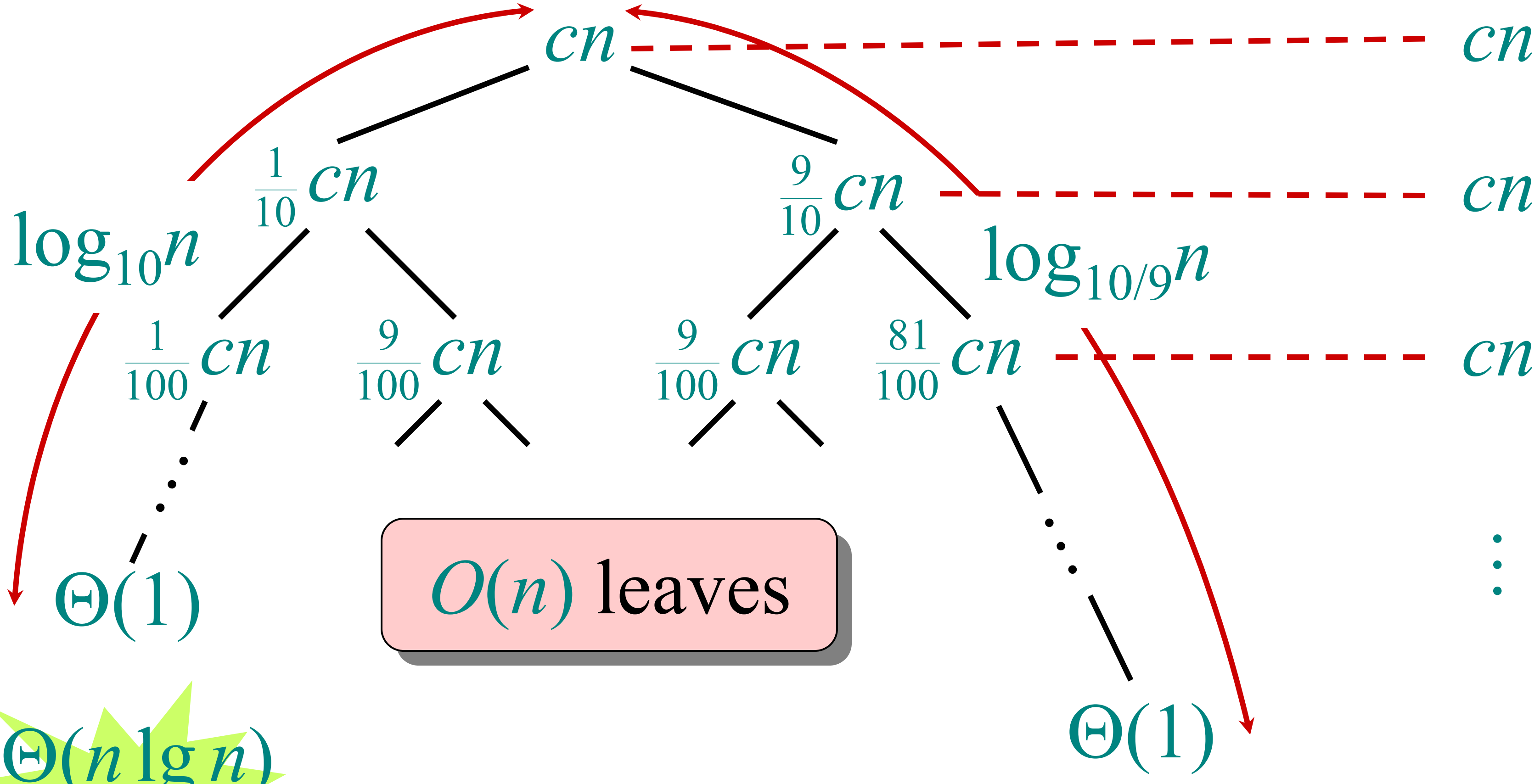
$\frac{1}{10}cn$ $\frac{9}{10}cn$

$T\left(\frac{1}{100}n\right)$ $T\left(\frac{9}{100}n\right)$ $T\left(\frac{9}{100}n\right)$ $T\left(\frac{81}{100}n\right)$

Analysis of “almost-best” case



Analysis of “almost-best” case



$\Theta(n \lg n)$

Lucky!

$$cn \log_{10}n \le T(n) \le cn \log_{10/9}n + O(n)$$

More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n)$$ ***lucky***

$$U(n) = L(n-1) + \Theta(n)$$ ***unlucky***

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Lucky!

How can we make sure we are usually lucky?

Randomized quicksort

IDEA: Partition around a ***random*** element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if } \textsc{Partition} \text{ generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Analysis (continued)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \quad\vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \big(T(k) + T(n-k-1) + \Theta(n)\big).$$

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \bigl(T(k) + T(n-k-1) + \Theta(n)\bigr)\right]$$

Take expectations of both sides.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k \big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

Linearity of expectation.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Independence of X_k from other random choices.

Calculating expectation

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k\big(T(k)+T(n-k-1)+\Theta(n)\big)\right] \\
&= \sum_{k=0}^{n-1} E\big[X_k\big(T(k)+T(n-k-1)+\Theta(n)\big)\big] \\
&= \sum_{k=0}^{n-1} E[X_k]\cdot E[T(k)+T(n-k-1)+\Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)
\end{aligned}
$$

Linearity of expectation; $E[X_k] = 1/n$.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(k)+T(n-k-1)+\Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(k)+T(n-k-1)+\Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k]\cdot E[T(k)+T(n-k-1)+\Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1}\Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.

Hairy recurrence

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \leq a n \lg n$ for constant $a > 0$.

- Choose a large enough so that $a n \lg n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

Substitute inductive hypothesis.

Substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

Use fact.

Substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

Express as ***desired*** – ***residual***.

Substitution method

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&= \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an\lg n\,,
\end{aligned}
$$

if a is chosen large enough so that $an/4$ dominates the $\Theta(n)$.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 5

Prof. Erik Demaine

How fast can we sort?

All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

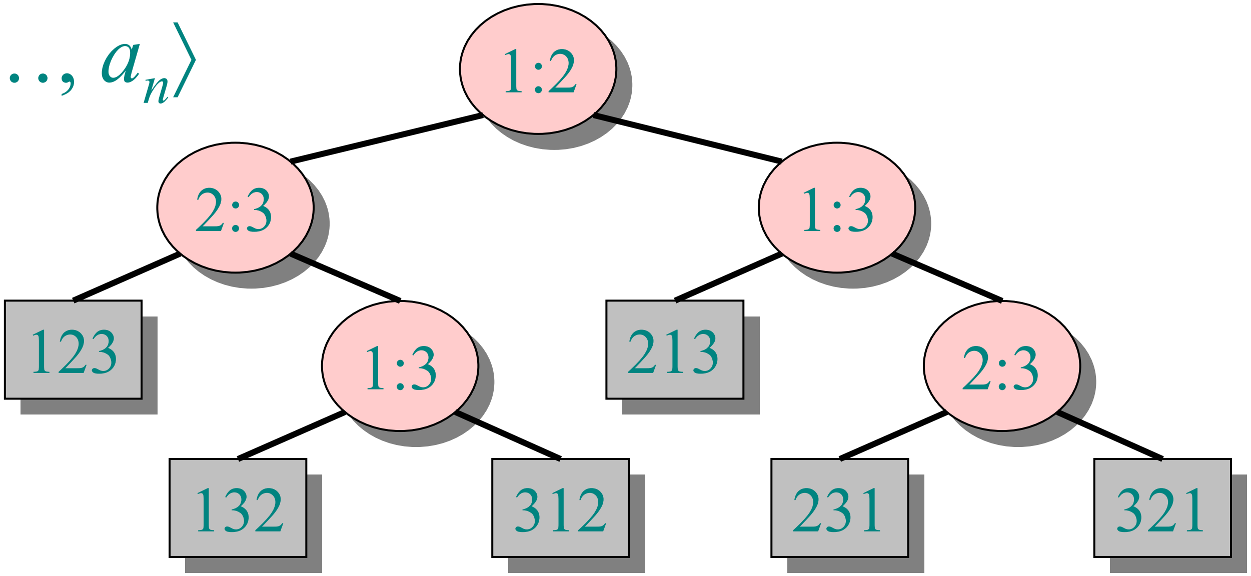
The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.

Decision-tree example

Sort $\langle a_1, a_2, \ldots, a_n \rangle$

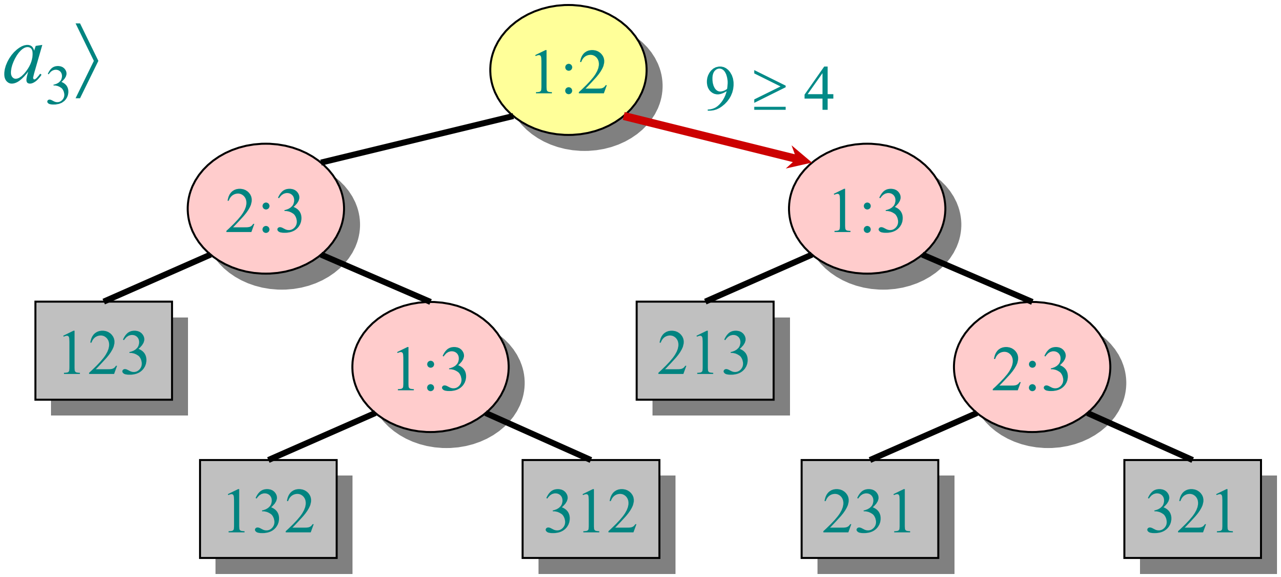


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

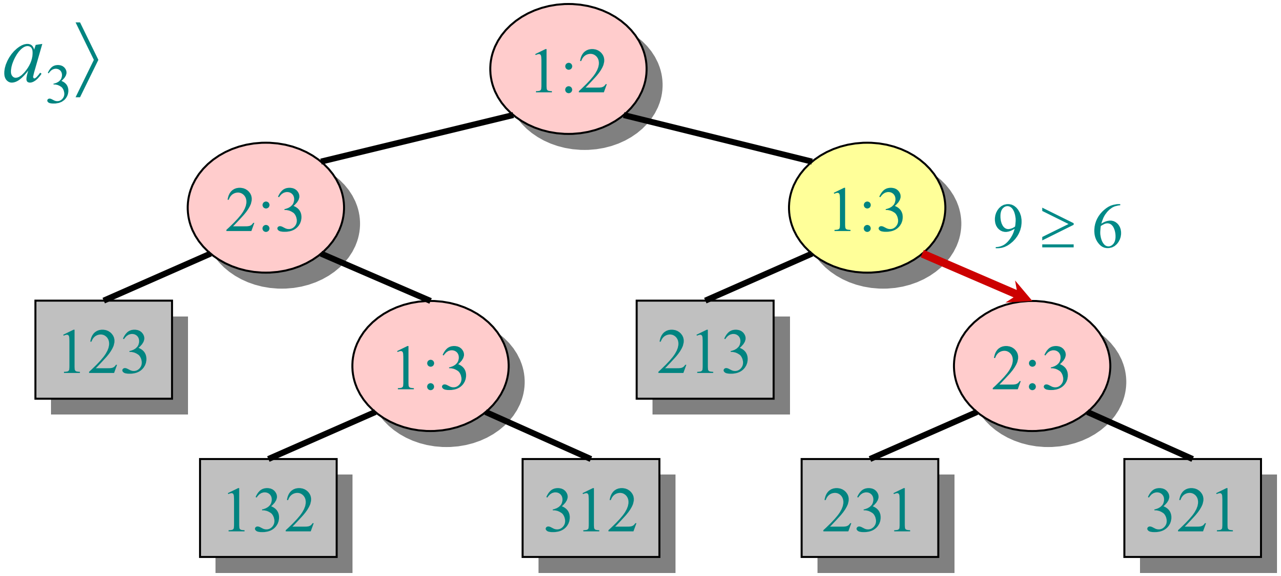


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2,\ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\, 9, 4, 6 \,\rangle$:

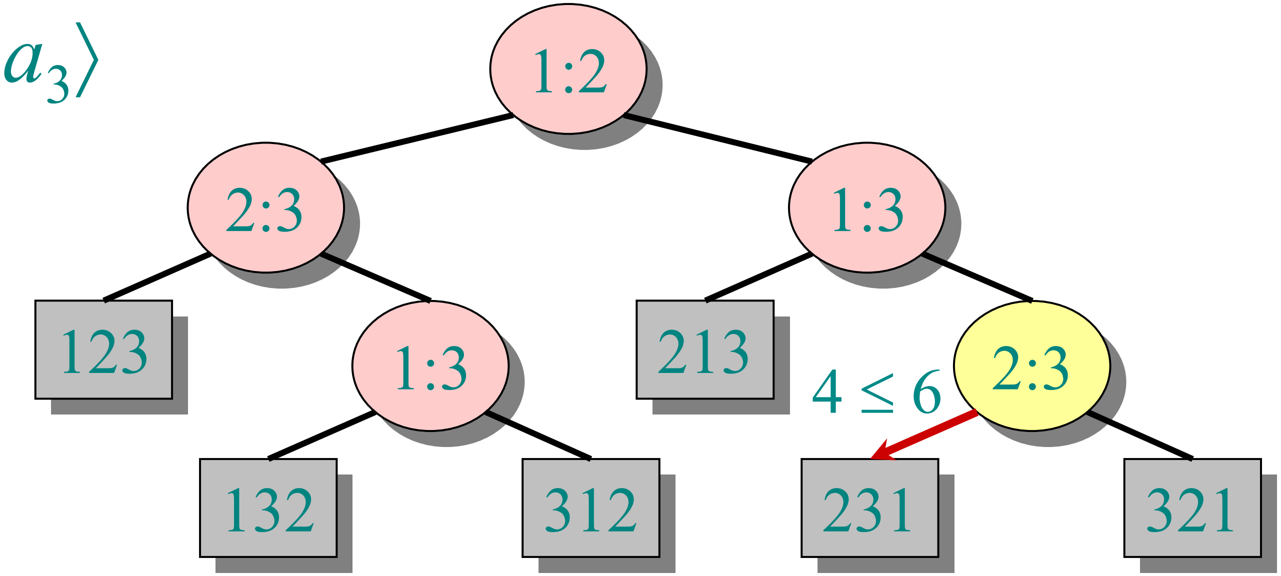


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

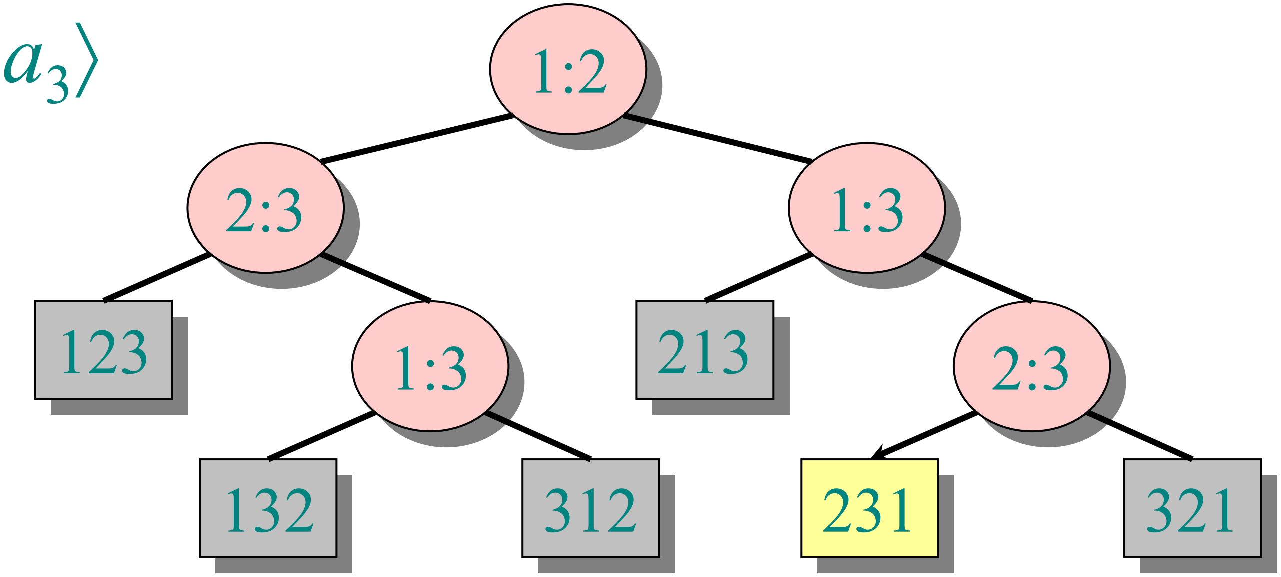


Each internal node is labeled i:j for $i, j \in \{1, 2,\ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



$4 \leq 6 \leq 9$

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm $=$ the length of the path taken.
- Worst-case running time $=$ height of tree.

Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\ge n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\le 2^h$ leaves. Thus, $n! \le 2^h$.

$$\begin{aligned}
\therefore\ h &\ge \lg(n!) && \text{(lg is mono. increasing)}\\
&\ge \lg((n/e)^n) && \text{(Stirling's formula)}\\
&= n \lg n - n \lg e\\
&= \Omega(n \lg n)\,. \quad \square
\end{aligned}$$

Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. □

Sorting in linear time

Counting sort: No comparisons between elements.

- ***Input***: $A[1 \,.\,.\, n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \,.\,.\, n]$, sorted.
- ***Auxiliary storage***: $C[1 \,.\,.\, k]$.

Counting sort

```
for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$          ▷ $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$           ▷ $C[i] = |\{\text{key} \le i\}|$
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Loop 1

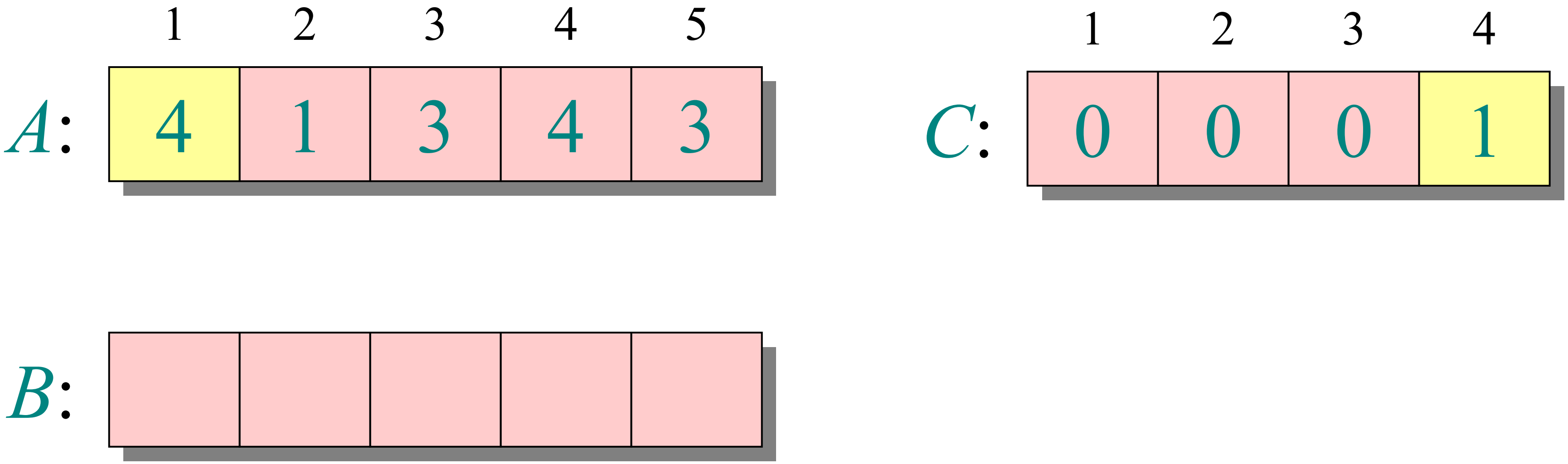
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

```
for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$
```

Loop 2



A:

1	2	3	4	5
4	1	3	4	3

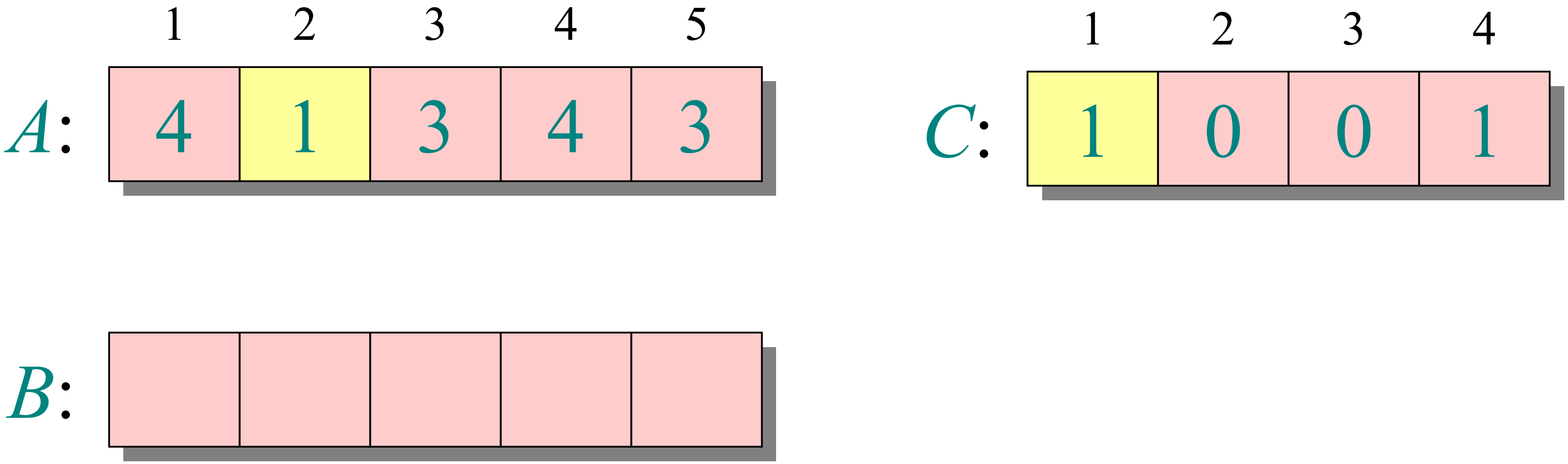
C:

1	2	3	4
0	0	0	1

B:


```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$    $\triangleright$ $C[i] = |\{\text{key} = i\}|$
```

Loop 2



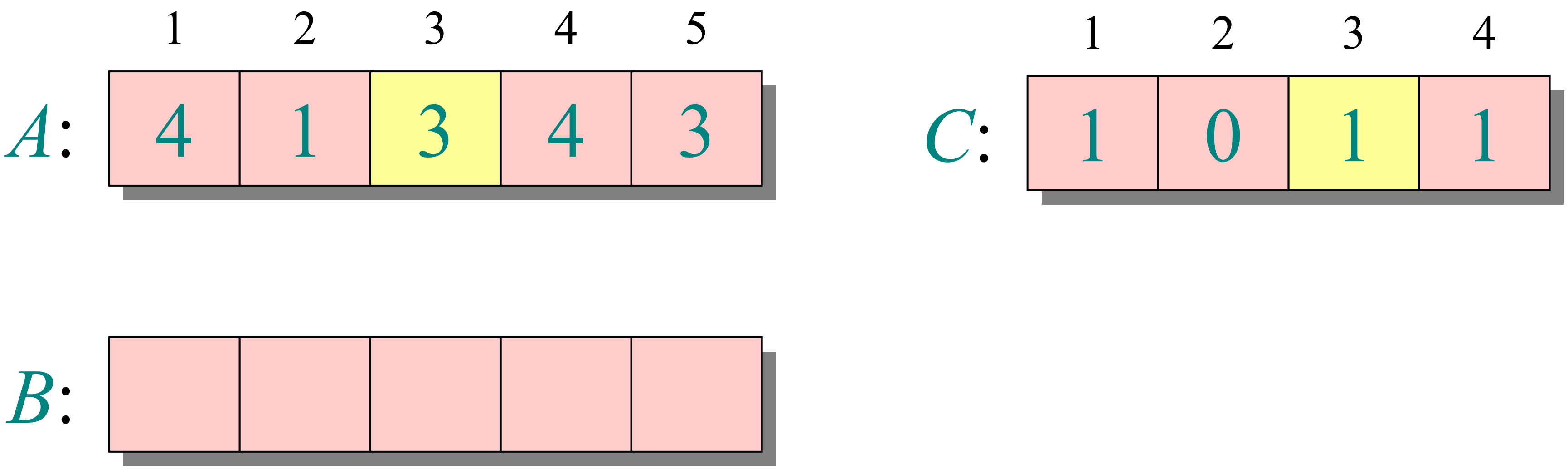
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

B:					

```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$    $\triangleright$ $C[i] = |\{\text{key} = i\}|$
```

Loop 2

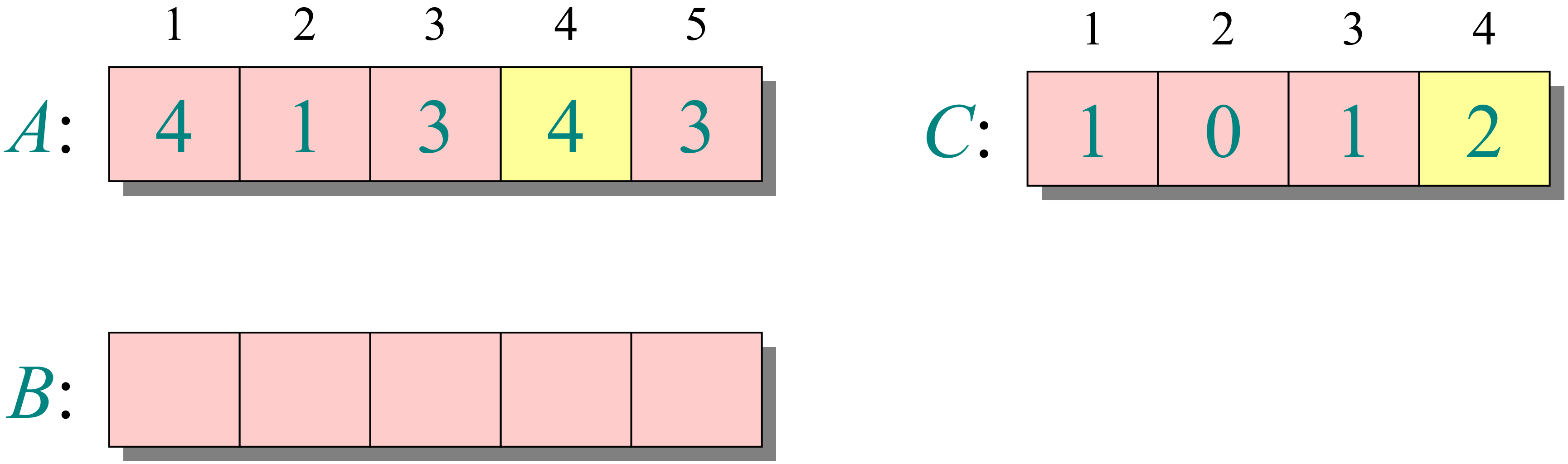


	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	1	1

```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$    $\triangleright$ $C[i] = |\{\text{key} = i\}|$
```

Loop 2



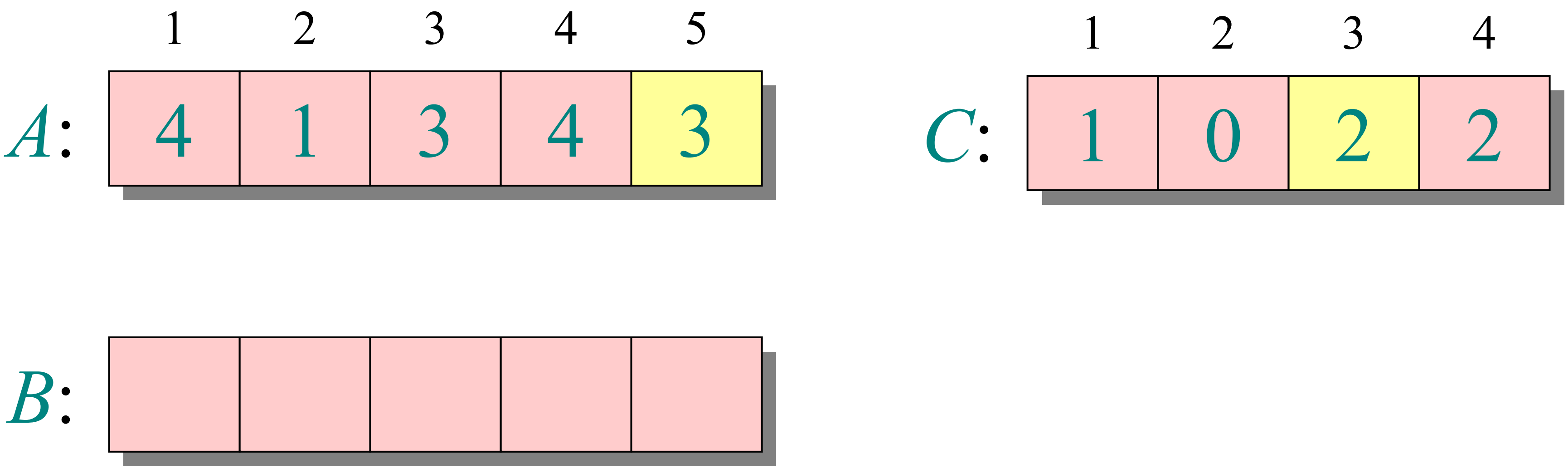
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	2

B:					

```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$    $\triangleright$ $C[i] = |\{\text{key} = i\}|$
```

Loop 2



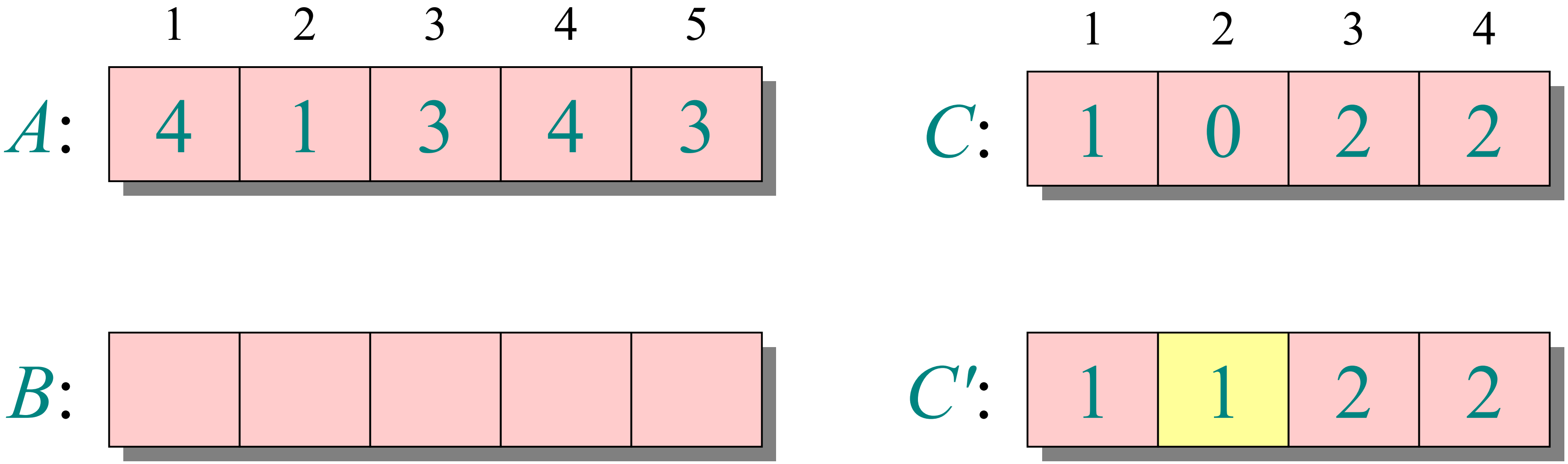
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					

```
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$    $\triangleright\ C[i] = |\{\text{key} = i\}|$
```

Loop 3

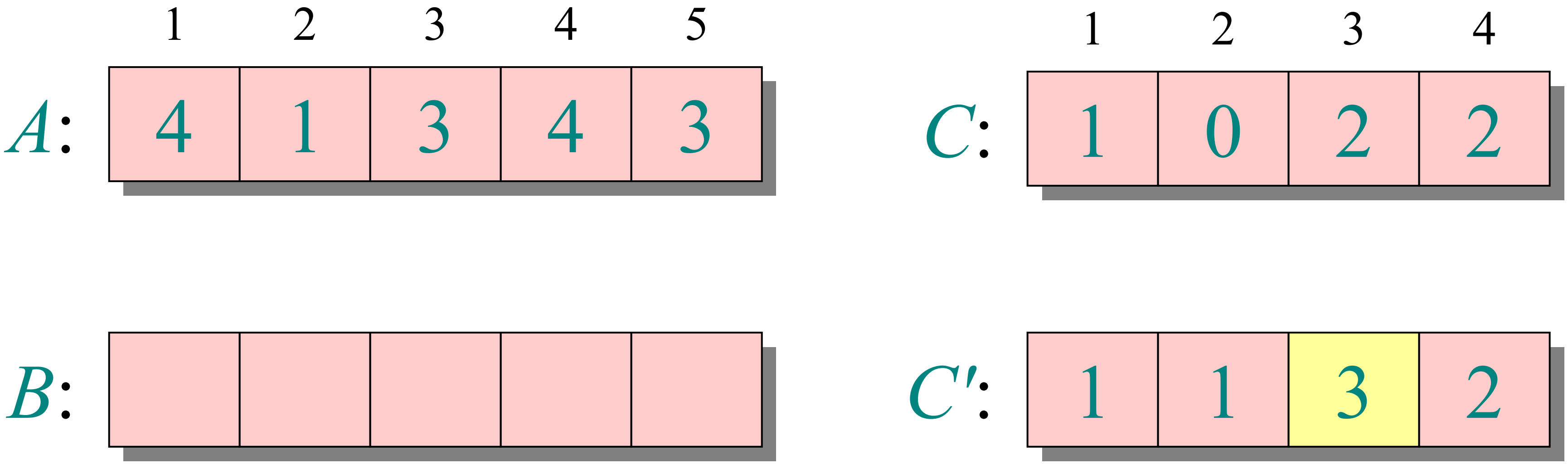


	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright\ C[i] = |\{\text{key} \leq i\}|$
```

Loop 3



A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

B:

C':

1	1	3	2

```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        $\triangleright$ $C[i] = |\{\text{key} \le i\}|$
```

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3

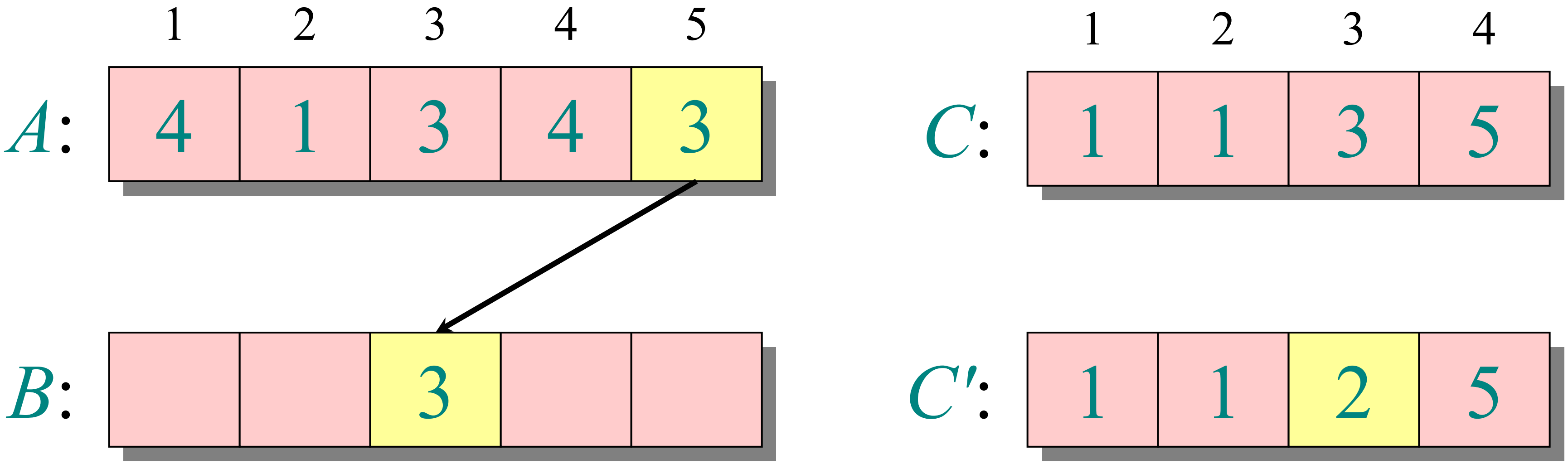
	1	2	3	4
C:	1	0	2	2

B:					

C':	1	1	3	5

```
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$        ▷ $C[i] = |\{\text{key} \leq i\}|$
```

Loop 4

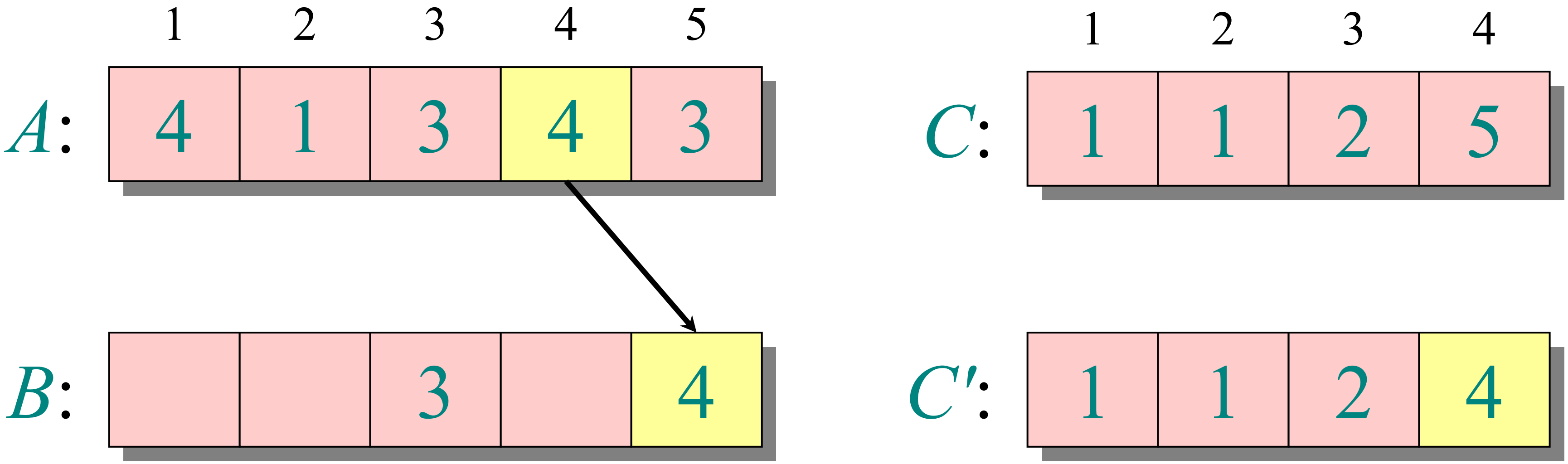


	1	2	3	4	5
A:	4	1	3	4	3
B:			3		

	1	2	3	4
C:	1	1	3	5
C':	1	1	2	5

```
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4

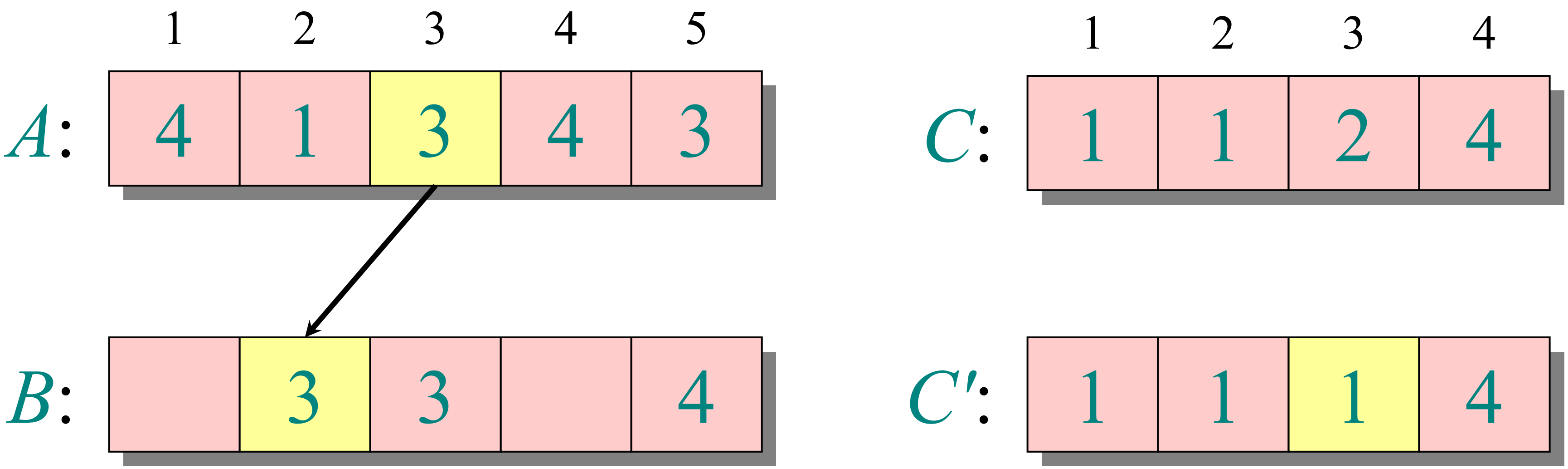


	1	2	3	4	5
A:	4	1	3	4	3
B:			3		4

	1	2	3	4
C:	1	1	2	5
C′:	1	1	2	4

```
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4

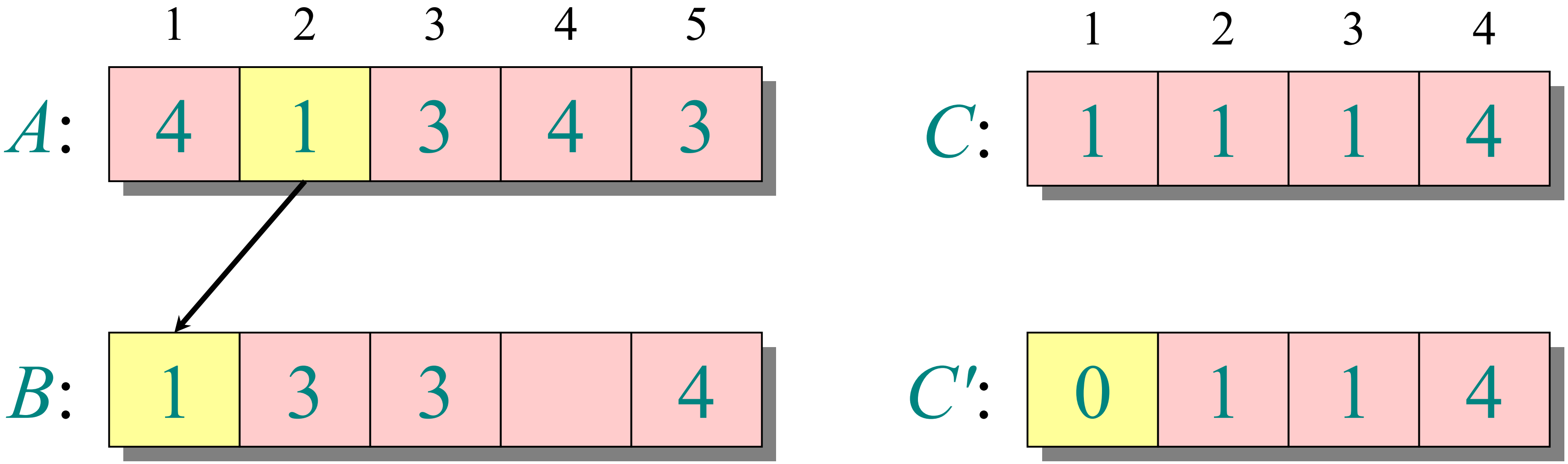


	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	2	4
C':	1	1	1	4

```
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4

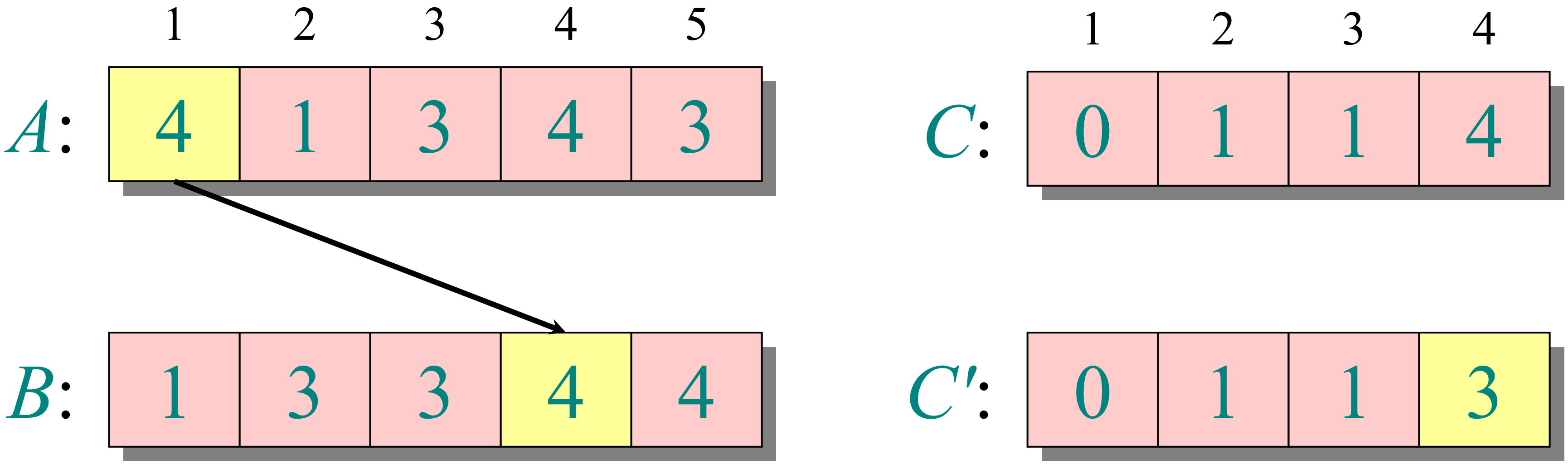


	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	4

```
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4



	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C′:	0	1	1	3

```
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$
```

Analysis

```
$\Theta(k)$    for $i \leftarrow 1$ to $k$
                   do $C[i] \leftarrow 0$
$\Theta(n)$    for $j \leftarrow 1$ to $n$
                   do $C[A[j]] \leftarrow C[A[j]] + 1$
$\Theta(k)$    for $i \leftarrow 2$ to $k$
                   do $C[i] \leftarrow C[i] + C[i–1]$
$\Theta(n)$    for $j \leftarrow n$ downto $1$
                   do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
                      $C[A[j]] \leftarrow C[A[j]] – 1$
```

$$\Theta(n + k)$$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

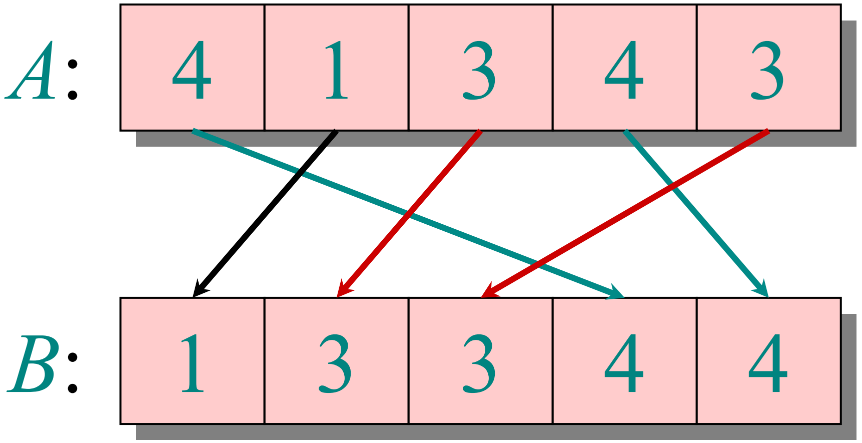
- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.

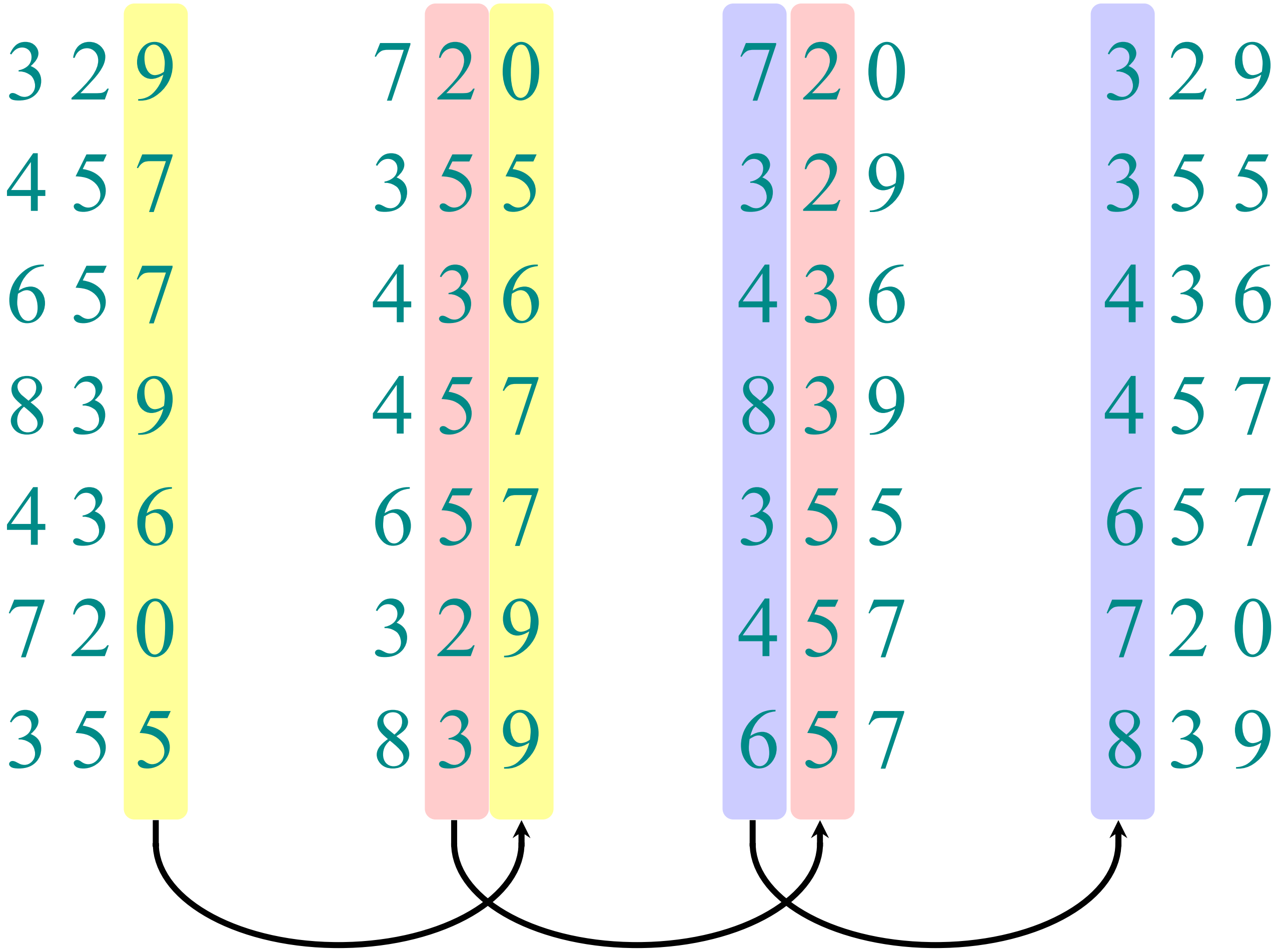


Exercise: What other sorts have this property?

Radix sort

- ***Origin***: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix ℹ.)
- Digit-by-digit sort.
- Hollerith’s original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

Operation of radix sort

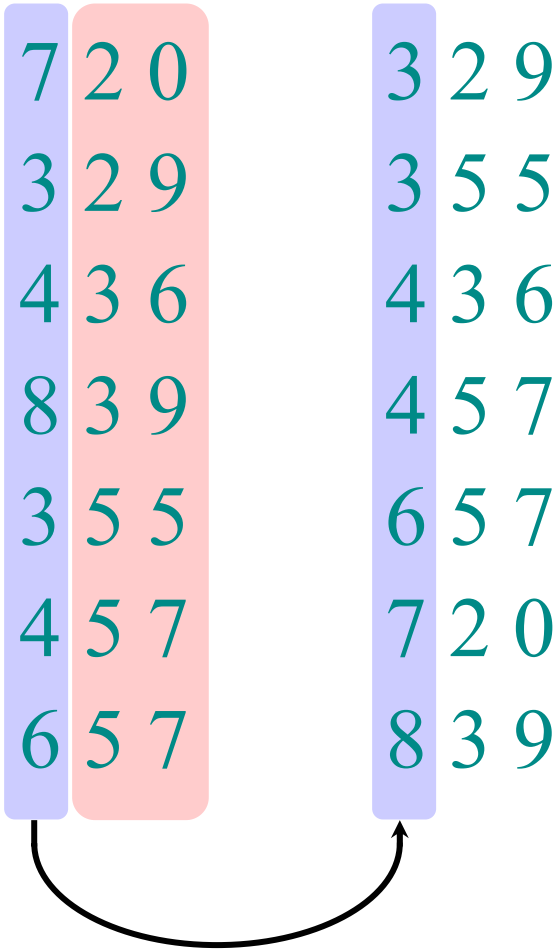


3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



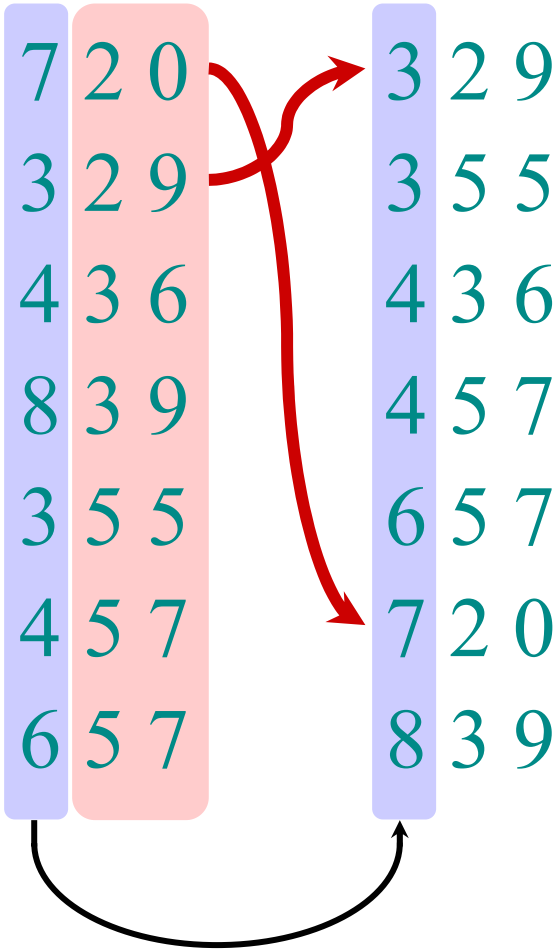
7 2 0 → 3 2 9
3 2 9 → 3 5 5
4 3 6 → 4 3 6
8 3 9 → 4 5 7
3 5 5 → 6 5 7
4 5 7 → 7 2 0
6 5 7 → 8 3 9

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.

- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7

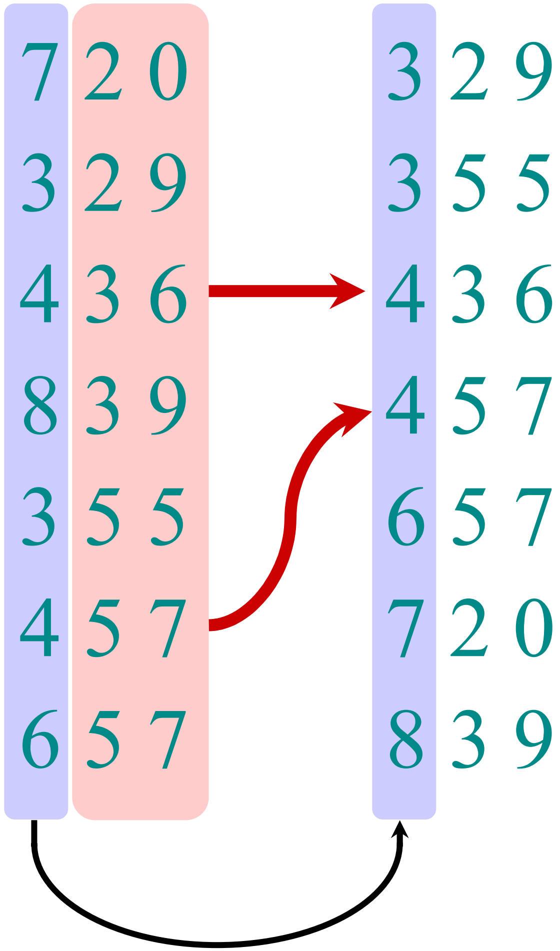
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9

Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.

- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input ⇒ correct order.



7 2 0 3 2 9

3 2 9 3 5 5

4 3 6 4 3 6

8 3 9 4 5 7

3 5 5 6 5 7

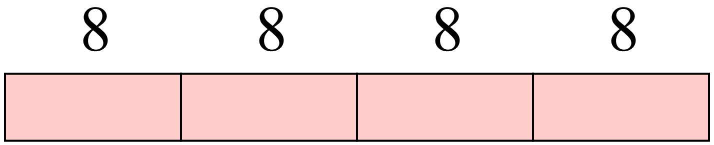
4 5 7 7 2 0

6 5 7 8 3 9

Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.

Choosing r

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(d\,n)$ time.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil =$ 11 passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith's tabulating system
- Operation of the sorter
- Origin of radix sort
- "Modern" IBM card
- Web resources on punched-card technology

Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.

Replica of punch
card from the
1900 U.S. census:
[Howells 2000]

Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

See figure from [Howells 2000].

Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.

Origin of radix sort

Holleriths original 1889 patent alludes to a most-significant-digit-first radix sort:

> *"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."*

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

“Modern” IBM card

- One character per column.

See examples on the WWW Virtual Punch-Card Server.

.

So, that’s why text windows have 80 columns!

Web resources on punched-card technology

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Hollerith's patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 6

Prof. Erik Demaine

Order statistics

Select the ith smallest of n elements (the element with ***rank*** i).

- $i = 1$: ***minimum***;
- $i = n$: ***maximum***;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: ***median***.

Naive algorithm: Sort and index ith element.

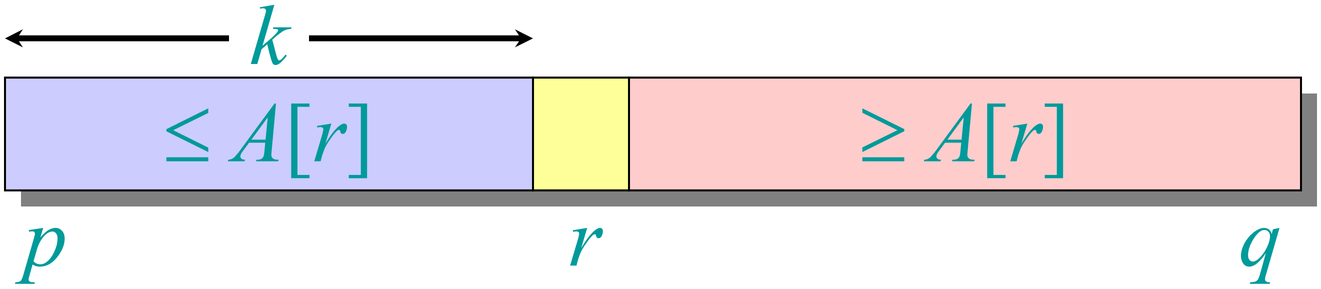
Worst-case running time $= \Theta(n \lg n) + \Theta(1)$

$= \Theta(n \lg n)$,

using merge sort or heapsort (*not* quicksort).

Randomized divide-and-conquer algorithm

```
RAND-SELECT$(A, p, q, i)$        ▷ $i$th smallest of $A[p\,..\,q]$
    if $p = q$ then return $A[p]$
    $r \leftarrow$ RAND-PARTITION$(A, p, q)$
    $k \leftarrow r - p + 1$                 ▷ $k = \text{rank}(A[r])$
    if $i = k$ then return $A[r]$
    if $i < k$
        then return RAND-SELECT$(A, p, r - 1, i)$
        else return RAND-SELECT$(A, r + 1, q, i - k)$
```



Example

Select the $i = 7$th smallest:

6	10	13	5	8	3	2	11

$i = 7$

pivot

Partition:

2	5	3	6	8	13	10	11

$k = 4$

$$2 \quad 5 \quad 3 \quad 6 \quad \underbrace{8 \quad 13 \quad 10 \quad 11}$$

Select the $7 - 4 = 3$rd smallest recursively.

Intuition for analysis

(All our analyses today assume that all elements are distinct.)

Lucky:

$$T(n) = T(9n/10) + \Theta(n) \qquad n^{\log_{10/9} 1} = n^0 = 1$$

$$= \Theta(n) \qquad \text{Case 3}$$

Unlucky:

$$T(n) = T(n-1) + \Theta(n) \qquad \text{arithmetic series}$$

$$= \Theta(n^2)$$

Worse than sorting!

Analysis of expected time

The analysis follows that of randomized quicksort, but it's a little different.

Let $T(n)$ = the random variable for the running time of RAND-SELECT on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n\text{–}1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n\text{–}k\text{–}1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

Analysis (continued)

To obtain an upper bound, assume that the ith element always falls in the larger side of the partition:

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \quad\vdots & \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \big(T(\max\{k, n-k-1\}) + \Theta(n)\big).$$

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \bigl(T(\max\{k, n-k-1\}) + \Theta(n)\bigr)\right]$$

Take expectations of both sides.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

Linearity of expectation.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]$$

Independence of X_k from other random choices.

Calculating expectation

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k\bigl(T(\max\{k, n-k-1\}) + \Theta(n)\bigr)\right] \\
&= \sum_{k=0}^{n-1} E\bigl[X_k\bigl(T(\max\{k, n-k-1\}) + \Theta(n)\bigr)\bigr] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)
\end{aligned}
$$

Linearity of expectation; $E[X_k] = 1/n$.

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

Upper terms appear twice.

Hairy recurrence

(But not quite as hairy as the quicksort one.)

$$E[T(n)] = \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

Prove: $E[T(n)] \le cn$ for constant $c > 0$.

- The constant c can be chosen large enough so that $E[T(n)] \le cn$ for the base cases.

Use fact: $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \le \frac{3}{8} n^2$ (exercise).

Substitution method

$$E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n)$$

Substitute inductive hypothesis.

Substitution method

$$E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n)$$

$$\le \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n)$$

Use fact.

Substitution method

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\
&\le \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n) \\
&= cn - \left(\frac{cn}{4} - \Theta(n)\right)
\end{aligned}
$$

Express as ***desired*** – ***residual***.

Substitution method

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\
&\le \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n) \\
&= cn - \left(\frac{cn}{4} - \Theta(n)\right) \\
&\le cn\,,
\end{aligned}
$$

if c is chosen large enough so that $cn/4$ dominates the $\Theta(n)$.

Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is ***very*** bad: $\Theta(n^2)$.

Q. Is there an algorithm that runs in linear time in the worst case?

A. Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

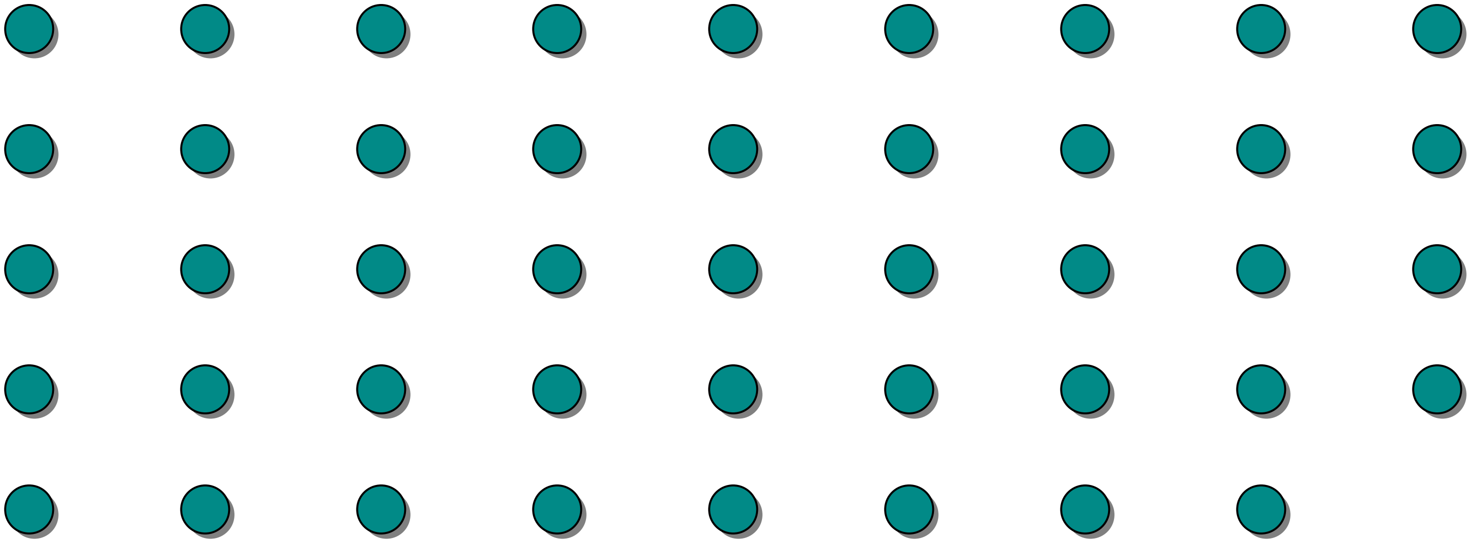
IDEA: Generate a good pivot recursively.

Worst-case linear-time order statistics

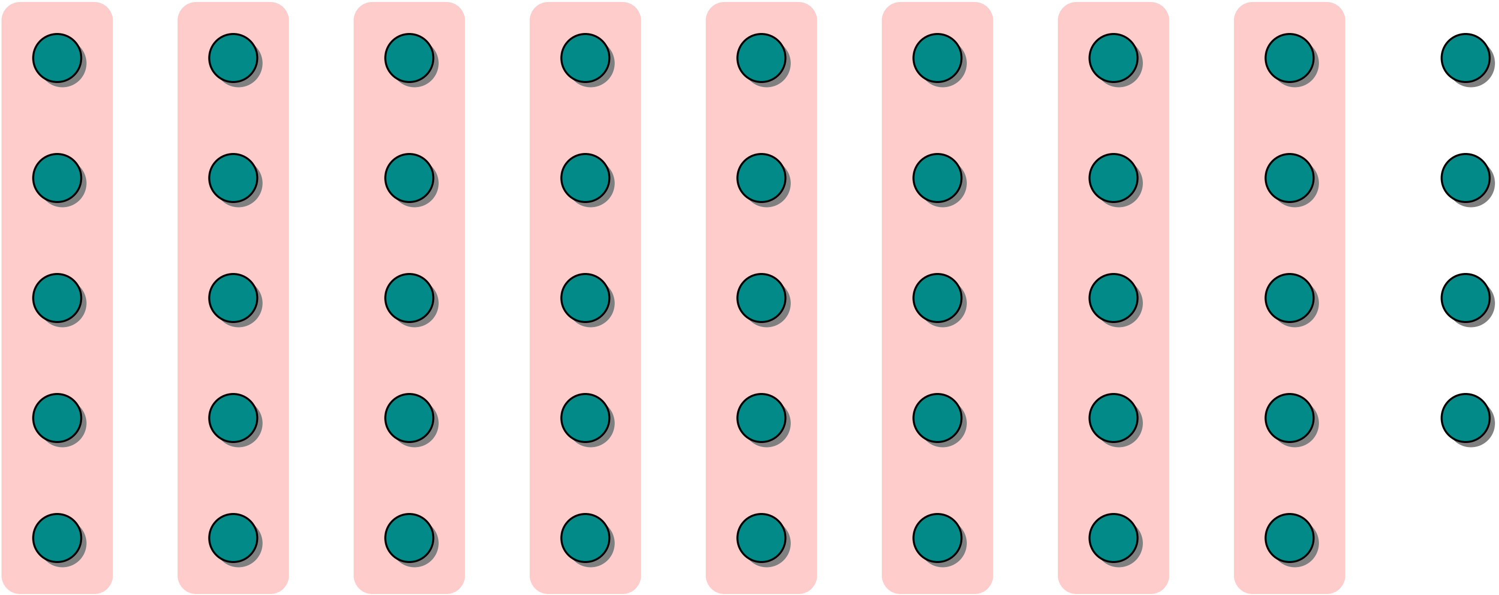
```
SELECT($i$, $n$)
1. Divide the $n$ elements into groups of $5$. Find
   the median of each $5$-element group by rote.
2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$
   group medians to be the pivot.
3. Partition around the pivot $x$. Let $k = \mathrm{rank}(x)$.
4. if $i = k$ then return $x$
   elseif $i < k$
       then recursively SELECT the $i$th
               smallest element in the lower part
       else recursively SELECT the $(i–k)$th
               smallest element in the upper part
```

Same as RAND-SELECT

Choosing the pivot

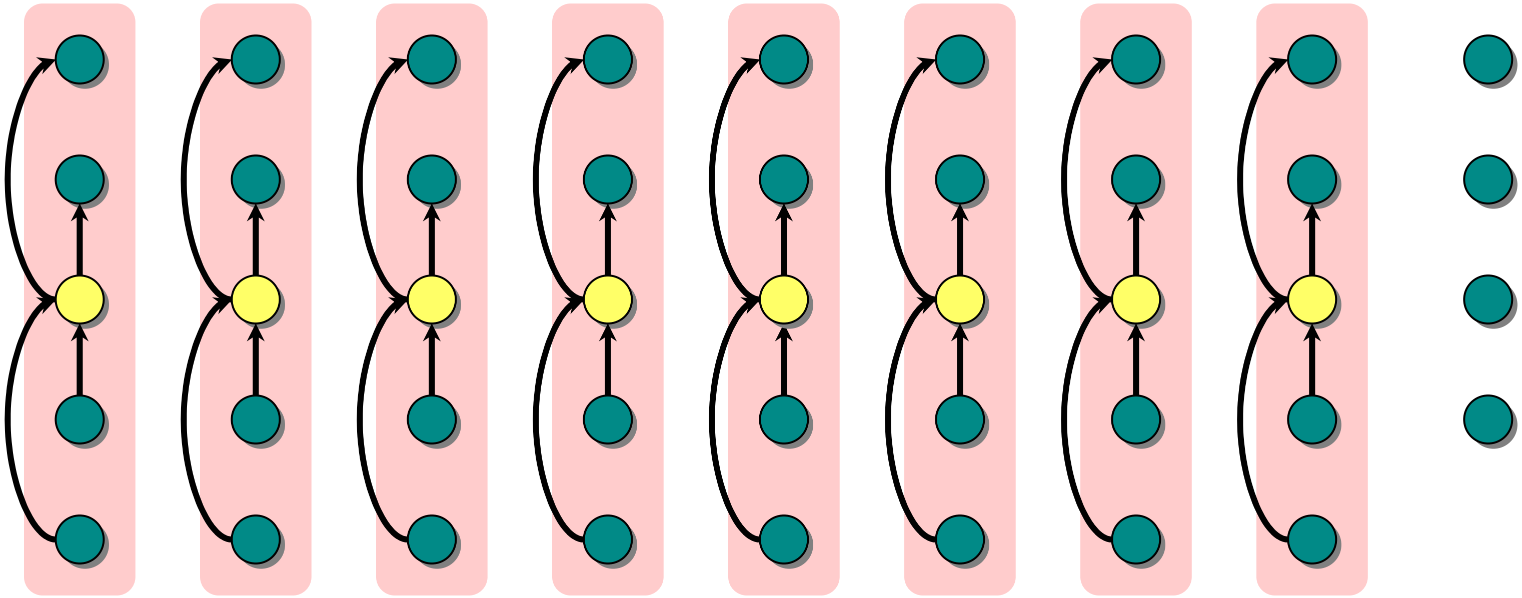


Choosing the pivot

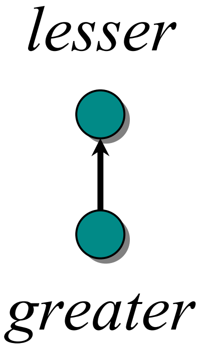


1. Divide the n elements into groups of 5.

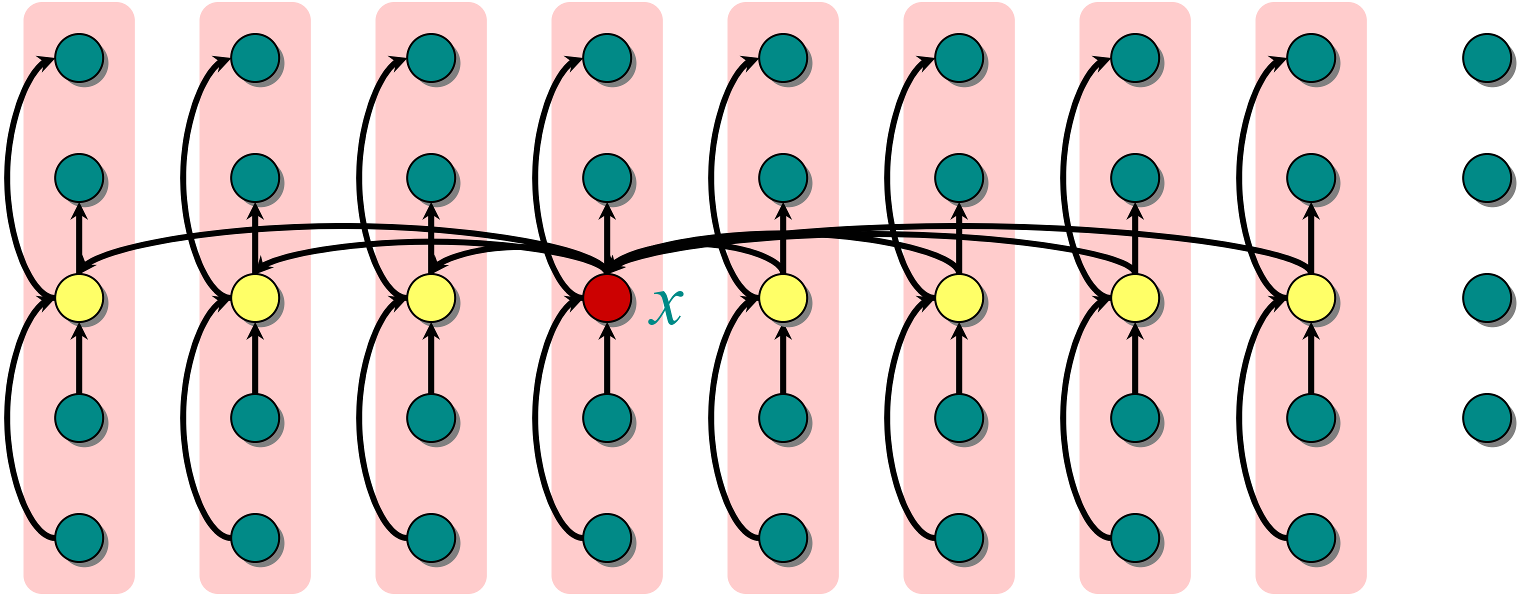
Choosing the pivot



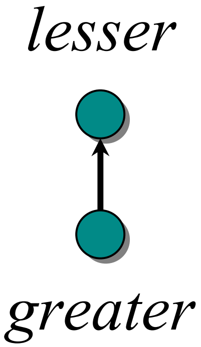
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.



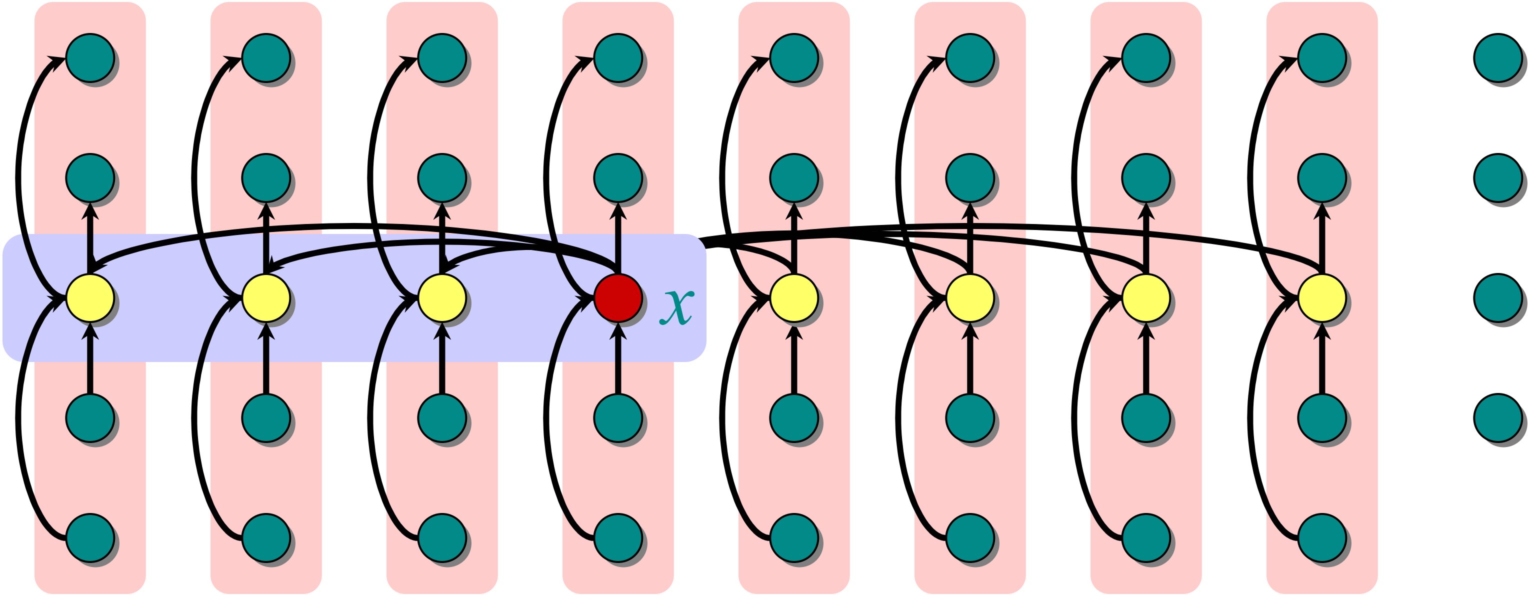
Choosing the pivot



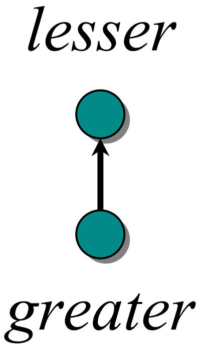
1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.



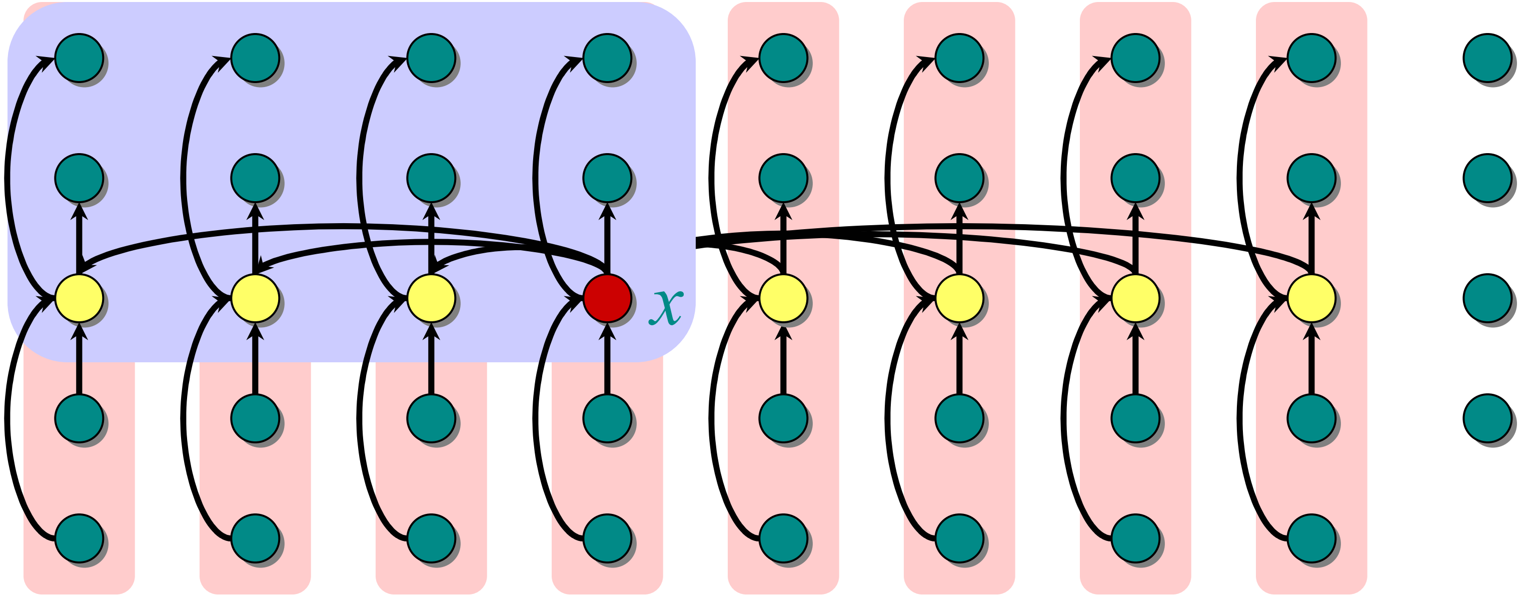
Analysis



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

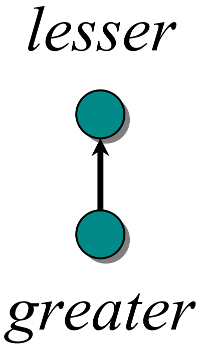


Analysis (Assume all elements are distinct.)

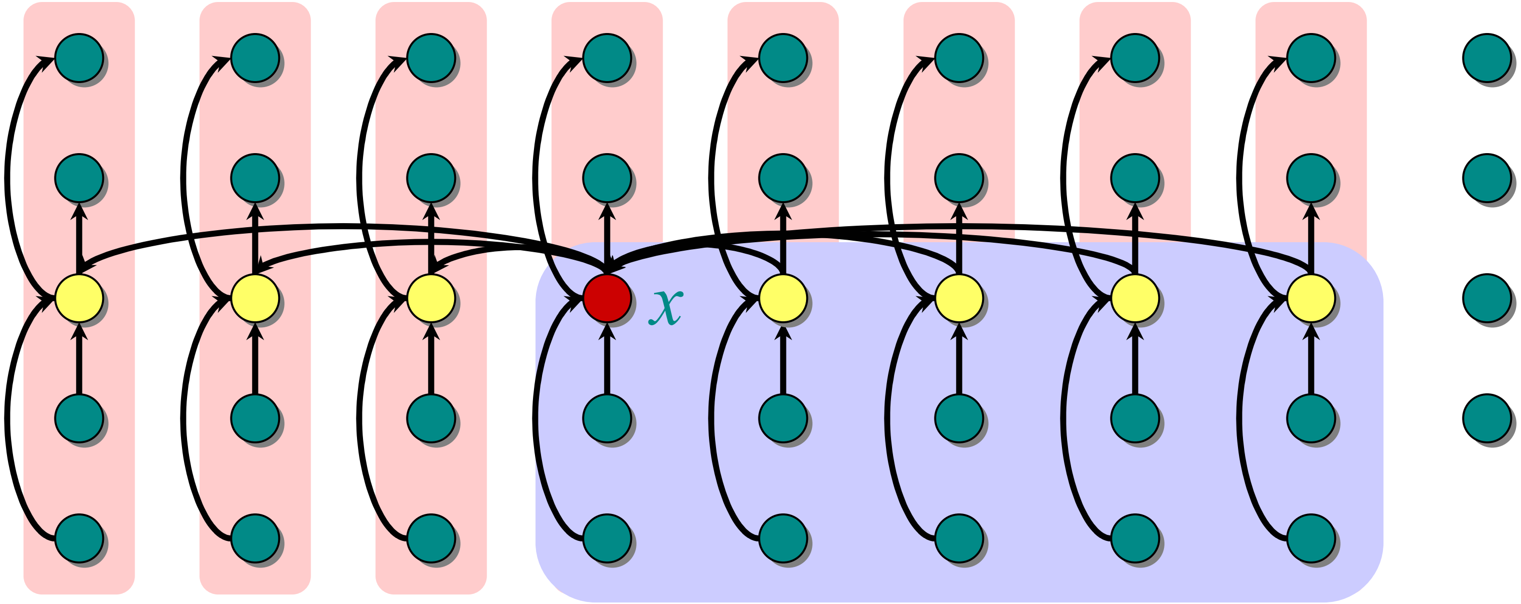


At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\le x$.

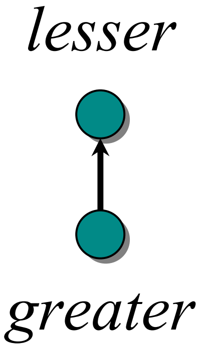


Analysis (Assume all elements are distinct.)



At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\le x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\ge x$.



Minor simplification

- For $n \geq 50$, we have $3\lfloor n/10 \rfloor \geq n/4$.
- Therefore, for $n \geq 50$ the recursive call to SELECT in Step 4 is executed recursively on $\leq 3n/4$ elements.
- Thus, the recurrence for running time can assume that Step 4 takes time $T(3n/4)$ in the worst case.
- For $n < 50$, we know that the worst-case time is $T(n) = \Theta(1)$.

Developing the recurrence

```
$T(n)$         SELECT$(i, n)$
$\Theta(n)$    1. Divide the $n$ elements into groups of $5$.  Find
                  the median of each $5$-element group by rote.
$T(n/5)$       2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$
                  group medians to be the pivot.
$\Theta(n)$    3. Partition around the pivot $x$.  Let $k = \text{rank}(x)$.
$T(3n/4)$      4. if  $i = k$ then return $x$
                  elseif  $i < k$
                      then recursively SELECT the $i$th
                              smallest element in the lower part
                      else  recursively SELECT the $(i–k)$th
                              smallest element in the upper part
```

Solving the recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

Substitution:

$T(n) \le cn$

$$\begin{aligned} T(n) &\le \frac{1}{5}cn + \frac{3}{4}cn + \Theta(n) \\ &= \frac{19}{20}cn + \Theta(n) \\ &= cn - \left(\frac{1}{20}cn - \Theta(n)\right) \\ &\le cn \quad , \end{aligned}$$

if c is chosen large enough to handle both the $\Theta(n)$ and the initial conditions.

Conclusions

- Since the work at each level of recursion is a constant fraction (19/20) smaller, the work per level is a geometric series dominated by the linear work at the root.
- In practice, this algorithm runs slowly, because the constant in front of n is large.
- The randomized algorithm is far more practical.

Exercise: *Why not divide into groups of 3?*

Introduction to Algorithms

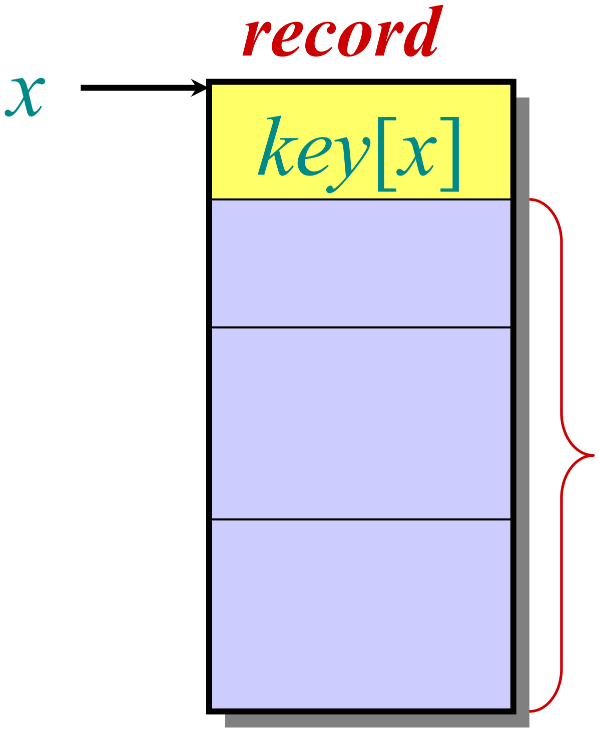
6.046J/18.401J/SMA5503

Lecture 7

Prof. Charles E. Leiserson

Symbol-table problem

Symbol table T holding n ***records***:



Other ***fields*** containing ***satellite data***

Operations on $\boldsymbol{T}$:

- INSERT(T, x)
- DELETE(T, x)
- SEARCH(T, k)

How should the data structure T be organized?

Direct-access table

IDEA: Suppose that the set of keys is $K \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \,..\, m-1]$:

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

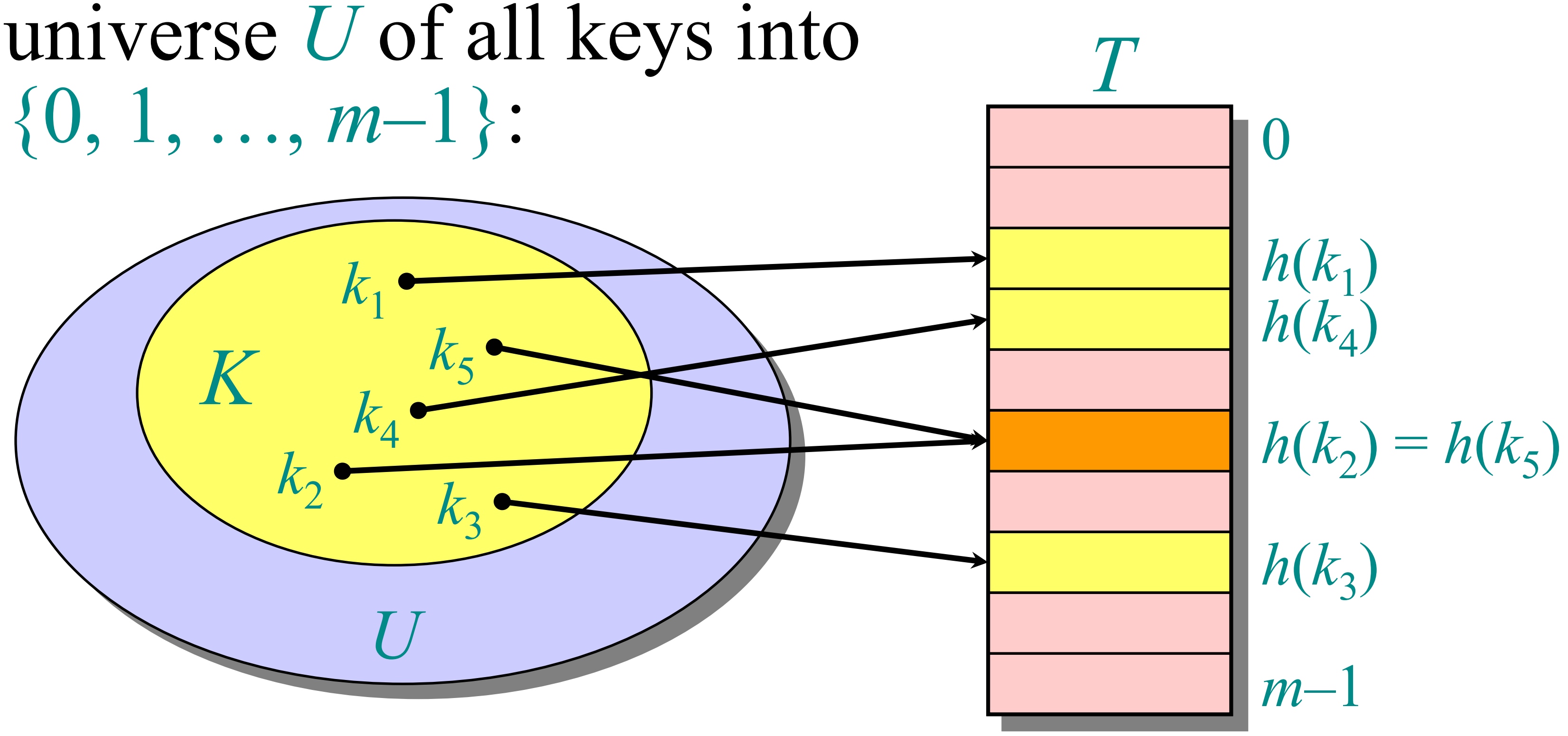
Then, operations take $\Theta(1)$ time.

Problem: The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- character strings (even larger!).

Hash functions

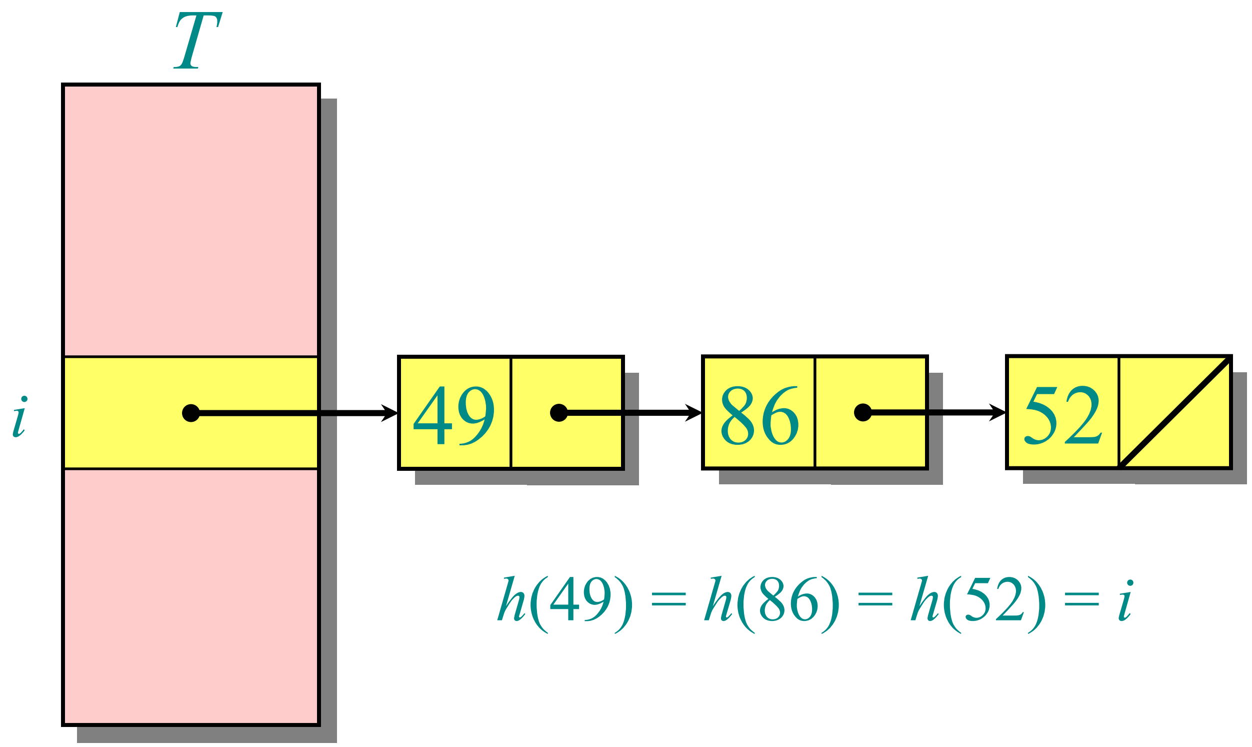
Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, m–1\}$:



When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs.

Resolving collisions by chaining

- Records in the same slot are linked into a list.



$$h(49) = h(86) = h(52) = i$$

Analysis of chaining

We make the assumption of ***simple uniform hashing:***

- Each key $k \in K$ of keys is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\alpha = n/m$$

$=$ average number of keys per slot.

Search cost

Expected time to search for a record with a given key $= \Theta(1 + \alpha)$.

apply hash function and access slot

search the list

Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

Division method

Assume all keys are integers, and define

$$h(k) = k \bmod m.$$

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:

- If $k = 1011000111\underbrace{011010}_{h(k)}{}_2$ and $r = 6$, then $h(k) = 011010_2$.

Division method (continued)

$$h(k) = k \bmod m.$$

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.

Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where rsh is the “bit-wise right-shift” operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

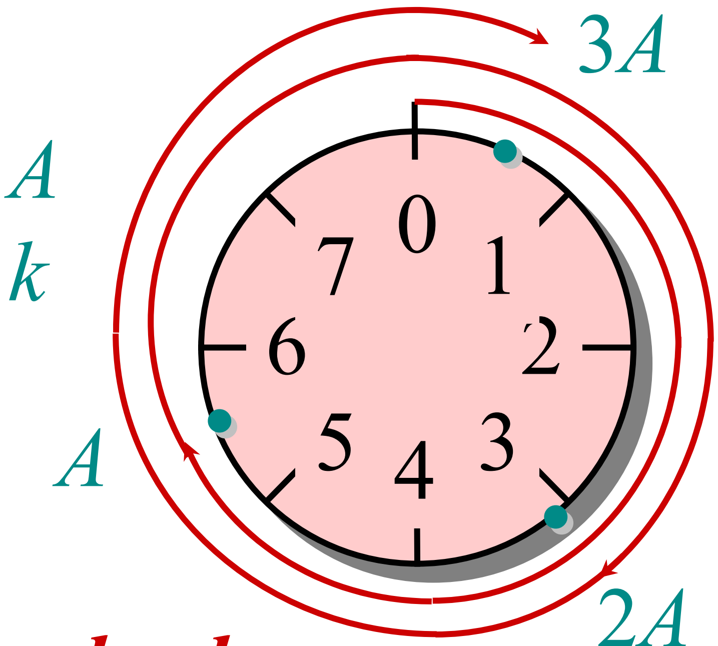
- Don’t pick A too close to 2^w.
- Multiplication modulo 2^w is fast.
- The rsh operator is fast.

Multiplication method example

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$\begin{array}{lr}
 & 1\,0\,1\,1\,0\,0\,1 = A \\
\times & 1\,1\,0\,1\,0\,1\,1 = k \\
\hline
 & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1
\end{array}$$



Modular wheel

Dot-product method

Randomized strategy:

Let m be prime. Decompose key k into $r + 1$ digits, each with value in the set $\{0, 1, \ldots, m-1\}$. That is, let $k = \langle k_0, k_1, \ldots, k_{m-1} \rangle$, where $0 \le k_i < m$.

Pick $a = \langle a_0, a_1, \ldots, a_{m-1} \rangle$ where each a_i is chosen randomly from $\{0, 1, \ldots, m-1\}$.

Define $h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$.

- Excellent in practice, but expensive to compute.

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

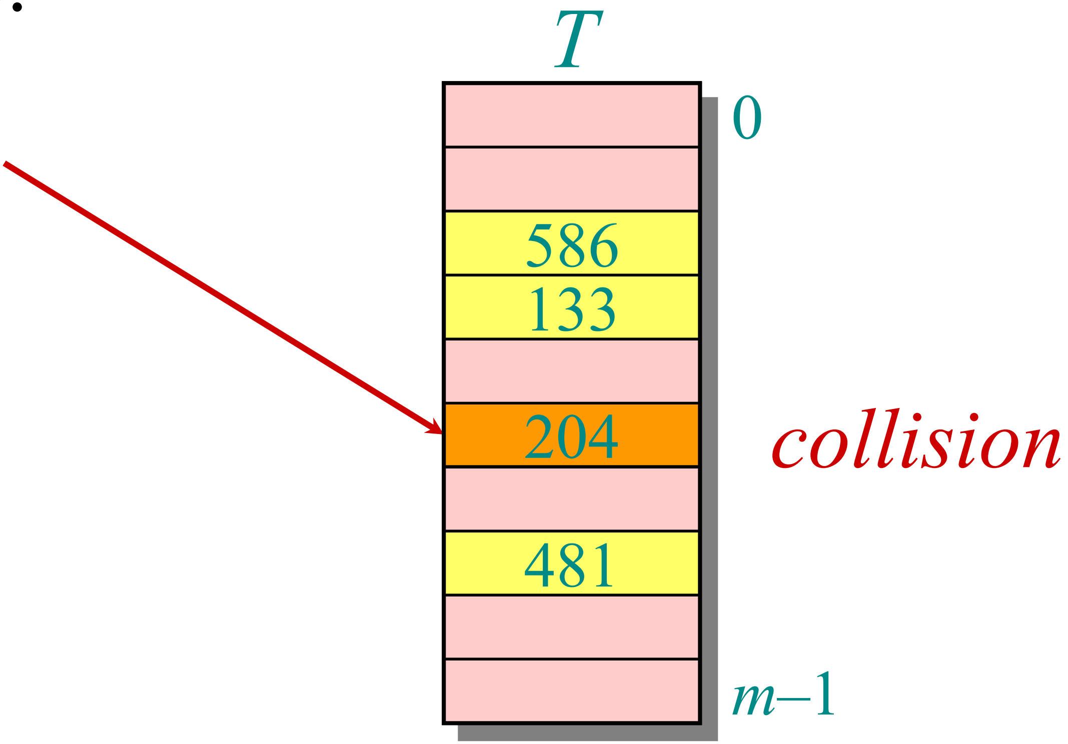
  $$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible).

Example of open addressing

Insert key $k = 496$:

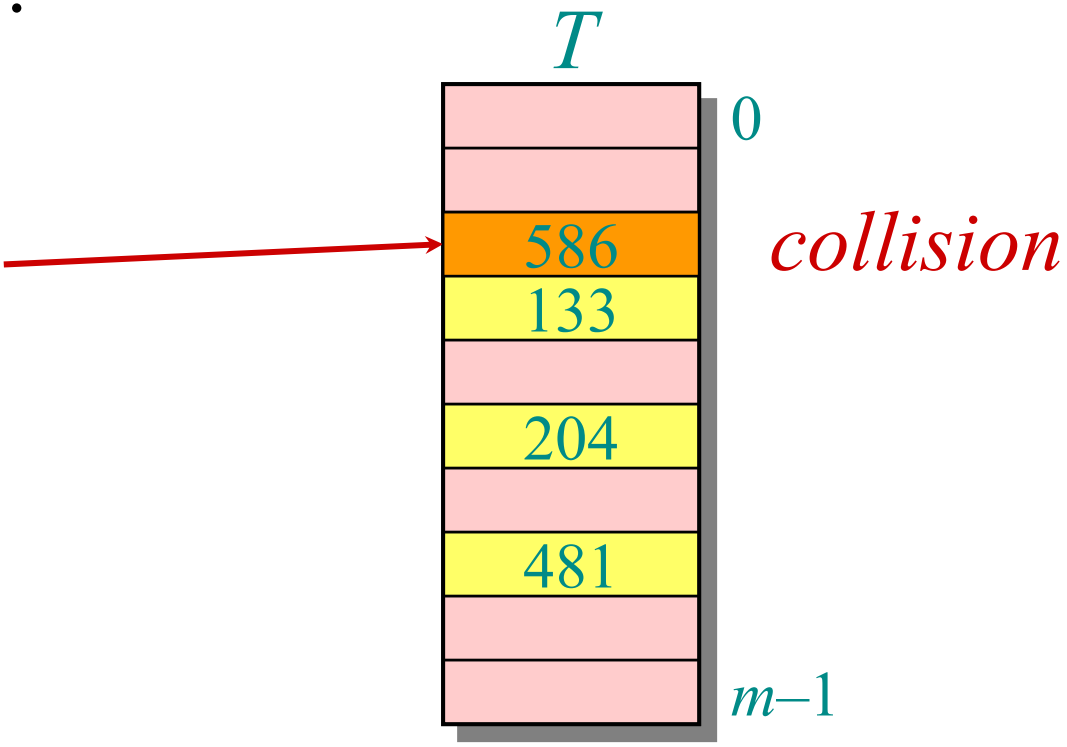
0. Probe $h(496,0)$



Example of open addressing

Insert key $k = 496$:

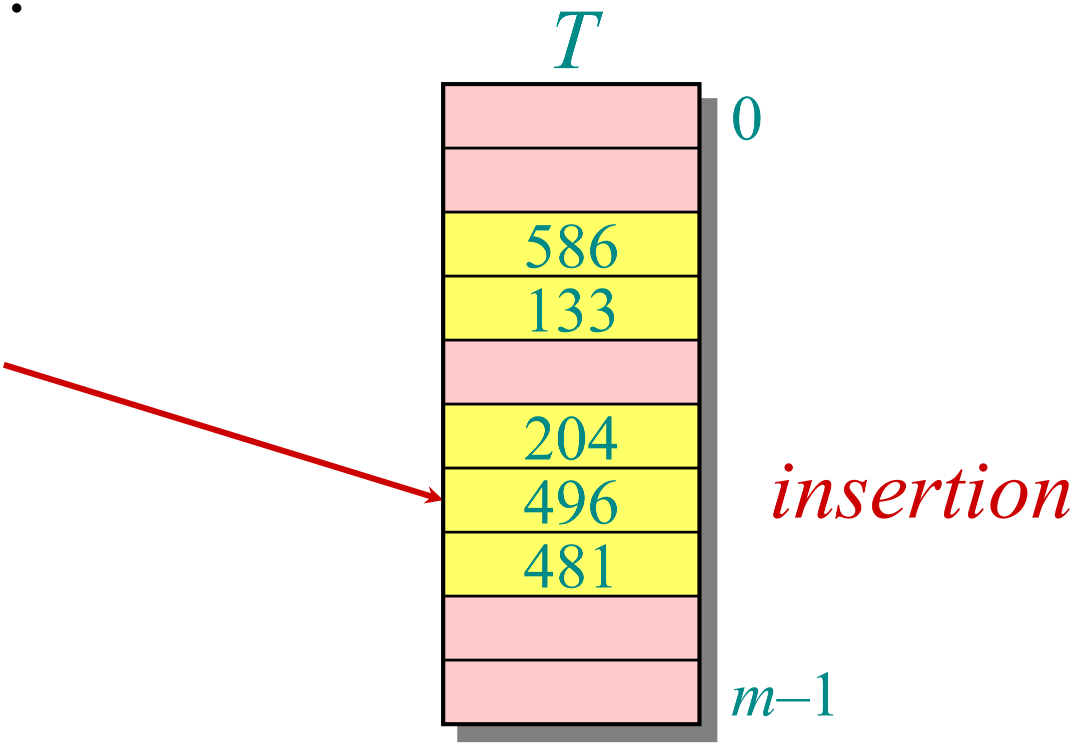
0. Probe $h(496,0)$
1. Probe $h(496,1)$



Example of open addressing

Insert key $k = 496$:

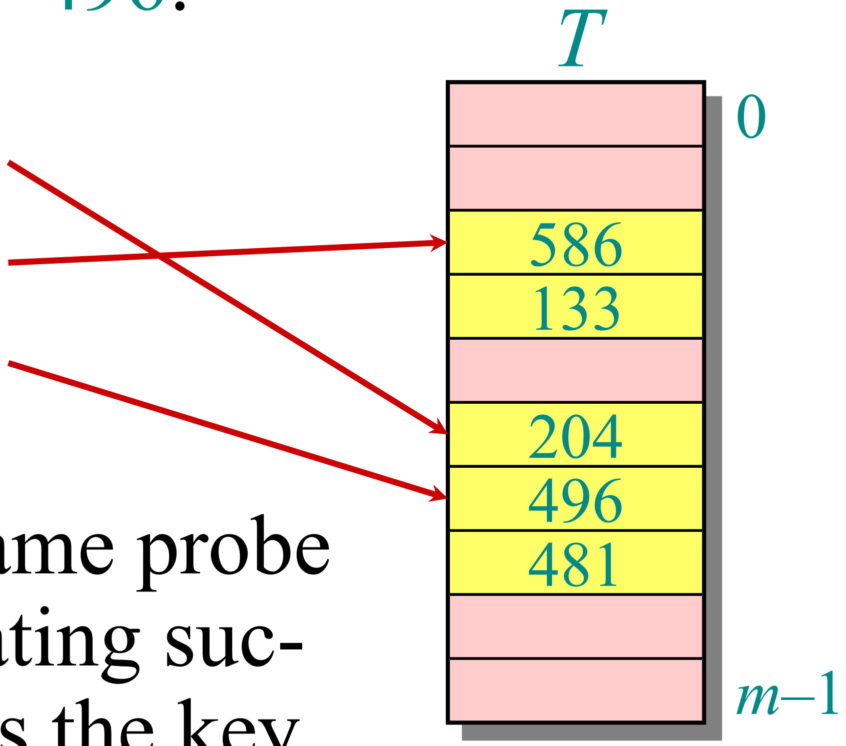
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m.$$

This method, though simple, suffers from ***primary clustering***, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

Probing strategies

Double hashing

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of open addressing

We make the assumption of ***uniform hashing:***

- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1–\alpha)$.

Proof of the theorem

Proof.

- At least one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n–1)/(m–1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n–2)/(m–2)$, the third probe hits an occupied slot, etc.

Observe that $\dfrac{n-i}{m-i} < \dfrac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.

Proof (continued)

Therefore, the expected number of probes is

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

$$\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))$$

$$\le 1+\alpha+\alpha^2+\alpha^3+\cdots$$

$$=\sum_{i=0}^{\infty}\alpha^i$$

$$=\frac{1}{1-\alpha}\,.\;\square$$

The textbook has a more rigorous proof.

Implications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1–0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1–0.9) = 10$.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 8

Prof. Charles E. Leiserson

A weakness of hashing

Problem: For any hash function h, a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from $\{k \in U : h(k) = i\}$ for some slot i.

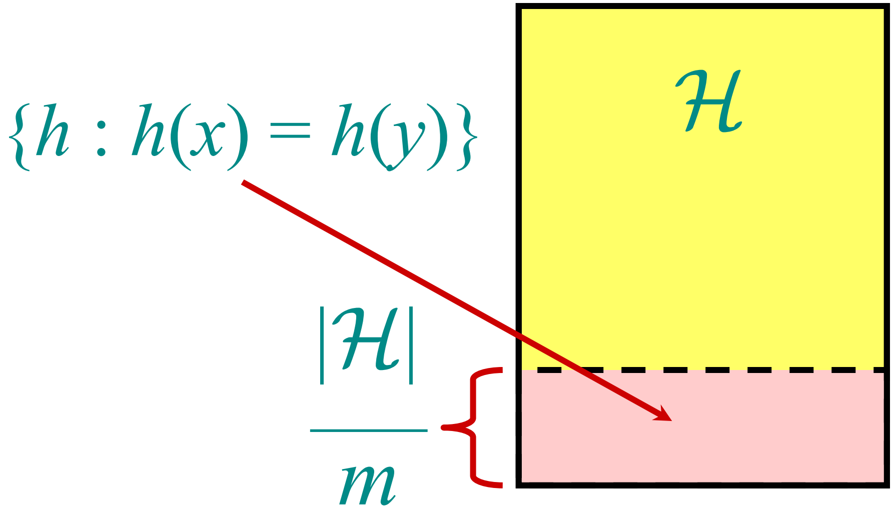
Idea: Choose the hash function at random, independently of the keys.

- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.

Universal hashing

Definition. Let U be a universe of keys, and let $\mathcal{H}$ be a finite collection of hash functions, each mapping U to $\{0, 1, \ldots, m-1\}$. We say $\mathcal{H}$ is ***universal*** if for all $x, y \in U$, where $x \neq y$, we have $|\{h \in \mathcal{H} : h(x) = h(y)\}| = |\mathcal{H}|/m$.

That is, the chance of a collision between x and y is $1/m$ if we choose h randomly from $\mathcal{H}$.



Universality is good

Theorem. Let h be a hash function chosen (uniformly) at random from a universal set $\mathcal{H}$ of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T. Then, for a given key x, we have

$$E[\#\text{collisions with } x] < n/m.$$

Proof of theorem

Proof. Let C_x be the random variable denoting the total number of collisions of keys in T with x, and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Note: $E[c_{xy}] = 1/m$ and $C_x = \sum_{y \in T - \{x\}} c_{xy}$.

Proof (continued)

$$E[C_x] = E\left[\sum_{y \in T - \{x\}} c_{xy}\right]$$

- Take expectation of both sides.

Proof (continued)

$$E[C_x] = E\left[\sum_{y \in T - \{x\}} c_{xy}\right]$$

$$= \sum_{y \in T - \{x\}} E[c_{xy}]$$

- Take expectation of both sides.
- Linearity of expectation.

Proof (continued)

$$E[C_x] = E\left[\sum_{y \in T - \{x\}} c_{xy}\right]$$

- Take expectation of both sides.

$$= \sum_{y \in T - \{x\}} E[c_{xy}]$$

- Linearity of expectation.

$$= \sum_{y \in T - \{x\}} 1/m$$

- $E[c_{xy}] = 1/m$.

Proof (continued)

$$E[C_x] = E\left[\sum_{y \in T-\{x\}} c_{xy}\right]$$

- Take expectation of both sides.

$$= \sum_{y \in T-\{x\}} E[c_{xy}]$$

- Linearity of expectation.

$$= \sum_{y \in T-\{x\}} 1/m$$

- $E[c_{xy}] = 1/m$.

$$= \frac{n-1}{m} . \quad \square$$

- Algebra.

Constructing a set of universal hash functions

Let m be prime. Decompose key k into $r + 1$ digits, each with value in the set $\{0, 1, \ldots, m{-}1\}$. That is, let $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

Randomized strategy:

Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0, 1, \ldots, m{-}1\}$.

Define $h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$.

Dot product, modulo m

How big is $\mathcal{H} = \{h_a\}$? $|\mathcal{H}| = m^{r+1}$. ← **REMEMBER THIS!**

Universality of dot-product hash functions

Theorem. The set $\mathcal{H} = \{h_a\}$ is universal.

Proof. Suppose that $x = \langle x_0, x_1, \ldots, x_r \rangle$ and $y = \langle y_0, y_1, \ldots, y_r \rangle$ be distinct keys. Thus, they differ in at least one digit position, wlog position 0. For how many $h_a \in \mathcal{H}$ do x and y collide?

We must have $h_a(x) = h_a(y)$, which implies that

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod{m} .$$

Proof (continued)

Equivalently, we have

$$\sum_{i=0}^{r} a_i(x_i - y_i) \equiv 0 \quad (\operatorname{mod} m)$$

or

$$a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \equiv 0 \quad (\operatorname{mod} m) \ ,$$

which implies that

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \quad (\operatorname{mod} m) \ .$$

Fact from number theory

Theorem. Let m be prime. For any $z \in \mathbb{Z}_m$ such that $z \neq 0$, there exists a unique $z^{-1} \in \mathbb{Z}_m$ such that

$$z \cdot z^{-1} \equiv 1 \pmod{m}.$$

Example: $m = 7$.

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

Back to the proof

We have

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m},$$

and since $x_0 \neq y_0$, an inverse $(x_0 - y_0)^{-1}$ must exist, which implies that

$$a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right) \cdot (x_0 - y_0)^{-1} \pmod{m}.$$

Thus, for any choices of $a_1, a_2, \ldots, a_r$, exactly one choice of a_0 causes x and y to collide.

Proof (completed)

Q. How many h_a's cause x and y to collide?

A. There are m choices for each of $a_1, a_2, \ldots, a_r$, but once these are chosen, exactly one choice for a_0 causes x and y to collide, namely

$$a_0 = \left(\left(-\sum_{i=1}^{r} a_i (x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \right) \bmod m .$$

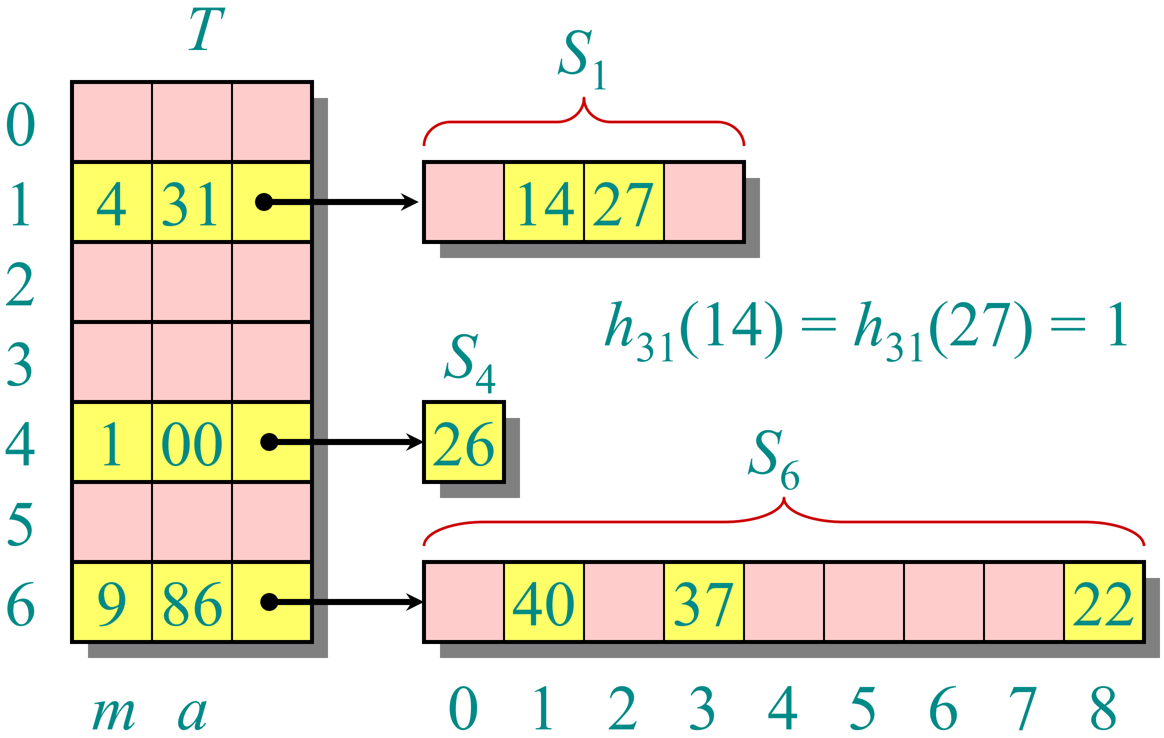
Thus, the number of h_a's that cause x and y to collide is $m^r \cdot 1 = m^r = |\mathcal{H}|/m$. □

Perfect hashing

Given a set of n keys, construct a static hash table of size $m = O(n)$ such that SEARCH takes $\Theta(1)$ time in the *worst case*.

IDEA: Two-level scheme with universal hashing at both levels.

No collisions at level 2!



$h_{31}(14) = h_{31}(27) = 1$

Collisions at level 2

Theorem. Let $\mathcal{H}$ be a class of universal hash functions for a table of size $m = n^2$. Then, if we use a random $h \in \mathcal{H}$ to hash n keys into the table, the expected number of collisions is at most $1/2$.

Proof. By the definition of universality, the probability that 2 given keys in the table collide under h is $1/m = 1/n^2$. Since there are $\binom{n}{2}$ pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2} . \qquad \square$$

No collisions at level 2

Corollary. The probability of no collisions is at least $1/2$.

Proof. ***Markov's inequality*** says that for any nonnegative random variable X, we have

$$\Pr\{X \geq t\} \leq E[X]/t.$$

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most $1/2$. □

Thus, just by testing random hash functions in $\mathcal{H}$, we'll quickly find one that works.

Analysis of storage

For the level-1 hash table T, choose $m = n$, and let n_i be random variable for the number of keys that hash to slot i in T. By using n_i^2 slots for the level-2 hash table S_i, the expected total storage required for the two-level scheme is therefore

$$E\left[\sum_{i=0}^{m-1} \Theta\left(n_i^2\right)\right] = \Theta(n),$$

since the analysis is identical to the analysis from recitation of the expected running time of bucket sort. (For a probability bound, apply Markov.)

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 9

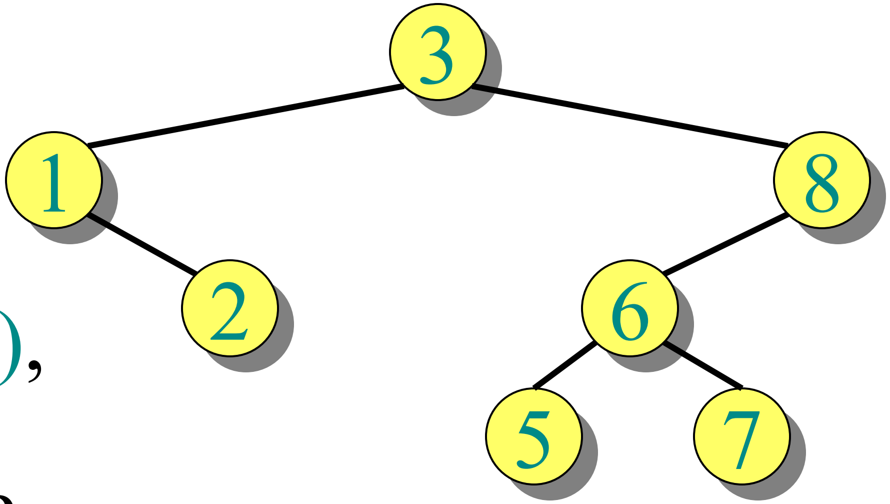
Prof. Charles E. Leiserson

Binary-search-tree sort

```
$T \leftarrow \emptyset$        ▷ Create an empty BST
for $i = 1$ to $n$
    do TREE-INSERT($T$, $A[i]$)
Perform an inorder tree walk of $T$.
```

Example:

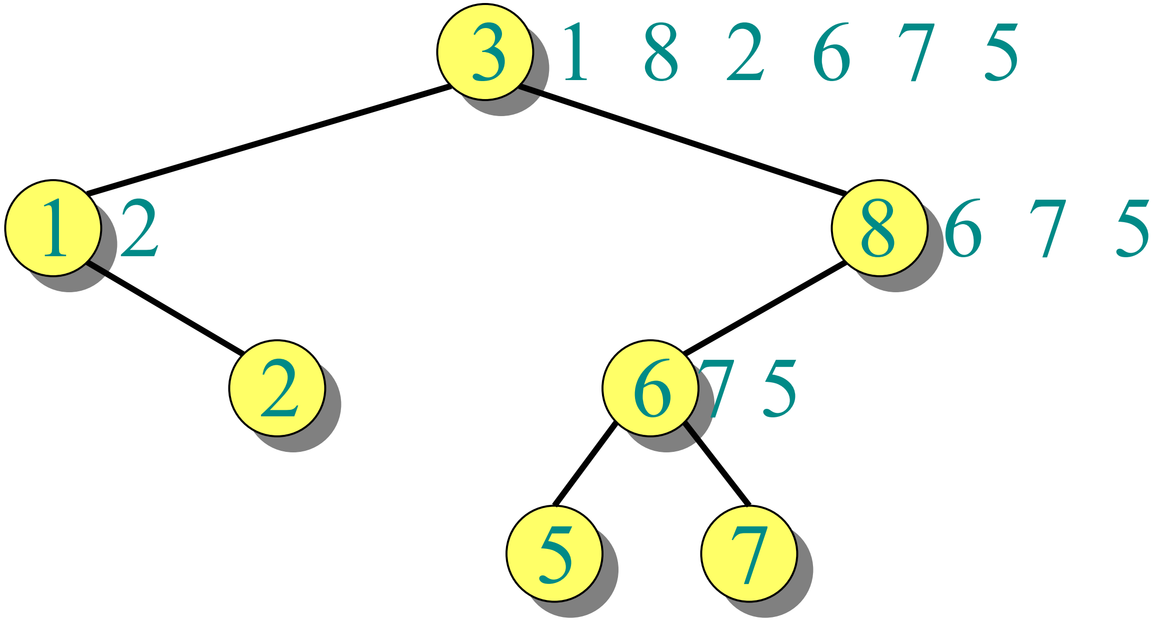
$A = [3\ 1\ 8\ 2\ 6\ 7\ 5]$



Tree-walk time = $O(n)$, but how long does it take to build the BST?

Analysis of BST sort

BST sort performs the same comparisons as quicksort, but in a different order!



The expected time to build the tree is asymptotically the same as the running time of quicksort.

Node depth

The depth of a node = the number of comparisons made during TREE-INSERT. Assuming all input permutations are equally likely, we have

Average node depth

$$= \frac{1}{n} E\left[\sum_{i=1}^{n} (\#\text{comparisons to insert node } i)\right]$$

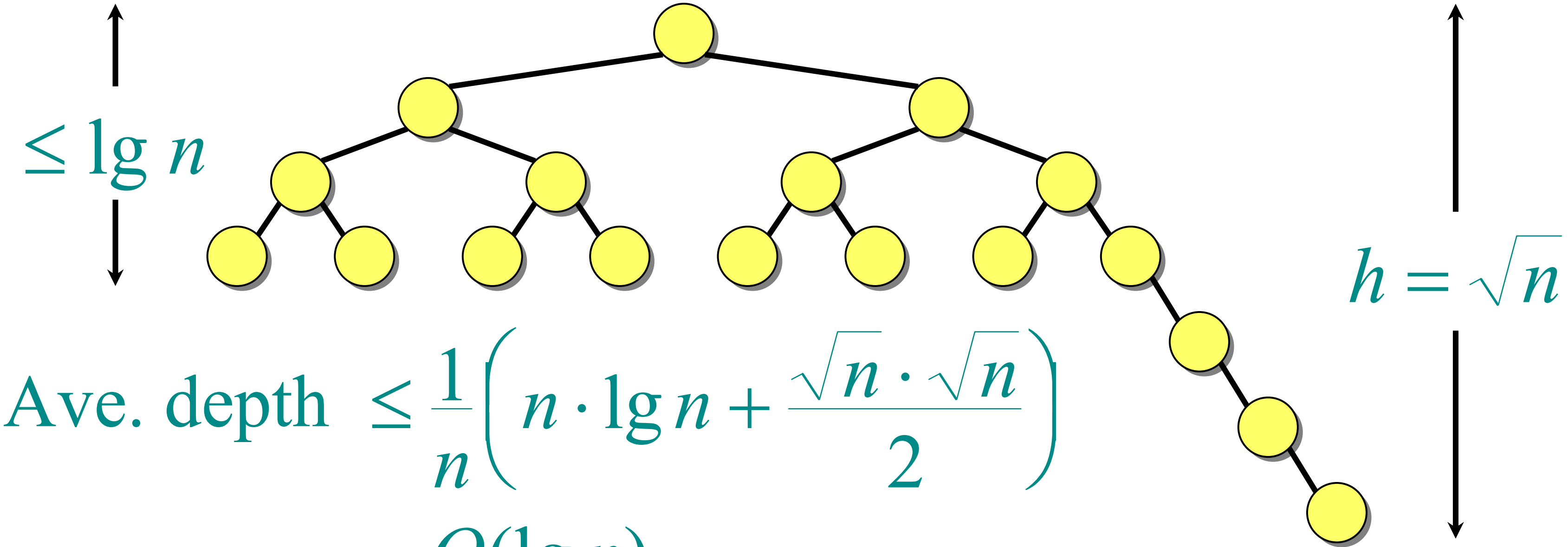
$$= \frac{1}{n} O(n \lg n) \qquad \text{(quicksort analysis)}$$

$$= O(\lg n) \; .$$

Expected tree height

But, average node depth of a randomly built BST = $O(\lg n)$ does not necessarily mean that its expected height is also $O(\lg n)$ (although it is).

Example.



$\le \lg n$

$h = \sqrt{n}$

$$\text{Ave. depth} \le \frac{1}{n}\left(n \cdot \lg n + \frac{\sqrt{n} \cdot \sqrt{n}}{2}\right)$$
$$= O(\lg n)$$

Height of a randomly built binary search tree

Outline of the analysis:

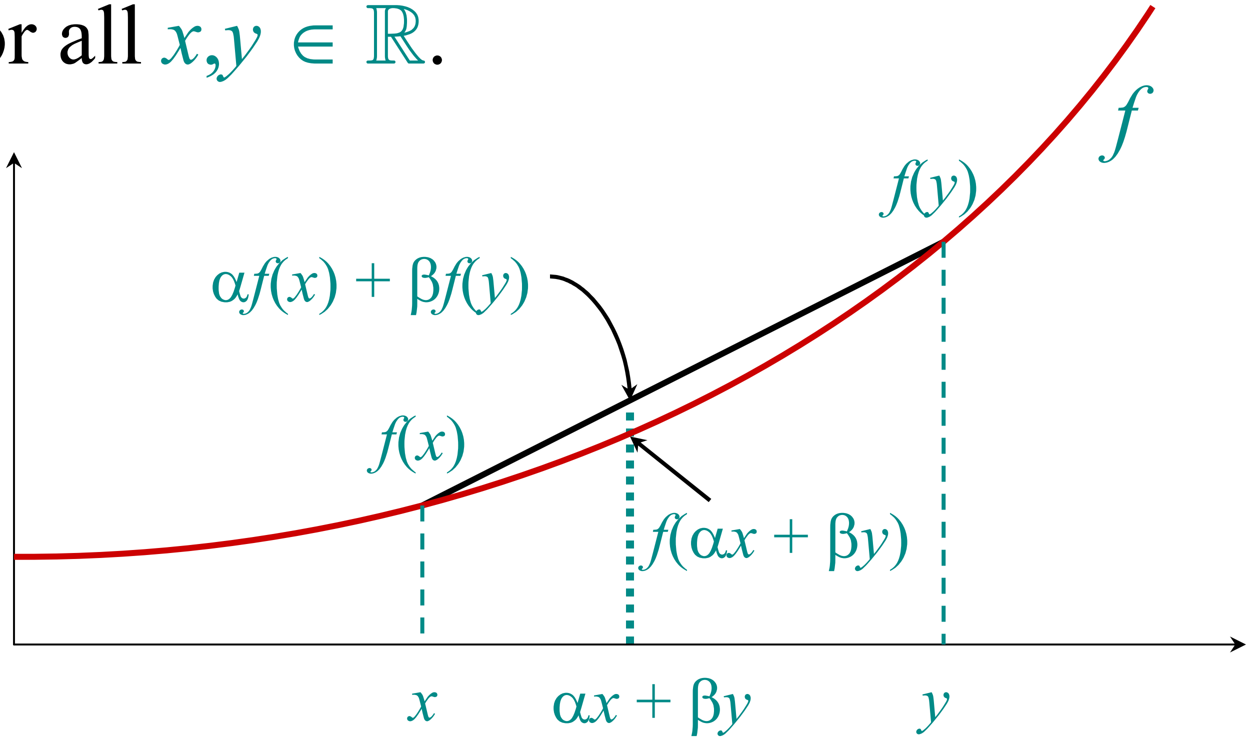
- Prove ***Jensen's inequality***, which says that $f(E[X]) \le E[f(X)]$ for any convex function f and random variable X.
- Analyze the ***exponential height*** of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] = O(n^3)$, and hence that $E[X_n] = O(\lg n)$.

Convex functions

A function $f : \mathbb{R} \to \mathbb{R}$ is ***convex*** if for all $\alpha, \beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all $x, y \in \mathbb{R}$.



Convexity lemma

Lemma. Let $f : \mathbb{R} \to \mathbb{R}$ be a convex function, and let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a set of nonnegative constants such that $\sum_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \ldots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^{n} \alpha_k x_k\right) \le \sum_{k=1}^{n} \alpha_k f(x_k) .$$

Proof. By induction on n. For $n = 1$, we have $\alpha_1 = 1$, and hence $f(\alpha_1 x_1) \le \alpha_1 f(x_1)$ trivially.

Proof (continued)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

Algebra.

Proof (continued)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\le \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

Convexity.

Proof (continued)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\le \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\le \alpha_n f(x_n) + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}f(x_k)$$

Induction.

Proof (continued)

Inductive step:

$$
\begin{aligned}
f\left(\sum_{k=1}^{n} \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\
&\le \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\
&\le \alpha_n f(x_n) + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} f(x_k) \\
&= \sum_{k=1}^{n} \alpha_k f(x_k). \quad \square
\end{aligned}
$$

Algebra.

Jensen's inequality

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \le E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

Definition of expectation.

Jensen's inequality

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \le E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

$$\le \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\}$$

Convexity lemma (generalized).

Jensen’s inequality

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \le E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

$$\le \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\}$$

$$= E[f(X)]. \quad \square$$

Tricky step, but true—think about it.

Analysis of BST height

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k, then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\} ,$$

since each of the left and right subtrees of the root are randomly built. Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\} .$$

Analysis (continued)

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\Pr\{Z_{nk} = 1\} = \mathrm{E}[Z_{nk}] = 1/n$, and

$$Y_n = \sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right) .$$

Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

Take expectation of both sides.

Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}(2 \cdot \max\{Y_{k-1}, Y_{n-k}\})]$$

Linearity of expectation.

Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}(2 \cdot \max\{Y_{k-1}, Y_{n-k}\})]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

Independence of the rank of the root from the ranks of subtree roots.

Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2\cdot\max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}(2\cdot\max\{Y_{k-1}, Y_{n-k}\})]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}]\cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\le \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

The max of two nonnegative numbers is at most their sum, and $\mathrm{E}[Z_{nk}] = 1/n$.

Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

$$= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

Each term appears twice, and reindex.

Solving the recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

Solving the recurrence

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$
\begin{aligned}
E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\
&\leq \frac{4}{n}\sum_{k=0}^{n-1} ck^3
\end{aligned}
$$

Substitution.

Solving the recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\ &\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\ &\le \frac{4c}{n}\int_0^n x^3 dx \end{aligned}$$

Integral method.

Solving the recurrence

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned} E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\ &\leq \frac{4c}{n}\int_0^n x^3\,dx \\ &= \frac{4c}{n}\left(\frac{n^4}{4}\right) \end{aligned}$$

Solve the integral.

Solving the recurrence

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$
\begin{aligned}
E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\
&\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\
&\le \frac{4c}{n}\int_0^n x^3 dx \\
&= \frac{4c}{n}\left(\frac{n^4}{4}\right) \\
&= cn^3. \quad \text{Algebra.}
\end{aligned}
$$

The grand finale

Putting it all together, we have

$$2^{E[X_n]} \le E[2^{X_n}]$$

Jensen's inequality, since $f(x) = 2^x$ is convex.

The grand finale

Putting it all together, we have

$$
\begin{aligned}
2^{E[X_n]} &\le E[2^{X_n}] \\
&= E[Y_n]
\end{aligned}
$$

Definition.

The grand finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\le E[2^{X_n}] \\ &= E[Y_n] \\ &\le cn^3 . \end{aligned}$$

What we just showed.

The grand finale

Putting it all together, we have

$$
\begin{aligned}
2^{E[X_n]} &\le E[2^{X_n}] \\
&= E[Y_n] \\
&\le cn^3 .
\end{aligned}
$$

Taking the lg of both sides yields

$$E[X_n] \le 3 \lg n + O(1) .$$

Post mortem

Q. Does the analysis have to be this hard?

Q. Why bother with analyzing exponential height?

Q. Why not just develop the recurrence on

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

directly?

Post mortem (continued)

A. The inequality

$$\max\{a, b\} \le a + b\ .$$

provides a poor upper bound, since the RHS approaches the LHS slowly as $|a - b|$ increases. The bound

$$\max\{2^a, 2^b\} \le 2^a + 2^b$$

allows the RHS to approach the LHS far more quickly as $|a - b|$ increases. By using the convexity of $f(x) = 2^x$ via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

Thought exercises

- See what happens when you try to do the analysis on X_n directly.
- Try to understand better why the proof uses an exponential. Will a quadratic do?
- See if you can find a simpler argument. (This argument is a little simpler than the one in the book—I hope it's correct!)

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 10

Prof. Erik Demaine

Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

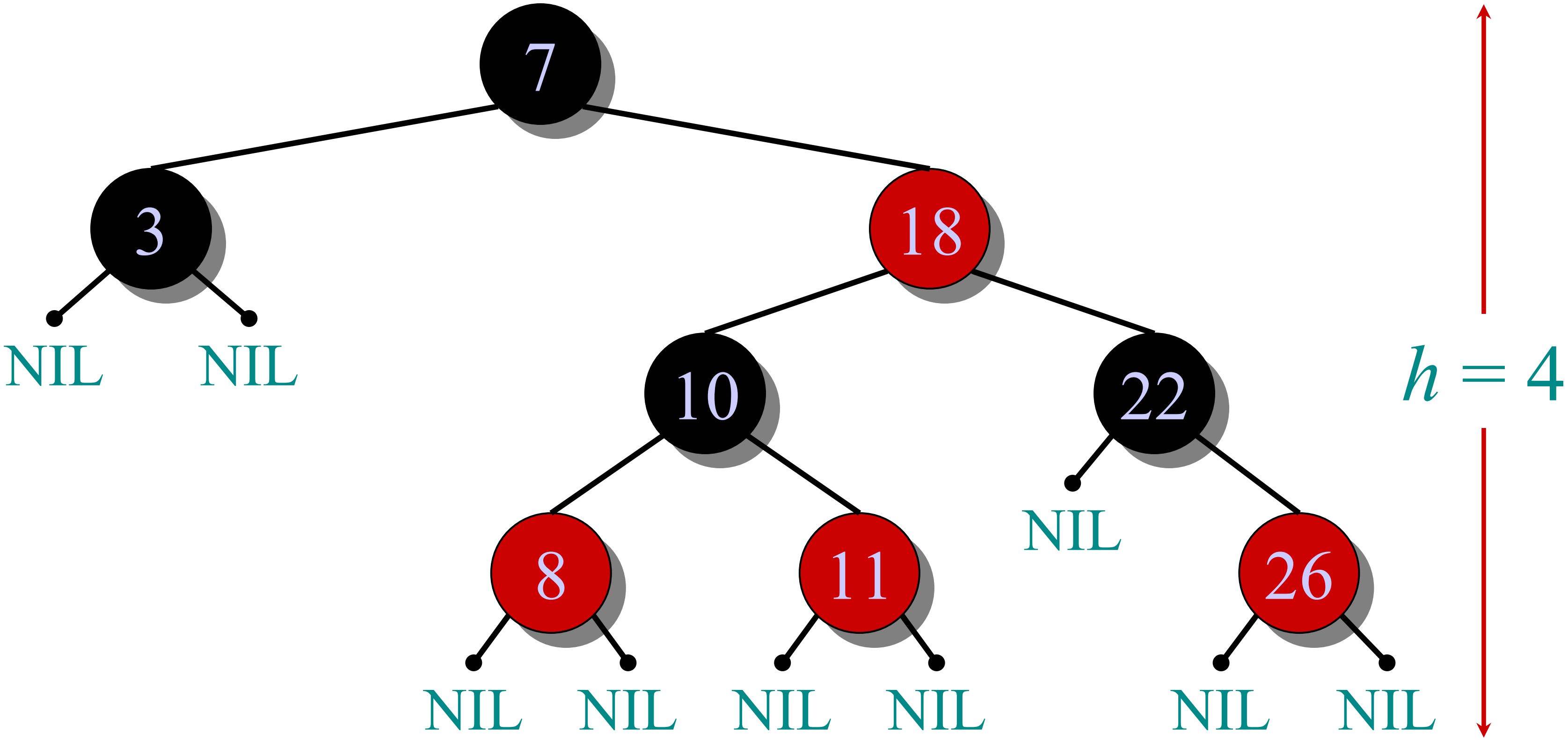
Red-black trees

This data structure requires an extra one-bit color field in each node.

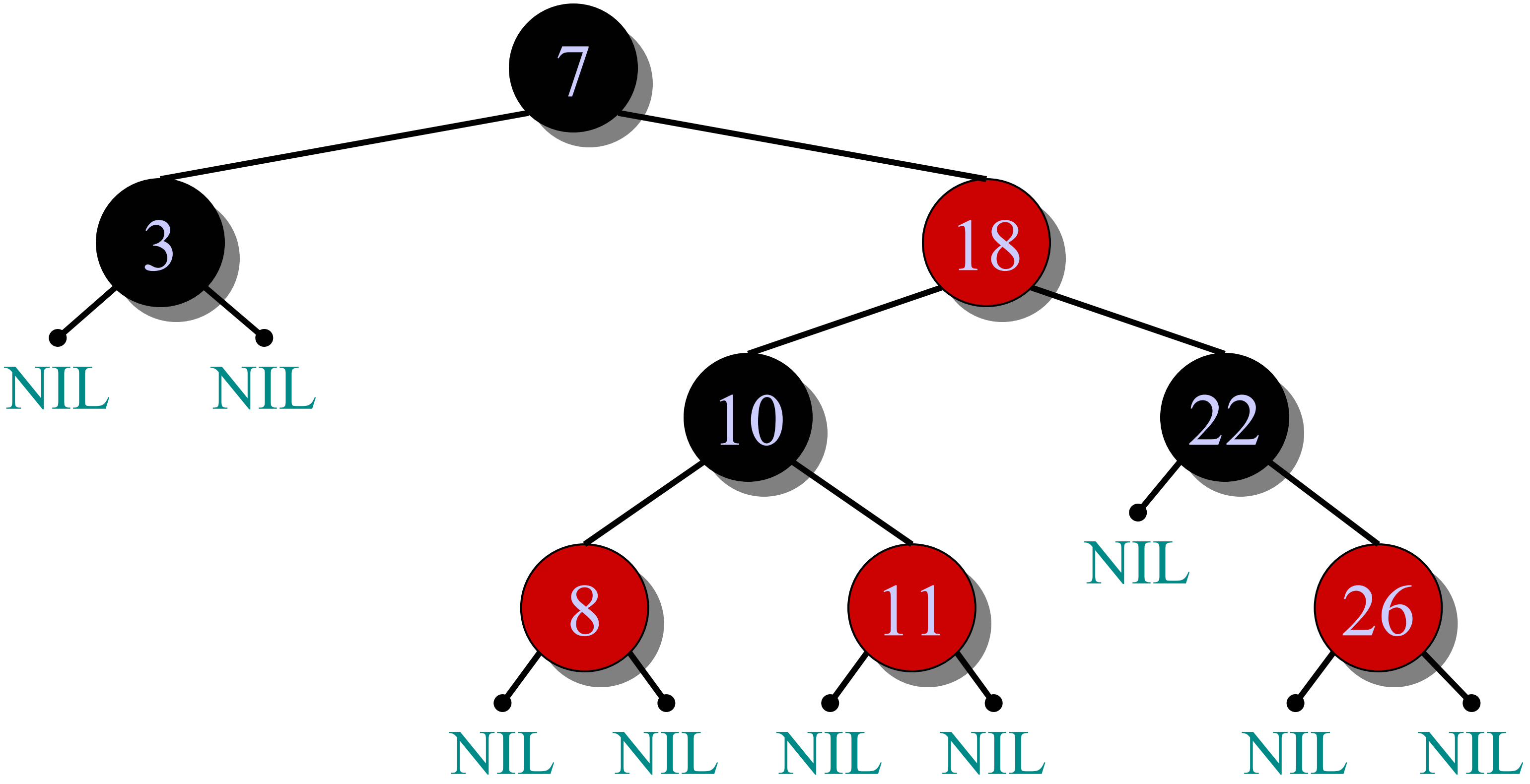
Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Example of a red-black tree

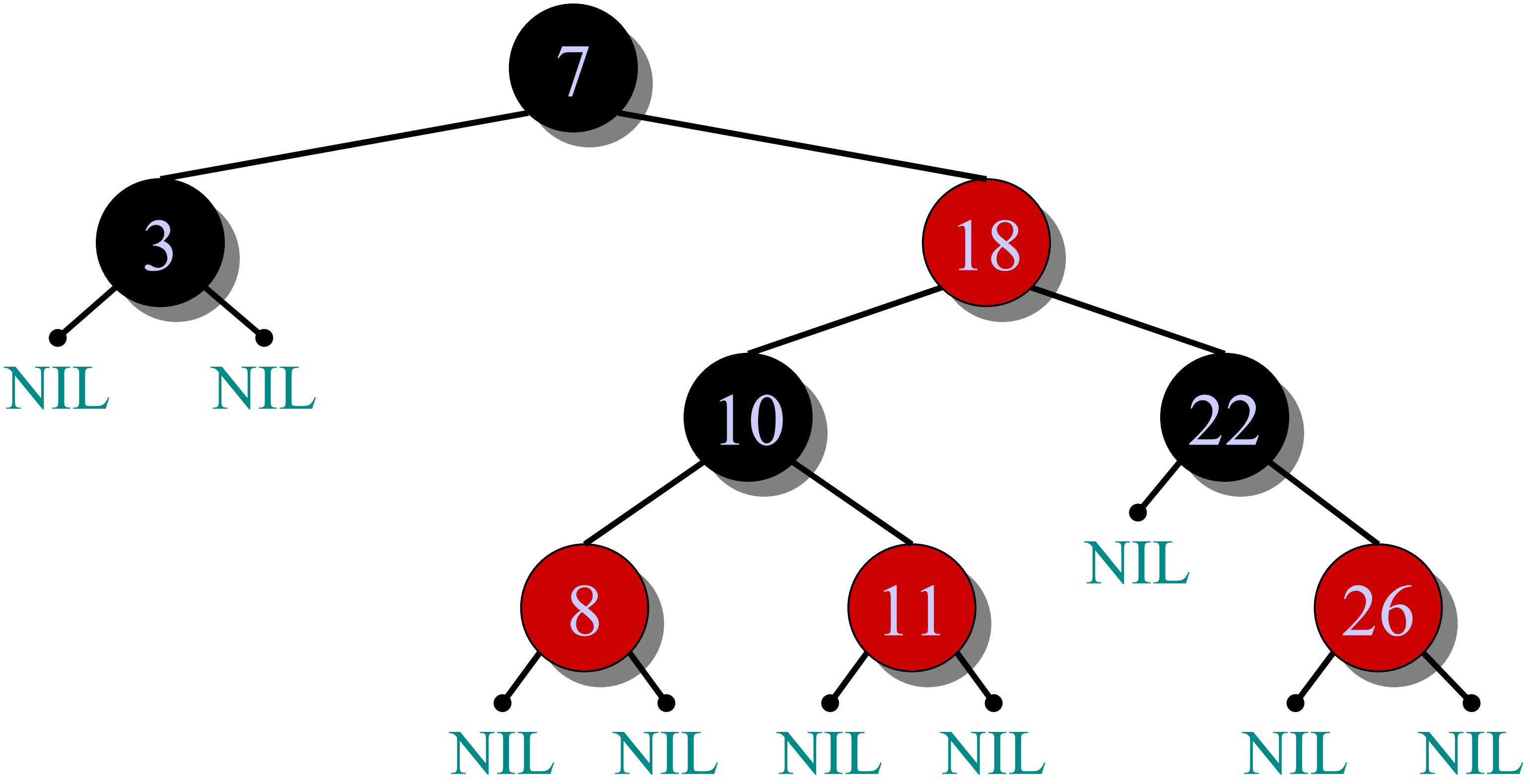


Example of a red-black tree



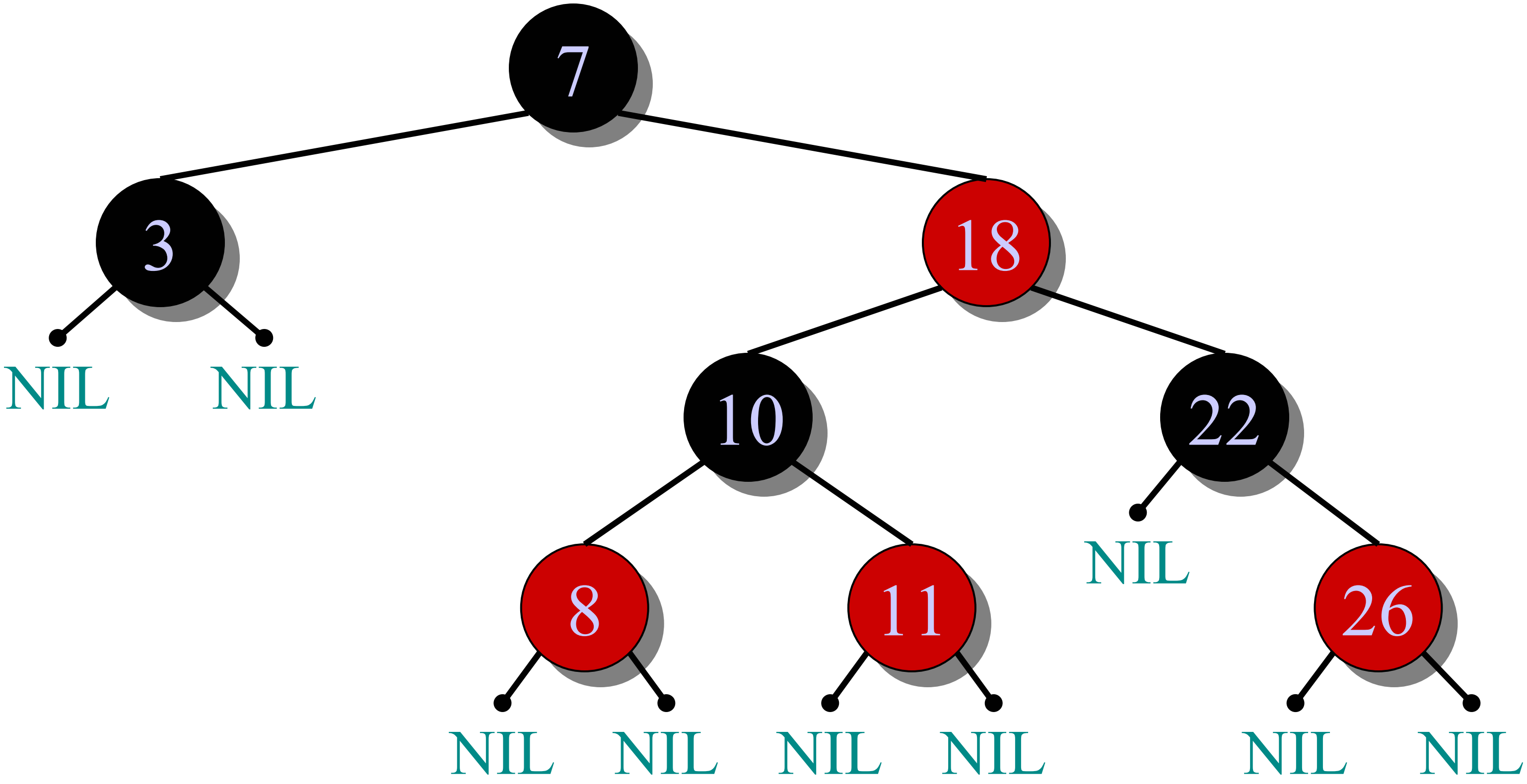
1. Every node is either red or black.

Example of a red-black tree



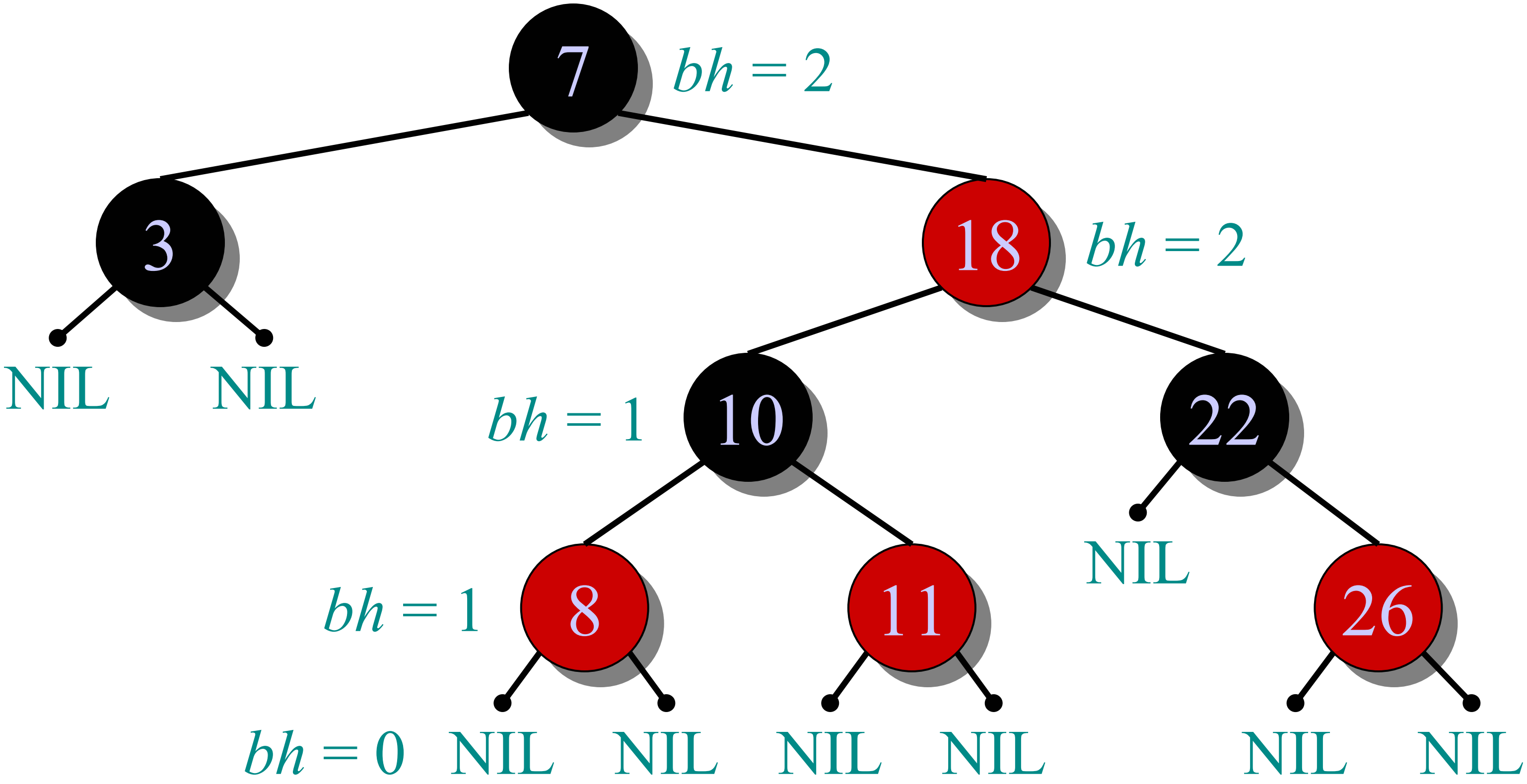
2. The root and leaves (NIL’s) are black.

Example of a red-black tree



3. If a node is red, then its parent is black.

Example of a red-black tree



4. All simple paths from any node x to a descendant leaf have the same number of black nodes = *black-height*(x).

Height of a red-black tree

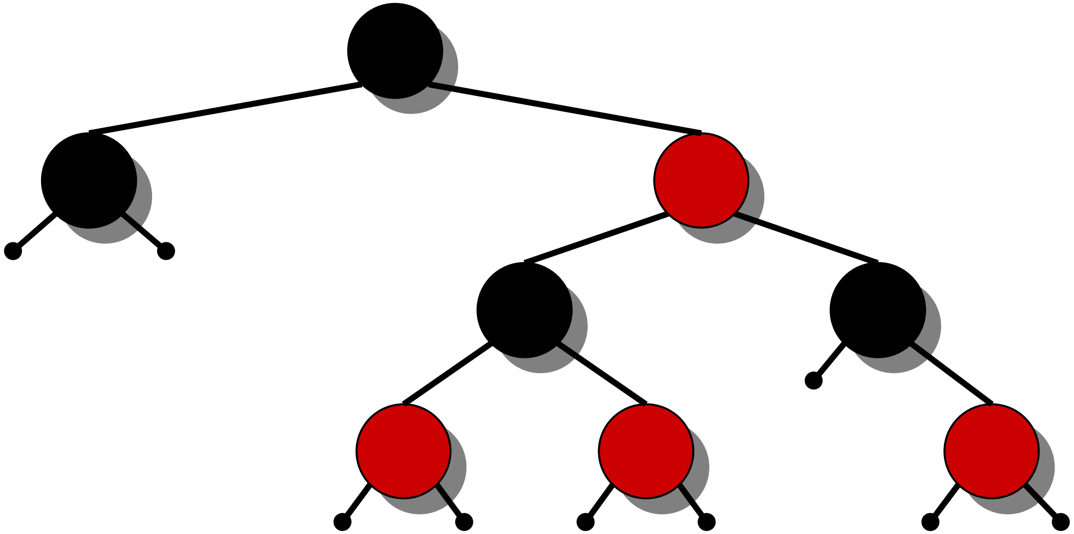
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

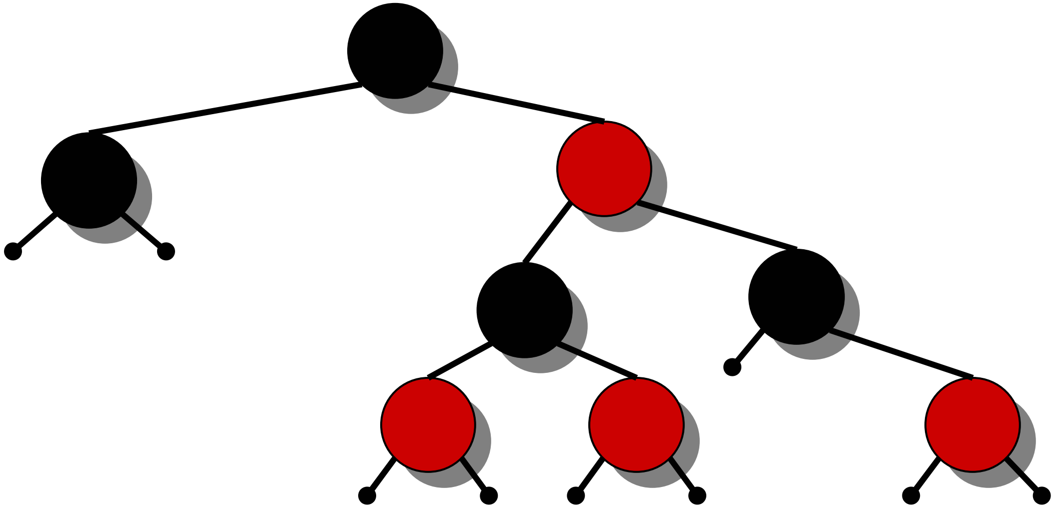
Theorem. A red-black tree with n keys has height

$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

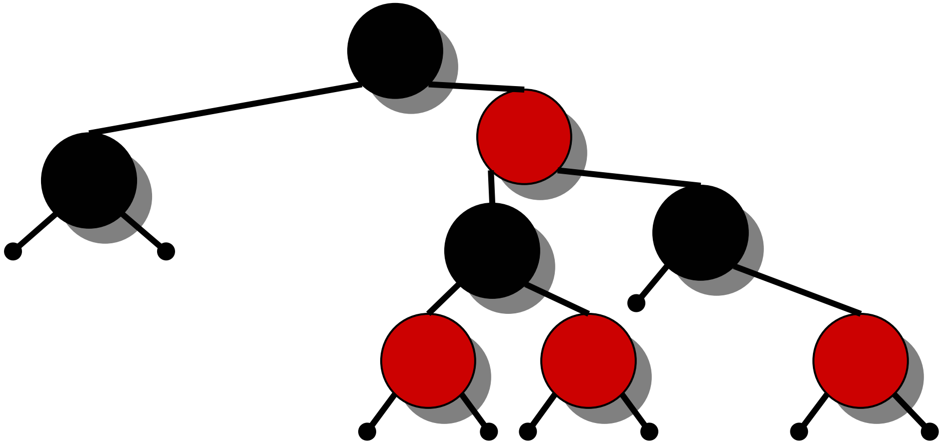
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

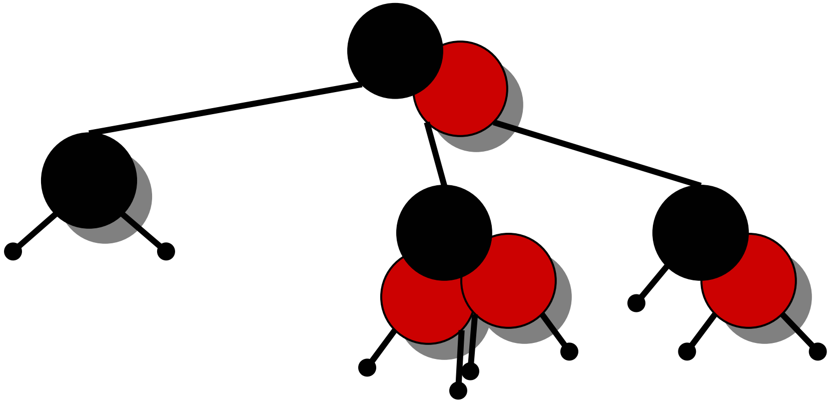
Theorem. A red-black tree with n keys has height

$$h \le 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

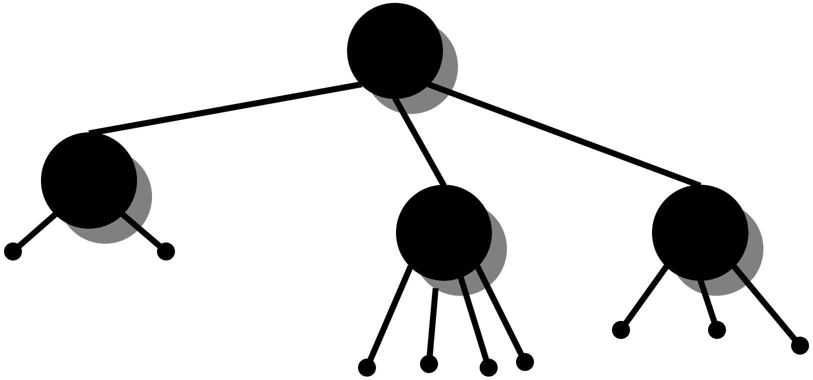
Theorem. A red-black tree with n keys has height

$$h \le 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



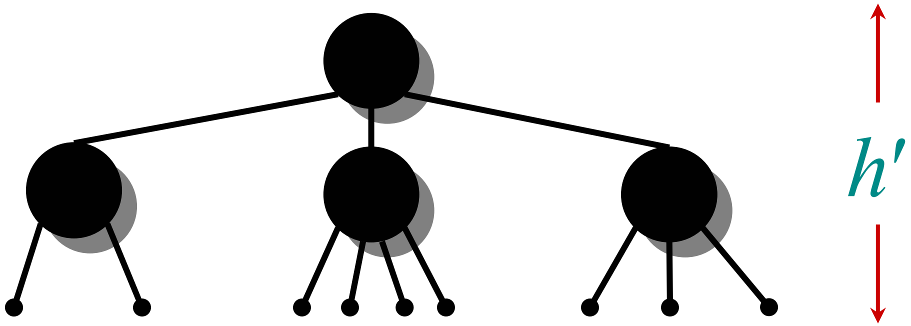
Height of a red-black tree

Theorem. A red-black tree with n keys has height

$$h \leq 2\lg(n+1).$$

Proof. (The book uses induction. Read carefully.)

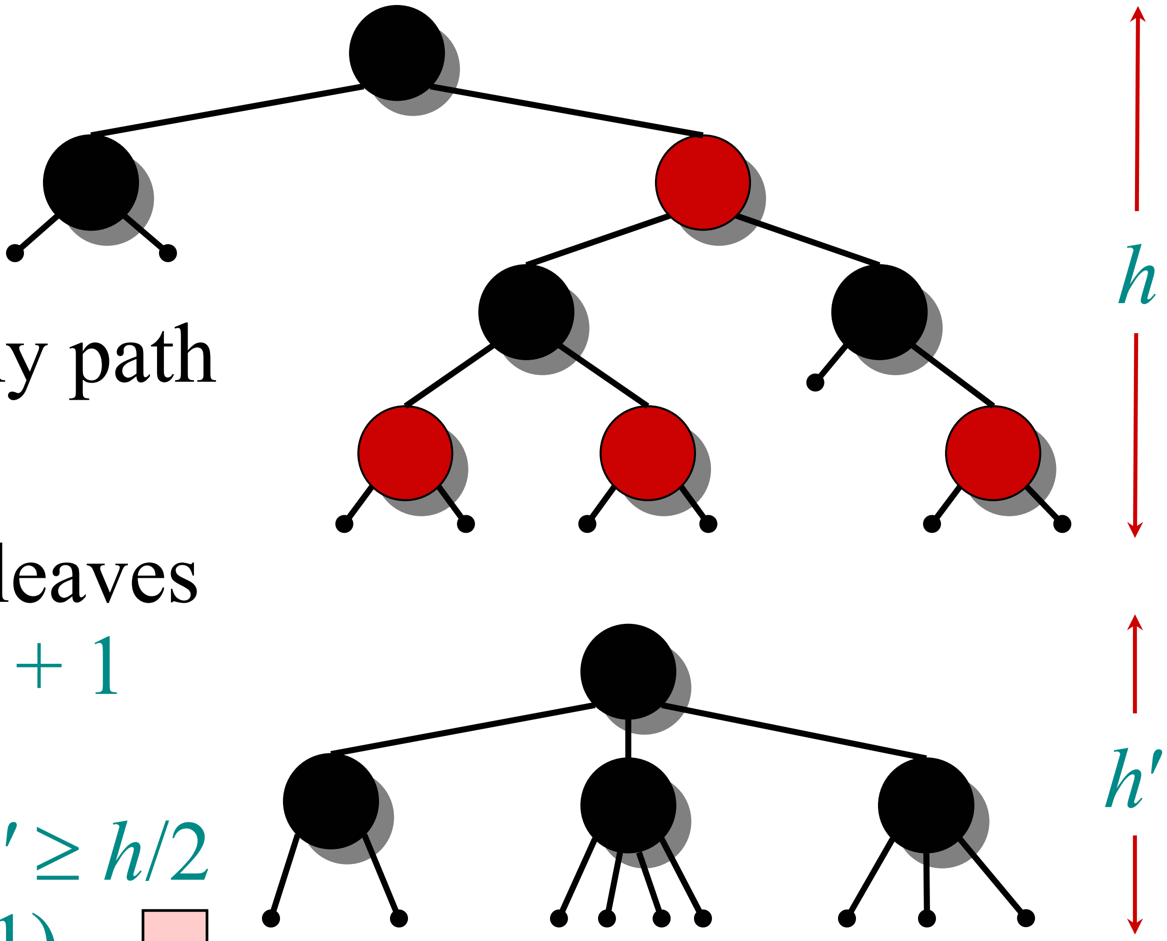
INTUITION:



- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

Proof (continued)

- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n + 1$
  $\Rightarrow n + 1 \geq 2^{h'}$
  $\Rightarrow \lg(n + 1) \geq h' \geq h/2$
  $\Rightarrow h \leq 2 \lg(n + 1)$. □



Query operations

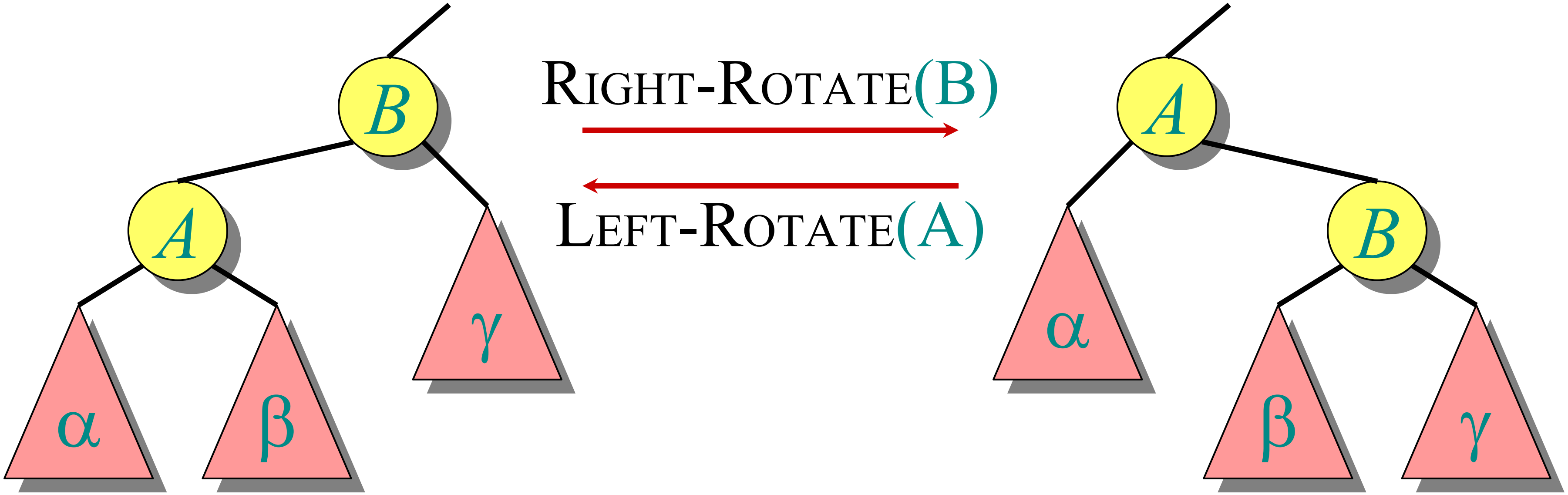
Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.

Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree via ***"rotations"***.

Rotations



Rotations maintain the inorder ordering of keys:

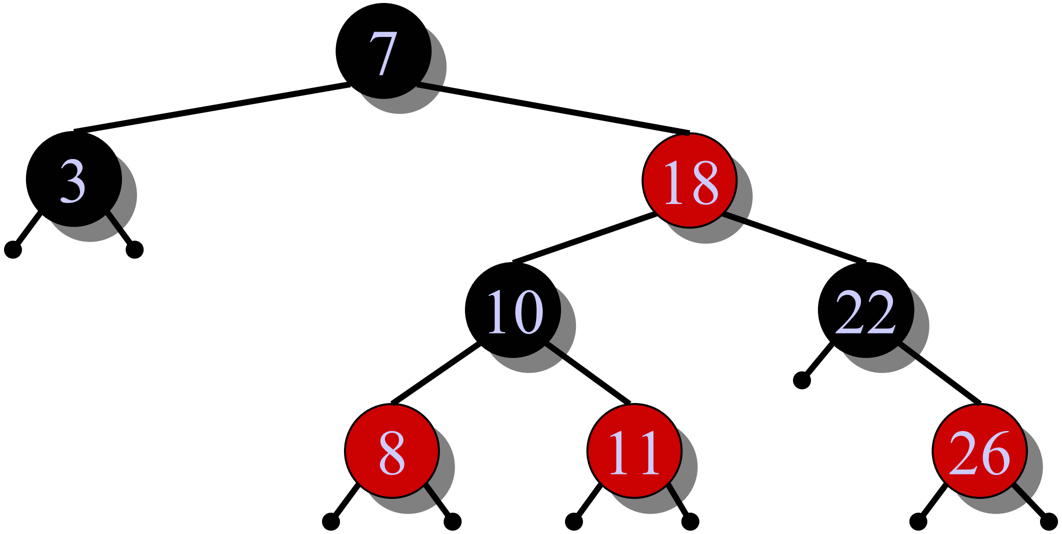
- $a \in \alpha$, $b \in \beta$, $c \in \gamma \Rightarrow a \le A \le b \le B \le c$.

A rotation can be performed in $O(1)$ time.

Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

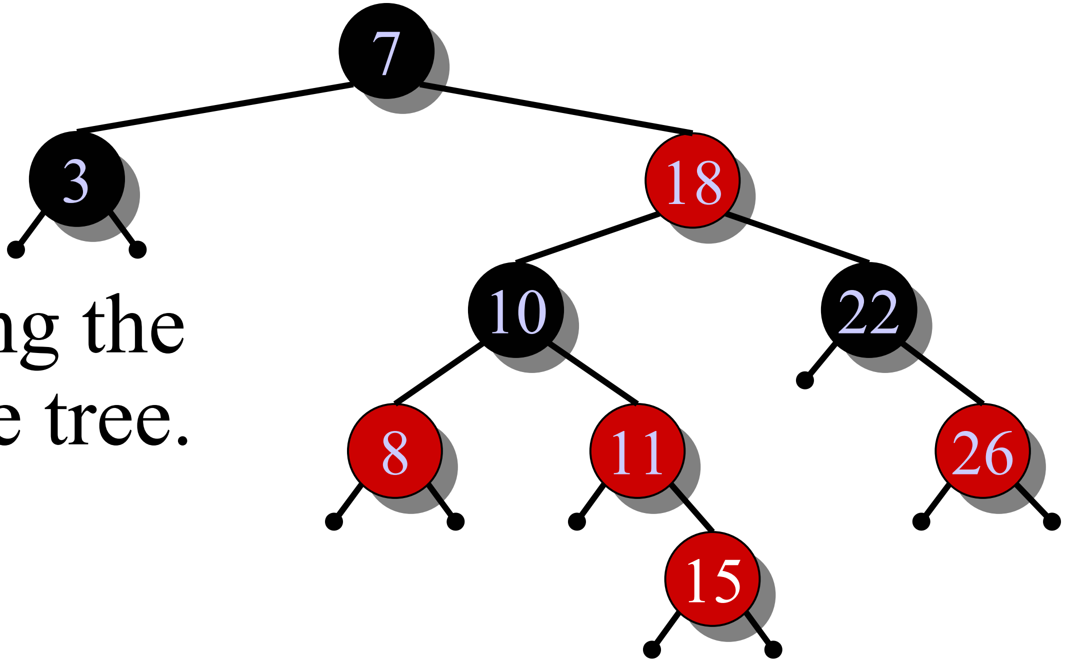


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.

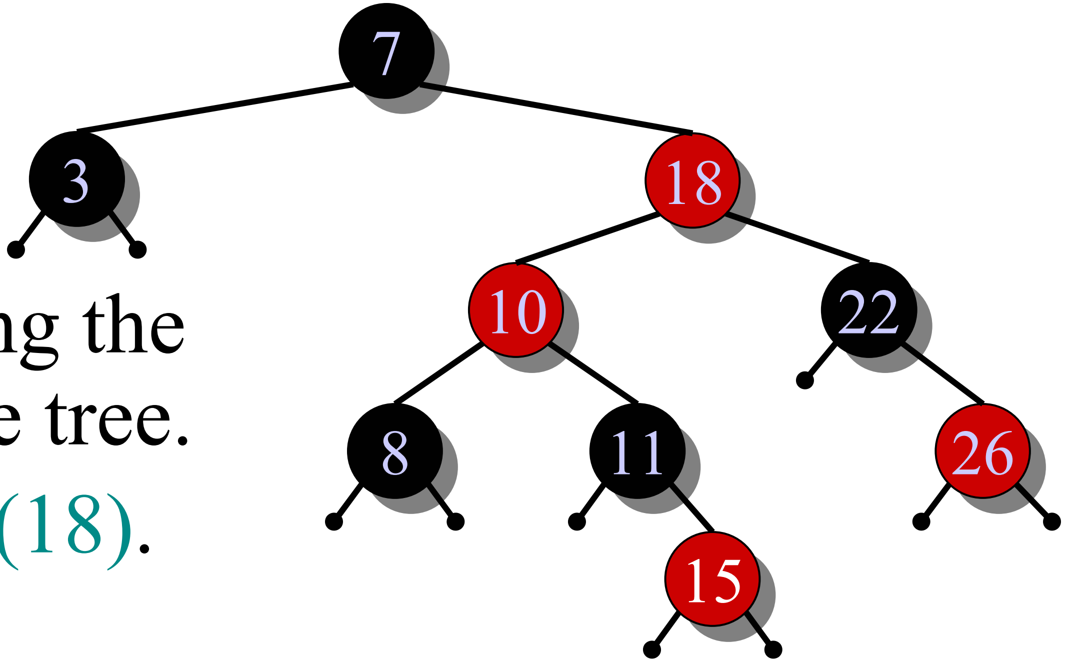


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).

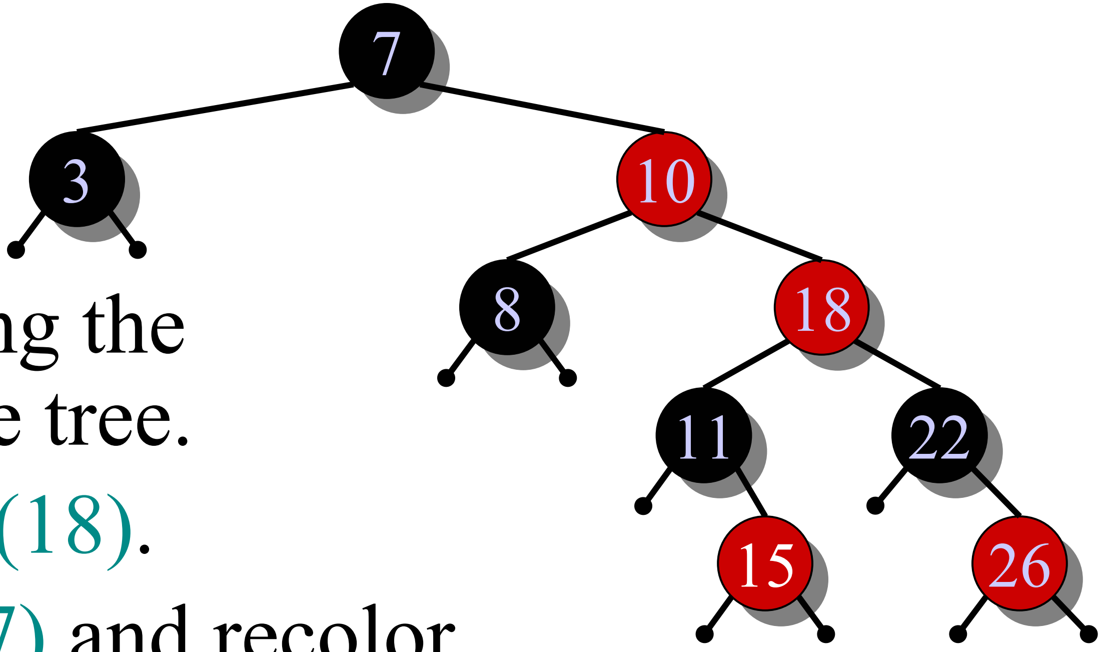


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

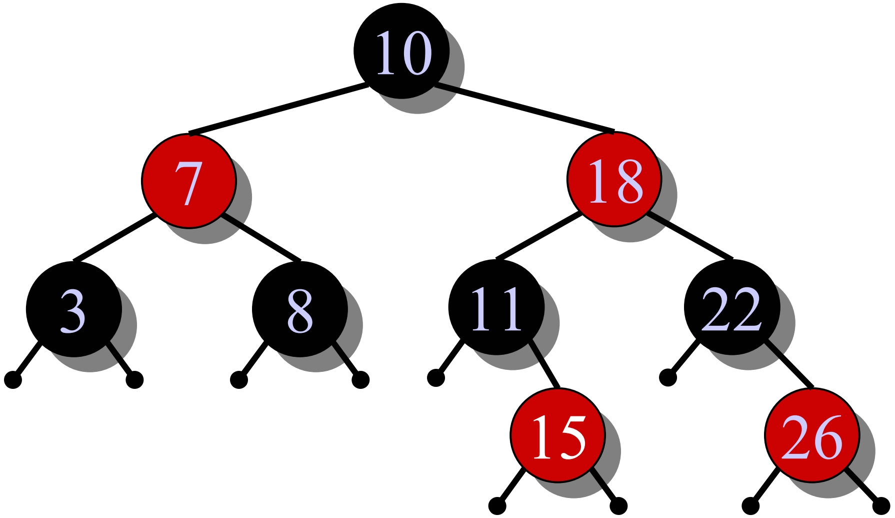


Insertion into a red-black tree

Idea: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- Right-Rotate(18).
- Left-Rotate(7) and recolor.



Pseudocode

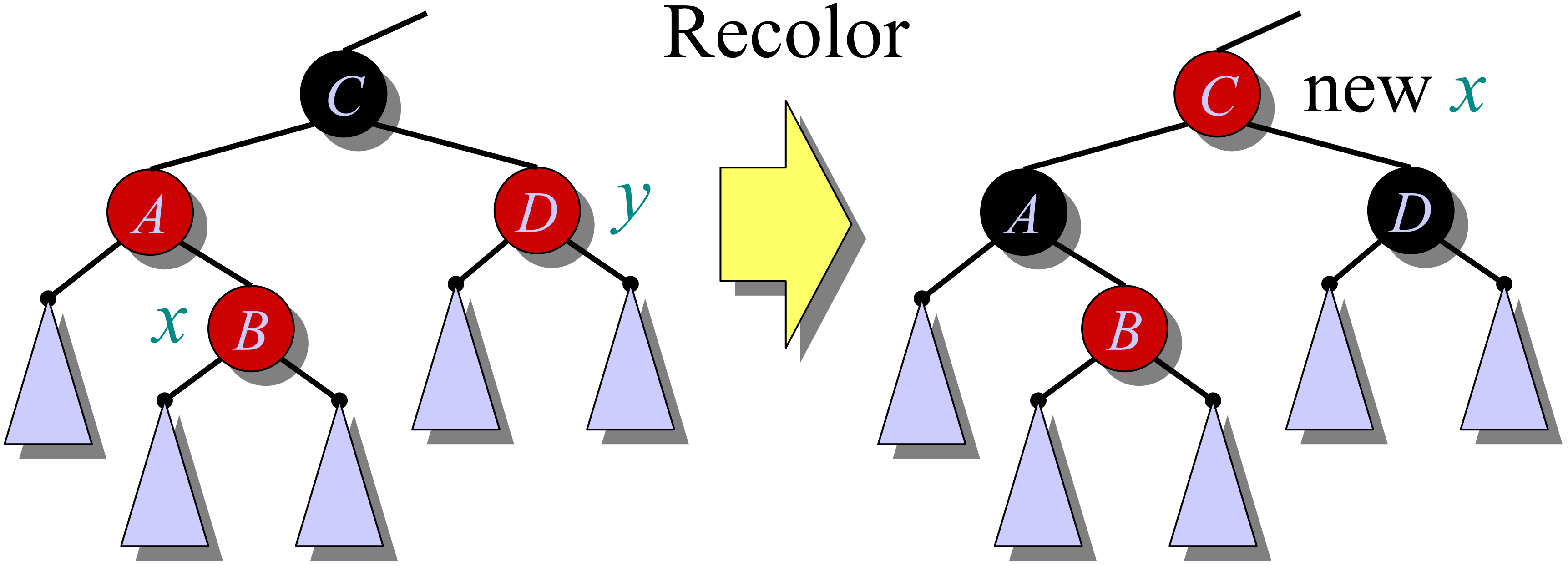
```
RB-INSERT($T$, $x$)
    TREE-INSERT($T$, $x$)
    $color[x] \leftarrow$ RED      ▷ only RB property 3 can be violated
    while $x \neq root[T]$ and $color[p[x]] =$ RED
        do if $p[x] = left[p[p[x]]]$
            then $y \leftarrow right[p[p[x]]]$          ▷ $y =$ aunt/uncle of $x$
                if $color[y] =$ RED
                 then ⟨Case 1⟩
                 else  if $x = right[p[x]]$
                        then ⟨Case 2⟩      ▷ Case 2 falls into Case 3
                       ⟨Case 3⟩
            else ⟨“then” clause with “left” and “right” swapped⟩
    $color[root[T]] \leftarrow$ BLACK
```

Graphical notation

Let △ denote a subtree with a black root.

All △'s have the same black-height.

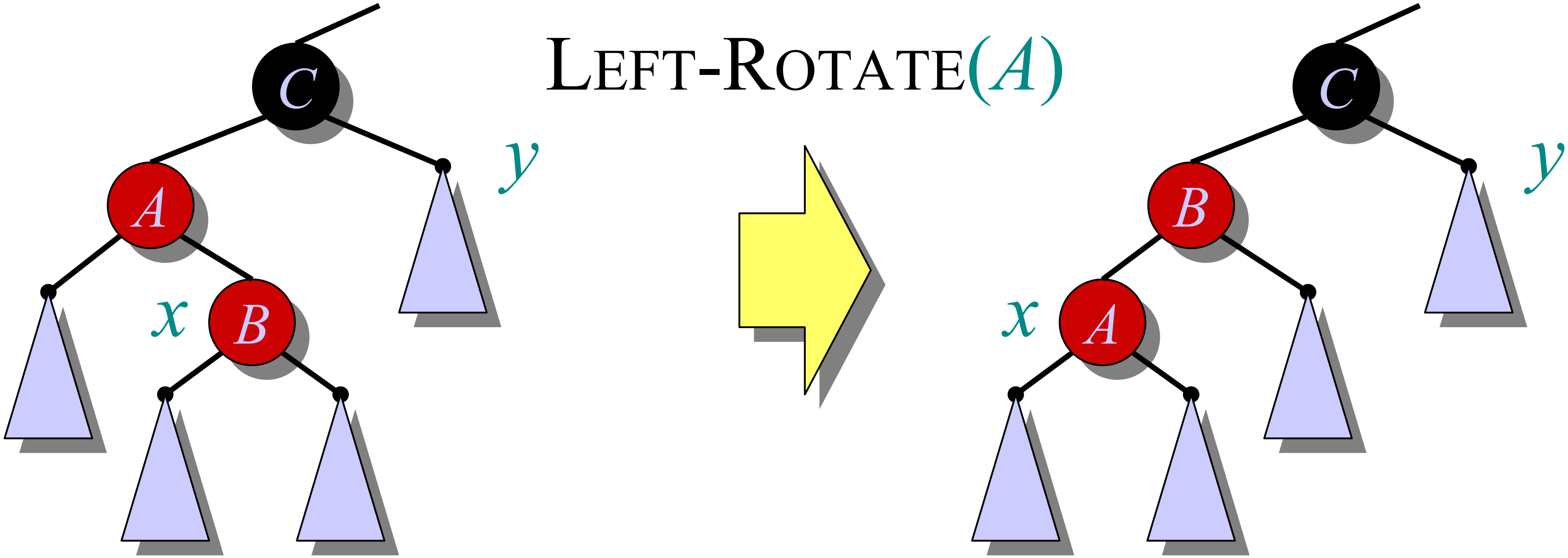
Case 1



(Or, children of *A* are swapped.)

Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

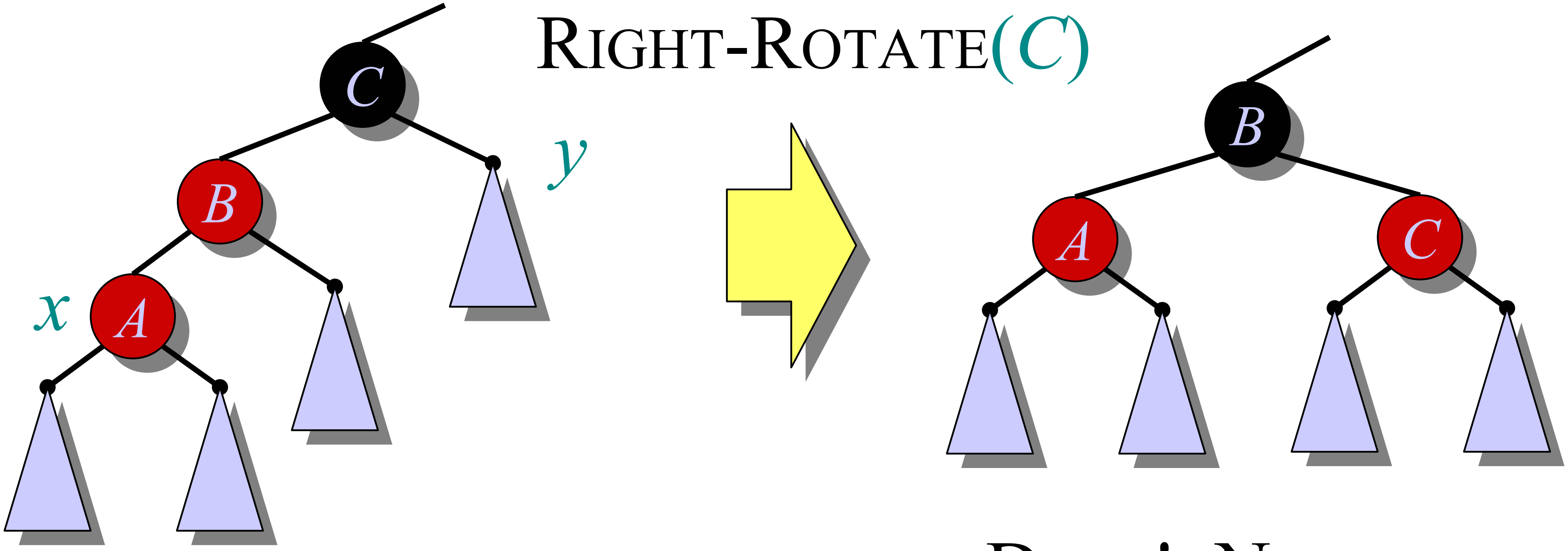
Case 2



LEFT-ROTATE(A)

Transform to Case 3.

Case 3



RIGHT-ROTATE(C)

Done! No more violations of RB property 3 are possible.

Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running time: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 11

Prof. Erik Demaine

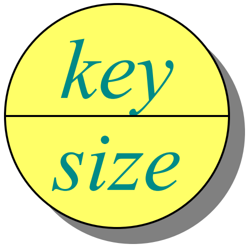
Dynamic order statistics

OS-SELECT(i, S): returns the ith smallest element in the dynamic set S.

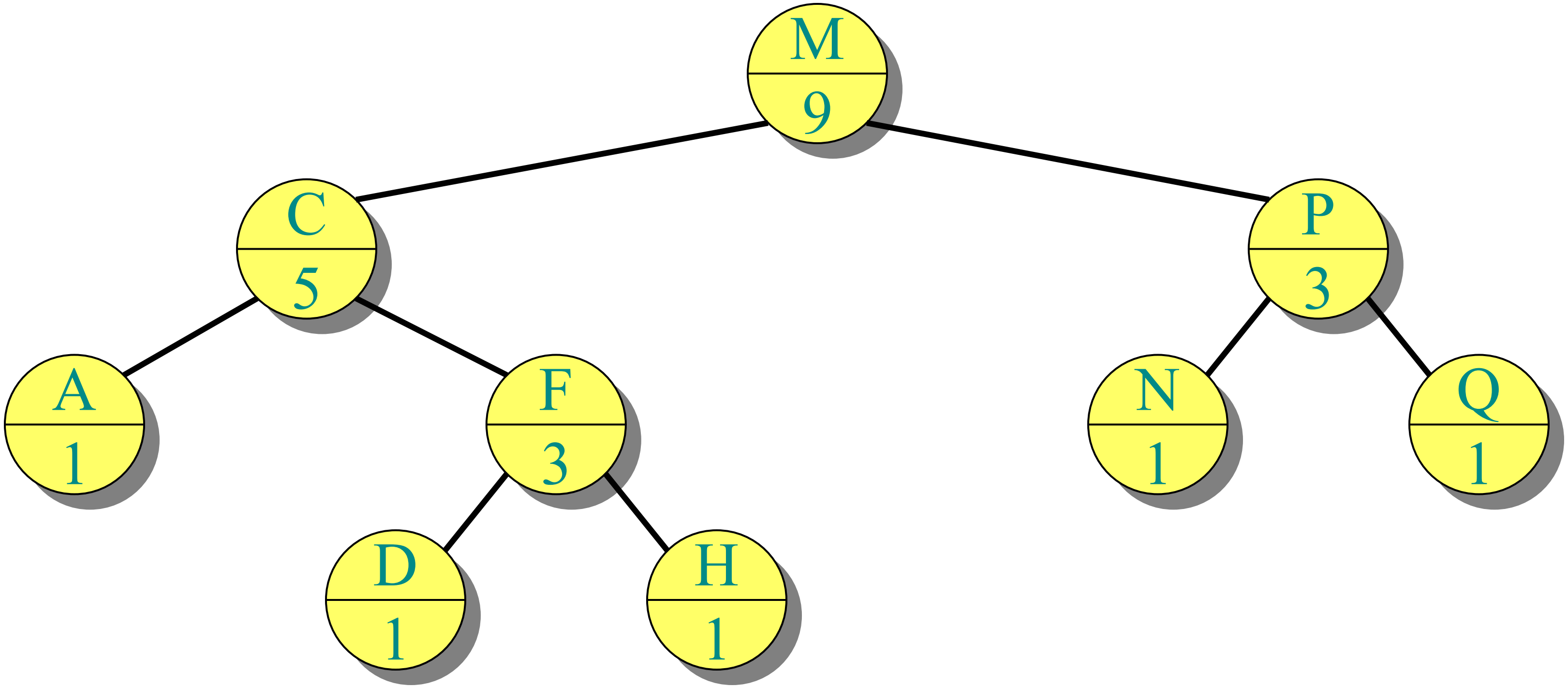
OS-RANK(x, S): returns the rank of $x \in S$ in the sorted order of S's elements.

IDEA: Use a red-black tree for the set S, but keep subtree sizes in the nodes.

Notation for nodes:



Example of an OS-tree



$$size[x] = size[left[x]] + size[right[x]] + 1$$

Selection

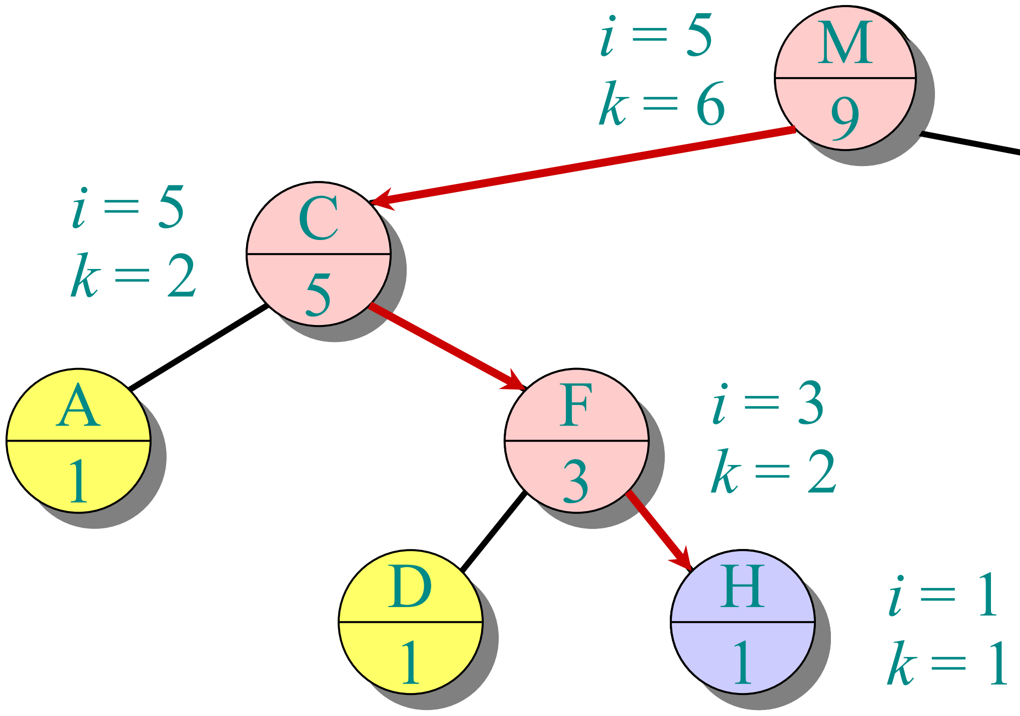
Implementation trick: Use a ***sentinel*** (dummy record) for NIL such that $size[\text{NIL}] = 0$.

```
OS-SELECT$(x, i)$   ▷ $i$th smallest element in the
                      subtree rooted at $x$
  $k \leftarrow size[left[x]] + 1$   ▷ $k = \text{rank}(x)$
  if $i = k$ then return $x$
  if $i < k$
     then return OS-SELECT$(left[x], i)$
     else return OS-SELECT$(right[x], i - k)$
```

(OS-RANK is in the textbook.)

Example

OS-SELECT(*root*, 5)



Running time = $O(h) = O(\lg n)$ for red-black trees.

Data structure maintenance

Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?

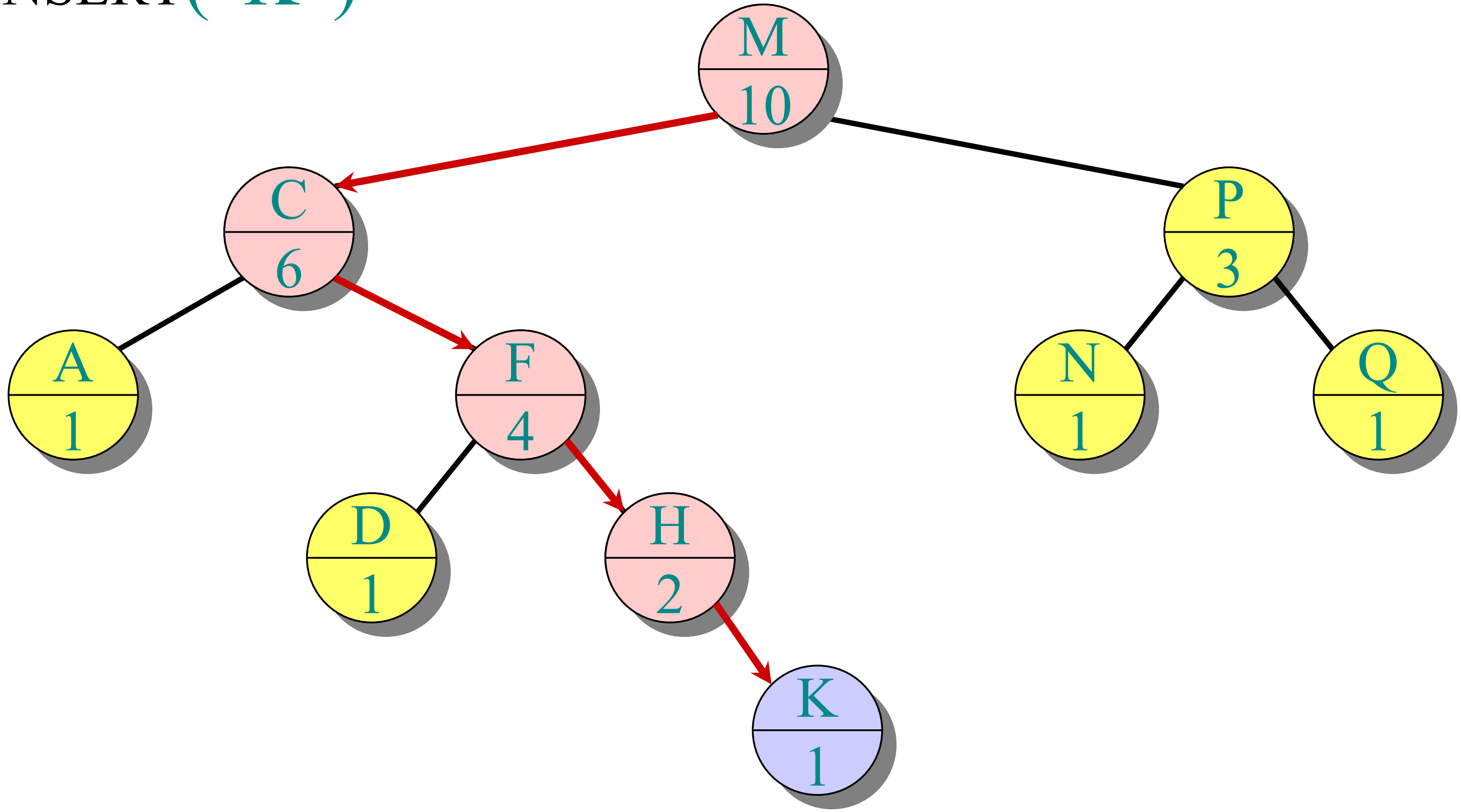
A. They are hard to maintain when the red-black tree is modified.

Modifying operations: INSERT and DELETE.

Strategy: Update subtree sizes when inserting or deleting.

Example of insertion

INSERT("K")

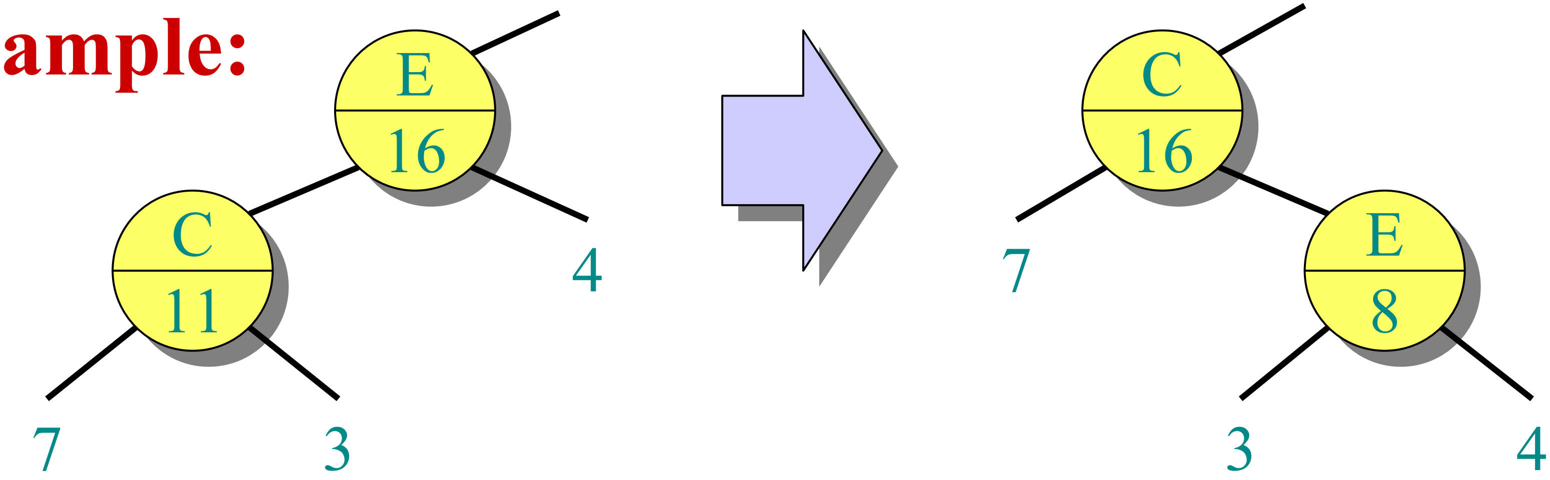


Handling rebalancing

Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

- *Recolorings*: no effect on subtree sizes.
- *Rotations*: fix up subtree sizes in $O(1)$ time.

Example:



∴ RB-INSERT and RB-DELETE still run in $O(\lg n)$ time.

Data-structure augmentation

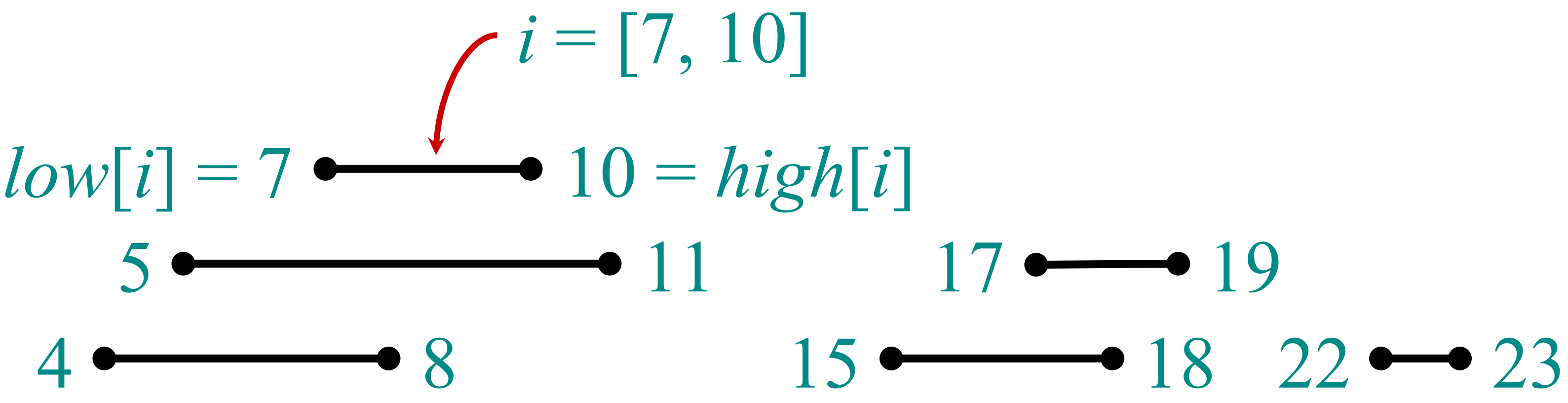
Methodology: (*e.g., order-statistics trees*)

1. Choose an underlying data structure (*red-black trees*).
2. Determine additional information to be stored in the data structure (*subtree sizes*).
3. Verify that this information can be maintained for modifying operations (*RB-INSERT, RB-DELETE — don't forget rotations*).
4. Develop new dynamic-set operations that use the information (*OS-SELECT and OS-RANK*).

These steps are guidelines, not rigid rules.

Interval trees

Goal: To maintain a dynamic set of intervals, such as time intervals.



$i = [7, 10]$

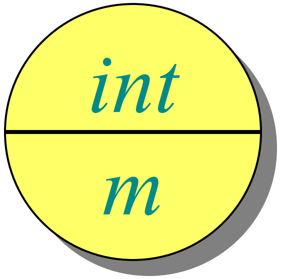
$low[i] = 7$

$10 = high[i]$

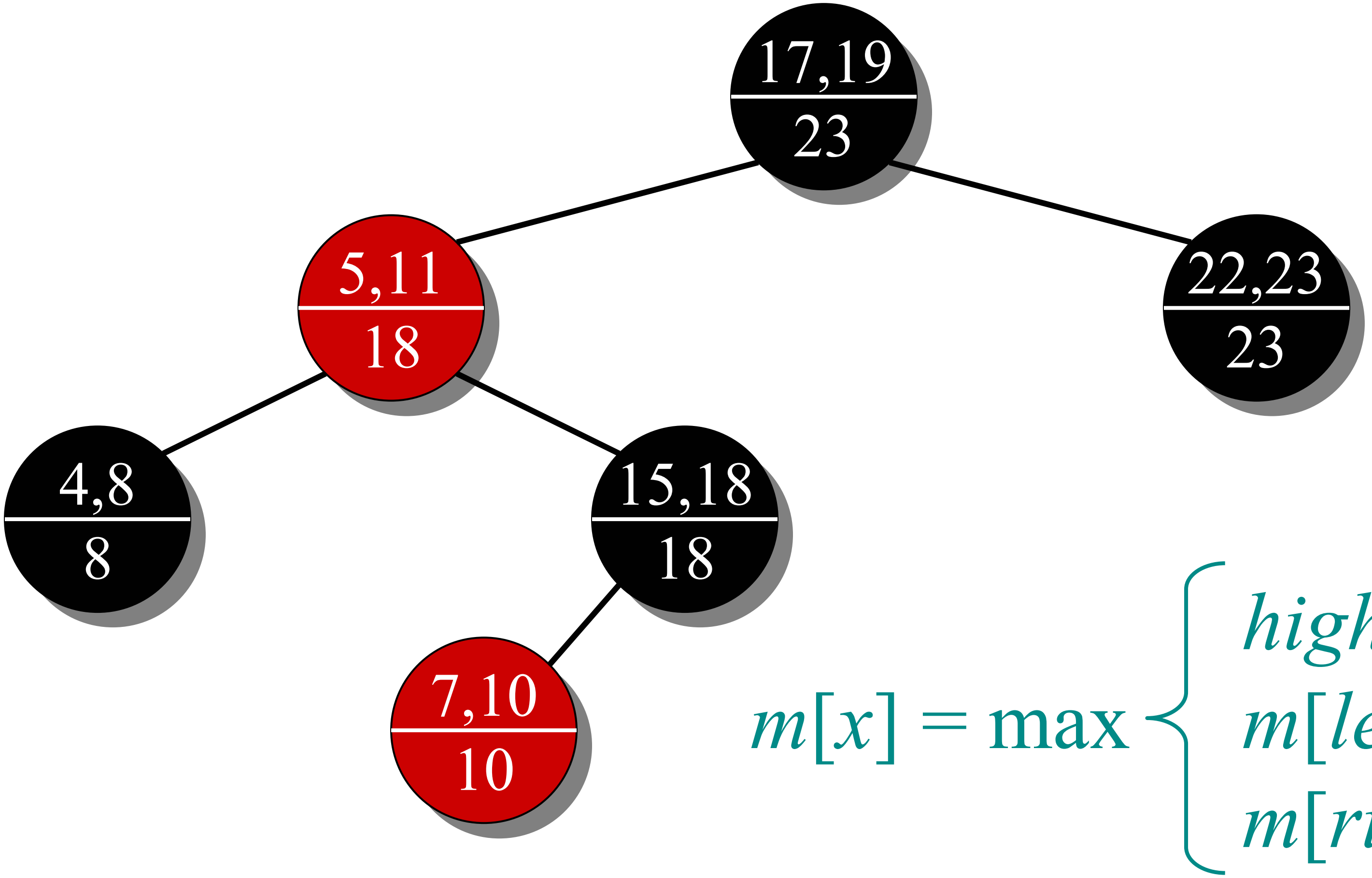
Query: For a given query interval i, find an interval in the set that overlaps i.

Following the methodology

1. *Choose an underlying data structure.*
   - Red-black tree keyed on low (left) endpoint.
2. *Determine additional information to be stored in the data structure.*
   - Store in each node x the largest value $m[x]$ in the subtree rooted at x, as well as the interval $int[x]$ corresponding to the key.



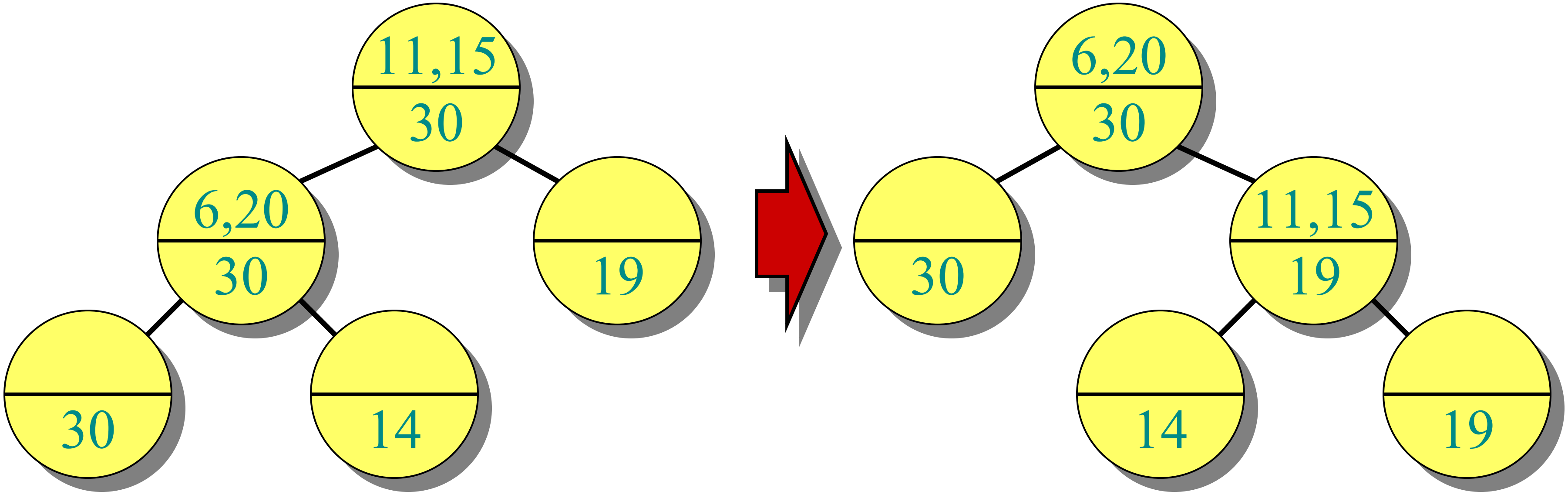
Example interval tree



$$m[x] = \max \begin{cases} high[int[x]] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

Modifying operations

3. *Verify that this information can be maintained for modifying operations.*
   - INSERT: Fix *m*'s on the way down.
   - Rotations — Fixup = $O(1)$ time per rotation:



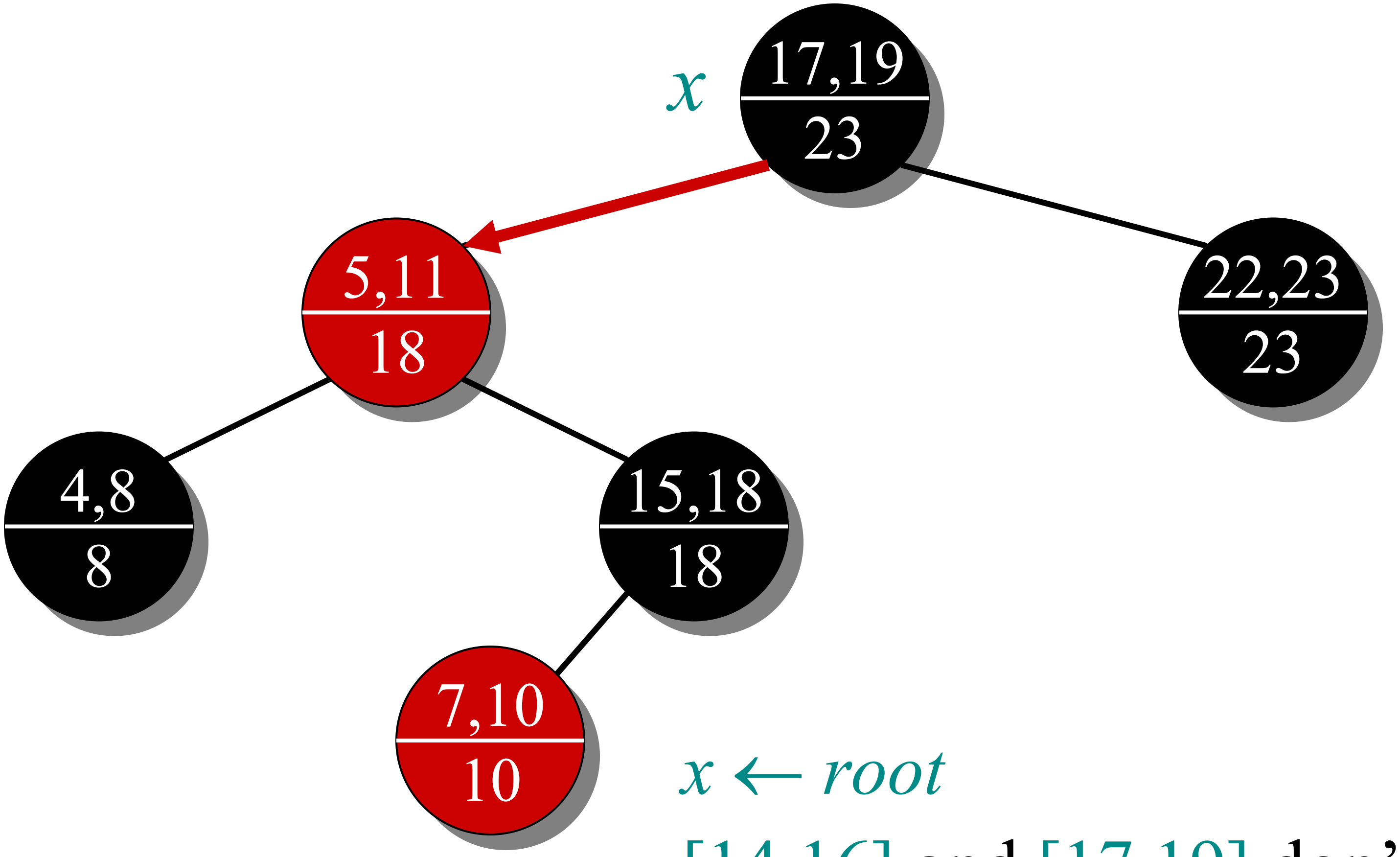
Total INSERT time = $O(\lg n)$; DELETE similar.

New operations

4. Develop new dynamic-set operations that use the information.

```
INTERVAL-SEARCH($i$)
    $x \leftarrow root$
    while $x \neq$ NIL and ($low[i] > high[int[x]]$
                               or $low[int[x]] > high[i]$)
        do ▷ $i$ and $int[x]$ don't overlap
            if $left[x] \neq$ NIL and $low[i] \leq m[left[x]]$
                then $x \leftarrow left[x]$
                else $x \leftarrow right[x]$
    return $x$
```

Example 1: INTERVAL-SEARCH([14,16])

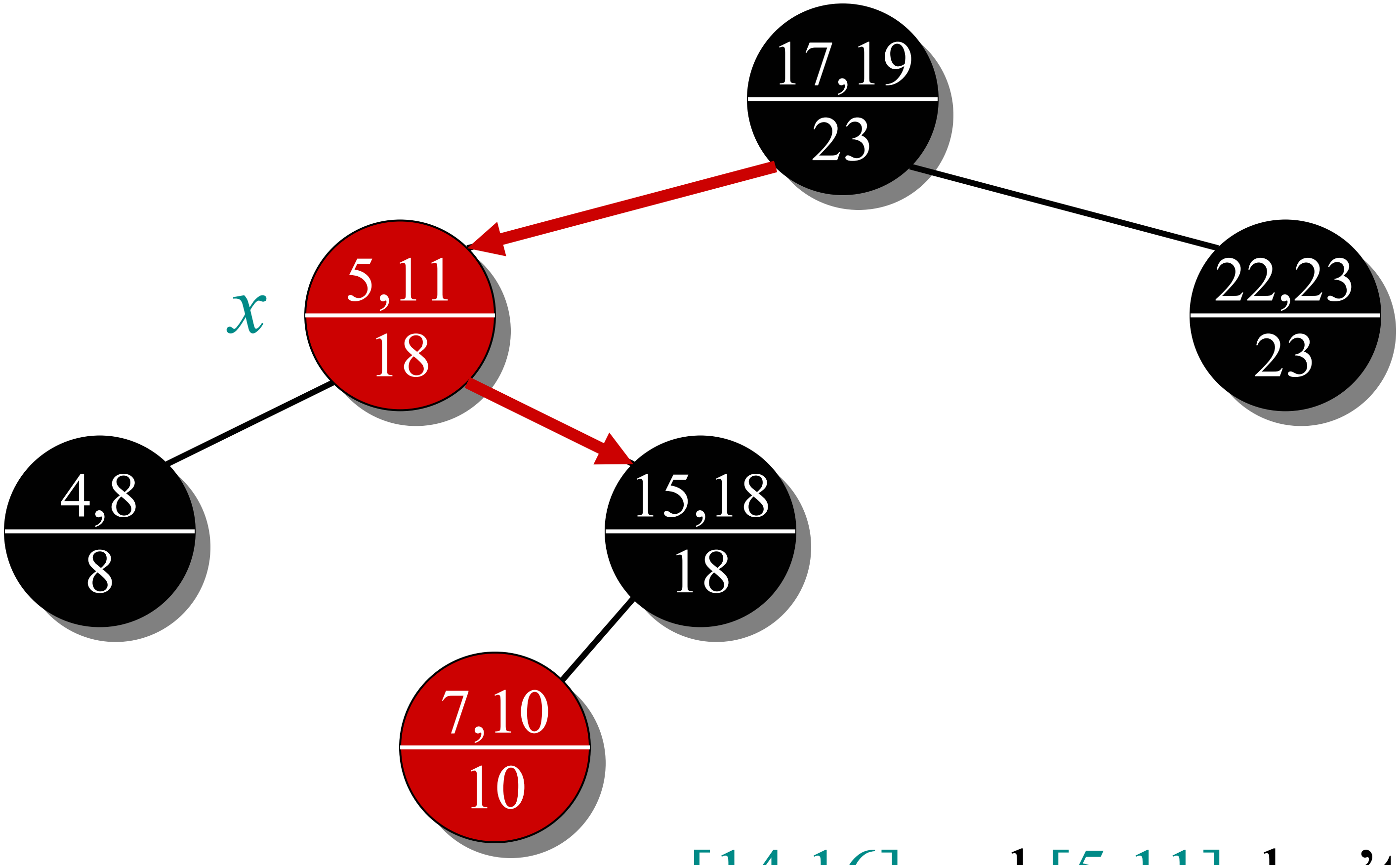


$x \leftarrow root$

[14,16] and [17,19] don’t overlap

$14 \leq 18 \Rightarrow x \leftarrow left[x]$

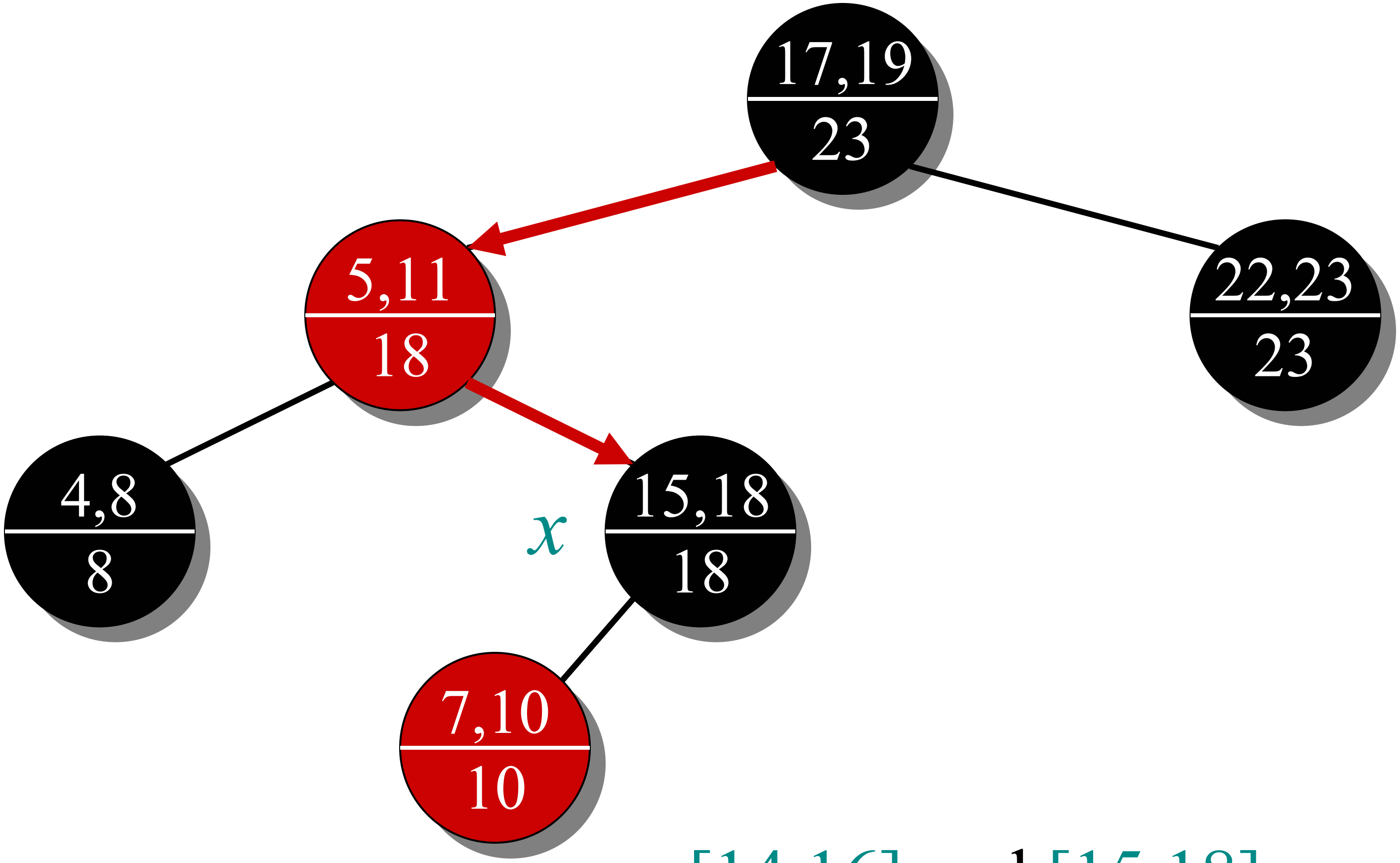
Example 1: INTERVAL-SEARCH([14,16])



[14,16] and [5,11] don’t overlap

$14 > 8 \Rightarrow x \leftarrow \mathit{right}[x]$

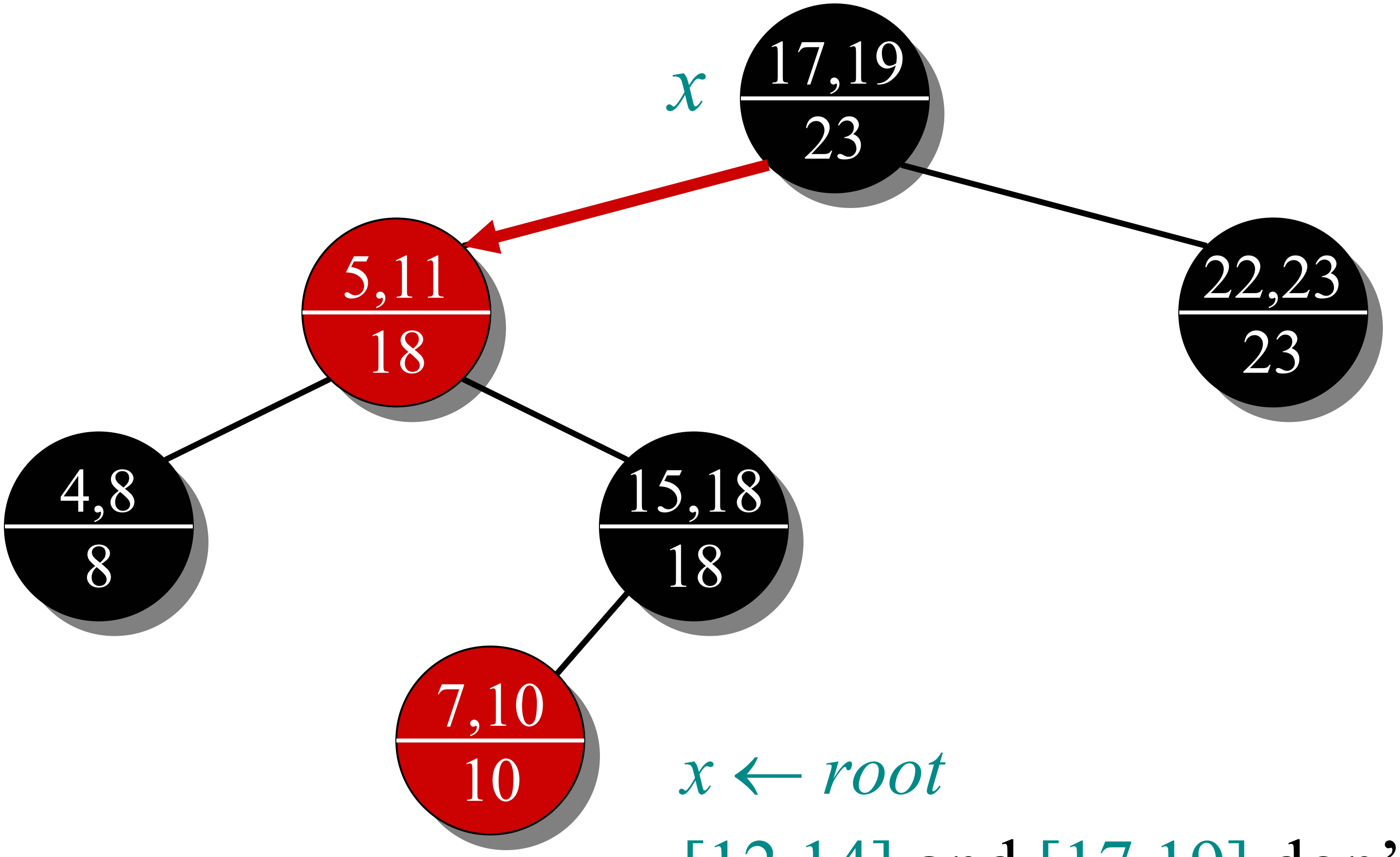
Example 1: INTERVAL-SEARCH([14,16])



[14,16] and [15,18] overlap

return [15,18]

Example 2: INTERVAL-SEARCH([12,14])

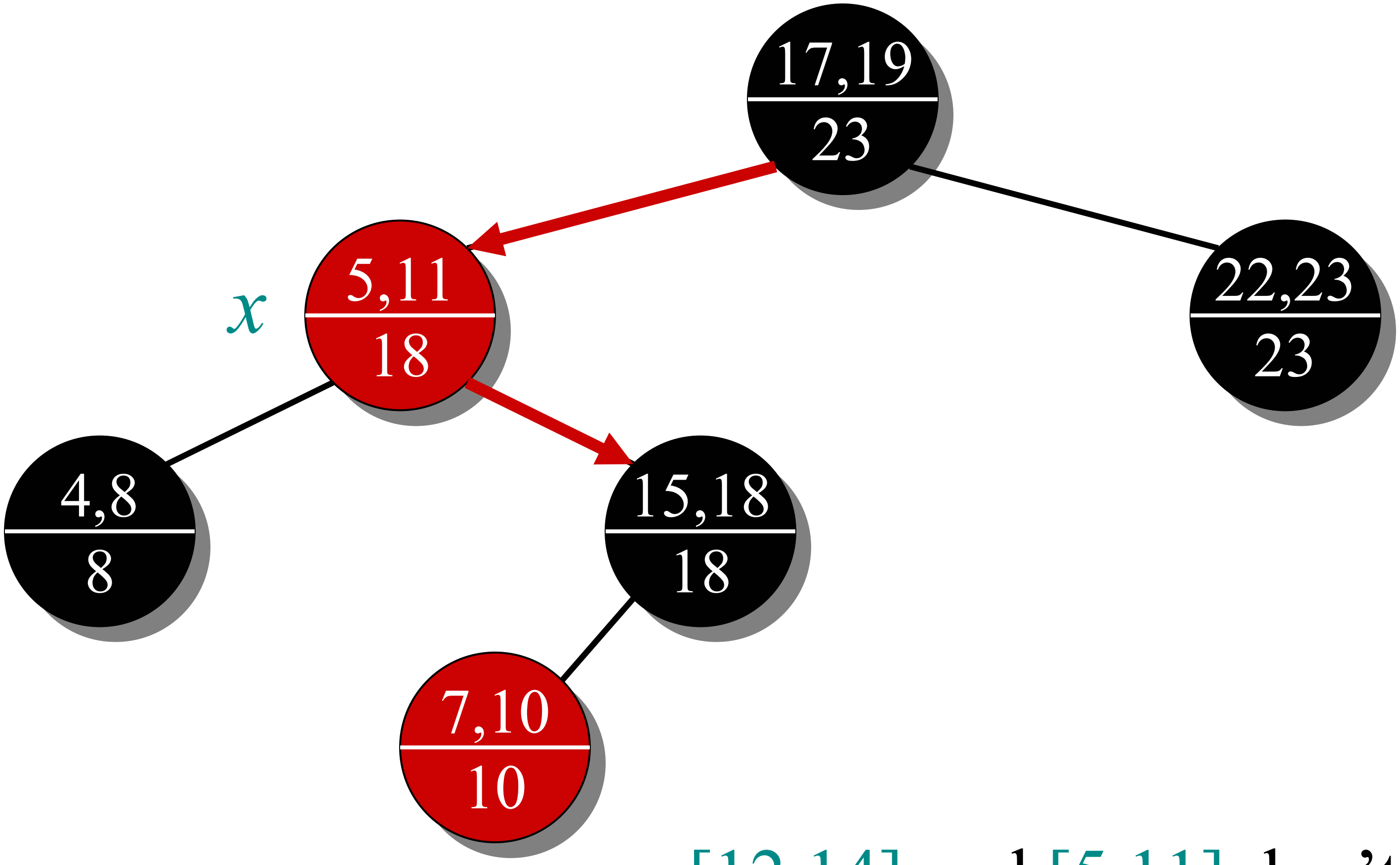


$x \leftarrow root$

[12,14] and [17,19] don't overlap

$12 \leq 18 \Rightarrow x \leftarrow left[x]$

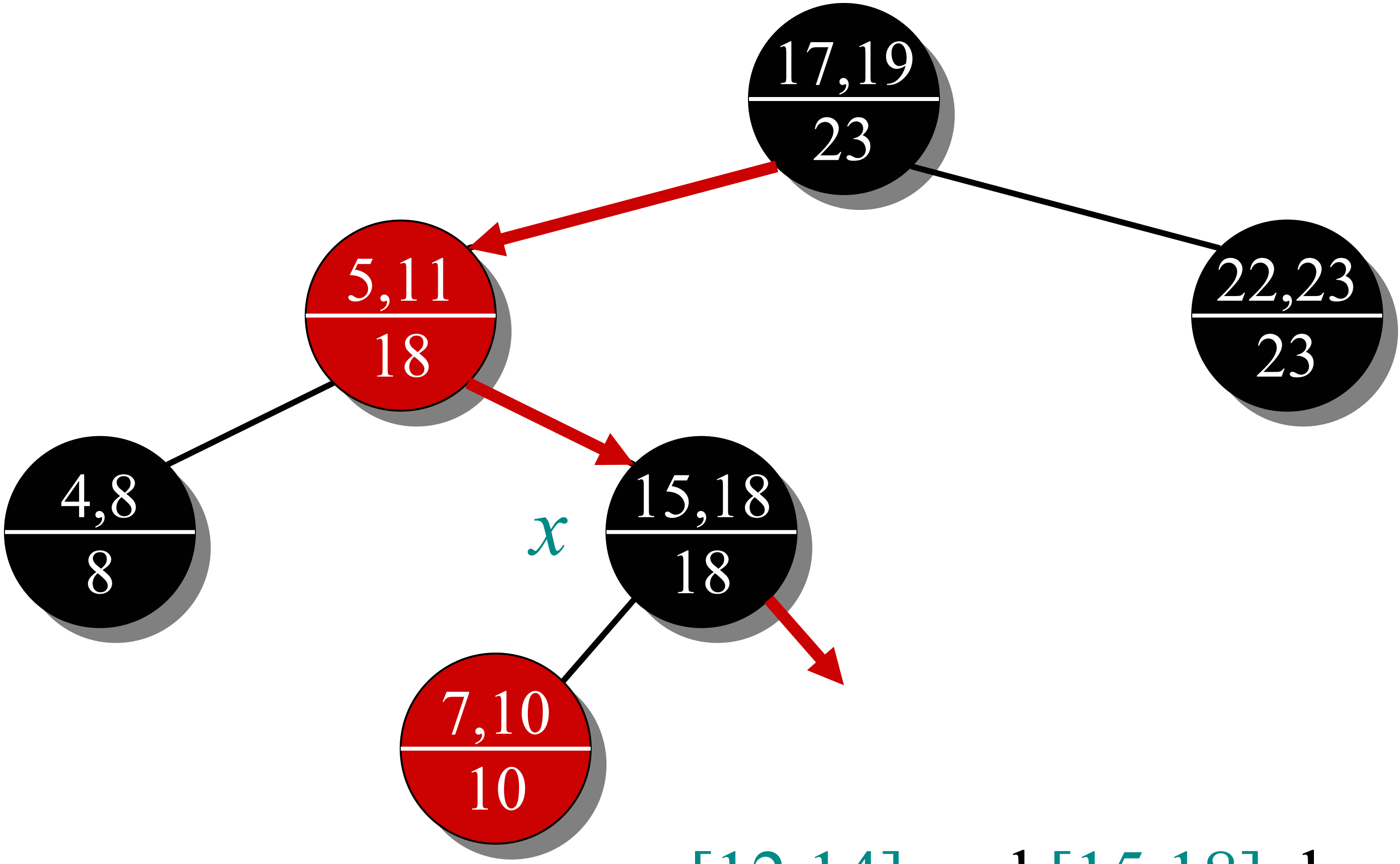
Example 2: INTERVAL-SEARCH([12,14])



[12,14] and [5,11] don't overlap

$12 > 8 \Rightarrow x \leftarrow right[x]$

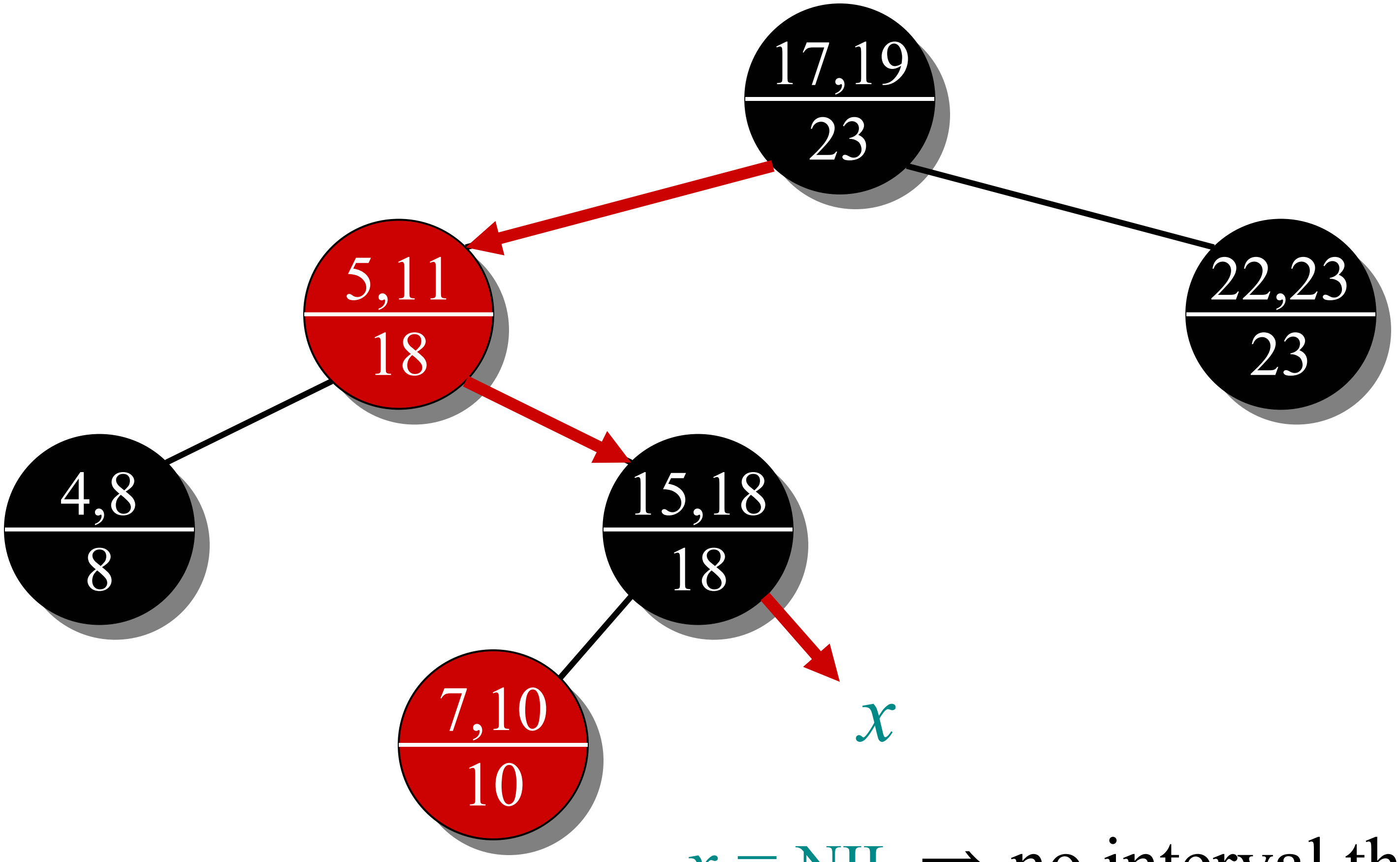
Example 2: INTERVAL-SEARCH([12,14])



[12,14] and [15,18] don’t overlap

$12 > 10 \Rightarrow x \leftarrow right[x]$

Example 2: INTERVAL-SEARCH([12,14])



x = NIL ⇒ no interval that overlaps [12,14] exists

Analysis

Time $= O(h) = O(\lg n)$, since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

List *all* overlapping intervals:

- Search, list, delete, repeat.
- Insert them all again at the end.

Time $= O(k \lg n)$, where k is the total number of overlapping intervals.

This is an ***output-sensitive*** bound.

Best algorithm to date: $O(k + \lg n)$.

Correctness

Theorem. Let L be the set of intervals in the left subtree of node x, and let R be the set of intervals in x's right subtree.

- If the search goes right, then

$$\{\, i' \in L : i' \text{ overlaps } i \,\} = \emptyset.$$

- If the search goes left, then

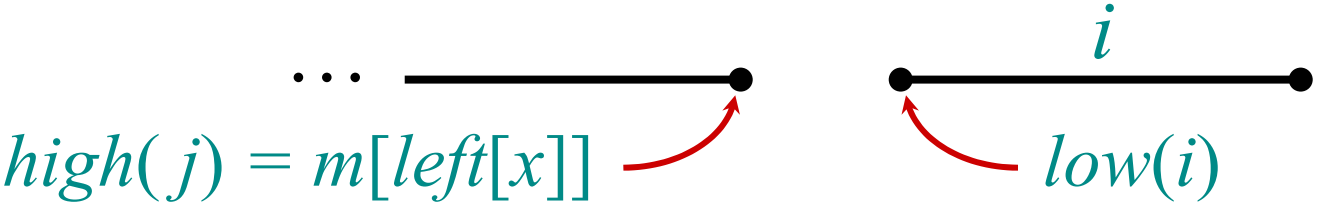
$$\{ i' \in L : i' \text{ overlaps } i \,\} = \emptyset$$
$$\Rightarrow \{ i' \in R : i' \text{ overlaps } i \,\} = \emptyset.$$

In other words, it's always safe to take only 1 of the 2 children: we'll either find something, or nothing was to be found.

Correctness proof

Proof. Suppose first that the search goes right.

- If $left[x] = \text{NIL}$, then we're done, since $L = \emptyset$.
- Otherwise, the code dictates that we must have $low[i] > m[left[x]]$. The value $m[left[x]]$ corresponds to the right endpoint of some interval $j \in L$, and no other interval in L can have a larger right endpoint than $high(j)$.



$\ldots$

i

$high(j) = m[left[x]]$

$low(i)$

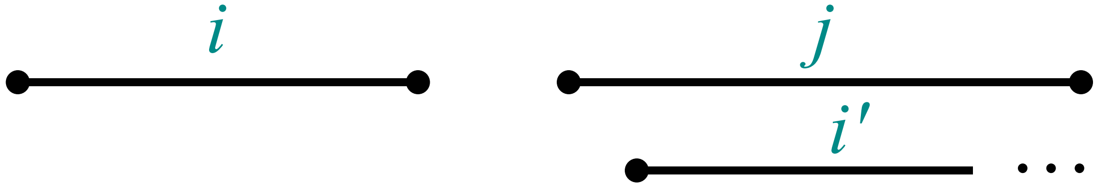
- Therefore, $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$.

Proof (continued)

Suppose that the search goes left, and assume that

$$\{i' \in L : i' \text{ overlaps } i\} = \varnothing.$$

- Then, the code dictates that $low[i] \le m[left[x]] = high[j]$ for some $j \in L$.
- Since $j \in L$, it does not overlap i, and hence $high[i] < low[j]$.
- But, the binary-search-tree property implies that for all $i' \in R$, we have $low[j] \le low[i']$.
- But then $\{i' \in R : i' \text{ overlaps } i\} = \varnothing$. □



Introduction to Algorithms

6.046J/18.401J/SMA5503

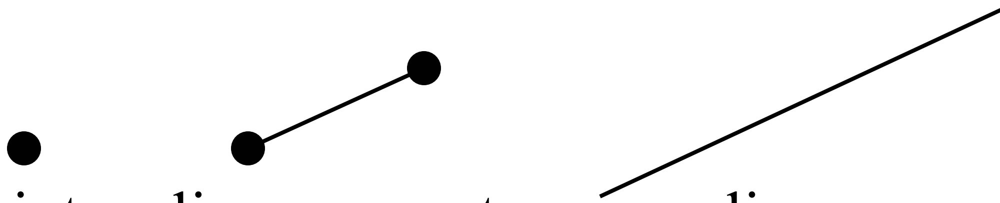
Lecture 12

Prof. Erik Demaine

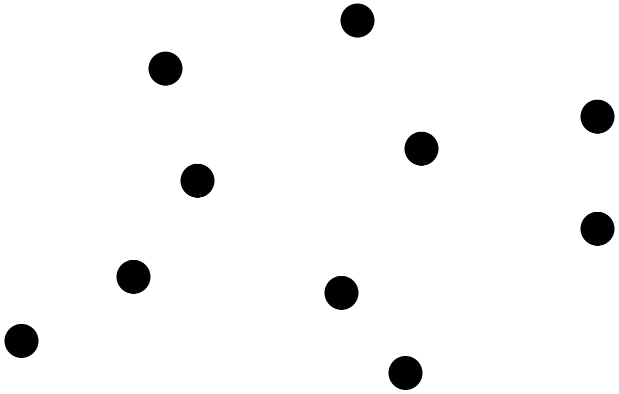
Computational geometry

Algorithms for solving "geometric problems" in 2D and higher.

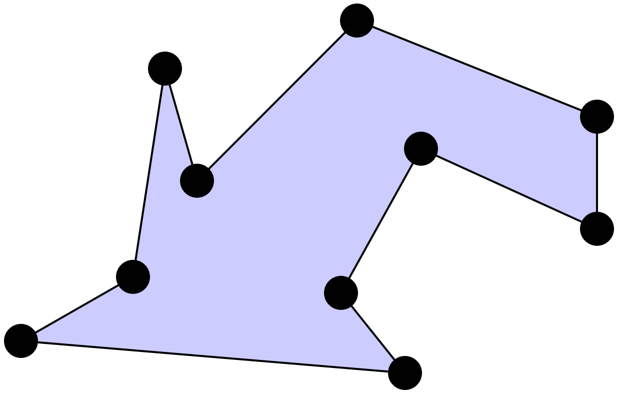
Fundamental objects:



Basic structures:



point set

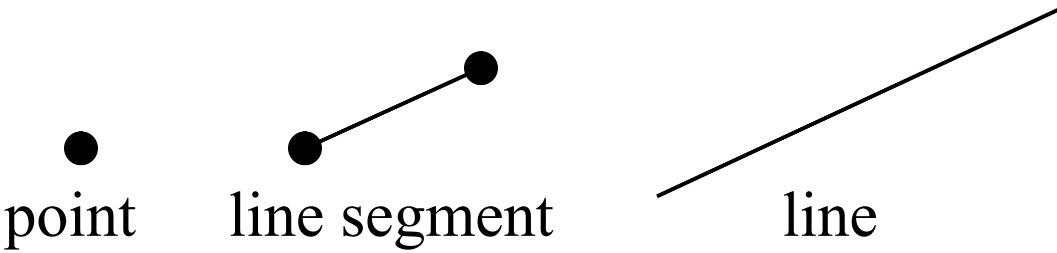


polygon

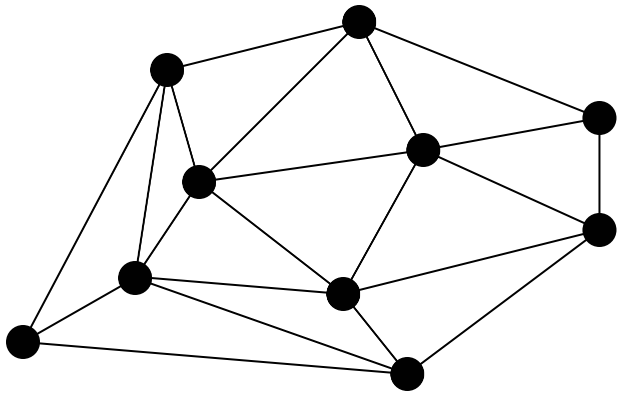
Computational geometry

Algorithms for solving “geometric problems” in 2D and higher.

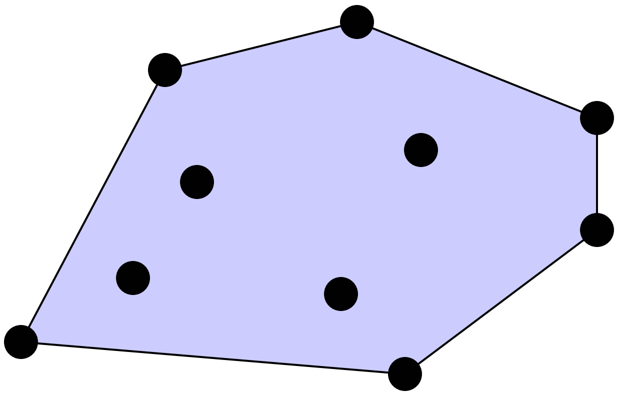
Fundamental objects:



Basic structures:



triangulation



convex hull

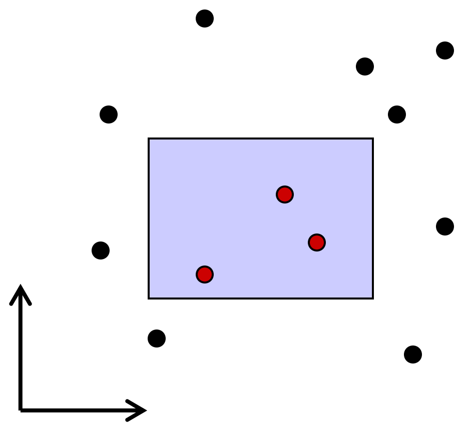
Orthogonal range searching

Input: *n* points in *d* dimensions

- E.g., representing a database of *n* records each with *d* numeric fields

Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.



Orthogonal range searching

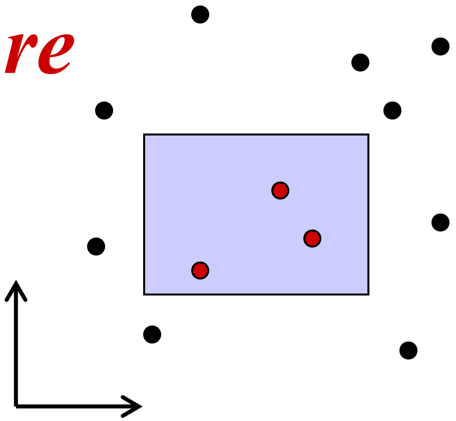
Input: n points in d dimensions

Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box

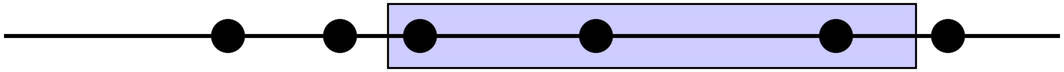
Goal: Preprocess points into a data structure to support fast queries

- Primary goal: ***Static data structure***
- In 1D, we will also obtain a dynamic data structure supporting insert and delete



1D range searching

In 1D, the query is an interval:

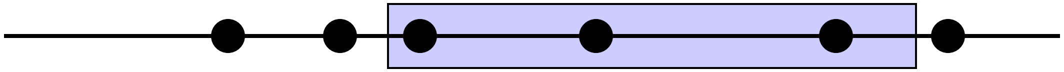


First solution using ideas we know:

- Interval trees
  - Represent each point *x* by the interval [*x*, *x*].
  - Obtain a dynamic structure that can list *k* answers in a query in $O(k \lg n)$ time.

1D range searching

In 1D, the query is an interval:



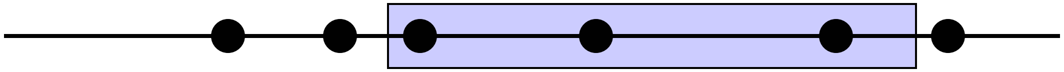
Second solution using ideas we know:

- Sort the points and store them in an array
  - Solve query by binary search on endpoints.
  - Obtain a static structure that can list k answers in a query in $O(k + \lg n)$ time.

Goal: Obtain a dynamic structure that can list k answers in a query in $O(k + \lg n)$ time.

1D range searching

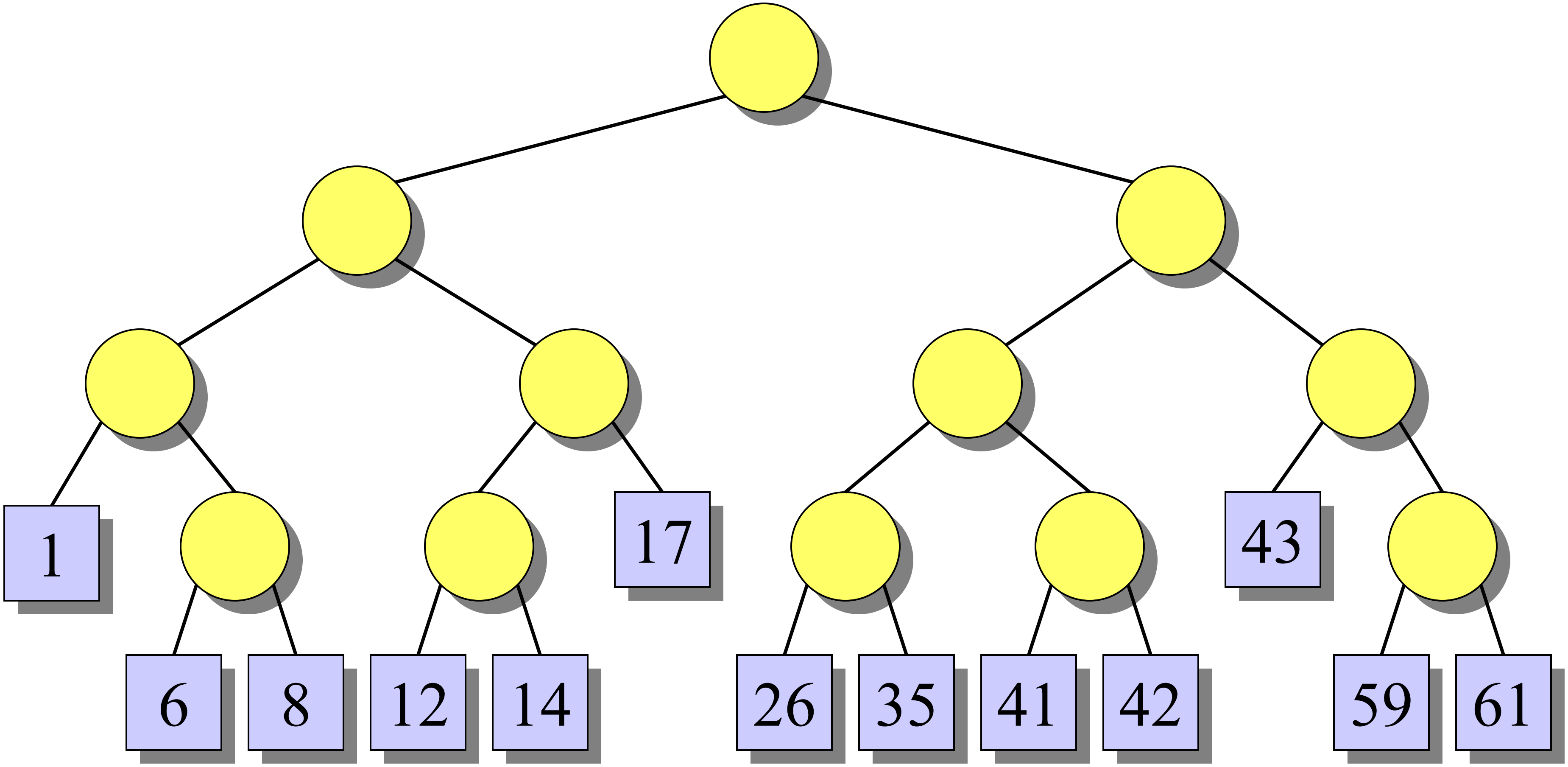
In 1D, the query is an interval:



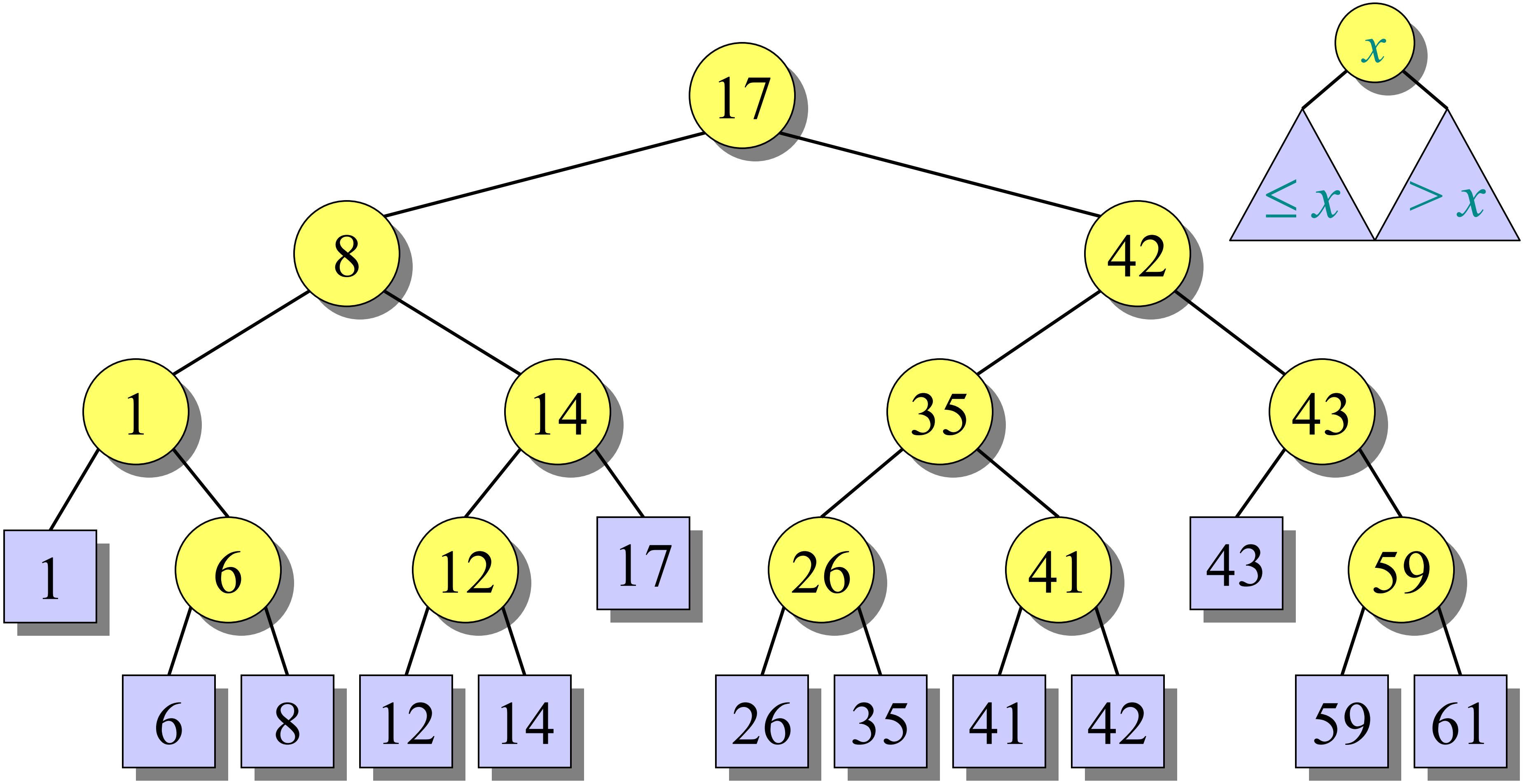
New solution that extends to higher dimensions:

- Balanced binary search tree
  - New organization principle:
    Store points in the *leaves* of the tree.
  - Internal nodes store copies of the leaves to satisfy binary search property:
    - Node x stores in $key[x]$ the maximum key of any leaf in the left subtree of x.

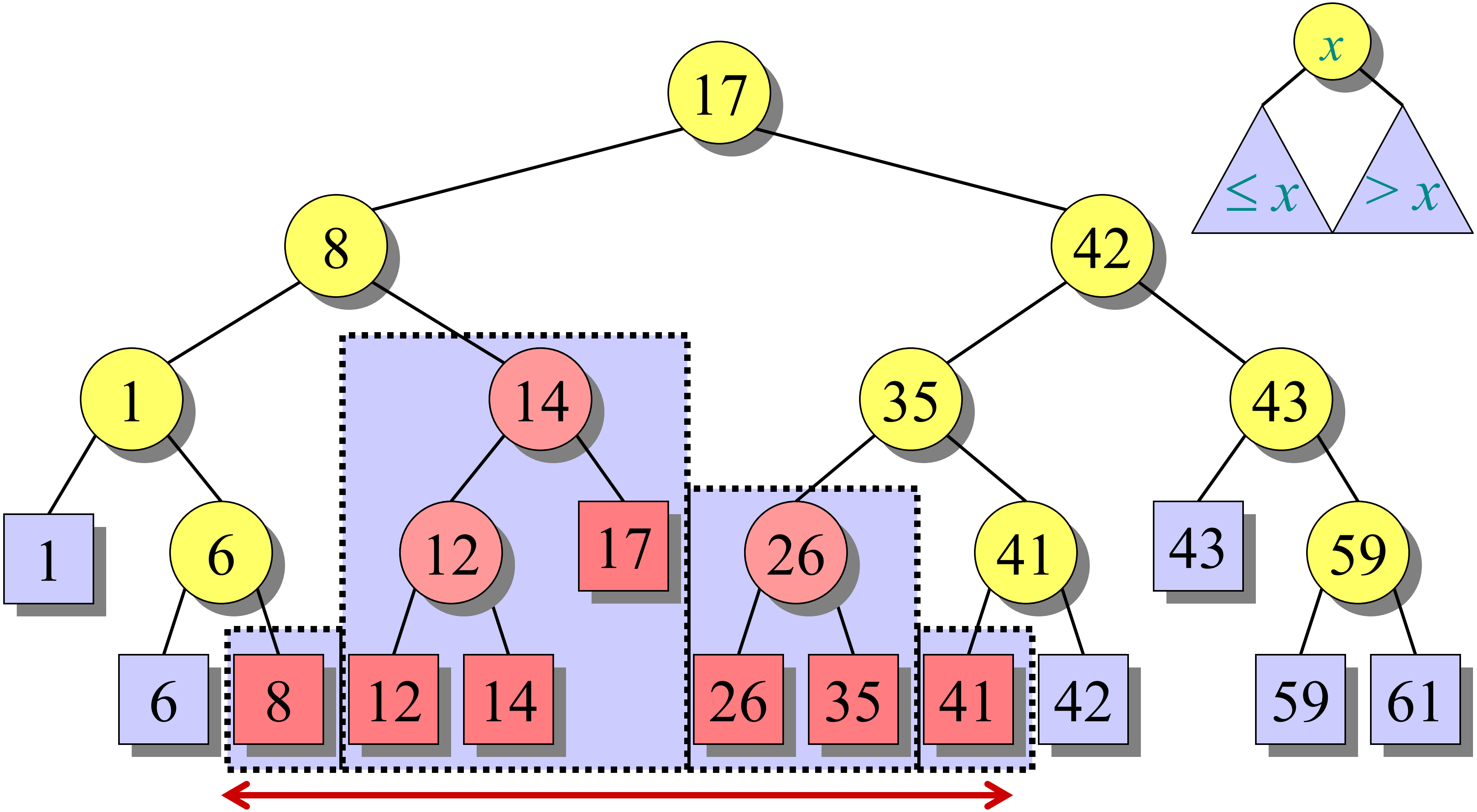
Example of a 1D range tree



Example of a 1D range tree

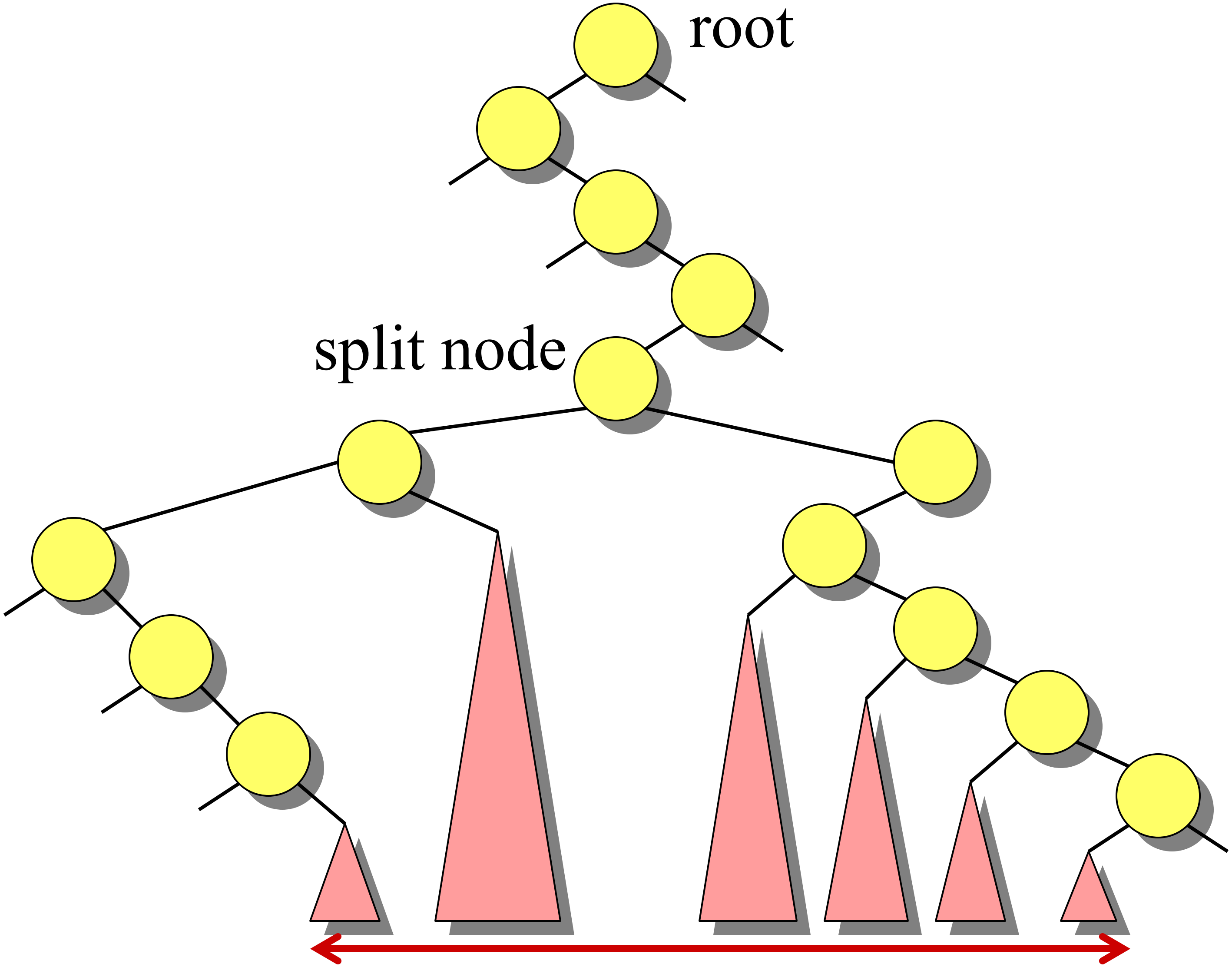


Example of a 1D range query



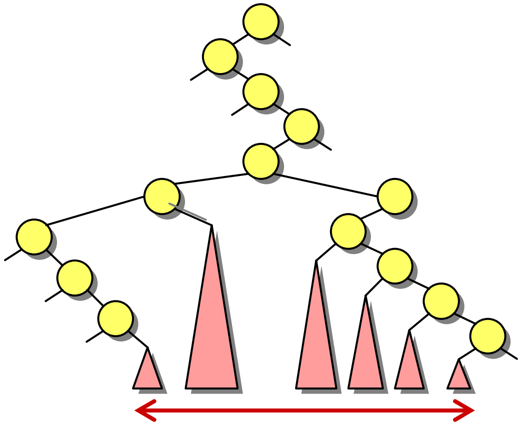
RANGE-QUERY([7, 41])

General 1D range query



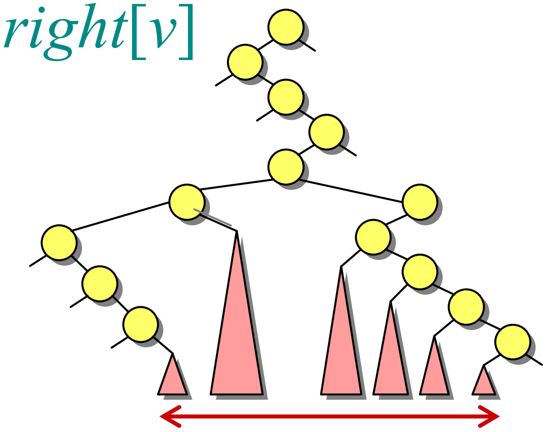
Pseudocode, part 1: Find the split node

```
1D-RANGE-QUERY($T$, $[x_1, x_2]$)
    $w \leftarrow \mathrm{root}[T]$
    while $w$ is not a leaf and ($x_2 \le key[w]$ or $key[w] < x_1$)
        do if $x_2 \le key[w]$
            then $w \leftarrow left[w]$
            else $w \leftarrow right[w]$
    ▷ $w$ is now the split node
    [traverse left and right from $w$ and report relevant subtrees]
```



Pseudocode, part 2: Traverse left and right from split node

```
1D-RANGE-QUERY($T$, [$x_1$, $x_2$])
    [find the split node]
    ▷ $w$ is now the split node
    if $w$ is a leaf
     then output the leaf $w$ if $x_1 \le key[w] \le x_2$
     else  $v \leftarrow left[w]$                                ▷ Left traversal
           while $v$ is not a leaf
              do if $x_1 \le key[v]$
                    then output the subtree rooted at $right[v]$
                         $v \leftarrow left[v]$
                    else  $v \leftarrow right[v]$
           output the leaf $v$ if $x_1 \le key[v] \le x_2$
           [symmetrically for right traversal]
```

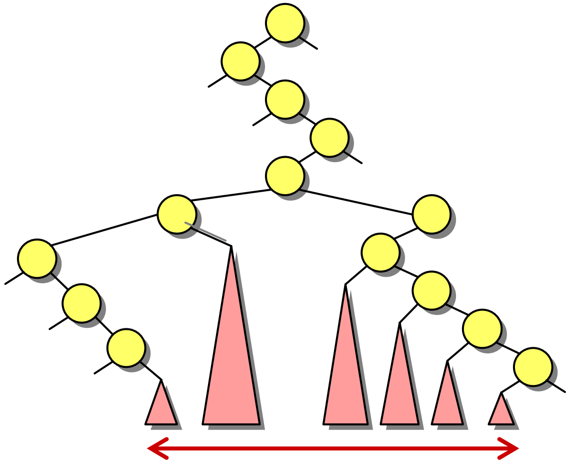


Analysis of 1D-RANGE-QUERY

Query time: Answer to range query represented by O(lg n) subtrees found in O(lg n) time.

Thus:

- Can test for points in interval in O(lg n) time.
- Can count points in interval in O(lg n) time if we augment the tree with subtree sizes.
- Can report the first k points in interval in O(k + lg n) time.



Space: O(n)

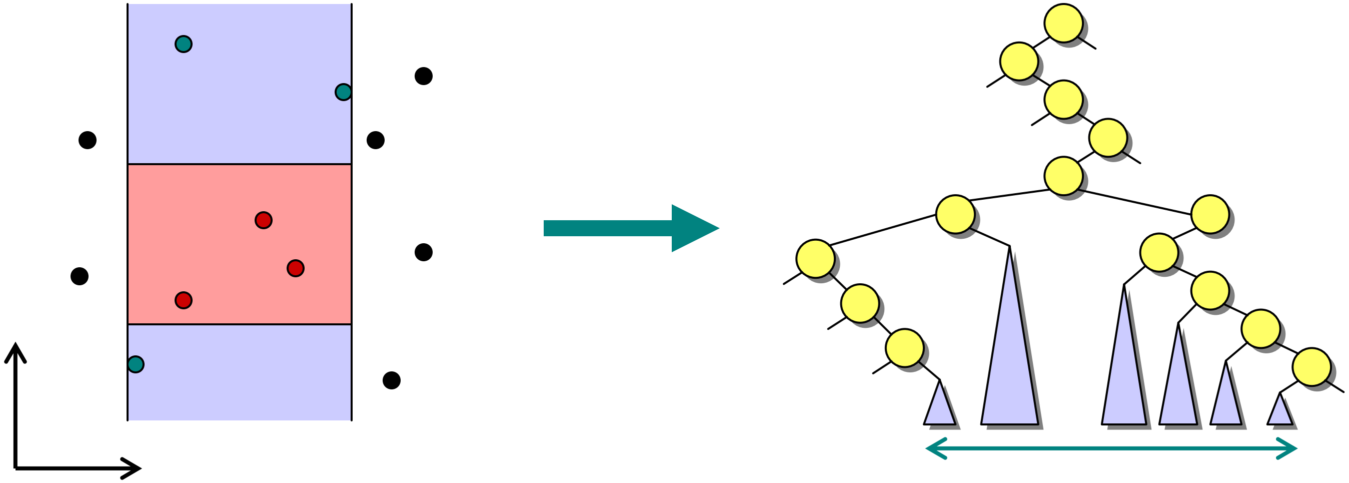
Preprocessing time: O(n lg n)

2D range trees

Store a ***primary*** 1D range tree for all the points based on x-coordinate.

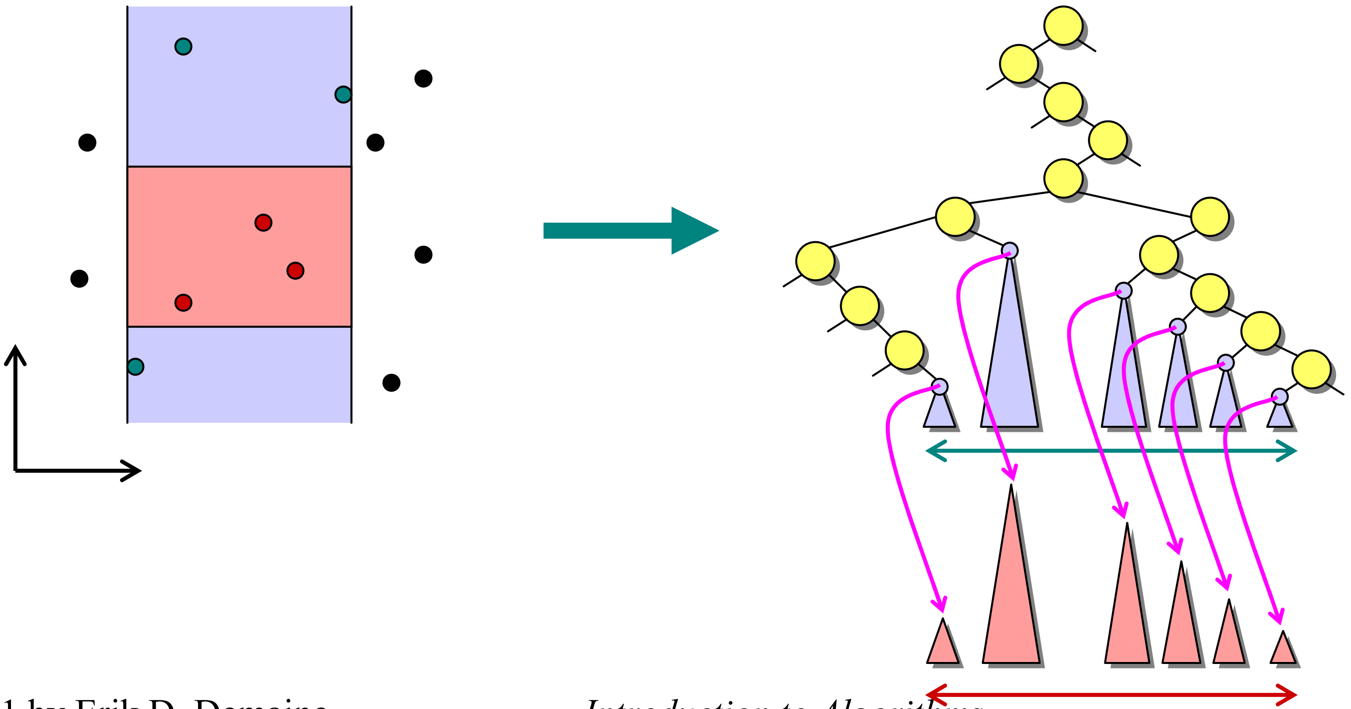
Thus in $O(\lg n)$ time we can find $O(\lg n)$ subtrees representing the points with proper x-coordinate.

How to restrict to points with proper y-coordinate?



2D range trees

Idea: In primary 1D range tree of *x*-coordinate, every node stores a ***secondary*** 1D range tree based on *y*-coordinate for all points in the subtree of the node. Recursively search within each.



Analysis of 2D range trees

Query time: In $O(\lg^2 n) = O((\lg n)^2)$ time, we can represent answer to range query by $O(\lg^2 n)$ subtrees. Total cost for reporting k points: $O(k + (\lg n)^2)$.

Space: The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root. Either way, we obtain that the space is $O(n \lg n)$.

Preprocessing time: $O(n \lg n)$

d-dimensional range trees

Each node of the secondary y-structure stores a tertiary z-structure representing the points in the subtree rooted at the node, etc.

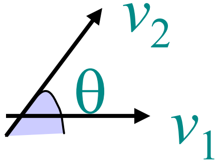
Query time: $O(k + \lg^d n)$ to report k points.
Space: $O(n \lg^{d-1} n)$
Preprocessing time: $O(n \lg^{d-1} n)$

Best data structure to date:
Query time: $O(k + \lg^{d-1} n)$ to report k points.
Space: $O(n (\lg n / \lg \lg n)^{d-1})$
Preprocessing time: $O(n \lg^{d-1} n)$

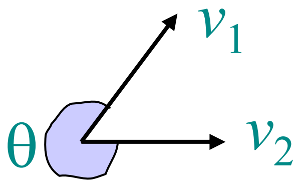
Primitive operations: Crossproduct

Given two vectors $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$, is their counterclockwise angle θ

- ***convex*** ($< 180^\circ$),
- ***reflex*** ($> 180^\circ$), or
- borderline (0 or 180°)?



convex



reflex

Crossproduct $v_1 \times v_2 = x_1 y_2 - y_1 x_2$

$$= |v_1|\,|v_2| \sin\theta\ .$$

Thus, $\operatorname{sign}(v_1 \times v_2) = \operatorname{sign}(\sin\theta)$ > 0 if θ convex,

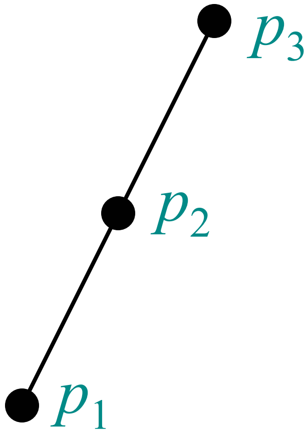
< 0 if θ reflex,

$= 0$ if θ borderline.

Primitive operations: Orientation test

Given three points p_1, p_2, p_3 are they

- in ***clockwise (cw) order***,
- in ***counterclockwise (ccw) order***, or
- ***collinear***?



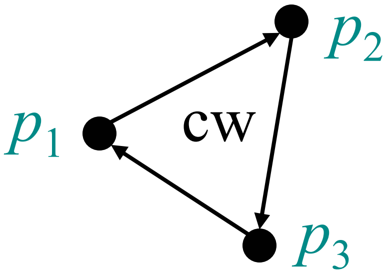
collinear

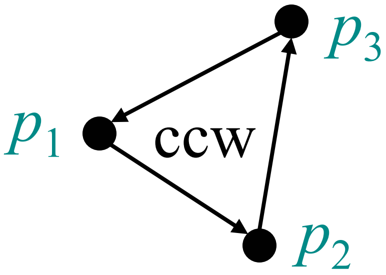
$$(p_2 - p_1) \times (p_3 - p_1)$$

> 0 if ccw

< 0 if cw

$= 0$ if collinear

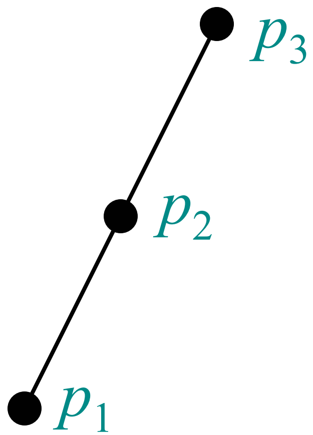




Primitive operations: Sidedness test

Given three points p_1, p_2, p_3 are they

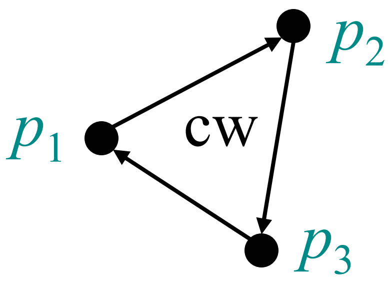
- in ***clockwise (cw) order***,
- in ***counterclockwise (ccw) order***, or
- ***collinear***?

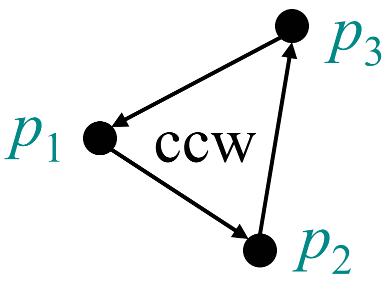


collinear

Let L be the oriented line from p_1 to p_2.
Equivalently, is the point p_3

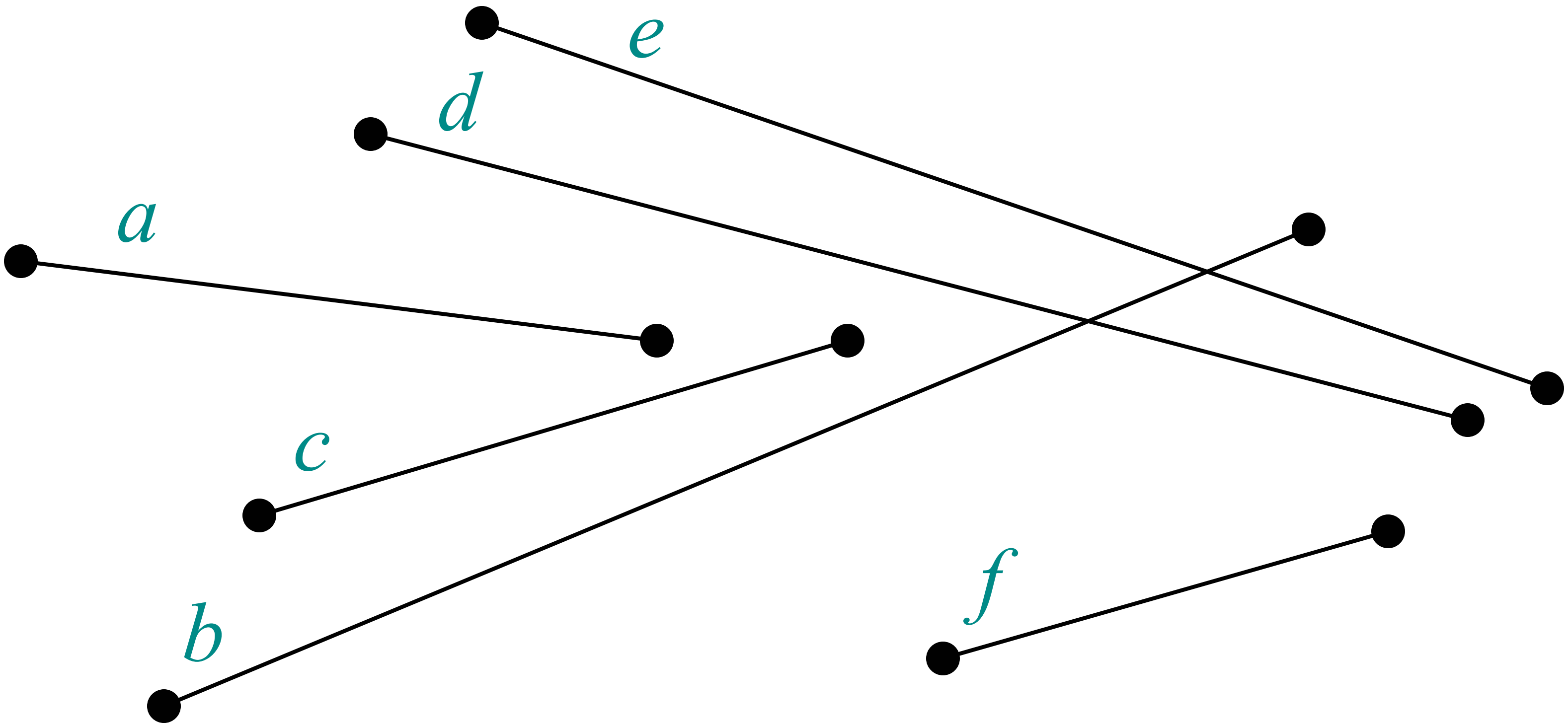
- ***right*** of L,
- ***left*** of L, or
- ***on*** L?





Line-segment intersection

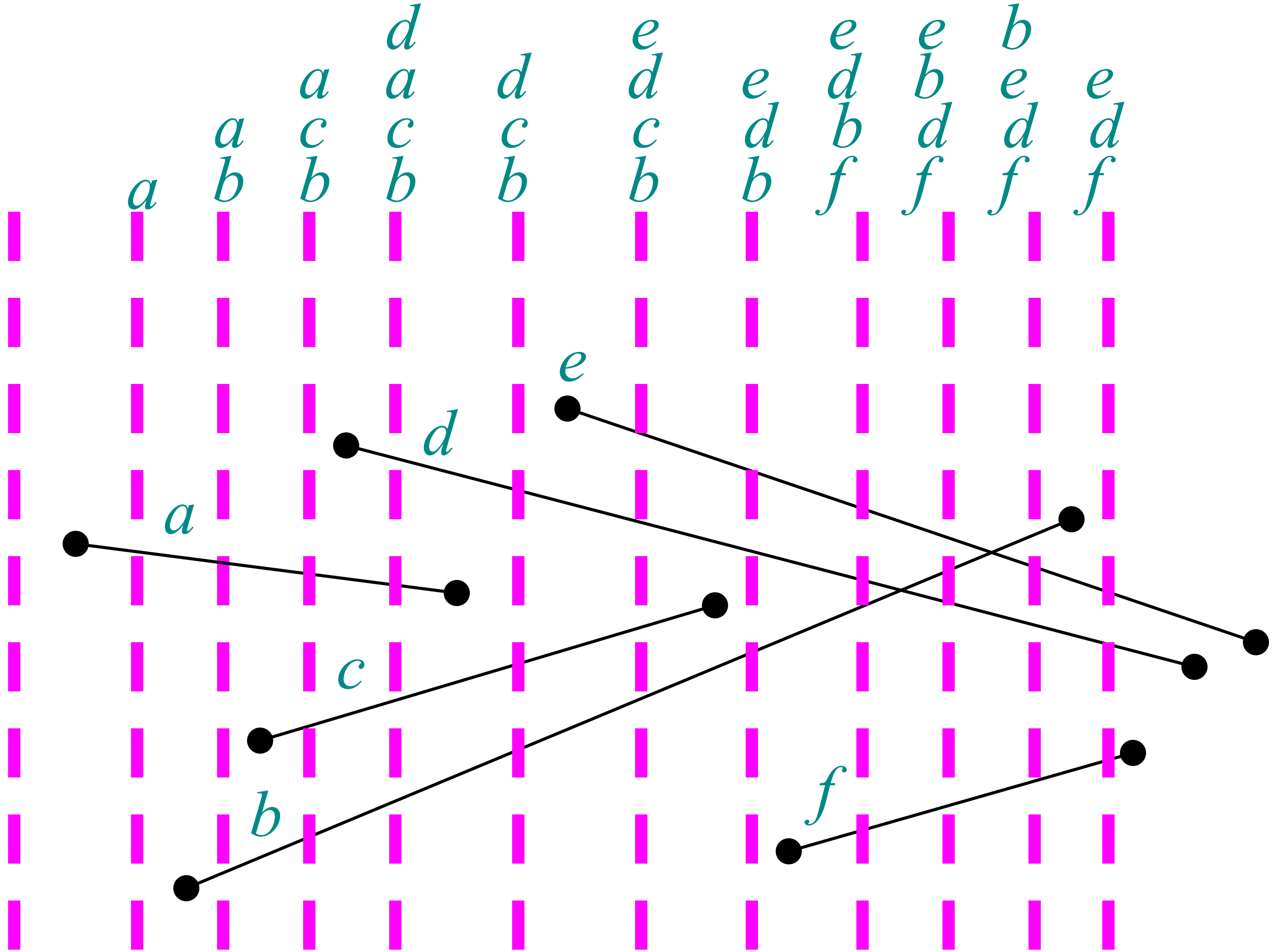
Given n line segments, does any pair intersect?
Obvious algorithm: $O(n^2)$.



Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing x-coordinate with time).
- Maintain dynamic set S of segments that intersect the sweep line, ordered (tentatively) by y-coordinate of intersection.
- Order changes when
    - new segment is encountered,
    - existing segment finishes, or

      } segment endpoints
    - two segments cross
- Key ***event points*** are therefore segment endpoints.


a
a b
a c b
d a c b
d c b
e d c b
e d b
e d b f
e b d f
b e d f
e d f
e
d
a
c
b
f

Sweep-line algorithm

Process event points in order by sorting segment endpoints by x-coordinate and looping through:

- For a left endpoint of segment s:
  - Add segment s to dynamic set S.
  - Check for intersection between s and its neighbors in S.
- For a right endpoint of segment s:
  - Remove segment s from dynamic set S.
  - Check for intersection between the neighbors of s in S.

Analysis

Use red-black tree to store dynamic set S.

Total running time: $O(n \lg n)$.

Correctness

Theorem: If there is an intersection, the algorithm finds it.

Proof: Let X be the leftmost intersection point. Assume for simplicity that

- only two segments s_1, s_2 pass through X, and
- no two points have the same x-coordinate.

At some point before we reach X, s_1 and s_2 become consecutive in the order of S. Either initially consecutive when s_1 or s_2 inserted, or became consecutive when another deleted. □

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 13

Prof. Erik Demaine

Fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u - 1\}$ of size u subject to these operations:

- **INSERT**$(x \in U \setminus S)$: Add x to S.
- **DELETE**$(x \in S)$: Remove x from S.
- **SUCCESSOR**$(x \in U)$: Find the next element in S larger than any element x of the universe U.
- **PREDECESSOR**$(x \in U)$: Find the previous element in S smaller than x.

Solutions to fixed-universe successor problem

Goal: Maintain a dynamic subset S of size n of the universe $U = \{0, 1, \ldots, u - 1\}$ of size u subject to INSERT, DELETE, SUCCESSOR, PREDECESSOR.

- Balanced search trees can implement operations in $O(\lg n)$ time, without fixed-universe assumption.
- In 1975, Peter van Emde Boas solved this problem in $O(\lg \lg u)$ time per operation.
  - If u is only polynomial in n, that is, $u = O(n^c)$, then $O(\lg \lg n)$ time per operation-- exponential speedup!

O(lg lg u)?!

Where could a bound of O(lg lg u) arise?

- Binary search over O(lg u) things
- $T(u) = T(\sqrt{u}) + O(1)$

  $T'(\lg u) = T'((\lg u)/2) + O(1)$

  $= O(\lg \lg u)$

(1) Starting point: Bit vector

Bit vector v stores, for each $x \in U$,

$$v_x = \begin{cases} 1 \text{ if } x \in S \\ 0 \text{ if } x \notin S \end{cases}$$

Example: $u = 16$; $n = 4$; $S = \{1, 9, 10, 15\}$.

0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insert/Delete run in O(1) time.

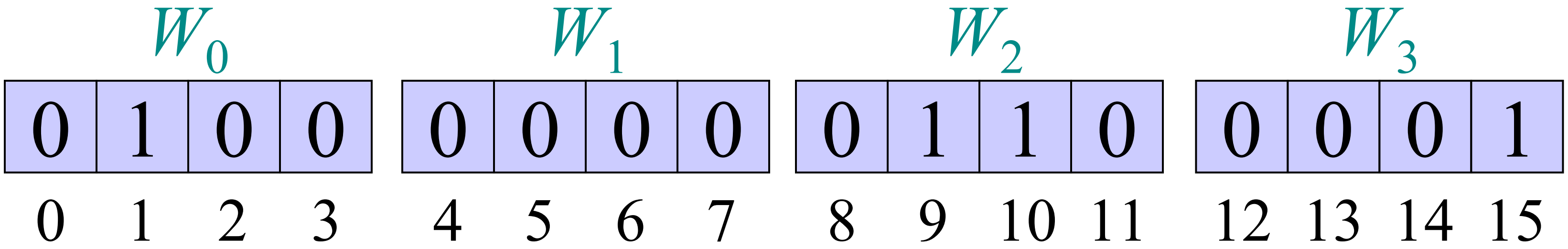
Successor/Predecessor run in O(u) worst-case time.

(2) Split universe into widgets

Carve universe of size u into $\sqrt{u}$ widgets

$W_0, W_1, \ldots, W_{\sqrt{u}-1}$ each of size $\sqrt{u}$.

Example: $u = 16, \sqrt{u} = 4$.



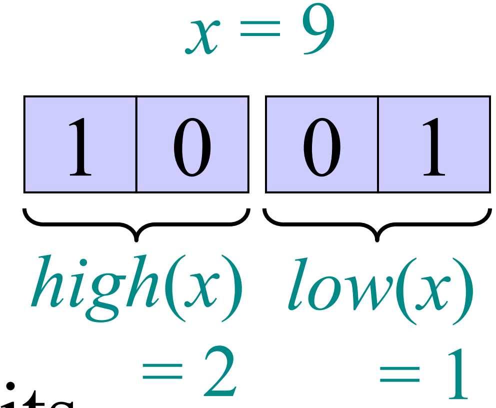
W_0				W_1				W_2				W_3			
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(2) Split universe into widgets

Carve universe of size u into $\sqrt{u}$ widgets $W_0, W_1, \ldots, W_{\sqrt{u}-1}$ each of size $\sqrt{u}$.

W_0 represents $0, 1, \ldots, \sqrt{u}-1 \in U$;
W_1 represents $\sqrt{u}, \sqrt{u}+1, \ldots, 2\sqrt{u}-1 \in U$;
$\vdots$
W_i represents $i\sqrt{u}, i\sqrt{u}+1, \ldots, (i+1)\sqrt{u}-1 \in U$;
$\vdots$
$W_{\sqrt{u}-1}$ represents $u-\sqrt{u}, u-\sqrt{u}+1, \ldots, u-1 \in U$.

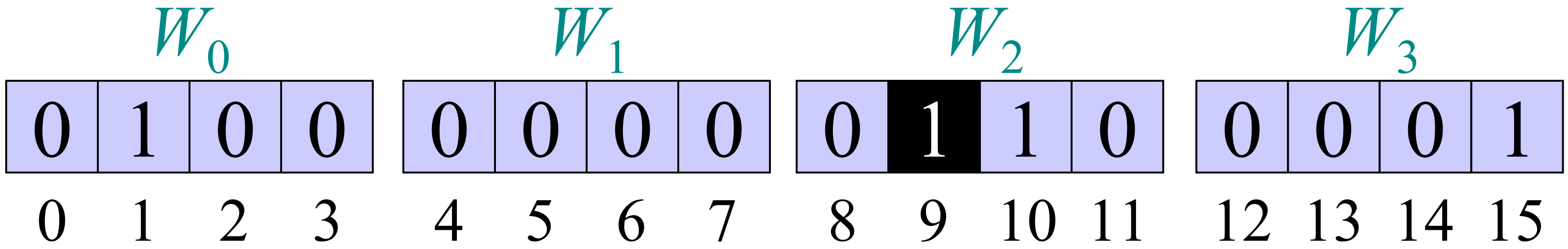
(2) Split universe into widgets



Define $high(x) \geq 0$ and $low(x) \geq 0$ so that $x = high(x)\sqrt{u} + low(x)$.

That is, if we write $x \in U$ in binary, $high(x)$ is the high-order half of the bits, and $low(x)$ is the low-order half of the bits.

For $x \in U$, $high(x)$ is index of widget containing x and $low(x)$ is the index of x within that widget.



(2) Split universe into widgets

```
INSERT($x$)
    insert $x$ into widget $W_{high(x)}$ at position $low(x)$.
    mark $W_{high(x)}$ as nonempty.
```

Running time $T(n) = O(1)$.

(2) Split universe into widgets

```
SUCCESSOR($x$)
    look for successor of $x$ within widget $W_{high(x)}$     } $O(\sqrt{u})$
        starting after position $low(x)$.
    if successor found
     then return it
     else find smallest $i > high(x)$                         } $O(\sqrt{u})$
                for which $W_i$ is nonempty.
          return smallest element in $W_i$                    } $O(\sqrt{u})$
```

Running time $T(u) = O(\sqrt{u})$.

Revelation

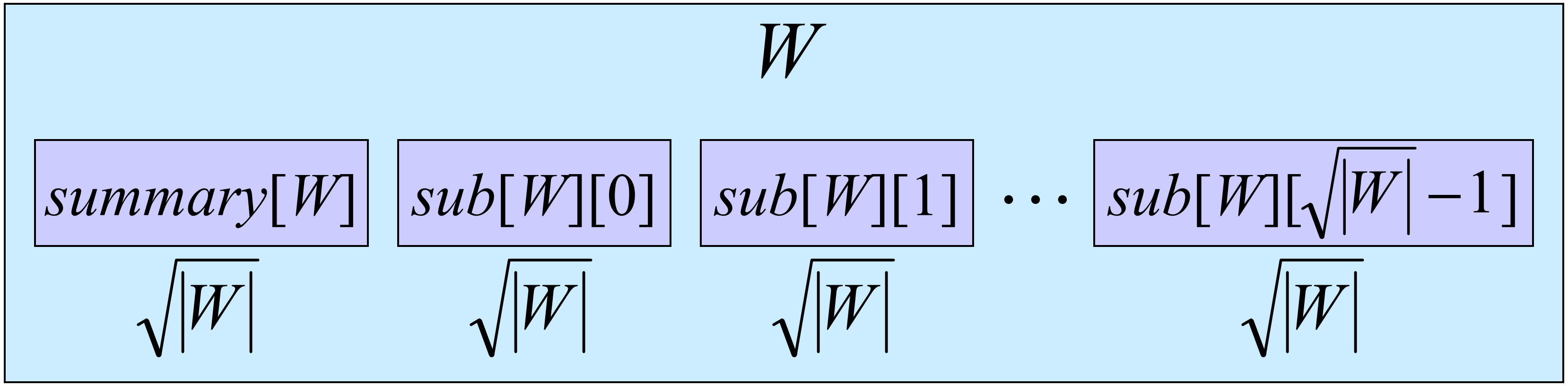
```
SUCCESSOR($x$)
    look for successor of $x$ within widget $W_{high(x)}$        } recursive
        starting after position $low(x)$.                          successor
    if successor found
     then return it
     else find smallest $i > high(x)$                            } recursive
                for which $W_i$ is nonempty.                       successor
            return smallest element in $W_i$                     } recursive
                                                                   successor
```

(3) Recursion

Represent universe by ***widget*** of size u.

Recursively split each widget W of size $|W|$ into $\sqrt{|W|}$ ***subwidgets*** $sub[W][0]$, $sub[W][1]$, …, $sub[W][\sqrt{|W|}-1]$ each of size $\sqrt{|W|}$.

Store a ***summary widget*** $summary[W]$ of size $\sqrt{|W|}$ representing which subwidgets are nonempty.



W

$summary[W]$ $\sqrt{|W|}$

$sub[W][0]$ $\sqrt{|W|}$

$sub[W][1]$ $\sqrt{|W|}$

$\cdots$

$sub[W][\sqrt{|W|}-1]$ $\sqrt{|W|}$

(3) Recursion

Define $high(x) \ge 0$ and $low(x) \ge 0$ so that $x = high(x)\sqrt{|W|} + low(x)$.

```
INSERT($x$, $W$)
    if $sub[W][high(x)]$ is empty
     then INSERT($high(x)$, $summary[W]$)
    INSERT($low(x)$, $sub[W][high(x)]$)
```

Running time $T(u) = 2\,T(\sqrt{u}) + O(1)$

$$T'(\lg u) = 2\,T'((\lg u) / 2) + O(1)$$

$$= O(\lg u)\ .$$

(3) Recursion

```
SUCCESSOR$(x, W)$
    $j \leftarrow$ SUCCESSOR$(low(x), sub[W][high(x)])$        $\}\ T(\sqrt{u})$
    if $j < \infty$
      then return $high(x)\sqrt{|W|} + j$
      else $i \leftarrow$ SUCCESSOR$(high(x), summary[W])$     $\}\ T(\sqrt{u})$
           $j \leftarrow$ SUCCESSOR$(-\infty, sub[W][i])$      $\}\ T(\sqrt{u})$
           return $i\sqrt{|W|} + j$
```

Running time $T(u) = 3\,T(\sqrt{u}) + O(1)$

$$T'(\lg u) = 3\,T'((\lg u)/2) + O(1)$$

$$= O((\lg u)^{\lg 3})\ .$$

Improvements

Need to reduce INSERT and SUCCESSOR down to 1 recursive call each.

- 1 call: $T(u) = 1\,T(\sqrt{u}) + O(1)$
  $= O(\lg \lg n)$
- 2 calls: $T(u) = 2\,T(\sqrt{u}) + O(1)$
  $= O(\lg n)$
- 3 calls: $T(u) = 3\,T(\sqrt{u}) + O(1)$
  $= O((\lg u)^{\lg 3})$

We're closer to this goal than it may seem!

Recursive calls in successor

If x has a successor within $sub[W][high(x)]$, then there is only 1 recursive call to SUCCESSOR. Otherwise, there are 3 recursive calls:

- SUCCESSOR$(low(x), sub[W][high(x)])$ discovers that $sub[W][high(x)]$ hasn't successor.
- SUCCESSOR$(high(x), summary[W])$ finds next nonempty subwidget $sub[W][i]$.
- SUCCESSOR$(-\infty, sub[W][i])$ finds smallest element in subwidget $sub[W][i]$.

Reducing recursive calls in successor

If x has no successor within $sub[W][high(x)]$, there are 3 recursive calls:

- Successor$(low(x), sub[W][high(x)])$ discovers that $sub[W][high(x)]$ hasn't successor.
  - Could be determined using the ***maximum value*** in the subwidget $sub[W][high(x)]$.
- Successor$(high(x), summary[W])$ finds next nonempty subwidget $sub[W][i]$.
- Successor$(-\infty, sub[W][i])$ finds ***minimum element*** in subwidget $sub[W][i]$.

(4) Improved successor

```
INSERT$(x, W)$
    if $sub[W][high(x)]$ is empty
     then INSERT$(high(x), summary[W])$
    INSERT$(low(x), sub[W][high(x)])$
    if $x < min[W]$ then $min[W] \leftarrow x$
    if $x > max[W]$ then $max[W] \leftarrow x$
```

new (augmentation)

Running time $T(u) = 2\,T(\sqrt{u}) + O(1)$

$$T'(\lg u) = 2\,T'((\lg u)/2) + O(1)$$
$$= O(\lg u)\,.$$

(4) Improved successor

```
SUCCESSOR$(x, W)$
    if $low(x) < max[sub[W][high(x)]]$
     then $j \gets$ SUCCESSOR$(low(x), sub[W][high(x)])$   $\}\ T(\sqrt{u})$
          return $high(x)\sqrt{|W|} + j$
    else $i \gets$ SUCCESSOR$(high(x), summary[W])$         $\}\ T(\sqrt{u})$
         $j \gets min[sub[W][i]]$
         return $i\sqrt{|W|} + j$
```

Running time $T(u) = 1\ T(\sqrt{u}) + O(1)$

$$= O(\lg \lg u)\ .$$

Recursive calls in insert

If $sub[W][high(x)]$ is already in $summary[W]$, then there is only 1 recursive call to INSERT.

Otherwise, there are 2 recursive calls:

- INSERT$(high(x), summary[W])$
- INSERT$(low(x), sub[W][high(x)])$

Idea: We know that $sub[W][high(x)])$ is empty.

Avoid second recursive call by specially storing a widget containing just 1 element.

Specifically, do not store *min* recursively.

(5) Improved insert

```
INSERT($x$, $W$)
    if $x < min[W]$ then exchange $x \leftrightarrow min[W]$
    if $sub[W][high(x)]$ is nonempty, that is,
      $min[sub[W][high(x)] \neq \text{NIL}$
     then INSERT($low(x)$, $sub[W][high(x)]$)
     else  $min[sub[W][high(x)]] \leftarrow low(x)$
           INSERT($high(x)$, $summary[W]$)
    if $x > max[W]$ then $max[W] \leftarrow x$
```

Running time $T(u) = 1\ T(\sqrt{u}) + O(1)$

$$= O(\lg \lg u)\ .$$

(5) Improved insert

```
SUCCESSOR$(x, W)$
    if $x < min[W]$ then return $min[W]$   $\}$ new
    if $low(x) < max[sub[W][high(x)]]$   $\}\ T(\sqrt{u})$
     then $j \leftarrow$ SUCCESSOR$(low(x), sub[W][high(x)])$
          return $high(x)\sqrt{|W|} + j$   $\}\ T(\sqrt{u})$
     else $i \leftarrow$ SUCCESSOR$(high(x), summary[W])$
          $j \leftarrow min[sub[W][i]]$
          return $i\sqrt{|W|} + j$
```

Running time $T(u) = 1\ T(\sqrt{u}) + O(1)$

$$= O(\lg \lg u)\ .$$

Deletion

```
DELETE($x$, $W$)
    if $min[W] = \text{NIL}$ or $x < min[W]$ then return
    if $x = min[W]$
     then $i \leftarrow min[summary[\text{W}]]$
          $x \leftarrow \text{i}\sqrt{|W|} + min[sub[W][i]]$
          $min[W] \leftarrow x$
    DELETE($low(x)$, $sub[W][high(x)]$)
    if $sub[W][high(x)]$ is now empty, that is,
      $min[sub[W][high(x)] = \text{NIL}$
    then DELETE($high(x)$, $summary[W]$)
          (in this case, the first recursive call was cheap)
```

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 14

Prof. Charles E. Leiserson

How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

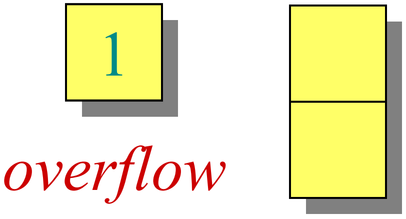
Problem: What if we don't know the proper size in advance?

Solution: ***Dynamic tables.***

IDEA: Whenever the table overflows, "grow" it by allocating (via **`malloc`** or **`new`**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

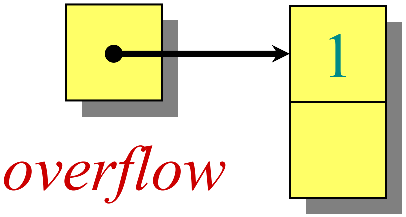
Example of a dynamic table

1. INSERT
2. INSERT



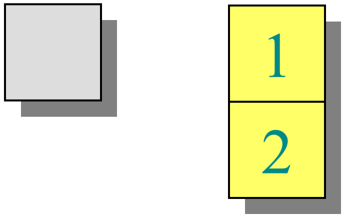
Example of a dynamic table

1. INSERT
2. INSERT



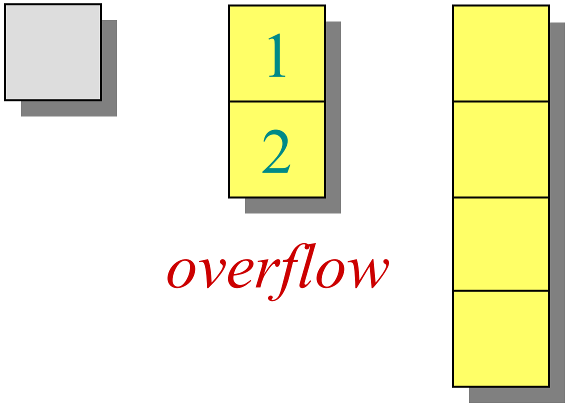
Example of a dynamic table

1. INSERT
2. INSERT



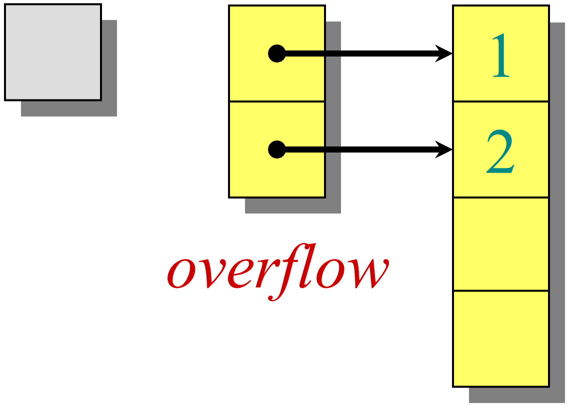
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



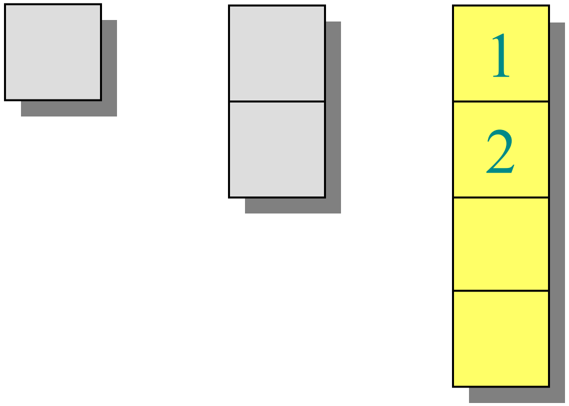
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



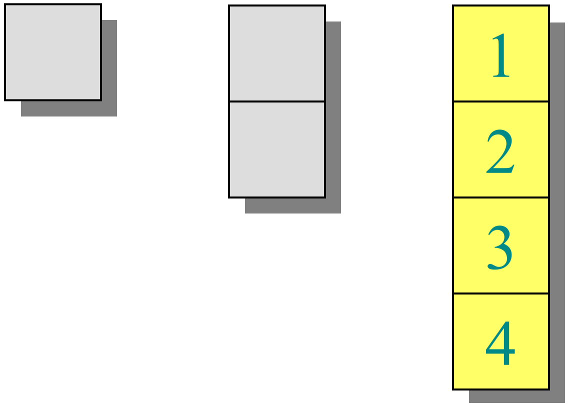
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



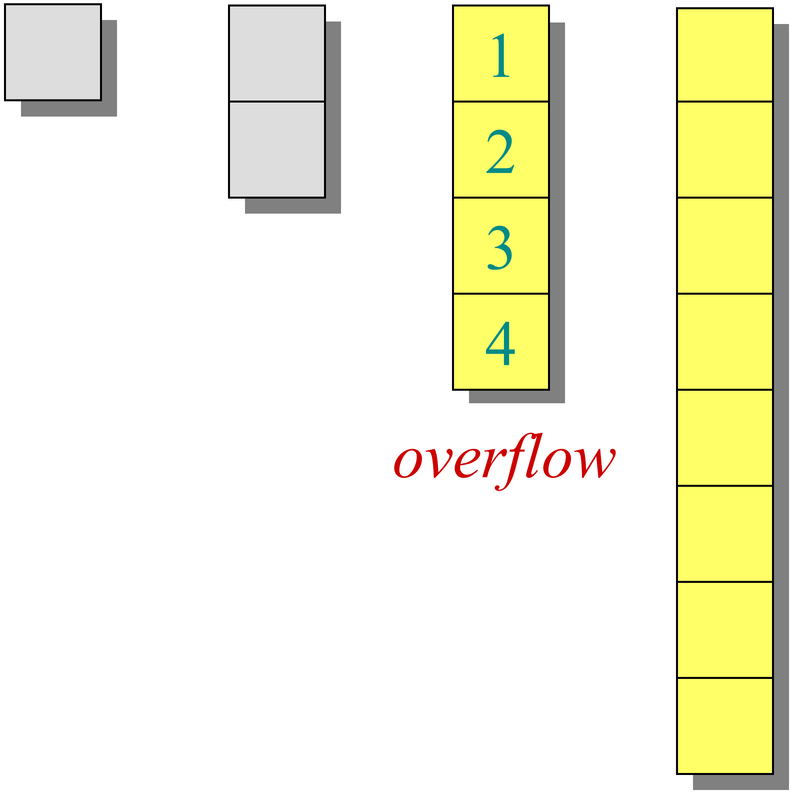
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



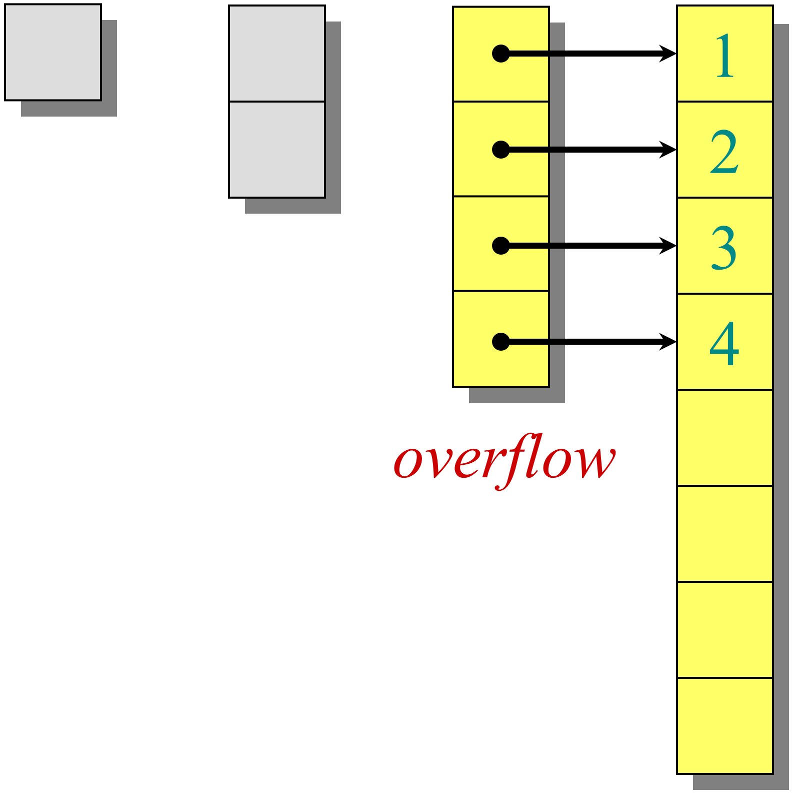
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



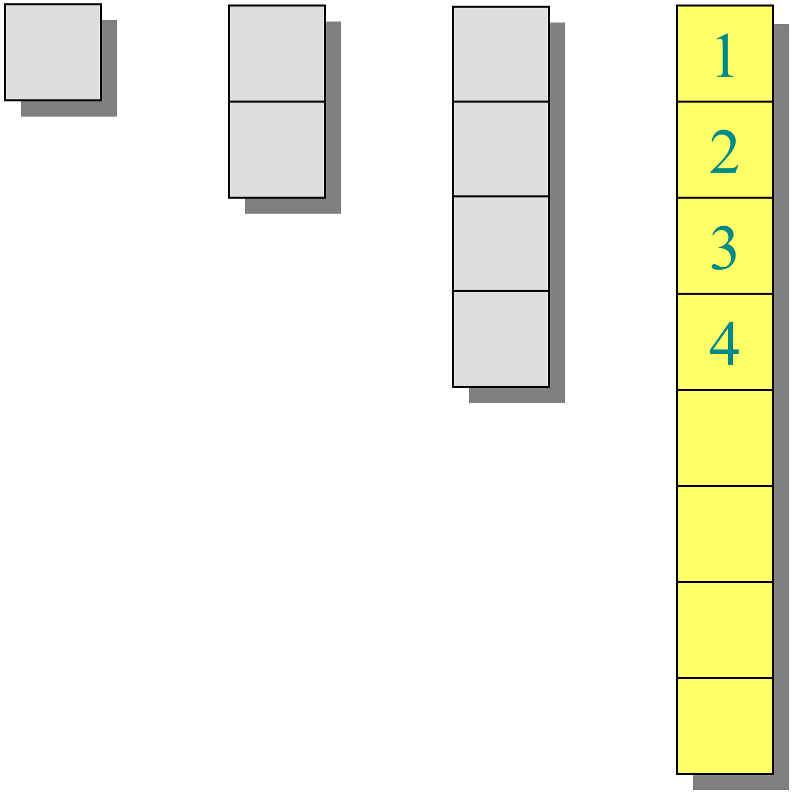
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



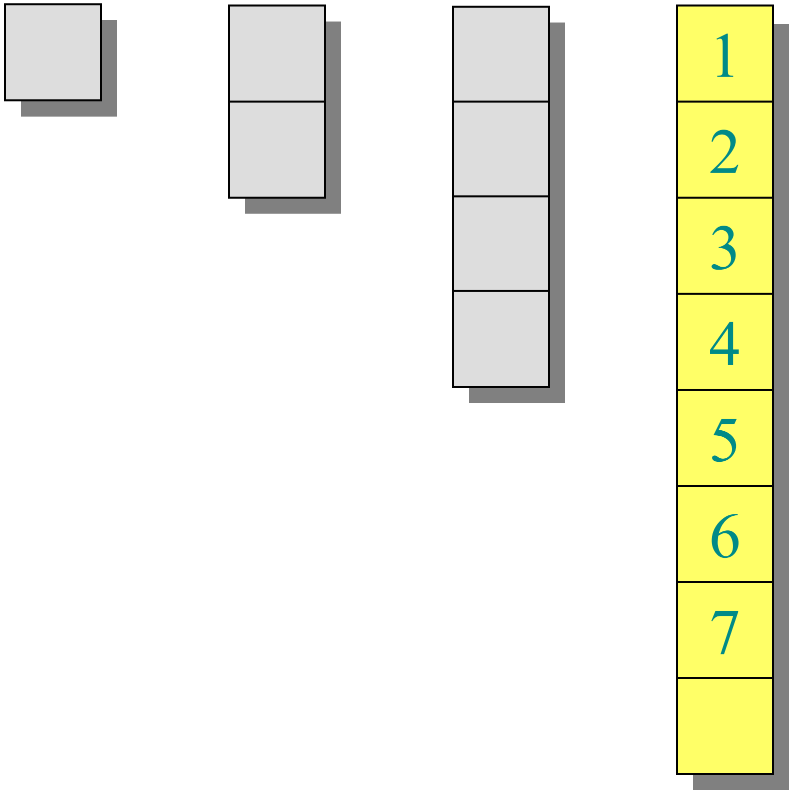
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG! In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n^2)$.

Let's see why.

Tighter analysis

Let $c_i =$ the cost of the i th insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Tighter analysis

Let $c_i =$ the cost of the i th insertion

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Tighter analysis (continued)

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\le 3n \\ &= \Theta(n)\,.\end{aligned}$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Amortized analysis

An ***amortized analysis*** is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we're taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the *worst case*.

Types of amortized analyses

Three common amortization arguments:

- the ***aggregate*** method,
- the ***accounting*** method,
- the ***potential*** method.

We've just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

Accounting method

- Charge i th operation a fictitious ***amortized cost*** $\hat{c}_i$, where \$1 pays for 1 unit of work (*i.e.*, time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The bank balance must not go negative! We must ensure that

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

  for all n.
- Thus, the total amortized costs provide an upper bound on the total true costs.

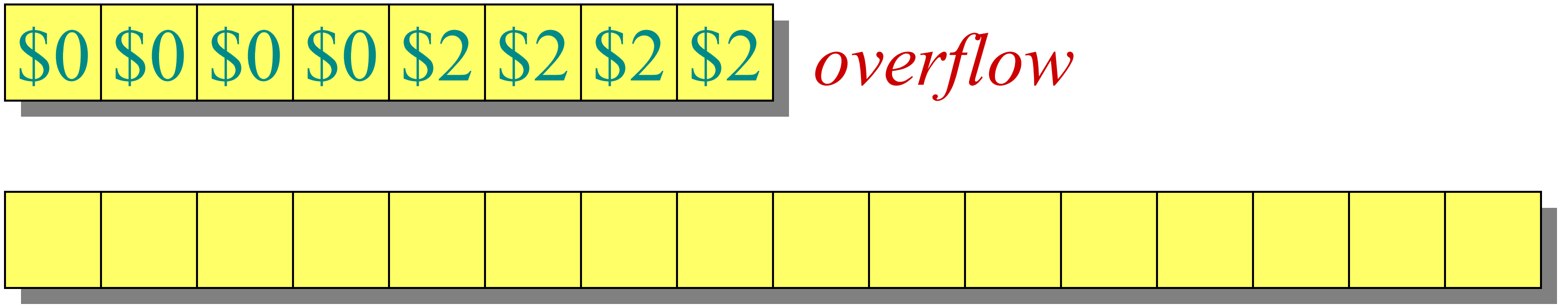
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



$0	$0	$0	$0	$2	$2	$2	$2

overflow

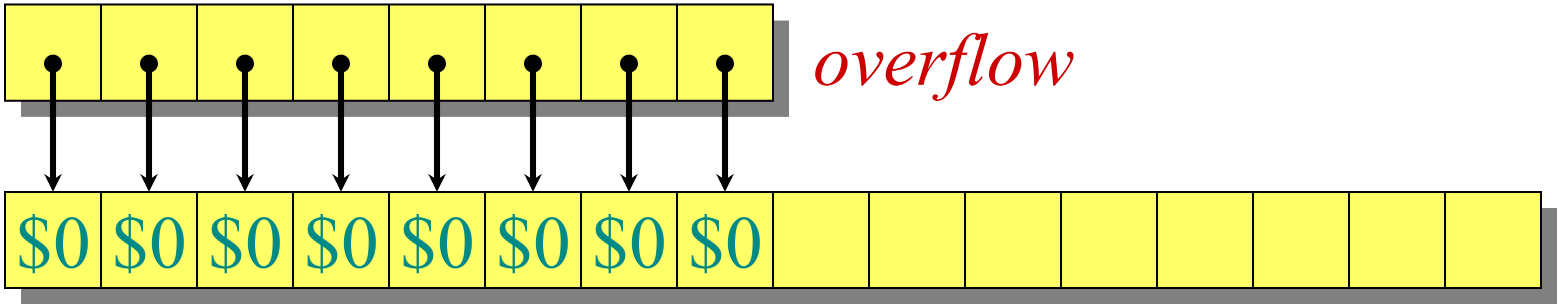
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



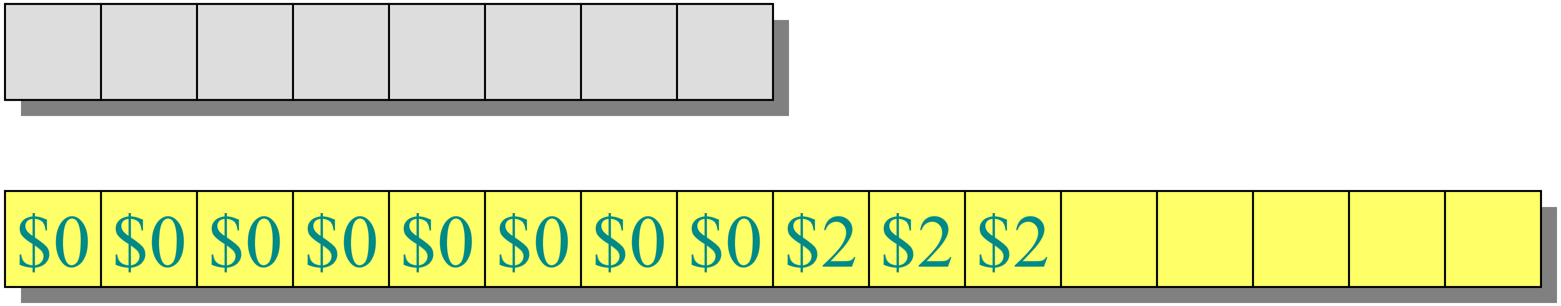
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



Accounting analysis (continued)

Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

*Okay, so I lied. The first operation costs only \$2, not \$3.

Potential method

IDEA: View the bank account as the potential energy (*à la* physics) of the dynamic set.

Framework:

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \to \mathbb{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

The amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

Summing both sides.

The amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

The series telescopes.

The amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

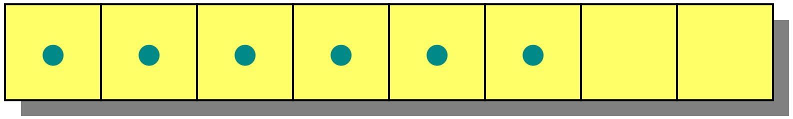
Potential analysis of table doubling

Define the potential of the table after the ith insertion by $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$. (Assume that $2^{\lceil \lg 0 \rceil} = 0$.)

Note:

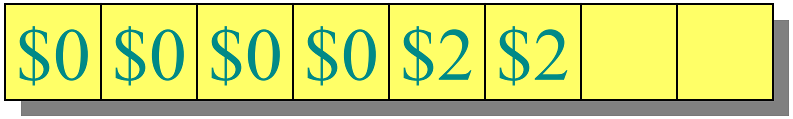
- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:



•	•	•	•	•	•		

$\Phi = 2 \cdot 6 - 2^3 = 4$



(

$0	$0	$0	$0	$2	$2		

accounting method)

Calculation of amortized costs

The amortized cost of the i th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \begin{cases} i + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) & \text{if } i - 1 \text{ is an exact power of } 2, \\ 1 + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) & \text{otherwise.} \end{cases}$$

Calculation (Case 1)

Case 1: $i - 1$ is an exact power of 2.

$$
\begin{aligned}
\hat{c}_i &= i + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) \\
&= i + 2 - \left(2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}\right) \\
&= i + 2 - \left(2(i-1) - (i-1)\right) \\
&= i + 2 - 2i + 2 + i - 1 \\
&= 3
\end{aligned}
$$

Calculation (Case 2)

Case 2: $i - 1$ is not an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= 1 + 2 - \left(2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Exercise: Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.

Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 15

Prof. Charles E. Leiserson

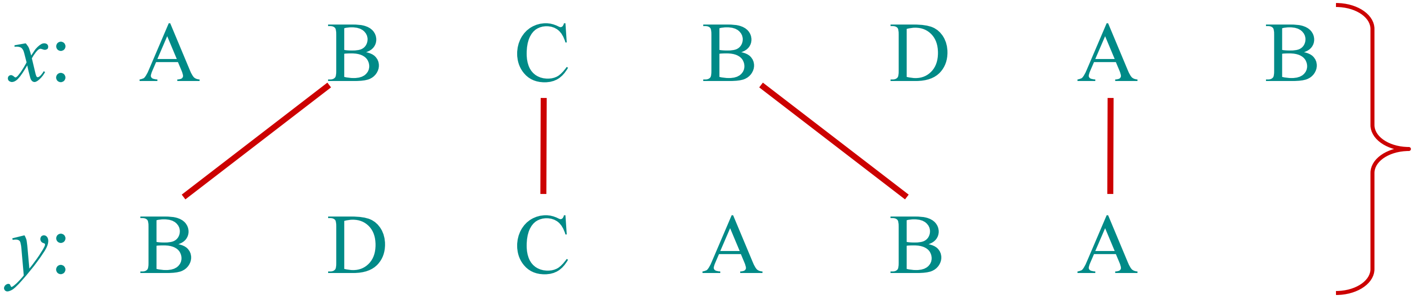
Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"



BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1\,.\,.\,m]$ to see if it is also a subsequence of $y[1\,.\,.\,n]$.

Analysis

- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$=$ exponential time.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

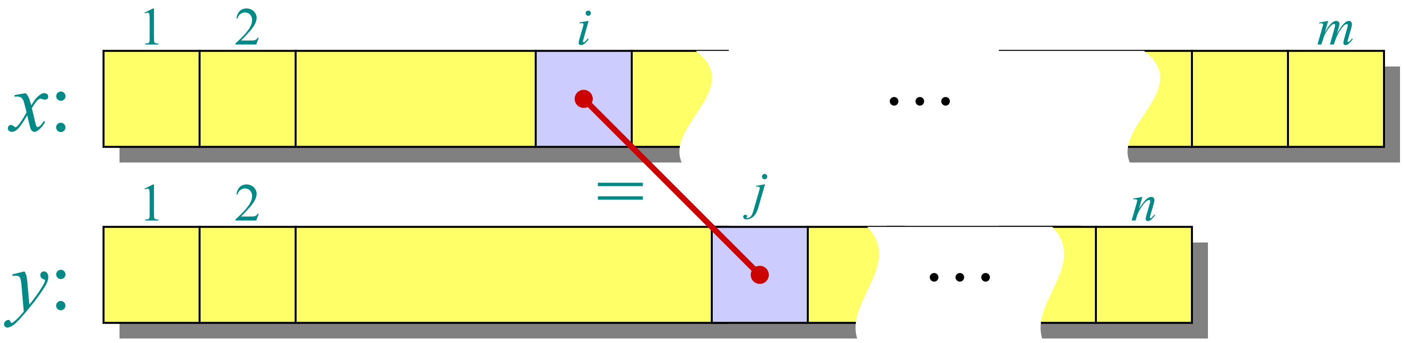
- Define $c[i, j] = |\text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:



Let $z[1\,.\,.\,k] = \text{LCS}(x[1\,.\,.\,i], y[1\,.\,.\,j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1\,.\,.\,k-1]$ is CS of $x[1\,.\,.\,i-1]$ and $y[1\,.\,.\,j-1]$.

Proof (continued)

Claim: $z[1 \,.\,.\, k-1] = \mathrm{LCS}(x[1 \,.\,.\, i-1], y[1 \,.\,.\, j-1])$.

Suppose w is a longer CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \,.\,.\, i]$ and $y[1 \,.\,.\, j]$ with $\left| w \,||\, z[k] \right| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. □

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

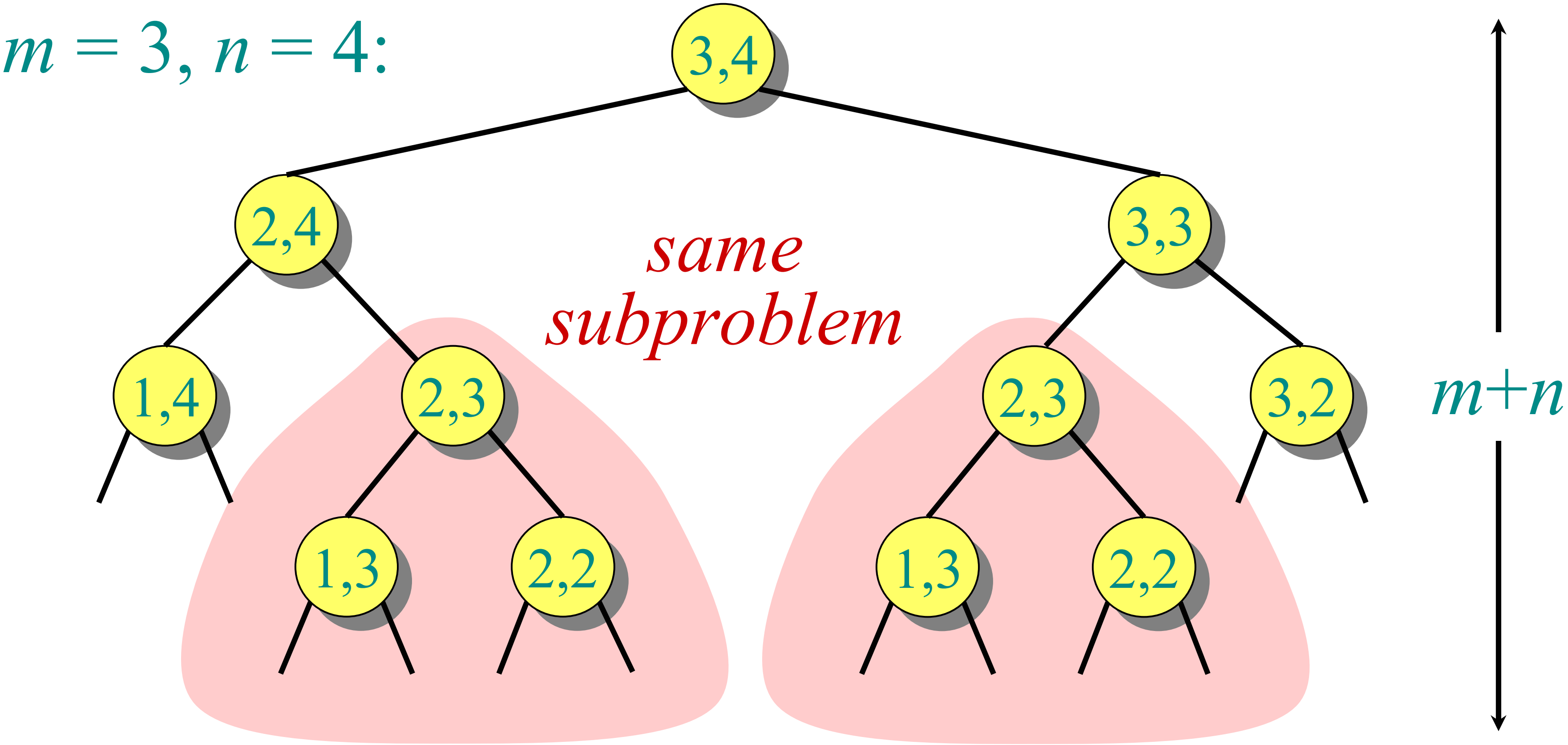
If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

```
LCS$(x, y, i, j)$
    **if** $x[i] = y[j]$
        **then** $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
        **else** $c[i, j] \leftarrow \max\{$LCS$(x, y, i-1, j),$
                                  LCS$(x, y, i, j-1)\}$
```

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree



Height = $m + n$ ⇒ work potentially exponential, but we're solving subproblems already solved!

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS($x, y, i, j$)
    if $c[i, j]$ = NIL
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow$ LCS($x, y, i–1, j–1$) + 1
            else $c[i, j] \leftarrow \max\{$LCS($x, y, i–1, j$),
                                LCS($x, y, i, j–1$)$\}$
```

same as before

Time = $\Theta(mn)$ = constant work per table entry.

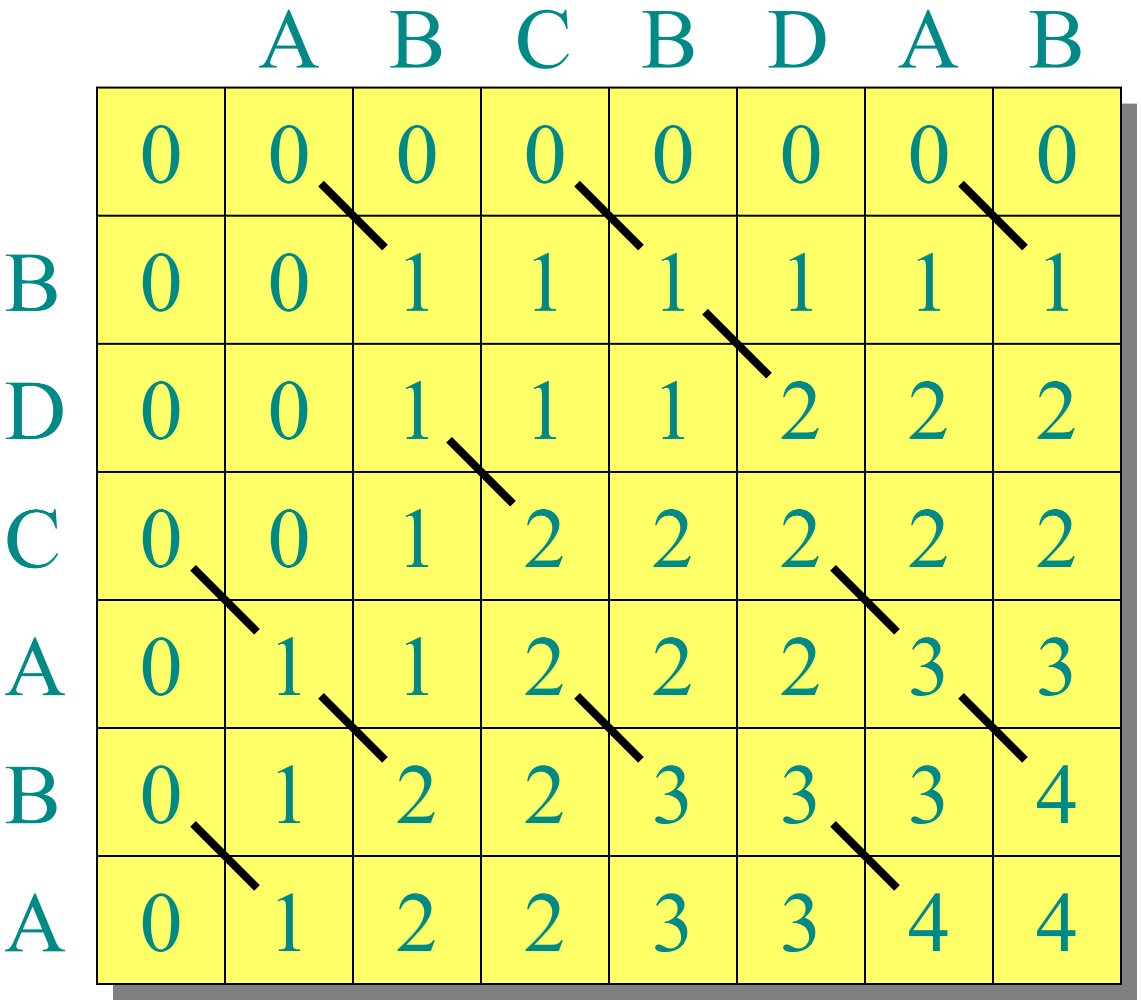
Space = $\Theta(mn)$.

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

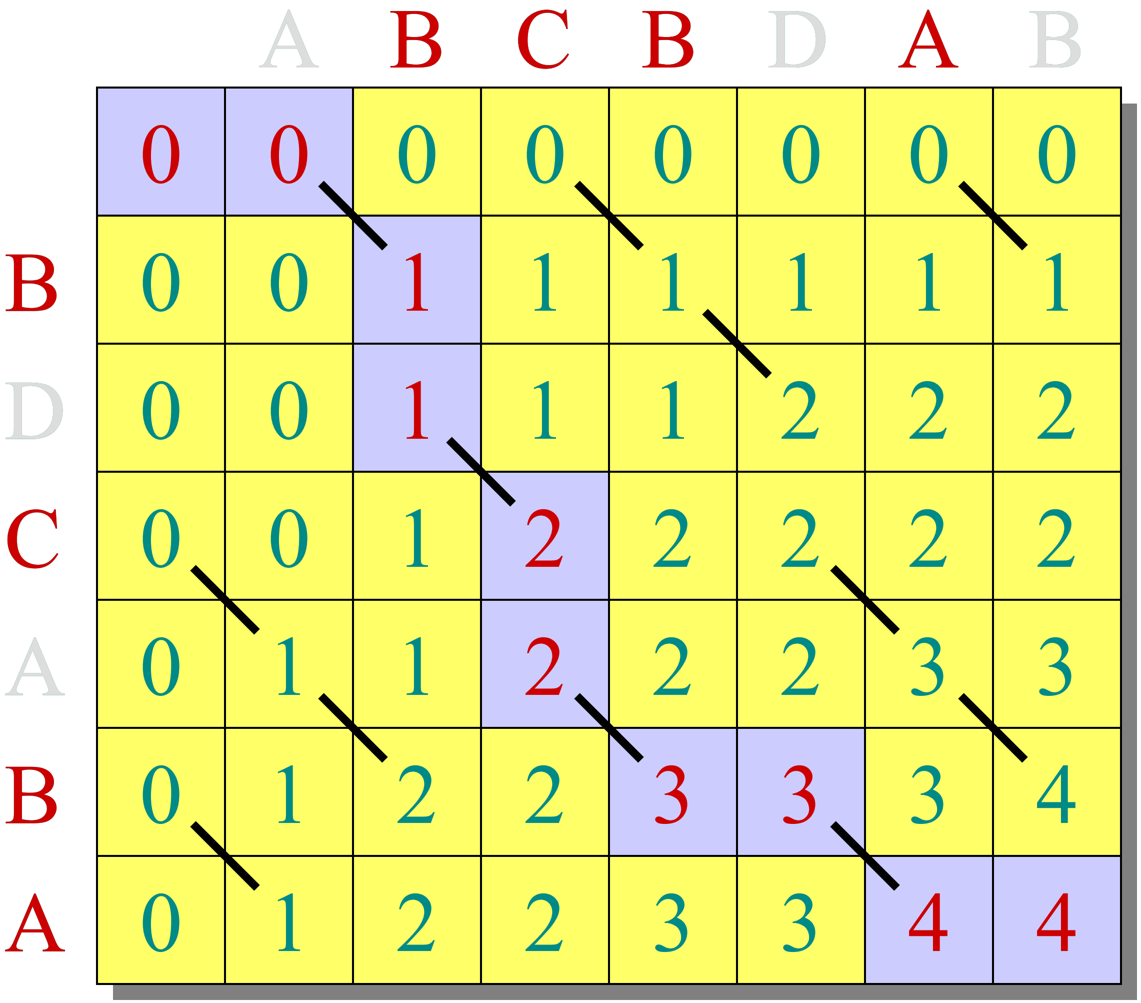
Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

Exercise: $O(\min\{m, n\})$.



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 16

Prof. Charles E. Leiserson

Graphs (review)

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

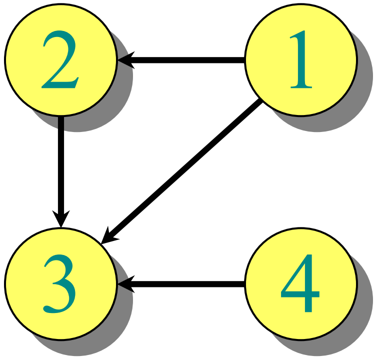
In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg|E| = \Theta(\lg V)$.

(Review CLRS, Appendix B.)

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,..\, n, 1 \,..\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in \mathrm{E}, \\ 0 & \text{if } (i, j) \notin \mathrm{E}. \end{cases}$$

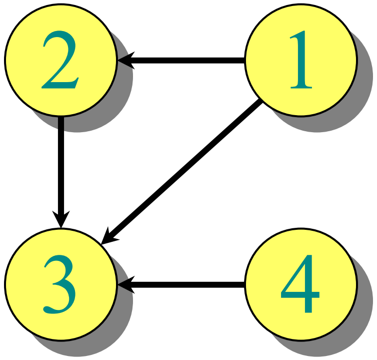


$$\begin{array}{c|cccc} A & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} = 2|E|$ for undirected graphs ⇒ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation (for either type of graph).

Minimum spanning trees

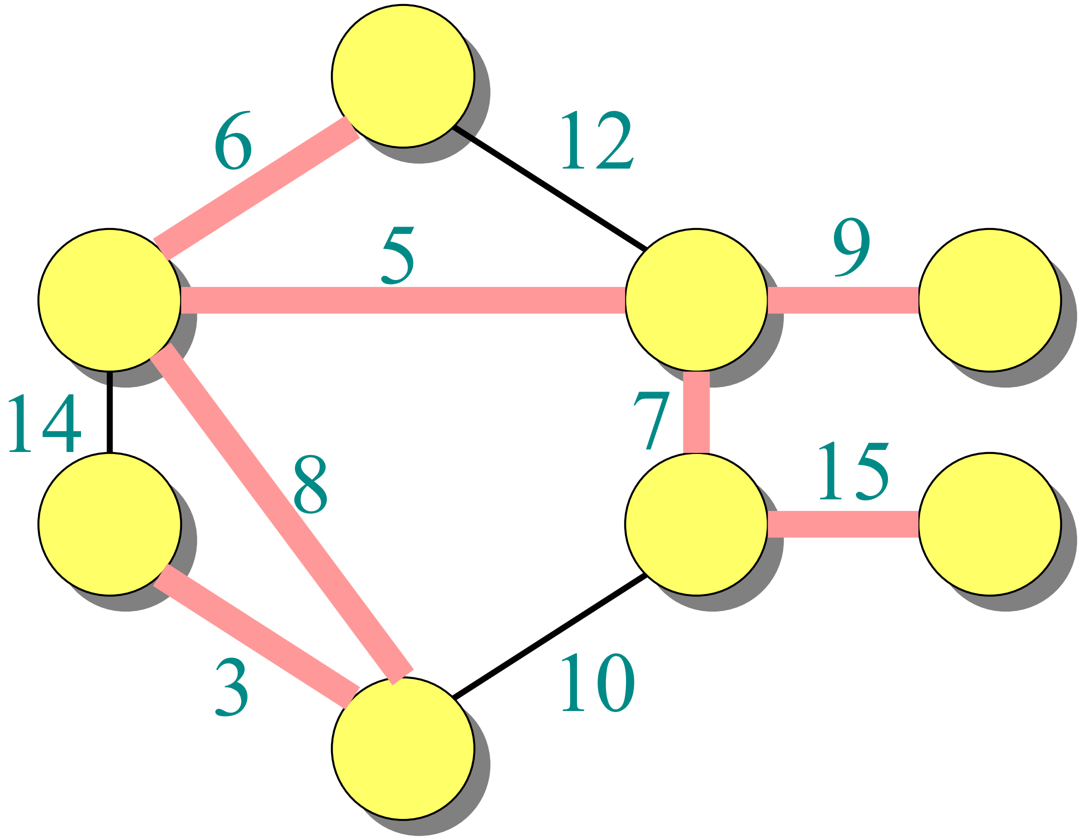
Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v)\in T} w(u,v) .$$

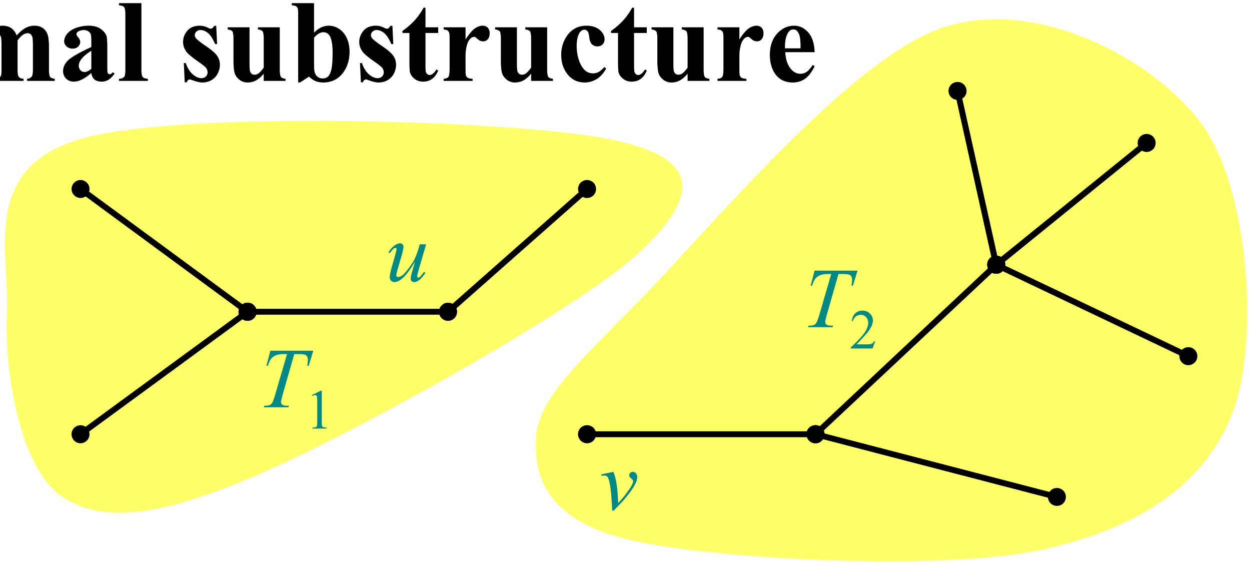
Example of MST



Optimal substructure

MST T:

(Other edges of G are not shown.)



Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.

Theorem. The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the subgraph of G ***induced*** by the vertices of T_1:

$$V_1 = \text{vertices of } T_1,$$
$$E_1 = \{\, (x, y) \in E : x, y \in V_1 \,\}.$$

Similarly for T_2.

Proof of optimal substructure

Proof. Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If T_1' were a lower-weight spanning tree than T_1 for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G. □

Do we also have overlapping subproblems?

- Yes.

Great, then dynamic programming may work!

- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.

Hallmark for “greedy” algorithms

Greedy-choice property

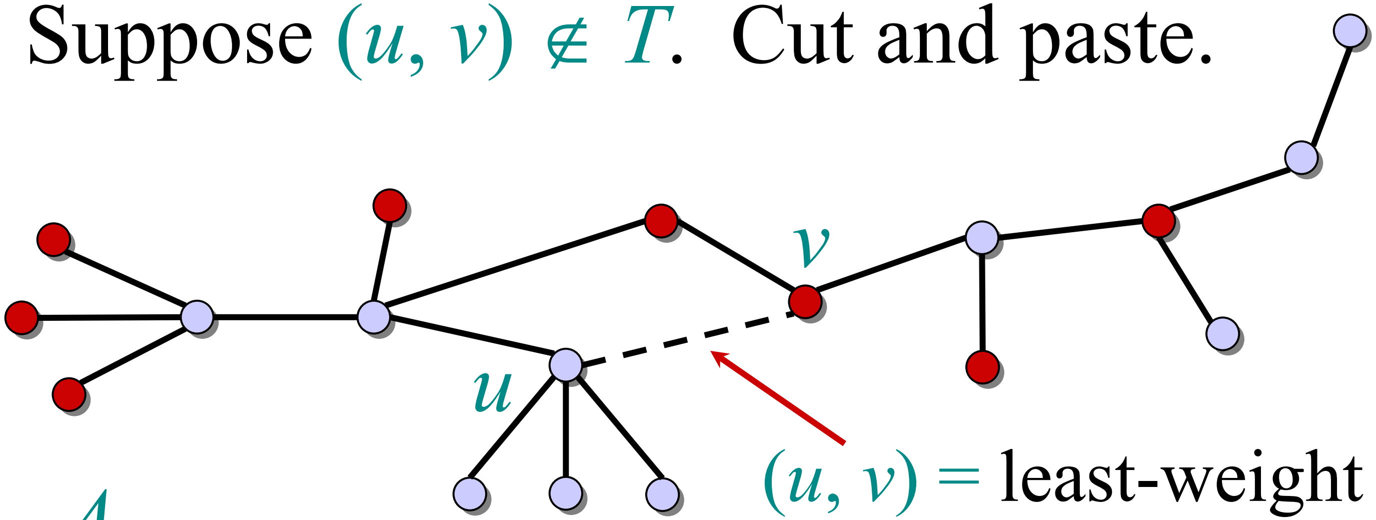
A locally optimal choice is globally optimal.

Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



○ $\in A$

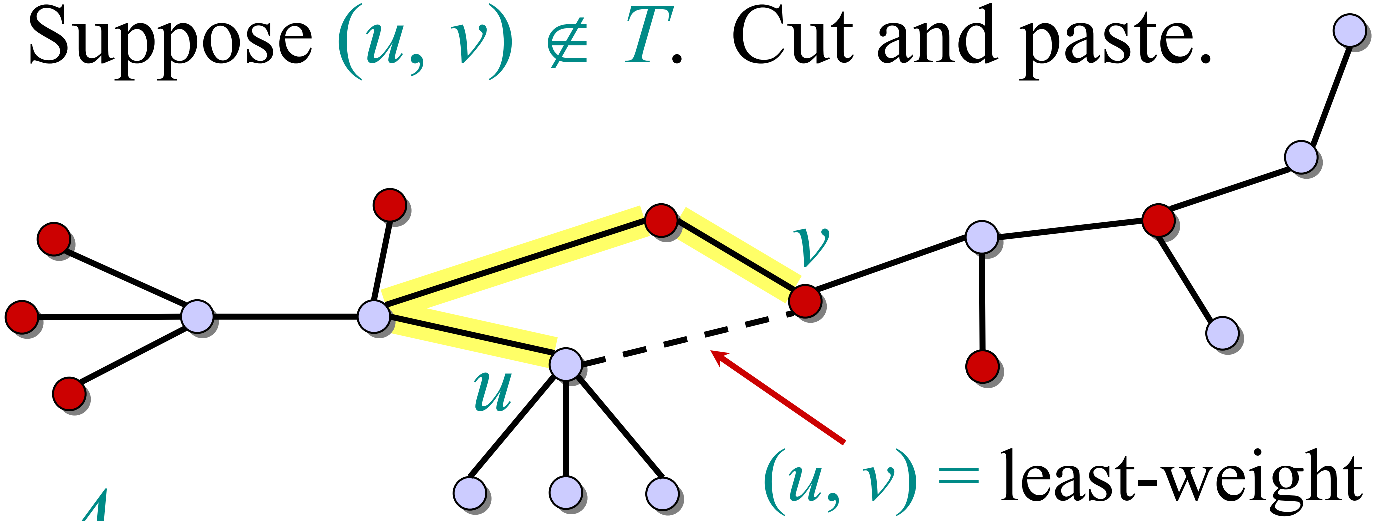
● $\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T:



○ $\in A$

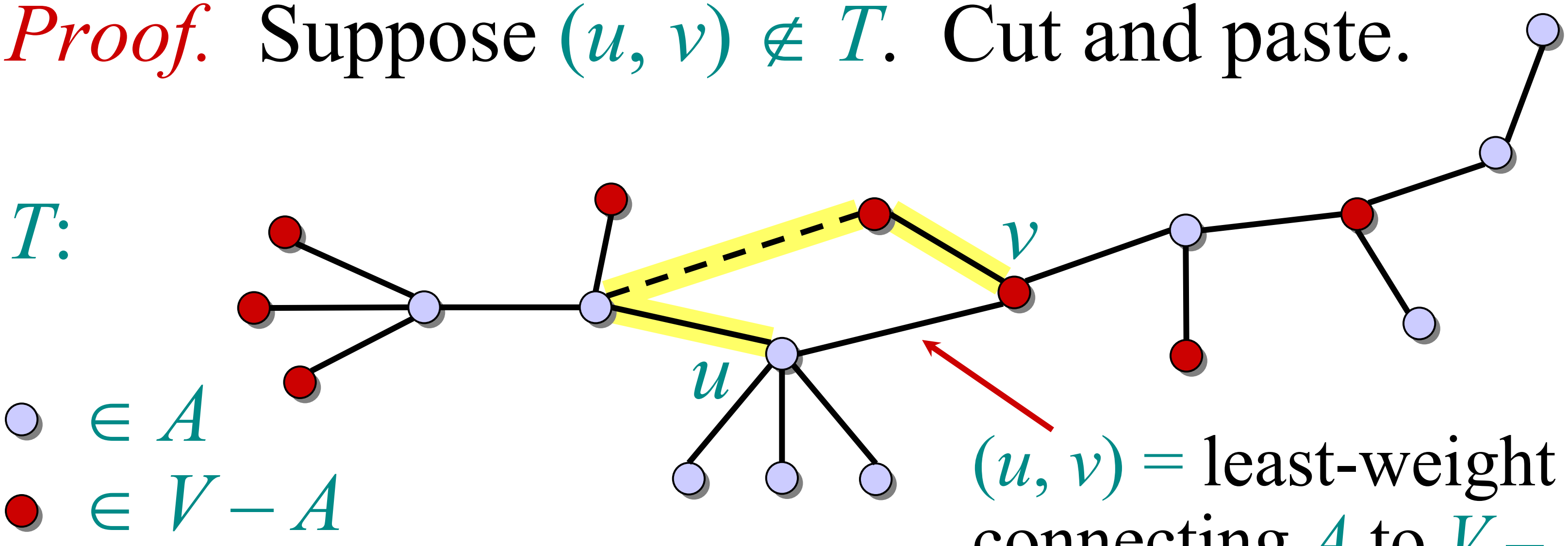
● $\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

Consider the unique simple path from u to v in T.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.



(u, v) = least-weight edge connecting A to $V - A$

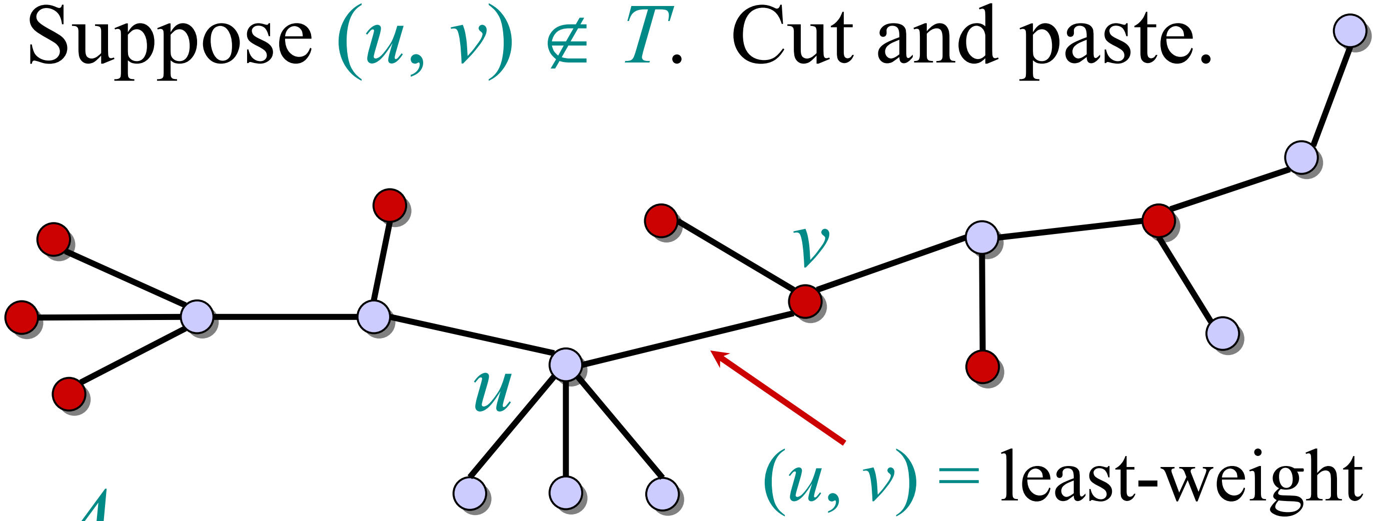
Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.

T':



○ $\in A$

● $\in V - A$

(u, v) = least-weight edge connecting A to $V - A$

Consider the unique simple path from u to v in T.

Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.

A lighter-weight spanning tree than T results. □

Prim's algorithm

IDEA: Maintain $V - A$ as a priority queue Q. Key each vertex in Q with the weight of the least-weight edge connecting it to a vertex in A.

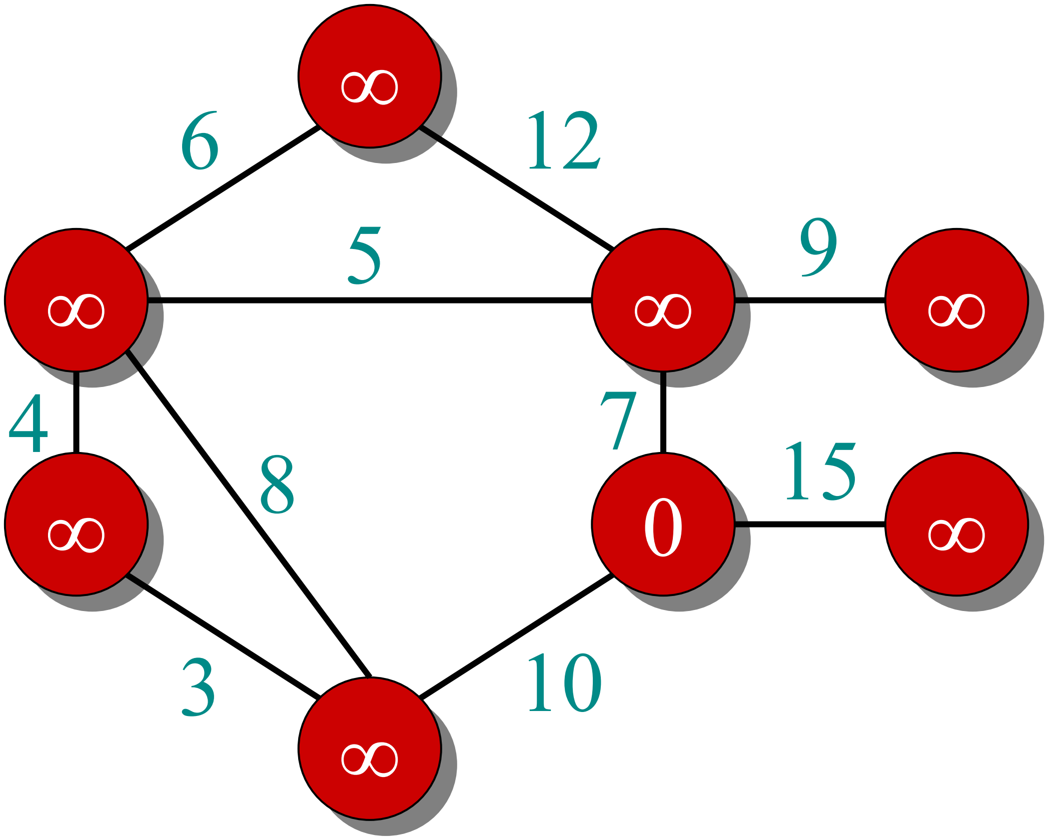
```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
        for each $v \in Adj[u]$
            do if $v \in Q$ and $w(u, v) < key[v]$
                then $key[v] \leftarrow w(u, v)$      ▷ DECREASE-KEY
                     $\pi[v] \leftarrow u$
```

At the end, $\{(v, \pi[v])\}$ forms the MST.

Example of Prim’s algorithm

○ $\in A$

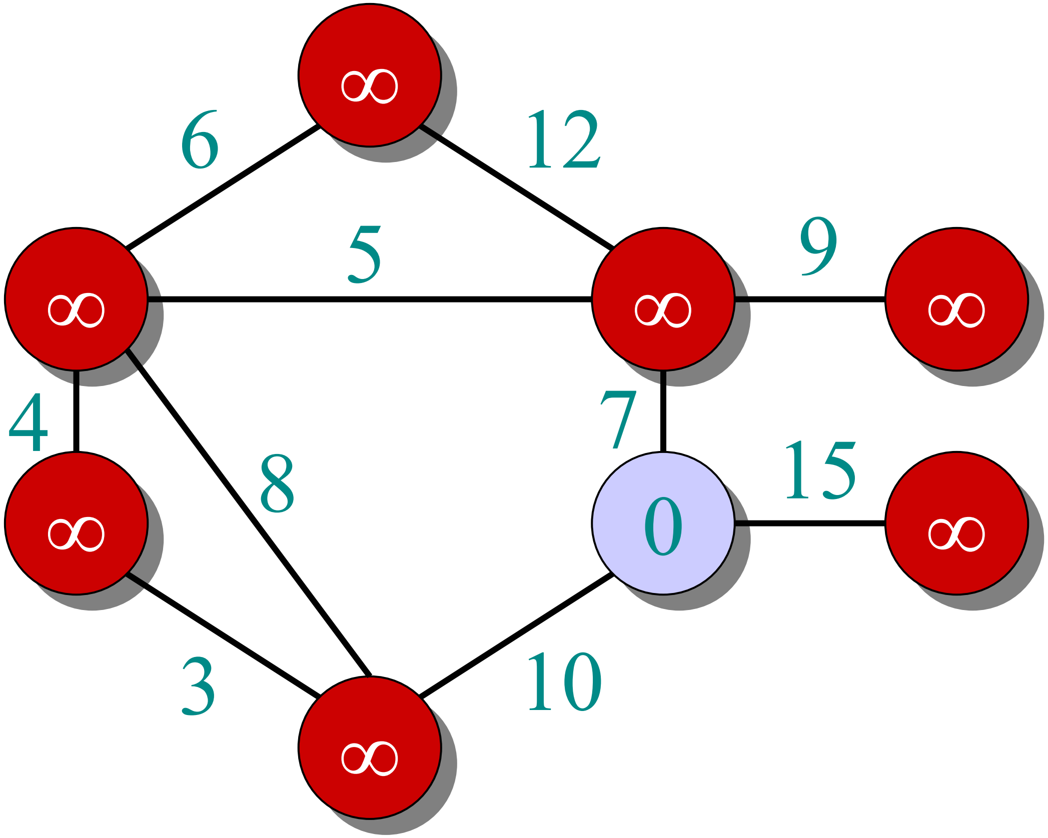
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

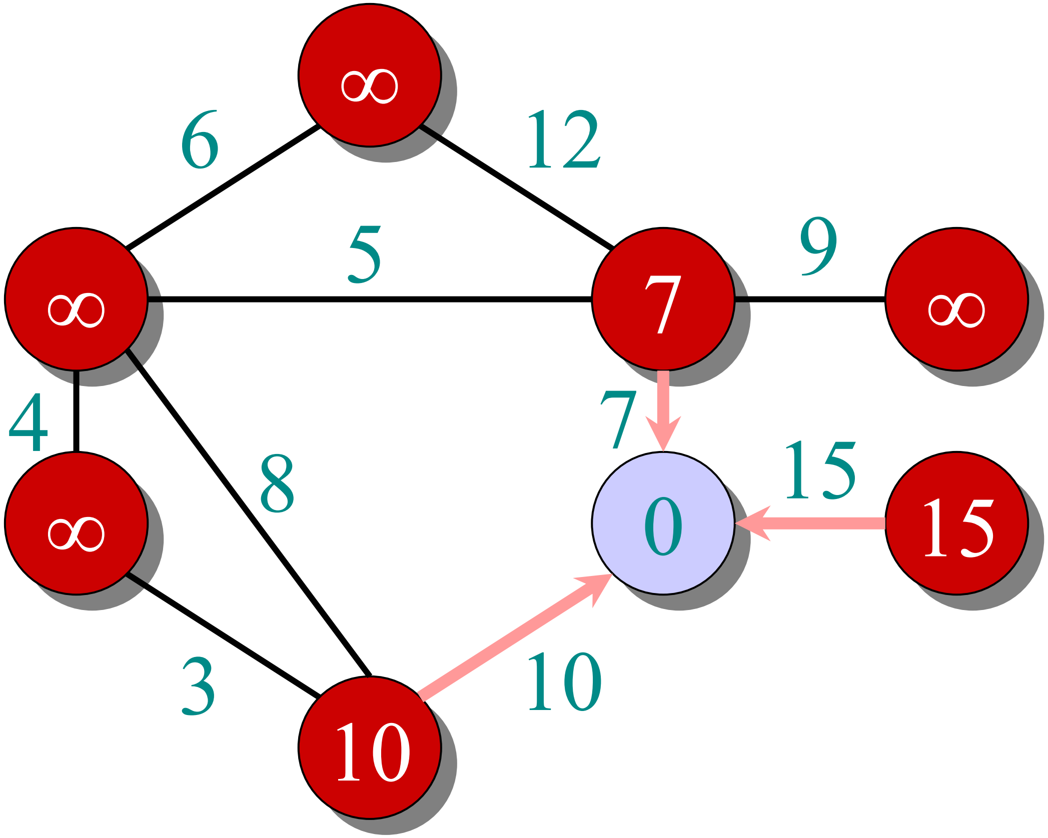
● $\in V - A$



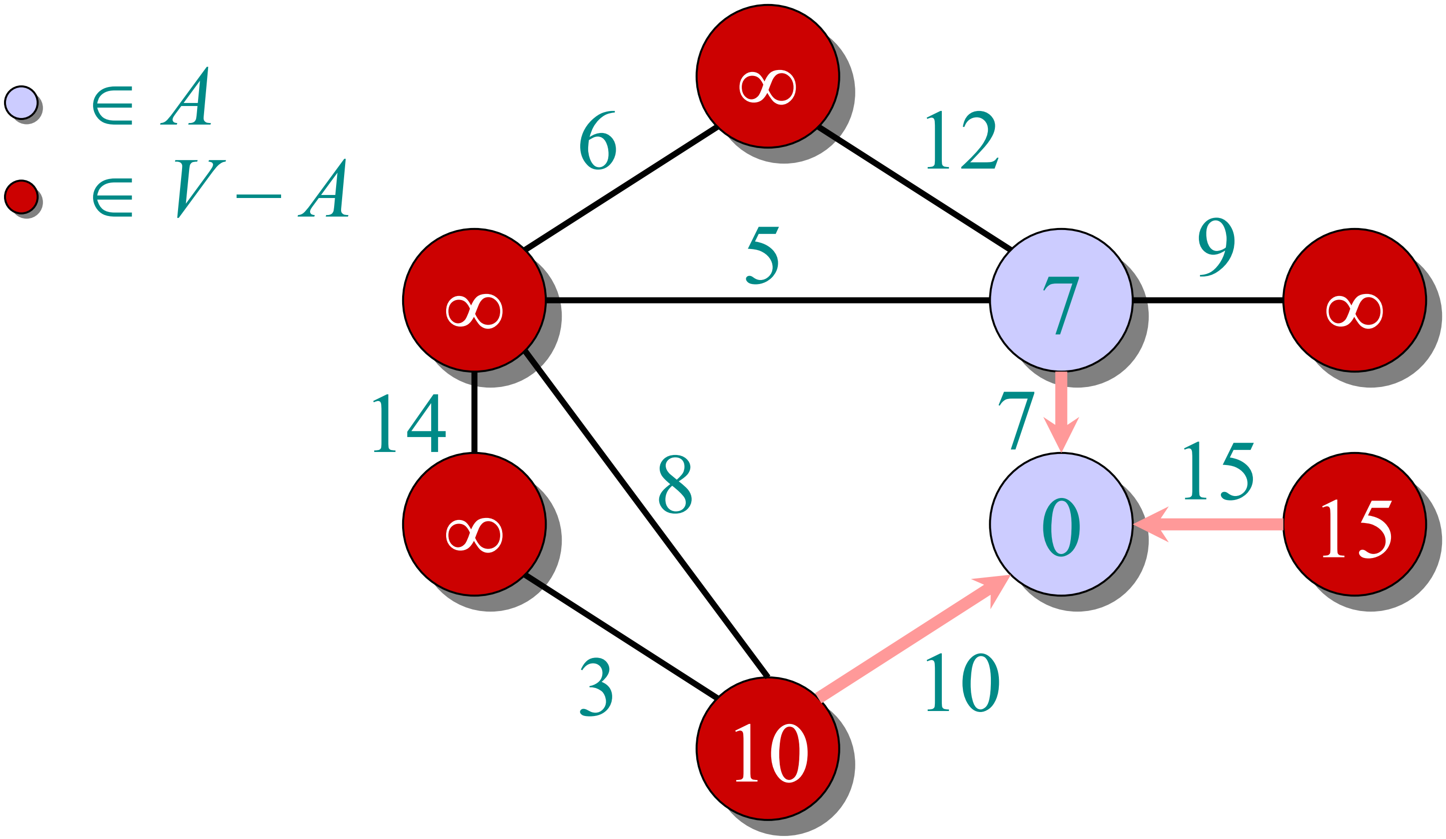
Example of Prim's algorithm

○ $\in A$

● $\in V - A$



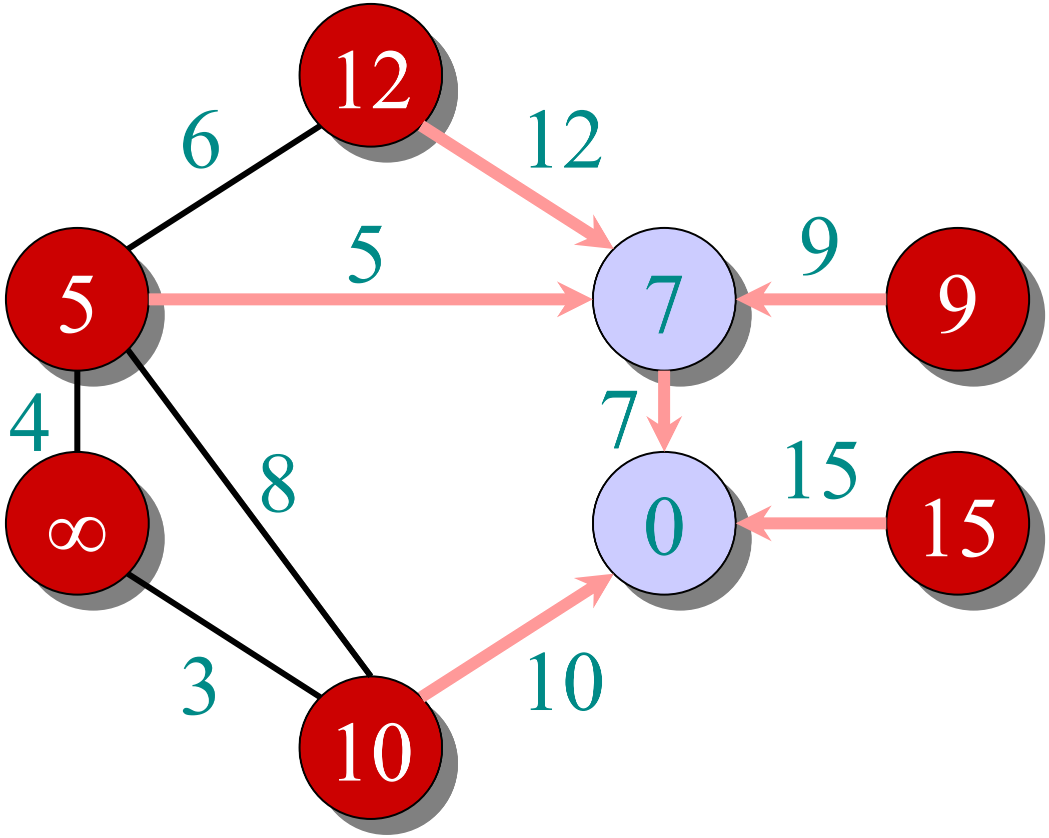
Example of Prim’s algorithm



Example of Prim's algorithm

○ $\in A$

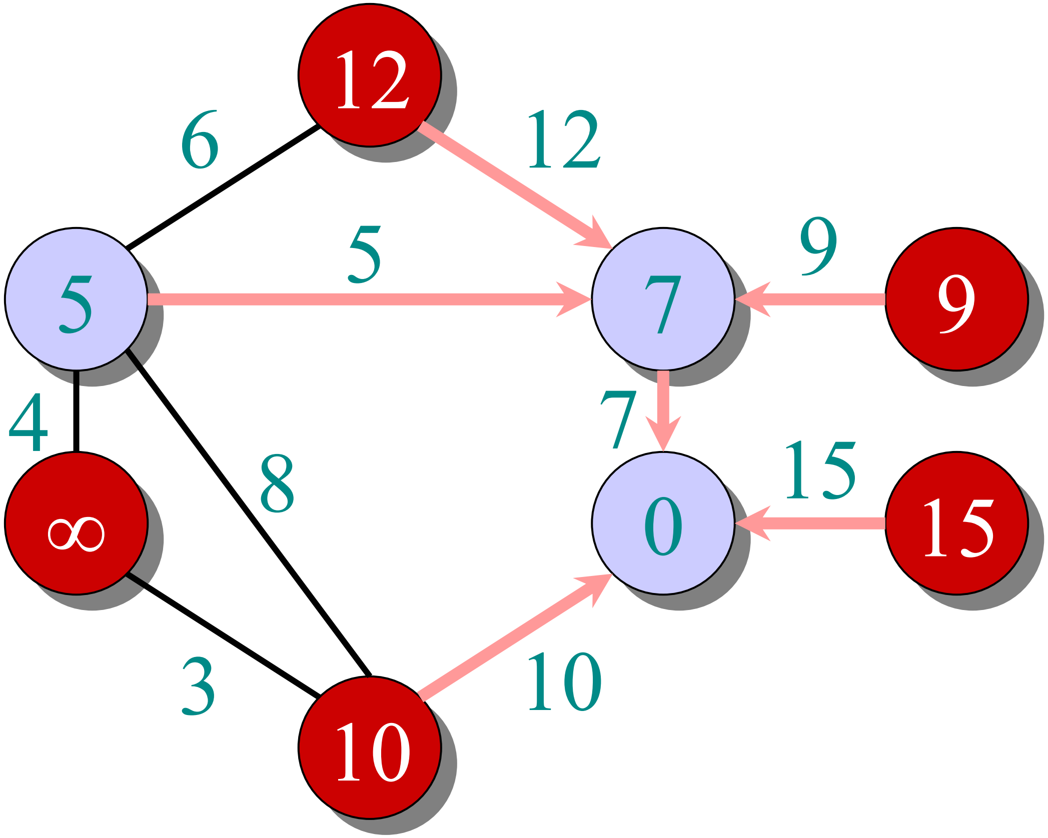
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

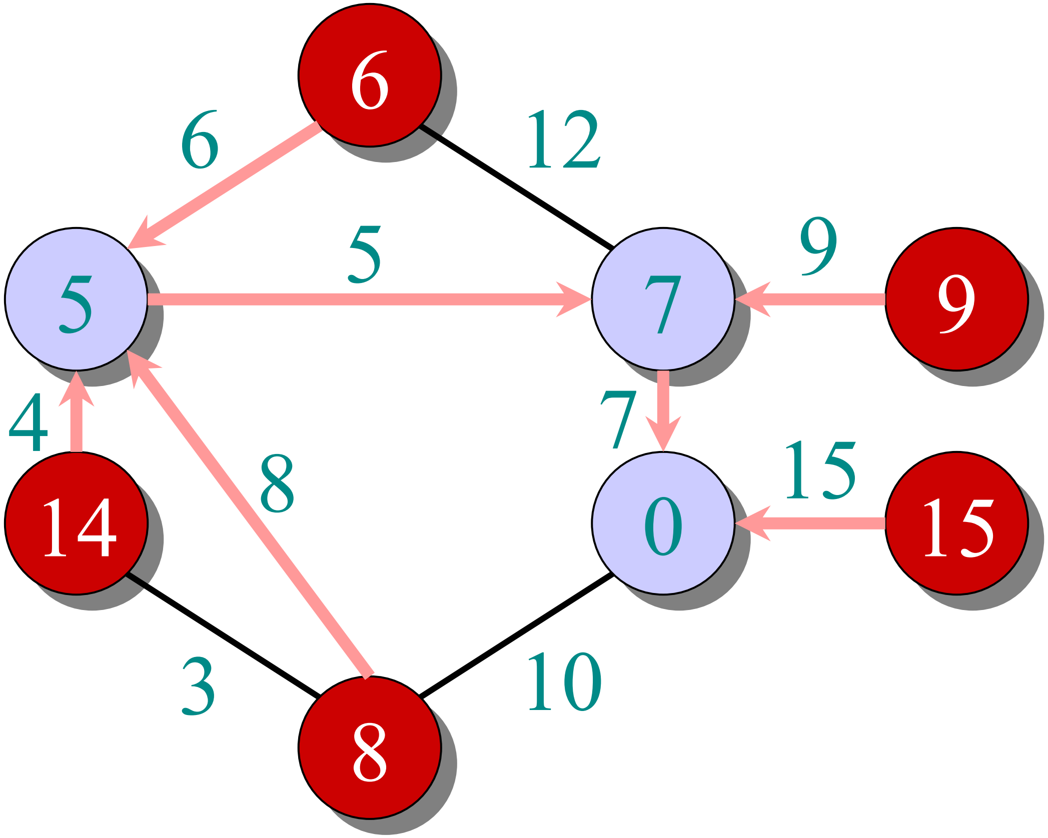
● $\in V - A$



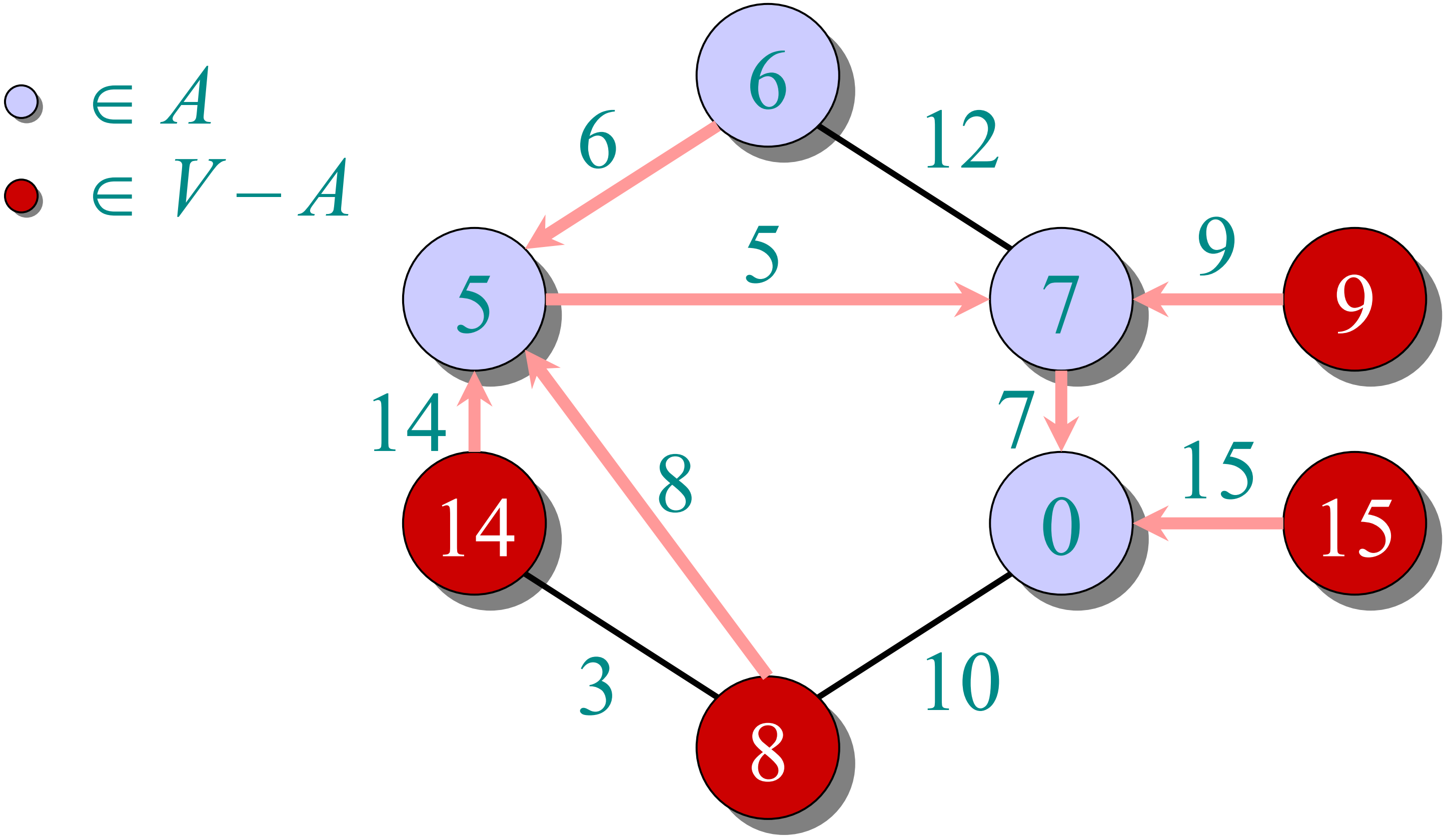
Example of Prim's algorithm

○ $\in A$

● $\in V - A$



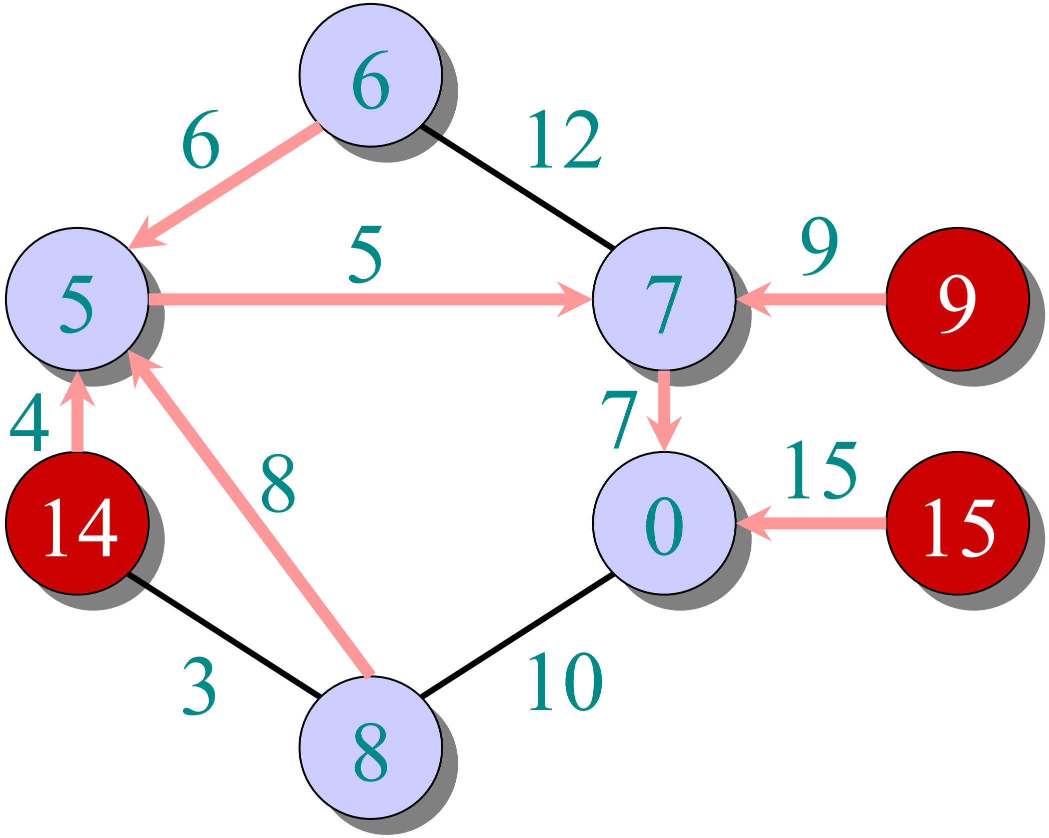
Example of Prim's algorithm



Example of Prim's algorithm

○ $\in A$

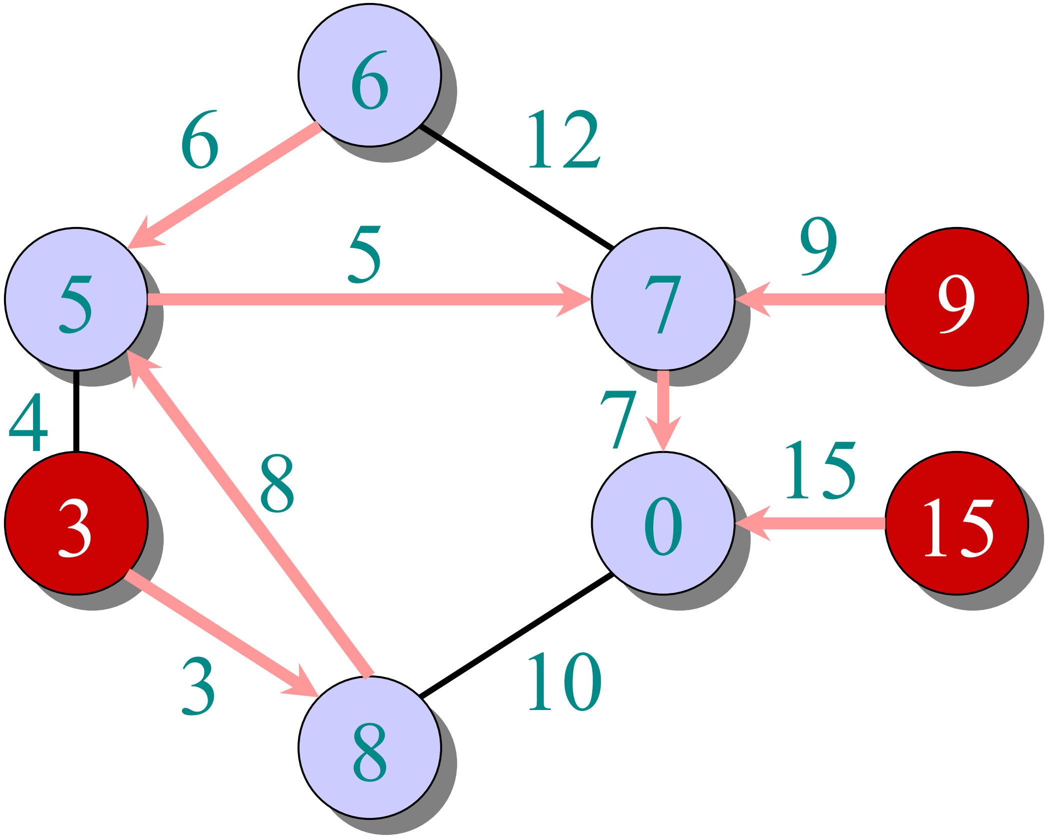
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

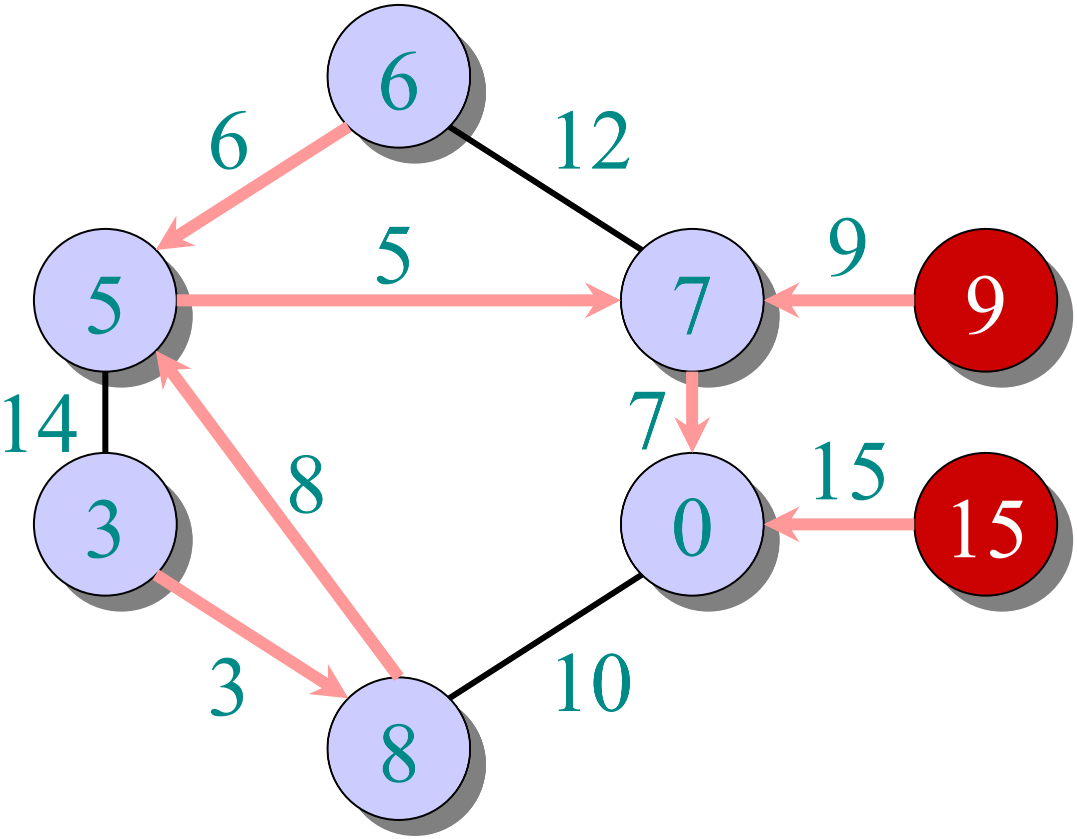
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

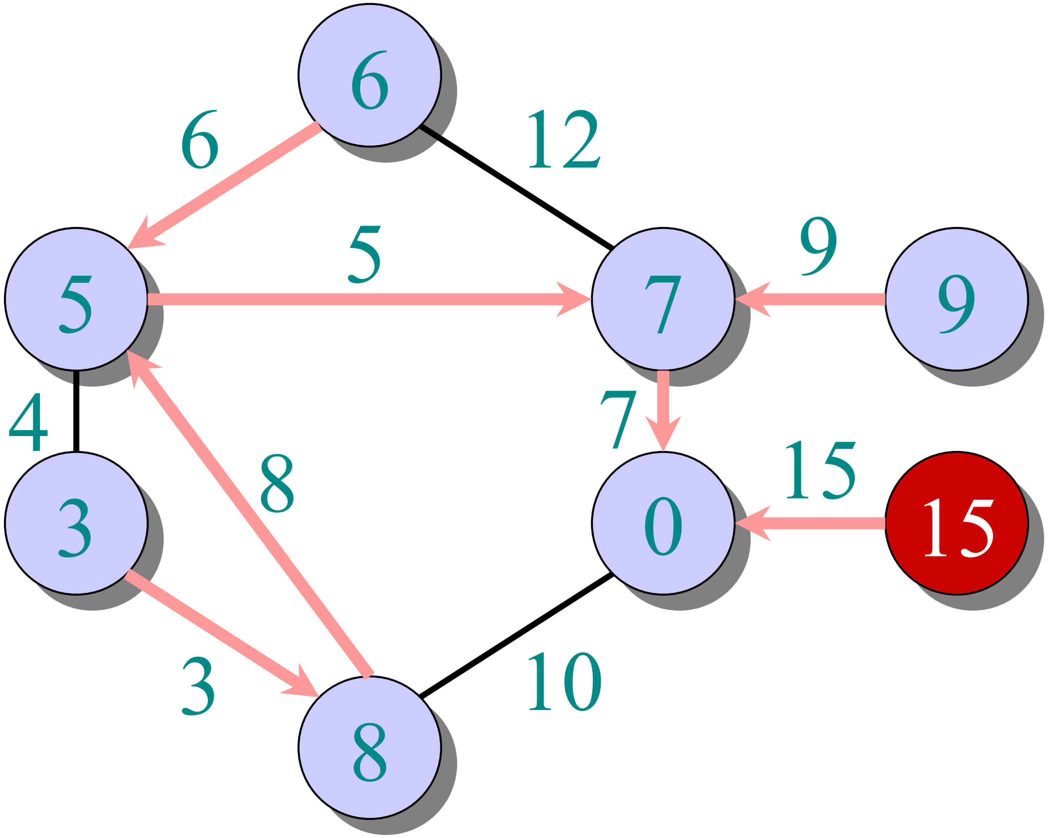
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

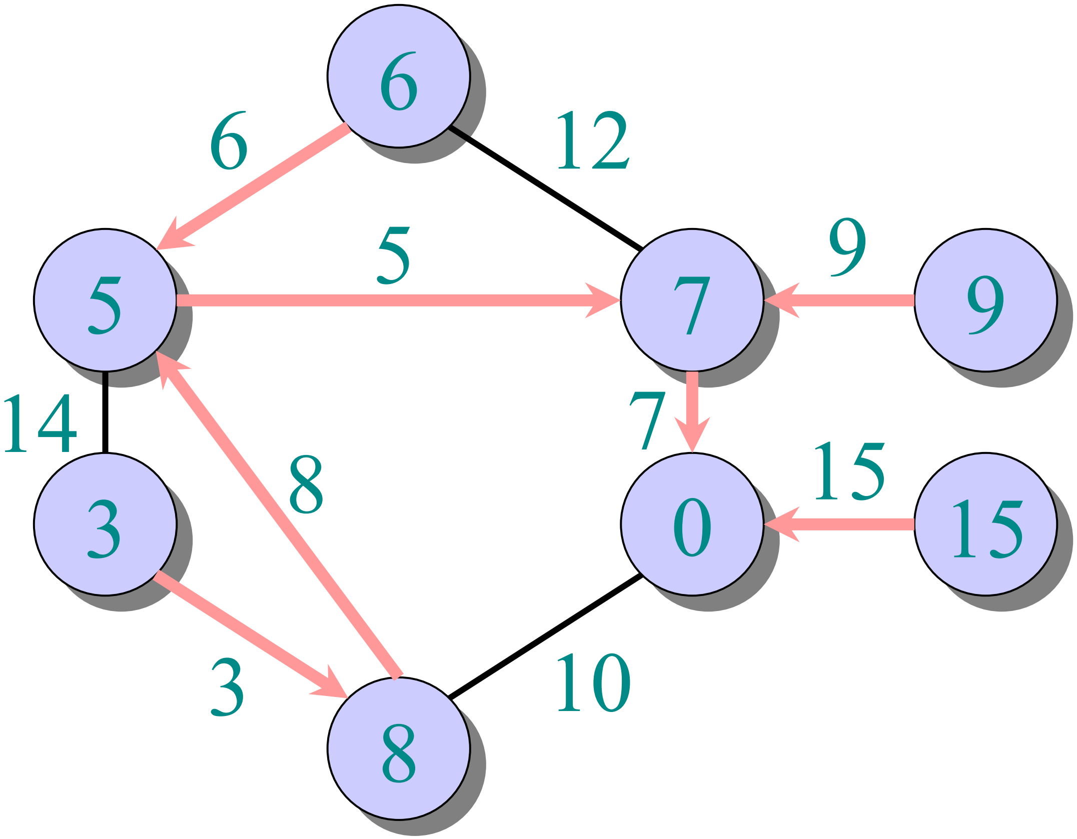
● $\in V - A$



Example of Prim's algorithm

○ $\in A$

● $\in V - A$



Analysis of Prim

$\Theta(V)$ total

$|V|$ times

degree(u) times

```
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                 then $key[v] \leftarrow w(u, v)$
                      $\pi[v] \leftarrow u$
```

Handshaking Lemma ⇒ $\Theta(E)$ implicit DECREASE-KEY's.

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Analysis of Prim (continued)

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

MST algorithms

Kruskal's algorithm (see CLRS):

- Uses the ***disjoint-set data structure*** (Lecture 20).
- Running time $= O(E \lg V)$.

Best to date:

- Karger, Klein, and Tarjan [1993].
- Randomized algorithm.
- $O(V + E)$ expected time.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 17

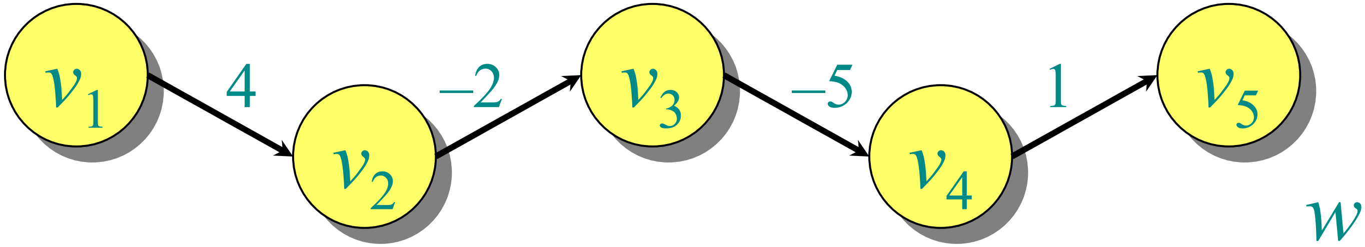
Prof. Erik Demaine

Paths in graphs

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

Example:



$w(p) = –2$

Shortest paths

A ***shortest path*** from u to v is a path of minimum weight from u to v. The ***shortest-path weight*** from u to v is defined as

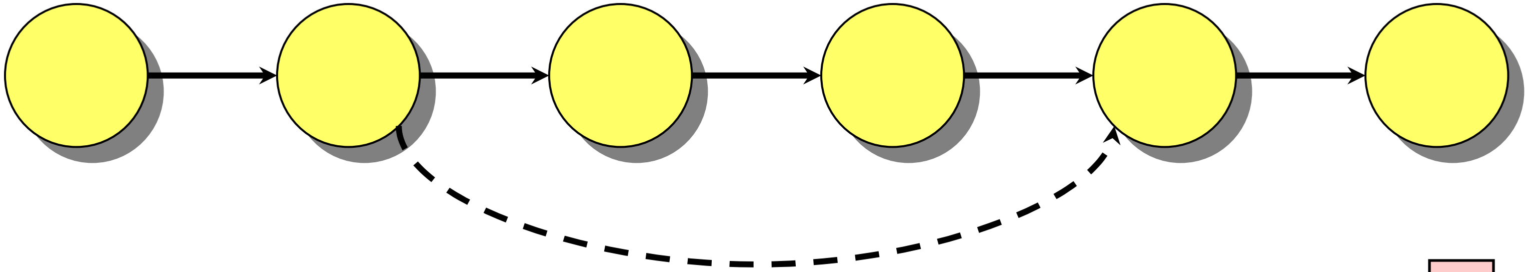
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

Note: $\delta(u, v) = \infty$ if no path from u to v exists.

Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

Proof. Cut and paste:



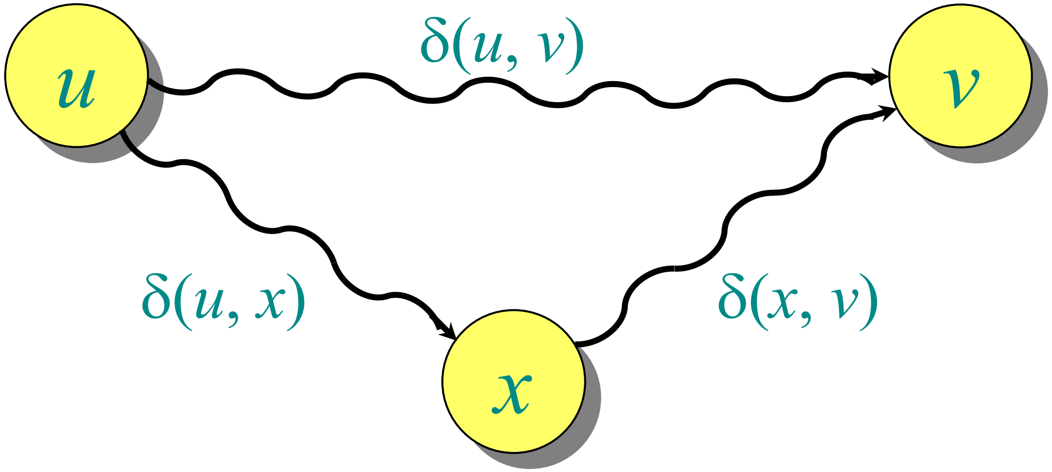
□

Triangle inequality

Theorem. For all $u, v, x \in V$, we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v).$$

Proof.

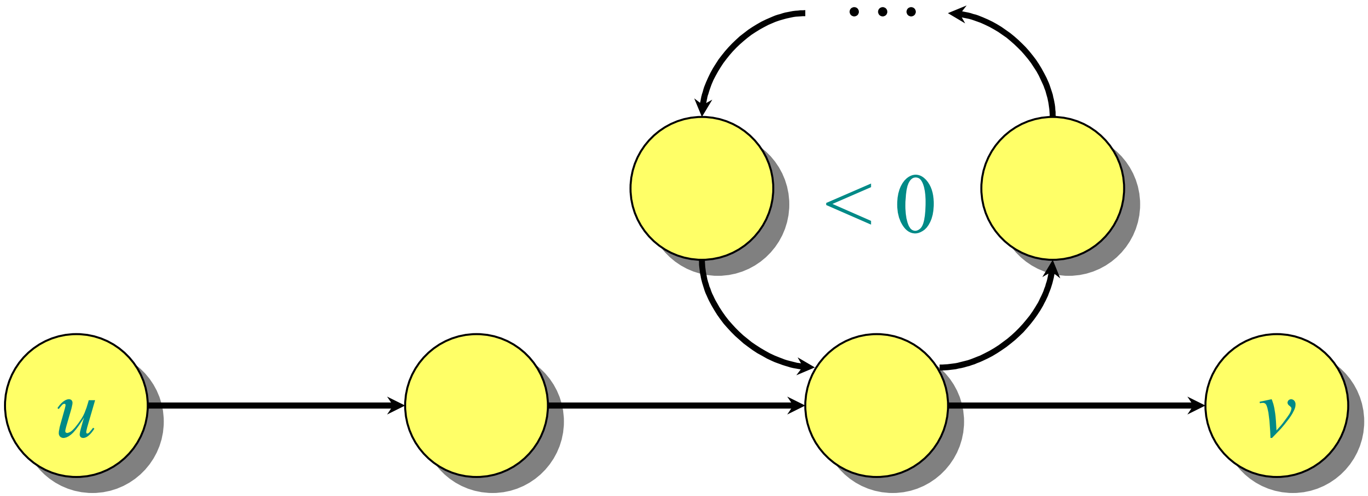


□

Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Single-source shortest paths

Problem. From a given source vertex $s \in V$, find the shortest-path weights $\delta(s, v)$ for all $v \in V$.

If all edge weights $w(u, v)$ are *nonnegative*, all shortest-path weights must exist.

IDEA: Greedy.

1. Maintain a set S of vertices whose shortest-path distances from s are known.
2. At each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
3. Update the distance estimates of vertices adjacent to v.

Dijkstra's algorithm

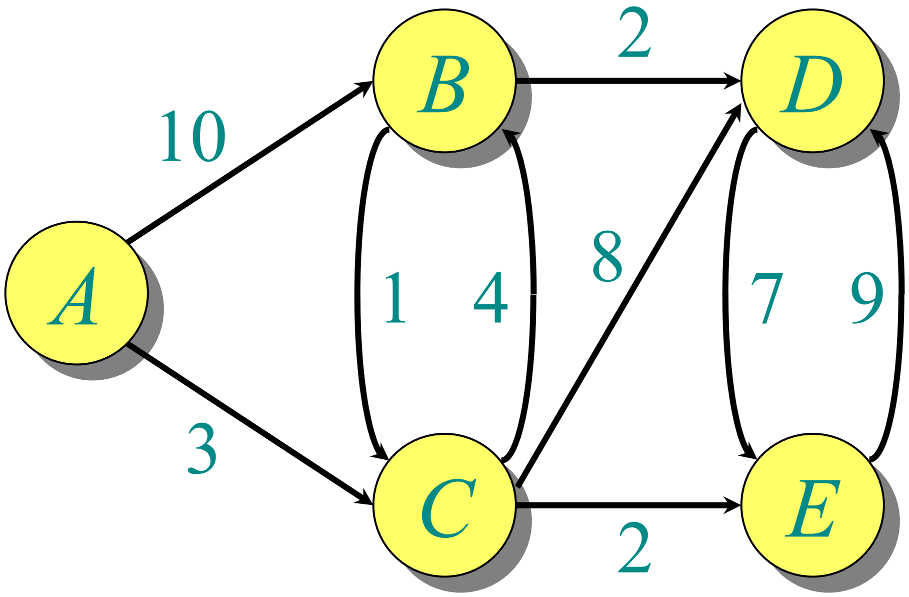
```
$d[s] \leftarrow 0$
**for** each $v \in V - \{s\}$
    **do** $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$            ⊳ $Q$ is a priority queue maintaining $V - S$
**while** $Q \neq \varnothing$
    **do** $u \leftarrow$ EXTRACT-MIN$(Q)$
        $S \leftarrow S \cup \{u\}$
        **for** each $v \in Adj[u]$
            **do if** $d[v] > d[u] + w(u, v)$
                 **then** $d[v] \leftarrow d[u] + w(u, v)$
```

relaxation step

Implicit DECREASE-KEY

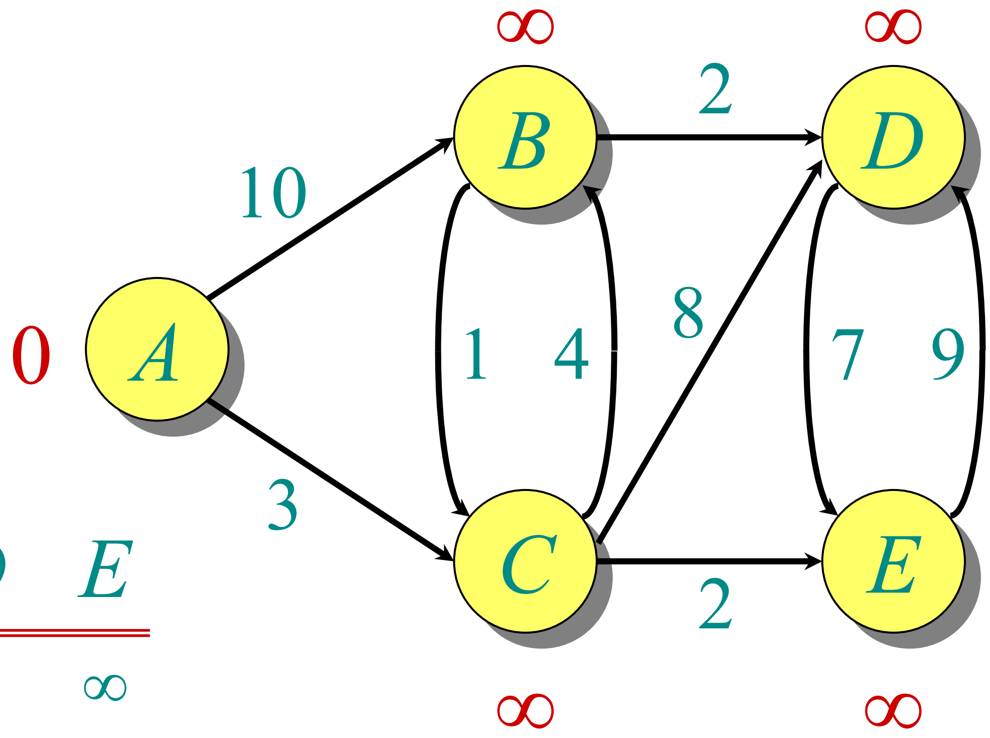
Example of Dijkstra's algorithm

Graph with nonnegative edge weights:



Example of Dijkstra's algorithm

Initialize:

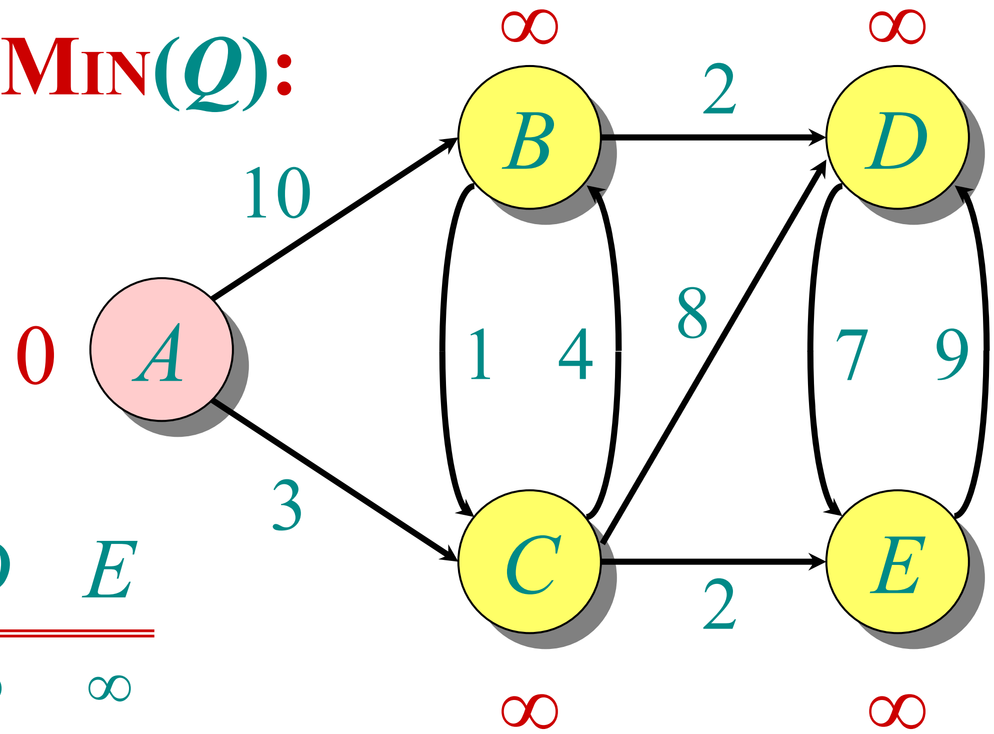


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

"*A*" ← **EXTRACT-MIN**(*Q*):

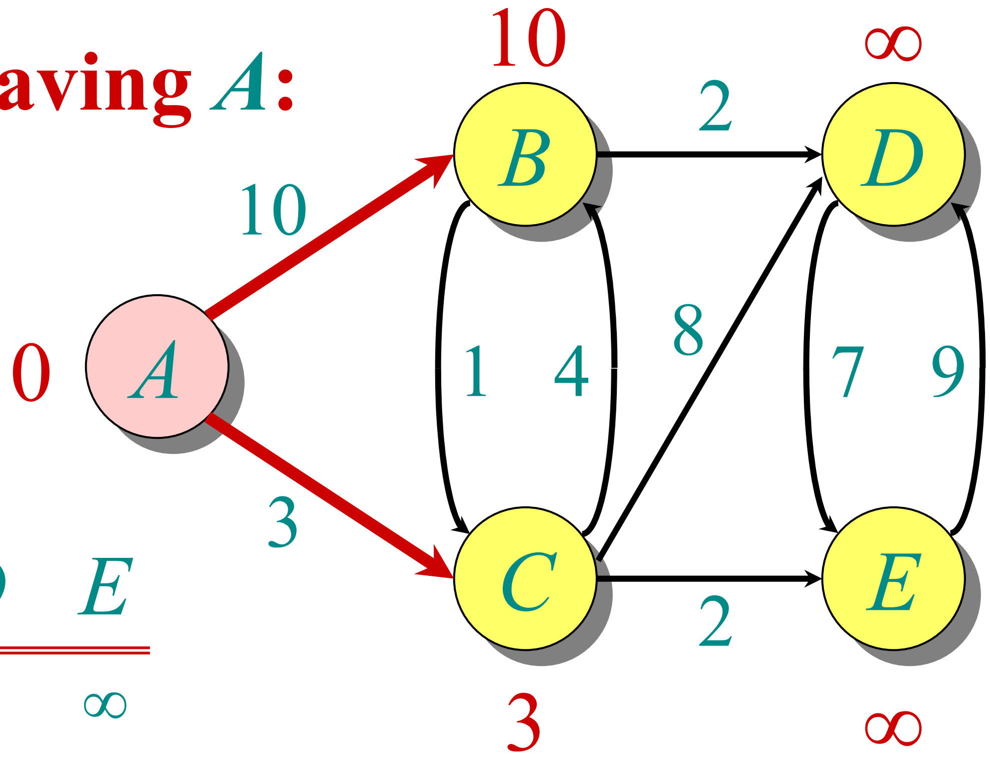


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

Relax all edges leaving *A*:

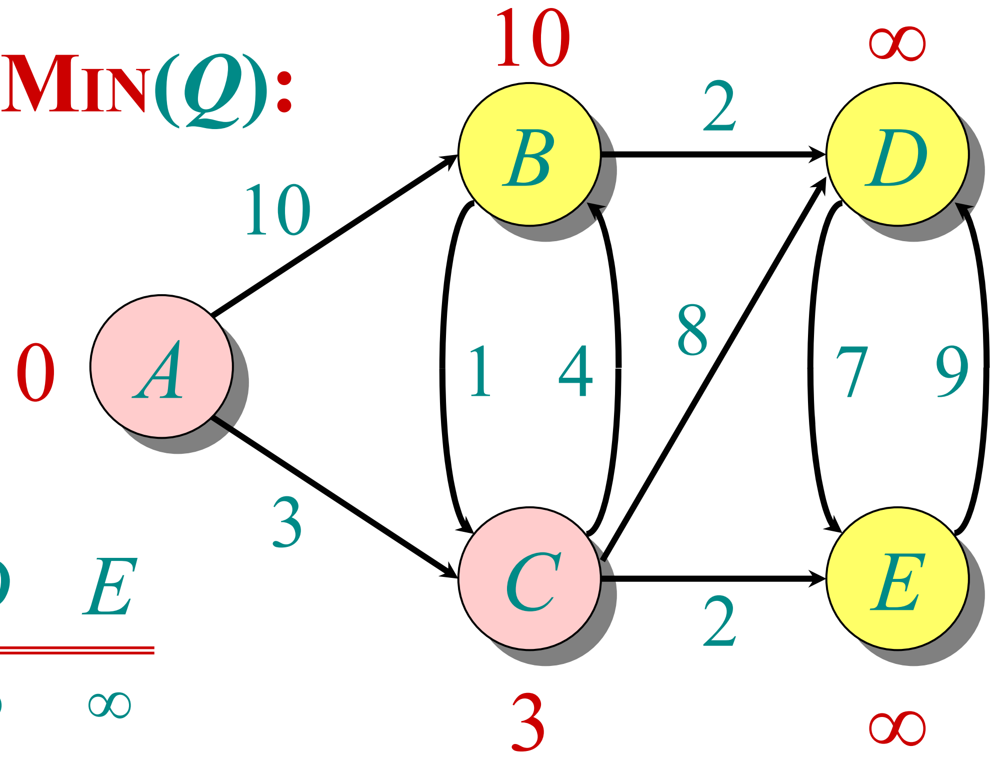


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	–	–

S: { *A* }

Example of Dijkstra's algorithm

"C" ← EXTRACT-MIN(Q):

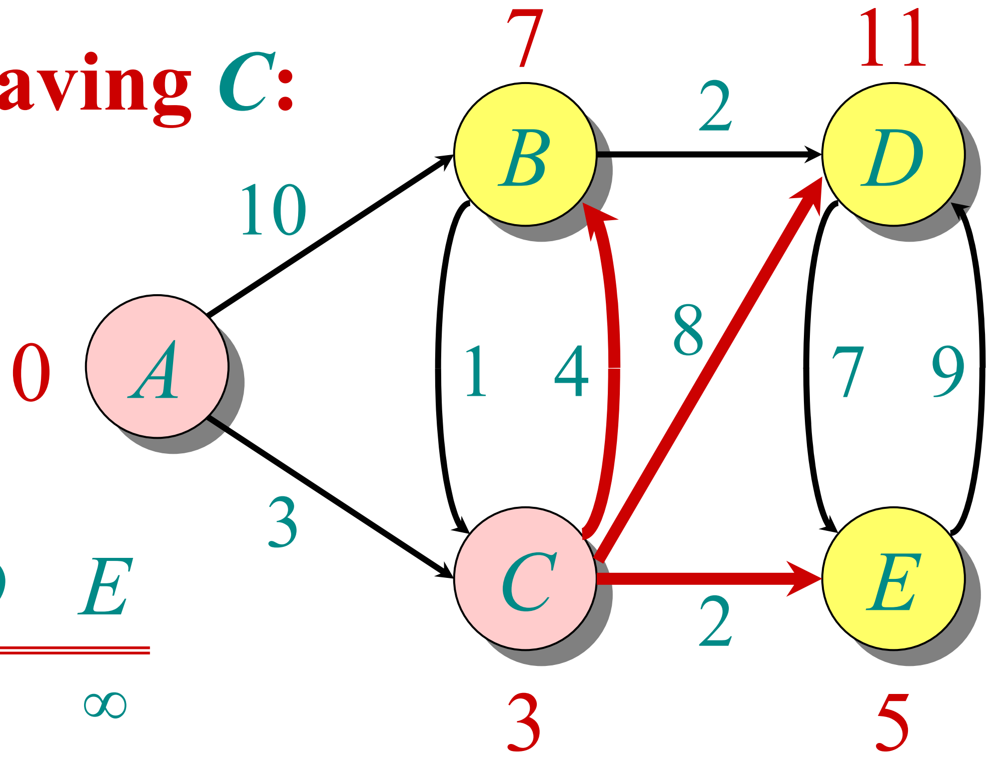


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S: { A, C }

Example of Dijkstra's algorithm

Relax all edges leaving *C*:

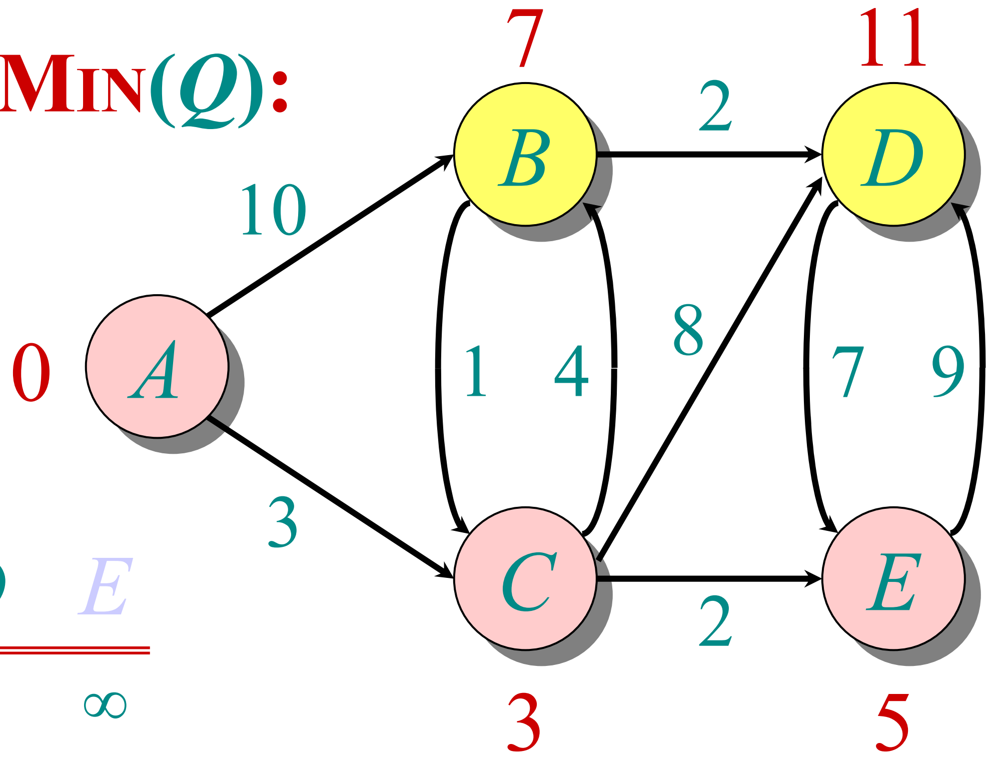


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S: { *A*, *C* }

Example of Dijkstra's algorithm

"E" ← EXTRACT-MIN(*Q*):

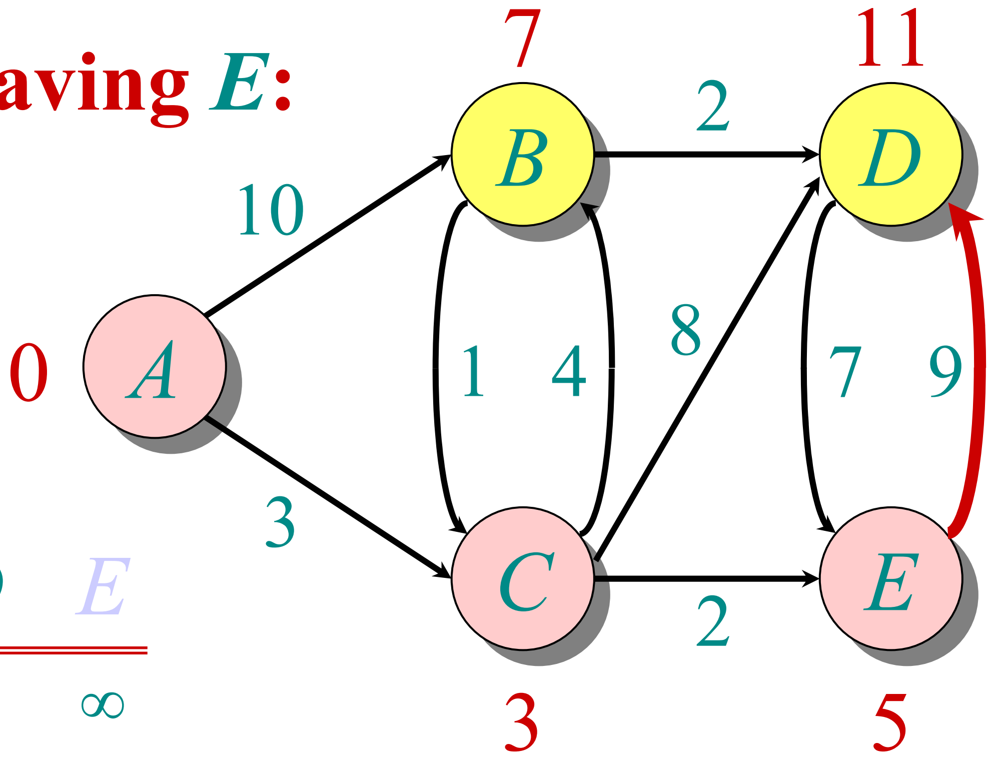


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

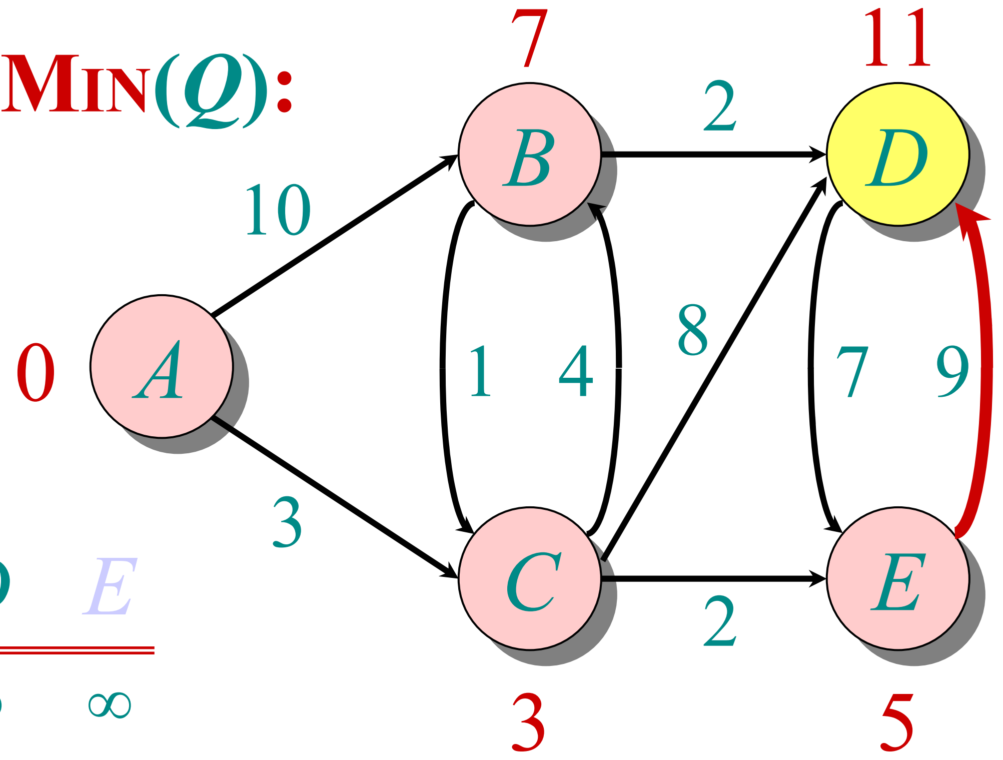


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

"B" ← EXTRACT-MIN(Q):

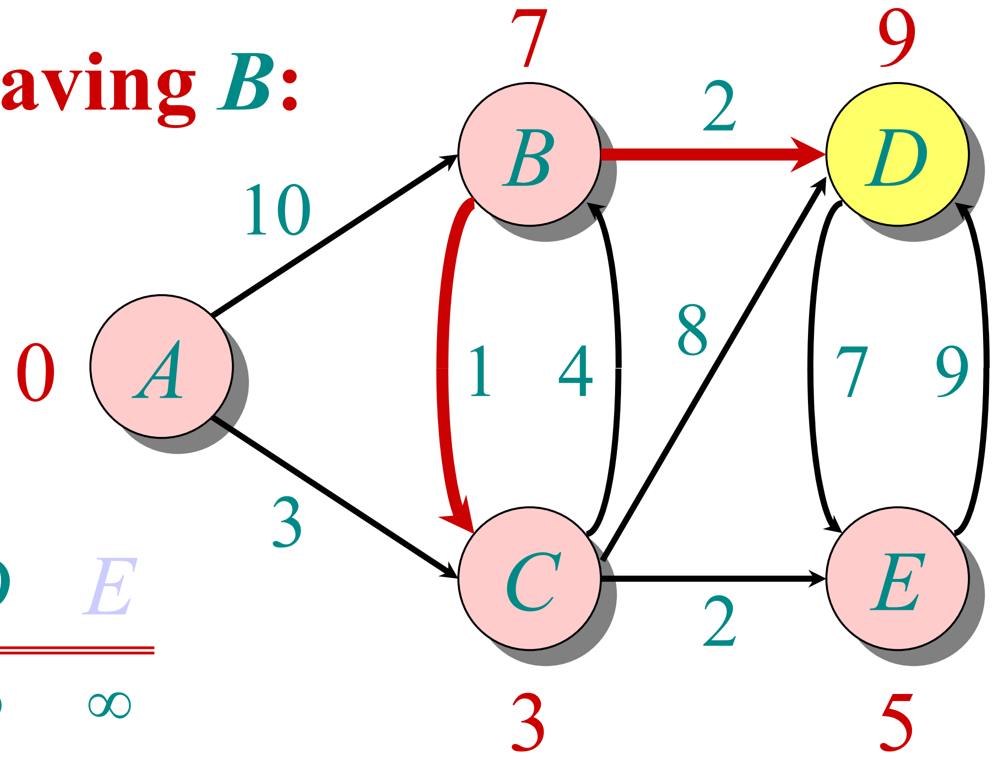


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example of Dijkstra’s algorithm

Relax all edges leaving *B*:

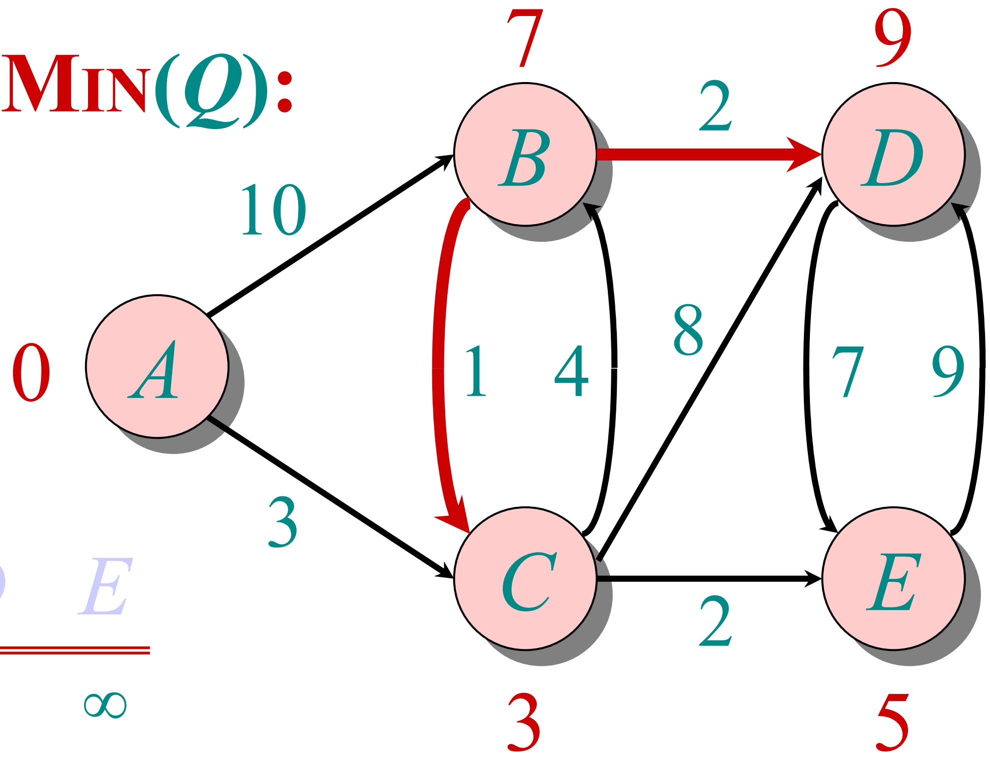


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

"*D*" ← EXTRACT-MIN(*Q*):



Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }

Correctness — Part I

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Suppose not. Let v be the first vertex for which $d[v] < \delta(s, v)$, and let u be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u, v)$. Then,

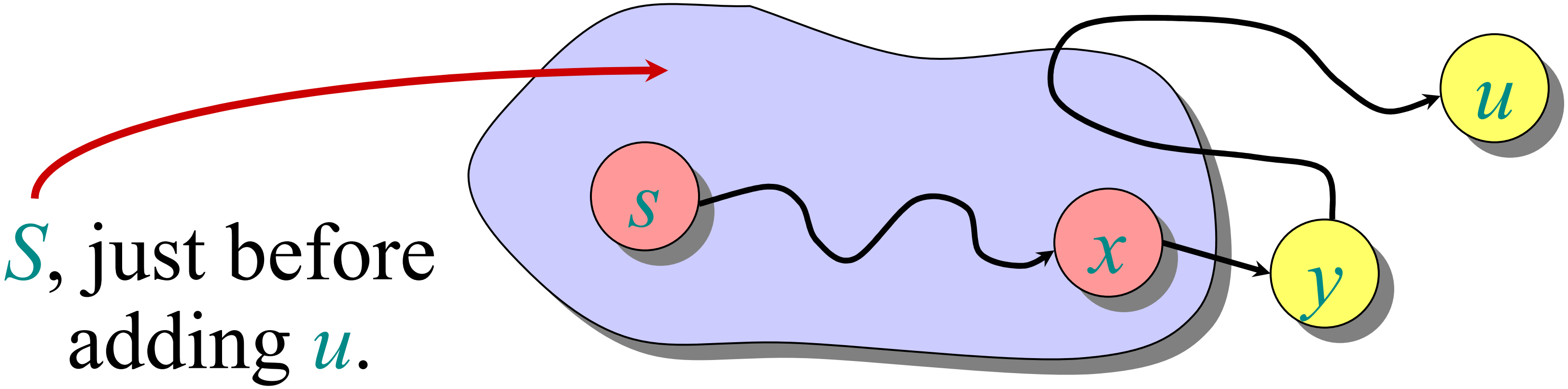
$$\begin{aligned} d[v] &< \delta(s, v) && \text{supposition} \\ &\leq \delta(s, u) + \delta(u, v) && \text{triangle inequality} \\ &\leq \delta(s,u) + w(u, v) && \text{sh. path} \leq \text{specific path} \\ &\leq d[u] + w(u, v) && v \text{ is first violation} \end{aligned}$$

Contradiction. □

Correctness — Part II

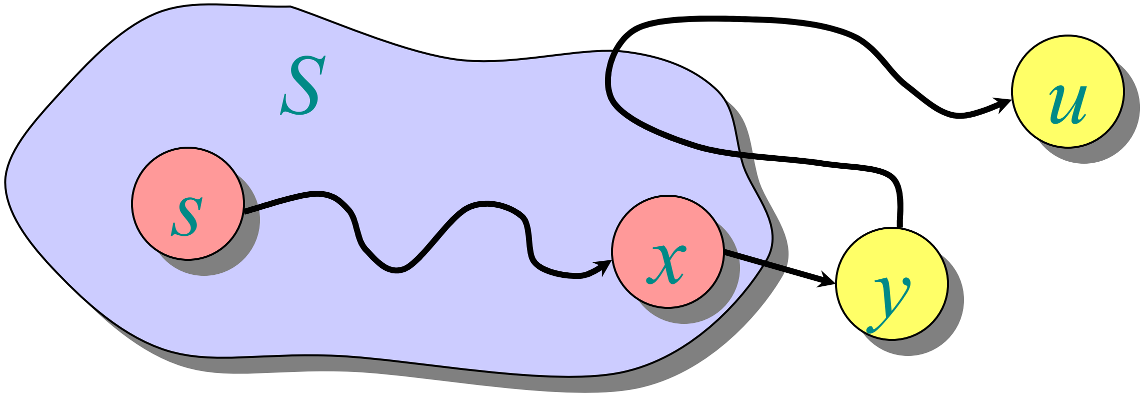
Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] \neq \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:



S, just before adding u.

Correctness — Part II (continued)



Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. Since subpaths of shortest paths are shortest paths, it follows that $d[y]$ was set to $\delta(s, x) + w(x, y) = \delta(s, y)$ when (x, y) was relaxed just after x was added to S. Consequently, we have $d[y] = \delta(s, y) \le \delta(s, u) \le d[u]$. But, $d[u] \le d[y]$ by our choice of u, and hence $d[y] = \delta(s, y) = \delta(s, u) = d[u]$. Contradiction. □

Analysis of Dijkstra

$|V|$ times

```
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN$(Q)$
       $S \leftarrow S \cup \{u\}$
       for each $v \in Adj[u]$
           do if $d[v] > d[u] + w(u, v)$
                 then $d[v] \leftarrow d[u] + w(u, v)$
```

degree(u) times

Handshaking Lemma ⇒ $\Theta(E)$ implicit DECREASE-KEY's.

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.

Analysis of Dijkstra (continued)

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Unweighted graphs

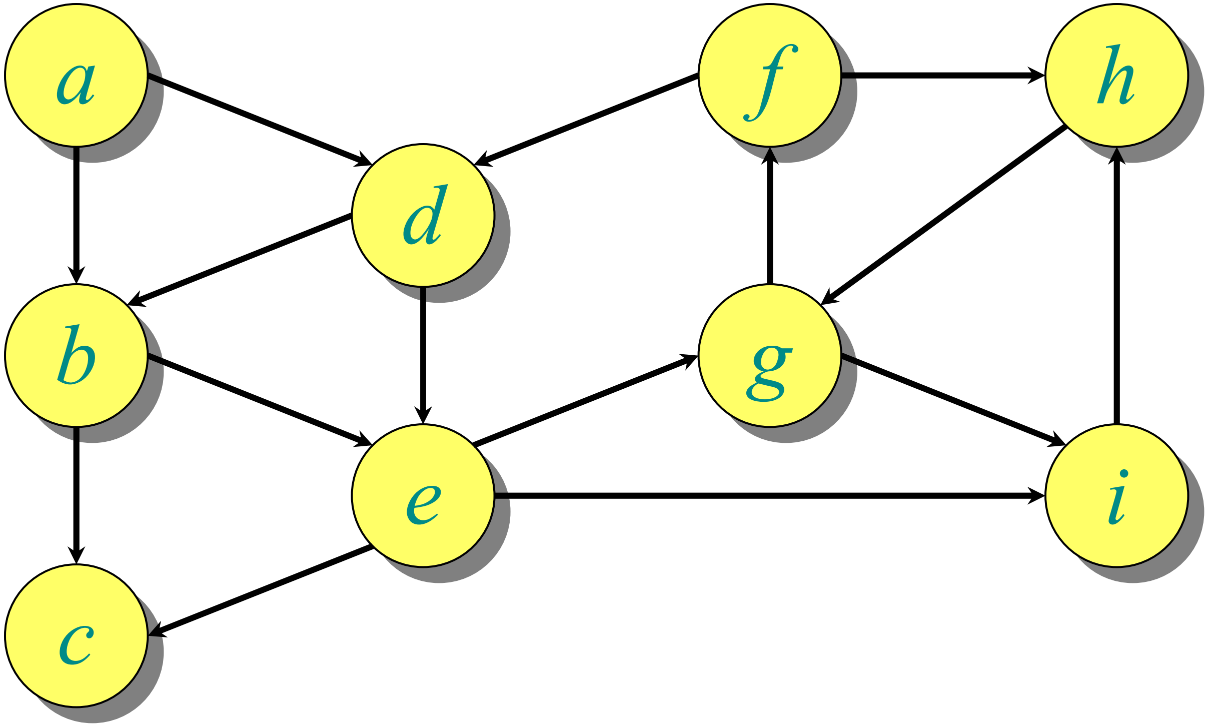
Suppose $w(u, v) = 1$ for all $(u, v) \in E$. Can the code for Dijkstra be improved?

- Use a simple FIFO queue instead of a priority queue.
- ***Breadth-first search***

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE$(Q)$
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                     ENQUEUE$(Q, v)$
```

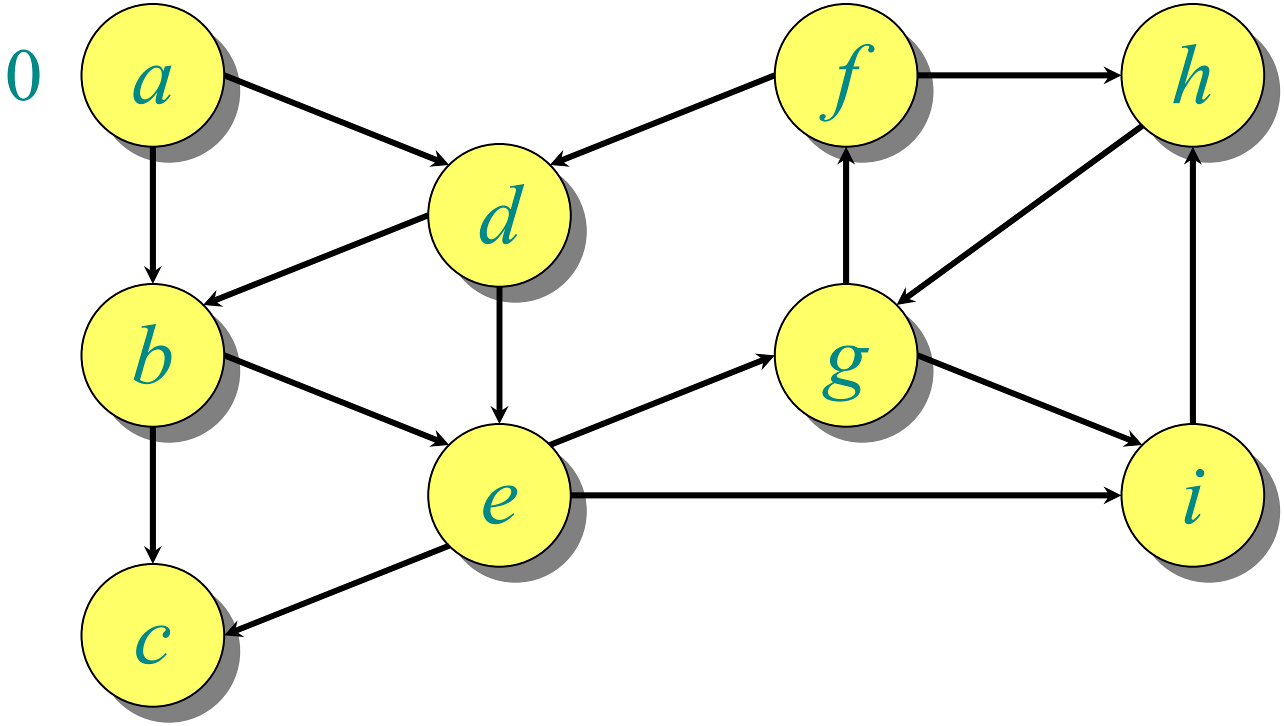
Analysis: Time = $O(V + E)$.

Example of breadth-first search



Q:

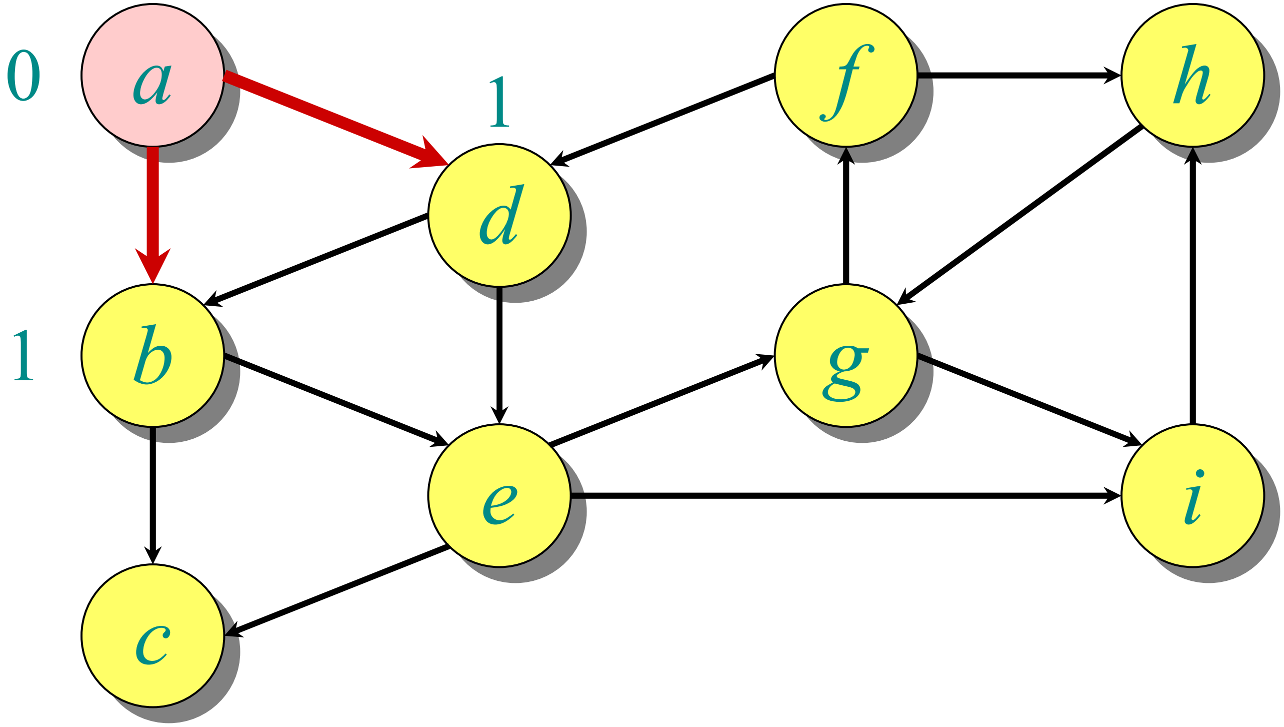
Example of breadth-first search



0

Q: *a*

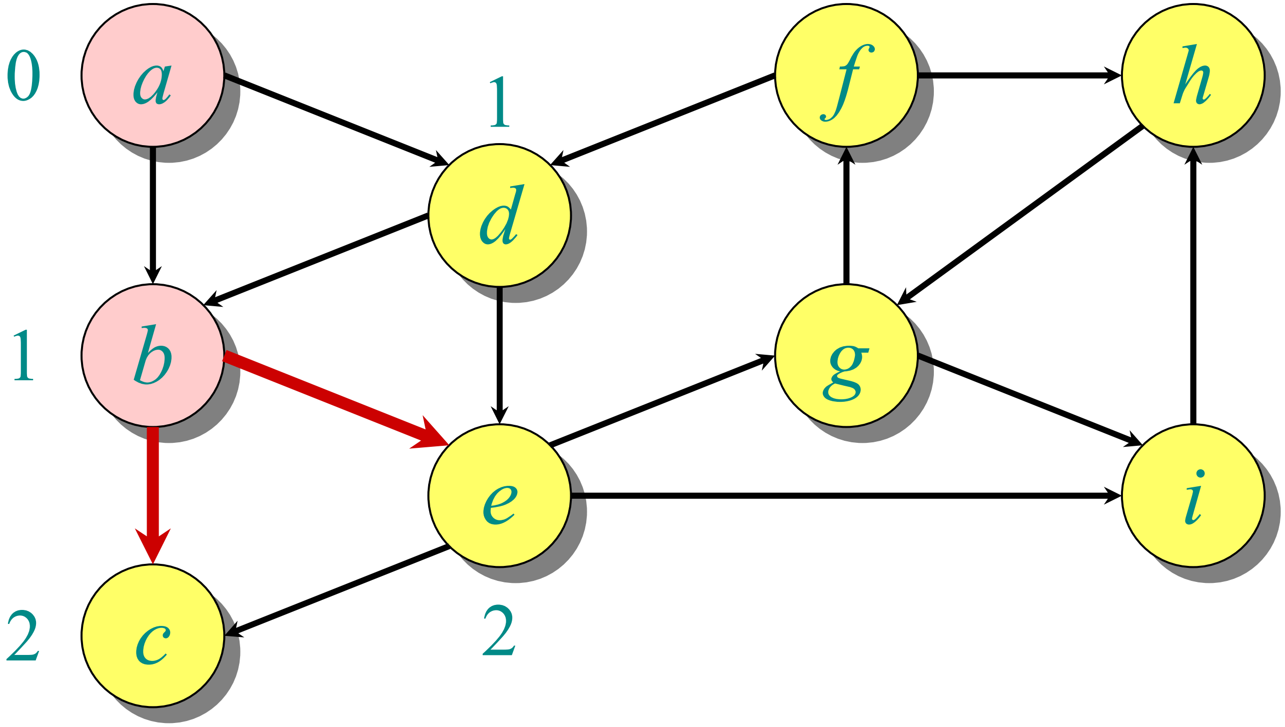
Example of breadth-first search



1 1

Q: *a* *b* *d*

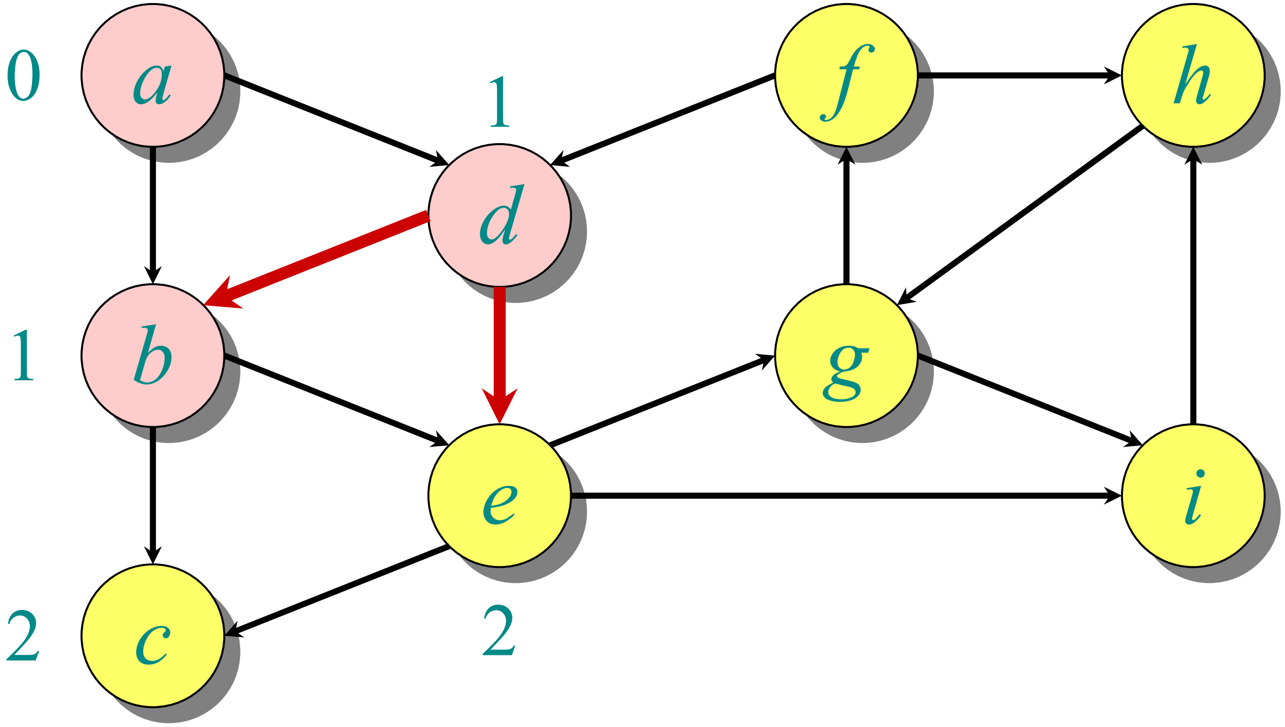
Example of breadth-first search



1 2 2

Q: a b d c e

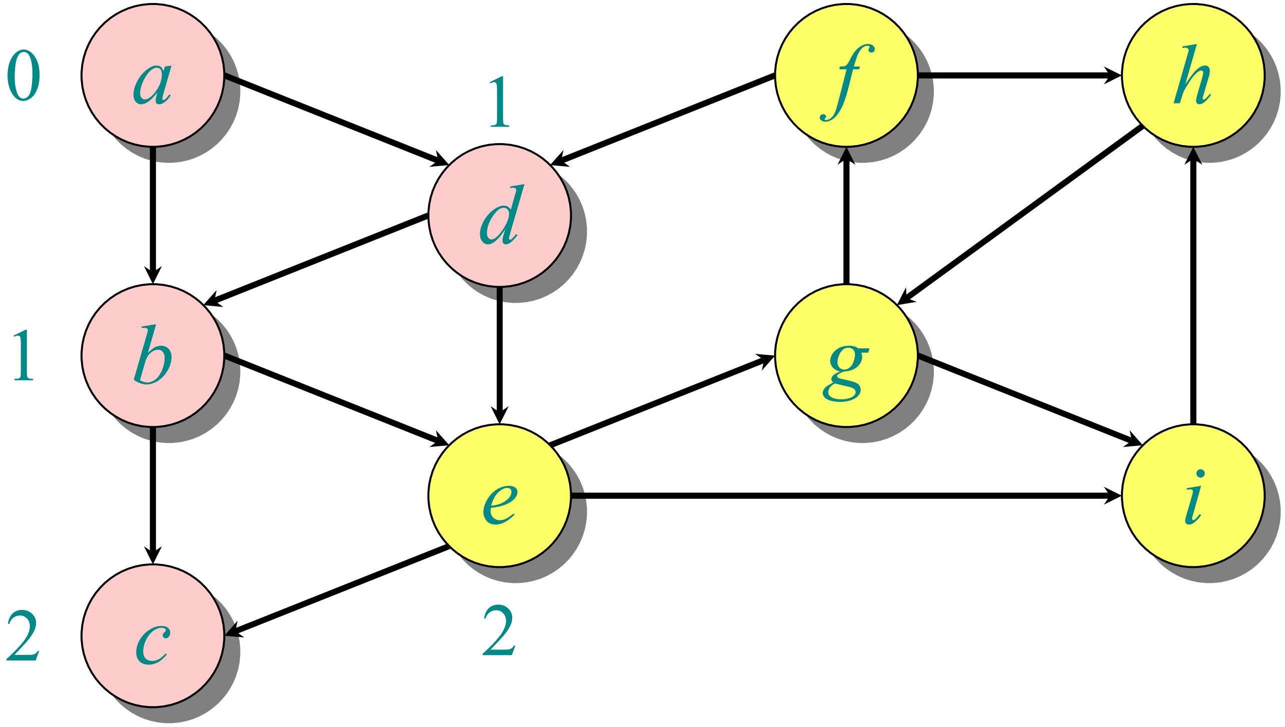
Example of breadth-first search



2 2

Q: *a b d c e*

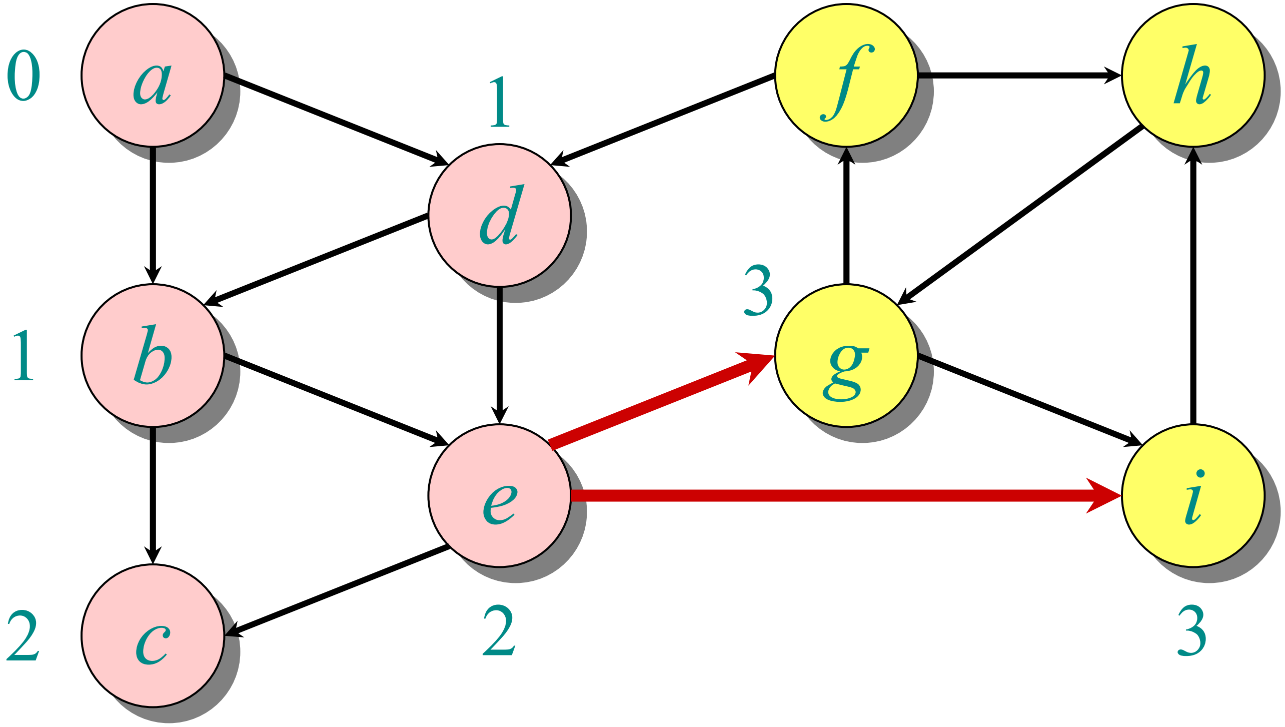
Example of breadth-first search



2

Q: *a b d c e*

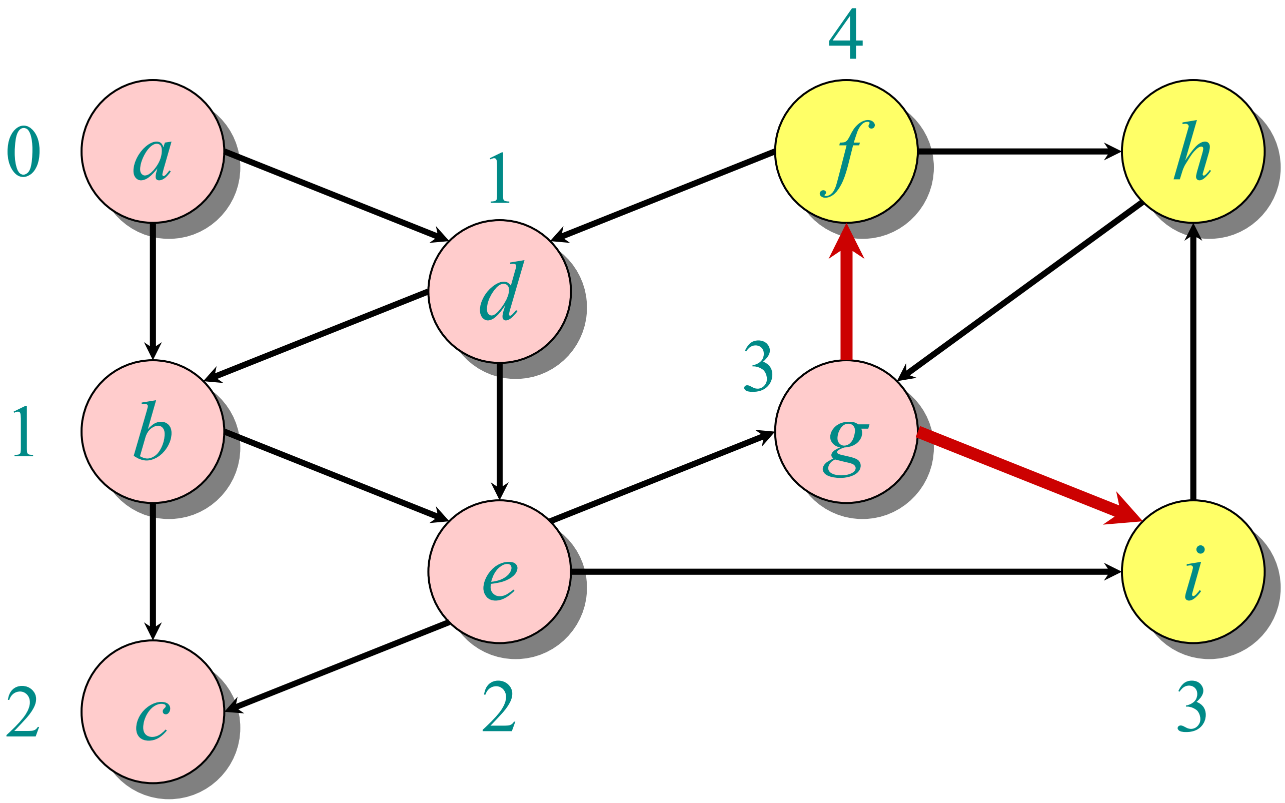
Example of breadth-first search



3 3

Q: *a* *b* *d* *c* *e* *g* *i*

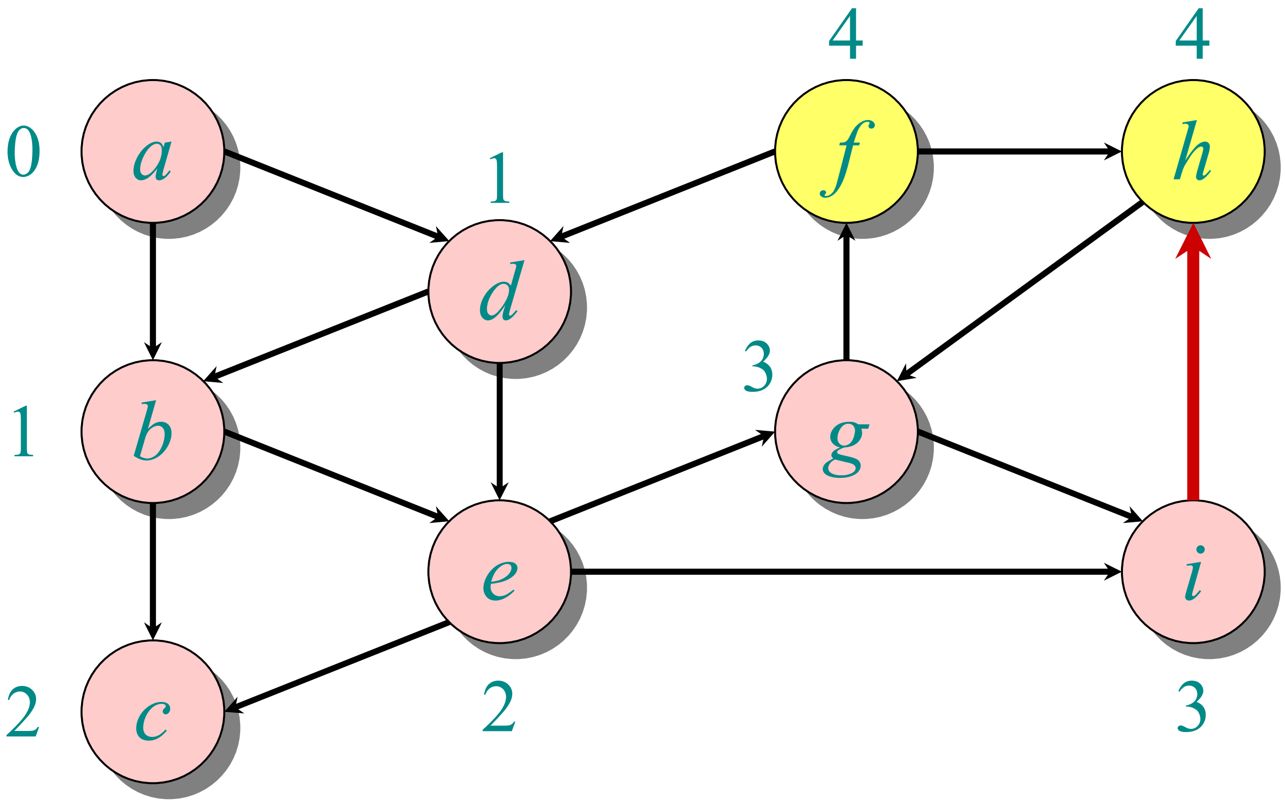
Example of breadth-first search



3 4

Q: *a b d c e g i f*

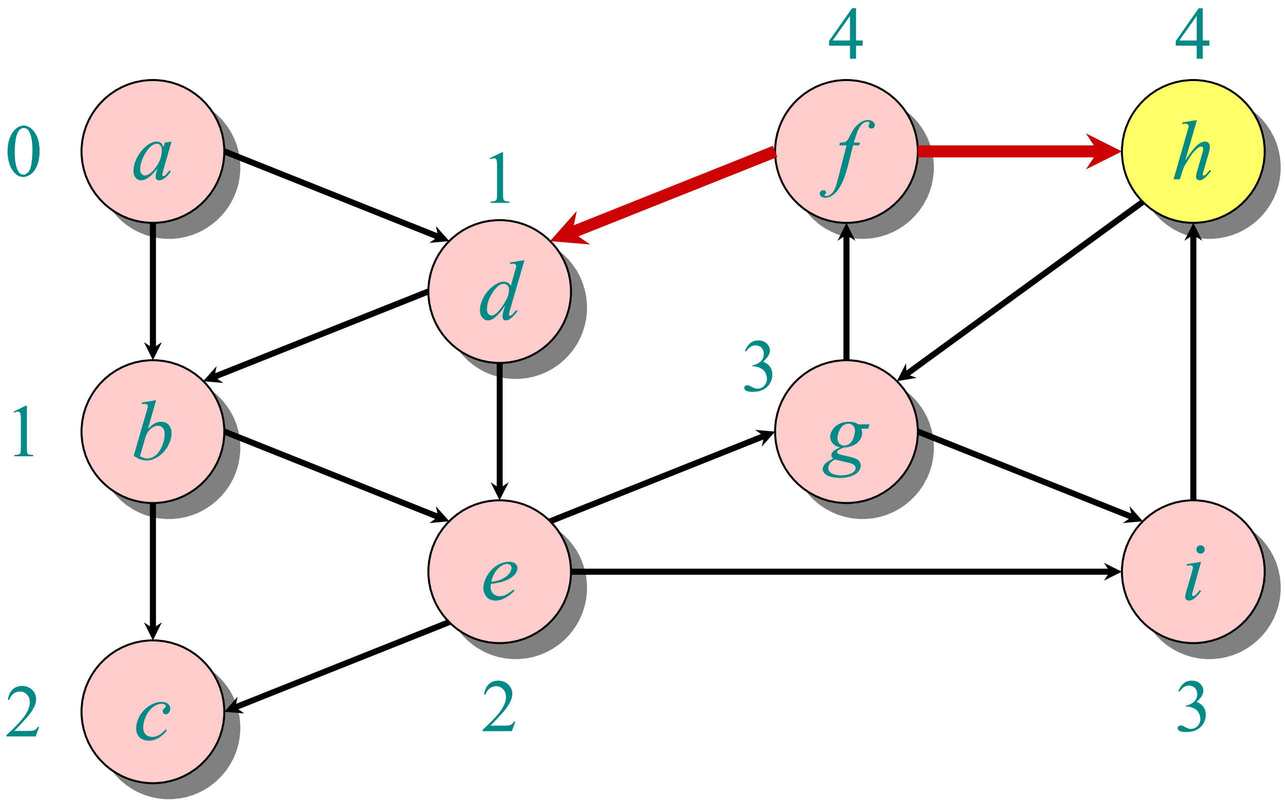
Example of breadth-first search



4 4

Q: *a b d c e g i f h*

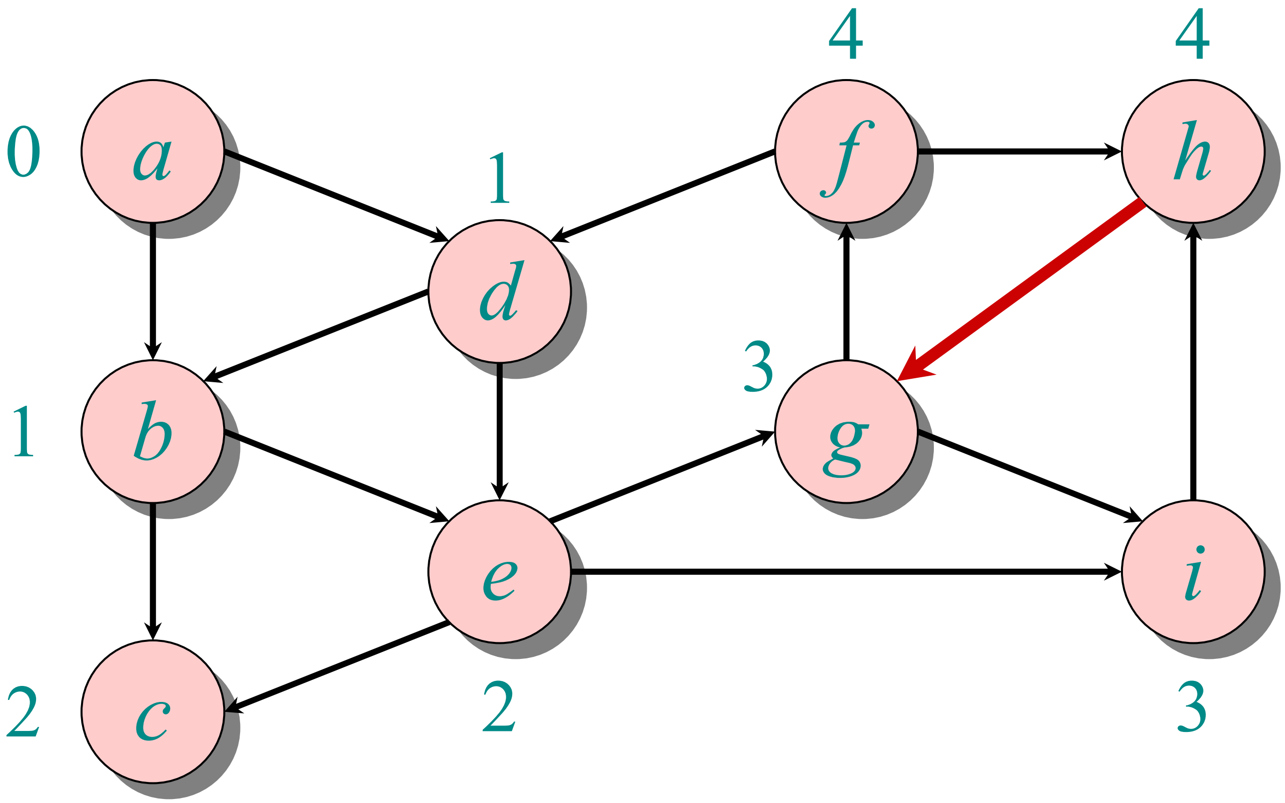
Example of breadth-first search



4

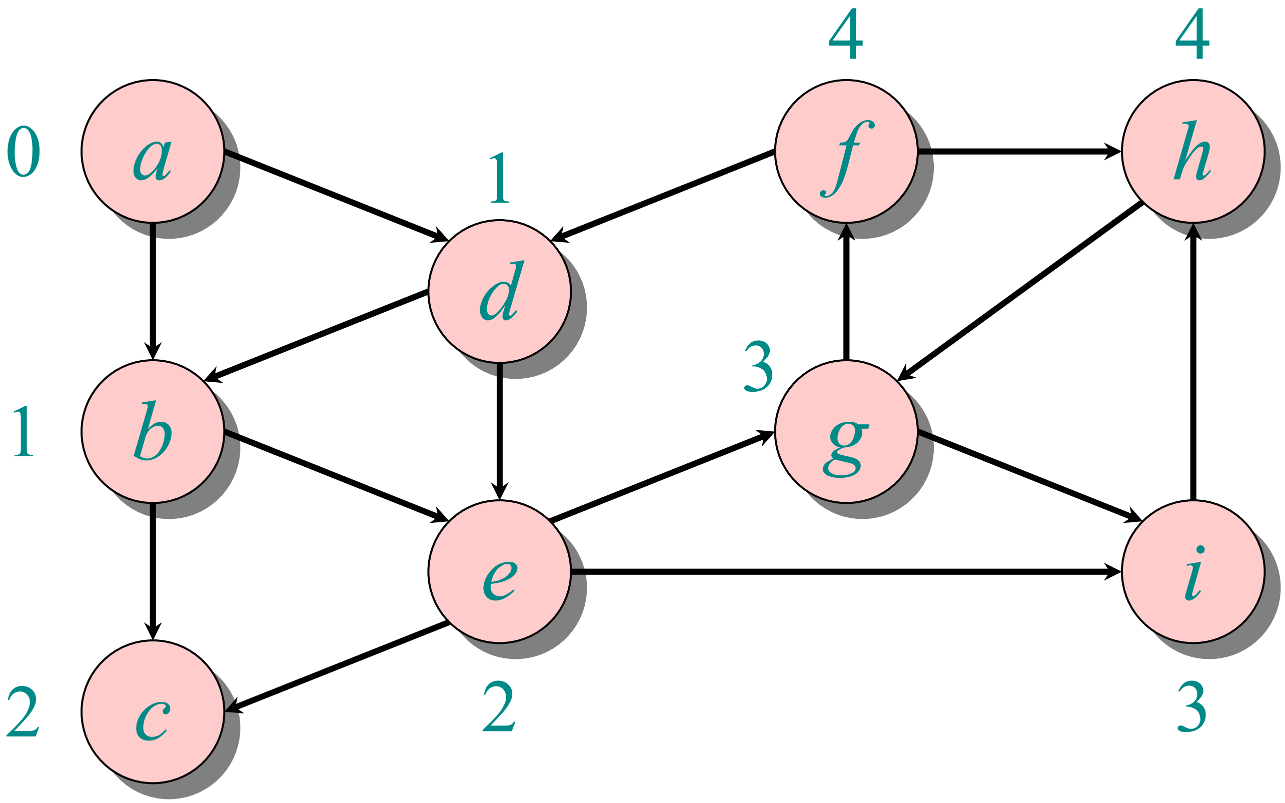
Q: *a b d c e g i f h*

Example of breadth-first search



Q: *a b d c e g i f h*

Example of breadth-first search



Q: *a b d c e g i f h*

Correctness of BFS

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                     ENQUEUE($Q$, $v$)
```

Key idea:

The FIFO Q in breadth-first search mimics the priority queue Q in Dijkstra.

- **Invariant:** v comes after u in Q implies that $d[v] = d[u]$ or $d[v] = d[u] + 1$.

Introduction to Algorithms

6.046J/18.401J/SMA5503

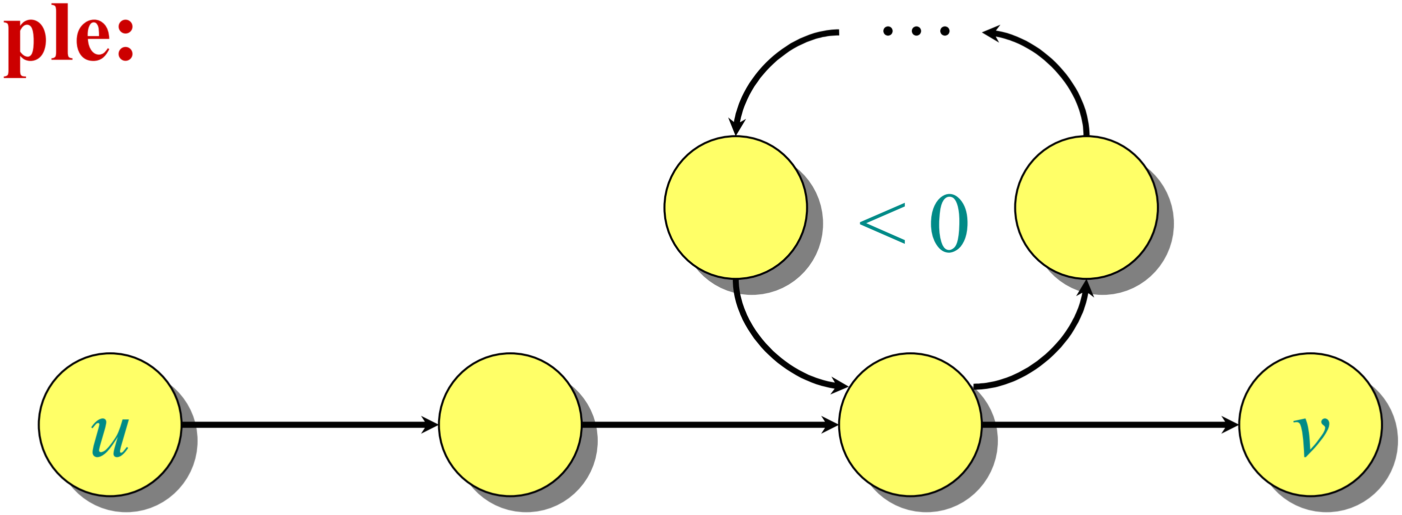
Lecture 18

Prof. Erik Demaine

Negative-weight cycles

Recall: If a graph $G = (V, E)$ contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Bellman-Ford algorithm: Finds all shortest-path lengths from a ***source*** $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Bellman-Ford algorithm

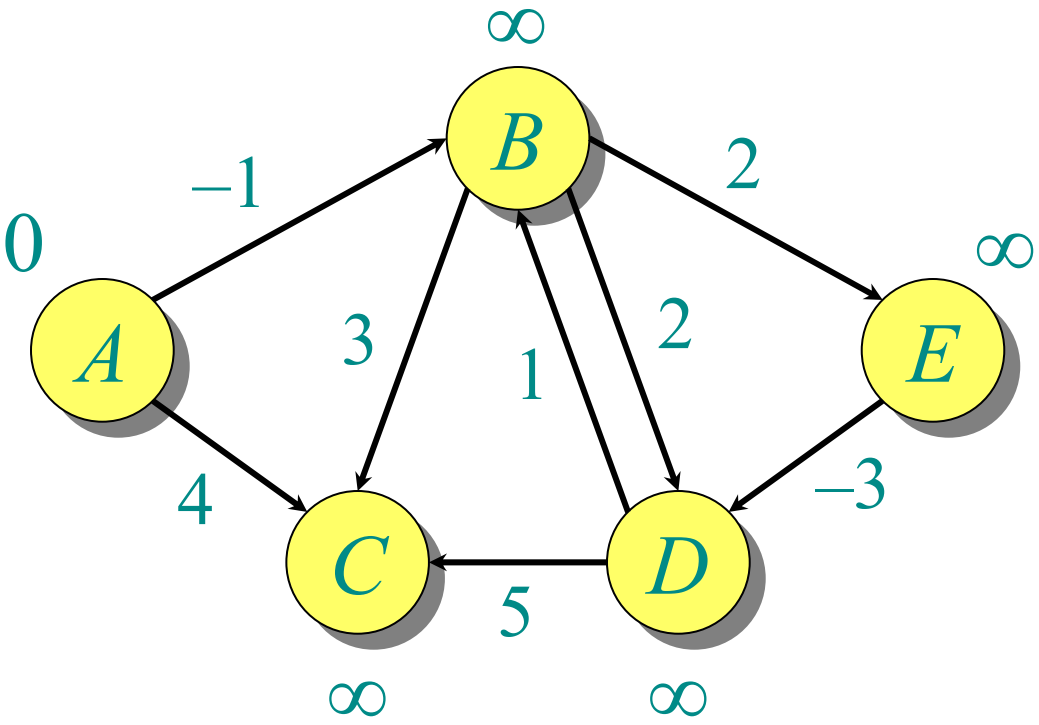
```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$                         } initialization
    do $d[v] \leftarrow \infty$

for $i \leftarrow 1$ to $|V| - 1$
    do for each edge $(u, v) \in E$
        do if $d[v] > d[u] + w(u, v)$               } relaxation
            then $d[v] \leftarrow d[u] + w(u, v)$   } step

for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
        then report that a negative-weight cycle exists
```

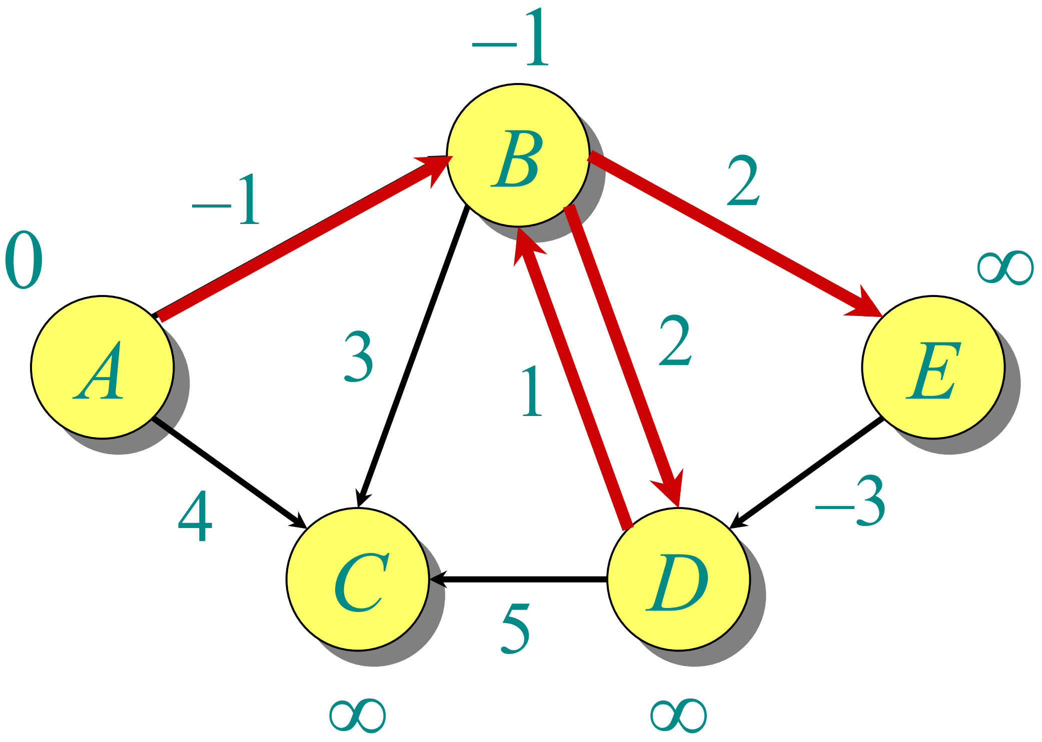
At the end, $d[v] = \delta(s, v)$. Time $= O(VE)$.

Example of Bellman-Ford



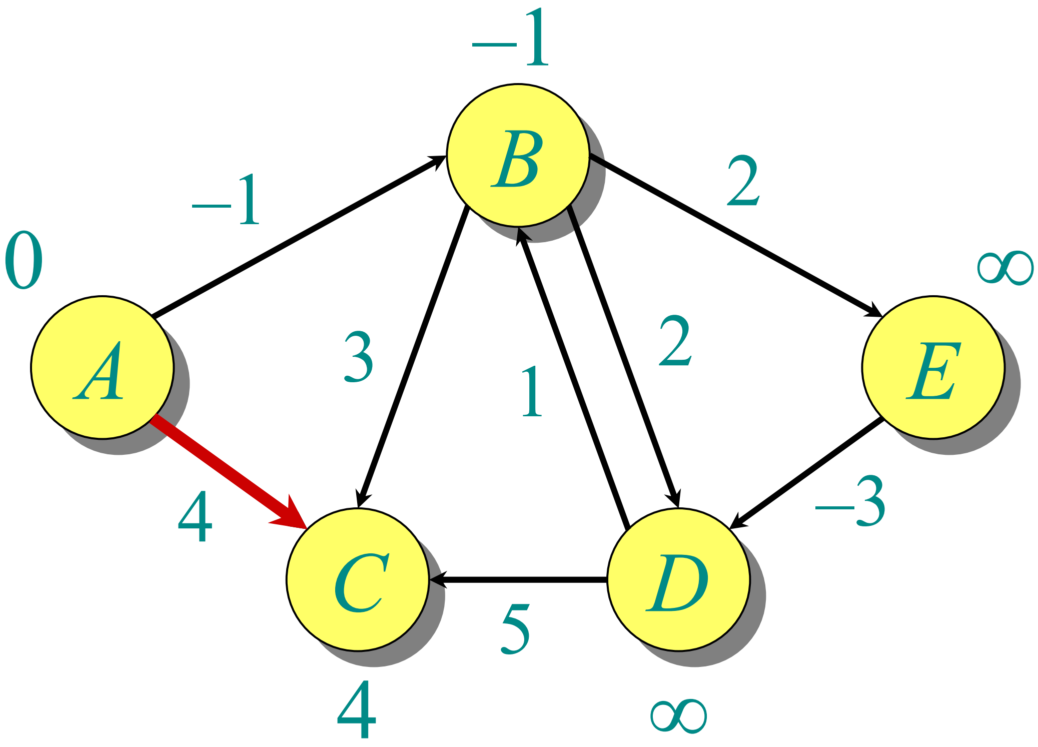
A	B	C	D	E
0	∞	∞	∞	∞

Example of Bellman-Ford



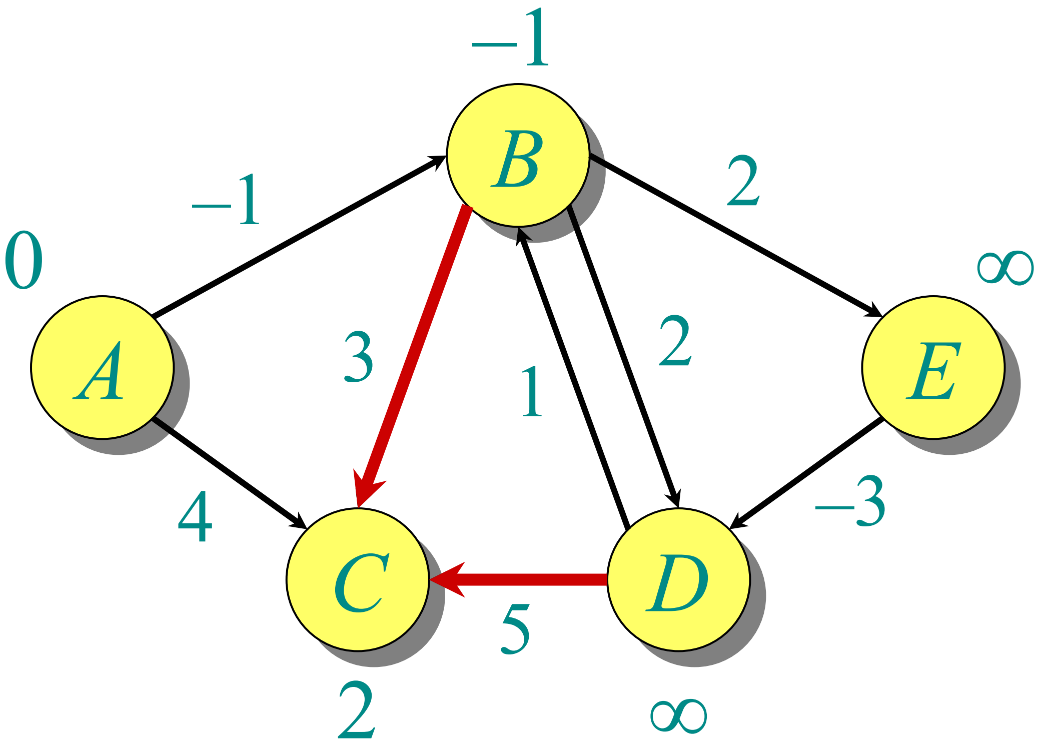
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞

Example of Bellman-Ford



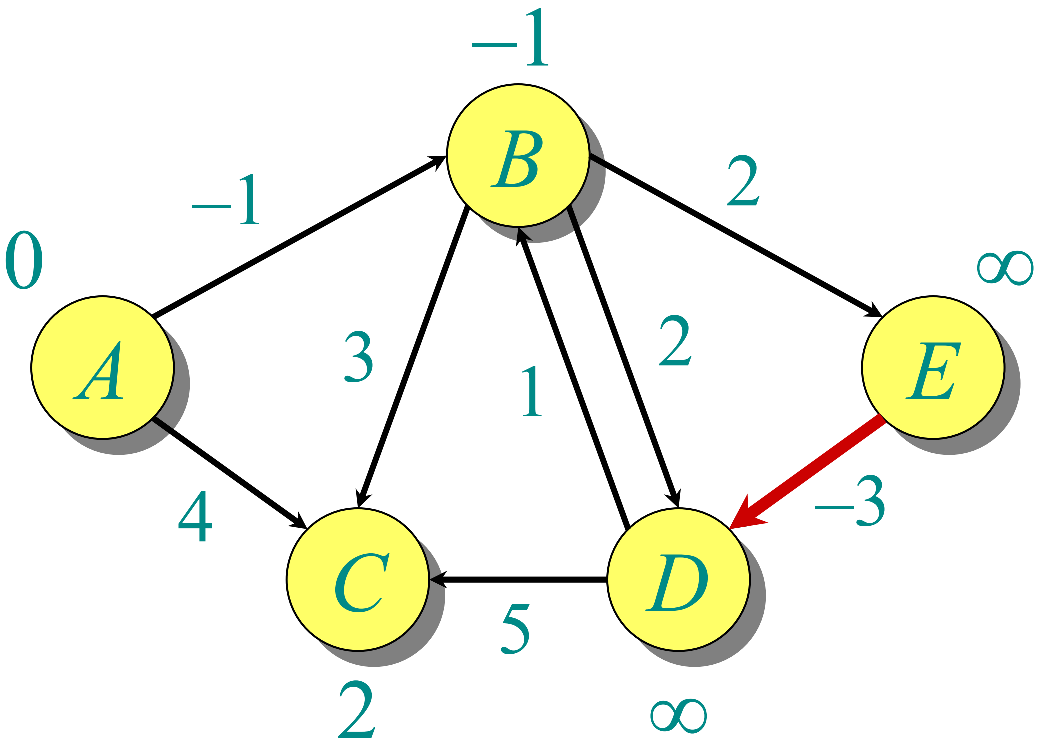
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞

Example of Bellman-Ford



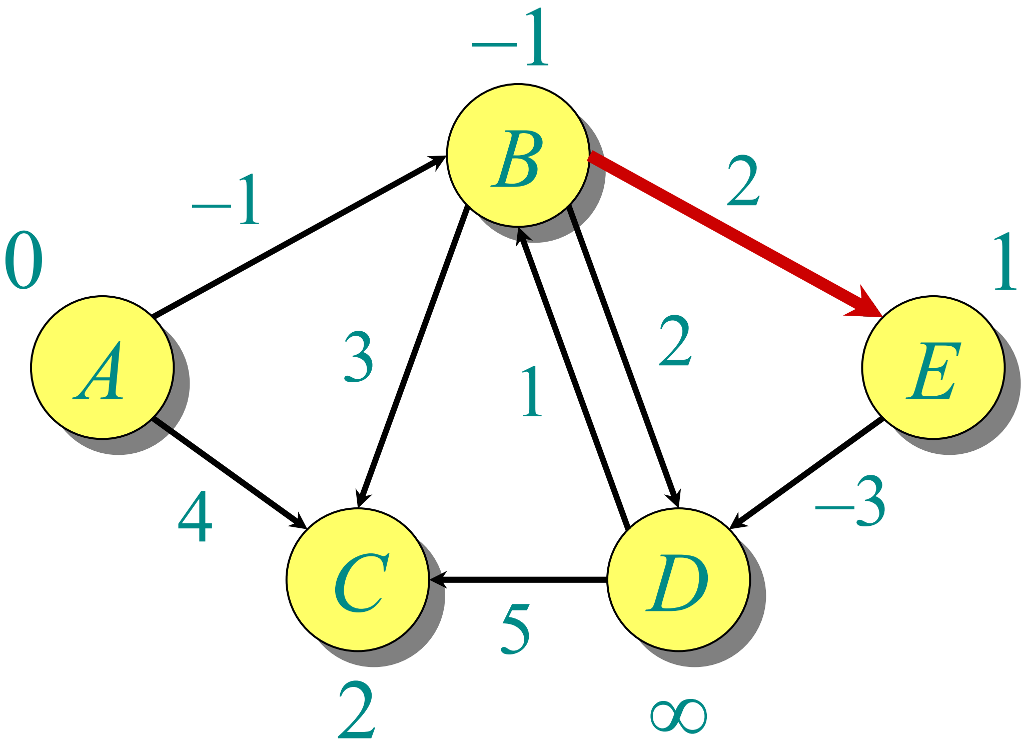
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞

Example of Bellman-Ford



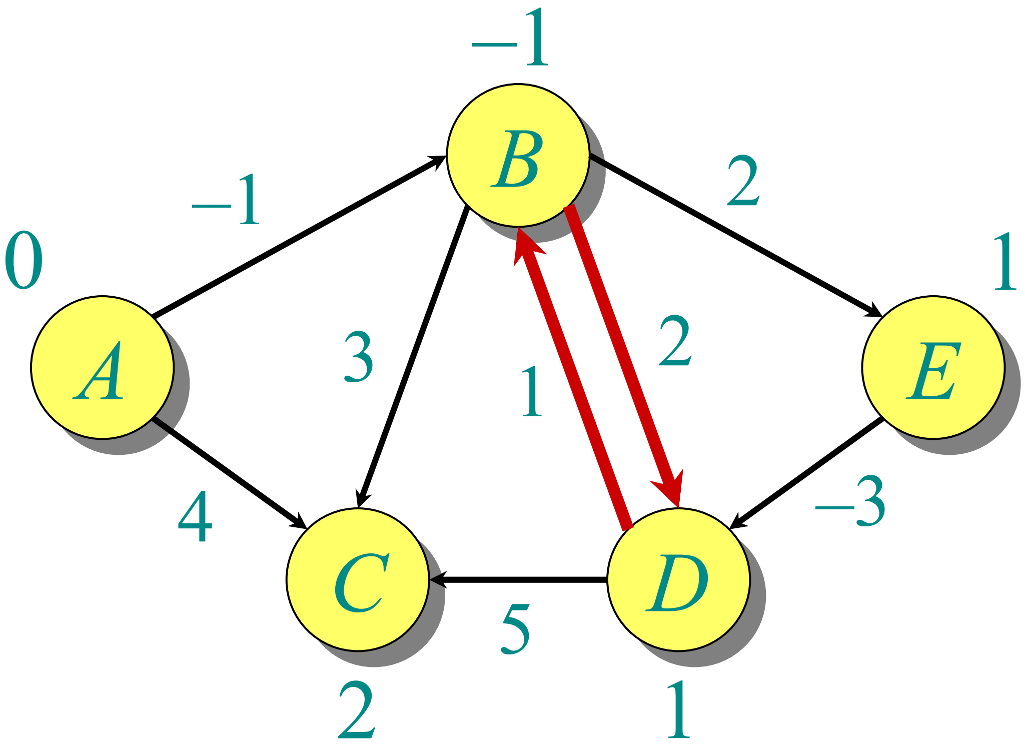
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞

Example of Bellman-Ford



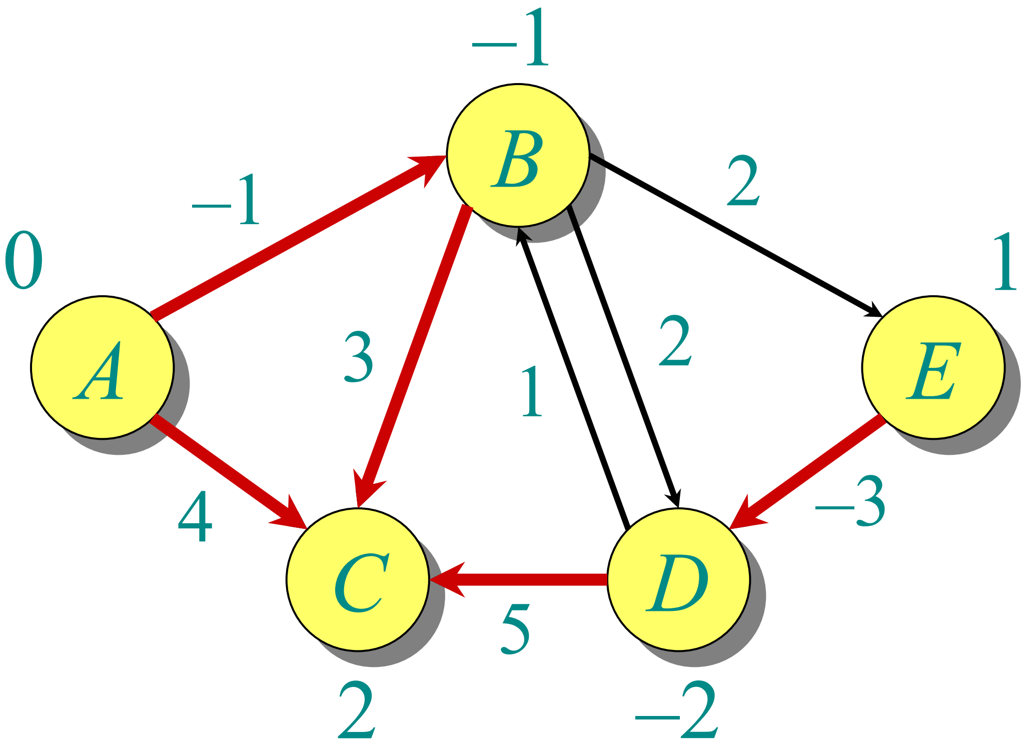
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1

Example of Bellman-Ford



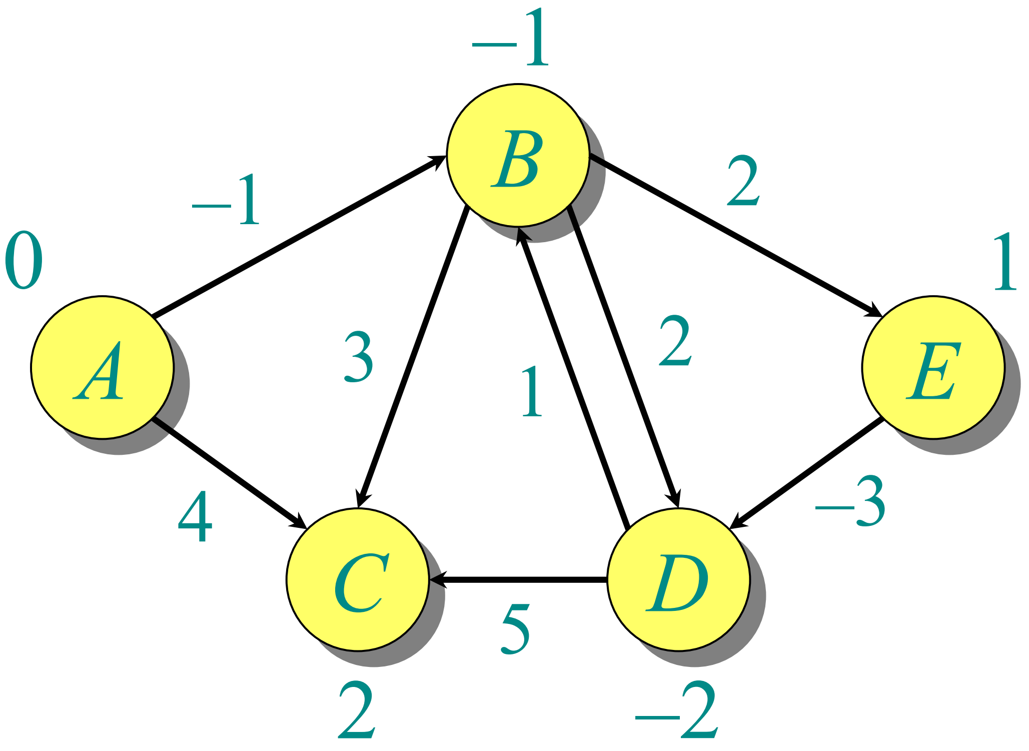
A	B	C	D	E
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1
0	–1	2	1	1

Example of Bellman-Ford



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1
0	–1	2	1	1
0	–1	2	–2	1

Example of Bellman-Ford



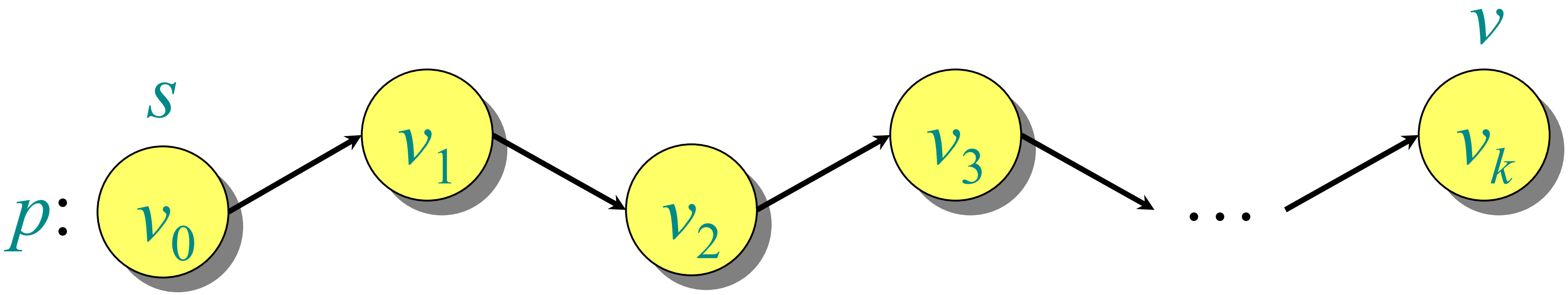
A	B	C	D	E
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1
0	–1	2	1	1
0	–1	2	–2	1

Note: Values decrease monotonically.

Correctness

Theorem. If $G = (V, E)$ contains no negative-weight cycles, then after the Bellman-Ford algorithm executes, $d[v] = \delta(s, v)$ for all $v \in V$.

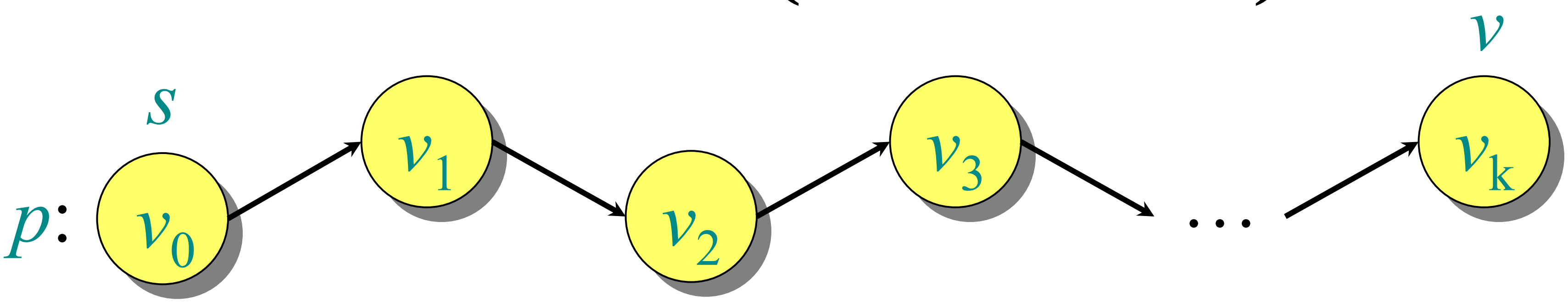
Proof. Let $v \in V$ be any vertex, and consider a shortest path p from s to v with the minimum number of edges.



Since p is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \ .$$

Correctness (continued)



Initially, $d[v_0] = 0 = \delta(s, v_0)$, and $d[s]$ is unchanged by subsequent relaxations (because of the lemma from Lecture 17 that $d[v] \geq \delta(s, v)$).

- After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$.
- After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$.

  ⋮

- After k passes through E, we have $d[v_k] = \delta(s, v_k)$.

Since G contains no negative-weight cycles, p is simple. Longest simple path has $\leq |V| - 1$ edges. □

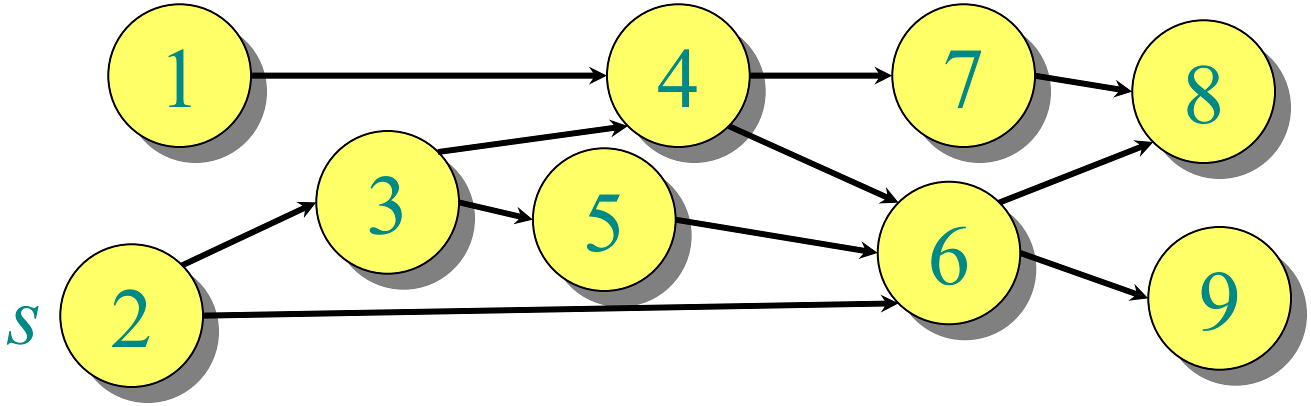
Detection of negative-weight cycles

Corollary. If a value $d[v]$ fails to converge after $|V| - 1$ passes, there exists a negative-weight cycle in G reachable from s. □

DAG shortest paths

If the graph is a ***directed acyclic graph*** (***DAG***), we first ***topologically sort*** the vertices.

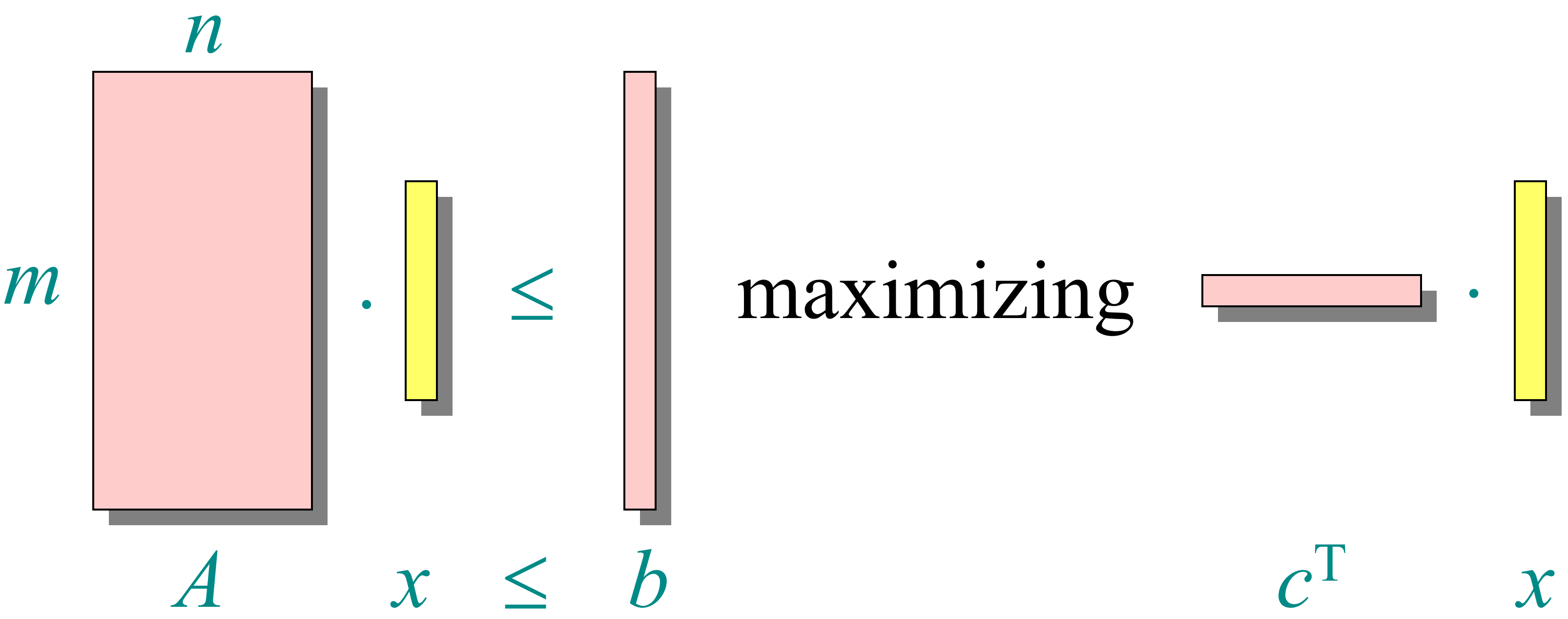
- Determine $f : V \to \{1, 2, \ldots, |V|\}$ such that $(u, v) \in E \Rightarrow f(u) < f(v)$.
- $O(V + E)$ time using depth-first search.



Walk through the vertices $u \in V$ in this order, relaxing the edges in $Adj[u]$, thereby obtaining the shortest paths from s in a total of $O(V + E)$ time.

Linear programming

Let A be an $m \times n$ matrix, b be an m-vector, and c be an n-vector. Find an n-vector x that maximizes $c^{\mathrm{T}}x$ subject to $Ax \le b$, or determine that no such solution exists.



Linear-programming algorithms

Algorithms for the general problem

- Simplex methods — practical, but worst-case exponential time.
- Ellipsoid algorithm — polynomial time, but slow in practice.
- Interior-point methods — polynomial time and competes with simplex.

Feasibility problem: No optimization criterion. Just find x such that $Ax \le b$.

- In general, just as hard as ordinary LP.

Solving a system of difference constraints

Linear programming where each row of A contains exactly one 1, one $–1$, and the rest 0’s.

Example:

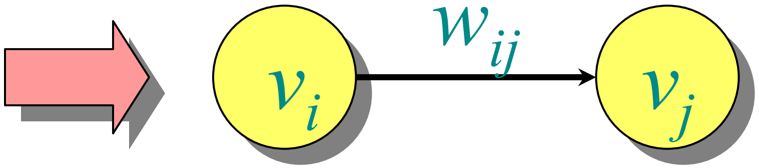
$$\left.\begin{aligned} x_1 - x_2 &\le 3 \\ x_2 - x_3 &\le –2 \\ x_1 - x_3 &\le 2 \end{aligned}\right\} \quad x_j - x_i \le w_{ij}$$

Solution:

$$\begin{aligned} x_1 &= 3 \\ x_2 &= 0 \\ x_3 &= 2 \end{aligned}$$

Constraint graph:

$x_j - x_i \le w_{ij}$



(The “A” matrix has dimensions $|E| \times |V|$.)

Unsatisfiable constraints

Theorem. If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

Proof. Suppose that the negative-weight cycle is $v_1 \to v_2 \to \cdots \to v_k \to v_1$. Then, we have

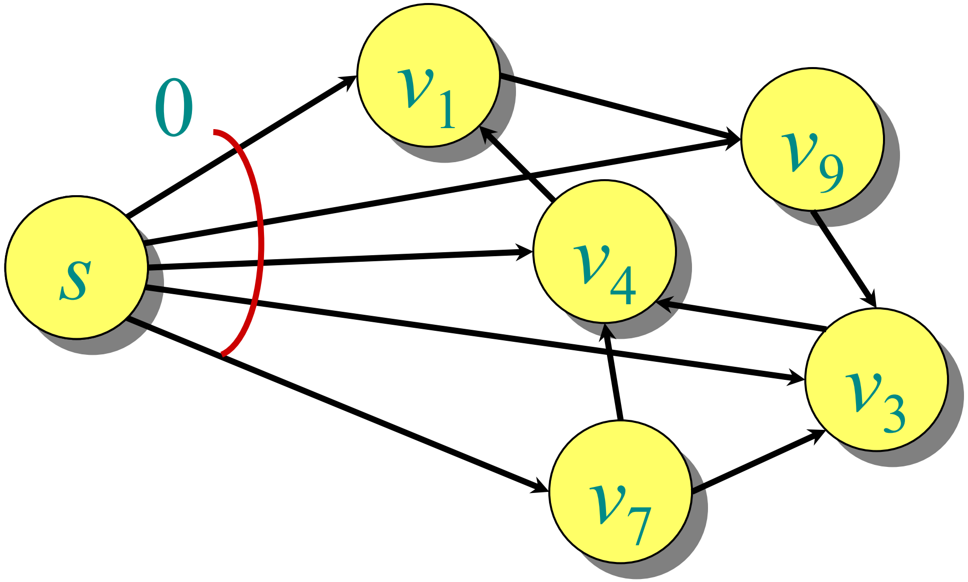
$$
\begin{array}{rcll}
x_2 - & x_1 & \le & w_{12} \\
x_3 - & x_2 & \le & w_{23} \\
 & & \vdots & \\
x_k - & x_{k-1} & \le & w_{k-1,\,k} \\
x_1 - & x_k & \le & w_{k1} \\
\hline
 & 0 & \le & \text{weight of cycle} \\
 & & < & 0
\end{array}
$$

Therefore, no values for the x_i can satisfy the constraints. □

Satisfying the constraints

Theorem. Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

Proof. Add a new vertex s to V with a 0-weight edge to each vertex $v_i \in V$.

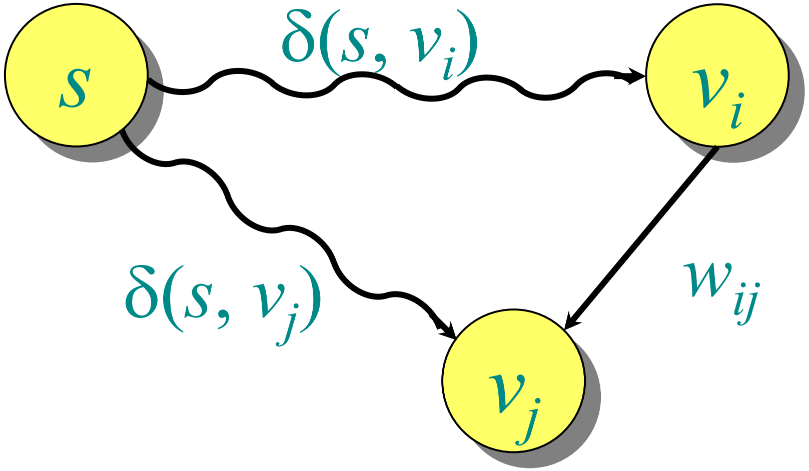


Note:
No negative-weight cycles introduced ⇒ shortest paths exist.

Proof (continued)

Claim: The assignment $x_i = \delta(s, v_i)$ solves the constraints.

Consider any constraint $x_j - x_i \le w_{ij}$, and consider the shortest paths from s to v_j and v_i:



The triangle inequality gives us $\delta(s,v_j) \le \delta(s, v_i) + w_{ij}$. Since $x_i = \delta(s, v_i)$ and $x_j = \delta(s, v_j)$, the constraint $x_j - x_i \le w_{ij}$ is satisfied. □

Bellman-Ford and linear programming

Corollary. The Bellman-Ford algorithm can solve a system of m difference constraints on n variables in $O(mn)$ time. □

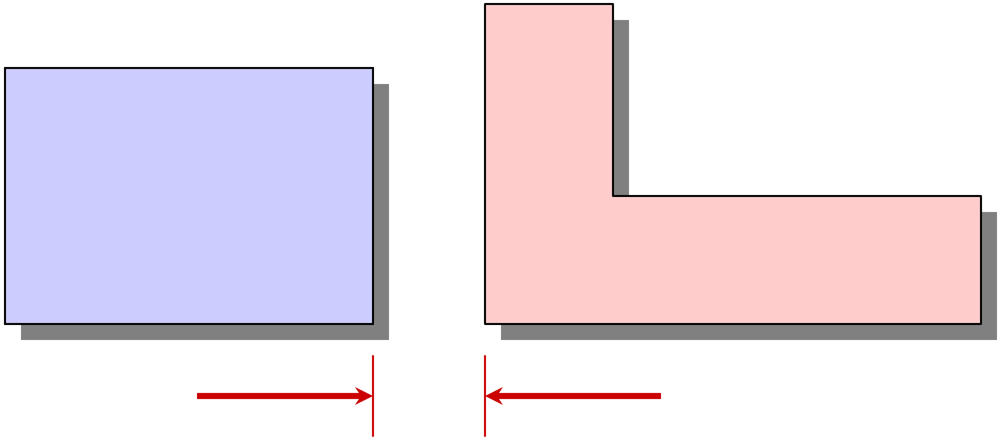
Single-source shortest paths is a simple LP problem.

In fact, Bellman-Ford maximizes $x_1 + x_2 + \cdots + x_n$ subject to the constraints $x_j - x_i \le w_{ij}$ and $x_i \le 0$ (exercise).

Bellman-Ford also minimizes $\max_i\{x_i\} - \min_i\{x_i\}$ (exercise).

Application to VLSI layout compaction

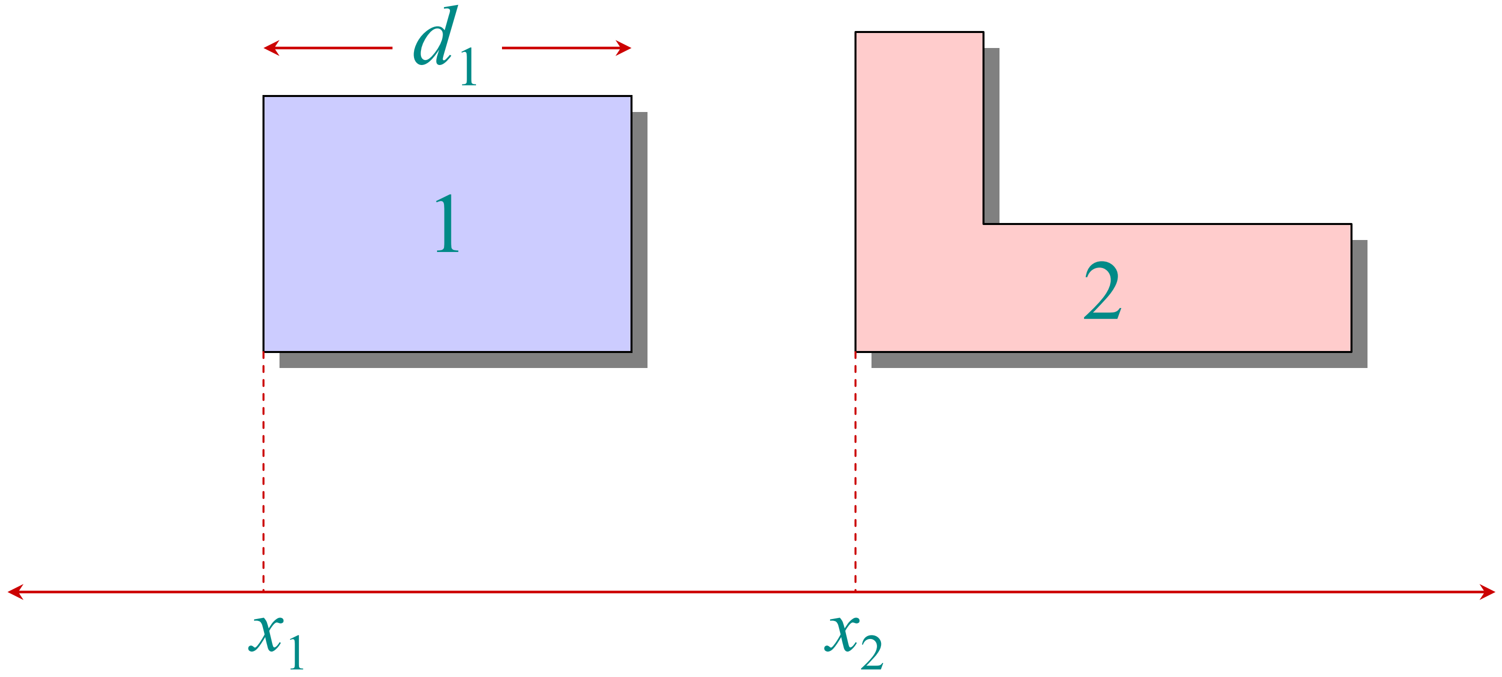
Integrated -circuit features:



minimum separation λ

Problem: Compact (in one dimension) the space between the features of a VLSI layout without bringing any features too close together.

VLSI layout compaction



Constraint: $x_2 - x_1 \geq d_1 + \lambda$

Bellman-Ford minimizes $\max_i\{x_i\} - \min_i\{x_i\}$, which compacts the layout in the x-dimension.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 19

Prof. Erik Demaine

Shortest paths

Single-source shortest paths

- Nonnegative edge weights
  - Dijkstra’s algorithm: $O(E + V \lg V)$
- General
  - Bellman-Ford: $O(VE)$
- DAG
  - One pass of Bellman-Ford: $O(V + E)$

All-pairs shortest paths

- Nonnegative edge weights
  - Dijkstra’s algorithm $|V|$ times: $O(VE + V^2 \lg V)$
- General
  - Three algorithms today.

All-pairs shortest paths

Input: Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to \mathbb{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

IDEA #1:

- Run Bellman-Ford once from each vertex.
- Time $= \mathrm{O}(V^2E)$.
- Dense graph ⇒ $\mathrm{O}(V^4)$ time.

Good first try!

Dynamic programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define

$d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges.

Claim: We have

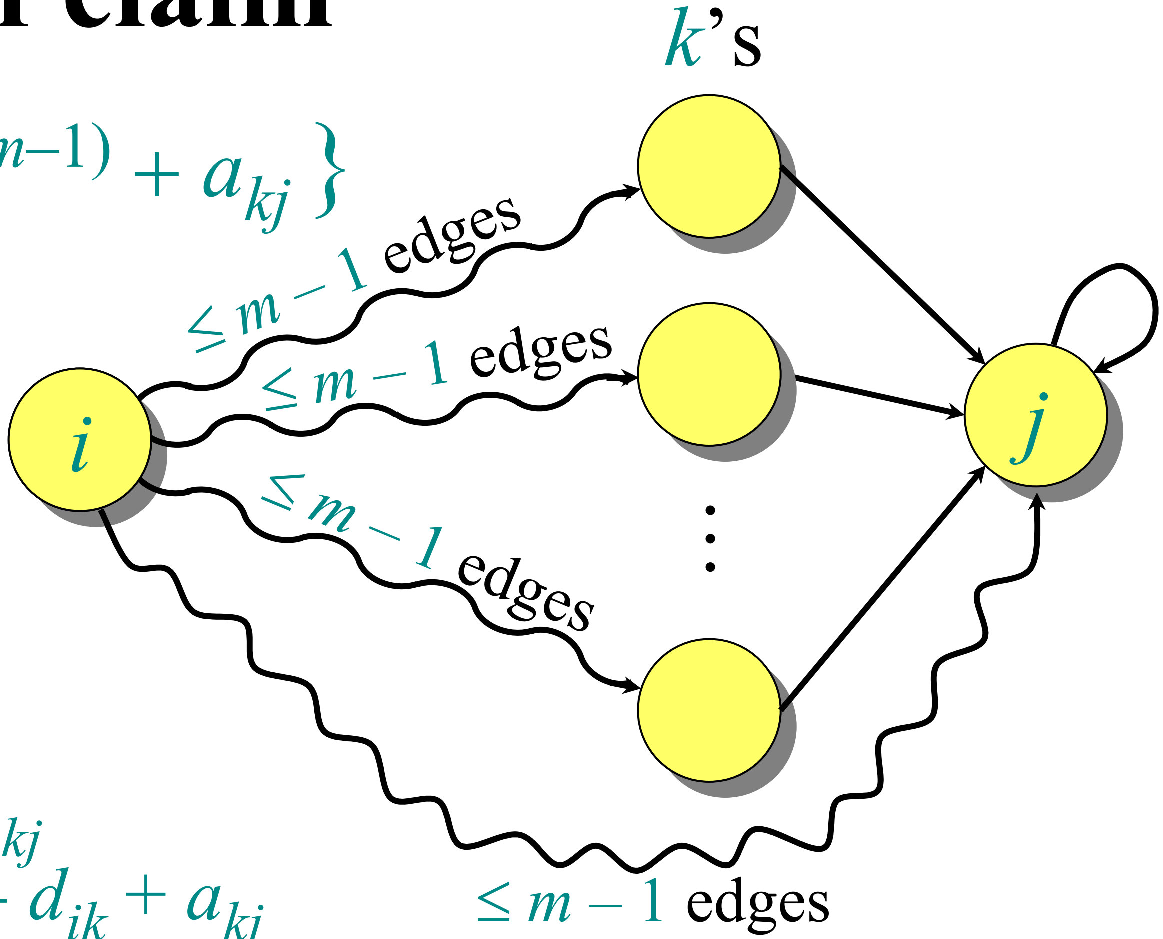
$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

and for $m = 1, 2, \ldots, n - 1$,

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$

Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$



Relaxation!

```
for $k \leftarrow 1$ to $n$
    do if $d_{ij} > d_{ik} + a_{kj}$
        then $d_{ij} \leftarrow d_{ik} + a_{kj}$
```

Note: No negative-weight cycles implies

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$$

Matrix multiplication

Compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Time $= \Theta(n^3)$ using the standard algorithm.

What if we map "$+$" → "$\min$" and "$\cdot$" → "$+$"?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\} .$$

Thus, $D^{(m)} = D^{(m-1)}$ "$\times$" A.

Identity matrix $= \mathrm{I} = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^0 = (d_{ij}^{(0)})$.

Matrix multiplication (continued)

The (min, +) multiplication is ***associative***, and with the real numbers, it forms an algebraic structure called a ***closed semiring***.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot A &&= A^1 \\ D^{(2)} &= D^{(1)} \cdot A &&= A^2 \\ &\vdots && \vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot A &&= A^{n-1}, \end{aligned}$$

yielding $D^{(n-1)} = (\delta(i, j))$.

Time $= \Theta(n \cdot n^3) = \Theta(n^4)$. No better than $n \times$ B-F.

Improved matrix multiplication algorithm

Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $\underbrace{A^2, A^4, \ldots, A^{2^{\lceil \lg(n-1) \rceil}}}_{O(\lg n) \text{ squarings}}$.

Note: $A^{n-1} = A^n = A^{n+1} = \cdots$.

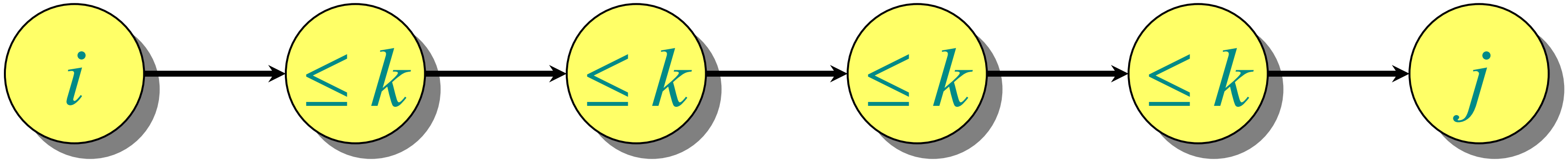
Time $= \Theta(n^3 \lg n)$.

To detect negative-weight cycles, check the diagonal for negative values in $O(n)$ additional time.

Floyd-Warshall algorithm

Also dynamic programming, but faster!

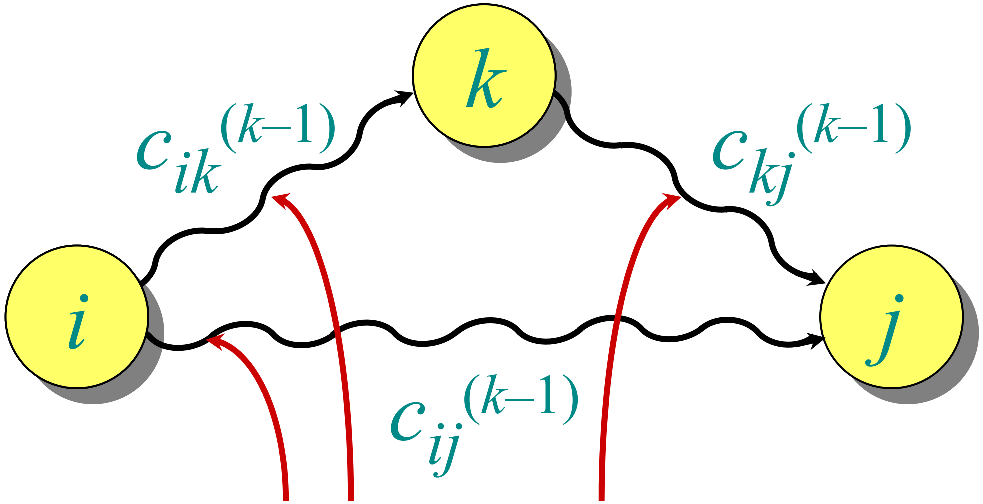
Define $c_{ij}^{(k)} =$ weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.



Thus, $\delta(i, j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

Floyd-Warshall recurrence

$$c_{ij}^{(k)} = \min_k \{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$



intermediate vertices in $\{1, 2, \ldots, k\}$

Pseudocode for Floyd-Warshall

```
for $k \leftarrow 1$ to $n$
    do for $i \leftarrow 1$ to $n$
        do for $j \leftarrow 1$ to $n$
            do if $c_{ij} > c_{ik} + c_{kj}$
                then $c_{ij} \leftarrow c_{ik} + c_{kj}$
```

relaxation

Notes:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in $\Theta(n^3)$ time.
- Simple to code.
- Efficient in practice.

Transitive closure of a directed graph

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Time = $\Theta(n^3)$.

Graph reweighting

Theorem. Given a label $h(v)$ for each $v \in V$, ***reweight*** each edge $(u, v) \in E$ by

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v).$$

Then, all paths between the same two vertices are reweighted by the same amount.

Proof. Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path in the graph.

Then, we have
$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k-1} \hat{w}(v_i, v_{i+1}) \\
&= \sum_{i=1}^{k-1} \big(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})\big) \\
&= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k) \\
&= w(p) + h(v_1) - h(v_k).
\end{aligned}$$ □

Johnson's algorithm

1. Find a vertex labeling h such that $\hat{w}(u, v) \geq 0$ for all $(u, v) \in E$ by using Bellman-Ford to solve the difference constraints
$$h(v) - h(u) \leq w(u, v),$$
or determine that a negative-weight cycle exists.
    - Time $= O(VE)$.
2. Run Dijkstra's algorithm from each vertex using $\hat{w}$.
    - Time $= O(VE + V^2 \lg V)$.
3. Reweight each shortest-path length $\hat{w}(p)$ to produce the shortest-path lengths $w(p)$ of the original graph.
    - Time $= O(V^2)$.

Total time $= O(VE + V^2 \lg V)$.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 20

Prof. Erik Demaine

Disjoint-set data structure (Union-Find)

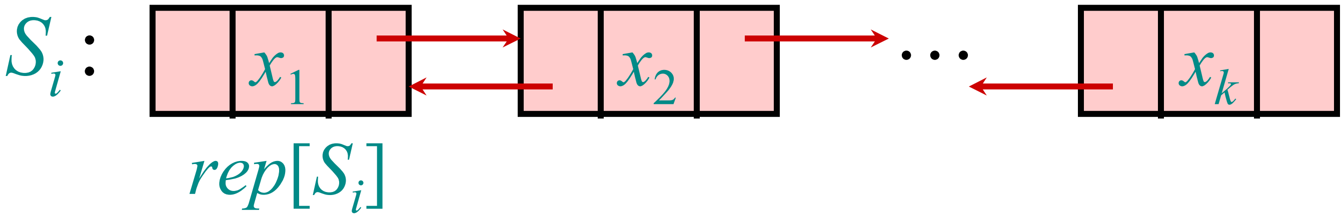
Problem: Maintain a dynamic collection of *pairwise-disjoint* sets $\mathsf{S} = \{S_1, S_2, \ldots, S_r\}$. Each set S_i has one element distinguished as the representative element, $rep[S_\mathrm{i}]$.

Must support 3 operations:

- MAKE-SET(x): adds new set $\{x\}$ to S with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i).
- UNION(x, y): replaces sets S_x, S_y with $S_x \cup S_y$ in S for any x, y in distinct sets S_x, S_y.
- FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x.

Simple linked-list solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an (unordered) doubly linked list. Define representative element $rep[S_i]$ to be the front of the list, x_1.



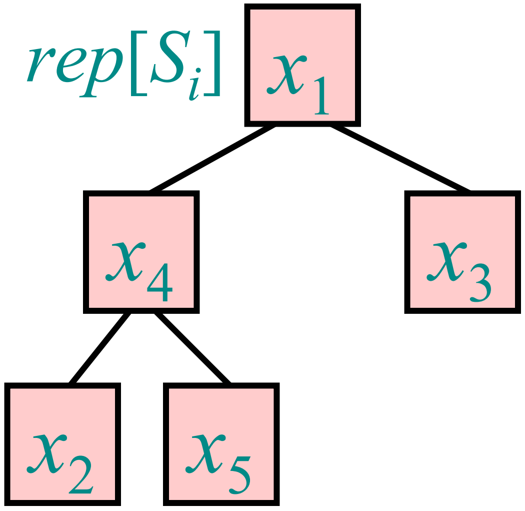
- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks left in the list containing x until it reaches the front of the list. – $\Theta(n)$
- UNION(x, y) concatenates the lists containing x and y, leaving rep. as FIND-SET$[x]$. – $\Theta(n)$

Simple balanced-tree solution

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as a balanced tree (ignoring keys). Define representative element $rep[S_i]$ to be the root of the tree.

- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root. – $\Theta(\lg n)$
- UNION(x, y) concatenates the trees containing x and y, changing rep. – $\Theta(\lg n)$

$S_i = \{x_1, x_2, x_3, x_4, x_5\}$



Plan of attack

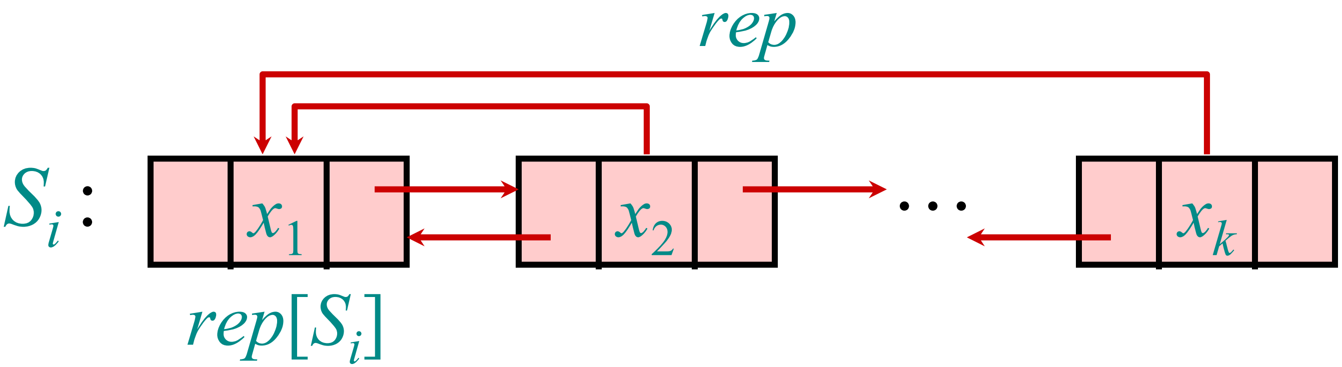
We will build a simple disjoint-union data structure that, in an amortized sense, performs significantly better than $\Theta(\lg n)$ per op., even better than $\Theta(\lg \lg n)$, $\Theta(\lg \lg \lg n)$, etc., but not quite $\Theta(1)$.

To reach this goal, we will introduce two key ***tricks***. Each trick converts a trivial $\Theta(n)$ solution into a simple $\Theta(\lg n)$ amortized solution. Together, the two tricks yield a much better solution.

First trick arises in an augmented linked list. Second trick arises in a tree structure.

Augmented linked-list solution

Store set $S_i = \{x_1, x_2, \ldots, x_k\}$ as unordered doubly linked list. Define $rep[S_i]$ to be front of list, x_1. Each element x_j also stores pointer $rep[x_j]$ to $rep[S_i]$.



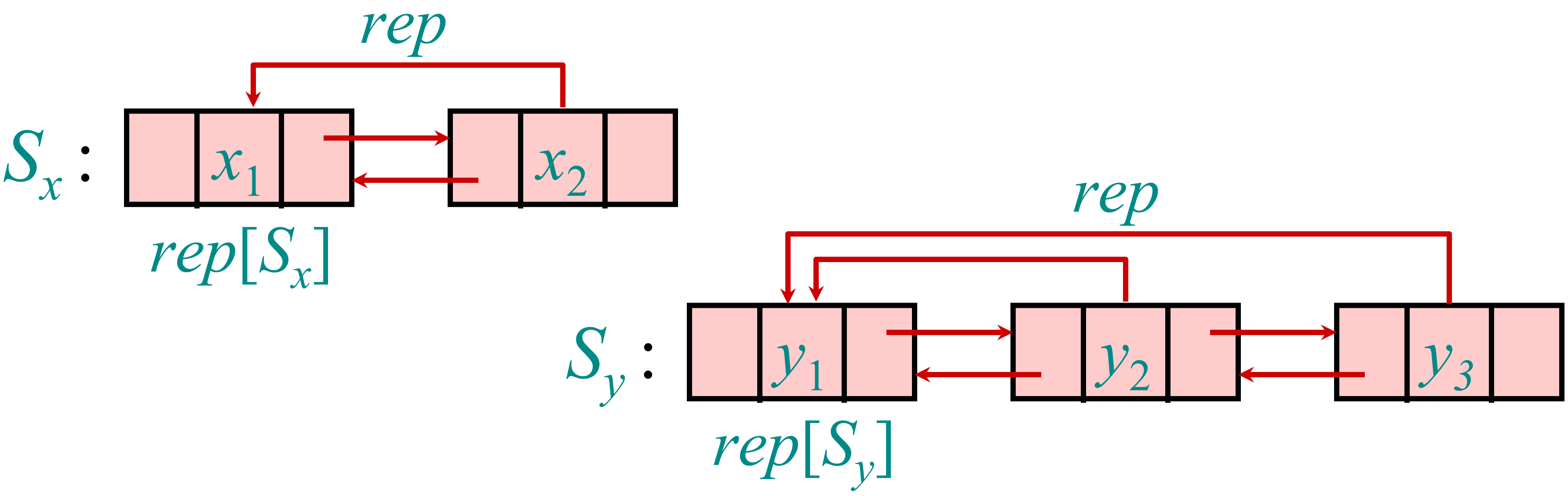
- FIND-SET(x) returns $rep[x]$. – $\Theta(1)$
- UNION(x, y) concatenates the lists containing x and y, and updates the *rep* pointers for all elements in the list containing y. – $\Theta(n)$

Example of augmented linked-list solution

Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

UNION(x, y)

- concatenates the lists containing x and y, and
- updates the *rep* pointers for all elements in the list containing y.



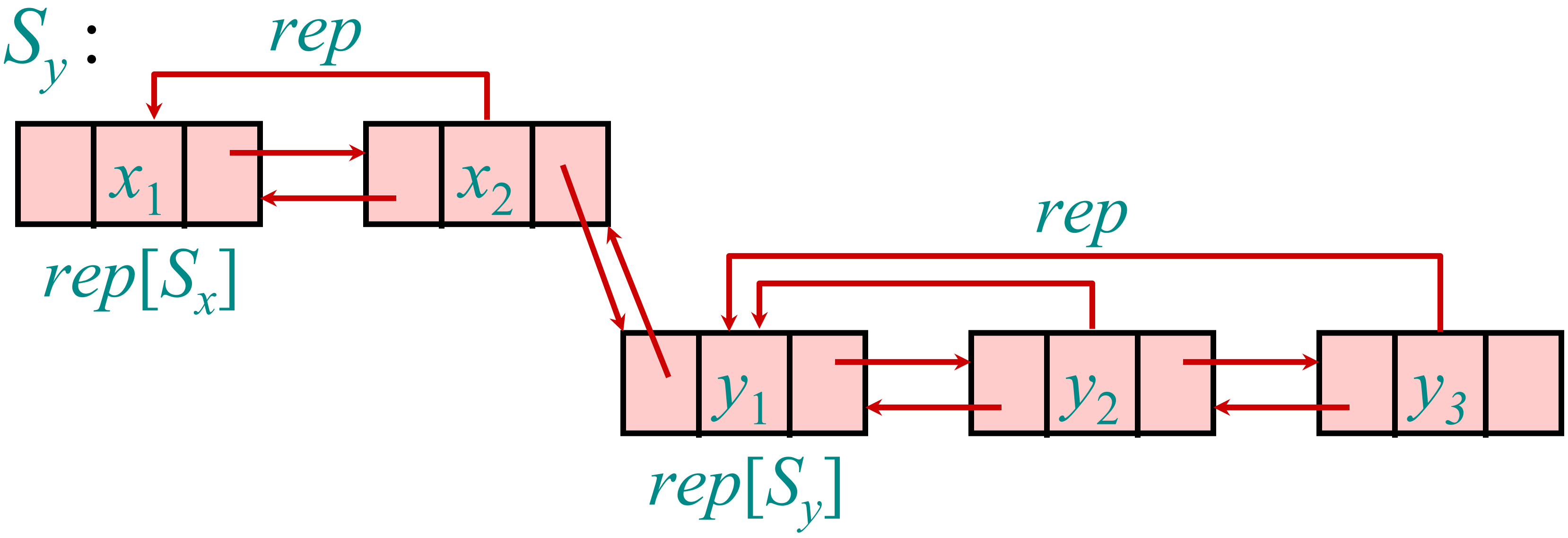
Example of augmented linked-list solution

Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

UNION(x, y)

- concatenates the lists containing x and y, and
- updates the *rep* pointers for all elements in the list containing y.

$S_x \cup S_y$:

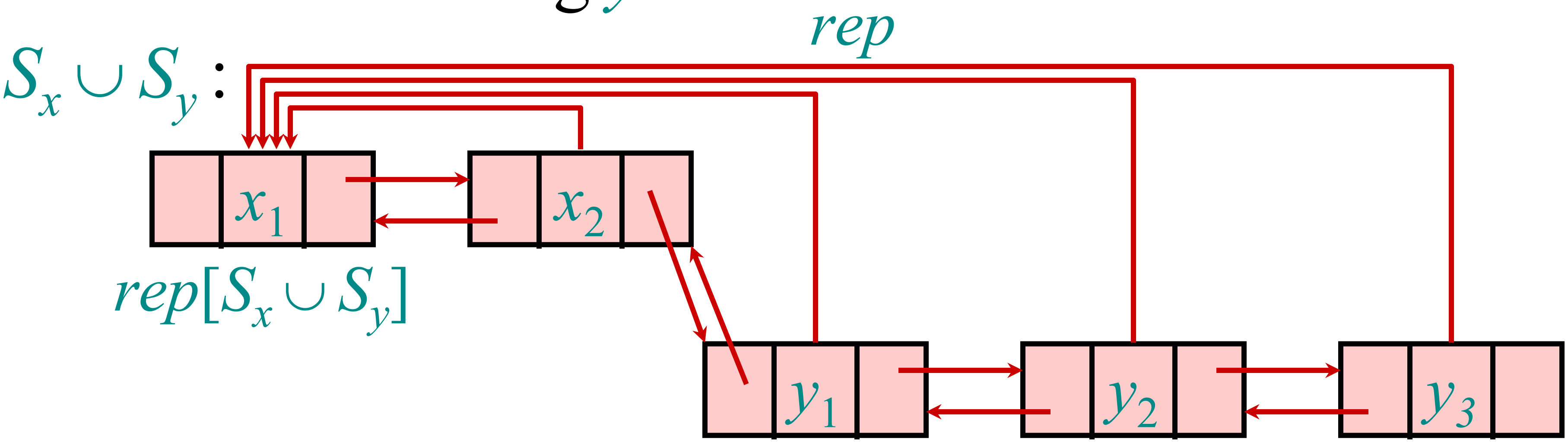


Example of augmented linked-list solution

Each element x_j stores pointer $rep[x_j]$ to $rep[S_i]$.

UNION(x, y)

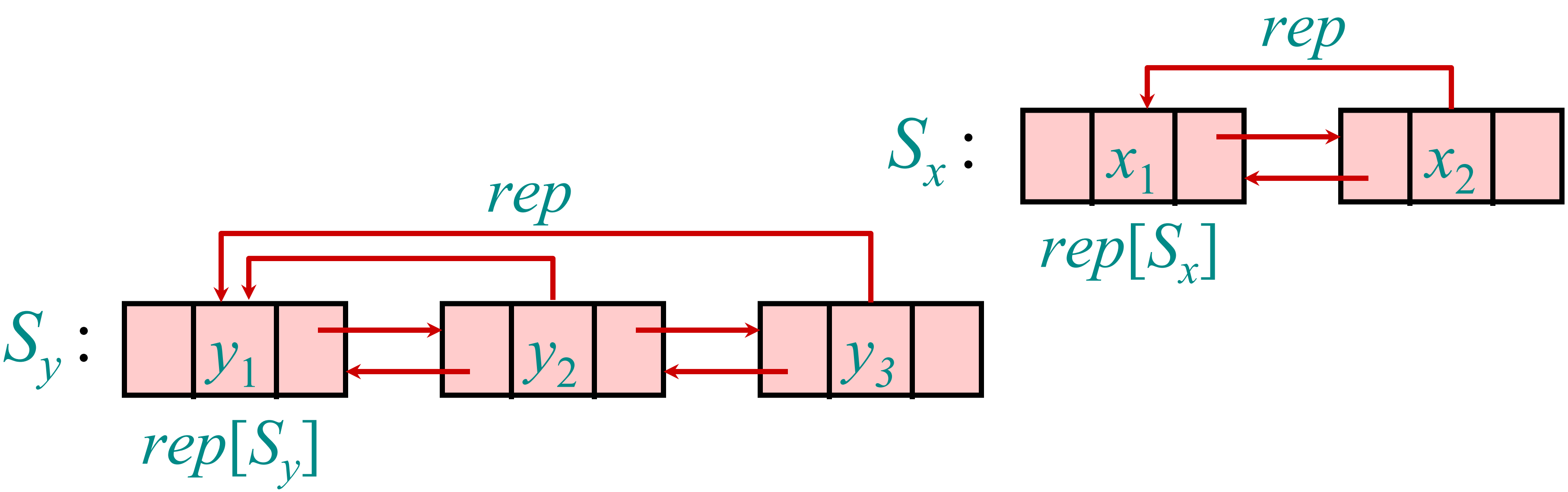
- concatenates the lists containing x and y, and
- updates the rep pointers for all elements in the list containing y.



Alternative concatenation

UNION(x, y) could instead

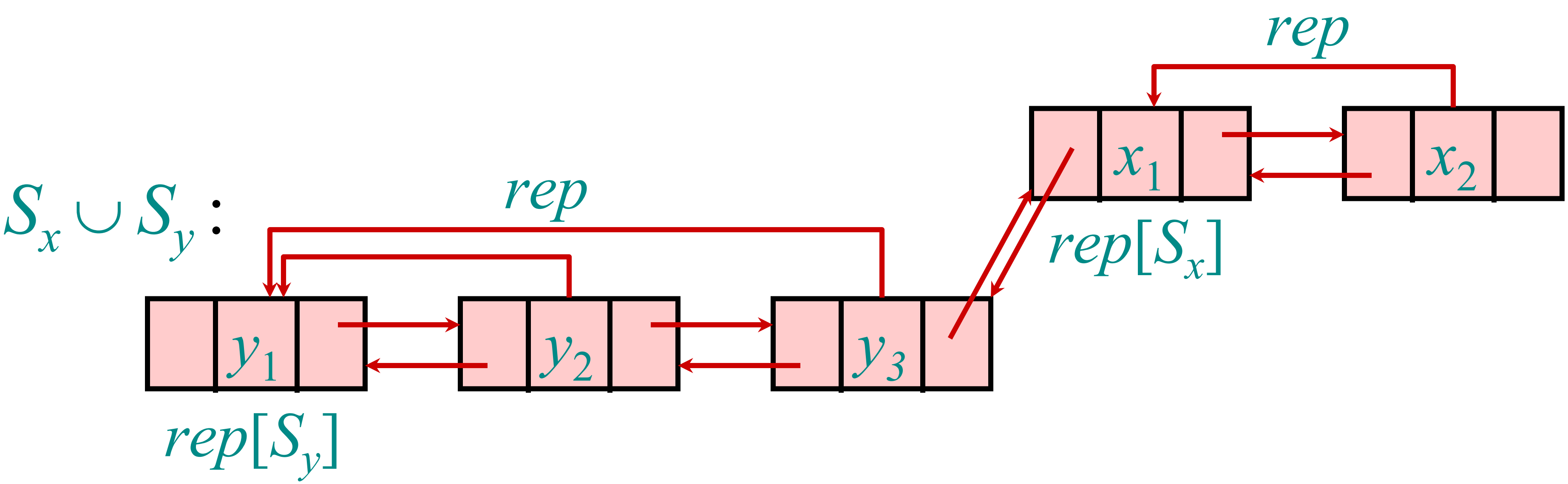
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Alternative concatenation

UNION(x, y) could instead

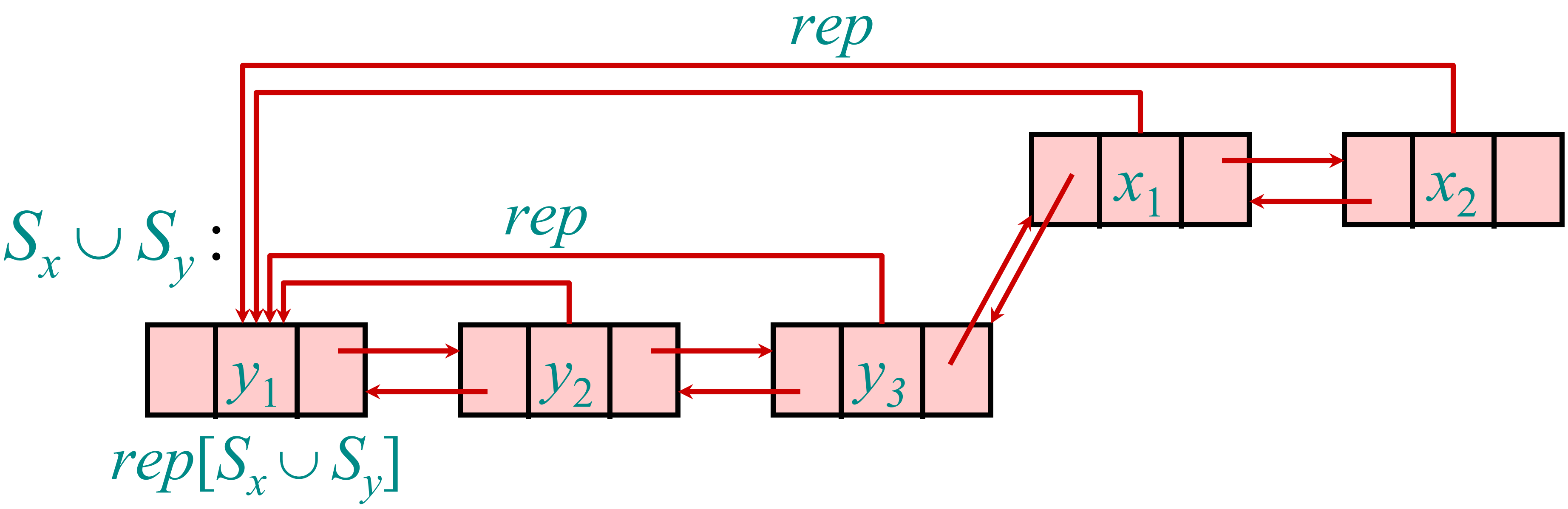
- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Alternative concatenation

UNION(x, y) could instead

- concatenate the lists containing y and x, and
- update the *rep* pointers for all elements in the list containing x.



Trick 1: Smaller into larger

To save work, concatenate smaller list onto the end of the larger list. Cost = Θ(length of smaller list). Augment list to store its ***weight*** (# elements).

Let n denote the overall number of elements (equivalently, the number of MAKE-SET operations).
Let m denote the total number of operations.
Let f denote the number of FIND-SET operations.

Theorem: Cost of all UNION's is $O(n \lg n)$.

Corollary: Total cost is $O(m + n \lg n)$.

Analysis of Trick 1

To save work, concatenate smaller list onto the end of the larger list. Cost $= \Theta(1 +$ length of smaller list$)$.

Theorem: Total cost of UNION's is $O(n \lg n)$.

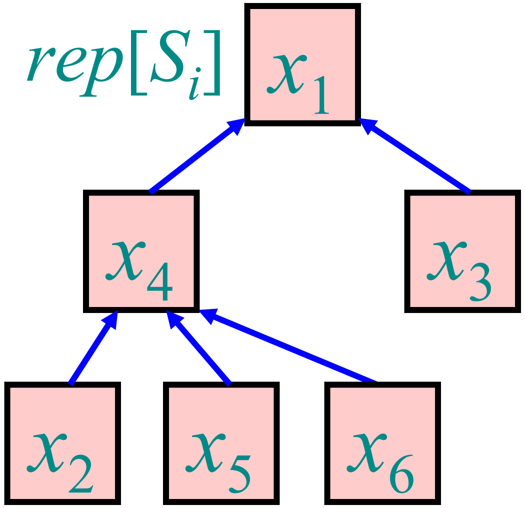
Proof. Monitor an element x and set S_x containing it. After initial MAKE-SET(x), $weight[S_x] = 1$. Each time S_x is united with set S_y, $weight[S_y] \geq weight[S_x]$, pay 1 to update $rep[x]$, and $weight[S_x]$ at least doubles (increasing by $weight[S_y]$). Each time S_y is united with smaller set S_y, pay nothing, and $weight[S_x]$ only increases. Thus pay $\leq \lg n$ for x. $\square$

Representing sets as trees

Store each set $S_i = \{x_1, x_2, \ldots, x_k\}$ as an unordered, potentially unbalanced, not necessarily binary tree, storing only *parent* pointers. $rep[S_i]$ is the tree root.

- MAKE-SET(x) initializes x as a lone node. – $\Theta(1)$
- FIND-SET(x) walks up the tree containing x until it reaches the root. – $\Theta(depth[x])$
- UNION(x, y) concatenates the trees containing x and y…

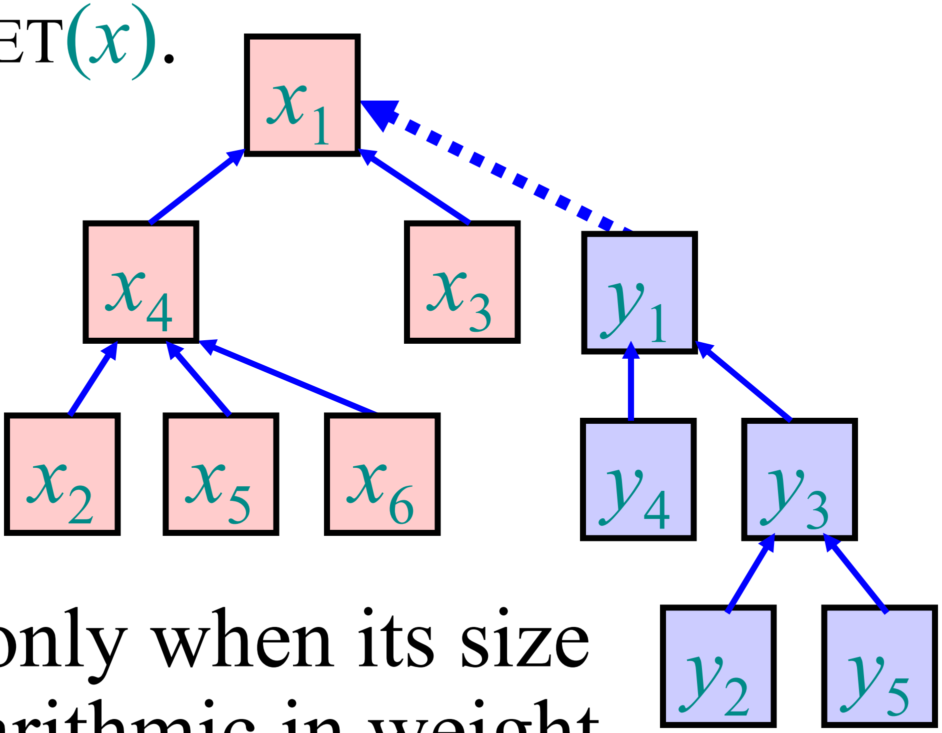
$S_i = \{x_1, x_2, x_3, x_4, x_5, x_6\}$



Trick 1 adapted to trees

UNION(x, y) can use a simple concatenation strategy: Make root FIND-SET(y) a child of root FIND-SET(x).
$\Rightarrow$ FIND-SET(y) = FIND-SET(x).

We can adapt Trick 1 to this context also: Merge tree with smaller weight into tree with larger weight.



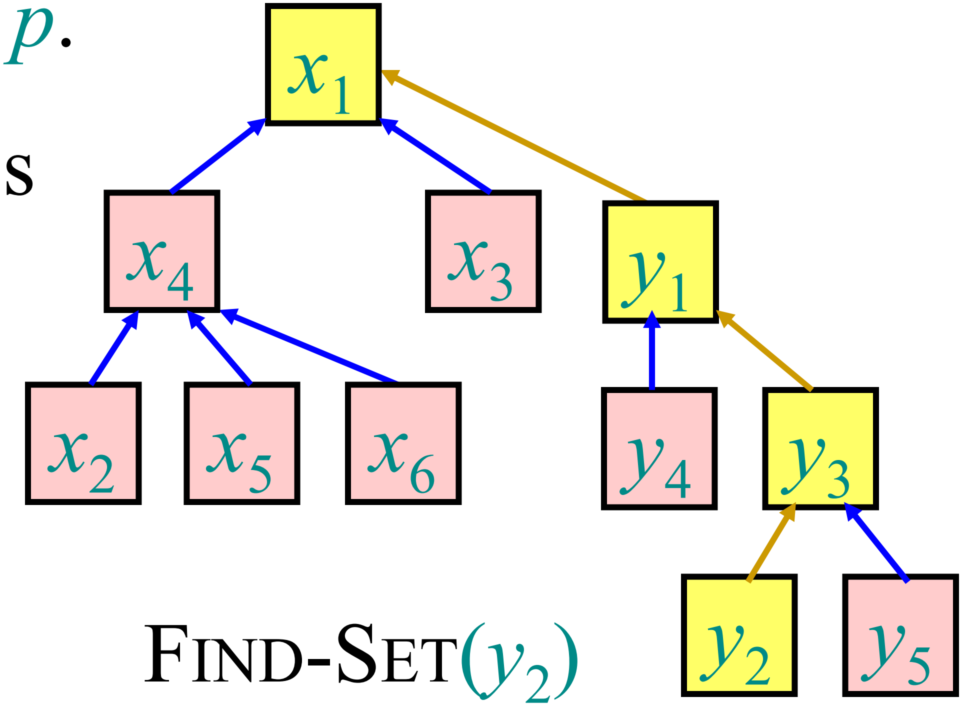
Height of tree increases only when its size doubles, so height is logarithmic in weight.
Thus total cost is $O(m + f \lg n)$.

Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path p to the root, we know the representative for all the nodes on path p.

Path compression makes all of those nodes direct children of the root.

Cost of FIND-SET(x) is still $\Theta(depth[x])$.



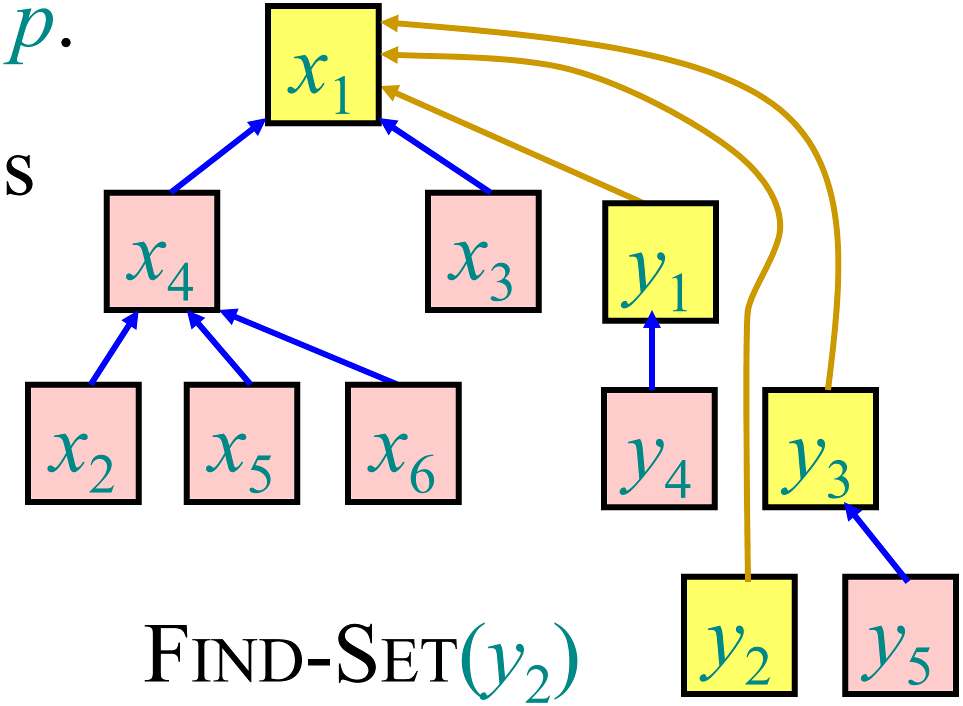
FIND-SET(y_2)

Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path p to the root, we know the representative for all the nodes on path p.

Path compression makes all of those nodes direct children of the root.

Cost of FIND-SET(x) is still $\Theta(depth[x])$.



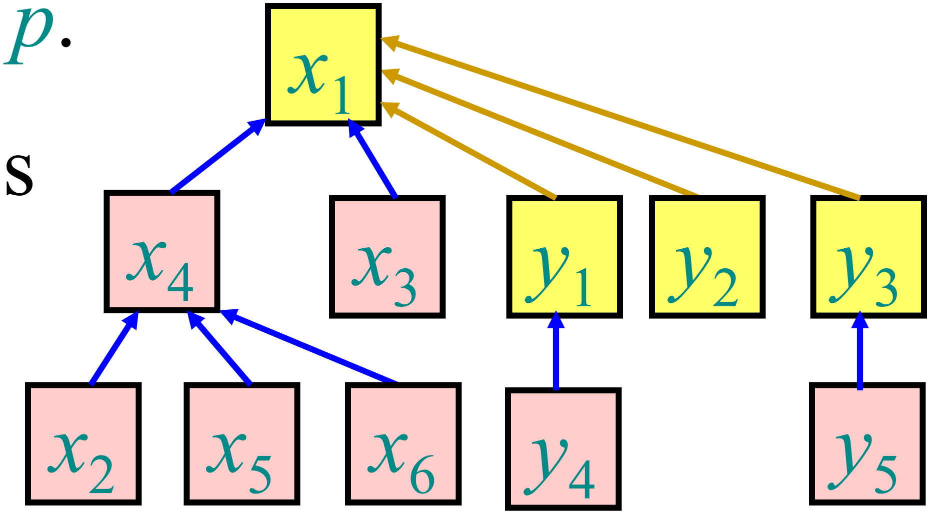
FIND-SET(y_2)

Trick 2: Path compression

When we execute a FIND-SET operation and walk up a path p to the root, we know the representative for all the nodes on path p.

Path compression makes all of those nodes direct children of the root.

Cost of FIND-SET(x) is still $\Theta(depth[x])$.



FIND-SET(y_2)

Analysis of Trick 2 alone

Theorem: Total cost of Find-Set's is $O(m \lg n)$.

Proof: Amortization by potential function.
The ***weight*** of a node x is # nodes in its subtree.
Define $\phi(x_1, \ldots, x_n) = \Sigma_i \lg \mathit{weight}[x_i]$.
Union(x_i, x_j) increases potential of root Find-Set(x_i) by at most $\lg \mathit{weight}[\text{root Find-Set}(x_j)] \le \lg n$.
Each step down $p \to c$ made by Find-Set(x_i), except the first, moves c's subtree out of p's subtree. Thus if $\mathit{weight}[c] \ge ½\, \mathit{weight}[p]$, ϕ decreases by ≥ 1, paying for the step down. There can be at most $\lg n$ steps $p \to c$ for which $\mathit{weight}[c] < ½\, \mathit{weight}[p]$. □

Analysis of Trick 2 alone

Theorem: If all UNION operations occur before all FIND-SET operations, then total cost is $O(m)$.

Proof: If a FIND-SET operation traverses a path with k nodes, costing $O(k)$ time, then $k - 2$ nodes are made new children of the root. This change can happen only once for each of the n elements, so the total cost of FIND-SET is $O(f + n)$. □

Ackermann's function A

Define $A_k(j) = \begin{cases} j+1 & \text{if } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$ – iterate $j+1$ times

$A_0(j) = j + 1$ $\qquad$ $A_0(1) = 2$

$A_1(j) \sim 2j$ $\qquad$ $A_1(1) = 3$

$A_2(j) \sim 2j\, 2^j > 2^j$ $\qquad$ $A_2(1) = 7$

$A_3(1) = 2047$

$A_3(j) > \left. 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^j}}}}}} \right\} j$

$A_4(j)$ is a lot bigger. $\qquad$ $A_4(1) > \left. 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2047}}}}}}} \right\} 2048$

Define $\alpha(n) = \min\ \{k : A_k(1) \geq n\} \leq 4$ for practical n.

Analysis of Tricks 1 + 2

Theorem: In general, total cost is $O(m\,\alpha(n))$.

(long, tricky proof – see Section 21.4 of CLRS)

Application: Dynamic connectivity

Suppose a graph is given to us ***incrementally*** by

- ADD-VERTEX(v)
- ADD-EDGE(u, v)

and we want to support ***connectivity*** queries:

- CONNECTED(u, v): Are u and v in the same connected component?

For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

Application: Dynamic connectivity

Sets of vertices represent *connected components*.
Suppose a graph is given to us ***incrementally*** by

- ADD-VERTEX(v) – MAKE-SET(v)
- ADD-EDGE(u, v) – **if** not CONNECTED(u, v)
  then UNION(v, w)

and we want to support ***connectivity*** queries:

- CONNECTED(u, v): – FIND-SET(u) = FIND-SET(v)
  Are u and v in the same connected component?

For example, we want to maintain a spanning forest, so we check whether each new edge connects a previously disconnected pair of vertices.

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 21

Prof. Charles E. Leiserson

Take-home quiz

No notes (except this one).

Introduction to Algorithms

6.046J/18.401J/SMA5503

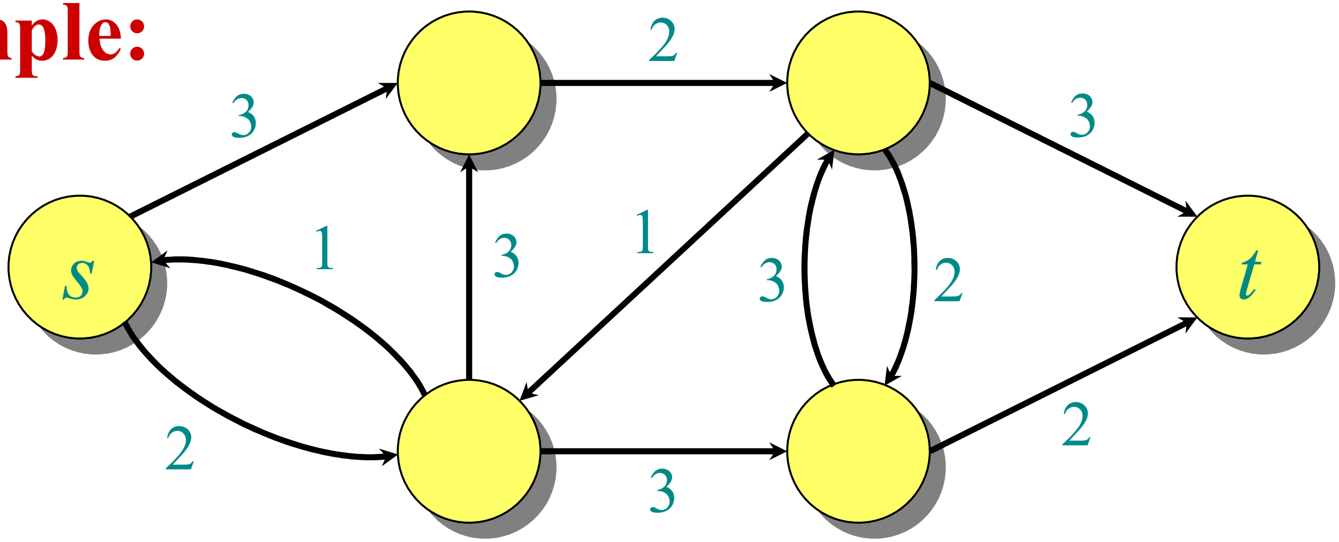
Lecture 22

Prof. Charles E. Leiserson

Flow networks

Definition. A ***flow network*** is a directed graph $G = (V, E)$ with two distinguished vertices: a ***source*** s and a ***sink*** t. Each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$. If $(u, v) \notin E$, then $c(u, v) = 0$.

Example:



Flow networks

Definition. A ***positive flow*** on G is a function $p : V \times V \rightarrow \mathbb{R}$ satisfying the following:

- ***Capacity constraint:*** For all $u, v \in V$,

$$0 \le p(u, v) \le c(u, v).$$

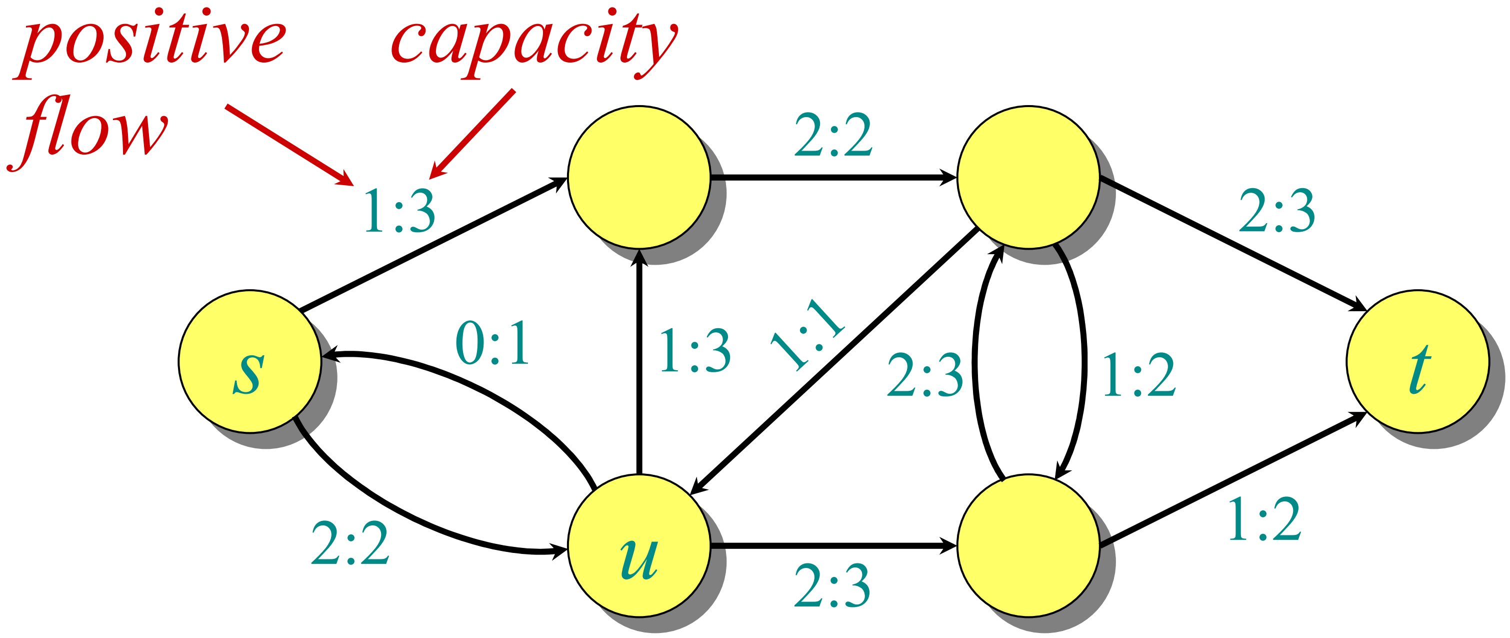
- ***Flow conservation:*** For all $u \in V - \{s, t\}$,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = 0.$$

The ***value*** of a flow is the net flow out of the source:

$$\sum_{v \in V} p(s,v) - \sum_{v \in V} p(v,s).$$

A flow on a network



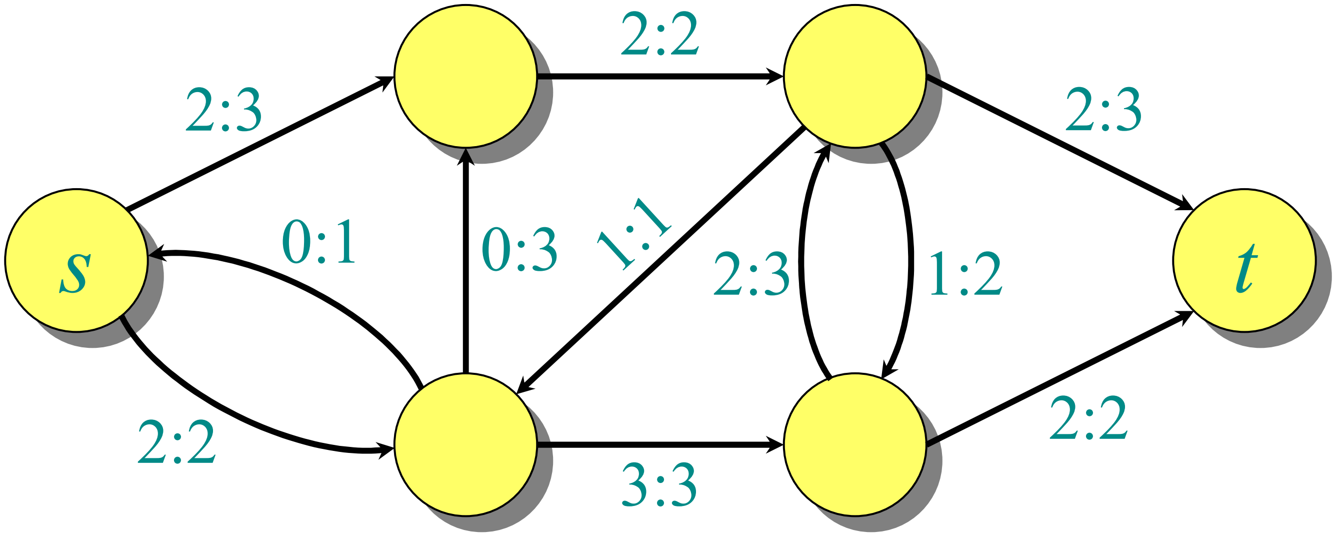
Flow conservation (like Kirchoff's current law):

- Flow into u is $2 + 1 = 3$.
- Flow out of u is $0 + 1 + 2 = 3$.

The value of this flow is $1 - 0 + 2 = 3$.

The maximum-flow problem

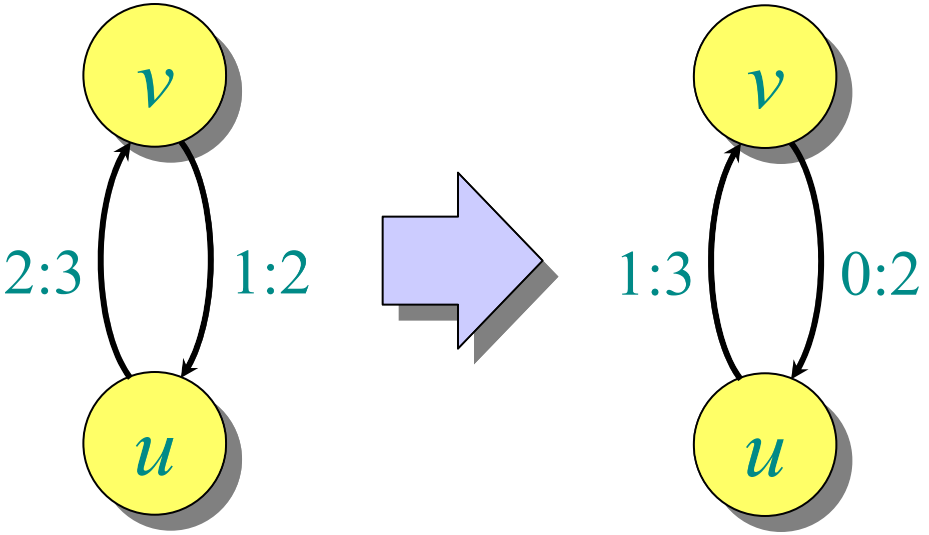
Maximum-flow problem: Given a flow network *G*, find a flow of maximum value on *G*.



The value of the maximum flow is 4.

Flow cancellation

Without loss of generality, positive flow goes either from *u* to *v*, or from *v* to *u*, but not both.



Net flow from *u* to *v* in both cases is 1.

The capacity constraint and flow conservation are preserved by this transformation.

INTUITION: View flow as a *rate*, not a *quantity*.

A notational simplification

IDEA: Work with the net flow between two vertices, rather than with the positive flow.

Definition. A ***(net) flow*** on G is a function $f : V \times V \to \mathbb{R}$ satisfying the following:

- ***Capacity constraint:*** For all $u, v \in V$,

$$f(u, v) \le c(u, v).$$

- ***Flow conservation:*** For all $u \in V - \{s, t\}$,

$$\sum_{v \in V} f(u,v) = 0. \longleftarrow$$

One summation instead of two.

- ***Skew symmetry:*** For all $u, v \in V$,

$$f(u, v) = -f(v, u).$$

Equivalence of definitions

Theorem. The two definitions are equivalent.

Proof. (⇒) Let $f(u, v) = p(u, v) - p(v, u)$.

- ***Capacity constraint:*** Since $p(u, v) \le c(u, v)$ and $p(v, u) \ge 0$, we have $f(u, v) \le c(u, v)$.
- ***Flow conservation:***

$$\begin{aligned}\sum_{v\in V} f(u,v) &= \sum_{v\in V}\bigl(p(u,v) - p(v,u)\bigr) \\ &= \sum_{v\in V} p(u,v) - \sum_{v\in V} p(v,u)\end{aligned}$$

- ***Skew symmetry:***

$$\begin{aligned}f(u, v) &= p(u, v) - p(v, u) \\ &= -\,(p(v, u) - p(u, v)) \\ &= -f(v, u).\end{aligned}$$

Proof (continued)

($\Leftarrow$) Let

$$p(u, v) = \begin{cases} f(u, v) & \text{if } f(u, v) > 0, \\ 0 & \text{if } f(u, v) \le 0. \end{cases}$$

- ***Capacity constraint:*** By definition, $p(u, v) \ge 0$. Since $f(u, v) \le c(u, v)$, it follows that $p(u, v) \le c(u, v)$.
- ***Flow conservation:*** If $f(u, v) > 0$, then $p(u, v) - p(v, u) = f(u, v)$. If $f(u, v) \le 0$, then $p(u, v) - p(v, u) = -f(v, u) = f(u, v)$ by skew symmetry. Therefore,

$$\sum_{v \in V} p(u,v) - \sum_{v \in V} p(v,u) = \sum_{v \in V} f(u,v). \quad \square$$

Notation

Definition. The ***value*** of a flow f, denoted by $|f|$, is given by

$$|f| = \sum_{v \in V} f(s, v) \\ = f(s, V) .$$

Implicit summation notation: A set used in an arithmetic formula represents a sum over the elements of the set.

- **Example** — flow conservation: $f(u, V) = 0$ for all $u \in V - \{s, t\}$.

Simple properties of flow

Lemma.

- $f(X, X) = 0$,
- $f(X, Y) = -f(Y, X)$,
- $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ if $X \cap Y = \varnothing$. □

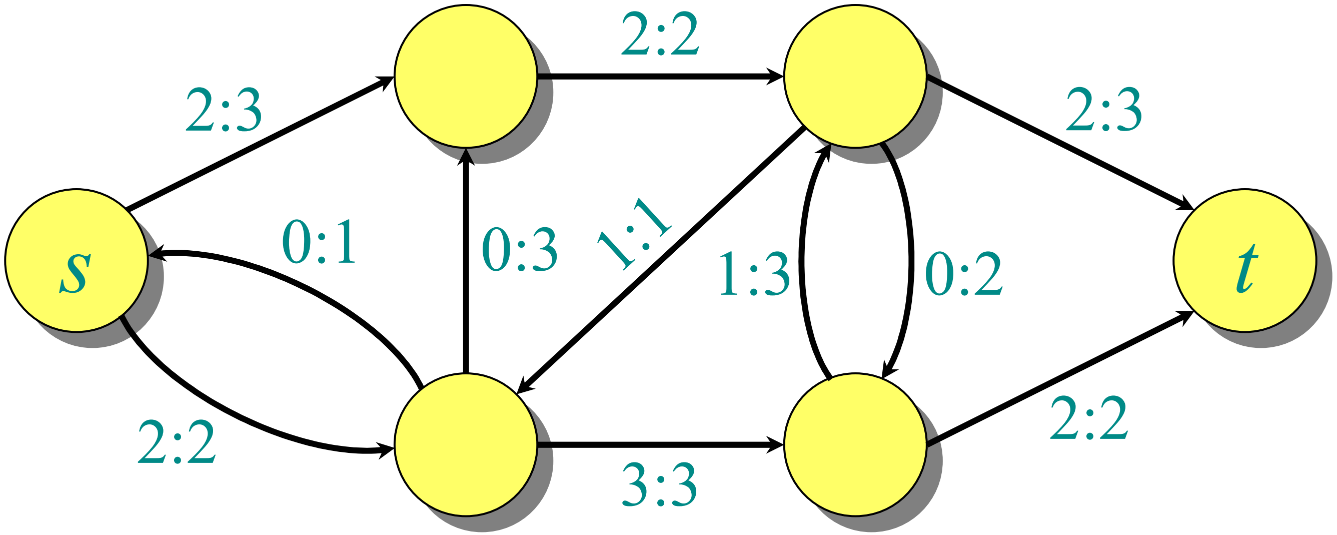
Theorem. $|f| = f(V, t)$.

Proof.

$$\begin{aligned} |f| &= f(s, V) \\ &= f(V, V) - f(V-s, V) \\ &= f(V, V-s) \\ &= f(V, t) + f(V, V-s-t) \\ &= f(V, t). \end{aligned}$$ □

Omit braces.

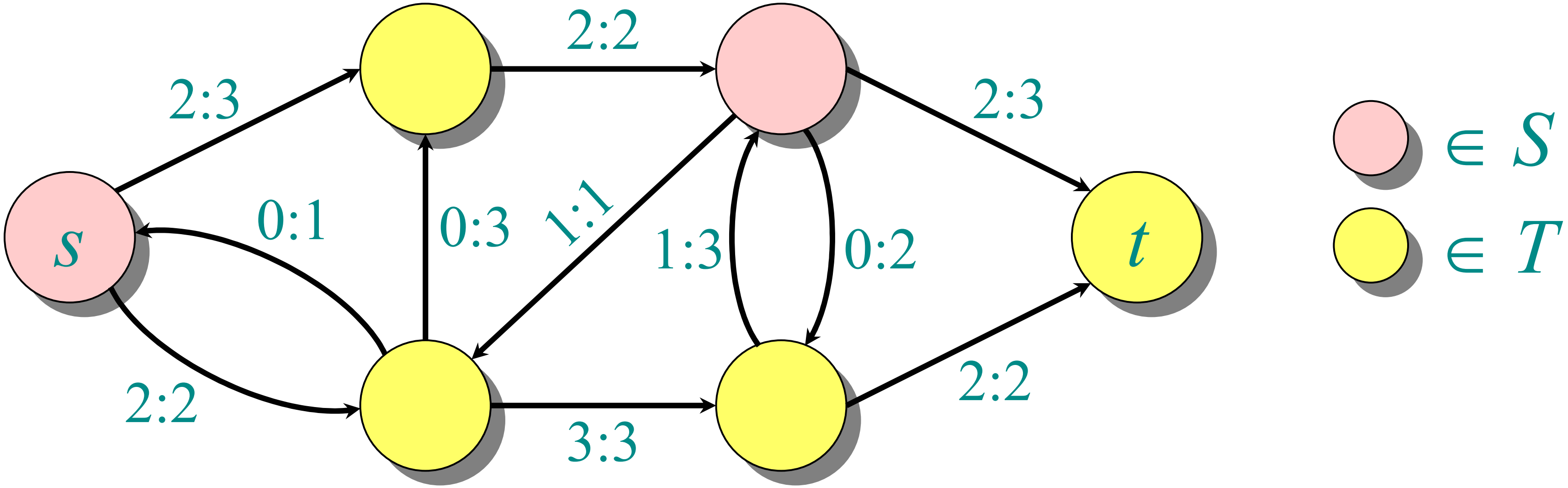
Flow into the sink



$|f| = f(s, V) = 4$ $\qquad$ $f(V, t) = 4$

Cuts

Definition. A ***cut*** (S, T) of a flow network $G = (V, E)$ is a partition of V such that $s \in S$ and $t \in T$. If f is a flow on G, then the ***flow across the cut*** is $f(S, T)$.



$$f(S, T) = (2 + 2) + (-2 + 1 - 1 + 2)$$
$$= 4$$

Another characterization of flow value

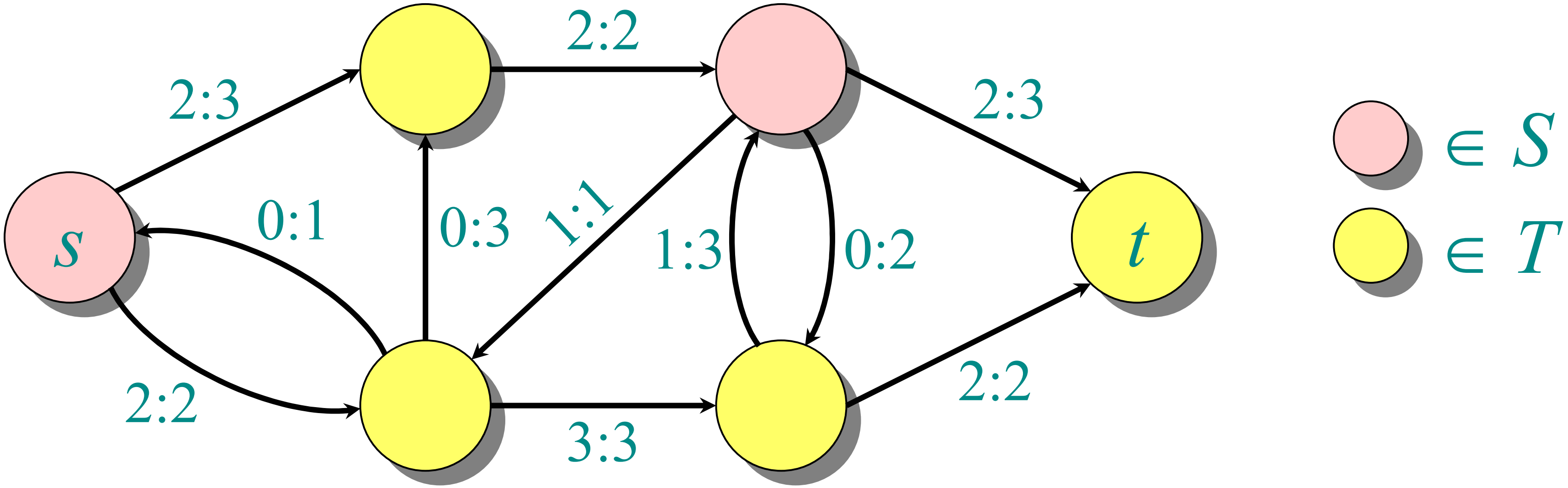
Lemma. For any flow f and any cut (S, T), we have $|f| = f(S, T)$.

Proof.

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S-s, V) \\ &= f(s, V) \\ &= |f|. \quad □ \end{aligned}$$

Capacity of a cut

Definition. The ***capacity of a cut*** (S, T) is $c(S, T)$.



$$c(S, T) = (3 + 2) + (1 + 2 + 3)$$
$$= 11$$

Upper bound on the maximum flow value

Theorem. The value of any flow is bounded above by the capacity of any cut.

Proof.

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T)\,. \quad \square \end{aligned}$$

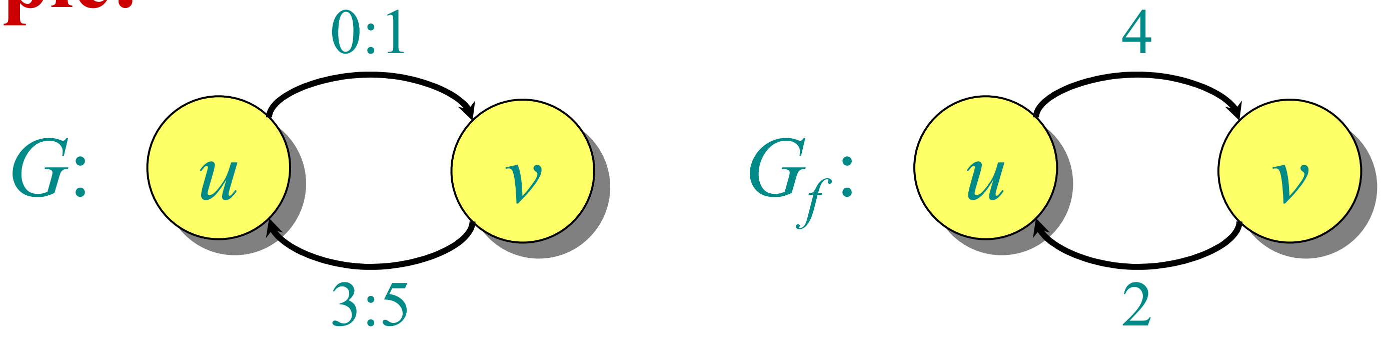
Residual network

Definition. Let f be a flow on $G = (V, E)$. The ***residual network*** $G_f(V, E_f)$ is the graph with strictly positive ***residual capacities***

$$c_f(u, v) = c(u, v) - f(u, v) > 0.$$

Edges in E_f admit more flow.

Example:

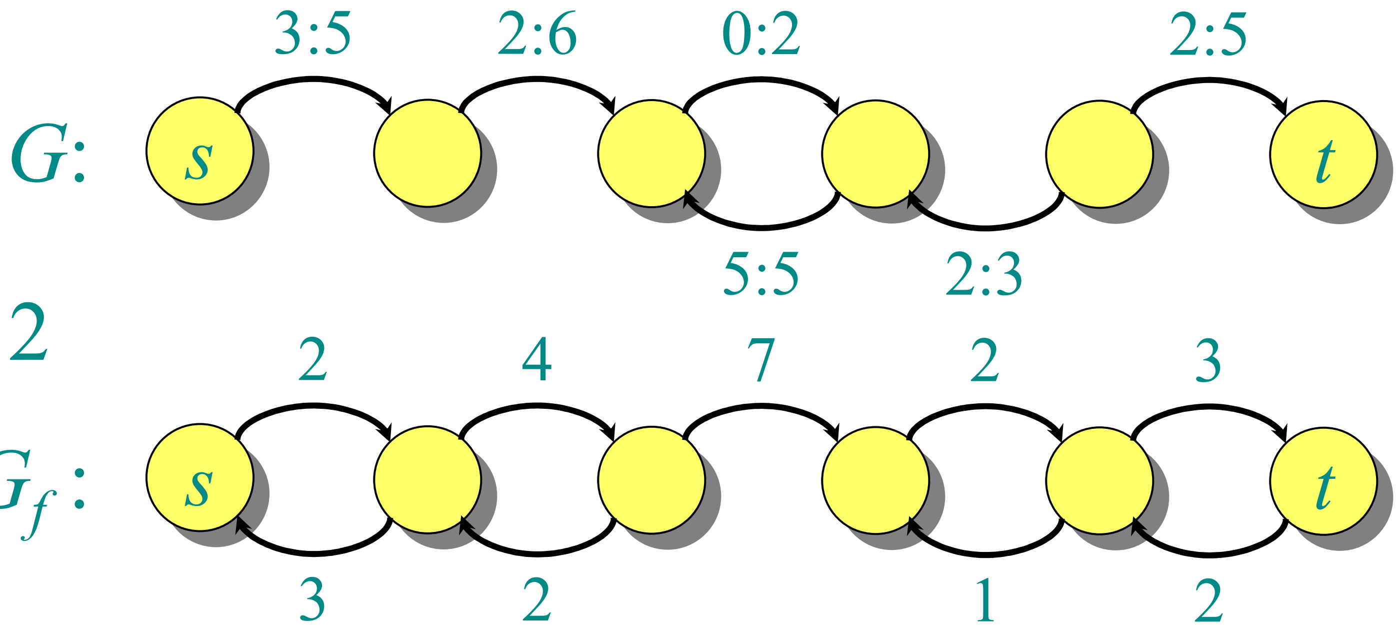


Lemma. $|E_f| \le 2|E|$. □

Augmenting paths

Definition. Any path from s to t in G_f is an ***augmenting path*** in G with respect to f. The flow value can be increased along an augmenting path p by $c_f(p) = \min_{(u,v)\in p} \{c_f(u,v)\}$.

Ex.:



$c_f(p) = 2$

Max-flow, min-cut theorem

Theorem. The following are equivalent:

1. f is a maximum flow.
2. f admits no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T).

Proof (and algorithms). Next time. □

Introduction to Algorithms

6.046J/18.401J/SMA5503

Lecture 23

Prof. Charles E. Leiserson

Recall from Lecture 22

- ***Flow value:*** $|f| = f(s, V)$.
- ***Cut:*** Any partition (S, T) of V such that $s \in S$ and $t \in T$.
- **Lemma.** $|f| = f(S, T)$ for any cut (S, T).
- **Corollary.** $|f| \le c(S, T)$ for any cut (S, T).
- ***Residual graph:*** The graph $G_f = (V, E_f)$ with strictly positive ***residual capacities*** $c_f(u, v) = c(u, v) - f(u, v) > 0$.
- ***Augmenting path:*** Any path from s to t in G_f.
- ***Residual capacity*** of an augmenting path:

$$c_f(p) = \min_{(u,v)\in p} \{c_f(u,v)\}.$$

Max-flow, min-cut theorem

Theorem. The following are equivalent:

1. $|f| = c(S, T)$ for some cut (S, T).
2. f is a maximum flow.
3. f admits no augmenting paths.

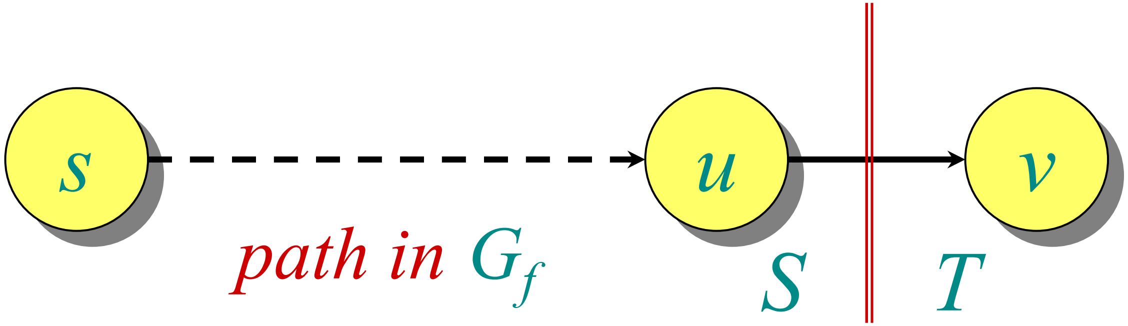
Proof.

(1) ⇒ (2): Since $|f| \le c(S, T)$ for any cut (S, T) (by the corollary from Lecture 22), the assumption that $|f| = c(S, T)$ implies that f is a maximum flow.

(2) ⇒ (3): If there were an augmenting path, the flow value could be increased, contradicting the maximality of f.

Proof (continued)

$(3) \Rightarrow (1)$: Suppose that f admits no augmenting paths. Define $S = \{v \in V : \text{there exists a path in } G_f \text{ from } s \text{ to } v\}$, and let $T = V - S$. Observe that $s \in S$ and $t \in T$, and thus (S, T) is a cut. Consider any vertices $u \in S$ and $v \in T$.



We must have $c_f(u, v) = 0$, since if $c_f(u, v) > 0$, then $v \in S$, not $v \in T$ as assumed. Thus, $f(u, v) = c(u, v)$, since $c_f(u, v) = c(u, v) - f(u, v)$. Summing over all $u \in S$ and $v \in T$ yields $f(S, T) = c(S, T)$, and since $|f| = f(S, T)$, the theorem follows. □

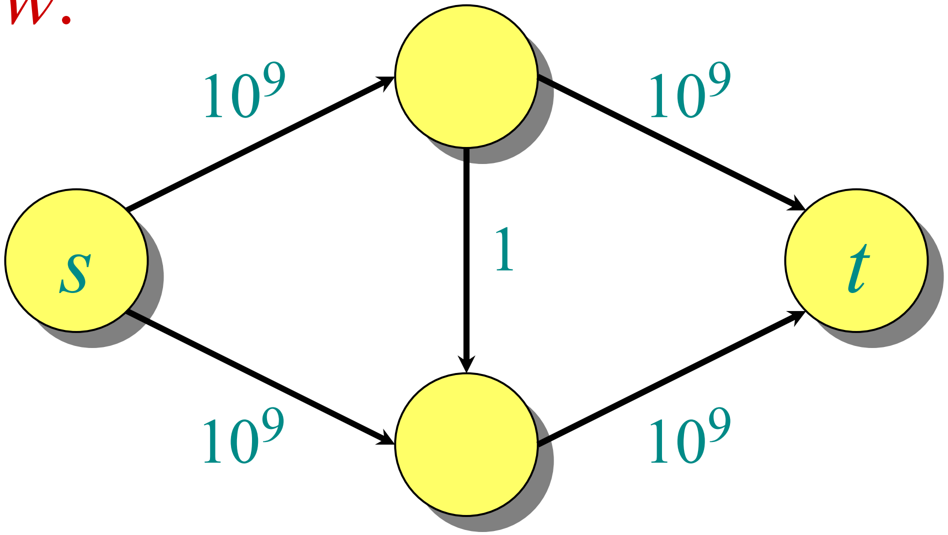
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



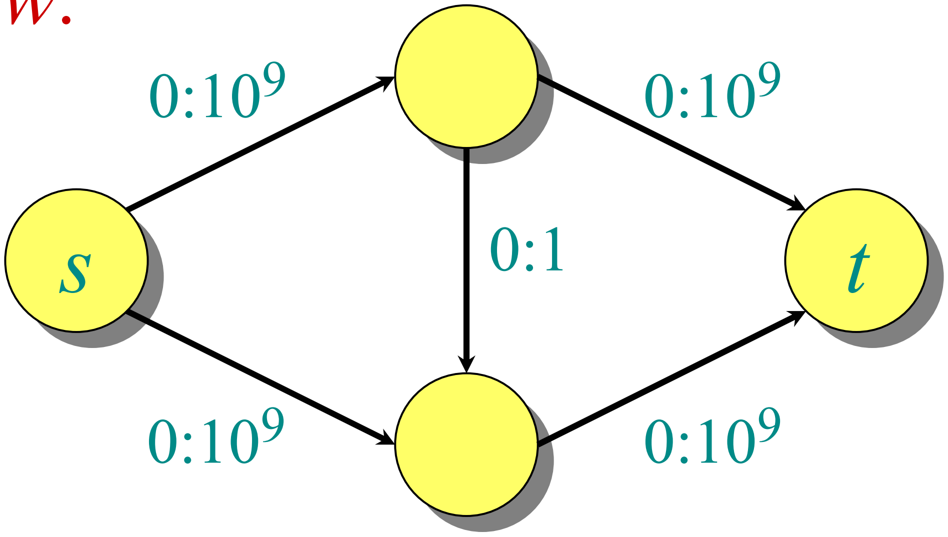
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



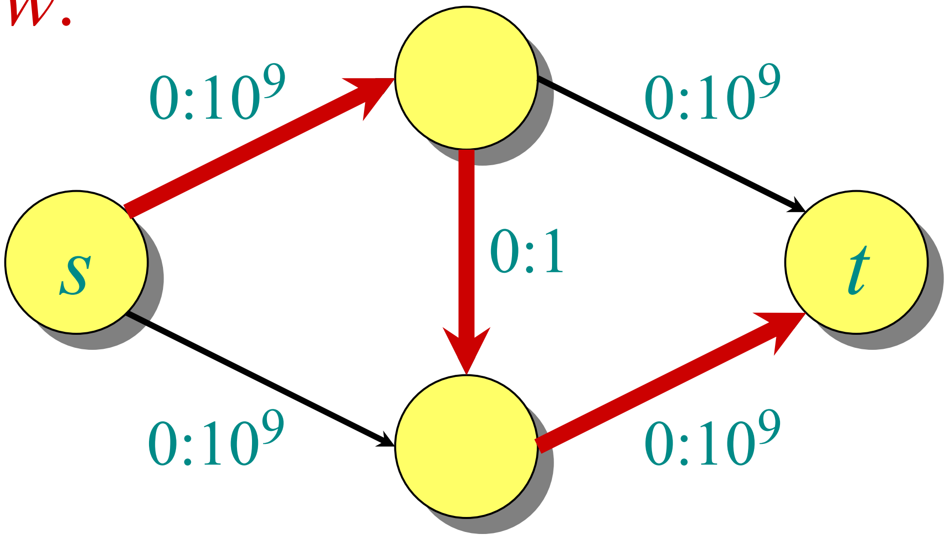
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



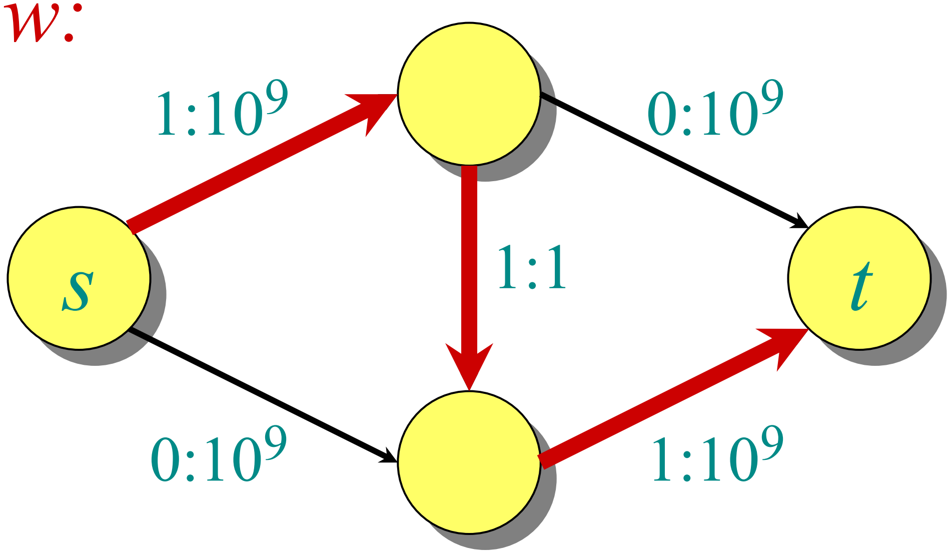
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:

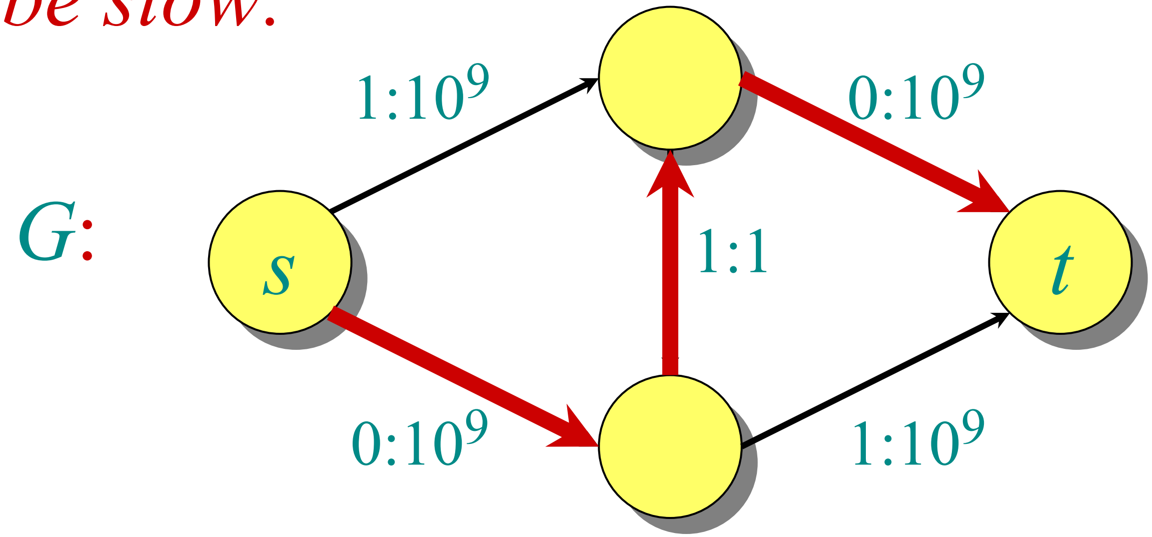


Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:



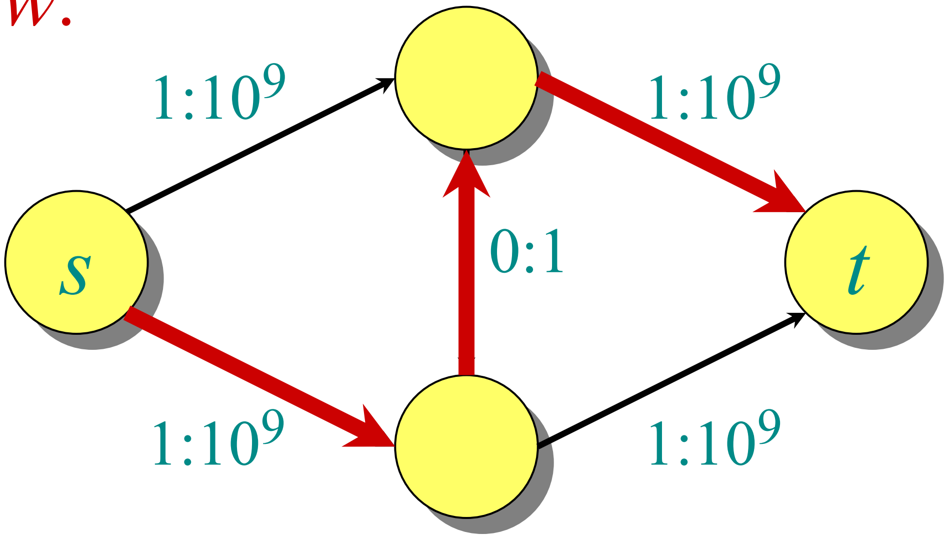
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



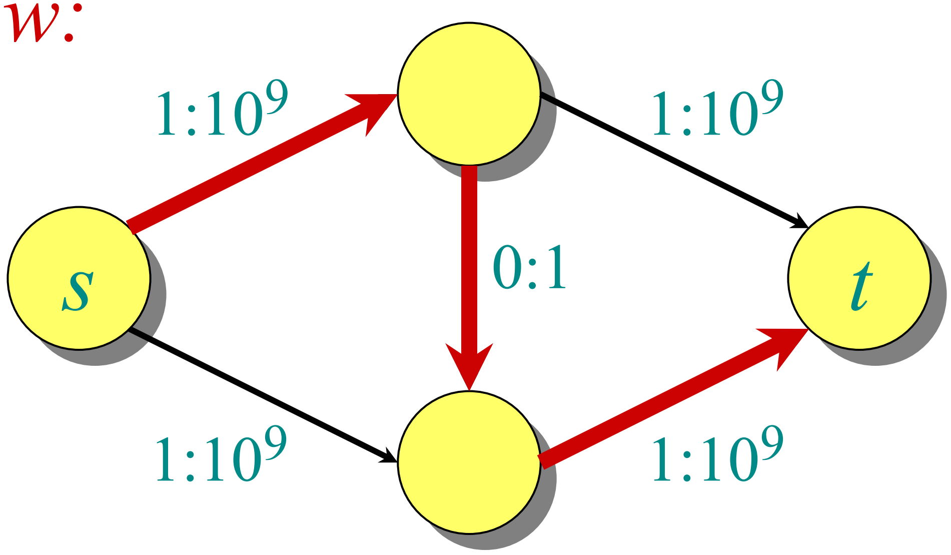
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



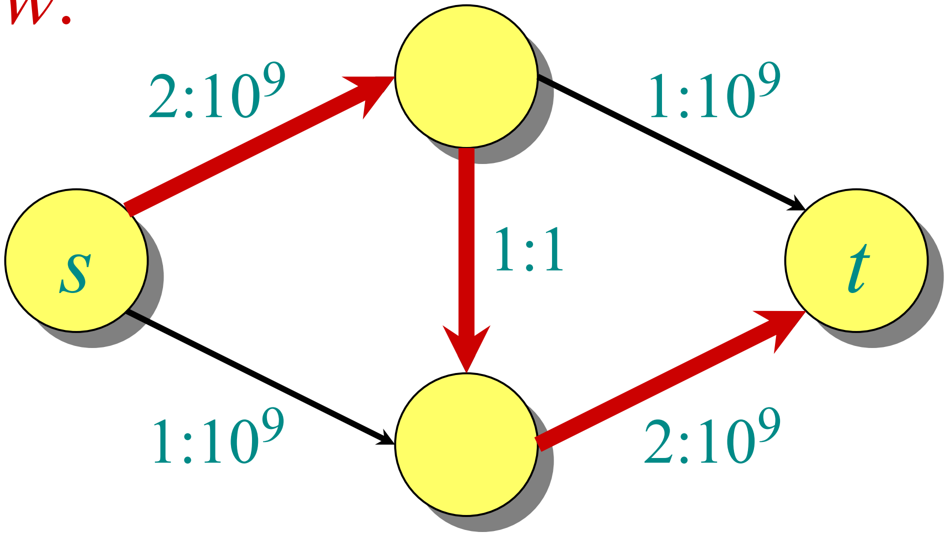
Ford-Fulkerson max-flow algorithm

Algorithm:

```
$f[u, v] \leftarrow 0$ for all $u, v \in V$
while an augmenting path $p$ in $G$ wrt $f$ exists
    do augment $f$ by $c_f(p)$
```

Can be slow:

G:



2 billion iterations on a graph with 4 vertices!

Edmonds-Karp algorithm

Edmonds and Karp noticed that many people's implementations of Ford-Fulkerson augment along a ***breadth-first augmenting path***: a shortest path in G_f from s to t where each edge has weight 1. These implementations would always run relatively fast.

Since a breadth-first augmenting path can be found in $O(E)$ time, their analysis, which provided the first polynomial-time bound on maximum flow, focuses on bounding the number of flow augmentations.

(In independent work, Dinic also gave polynomial-time bounds.)

Monotonicity lemma

Lemma. Let $\delta(v) = \delta_f(s, v)$ be the breadth-first distance from s to v in G_f. During the Edmonds-Karp algorithm, $\delta(v)$ increases monotonically.

Proof. Suppose that f is a flow on G, and augmentation produces a new flow f'. Let $\delta'(v) = \delta_{f'}(s, v)$. We'll show that $\delta'(v) \geq \delta(v)$ by induction on $\delta(v)$. For the base case, $\delta'(s) = \delta(s) = 0$.

For the inductive case, consider a breadth-first path $s \to \cdots \to u \to v$ in $G_{f'}$. We must have $\delta'(v) = \delta'(u) + 1$, since subpaths of shortest paths are shortest paths. Certainly, $(u, v) \in E_{f'}$, and now consider two cases depending on whether $(u, v) \in E_f$.

Case 1

Case: $(u, v) \in E_f$.

We have

$$\begin{aligned} \delta(v) &\le \delta(u) + 1 && \text{(triangle inequality)} \\ &\le \delta'(u) + 1 && \text{(induction)} \\ &= \delta'(v) && \text{(breadth-first path),} \end{aligned}$$

and thus monotonicity of $\delta(v)$ is established.

Case 2

Case: $(u, v) \notin E_f$.

Since $(u, v) \in E_{f'}$, the augmenting path p that produced f' from f must have included (v, u). Moreover, p is a breadth-first path in G_f:

$$p = s \to \cdots \to v \to u \to \cdots \to t\,.$$

Thus, we have

$$\begin{aligned} \delta(v) &= \delta(u) - 1 && \text{(breadth-first path)} \\ &\le \delta'(u) - 1 && \text{(induction)} \\ &\le \delta'(v) - 2 && \text{(breadth-first path)} \\ &< \delta'(v)\,, \end{aligned}$$

thereby establishing monotonicity for this case, too. □

Counting flow augmentations

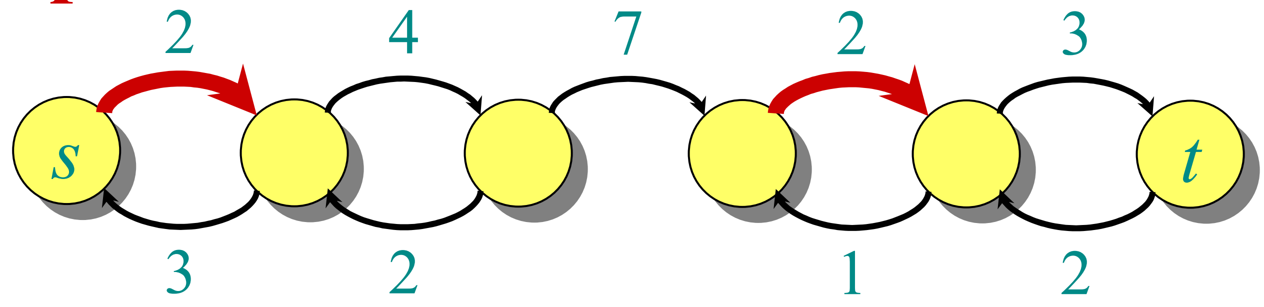
Theorem. The number of flow augmentations in the Edmonds-Karp algorithm (Ford-Fulkerson with breadth-first augmenting paths) is $O(VE)$.

Proof. Let p be an augmenting path, and suppose that we have $c_f(u, v) = c_f(p)$ for edge $(u, v) \in p$. Then, we say that (u, v) is ***critical***, and it disappears from the residual graph after flow augmentation.

Example:

$c_f(p) = 2$

G_f :



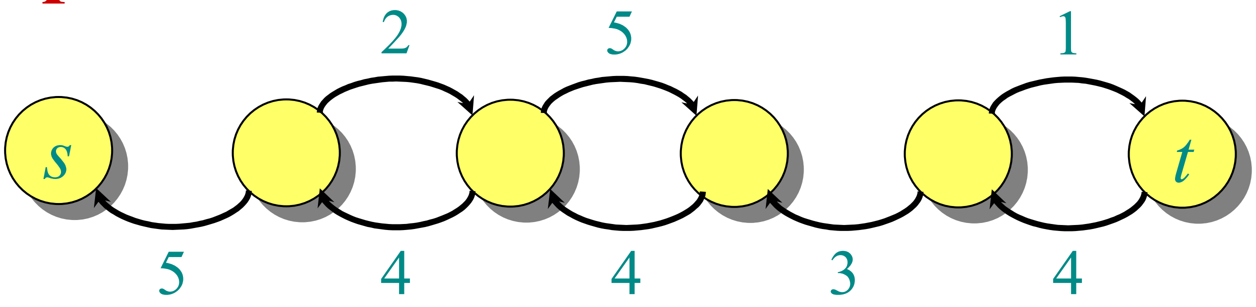
Counting flow augmentations

Theorem. The number of flow augmentations in the Edmonds-Karp algorithm (Ford-Fulkerson with breadth-first augmenting paths) is $O(VE)$.

Proof. Let p be an augmenting path, and suppose that the residual capacity of edge $(u, v) \in p$ is $c_f(u, v) = c_f(p)$. Then, we say (u, v) is ***critical***, and it disappears from the residual graph after flow augmentation.

Example:

$G_{f'}$:

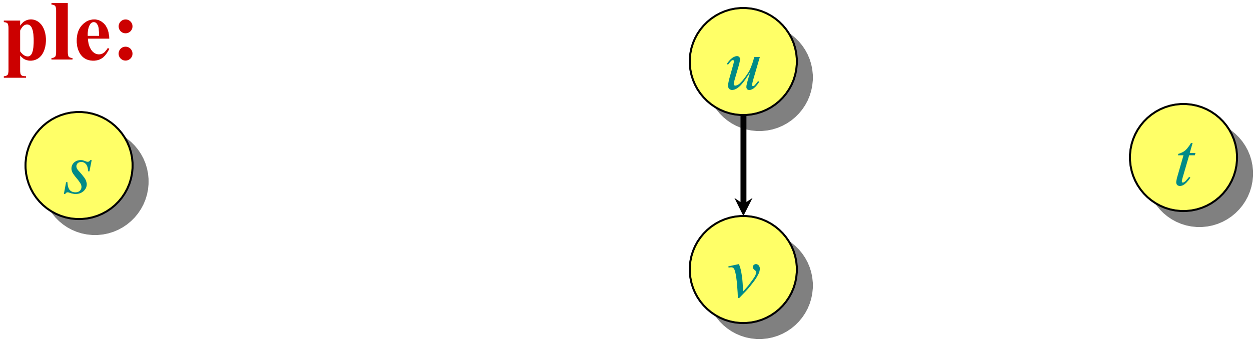


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned} \delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)} \\ &\geq \delta(v) + 1 && \text{(monotonicity)} \\ &= \delta(u) + 2 && \text{(breadth-first path).} \end{aligned}$$

Example:

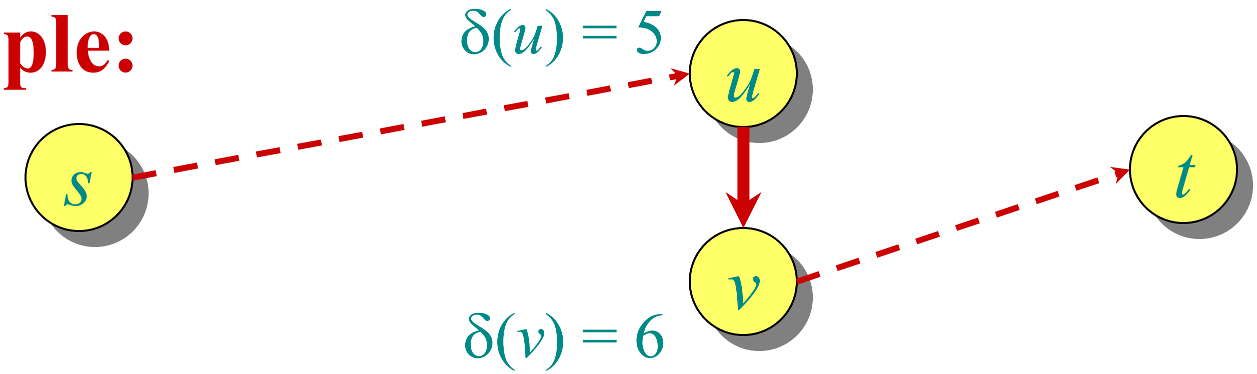


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned}\delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)}\\ &\geq \delta(v) + 1 && \text{(monotonicity)}\\ &= \delta(u) + 2 && \text{(breadth-first path).}\end{aligned}$$

Example:

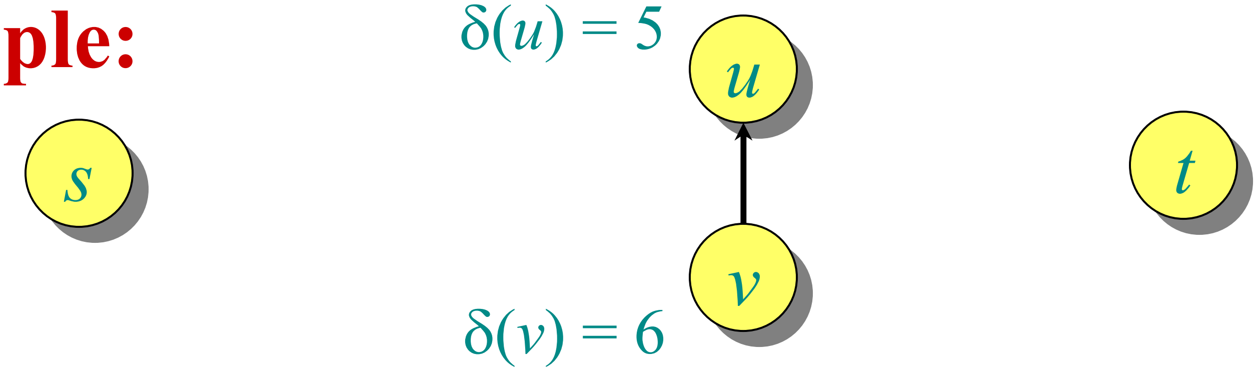


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned} \delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)} \\ &\ge \delta(v) + 1 && \text{(monotonicity)} \\ &= \delta(u) + 2 && \text{(breadth-first path).} \end{aligned}$$

Example:

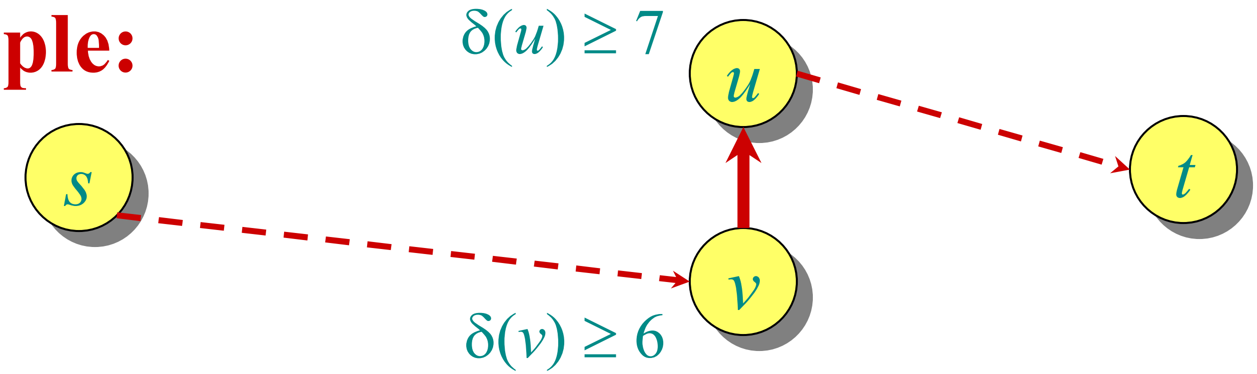


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned} \delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)} \\ &\geq \delta(v) + 1 && \text{(monotonicity)} \\ &= \delta(u) + 2 && \text{(breadth-first path).} \end{aligned}$$

Example:

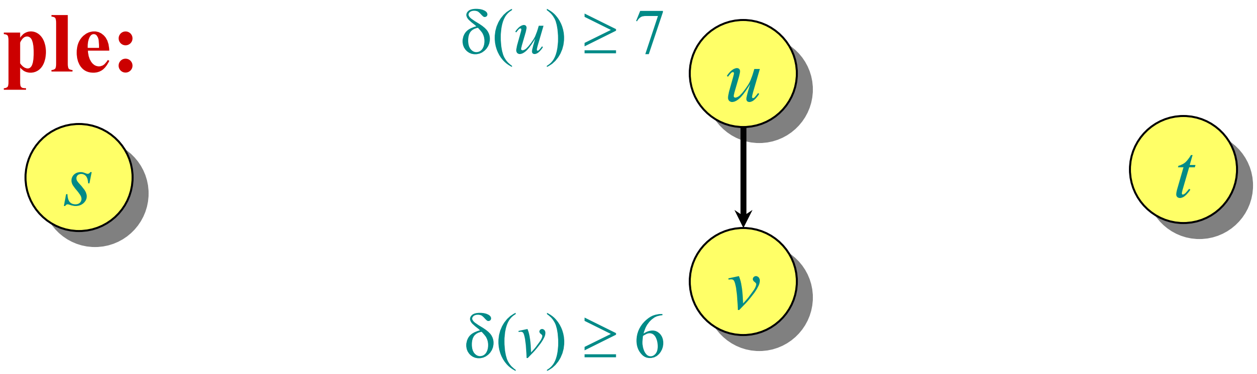


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned} \delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)} \\ &\geq \delta(v) + 1 && \text{(monotonicity)} \\ &= \delta(u) + 2 && \text{(breadth-first path).} \end{aligned}$$

Example:

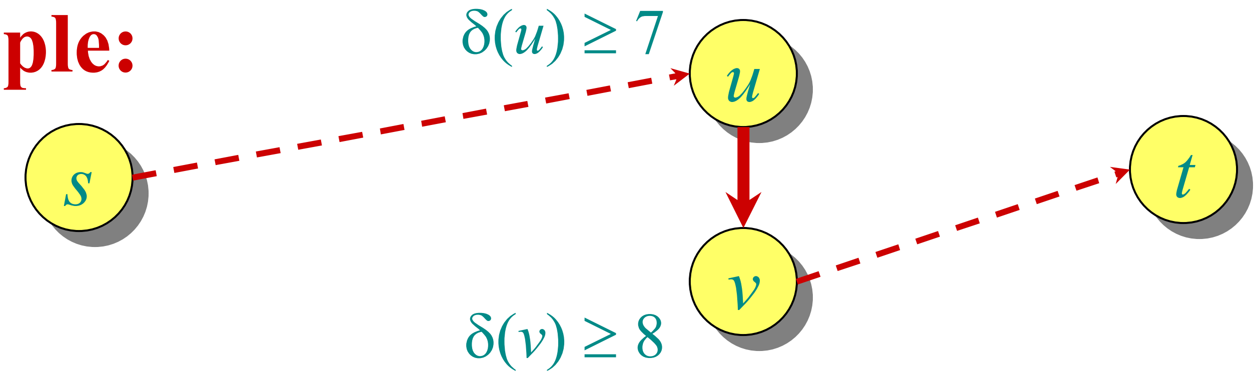


Counting flow augmentations (continued)

The first time an edge (u, v) is critical, we have $\delta(v) = \delta(u) + 1$, since p is a breadth-first path. We must wait until (v, u) is on an augmenting path before (u, v) can be critical again. Let δ' be the distance function when (v, u) is on an augmenting path. Then, we have

$$\begin{aligned}
\delta'(u) &= \delta'(v) + 1 && \text{(breadth-first path)} \\
&\ge \delta(v) + 1 && \text{(monotonicity)} \\
&= \delta(u) + 2 && \text{(breadth-first path).}
\end{aligned}$$

Example:



Running time of Edmonds-Karp

Distances start out nonnegative, never decrease, and are at most $|V| - 1$ until the vertex becomes unreachable. Thus, (u, v) occurs as a critical edge $O(V)$ times, because $\delta(v)$ increases by at least 2 between occurrences. Since the residual graph contains $O(E)$ edges, the number of flow augmentations is $O(V E)$. □

Corollary. The Edmonds-Karp maximum-flow algorithm runs in $O(V E^2)$ time.

Proof. Breadth-first search runs in $O(E)$ time, and all other bookkeeping is $O(V)$ per augmentation. □

Best to date

- The asymptotically fastest algorithm to date for maximum flow, due to King, Rao, and Tarjan, runs in $O(VE \log_{E/(V \lg V)} V)$ time.
- If we allow running times as a function of edge weights, the fastest algorithm for maximum flow, due to Goldberg and Rao, runs in time
$$O\left(\min\{V^{2/3}, E^{1/2}\} \cdot E \lg(V^2/E + 2) \cdot \lg C\right),$$
where C is the maximum capacity of any edge in the graph.