

Algorithms and Data Structures

Lecture notes: Definitions, terminology, Brassard

Chap. 2

Lecturer: Michel Toulouse

Vietnamese-German University
michel.toulouse@vgu.edu.vn

4 avril 2016

Outline

- ▶ Definitions of terms like *algorithm*, *input size*, *formal parameters*, *output*, *running time*, *time complexity*, *space complexity*, *basic operations*, *rate of growth*, etc.
- ▶ Distinction between terms like *problem* and *problem's instance*, or *algorithm* and *program*.
- ▶ Introduction of the basic procedure and steps to analyze algorithms.
- ▶ For a YouTube history of computers, see
<https://www.youtube.com/watch?v=hGXZj1DSXyg>

Algorithms

- ▶ What is an algorithm ?
- ▶ Origin of the word : al-Khowarizmi, Persian scholar (780-850), early "textbook" on arithmetic operations.
- ▶ The following is an intuitive explanation of "algorithm", not a formal definition : *An algorithm is a set of instructions for some calculation that can be delegated to a machine.*
- ▶ Hence it must - be precise, unambiguous, - not allow subjective decisions.

Problem & Problem's instance

- ▶ An algorithm solves some general problem, i.e. it works for a whole class of instances, not only for a particular one.
- ▶ A problem is a set and a problem's instance is an element of this set. An algorithm solves all the problem's instances of a set.

Problem & Problem's instance

- ▶ Example, the Prime Problem : “Given a positive integer greater than one” (parameter), does this number can be divided by an integer other than one, “yes or no” (property to be satisfied by the answer) ?
- ▶ An **instance** of a problem is obtained by associating values to the formal parameters of the problem.
- ▶ An instance of Problem Prime is “Is 9 prime ?”

An every-day example of an algorithm

- ▶ Paper-and-pencil methods for multiplying two integers, as learned at school.
- ▶ Problem : Given two (arbitrary !) integers, compute their product.
- ▶ Instance : a pair of integers.

How to describe algorithms

By programs : Specific realization of a (more abstract) algorithm.

By pseudocode : Similar to usual procedural languages, free of non-essential details.

```
input  
SequentialSearch( $\overbrace{A[1..n]}$ , val)  
int loc = 1;  
while (loc  $\leq$  n & A[loc]  $\neq$  val)  
    loc = loc + 1;  
output  
return  $\overbrace{\text{loc}}$ ;
```

How to describe algorithms

Informally, in words. Programmer must fill in all details.

Given an array of integers

222, 179, 77, 156, 18, 96, 12

sort it in increasing order.

Algorithm :

- ▶ search for the smallest integer in the array and swap it with the integer in the first entry
- ▶ search for the second smallest integer in the array and swap it with the integer in the second entry
- ⋮
- ▶ search for the largest integer in the array and swap it with the integer in the last entry

How to describe algorithms

- ▶ search for the smallest integer in the array and swap it with the integer in the first entry
- ▶ search for the second smallest integer in the array and swap it with the integer in the second entry
- ⋮
- ▶ search for the largest integer in the array and swap it with the integer in the last entry

Same algo in pseudocode

```
Selection sort(A[1..n])
  for i = 1 to n - 1 do
    minj = i; minx = A[i];
    for j = i + 1 to n do
      if A[j] < minx then
        minj = j; minx = A[j];
    A[minj] = A[i]; A[i] = minx;
```

Problems, Algorithms and Programs

- ▶ We design algorithms and programs for a problem, not for a problem instance. A given algorithm or program should work for all the problem's instances !
- ▶ There is *often* many different algorithms for a same problem (ex. multiplication : à la russe, Arabic, classic [American, English], divide-and-conquer)
- ▶ There is *always* many different programs for a same algorithm.

Algorithm analysis

- ▶ Given that there is often several algorithms for a same problem, one might want to compare the algorithms with each others
- ▶ If you discover a new algorithm, you might want to compare it with existing algorithms
- ▶ Then the question becomes :
 - ▶ By what should we compare two algorithms with each others ?
 - ▶ What should we use as a measure of how “good” an algorithm is ?
- ▶ Algorithm analysis is about comparing algorithms as well as making sure that algorithms solve the problem they have been designed to solve.

Algorithm behavior : running time

- ▶ The time taken by an algorithm to solve a problem instance is a critical component of an algorithm behavior
- ▶ But the running time for every instance is too complicated and cumbersome to calculate.

222, 179, 77, 156, 18, 96, 12

112, 189, 976, 551, 94, 18, 55

- ▶ Rather we compute the *running time on instances of a given input size.*
 - ▶ Consider maximum or average of running times for all instances of size n . (Worst-case/ average-case analysis.)
 - ▶ **Size n** must be defined appropriately : e.g. number of bits, digits, items.

Algorithm behavior : running time

- ▶ We need to be little bit careful when measuring the running time of an algorithm because other factors can affect the computational time. For example :
 - ▶ Characteristics of the computer (e.g. processor speed, amount of memory, file-system type, number and type of elementary operations, size of the registers, etc.)
 - ▶ The way the algorithm is coded (the program !!!)
 - ▶ The particular instance of data the algorithm is operating on (e.g., amount of data, type of data).

“Rate of growth”

- ▶ Show how time scale when problem instance size grow ?

222, 179, 77, 156, 18, 96, 12

222, 179, 77, 156, 18, 96, 12, 112, 189, 976, 551, 94, 18, 55

- ▶ Example : compare n ; n^2 ; 2^n .
- ▶ The rate of growth ignores constant factors in running time as they depend on other factors than the algorithm itself : - speed of machine, - minor implementation details, etc.
- ▶ Using the rate of growth, we can compare algorithms, independent of the implementation details

Time taken for different running times and input sizes

$f(n)$	$n = 2$	$n = 64$	$n = 1024$	$n = 65536$
1	1 μs	1 μs	1 μs	1 μs
$\log_2 n$	1 μs	6 μs	10 μs	16 μs
n	2 μs	64 μs	1.02 ms	65.5 ms
$n \log_2 n$	2 μs	384 μs	10.2 ms	1.05 s
n^2	4 μs	4.1 ms	1.04 s	71.6 min
n^3	8 μs	0.26 s	17.9 min	8.9 yr
2^n	4 μs	5849 cen	$2^{972.5}$ cen	2^{65484} cen

Note : μs = microsecond = one millionth of a second, ms = millisecond = one thousandth of a second, s = second, min = minute, yr = year, and cen = century. (Assume 1 μs per op.)

Time is not “wall-clock” time

- ▶ *Elementary operations* are used to measure time complexity of algorithms.
It is assumed that one elementary operation requires one time “unit”.
- ▶ What is “elementary”? Depends on type of data, like “input size”.
- ▶ Some simple measure of the “size” of the data that the algorithm is operating on are needed, e.g., the size of an array, the number of nodes in a graph, the dimensions of a matrix.
- ▶ Some examples below show appropriate viewpoints.

Example

SequentialSearch(A[1..n], val)

loc = 0 ; (1 instr but 10 machine opts)

while (*loc < n & A[loc] ≠ val*) *(1 instr but 200 machine opts)*

loc = loc + 1; (1 instr but 20 machine opts)

return *loc*; *(1 instr but 5 machine opts)*

- ▶ What is the size of the input ?
- ▶ What are the elementary operations ?
- ▶ How many elementary operations, as a function of the array size n , are required by SequentialSearch ?

Example

```
Selection sort(A[1..n])
for  $i = 1$  to  $n - 1$  do
     $minj = i$ ;  $minx = A[i]$ ;
    for  $j = i + 1$  to  $n$  do
        if  $A[j] < minx$  then
             $minj = j$ ;  $minx = A[j]$ ;
     $A[minj] = A[i]$ ;  $A[i] = minx$ ;
```

- ▶ What is the size of the input ?
- ▶ What are the elementary operations ?
- ▶ How many elementary operations, as a function of the array size n , are required by Selection sort ?

"Analyzing the running time behavior"

- ▶ Identify typical input data
- ▶ Identify elementary operations
- ▶ Derive a mathematical analysis (exact analysis followed by the grow rate stated using asymptotic notation)
- ▶ Associate the algorithm to a *complexity class* (polynomial, exponential).

Typical input data

- ▶ We first need to determine what the input is, and *how much data* is being input.
- ▶ We need to determine which of the data affects the running time.
- ▶ We usually use n to denote the number of data items to be processed.
- ▶ This could be
 - ▶ size of a file
 - ▶ size of an array or matrix
 - ▶ number of nodes in a tree or graph
 - ▶ degree of a polynomial

Elementary operation

We talk about elementary operations (instructions) when we consider operations in a hardware independent fashion.

Recall that we are interested in rate of growth, not the exact running time. Thus, we can pick operations (instructions) that will run most often in the code.

Determine the number of times these operations will be executed as a function of the size of the input data.

Need to pick the operations that are executed most often, need to recognize when an operation can or cannot be performed in a constant amount of time.

Example : Sorting

- ▶ Problem statement : Sort in increasing order an array of n integers, output the sorted array.
- ▶ Algorithm :

Selection Sort(A[1..n])

for $i = 1$ **to** $n - 1$ **do**

$minj = i$; $minx = A[i]$;

for $j = i + 1$ **to** n **do**

if $A[j] < minx$ **then**

$minj = j$; $minx = A[j]$;

$A[minj] = A[i]$; $A[i] = minx$;

- ▶ We focus on the comparison ($<$) (this is the elementary operation) and ignore all the other instructions.

Example : Sorting

```
Selection Sort(A[1..n])
for  $i = 1$  to  $n - 1$  do
     $minj = i$ ;  $minx = A[i]$ ;
    for  $j = i + 1$  to  $n$  do
        if  $A[j] < minx$  then
             $minj = j$ ;  $minx = A[j]$ ;
     $A[minj] = A[i]$ ;  $A[i] = minx$ ;
```

- ▶ If we calculate the number of operations in this function based on the comparison operator, we have :
 - ▶ for $n = 1$, 0 comparison
 - ▶ for $n = 2$, 1 comparison
 - ⋮
 - ▶ for n , $n^2/2$ comparisons

Example : Search for Maximum

- ▶ Problem statement : Search the maximum element in an array of n integers.

- ▶ Algorithm :

```
max(A[1..n])
```

```
    max_value = int.MIN_VAL ;
```

```
    for ( i = 0; i < n; i ++ )
```

```
        max_value = MAXIMUM(max_value, A[i]);
```

```
    return max_value;
```

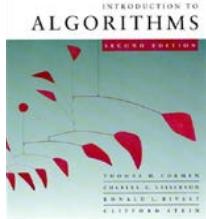
- ▶ We focus on the assignment (=) inside the loop and ignore the other instructions.

- ▶ for an array of length 1, 1 comparison

- ▶ for an array of length 2, 2 comparisons

⋮

- ▶ for an array of length n , n comparisons

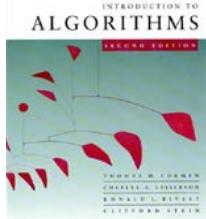


Analysis of algorithms

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

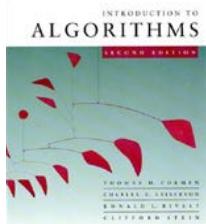
What's more important than performance?

- 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.
- Performance is the *currency* of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!



The problem of sorting

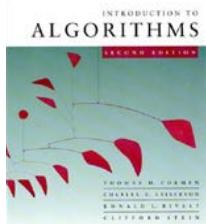
Input: sequence $\langle a_1, a_2, \dots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

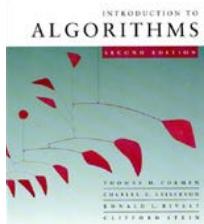
Output: 2 3 4 6 8 9



Insertion sort

“pseudocode”

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

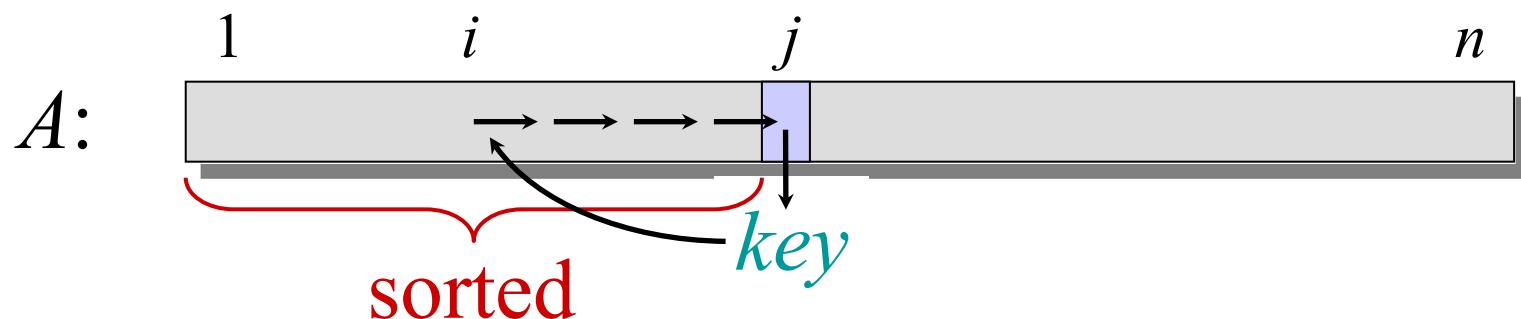


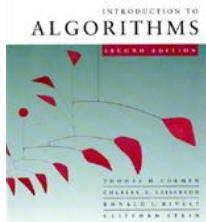
Insertion sort

“pseudocode”

{

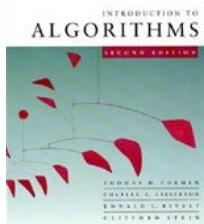
```
INSERTION-SORT ( $A, n$ )      ▷  $A[1 \dots n]$ 
  for  $j \leftarrow 2$  to  $n$ 
    do  $key \leftarrow A[j]$ 
         $i \leftarrow j - 1$ 
        while  $i > 0$  and  $A[i] > key$ 
          do  $A[i+1] \leftarrow A[i]$ 
               $i \leftarrow i - 1$ 
     $A[i+1] = key$ 
```





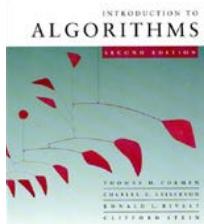
Example of insertion sort

8 2 4 9 3 6



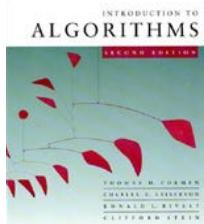
Example of insertion sort





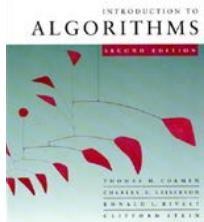
Example of insertion sort



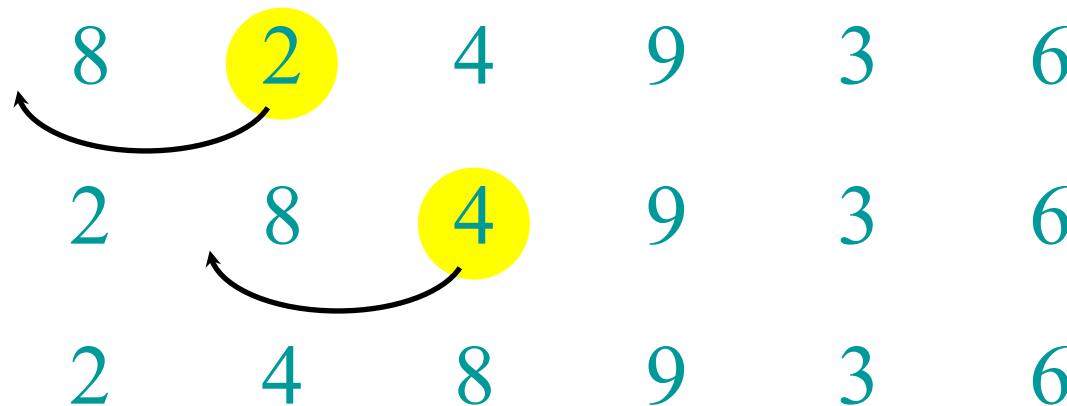


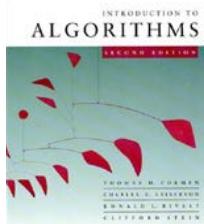
Example of insertion sort



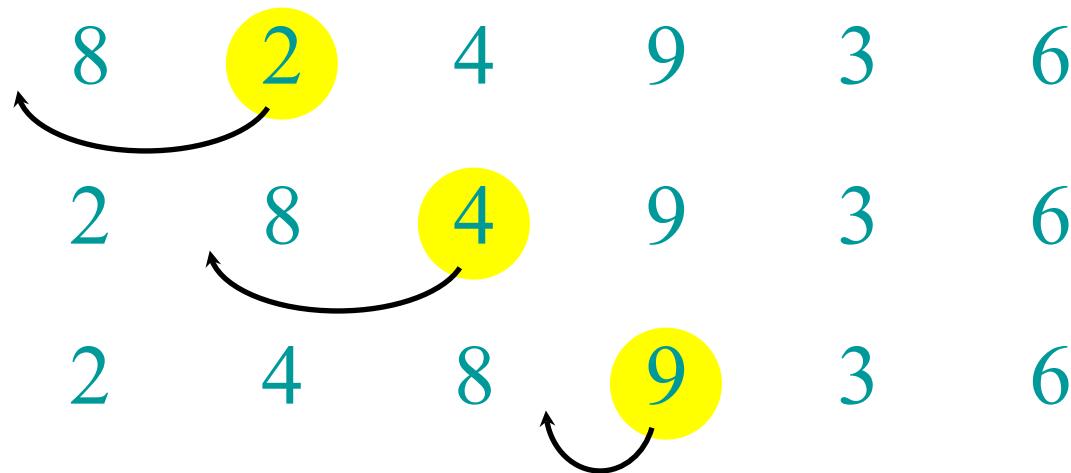


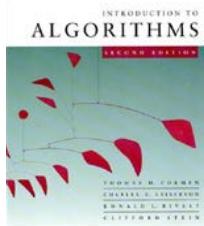
Example of insertion sort



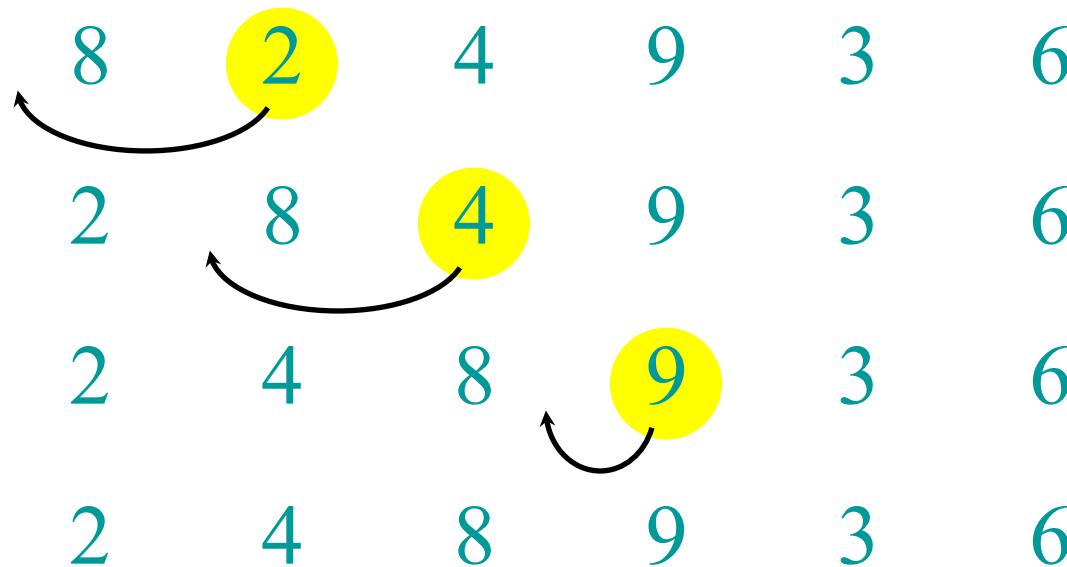


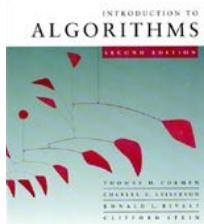
Example of insertion sort



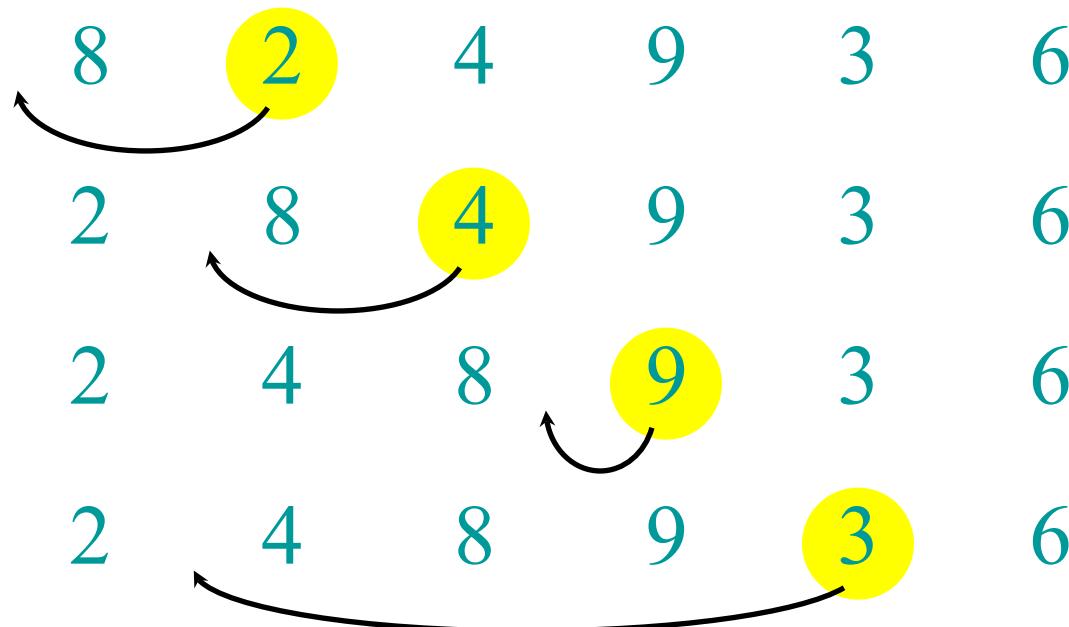


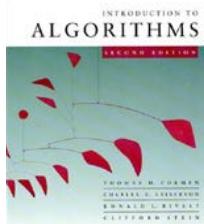
Example of insertion sort



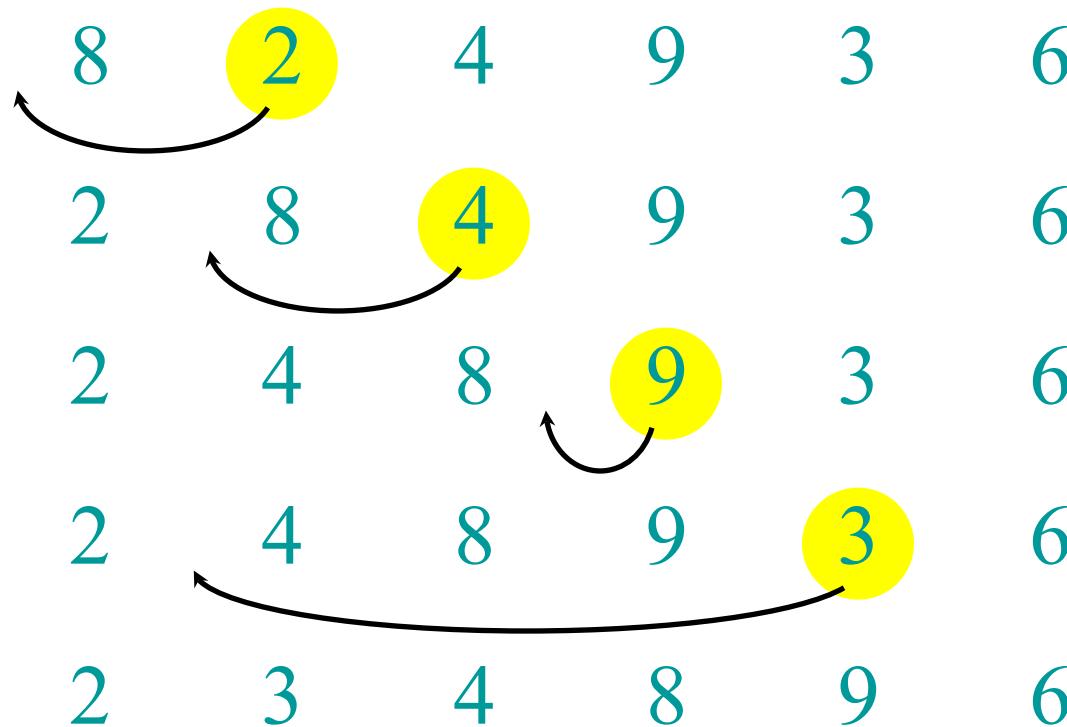


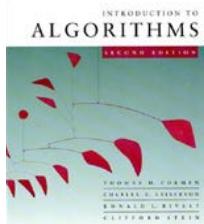
Example of insertion sort



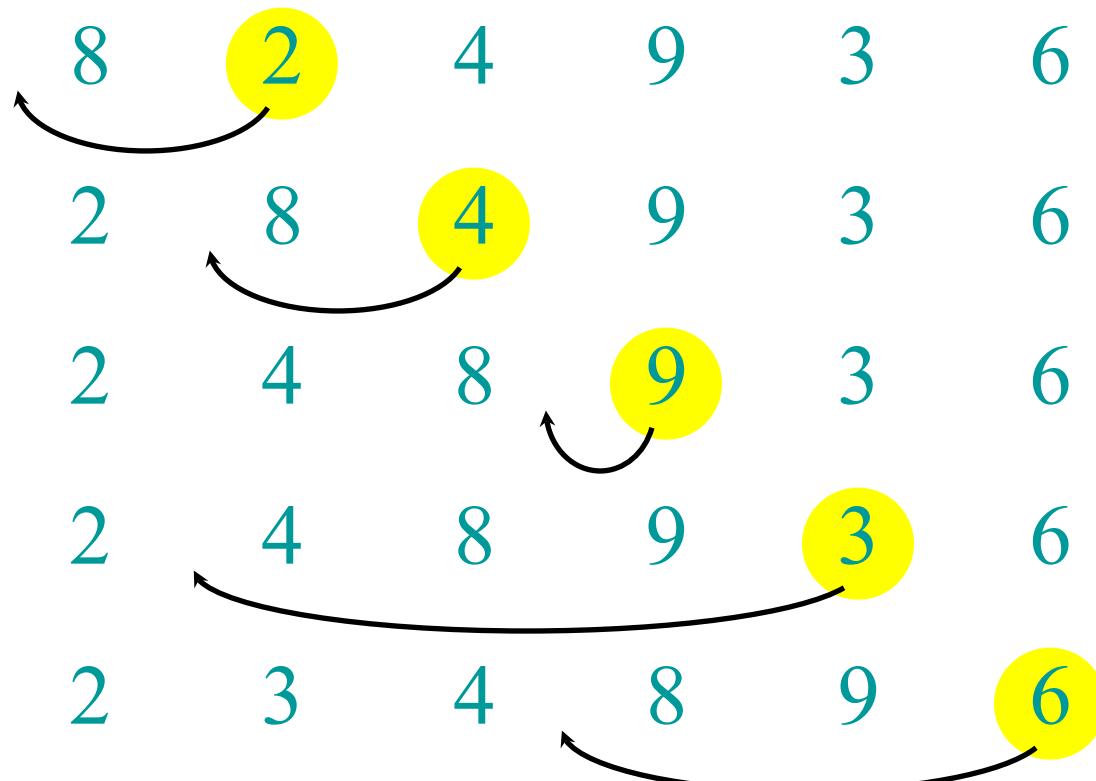


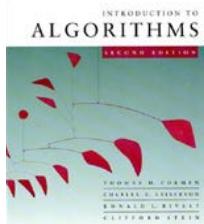
Example of insertion sort



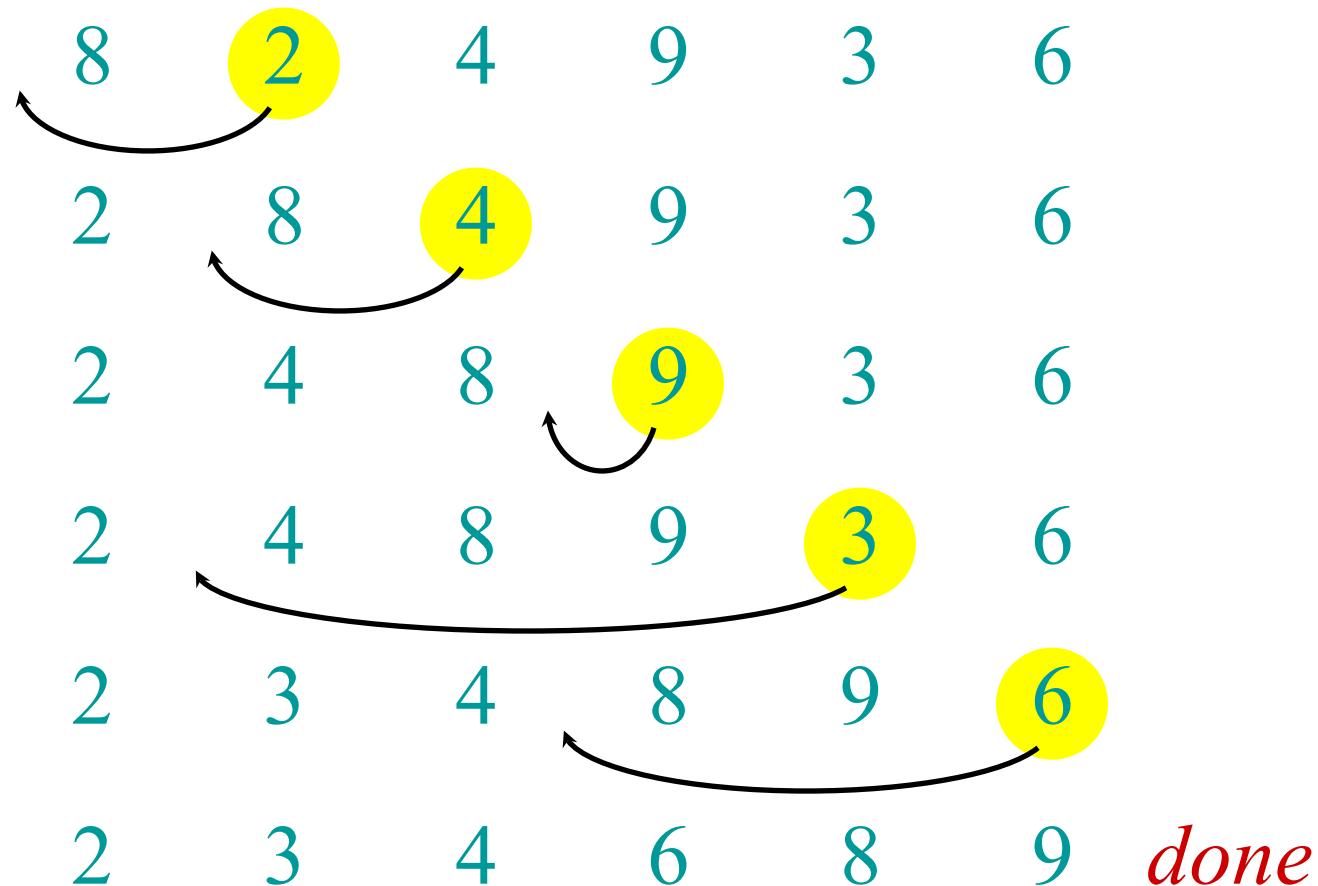


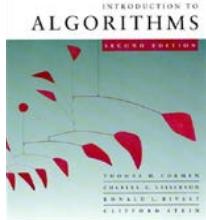
Example of insertion sort





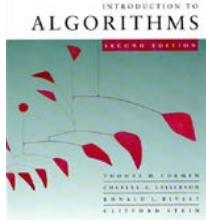
Example of insertion sort





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)

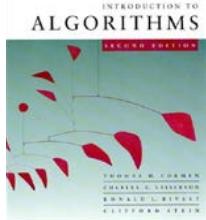
- $T(n)$ = maximum time of algorithm on any input of size n .

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

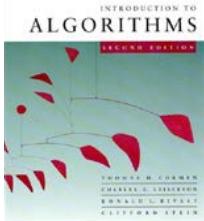
What is insertion sort's worst-case time?

- 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 $n \rightarrow \infty$.

“Asymptotic Analysis”



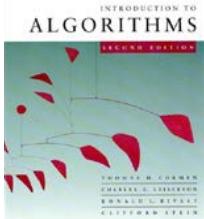
Θ -notation

Math:

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

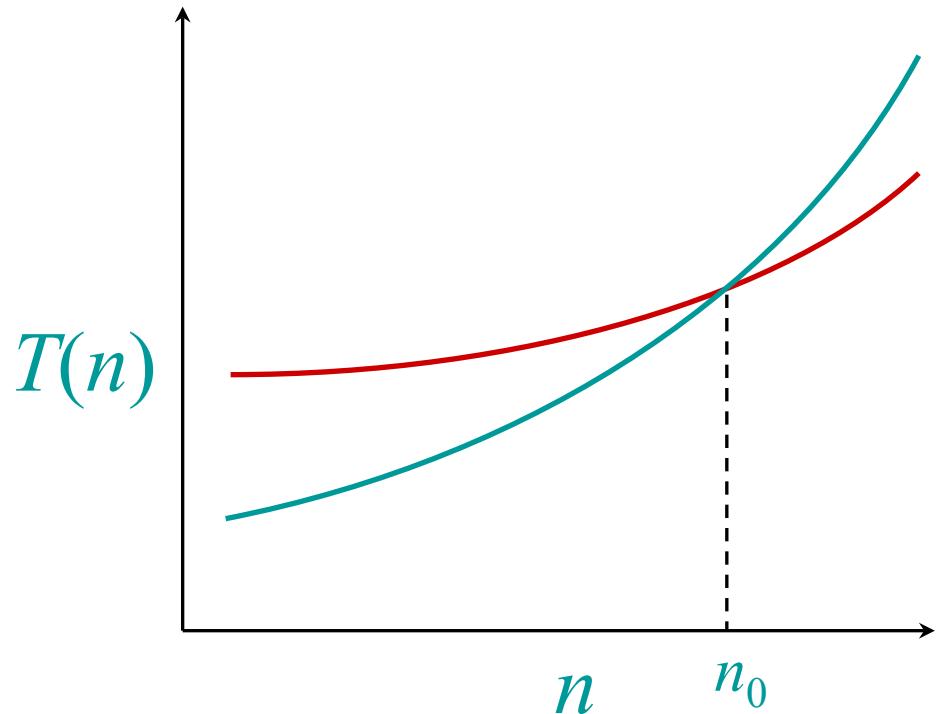
Engineering:

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

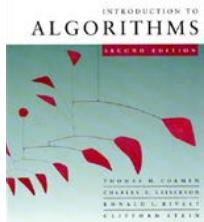


Asymptotic performance

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.

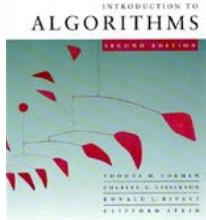
$$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(n^2) \quad [\text{arithmetic series}]$$

Average case: All permutations equally likely.

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

Is insertion sort a fast sorting algorithm?

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

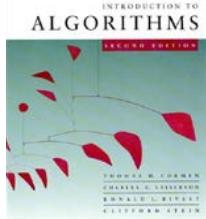


Merge sort

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

1. If $n = 1$, done.
2. Recursively sort $A[1 \dots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \dots n]$.
3. “*Merge*” the 2 sorted lists.

Key subroutine: MERGE



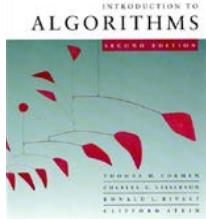
Merging two sorted arrays

20 12

13 11

7 9

2 1



Merging two sorted arrays

20 12

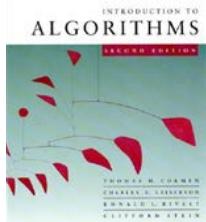
13 11

7 9

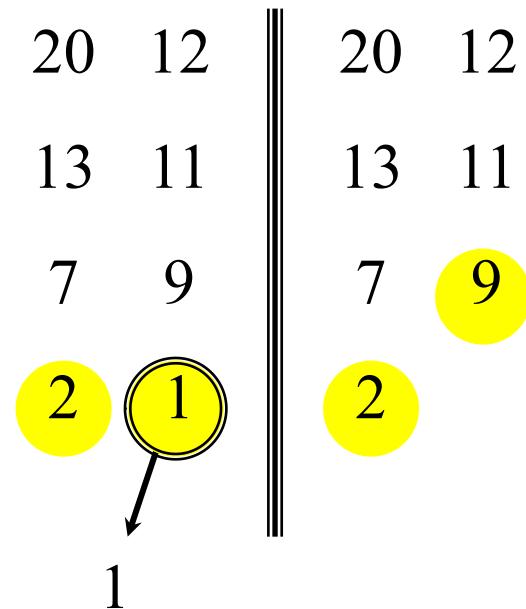
2 1

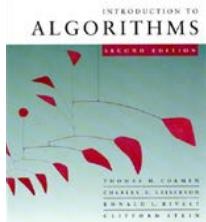


1

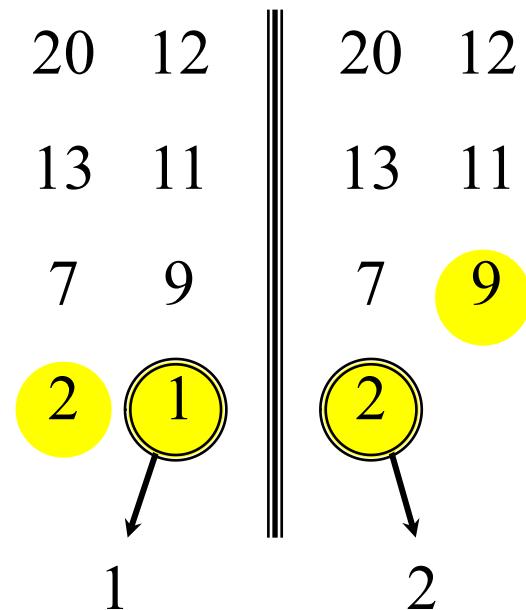


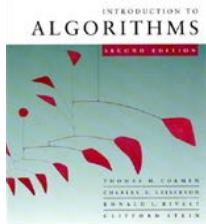
Merging two sorted arrays



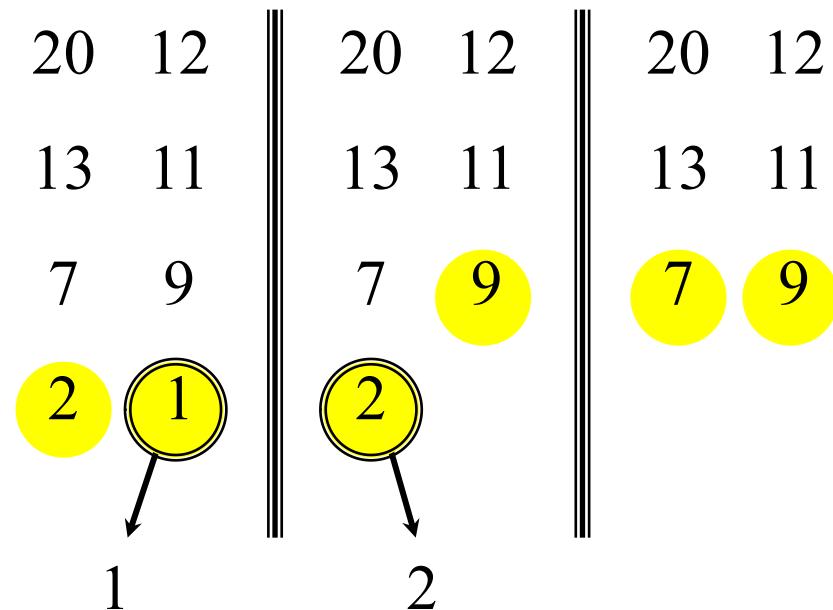


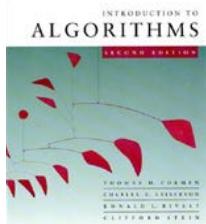
Merging two sorted arrays



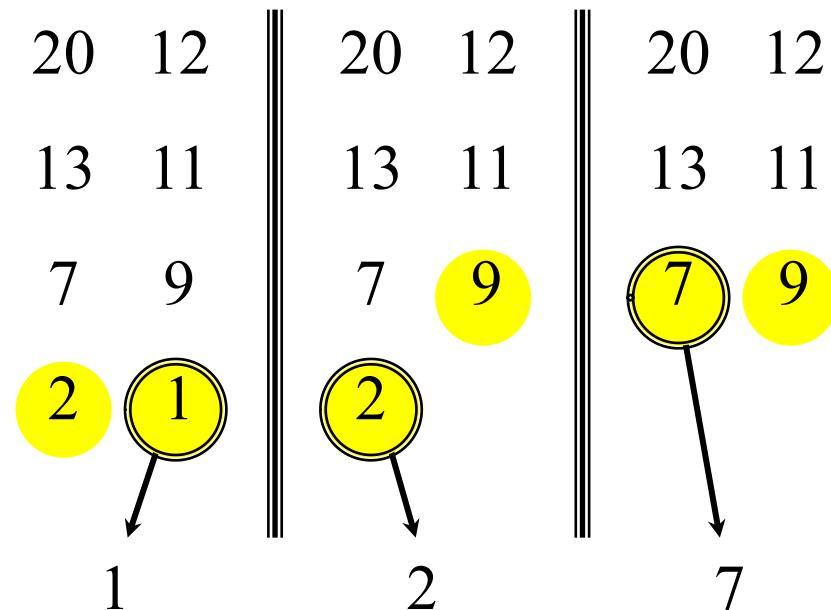


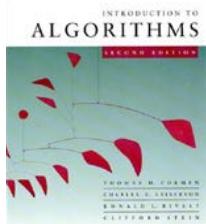
Merging two sorted arrays



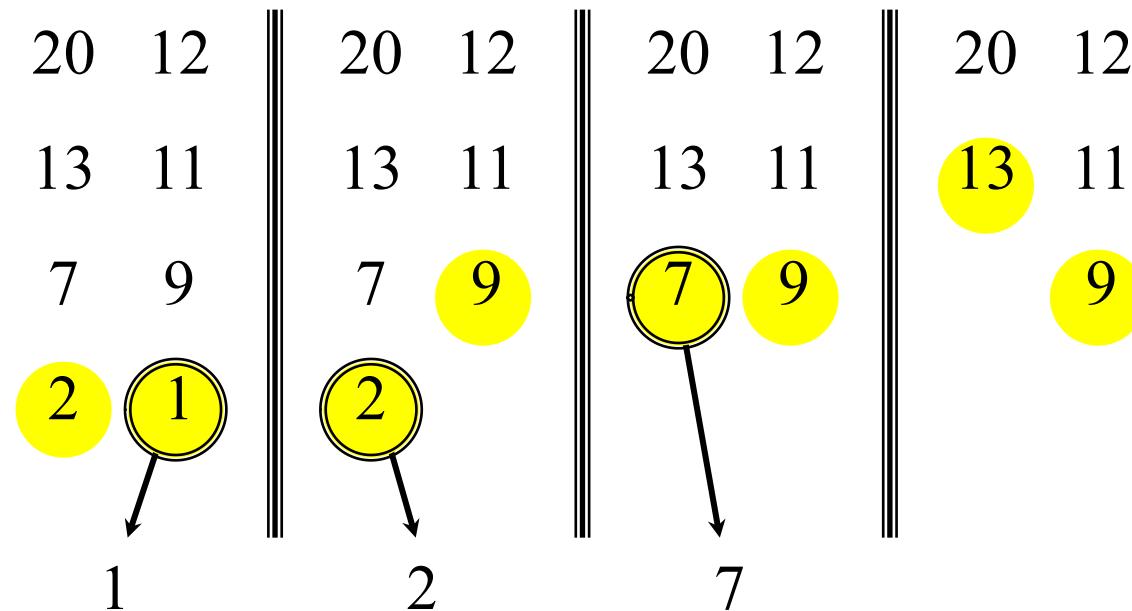


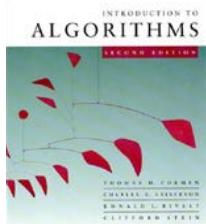
Merging two sorted arrays



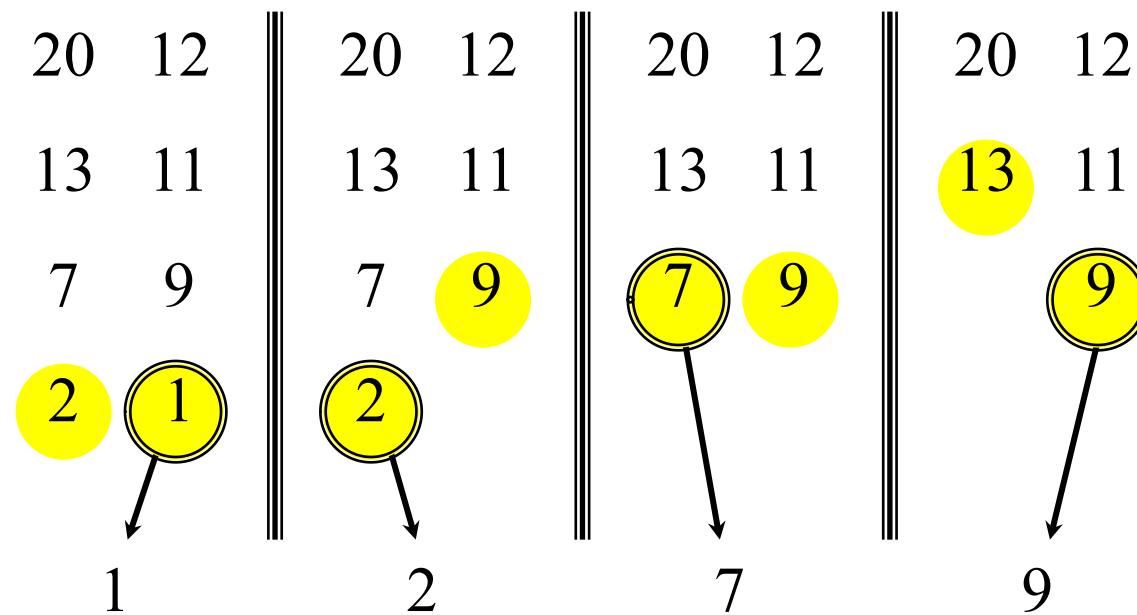


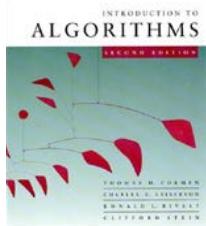
Merging two sorted arrays



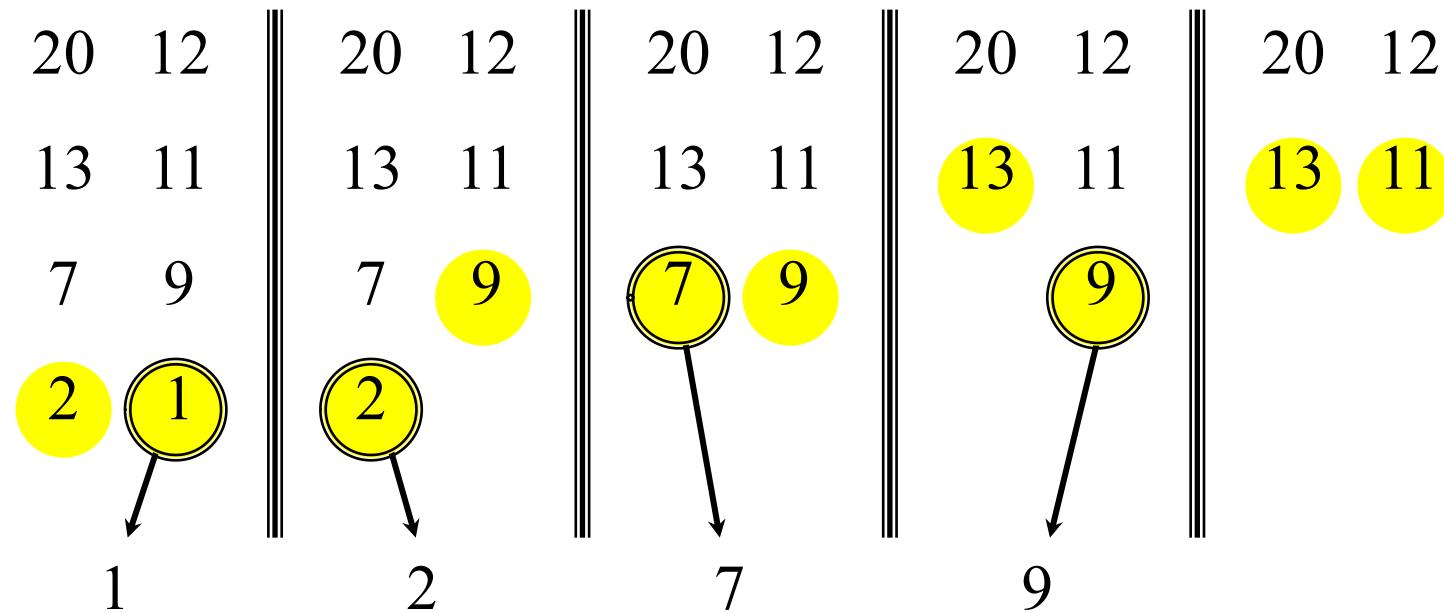


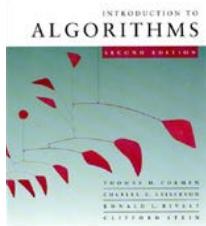
Merging two sorted arrays



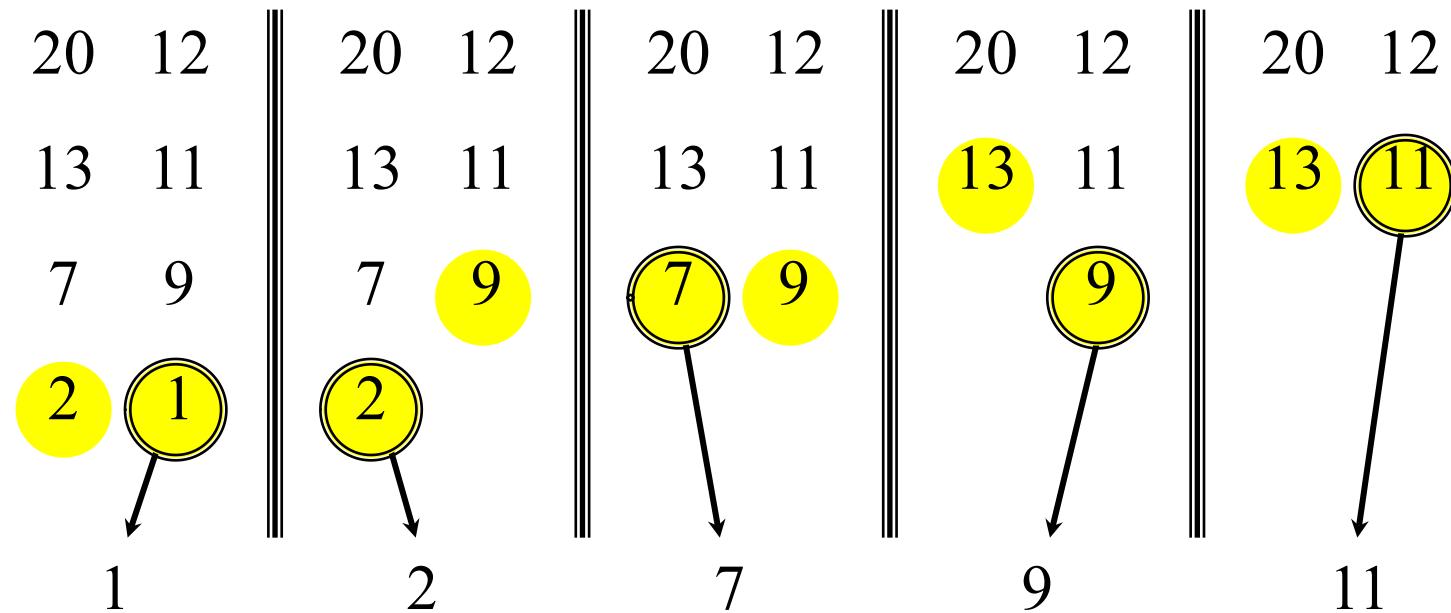


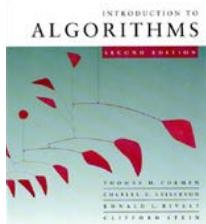
Merging two sorted arrays



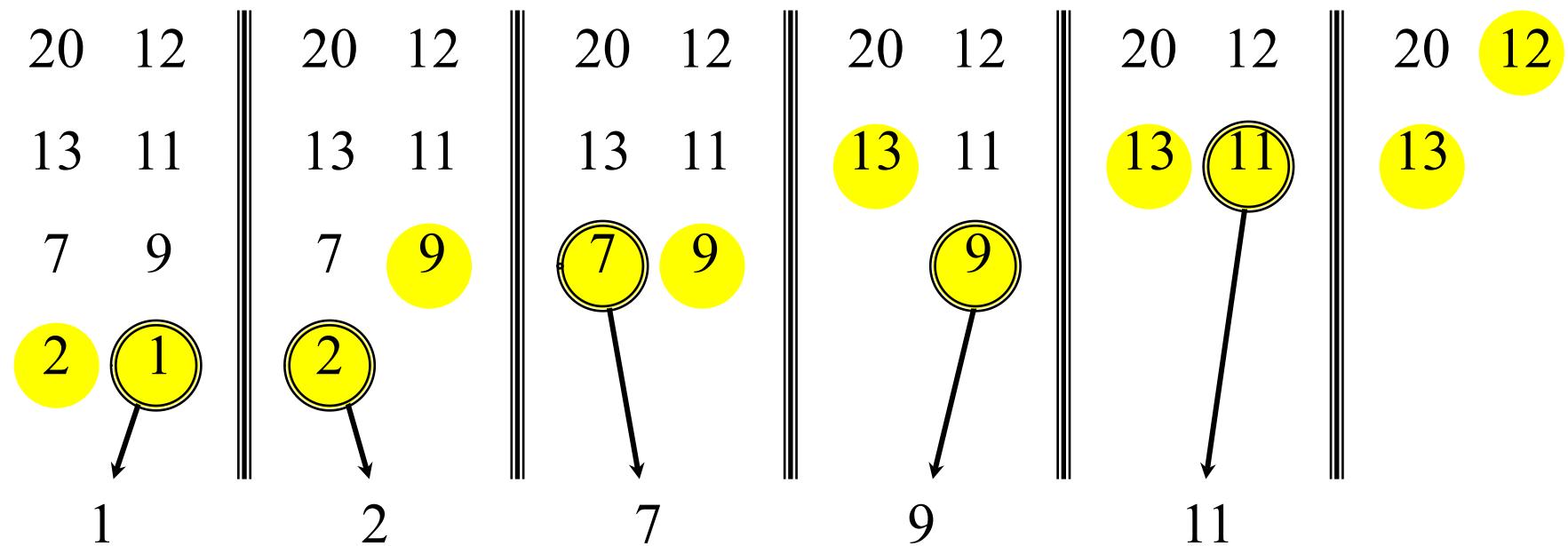


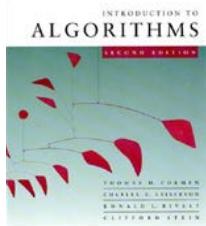
Merging two sorted arrays



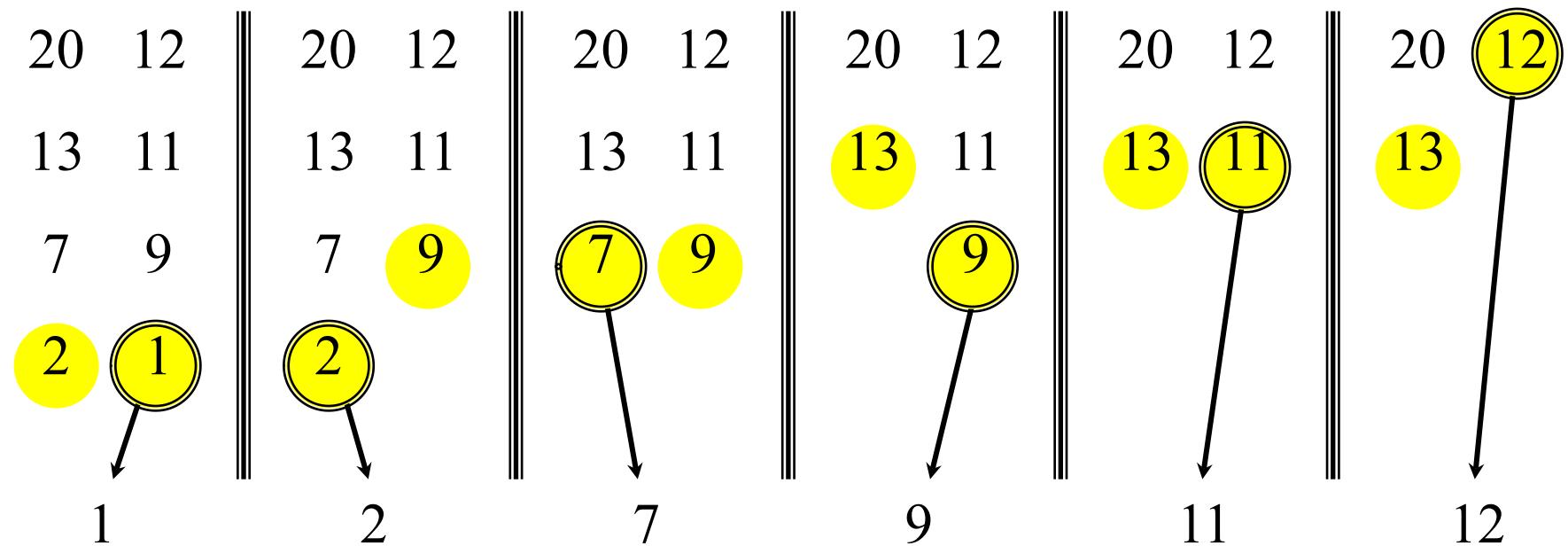


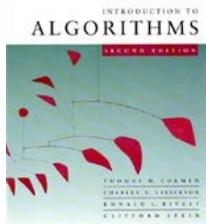
Merging two sorted arrays



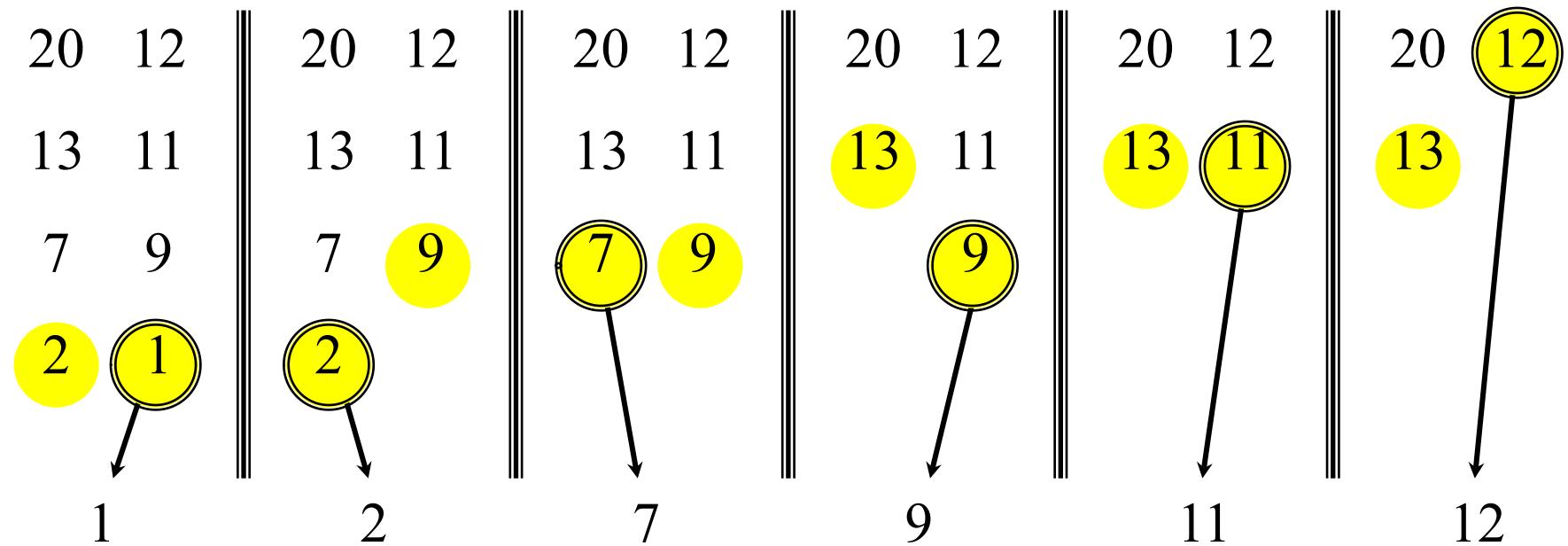


Merging two sorted arrays

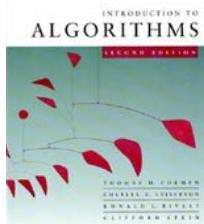




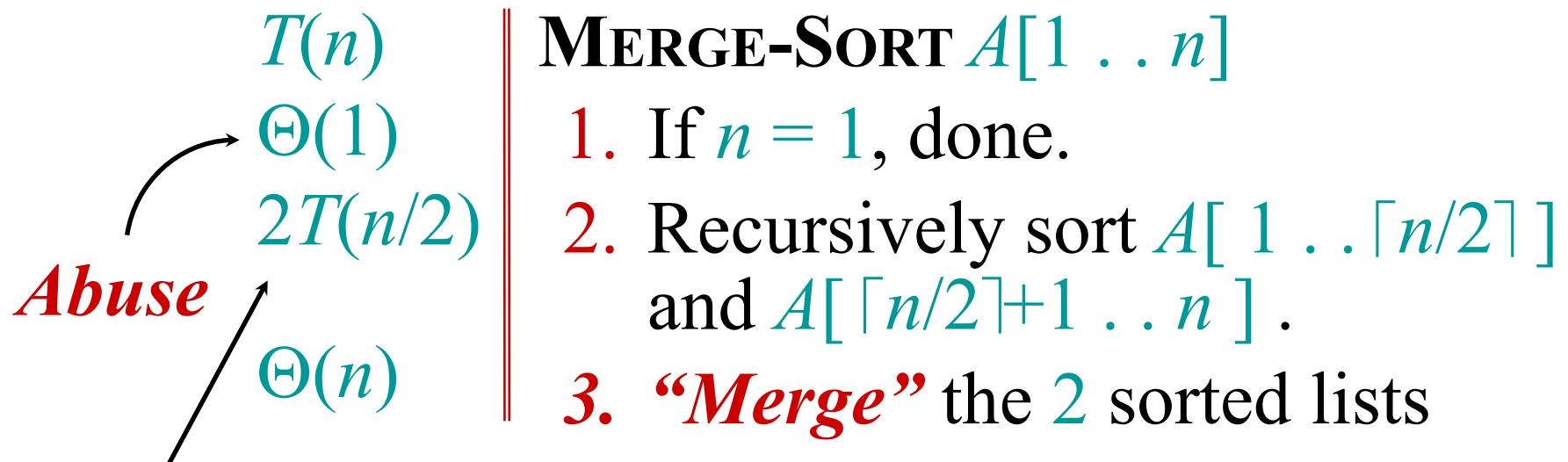
Merging two sorted arrays



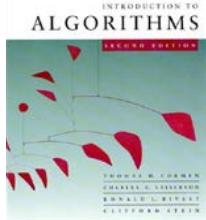
Time = $\Theta(n)$ to merge a total
of n elements (linear time).



Analyzing merge sort



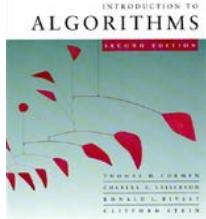
Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.



Recurrence for merge sort

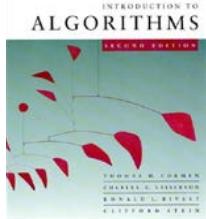
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

- 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

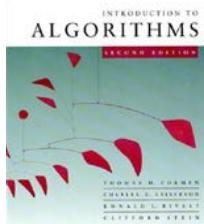
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

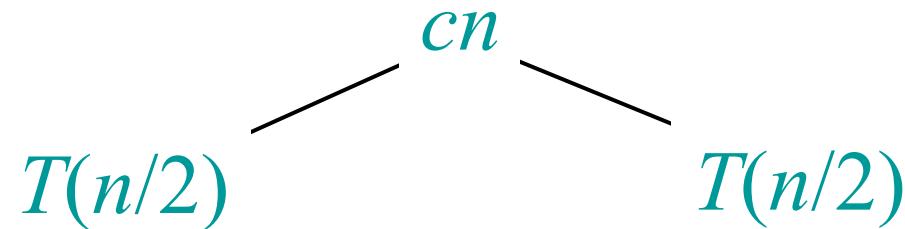
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

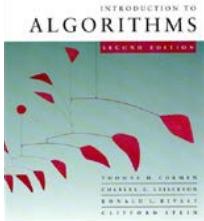
$$T(n)$$



Recursion tree

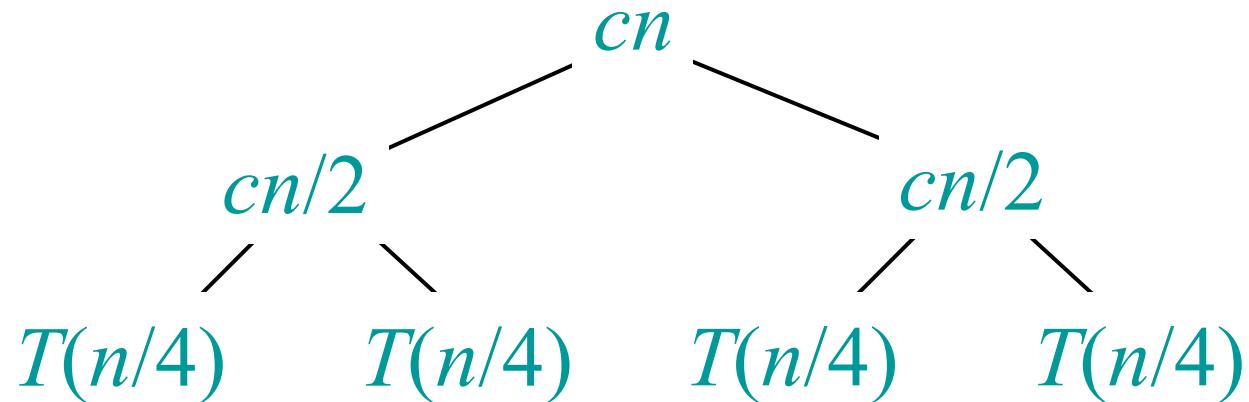
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

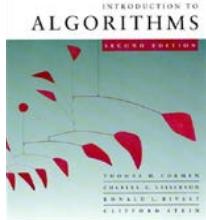




Recursion tree

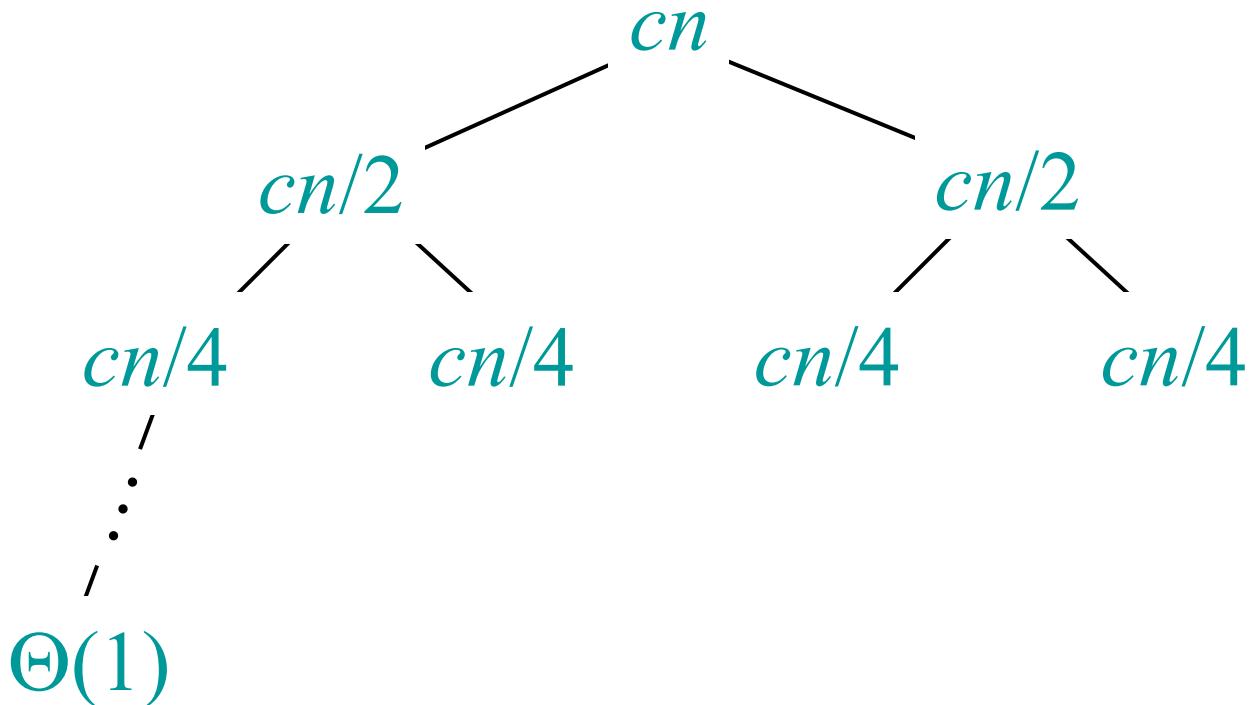
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

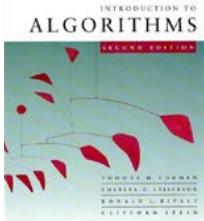




Recursion tree

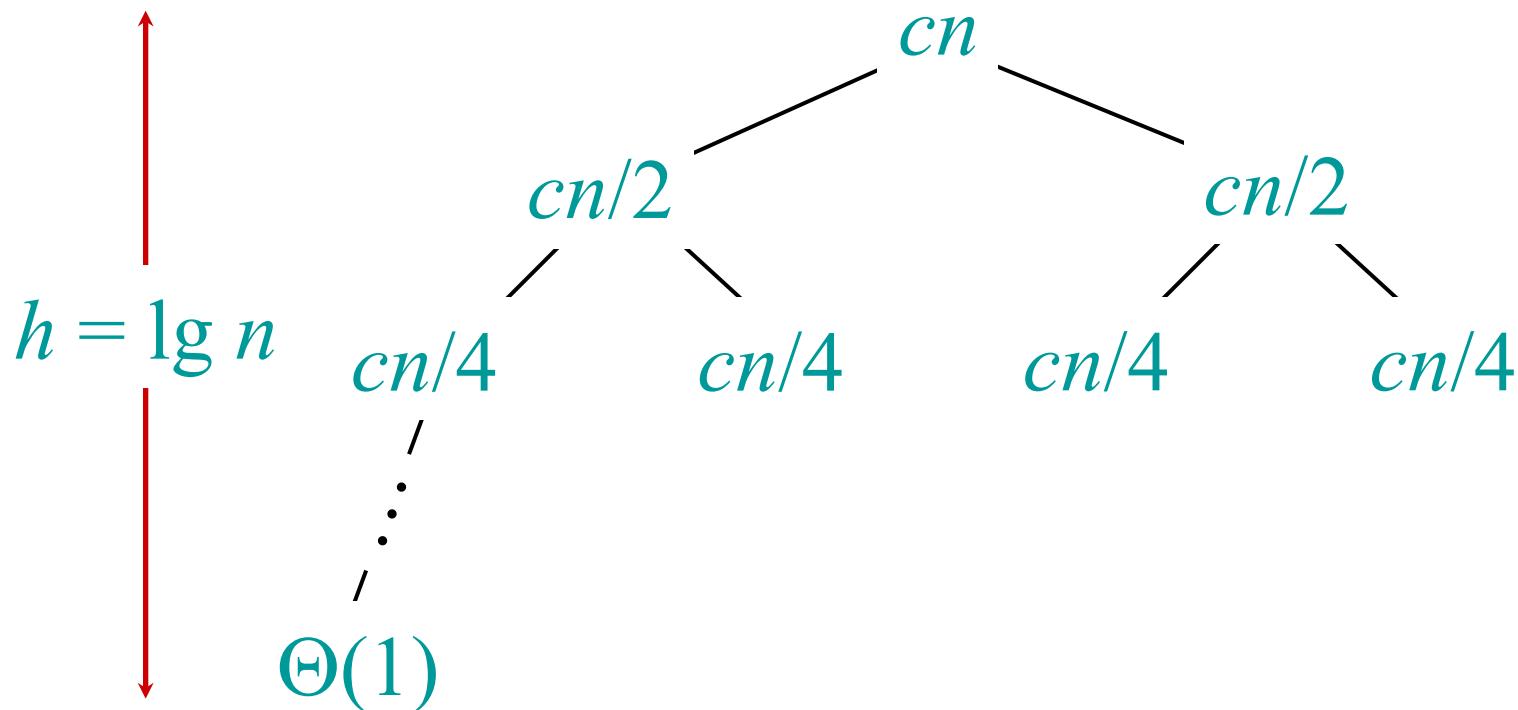
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

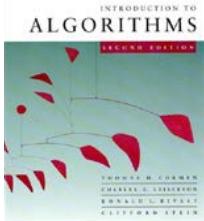




Recursion tree

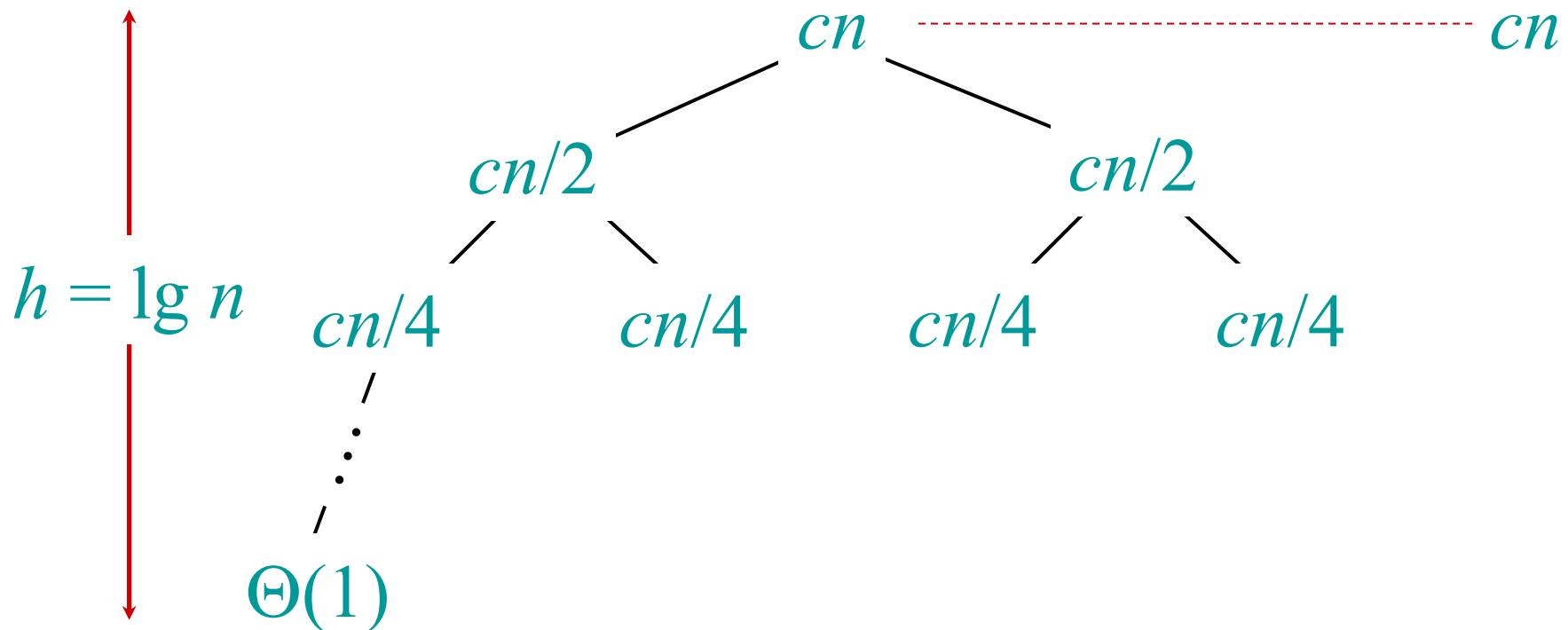
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

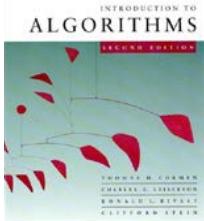




Recursion tree

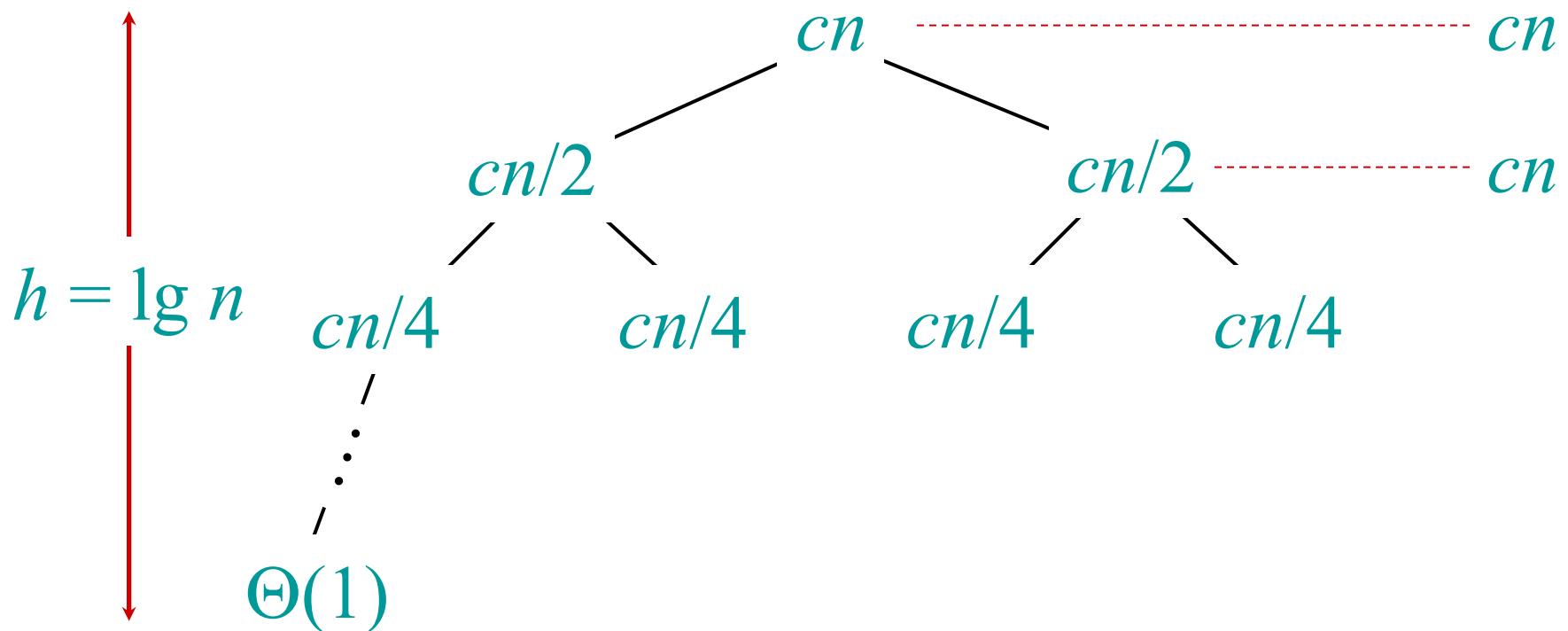
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

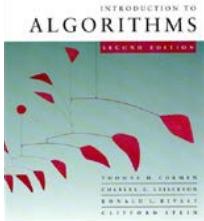




Recursion tree

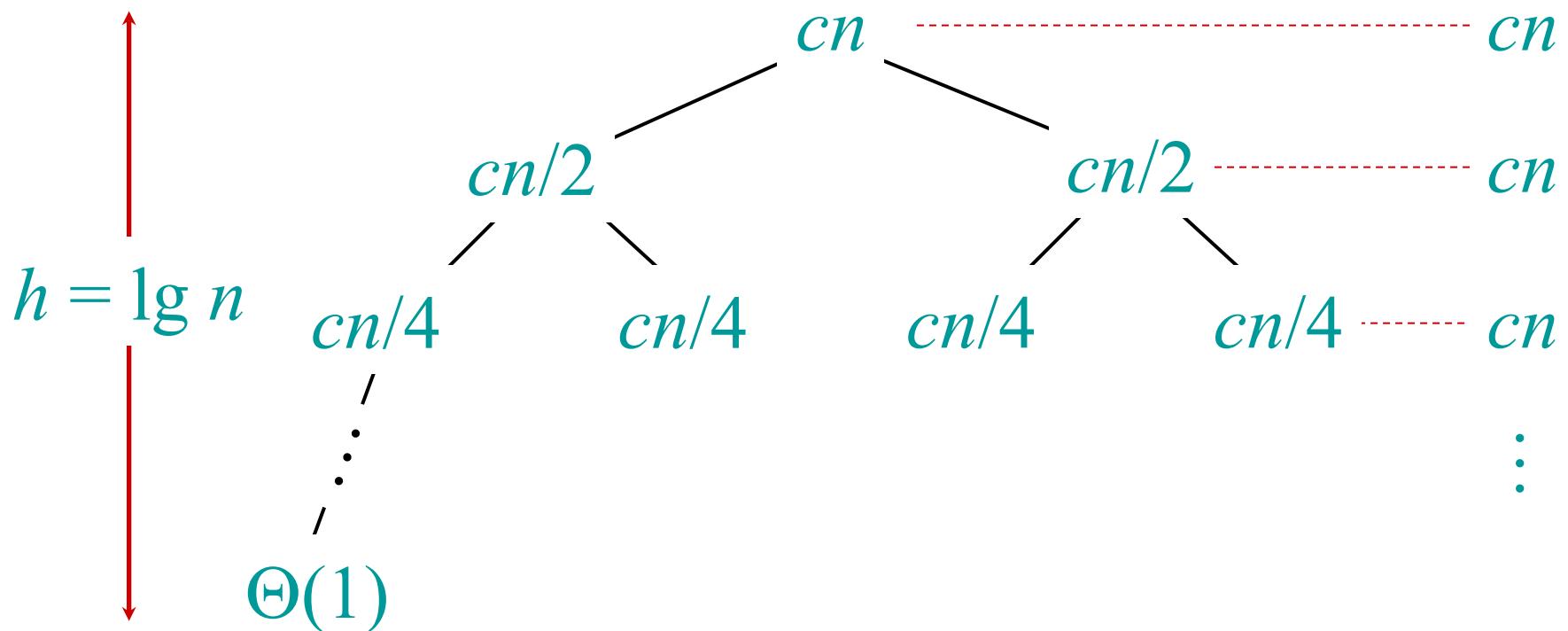
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

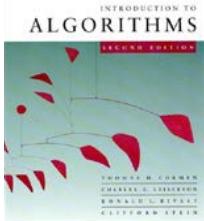




Recursion tree

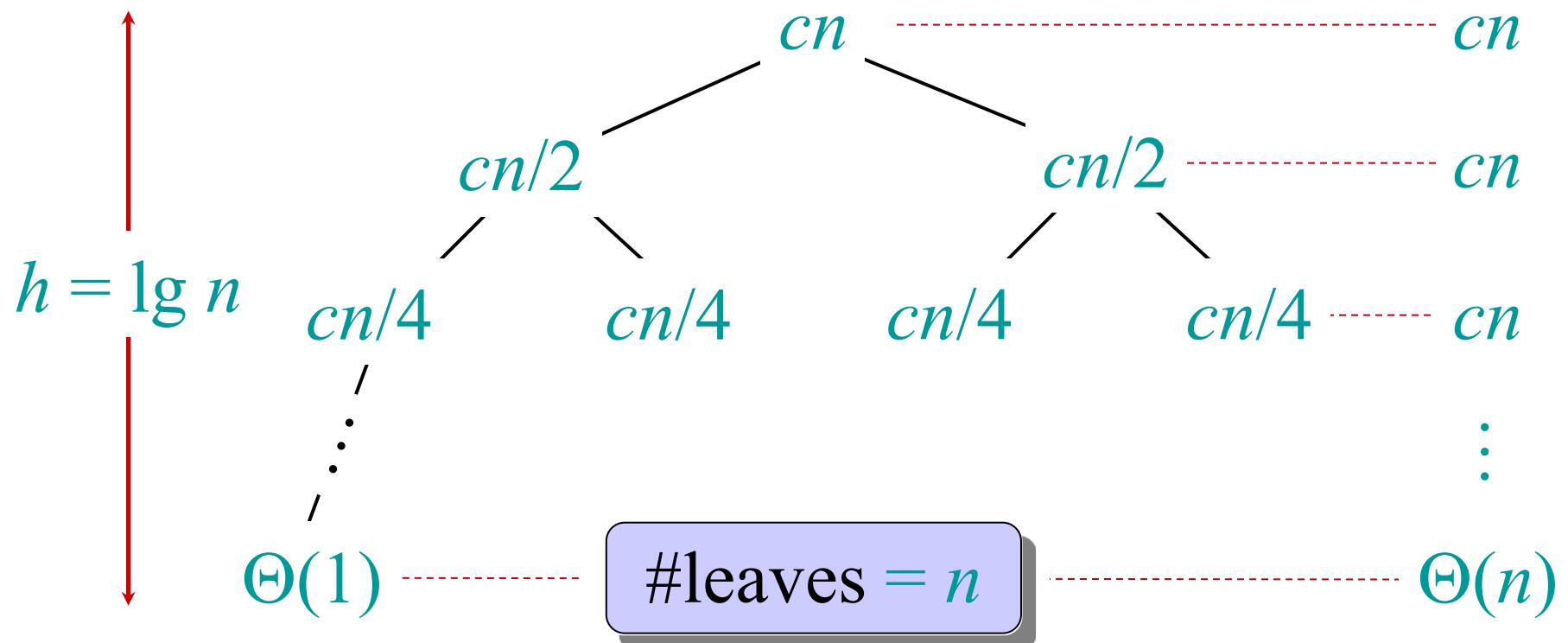
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

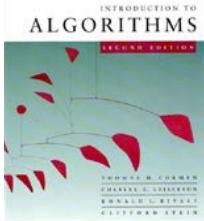




Recursion tree

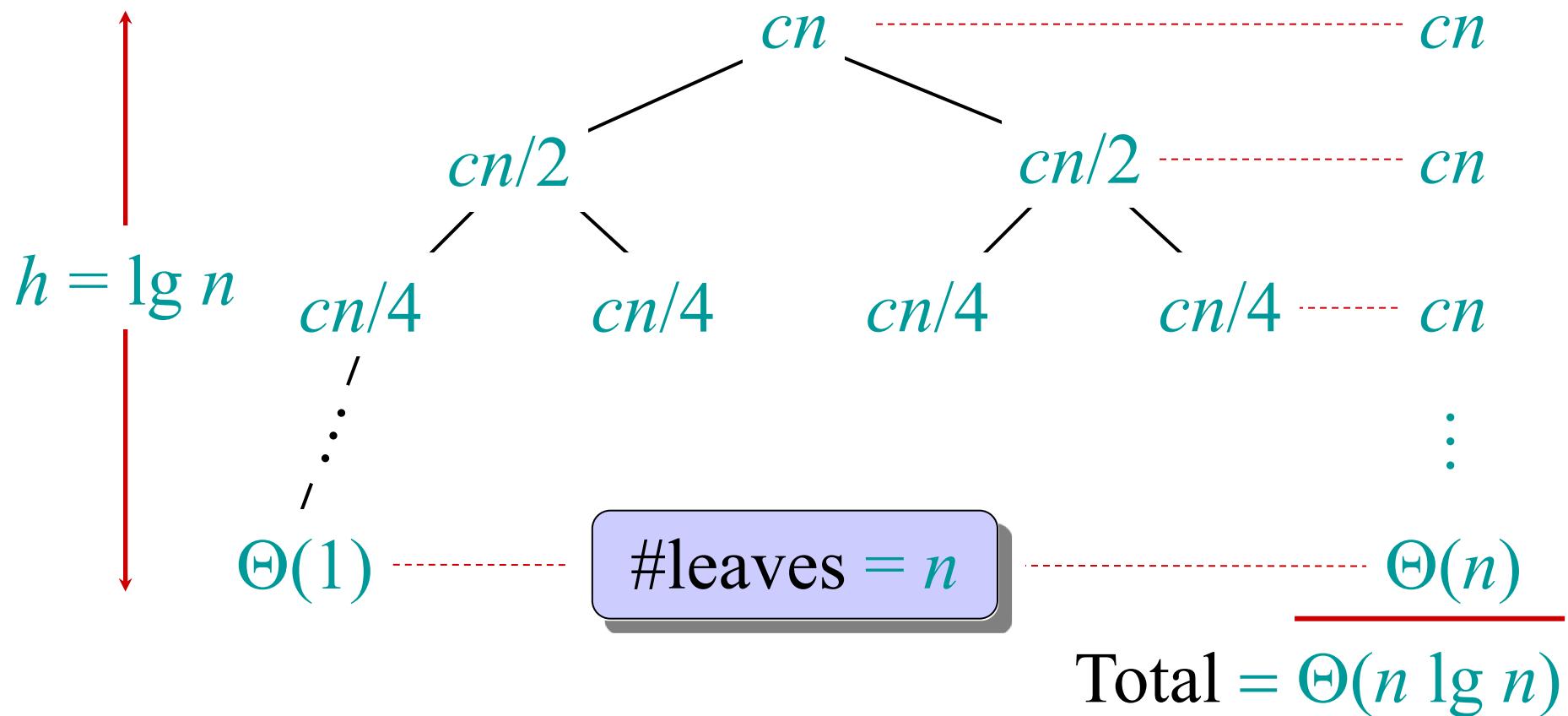
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

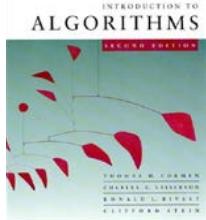




Recursion tree

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!

Topics covered Analyzing algorithms: recurrences

1. Introduction
2. Methods for solving recurrence relations
 - ▶ Substitution
 - ▶ Recursion-tree
 - ▶ Master theorem
3. Appendix : Linear Recurrence Relations

Introduction : Recurrences

1. Recurrences go hand in hand with the divide-and-conquer (DC) paradigm because they give us a natural way to characterize the running time of DC algorithms.
2. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
3. Recurrences have 2 types of terms : The *recursive* term(s) and the *non-recursive* terms.
 - ▶ The *recursive* terms refer to the recurrence relation.
 - ▶ The *non-recursive* terms refer to the time in one execution of the recursive function.
4. Example : The worst-case running time of the Merge-Sort procedure is given by a recurrence form :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

A few relations to remember

$$\log_a(xy) = \log_a x + \log_a y$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$\log_a(x^r) = r \log_a x$$

$$\log_a a = 1; \log_a 1 = 0$$

$$\log_a \frac{1}{x} = -\log_a x$$

$$\log_a x = \frac{\log x}{\log a}$$

$$x^{\frac{a}{b}} = \sqrt[b]{x^a}$$

$x^{1/b} = z$ means $z^b = x$; $x^{a/b} = z$ means $z^b = x^a$

Recurrence for Factorial

```
int factorial(int n)
if (n==1)
    return 1;
else
    return n*factorial(n-1);
```

Running-time of factorial, $T(n)$, is given by the following recurrence relation :

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ T(n - 1) + 1 & \text{otherwise} \end{cases}$$

Mapping recurrences to algorithms

How do these terms relate to algorithms ?

- ▶ Recursive terms refer to recursive calls.
- ▶ Non-recursive terms computation in the subroutine, including any splitting or combining of data.

```
int factorial(int n)
    if (n==1)
        return 1;
    else
        return n*factorial(n-1);
```

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ T(n - 1) + 1 & \text{otherwise} \end{cases}$$

Recurrence for Binary Search

```
procedure BinarySearch( $L[i..j]$ ,  $x$ )
    if  $i = j$  then return  $i$ 
     $k = \frac{(i+j)}{2}$ ;
    if  $x \leq L[k]$  then
        return BinarySearch( $L[i..k]$ ,  $x$ )
    else
        return BinarySearch( $L[k+1..j]$ ,  $x$ )
```

- ▶ To solve an instance of binary search, we need to do a compare operation, followed by an instance of binary search of half the size.
- ▶ Thus, the recurrence for the (worst-case) run-time of this algorithm is

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$$

Other examples of recurrences

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n - 1) + 1 & \text{if } n > 1 \end{cases}$$

Solution : $T(n) = n$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n \geq 1 \end{cases}$$

Solution : $T(n) = n \lg n + n$

$$T(n) = \begin{cases} 0 & \text{if } n = 2 \\ T(\sqrt{n}) + n & \text{if } n > 2 \end{cases}$$

Solution : $T(n) = \lg \lg n$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases}$$

Solution : $T(n) = \Theta(n \lg n)$

How to analyze recursive functions

Identify the number of elementary operations perform in each call of the recursive function. Usually elementary operations are :

1. operations executed in each instantiation of the r.f. to divide its input in several recursive calls and
2. operations executed in each instantiation of the r.f. to combine the solutions of the recursive calls.

Write the recurrence relation (rather than the summation) expressing the number of basic operations executed based on n , the input.

Find the closed form of the recurrence relation.

The substitution method

The substitution method consists of two steps :

1. Guess the form of the solution.
2. Use mathematical induction to show that the guess is correct.

It can be used to obtain either upper ($O()$) or lower bounds ($\Omega()$) on a recurrence.

A good guess is vital when applying this method. If the initial guess is wrong, the guess needs to be adjusted later.

Example 1

Solve the recurrence $T(n) = T(n - 1) + 2n - 1$, $T(0) = 0$

Initial guess. Use the method of forward substitution :

n	0	1	2	3	4	5
$T(n)$	0	1	4	9	16	25

Guess is $T(n) = n^2$

Proof by induction :

Base case $n = 0$: $T(0) = 0 = 0^2$

Induction step : Assume the inductive hypothesis is true for $n = n - 1$.

We have

$$\begin{aligned}T(n) &= T(n - 1) + 2n - 1 \\&= (n - 1)^2 + 2n - 1 \quad (\text{Induction hypothesis}) \\&= n^2 - 2n + 1 + 2n - 1 \\&= n^2\end{aligned}$$

Example 2

Solve the recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n$, $T(0) = 0$

Guess using the method of forward substitution :

n	0	1	2	3	4	5	8	16	32	64
$T(n)$	0	1	3	4	7	8	15	31	63	127

Guess : $T(n) \leq 2n$

Base case $n = 0$: $T(n) = 0 \leq 2 \times 0$

Induction step : Assume the inductive hypothesis is true for some $n = \lfloor \frac{n}{2} \rfloor$. We have

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + n \leq 2\lfloor \frac{n}{2} \rfloor + n \leq 2(n/2) + n = 2n, \text{ i.e.}$$

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

Example 3

Solve $T(n) = 2T(n/2) + n$

Guess $T(n) \leq cn \log n$ for some constant c (that is,
 $T(n) = O(n \log n)$)

Proof :

Base case : $T(1) = 1 = n \log n + n$ (note : we need to show that our guess holds for some base case (not necessarily $n = 1$, some small n is ok)).

Induction step : Assume the inductive hypothesis holds for $n/2$ (n is a power of 2) : $T(n/2) \leq c \frac{n}{2} \log \frac{n}{2}$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2(c \frac{n}{2} \log \frac{n}{2}) + n \\ &= cn \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \end{aligned}$$

Which is true if $c \geq 1$

Exercise

Assume that the running time $T(n)$ satisfies the recurrence relation

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n.$$

Show that $T(n) \in \Theta(n \lg n)$.

Remark : To have a good guess, we could use the recursion-tree or changing variables to reduce to the linear recurrence that could be solved (see [Appendix](#)).

The recursion tree

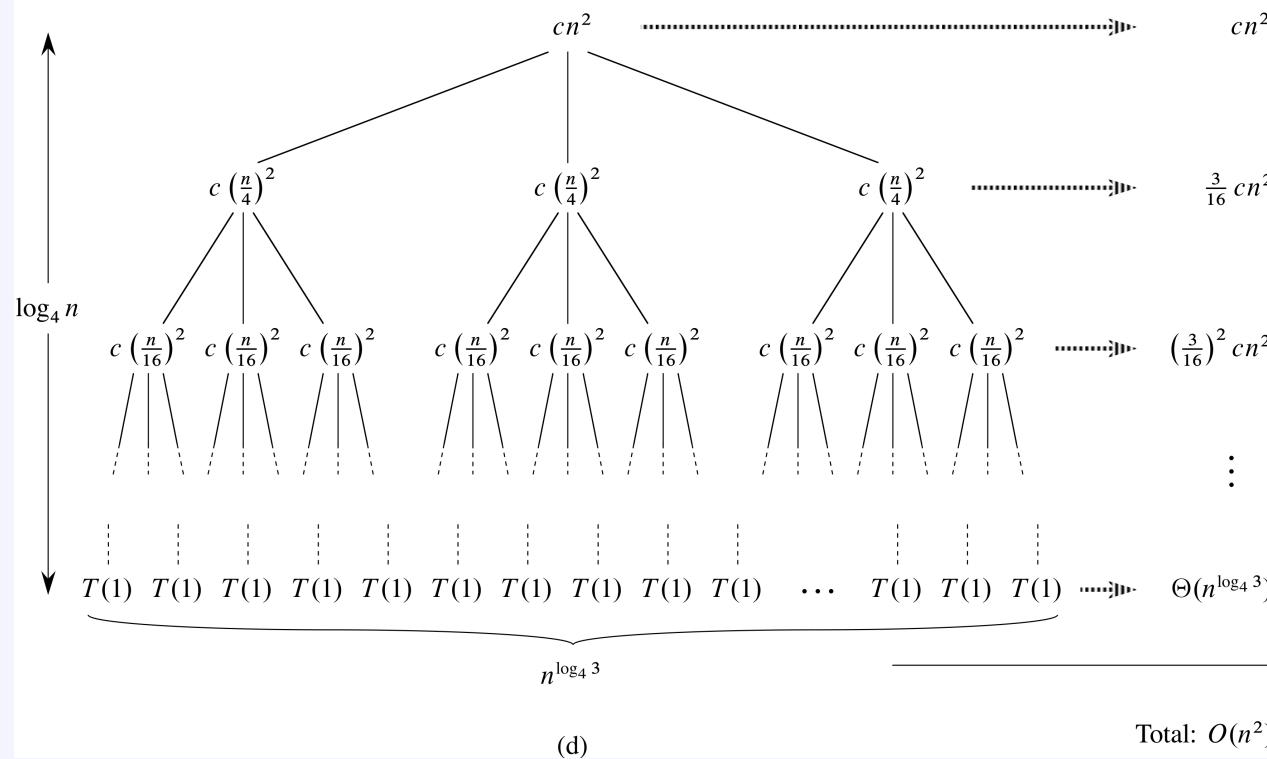
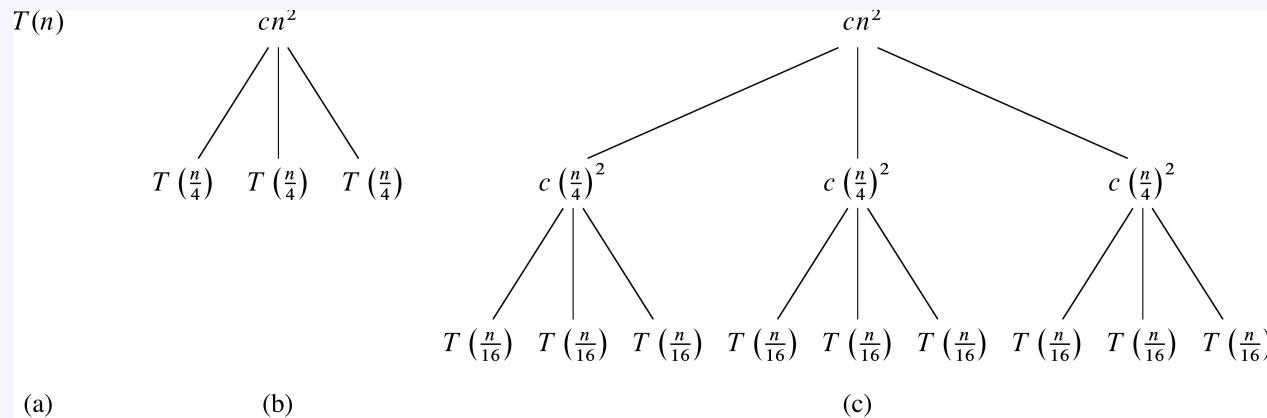
To analyze the recurrence by using the recursion tree, we will

- ▶ draw a recursion tree with cost of single call in each node.
- ▶ find the running time at each level of the tree by summing the running time of each call at that level.
- ▶ find the number of levels
- ▶ Last, the running time is sum of costs in all nodes.

Example of a recurrence relation :

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2.$$

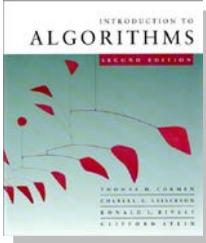
The recursion tree



The recursion tree

Remark : The number of levels depends by how much subproblem sizes decrease. In this example, subproblems decrease by a factor of 4 at each new level, the level where $n = 1$ is where $n/4^i = 1$, i.e. when $i = \log_4 n$. Therefore, the number of terms in the above summation = number of levels in the tree = $\log_4 n$. More precisely,

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2cn^2 + \cdots + \left(\frac{n}{16}\right)^{\log_4(n-1)}cn^2 + cn^{\log_4 3} \\ &= cn^2 \left[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4(n-1)} \right] + cn^{\log_4 3} \\ &= O(n^2) \quad (\text{as geometric series and maximum rule}) \end{aligned}$$



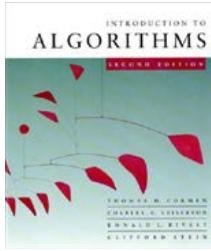
Master theorem (reprise)

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

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$
 $\Rightarrow T(n) = \Theta(n^{\log_b a})$.

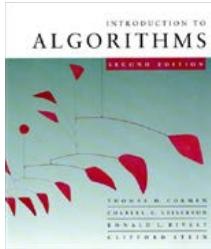
CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
 $\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition
 $\Rightarrow T(n) = \Theta(f(n))$.



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

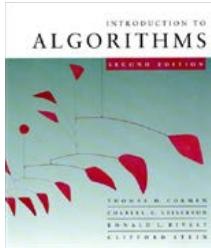
Quicksort an n -element array:

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



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

Key: *Linear-time partitioning subroutine.*

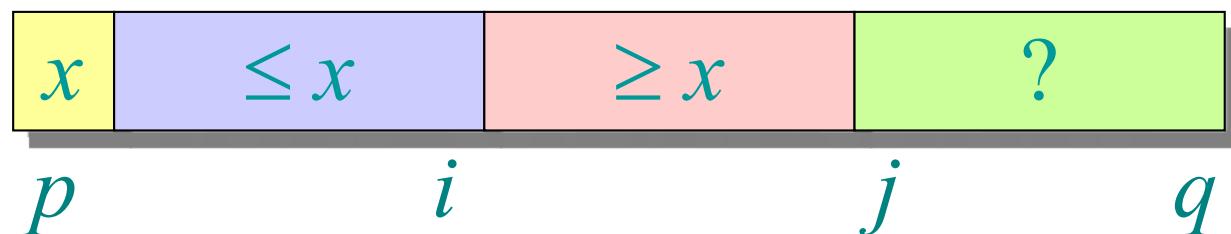


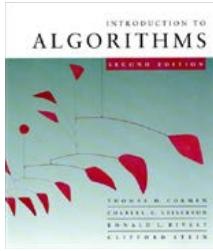
Partitioning subroutine

```
PARTITION( $A, p, q$ )
   $x \leftarrow A[p]$             $\triangleright A[p \dots q]$ 
   $i \leftarrow p$               $\triangleright \text{pivot} = A[p]$ 
  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  $i$ 
```

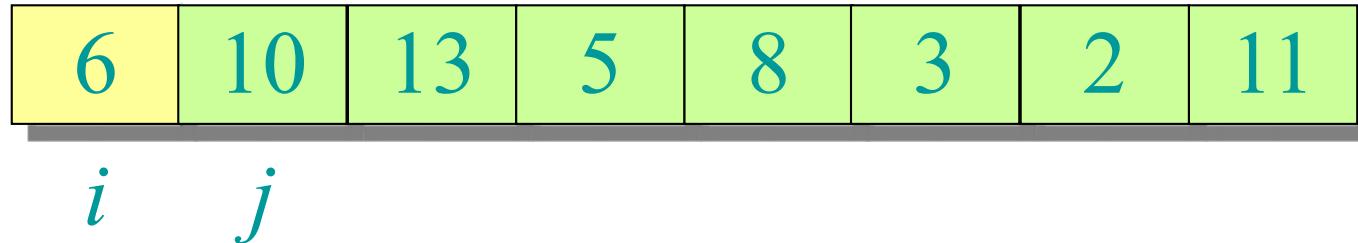
Running time
 $= O(n)$ for n elements.

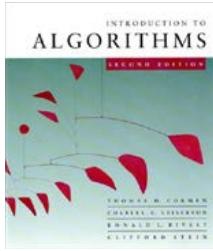
Invariant:





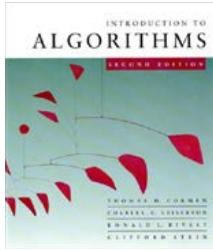
Example of partitioning



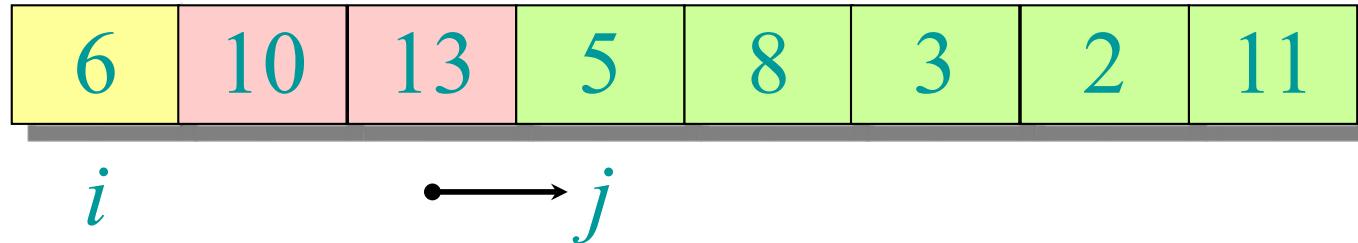


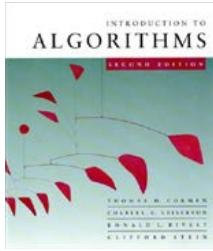
Example of partitioning



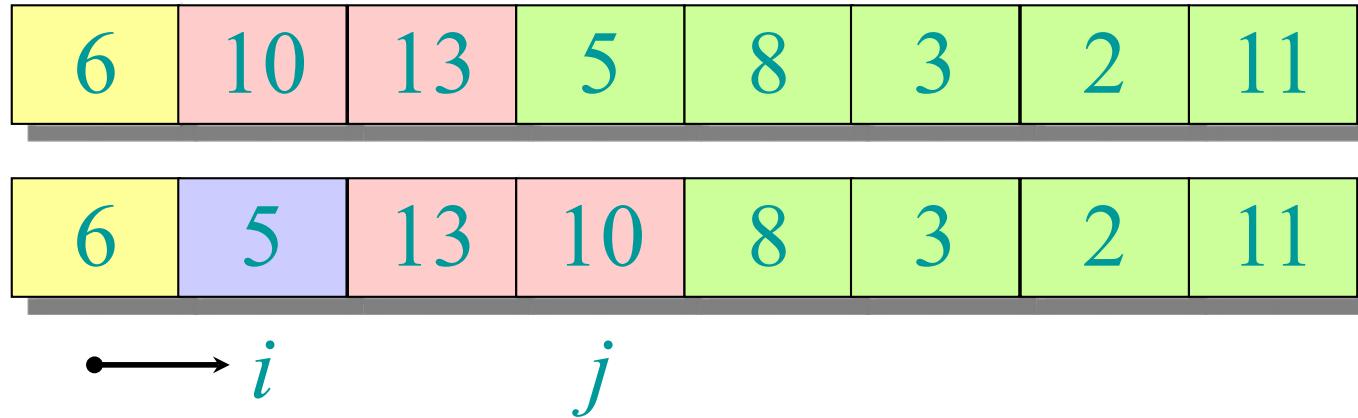


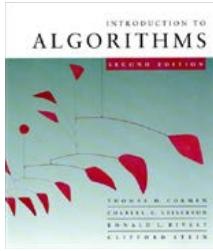
Example of partitioning



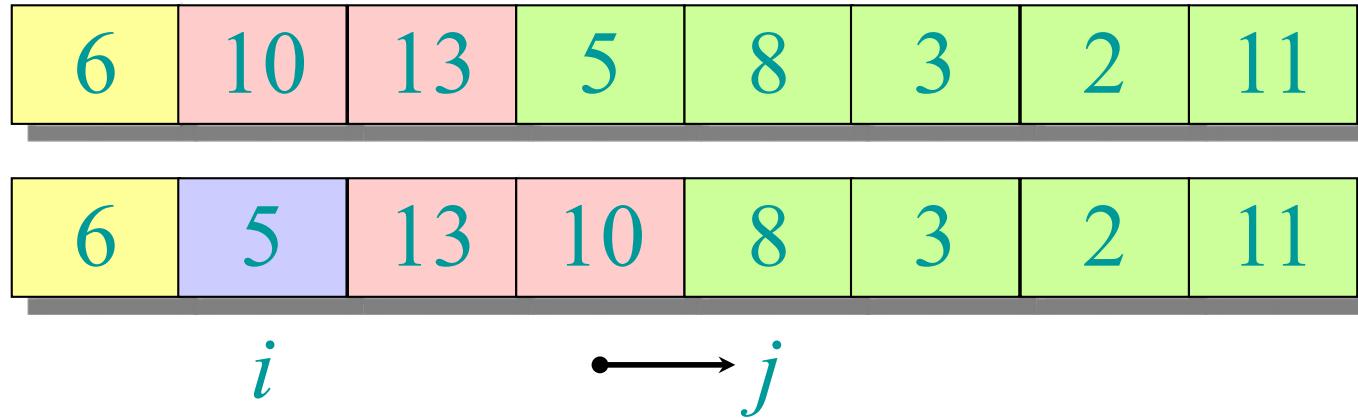


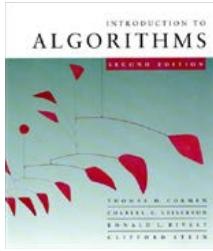
Example of partitioning



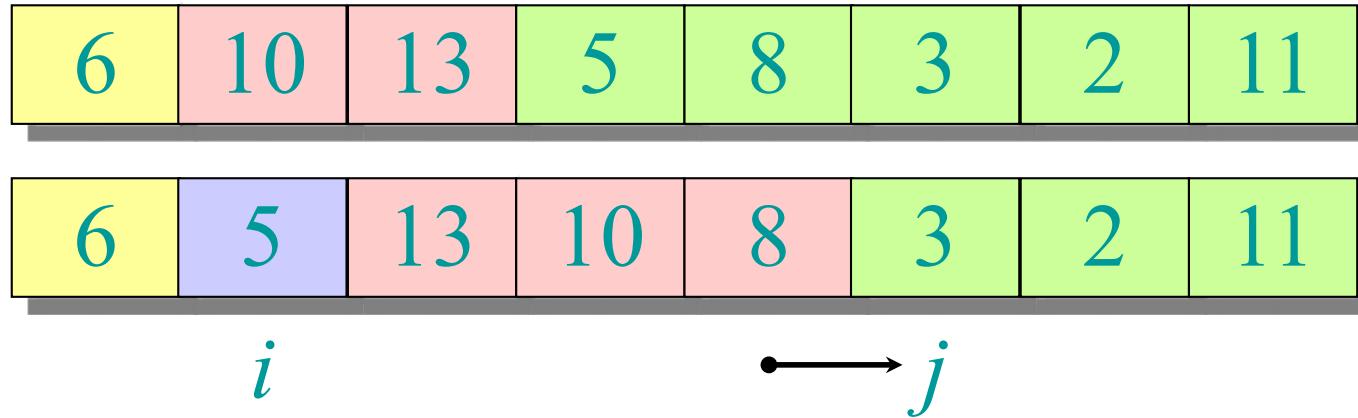


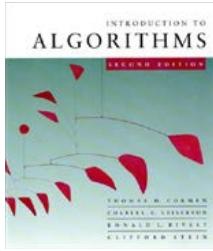
Example of partitioning



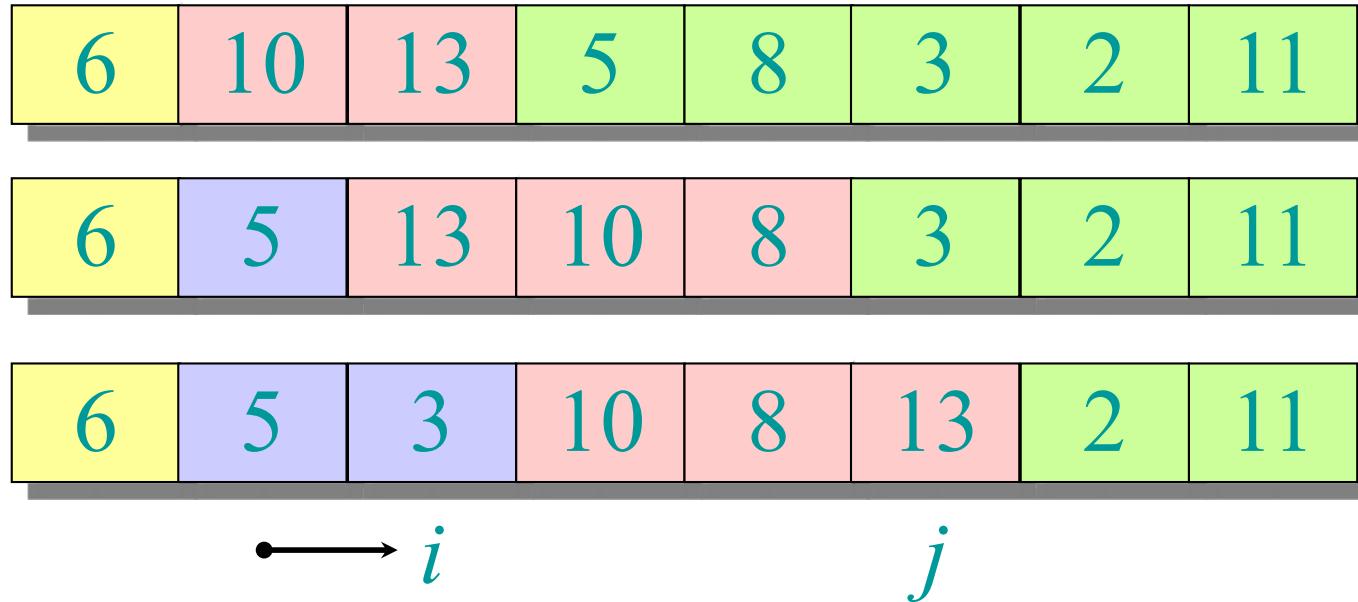


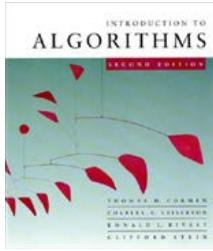
Example of partitioning



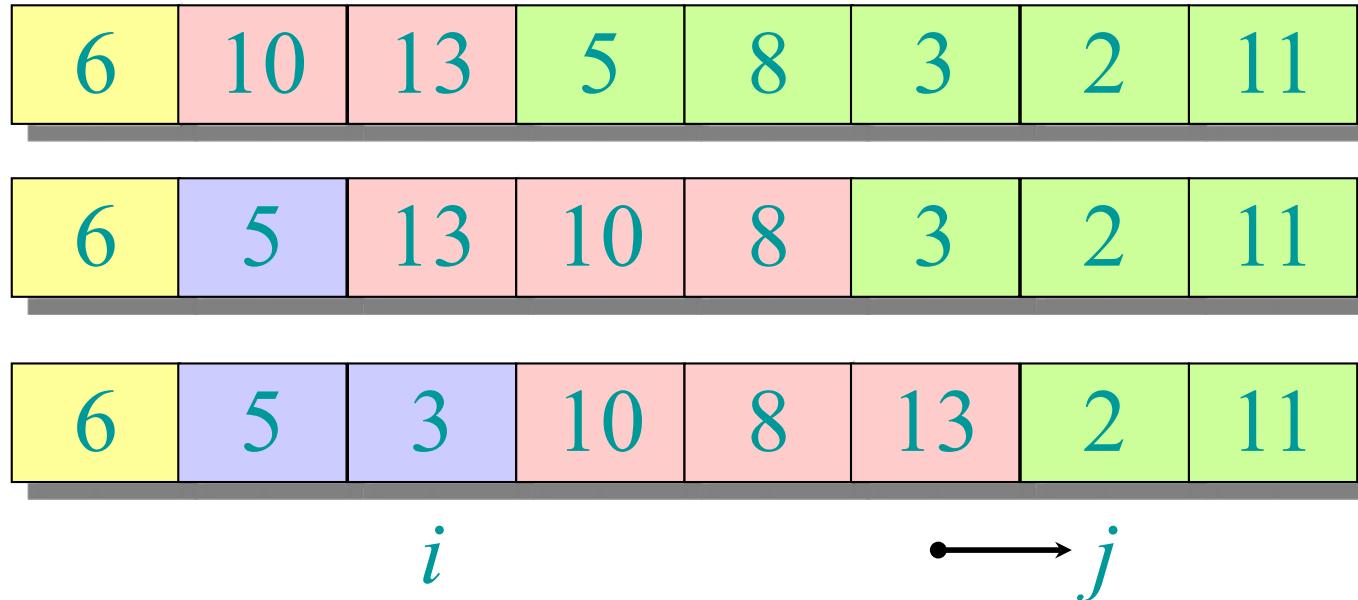


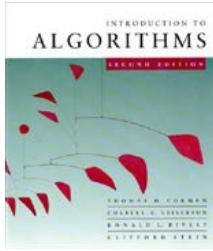
Example of partitioning



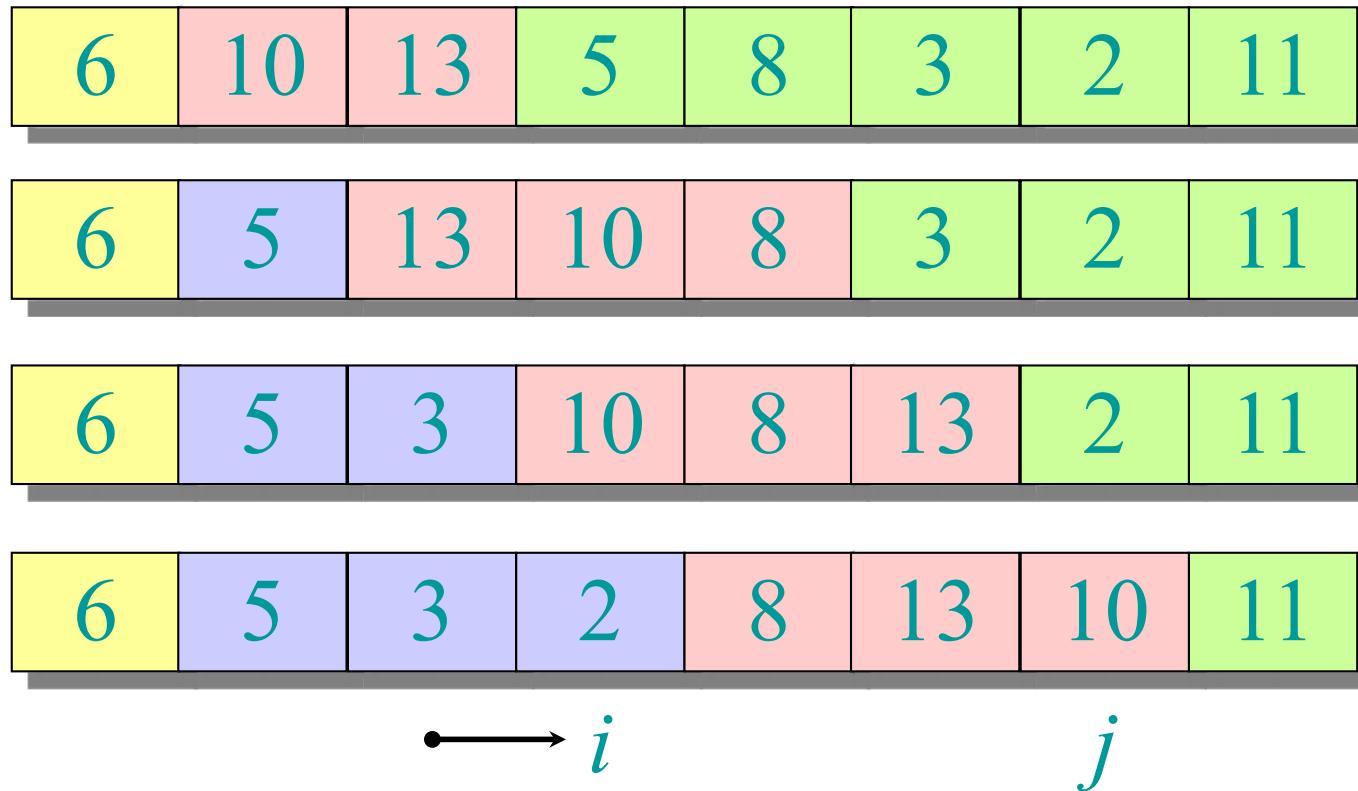


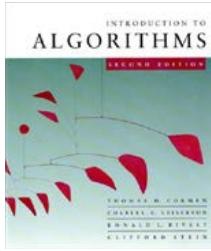
Example of partitioning



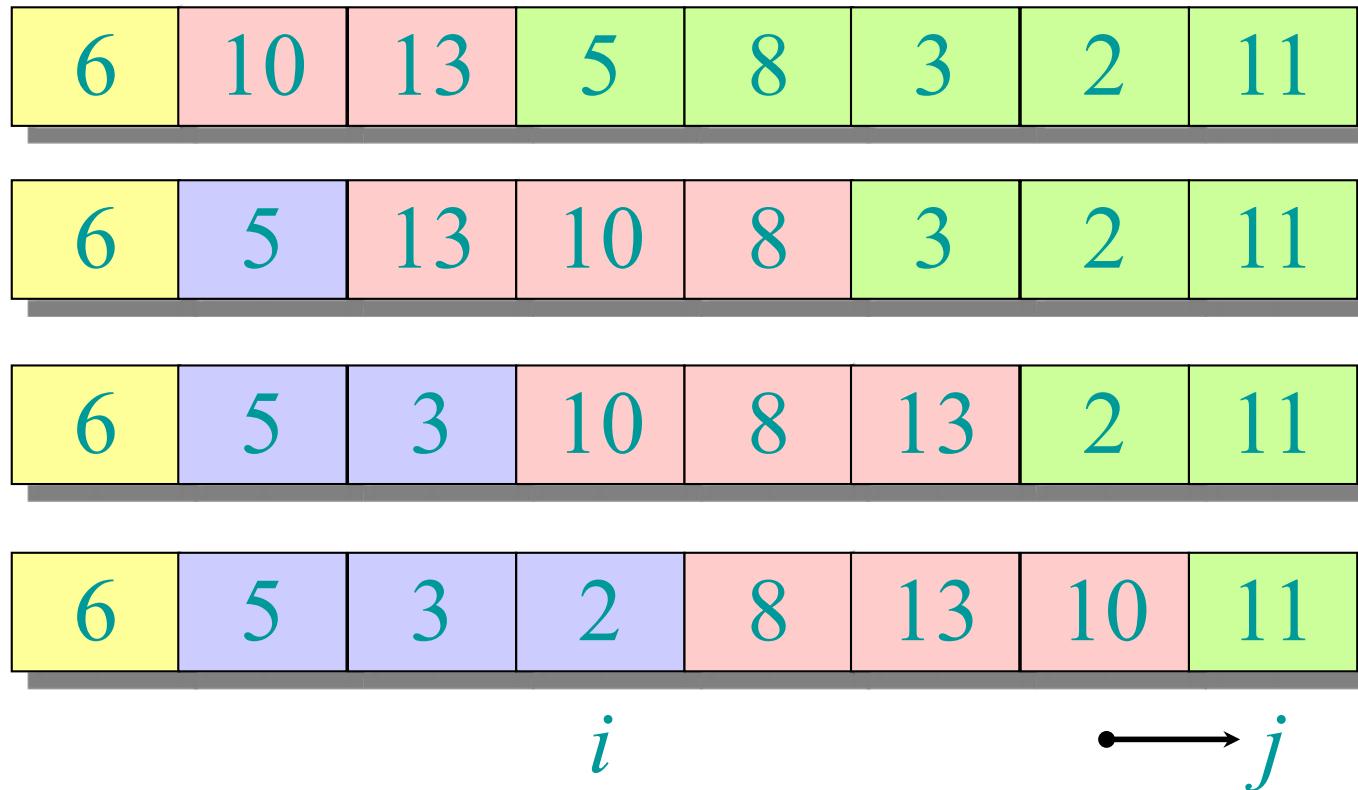


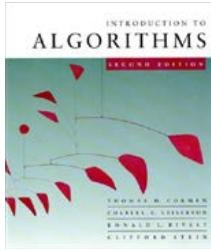
Example of partitioning



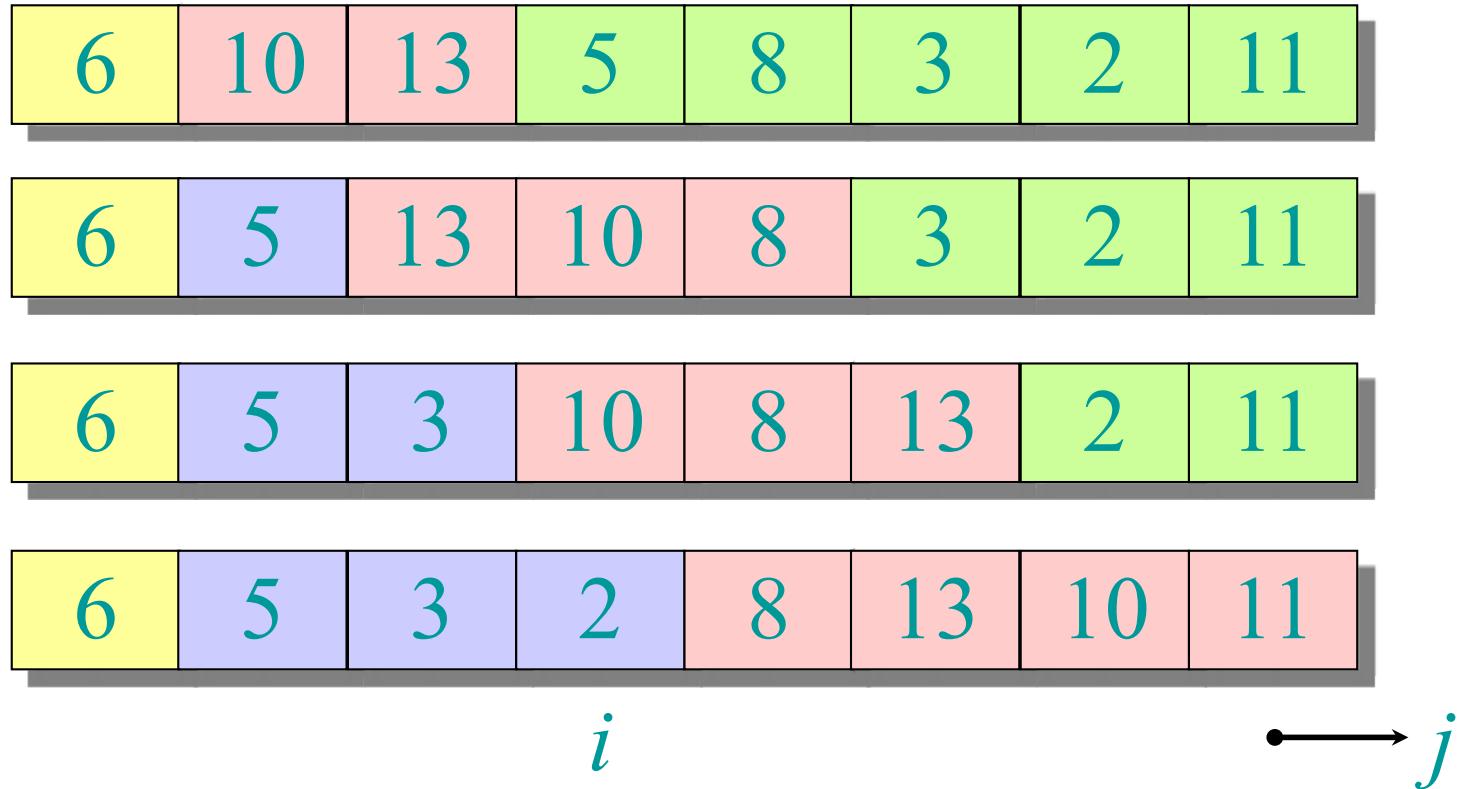


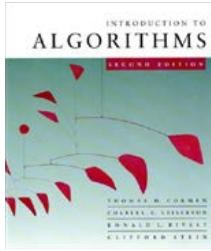
Example of partitioning



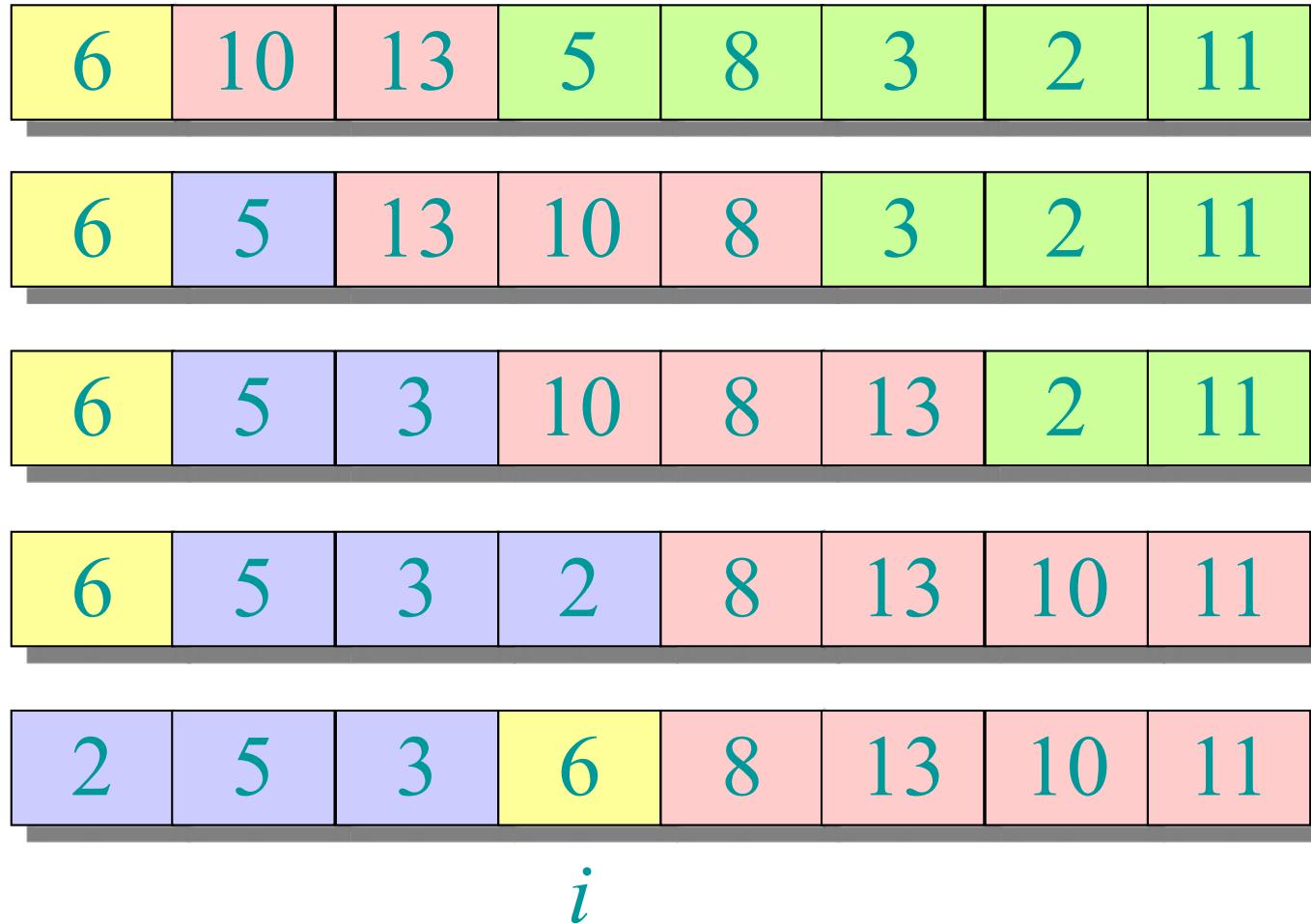


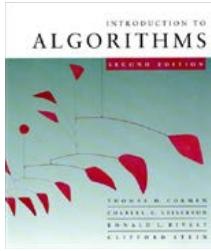
Example of partitioning





Example of partitioning





Pseudocode for quicksort

QUICKSORT(A, p, r)

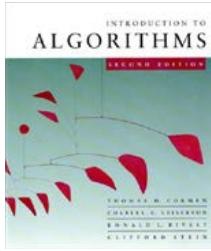
if $p < r$

then $q \leftarrow \text{PARTITION}(A, p, r)$

 QUICKSORT($A, p, q-1$)

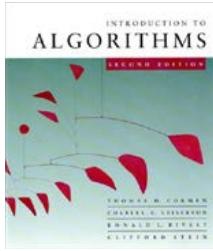
 QUICKSORT($A, q+1, r$)

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



Analysis of quicksort

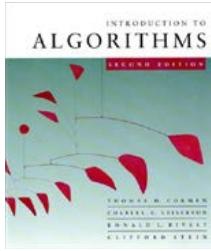
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.



Worst-case of quicksort

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

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



Best-case analysis

(For intuition only!)

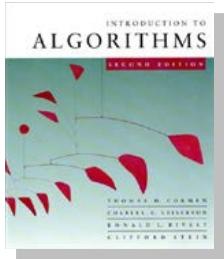
If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \quad (\text{same as merge sort}) \end{aligned}$$

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

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

What is the solution to this recurrence?

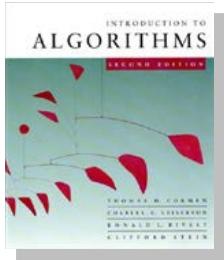


Matrix multiplication

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

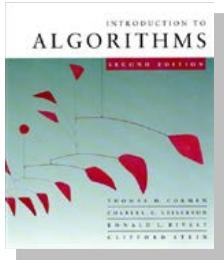
$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots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

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



Standard algorithm

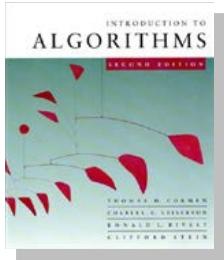
```
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $c_{ij} \leftarrow 0$ 
            for  $k \leftarrow 1$  to  $n$ 
                do  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```



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:

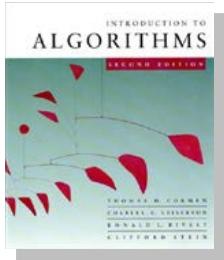
$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

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

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



Divide-and-conquer algorithm

IDEA:

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

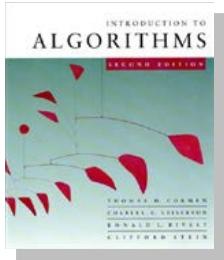
$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

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

recursive

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

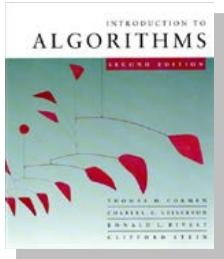


Analysis of D&C algorithm

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

submatrices ↗
 ↓
 submatrix size
 ↗
 work adding
 submatrices

The diagram illustrates the recurrence relation for the D&C algorithm. The equation $T(n) = 8T(n/2) + \Theta(n^2)$ is shown with two annotations. An arrow labeled "# submatrices" points to the term $8T(n/2)$. Another arrow labeled "submatrix size" points to the term $\Theta(n^2)$. A third annotation, "work adding submatrices", is positioned to the right of the equation, with an arrow pointing towards the $\Theta(n^2)$ term.



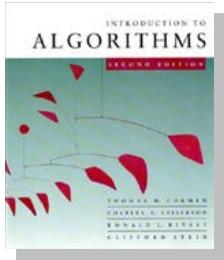
Analysis of D&C algorithm

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

submatrices ↗
 ↓
 submatrix size
 ↗
 work adding
 submatrices

The diagram illustrates the recurrence relation for the D&C algorithm. The equation $T(n) = 8T(n/2) + \Theta(n^2)$ is shown at the top. Two arrows point from the text below to specific parts of the equation: one arrow from "# submatrices" points to the term $8T(n/2)$, and another arrow from "submatrix size" points to the term $\Theta(n^2)$. A third arrow from "work adding submatrices" points to the plus sign between the two terms.

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



Analysis of D&C algorithm

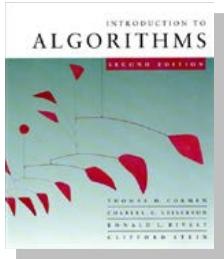
$$T(n) = 8T(n/2) + \Theta(n^2)$$

submatrices ↗
 ↓
 submatrix size
 ↗
 work adding
 submatrices

The diagram illustrates the recurrence relation for the D&C algorithm. The equation $T(n) = 8T(n/2) + \Theta(n^2)$ is shown at the top. Two arrows point from the text below to specific parts of the equation: one arrow from "# submatrices" points to the term $8T(n/2)$, and another arrow from "submatrix size" points to the term $\Theta(n^2)$. A third arrow from "work adding submatrices" points to the plus sign between the two terms.

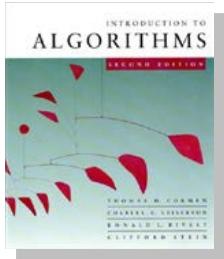
$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.



Strassen's idea

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



Strassen's idea

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

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

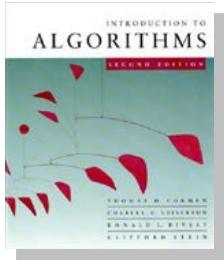
$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$



Strassen's idea

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

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

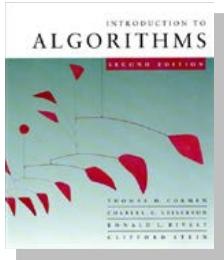
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$



Strassen's idea

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

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

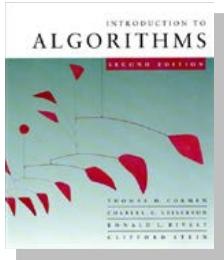
$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$

7 mults, 18 adds/subs.

Note: No reliance on commutativity of mult!



Strassen's idea

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

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

$$= (a + d)(e + h)$$

$$+ d(g - e) - (a + b)h$$

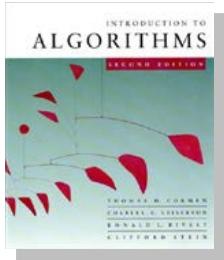
$$+ (b - d)(g + h)$$

$$= ae + ah + de + dh$$

$$+ dg - de - ah - bh$$

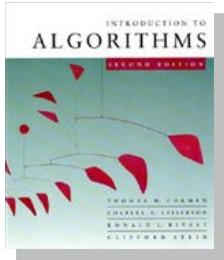
$$+ bg + bh - dg - dh$$

$$= ae + bg$$



Strassen's algorithm

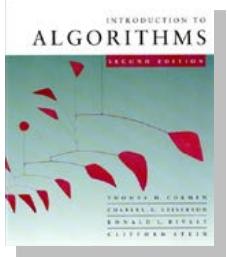
1. **Divide:** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. **Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
3. **Combine:** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.



Strassen's algorithm

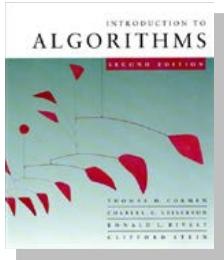
1. **Divide:** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. **Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
3. **Combine:** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

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



Analysis of Strassen

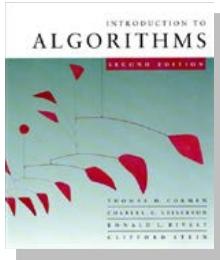
$$T(n) = 7 T(n/2) + \Theta(n^2)$$



Analysis of Strassen

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

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

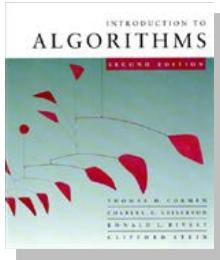


Analysis of Strassen

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

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

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 $n \geq 32$ or so.



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 $n \geq 32$ or so.

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

Algorithms and Data Structures

Lecture notes: Binary Search Trees, Cormen

chapters 12

Lecturer: Michel Toulouse

Vietnamese-German University
michel.toulouse@vgu.edu.vn

2 avril 2016

Binary Search Trees

A set of elementary data structures (elements) link together by pointers such to form a binary tree data structure as a whole.

Each element has :

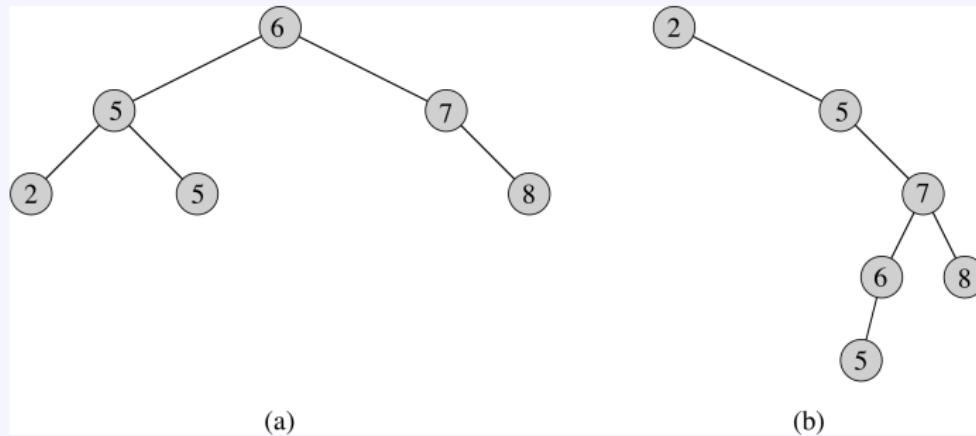
- ▶ *key* : an identifying field inducing a total ordering
- ▶ *left* : pointer to a left child (may be NULL)
- ▶ *right* : pointer to a right child (may be NULL)
- ▶ *p* : pointer to a parent node (NULL for root)

Binary Search Trees

Binary search tree should satisfy the following property :

$$\text{key}[\text{leftSubtree}(x)] \leq x.\text{key} \leq \text{key}[\text{rightSubtree}(x)]$$

Examples :



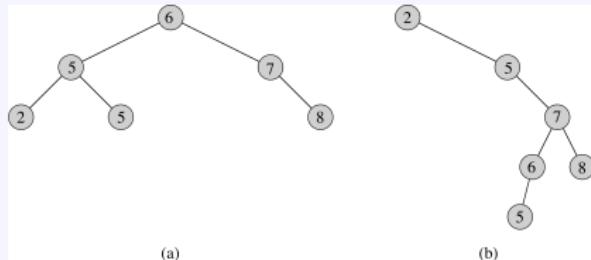
Inorder tree traversal

Inorder tree traversal :

- Visits elements in sorted (increasing) order

InorderTreeWalk(x)

```
if  $x \neq NIL$ 
    InorderTreeWalk( $x.left$ );
    print( $x.key$ );
    InorderTreeWalk( $x.right$ );
```



Preorder tree traversal

Preorder tree traversal :

- ▶ Visits root before left and right subtrees are visited

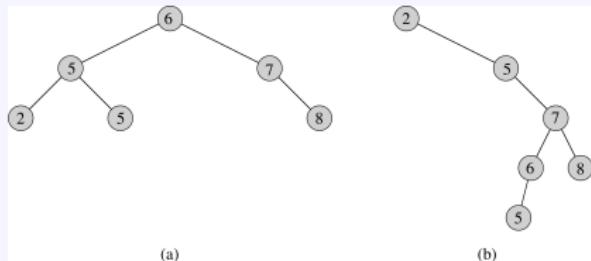
PreorderTreeWalk(x)

if $x \neq NIL$

 print($x.key$) ;

 PreorderTreeWalk($x.left$) ;

 PreorderTreeWalk($x.right$) ;



(a)

(b)

Postorder tree traversal

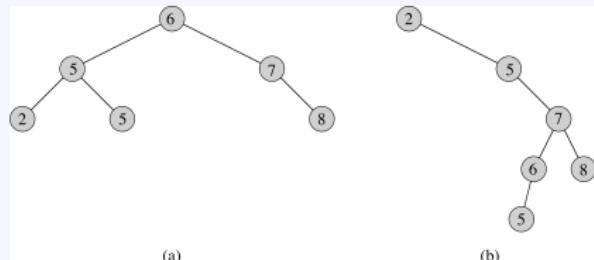
Postorder tree traversal :

- ▶ Visits root after visiting left and right subtrees

PostorderTreeWalk(x)

if $x \neq NIL$

```
    PostorderTreeWalk(x.left);  
    PostorderTreeWalk(x.right);  
    print(x.key);
```



Cost of tree traversals

Tree traversal, Inorder, Postorder and Preorder cost $\Theta(n)$ where n is the number of nodes in the tree. Proof :

- ▶ $T(n)$ is the time for tree traversal called on the root of an n -node subtree
- ▶ Each traversal visits the n nodes, $T(n) \in \Omega(n)$
- ▶ Traversal of an empty subtree cost c
- ▶ For $n > 0$, $T(n) = T(k) + T(n - k - 1) + d$ where d represents the cost of printing key x

Cost of tree traversals

For $n > 0$, $T(n) \leq T(k) + T(n - k - 1) + d$

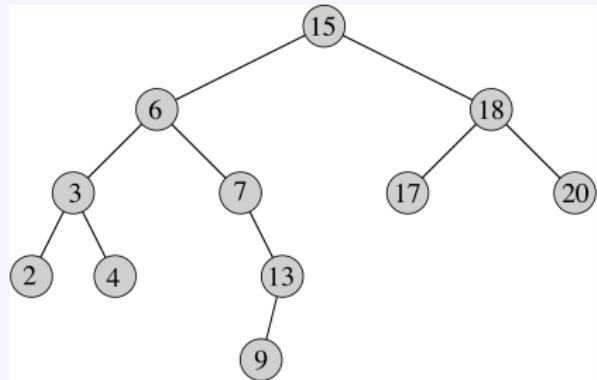
$T(n) \in O(n)$. Proof by substitution, we guess $T(n) = (c + d)n + c$ where $c + d$ is the constant amount time for each call + some constant c for the base case.

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \end{aligned}$$

Operations on BSTs : Recursive Search

Search for a key k at node x (x is a pointer)

```
TreeSearch(x, k)
if (x = NULL or k = x.key)
    return x;
if (k < x.key)
    return TreeSearch(x.left, k);
else
    return TreeSearch(x.right, k);
```



Cost $\Theta(h)$ (where h is the height of the tree) since search is performed along one path in the tree

Operations on BSTs : Iterative Search

Given a key k and a pointer x to a node, the following iterative procedure returns an element with that key or NULL :

```
TreeSearch(x, k)
    while (x != NULL and k != x.key)
        if (k < x.key)
            x = x.left;
        else
            x = x.right;
    return x;
```

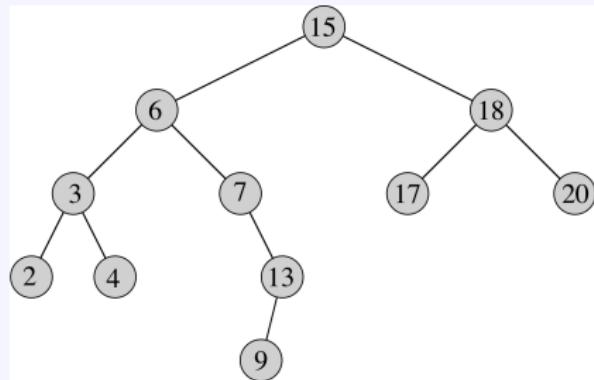
Same asymptotic time complexity $\Theta(h)$, but faster in practice.

Other BST Operations

Get the key with the minimum or the maximum value.

```
Tree-Minimum(x)
    while x.left ≠ NIL
        x = x.left
    return x

Tree-Maximum(x)
    while x.right ≠ NIL
        x = x.right
    return x
```



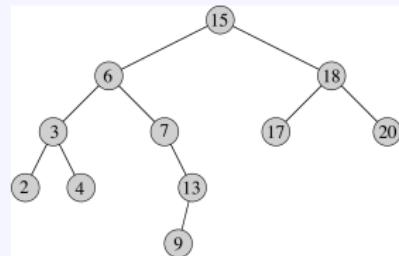
$\Theta(h)$

Successor operation

Given a node x , get its successor node in the sorted order of the keys.

Successor :

- ▶ x has a right subtree : successor is minimum node in right subtree
- ▶ x has no right subtree : successor is first ancestor of x whose left child is also ancestor of x
 - ▶ Intuition : As long as you move to the left up the tree, you're visiting smaller nodes.



Successor Operation

```
Tree-Successor(x)
```

```
    if x.right ≠ NIL
```

```
        return Tree-Minimum(x.right)
```

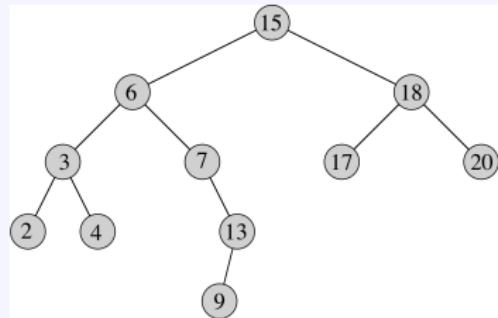
```
    y = x.p
```

```
    while y ≠ NIL and x == y.right
```

```
        x = y; y = y.p
```

```
    return y
```

$\Theta(h)$. Tree-predecessor symmetric to Tree-successor



Operations of BSTs : Insert

Adds an element x to the tree so that the binary search tree property continues to hold

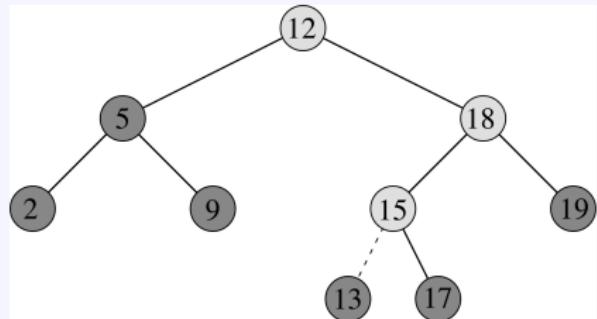
The basic algorithm is like the search procedure above

- ▶ Insert x in place of a NIL pointer
- ▶ Use a "trailing pointer" to keep track of where you came from

Operations of BSTs : Insert

Tree-Insert(T, z)

```
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL /* Tree was empty */
10    T.root = z
11  elseif z.key < y.key
12    y.left = z
13  else y.right = z
```



BST Search/Insert : Running Time

The running time of TreeSearch() or Tree-Insert() is $O(h)$, where $h =$ height of tree

The height of a binary search tree in the worst case is $h = O(n)$ when tree is just a linear string of left or right children

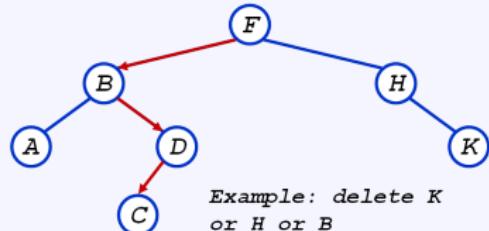
This worst case happens if keys are selected to be inserted in the tree in increasing or decreasing order

- ▶ We kept all analysis in terms of h so far for now
- ▶ We can maintain $h = O(\lg n)$ in a similar way as for quick sort by randomly picking among the keys the next one that is inserted in the tree

Operations of BSTs : Delete

The operation to delete a key z has 3 cases :

1. z has no children : Remove z
2. z has one child : Replace z by his child
3. z has two children :
 - ▶ Swap z with successor
 - ▶ Perform case 1 or 2 to delete it



Transplant routine

`TRANSPLANT(T, u, v)` replaces the subtree rooted at u by the subtree rooted at v

`Transplant(T, u, v)`

```
1  if u.p == NIL  
2      T.root = v  
3  elseif u == u.p.left  
4      u.p.left = v  
5  else u.p.right = v  
6  if v ≠ NIL  
7      v.p = u.p
```

Lines 1 - 2 handle the case in which u is the root of T . Otherwise, u is either a left child or a right child of its parent. Lines 3 - 4 take care of updating $u.p.left$ for u left child, line 5 updates $u.p.right$ for u a right child. `TRANSPLANT` does not attempt to update $v : left$ and $v : right$

BST delete routine

Tree-Delete(T, z)

if $z.\text{left} == \text{NIL}$

 Transplant($T, z, z.\text{right}$)

elseif $z.\text{right} == \text{NIL}$

 Transplant($T, z, z.\text{left}$)

else

$y = \text{Tree-minimum}(z.\text{right})$

 if $y.p \neq z$

 Transplant($T, y, y.\text{right}$)

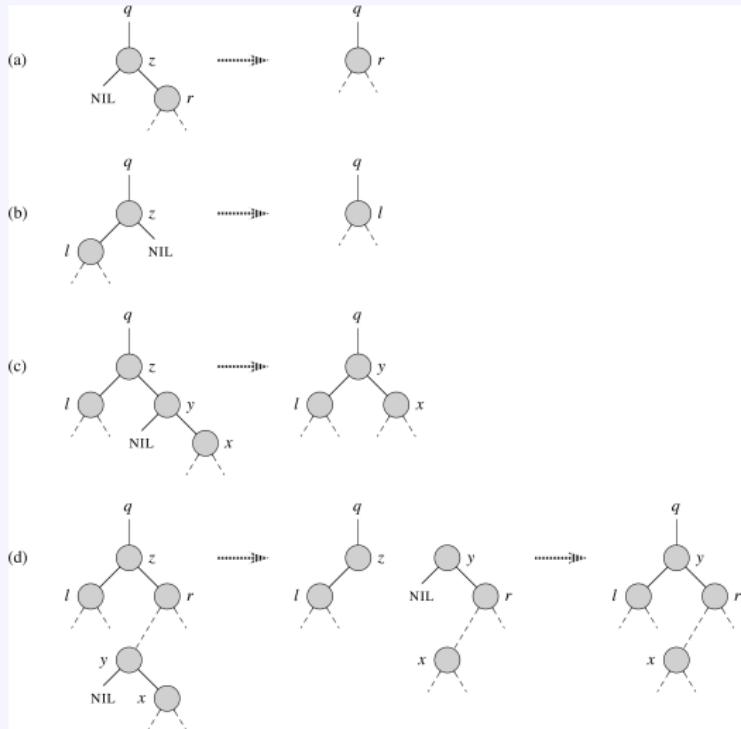
$y.\text{right} = z.\text{right}$

$y.\text{right}.p = y$

 Transplant(T, z, y)

$y.\text{left} = z.\text{left}$

$y.\text{left}.p = y$



Cases of procedure delete

This procedure for deleting a given node z has 3 cases :

1. If z has no left child (part (a) of fig.), replace z by its right child (which may or may not be NIL).
2. If z has just a left child (part (b) of fig.), replace z by its left child.
3. z has both a left and a right child. Find z 's successor y . Want splice y out of its current location and have it replace z in the tree.
 - ▶ If y is z 's right child (part (c)), then we replace z by y , leaving y 's right child alone.
 - ▶ Otherwise, y lies within z 's right subtree but is not z 's right child (part (d)). In this case, we first replace y by its own right child, and then we replace z by y .

Sorting with BST

Informal code for sorting array A of length n :

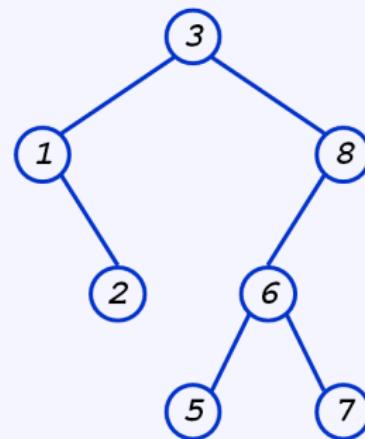
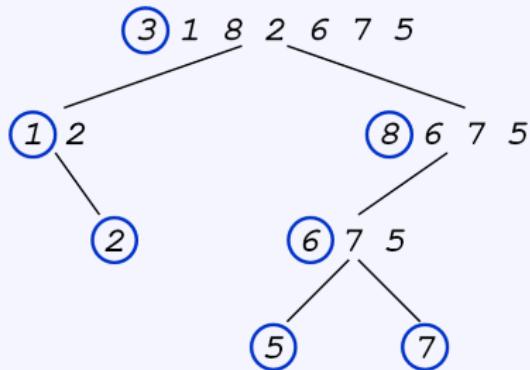
```
BSTSort(A)
for i=1 to n
    Tree-Insert(A[i]);
InorderTreeWalk(root);
```

- ▶ Best case is $\Omega(n \lg n)$
- ▶ Worst case is $O(n^2)$ (when tree is just a linear string of left or right children)
- ▶ Average case is $O(n \lg n)$

BST sort : Average case analysis

Average case analysis

- ▶ It's a form of quicksort !



BST sort : Average case analysis

Same partitions are done as with quicksort, but in a different order

In previous example

- ▶ Everything was compared to 3 once
- ▶ Then those items < 3 were compared to 1 once
- ▶ Etc.

Same comparisons as quicksort, different order !

- ▶ Example : consider inserting 5

BST sort : Average case analysis

Since run time is proportional to the number of comparisons, same average time as quicksort : $O(n \lg n)$

But quicksort better than BSTSort because quicksort has

- ▶ Better constants
- ▶ Sorts in place
- ▶ Doesn't need to build data structure

Algorithms and Data Structures

Lecture notes: Red-Black Trees, Cormen Chap. 13

Lecturer: Michel Toulouse

Vietnamese-German University
michel.toulouse@vgu.edu.vn

3 avril 2016

Red-Black trees

Red-Black trees are binary search trees augmented with node color

Operations on Red-Black trees are designed to guarantee that the height of the tree is always $h = O(\lg n)$

In this lecture :

- ▶ First : describe the properties of red-black trees
- ▶ Then : prove that these guarantee $h = O(\lg n)$
- ▶ Finally : describe operations on red-black trees

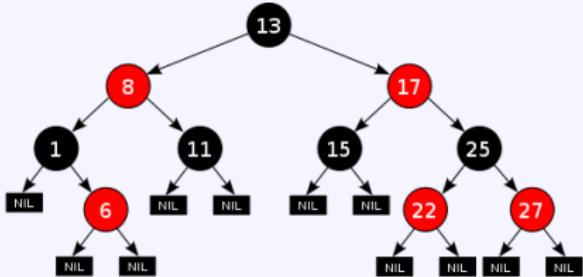
Red-black tree properties

1. Every node is either red or black
2. The root is always black
3. Every leaf (NIL pointer) is black (every "real" node has 2 children)
4. If a node is red, both children are black (can't have 2 consecutive reds on a path)
5. For each node, all simple paths from the node to descendant leaves contains the same number of black nodes

Red-black tree properties

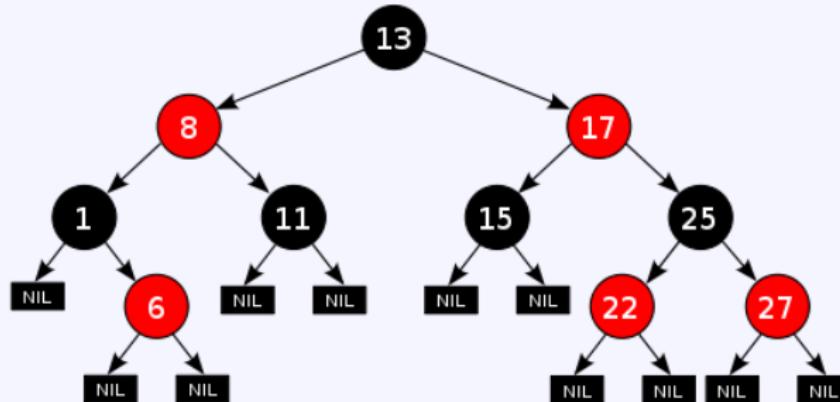
Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes



Black-height

The *black-height* (bh) of a node x is the number of black nodes, not including x , on a path from x to a leaf



A node of height h has black-height $\geq h/2$ since, by property 4, at least half of the nodes on a simple path must be black

Proving height bound of RB

Theorem

A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$

Claim : A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes

Proof by mathematical induction on height h

Basic step : x has height 0 (i.e., NIL leaf node)

- ▶ $bh(x) = 0$
- ▶ Subtree contains $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

Proving height bound of RB

Inductive step : x has positive height and 2 children

- ▶ Each child has black-height of $bh(x)$ or $bh(x) - 1$
- ▶ The height of a child = (height of x) - 1
- ▶ Inductive hypothesis : subtrees rooted at each child contain at least $2^{bh(x)-1} - 1$ internal nodes
- ▶ Thus subtree at x contains

$$\begin{aligned}& (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \\&= 2 \times 2^{bh(x)-1} - 1 \text{ internal nodes} \\&= 2^{bh(x)} - 1 \text{ internal nodes}\end{aligned}$$

By property 4, black height of the root r ($bh(r)$) is at least $h/2$

Thus at the root r of the red-black tree : $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$

- ▶ adding 1 on each side $n + 1 \geq 2^{h/2}$,
- ▶ taking the log on each side $\lg(n + 1) \geq h/2$,
- ▶ thus $2 \lg(n + 1) \geq h$.

Operations on RB trees : Worst-Case Time

So we proved that a red-black tree has $O(\lg n)$ height

Corollary : These operations take $O(\lg n)$ time :

- ▶ Minimum(), Maximum()
- ▶ Successor(), Predecessor()
- ▶ Search()
- ▶ Insert() and Delete()
 - ▶ also take $O(\lg n)$ time
 - ▶ But will need special care since they modify tree

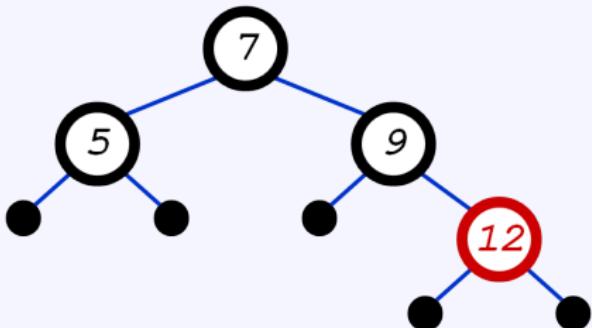
Note, the algorithms for the operations Minimum(), Maximum(), Successor(), Predecessor() and Search() are the same as for the binary search trees

Red-Black trees : an example

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Coloring this tree :



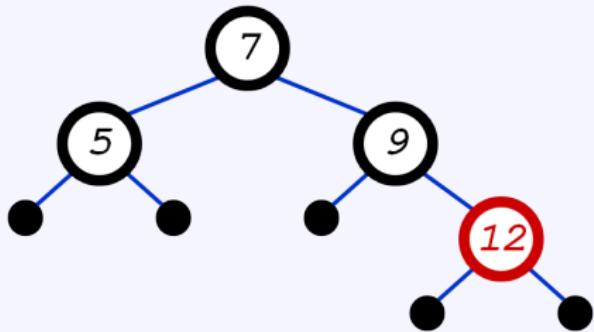
Red-Black trees : Insertion

Insertion

1. Use Tree-Insert algorithm in binary search

Insert node 8 :

- ▶ Where does it go ?



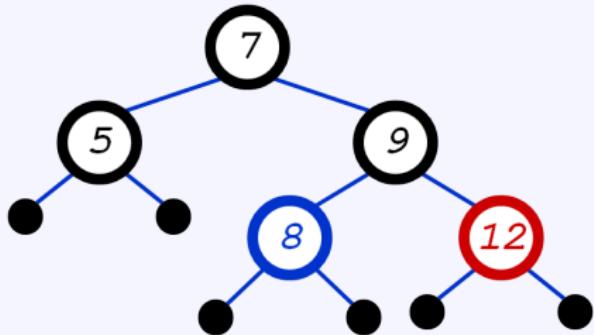
Red-Black trees : Insertion

Red-Black trees properties

5. Every path from node to descendant leaf contains the same number of black nodes

Insert node 8 :

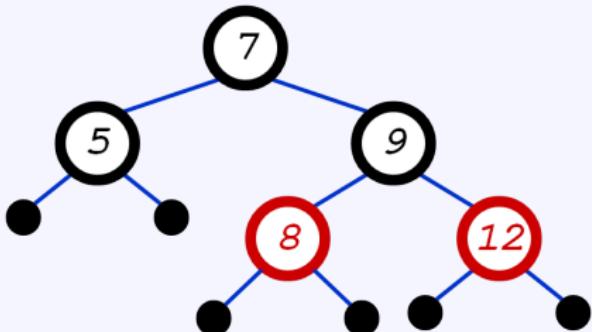
- ▶ Which color it should be ?



Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

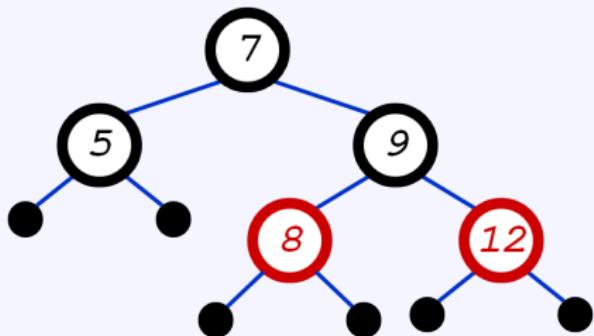


Red-Black trees : Insertion

Insertion

Insert 11 :

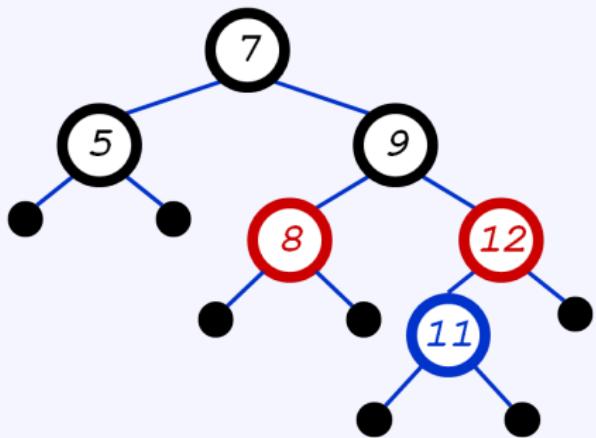
- ▶ Where node 11 goes
- ▶ Which color it should be ?



Red-Black trees : Insertion

Insertion

- ▶ Node 11 inserted



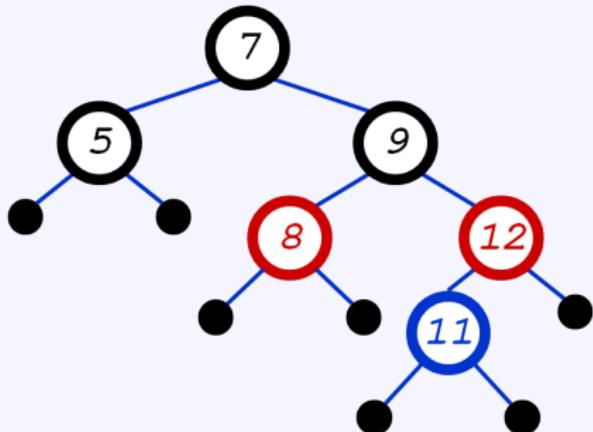
Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Insert 11 :

- ▶ Can't be red, violate property 4
- ▶ Can't be black, violate property 5

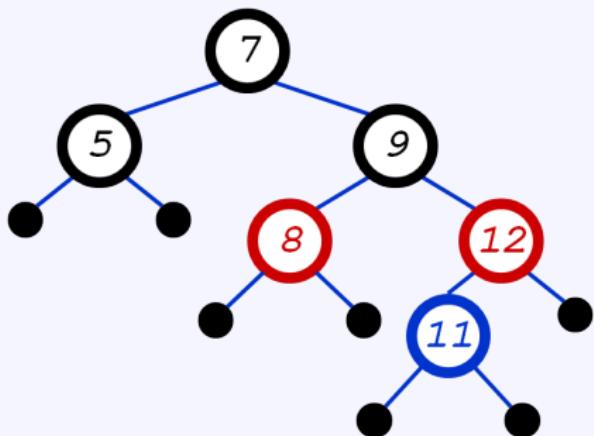


Red-Black trees : Insertion

Re-color

- ▶ The rotation operation is used to re-color the tree
- ▶ Operation described in the next slides

The solution is to re-color the tree !

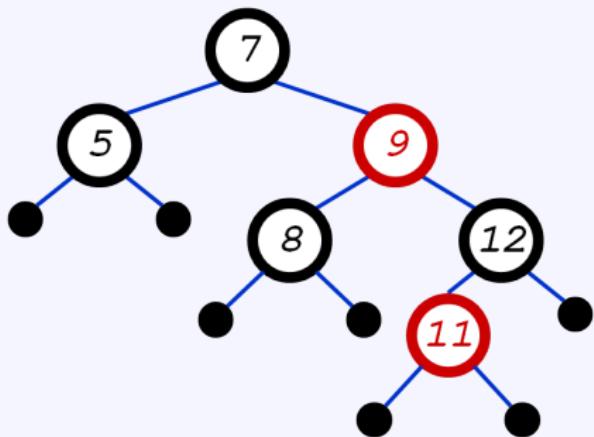


Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

Insert 10 :

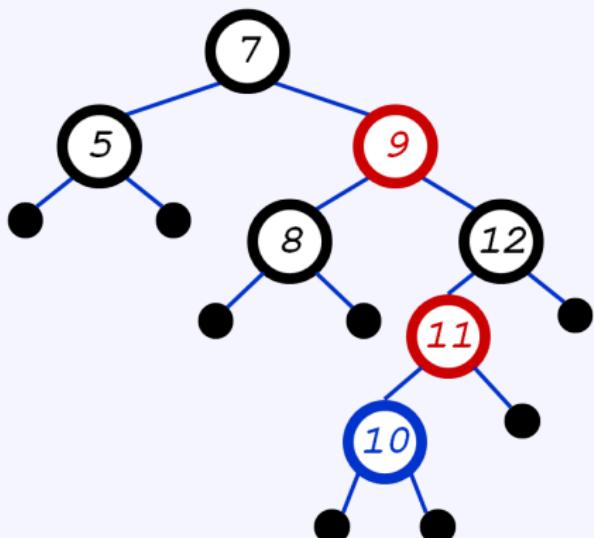


Red-Black trees : Insertion

Red-Black trees properties

1. Every node is either red or black
2. The root is always black
3. Every leaf is black
4. If a node is red, both children are black
5. Every path from node to descendant leaf contains the same number of black nodes

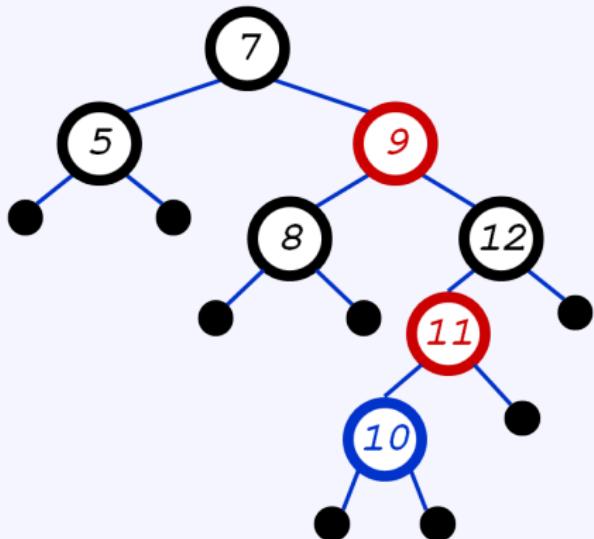
Insert 10 :



Red-Black trees : Insertion

- ▶ We cannot decide the color for 10, the tree is too unbalanced
- ▶ Must change tree structure to allow recoloring
- ▶ Goal : rebalance tree in $O(\lg n)$ time

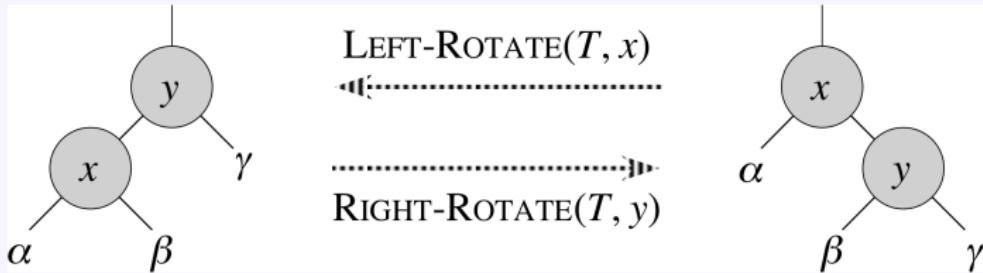
Insert 10 :



Red-Black trees : Rotation

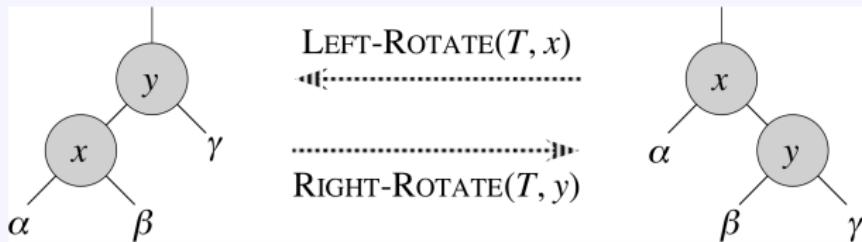
Our basic operation for changing tree structure is called **rotation** :

Two types of rotation operations : left rotation and right rotation

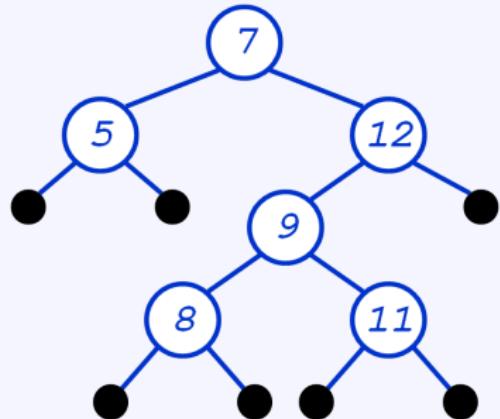
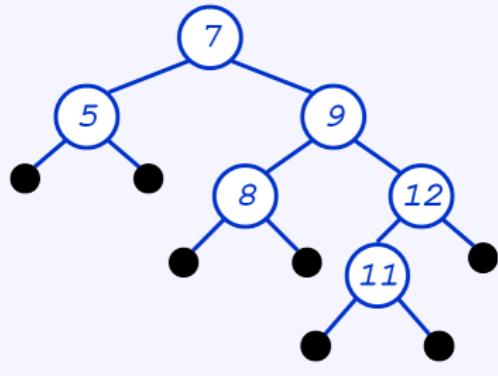


Rotation operations must preserve the inorder key ordering

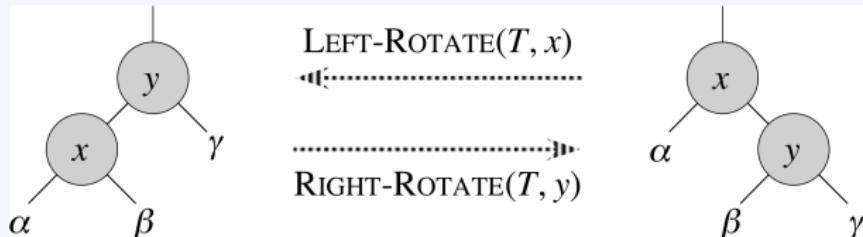
Left rotation



Rotate left about 9 :



Red-Black trees : Rotation



Lot of pointers manipulation

Right-Rotate(T, y) :

- ▶ x keeps its left child
- ▶ y keeps its right child
- ▶ x 's right child becomes y 's left child
- ▶ x 's and y 's parents change

Red-Black trees : Left rotation

Left-Rotate(T, x)

$y = x.\text{right}$

$x.\text{right} = y.\text{left}$

if $y.\text{left} \neq T.\text{nil}$

$y.\text{left.p} = x$

$y.p = x.p$

if $x.p == T.\text{nil}$

$T.\text{root} = y$

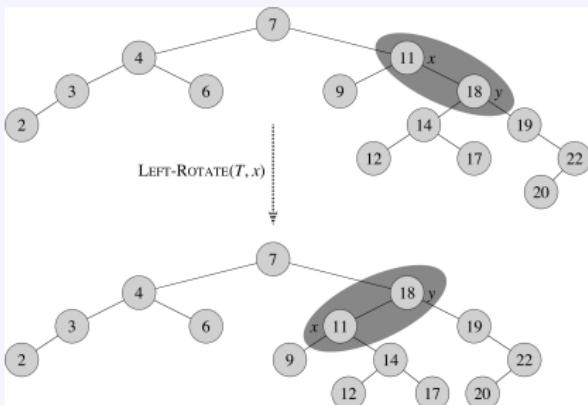
elseif $x == x.p.\text{left}$

$x.p.\text{left} = y$

else $x.p.\text{right} = y$

$y.\text{left} = x$

$x.p = y$



Red-Black trees : Insertion

Red-black insertion of a node z : the basic idea

- ▶ Start by doing regular binary-search-tree insertion
- ▶ Color z red
- ▶ Only RB properties 2 (z is a root) and 4 might be violated ($z.p$ is red)
- ▶ If so, move violation up tree until a place is found where it can be fixed
- ▶ Total time will be $O(\lg n)$

Red-Black tree operation : Insertion

RB-Insert(T, z)

```
1   y = T.nil
2   x = T.root
3   while x ≠ T.nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == T.nil
10      T.root = z
11   elseif z.key < y.key
12      y.left = z
13   else y.right = z
14   z.left = T.nil
15   z.right = T.nil
16   z.color = RED
17   RB-Insert-Fixup( $T, z$ )
```

Which properties of RB tree can be violated

Which of the red-black properties might be violated upon the call to RB-Insert-Fixup ?

Properties 1 and 3 still hold since both children of the newly inserted red node are the sentinel T :nil.

Property 5 (number of black nodes is the same on every simple path from a given node) is satisfied because node z replaces the (black) sentinel, and node z is red with sentinel children.

Thus, only property 2 (the root is black) and property 4 (a red node cannot have a red child) might be violated.

Both possible violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red.

Red-Black trees : Insertion

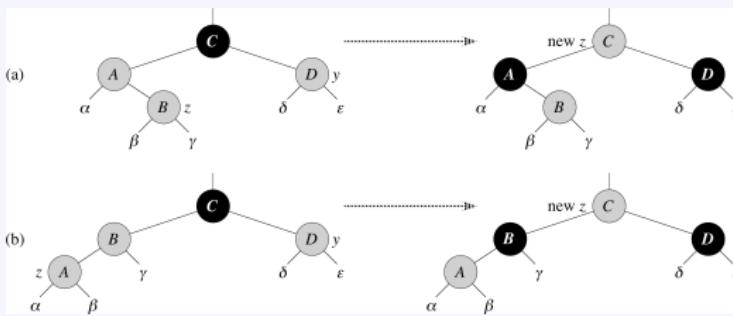
Remove the violation of property 4 :

RB-Insert-Fixup(T, z)

```
1   while  $z.p.color == \text{RED}$ 
2     if  $z.p == z.p.p.left$ 
3        $y = z.p.p.right$ 
4       if  $y.color == \text{RED}$ 
5          $z.p.color = \text{BLACK}$            //case 1
6          $y.color = \text{BLACK}$            //case 1
7          $z.p.p.color = \text{RED}$         //case 1
8          $z = z.p.p$                  //case 1
9       elseif  $z == z.p.right$ 
10       $z = z.p$                    //case 2
11      Left-Rotate( $T, z$ )          //case 2
12       $z.p.color = \text{BLACK}$         //case 3
13       $z.p.p.color = \text{RED}$        //case 3
14      Right-Rotate( $T, z.p.p$ )    //case 3
15    else (same as then clause with "right" and "left" exchanged)
16   $T.root.color = \text{BLACK}$ 
```

RB Insert : Case 1

Case 1 : uncle (y) is RED

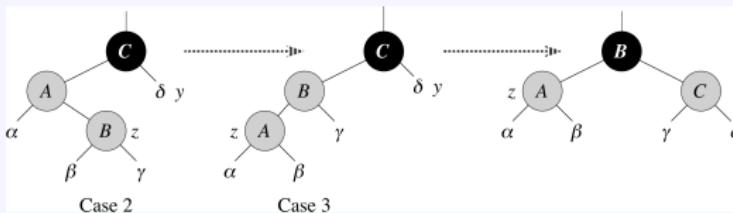


- 4 if $y.\text{color} == \text{RED}$
- 5 $z.p.\text{color} = \text{BLACK}$
- 6 $y.\text{color} = \text{BLACK}$
- 7 $z.p.p.\text{color} = \text{RED}$
- 8 $z = z.p.p$

- ▶ z.p.p must be black, since z and $z.p$ are both red
- ▶ Make $z.p$ and y black ; z and $z.p$ are not both red. But property 5 might be violated.
- ▶ Make $z.p.p$ red ; restores property 5
- ▶ The next iteration has $z.p.p$ as new z (z has moved up 2 levels)

RB Insert : Case 2

Case 2 : uncle (y) is BLACK ; Node z is a right child



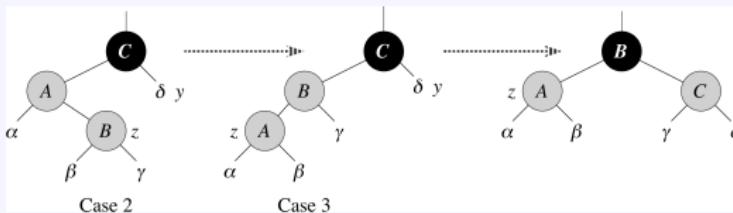
```

9   elseif z == z.p.right
10  z = z.p
11  Left-Rotate(T,z)
    
```

- ▶ Left rotate around z.p ; z is a left child, and both z and z.p are RED
- ▶ Takes us to case 3

RB Insert : Case 3

Case 3 : uncle (y) is BLACK ; Node z is a left child



- 12 $z.p.color = \text{BLACK}$
- 13 $z.p.p.color = \text{RED}$
- 14 Right-Rotate(T , $z.p.p$)

- ▶ Make $z.p$ BLACK and $z.p.p$ RED
- ▶ Then right rotate on $z.p.p$
- ▶ No longer have 2 reds in a row
- ▶ $z.p$ is BLACK ; no more iterations

RB Insert : Cases 4-6

Cases 1-3 hold if z's parent is a left child

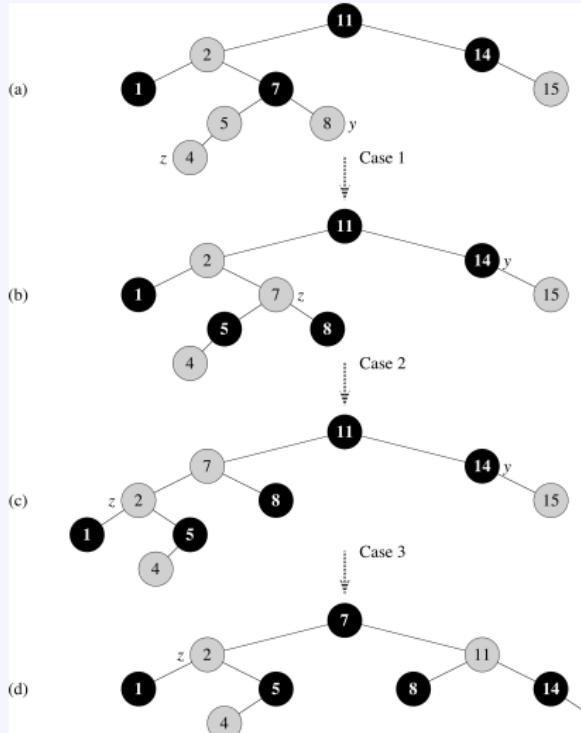
If z's parent is a right child, cases 4-6 are symmetric (swap left for right)

Insertion : final example

RB-Insert-Fixup(T, z)

```

1  while  $z.p.color == \text{RED}$ 
2    if  $z.p == z.p.p.left$ 
3       $y = z.p.p.right$ 
4      if  $y.color == \text{RED}$ 
5         $z.p.color = \text{BLACK}$  //case 1
6         $y.color = \text{BLACK}$  //case 1
7         $z.p.p.color = \text{RED}$  //case 1
8         $z = z.p.p$  //case 1
9      elseif  $z == z.p.right$ 
10      $z = z.p$  //case 2
11     Left-Rotate( $T, z$ ) //case 2
12      $z.p.color = \text{BLACK}$  //case 3
13      $z.p.p.color = \text{RED}$  //case 3
14     Right-Rotate( $T, z.p.p$ ) //case 3
15   else (same as then clause
        with "right" and "left" exchanged)
16    $T.root.color = \text{BLACK}$ 
```



RB Insert : analysis

$O(\lg n)$ to get through RB-Insert up to the call of RB-Insert-Fixup

Within RB-insert-Fixup :

- ▶ Each iteration takes $O(1)$.
- ▶ Each iteration is either the last one or it moves z up 2 levels.
- ▶ $O(\lg n)$ levels, therefore $O(\lg n)$ time.
- ▶ Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

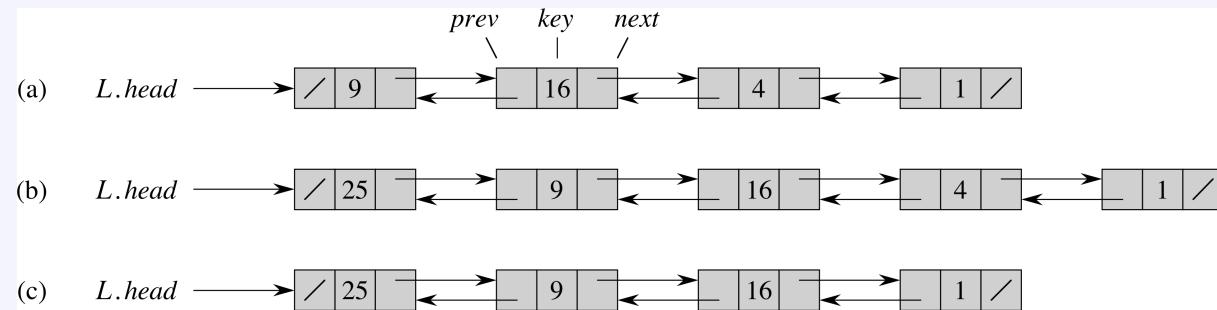
Elementary data structures

Stack : The element deleted is always the most recent one to have entered the data structure (LIFO). Insert and delete at the top

Queue : The element deleted is always the oldest one to have entered the data structure (FIFO). Insert at the end, delete at the beginning

Like heaps, stacks and queues can be implemented using an array

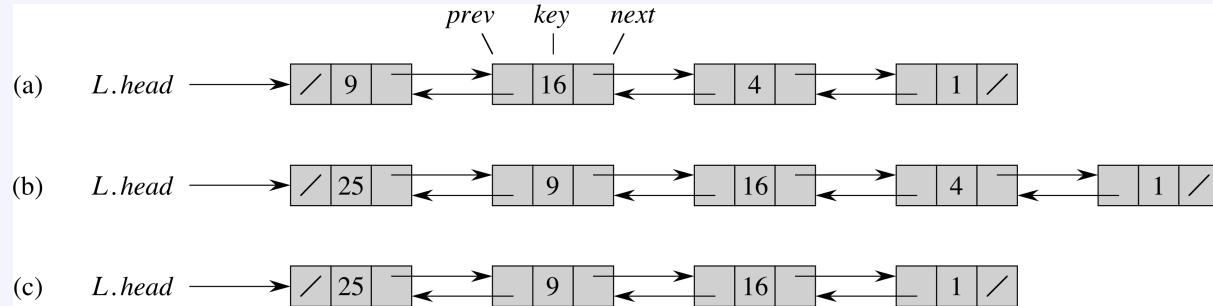
Elementary data structures



Link list : Elements in the set S are organized in a sequence connected by pointers

Doubly linked list : Can be traversed both from the beginning (head) or the end (tail)

Elementary data structures



To search an element in a link list, one needs a key, therefore we assume that each element has a *key* field

LIST-SEARCH(L, k)

x = *L.head*

while (*x* ≠ NILL and *x. key* ≠ *k*)

x = *x.next*

return *x*

Assumptions

It is common occurrence to assume that elements stored in a data structure have a key field which identify each element

- ▶ the key is used to find, delete, add the element in a data structure

Keys of elements are drawn from some set (Universe) U of keys

This set could be a range over the positive integers, physical memory addresses or the strings on an alphabet (like the strings based on the English alphabet that act like keys in the symbol table of a program)

Direct addressing

The key is used to index the entries in the data structure. Suppose :

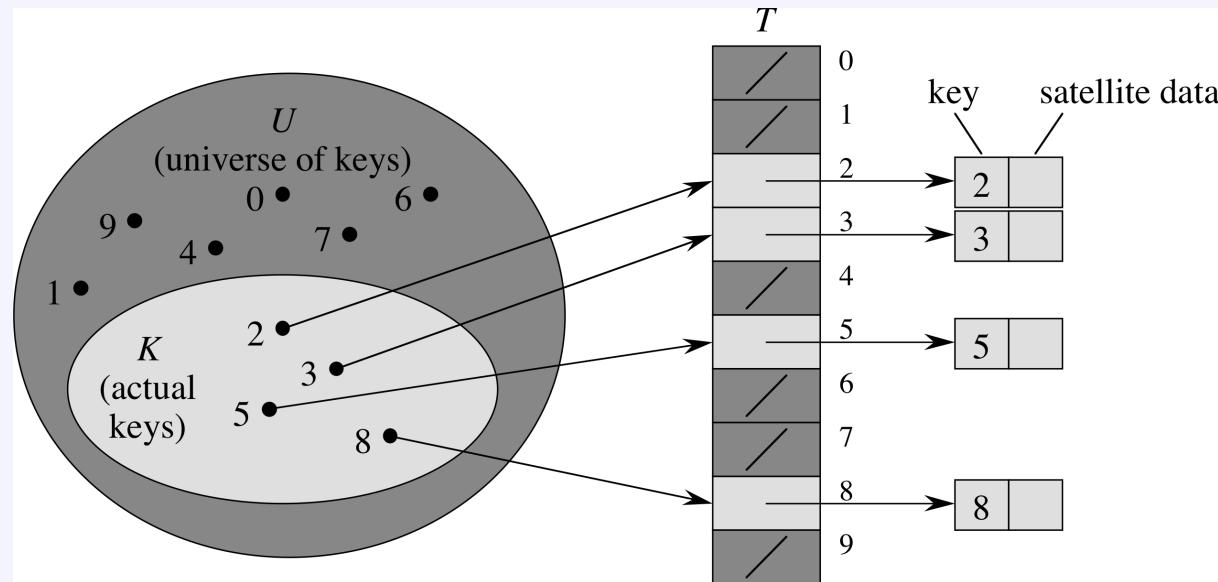
- ▶ the set of keys is the range $0..m - 1$
- ▶ keys are distinct

The idea :

- ▶ Set up an array $T[0..m - 1]$, the same size as the range of keys
- ▶ $T[i] = x$ if $x \in T$ and $\text{key}[x] = i$
- ▶ $T[i] = \text{NULL}$ otherwise

This is called a direct-addressing table

Direct addressing example



Operations take $O(1)$ time

$\text{SEARCH}(T, k)$
return $T[k]$

$\text{INSERT}(T, x)$
 $T[\text{key}[x]] = x$

$\text{DELETE}(T,x)$
 $T[\text{key}[x]] = \text{NILL}$

Problem with direct addressing

Direct addressing works well when the range m of keys is relatively small

But if the keys are 32-bit integers

- ▶ Problem 1 : direct-address table will have 2^{32} entries, more than 4 billion
- ▶ Problem 2 : even if memory is not an issue, the time to initialize the elements to NULL may be

Solution : map the universe of keys to a smaller range $0..m - 1$

This mapping is performed by a *hash function*

Hash table issues

Solution : map the universe of keys to a smaller range $0..m - 1$ called a *hash table*

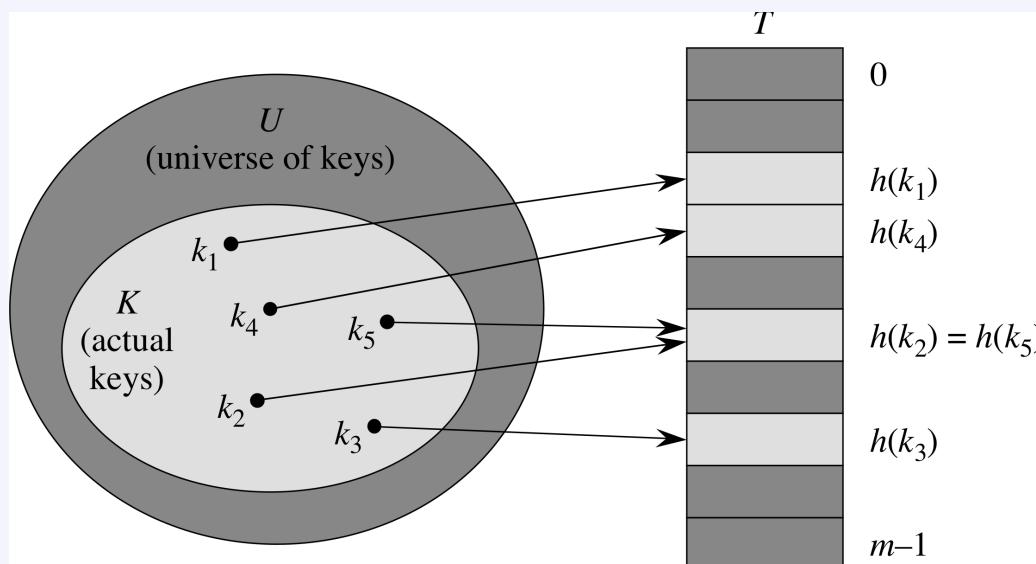
This mapping is performed by a *hash function*

- ▶ How to compute hash functions.
 - ▶ We will look at the *multiplication* and *division* methods.
- ▶ The hash function could map multiple keys to the same table index.
 - ▶ We will look at two techniques to address these "collisions" :
chaining and *open addressing*.

Mapping keys to smaller ranges

Instead of storing an element with key k in index k , use a function h and store the element in index $h(k)$

- ▶ The function h is the hash function.
- ▶ $h : U \rightarrow \{0, 1, \dots, m - 1\}$ so that $h[k]$ is a legal index in T .
- ▶ We say that k hashes to index $h(k)$



Collisions

Two or more keys hash to the same index in the hash table

Can happen when there are more possible keys than entries ($|U| > m$).

For a given set U of keys with $|U| < m$, may or may not happen.

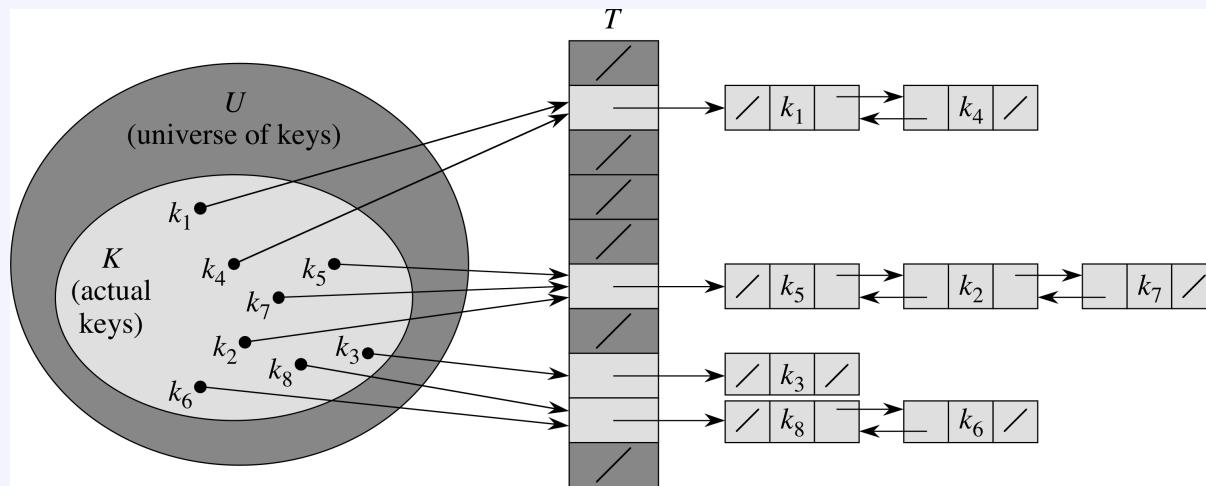
Definitely happens if $|U| > m$.

Must be prepared to handle collisions in all cases. Use two methods : chaining and open addressing.

Chaining is usually better than open addressing.

Chaining

Chaining puts elements that hash to the same index in a linked list :



SEARCH(T, k)

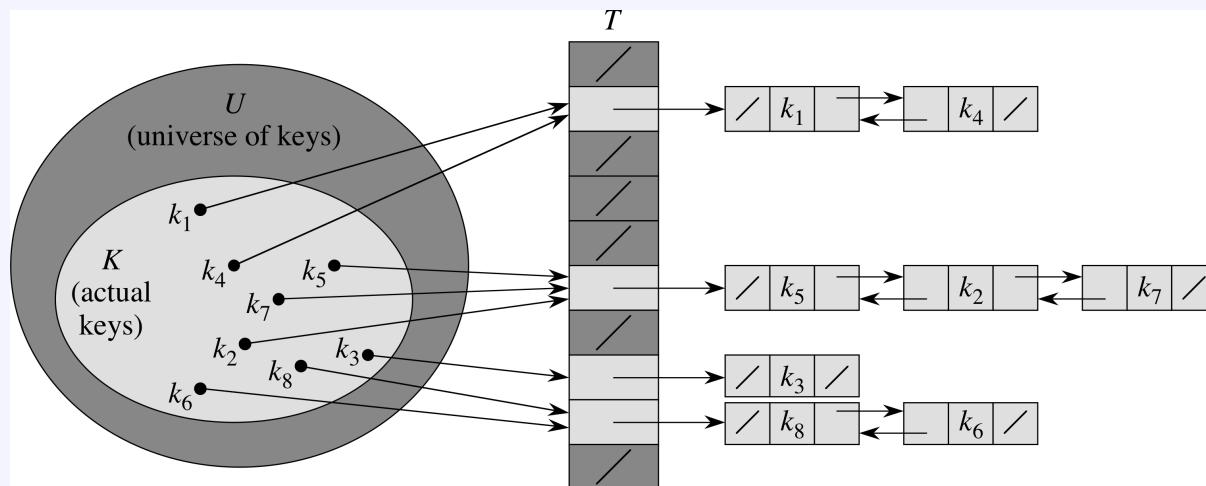
search for an element with key k in list $T[h(k)]$. Time proportional to length of the list of elements in $h(k)$

INSERT(T, x)

insert x at the head of the list $T[h(\text{key}[x])]$. Worst-case $O(1)$

Chaining

Chaining puts elements that hash to the same index in a linked list :



$\text{DELETE}(T, x)$

delete x from the list $T[h(\text{key}[x])]$. If lists are singly linked, then deletion takes as long as searching, $O(1)$ if doubly linked

Analysis of running time for chaining

load factor : Given n keys and m indexes in the table : the $\alpha = n/m$ = average # keys per index

Worst case : All keys hash in the same index, creating a list of length n . Searching for a key takes $\Theta(n)$ + the time to hash the key. We don't do hashing for the worst case !

Average case : Assume simple uniform hashing : each key in table is equally likely to be hashed to any index

- ▶ The average cost of an unsuccessful search for a key is $\Theta(1 + \alpha)$
- ▶ The average cost of a successful search is $\Theta(1 + \alpha/2) = O(1 + \alpha)$

Analysis of running time for chaining

If the size m of the hash table is proportional to the number n of elements in the table, say $\frac{1}{2}$, then $n = 2m$, and we have $n \in O(m)$

Consequently, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$, in other words, we can make the expected cost of searching constant if we make α constant

Choosing a hash function

A good hash function satisfies (approximately) the assumption of **simple uniform hashing** :

"given a hash function h , and a hash table of size m , the probability that two non-equal keys a and b will hash to the same slot is $P(h(a) = h(b)) = \frac{1}{m}$ "

In other words, each key is equally likely to hash in any of the m slots of the hash table

Under the assumption of uniform hashing, the load factor α and the average chain length of a hash table of size m with n elements will be

$$\alpha = \frac{n}{m}$$

Hash functions : the division method

$h(k) = k \bmod m$, i.e. hash k into a table with m entries using the index given by the remainder of k divided by m

Example : $k \bmod 20$ and $k = 91$, then $h(k) = 11$.

It's fast, requires just one division operation.

Disadvantage of the division method

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

Disadvantage : Have to avoid certain values of m :

- ▶ Powers of 2. If $m = 2^p$ for integer p , then $h(k)$ is just the least significant p bits of k . Examples : $m = 8 = 2^3$
 - ▶ $h(52) = 4$: $h(110100) = 100$
 - ▶ $h(37) = 5$: $h(100101) = 101$
- ▶ The implication is, for example, all keys that end with 100 map to the same index
- ▶ Solution : pick table size $m =$ a prime number not too close to a power of 2 (or 10). Example $m = 5$
 - ▶ $h(52) = 2$: $h(110100) = 010$
 - ▶ $h(36) = 1$: $h(100100) = 001$

Division method : example

More examples with the division method $h(k) = k \bmod m$

Recall : A number in base 10 is

$$6789 = (6 \times 10^3) + (7 \times 10^2) + (8 \times 10^1) + (9 \times 10^0) = 6789$$

Keys may be strings of characters. Have to be converted into integers using base 128 (7-bit ASCII values). For "CLRS", ASCII values are : C = 67, L = 76, R = 82, S = 83

Interpret CLRS as

$$(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) = 141,764,947$$

Hash fct : division method

If $m = 2^p$ for integer p , then $h(k)$ is just the least significant p bits of k .

$$128 = 2^7$$

So CLRS as $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0)$
 $\text{mod } 2^7 = 100001110011001010010\mathbf{1010011} = 83 = 1010011$

ABCS as $(65 \times 128^3) + (66 \times 128^2) + (67 \times 128^1) + (83 \times 128^0)$
 $\text{mod } 2^7 = 100000110000101000010\mathbf{1010011} = 83 = 1010011$

Hash fct : division method

If k is a character string interpreted in base 2^p (as in CLRS example), then $m = 2^p - 1$ is a poor choice as well : permuting characters in a string does not change its hash value (Exercise 11.3-3).

For example CLRS $\mod 2^7 - 1 =$ RLCS $\mod 2^7 - 1$. Assume
 $m = 2^7 - 1 = 127$

CLRS as $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0)$
 $\mod 127 = 54$

SRLC as $(83 \times 128^0) + (67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1)$
 $\mod 127 = 54$

$m = 9$ is a poor choice for integer keys. Try keys 123, 321, 231

Hash fct : multiplication method

1. Choose a constant A in the range $0 < A < 1$
2. Multiply key k by A
3. Extract the fractional part of kA
4. Multiply the fractional part by m
5. Take the floor of the result

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Examples for $m = 8 = 2^3$ and $A = 0.6180$:

$$\begin{aligned} h(52) &= \lfloor 8(52 \times 0.6180 \bmod 1) \rfloor \\ &= \lfloor 8(32.136 \bmod 1) \rfloor \\ &= \lfloor 8(0.136) \rfloor \\ &= \lfloor 1.088 \rfloor \\ &= 1 \end{aligned}$$

$$\begin{aligned} h(36) &= \lfloor 8(36 \times 0.6180 \bmod 1) \rfloor \\ &= \lfloor 8(22.248 \bmod 1) \rfloor \\ &= \lfloor 8(0.248) \rfloor \\ &= \lfloor 1.984 \rfloor \\ &= 1 \end{aligned}$$

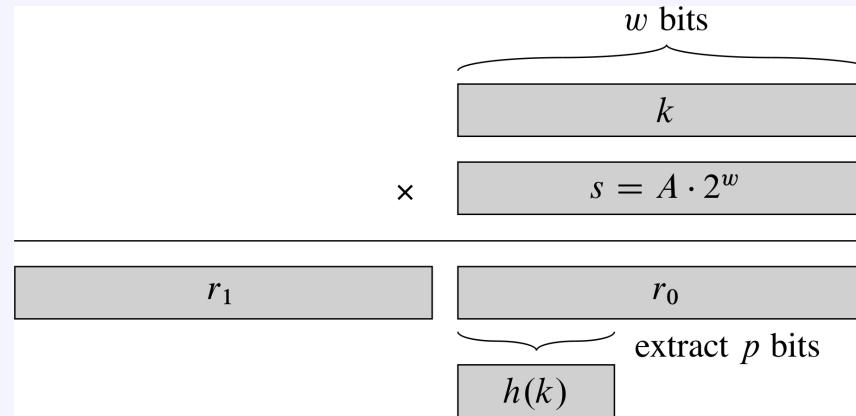
Multiplication method : advantage and disadvantage

- ▶ Choose $m = 2^p$ for implementation reasons
- ▶ Choose A not too close to 0 or 1
- ▶ Knuth : Good choice for $A = (\sqrt{5} - 1)/2$

Disadvantage : Slower than division method

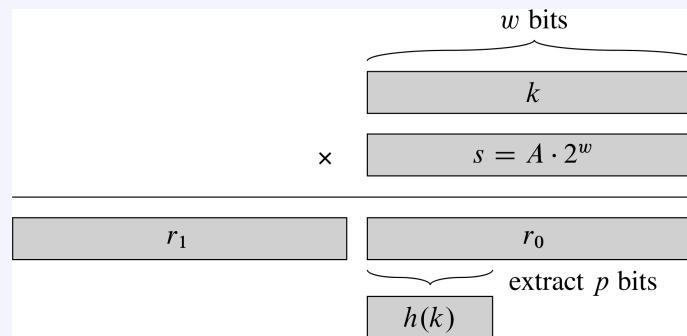
Advantage : Value of m not critical

Multiplication method : implementation



- ▶ w bits is the memory word size ; $0 < s < 2^w$
- ▶ Multiplying two w -bit words, the result is $2w$ bits, $r_1 2^w + r_0$
- ▶ r_1 is the high-order word of the product and r_0 is the low-order word
- ▶ $\lfloor m(kA \text{ mod } 1) \rfloor$ are the p most significant bits of r_0

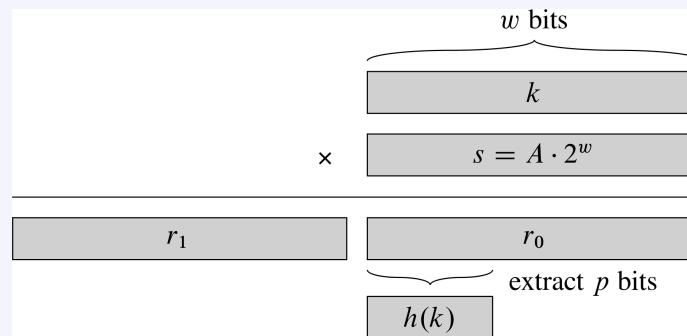
Multiplication method : implementation



Example : $m = 8 = 2^3$ ($p = 3$); $w = 5$; $k = 21$; $0 < s < 2^5$; $s = 13 \Rightarrow A = 13/32 = 0.40625$

$$\begin{aligned}
 h(21) &= \lfloor 8(21 \times 0.40625 \bmod 1) \rfloor &= ks = 21 \times 13 = 273 \\
 &= \lfloor 8(8.53125 \bmod 1) \rfloor &= 8 \times 2^5 + 17 \\
 &= \lfloor 8(0.53125) \rfloor &= r_1 = 8, r_0 = 17 = 10001 \\
 &= \lfloor 4.25 \rfloor &= \text{takes the } p = 3 \text{ most significant bits of } r_0 \\
 &= 4 &= 100 = 4
 \end{aligned}$$

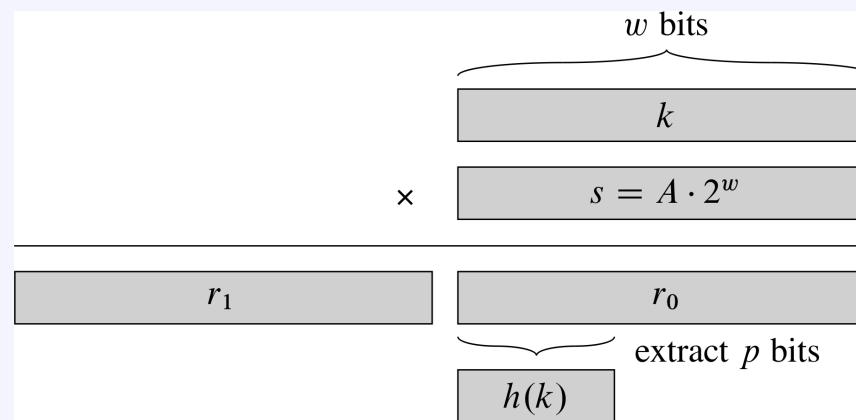
Multiplication method : implementation



Example : $m = 8 = 2^3$ ($p = 3$); $w = 6$; $k = 52$; $0 < s < 2^6$; $A = 0.625 \Rightarrow s = 0.625 \times 2^6 = 40$

$$\begin{aligned}
 h(52) &= \lfloor 8(52 \times 0.625 \bmod 1) \rfloor &= ks = 52 \times 40 = 2080 \\
 &= \lfloor 8(32.5 \bmod 1) \rfloor &= 2080 = 32^6 + 32 \\
 &= \lfloor 8(0.5) \rfloor &= r_1 = 32, r_0 = 32 = 100000 \\
 &= \lfloor 4 \rfloor &= \text{takes the } p = 3 \text{ most significant bits of } r_0 \\
 &= 4 &= 100 = 4
 \end{aligned}$$

Multiplication method : implementation



Example :

$$m = 8 = 2^3; \quad w = 6; \quad k = 52; \quad 0 < s < 2^6; \quad s = 40; \quad A = 0.625$$

We extract the p most significant bits of r_0 because we have selected the table to be of size $m = 2^p$, therefore we need p bits to generate a number in the range $0..m - 1$.

Open addressing

When collisions are handled through chaining, the keys that collide are placed in a link list in the same entry where the collision occurred

Open addressing place the keys that collide in another entry of the hash table by searching an empty slot in hash table using a *probe sequence*

The hash function is modified to include a *probe number* :

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

The probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ should be a permutation of $\langle 0, 1, \dots, m - 1 \rangle$ so every position in the table is eventually considered

Open addressing : Insertion

HASH-INSERT(T , k)

$i = 0$

 repeat

$j = h(k, i)$

 if $T[j] == \text{NIL}$

$T[j] = k$

 return j

 else $i = i + 1$ // probing

 until $i == m$

 error "hash table overflow"

Open addressing : Searching

HASH-SEARCH(T, k)

$i = 0$

repeat

$j = h(k, i)$

if $T[j] == k$

 return j

else $i = i+1$

until $T[j] == \text{NIL}$ or $i == m$

return NIL

Computing probe sequences

Probe sequences are dealing with collisions. We examine three techniques to generate probe sequences :

- ▶ Linear probing
- ▶ Quadratic probing
- ▶ Double hashing

Linear probing

$h(k, i)$ is based on an ordinary hash function
 $h' : U \rightarrow \{0, 1, \dots, m - 1\}$. Thus

$h(k, i) = (h'(k) + i) \bmod m$ for $i = 0, 1, \dots, m - 1$, thus $h(k, 0)$ is the initial hashing. If $h(k, 0) \neq NIL$, then there is a collision.

Linear probing resolves collisions by looking at the next entry in the table.

Assume we have the following table T ($h' = k \bmod 11$) :

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19

Linear probing

A call to $h(28, 0) = 6$ is a collision. The linear probing strategy will examine entries 7, 8 and finally 9. $h(28, 1) = 7$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
						↑				

$$h(28, 2) = 8$$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
							↑			

$$h(28, 3) = 9$$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
								↑		

Quadratic probing

Linear probing suffers from primary clustering : long runs of occupied sequences build up : an empty slot that follows i full slots has probability $\frac{i+1}{m}$ to be filled.

Quadratic probing jumps around in the table according to a quadratic function of the probe number : $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $c_1, c_2 \neq 0$ are constants and $i = 0, 1, \dots, m - 1$. Thus, the initial position probed is $T[h'(k)]$.

Issue : Can get a milder form of clustering : secondary clustering : if two distinct keys have the same $h(\text{key})$ value, then they have the same probe sequence.

Quadratic probing

Assuming $c_1 = c_2 = 1$, in this case, the probe sequence will be
 $h(k, i) = h'(k) + c_1 i + c_2 i^2 \bmod m = h'(k), h'(k) + 2, h'(k) + 6$, etc.

$$h(28, 1) = 8$$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
							↑			

$$h(28, 2) = 1$$

1	2	3	4	5	6	7	8	9	10	11
28	44	NIL	29	NIL	52	81	56	NIL	NIL	19
↑										

Quadratic probing : Issue

Can get a milder form of clustering : secondary clustering : if two distinct keys have the same $h(key)$ value, then they have the same probe sequence.

Double hashing

Use two auxiliary hash functions, h_1 and h_2 .

h_1 gives the initial probe, and h_2 gives the remaining probes :

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Issue : Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $[0, 1, \dots, m - 1]$

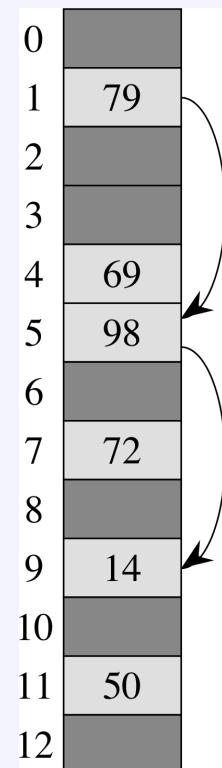
- ▶ Could choose m to be a power of 2 and h_2 to always produce an odd number > 1 .
- ▶ or m prime and h_2 to always produce an integer less than m

Handling collisions : Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Example : $m = 13$, $k = 14$, $h_1(k) = k \bmod 13$,
 $h_2(k) = 1 + (k \bmod 11)$. For $i = 0$

$$\begin{aligned} h(14, 0) &= h_1(14) + 0(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 0(1 + (14 \bmod 11)) \\ &= 1 + 0(1 + 3) \\ &= 1 \end{aligned}$$

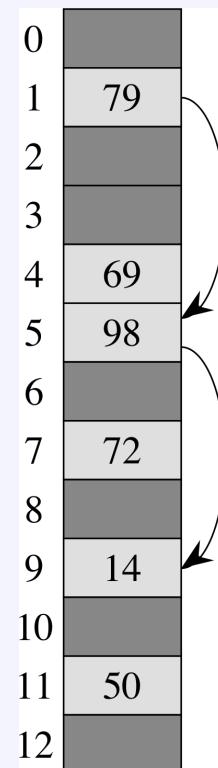


Handling collisions : Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Example : $m = 13$, $k = 14$, $h_1(k) = k \bmod 13$,
 $h_2(k) = 1 + (k \bmod 11)$. For $i = 1$

$$\begin{aligned} h(14, 1) &= h_1(14) + 1(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 1(1 + (14 \bmod 11)) \\ &= 1 + 1(1 + 3) \\ &= 5 \end{aligned}$$

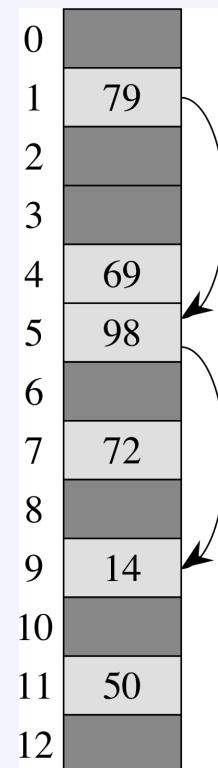


Handling collisions : Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Example : $m = 13$, $k = 14$, $h_1(k) = k \bmod 13$,
 $h_2(k) = 1 + (k \bmod 11)$. For $i = 2$

$$\begin{aligned} h(14, 2) &= h_1(14) + 2(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 2(1 + (14 \bmod 11)) \\ &= 1 + 2(1 + 3) \\ &= 9 \end{aligned}$$



Open addressing : Deleting

Use a special value DELETED instead of NIL when marking a index as empty during deletion.

- ▶ Suppose we want to delete key k at index j .
- ▶ And suppose that sometime after inserting key k , we were inserting key k' , and during this insertion we had probed index j (which contained key k).
- ▶ And suppose we then deleted key k by storing NIL into index j .
- ▶ And then we search for key k' .
- ▶ During the search, we would probe index j before probing the index into which key k' was eventually stored.
- ▶ Thus, the search would be unsuccessful, even though key k' is in the table.

Dynamic Programming

Divide-and-conquer algorithms decompose problems into subproblems, and then combine solutions to subproblems to obtain solutions for the larger problems.

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

However, during the divide part of divide-and-conquer some subproblems could appear more than once.

Dynamic programming

However, during the divide part of divide-and-conquer some subproblems could appear more than once.

If subproblems are duplicated, the computation of the solution of the subproblems is also duplicated, solving the same subproblems several times obviously yield very poor running times.

Dynamic programming algorithms avoid recomputing the solution of same subproblems by storing the solution of subproblems the first time they are computed, and referring to the stored solution when needed.

Example : Fibonacci numbers

A divide-and-conquer algorithm :

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

To compute Fibonacci of 7 ($\text{Fib}(7)$), the following decompositions take place :

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = \text{Fib}(0) + \text{Fib}(1) = 0 + 1 = 1$$

$$\text{Fib}(3) = \text{Fib}(1) + \text{Fib}(2) = 1 + 1 = 2$$

$$\text{Fib}(4) = \text{Fib}(2) + \text{Fib}(3) = 1 + 2 = 3$$

$$\text{Fib}(5) = \text{Fib}(3) + \text{Fib}(4) = 2 + 3 = 5$$

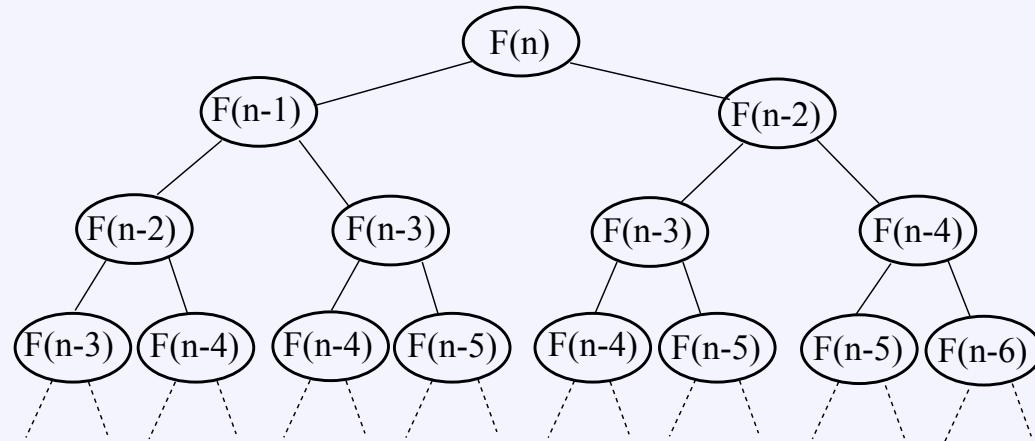
$$\text{Fib}(6) = \text{Fib}(4) + \text{Fib}(5) = 3 + 5 = 8$$

$$\text{Fib}(7) = \text{Fib}(5) + \text{Fib}(6) = 5 + 8 = 13$$

Another view of Fibonacci

```
function Fib(n)
    if (n ≤ 1) then return n;
    else
        return(Fib(n - 1) + Fib(n - 2));
```

The divide-and-conquer algorithm generates the following call tree :



The running time of Fib is $O((\frac{1+\sqrt{5}}{2})^n)$, it grows exponentially.

Dynamic Programming for Fibonacci

1. Divide-and-conquer :

```
function Fib_rec(n)
    if (n ≤ 1) then return n;
    else
        return(Fib_rec(n - 1) + Fib_rec(n - 2));
```

2. Dynamic Programming :

```
function fib_dyn(n)
    int *f, i;
    f = malloc((n + 1) * sizeof(int));
    for (i = 0; i ≤ n; i++)
        if (i ≤ 1)
            f[i] = i;
        else
            f[i] = f[i - 1] + f[i - 2];
    return f[n];
```

Dynamic Programming for Fibonacci

```
function fib_dyn(n)
    int *f, i ;
    f = malloc((n + 1) * sizeof(int)) ;
    for (i = 0; i ≤ n; i++)
        if (i ≤ 1)
            f[i] = i ;
        else
            f[i] = f[i - 1] + f[i - 2] ;
    return f[n] ;
```

0	1													
---	---	--	--	--	--	--	--	--	--	--	--	--	--	--

$\text{fib_dyn} \in \Theta(n)$ as opposed to the exponential complexity $O((\frac{1+\sqrt{5}}{2})^n)$ for fib_rec .

Summary

Instead of solving the same subproblem repeatedly, arrange to solve each subproblem only one time

Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem

"Store, don't recompute"

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3							

- ▶ here computing Fib(4) and Fib(5) both require Fib(3), but Fib(3) is computed only once

Can turn an exponential-time solution into a polynomial-time solution

When do we need DP

Dynamic programming is useful because it solves each subproblem only once.

Before writing a dynamic programming algorithm, first do the following :

- ▶ Write a divide-and-conquer algorithm to solve the problem
- ▶ Next, analyze its running time, if it is exponential then :
 - ▶ it is likely that the divide-and-conquer generates a large number of identical subproblems
 - ▶ therefore solving many times the same subproblems

If D&C has poor running times, we can consider DP.

But successful application of DP requires that the problem satisfies some conditions, which will be introduced later...

Writing a DP algorithm : the bottom-up approach

Create a table that will store the solution of the subproblems

Use the “base case” of D&C to initialize the table

Devise look-up template using the recursive calls of the D&C algorithm

Devise for-loops that fill the table using look-up template

The function containing the for loop returns the last entry that has been filled in the table.

An example : making change

Devise an algorithm for paying back a customer a certain amount using the smallest possible number of coins.

For example, what is the smallest amount of coins needed to pay back \$2.89 (289 cents) using as denominations "one dollars", "quarters", "dimes" and "pennies".

The solution is 10 coins, i.e. 2 one dollars, 3 quarters, 1 dime and 4 pennies.

Making change : a recursive solution

Assume we have an infinite supply of n different denominations of coins.

A coin of denomination i worth d_i units, $1 \leq i \leq n$.

We need to return change for N units.

```
function Make_Change(i,j)
  if (j == 0) then return 0 ;
  else
    return min(make_change(i - 1,j), make_change(i,j - di)+ 1);
```

The function is called initially as Make_Change(n, N).

Making change : DP approach

Assume $n = 3$, $d_1 = 1$, $d_2 = 4$ and $d_3 = 6$. Let $N = 8$.

To solve this problem by dynamic programming we set up a table $t[1..n, 0..N]$, one row for each denomination and one column for each amount from 0 unit to N units.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$									
$d_2 = 4$									
$d_3 = 6$									

The entry $t[i, j]$ indicates the minimum number of coins needed to refund an amount of j units using only coins from denominations 1 to i .

Making change : DP approach

The initialization of the table is obtained from the D&C base case :

if ($j == 0$) then return 0

i.e. $t[i, 0] = 0$, for $i = 1, 2, 3$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0								
$d_2 = 4$	0								
$d_3 = 6$	0								

Making change : DP approach

If $i = 1$, then only denomination $d_1 = 1$ can be used to return change.

Therefore $t[1, j]$ for $j = 1..8$ is $t[i, j] = t[i, j - d_i] + 1$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0								
$d_3 = 6$	0								

For example, the content of entry $t[1, 4] = t[1, 3] + 1$ means that the minimum number of coins to return 4 units using only denomination 1 is the minimum number of coins to return 3 units + 1 = 4 coins.

Making change : DP approach

If the amount of change to return is smaller than domination d_i , then the change needs to be returned using denominations smaller than d_i

For those cases, i.e. if $(j < d_i)$ then $t[i, j] = t[i - 1, j]$

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3					
$d_3 = 6$	0	1	2	3	1	2			

For all the other entries of the table we write the code of the DP algorithm using the recursive function

Making change : DP approach

The recursive function Make_Change

```
function Make_Change(i,j)
    if (j == 0) then return 0 ;
    else
        return min(make_change(i - 1,j), make_change(i,j - di)+ 1);
```

tells us that to fill entry $t[i,j], j > 0$, we have two choices :

1. Don't use a coin from d_i , then $t[i,j] = t[i - 1,j]$
2. Use at least one coin from d_i , then $t[i,j] = t[i,j - d_i] + 1$.

The recursive function also tells us that we take the min of these two values :

$$t[i,j] = \min(t[i - 1,j], t[i,j - d_i] + 1)$$

The DP algorithm

coins(n, N)

```
int d[1..n] = d[1, 4, 6];
int t[1..n, 0..N];
for (i = 1; i ≤ n; i++) t[i, 0] = 0; /*base case*/
for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ N; j++)
        if (i = 1) then t[i, j] = t[i, j - d[i]] + 1
        else if (j < d[i]) then t[i, j] = t[i - 1, j]
        else t[i, j] = min(t[i - 1, j], t[i, j - d[i]] + 1)
return t[n, N];
```

The algorithm runs in $\Theta(nN)$.

Making change : DP approach

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

To fill entry $t[i, j], j > 0$, we have two choices :

1. Don't use a coin from d_i , then $t[i, j] = t[i - 1, j]$
2. Use at least one coin from d_i , then $t[i, j] = t[i, j - d_i] + 1$.

Since we seek to minimize the number of coins return, we have

$$t[i, j] = \min(t[i - 1, j], t[i, j - d_i] + 1)$$

The solution is in entry $t[n, N]$

Making change : getting the coins

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

We can use the information in the table to get the list of coins that should be returned :

- ▶ Start at entry $t[n, N]$;
- ▶ If $t[i, j] = t[i - 1, j]$ then no coin of denomination i has been used to calculate $t[i, j]$, then move to entry $t[i - 1, j]$;
- ▶ If $t[i, j] = t[i, j - d_i] + 1$, then add one coin of denomination i and move to entry $t[i, j - d_i]$.

Optimal Substructure

DP is often used to solve optimization problems that have the following form

$$\begin{aligned} & \min f(x) \text{ or} \\ & \max f(x) \\ \text{s.t. } & \textit{some constraints} \end{aligned} \tag{1}$$

Making change is an optimization problem.

The function $f(x)$ to minimize is the number of coins

The constraint is the sum of the value of the coins is equal to the amount to return

Optimal Substructure

In solving optimization problems with DP, we find the optimal solution of a problem of size n by solving smaller problems of same type

The optimal solution of the original problem is made of optimal solutions from subproblems

Thus the subsolutions within an optimal solution are optimal subsolutions

Solutions to optimization problems that exhibit this property are say to be based on **optimal substructures**

Optimal Substructure

`Make_Change()` exhibits the optimal substructure property :

- ▶ The optimal solution of problem (i, j) is obtained using optimal solutions (minimum number of coins) of sub-problems $(i - 1, j)$ and $(i, j - d_i)$.

Amount	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Each entry $t[i, j]$ in the table is the optimal solution (minimum number of coins) that can be used to return an amount of j units using only denominations d_1 to d_i .

The optimal solution for $t[i, j]$ is obtained by comparing $t[i - 1, j]$ and $t[i - 1, j - d_i]$, taking the smallest of the two.

Optimal Substructure

To compute the optimal solution, we can compute all optimal subsolutions

Often we start with all optimal subsolutions of size 1, then compute all optimal subsolutions of size 2 combining some subsolutions of size 1. We continue in this fashion until we have out solution for n .

Note, optimal substructure does not apply to all optimization problems. When it fails to apply, we cannot use DP.

DP for optimization problems

The basic steps are :

- ▶ Characterize the structure of an optimal solution.
- ▶ Give a recursive definition for computing the optimal solution based on optimal solutions of smaller problems.
- ▶ Compute the optimal solutions and/or the value of the optimal solution in a bottom-up fashion.

Integer 0-1 Knapsack problem

Given n objects with integer weights w_i and values v_i , you are asked to pack a knapsack with no more than W weight (W is integer) such that the load is as valuable as possible (maximize). You cannot take part of an object, you must either take an object or leave it out.

Example : Suppose we are given 4 objects with the following weights and values :

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

Suppose $W = 5$ units of weight in our knapsack.

Seek a load that maximize the value

Problem formulation

Given

- ▶ n integer weights w_1, \dots, w_n ,
- ▶ n values v_1, \dots, v_n , and
- ▶ an integer capacity W ,

assign either 0 or 1 to each of x_1, \dots, x_n so that the sum

$$f(x) = \sum_{i=1}^n x_i v_i$$

is maximized, s.t.

$$\sum_{i=1}^n x_i w_i \leq W.$$

Explanation

$x_i = 1$ represents putting Object i into the knapsack and $x_i = 0$ represents leaving Object i out of the knapsack.

The value of the chosen load is $\sum_{i=1}^n x_i v_i$. We want the most valuable load, so we want to maximize this sum.

The weight of the chosen load is $\sum_{i=1}^n x_i w_i$. We can't carry more than W units of weight, so this sum must be $\leq W$.

Solving the 0-1 Knapsack

0-1 knapsack is an optimization problem.

Should we apply dynamic programming to solve it ? To answer this question we need to investigate two things :

1. Whether subproblems are solved repeatedly when using a recursive algorithm.
2. An optimal solution contains optimal sub-solutions, the problem exhibits optimal substructure

Optimal Substructure

Does integer 0-1 knapsack exhibits the optimal substructure property ?

Let $\{x_1, x_2, \dots, x_k\}$ be the objects in an optimal solution x .

The optimal value is $V = v_{x_1} + v_{x_2} + \dots + v_{x_k}$.

We must also have that $w_{x_1} + w_{x_2} + \dots + w_{x_k} \leq W$ since x is a feasible solution.

Claim :

If $\{x_1, x_2, \dots, x_k\}$ is an optimal solution to the knapsack problem with weight W , then $\{x_1, x_2, \dots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Optimal Substructure

Proof : Assume $\{x_1, x_2, \dots, x_{k-1}\}$ is not an optimal solution to the subproblem. Then there are objects $\{y_1, y_2, \dots, y_l\}$ such that

$$w_{y_1} + w_{y_2} + \dots + w_{y_l} \leq W',$$

and

$$v_{y_1} + v_{y_2} + \dots + v_{y_l} > v_{x_1} + v_{x_2} + \dots + v_{x_{k-1}}.$$

Then

$$v_{y_1} + v_{y_2} + \dots + v_{y_l} + v_{x_k} > v_{x_1} + v_{x_2} + \dots + v_{x_{k-1}} + v_{x_k}.$$

However, this implies that the set $\{x_1, x_2, \dots, x_k\}$ is not an optimal solution to the knapsack problem with weight W .

This contradicts our assumption. Thus $\{x_1, x_2, \dots, x_{k-1}\}$ is an optimal solution to the knapsack problem with $W' = W - w_{x_k}$.

Behavior of recursive solutions

Define $K[i, j]$ to be the maximal value for the 0-1-knapsack involving the first i objects for a knapsack of capacity j .

Then we have

$$K[1, j] = \begin{cases} v_1 & \text{if } w_1 \leq j \\ 0 & \text{if } w_1 > j. \end{cases}$$

To compute $K[i, j]$, notice that

- ▶ if we add the i th element to the knapsack, the sack had weight $j - w_i$ before it was added, and
- ▶ if we don't add the i th element, then $K[i, j] = K[i - 1, j]$.

Behavior of recursive solutions

To compute $K[i, j]$, notice that

- ▶ if we add the i th element to the knapsack, the sack had weight $j - w_i$ before it was added, and
- ▶ if we don't add the i th element, then $K[i, j] = K[i - 1, j]$.

Thus,

$$K[i, j] = \begin{cases} K[i - 1, j] & \text{if } w_i > j \\ \max\{K[i - 1, j], K[i - 1, j - w_i] + v_i\} & \text{if } w_i \leq j. \end{cases}$$

The maximum value is $K[n, W]$.

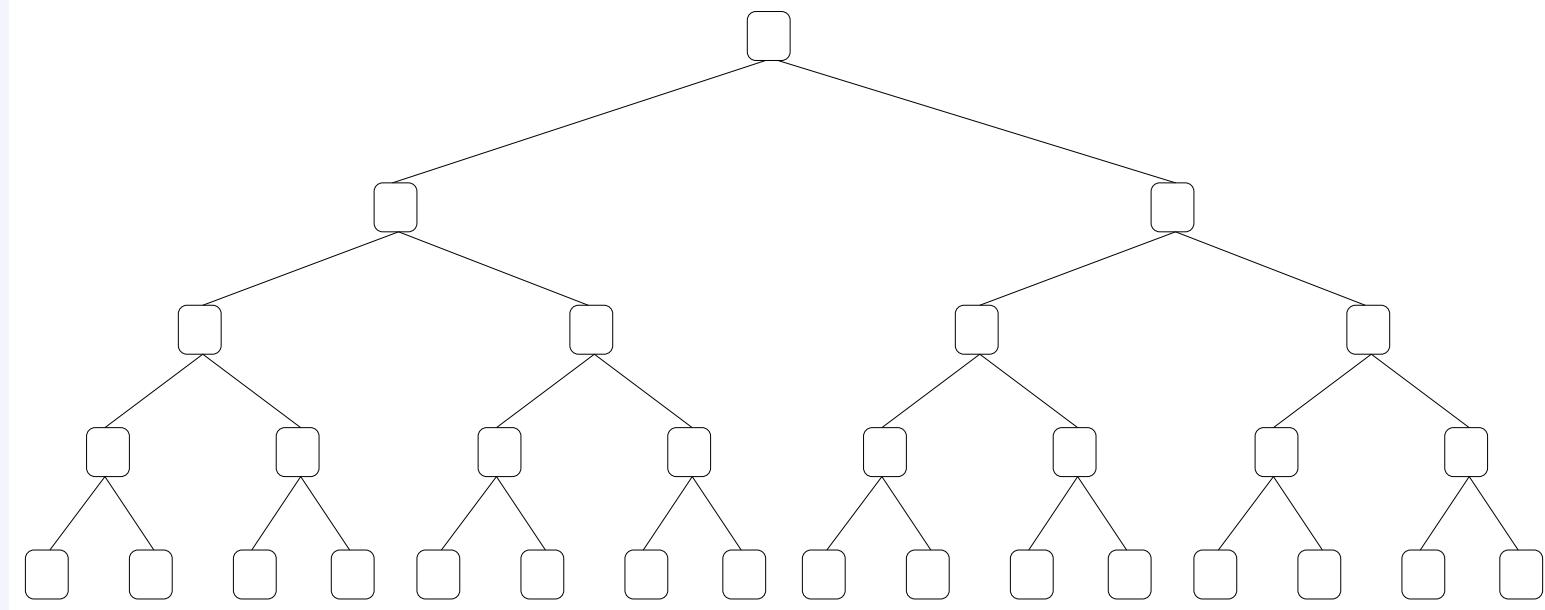
Divide & Conquer 0-1 Knapsack

```
int K(i, W)
    if (i = 1) return (W < w[1]) ? 0 : v[1];
    if (W < w[i]) return K(i - 1, W);
    return max(K(i - 1, W), K(i - 1, W - w[i]) + v[i]);
```

Solve for the following problem instance where $W = 10$:

i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

Optimal substructure



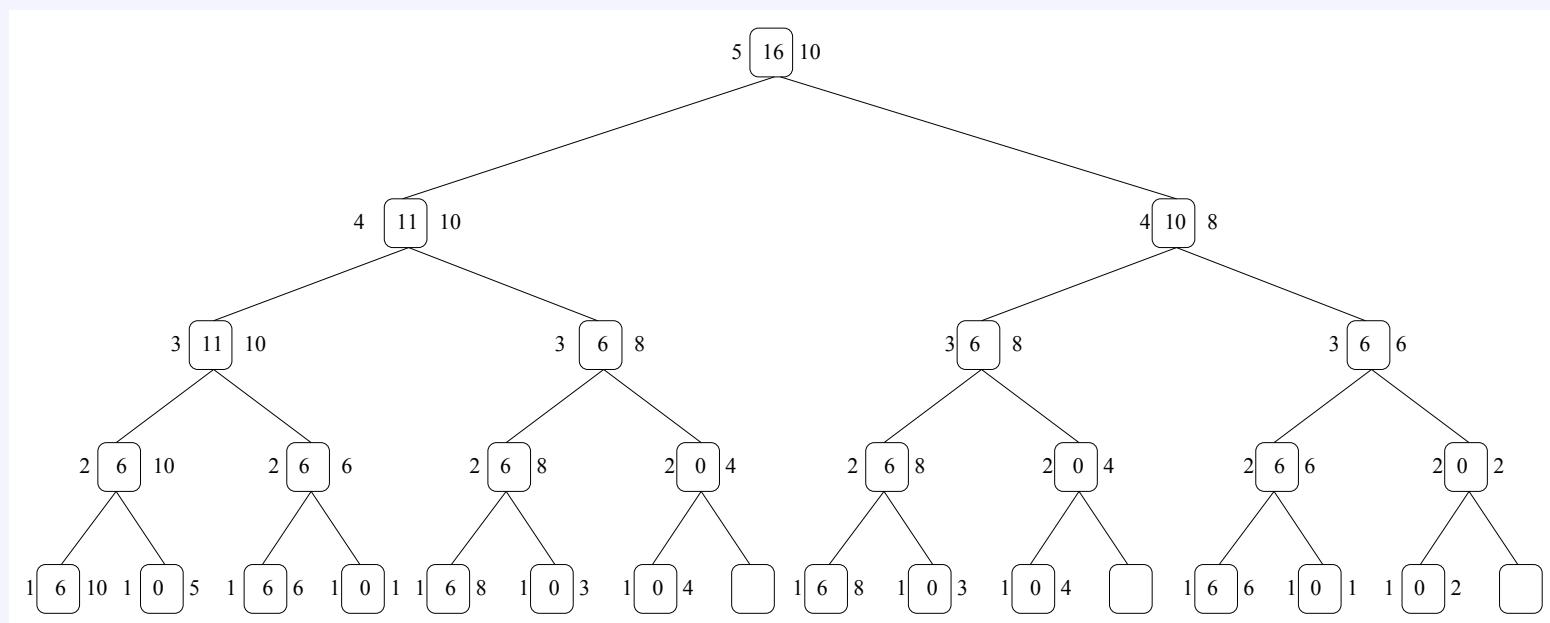
i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

int $K(i, W)$

```

if ( $i = 1$ ) return ( $W < w[1]$ ) ? 0 :  $v[1]$ ;
if ( $W < w[i]$ ) return K( $i - 1, W$ );
return max(K( $i - 1, W$ ), K( $i - 1, W - w[i]$ ) +  $v[i]$ );

```



Analysis of the Recursive Solution

Let $T(n)$ be the worst-case running time on an input with n objects.

If there is only one object, we do a constant amount of work.

$$T(1) = 1.$$

If there is more than one object, this algorithm does a constant amount work plus two recursive calls involving $n - 1$ objects.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + c & \text{if } n > 1 \end{cases}$$

Solving the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n - 1) + c & \text{if } n > 1 \end{cases}$$

we first observe that we cannot apply the Master Theorem because $b \leq 1$. So we use the substitution method here.

Solving the recurrence

Educated guess, $T(n) \in \Theta(2^n)$:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2[2T(n-2) + 1] + 1 \\ &= 2^2 T(n-2) + 2 + 1 \\ &= 2^2 [2T(n-3) + 1] + 2 + 1 \\ &= 2^3 T(n-3) + 2^2 + 2 + 1 \\ &= \dots \\ &= 2^{n-1} T(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} T(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Solving the recurrence

Basic Step ($n = 1$) :

$$\begin{aligned}T(1) &= 1 \text{ by definition} \\&= 2^1 - 1 = 1\end{aligned}$$

Solving the recurrence

Inductive Step :

Assume the closed formula holds for $n-1$, that is, $T(n-1) = 2^{n-1} - 1$, for all $n \geq 2$. Show the formula also holds for n , that is, $T(n) = 2^n - 1$.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \text{ by recursive definition} \\ &= 2(2^{n-1} - 1) + 1 \text{ by inductive hypothesis} \\ &= 2(2^{n-1}) - 2 + 1 \\ &= 2(2^{n-1}) - 1 \\ &= 2^n - 1 \end{aligned}$$

Therefore $T(n) \in \Theta(2^n)$

Overlapping Subproblems

We have seen that the maximal value is $K[n, W]$.

But computing $K[n, W]$ recursively cost $2^n - 1$.

But the number of subproblems is nW .

Thus, if $nW < 2^n$, then the 0-1 knapsack problem will certainly have overlapping subproblems, therefore using dynamic programming is most likely to provide a more efficient algorithm.

0-1 knapsack satisfies the two pre-conditions (optimal substructure and repeated solutions of identical subproblems) justifying the design of an DP algorithm for this problem.

0-1 Knapsack : DP algorithm

Declare a table K of size $n \times W + 1$ that stores the optimal solutions of all the possible subproblems. Let $n = 6$, $W = 10$ and

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											
4											
5											
6											

0-1 Knapsack : DP algorithm

Initialization of the table :

The value of the knapsack is 0 when the capacity is 0. Therefore,
 $K[i, 0] = 0, i = 1..10.$

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1	0			✓	✓	✓	✓	✓	✓	✓	✓
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

Initialization of the table using the base case of the recursive function :
if ($i = 1$) return ($W < w[1]$) ? 0 : $v[1]$

This said that if the capacity is smaller than the weight of object 1, then the value is 0 (cannot add object 1), otherwise the value is $v[1]$

Since $w[1] = 3$ we have :

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

The DP code for computing the other entries of the table is based on the recursive function for 0-1 knapsack :

```
int K(i, W)
    if (i == 1) return (W < w[1]) ? 0 : v[1];
    if (W < w[i]) return K(i - 1, W);
    return max(K(i - 1, W), K(i - 1, W - w[i]) + v[i] );
```

<i>i</i> \ <i>j</i>	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0										
3	0										
4	0										
5	0										
6	0										

0-1 Knapsack : DP algorithm

The dynamic programming algorithm is now (more or less) straightforward.

```
function 0-1-Knapsack( $w, v, n, W$ )
    int K[ $n, W + 1$ ] ;
    for( $i = 1; i \leq n; i ++$ )  $K[i, 0] = 0$  ;
    for( $j = 0; j \leq W; j ++$ )
        if ( $w[1] \leq j$ ) then  $K[1, j] = v[1]$  ;
        else  $K[1, j] = 0$  ;
    for ( $i = 2; i \leq n; i ++$ )
        for ( $j = 1; j \leq W; j ++$ )
            if ( $j \geq w[i] \ \&\& \ K[i - 1, j - w[i]] + v[i] > K[i - 1, j]$ )
                 $K[i, j] = K[i - 1, j - w[i]] + v[i]$  ;
            else
                 $K[i, j] = K[i - 1, j]$  ;
    return K[ $n, W$ ] ;
```

0-1 Knapsack Example

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

```

for ( $i = 2; i \leq n; i ++$ )
    for ( $j = 1; j \leq W; j ++$ )
        if ( $j \geq w[i] \&& K[i - 1, j - w[i]] + v[i] > K[i - 1, j]$ )
             $K[i, j] = K[i - 1, j - w[i]] + v[i];$ 
        else
             $K[i, j] = K[i - 1, j];$ 
    
```

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

Finding the Knapsack

How do we compute an optimal knapsack ?

With this problem, we don't have to keep track of anything extra. Let $K[n, k]$ be the maximal value.

If $K[n, k] \neq K[n - 1, k]$, then $K[n, k] = K[n - 1, k - w_n] + v_n$, and the n th item is in the knapsack.

Otherwise, we know $K[n, k] = K[n - 1, k]$, and we assume that the n th item is not in the optimal knapsack.

Finding the Knapsack

In either case, we have an optimal solution to a subproblem.

Thus, we continue the process with either $K[n - 1, k]$ or $K[n - 1, k - w_n]$, depending on whether n was in the knapsack or not.

When we get to the $K[1, k]$ entry, we take item 1 if $K[1, k] \neq 0$ (equivalently, when $k \geq w[1]$)

Finishing the Example

- Recall we had :

i	1	2	3	4	5	6
w_i	3	2	6	1	7	4
v_i	7	10	2	3	2	6

- We work backwards through the table

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	7	7	7	7	7	7	7	7
2	0	0	10	10	10	17	17	17	17	17	17
3	0	0	10	10	10	17	17	17	17	17	17
4	0	3	10	13	13	17	20	20	20	20	20
5	0	3	10	13	13	17	20	20	20	20	20
6	0	3	10	13	13	17	20	20	20	23	26

- The optimal knapsack contains $\{1, 2, 4, 6\}$

0-1 Knapsack Example

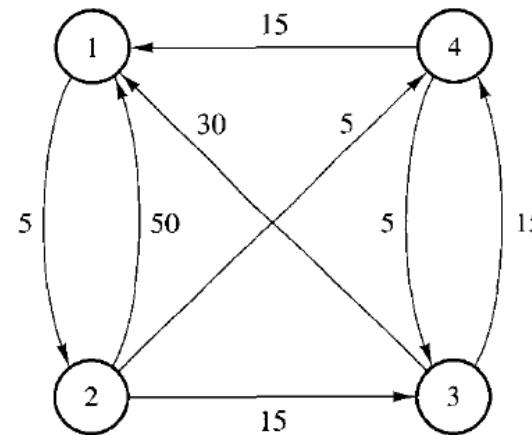
Given the following instance with $W = 10$:

i	1	2	3	4	5
w_i	6	5	4	2	2
v_i	6	3	5	4	6

$i \setminus j$	0	1	2	3	4	5	6	7	8	9	10
1	0										
2	0										
3	0										
4	0										
5	0										

What is the optimal value is? Which items should we take?

The All-Pairs Shortest-Path Problem



Problem definition :

- ▶ Let $G = \{V, E\}$ be a connected “directed” graph where V is the set of nodes and E is the set of edges.
- ▶ Each edge has an associated nonnegative length
- ▶ **Find the shortest path between all pairs of nodes in G**

We study a dynamic programming approach : Floyd's algorithm.

Floyd Algorithm

We have a distance matrix $L[i, j]$ that gives the length of each edge :

- ▶ $L[i, i] = 0$, $L[i, j] \geq 0$ if $i \neq j$,
- ▶ $L[i, j] = \infty$ if the edge (i, j) does not exist.

Algorithm Floyd($L[n, n]$)

```
D = L
for (k = 1; k ≤ n; k++)
    for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ n; j++)
            D[i, j] = min(D[i, j], D[i, k] + D[k, j])
return D
```

Algorithm constructs a matrix D that gives the length of the shortest path between each pair of nodes.

D is initialized to L , the direct distances between nodes.

After each iteration k , D contains the length of the shortest paths that only use nodes in $\{1, 2, \dots, k\}$ as intermediate nodes.

Floyd Algorithm

Algorithm Floyd($L[n, n]$)

```
 $D = L$ 
for ( $k = 1; k \leq n; k ++$ )
    for ( $i = 1; i \leq n; i ++$ )
        for ( $j = 1; j \leq n; j ++$ )
             $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 
```

At iteration k , the algo checks each pair of nodes (i, j) whether or not there exists a path from i to j passing through node k that is better than the present optimal path passing only through nodes in $\{1, 2, \dots, k - 1\}$.

If D_k represents the matrix D after the k -th iteration (so $D_0 = L$), the necessary check can be implemented by

$$D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$$

Execution of Floyd's algorithm

Algorithm Floyd($L[n, n]$)

```
 $D = L$ 
for ( $k = 1; k \leq n; k ++$ )
    for ( $i = 1; i \leq n; i ++$ )
        for ( $j = 1; j \leq n; j ++$ )
             $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 
```

Base case $D = L$, the smallest problem instances

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

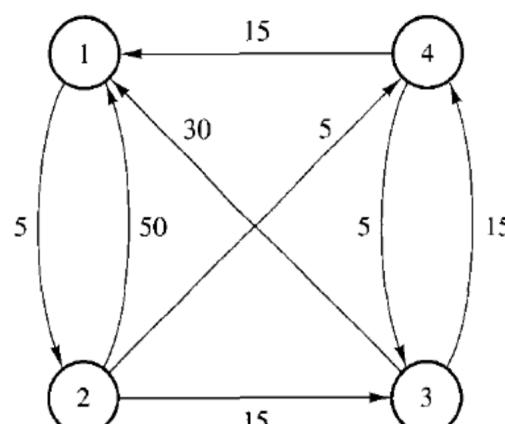
Algorithm Floyd($L[n, n]$)

```

 $D = L$ 
for ( $k = 1; k \leq n; k++$ )
  for ( $i = 1; i \leq n; i++$ )
    for ( $j = 1; j \leq n; j++$ )
       $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 

```

For $k = 1$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through node 1.



$$\begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

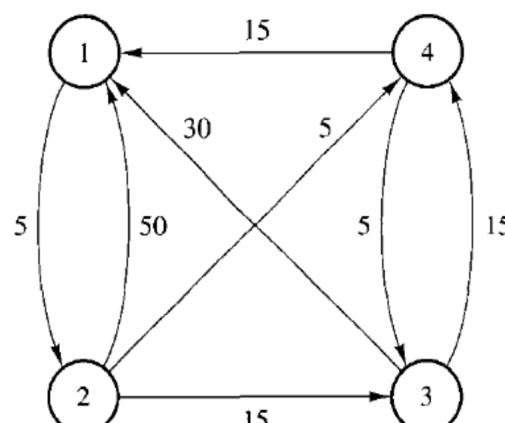
Algorithm Floyd($L[n, n]$)

```

 $D = L$ 
for ( $k = 1; k \leq n; k ++$ )
    for ( $i = 1; i \leq n; i ++$ )
        for ( $j = 1; j \leq n; j ++$ )
             $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 

```

For $k = 2$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through nodes 1 and 2.



$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Execution of Floyd's algorithm

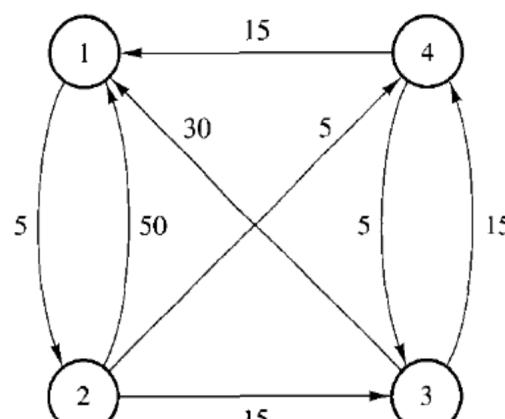
Algorithm Floyd($L[n, n]$)

```

 $D = L$ 
for ( $k = 1; k \leq n; k++$ )
  for ( $i = 1; i \leq n; i++$ )
    for ( $j = 1; j \leq n; j++$ )
       $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 

```

For $k = 3$, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through nodes $\{1, 2, 3\}$.



$$D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \end{pmatrix}$$

Execution of Floyd's algorithm

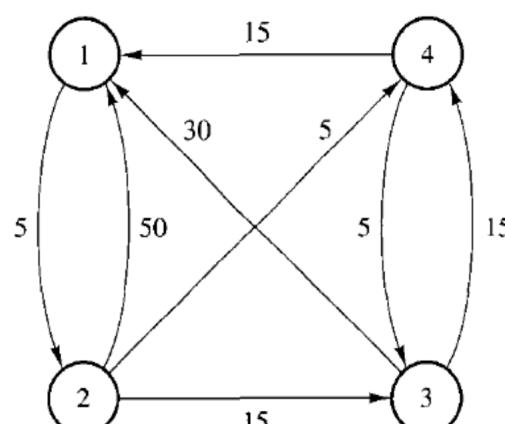
Algorithm Floyd($L[n, n]$)

```

 $D = L$ 
for ( $k = 1; k \leq n; k ++$ )
    for ( $i = 1; i \leq n; i ++$ )
        for ( $j = 1; j \leq n; j ++$ )
             $D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 

```

For $k = 4$, solution, compute the shortest path between each pair of nodes (i, j) when the path is allowed to pass through any nodes.



$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Computing the shortest paths

- ▶ We want to know the shortest paths, not just their length.
- ▶ For that we create a new matrix P of size $n \times n$.
- ▶ Then use the following algorithm in place of the previous one :

Algorithm Floyd($D[n, n]$)

Input : An array D of shortest path lengths

Output : The shortest path between every pair of nodes

$P[n, n]$ an $n \times n$ array initialized to 0

for ($k = 1; k \leq n; k++$)

for ($i = 1; i \leq n; i++$)

for ($j = 1; j \leq n; j++$)

if $D[i, k] + D[k, j] < D[i, j]$ **then**

$D[i, j] = D[i, k] + D[k, j]$

$P[i, j] = k$;

Computing the shortest paths

- ▶ The matrix P is initialized to 0.
- ▶ When the previous algorithm stops, $P[i, j]$ contains the number of the last iteration that caused a change in $D[i, j]$.

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- ▶ If $P[i, j] = 0$, then the shortest path between i and j is directly along the edge (i, j) .
- ▶ If $P[i, j] = k$, the shortest path from i to j goes through k .

Computing the shortest paths

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

- ▶ Look recursively at $P[i, k]$ and $P[k, j]$ to find other intermediate vertex along the shortest path.
- ▶ In the table above, since, $P[1, 3] = 4$, the shortest path from 1 to 3 goes through 4. If we look recursively at $P[1, 4]$ we find that the path between 1 and 4 goes through 2. Recursively again, if we look at $P[1, 2]$ and $P[2, 4]$ we find direct edge.
- ▶ Similarly if we look recursively to $P[4, 3]$ we find a direct edge (because $P[4, 3] = 0$). Then the shortest path from 1 to 3 is 1,2,4,3.

Greedy Algorithms

Greedy algorithms apply to problems that exhibit the following properties :

- ▶ greedy choice property,
- ▶ optimal substructure.

We have seen that optimal substructure means that optimal solutions are composed of optimal subsolutions.

Sometime there are too many subsolutions to check.

The "greedy choice property" means that an optimal solution can be obtained by making the "greedy" choice at every step.

Don't need to know the solutions of all the subproblems in order to make a choice, always make the choice that looks best at the moment.

Greedy Algorithms

Greedy algorithms are usually very simple. In most common situation, we distinguishes the following of an greedy algorithm :

- ▶ A **candidate set** (e.g. the nodes of a graph)
- ▶ A set of **chosen candidates** that have been selected to be part of the solution
- ▶ A **selection function (greedy criterion)** that indicates at any time which is the most promising of the candidates not yet used
- ▶ A function that checks whether a particular set of candidates *is a solution* to our problem
- ▶ A function that checks whether a set of candidates is *feasible*
- ▶ A function to optimize, the **objective function** (greedy algorithms are applied to optimization problems).

Example : the 0-1 knapsack problem

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

In this example, we have a set of 4 objects each with weight $w[i]$ and value $v[i]$, knapsack capacity $W = 5$ and decision variable x_i , $x_i = 1$ if object i is in the solution, otherwise $x_i = 0$.

To solve the knapsack problem one has to select a subset of objects such to $\max \sum_{i=1}^4 x_i v[i]$ while $\sum_{i=1}^4 x_i w[i] \leq W$

The way dynamic programming solves this problem is not very intuitive, it finds the optimal subset of objects for each capacity value

Greedy on the other hand select one object, put it in the knapsack and then select another one, until the knapsack is full

Example : the 0-1 knapsack problem

Object	1	2	3	4
Weight	1	1	2	2
Value	3	4	5	1

- ▶ The set of 4 objects is the initial *candidate set*
- ▶ The *selection function (greedy criterion)* could be the object in the candidate set that has the largest value (the most promising candidate)
- ▶ Adding object i such that $\sum_{i=1}^4 x_i w[i] > W$ could be the function that checks whether a particular set of candidates *provides a solution*
- ▶ $\sum_{i=1}^4 x_i w[i] \leq W$ is the function that checks whether a set of candidates is *feasible*
- ▶ $\max \sum_{i=1}^4 x_i v[i]$ is the *objective function*

Steps of a greedy Algorithm

- ▶ Initially, the set of chosen candidates is empty
- ▶ At each step, try to add to this set the best remaining candidate according to greedy criterion
- ▶ If the **increased** set of chosen candidates not feasible, remove the candidate just added, it is never considered again. Otherwise, candidate just added stays in the set of chosen candidates
- ▶ Each time the set of chosen candidates is increased, check whether the set now constitutes a solution to the problem
- ▶ When a greedy algorithm works correctly, the first solution found in this way is always optimal !

Note, greedy algorithms do not work for the 0-1 knapsack, it provides a feasible solution but not necessary optimal.

The making change problem : a greedy algorithm

Design an algorithm for paying back an customer a certain amount using the smallest possible number of coins.

For example, the smallest amount of coins to pay back \$2.89 (289 cents) is 10 : 2 one dollars, 3 quarters, 1 dime and 4 pennies.

Greedy algorithm :

- ▶ Choose the largest coin that is no larger than n (here n is the number of cents). Suppose that coin is the c cent coin.
- ▶ Give out $\lfloor n/c \rfloor$ of the c cent coins.
- ▶ Make change recursively for $n - \lfloor n/c \rfloor c$ cents.

Example of making change

Assume the coin denominations are 25, 10, 5 and 1 cents and we make change for 97 cents using the above greedy algorithm.

- ▶ Choose the largest coin that is no larger than n (here 25 cents is the largest coin no larger than 97 cents)
- ▶ Give out $\lfloor n/c \rfloor$ of the c cent coins ($\lfloor 97/25 \rfloor =$ three quarters)
- ▶ Make change recursively for $n - \lfloor n/c \rfloor c$ cents ($97 - 3 \times 25 = 22$ cents)
 - ▶ 10 cents is the largest coin no larger than 22 cents, so we hand out two dimes and make change for $22 - 20 = 2$ cents
 - ▶ a penny is the largest coin no larger than 2 cents, so we hand out two pennies and we're done!

Altogether we hand out 3 quarters, 2 dimes and 2 pennies, for a total of 7 coins

Greedy characteristics of the making change problem

The elements of the problem are :

- ▶ Candidate set : a finite set of coins, representing for instance 1, 5, 10, 25 units, and containing enough coins of each type
- ▶ Solution : the total value of the chosen set of coins is exactly the amount we have to pay back
- ▶ Feasible set : the total value of the chosen set *does not exceed* the amount to be returned
- ▶ Greedy criterion : chose the highest-valued coin remaining in the set of candidates ; and
- ▶ Objective function : minimize the number of coins used in the solution.

Does greedy algorithms always work ?

Depends. The greedy algorithm always gives out the fewest number of coins in change, when the coins are quarters, dimes, nickels and pennies.

It doesn't give out the fewest number of coins if the coins are 15 cents, 7 cents, 2 cents and 1 cent.

Example : For 21 cents the greedy algorithm would

- ▶ give out one 15-cent coin, remainder $21 - 15 = 6$ and
- ▶ give out three 2-cent coins,
- ▶ for a total of four coins.

Better : give out three 7-cent coins for a total of three coins !

Greedy problems

We can find counter examples for 0-1 knapsack and making change problems where greedy algorithms fail to find an optimal solution

However, both problems exhibit the optimal substructure, the optimal solution to a problem instance contains in it the optimal solution of subproblems

The difference is in the way greedy algorithms solve problems compared to dynamic programming algorithms

Greedy problems

Typically dynamic programming algorithms solve problems bottom-up

- ▶ finding the optimal solution to the smallest instances
- ▶ then building optimal solutions to largest instances from the optimal solutions of the smallest.

Greedy algorithms solve in top-down fashion making the best choice at the moment and then solving recursively the subproblem that remains

A problem that can be solved in this way is said to exhibit the **greedy property**

To use a greedy algorithm one must prove that the problem exhibit the greedy property, the prove is often by induction

Minimum Spanning Tree (MST)

Let $G = (V, E)$ be a connected, weighted graph.

A weighted graph is a graph where a real number called **weight** is associated with each edge.

A **spanning tree** of G is a subgraph T of G which is a tree that spans all vertices of G . In other words, T contains all of the vertices of G .

Minimum Spanning Tree

The weight of a spanning tree T is the sum of the weights of its edges. That is,

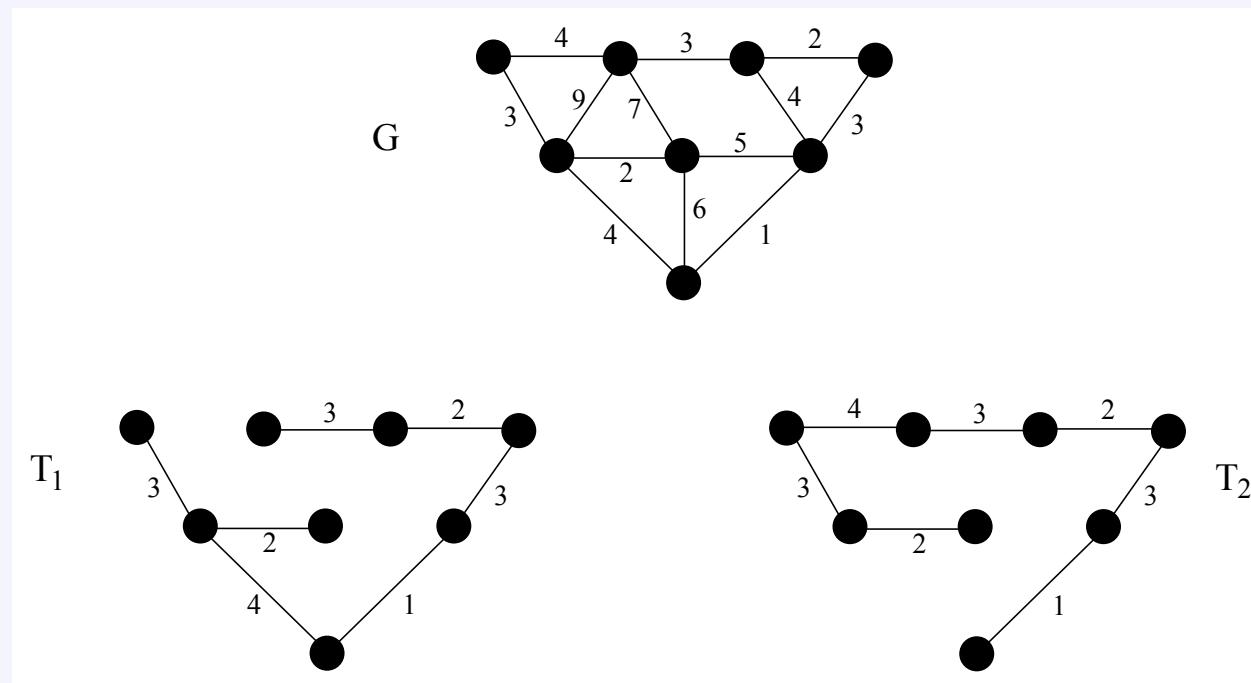
$$w(T) = \sum_{(u,v) \in T} w(u, v).$$

A minimum spanning tree (MST) of G is spanning tree T of minimum weight.

It should be clear that a minimum spanning tree always exists.

Minimum Spanning Tree : Examples

- Here is an example of a graph and two minimum spanning trees.



Constructing an MST

We have proved that the MST problem exhibit the optimal substructure and greedy properties.

Therefore minimum spanning trees can be constructed using greedy algorithms.

There are two common greedy algorithms to construct MSTs :

- ▶ **Kruskal's algorithm**
- ▶ **Prim's algorithm**

Both of these algorithms use the same basic ideas, but in a slightly different fashion.

Prim's Algorithm—More Details

For each node x , we store

- ▶ The predecessor $p(x)$. This is the vertex y in T which we join x to when edge (x, y) is added to T .
- ▶ The key(x). The weight of the minimum weight edge that connects x to some vertex in T .

$\text{key}(r) = 0$, and $p(r) = \text{NIL}$ throughout.

Each node $x \neq r$ starts with $\text{key}(x) = \infty$.

The value $\text{key}(x)$ only changes if some node adjacent to x is added to T .

Prim's Algorithm—More Details

Thus, when a node y is added to T , the key values of the nodes adjacent to y is updated

The vertices in $V - T$ are stored in a priority queue Q based on $\text{key}(x)$. This allows to pick the minimum weight edge to add to T .

T is not stored explicitly. The MST is reconstructed using the predecessors $p(x)$ for all $p \neq r$.

Prim's Algorithm

Prim_MST(G, r)

$\forall u \in G$

$key[u] = \text{Max_Int}$;

$key[r] = 0$;

$p[r] = \text{NIL}$;

$Q = \text{MinPriorityQueue}(V[G])$

while ($Q \neq \emptyset$)

$u = \text{ExtractMin}(Q)$;

for each v adjacent to u

if (($v \in Q$) & ($w(u, v) < key[v]$))

$p[v] = u$;

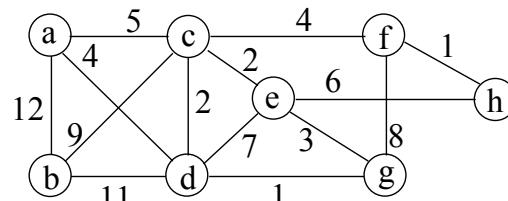
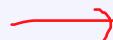
$key[v] = w(u, v)$;

Prim's algorithm

```

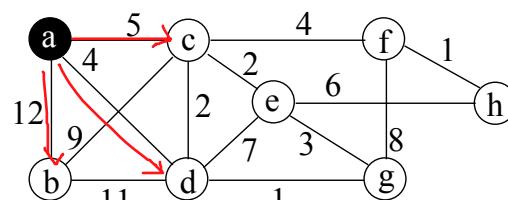
while ( $Q \neq \emptyset$ )
     $u = \text{ExtractMin}(Q);$ 
    for each  $v$  adjacent to  $u$ 
        if ( $(v \in Q)$ 
            & ( $w(u, v) < \text{key}[v]$ ))
             $p[v] = u;$ 
             $\text{key}[v] = w(u, v);$ 

```



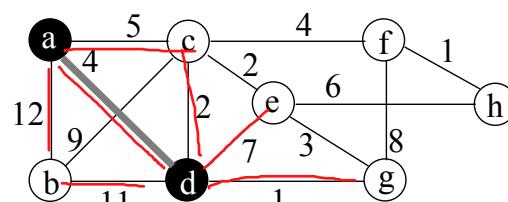
v	a	b	c	d	e	f	g	h
k	0	∞						
p	nil	?	?	?	?	?	?	?

$Q = [a, b, c, d, e, f, g, h]$



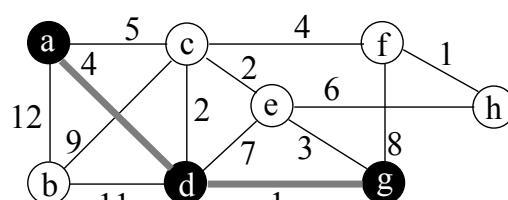
v	a	b	c	d	e	f	g	h
k	∞	12	5	4	∞	∞	∞	∞
p	nil	a	a	a	?	?	?	?

$Q = [d, c, b, e, f, g, h]$



v	a	b	c	d	e	f	g	h
k	∞	11	2	∞	7	∞	1	∞
p	nil	d	d	a	d	?	d	?

$Q = [g, c, e, b, f, h]$



v	a	b	c	d	e	f	g	h
k	∞	11	2	∞	3	8	∞	∞
p	nil	d	d	a	g	g	d	?

$Q = [c, e, f, b, h]$

while ($Q \neq \emptyset$)

$u = \text{ExtractMin}(Q)$;

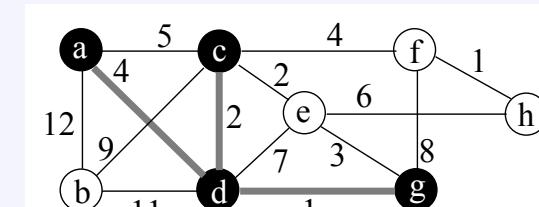
for each v adjacent to u

if ($(v \in Q)$

 & ($w(u, v) < \text{key}[v]$))

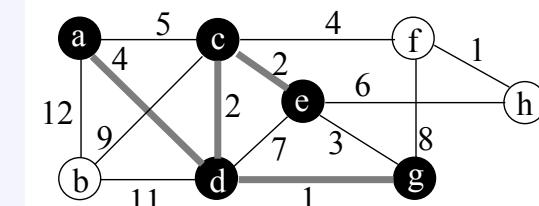
$p[v] = u$;

$\text{key}[v] = w(u, v)$;



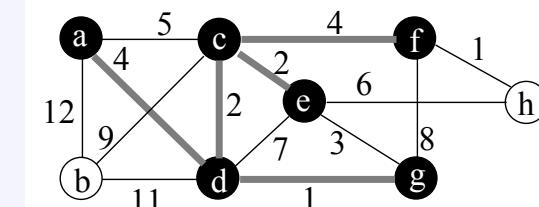
v	a	b	c	d	e	f	g	h
k	∞	9	∞	∞	2	4	∞	∞
p	nil	c	d	a	c	c	d	?

$Q = [e, f, b, h]$



v	a	b	c	d	e	f	g	h
k	∞	9	∞	∞	∞	4	∞	6
p	nil	c	d	a	c	c	d	e

$Q = [f, h, b]$



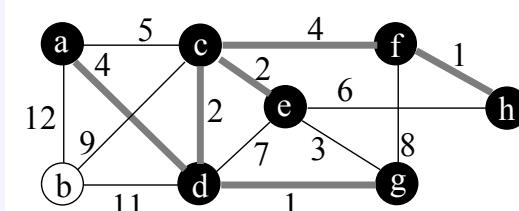
v	a	b	c	d	e	f	g	h
k	∞	9	∞	∞	∞	4	∞	1
p	nil	c	d	a	c	c	d	f

$Q = [h, b]$

```

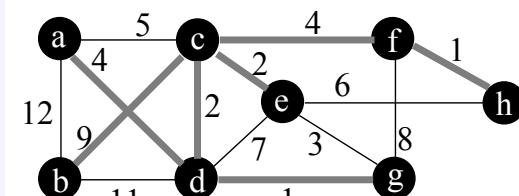
while ( $Q \neq \emptyset$ )
   $u = \text{ExtractMin}(Q)$ ;
  for each  $v$  adjacent to  $u$ 
    if ( $(v \in Q)$ 
      & ( $w(u, v) < \text{key}[v]$ ))
       $p[v] = u$ ;
       $\text{key}[v] = w(u, v)$ ;

```



v	a	b	c	d	e	f	g	h
k	∞	9	∞	∞	∞	∞	∞	∞
p	nil	c	d	a	c	c	d	f

Q=[b]



v	a	b	c	d	e	f	g	h
k	∞	∞	9	∞	∞	∞	∞	∞
p	nil	c	d	a	c	c	d	f

Q=[]

Prim's algorithm : running time

The running time of Prim's algorithm is $O(E \lg V)$.

Building the priority queue using min heap can be done in $O(V)$.

The **while** loop is performed V times

- ▶ ExtractMin cost $O(\lg V)$, total $O(V \lg V)$
- ▶ The cost of Prim's algorithm is driven by the **for** loop
 - ▶ This **for** loop is executed in total $2|E|$ times (the sum of the length of the adjacency list)
 - ▶ Each time it is executed, it can potentially change the value of $\text{key}[v]$
 - ▶ which means an update of the priority queue that cost $O(\lg V)$
- ▶ Total cost of the **for** loop is $O(E \lg V)$
- ▶ The total cost of the **while** loop is $O(V \lg V + E \lg V) \in O(E \lg V)$

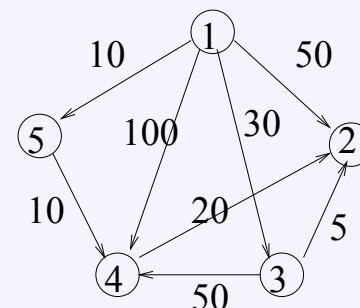
See also the analysis in Cormen, on page 636

Shortest Paths Problem

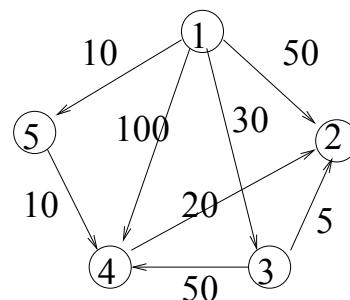
Let $G = \{V, A\}$ be a connected **directed** graph where V is the set of nodes and A is the set of arcs.

- ▶ Each arc $a \in A$ has a nonnegative length.
- ▶ One of the node is designated as the **source** node.
- ▶ The problem is to determine the length of the shortest path from the **source node** to each of the other nodes of the graph

This problem can be solved by a greedy algorithm called **Dijkstra's algorithm**



Dijkstra's algorithm



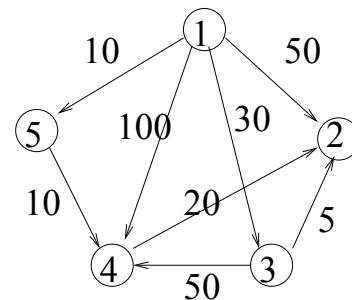
Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

C is the candidate set ($V \setminus$ source node)

S be the set of selected nodes, i.e. the nodes whose minimal distance from the source is already known

A path from the source to another node is **special** if all intermediate nodes along the path belong to S

Dijkstra's algorithm



Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

Let D be an array such that at each step of the algo., D contains the length of the shortest special path to each node in the graph

When a new node v is added to S , the shortest special path to v is also the shortest of all the paths to v

Dijkstra's algorithm

Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

```

graph TD
    1((1)) -- 10 --> 5((5))
    1 -- 50 --> 2((2))
    1 -- 100 --> 4((4))
    2 -- 30 --> 3((3))
    3 -- 5 --> 2
    5 -- 10 --> 4
    4 -- 20 --> 3
    
```

Algorithm Dijkstra(G)

$$C = \{2, 3, \dots, n\} \quad S = V \setminus C$$

for $i = 2$ **to** n **do** $D[i] = L[1, i]$

repeat $n - 1$ **times**

$v =$ some element of C minimizing $D[v]$

$C = C \setminus \{v\}$

for each $w \in C$ **do**

$D[w] = \min(D[w], D[v] + L[v, w])$

return D

Dijkstra's algorithm

Step	v	C	D	S
init		{2,3,4,5}	{50,30,100,10}	{1}
1	5	{2,3,4}	{50,30,20,10}	{1,5}
2	4	{2,3}	{40,30,20,10}	{1,4,5}
3	3	{2}	{35,30,20,10}	{1,3,4,5}
				{1,2,3,4,5}

```

graph TD
    1((1)) -- 10 --> 5((5))
    1 -- 50 --> 2((2))
    1 -- 10 --> 4((4))
    2 -- 30 --> 3((3))
    2 -- 5 --> 4
    3 -- 20 --> 4
    4 -- 100 --> 5
  
```

Algorithm Dijkstra(G)

$$C = \{2, 3, \dots, n\} \quad S = V \setminus C$$

for $i = 2$ **to** n **do** $D[i] = L[1, i]$

repeat $n - 1$ **times**

$v = \text{some element of } C \text{ minimizing } D[v]$

$C = C \setminus \{v\}$

for each $w \in C$ **do**

$D[w] = \min(D[w], D[v] + L[v, w])$

return D

Running time $O(|V|^2)$.

Encoding data : Huffman algorithm

Alphabetic characters used to be stored in *ASCII* on computers, which requires 7 bits per character.

ASCII is a **fixed-length code**, since each character requires the same number of bits to store.

Encoding data : Huffman algorithm

Notice that some characters, (e.g. **q**, **x**, **z**, **v**) are rare, and others (e.g. **e**, **s**, **t**, **a**) are common.

It might make more sense to use less bits to store the common characters, and more bits to store the rare characters.

An encoding that does this is called a [variable-length code](#).

A code is called [optimal](#) if the space required to store data with the given distribution is a minimum.

Optimal codes are important for many applications.

Variable Length Code Example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

Thus, the string *treat* is encoded as 110111011

The problem with this encoding is that the string could also be *ktve*.

Variable Length Code Example

Assume a file has the following distribution of characters.

Letter	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

One good encoding might be the following :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	10	11	101	1	01	001	110	011

With this code, we have to somehow keep track of which letter is which.

(e.g. 11_01_1_10_11)

To do this requires more space, and may make the code worse than a fixed-length code. Rather we use a [prefix code](#).

Prefix Codes

A code in which no word is a prefix of another word.

To encode a string of data, concatenate the codewords together.

To decode, just read the bits until a codeword is recognized.

Since no codeword is a prefix of another, this works.

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

Notice that no codeword is the prefix of another.

Now, we can encode *treat* as *1001010100100*, and it is uniquely decodable.

Decoding prefix codes

Decode *01111011010010100100* based on the following decoding table :

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

The most obvious way is to read one bit, see if it is a character, read another, see if the two are a character, etc. : *not very efficient.*

Decoding prefix codes

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

A better way is to represent the encoding of each letter as a path a binary tree :

- ▶ Each leaf node stores a character.
- ▶ A '0' means go to the left child
- ▶ A '1' means go to the right child
- ▶ The process continues until a leaf is found.

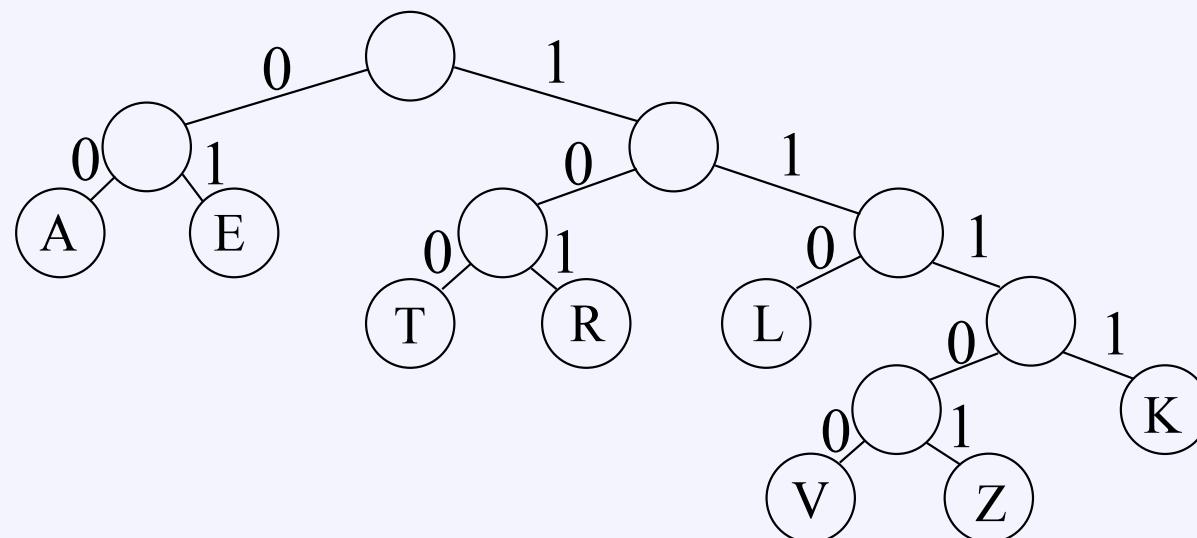
Any code represented this way is a prefix code. Why ?

Decoding prefix code

Decode *01111011010010100100*

Letters	A	T	V	E	R	Z	K	L
Frequency	20	15	3	23	19	2	7	13
Encoding	00	100	11100	01	101	11101	1111	110

We will represent the data by the following binary tree :



It is now not too hard to see that the answer is *ezrarat*.

Constructing Codes

Prefix codes make encoding and decoding data very easy.

It can be shown that optimal data compression using a character code can be obtained using a prefix code.

How can we construct such a code ?

Huffman code is an algorithm to construct prefix codes

Huffman Code

A Huffman code can be constructed by building the encoding tree from the bottom-up in a greedy fashion.

Since the less frequent nodes should end up near the bottom of the tree, it makes sense that we should consider these first.

We'll see an example, then the algorithm.

Huffman Code Example

Suppose I want to store this very long word in an electronic dictionary : *floccinaucinihilipilification*.

I want to store it using as few bits as possible.

The frequency of letters, sorted in decreasing order, is as follows :

i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

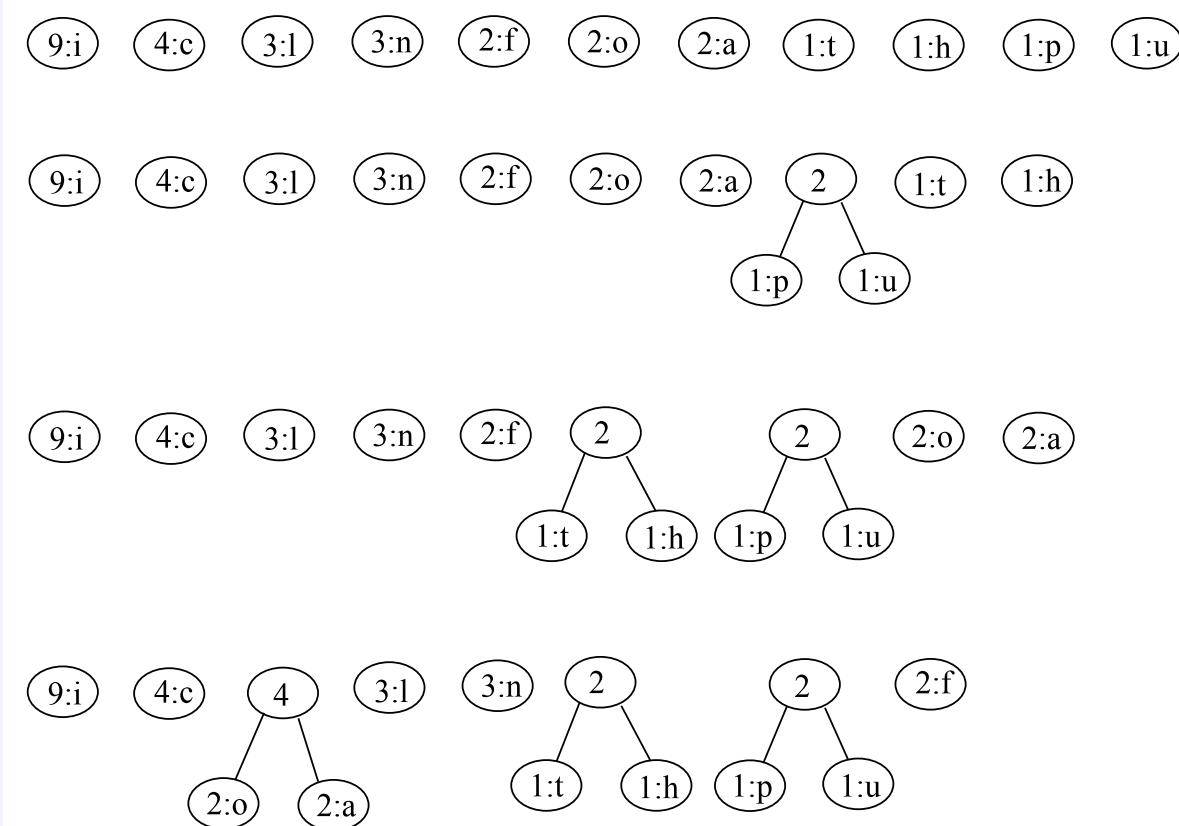
Huffman Code Example

i	c	l	n	f	o	a	t	h	p	u
9	4	3	3	2	2	2	1	1	1	1

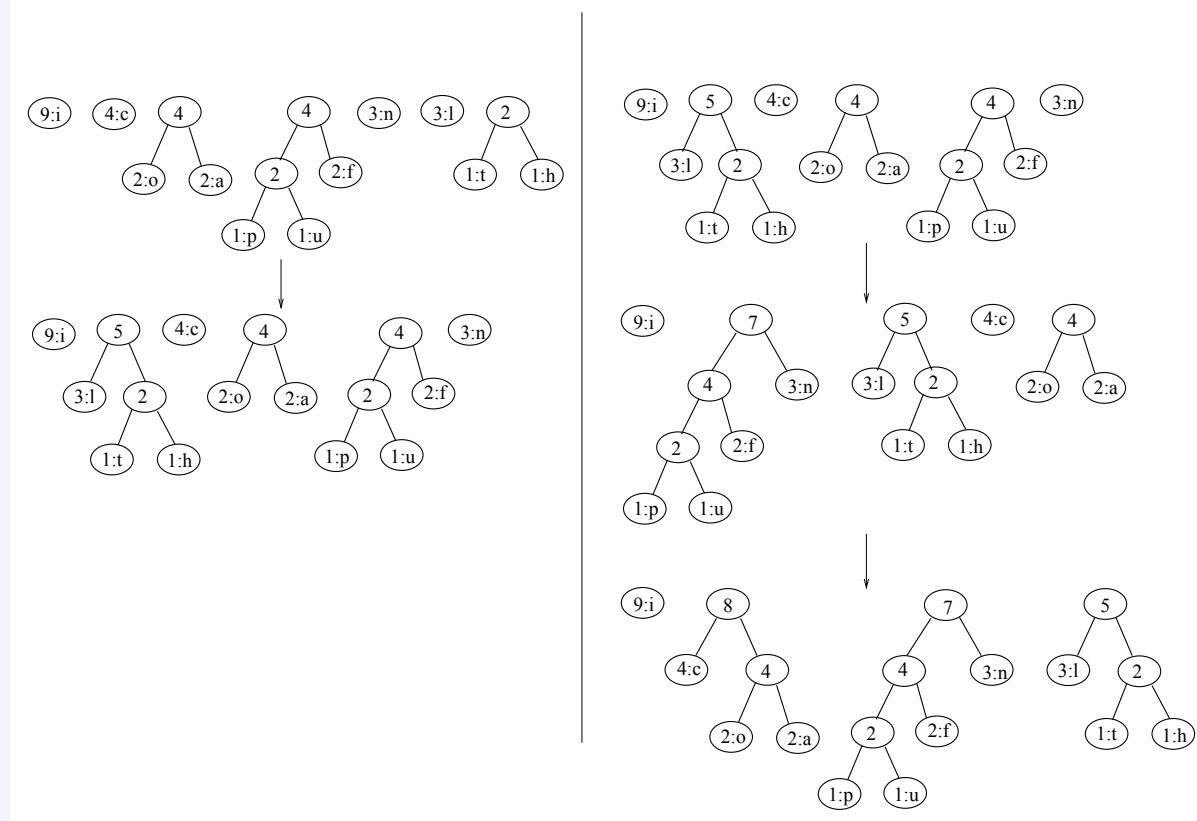
The algorithm works sort of like this :

- ▶ Consider each letter as a node in a not-yet-constructed tree.
- ▶ Label each node with its letter and frequency.
- ▶ Pick two nodes x and y of least frequency.
- ▶ Insert a new node, and let x and y be its children. Let its frequency be the combined frequency of x and y .
- ▶ Take x and y off the list.
- ▶ Continue until only 1 node is left on the list.

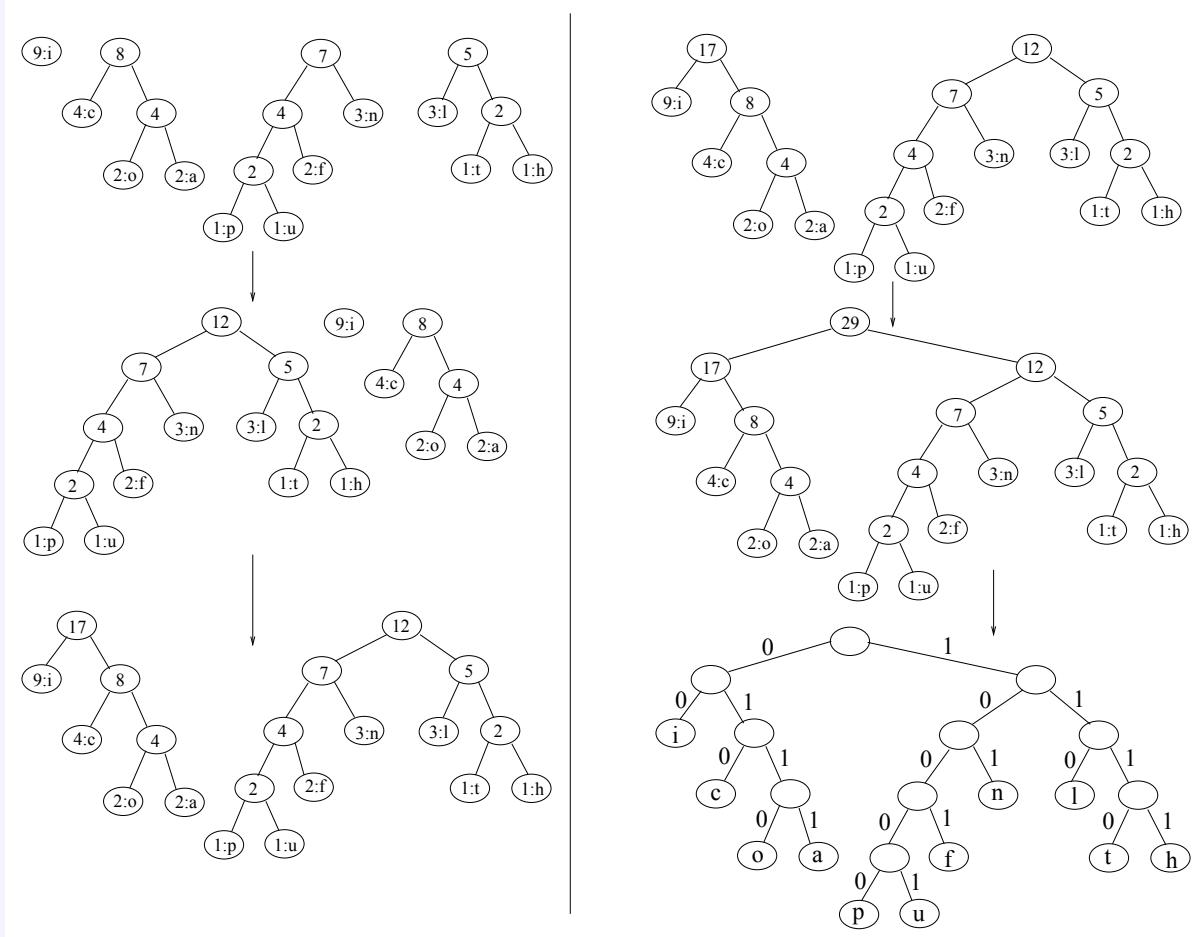
Huffman Example



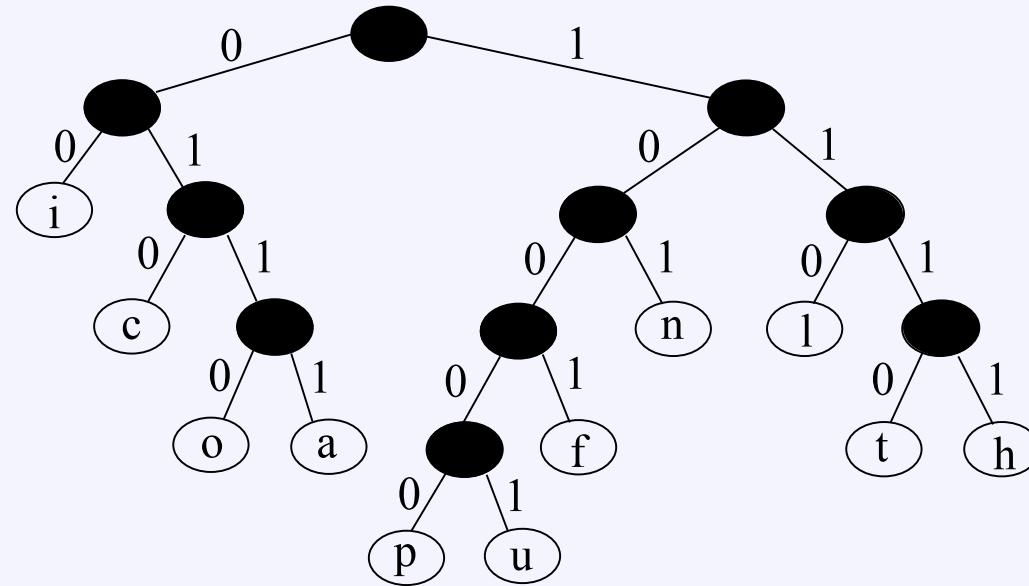
Huffman's encoding



Huffman's encoding



We can now list the code :



i (9)	00	f (2)	1001
c (4)	010	t (1)	1110
n (3)	101	h (1)	1111
l (3)	110	p (1)	10000
o (2)	0110	u (1)	10001
a (2)	0111		