## Introduction to Algorithms

Department of Computer Science and Engineering

East China University of Science and Technology

**Recursion algorithm**

**&**

*divide-and-conquer*

## Recursion

- Recursion occurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic.
- The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition. While this apparently defines an infinite number of instances (function values), it is often done in such a way that no loop or infinite chain of references can occur.

## Formal definitions

In mathematics and computer science, a class of objects or methods exhibit recursive behavior when they can be defined by two properties:

1.A simple base case (or cases)—a terminating scenario that does not use recursion to produce an answer

2.A set of rules that reduce all other cases toward the base case

- For example, the following is a recursive definition of a person's ancestors:
- One's parents are one's ancestors (base case).
- The ancestors of one's ancestors are also one's ancestors (recursion step).

- Substitution method
- 1.Guess the form of the solution.
- 2.Verify by induction.
- 3.Solve for constants.
- The most general method:
- Example:T(n) = 4T(n/2) + n
- [Assume that T(1) = Θ(1).] •
- Guess O(n3). (Prove O and Ω separately.)
- Assume that T(k) ≤ck3for k< n .
- Prove T(n) ≤cn3by induction

## The Fibonacci sequence

**Fibonacci numbers**

**Recursive definition:**

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

0 1 1 2 3 5 8 13 21 34 ⋯

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

# The Fibonacci sequence

- The Fibonacci sequence is a classic example of recursion:

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

- Fib （0）= 0 as base case 1,
- Fib （1）= 1 as base case 2,
- For all integers n > 1， Fib （n）:= Fib （n − 1）+ Fib （n − 2）. {For all integers }n>1
- Many mathematical axioms are based upon recursive rules. For example, the formal definition of the natural numbers by the Peano axioms can be described as: 0 is a natural number, and each natural number has a successor, which is also a natural number. By this base case and recursive rule, one can generate the set of all natural numbers.
- Recursively defined mathematical objects include functions, sets, and especially fractals.

**Example** The Fibonacci sequence

无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。它可以递归地定义为：

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

边界条件

递归方程

第n个Fibonacci数可递归地计算如下：

```
int fibonacci(int n)
  {
     if (n <= 1) return 1;
     return fibonacci(n-1)+fibonacci(n-2);
  }
```



```
• [java] view plain copy
• public class Fibonacci {
• atic void main(String[] args) {
•       Fibonacci fibonacci=new Fibonacci();
•       int result=fibonacci.fib(5);
•           System.out.println(result);
•   }
•   public int fib(int index){
•    if(index==1||index==2){
•      return 1;
•     }else{
•        return fib(index-1)+fib(index-2);
•     }
```

```
if(index==1 || index==2)
{
        return 1;
}
```

# Recursion (computer science)

- A common method of simplification is to divide a problem into subproblems of the same type.
- As a computer programming technique, this is called divide and conquer and is key to the design of many important algorithms.
- Divide and conquer serves as a top-down approach to problem solving, where problems are solved by solving smaller and smaller instances.
- A contrary approach is dynamic programming.
- This approach serves as a bottom-up approach, where problems are solved by solving larger and larger instances, until the desired size is reached.

# Recursion (computer science)

- A classic example of recursion is the definition of the factorial function, given here in C code:

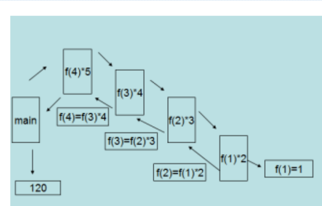
```
• unsigned int factorial(unsigned int n) {
•   if (n == 0) {
•       return 1;
•   } else {
•       return n * factorial(n - 1);
•   }
• }
```

# Recursion-----n!

```
function fib(n)
    if n <= 1 return n
    return fib(n − 1) + fib(n − 2)
```

```
 5!=?
description:递归求n的阶乘
result=factorial_Five.factorial(5);
public int factorial(int index){
      if(index==1){
      return 1;
      }else{
       return factorial(index-1)*index;
      }
   }

}
```



## 5!=?

```
function fib(n)
    if n <= 1 return n
    return fib(n − 1) + fib(n − 2)
```

- Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:
- 1.fib(5)
- 2.fib(4) + fib(3)
- 3.(fib(3) + fib(2)) + (fib(2) + fib(1))
- 4.((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
- 5.(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

- This technique of saving values that have already been calculated is called memoization; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

- In the bottom-up approach, we calculate the smaller values of fib first, then build larger values from them. This method also uses O(n) time since it contains a loop that repeats n − 1 times, but it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(n) space to store the map.

- In both examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.
- Note that the above method actually takes Ω ( n $^{2}$ ) (n^{2})} ) time for large n because addition of two integers with Ω ( n ) bits each takes Ω ( n ) time.

- Also, there is a closed form for the Fibonacci sequence, known as Binet's formula, from which the n
- n-th term can be computed in approximately *O ( n (logn )*$^{2}$ *)* O(n(\log n)^{2}) time, which is more efficient than the above dynamic programming technique.
- However, the simple recurrence directly gives the matrix form that leads to an approximately *O ( n (logn )* O(nlog n) algorithm by fast matrix exponentiation.

例：假设S（n）是前n个整数的和，那么S（1）= 1，并且我们可以将S（n）写成S（n）= S（n-1）+ n。
根据递归公式，我们可以得到对应的递归函数：

```
int S(int n) //求前n个整数的和
{
    if (n == 1)
        return 1;
    else
        return S(n-1) + n;
}
```

流程图

函数由递归公式得到，应该是好理解的，要想求出S（n），得先求出S(n-1)，递归终止的条件（递归出口）是(n == 1)。

前2例中的函数都可以找到相应的非递归方式定义：

$$n! = 1\cdot 2\cdot 3\cdot\cdots\cdot(n-1)\cdot n$$

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

**但是, 有的函数却无法找到非递归定义, 比如Ackerman函数.**

## The *divide-and-conquer* design paradigm

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

## Binary search

- Example: Find 9
- 3 5 7 8 9 12 15
- Find an element in a sorted array:
- 1.Divide:Check middle element.
  2.Conquer:Recursively search 1subarray.
  3.Combine:Trivial

3 5 7 8 9 12 15

## Divide-and-Conquer

- **Recursive in structure**
- **To solve P:**

– **Divide P into smaller problems $P_1$, $P_2$, …, $P_k$.**

– **Conquer by solving the (smaller) subproblems recursively. Recursively**

– **Combine the solutions to $P_1$, $P_2$, …, $P_k$ *into the solution for P***

## 7 Quicksort

- Proposed by 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 ≤ *x* ≤ elements in upper subarray.


***2. Conquer:*** Recursively sort the two subarrays.

***3. Combine:*** Trivial.

## Pseudocode for quicksort

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

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

## Partitioning subroutine

The key to the algorithm is the PARTITION procedure,which rearranges the subarray A[p..r] in place.

```
PARTITION(A, p, q)  ⊳ A[p . . q]
   x ← A[p]          ⊳ pivot = A[p]
    i ← p
   for j ← p + 1 to q
       do if A[ j] ≤ x
                   then i ← i + 1
                 exchange A[i] ↔ A[ j]
   exchange A[p] ↔ A[i]
   return i
```

***invariant:***

| | | | |
|---|---|---|---|
| $x$ | $\le x$ | $\ge x$ | ? |

$p$ $i$ $j$ $q$

## Example of partitioning

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

$i$ $j$

## Example of partitioning

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

$i$ $\longrightarrow j$

## Example of partitioning

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

$i$ $\longrightarrow j$

## Example of partitioning

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

$\longrightarrow i$ $j$

## Example of partitioning

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

$i$ $\longrightarrow j$

## Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| | $i$ | | | $\longrightarrow j$ | | | |

## Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| | | $\longrightarrow i$ | | | $j$ | | |

## Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| | | $i$ | | | | $\longrightarrow j$ | |

## Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 |
| | | | $\longrightarrow i$ | | | $j$ | |

## Example of partitioning

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 |
| | | | $i$ | | | | $\longrightarrow j$ |

## Example of partitioning

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 | |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 | |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 | |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 | |
| | | | $i$ | | | | | $\longrightarrow j$ |

## Example of partitioning

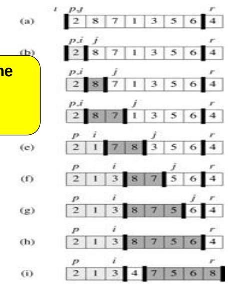
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 13 | 5 | 8 | 3 | 2 | 11 |
| 6 | 5 | 13 | 10 | 8 | 3 | 2 | 11 |
| 6 | 5 | 3 | 10 | 8 | 13 | 2 | 11 |
| 6 | 5 | 3 | 2 | 8 | 13 | 10 | 11 |
| 2 | 5 | 3 | 6 | 8 | 13 | 10 | 11 |

## Partitioning subroutine

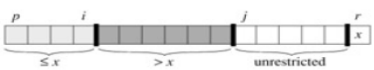
- PARTITION(A,p,r)
- x← A[r]
- i← p-1
- **For** j← p to r-1
- **do if** A [j] ≤x
- **then** i ←i+1
- exchange A[i] ←→A[j]
- exchange A[i+1] ←→A[r]
- **return** i+1

**Running time**

**= *O*(*n*) for *n* elements.**



## Anlysis of the partition:



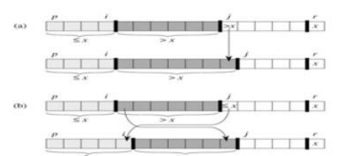

- **If $p \le k \le i$, then $A[k] \le x$.**
- **If $i + 1 \le k \le j - 1$, then $A[k] > x$.**
- **If $k = r$, then $A[k] = x$**
- **If $j \le k \le r-1$, then $A[k]$ can take on any values**

- **Initialization: the first two conditions of the loop invariant are trivially satisfied.**
- **Maintenance:there are two cases to consider:**

**1.(a) shows what happens when *A*[*j*] > *x*;**

**2.(b) shows what happenswhen *A*[*j*] ≤ *x*;**

- **Termination:At termination, *j* = *r*. and we have partitioned the values in the array into three sets: those less than or equal to *x*, those greater than *x*, and a singleton set containing *x*.**



## Analysis of quicksort

- 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.

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

## Worst-case recursion tree

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

## Worst-case recursion tree

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

$T(n)$

## Worst-case recursion tree

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

$cn$

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

## Worst-case recursion tree

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

$cn$

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

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

## Worst-case recursion tree

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

$cn$

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

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

$T(0)$

$\Theta(1)$

## Best-case analysis

*(For intuition only!)*

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

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

$= \Theta(n \lg n)$ (same as merge sort)

What if the split is always 1/10:9/10?

$T(n) = T(1/10n) + T(9/10n) + \Theta(n)$

What is the solution to this recurrence?

## Analysis of “almost-best” case

$T(n)$

## Analysis of “almost-best” case

$cn$

$T(1/10n)$ $T(9/10n)$

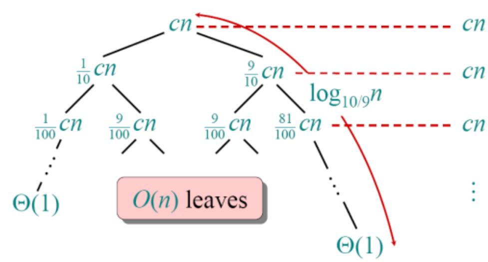
## Analysis of “almost-best” case

$cn$

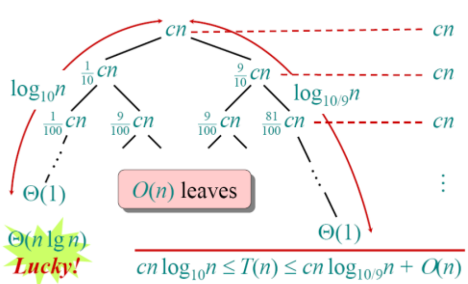
$1/10cn$ $9/10cn$

$T(1/100n)$ $T(9/100n)$ $T(9/100n)$ $T(81/100n)$

## Analysis of “almost-best” case




## Analysis of “almost-best” case




## More intuition

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

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

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

Solving:

$L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)$

$= 2L(n/2 - 1) + \Theta(n)$

$= \Theta(n \lg n)$

## Randomized quicksort

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

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

## Randomized quicksort analysis

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

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

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

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

## Analysis (continued)

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

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

## Calculating expectation

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

T

## Calculating expectation

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

Linearity of expectation.

## Calculating expectation

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

Independence of $X_k$ from other random choices.

## Calculating expectation

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

## Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right]$$
$$= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))]$$
$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$
$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$
$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.

## Hairy recurrence

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

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

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

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

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

## Substitution method

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

S

## Substitution method

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

## Substitution method

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

E›

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

## Substitution method

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

if *an*/4 dominates the Θ(*n*).

## Quicksort in practice

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

## Conclusion

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

## Example

Problem: sorting {12,3,25,5,31,7,9,1,10}

- Divide: it is too big to solve, if it is smaller we may solve it, so we divide it into two problem:
- Sub-problem1: sorting{12,3,25,5,31}
- Sub-problem2: sorting{7,9,1,10}

## Example

- Conquer: we use some methods (any kind of) to solve it and get the sub-problems' solutions, that means we sort the two sub-problem:

sorting{12,3,25,5,31}

Solution of sub-problem1:{3,5,12,25,31}

sorting{7,9,1,10}

Solution of sub-problem2:{1,7,9,10}

## Example

- Combine: we should use these two solutions to construct the solution of the original problem.

## Designing algorithms

**Divide** the problem into a number of subproblems.

**Conquer** the subproblems by solving them recursively.

**Combine** the subproblem solutions to give a solution to the original problem.

## Merge Sort Algorithm

- **Using divide-and-conquer, we can obtain a merge-sort algorithm**

– **Divide:** Divide the *n elements into two subsequences of n/2 elements each.*
– **Conquer:** Sort the two subsequences recursively.
– **Combine:** Merge the two sorted subsequences to produce the sorted answer.

- Assume we have procedure **MERGE(*A, p, q, r*)** *which* merges sorted *A[p...q] with sorted A[q+1..r] in (r - p) time.*

## Merge Sort (1) By Divide-and-Conquer

**基本思想：**将元素分成2个子集合，分别对子集合进行排序，最终将排好序的子集合合并为有序集合。n=1时中止。

```
void MergeSort(Type a[ ], int left, int right)
  {
    if (left<right) {//至少有2个元素
    int i=(left+right)/2;  //取中点
    MergeSort(a, left, i);
    MergeSort(a, i+1, right);
    merge(a, b, left, i, right);  //合并到数组b
    copy(a, b, left, right);    //复制回数组a
    }
  }
```

**复杂度分析**

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

T(n)=O(nlogn) 渐进意义下的最优算法

## MergeSort

**思想：**相邻元素（子数组段）两两配对，用合并算法将其排序，直至整个数组排好序。

算法mergeSort的递归过程可以消去，

初始序列 [49] [38] [65] [97] [76] [13] [27]

第一步 [38 49] [65 97] [13 76] [27]

第二步 [38 49 65 97] [13 27 76]

第三步 [13 27 38 49 65 76 97]

77

## Merge Sort(3) 自然排序

➢自然排序是合并排序算法的一个变形。
e. g. {4, 8, 3, 7, 1, 5, 6, 2}

➢1. 用一次对初始数组的扫描找出所有已排好序的子数组段; {4, 8}, {3, 7}, {1, 5, 6}, {2}

➢2. 将相邻排好序的数组两两合并, 直至完成整个数组的排序. {3, 4, 7, 8}, {1, 2, 5, 6}, …………

📖最坏时间复杂度：**O(nlogn)**
📖平均时间复杂度：**O(nlogn)**
📖最好时间复杂度：**O(n)** （初始已排好序）

78

## Merge sort

➢A sorting algorithm based on **divide and conquer**.

➢Its worst-case running time has a **lower order** of growth than insertion sort.

**MERGE-SORT** $A[1 \, . \, . \, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\, 1 \, . \, . \quad n/2 \,]$ and $A[n/2 \quad +1 \, . \, . \, n \,]$ .
3. "***Merge***" the 2 sorted lists.
   ***Key subroutine:*** **MERGE**

Me

## Merging two sorted

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

## Merging two sorted

| | |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

1

## Merging two sorted

| | | | |
|---|---|---|---|
| 20 | 12 | 20 | 12 |
| 13 | 11 | 13 | 11 |
| 7 | 9 | 7 | 9 |
| 2 | 1 | 2 | |

1

## Merging two sorted

| | | | |
|---|---|---|---|
| 20 | 12 | 20 | 12 |
| 13 | 11 | 13 | 11 |
| 7 | 9 | 7 | 9 |
| 2 | 1 | 2 | |

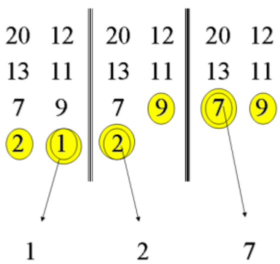
1 2

## Merging two sorted

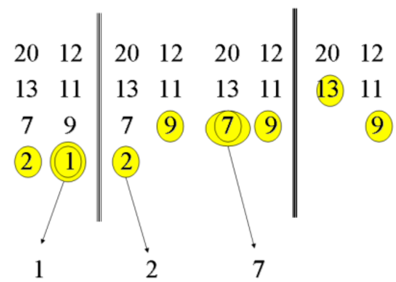
| | | | | | |
|---|---|---|---|---|---|
| 20 | 12 | 20 | 12 | 20 | 12 |
| 13 | 11 | 13 | 11 | 13 | 11 |
| 7 | 9 | 7 | 9 | 7 | 9 |
| 2 | 1 | 2 | | | |

1 2

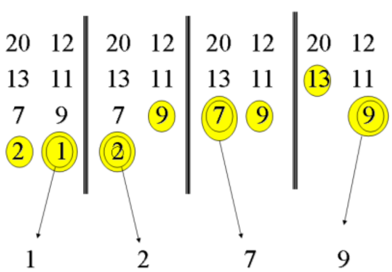
## Merging two sorted arrays



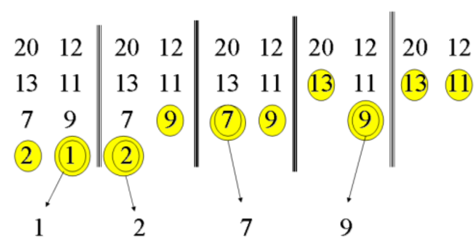

## Merging two sorted arrays



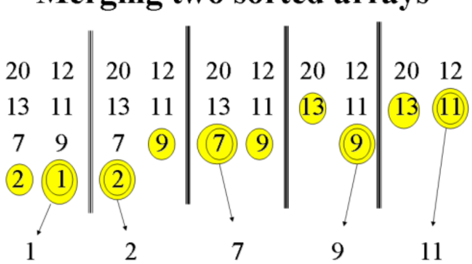

## Merging two sorted arrays



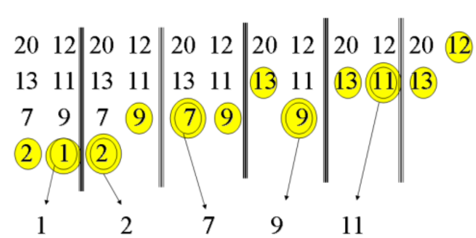

## Merging two sorted arrays



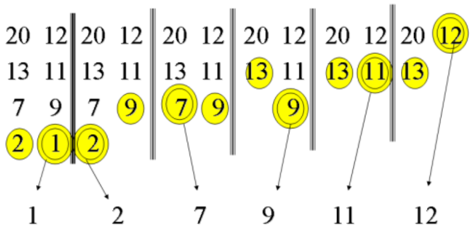

## Merging two sorted arrays



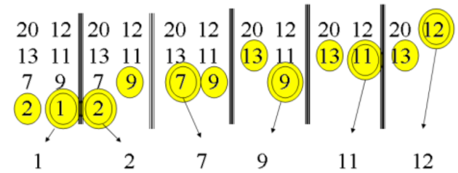

## Merging two sorted arrays



## Merging two sorted arrays



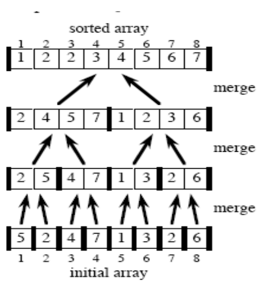
## Merging two sorted arrays



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

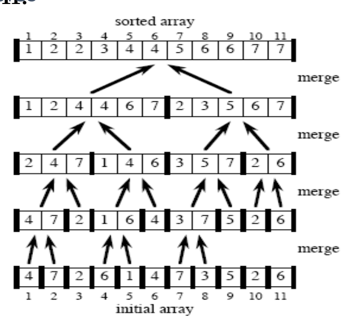
Examples

$n = 8$:




Examples

$n = 11$:




**Merge-Sort (A, p, r)**

**INPUT: a sequence of *n numbers stored in array A***

**OUTPUT: an ordered sequence of *n numbers***

**MERGESORT**

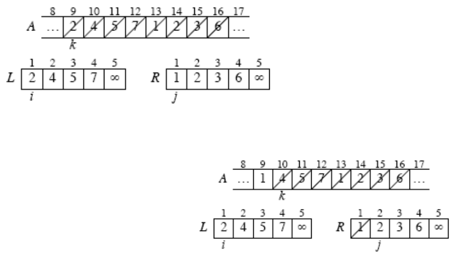
```
MERGE-SORT(A,p,r)
if  p < r
    then  q ← ⌊(p+r)/2⌋
          MERGE-SORT (A, p, q)
          MERGE-SORT (A, q+1, r)
          MERGE (A, p, q, r)
```

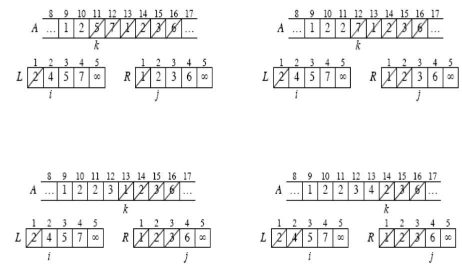
## *Pseudocode*

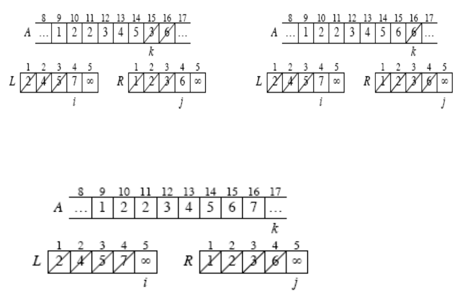
- MERGE($A, p, q, r$)
- $n_1 \leftarrow q - p + 1$
- $n_2 \leftarrow r - q$
- create arrays $L[1 . . n_1 + 1]$ and $R[1 . . n_2 + 1]$
- for $i \leftarrow 1$ to $n_1$
-     do $L[i] \leftarrow A[p + i - 1]$
- for $j \leftarrow 1$ to $n_2$
-     do $R[j] \leftarrow A[q + j]$
- $L[n_1 + 1] \leftarrow \infty$
- $R[n_2 + 1] \leftarrow \infty$
- $i \leftarrow 1$  $j \leftarrow 1$
- for $k \leftarrow p$ to $r$
-     do if $L[i] \le R[j]$
-         then $A[k] \leftarrow L[i]$
-         $i \leftarrow i + 1$
-     else $A[k] \leftarrow R[j]$
-     $j \leftarrow j + 1$

*Running time:*
The first two for loops take $\Theta(n_1 + n_2) = \Theta(n)$ time.
The last for loop makes $n$ iterations, each taking constant time, for $\Theta(n)$ time.
Total time: ***$\Theta(n)$.***

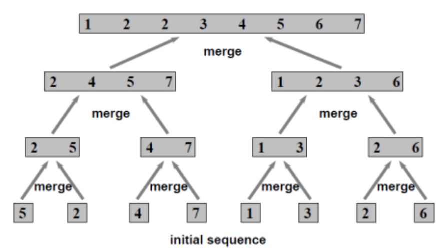
***Example:*** A call of MERGE(9, 12, 16)







**Merge-Sort (A, 1, length[A])**




## Solution!

{5,2,4,7, 1,3,2,6}

- Using the merge algorithm we get the
- solution:{1,2,2,3,4,5,6,7}
- Any left problems?

## Correctness

- Is your solution correct?
- Does your merge algorithm outcome a sorted array in any case?

- ***loop invariant***
  - **Initialization:** It is true prior to the first iteration of the loop.
  - **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination:** When the loop terminates, the invariant.

## Complexity

- How many memories does the algorithm use?
- How many steps does the algorithm use?

## Analyzing divide-and-conquer algorithms

**Let $T(n)$ = running time on a problem of size $n$.**

◆to make analysis cleaner, assume *n is a power of 2*

## Analyzing merge sort

$T(n)$ **MERGE-SORT** $A[1..n]$

$\Theta(1)$ 1. If $n = 1$, done.

$2T(n/2)$ 2. Recursively sort $A[1..n/2]$

Abuse and $A[n/2 + 1..n]$.

$\Theta(n)$ ***3. "Merge"*** the 2 sorted lists

***Sloppiness:*** Should be $T(n/2) + T(n/2)$,
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.

## Left problems

- What is the upper bounds and lower bounds for the sorting problem?
- How to solve the recursive equality(or inequality)?
  $T(n)=2T(n/2)+\Theta(n)$
- How to solve the problem?
  - **Recursion-tree method**
  - **Substitution method**
  - **Master method**

## Recursion- tree method

- **• A recursion tree models the costs (time) of a recursive execution of an algorithm.**
- **• The recursion tree method is good for generating guesses for the substitution method.**
- **• The recursion-tree method promotes intuition, however.**

## Recursion tree

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

## Recursion tree

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

$T(n)$

## Recursion tree

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

$cn$

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

For the original problem, we have a cost of *cn*, plus the two subproblems, each costing *T (n/2)*:

## Recursion tree

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

$cn$

$cn/2$ $cn/2$

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

For each of the size-*n*/2 subproblems, we have a cost of *cn*/2, plus two subproblems, each costing *T (n/4)*:

## Recursion tree

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

$cn$

$cn/2$ $cn/2$

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

c

## Recursion tree

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

$cn$

$cn/2$ $cn/2$

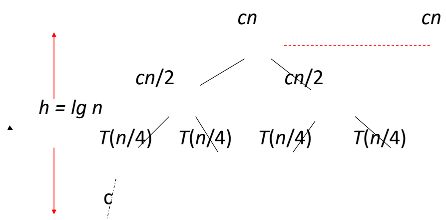
$h = lg\ n$

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

c

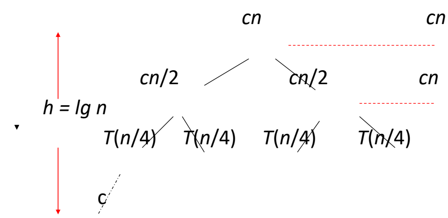
## Recursion tree

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



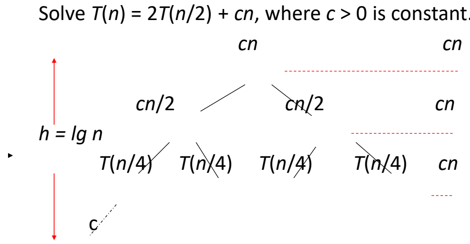

## Recursion tree

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



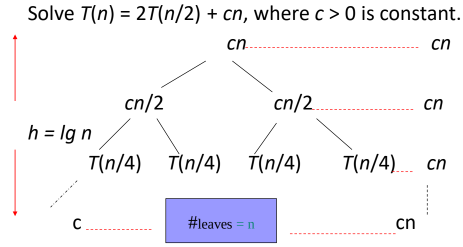

## Recursion tree

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



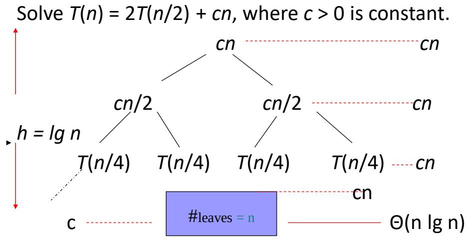

## Recursion tree

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

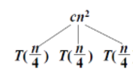
- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n >$ 30 or so.
- Go test it out for yourself!

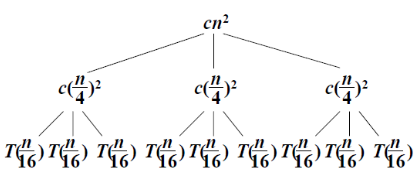
## Example

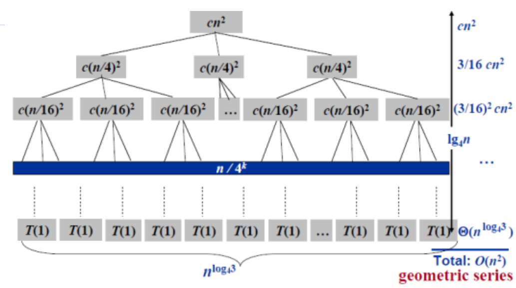
Ex. ***T(n) = T(n/4)+ T(n/2) + n²***

***T(n) = 3T(n/4) + Θ(n²)***

*T(n) = 3T(n/4) + Θ(n²)*




## Substitution method

- • The most general method:
- – Guess the form of the solution.
- – Verify by induction.
- – Solve for constants.

## Example

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

- – Assume that ***T(1) = Θ(1).***
- – Guess O(***n³***).
- – Assume that ***T(k) ≤ ck³ for k < n.***
- – Prove ***T(n) ≤ cn³ by induction.***

## Master Method

- • It provides a "cookbook" method for solving recurrences of the form:

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

- where *$a \geq 1$ and $b > 1$ are constants and $f(n)$ is* an asymptotically positive function

Three common cases

- **Based on the *master theorem*,**
- **Compare $n^{\log_b a}$ vs. $f(n)$:**

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

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

- **Compare $n^{\log_b a}$ vs. f (n):**

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$
- **$f(n)$ and $n^{\log_b a}$ grow at similar rates.**

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

- **Compare $n^{\log_b a}$ vs. f (n):**

3. f (n) = $\Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.

- f (n) grows polynomially faster than $n^{\log_b a}$ and f (n) satisfies the regularity condition that a f (n/b) ≤ c f (n) for some constant c < 1.

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

Examples

*Ex.* *Merge sort*

*$T(n)=2T(n/2)+\Theta(n)$*

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

CASE 2: k=0

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

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

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

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

∴ $T(n) = \Theta(n^2)$.

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

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

CASE 2: $f(n) = \Theta(n^2)$,

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

## Examples

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

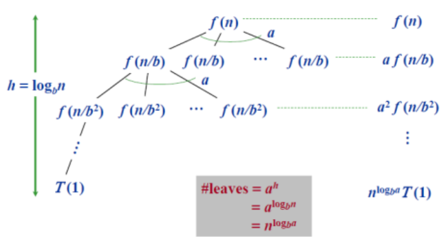
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

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

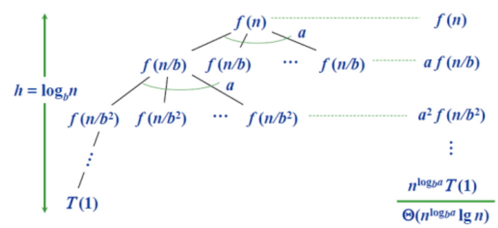
and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

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

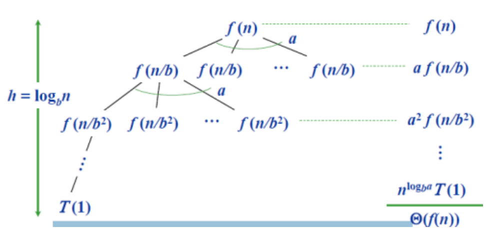
- Recursion tree




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




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




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