

CS430-01

Introduction to Algorithms

## 3. Divide & Conquer Algorithm II

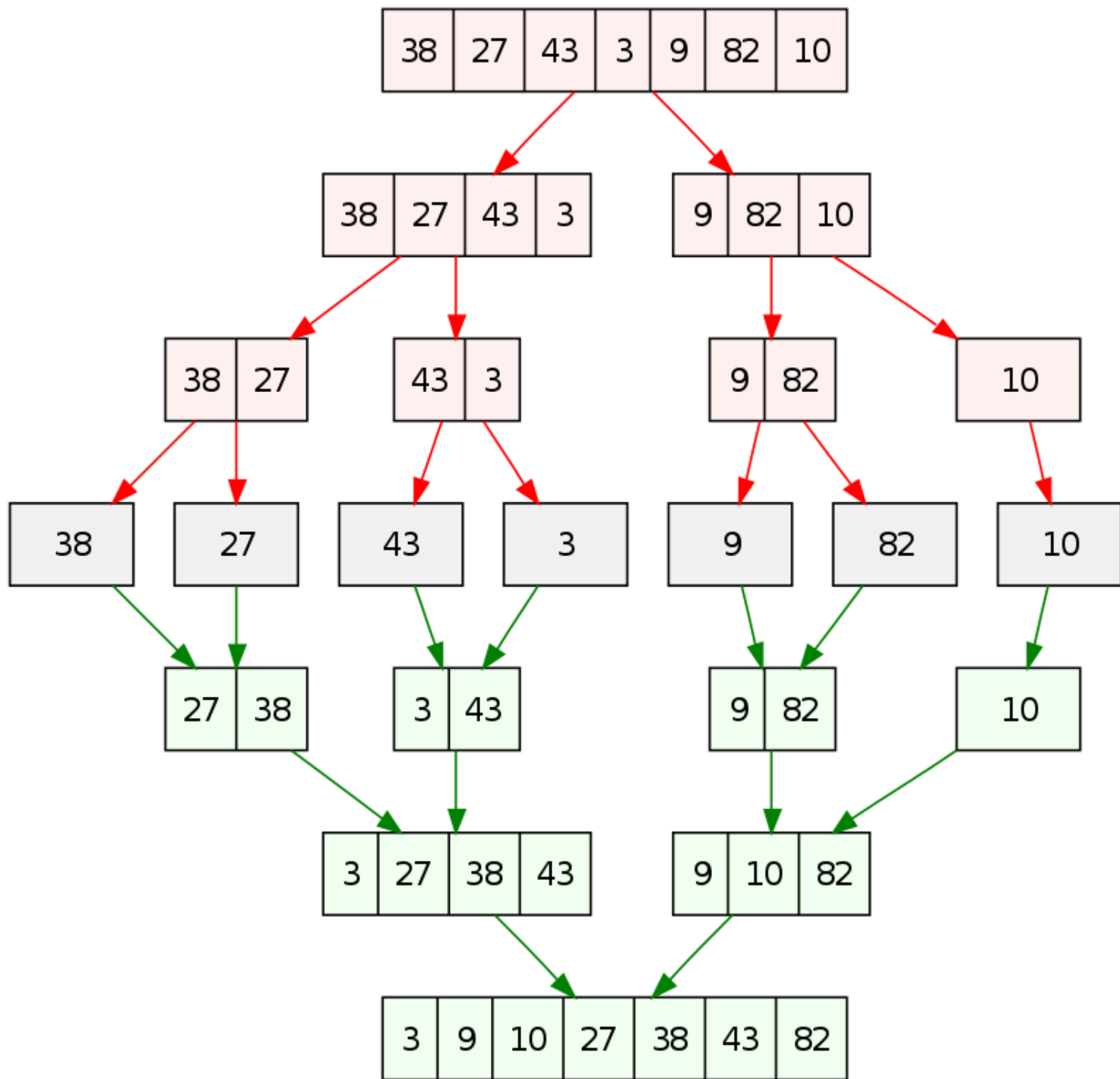
Michael Choi

Dept. of Computer Science

IIT

# D & C algorithms

- **divide and conquer** is an algorithm design paradigm based on **multi-branched recursion**
- A divide-and-conquer algorithm works by **recursively** breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly
- The solutions to the sub-problems are then combined
- This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), [multiplying large numbers](#) (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT) to give a solution to the original problem



# D&C: Advantages

## Solving difficult problems

- Divide and conquer is a powerful tool for solving conceptually difficult problems:
- all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.
- Similarly, decrease and conquer only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height  $n$  to moving a tower of height  $n - 1$ .

# D&C: Advantages

## Algorithm efficiency

- The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.
- an improvement in the **asymptotic cost** of the solution
- if (a) the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size  $n$ , and
- (b) there is a bounded number  $p$  of sub-problems of size  $\sim n/p$  at each stage, then the cost of the divide-and-conquer algorithm will be  $O(n \log_p n)$ .

# D&C: Advantages

## Parallelism

- D&C algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

# D&C: Advantages

## Memory access

- naturally tend to make efficient use of memory caches.
- once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory.

# D&C: Implantation issues

## Recursion

- D&C algorithms are naturally implemented as recursive procedures.
- In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack. A recursive function is a function that calls itself within its definition.



# D&C: Implantation issues

## Stack size

- In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of [stack overflow](#).
- D&C algorithms that are time-efficient often have relatively small recursion depth.
- Stack overflow may be difficult to avoid when using recursive procedures, since many compilers assume that the recursion stack is a contiguous area of memory, and some allocate a fixed amount of space for it.

# D&C: Implantation issues

## Stack size

- Compilers may also save more information in the recursion stack than is strictly necessary, such as return address, unchanging parameters, and the internal variables of the procedure.
- Thus, the risk of stack overflow can be reduced by minimizing the parameters and internal variables of the recursive procedure or by using an explicit stack structure.

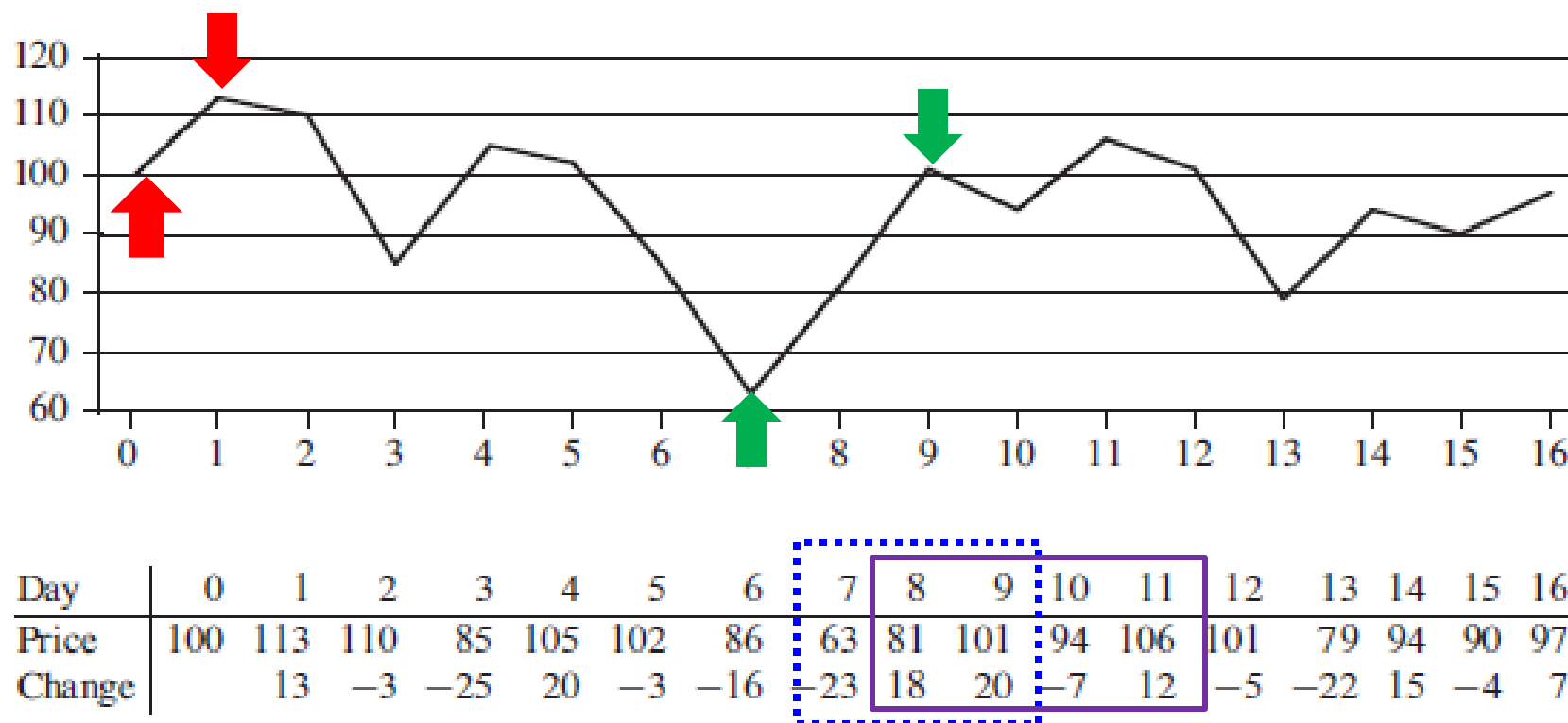
# Maximum-subarray problem

## Investing stock

- “Buy low, Sell high” strategy
- Goal: to achieve maximum profit, buy stocks at the lowest price and sell them at the highest price
- Restriction 1: buy one unit of stock only one time
- Restriction 2: sell one it a later date
- Restriction 3: Buying and selling after close of trading for the day
- Compensation: you are allowed to learn what the price of stock will be in the future
- Next chart shows 17 days of stock prices

# Maximum-subarray problem

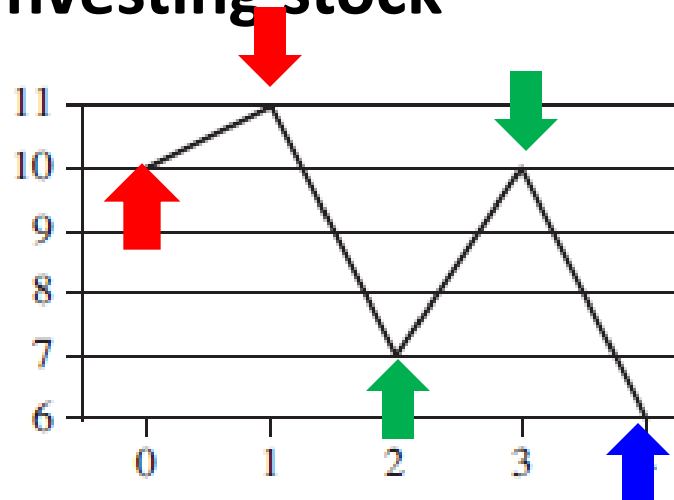
## Investing stock



**Figure 4.1** Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

# Maximum-subarray problem

## Investing stock



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

**Figure 4.2** An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

# Maximum-subarray problem

## Brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of  $n$  days has  $\binom{n}{2}$  such pairs of dates. Since  $\binom{n}{2}$  is  $\Theta(n^2)$ , and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take  $\Omega(n^2)$  time. Can we do better?

# Maximum-subarray problem

## Transformation

Fig 4.1 shows daily change  $\rightarrow$  store these data into array A

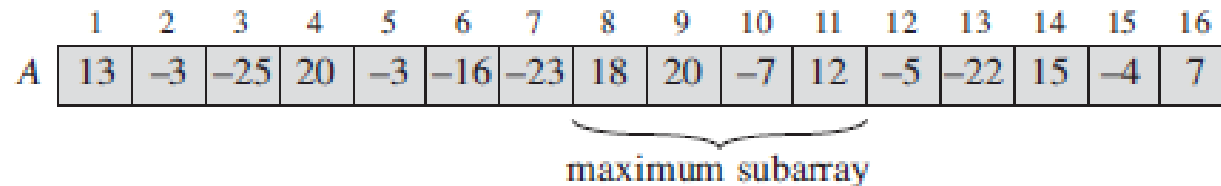


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray  $A[8 \dots 11]$ , with sum 43, has the greatest sum of any contiguous subarray of array A.

Find the nonempty, contiguous subarray A whose values have the largest sum: max subarray

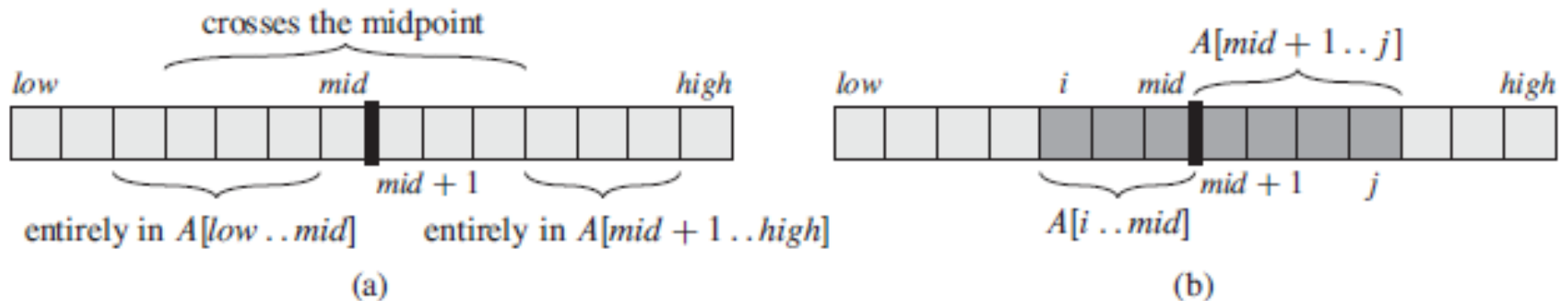
Still  $\Theta(n^2)$

# Using D&C

Find subarray  $A[\text{low}..\text{high}]$  by D&C technique

Find mid location of the array  $A$  and divide as  $A[\text{low}..\text{mid}]$  and  $A[\text{mid}+1..\text{high}]$

- entirely in the subarray  $A[\text{low}..\text{mid}]$ , so that  $\text{low} \leq i \leq j \leq \text{mid}$ ,
- entirely in the subarray  $A[\text{mid} + 1..\text{high}]$ , so that  $\text{mid} < i \leq j \leq \text{high}$ , or
- crossing the midpoint, so that  $\text{low} \leq i \leq \text{mid} < j \leq \text{high}$ .



**Figure 4.4** (a) Possible locations of subarrays of  $A[\text{low}..\text{high}]$ : entirely in  $A[\text{low}..\text{mid}]$ , entirely in  $A[\text{mid} + 1..\text{high}]$ , or crossing the midpoint  $\text{mid}$ . (b) Any subarray of  $A[\text{low}..\text{high}]$  crossing the midpoint comprises two subarrays  $A[i..\text{mid}]$  and  $A[\text{mid} + 1..j]$ , where  $\text{low} \leq i \leq \text{mid}$  and  $\text{mid} < j \leq \text{high}$ .



# Using D&C

FIND-MAX-CROSSING-SUBARRAY(*A, low, mid, high*)

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

# Using D&C

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])           // base case: only one element
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =      // 3..11: handles recursive case
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =    // 6 .. 11: Combine part
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum  $\geq$  right-sum and left-sum  $\geq$  cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum  $\geq$  left-sum and right-sum  $\geq$  cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

// if no left and right subarrays has max sum, then the max subarray  
// must cross the midpoint

# Exercise

**Q. What does FIND-MAXIMIM-SUBARRAY return when all elements of A are negative?**

A. It will return the least negative position. As each of the cross sums are computed, the most positive one must have the shortest possible lengths. The algorithm does not consider length zero sub arrays, so it must have length 1.

# Exercise

Q. Write pseudocode for the brute-force method of solving the maximum-subarray problem. The procedure should run in  $\Theta(n^2)$  time.

**HW2** by next week 2/13 6 PM

Type in by MS Word, PPT or PDF format only

# Three approaches

**Substitution method:** when you have a good guess of the solution, prove that it's correct

**Recursion-tree method:** If you don't have a good guess, the recursion tree can help. Then solve with substitution method.

**Master method:** Provides solutions for recurrences of the form:

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

# Substitution method

Guess the form of the solution

Then prove it's correct by induction

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

Halves the input then constant amount of work

Guesses?

# Substitution method

Guess the form of the solution

Then prove it's correct by induction

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

Halves the input then constant amount of work

Similar to binary search:

Guess:  $O(\log_2 n)$

# Proof?

$$T(n) = T(n / 2) + d = O(\log_2 n)?$$

## Ideas?



# Proof?

$$T(n) = T(n / 2) + d = O(\log_2 n)?$$

Proof by induction!

-Assume it's true for smaller  $T(k)$ ,

i.e.  $k < n$

-prove that it's then true for current  $T(n)$

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

Assume  $T(k) = O(\log_2 k)$  for all  $k < n$

Show that  $T(n) = O(\log_2 n)$

From our assumption,  $T(n/2) = O(\log_2 n)$ :

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

From the definition of big- $O$ :  $T(n/2) \leq c \log_2(n/2)$

How do we now prove  $T(n) = O(\log n)$ ?

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

To prove that  $T(n) = O(\log_2 n)$  identify the appropriate constants:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

i.e. some constant  $c$  such that  $T(n) \leq c \log_2 n$

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

$$\leq c \log_2(n/2) + d \quad \text{from our inductive hypothesis}$$

$$\leq c \log_2 n - c \log_2 2 + d$$

$$\leq c \log_2 n - c + d \quad \text{residual}$$

$$\leq c \log_2 n$$

$$\text{if } c \geq d$$



# Base case?

For an inductive proof we need to show two things:

- Assuming it's true for  $k < n$  show it's true for  $n$
- *Show that it holds for some base case*

What is the base case in our situation?

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

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

## Guess the solution?

- At each iteration, does a linear amount of work (i.e. iterate over the data) and reduces the size by one at each step
- $O(n^2)$

Assume  $T(k) = O(k^2)$  for all  $k < n$

- again, this implies that  $T(n-1) \leq c(n-1)^2$

Show that  $T(n) = O(n^2)$ , i.e.  $T(n) \leq cn^2$

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

$$\leq c(n-1)^2 + n \quad \text{from our inductive hypothesis}$$

$$= c(n^2 - 2n + 1) + n$$

$$= cn^2 - 2cn + c + n \quad \text{residual}$$

$$\leq cn^2$$

$$\text{if } -2cn + c + n \leq 0$$

$$-2cn + c \leq -n$$

$$c(-2n + 1) \leq -n$$

$$c \geq \frac{n}{2n-1}$$

$$c \geq \frac{1}{2-1/n}$$

which holds for any  $c$   
 $\geq 1$  for  $n \geq 1$

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

## Guess the solution?

- Recurses into 2 sub-problems that are half the size and performs some operation on all the elements
- $O(n \log n)$

What if we guess wrong, e.g.  $O(n^2)$ ?

Assume  $T(k) = O(k^2)$  for all  $k < n$

- again, this implies that  $T(n/2) \leq c(n/2)^2$

Show that  $T(n) = O(n^2)$

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

$$\leq 2c(n/2)^2 + n \quad \text{from our inductive hypothesis}$$

$$= 2cn^2 / 4 + n$$

$$= 1/2cn^2 + n$$

$$= cn^2 - (1/2cn^2 - n) \quad \text{residual}$$

$$\leq cn^2$$

if

$$-(1/2cn^2 - n) \leq 0$$

$$-1/2cn^2 + n \leq 0$$

$$cn \geq 2$$

overkill?





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

What if we guess wrong, e.g.  $O(n)$ ?

Assume  $T(k) = O(k)$  for all  $k < n$

– again, this implies that  $T(n/2) \leq c(n/2)$


Show that  $T(n) = O(n)$

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

$$\leq 2cn/2 + n$$

$$= cn + n$$

$$\leq cn$$



factor of  $n$  so we can  
just roll it in?

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

What if we guess wrong, e.g.  $O(n)$ ?

Assume  $T(k) = O(k)$  for all  $k < n$

– again, this implies that  $T(n/2) \leq c(n/2)$

Show that  $T(n) = O(n)$

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

$$\leq 2cn/2 + n$$

$$= cn + n$$

$$\leq cn$$

Must prove the  
exact form!

$cn + n \leq cn$  ??

factor of  $n$  so we can  
just roll it in?

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

Prove  $T(n) = O(n \log_2 n)$

Assume  $T(k) = O(k \log_2 k)$  for all  $k < n$

– again, this implies that  $T(k) = ck \log_2 k$

Show that  $T(n) = O(n \log_2 n)$

$$\begin{aligned}
 T(n) &= 2T(n/2) + n \\
 &\leq 2cn/2 \log(n/2) + n \\
 &\leq cn(\log_2 n - \log_2 2) + n \\
 &\leq cn \log_2 n - cn + n \quad \text{residual} \\
 &\leq cn \log_2 n
 \end{aligned}$$

if  $cn \geq n, c > 1$

# Recursion Tree

Guessing the answer can be difficult

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

$$T(n) = T(n/3) + 2T(2n/3) + cn$$

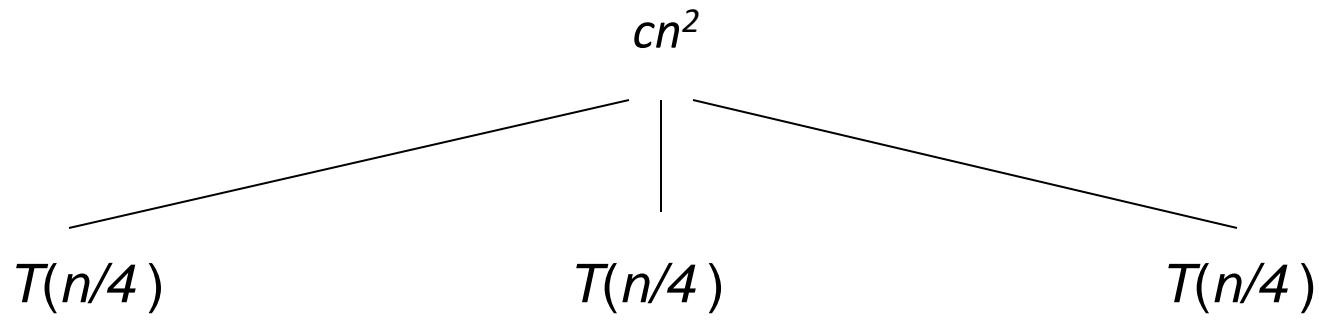
The recursion tree approach

- Draw out the cost of the tree at each level of recursion
- Sum up the cost of the levels of the tree
  - Find the cost of each level with respect to the depth
  - Figure out the depth of the tree
  - Figure out (or bound) the number of leaves
- Verify your answer using the substitution method

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

cost

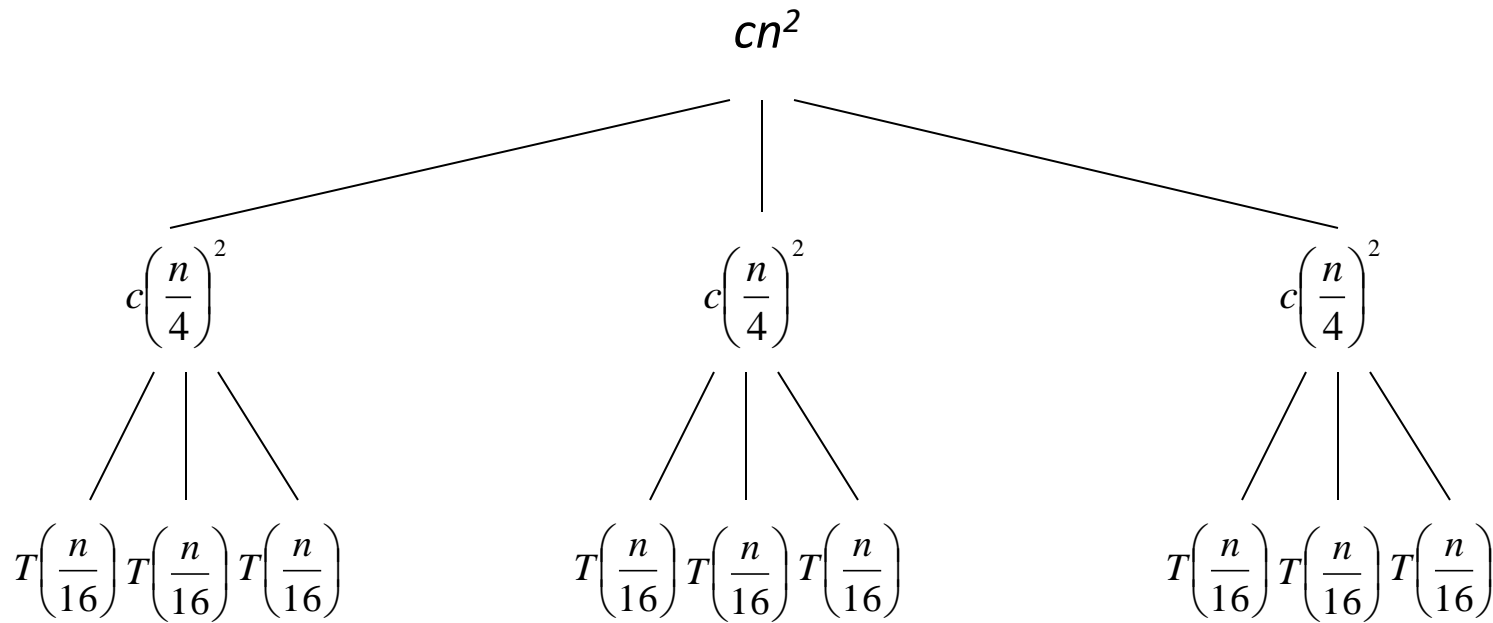
$cn^2$



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

cost

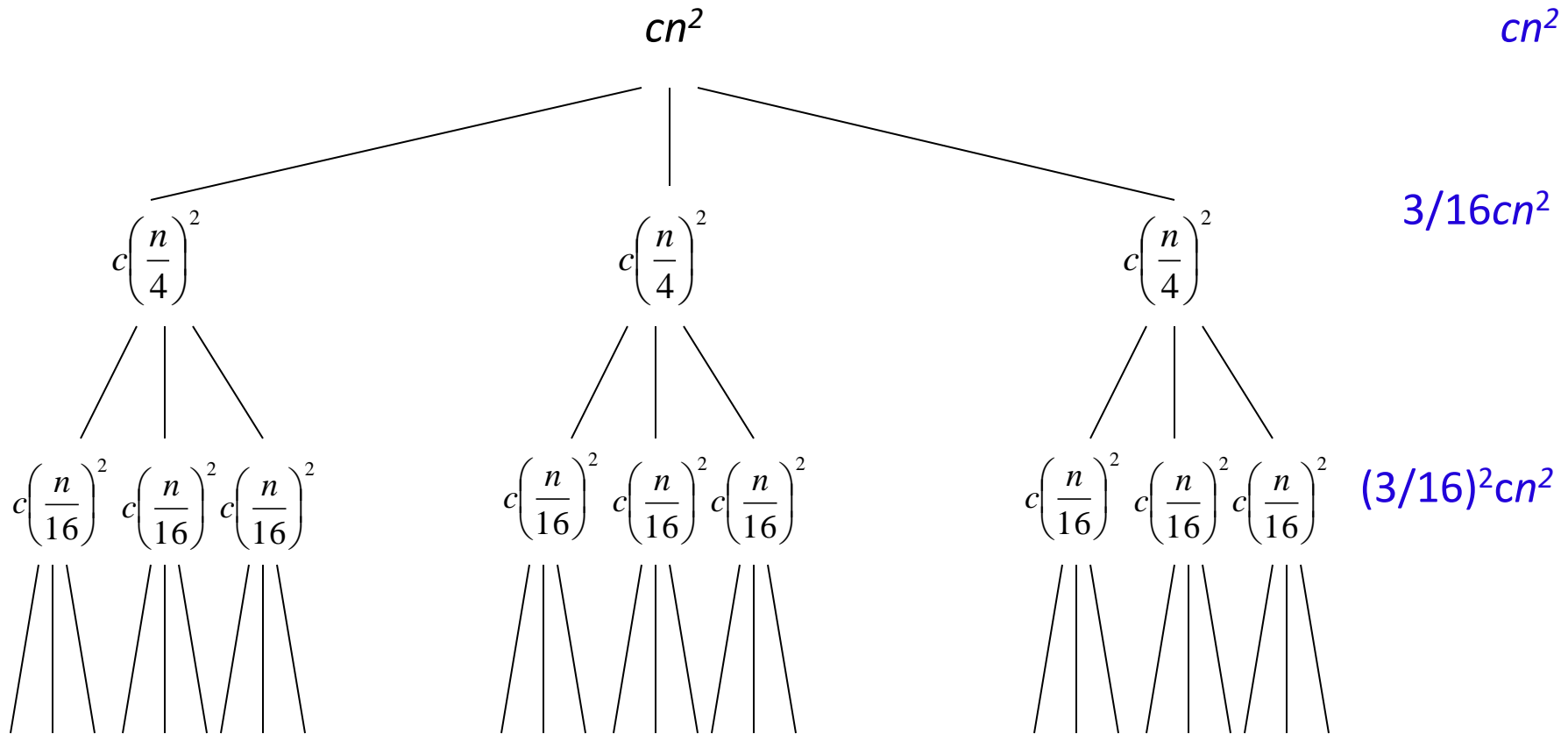
$cn^2$



$3/16cn^2$

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

cost



What is the cost at each level?

$$\left(\frac{3}{16}\right)^d cn^2$$

# What is the depth of the tree?

At each level, the size of the data is divided by 4

$$\frac{n}{4^d} = 1$$

$$\log\left(\frac{n}{4^d}\right) = 0$$

$$\log n - \log 4^d = 0$$

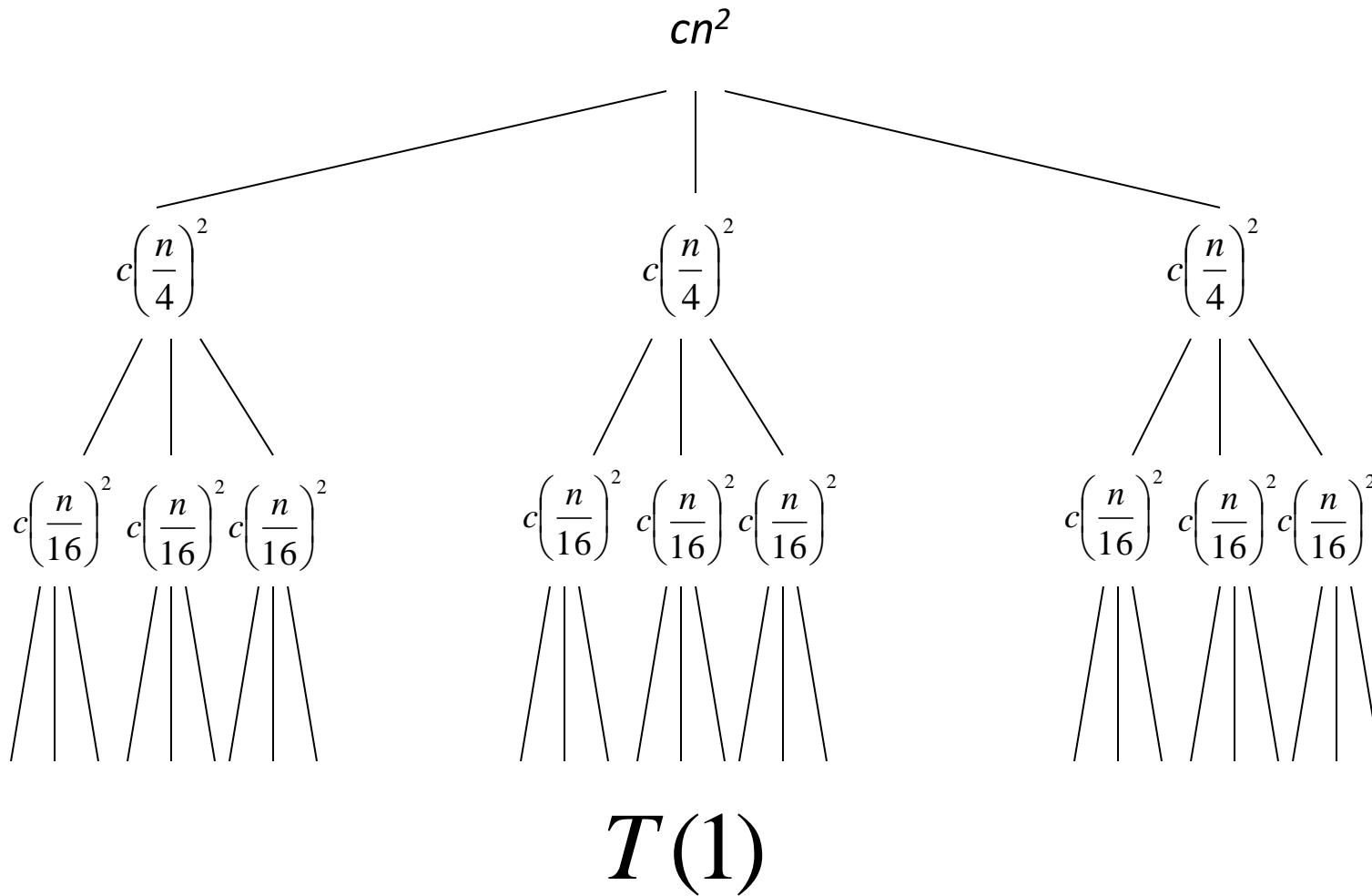
$$d \log 4 = \log n$$

$$d = \log_4 n$$





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



How many leaves are there?

# How many leaves?

How many leaves are there in a complete ternary tree of depth  $d$ ?

$$3^d = 3^{\log_4 n}$$

# Total cost

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{d-1} cn^2 + \Theta(3^{\log_4 n})$$

$$= cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + \Theta(3^{\log_4 n})$$

$$< cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(3^{\log_4 n})$$

$$= \frac{1}{1 - (3/16)} cn^2 + \Theta(3^{\log_4 n})$$

$$= \frac{16}{13} cn^2 + \Theta(3^{\log_4 n}) \quad ?$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

let  $x = 3/16$

# Total cost

$$T(n) = \frac{16}{13}cn^2 + \Theta(3^{\log_4 n})$$

$$\begin{aligned} 3^{\log_4 n} &= 4^{\log_4 3^{\log_4 n}} \\ &= 4^{\log_4 n \log_4 3} \\ &= 4^{\log_4 n^{\log_4 3}} \\ &= n^{\log_4 3} \end{aligned}$$

$$T(n) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = O(n^2)$$



# Verify solution using substitution

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

Assume  $T(k) = O(k^2)$  for all  $k < n$

Show that  $T(n) = O(n^2)$

Given that  $T(n/4) = O((n/4)^2)$ , then

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

$$T(n/4) \leq c(n/4)^2$$

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

To prove that Show that  $T(n) = O(n^2)$  we need to identify the appropriate constants:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

i.e. some constant  $c$  such that  $T(n) \leq cn^2$

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

$$\leq 3c(n/4)^2 + n^2$$

$$= cn^2 3/16 + n^2$$

$$\leq cn^2$$

if

$$c \geq \frac{16}{13}$$



# Master Method

Provides solutions to the recurrences of the form:

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

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$T(n) = 16T(n/4) + n$$

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a = 16 & n^{\log_b a} = n^{\log_4 16} \\ b = 4 & = n^2 \\ f(n) = n & \end{array}$$

is  $n = O(n^{2-\varepsilon})$ ?

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

is  $n = \Omega(n^{2+\varepsilon})$ ?

**Case 1:  $\Theta(n^2)$**



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

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a = 1 & n^{\log_b a} = n^{\log_2 1} \\ b = 2 & = n^0 \\ f(n) = 2^n & \end{array}$$

### Case 3?

is  $2^n = O(n^{0-\varepsilon})$ ?

is  $2^n = \Theta(n^0)$ ?

is  $2^n = \Omega(n^{0+\varepsilon})$ ?

is  $2^{n/2} \leq c2^n$  for  $c < 1$ ?

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

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

is  $2^{n/2} \leq c2^n$  for  $c < 1$ ?

Let  $c = 1/2$

$$2^{n/2} \leq (1/2)2^n$$

$$2^{n/2} \leq 2^{-1}2^n$$

$$2^{n/2} \leq 2^{n-1}$$

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

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

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a &= 2 \\ b &= 2 \\ f(n) &= n \end{array} \qquad \begin{array}{ll} n^{\log_b a} &= n^{\log_2 2} \\ &= n^1 \end{array}$$

is  $n = O(n^{1-\varepsilon})$ ?

is  $n = \Theta(n^1)$ ?

is  $n = \Omega(n^{1+\varepsilon})$ ?

**Case 2:**  $\Theta(n \log n)$

$$T(n) = 16T(n/4) + n!$$

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a = 16 & n^{\log_b a} = n^{\log_4 16} \\ b = 4 & = n^2 \\ f(n) = n! & \end{array}$$

### Case 3?

is  $n! = O(n^{2-\varepsilon})$ ?

is  $n! = \Theta(n^2)$ ?

is  $n! = \Omega(n^{2+\varepsilon})$ ?

is  $16(n/4)! \leq cn!$  for  $c < 1$ ?

$$T(n) = 16T(n/4) + n!$$

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

is  $16(n/4)! \leq cn!$  for  $c < 1$ ?

Let  $c = 1/2$

$$cn! = 1/2n!$$

$$> (n/2)!$$

$$T(n) = \Theta(n!)$$

therefore,

$$16(n/4)! \leq (n/2)! < 1/2n!$$

$$T(n) = \sqrt{2}T(n/2) + \log n$$

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a = \sqrt{2} & n^{\log_b a} = n^{\log_2 \sqrt{2}} \\ b = 2 & = n^{\log_2 2^{1/2}} \\ f(n) = \log n & = \sqrt{n} \end{array}$$

is  $\log n = O(n^{1/2 - \varepsilon})$ ?

is  $\log n = \Theta(n^{1/2})$ ?

is  $\log n = \Omega(n^{1/2 + \varepsilon})$ ?

**Case 1:  $\Theta(\sqrt{n})$**

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

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$   
then  $T(n) = \Theta(f(n))$

$$\begin{array}{ll} a &= 4 \\ b &= 2 \\ f(n) &= n \end{array} \qquad \begin{array}{ll} n^{\log_b a} &= n^{\log_2 4} \\ &= n^2 \end{array}$$

is  $n = O(n^{2-\varepsilon})$ ?

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

is  $n = \Omega(n^{2+\varepsilon})$ ?

**Case 1:  $\Theta(n^2)$**

# Recurrences

$$T(n) = 2T(n/3) + d$$

$$T(n) = 7T(n/7) + n$$

if  $f(n) = O(n^{\log_b a - \varepsilon})$  for  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

if  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$

if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for  $\varepsilon > 0$  and  $af(n/b) \leq cf(n)$  for  $c < 1$

then  $T(n) = \Theta(f(n))$

$$T(n) = T(n-1) + \log n$$

$$T(n) = 8T(n/2) + n^3$$



# Appendix

# Multiplication Algorithm

- A **multiplication algorithm** is to multiply two numbers
- Depending on the size of the numbers, different algorithms are used

# Multiplication Algorithm

## Grid method

- Both factors are broken up into their hundreds, tens and units parts, and the products of the parts are then calculated explicitly in a relatively simple multiplication-only stage

$$34 \times 13 = ?$$

×	30	4
10	300	40
3	90	12

- $300 + 90 + 40 + 12 = 442$

# Multiplication Algorithm

## Long Multiplication

- a natural way of multiplying numbers is taught in schools as **long multiplication**

$$\begin{array}{r} 23958233 \\ \times \quad 5830 \\ \hline 00000000 \quad ( = 23,958,233 \times 0 ) \\ 71874699 \quad ( = 23,958,233 \times 30 ) \\ 191665864 \quad ( = 23,958,233 \times 800 ) \\ + 119791165 \quad ( = 23,958,233 \times 5,000 ) \\ \hline 139676498390 \quad ( = 139,676,498,390 ) \end{array}$$

# Multiplication Algorithm

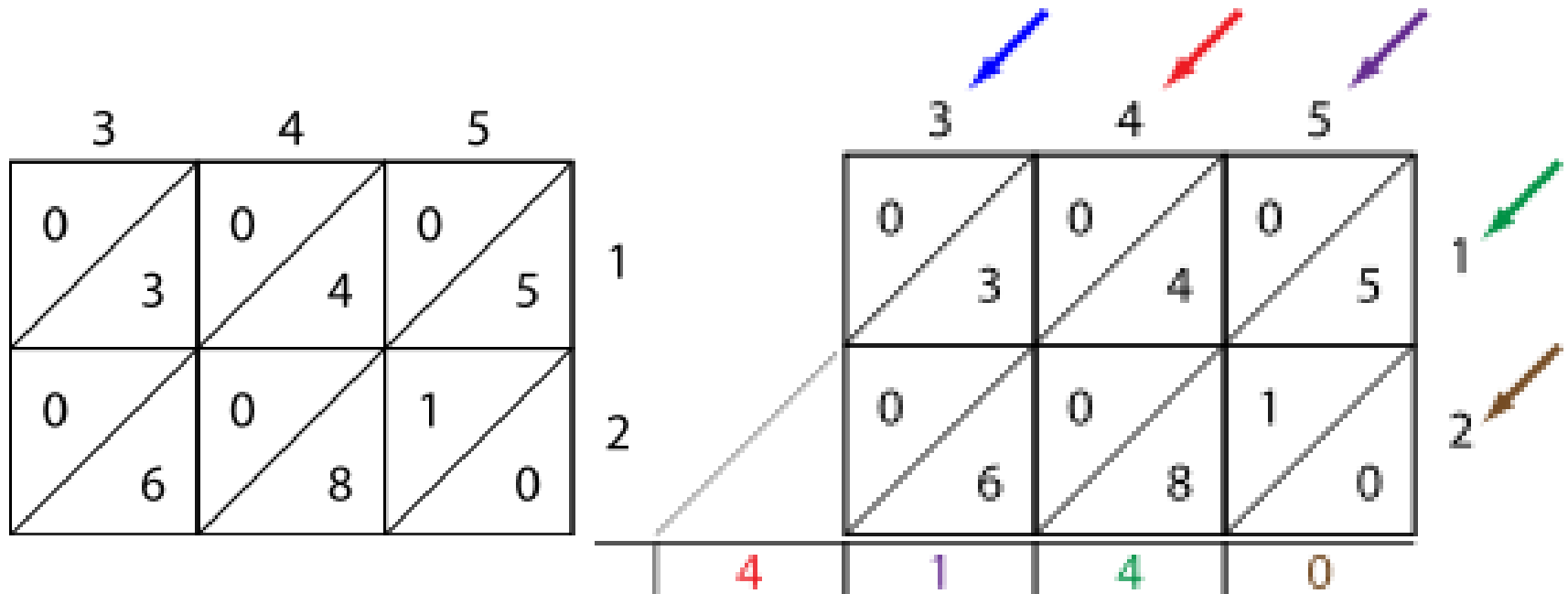
## Long Multiplication

```
multiply(a[1..p], b[1..q], base)
  product = [1..p+q]
  for b_i = 1 to q
    carry = 0
    for a_i = 1 to p
      product[a_i + b_i - 1] += carry + a[a_i] * b[b_i]
      carry = product[a_i + b_i - 1] / base
      product[a_i + b_i - 1] = product[a_i + b_i - 1] mod base
    product[b_i + p] = carry
  return product
```

# Multiplication Algorithm

## Lattice Multiplication

$$345 \times 12 = ?$$



# Multiplication Algorithm

## Lattice Multiplication

$$23958233 \times 5830 = ?$$

		2	3	9	5	8	2	3	3	
		+---	+---	+---	+---	+---	+---	+---	+---	+
		1 /	1 /	4 /	2 /	4 /	1 /	1 /	1 /	
		/	/	/	/	/	/	/	/	5
01	/	0	/ 5	/ 5	/ 5	/ 0	/ 0	/ 5	/ 5	
		+---	+---	+---	+---	+---	+---	+---	+---	+
		1 /	2 /	7 /	4 /	6 /	1 /	2 /	2 /	
		/	/	/	/	/	/	/	/	8
02	/	6	/ 4	/ 2	/ 0	/ 4	/ 6	/ 4	/ 4	
		+---	+---	+---	+---	+---	+---	+---	+---	+
		0 /	0 /	2 /	1 /	2 /	0 /	0 /	0 /	
		/	/	/	/	/	/	/	/	3
17	/	6	/ 9	/ 7	/ 5	/ 4	/ 6	/ 9	/ 9	
		+---	+---	+---	+---	+---	+---	+---	+---	+
		0 /	0 /	0 /	0 /	0 /	0 /	0 /	0 /	
		/	/	/	/	/	/	/	/	0
24	/	0	/ 0	/ 0	/ 0	/ 0	/ 0	/ 0	/ 0	
		+---	+---	+---	+---	+---	+---	+---	+---	+
		26	15	13	18	17	13	09	00	

```

01
002
0017
00024
000026
0000015
00000013
000000018
0000000017
00000000013
000000000009
00000000000000
-----
139676498390

```

= 139,676,498,390

# Multiplication Algorithm

## Karatsuba algorithm

- reduces the multiplication of two  $n$ -digit numbers to at most  $n^{\log_2 3} \approx n^{1.58}$  single-digit multiplications in general
- faster than the classical algorithm, which requires  $n^2$  single-digit products
- For example, the Karatsuba algorithm requires  $3^{10} = 59,049$  single-digit multiplications to multiply two 1024-digit numbers ( $n = 1024 = 2^{10}$ ), whereas the classical algorithm requires  $(2^{10})^2 = 1,048,576$  (a speedup of 17.75 times)



# Multiplication Algorithm

## Karatsuba algorithm

Let  $x$  and  $y$  be represented as  $n$ -digit strings in some base  $B$ .

$$\begin{aligned} x &= x_1 B^m + x_0, \\ y &= y_1 B^m + y_0, \end{aligned} \quad \begin{array}{l} \text{For any positive integer } m \text{ less than } n, \\ \text{one can write the two given numbers as} \end{array}$$

where  $x_0$  and  $y_0$  are less than  $B^m$ . The product is then

$$\begin{aligned} xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ &= z_2 B^{2m} + z_1 B^m + z_0, \end{aligned}$$

where

$$z_2 = x_1 y_1,$$

$$z_1 = x_1 y_0 + x_0 y_1,$$

$$z_0 = x_0 y_0.$$

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0.$$

# Multiplication Algorithm

## Karatsuba algorithm

- To compute the product of 12345 and 6789, where  $B = 10$ , choose  $m = 3$ . We use  $m$  right shifts for decomposing the input operands using the resulting base ( $B^m = 1000$ ), as:

$$12345 = \mathbf{12} \cdot 1000 + \mathbf{345}$$

$$6789 = \mathbf{6} \cdot 1000 + \mathbf{789}$$

- Only three multiplications, which operate on smaller integers, are used to compute three partial results:

$$z_2 = \mathbf{12} \times \mathbf{6} = 72$$

$$z_0 = \mathbf{345} \times \mathbf{789} = 272205$$

$$z_1 = (\mathbf{12} + \mathbf{345}) \times (\mathbf{6} + \mathbf{789}) - z_2 - z_0 = 357 \times 795 - 72 - 272205 = 283815 - 72 - 272205 = 11538$$

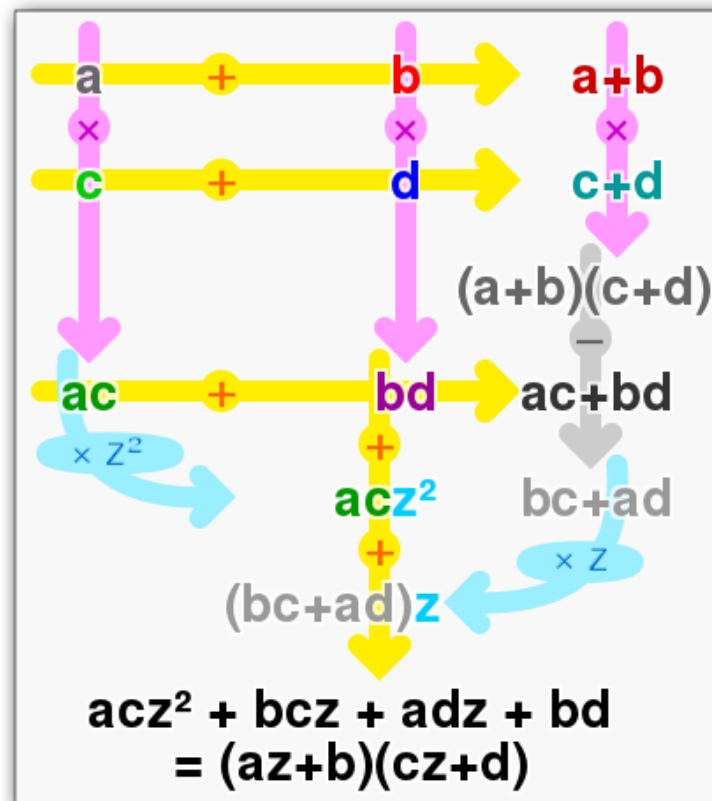
- We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

$$\text{result} = z_2 \cdot (B^m)^2 + z_1 \cdot (B^m)^1 + z_0 \cdot (B^m)^0, \text{ i.e.}$$

$$\text{result} = 72 \cdot 1000^2 + 11538 \cdot 1000 + 272205 = \mathbf{83810205}.$$

M

# Karatsuba algorithm



$$1234 \times 567 = ?$$

