Chap.4 Divide and Conquer

- Maximum subarray
- The substitution method
- The recursion-tree method
- The master method

Designing Algorithms

- There are a number of design paradigms for algorithms that have proven useful for many types of problems
- Insertion sort – incremental approach
- Other examples of design approaches
  - divide and conquer
  - greedy algorithms
  - dynamic programming…

Divide and Conquer

- A good divide and conquer algorithm generally implies an easy recursive version of the algorithm
- Three steps
  - <u>**Divide**</u> the problem into a number of subproblems
  - <u>**Conquer**</u> the subproblems by solving them recursively. When the subproblem size is small enough, just solve the subproblem.
  - <u>**Combine**</u> subproblems to form the solution of the original problem

Recurrence

- Definition
  a recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
- Ex)

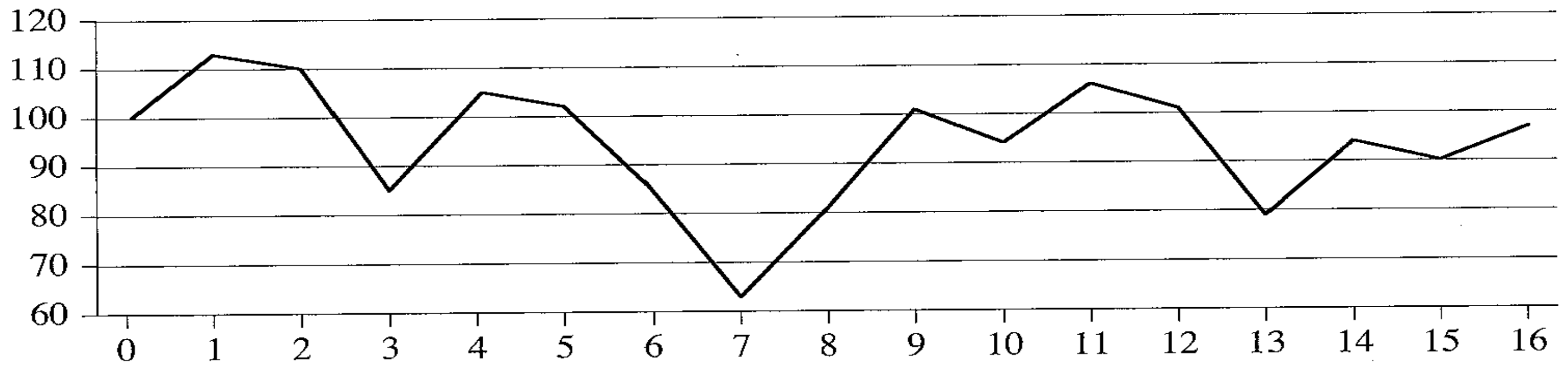
$$T(n) = \begin{cases} \Theta(1) \\ \underbrace{aT(n/b)}_{\text{Conquer cost}} + \underbrace{D(n)}_{\text{Divide cost}} + \underbrace{C(n)}_{\text{Combine cost}} \end{cases}$$

Why Recurrences?

- The complexity of many interesting algorithms is easily expressed as a recurrence – especially divide and conquer algorithms
- The complexity of recursive algorithms is readily expressed as a recurrence.

Maximum Subarray Problem

- Stock investment: Buy one unit of stock only one time and then sell it at a later date
- Goal: to maximize the profit

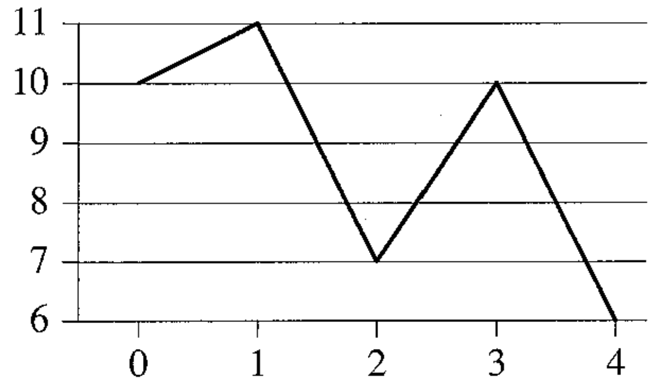


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

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.

One potential solution?

- Find the highest price and search left to find the lowest prior price
- Find the lowest price and search right to find the highest later price
- Take the pair with the greater difference
- Do not work! See counterexample below.

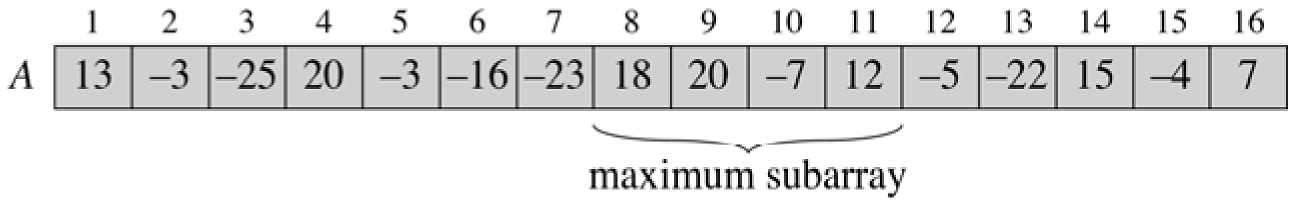


Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

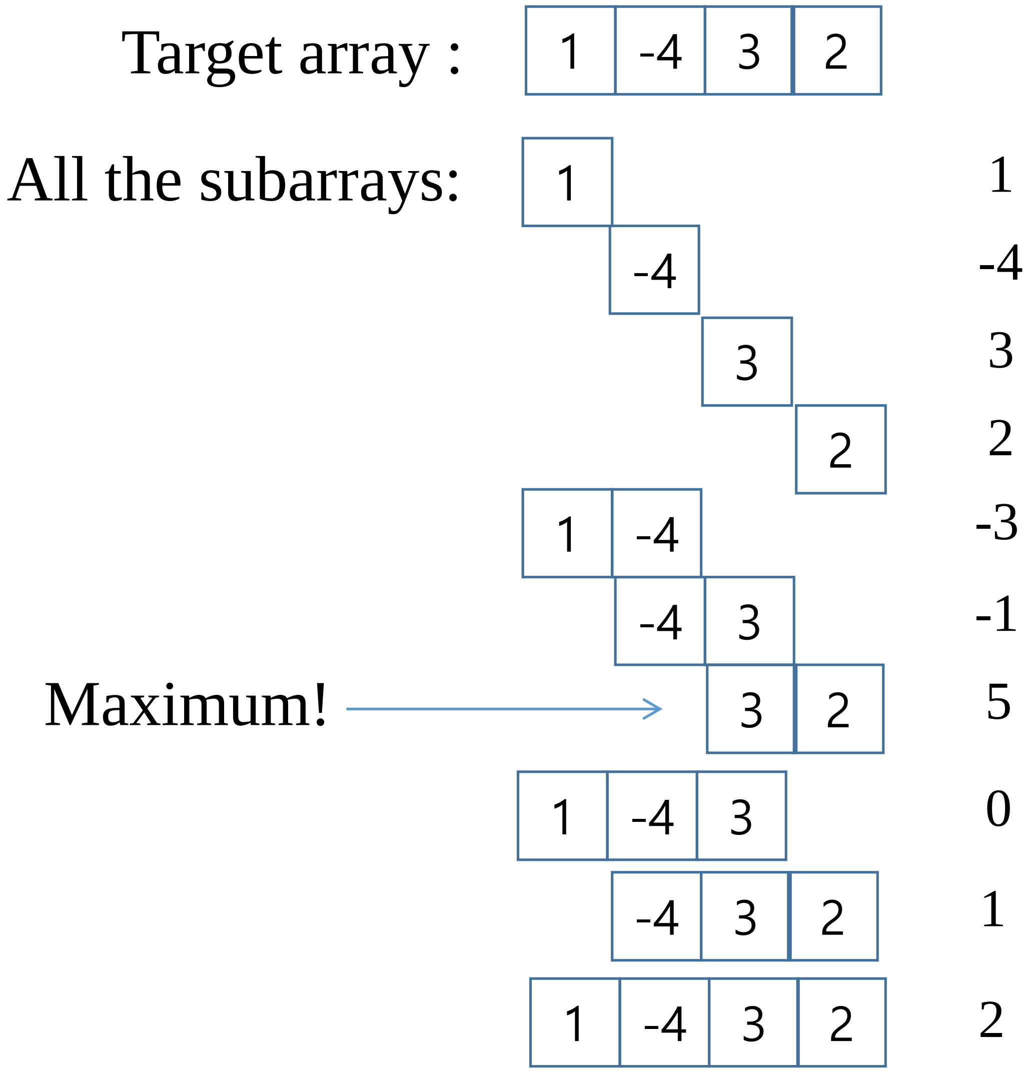
Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling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

- Consider the daily change in price
- Maximum subarray problem: find the nonempty, contiguous subarray of A whose values have the largest sum.



What is a Maximum Subarray ?



The subarray with the largest sum

What is the brute-force T(n)?

$O(n^2)$

Maximum Subarray Problem

- We need to find the better algorithm than brute force algorithm.
- How about Divide & Conquer algorithm?

Target array :

All the subarrays:

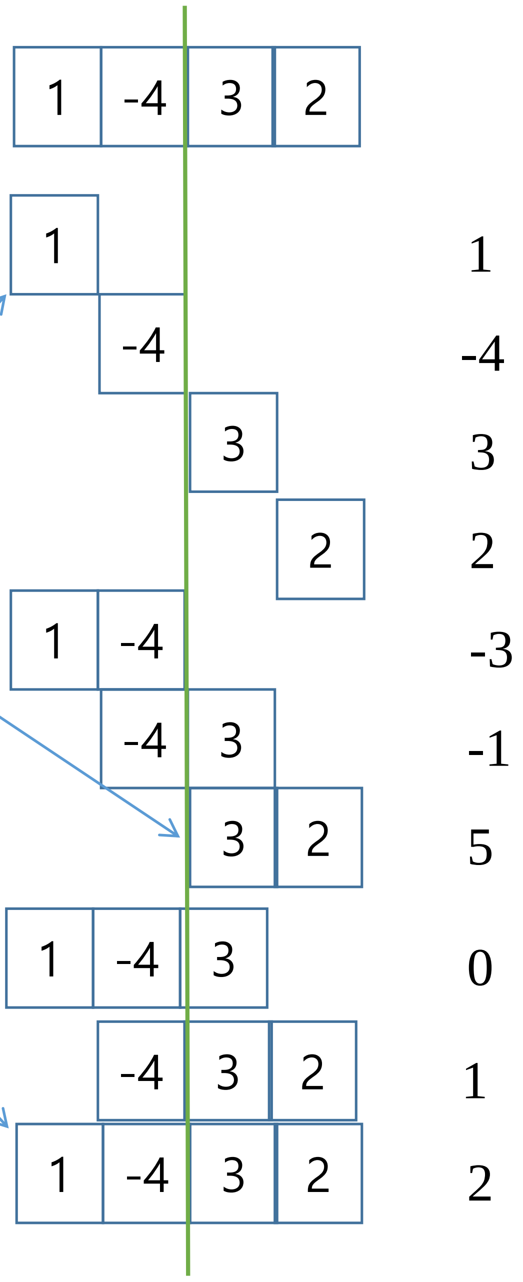
The problem can be then solved by:

1. Find the max in left subarrays

2. Find the max in right subarrays

3. Find the max in crossing subarrays

4. Choose the largest one from those 3 as the final result



Divide target array into 2 arrays.

We then have 3 types of subarrays:

1.The ones belong to the left array

2.The ones belong to the right array

3.The ones crossing the mid point

The whole algorithm

FindMaxSub ($\boxed{1}\boxed{-4}\boxed{3}\boxed{2}$)

1. Find the max in left subarrays **FindMaxSub** ($\boxed{1}\boxed{-4}$)

2. Find the max in right subarrays **FindMaxSub** ($\boxed{3}\boxed{2}$)

3. Find the max in crossing subarrays

   Scan $\boxed{1}\boxed{-4}$ once, and scan $\boxed{3}\boxed{2}$ once

4. Choose the largest one from those 3 as the final result

Divide and Conquer

- Suppose we want to find a maximum subarray of A[*low..high]*
- Divide and conquer will find the midpoint, say mid, of the subarray, and consider the subarrays A[*low..mid] and A[mid+1..high]*
- Any contiguous subarray A[*i..j] must lie in* one area out of three possibilities

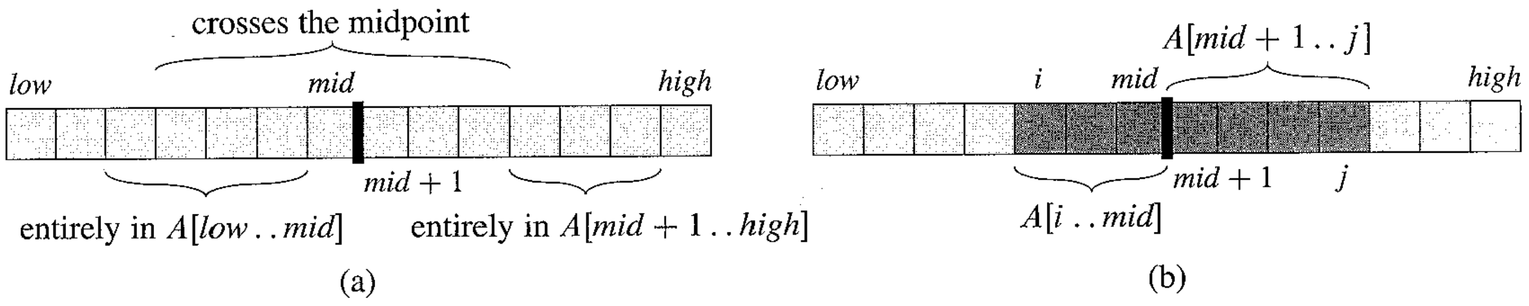


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where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

Find Max Crossing Subarray

- First, it is easy to find a maximum subarray crossing the midpoint
- We just need to find maximum subarrays of the form *A[i..mid] and A[mid+1..j]* and combine them

Find Max Crossing Subarray

```
FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
left-sum = $-\infty$
sum = 0
for $i$ = mid downto low
    sum = sum + $A[i]$
    if sum > left-sum
        left-sum = sum
        max-left = $i$
right-sum = $-\infty$
sum = 0
for $j$ = mid + 1 to high
    sum = sum + $A[j]$
    if sum > right-sum
        right-sum = sum
        max-right = $j$
return (max-left, max-right, left-sum + right-sum)
```

Find Maximum subarray

- We can then write a divide and conquer algorithm to solve the maximum subarray problem.
- Divide into three cases, and choose the best solution
  - Left subarray
  - Crossing subarray
  - Right subarray

Find Maximum Subarray

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

Time complexity? $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(\text{n})$ $O(nlogn)$

Methods for Solving Recurrences

- Substitution method
- Recursion-tree
- Master method

Substitution Method

- (1) Guess the form of the solution
- (2) Use mathematical induction to find the constants and show the solution works
    - Works well when it is easy to guess
    - Can be used for upper or lower bounds
- *Example*: merge sort
    - T(n) = 2T(n/2) + cn
    - We guess that the answer is O(nlogn)
    - Prove it by induction

Substitution Method

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

- Guess T(n) = O(*n* log *n*)
- Prove T(n) ≤ *cn* log *n*   for some c

Inductive base: prove the inequality holds for some small n

T(2)=2T(1) + 2 = 4

cn lg *n* = c * 2 * log 2 = 2c   choose any c ≥ 2

Assume true for n/2

$$T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor)$$

Substitution Method

Prove that it then must hold <u>for n</u>:

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\le 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor) + n \\
&\le cn \lg \lfloor n/2 \rfloor + n \\
&\le cn(\lg n - \lg 2) + n \\
&\le cn \lg n - cn + n \\
&\le cn \lg n
\end{aligned}
$$

Substitution Method

- Experience helps… you know it when you see it…
  - If a recurrence looks familiar… guess a similar solution

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

Guess: T(n) = O(*n* lg *n*)

Why?

- The 17 cannot substantially affect the solution to the recurrence (just a constant)

Iterative Substitution

- Look at the recurrence relation:

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

- Substituting n – 1 for n in the relation above we get:
  $T(n - 1) = T(n – 2) + (n – 1)$
- Substitute for n – 1 in the original relation:
  $T(n) = (T(n – 2) + (n – 1)) + n$
- We know that $T(n – 2) = T(n – 3) + (n – 2)$
- So substitute this for $T(n – 2)$ above:
  $T(n) = (T(n – 3) + (n – 2)) + (n – 1) + n$

Iterative Substitution

- We see the following pattern:
  $T(n) = T(n - 1) + n$
  $T(n) = (T(n - 2) + (n - 1)) + n$
  $T(n) = (T(n - 3) + (n - 2)) + (n - 1) + n$
  . . .
  $T(n) = T(n - (n - 2)) + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$
  $T(n) = T(n - (n - 1)) + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$
  $T(n) = T(n - (n - 0)) + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$
- We can rewrite $(n - (n - 0))$ as $(n - n)$ or as (0), thus:
  $T(n) = T(0) + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$

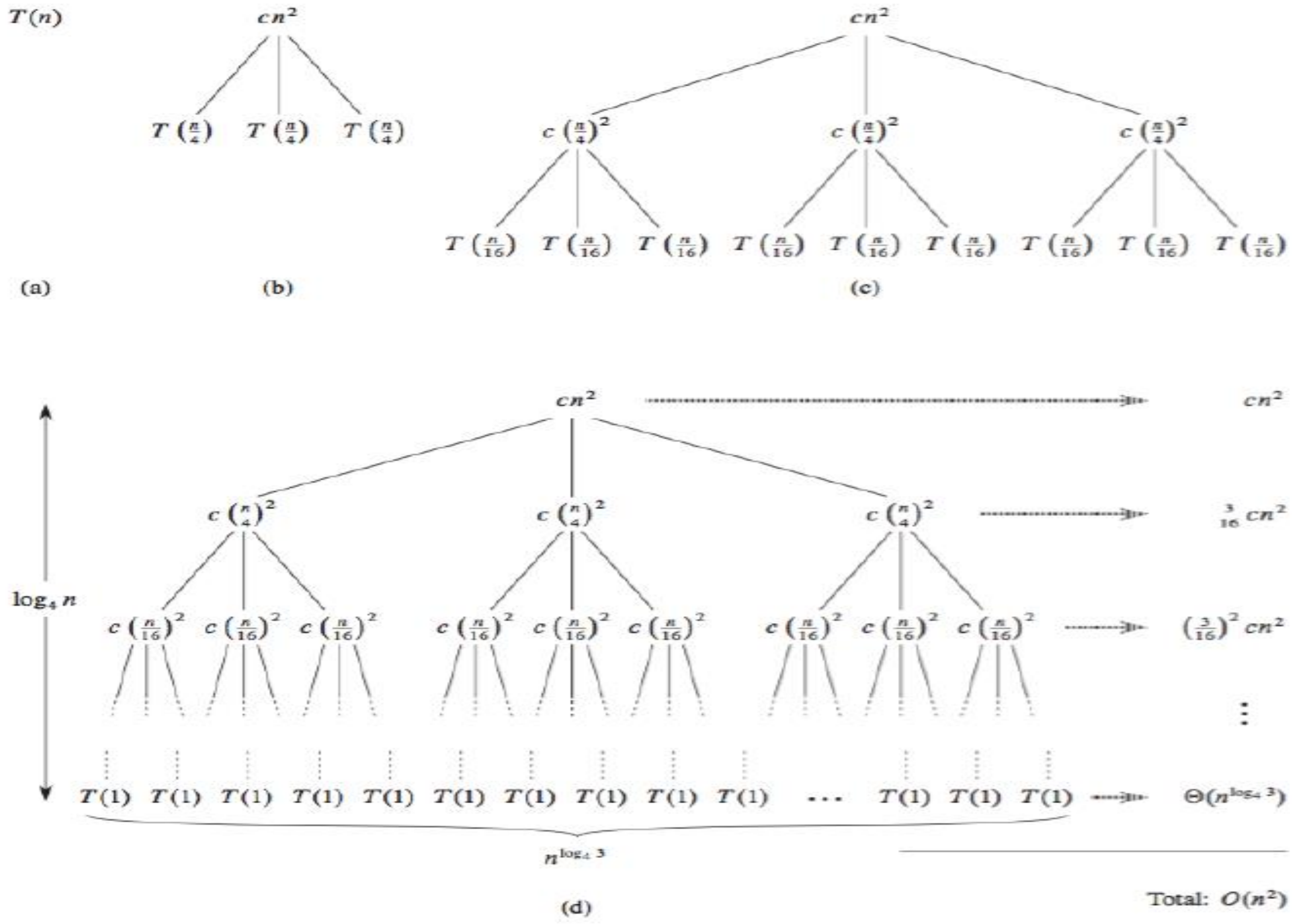
Iterative Substitution

- But we know that $T(0) = 0$ is the base case, so:
  $T(n) = 0 + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$
- The summation of
  $T(n) = 0 + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n$
  is $T(n) = (n (n + 1) /2) = \frac{1}{2} n^2 + \frac{1}{2} n$
  which we recognize as $O(n^2)$.

Recursion Tree

- Recursion tree
  - Each node represents the cost of a single sub-problem in the set of recursive function invocations
  - Sum the costs within each level of the tree to obtain a set of per-level costs
  - Sum all the per-level costs to determine the total cost of all levels of the recursion
- Useful for generating a good guess for the substitution method

Recursion tree for T(n) = 3T(n/4) + cn^2



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
Total: $O(n^2)$
(d)

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

- The sub-problem size for a node at depth i is $n/4^i$
  - When the sub-problem size is 1 → $n/4^i = 1$ → $i=\log_4 n$
  - The tree has $\log_4 n+1$ levels (0, 1, 2,.., $\log_4 n$)
- The cost at each level of the tree (0, 1, 2,.., $\log_4 n-1$)
  - Number of nodes at depth i is 3^i
  - Each node at depth i has a cost of $c(n/4^i)^2$
  - The total cost over all nodes at depth i is $3^i c(n/4^i)^2=(3/16)^i cn^2$
- The cost at depth $\log_4 n$
  - Number of nodes is $3^{\log_4 n} = n^{\log_4 3}$
  - Each contributing cost T(1)
  - The total cost

$$n^{\log_4 3}T(1) = \Theta(n^{\log_4 3})$$

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

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

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

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

$$= \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

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

Prove $T(n) = O(n^2)$ is an upper bound

Use the substitution method
i.e. $T(n) \leq dn^2$ for some constant $d > 0$

$$
\begin{aligned}
T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\leq 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16}dn^2 + cn^2 \\
&\leq dn^2
\end{aligned}
$$

(for $d \geq (16/13)c$)

Master Theorem

- Provides a cookbook method for solving recurrences of the form
  $T(n)=aT(n/b) + f(n)$
  where a ≥ 1 and b > 1 and $f(n)$ is an asymptotically positive function
- The form of the master theorem is very convenient because divide and conquer algorithms have recurrences of the form
  $T(n) = aT(n/b)+D(n)+C(n)$

Master Theorem

- Provides solutions to the recurrences of the form:

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

case1) if $f(n) = O(n^{\log_b a-\varepsilon})$ for $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

case2) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$

case3) if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for $\varepsilon > 0$ and $af(n/b) \le cf(n)$ for $c < 1$ then $T(n) = \Theta(f(n))$

What does the Master Theorem say?

- Compare two functions:
  f(n) and $n^{\log_b a}$
- When *f(n)* grows asymptotically slower (Case 1)

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

- When the growth rates are the same (Case 2)

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

- When *f(n)* grows asymptotically faster (Case 3)

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

Some Examples of master theorem

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

- Compare two function: *f(n)* and $n^{\log_b a}$
- *f(n)= n*
- *a=16 b= 4* $n^{\log_b a} = n^{\log_4 16} = n^2$
- *f(n)=n* grows asymptotically slower than n^2 case 1 $\Theta(n^2)$

Some Examples of master theorem

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

- Compare two function: *f(n)* and $n^{\log_b a}$
- *f(n)= n*
- *a=2 b=2*

$$\begin{aligned} n^{\log_b a} &= n^{\log_2 2} \\ &= n^1 \end{aligned}$$

- *f(n)=n* grows asymptotically same with *n¹*
  case 2 Θ(*nlogn*)

Some Examples of master theorem

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

- Compare two function: *f(n)* and $n^{\log_b a}$
- *f(n)= 2ⁿ*
- *a=1 b=2*

$$n^{\log_b a} = n^{\log_2 1}$$
$$= n^0$$

- *f(n)= 2ⁿ* grows asymptotically faster with *n ⁰* case 3 Θ(*2ⁿ*)

Conclusion

- We talked about:
  - The substitution method
  - The recursion-tree method
  - The master method
- Be able to solve recurrences using all three of these methods