UNIT 1 Part II
DIVIDE AND CONQUER

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Preliminaries: Mathematical Induction

Mathematical Induction

- The first domino falls.
- If a domino falls,
  so will the next domino.
- All dominoes will fall!

Weak Induction

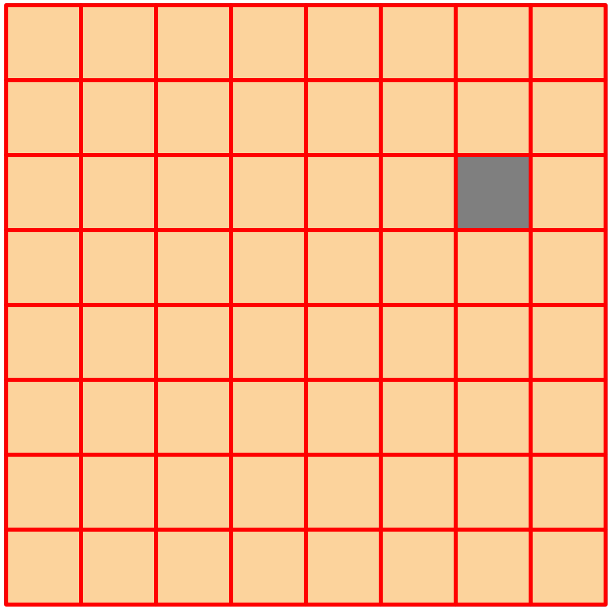
- Given the proposition $P(n)$ where $n \in \mathbb{N}$, a proof by mathematical induction is of the form:
  - Basis step: The proposition $P(0)$ is shown to be true
  - Inductive step: The implication $P(k) \rightarrow P(k+1)$ is shown to be true for every $k \in \mathbb{N}$
    - In the inductive step, statement *P*(*k*) is called the inductive hypothesis

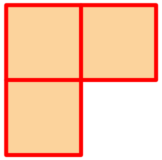
Strong Induction

- Given the proposition $P(n)$ where $n \in \mathbb{N}$, a proof by second principle of mathematical induction (or strong induction) is of the form:
  - Basis step: The proposition $P(0)$ is shown to be true
  - Inductive step: The implication $P(0) \land P(1) \land \cdots \land P(k) \to P(k+1)$ is shown to be true for every $k \in \mathbb{N}$

Example: A Defective Chessboard

- Any 8×8 defective chessboard can be covered with twenty-one triominoes
- Q: How?





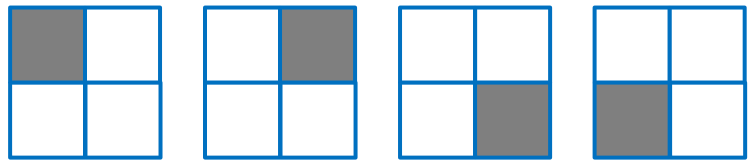
Triomino

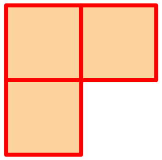
Example: A Defective Chessboard

- Any 8×8 defective chessboard can be covered with twenty-one triominoes
- Any $2^n \times 2^n$ defective chessboard can be covered with $1/3(2^n \times 2^n - 1)$ triominoes
- Prove by mathematical induction!

Proof by Mathematical Induction

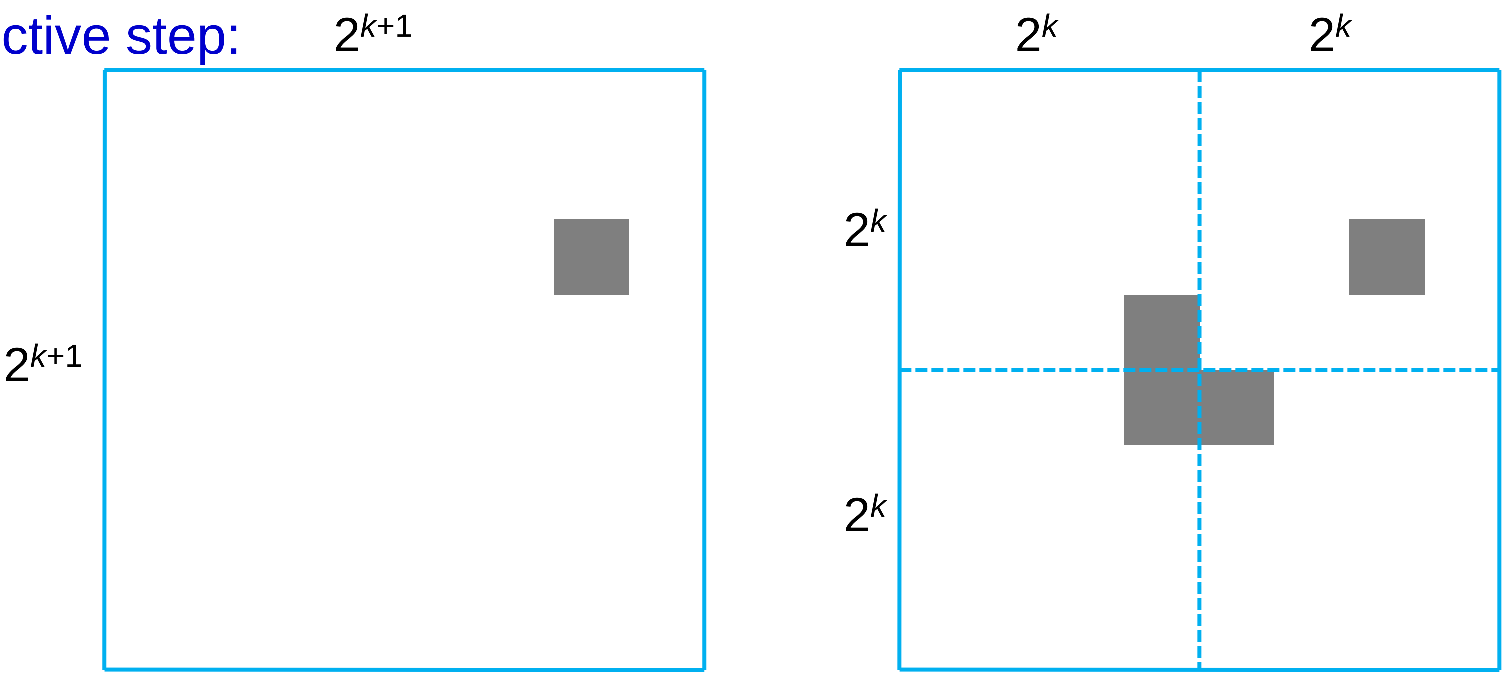
- Any $2^n \times 2^n$ defective chessboard can be covered with $1/3(2^n \times 2^n - 1)$ triominoes
  - Basis step:
    - $n=1$





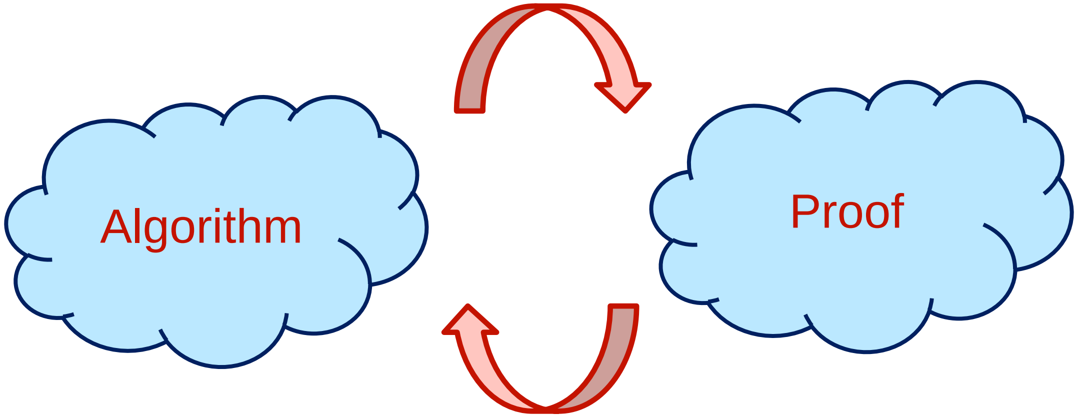
Triomino

  - Inductive step:



Proof vs. Algorithm

- From the defective chessboard example, we can see



Outline

- Content:
  - A first recurrence: merge sort
  - Master theorem
  - Maximum subarray
  - Strassen’s method for matrix multiplication
- Reading:
  - Section 2.3, Chapter 4

Warm Up: Searching

- Problem: Searching
- Input
  - A sorted list of n distinct numbers $A=<a_1, a_2, \ldots, a_n>$
  - Value x
- Output
  - i if $x = A[i]$

- Solution:
  - Naïve idea: compare one by one (linear search)
    - Correct but slow: $\Theta(n)$
  - Better idea?
    - Hint: input is sorted

Use known information
to improve your solution

Binary Search

- **D&C paradigm**
- Divide the problem into several subproblems of the same type
- Conquer subproblems recursively. Solve trivial case directly.
- Combine the solutions to subproblems into an overall solution

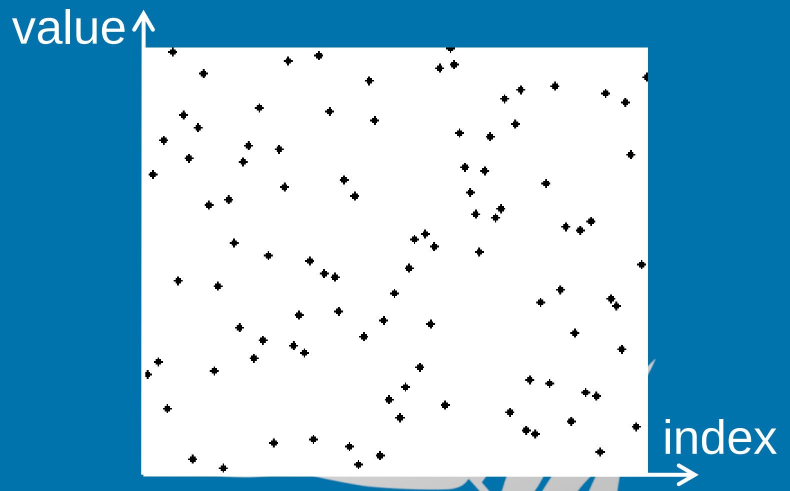
- **Search a sorted array**
- Divide: check the middle element
- Conquer: search the subarray recursively
- Combine: trivial
- $\Theta(\lg n)$

0	5	13	19	22	41	55	68	72	81	98	<	55
						55	68	72	81	98	>	55
						55	68				=	55

6 5 3 1 8 7 2 4

Merge Sort

John von Neumann, 1945



http://en.wikipedia.org/wiki/File:Merge_sort_animation2.gif

Divide and Conquer

- Divide-and-conquer
  - **Divide** the problem into several subproblems of the same type
  - **Conquer** subproblems recursively. Solve trivial case directly
  - **Combine** the solutions of subproblems into an overall solution
- Complexity: recurrence
  - A divide and conquer algorithm is naturally implemented by a recursive procedure
  - The running time of a D&C algorithm is generally represented by a recurrence that bounds the running time recursively in terms of the running time on smaller instances
- Correctness: mathematical induction
  - The basic idea is mathematical induction!

A Divide-and-Conquer Template

Merge sort

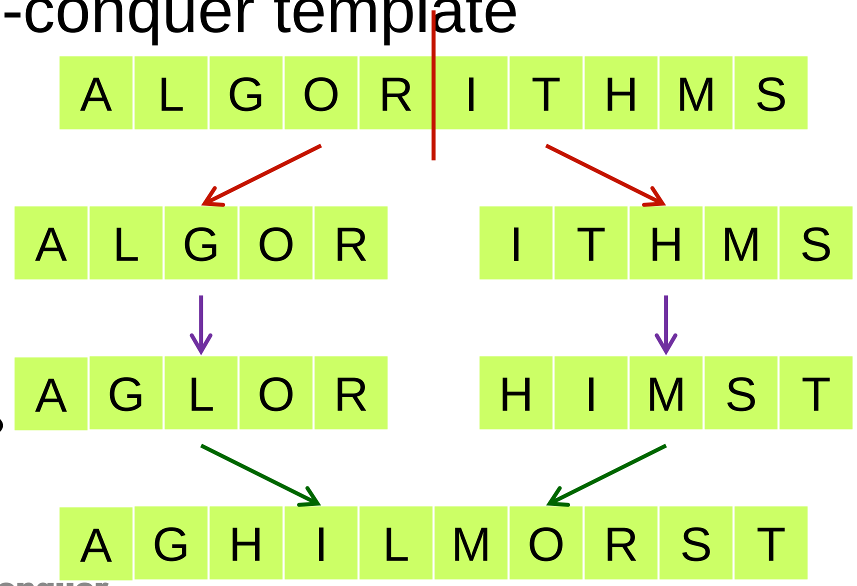
- **Divide** the problem into two subproblems of equal size
- **Conquer** the two subproblems separately by recursion
- **Combine** the two results into an overall solution

- Spend only linear time for the initial division and final combining

Merge Sort (1/2)

- Problem: Sorting
- Input
  - A set of *n* numbers
- Output
  - Sorted list in ascending order
- Solution: many!
- Merge sort fits the divide-and-conquer template
  - **Divide** the input into two halves
  - **Sort** each half recursively
    - Need base case
  - **Merge** two halves into one

Stop recursion



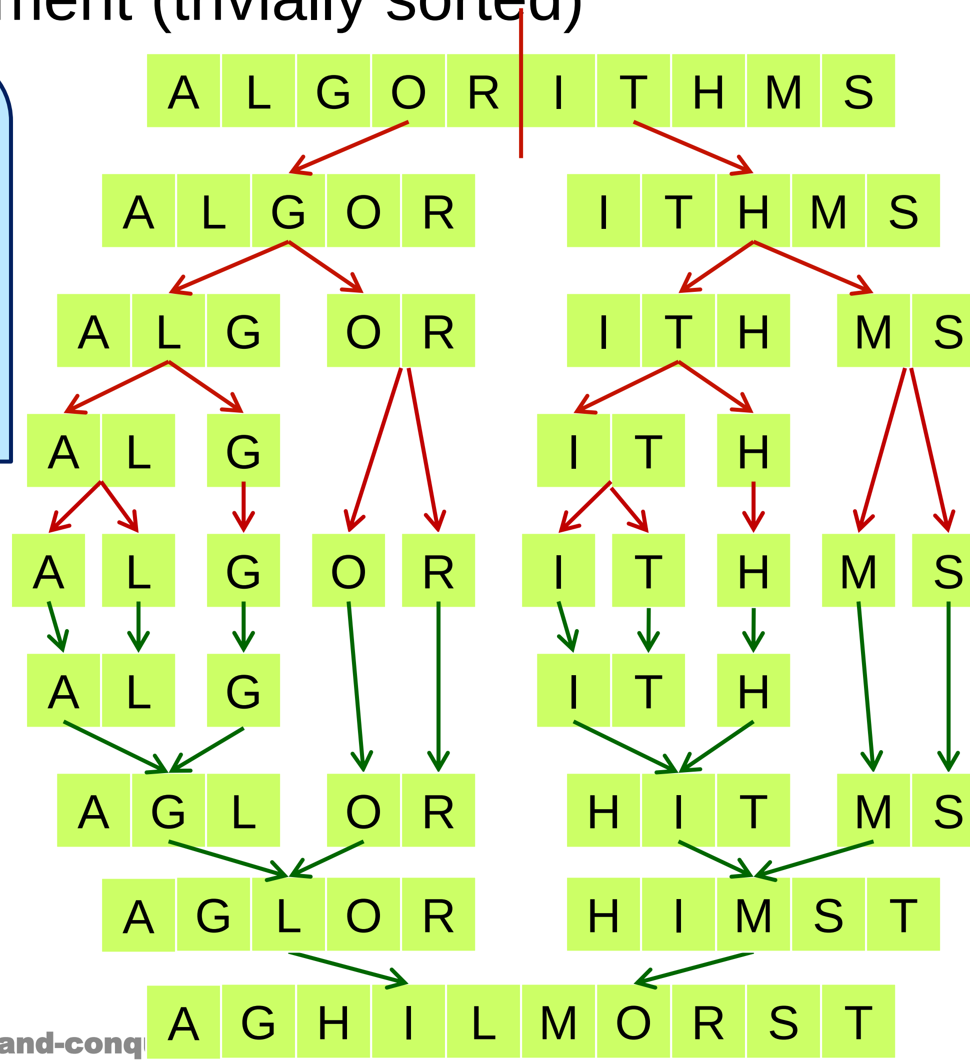
Merge Sort (2/2)

- The base case: single element (trivially sorted)

```
MergeSort(A, p, r)
// A[p..r]: initially unsorted
1.  if (p < r) then
2.      q = ⌊(p+r)/2⌋
3.      MergeSort(A, p, q)
4.      MergeSort(A, q+1, r)
5.      Merge(A, p, q, r)
```

  – MergeSort(A, 1, $A.length$)

- Running time:
  – $T(n)$ for input size n
  – Divide: lines 1-2, $D(n)$
  – Conquer: lines 3-4, $2T(n/2)$
  – Combine: line 5, $C(n)$
  – $T(n) = 2T(n/2) + D(n) + C(n)$

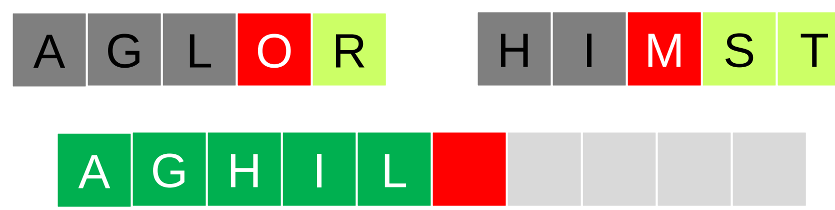


Implementation: Division and Merging

- Running time: $T(n)$
  - $T(n)$ for input size n
  - Divide: lines 1-2, $D(n)$, $\Theta(1)$ for array
  - Conquer: lines 3-4, $2T(n/2)$
  - Combine: line 5, $C(n)$

MergeSort(A, p, r)	$T(n)$
// $A[p..r]$: initially unsorted	
1. **if** ($p < r$) **then**	$\Theta(1)$
2. $q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. MergeSort(A, p, q)	$T(n/2)$
4. MergeSort(A, q+1, r)	$T(n/2)$
5. Merge(A, p, q, r)	$\Theta(n)$

- Efficient merging: linear time?
  - Merge(A, p, q, r) merges two sorted subarrays $A[p..q]$ and $A[q+1..r]$ into sorted $A[p..r]$
  - Linear number of comparisons
  - Use auxiliary arrays
  - $\Theta(n)$

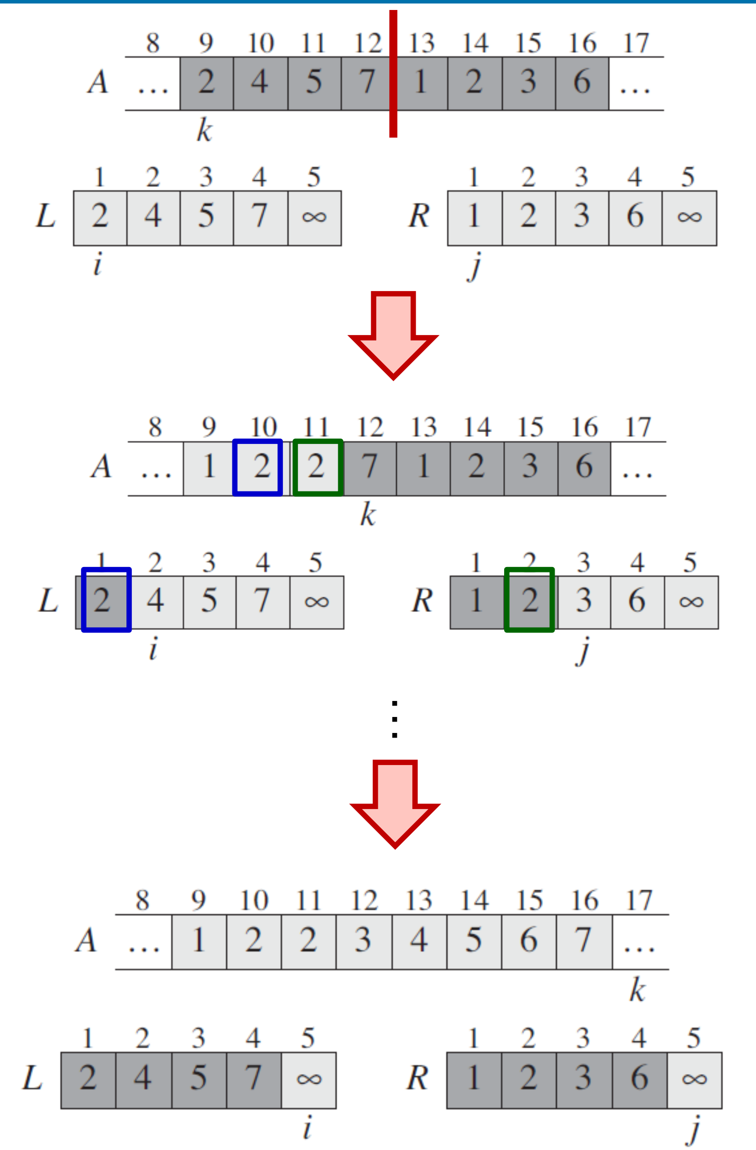


- Merge sort is often the best choice for sorting a linked list

Merge

```
Merge (A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1+1] and R[1..n2+1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1+1] = ∞ // sentinel
R[n2+1] = ∞ // sentinel
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

$\Theta(n)$ time!



Recurrence

- Describes a function recursively in terms of itself
- Describes performance of recursive algorithms

- Recurrence for merge sort
  1. Base case: for $n = 1$, $T(n) = \Theta(1)$
  2. $T(n) = 2T(n/2) + \Theta(1) + \Theta(n)$

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

MergeSort(A, p, r)	$T(n)$
// $A[p..r]$: initially unsorted	
1. **if** ($p < r$) **then**	$\Theta(1)$
2. $q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$
3. MergeSort(A, p, q)	$T(n/2)$
4. MergeSort(A, $q+1$, r)	$T(n/2)$
5. Merge(A, p, q, r)	$\Theta(n)$

- Q: Why not $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$?
- A: Asymptotic bounds are not affected by ignoring $\lfloor \rfloor$ & $\lceil \rceil$

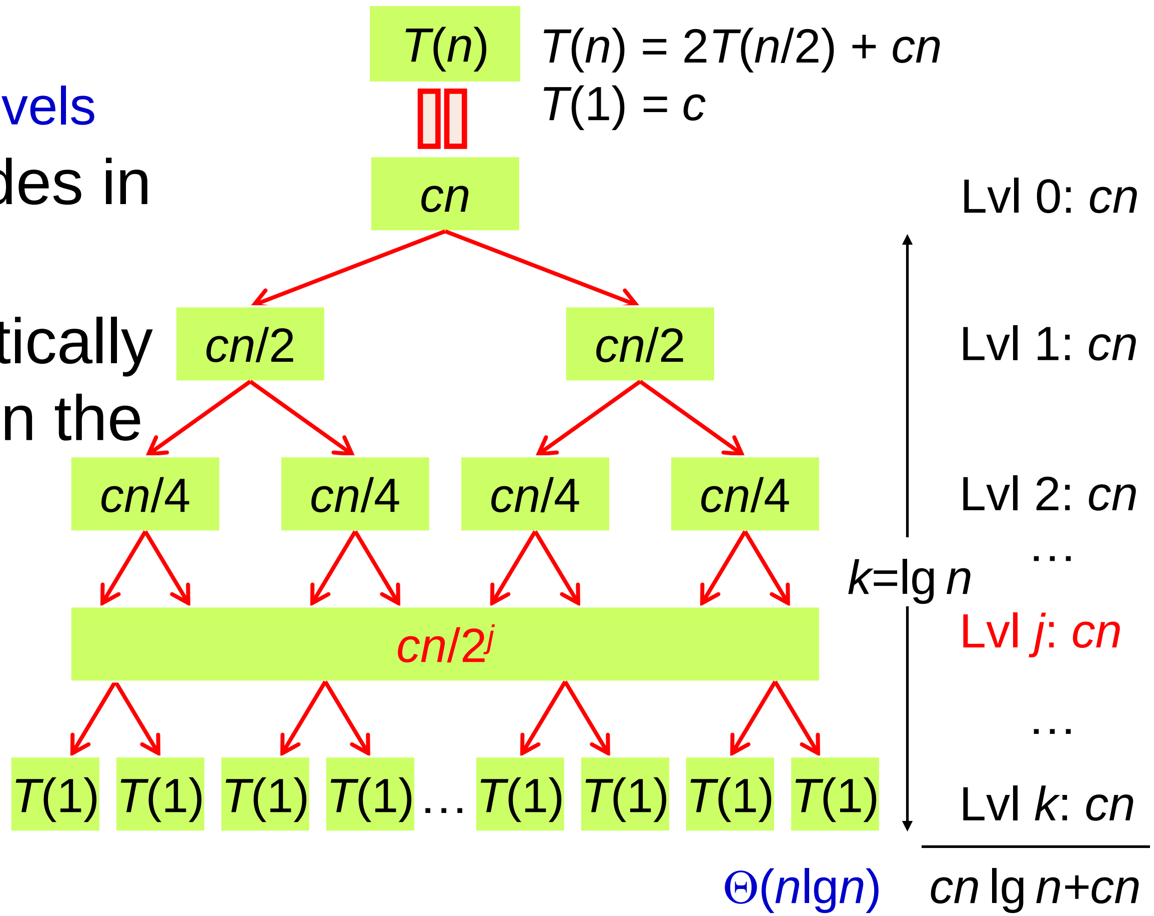
Solving Recurrences

- Three general ways to solve a recurrence
  - Unrolling the recurrence (iteration or recursion tree)
  - Substituting a guess
  - Master theorem

- Initially, we assume n is a power of 2 and replace $\leq$ with $=$
  - $T(n) = 2T(n/2) + cn$
  - Solve the worst case
  - Simplify the problem by omitting floors and ceilings
  - Assume base cases are constant, i.e., $T(n) = \Theta(1)$ for small n

Unrolling – Recursion Tree

- Procedure
  1. Analyzing the first few levels
  2. Identifying a pattern
  3. Summing over all levels
- $T(n)$ = sum of all nodes in the tree
- Merge sort asymptotically beats insertion sort in the worst case
  - insertion sort:
    - **stable**, **in-place**
  - merge sort:
    - **stable**, **not in-place**



Substitution

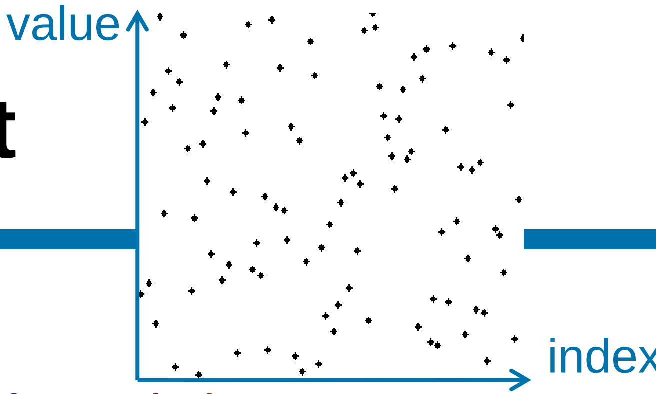
- Any function $T(.)$ satisfying this recurrence
  $T(n) \leq 2T(n/2) + cn$ when $n > 1$, and $T(n) \leq c$ for $n \leq \cancel{1}$ 2
  is bounded by $O(n \lg n)$, when $n > 1$.
- Pf: Guess and prove by induction

assume n is a power of 2

- Suppose we believe that $T(n) \leq cn \lg n$ for all $n \geq \cancel{1}$ 2
  1. Base case:
     - $n = 1$, doesn't hold! Try next!
     - $n = 2$, $T(2) \leq c \leq 2c$. Indeed true
  2. Inductive step:
     - Inductive hypothesis: $T(m) \leq cm \lg m$ for all $m < n$
     - $T(n/2) \leq c(n/2) \lg (n/2)$; $\lg (n/2) = (\lg n) - 1$
     - $T(n) \leq 2T(n/2) + cn$

$$\begin{aligned} &\leq 2c(n/2) \lg (n/2) + cn \\ &= cn\,[(\lg n) - 1] + cn \\ &= (cn \lg n) - cn + cn = cn \lg n \end{aligned}$$

Quick Summary: Merge Sort



- **Divide-and-conquer approach**
  - **Divide** the problem into two subproblems of equal size
  - **Conquer** the two subproblems separately by recursion
  - **Combine** the two results into an overall solution
- **Not in-place**:
  - Use auxiliary arrays
- **Stable:**
  - Numbers with the same value appear in the output array in the same order as they do in the input array
  - 23_a43_b -> 23_a3_b4
- Correctness proof by **mathematical induction**
- Reach the asymptotic lower bound $\Theta(n \lg n)$

Solving More Recurrences

Analyzing Divide-and-Conquer Algorithms

- Recurrence for a divide-and-conquer algorithm

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

  - *a*: # of subproblems
  - *n*/*b*: size of the subproblems
  - *D*(*n*): time to divide the problem of size *n* into subproblems
  - *C*(*n*): time to combine the subproblem solutions to get the answer for the problem of size *n*
- Merge sort:

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ 2T(n/2) + \theta(n), & \text{otherwise} \end{cases}$$

  - *a* = 2: two subproblems
  - *n*/*b* = *n*/2: each subproblem has size ≈ *n*/2
  - *D*(*n*) = $\Theta(1)$: compute midpoint of array
  - *C*(*n*) = $\Theta(n)$: merging by scanning sorted subarrays

Divide-and-Conquer: Binary Search

- Binary search on a **sorted** array:
  - **Divide:** Check middle element
  - **Conquer:** Search the subarray
  - **Combine:** Trivial
- Recurrence:

0	5	13	19	22	41	55	68	72	81	98	<	55
						55	68	72	81	98	>	55
						55	68				=	55

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

$$T(n) = \begin{cases} \theta(1), & \text{if } n \leq c \\ T(n/2) + \theta(1), & \text{otherwise} \end{cases}$$

  - $a = 1$: search one subarray
  - $n/b = n/2$: each subproblem has size $\approx n/2$
  - $D(n) = \Theta(1)$: compute midpoint of array
  - $C(n) = \Theta(1)$: trivial

Solving Recurrences

- Three general methods for solving recurrences
  - **Unrolling:** Convert the recurrence into a summation by expanding some terms and then bound the summation
    - Iteration or recursion tree
  - **Substitution:** Guess a solution and verify it by induction
  - **Master Theorem:** if the recurrence has the form

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

    then **most likely** there is a formula that can be applied
- Two **simplifications** that won't affect asymptotic analysis
  - Ignore floors and ceilings
  - Assume base cases are constant, i.e., $T(n) = \Theta(1)$ for small n

Solving Recurrences: Unrolling by Iteration

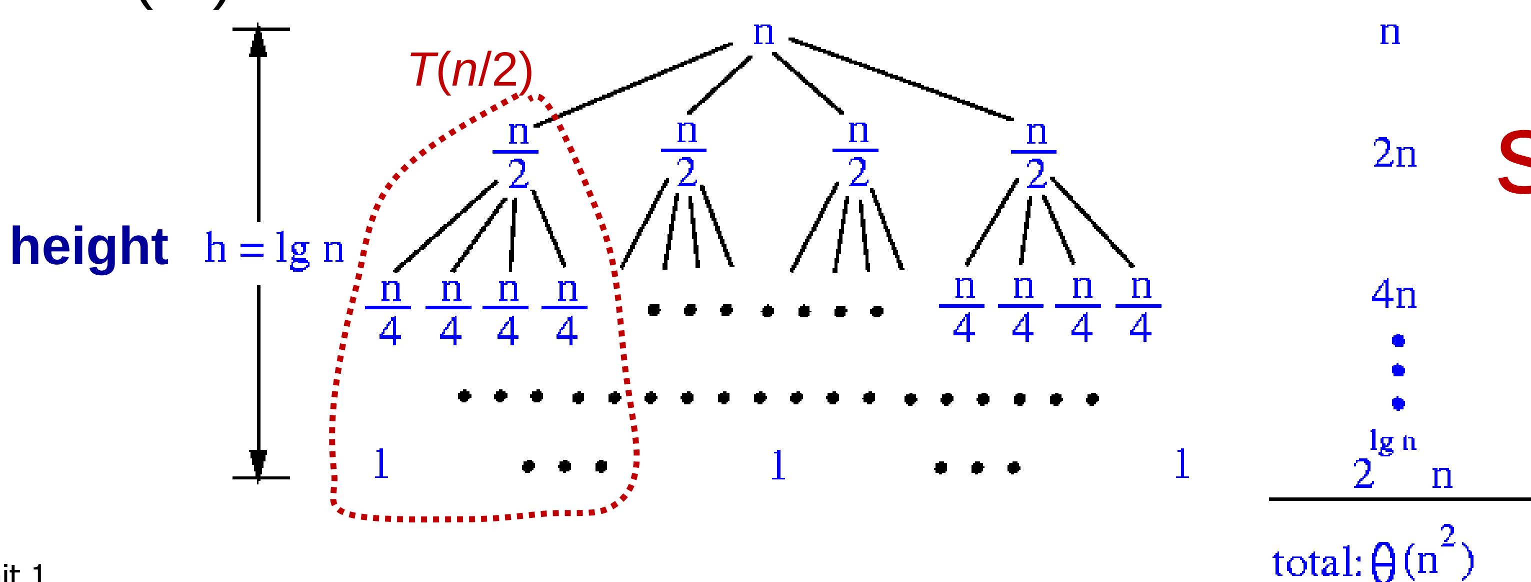
- **Example:** $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4(4T(n/4) + n/2) + n && \text{/* expand */} \\
&= 16T(n/4) + 2n + n && \text{/* simplify */} \\
&= 16(4T(n/8) + n/4) + 2n + n && \text{/* expand */} \\
&= 64T(n/8) + 4n + 2n + n && \text{/* simplify */} \\
&= 4^{\lg n}T(1) + \ldots + 4n + 2n + n && \text{/* \#level} = \lg n \text{ */} \\
&= 4^{\lg n}c + n\textstyle\sum_{k=0}^{\lg n - 1} 2^k && \text{/* convert to summation */} \\
&= cn^{\lg 4} + n\left(\frac{2^{\lg n} - 1}{2 - 1}\right) && \text{/* } a^{\lg b} = b^{\lg a} \text{ */} \\
&= cn^2 + n(n^{\lg 2} - 1) && \text{/* } 2^{\lg n} = n^{\lg 2} \text{ */} \\
&= (c+1)n^2 - n \\
&= \Theta(n^2)
\end{aligned}
$$

skip

Unrolling by Using Recursion Trees

- **Example:** $T(n) = 4T(n/2) + n$
- Root: computation ($D(n) + C(n)$) at top level of recursion
- Node at level i: Subproblem at level i in the recursion
- Height of tree: #level in the recursion
- $T(n)$ = sum of all nodes in the tree
- $T(1) = 1 \Rightarrow T(n)=4T(n/2) + n = n + 2n + 4n + \ldots + 2^{\lg n}n = \Theta(n^2)$



skip

Solving Recurrences: Substitution (Guess & Verify)

1. Guess form of a solution
2. Apply math. induction to find the constant & verify solution
3. Is used to find an upper or a lower bound

- **Example:** Guess $T(n) = 4T(n/2) + n = O(n^3)$ ($T(1) = 1$)
  - Show $T(n) \leq cn^3$ for some $c > 0$ (**we must find** c)

  1. Basis: $T(2) = 4T(1) + 2 = 6 \leq 2^3c$ (pick $c = 1$)
  2. Assume $T(k) \leq ck^3$ for $k < n$, and prove $T(n) \leq cn^3$

$$\begin{aligned} T(n) &= 4\,T(n/2) + n \\ &\leq 4\,(c\,(n/2)^3) + n \\ &= cn^3/2 + n \\ &= cn^3 - (cn^3/2 - n) \\ &\leq cn^3, \end{aligned}$$

where $c \geq 2$ and $n \geq 1$. (**Pick $c \geq 2$ for Steps 1 & 2!**)

skip

- **Useful tricks:** subtract a lower order term, change variables (e.g., $T(n) = T(\sqrt{n}\,) + \lg n$)

Pitfall in Substitution

- **Example:** Guess $T(n) = 2T(n/2) + n = O(n)$ (wrong guess!)
  - Show $T(n) \le cn$ for some $c > 0$ (we must find c)
  1. Basis: $T(2) = 2T(1) + 2 = 4 \le 2c$ (pick $c = 2$)
  2. Assume $T(k) \le ck$ for $k < n$, and prove $T(n) \le cn$

$$\begin{aligned} T(n) &= 2\,T(n/2) + n \\ &\le 2\,(cn/2) + n \\ &= \boldsymbol{cn + n} \\ &= \boldsymbol{O(n)?} \qquad \textbf{/* Wrong!! */} \end{aligned}$$

- What's wrong?
- How to fix? Subtracting a lower-order term may help!

Fixing Wrong Substitution

- Guess $T(n) = 4T(n/2) + n = O(n^2)$ (right guess!)
  - Assume $T(k) \leq ck^2$ for $k < n$, and prove $T(n) \leq cn^2$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4c(n/2)^2 + n \\
&= \mathbf{cn^2 + n} \\
&= \mathbf{O(n^2)} \qquad \textbf{/* Wrong!! */}
\end{aligned}
$$

- Fix by subtracting a lower-order term
  - Assume $T(k) \leq c_1k^2 - c_2k$ for $k < n$, and prove $T(n) \leq c_1n^2 - c_2n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\
&= \mathbf{c_1n^2 - 2c_2n + n} \\
&\leq \mathbf{c_1n^2 - c_2n} \qquad \mathbf{(\text{if } c_2 \geq 1)}
\end{aligned}
$$

  - Pick c_1 big enough to handle initial conditions

Master Theorem

- Let **$a \geq 1$** and **$b > 1$** be constants, *f*(*n*) be a nonnegative function, and *T*(*n*) be defined on nonnegative integers as

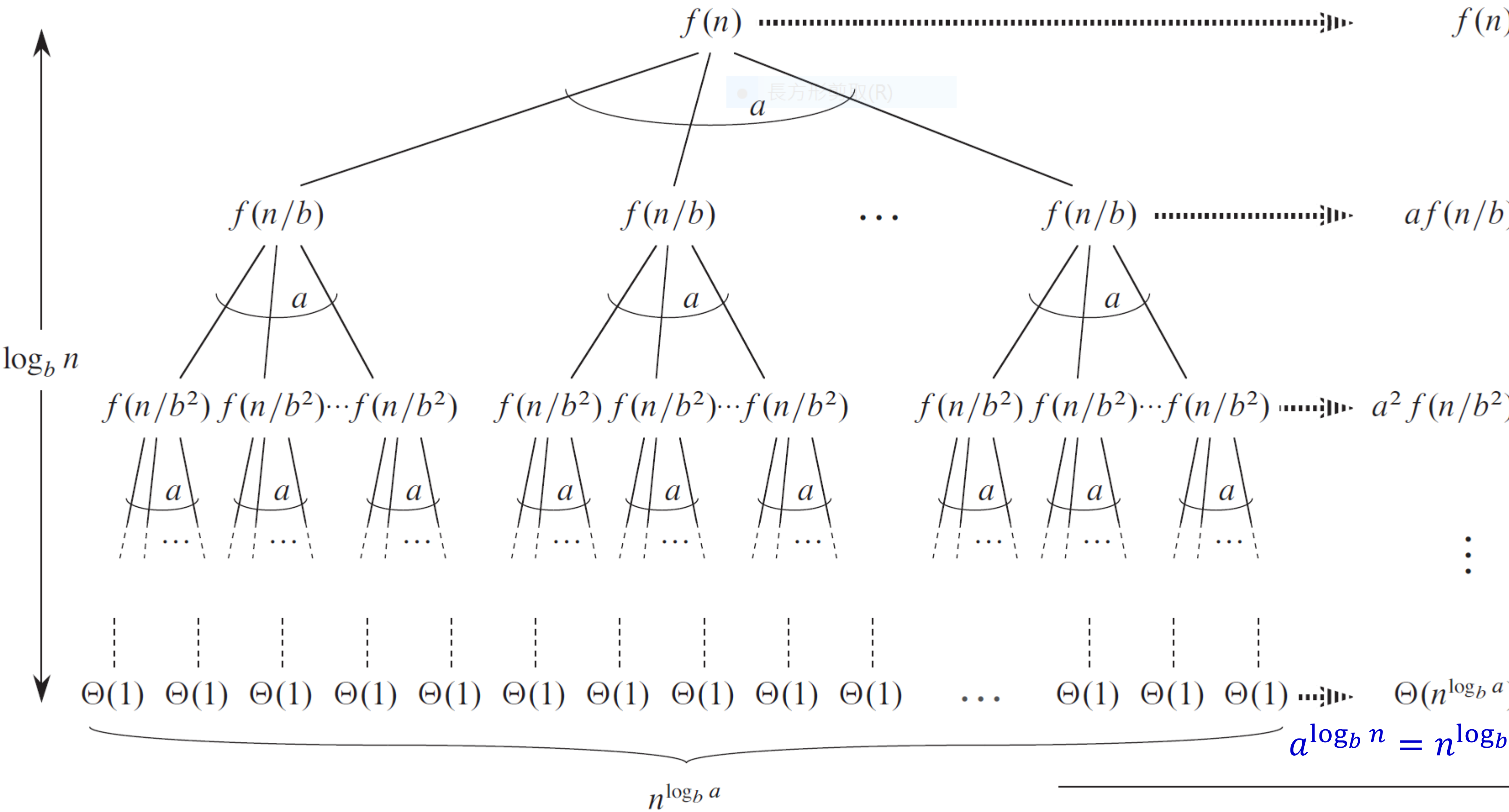
$$T(n) = aT(n/b) + f(n)$$

- Then, *T*(*n*) can be bounded asymptotically as follows:

  1. $T(n) = \Theta(n^{\log_b a})$ if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$
  2. $T(n) = \Theta(n^{\log_b a} \lg n)$ if $f(n) = \Theta\left(n^{\log_b a}\right)$
  3. $T(n) = \Theta(f(n))$ if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ **and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (regularity condition)**

- Intuition: compare *f*(*n*) with $\Theta\left(n^{\log_b a}\right)$
  - Case 1: *f*(*n*) is polynomially smaller than $\Theta\left(n^{\log_b a}\right)$
  - Case 2: *f*(*n*) is asymptotically equal to $\Theta\left(n^{\log_b a}\right)$
  - Case 3: *f*(*n*) is polynomially larger than $\Theta\left(n^{\log_b a}\right)$

General Form: $T(n) = aT(n/b) + f(n)$



$a^{\log_b n} = n^{\log_b a}$

Total: $\Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

Case 3: Regularity condition: $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n

Examples

- $T(n) = 2^n T(n/2) + n^n \Longrightarrow$ Does not apply (a is not constant)
- $T(n) = 0.5T(n/2) + 1/n \Longrightarrow$ Does not apply ($a < 1$)
- $T(n) = 64T(n/8) - n^2 \log n \Longrightarrow$ Does not apply ($f(n)$ is not positive)
- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ vs. n^2
  Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use Case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$
- $T(n) = 2T(n/2) + n$
  - n vs. n
  - Case 2 applies: $T(n) = n \lg n$

Examples

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

[This formulation of Case 2 is more general than in Theorem 4.1, and it is given in Exercise 4.6-2.]

($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)

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

(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)

Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
  Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$

Examples

- $T(n) = 5T(n/2) + \Theta(n^3)$
  $n^{\log_2 5}$ vs. n^3
  Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  Check regularity condition
  $af(n/b) = 5(n/2)^3 = 5n^3/8 \leq cn^3$ for $c = 5/8 < 1$
  Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

- $T(n) = 27T(n/3) + \Theta(n^3/\lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3/\lg n = n^3 \lg^{-1} n \neq \Theta(n^3 \lg^k n)$ for any $k \geq 0$.
  Cannot use the master method.

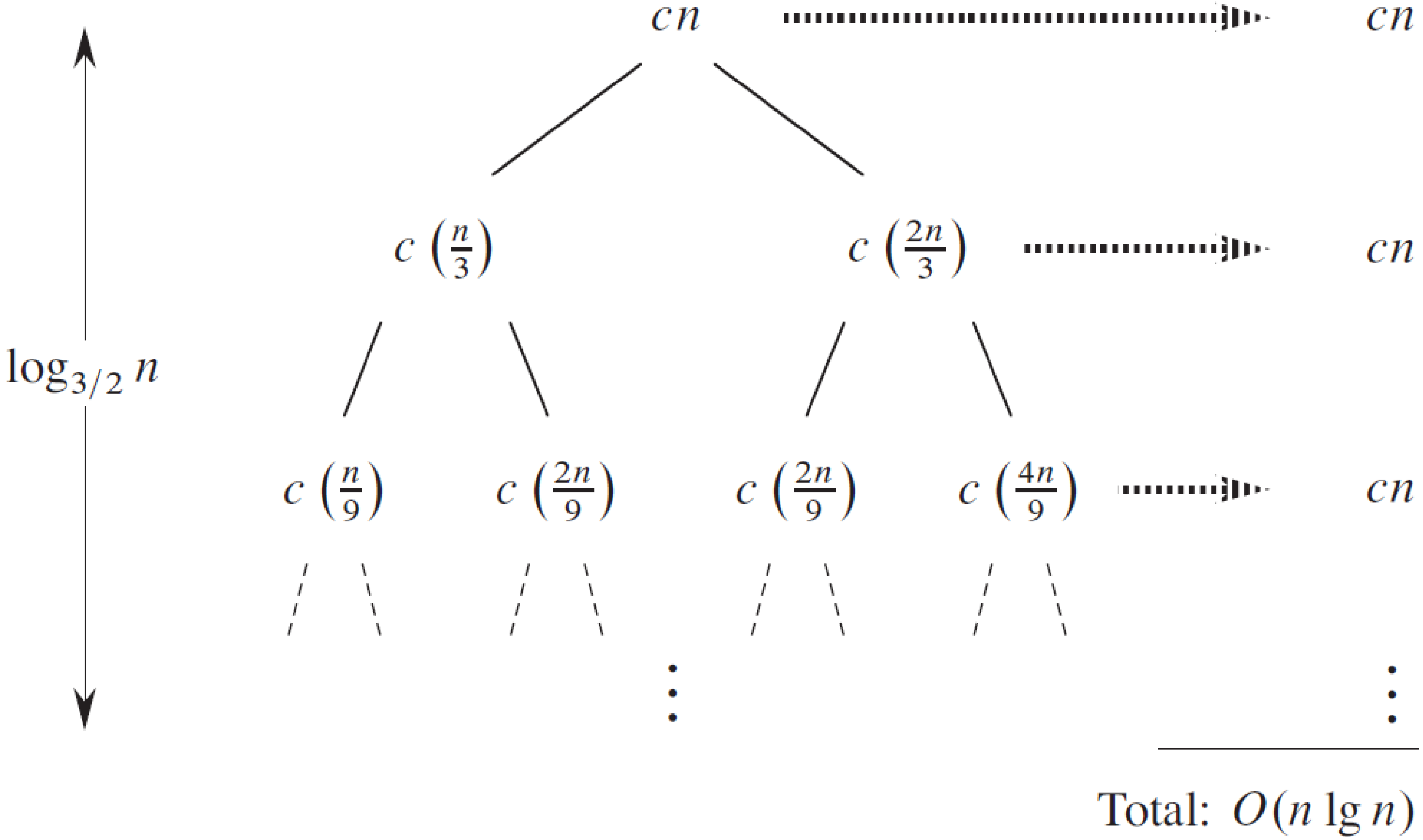
Changing Variables

- Consider the recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

- Floor/ceiling signs can be ignored for asymptotic analysis
- Let $m = \lg n \rightarrow T(n) = T(2^m) = 2T(2^{m/2}) + m$
- Let $S(m) = T(2^m) \rightarrow T(n) = S(m) = 2S(m/2) + m$
- By Master Theorem, $S(m) = O(m \lg m)$
- So $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$

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



How about Θ?

The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Maximum Subarray

Maximum Subarray

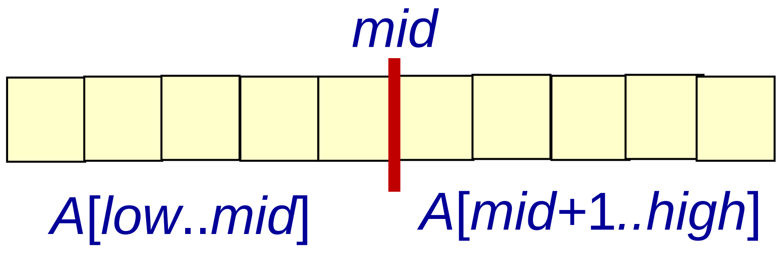
- Input: An array $A[1..n]$ of positive/negative numbers
- Output: **Indices *i* and *j*** such that $A[i..j]$ has the greatest sum of any nonempty, contiguous subarray of A, along with the **sum of the values in *A*[*i*..*j*]**
- "Impractical" example: maximize your earning in a stock market

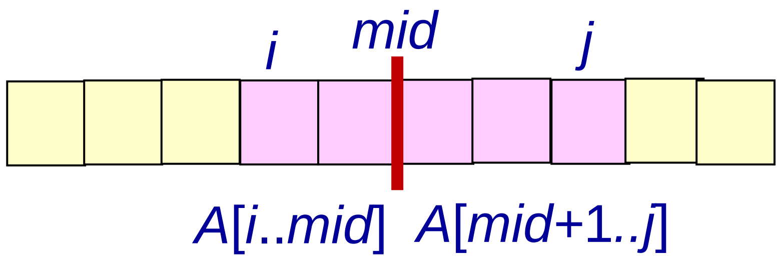
Day	0	1	2	3	4
Price	10	11	7	9	6
Change		1	-4	2	-3

- Brute force: check all $C(n, 2) = \Theta(n^2)$ subarrays. Time??
- Better algorithm? Focus on daily changes

Divide-and-Conquer Maximum Subarray

- Subproblem: Find a maximum subarray of $A[low..high]$
- **Divide** the subarray into two subarrays of "equal" size at the midpoint mid: $A[low..mid]$ & $A[mid+1..high]$
- **Conquer** by finding maximum subarrays of $A[low..mid]$ & $A[mid+1..high]$
- **Combine** by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three (the subarray crossing the midpoint and the two solutions found in the conquer step)

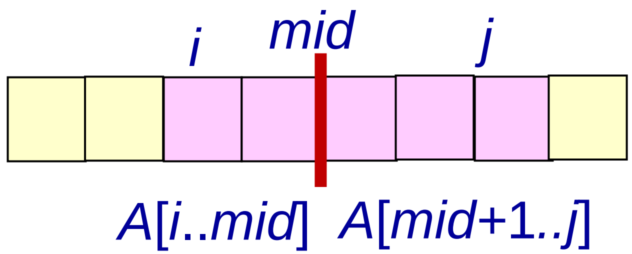




Finding the Maximum Subarray Crossing Midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i..mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1..j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

$\Theta(n)$ time!!



- The maximum subarray crossing the midpoint can be solved in linear time
  - Any subarray crossing the midpoint *A*[*mid*] is made of two *subarrays A*[*i*..*mid*] and *A*[*mid*+1.. *j*]
  - Find maximum subarrays of the form *A*[*i*..*mid*] *and A*[*mid*+1.. *j*], and then combine them

Finding the Maximum Subarray

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])              // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

- Divide: $\Theta(1)$ time; Conquer: $2T(n/2)$; Combine: $\Theta(n)$ for finding the maximum subarray crossing the midpoint

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

$\Theta(n \lg n)$ time!!

Strassen's Method

Volker Strassen, 1969

Volker Strassen, Gaussian Elimination is not Optimal, *Numer. Math*. 13, p. 354-356, 1969

Matrix Multiplication

- Input: Two $n \times n$ matrices, $A = (a_{ij})$ and $B = (b_{ij})$
- Output: $n \times n$ matrix $C = (c_{ij})$, where $C = AB$,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \text{for } i, j = 1, 2, \ldots, n$$

- Need to compute n^2 entries of C; each entry is the sum of n values: $\Theta(n^3)$-time algorithm

```
Square-Matrix-Multiply(A, B, n)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik b_kj
return C
```

Simple Divide-and-Conquer Method

- Can we multiply matrices in ***o***(n^3) time?
- Can a simple divide-and-conquer method work?
- $n = 2^k$: partition each of A, B, C into 4 $n/2 \times n/2$ matrices

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- $C_{11} = A_{11}B_{11} + A_{12}B_{21},\ C_{12} = A_{11}B_{12} + A_{12}B_{22}$
  $C_{21} = A_{21}B_{11} + A_{22}B_{21},\ C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Recursive-Mat-Mult(A, B)
1. $n = A$.rows
2. let C be a new $n \times n$ matrix
3. **if** $n == 1$
4. $\quad c_{11} = a_{11}\, b_{11}$
5. **else** partition A, B, C into $n/2 \times n/2$ submatrices (as above)
6. $\quad C_{11}$ = Recursive-Mat-Mult(A_{11}, B_{11}) + Recursive-Mat-Mult(A_{12}, B_{21})
7. $\quad C_{12}$ = Recursive-Mat-Mult(A_{11}, B_{12}) + Recursive-Mat-Mult(A_{12}, B_{22})
8. $\quad C_{21}$ = Recursive-Mat-Mult(A_{21}, B_{11}) + Recursive-Mat-Mult(A_{22}, B_{21})
9. $\quad C_{22}$ = Recursive-Mat-Mult(A_{21}, B_{12}) + Recursive-Mat-Mult(A_{22}, B_{22})
10. **return** C

Simple Divide-and-Conquer Method Is Not Better

Recursive-Mat-Mult(A, B)
1. n = A.rows
2. let C be a new $n \times n$ matrix
3. **if** n == 1
4. $c_{11} = a_{11}\, b_{11}$
5. **else** partition A, B, C into $n/2 \times n/2$ submatrices (as before)
6. C_{11} = Recursive-Mat-Mult(A_{11}, B_{11}) + Recursive-Mat-Mult(A_{12}, B_{21})
7. C_{12} = Recursive-Mat-Mult(A_{11}, B_{12}) + Recursive-Mat-Mult(A_{12}, B_{22})
8. C_{21} = Recursive-Mat-Mult(A_{21}, B_{11}) + Recursive-Mat-Mult(A_{22}, B_{21})
9. C_{22} = Recursive-Mat-Mult(A_{21}, B_{12}) + Recursive-Mat-Mult(A_{22}, B_{22})
10. **return** C

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2), & \text{if } n > 1 \end{cases}$$

***$T(n) = \Theta(n^3)$*, not better!!**

- Dividing takes $\Theta(1)$ time using index calculations ($\Theta(n^2)$ time, otherwise)
- Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices: $8T(n/2)$ time
- Combining takes $\Theta(n^2)$ time to add $n/2 \times n/2$ matrices 4 times

Strassen's Method

- **Keys:** Make the recursion tree less bushy
  - Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8
  - Cost a constant number more additions of $n/2 \times n/2$ matrices; can still absorb the constant into the $\Theta(n^2)$ term

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2), & \text{if } n > 1 \end{cases}$$

$T(n) = \Theta(n^{\lg 7})$, **better!!**

Strassen's Matrix Multiplication

1. Partition each of the matrices into 4 $n/2 \times n/2$ submatrices. — $\Theta(1)$ or $\Theta(n^2)$
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$; each is $n/2 \times n/2$ and is the sum or difference of two matrices created in previous step. — $\Theta(n^2)$
3. Recursively compute 7 $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$. — $7T(n/2)$
4. Compute $n/2 \times n/2$ submatrices of C by adding and subtracting various combinations of the P_i — $\Theta(n^2)$

Strassen's Method

1. Partition each matrix into 4 $n/2 \times n/2$ submatrices
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$
   - $S_1 = B_{12} - B_{22}$, $S_2 = A_{11} + A_{12}$, $S_3 = A_{21} + A_{22}$, $S_4 = B_{21} - B_{11}$
     $S_5 = A_{11} + A_{22}$, $S_6 = B_{11} + B_{22}$, $S_7 = A_{12} - A_{22}$, $S_8 = B_{21} + B_{22}$
     $S_9 = A_{11} - A_{21}$, $S_{10} = B_{11} + B_{12}$
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$
   - $P_1 = A_{11}S_1 = A_{11} B_{12} - A_{11} B_{22}$, $P_2 = S_2 B_{22} = A_{11} B_{22} + A_{12} B_{22}$
   - $P_3 = S_3B_{11} = A_{21} B_{11} + A_{22} B_{11}$, $P_4 = A_{22}S_4 = A_{22} B_{21} - A_{22} B_{11}$
   - $P_5 = S_5S_6 = A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22}$
   - $P_6 = S_7S_8 = A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22}$
   - $P_7 = S_9S_{10} = A_{11} B_{11} + A_{11} B_{12} - A_{21} B_{11} - A_{21} B_{12}$
4. Add/subtract P_i to construct $n/2 \times n/2$ submatrices of C
   - $C_{11} = P_5 + P_4 - P_2 + P_6$, $C_{12} = P_1 + P_2$,
   - $C_{21} = P_3 + P_4$, $C_{22} = P_5 + P_1 - P_3 - P_7$

Illustration: Strassen's Method

- Expand each right-hand side, replacing each P_i with the submatrices of A and B that form it, and cancel terms:

$C_{11} = P_5 + P_4 - P_2 + P_6$

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$$\begin{array}{l}
A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} \\
\qquad - A_{22}\cdot B_{11} + A_{22}\cdot B_{21} \\
\qquad - A_{11}\cdot B_{22} - A_{12}\cdot B_{22} \\
\qquad - A_{22}\cdot B_{22} - A_{22}\cdot B_{21} + A_{12}\cdot B_{22} + A_{12}\cdot B_{21} \\
\hline
A_{11}\cdot B_{11} + A_{12}\cdot B_{21}
\end{array}$$

$C_{12} = P_1 + P_2$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$$\begin{array}{l}
A_{11}\cdot B_{12} - A_{11}\cdot B_{22} \\
\qquad + A_{11}\cdot B_{22} + A_{12}\cdot B_{22} \\
\hline
A_{11}\cdot B_{12} + A_{12}\cdot B_{22}
\end{array}$$

$C_{21} = P_3 + P_4$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$$\begin{array}{l}
A_{21}\cdot B_{11} + A_{22}\cdot B_{11} \\
\qquad - A_{22}\cdot B_{11} + A_{22}\cdot B_{21} \\
\hline
A_{21}\cdot B_{11} + A_{22}\cdot B_{21}
\end{array}$$

$C_{22} = P_5 + P_1 - P_3 - P_7$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

$$\begin{array}{l}
A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} \\
\qquad - A_{11}\cdot B_{22} + A_{11}\cdot B_{12} \\
\qquad - A_{22}\cdot B_{11} - A_{21}\cdot B_{11} \\
- A_{11}\cdot B_{11} - A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} \\
\hline
A_{22}\cdot B_{22} + A_{21}\cdot B_{12}
\end{array}$$

Issues with Strassen's Method

- The constant factor hidden in the $\Theta(n^{\lg 7})$ running time is larger than that of the $\Theta(n^3)$-time Square-Matrix-Multiply
  - Fast implementation considers the matrix size over a "crossover point" for applying Strassen's Method; use Square-Matrix-Multiply for smaller problems, instead
- More efficient process for spare matrices exists
  - Strassen's Method is mainly for dense matrices
- Is less numerically stable than Square-Matrix-Multiply
  - Larger errors might accumulate
- The submatrices formed during recursion consume space