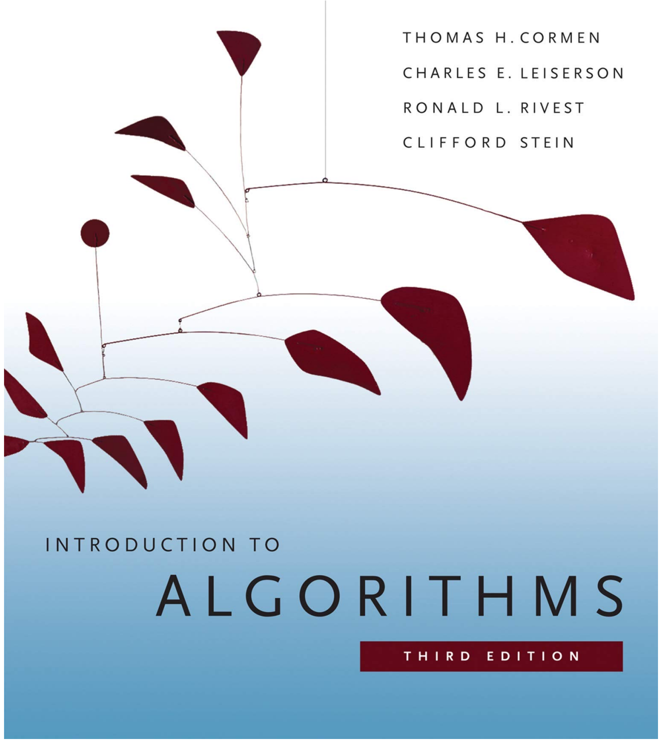
Algorithm



Chap 2: Asymptotic Notations

Outline

- Review of last lecture
- Sum of series
- Analyzing recursive algorithms

L' Hopital's rule

$$\lim_{n \to \infty} f(n) / g(n) = \lim_{n \to \infty} f(n)' / g(n)'$$

Condition:

If both lim f(n) and lim g(n) = ∞ or 0

Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \sqrt{2\pi}\, n^{n+1/2} e^{-n}$$

or

$$n! \approx \text{(constant)}\; n^{n+1/2} e^{-n}$$

Properties of asymptotic notations

- Textbook page 51
- Transitivity

  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$

  $\Rightarrow f(n) = \Theta(h(n))$

  (holds true for o, O, ω, and Ω as well).
- Symmetry

  $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry

  $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

logarithms

- $\lg n = \log_2 n$
- $\ln n = \log_e n$, $e \approx 2.718$
- $\lg^k n = (\lg n)^k$
- $\lg \lg n = \lg (\lg n) = \lg^{(2)} n$
- $\lg(^k) n = \lg \lg \lg \ldots \lg n$
- $\lg^2 4 = ?$
- $\lg^{(2)} 4 = ?$
- Compare $\lg^k n$ vs $\lg^{(k)} n$?

Useful rules for logarithms

- For all $a > 0$, $b > 0$, $c > 0$, the following rules hold
- $\log_b a = \log_c a / \log_c b = \lg a / \lg b$
- $\log_b a^n = n \log_b a$
- $b^{\log_b a} = a$
- $\log (ab) = \log a + \log b$
  - $\lg (2n) = ?$
- $\log (a/b) = \log (a) – \log(b)$
  - $\lg (n/2) = ?$
  - $\lg (1/n) = ?$
- $\log_b a = 1 / \log_a b$

Useful rules for exponentials

- For all $a > 0$, $b > 0$, $c > 0$, the following rules hold
- $a^0 = 1 \quad (0^0 = ?)$
- $a^1 = a$
- $a^{-1} = 1/a$
- $(a^m)^n = a^{mn}$
- $(a^m)^n = (a^n)^m$
- $a^m a^n = a^{m+n}$

More advanced dominance ranking

$$n^n >> n! >> 3^n >> 2^n >> n^3 >> n^2 >> n^{1+\varepsilon} >> n \log n \sim \log n!$$

$$>> n >> n / \log n >> \sqrt{n} >> n^\varepsilon >> \log^3 n >> \log^2 n >> \log n$$

$$>> \log n / \log \log n >> \log \log n >> \log^{(3)} n >> \alpha(n) >> 1$$

Find the order of growth for sums

- $T(n) = \sum_{i=1..n} i = \Theta(n^2)$
- $T(n) = \sum_{i=1..n} \log(i) = ?$
- $T(n) = \sum_{i=1..n} n / 2^i = ?$
- $T(n) = \sum_{i=1..n} 2^i = ?$
- …

- How to find out the actual order of growth?
  – Math…
  – Textbook Appendix A.1 (page 1058-60)

Arithmetic series

- An arithmetic series is a sequence of numbers such that the difference of any two successive members of the sequence is a constant.

  e.g.: 1, 2, 3, 4, 5

  or 10, 12, 14, 16, 18, 20
- In general:

$$a_j = a_{j-1} + d$$ ← Recursive definition

Or: $$a_j = a_1 + (j-1)d$$ ← Closed form, or explicit formula

Sum of arithmetic series

If $a_1, a_2, \ldots, a_n$ is an arithmetic series, then

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

e.g. 1 + 3 + 5 + 7 + … + 99 = ?

Geometric series

- A geometric series is a sequence of numbers such that the ratio between any two successive members of the sequence is a constant.

  e.g.: 1, 2, 4, 8, 16, 32

  or 10, 20, 40, 80, 160

  or 1, ½, ¼, 1/8, 1/16

- In general:

$$a_j = r a_{j-1}$$ ← Recursive definition

Or: $$a_j = r^{j-1} a_0$$ ← Closed form, or explicit formula

Sum of geometric series

$$\sum_{i=0}^{n} r^i = \begin{cases} (1-r^{n+1})/(1-r) & \text{if r < 1} \\ (r^{n+1}-1)/(r-1) & \text{if r > 1} \\ n+1 & \text{if r = 1} \end{cases}$$

$$\sum_{i=0}^{n} 2^i = ?$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{1}{2^i} = ?$$

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = ?$$

Sum of geometric series

$$\sum_{i=0}^{n} r^i = \begin{cases} (1-r^{n+1})/(1-r) & \text{if r < 1} \\ (r^{n+1}-1)/(r-1) & \text{if r > 1} \\ n+1 & \text{if r = 1} \end{cases}$$

$$\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1} - 1 \approx 2^{n+1}$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{1}{2^i} = \lim_{n\to\infty} \sum_{i=0}^{n} (1/2)^i = \frac{1}{1-1/2} = 2$$

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^i} = \lim_{n\to\infty} \sum_{i=0}^{n} \left(1/2\right)^0 - \left(1/2\right)^0 = 2 - 1 = 1$$

Important formulas

$$\sum_{i=1}^{n} 1 = n \in \Theta(n)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1} \in \begin{cases} \Theta(1) & (r<1) \\ \Theta(r^n) & (r>1) \end{cases}$$

$$\sum_{i=1}^{n} i^2 \approx \frac{n^3}{3} \in \Theta(n^3)$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2 \in \Theta(n2^n)$$

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\lg n)$$

$$\sum_{i=1}^{n} \lg i \in \Theta(n \lg n)$$

Sum manipulation rules

$$\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$$

$$\sum_i c a_i = c \sum_i a_i$$

$$\sum_{i=m}^{n} a_i = \sum_{i=m}^{x} a_i + \sum_{i=x+1}^{n} a_i$$

Example:

$$\sum_{i=1}^{n} (4i + 2^i) = ?$$

$$\sum_{i=1}^{n} \frac{n}{2^i} = ?$$

Sum manipulation rules

$$\sum_i (a_i + b_i) = \sum_i a_i + \sum_i b_i$$

$$\sum_i c a_i = c \sum_i a_i$$

$$\sum_{i=m}^{n} a_i = \sum_{i=m}^{x} a_i + \sum_{i=x+1}^{n} a_i$$

Example:

$$\sum_{i=1}^{n} (4i + 2^i) = 4\sum_{i=1}^{n} i + \sum_{i=1}^{n} 2^i = 2n(n+1) + 2^{n+1} - 2$$

$$\sum_{i=1}^{n} \frac{n}{2^i} = n \sum_{i=1}^{n} \frac{1}{2^i} \approx n$$

Examples

- $\sum_{i=1..n} n / 2^i = n * \sum_{i=1..n} (½)^i = ?$

- using the formula for geometric series:

  $\sum_{i=0..n} (½)^i = 1 + ½ + ¼ + \ldots (½)^n = 2$

- Application: algorithm for allocating dynamic memories

Examples

- $\sum_{i=1..n} \log(i) = \log 1 + \log 2 + \ldots + \log n$

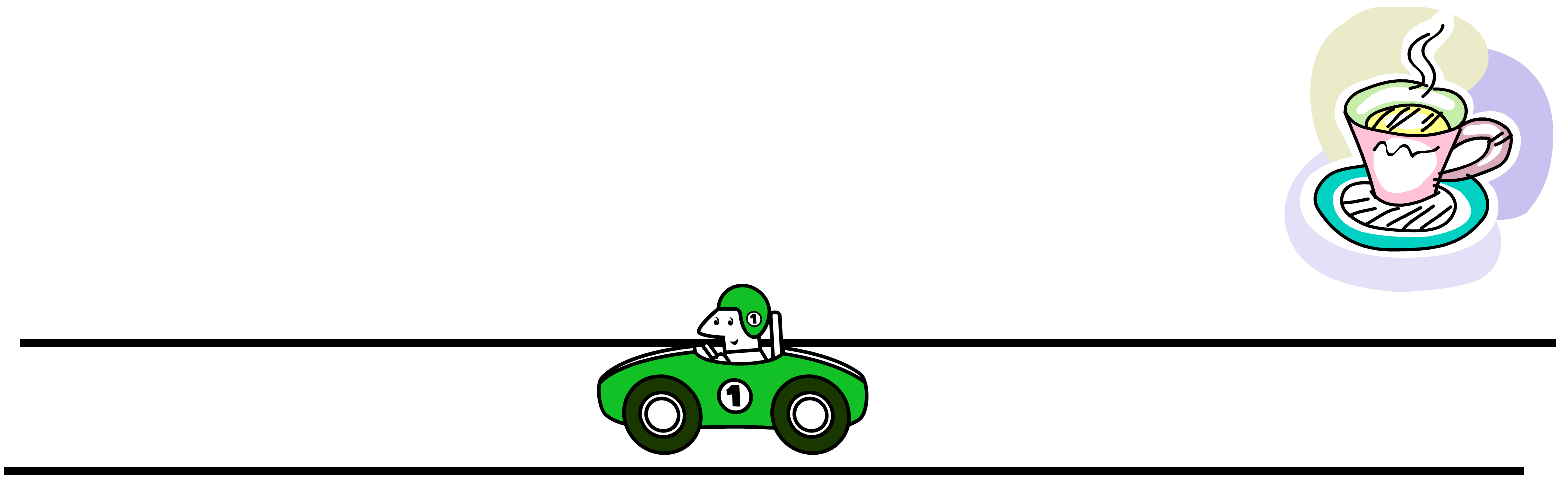
  $= \log 1 \times 2 \times 3 \times \ldots \times n$

  $= \log n!$

  $= n \log n$

- Application: algorithm for selection sort using priority queue

Problem of the day



How do you find a coffee shop if you don’t know on which direction it might be?

Recursive definition of sum of series

- $T(n) = \sum_{i=0..n} i$ is equivalent to:

$$\begin{cases} T(n) = T(n-1) + n & \longleftarrow \text{Recurrence relation} \\ T(0) = 0 & \longleftarrow \text{Boundary condition} \end{cases}$$

- $T(n) = \sum_{i=0..n} a^i$ is equivalent to:

$$\begin{cases} T(n) = T(n-1) + a^n \\ T(0) = 1 \end{cases}$$

Recursive definition is often intuitive and easy to obtain. It is very useful in analyzing recursive algorithms, and some non-recursive algorithms too.

Analyzing recursive algorithms

Recursive algorithms

- General idea:
  - Divide a large problem into smaller ones
    - By a constant ratio
    - By a constant or some variable
  - Solve each smaller one *recursively* or *explicitly*
  - Combine the solutions of smaller ones to form a solution for the original problem

Divide and Conquer

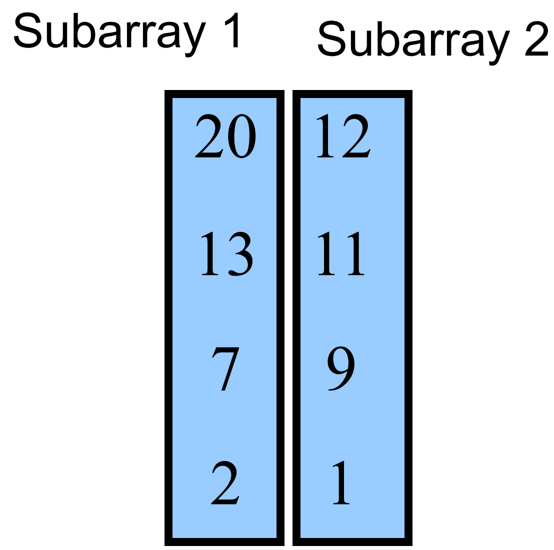
Merge sort

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

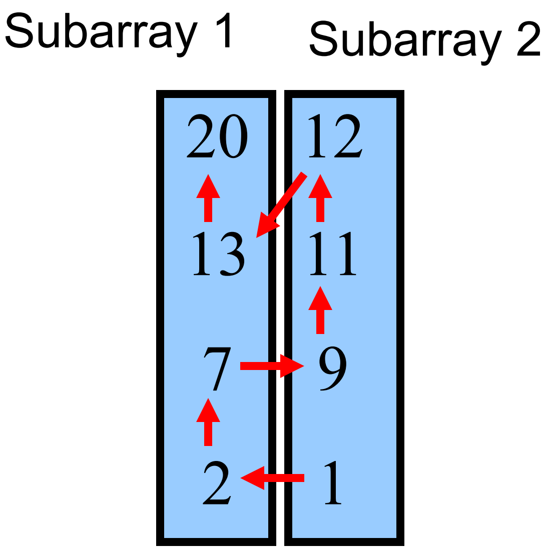
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays

20	12
13	11
7	9
2	1

Merging two sorted arrays

20	12
13	11
7	9
2	1

Merging two sorted arrays

20	12
13	11
7	9
2	1

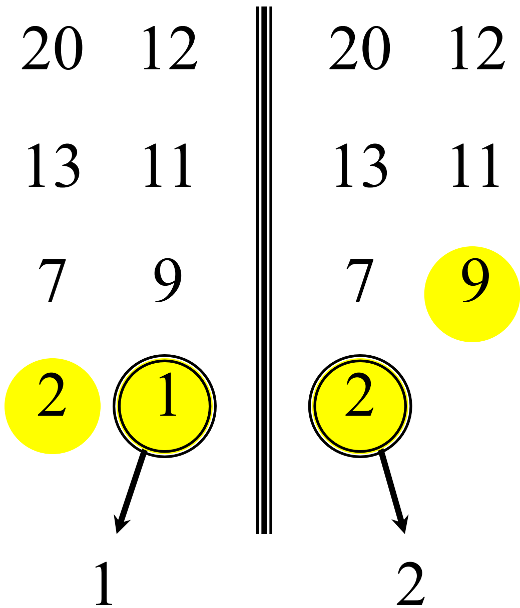
1

Merging two sorted arrays

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

1

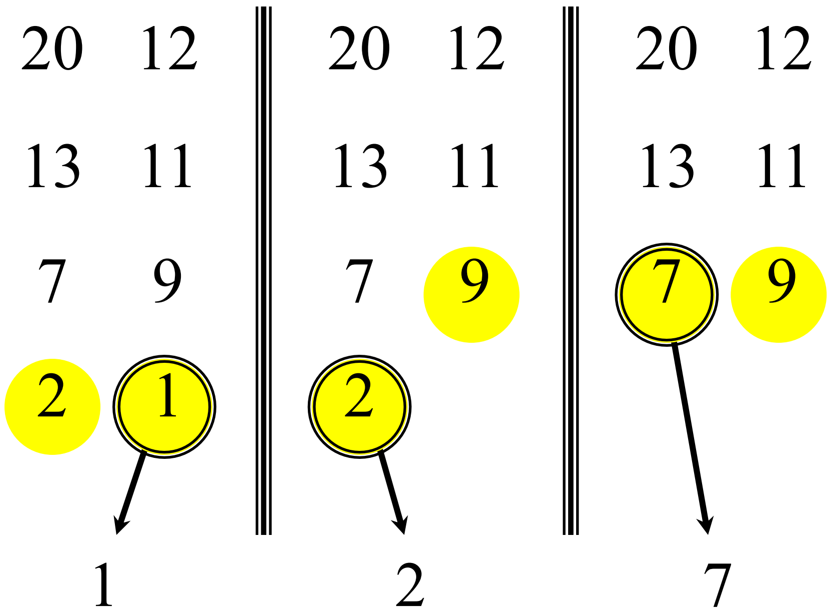
Merging two sorted arrays



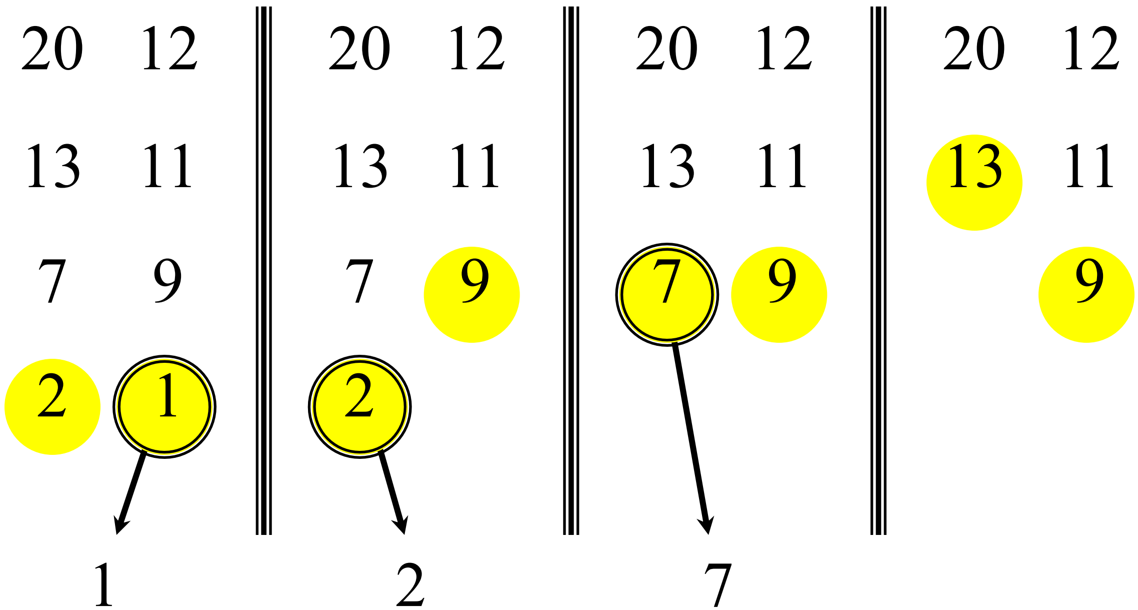
Merging two sorted arrays

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

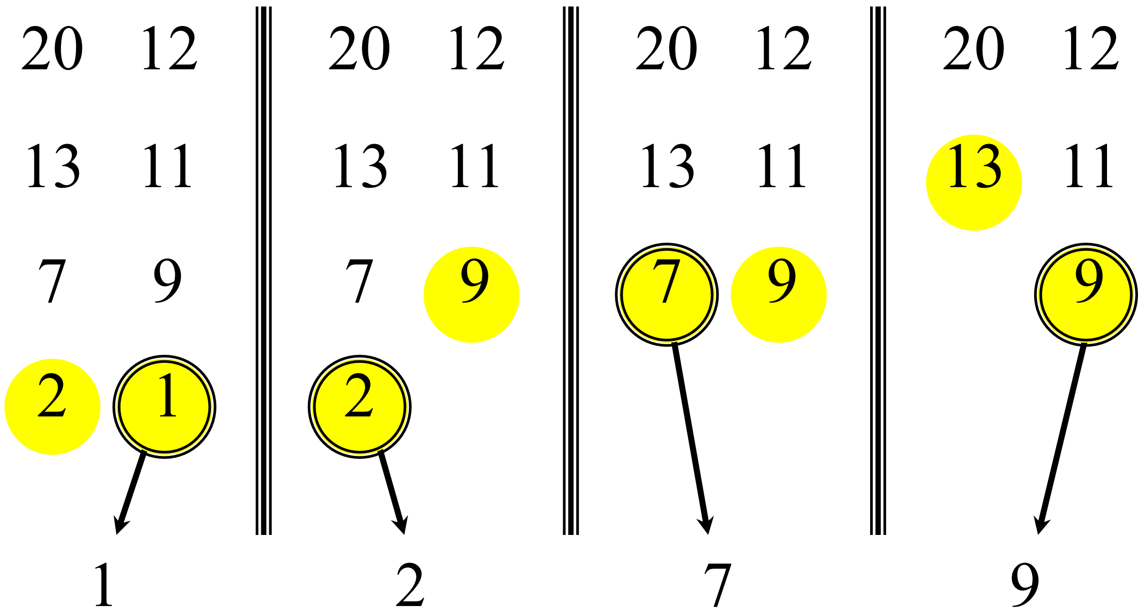
Merging two sorted arrays



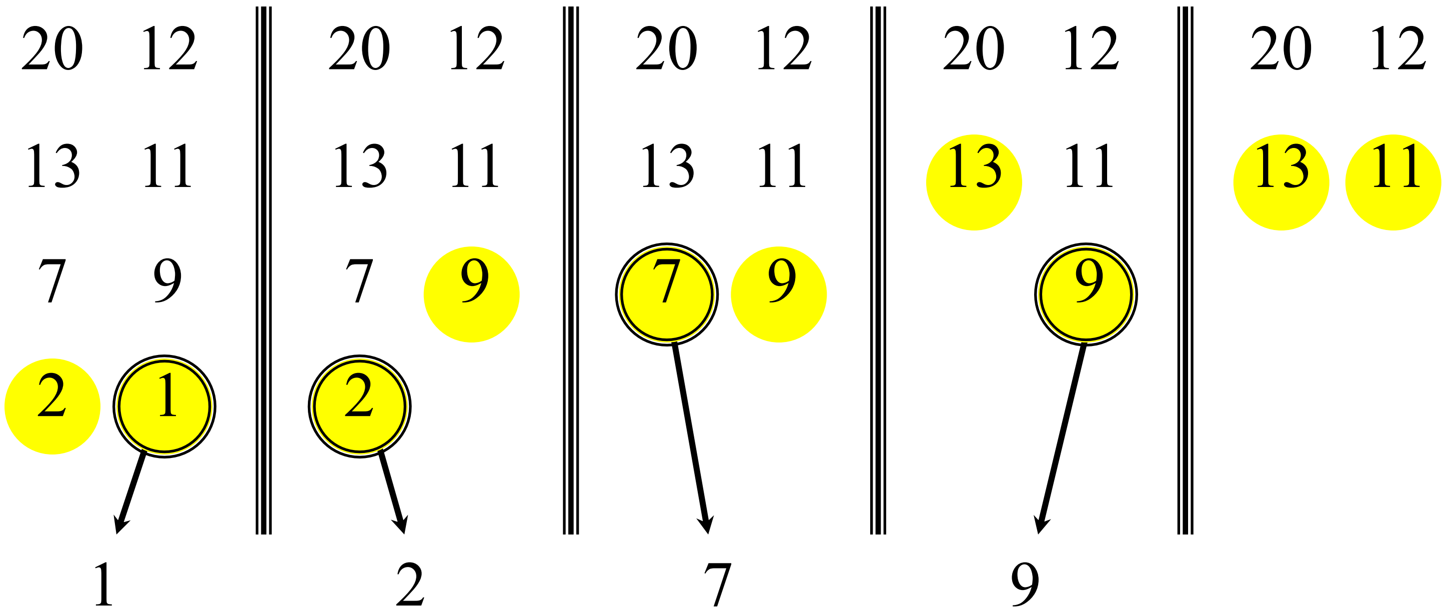
Merging two sorted arrays



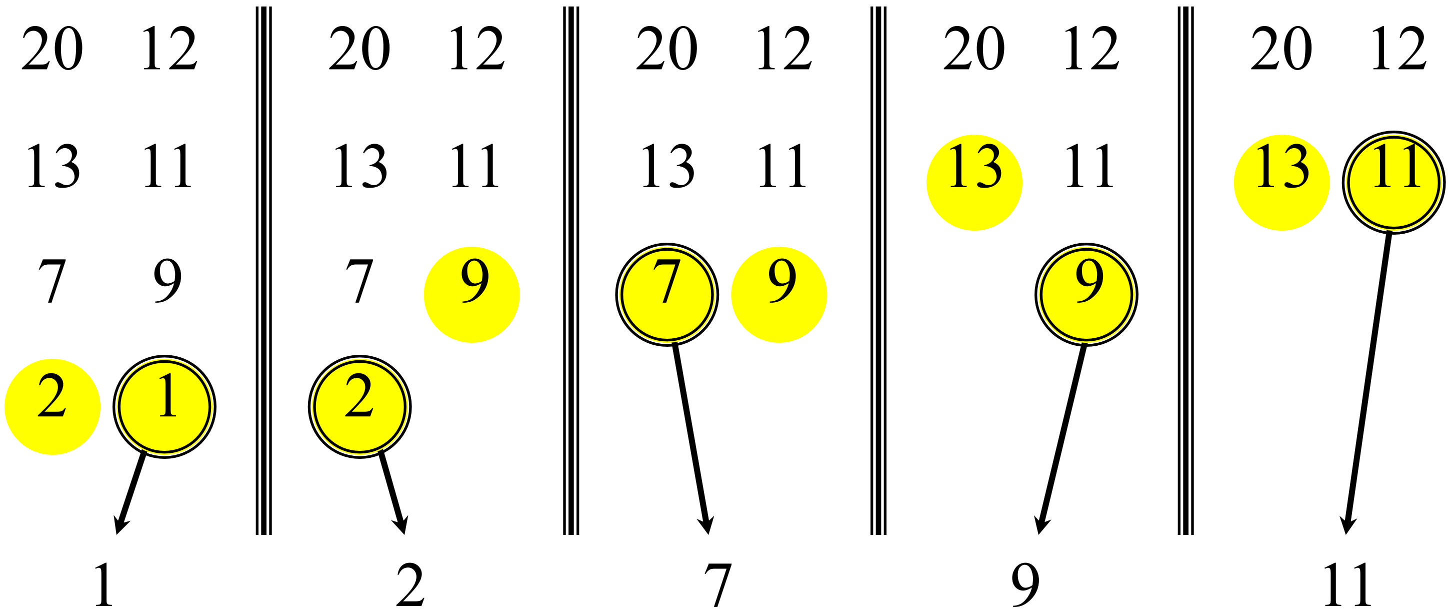
Merging two sorted arrays



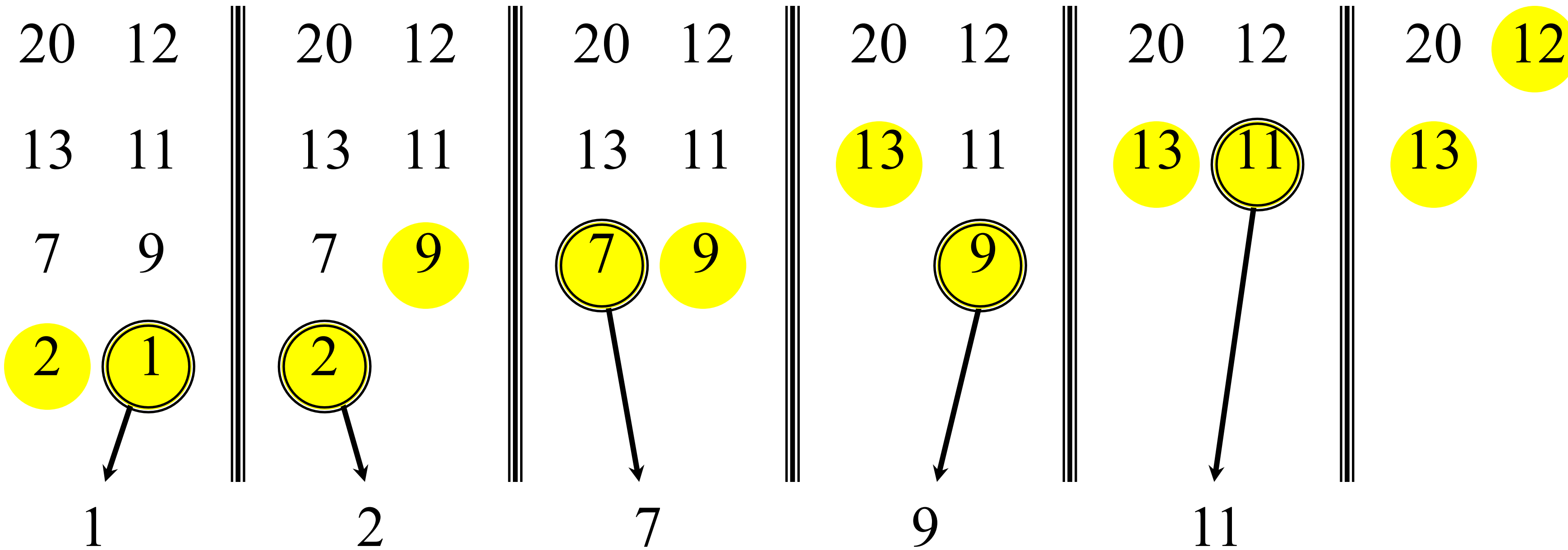
Merging two sorted arrays



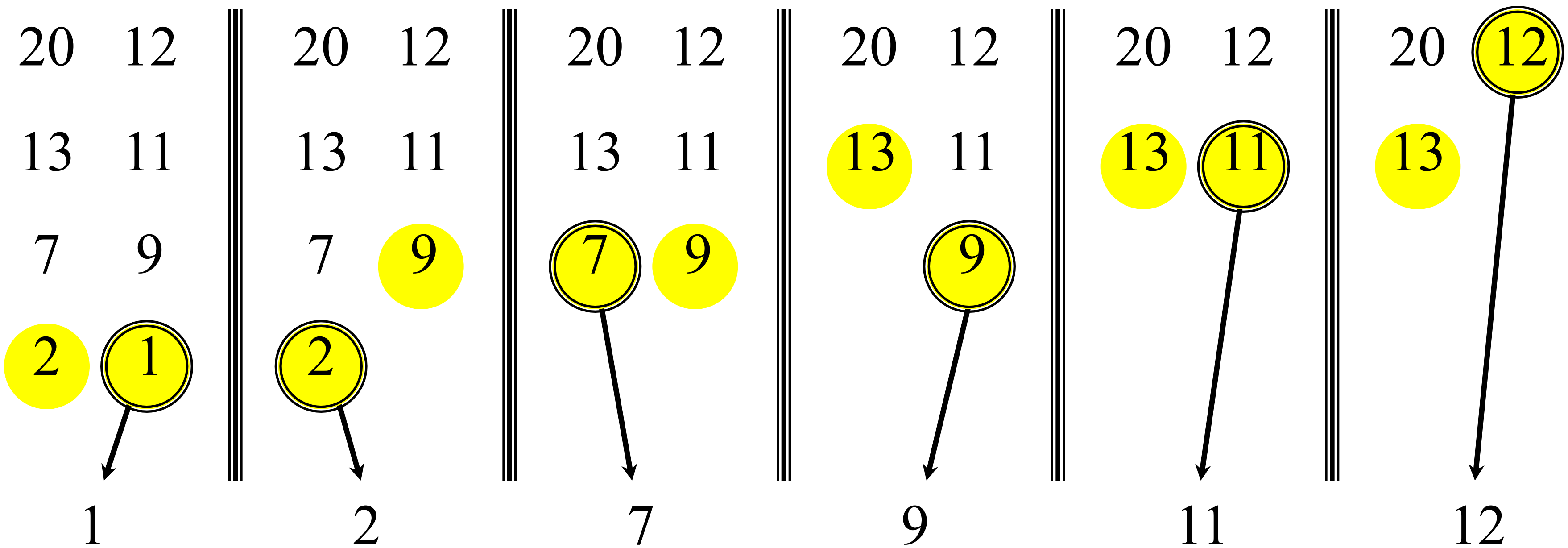
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



How to show the correctness of a recursive algorithm?

- By induction:
  - Base case: prove it works for small examples
  - Inductive hypothesis: assume the solution is correct for all sub-problems
  - Step: show that, if the inductive hypothesis is correct, then the algorithm is correct for the original problem.

Correctness of merge sort

MERGE-SORT $A[1 \mathrel{.\,.} n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \mathrel{.\,.} \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \mathrel{.\,.} n\,]$.
3. "***Merge***" the 2 sorted lists.

Proof:

1. Base case: if n = 1, the algorithm will return the correct answer because A[1..1] is already sorted.
2. Inductive hypothesis: assume that the algorithm correctly sorts $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$.
3. Step: if $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$ are both correctly sorted, the whole array $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil+1..n]$ is sorted after merging.

How to analyze the time-efficiency of a recursive algorithm?

- Express the running time on input of size n as a function of the running time on smaller problems

Analyzing merge sort

$T(n)$	**MERGE-SORT** $A[1 \,.\,.\, n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$f(n)$	3. ***"Merge"*** the 2 sorted lists

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

Analyzing merge sort

1. ***Divide:*** Trivial.
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Merge two sorted subarrays

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

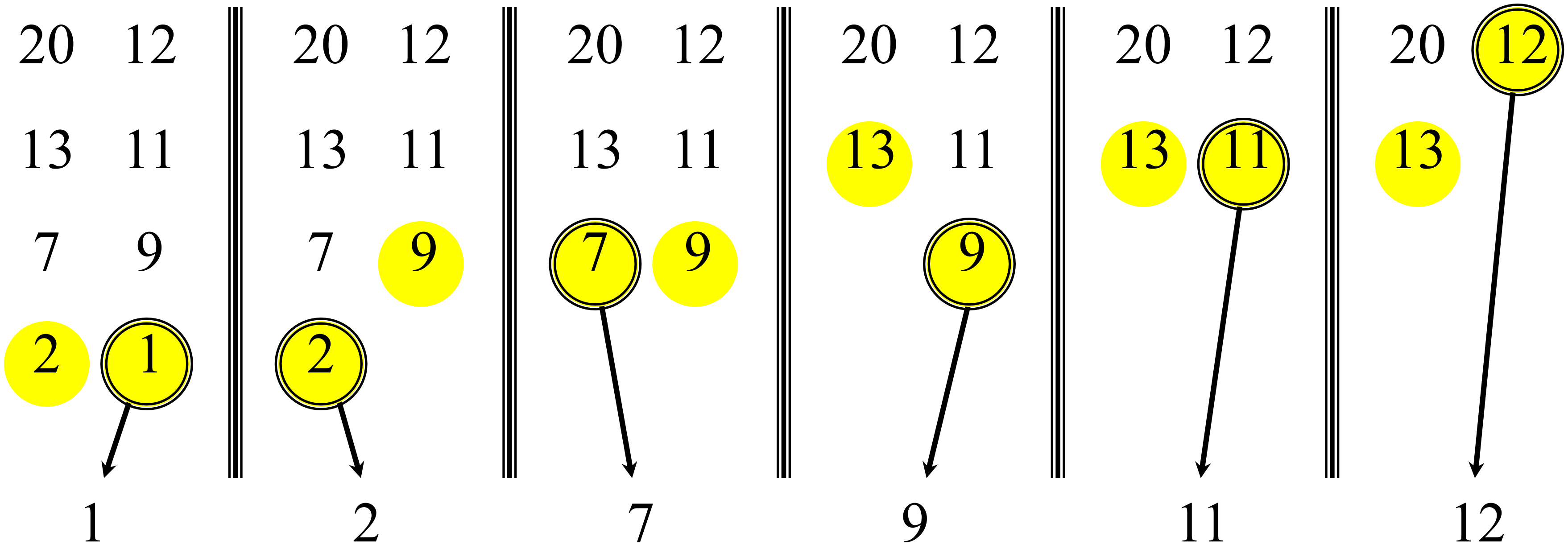
 # subproblems

subproblem size

Dividing and Combining

1. What is the time for the base case? Constant
2. What is $f(n)$?
3. What is the growth order of $T(n)$?

Merging two sorted arrays



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

Recurrence for merge sort

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

- Later we shall often 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.

- But what does $T(n)$ solve to? I.e., is it $O(n)$ or $O(n^2)$ or $O(n^3)$ or …?

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3	5	7	8	9	12	15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3	5	7	8	9	12	15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3	5	7	8	9	12	15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3	5	7	8	9	12	15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3	5	7	8	9	12	15

Binary Search

To find an element in a sorted array, we

1. Check the middle element
2. If ==, we've found it
3. else if less than wanted, search right half
4. else search left half

Example: Find 9

3 5 7 8 9 12 15

Binary Search

```
BinarySearch (A[1..N], value) {
    if (N == 0)
        return -1;                  // not found
    mid = (1+N)/2;
    if (A[mid] == value)
        return mid;                 // found
    else if (A[mid] < value)
        return BinarySearch (A[mid+1, N], value)
    else
        return BinarySearch (A[1..mid-1], value);
}
```

What’s the recurrence relation for its running time?

Recurrence for binary search

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

$$T(1) = \Theta(1)$$

Recursive Insertion Sort

RecursiveInsertionSort(A[1..n])

1. if (n == 1) do nothing;
2. ***RecursiveInsertionSort***(A[1..n-1]);
3. Find index *i* in A such that A[i] <= A[n] < A[i+1];
4. Insert A[n] after A[i];

Recurrence for insertion sort

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

$$T(1) = \Theta(1)$$

Compute factorial

Factorial (n)

```
if (n == 1) return 1;
return n * Factorial (n-1);
```

- Note: here we use n as the size of the input. However, usually for such algorithms we would use log(n), i.e., the bits needed to represent n, as the input size.

Recurrence for computing factorial

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

$$T(1) = \Theta(1)$$

- Note: here we use n as the size of the input. However, usually for such algorithms we would use log(n), i.e., the bits needed to represent n, as the input size.

What do these mean?

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

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

$$T(n) = T(n/2) + 1$$

$$T(n) = 2T(n/2) + 1$$

Challenge: how to solve the recurrence to get a closed form, e.g. $T(n) = \Theta(n^2)$ or $T(n) = \Theta(n \lg n)$, or at least some bound such as $T(n) = O(n^2)$?

Solving recurrence

- Running time of many algorithms can be expressed in one of the following two recursive forms

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

or

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

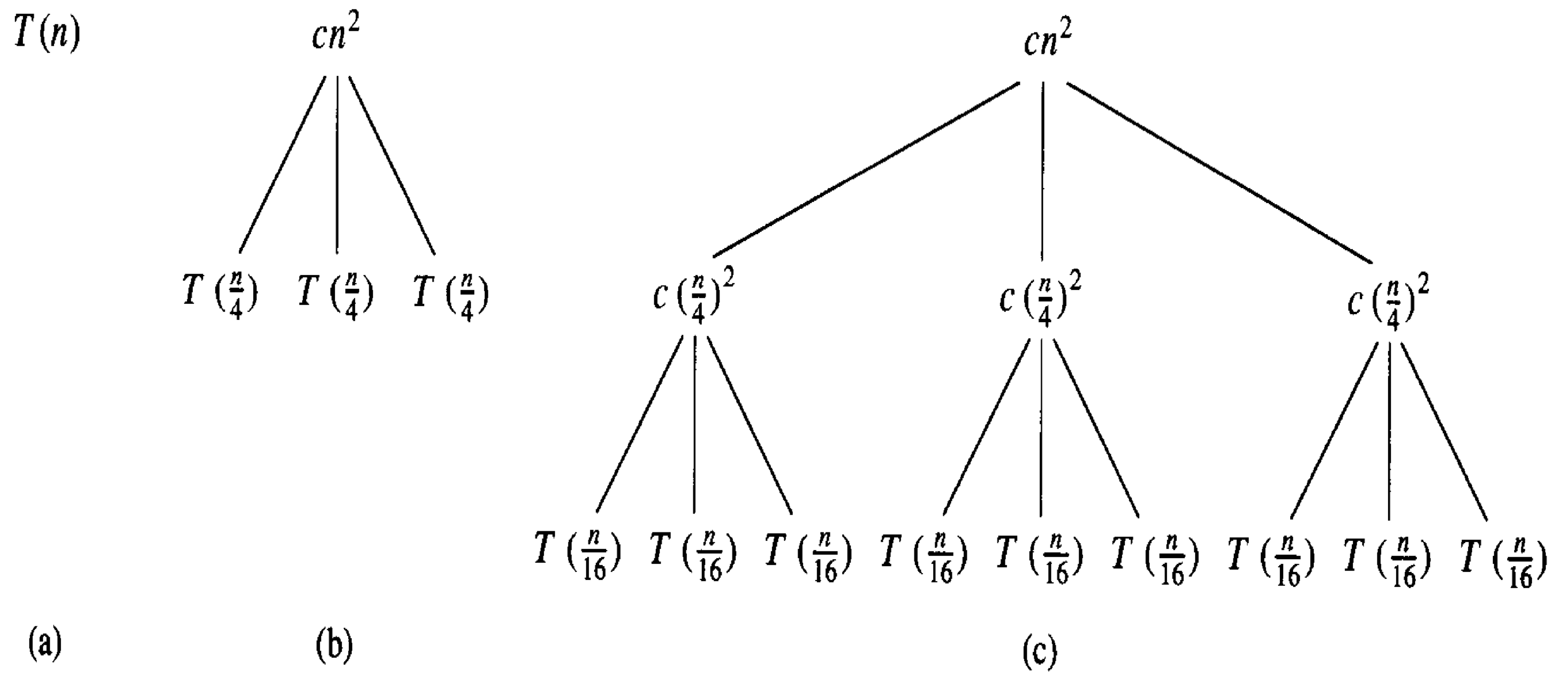
Both can be very hard to solve. We focus on relatively easy ones, which you will encounter frequently in many real algorithms (and exams…)

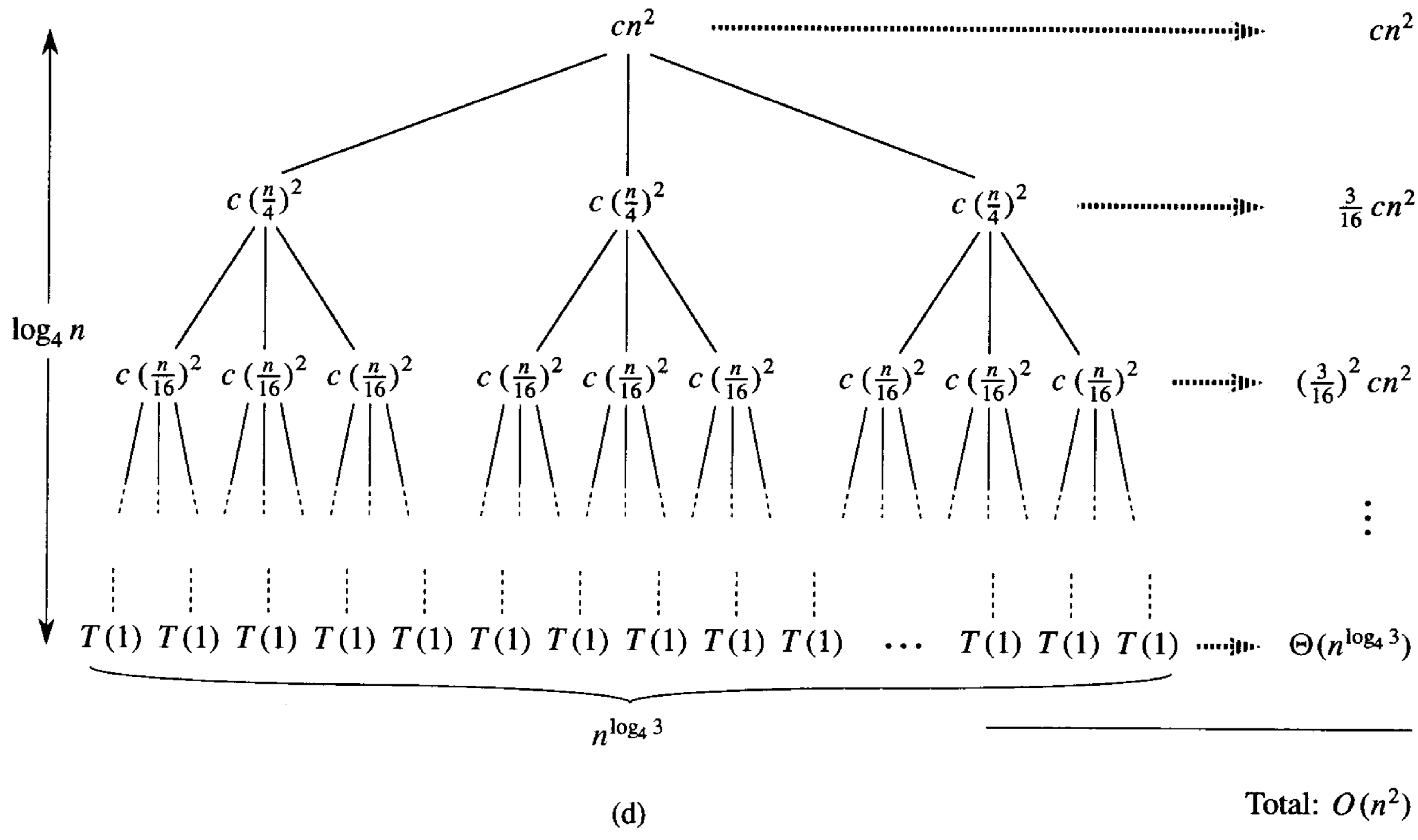
Solving recurrence

1. Recursion tree / iteration method
2. Master method
3. Math…

Recursion-tree method

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




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

(d)

The cost of the entire tree

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

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

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

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

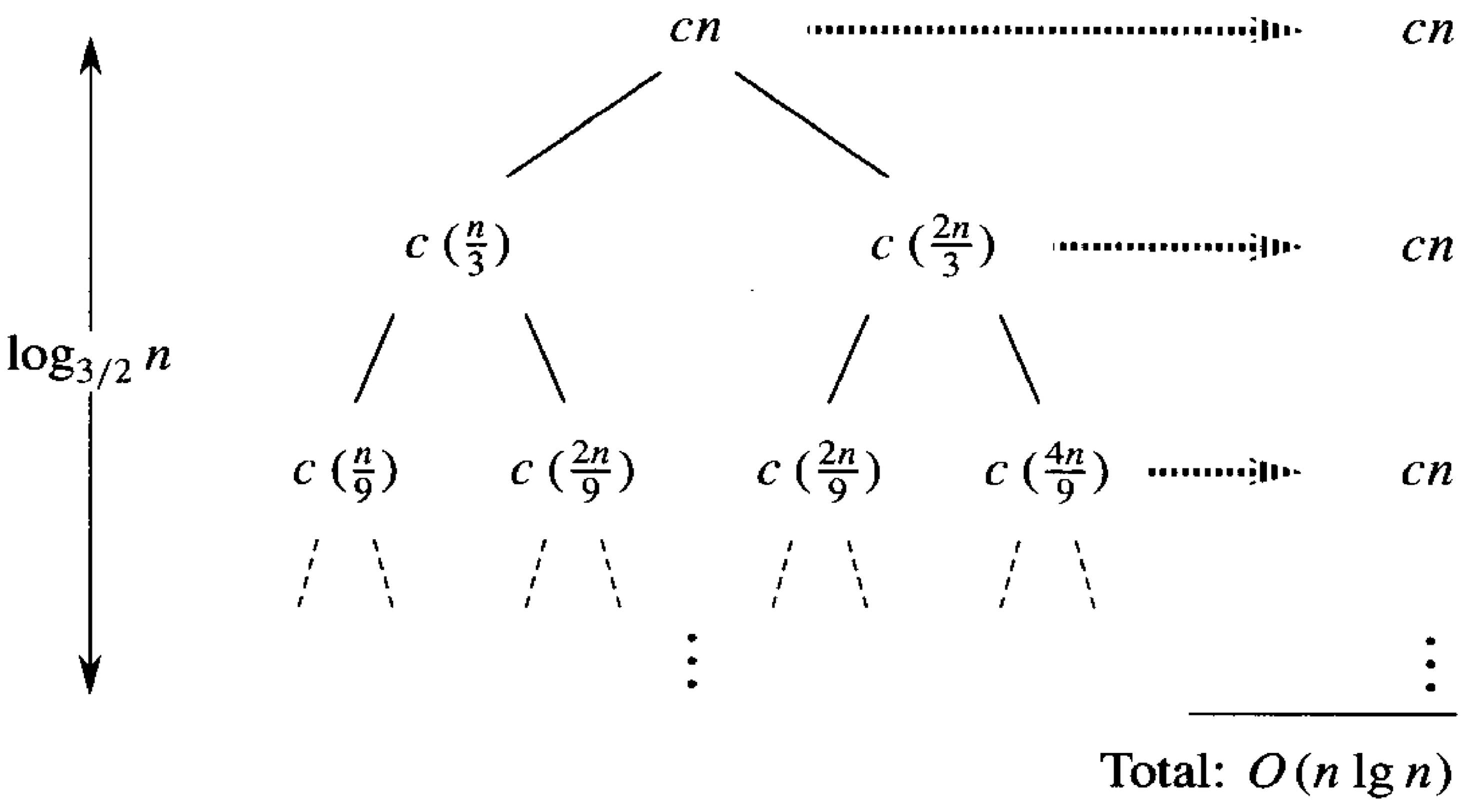
$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

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

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

$$= O(n^2)$$

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



Solving recurrence

1. Recursion tree / iteration method
   - Good for guessing an answer
2. Substitution method
   - Generic method, rigid, but may be hard
3. Master method
   - Easy to learn, useful in limited cases only
   - Some tricks may help in other cases

The master method

The master method applies to recurrences of the form

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

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

1. ***Divide*** the problem into a subproblems, **each** of size n/b
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions

   Divide + combine takes $f(n)$ time.

Master theorem

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

Key: compare $f(n)$ with $n^{\log_b a}$

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

CASE 2: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $\underline{a\,f(n/b) \le c\,f(n)}$

Regularity Condition

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

Case 1

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

Alternatively: $n^{\log_b a} / f(n) = \Omega(n^{\varepsilon})$

Intuition: $f(n)$ grows polynomially slower than $n^{\log_b a}$

Or: $n^{\log_b a}$ dominates $f(n)$ by an n^{ε} factor for some $\varepsilon > 0$

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

$T(n) = 4T(n/2) + n$
$b = 2, a = 4, f(n) = n$
$\log_2 4 = 2$
$f(n) = n = O(n^{2-\varepsilon})$, or
$n^2 / n = n^1 = \Omega(n^{\varepsilon})$, for $\varepsilon = 1$
$\therefore T(n) = \Theta(n^2)$

$T(n) = 2T(n/2) + n/\log n$
$b = 2, a = 2, f(n) = n / \log n$
$\log_2 2 = 1$
$f(n) = n/\log n \notin O(n^{1-\varepsilon})$, or
$n^1 / f(n) = \log n \notin \Omega(n^{\varepsilon})$, for any $\varepsilon > 0$
$\therefore$ CASE 1 does not apply

Case 2

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

Intuition: $f(n)$ and $n^{\log_b a}$ have the same asymptotic order.

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

e.g. $T(n) = T(n/2) + 1$ $\qquad \log_b a = 0$

$T(n) = 2\ T(n/2) + n$ $\qquad \log_b a = 1$

$T(n) = 4T(n/2) + n^2$ $\qquad \log_b a = 2$

$T(n) = 8T(n/2) + n^3$ $\qquad \log_b a = 3$

Case 3

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

Alternatively: $f(n) / n^{\log_b a} = \Omega(n^{\varepsilon})$

Intuition: $f(n)$ grows polynomially faster than $n^{\log_b a}$

Or: $f(n)$ dominates $n^{\log_b a}$ by an n^{ε} factor for some $\varepsilon > 0$

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

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

$b = 2,\ a = 1,\ f(n) = n$

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

$f(n) = n = \Omega(n^{0+\varepsilon})$, or

$n / 1 = n = \Omega(n^{\varepsilon})$

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

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

$b = 2,\ a = 1,\ f(n) = \log n$

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

$f(n) = \log n \notin \Omega(n^{0+\varepsilon})$, or

$f(n) / n^{\log_2 1} / = \log n \notin \Omega(n^{\varepsilon})$

$\therefore$ *CASE 3 does not apply*

Regularity condition

- $a f(n/b) \leq c f(n)$ for some $c < 1$ and all sufficiently large n
- This is needed for the master method to be mathematically correct.
  - to deal with some non-converging functions such as sine or cosine functions
- For most *f(n)* you'll see (e.g., polynomial, logarithm, exponential), you can safely ignore this condition, because it is implied by the first condition $f(n) = \Omega(n^{\log_b a + \varepsilon})$

Examples

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

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

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

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

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

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

Case 2: $f(n) = \Theta(n^2).$

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

Examples

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

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

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

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

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

$T(n) = 4T(n/2) + n^2/\lg n$

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

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

Examples

$T(n) = 4T(n/2) + n^{2.5}$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^{2.5}.$

Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 0.5$

and $4(n/2)^{2.5} \le cn^{2.5}$ (reg. cond.) for $c = 0.75$.

$\therefore\ T(n) = \Theta(n^{2.5})$.

$T(n) = 4T(n/2) + n^2 \log n$

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

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\log n)$.

How do I know which case to use? Do I need to try all three cases one by one?

- Compare $f(n)$ with $n^{\log_b a}$

check if $n^{\log_b a} / f(n) \in \Omega(n^{\varepsilon})$

- $f(n) \in \begin{cases} o(n^{\log_b a}) & \text{Possible CASE 1} \\ \Theta(n^{\log_b a}) & \text{CASE 2} \\ \omega(n^{\log_b a}) & \text{Possible CASE 3} \end{cases}$

check if $f(n) / n^{\log_b a} \in \Omega(n^{\varepsilon})$

Examples

a. $T(n) = 4T(n/2) + n;$ $\log_b a$ = 2. n = $o(n^2)$ => Check case 1

b. $T(n) = 9T(n/3) + n^2;$ $\log_b a$ = 2. $n^2 = o(n^2)$ => case 2

c. $T(n) = 6T(n/4) + n;$ $\log_b a$ = 1.3. n = $o(n^{1.3})$ => Check case 1

d. $T(n) = 2T(n/4) + n;$ $\log_b a$ = 0.5. n = $\omega(n^{0.5})$ => Check case 3

e. $T(n) = T(n/2) + n \log n;$ $\log_b a$ = 0. nlogn = $\omega(n^0)$ => Check case 3

f. $T(n) = 4T(n/4) + n \log n.$ $\log_b a$ = 1. nlogn = $\omega(n)$ => Check case 3

More examples

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

$$T(n) = 0.5T(n/2) + n\log n$$

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

$$T(n) = T(n/2) + n(2 - \cos n)$$

Some tricks

- Changing variables

- Obtaining upper and lower bounds
  - Make a guess based on the bounds
  - Prove using the substitution method

Changing variables

$$T(n) = 2T(n-1) + 1$$

- Let $n = \log m$, i.e., $m = 2^n$

=> $T(\log m) = 2\, T(\log (m/2)) + 1$

- Let $S(m) = T(\log m) = T(n)$

=> $S(m) = 2S(m/2) + 1$

=> $S(m) = \Theta(m)$

=> $T(n) = S(m) = \Theta(m) = \Theta(2^n)$

Changing variables

$$T(n) = T(\sqrt{n}) + 1$$

- Let n $=2^m$

=> sqrt(n) = $2^{m/2}$

- We then have $T(2^m) = T(2^{m/2}) + 1$
- Let $T(n) = T(2^m) = S(m)$

=> $S(m) = S(m/2) + 1$

$\Rightarrow S(m) = \Theta(\log m) = \Theta(\log \log n)$

$\Rightarrow T(n) = \Theta(\log \log n)$

Changing variables

- $T(n) = 2T(n-2) + 1$
- Let $n = \log m$, i.e., $m = 2^n$

$\Rightarrow T(\log m) = 2\, T(\log m/4) + 1$

- Let $S(m) = T(\log m) = T(n)$

$\Rightarrow S(m) = 2S(m/4) + 1$

$\Rightarrow S(m) = m^{1/2}$

$\Rightarrow T(n) = S(m) = (2^n)^{1/2} = (\mathrm{sqrt}(2))^n \approx 1.4^n$

Obtaining bounds

Solve the Fibonacci sequence:

$$T(n) = T(n-1) + T(n-2) + 1$$

- $T(n) >= 2T(n-2) + 1$ [1]
- $T(n) <= 2T(n-1) + 1$ [2]

- Solving [1], we obtain $T(n) >= 1.4^n$
- Solving [2], we obtain $T(n) <= 2^n$
- Actually, $T(n) \approx 1.62^n$

Obtaining bounds

- $T(n) = T(n/2) + \log n$
- $T(n) \in \Omega(\log n)$
- $T(n) \in O(T(n/2) + n^{\varepsilon})$
- Solving $T(n) = T(n/2) + n^{\varepsilon}$,

  we obtain $T(n) = O(n^{\varepsilon})$, for any $\varepsilon > 0$
- So: $T(n) \in O(n^{\varepsilon})$ for any $\varepsilon > 0$
  - $T(n)$ is unlikely polynomial
  - Actually, $T(n) = \Theta(\log^2 n)$ by extended case 2

Extended Case 2

CASE 2: $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.

Extended CASE 2: (k >= 0)

$f(n) = \Theta(n^{\log_b a} \, log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.