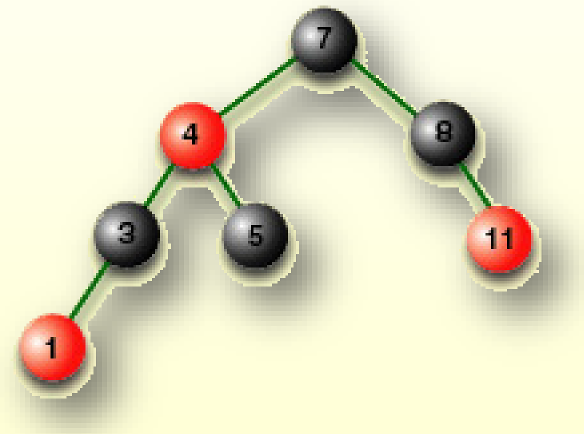
CS161: Design and Analysis of Algorithms



Lecture 2

Leonidas Guibas

Outline

- Review of last lecture
- Order of growth of functions
- Asymptotic notations
  - Big O, big Ω, Θ, etc
- Recurrences

Slides modified from
- http://www.cs.virginia.edu/~luebke/cs332/

Correctness of Algorithms

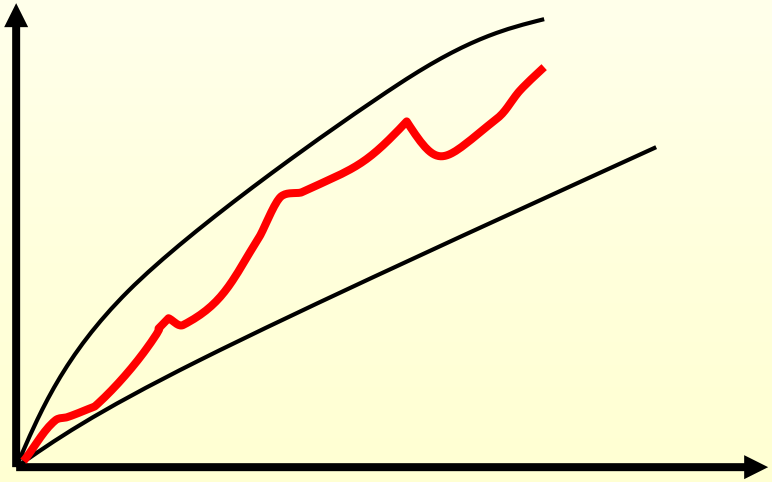
- For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem.
- For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Algorithm correctness is NOT obvious in many cases (e.g., optimization)

Efficiency of Algorithms

- Correctness alone is not sufficient
- Brute-force algorithms exist for most problems
- To sort n numbers, we can enumerate all permutations of these numbers and test which permutation has the correct order
  - Why cannot we do this?
  - Too slow!
  - By what standard?

Exact Algorithm Analysis is Hard

- Worst-case and average-case are difficult to analyze precisely -- the details can be very complicated



Easier to talk about upper and lower bounds on the function T(n), the count of the number of operations the algorithm performs.

Kinds of Analyses

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Best case – not very useful
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs
    - Real-life inputs

Analysis of Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j]
      i = j - 1;
      while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
      }
      A[i+1] = key
  }
}
```

How many times will this line execute?

Analysis of Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j]
      i = j - 1;
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
  }
}
```

How many times will this line execute?

Analysis of Insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`    key = A[j]`	c_2	(n-1)
`    i = j - 1;`	c_3	(n-1)
`    while (i > 0) and (A[i] > key) {`	c_4	S
`      A[i+1] = A[i]`	c_5	(S-(n-1))
`      i = i - 1`	c_6	(S-(n-1))
`    }`	0	
`    A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

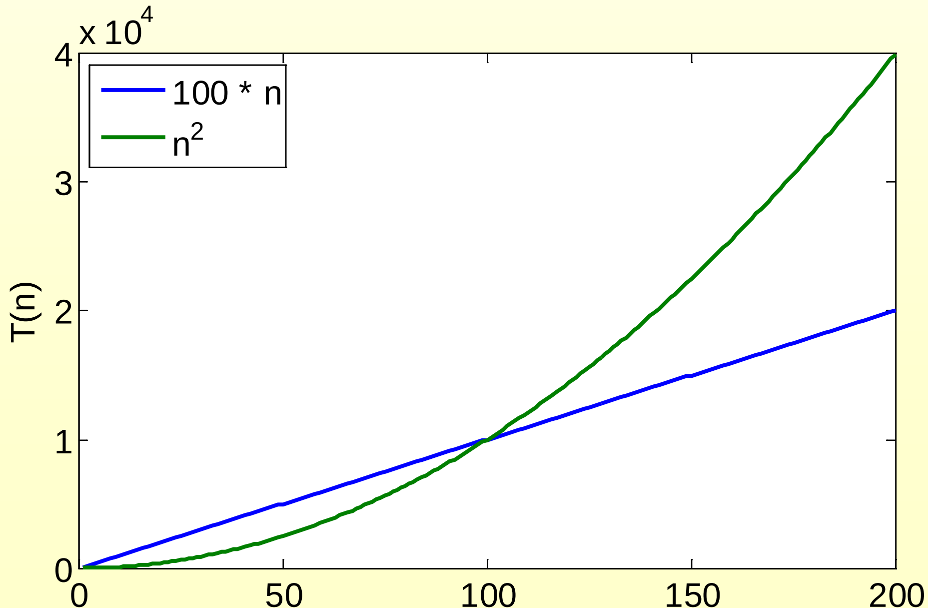
$S = t_2 + t_3 + \ldots + t_n$ where t_j is number of while expression evaluations for the j^{th} for loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 S + c_5(S - (n-1)) + c_6(S - (n-1)) + c_7(n-1)$
  $= c_8 S + c_9 n + c_{10}$
- What can S be?
  - Best case -- inner loop body never executed
    - $t_j = 1 \Rightarrow S = n - 1$
    - $T(n) = an + b$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_j = j \Rightarrow S = 2 + 3 + \dots + n = n(n+1)/2 - 1$
    - $T(n) = an^2 + bn + c$ is a quadratic function
  - Average case
    - Can assume that on average, we have to insert A[j] into the middle of A[1..j-1], so $t_j = j/2$
    - $S \approx n(n+1)/4$
    - T(n) is still a quadratic function

Asymptotic Analysis

- Abstract statement costs (don't care about c_1, c_2, etc)
- *Order of growth* (as a function of n, the input size) is the interesting measure:
  - Highest-order term is what counts
    - As the input size grows larger it is the high order term that dominates



Comparison of functions

	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	10^2	10^3	10^3	10^6
10^2	6.6	10^2	660	10^4	10^6	10^{30}	10^{158}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}	10^{15}		
10^6	20	10^6	10^7	10^{12}	10^{18}		

For a super computer that does 1 trillion operations per second, it will be longer than 1 billion years

Order of Growth

$1 \ll \log_2 n \ll n \ll n\log_2 n \ll n^2 \ll n^3 \ll 2^n \ll n!$

(We are slightly abusing of the “<<“ sign. It means a smaller order of growth).

Asymptotic Notations

- We say InsertionSort's worst-case running time is $\Theta(n^2)$
  - Properly we should say running time is *in* $\Theta(n^2)$
  - It is also in $O(n^2)$
  - What's the relationships between Θ and O?
- Formal definition comes next

Asymptotic Notations

- O: Big-Oh
- Ω: Big-Omega
- Θ: Theta
- o: Small-oh
- ω: Small-omega

Big "O"

- Informally, $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$, within a constant multiple
- If we say $f(n)$ is in $O(g(n))$, it means that $g(n)$ is an asymptotic upper bound on $f(n)$
  - Formally:
    - $\exists$ C (>0) & n_0, $f(n) \leq Cg(n)$ for $\forall$ $n >= n_0$
- What is $O(n^2)$?
  - The set of all functions that grow slower than or at the same order as n^2

Big "O"

So:

$n \in O(n^2)$

$n^2 \in O(n^2)$

$1000n \in O(n^2)$

$n^2 + n \in O(n^2)$

$100n^2 + n \in O(n^2)$

But:

$1/1000\ n^3 \notin O(n^2)$

O is an upper bound notation, like ≤

We ignore constants, lower order terms – get to the essential growth

Even though formally we should write $n \in O(n^2)$, in practice we write $n = O(n^2)$

Small "o"

- Informally, o(g(n)) is the set of all functions with a strictly smaller growth as g(n), within a constant factor
- What is $o(n^2)$?
  - The set of all functions that grow slower than n^2

So:

$1000n \in o(n^2)$

But:

$n^2 \notin o(n^2)$

o is a strict upper bound notation, like <

Formally,

$$f(n) \in o(g(n))$$

$$\frac{f(n)}{g(n)} \rightarrow 0 \; as \; n \rightarrow \infty$$

Big "Ω" [Omega]

- Informally, $\Omega(g(n))$ is the set of all functions with a larger or same order of growth as g(n), within a constant multiple
- $f(n) \in \Omega(g(n))$ means g(n) is an asymptotic lower bound of f(n)
  - Intuitively, it is like $f(n) \geq g(n)$

Intuitively, Ω is like ≥

a lower bound notation

So:

$n^2 \in \Omega(n)$

$1/1000\ n^2 \in \Omega(n)$

But:

$1000\ n \notin \Omega(n^2)$

Small "ω" [omega]

- Informally, $\omega(g(n))$ is the set of all functions with a strictly larger order of growth than $g(n)$, within a constant factor

So:

$n^2 \in \omega(n)$

$1/1000\ n^2 \in \omega(n)$

$n^2 \notin \omega(n^2)$

Intuitively, ω is like >

a strict lower bound

Theta ("Θ"): Θ = O and Ω

- Informally, $\Theta(g(n))$ is the set of all functions with the same order of growth as g(n), within a constant multiple
- $f(n) \in \Theta(g(n))$ means g(n) is an asymptotically tight bound on f(n)
  - Intuitively, it is like f(n) = g(n)
- What is $\Theta(n^2)$?
  - The set of all functions that grow at the same order as n^2

Big "Θ" [Theta]

So:

$n^2 \in \Theta(n^2)$

$n^2 + n \in \Theta(n^2)$

$100n^2 + n \in \Theta(n^2)$

$100n^2 + \log_2 n \in \Theta(n^2)$

But:

$n\log_2 n \notin \Theta(n^2)$

$1000n \notin \Theta(n^2)$

$1/1000\ n^3 \notin \Theta(n^2)$

Intuitively, Θ is like =

Tricky Cases

- How about sqrt(n) and $\log_2 n$?
- How about $\log_2 n$ and $\log_{10} n$
- How about 2^n and 3^n
- How about 3^n and n!?

Big "O", Formally

- Definition:

  There exist / For all

  $O(g(n)) = \{f(n): \exists$ positive constants C and n_0 such that $0 \le f(n) \le Cg(n)\ \forall\ n > n_0\}$

- $\lim_{n\to\infty} g(n)/f(n) > 0$ (if the limit exists)

- Abuse of notation (for convenience):

  $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

Big "O", Example

- Claim: $f(n) = 3n^2 + 10n + 5 \in O(n^2)$
- Proof from the definition

  To prove this claim by definition, we need to find some positive constants C and n_0 such that $f(n) <= Cn^2$ for all $n > n_0$.

  (Note: you just need to find one concrete example of c and n_0 satisfying the condition.)

  $3n^2 + 10n + 5 \leq 10n^2 + 10n + 10$

  $\leq 10n^2 + 10n^2 + 10n^2, \forall\, n \geq 1$

  $\leq 30\, n^2, \forall\, n \geq 1$

  Therefore, if we let $C = 30$ and $n_0 = 1$, we have $f(n) \leq C\, n^2, \forall\, n \geq n_0$.

  Hence according to the definition of big-Oh, $f(n) = O(n^2)$.
- Alternatively, we can show that

  $\lim_{n\to\infty} n^2 / (3n^2 + 10n + 5) = 1/3 > 0$

Big "Ω", Formally

- Definition:

  $\Omega(g(n)) = \{f(n)\!:\ \exists$ positive constants C and n_0 such that $0 \le Cg(n) \le f(n)\ \forall\ n > n_0\}$

- $\lim_{n\to\infty} f(n)/g(n) > 0$ (if the limit exists.)
- Abuse of notation (for convenience):

  $f(n) = \Omega(g(n))$ actually means $f(n) \in \Omega(g(n))$

Big "Ω", Example

- Claim: $f(n) = n^2 / 10 = \Omega(n)$
- Proof from the definition:

  $f(n) = n^2 / 10$, $g(n) = n$

  Need to find a C and a n_0 to satisfy the definition of $f(n) \in \Omega(g(n))$, i.e., $f(n) \geq Cg(n)$ for $n > n_0$

  $n \leq n^2 / 10$ when $n \geq 10$

  If we let $C = 1$ and $n_0 = 10$, we have $f(n) \geq Cn$, $\forall\ n \geq n_0$. Therefore, according to the definition, $f(n) = \Omega(n)$.
- Alternatively:

  $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} (n/10) = \infty$

Big "Θ", Formally

- Definition:
  - $\Theta(g(n)) = \{f(n)$: $\exists$ positive constants c_1, c_2, and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$, $\forall\, n \geq n_0\}$
- $\lim_{n\to\infty} f(n)/g(n) = c > 0$ and $c < \infty$
- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- Abuse of notation (for convenience):

  $f(n) = \Theta(g(n))$ actually means $f(n) \in \Theta(g(n))$

  $\Theta(1)$ means constant time.

Big "Θ", Example

- Claim: $f(n) = 2n^2 + n = \Theta(n^2)$
- Proof from the definition:
  - Need to find the three constants c_1, c_2, and n_0 such that
    $c_1 n^2 \le 2n^2 + n \le c_2 n^2$ for all $n > n_0$
  - A simple solution is $c_1 = 2$, $c_2 = 3$, and $n_0 = 1$
- Alternatively, $\lim_{n\to\infty}(2n^2+n)/n^2 = 2$

More Examples

- Prove $n^2 + 3n + \lg n$ is in $O(n^2)$
- Want to find c and n_0 such that

  $n^2 + 3n + \lg n <= cn^2$ for $n > n_0$
- Proof:

$$n^2 + 3n + \lg n <= 3n^2 + 3n + 3\lg n \qquad \text{for } n > 1$$
$$<= 3n^2 + 3n^2 + 3n^2$$
$$<= 9n^2$$

Or $n^2 + 3n + \lg n <= n^2 + n^2 + n^2$ for $n > 10$

$$<= 3n^2$$

More Examples

- Prove $n^2 + 3n + \lg n$ is in $\Omega(n^2)$
- Want to find c and n_0 such that

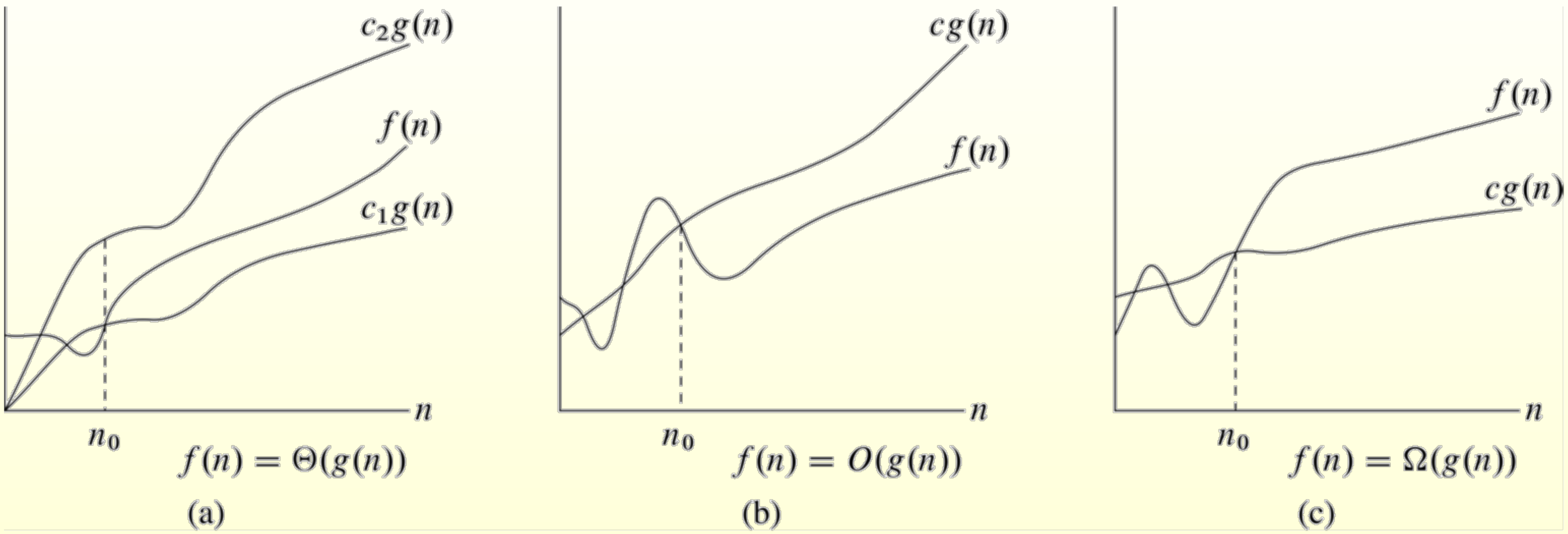
$$n^2 + 3n + \lg n >= cn^2 \text{ for } n > n_0$$

$$n^2 + 3n + \lg n >= n^2 \text{ for } n > 1$$

$n^2 + 3n + \lg n = O(n^2)$ and $n^2 + 3n + \lg n = \Omega\,(n^2)$

$=> n^2 + 3n + \lg n = \Theta(n^2)$

O, Ω, and Θ



The definitions imply a constant n_0 *beyond which* they are satisfied. We do not care about small values of n.

Using Limits to Compare Orders of Growth

- $$\lim_{n\to\infty} f(n) / g(n) = \begin{cases} 0 \\ c \quad > 0 \\ \infty \end{cases}$$

$0 \rightarrow f(n) \in o(g(n)) \rightarrow f(n) \in O(g(n))$

$c > 0 \rightarrow f(n) \in \Theta(g(n)) \rightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

$\infty \rightarrow f(n) \in \omega(g(n)) \rightarrow f(n) \in \Omega(g(n))$

Logarithms

- compare $\log_2 n$ and $\log_{10} n$

- $\log_a b = \log_c b \,/\, \log_c a$
- $\log_2 n = \log_{10} n \,/\, \log_{10} 2 \sim 3.3 \log_{10} n$
- Therefore $\lim(\log_2 n \,/\, \log_{10} n) = 3.3$
- $\log_2 n = \Theta\ (\log_{10} n)$

Exponentials

- Compare 2^n and 3^n
- $\lim_{n\to\infty} 2^n / 3^n = \lim_{n\to\infty}(2/3)^n = 0$
- Therefore, $2^n \in o(3^n)$, and $3^n \in \omega(2^n)$

- How about 2^n and 2^{n+1}?

$2^n / 2^{n+1} = ½$, therefore $2^n = \Theta(2^{n+1})$

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) are ∞ or 0

- You can apply this transformation as many times as you want, as long as the condition holds

- Compare $n^{0.5}$ and $\log n$
- $\lim_{n \to \infty} n^{0.5} / \ln n = ?$
- $(n^{0.5})' = 0.5 \, 1/n^{0.5}$
- $(\ln n)' = 1 / n$
- $\lim (1/n^{0.5} / 1/n) = \lim (n^{0.5}) = \infty$
- Therefore, $\ln n \in o(n^{0.5})$
- In fact, $\ln n \in o(n^{\varepsilon})$, for any $\varepsilon > 0$ – and so is $\log n$

Stirling's Formula (Useful)

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

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

- Compare 2^n and n! $\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{c\sqrt{n}n^n}{2^n e^n}=\lim_{n\to\infty}c\sqrt{n}\left(\frac{n}{2e}\right)^n=\infty$
- Therefore, $2^n = o(n!)$
- Compare n^n and n! $\lim_{n\to\infty}\frac{n!}{n^n}=\lim_{n\to\infty}\frac{c\sqrt{n}n^n}{n^n e^n}=\lim_{n\to\infty}\frac{c\sqrt{n}}{e^n}=0$
- Therefore, $n^n = \omega(n!)$
- How about log (n!)?

$$\log(n!) = \log \frac{c\sqrt{n}n^n}{e^n} = C + \log n^{n+1/2} - \log(e^n)$$

$$= C + n\log n + \frac{1}{2}\log n - n$$

$$= C + \frac{n}{2}\log n + (\frac{n}{2}\log n - n) + \frac{1}{2}\log n$$

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

More Advanced Dominance Rankings

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n / \log \log n \gg \log \log n \gg \alpha(n) \gg 1$$

Asymptotic Notation Summary

- O: Big-Oh
- Ω: Big-Omega
- Θ: Theta
- o: Small-oh
- ω: Small-omega
- Intuitively:

O is like $\leq$	Ω is like $\geq$	Θ is like =
o is like <	ω is like >	

Properties of Asymptotic Notations

- CLRS 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 $g(n) = \omega(f(n))$

Exponential and Logarithmic Functions

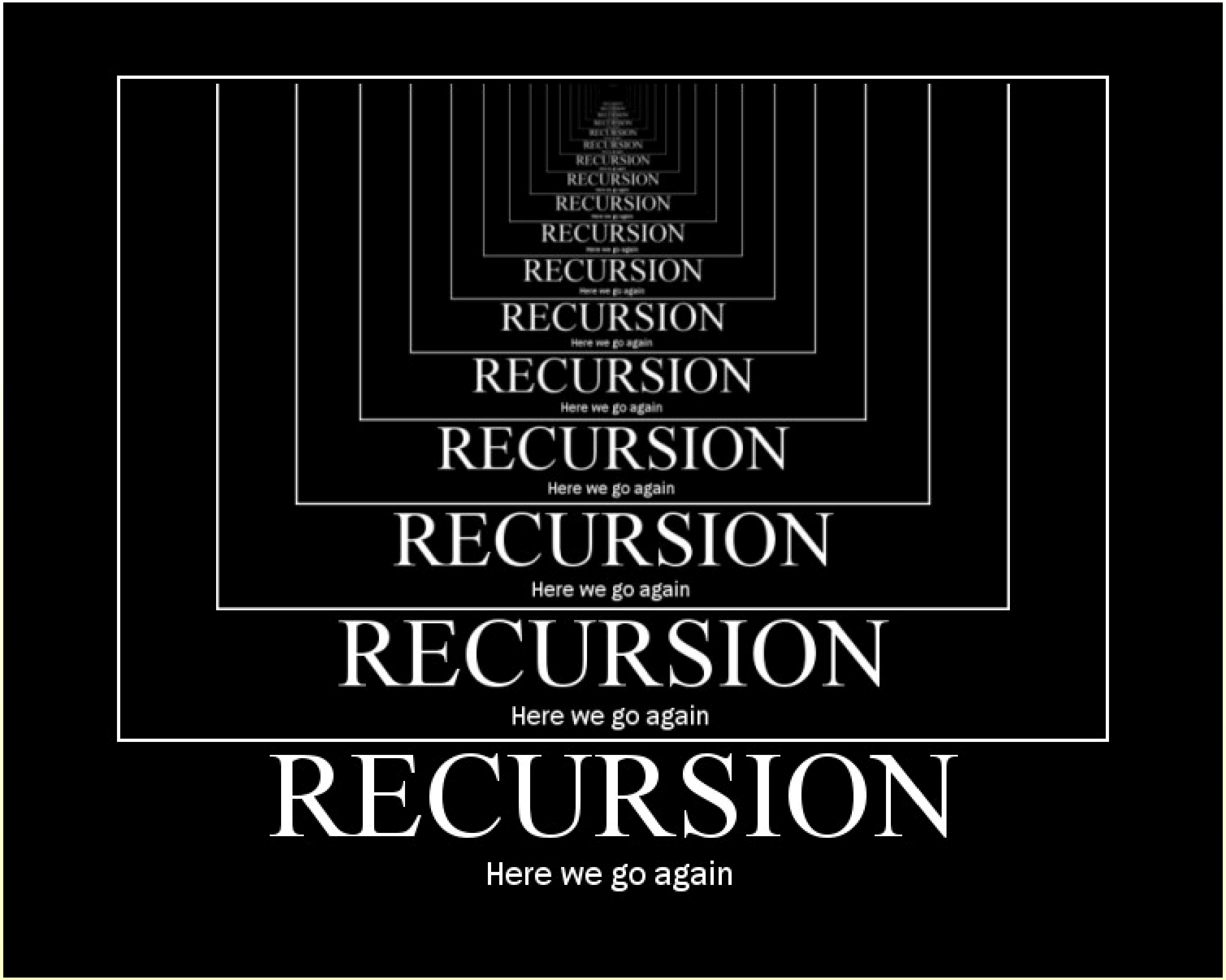
- CLRS textbook, pages 55-56
- It is important to understand what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function.
- Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.
- Logarithms reflect how many times we can double something until we get to n, or halve something until we get to 1.
- $\log_2 1 = ?$
- $\log_2 2 = ?$

Useful Rules for Logarithms

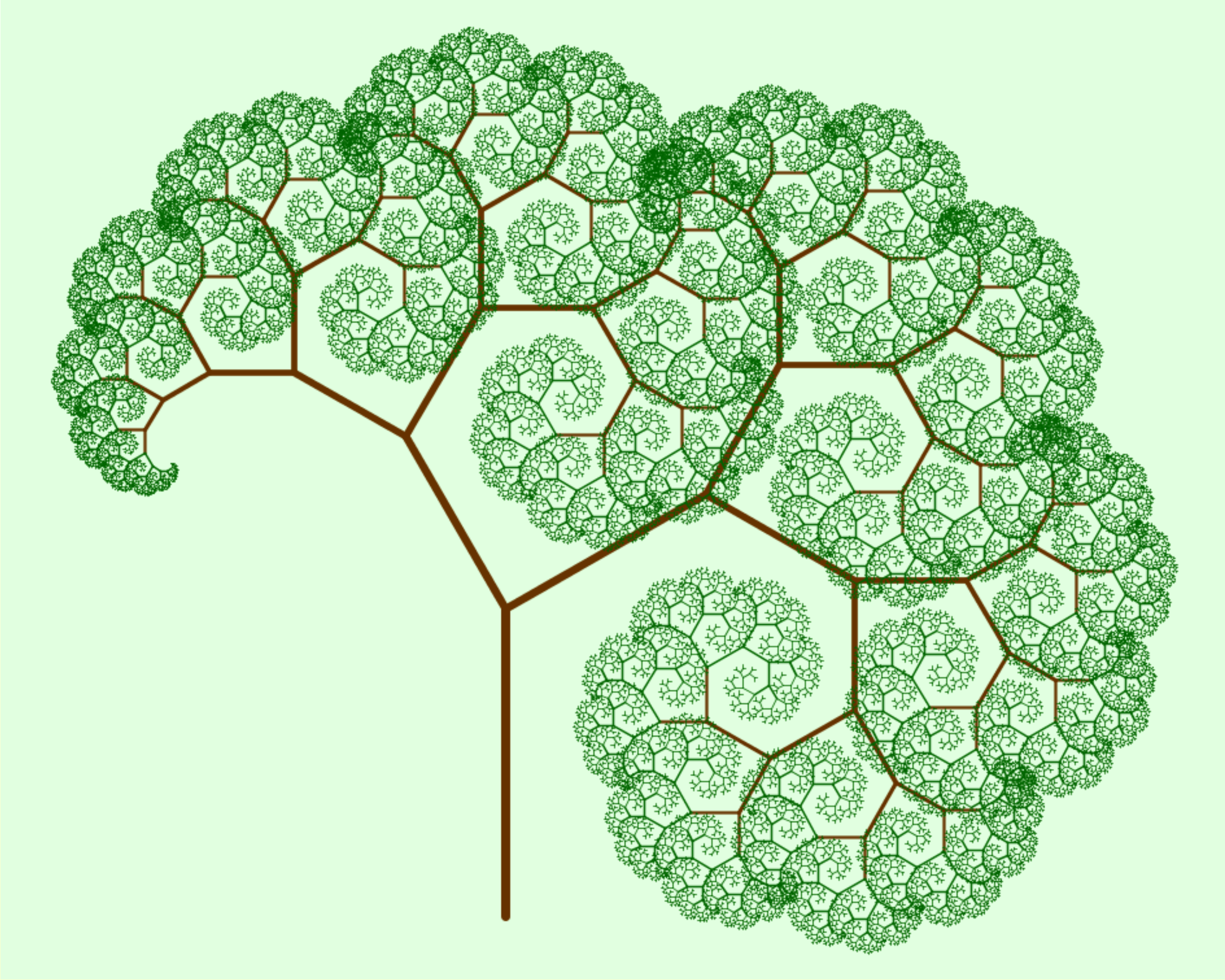
- For all $a > 0$, $b > 0$, $c > 0$, the following rules hold
- $\log_b a = \log_c a / \log_c b = \lg a / \lg b$
- $\log_b a^n = n \log_b a$
- $b^{\log_b a} = a$
- $\log (ab) = \log a + \log b$
  - $\lg (2n) = ?$
- $\log (a/b) = \log (a) – \log(b)$
  - $\lg (n/2) = ?$
  - $\lg (1/n) = ?$
- $\log_b a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

Useful Rules for Exponentials

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


RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION

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

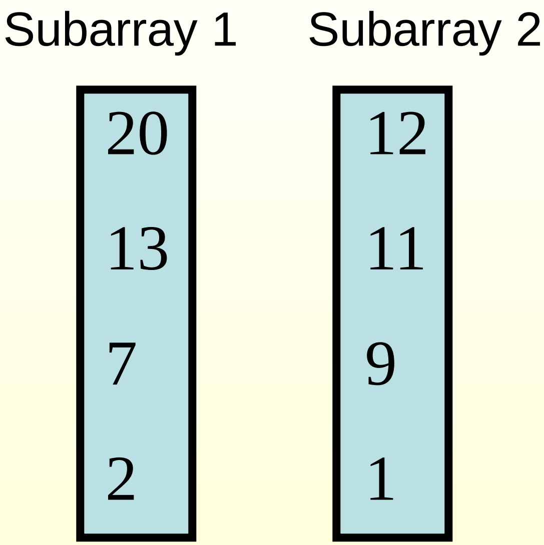
MergeSort

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

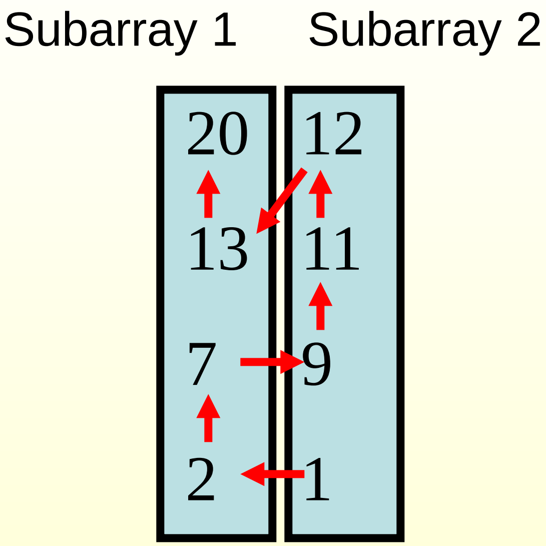
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
3. "***Merge***" the 2 sorted lists.

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

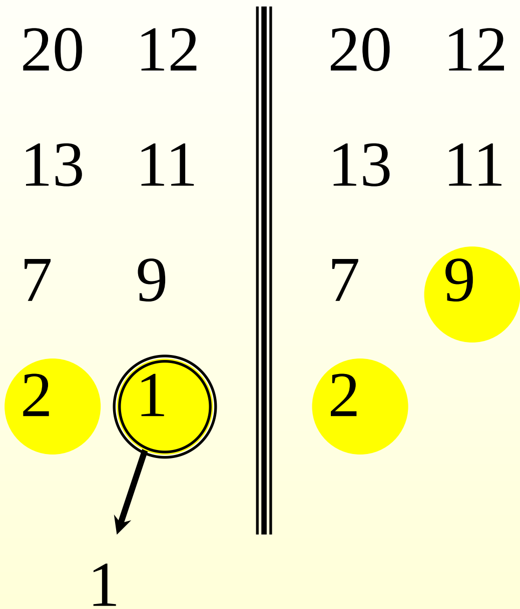
20	12
13	11
7	9
2	1

Merging Two Sorted Arrays

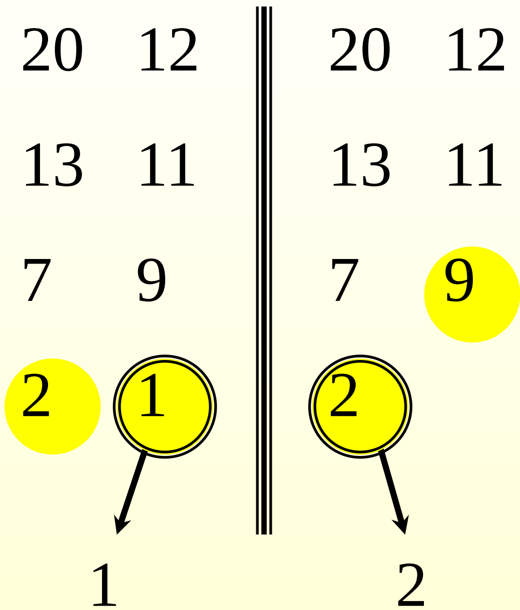
20	12
13	11
7	9
2	1

1

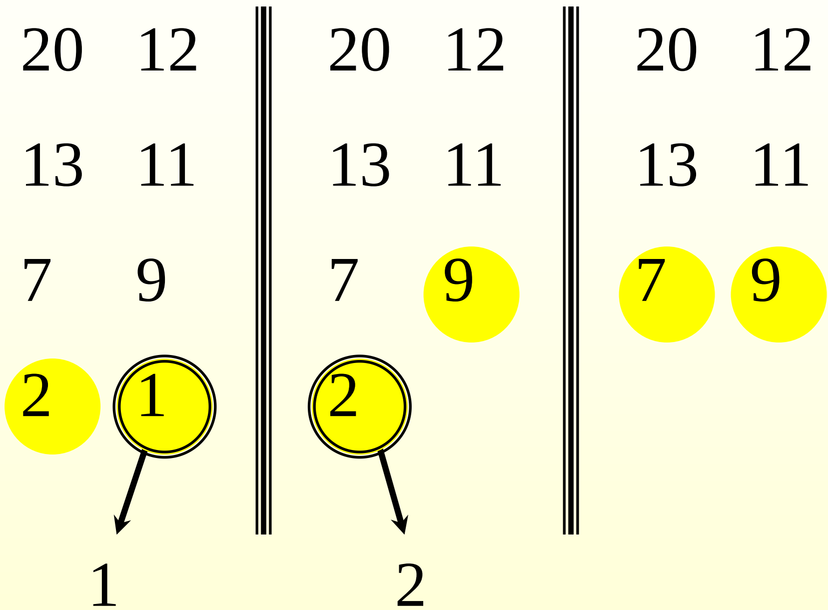
Merging Two Sorted Arrays



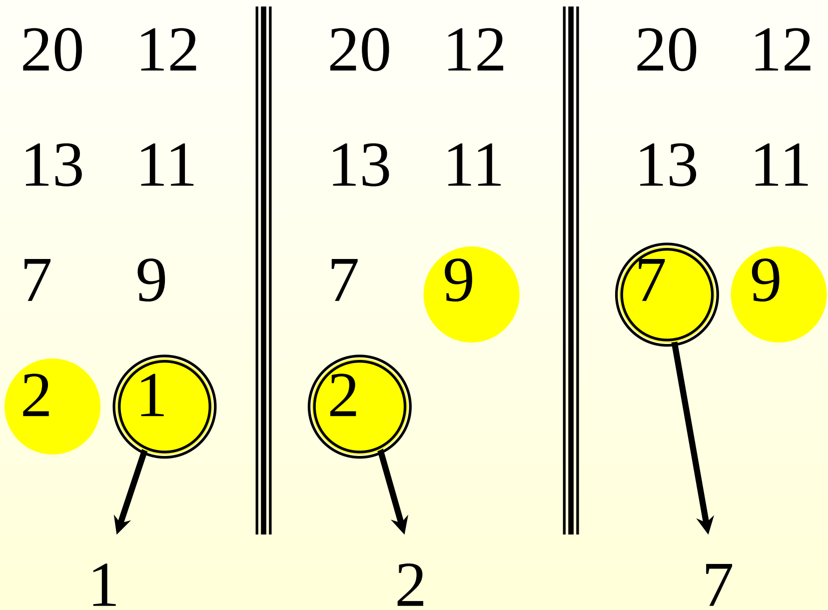
Merging Two Sorted Arrays



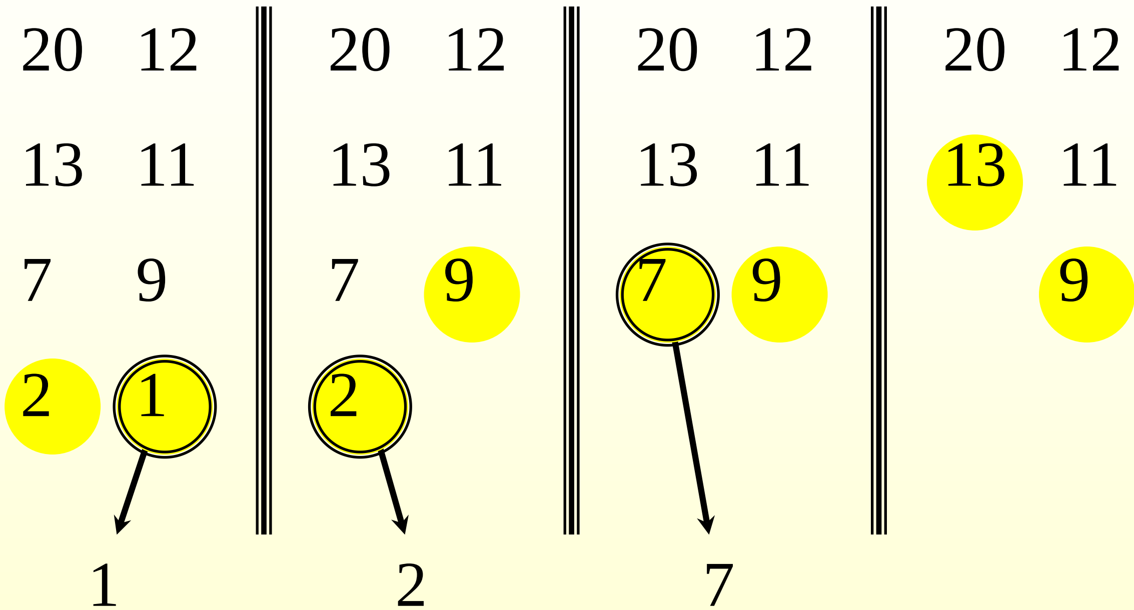
Merging Two Sorted Arrays



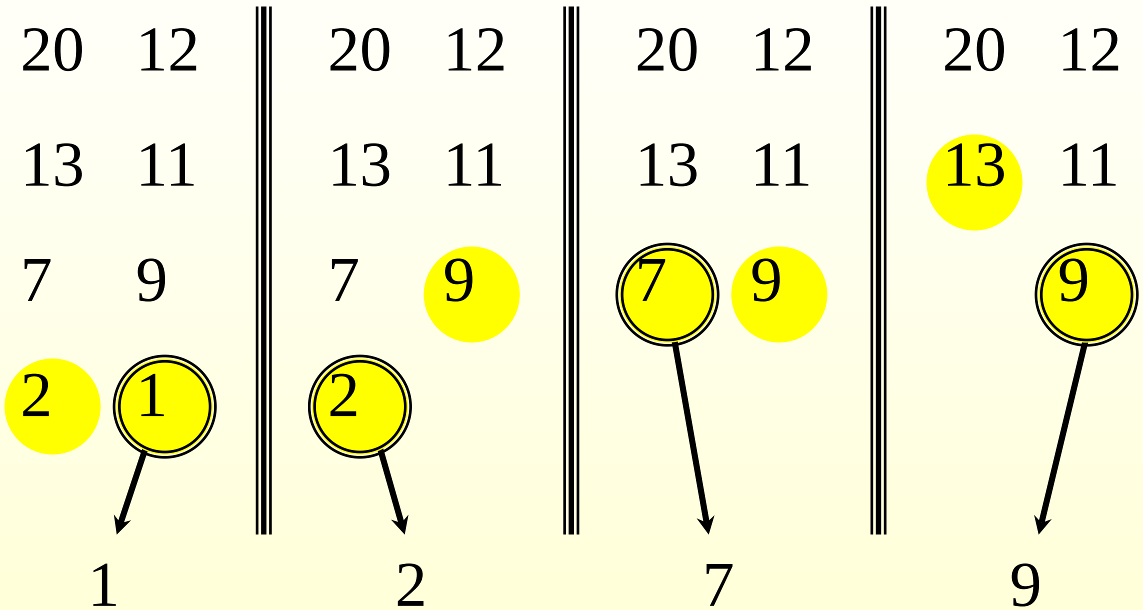
Merging Two Sorted Arrays



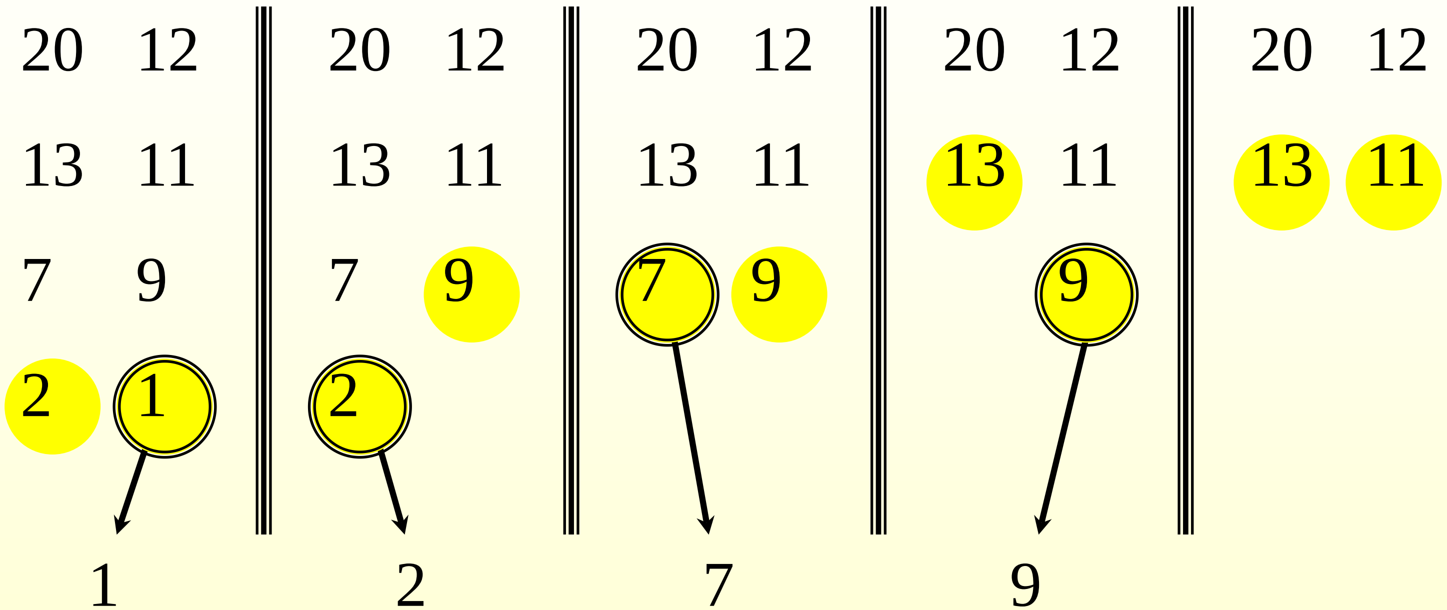
Merging Two Sorted Arrays



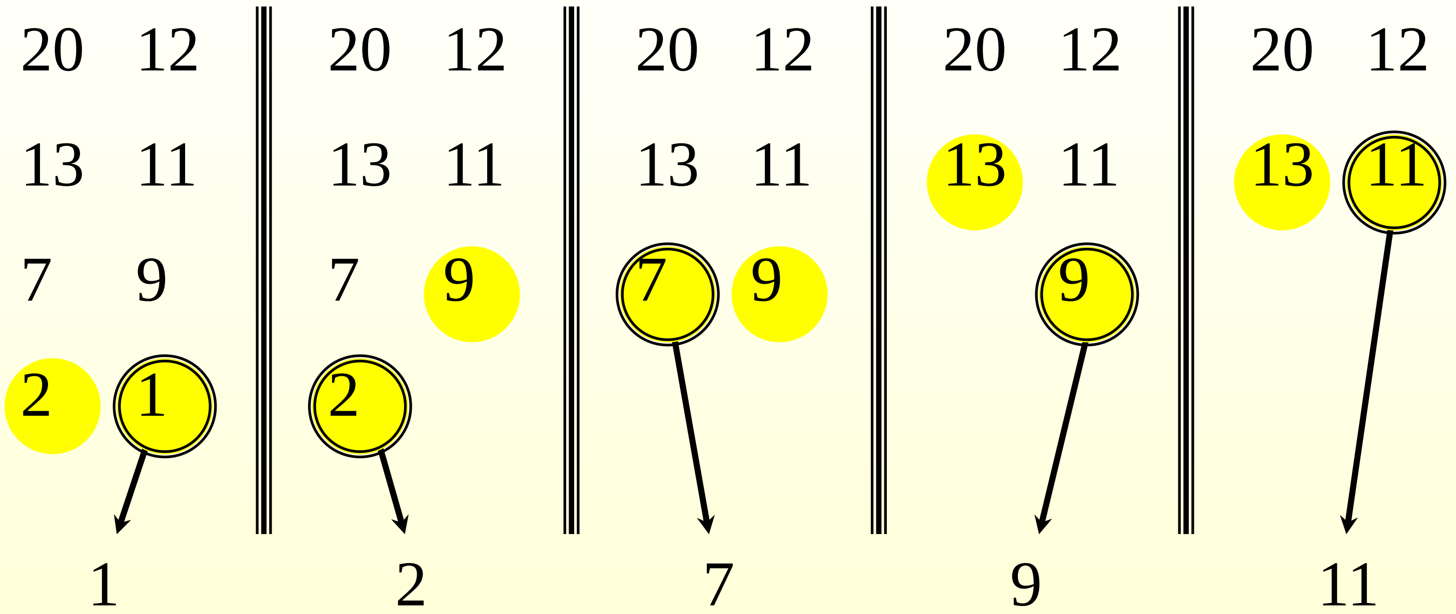
Merging Two Sorted Arrays



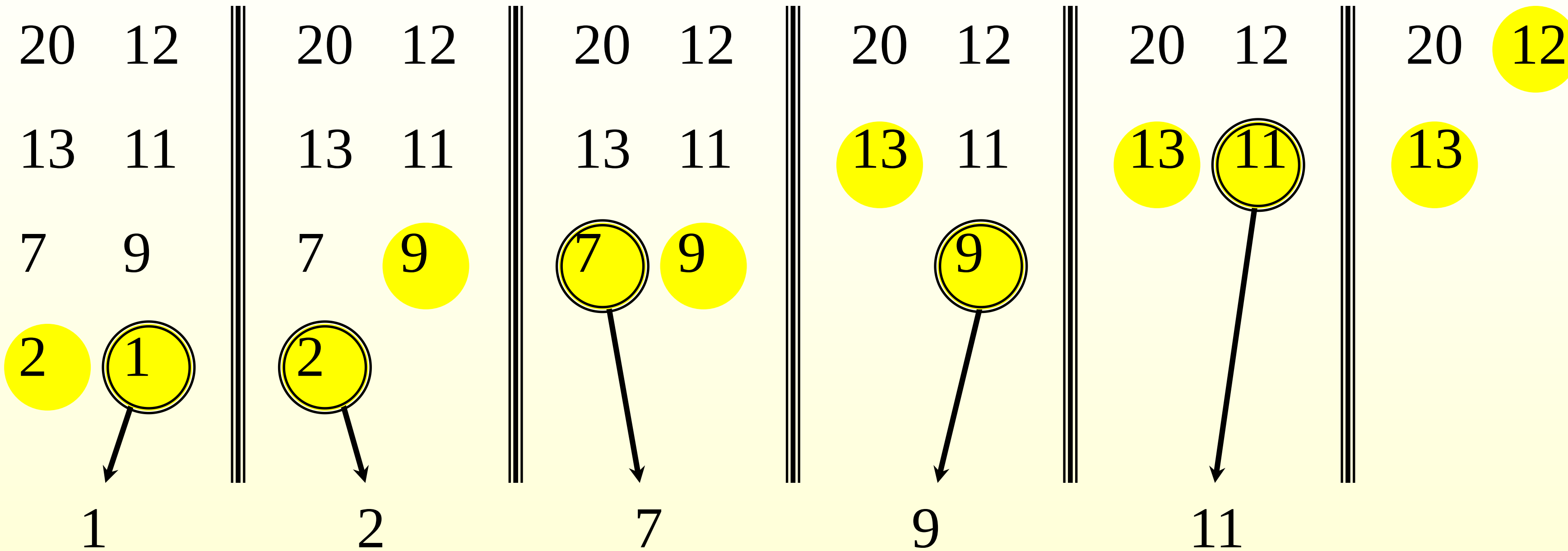
Merging Two Sorted Arrays



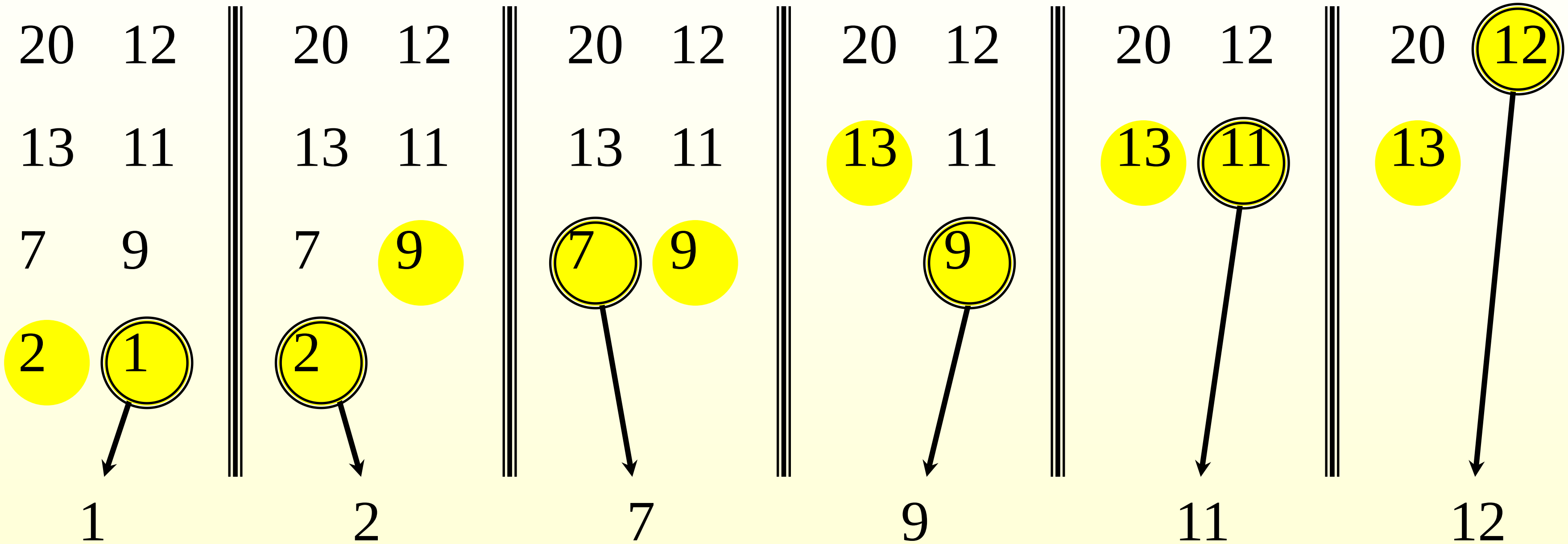
Merging Two Sorted Arrays



Merging Two Sorted Arrays



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 MergeSort

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.

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 MergeSort

$T(n)$	**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 MergeSort

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

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

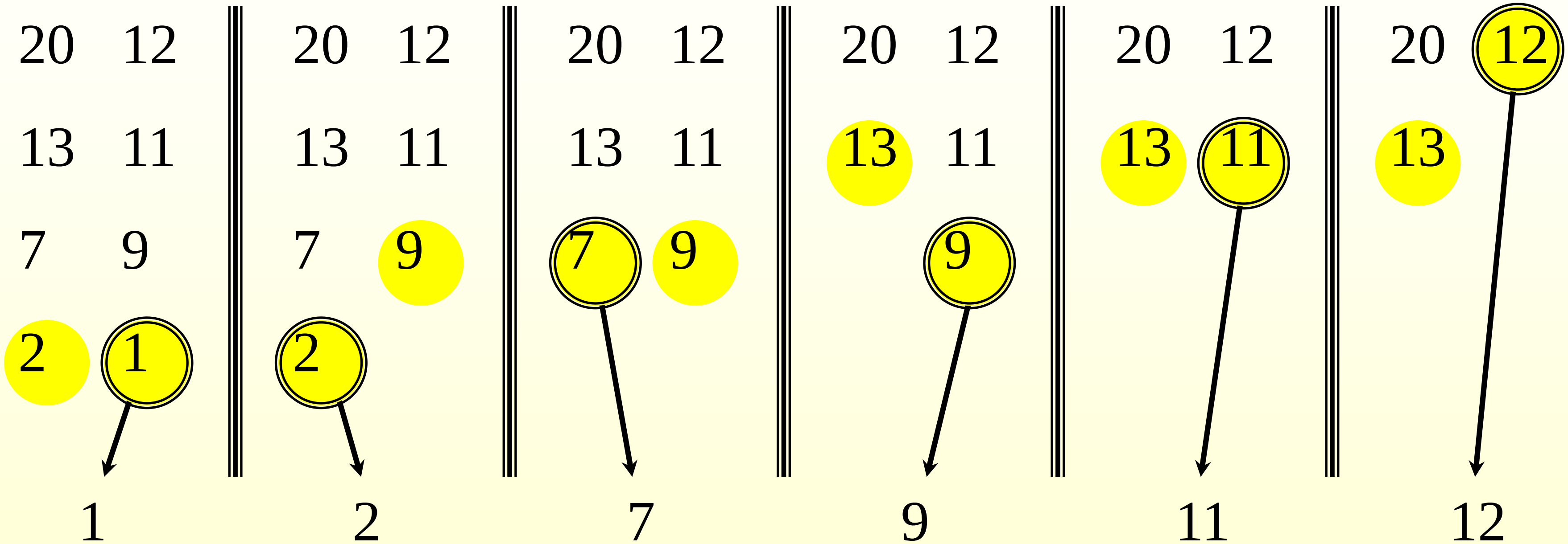
 # subproblems

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 MergeSort

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

- 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 InsertionSort

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 InsertionSort

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

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

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 Signify?

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

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

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

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

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

Solving Recurrences

- 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 Recurrences

1. Recursion tree / iteration method
2. Substitution method
3. Master method