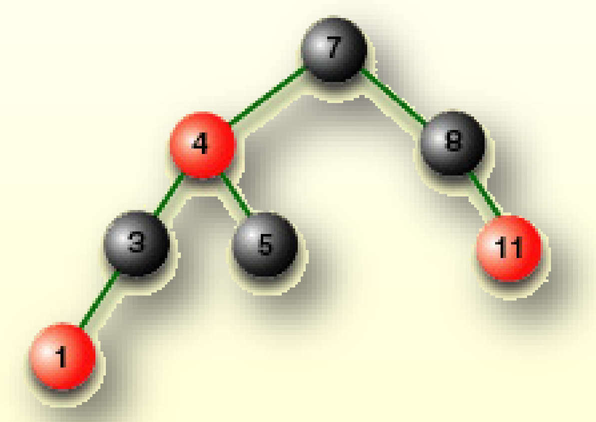
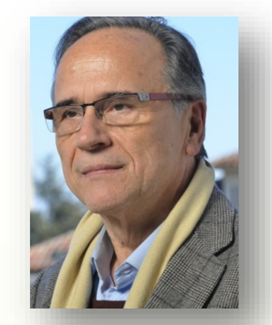
CS161: Design and Analysis of Algorithms



Lecture 1

Leonidas Guibas

The CS161 Team


Leonidas (Leo) Guibas

Abraham Starosta

Bryan Anenberg

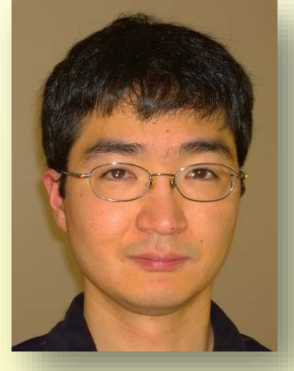
Benjamin Au

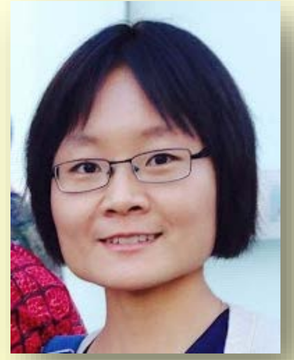
Ari Ekmekji

Kyle Griswold

Seth Hildick-Smith

James Hong


Anthony Kim


Yang Li

Jonathan Necamp

Robbie Ostrow

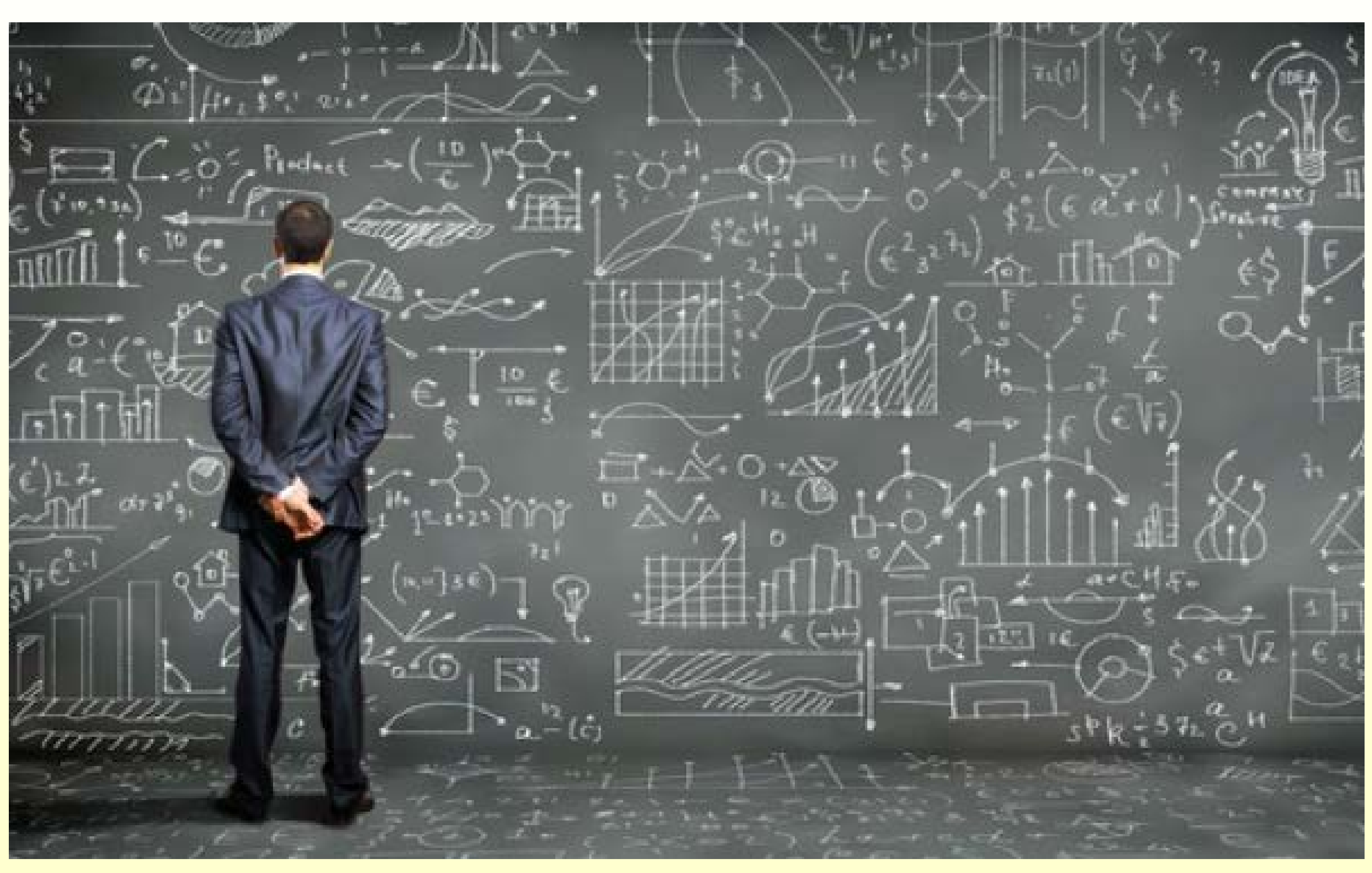
The Class

- CS161 Lectures:
  - Monday/Wednesday, 3:00 – 4:20 pm, in Hewlett 200

- Recitation A:
  - Tuesday, 4:30 – 5:20 pm in 200-034
- Recitation B:
  - Thursday, 5:30 – 6:20 pm in 200-305
- Recitation C:
  - Friday, 11:30 am – 12:20 pm in 200-303
- Recitation D:
  - Friday, 4:30 – 5:20 pm in 200-205

No recitations until the week of Jan 11

Algorithms

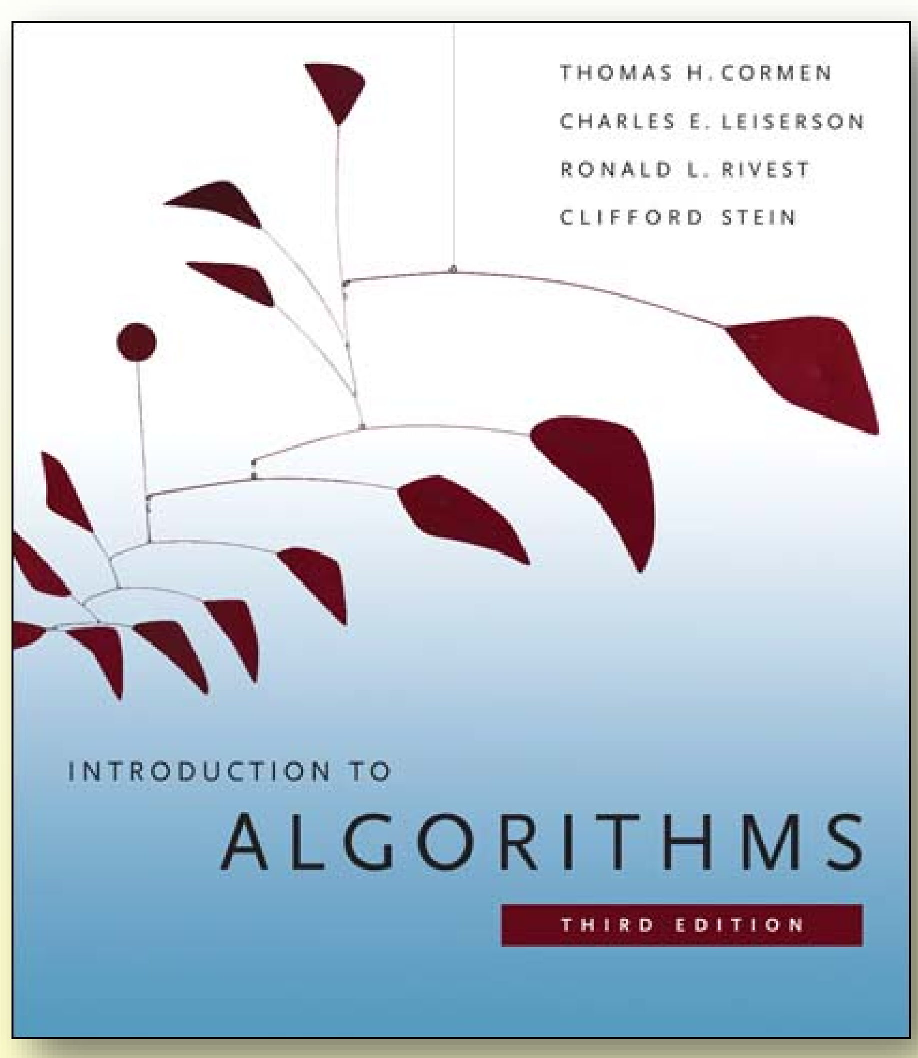


The Topics: Algorithm Design and Analysis

- Algorithm analysis; worst and average case
- Recurrences and asymptotics
- Algorithms for sorting and selection
- Randomized techniques
- Search structures: heaps, balanced trees, skip lists, hash tables
- Dynamic programming and greedy algorithms
- Amortized analysis
- Graph algorithms: breadth- and depth-first search, minimum spanning trees, shortest paths

Algorithms and Data Structures

The Textbook



The CS161 Course Work

- Six paper-and-pencil homeworks
- A programming project
- A midterm
- No final

More Tidbits

- We'll use Piazza as the class discussion board, Gradescope for grading
- You can email the staff at
  - staff-cs-161-16-winter@lists.stanford.edu
- The class web site is
  - http://graphics.stanford.edu/courses/cs161-16-winter, or
  - http://cs161.stanford.edu (redirects to the above)
- Please read the CS161 homework policies, especially regarding the Honor Code

CS 161 Homework Policies

Homework policies

- For both theoretical homeworks, as well as for the programming project, you may work in groups of up to three students per group.
- For the paper-and-pencil homeworks, each student must write up solutions individually. The names of the team collaborators must be listed in the homework write-up. Again, collaboration in a team of up to three is OK when developing the solutions -- but the solution write-ups themselves must be done separately and independently.
- For the programming project, a single write up per group will suffice .
- It is important that the homeworks be turned in on time. Each student is allowed two 24-hour grace periods during the quarter. That means a single homework can be handed in late by two days, or two homeworks may be handed in late by one day each. Other than these grace periods, late homeworks will not be accepted.
- Paper-and-pencil homeworks must be submitted by the day they are due, in electronic form, by 5:00 pm PDT. The programming project and write up must submitted by running the provided submission script, again by 5:00 pm PDT the day they are due.

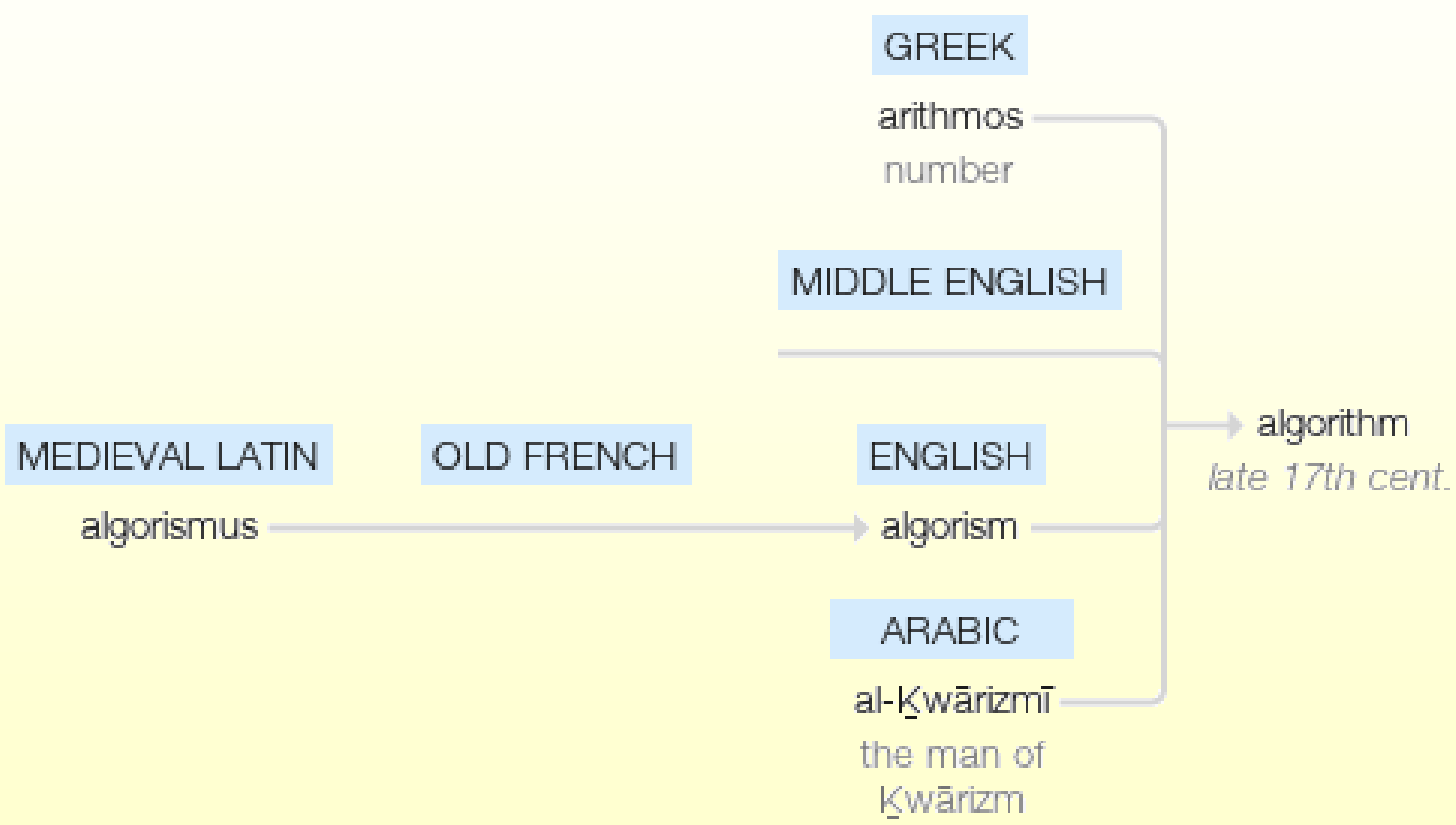
The Honor Code

- Each student is expected to do his/her own work on the problem sets in CS161. A good point to keep in mind is that you must b e able to explain and/or re-derive anything that you submit. Students may discuss problem sets with each other as well as the course staff -- in fact close collaboration in groups of up to three students is permitted, as explained above. Any discussion of problem set questions with others must be noted on a student's final write-up of the problem set answers. Each student must turn in his/her own write-up of the problem set solutions, except for the final project, as noted above. Questions regarding acceptable collaboration should be directed to the class instructor of CAs prior to the collaboration.
- It is a violation of the honor code to copy or derive problem set or exam question solutions from other students or anyone at all, textbooks, previous instances of this course, other courses covering the same topics either at Stanford or at other schools. or any web sources. Copying of solutions from other students, or from students who previously took a similar course is also clearly a violation of the honor code.

Again, the basic rules to follow are:

- You must not look at solutions or program code that are not your own.
- You must not share your solutions or code with other students, except within the group (as above) that you are part of .
- You must indicate on your submission any assistance you have received.

Etymology: Algorithm



Muḥammad ibn Mūsā al-Khwārizmī

(عَبْدَالله مُحَمَّد بِن مُوسَى اَلْخْوَارِزْمِي)

Khwarezm, Khiva

Khiva

City in Uzbekistan

Khiva is a city of approximately 50,000 people located in Xorazm Province, Uzbekistan. It is the former capital of Khwarezmia and the Khanate of Khiva.

What is an Algorithm?

- Algorithms are the high-level ideas behind computer programs.
- An algorithm is the thing that stays the same, whether the program is in C++ running on a Windows or is in JAVA running on a Macintosh.

Following slides modified from

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

What is an Algorithm? (cont'd)

- An algorithm is a precise and unambiguous specification of a sequence of steps that can be carried out to solve a given problem or to achieve a given condition.
- An algorithm accepts some value or set of values as input and produces a value or set of values as output.
- Algorithms [actions] are closely intertwined with the data structures [objects] used to go from input to output values

How to Express Algorithms?

Increasing precision

English

Pseudo-code

Real programming languages

Ease of expression

Describe the *ideas* of an algorithm in English.
Use pseudocode to clarify sufficiently more complex details of the algorithm.

Example: Sorting

- Input: A sequence of n numbers $a_1 \ldots a_n$
- Output: the permutation (reordering) of the input sequence so that $a_1 \leq a_2 \ldots \leq a_n$.
- Possible algorithms you may have seen
  - Insertionsort, Selectionsort, Bubblesort, Quicksort, Mergesort, …
  - These and more in this course
- We seek algorithms that are both *correct* and *efficient*

Insertion Sort: An Incremental Algorithm

```
InsertionSort(A, n) {
  for j = 2 to n {
```

▹ Pre condition: A[1..j-1] is sorted

1. Find position i in A[1..j-1] such that $A[i] \leq A[j] < A[i+1]$
2. Insert A[j] between A[i] and A[i+1]

▹ Post condition: A[1..j] is sorted

```
  }
}
```

1 *j*

sorted

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
  }
}
```

1 i j

sorted Key In-place sorting!

Correctness

Correctness

- What makes a sorting algorithm correct?
  - In the output sequence, the elements are ordered in non-decreasing order
  - Each element in the input sequence has a unique appearance in the output sequence
    - [2 3 1] => [1 2 2] X
    - [2 2 3 1] => [1 1 2 3] X

Correctness

- For any algorithm to be correct, 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)

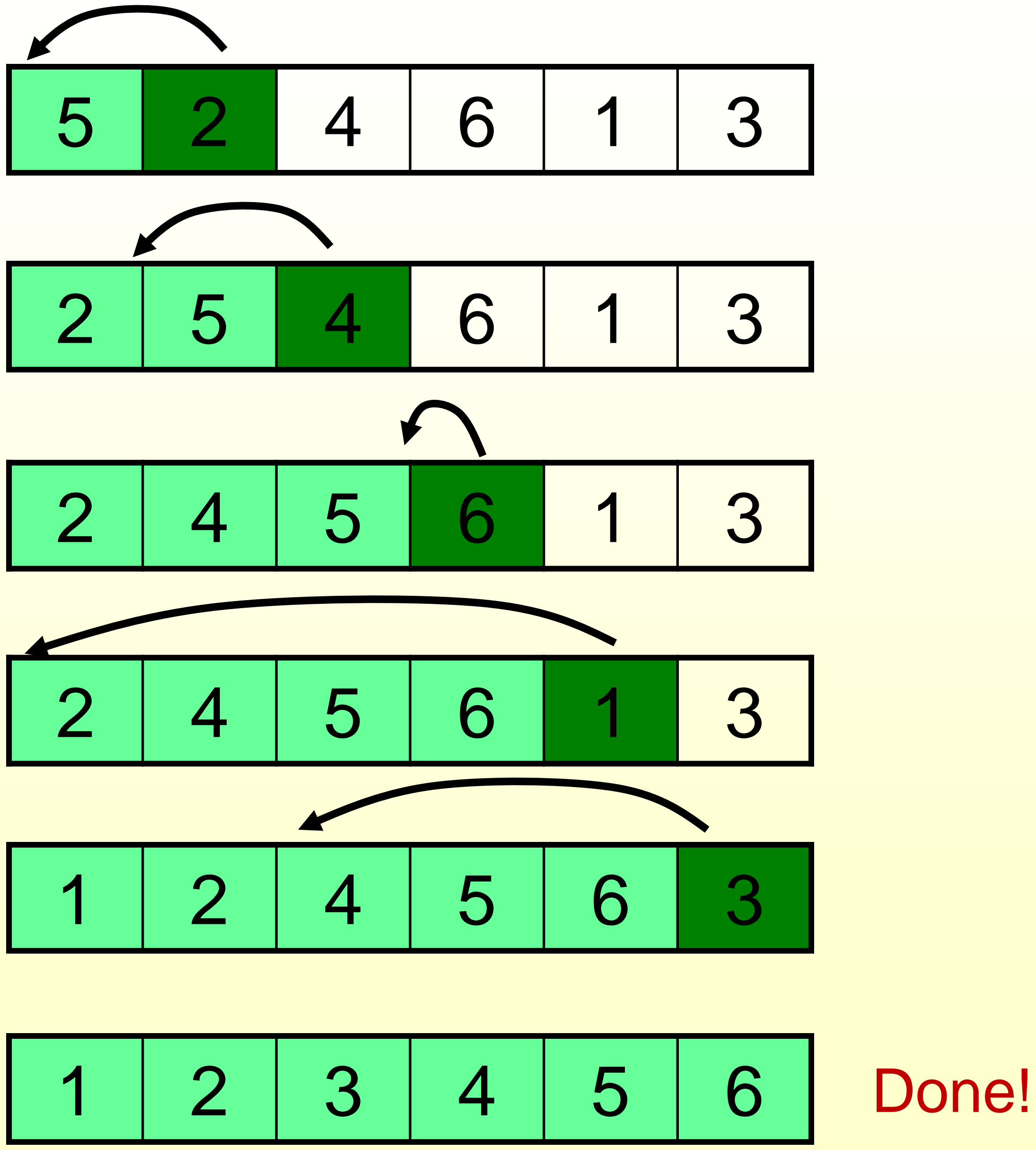
How to Prove Correctness?

- Given a particular input, e.g. <4,2,6,1,7> trace it and prove that it works.
- Given an abstract/general input, e.g. $<a_1, \ldots a_n>$ trace it and prove that it works.
- Sometimes it is easier to find a counterexample to show that an algorithm does NOT work.
  - Think about all small examples
  - Think about examples with “extreme” data values
  - Think about examples with ties
  - Failure to find a counterexample does NOT mean that the algorithm is correct

An Example: Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    ▷Insert A[j] into the sorted sequence A[1..j-1]
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key
  }
}
```

Example of Insertion Sort



Loop Invariants and Correctness of Insertion Sort

- Claim: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order.
- Proof: by induction

Review: Proof By Induction

- Claim: S(n) [Insertion Sort correctly sorts any array of length n] is true for all n >= 1
- Basis:
  - Show formula is true when n = 1
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n = k
- Step:
  - Show that formula is then true for n = k+1

Prove Correctness Using Loop Invariants

- Initialization (basis): the loop invariant is true prior to the first iteration of the loop
- Maintenance:
  - Assume that it is true before an iteration of the loop (Inductive hypothesis)
  - Show that it remains true before the next iteration (Step)
- Termination: show that when the loop terminates, the loop invariant gives us a useful property to show that the algorithm is correct

Prove Correctness Using Loop Invariants

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j];
      i = j - 1;
      ▷Insert A[j] into the sorted sequence A[1..j-1]
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i - 1;
      }
      A[i+1] = key
  }
}
```

Loop invariant: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Initialization

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j];
      i = j - 1;
      ▷Insert A[j] into the sorted sequence A[1..j-1]
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i];
            i = i – 1;
      }
      A[i+1] = key
  }
}
```

Subarray A[1] is sorted. So loop invariant is true before the loop starts.

Maintenance

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
     }
     A[i+1] = key
  }
}
```

Assume loop variant is true prior to iteration j

Loop variant will be true before iteration j+1

1 i j

sorted

Key

Termination

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;
     }
     A[i+1] = key
  }
}
```

The algorithm is correct!

Upon termination, A[1..n] contains all the original elements of A in sorted order.

1 *n* *j=n+1*

Sorted

Efficiency

Efficiency

- Correctness alone is not sufficient
- Simple, 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?

How to Measure Algorithm Complexity?

- Absolute running time is not always a good measure
- It depends on input
- It depends on the machine you used and on who implemented the algorithm
- Makes it hard to compare algorithms

- We would like to have a higher-level analysis that does not depend on those factors

Machine-Independent Complexity

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each simple operation (e.g. +, -, =, *, if, for) takes 1 step.
    - Loops and subroutine calls are *not* simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits

Running Time = Operation Count

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(x)
- We can be more exact if need be

Asymptotic Analysis

- Running time (i.e., operation count) depends on the size of the input
  - Larger array takes more time to sort
  - T(n): the time taken on input with size n
  - Look at ***growth*** of $T(n)$ as $n \to \infty$.

    "Asymptotic Analysis"
- Size of input is generally defined as the number of input elements
  - Some cases may be tricky

Running Time of Insertion Sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

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	
`}`		

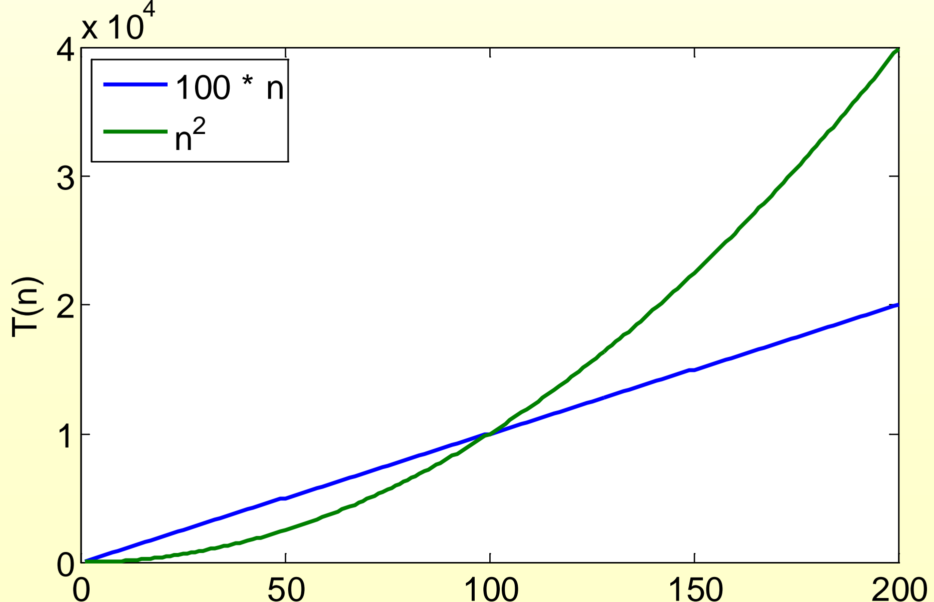
$S = t_2 + t_3 + \ldots + t_n$ where t_j is number of while expression evaluations for the jth for loop iteration

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
    - $t_j = j \Rightarrow S = 2 + 3 + \ldots + n = n(n+1)/2 - 1$
    - $T(n) = an^2 + bn + c$ 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

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

(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 next time

Another Example: 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

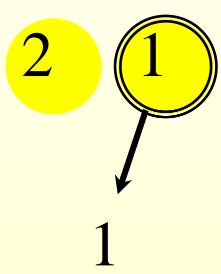
20	12
13	11
7	9
2	1

Merging Two Sorted Arrays

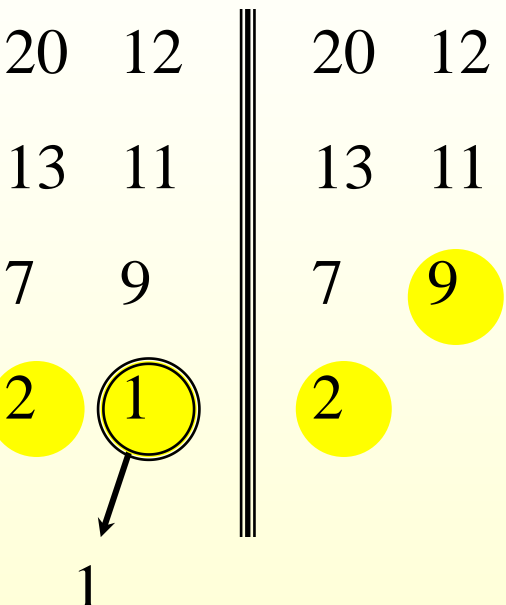
20 12

13 11

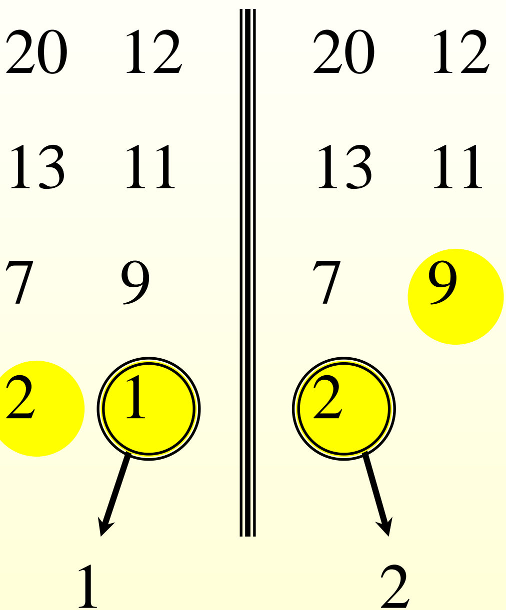
7 9



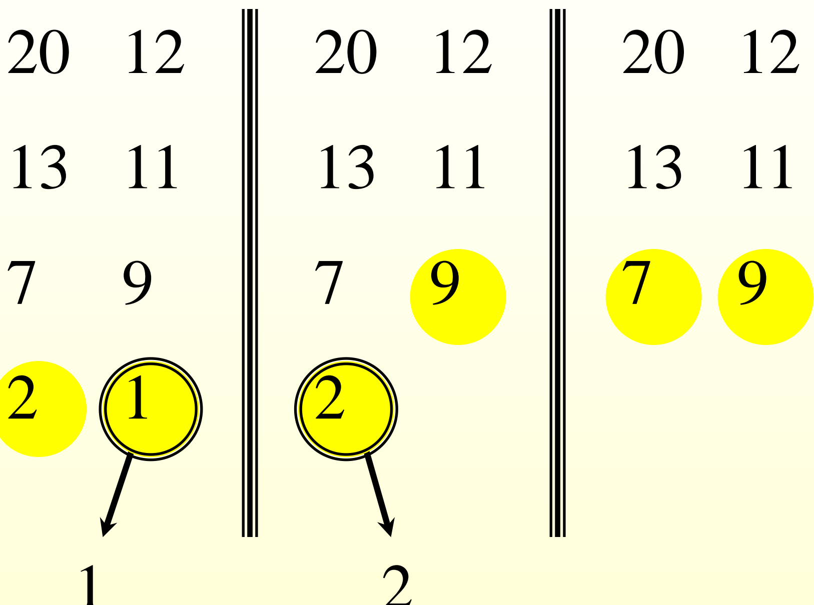
Merging Two Sorted Arrays



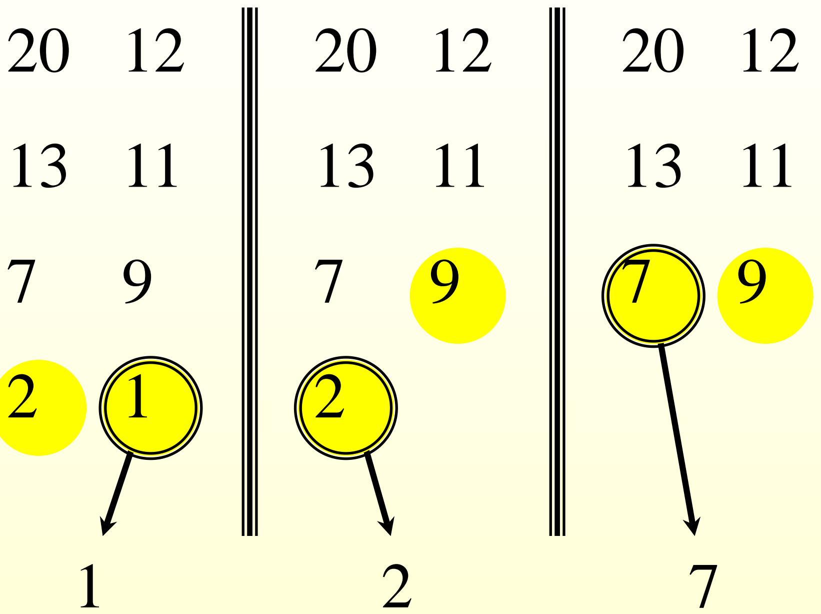
Merging Two Sorted Arrays



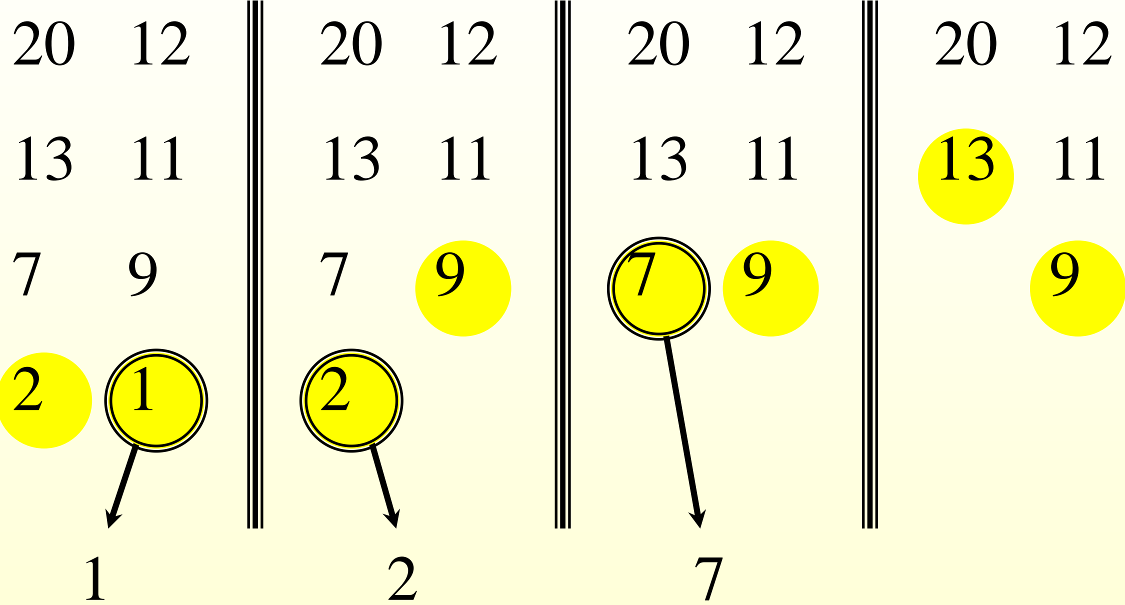
Merging Two Sorted Arrays



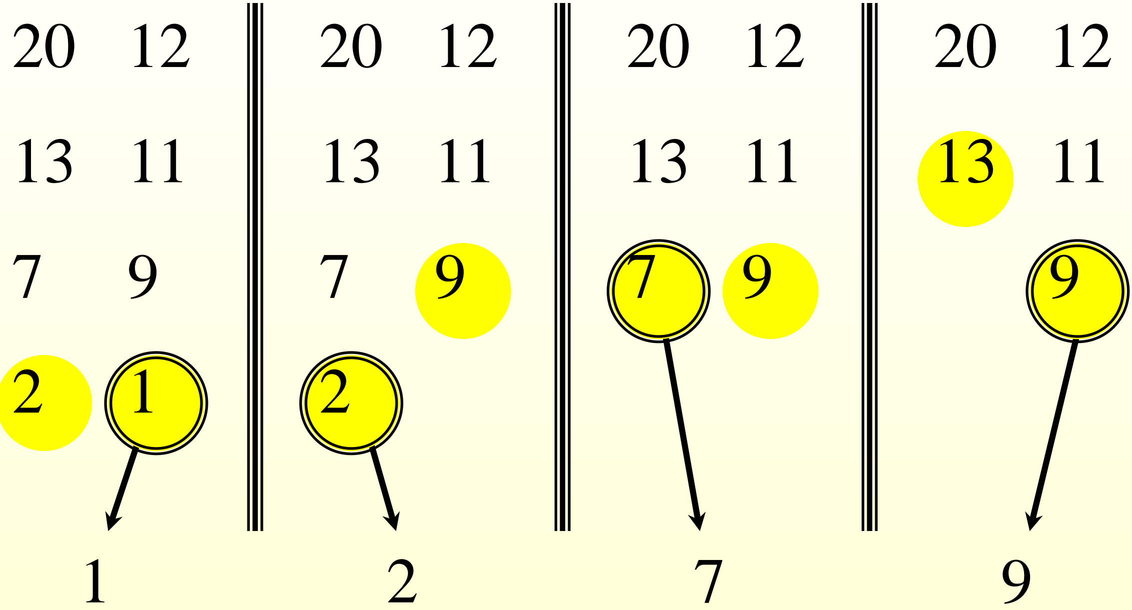
Merging Two Sorted Arrays



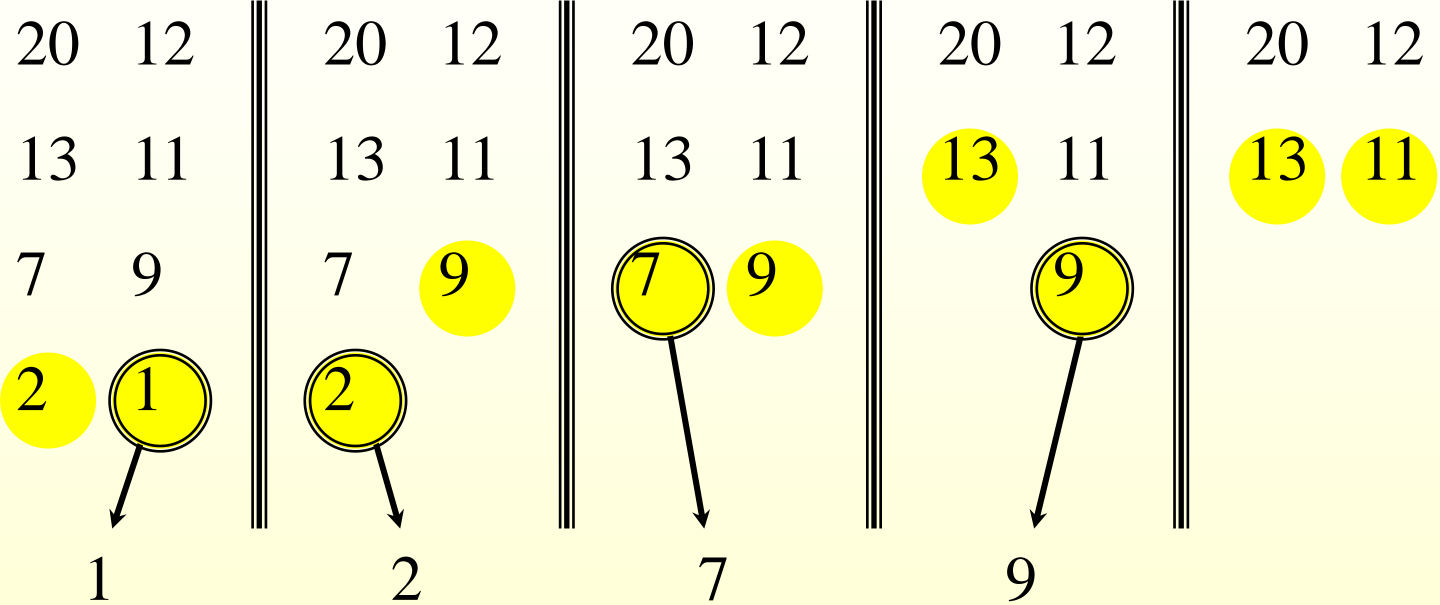
Merging Two Sorted Arrays



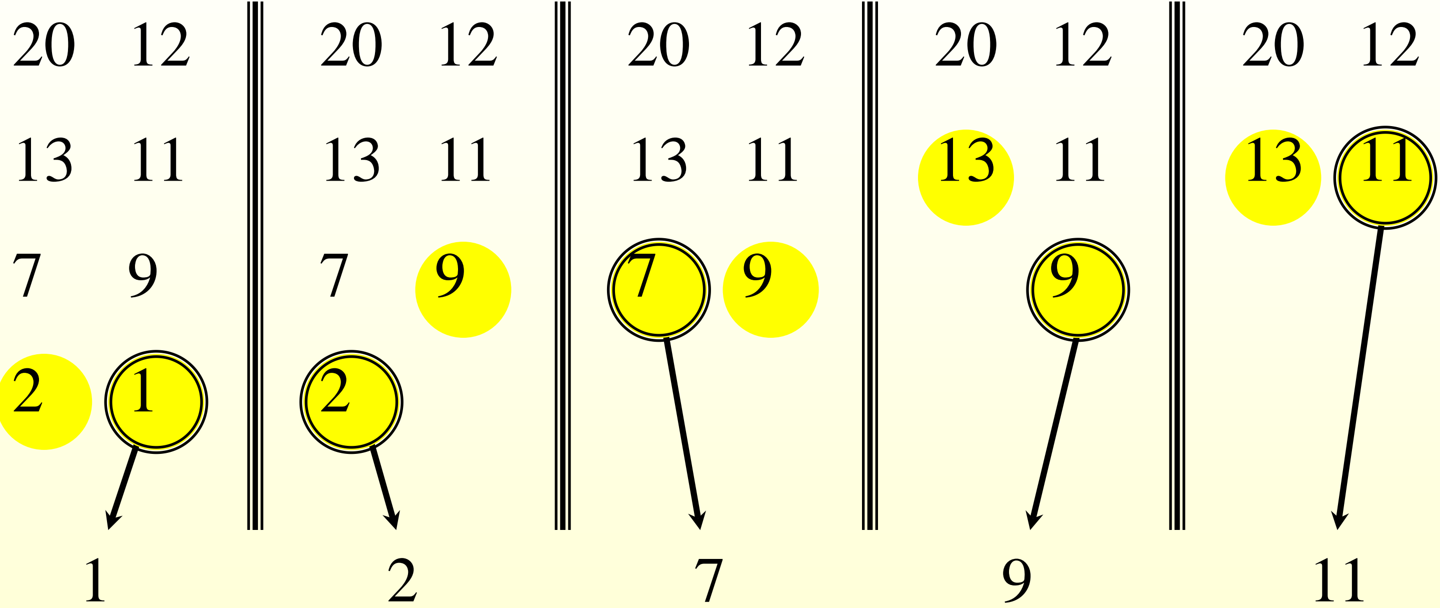
Merging Two Sorted Arrays



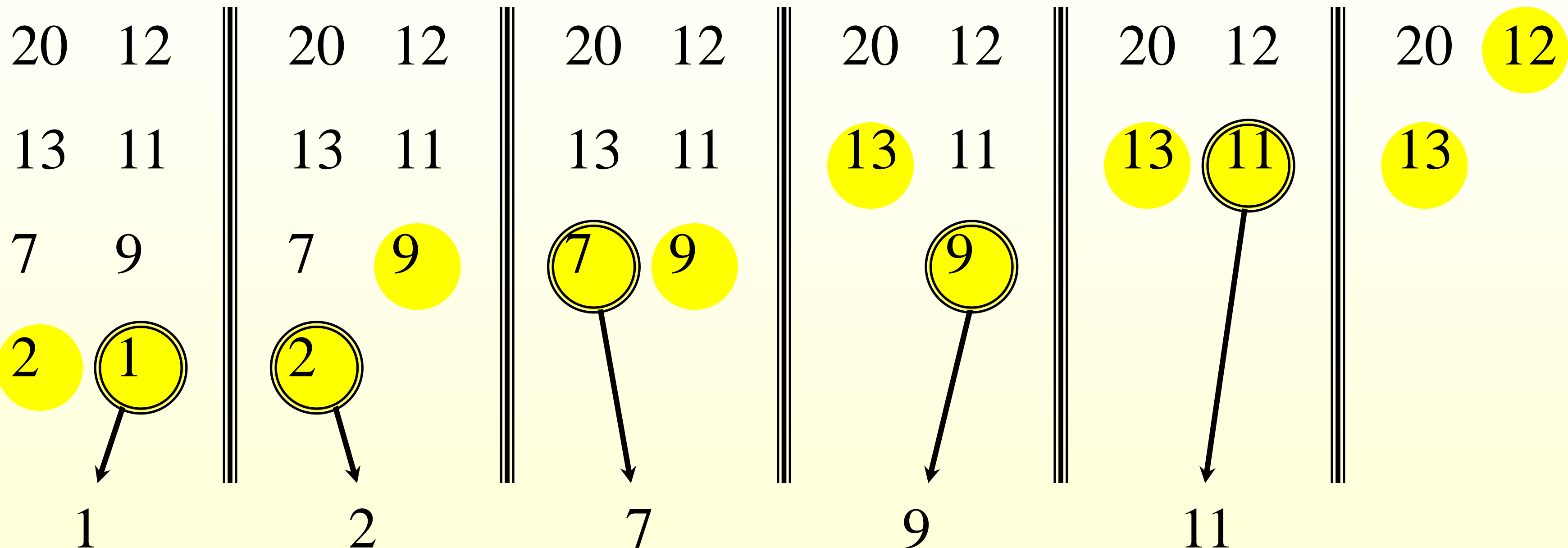
Merging Two Sorted Arrays



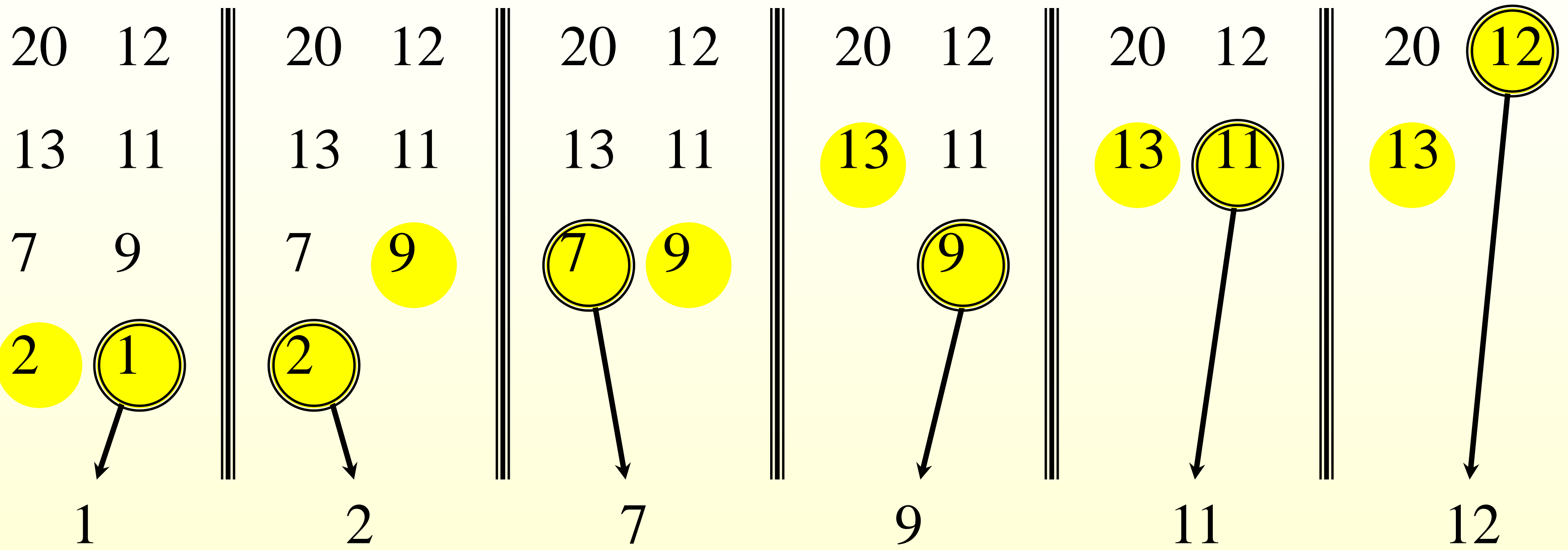
Merging Two Sorted Arrays



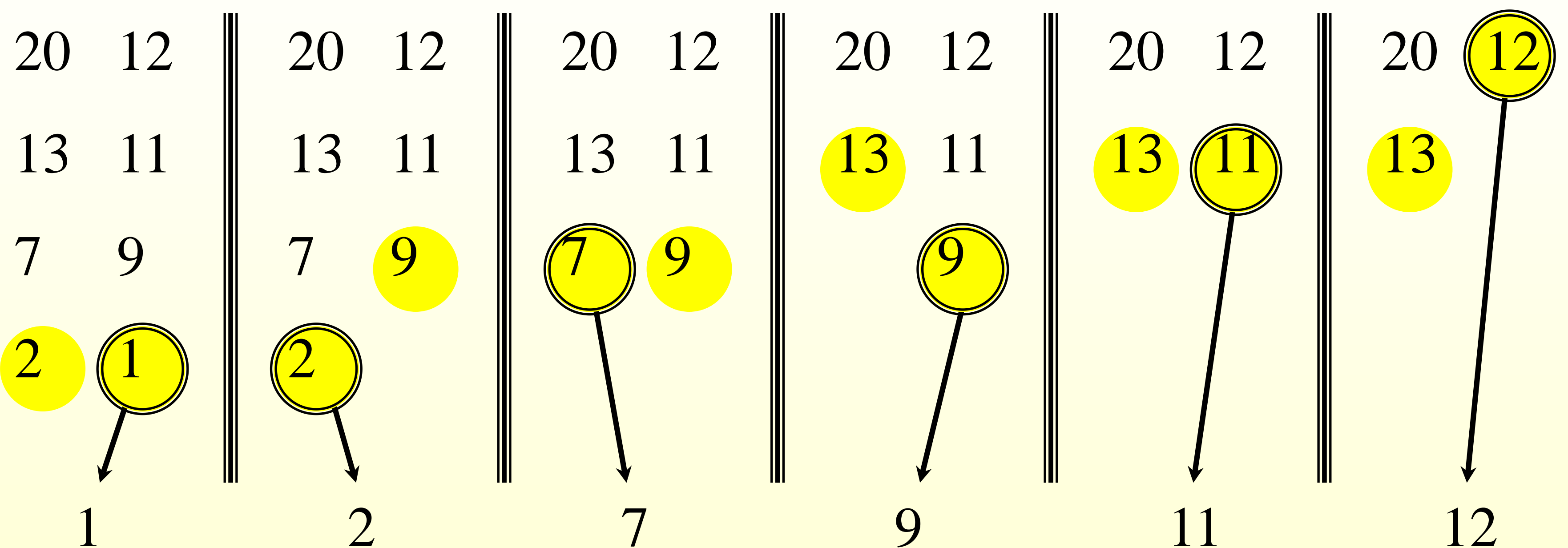
Merging Two Sorted Arrays



Merging Two Sorted Arrays



Merging Two Sorted Arrays



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

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 $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$\Theta(n)$	***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.

Recurrence for Merge Sort

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

- We shall usually 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.
- We'll soon see how to find a good upper bound on $T(n)$.

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

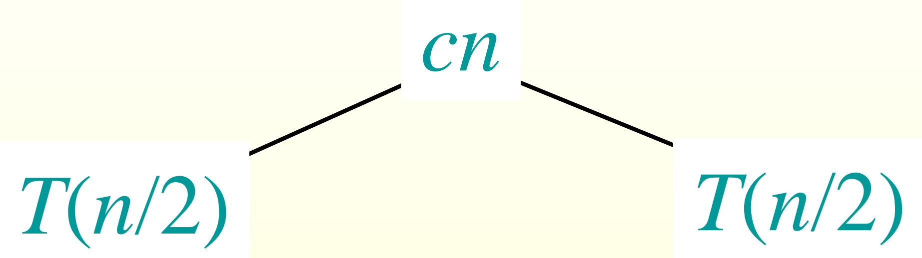
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

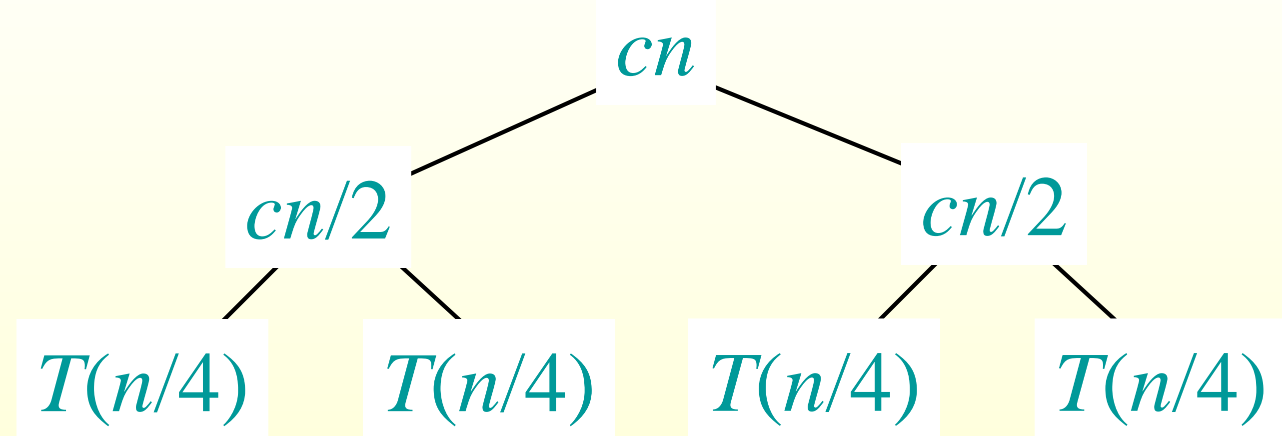
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



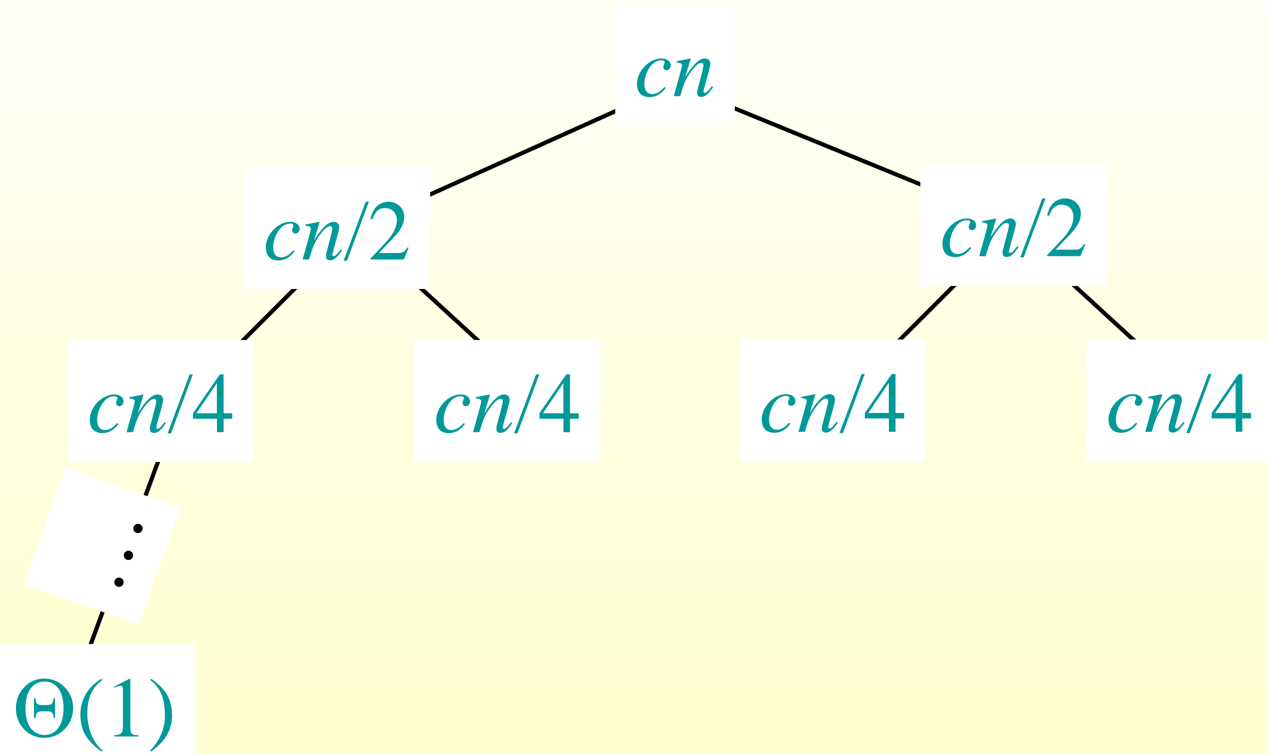
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



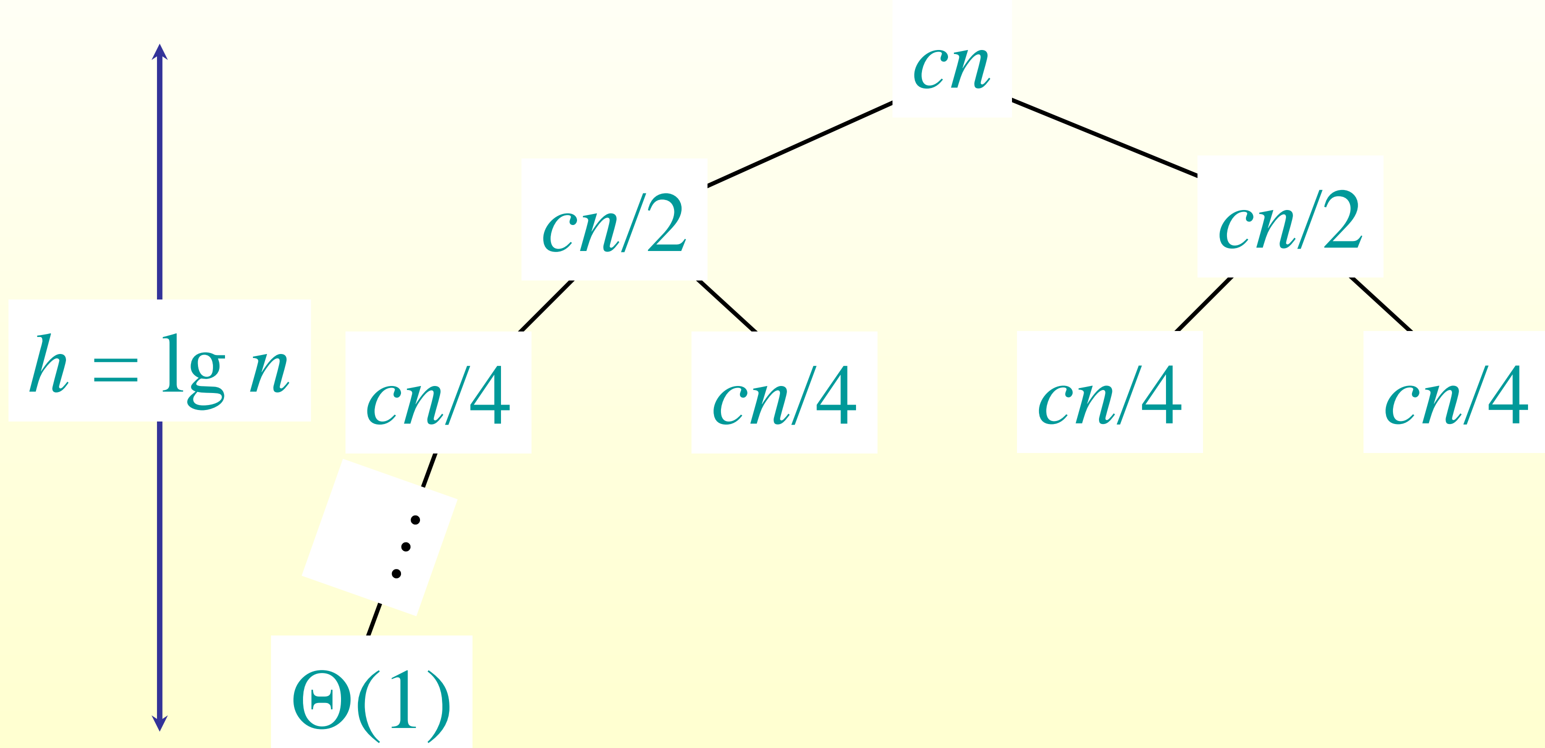
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



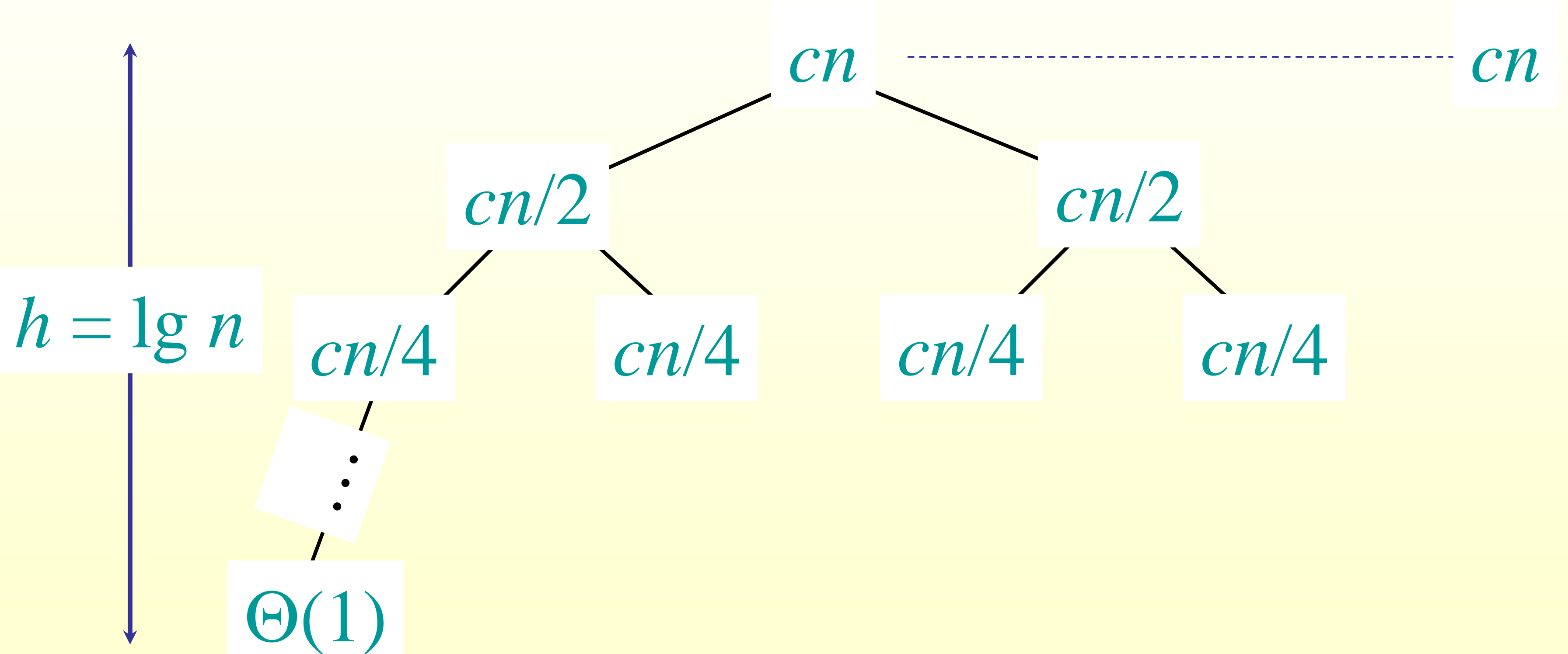
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



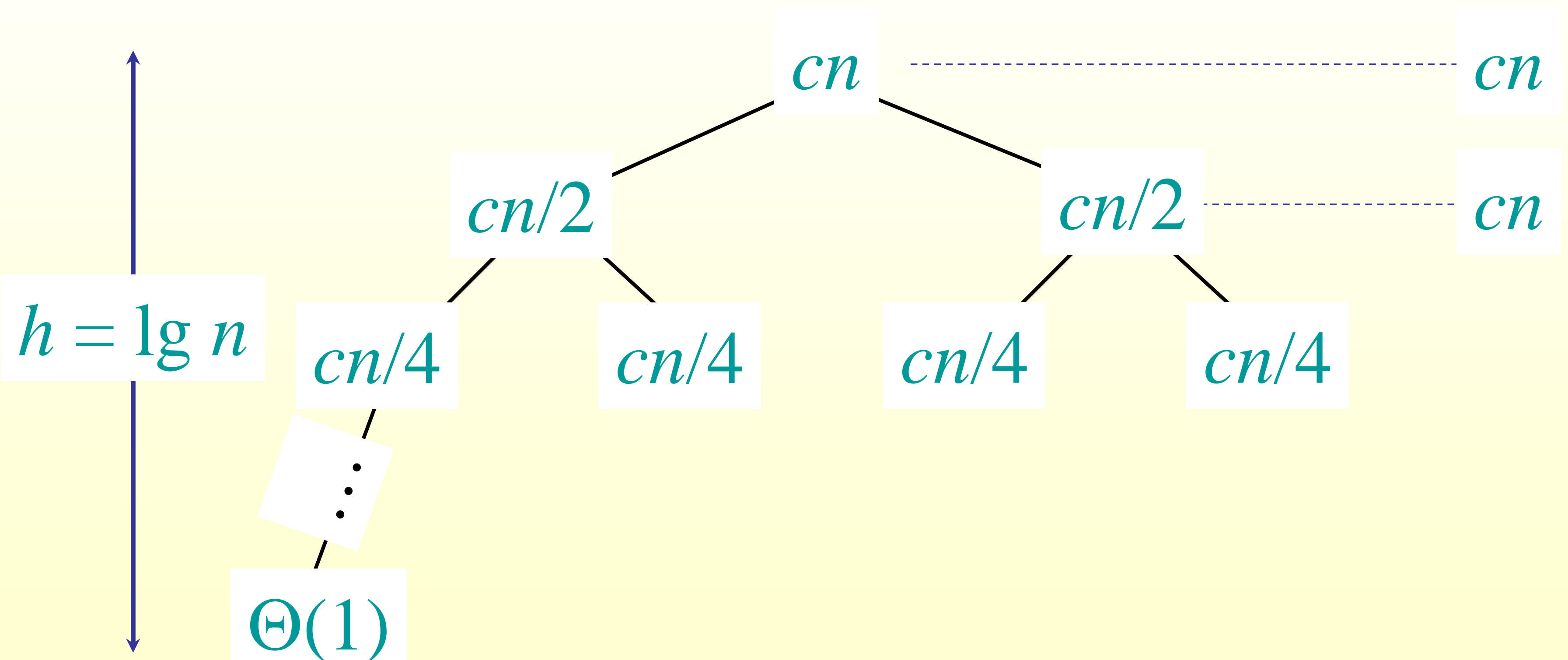
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



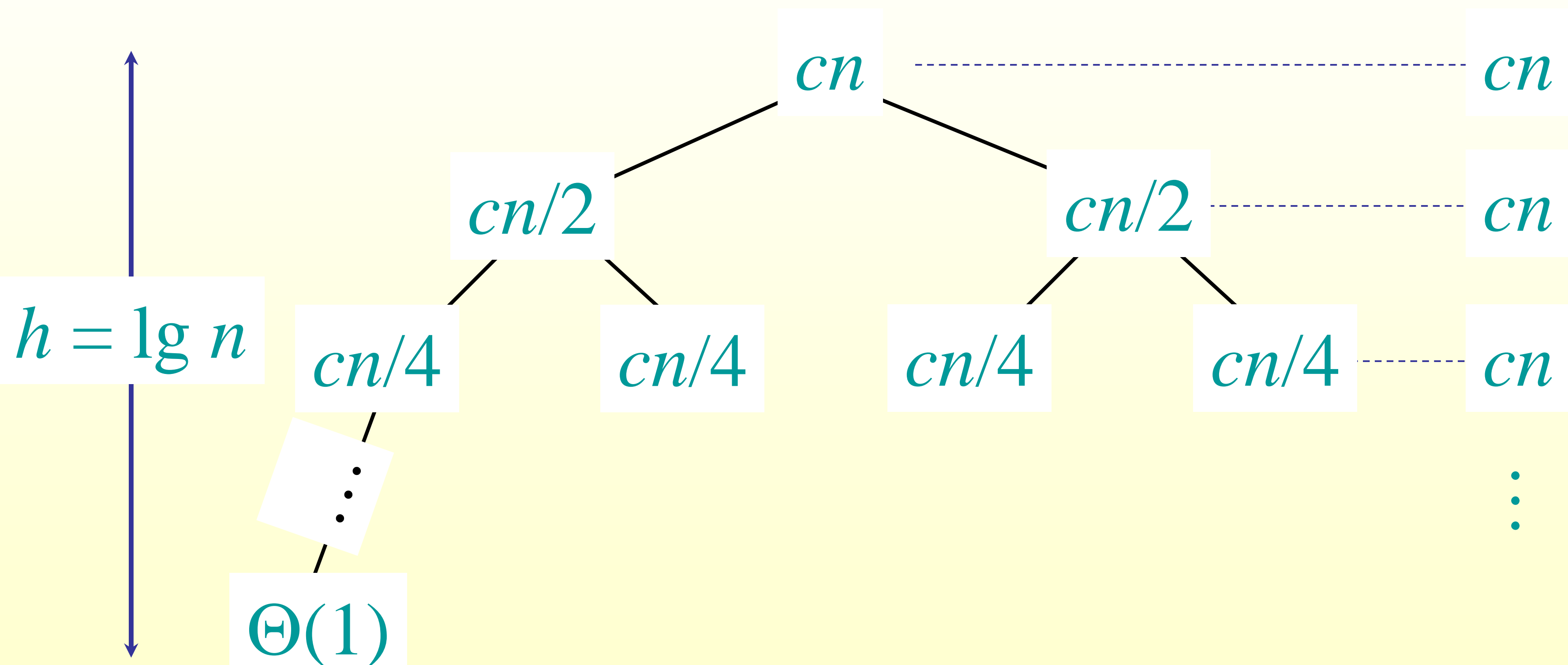
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



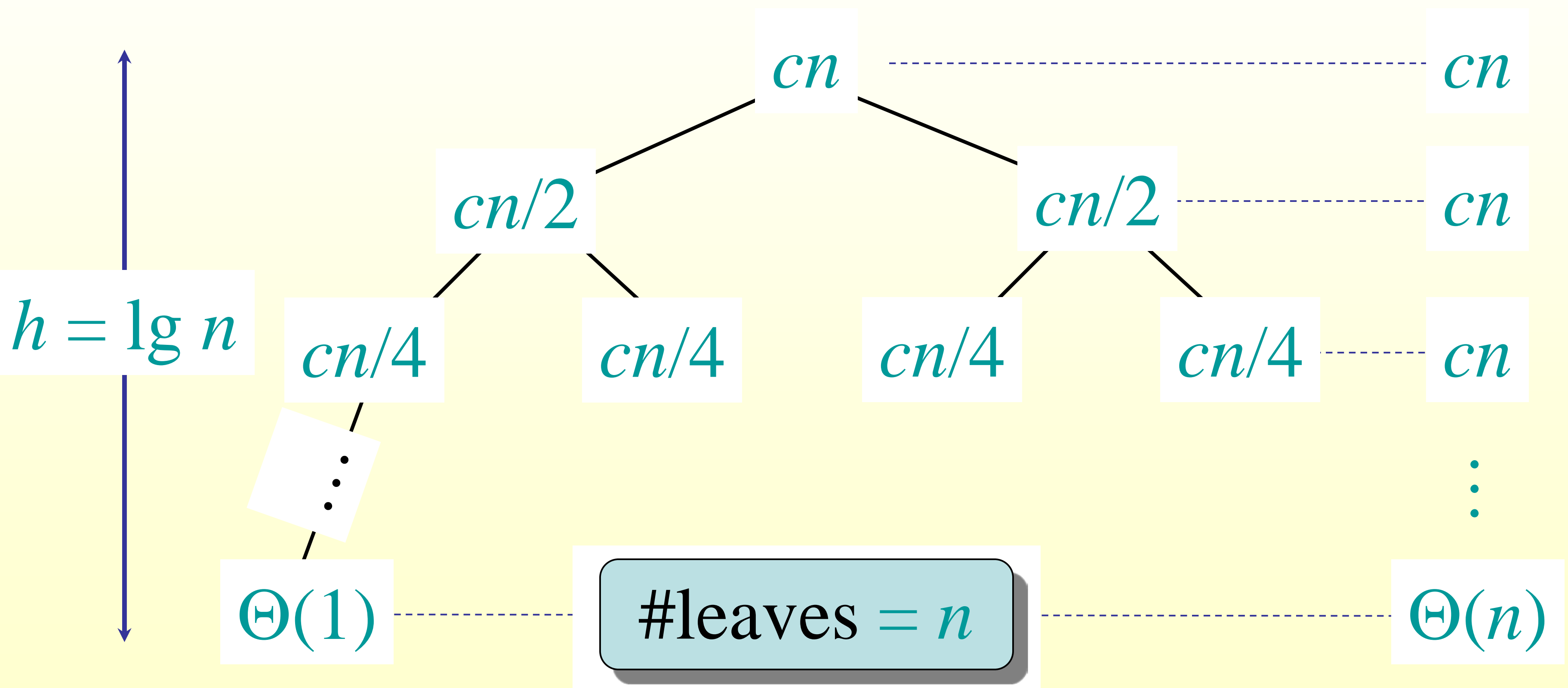
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



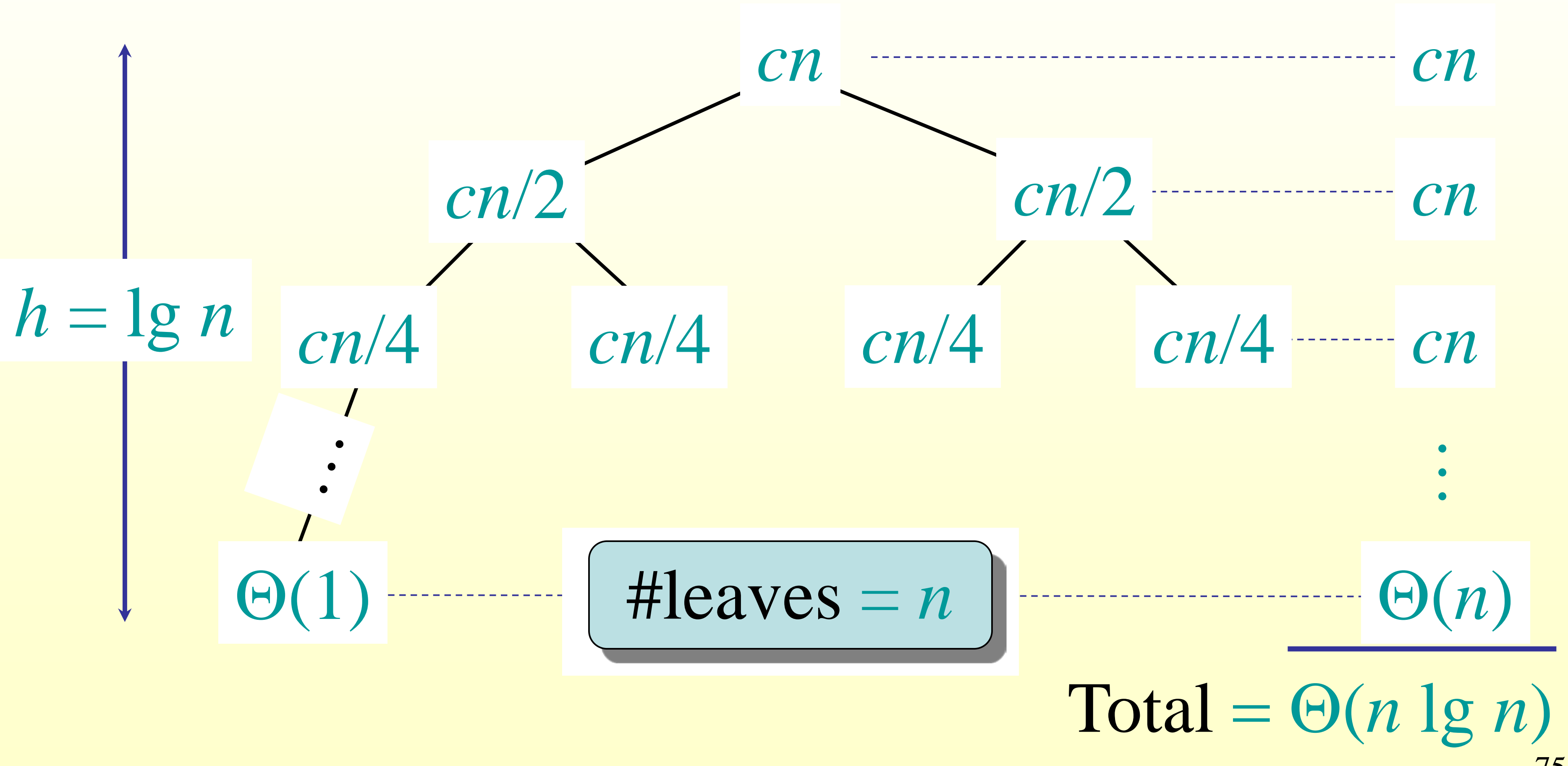
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



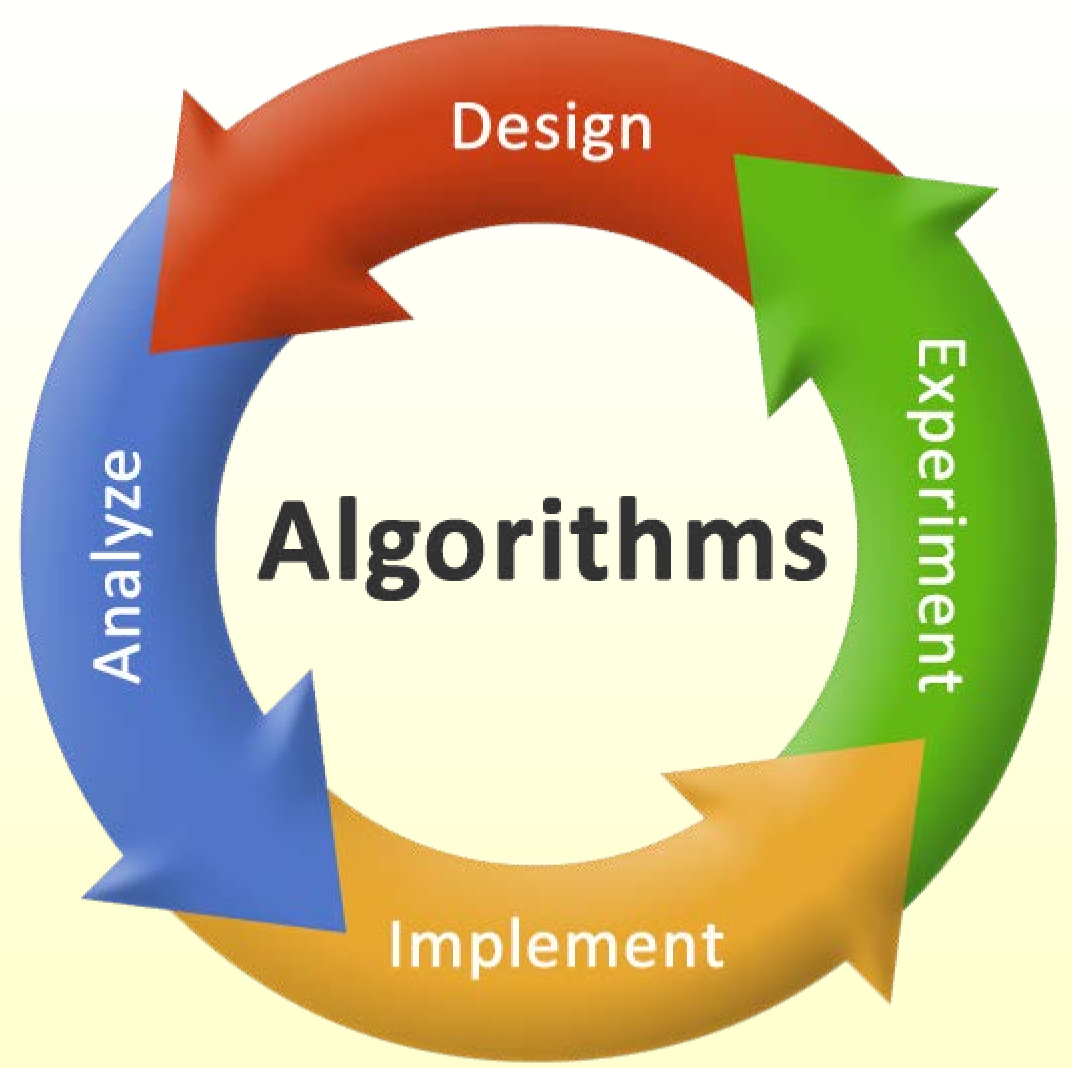
Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Conclusion

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, Merge Sort asymptotically beats Insertion Sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.


Design
Experiment
Algorithms
Analyze
Implement