CS 325

Syllabus

Which discrete math did you take?

a) MTH 231 at OSU
b) MTH 231 at LBCC
c) CS 225 at OSU
d) Other discrete math class
e) Never took discrete math

Did you take Stats?

a) ST 314 at OSU
b) Engineering Stats at LBCC
c) Other stats class Stats class
d) Never took stats

Which programming language would you use?

a) C
b) C++
c) Python
d) Other

CS 325 - Topics

- Asymptotic Analysis and Complexity Classes.
- Analysis on experimental run time data.
  - Linear, polynomial, exponential regression
- Recursion, recurrences
- Performance determines what is feasible and what is impossible.
- When to use a Heuristic or an Approximation Algorithm
  - Travelling Salesman Problem
  - Knapsack Problem
  - Scheduling

.

Algorithm Design Paradigms

There is no one “best” method to solve all problems

- Brute Force
- Divide and Conquer
- Greedy
- Dynamic Programming
- Graph Algorithms
- Approximation

Problems vs Algorithms

- Many algorithms exist to solve the sorting problem.
- Running time is associated with an algorithms.
- Bounds on running times may also be associated with the problem.

Example

- Problem: Sorting a list of integers
- Algorithms: Insertion Sort, Merge Sort, Naive Sort

How do we compare Algorithms?

We need to define a number of objective measures.

- Compare execution times?

  Not good: times are specific to a particular computer and programming language!!

- Count the number of statements executed?

  Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute **guarantee** that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest
- Average case = Expected Value
  - Provides a prediction about the running time
  - Assumes that the input is random

Input Size

Express running time as a function of the input size *n* (i.e., f(n)).

- size of an array
- # of elements in a matrix
- # of bits in the binary representation of the input
- vertices and edges in a graph

Asymptotic Analysis

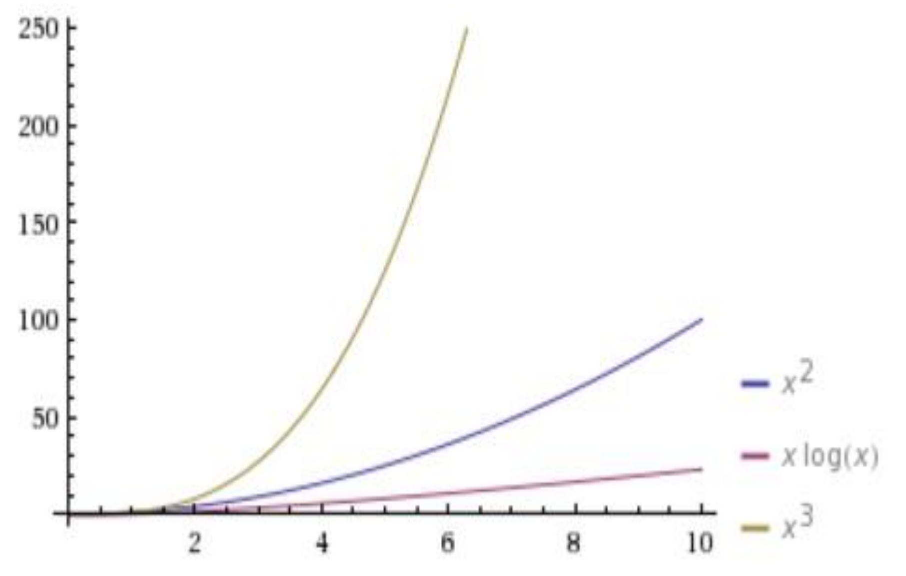
To compare two algorithms with running times *f(n)* and *g(n)*, we need a **rough measure** that characterizes **how fast each function grows.**

- Running time of an algorithm as a function of input size *n* **for large *n***.
- Compare functions in the limit, that is, **asymptotically!** (i.e., for large values of *n*)
- Worst Case Analysis

Problems vs Algorithms

- For any given problem there are potentially many different types of algorithms to solve it.
- Problem: Sorting a list of integers
- Algorithms: Insertion Sort, Merge Sort, Naive Sort
- Running time
  - Insertion Sort is $O(n^2)$
  - Merge Sort is $O(n\lg n)$
  - Naive Sort is $O(n^3)$

Plot:



Formally Define The Problem of Sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Importance of Sorting

- Maintain a directory of names, phone book, sort by grades of students, …
- Find the median
- Binary Search assumes array is sorted.
- Greedy Algorithms

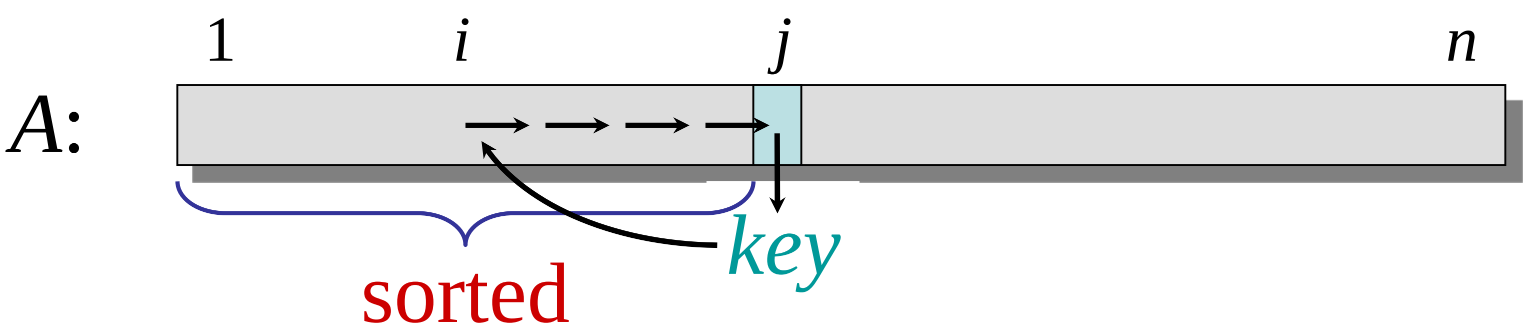
Problem vs Algorithm

- Many algorithms exist to solve the sorting problem.
- Running time is associated with an algorithms.
- Bounds on running times may also be associated with the problem.

Algorithm 1: Insertion sort

“pseudocode”

```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    for j ← 2 to n
          do key ← A[ j]
             i ← j – 1
             while i > 0 and A[i] > key
                   do A[i+1] ← A[i]
                      i ← i – 1
             A[i+1] = key
```



Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 $\quad$ 2 $\quad$ 4 $\quad$ 9 $\quad$ 3 $\quad$ 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 *done*

Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

$$T_A(n) = \text{ time of A on length n inputs}$$

- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of Analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (NEVER)

- Cheat with a slow algorithm that works fast on *some* input.

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} j = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

Insertion sort analysis

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} (j/2) = \Theta\left(n^2\right)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small *n*.
- Not at all, for large *n*.

Insertion sort analysis

Best Case: Already sorted. *Nearly Sorted??*

O(n)

Can we sort better?

Insertion upper bound $O(n^2)$

Are there other ways to sort??

Merge Sort

Sorting Problem: Sort a sequence of n elements into non-decreasing order.

- ***Divide:*** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each

- ***Conquer:*** Sort the two subsequences recursively using merge sort.

- ***Combine:*** Merge the two sorted subsequences to produce the sorted answer.

Merge sort – Pseudo code

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

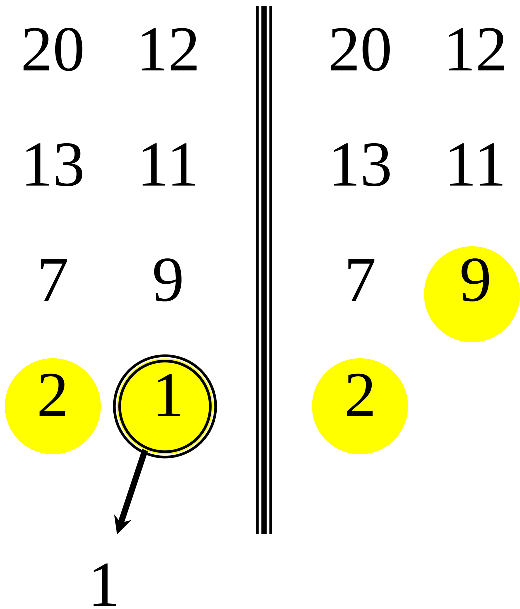
20	12
13	11
7	9
2	1

Merging two sorted arrays

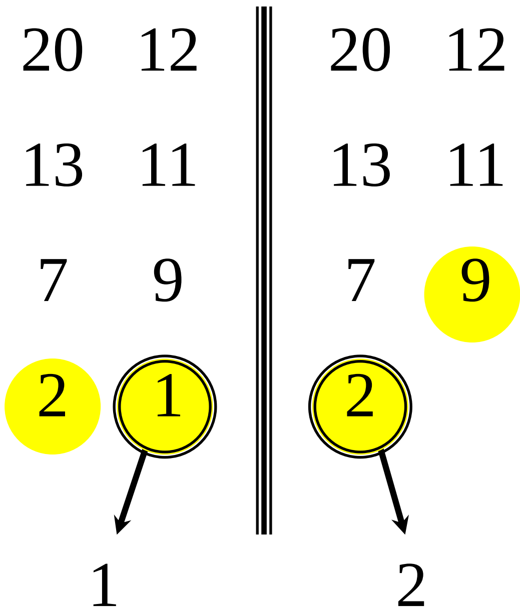
20	12
13	11
7	9
2	1

1

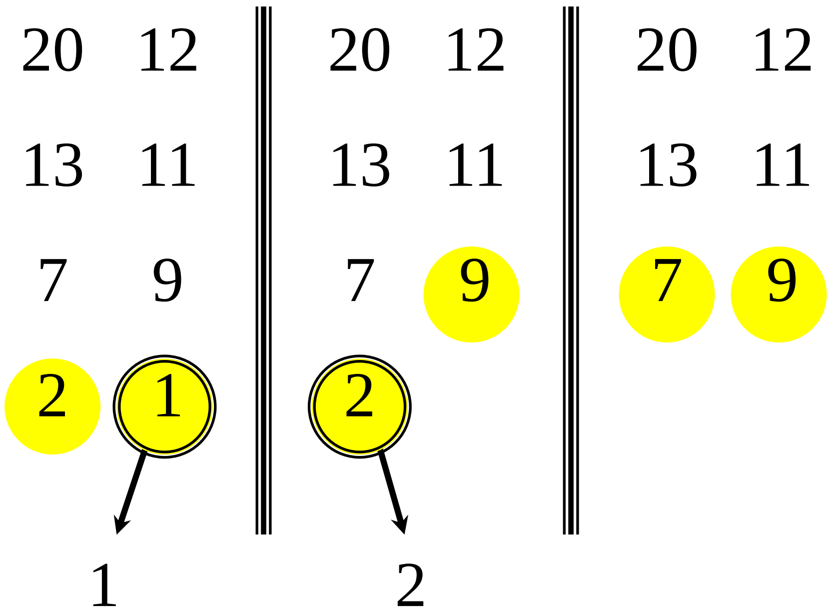
Merging two sorted arrays



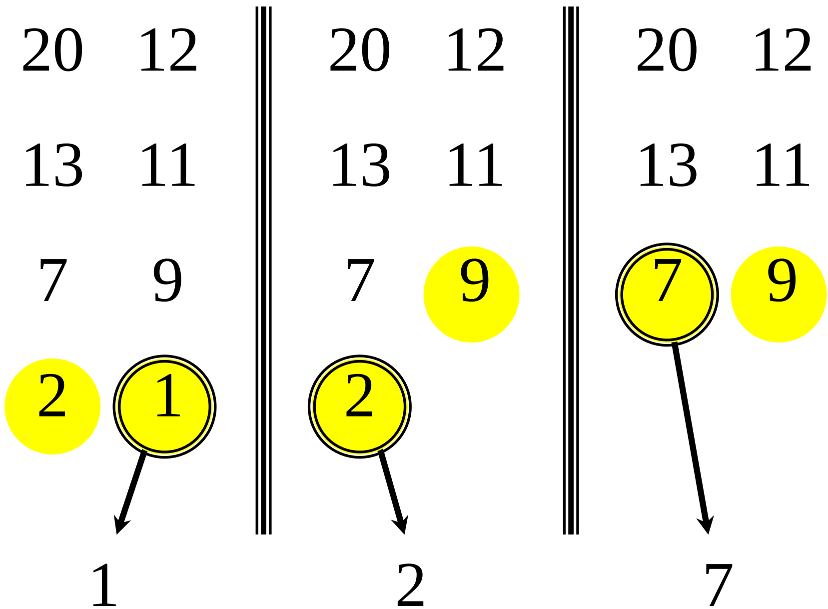
Merging two sorted arrays



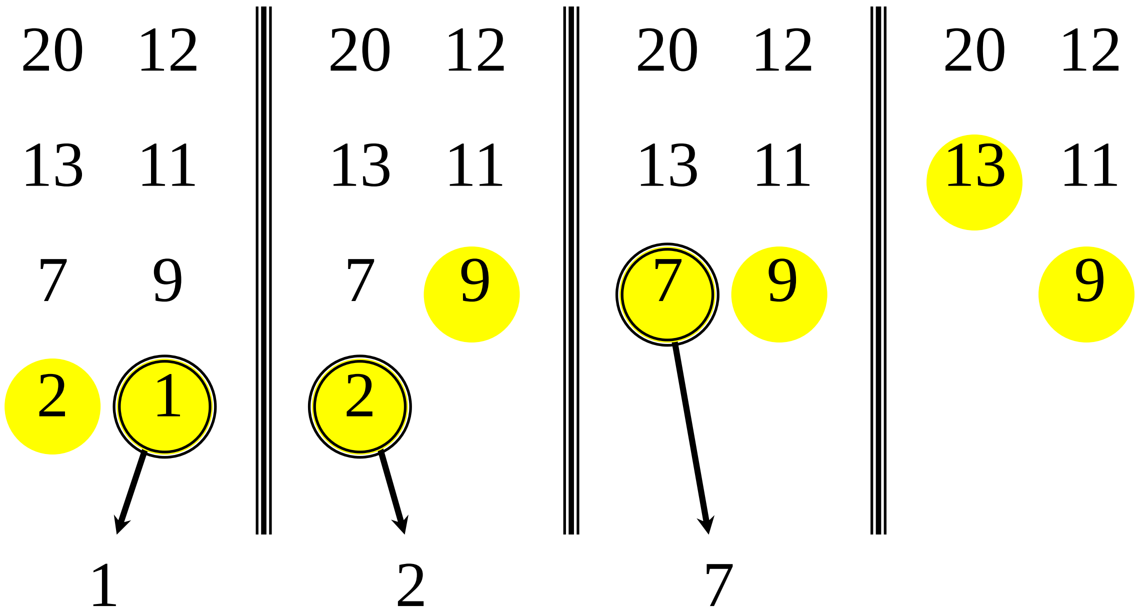
Merging two sorted arrays



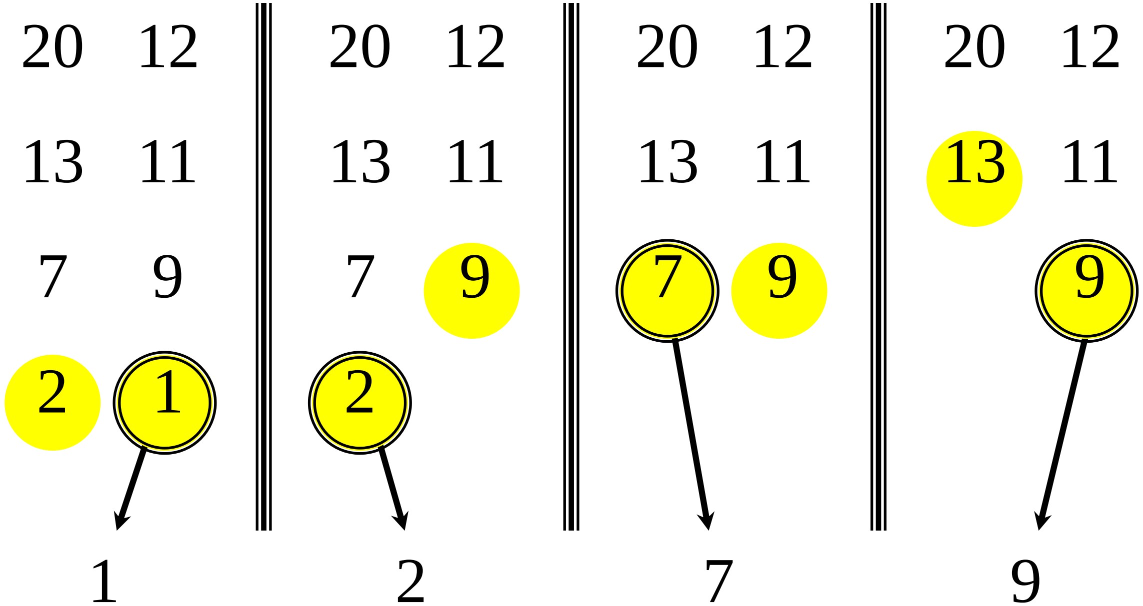
Merging two sorted arrays



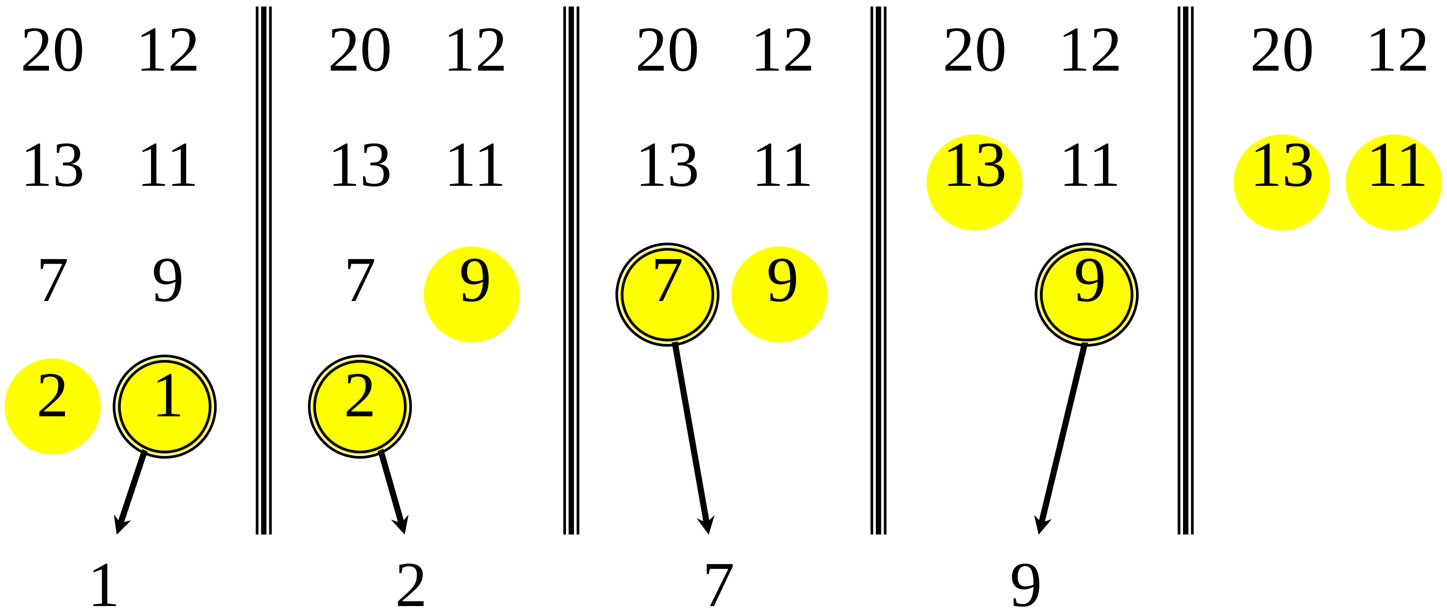
Merging two sorted arrays



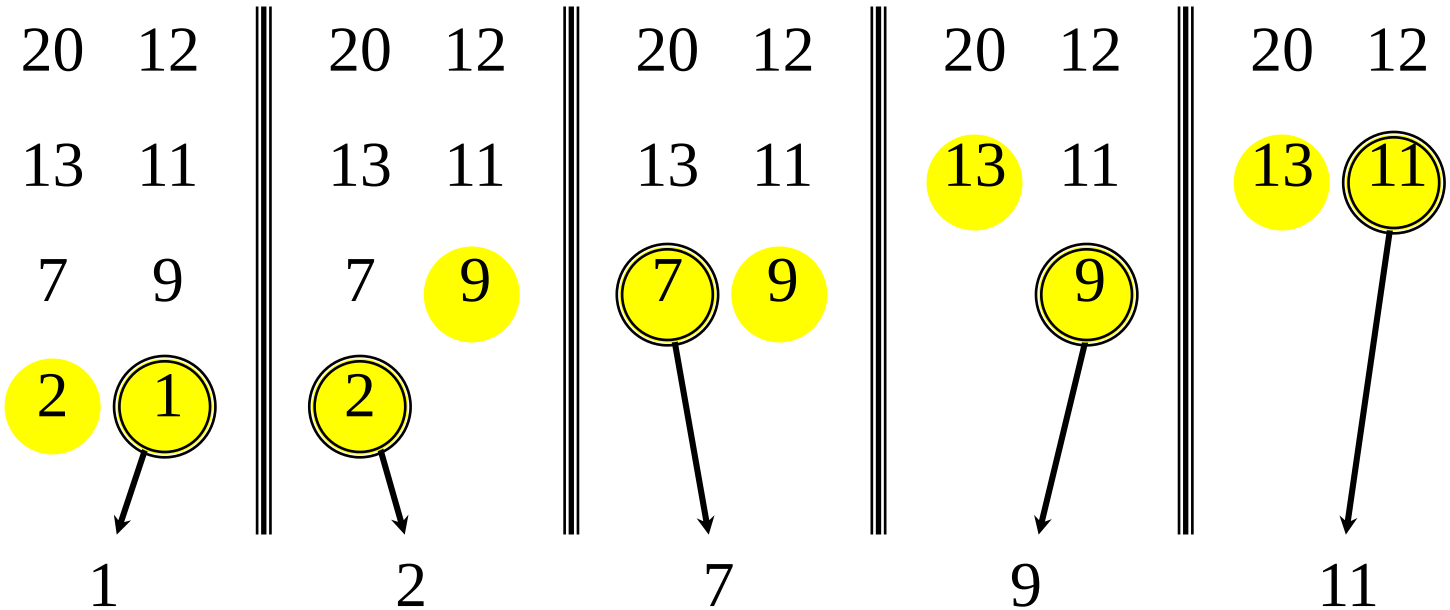
Merging two sorted arrays



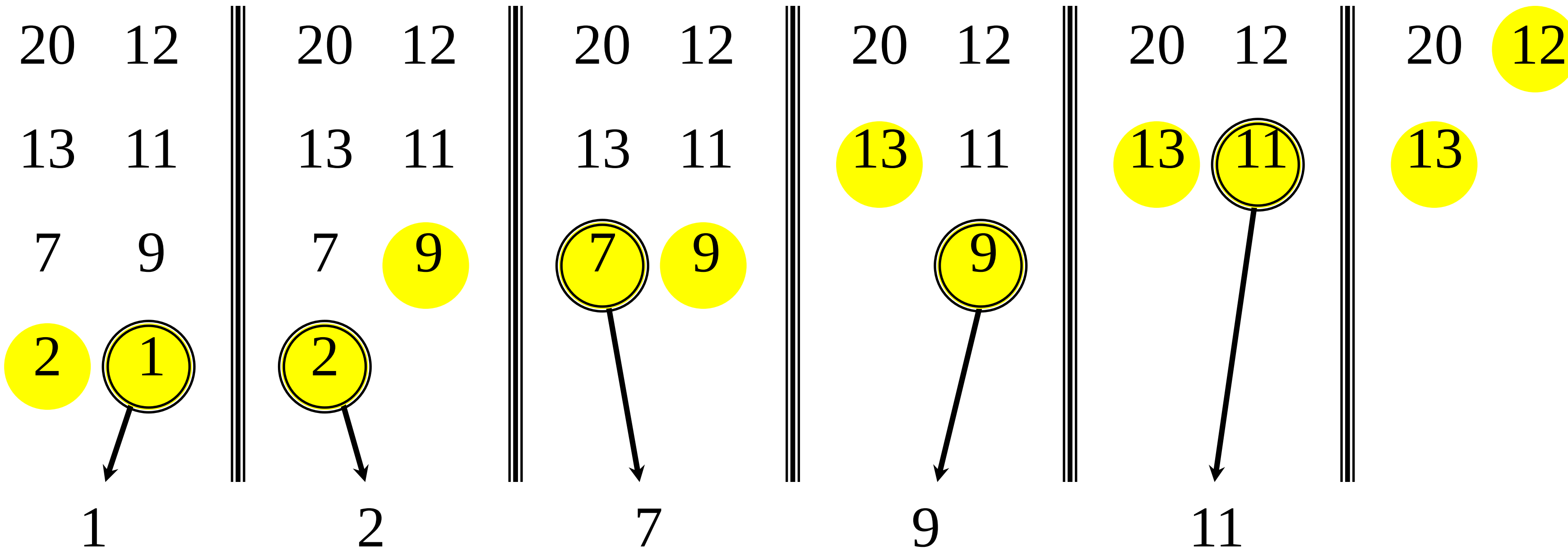
Merging two sorted arrays



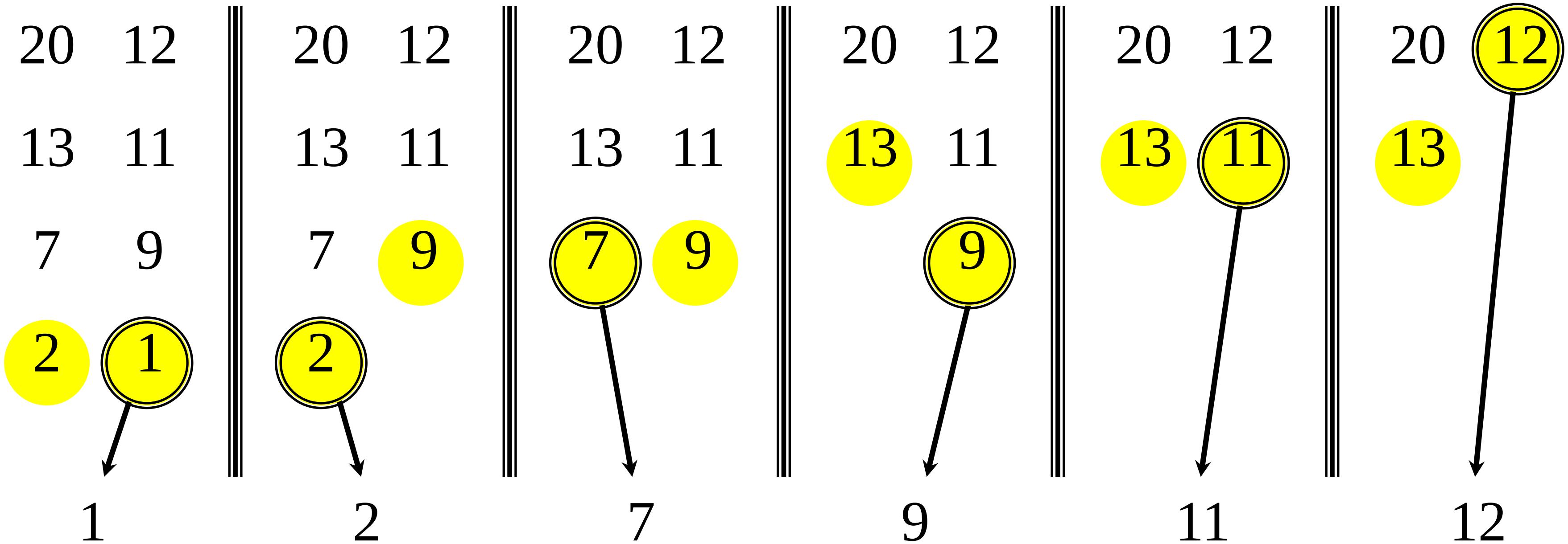
Merging two sorted arrays



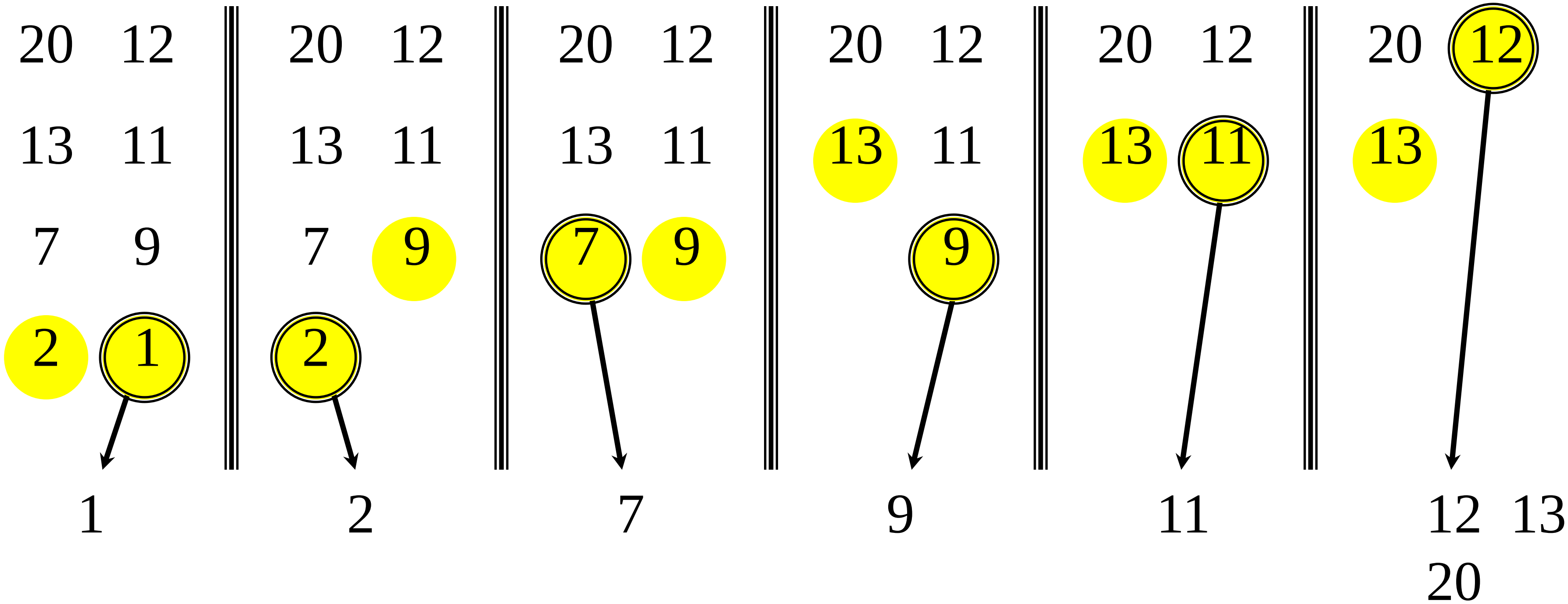
Merging two sorted arrays



Merging two sorted arrays

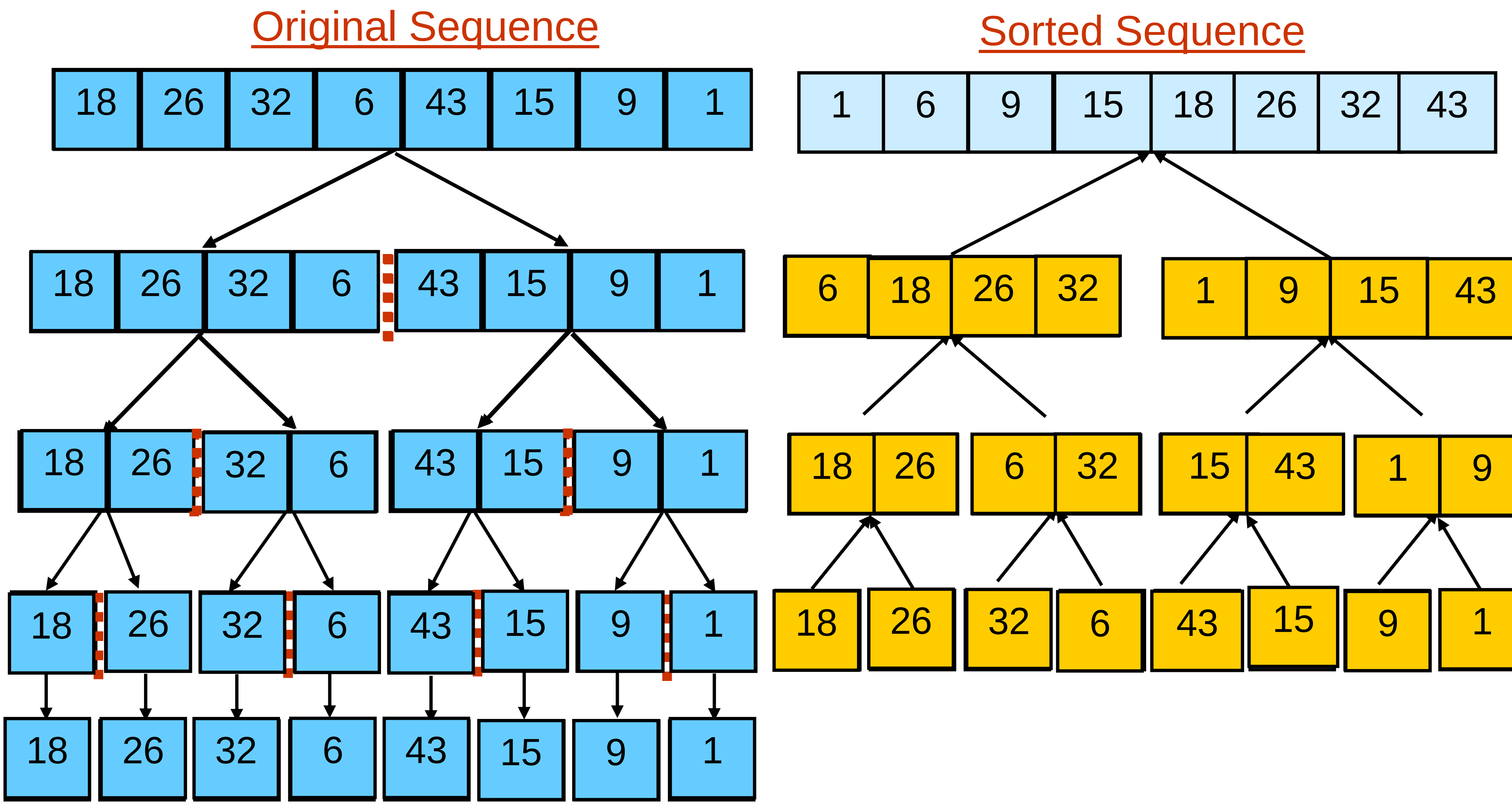


Merging two sorted arrays



Time = O(n) to merge a total of n elements (linear time).

Merge Sort – Example



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

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

- 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.
- Week 2 provides several ways 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.

$$
\begin{array}{ccc}
 & cn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$

Recursion tree

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

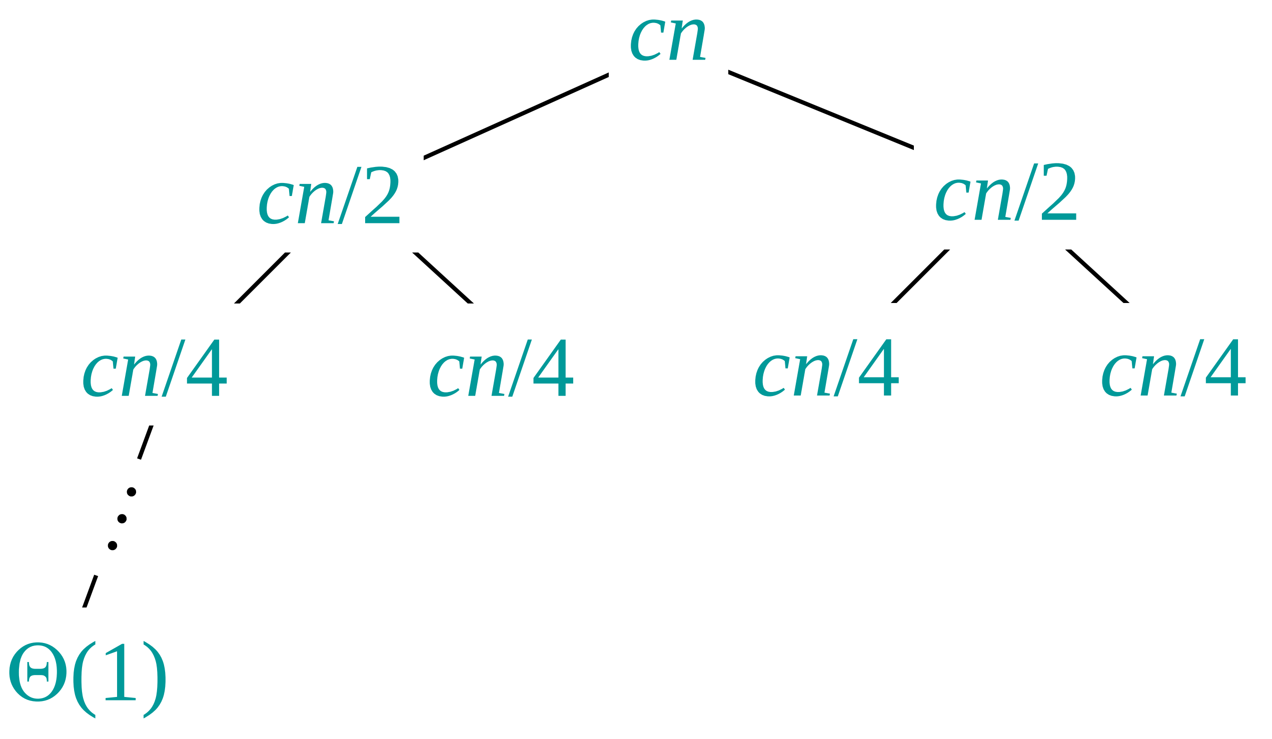
$$cn$$

$$cn/2 \qquad\qquad cn/2$$

$$T(n/4) \qquad T(n/4) \qquad T(n/4) \qquad T(n/4)$$

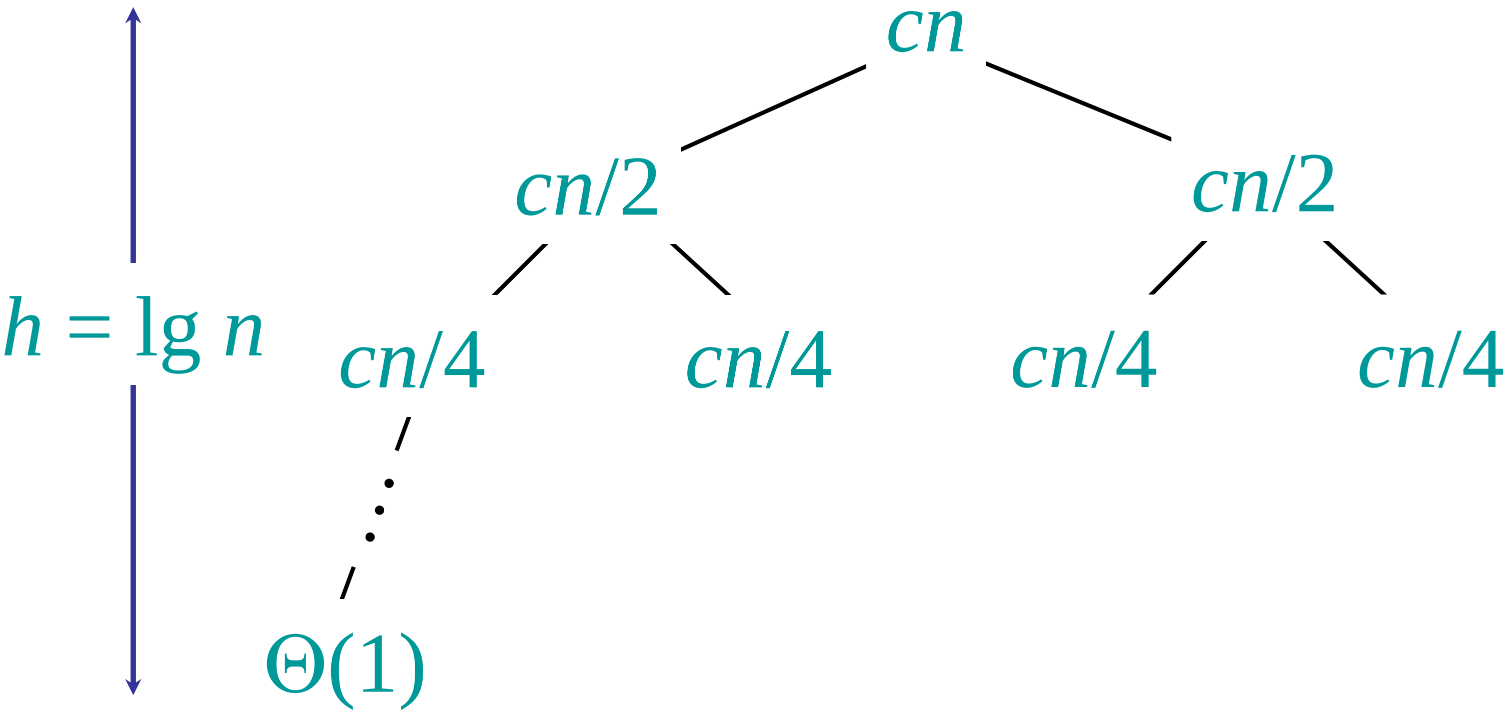
Recursion tree

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



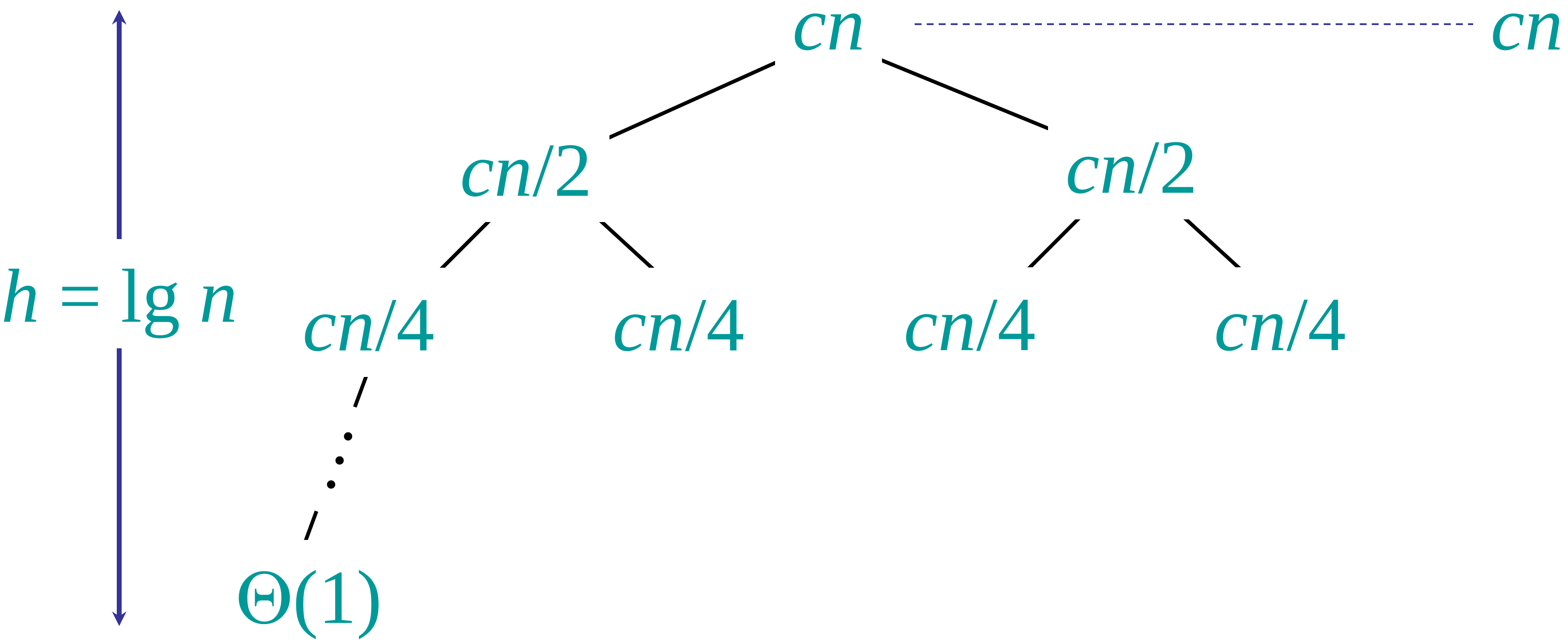
Recursion tree

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



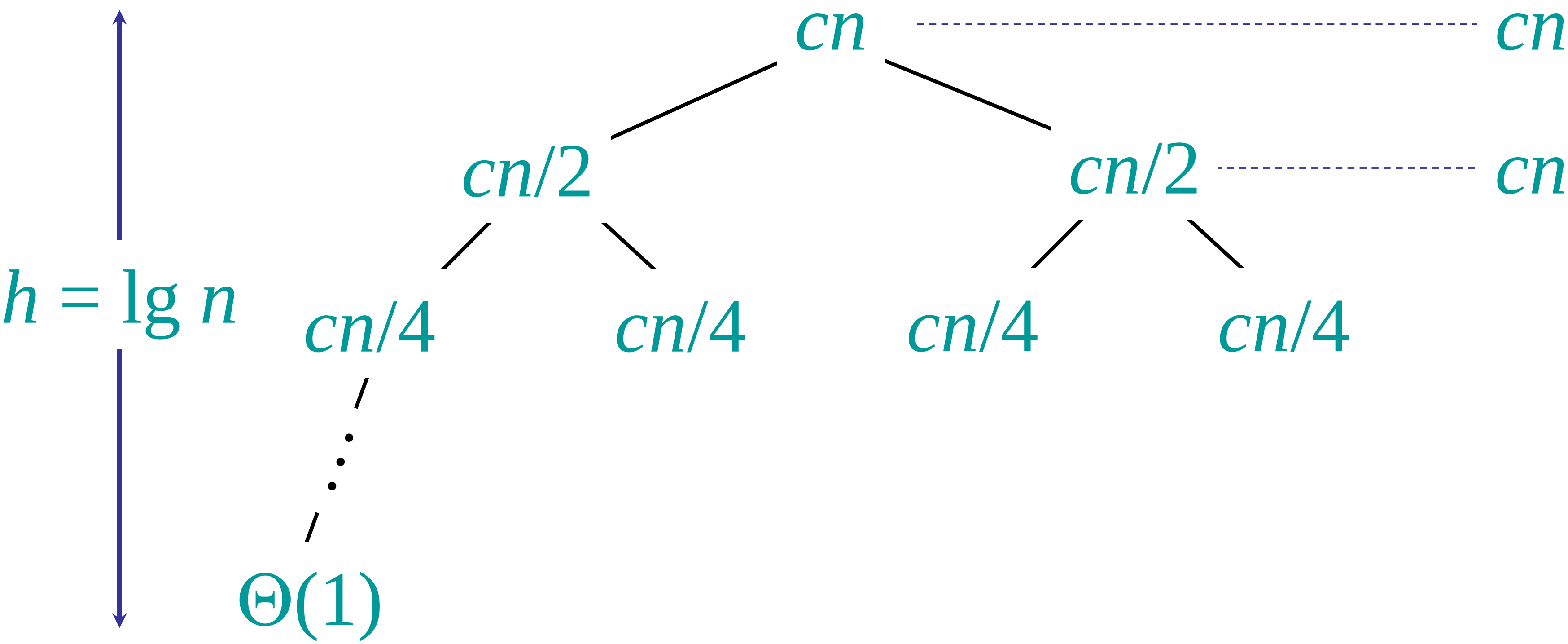
Recursion tree

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



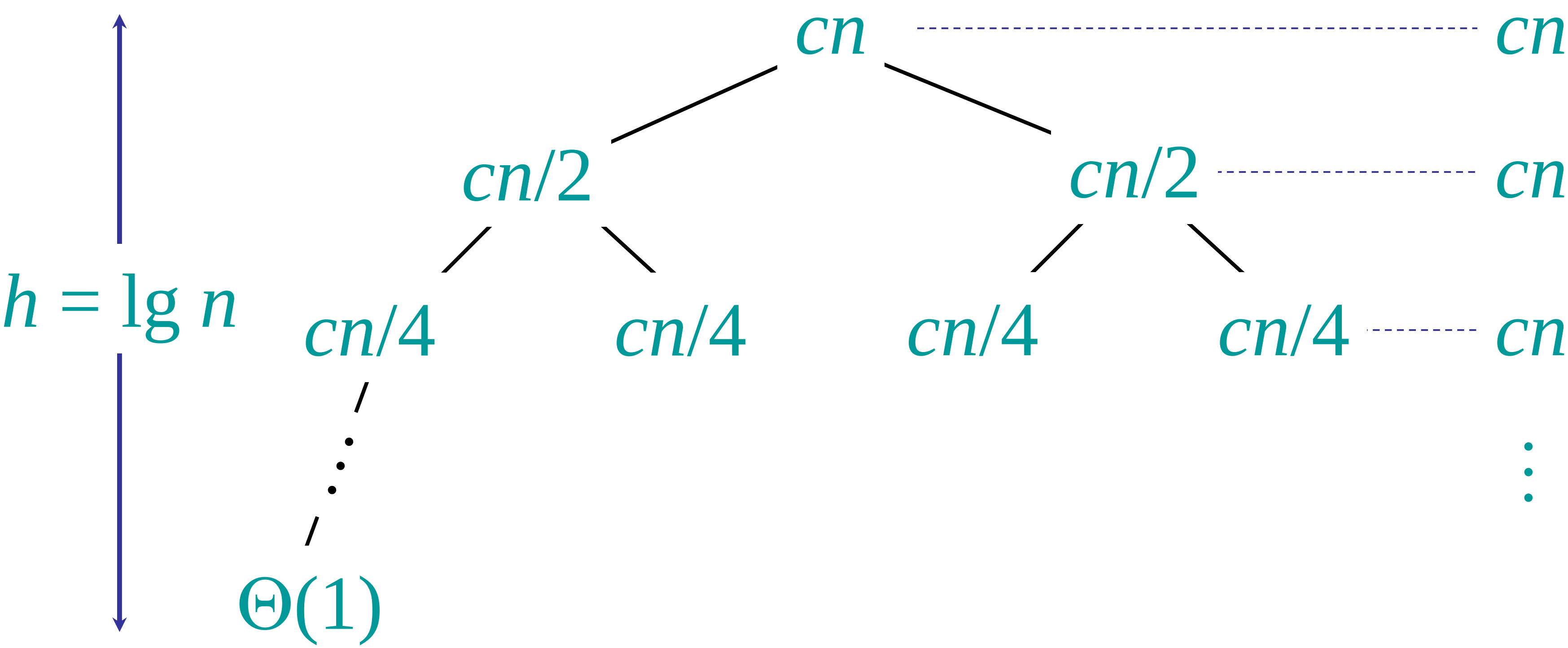
Recursion tree

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



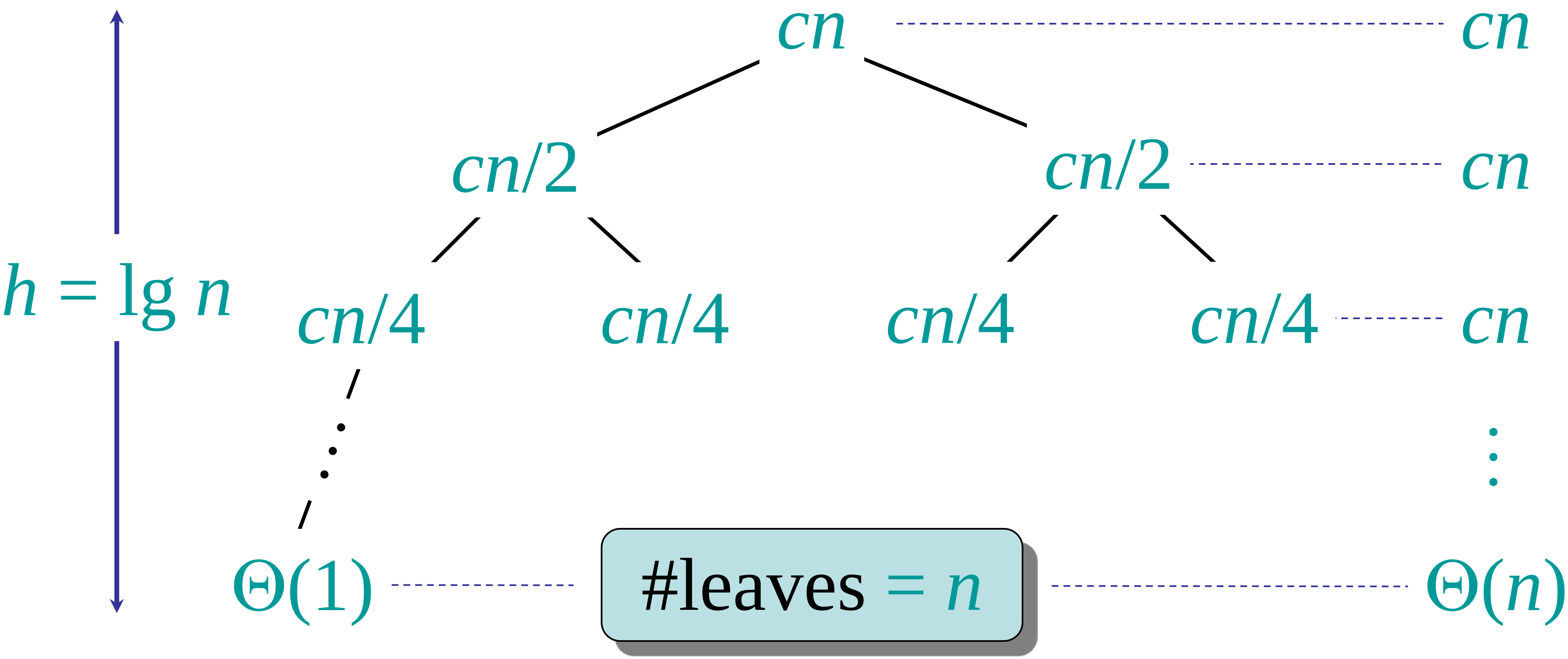
Recursion tree

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



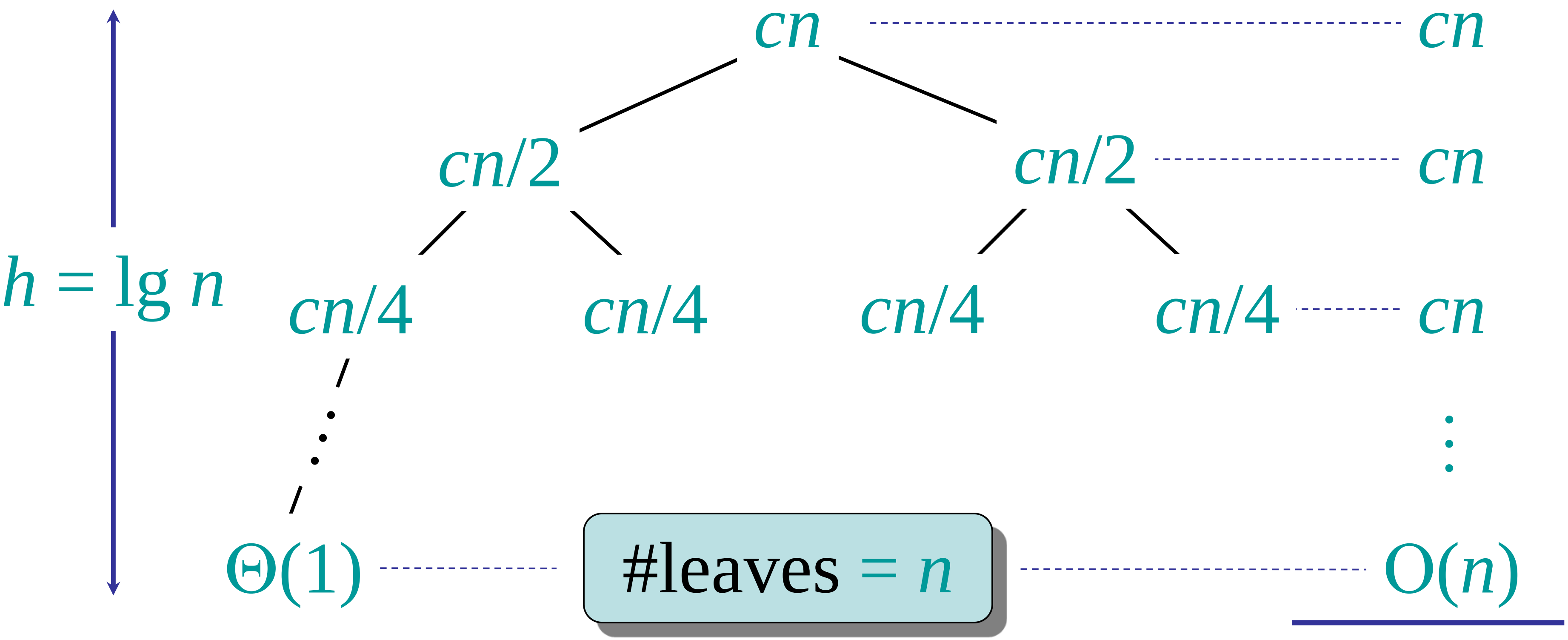
Recursion tree

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.



Total $= O(n \lg n)$

Conclusions

- $O(n \lg n)$ grows more slowly than $O(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- *Nearly Sorted??*

A tighter bound

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

```
Bubblesort( array A)
    for i from 1 to n
        for j from 0 to n - 1
            if A[j] > A[j + 1]
                swap( A[j], A[j + 1] )
```

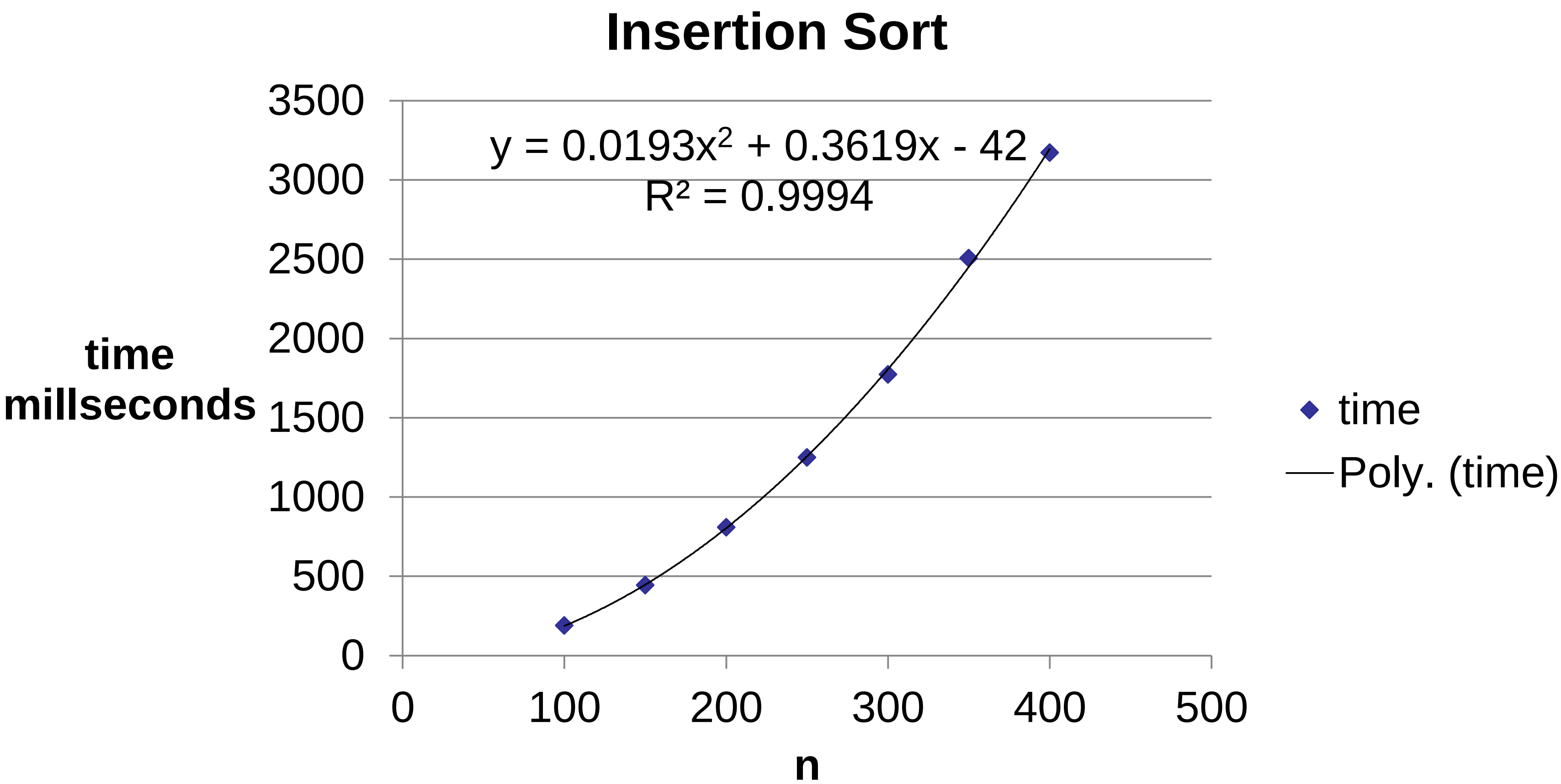
- Worst Case : $O(n^2)$
- Average Case: $O(n^2)$
- Best Case : $O(n^2)$

Benchmarking

- Algorithmic analysis is the first and best way, but not the final word
- What if two algorithms are both of the same complexity?
- Example: bubble sort and insertion sort are both $O(n^2)$
  - So, which one is the "faster" algorithm?
  - Benchmarking: run both algorithms on the same machine
  - Often indicates the constant multipliers and other "ignored" components
  - Still, different implementations of the same algorithm often exhibit different execution times – due to changes in the constant multiplier or other factors (such as adding an early exit to bubble sort)

Experimental Analysis

Implement the algorithm & Collect running time data



Experimental Analysis

Use for prediction

$$T(n) = 0.0193n^2 + 0.3619n - 42$$

Time in milliseconds a function of n.

Predict time for a list of 10,000 elements.

T(10,000) = 1933568 milliseconds

About 32 minutes

Problem: Sorting

Must be correct

Design Algorithms: Merge Sort | Insertion Sort

Asymptotic analysis

Theoretical Running Time: $O(n\log n)$ | $O(n^2)$

Implement, run, collect data

Experimental Running Time: $T(n) = 0.3n\log n + 0.01$ | $T(n) = 0.06n2 + 0.002n + 0.1$

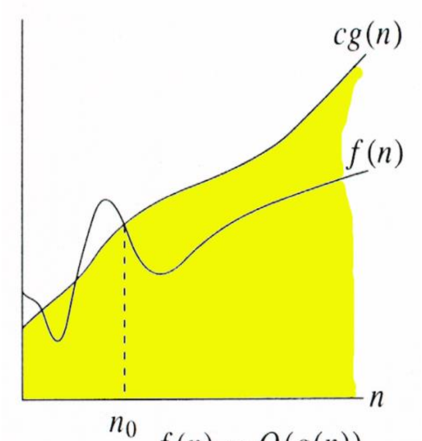
Problem Complexity Class: Polynomial P

Comparing functions

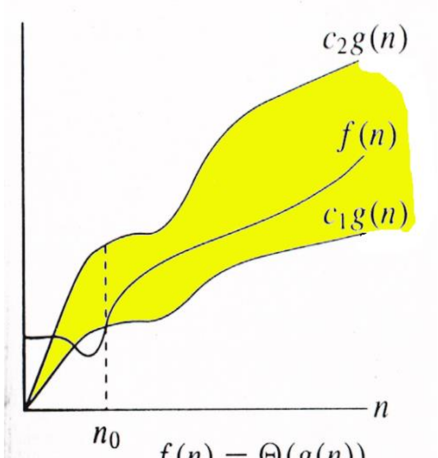
- f(n) = .2nlogn vs g(n) = .0.01n^2

- f(n) = O(g(n))

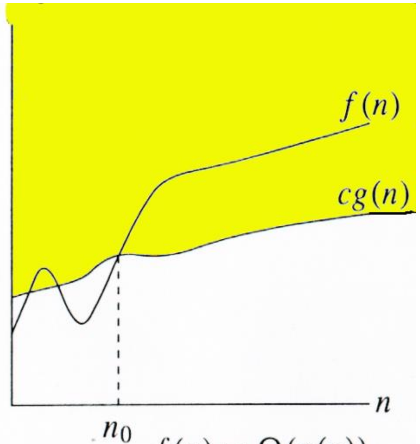
Relations Between Θ, *O*, Ω



$f(n) = O(g(n))$



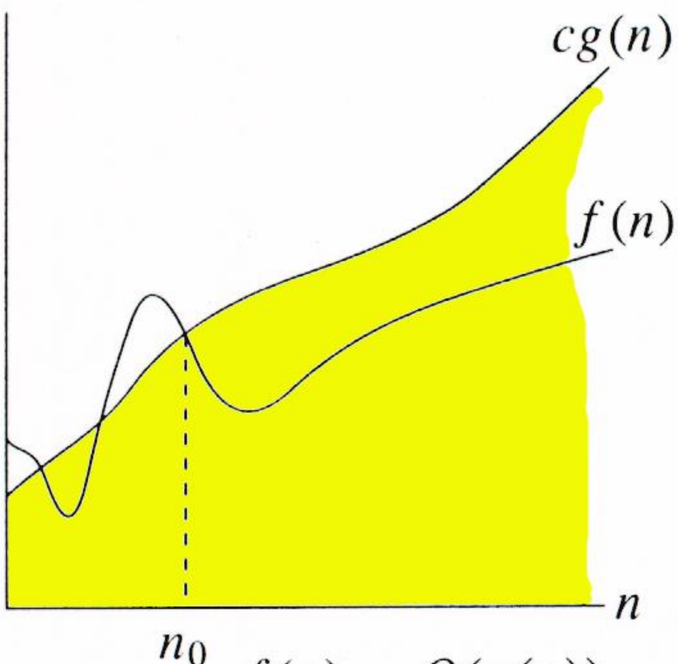
$f(n) = \Theta(g(n))$



$f(n) = \Omega(g(n))$

O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



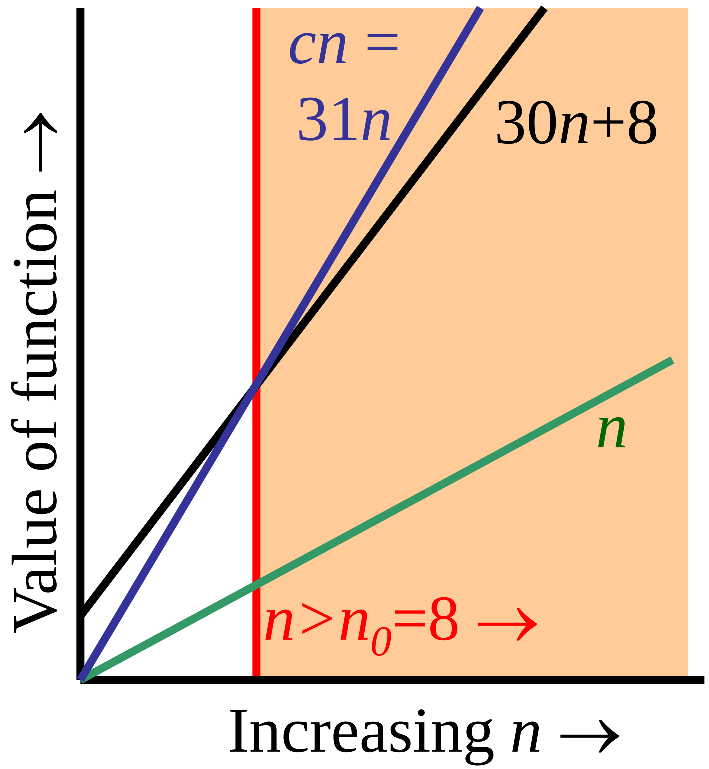
$f(n) = O(g(n))$

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.

$30n+8$ is $O(n)$



Examples

$2n^2 = O(n^3)$: $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 2$ and $n_0 = 1$

$n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c = 1$ and $n_0 = 1$

$1000n^2+1000n = O(n^2)$:

$1000n^2+1000n \le 1000n^2+ n^2 = 1001n^2 \Rightarrow c=1001$ and $n_0 = 1000$

$n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

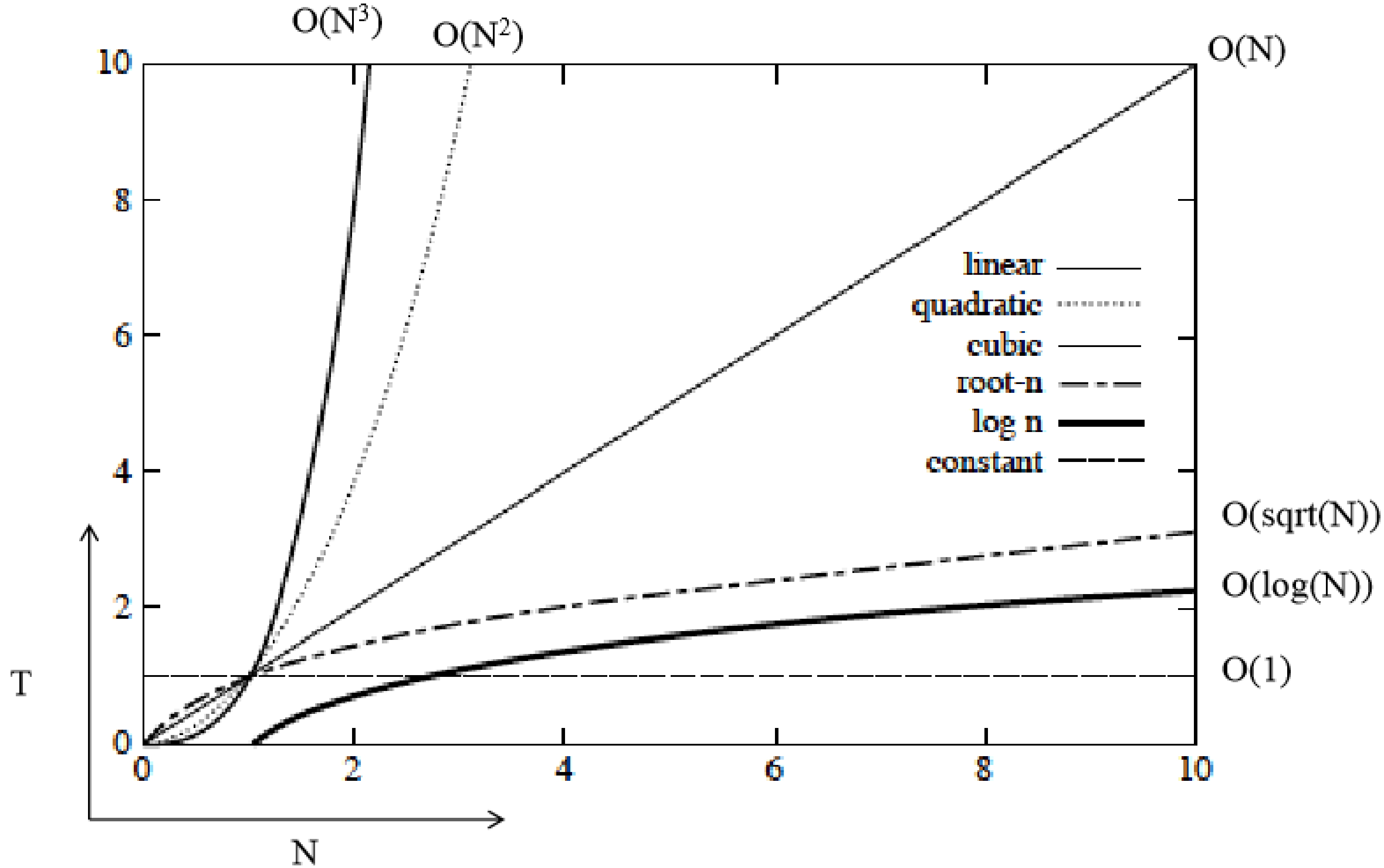
More Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$

constants

- 10 is $O(1)$
- 1273 is $O(1)$

Orders of Growth



A polynomial of degree k is $O(n^k)$

Recall: $f(n)$ *is* $O(g(n))$ *if there exist positive constants c and* n_0 *such that* $f(n) \le c \cdot g(n)$ *for all* $n \ge n_0$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

Let $a_i = | b_i |$

$$f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

$$f(n) \le n^k \left(a_k + a_{k-1} \frac{n^{k-1}}{n^k} + \ldots + a_1 \frac{n^1}{n^k} + a_0 \frac{1}{n^k} \right)$$

$$f(n) \le \quad n^k \sum a_i \frac{n^i}{n^k} \le n^k \sum a_i$$

$$let \;\; c = \sum a_i$$

$$f(n) \le \quad cn^k \quad \text{for n} \ge 1$$

Therefore all polynomial functions $f(n)$ of degree k are $O(n^k)$.

Big Oh Classes

• Constant	$O(1)$
• Logarithmic	$O(\log(n))$
• Linear	$O(n)$
• Quadratic	$O(n^2)$
• Cubic	$O(n^3)$
• Polynominal	$O(n^k)$ for any $k>0$
• Exponential	$O(k^n)$, where $k>1$
• Factorial	$O(n!)$

Rank the following functions in increasing order of growth

lg(2^n), 1000, sqrt(n), $3n^2$, n!, logn, 3^n

a) 1000, lg(2^n), sqrt(n), $3n^2$, n!, logn, 3^n

b) 1000, lg(2^n), sqrt(n), $3n^2$, logn, 3^n, n!

c) 1000, logn, lg(2^n), sqrt(n), $3n^2$, n!, 3^n

d) 1000, logn, sqrt(n), lg(2^n), $3n^2$, 3^n, n!

e) None of the above

A Simple Code Example

- Consider summing an array of n integers.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

Executes in constant time $\boldsymbol{c_1}$ (independent of n)

Executes in $\boldsymbol{c_2 \cdot n}$ time for for some constant $\boldsymbol{c_2}$

Executes in constant time $\boldsymbol{c_3}$ (independent of n)

- Total running time: $c_1 + c_2 n + c_3$
  - But the constants c_1, c_2, c_3 depend on hardware, compiler, etc.
- What is the big-Oh runtime? (big-Oh ignores factors)

O(n) also known as <u>linear time</u>

A Simple Example

- Consider summing an array of n integers.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

O(n)

What does this mean in practice?

```
for(s = 0; s < N; s++)
        sum = sum + 1;
```

$O(n)$

```
for(i = 0; i < N; i++)
  for(j =0; j < N; j++)
        printf( …)
```

$O(n^2)$

Total = $O(n) + O(n^2) = O(n + n^2) = O(n^2)$

Code Example

```
int isPrime (int n) {
  for (int i = 2; i * i <= n; i++) {
    if (0 == n % i) return 0;
  }
  return 1; /* 1 is true */
}
```

Question: What is the "Big-Oh" running time in terms of n?

a) O(n)
b) $O(n^2)$
c) O(lgn)
d) $O(\sqrt{n})$
e) O(nlgn)

Trouble with Big-Oh

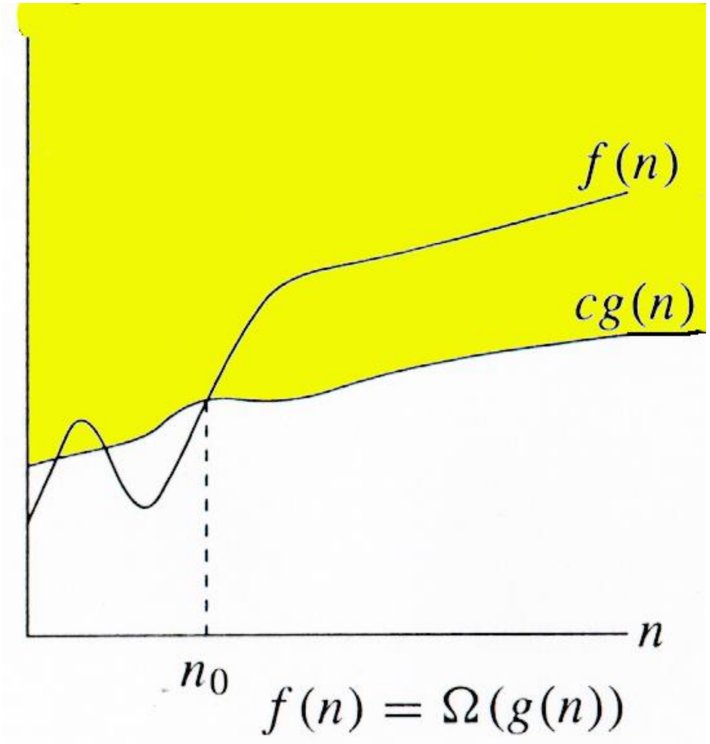
Just an upper bound. Factually true but practically meaningless.

- 3n is $O(n^2)$
- 3n is $O(n^4)$
- 3n is $O(n)$

Many times only Big-Oh is reported but it is assumed a “tight” upper bound.

Omega Ω -notation

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



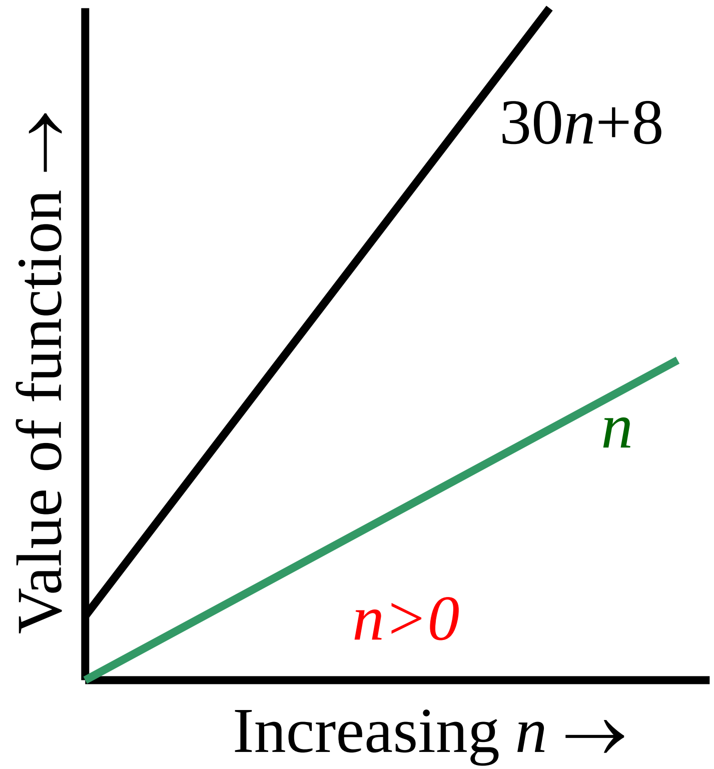
$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Omega Graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).

$30n+8$ is $\Omega(n)$



Questions

- $5n^2 = \Omega(n)$

  $\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \ne \Omega(n^2)$

  $\exists\ c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

  $100n + 5 \le 100n + 5n\ (\forall\ n \ge 1) = 105n$

  $cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

  Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

Property of Big-Oh and Omega

If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$

By definition of Big-Oh

$$f(n) \leq cg(n) \text{ for all } n \geq n_0 \text{ for some } n_0, c > 0.$$

Dividing by c yields

$$\frac{1}{c} f(n) \leq g(n) \text{ or } c_2 f(n) \leq g(n) \text{ where } c_2 = \frac{1}{c} \geq 0$$

If we use the same n_0, this implies that $g(n) = \Omega(f(n))$.

A non-negative polynomial of degree *k* is $\Omega(n^k)$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

$$f(n) = n^k \left(b_k + b_{k-1}\frac{n^{k-1}}{n^k} + \ldots + b_1 \frac{n^1}{n^k} + b_0 \frac{1}{n^k} \right)$$

$$f(n) = n^k \left(b_k + \frac{b_{k-1}}{n^1} + \ldots + \frac{b_1}{n^{k-1}} + \frac{b_0}{n^k} \right)$$

For large n's the fractions go to zero, so if we set n_0 large enough we can ignore all terms except b_k. Thus a value for n_0 must exist.

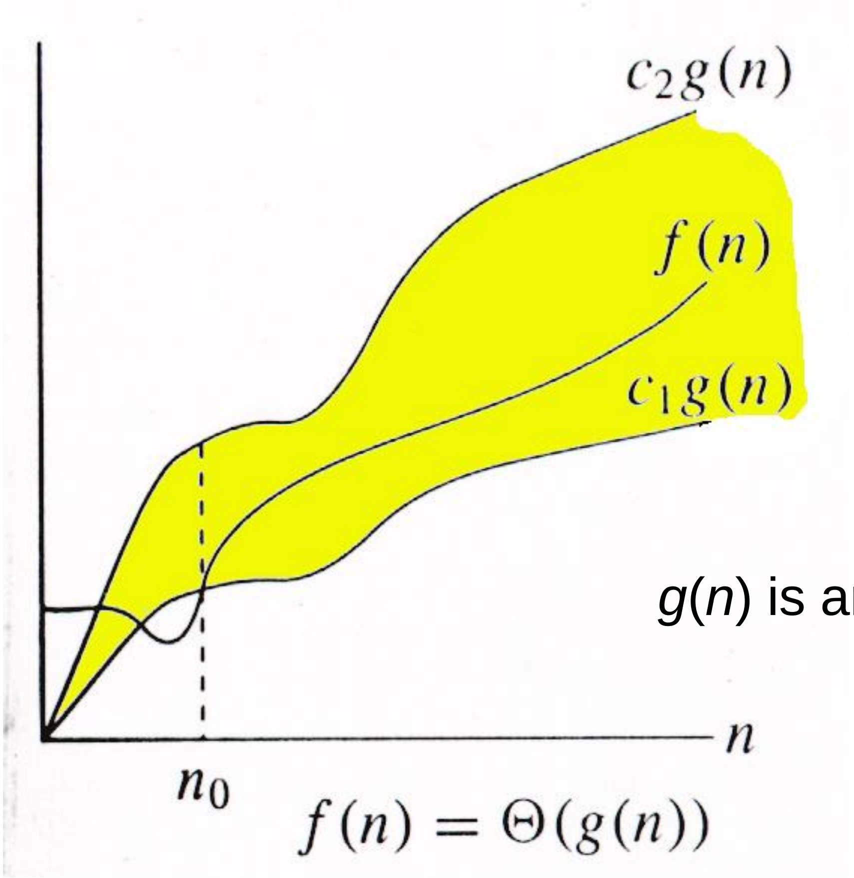
$$let\ \ c = \frac{\mathrm{b_k}}{2} \qquad\qquad cn^k \le f(\mathrm{n}) \text{ for } \mathrm{n} \ge \mathrm{n_0}$$

$$\frac{\mathrm{b_k}}{2} n^k \le \mathrm{b_k} n^k \text{ for } \mathrm{n} \ge \mathrm{n_0}$$

Therefore all polynomial functions *f(n)* of degree k are $\Omega(n^k)$.

Θ-notation

$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



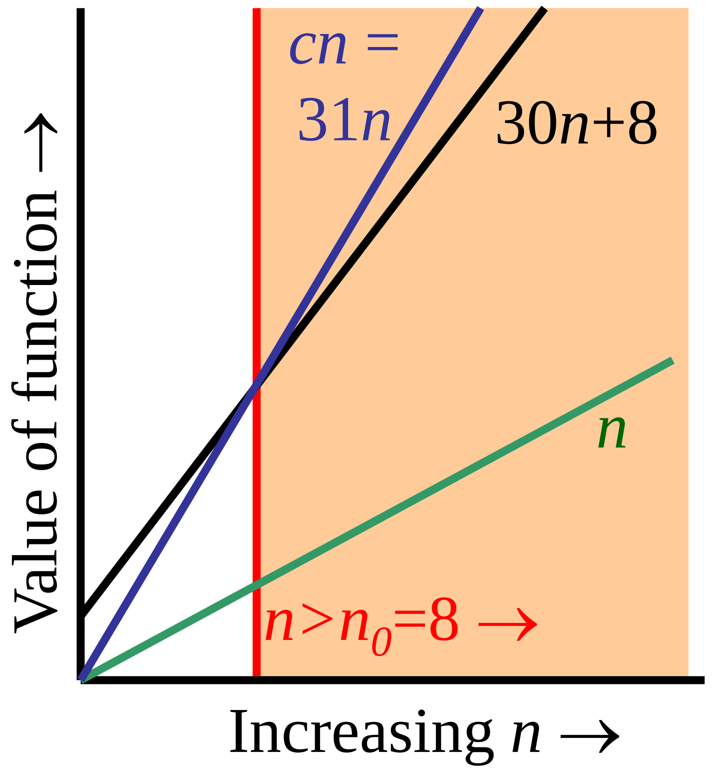
$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Big-Theta example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.

$$30n+8 \in \Theta(n)$$



A non-negative polynomial of degree *k* is $\Theta(n^k)$

Proof: From previous Big-Oh and Omega

$f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$ *is* $\Theta(n^k)$

Short-cut:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

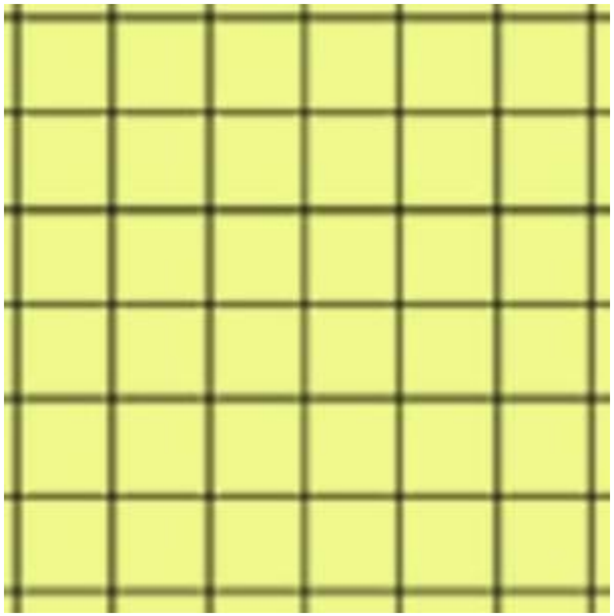
Iterative Algorithm Analysis

```
for (i=1; i<=n*n; i++)
  for (j=0; j<i; j++)
    sum++;
```

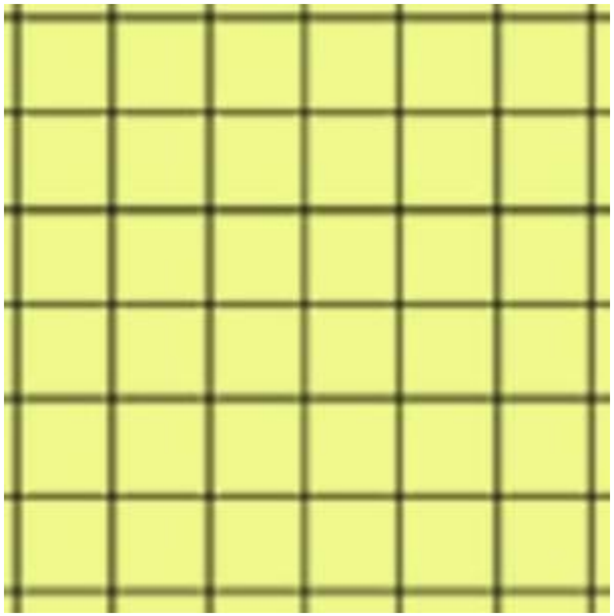
Determine the running time

a) $\Theta(n)$
b) $\Theta(n^2)$
c) $\Theta(\lg n)$
d) $\Theta(n^4)$
e) $\Theta(n \lg n)$

$$\sum_{k=1}^{n} k = 1 + 2 + ... + n = \frac{n(n+1)}{2}$$



$$\sum_{k=1}^{n} k = 1 + 2 + ... + n = \frac{n(n+1)}{2}$$



Iterative Algorithm Analysis

```
for (i=1; i<=n*n; i++)
  for (j=0; j<i; j++)
    sum++;
```

Determine the running time

a) $\Theta(n)$
b) $\Theta(n^2)$
c) $\Theta(lgn)$
d) $\Theta(n^4)$
e) $\Theta(nlgn)$

Iterative Algorithm Analysis

$i \leftarrow n$
while $(i > 1)$ **do**
  $i \leftarrow \lfloor i/2 \rfloor$
  $z \leftarrow z + 1$

Determine the running time

a) $\Theta(n)$
b) $\Theta(n^2)$
c) $\Theta(\lg n)$
d) $\Theta(n^4)$
e) $\Theta(n \lg n)$

Iterative Algorithm Analysis

$i \leftarrow n$
while $(i > 1)$ **do**
  $i \leftarrow \lfloor i/2 \rfloor$
  $z \leftarrow z + 1$

Rate of Growth as limits

The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

$$\lim_{n \to \infty} \frac{n^4 + 100n^2 + 10n + 50}{n^4} = 1$$

That is we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Mathematics a **tilde symbol** (~) indicating equivalency or similarity between two values.

Limit Method: The Process

Say we have functions f(n) and g(n). We set up a limit quotient between f and g as follows

$$\text{If } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & then\ f(n)\ is\ O(g(n)) \\ c > 0 & then\ f(n)\ is\ \Theta(g(n)) \\ \infty & then\ f(n)\ is\ \Omega(g(n)) \end{cases}$$

- The above can be proven using calculus, but for our purposes, the limit method is sufficient for showing asymptotic inclusions
- Always try to look for algebraic simplifications first
- If f and g both diverge or converge on zero or infinity, then you need to apply the l'Hôpital Rule

L’Hôpital Rule

Theorem (L’Hôpital Rule):

- Let f and g be two functions,
- if the limit between the quotient f(n)/g(n) exists,
- Then, it is equal to the limit of the derivative of the numerator and the denominator

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

Question

If $f(n) = \log n$ and $g(n) = \lg n$ then $g(n) = \Theta(f(n))$

a) True
b) False

Limit Method: Question

- Example: Let $f(n) = 2^n$, $g(n) = 3^n$. Determine a tight inclusion of the form $f(n) = \Delta\,(g(n))$

What is your intuition in this case?

a) Big-O
b) Theta
c) Omega

Limit Method: Question

Example: Let $f(n) = \log_2 n$, $g(n) = \log_3 n^2$. Determine a tight inclusion of the form

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

What is your intuition in this case?

a) Big-O
b) Theta
c) Omega

Using Wolfram Alpha

https://www.wolframalpha.com/

lim(x->inf)(2^x/3^x)

Important Result

- All logs have the same asymptotic growth rate no what the base is.
- In many CS algorithms the base is 2.
- But we get sloppy since lg(n) is $\Theta(\log n)$

Properties

Theorem:

$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- 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 $g(n) = \Omega(f(n))$

Let f, g and h be asymptotically positive functions. Prove or disprove each of the following conjectures.

Transitivity f(n) = Θ(g(n)) and g(n) = Θ(h(n)) $\Rightarrow$ f(n) = Θ(h(n))

1. By definition f(n) = Θ(g(n)) implies there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

2. By definition g(n) = Θ(h(n)) implies there exist positive constants c_3, c_4, and n_1 such that $0 \le c_3 h(n) \le g(n) \le c_4 h(n)$ for all $n \ge n_1$

3. Show f(n) = Θ(h(n)) that is there exist positive constants c_5, c_6, and n_2 such that $0 \le c_5 h(n) \le f(n) \le c_6 h(n)$ for all $n \ge n_2$

By combining 1 and 2: $c_1 c_3 h(n) \le c_1 g(n) \le f(n)$ let $c_5 = c_1 c_3$ so $c_5 h(n) \le f(n)$

Again from 1 and 2: $f(n) \le c_2 g(n) \le c_2 c_4 h(n)$ let $c_6 = c_2 c_4$ so $f(n) \le c_6 h(n)$

So $c_5 h(n) \le f(n) \le c_6 h(n)$ for all $n \ge n_2$

And let $n_2 = \max\{n_0, n_1\}$

Let f, g and h be asymptotically positive functions. Prove or disprove each of the following conjectures.

If $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n)+h(n) = O(g(n))$?

a) True
b) False

Let f, g and h be asymptotically positive functions. Prove or disprove each of the following conjectures.

If $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n)*h(n) = O(g(n))$?

Let f, g and h be asymptotically positive functions. Prove or disprove each of the following conjectures.

If $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n)*h(n) = O(g(n))$?

a) True
b) False

Constant factors and runtimes

Suppose we have two algorithms with exact running times of:

Algorithm 1		Algorithm 2
$1{,}000{,}000 \cdot n$	versus	$2 \cdot n^2$

Is it reasonable to say that runtime of Algorithm 2 is "worse" (slower) than Algorithm 1?

NO for small values of n Algorithm 2 is better

YES for large values of n and asymptotic analysis

Common Summations

- Arithmetic series: $$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$
- Geometric series: $$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \left(x \neq 1\right)$$
  - Special case: $|x| < 1$: $$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$
- Harmonic series: $$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$
- Other important formulas: $$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$