EECE 2560:
Fundamentals of Engineering Algorithms

Algorithm Efficiency

What Is a Good Solution? (1 of 2)

- A program incurs a real and tangible cost.
  - Computing time
  - Memory required
  - Difficulties encountered by users
  - Consequences of incorrect actions by program
- A solution is good if …
  - The total cost incurs over all phases of its life … is minimal

What Is a Good Solution? (2 of 2)

- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency

- Be concerned with efficiency when
  - Developing underlying algorithm
  - Choice of objects and design of interaction between those objects

Measuring Efficiency of Algorithms (1 of 3)

- Important because
  - Choice of algorithm has significant impact

- Examples
  - Responsive word processors
  - Grocery checkout systems
  - Automatic teller machines
  - Video machines
  - Life support systems

Measuring Efficiency of Algorithms (2 of 3)

- Analysis of algorithms
  - The area of computer science that provides tools for contrasting efficiency of different algorithms
  - Comparison of algorithms should focus on significant differences in efficiency
  - We consider comparisons of **algorithms**, not programs

Measuring Efficiency of Algorithms (3 of 3)

- Difficulties with comparing programs (instead of algorithms)
  - How are the algorithms coded
  - What computer will be used
  - What data should the program use

- Algorithm analysis should be independent of
  - Specific implementations, computers, and data

Program Execution Time

- Definitions:
  - IC: Number of CPU instructions in the executable program.
  - CPI: CPU average clock cycles per instruction.
  - T: Clock cycle time, F: CPU frequency (Hz) → F = 1 / T.

Program Execution Time = IC × CPI × T = (IC × CPI) / F

- Program execution time depends on
  - Algorithm affects the IC.
  - Programming language IC and CPI
  - Compiler affects the IC and CPI
  - CPU architecture affects the IC , CPI, and T (complex instruction needs more time if a single cycle architecture is used)
  - Microarchitecture (Hardware implementation) affects CPI and T
  - Semiconductor technology affects T

The Execution Time of Algorithms

- An algorithm's execution time is related to number of operations it requires.
  - Can we just use a stopwatch?
- Example: Towers of Hanoi
  - Solution for n disks required $2^n - 1$ moves
  - If each move requires time m units
  - Solution requires $(2^n - 1) \times m$ time units

Algorithm Growth Rates (1 of 3)

- Measure an algorithm's time requirement as function of problem size.
- Most important thing to learn
  - How quickly algorithm's time requirement grows as a function of problem size.

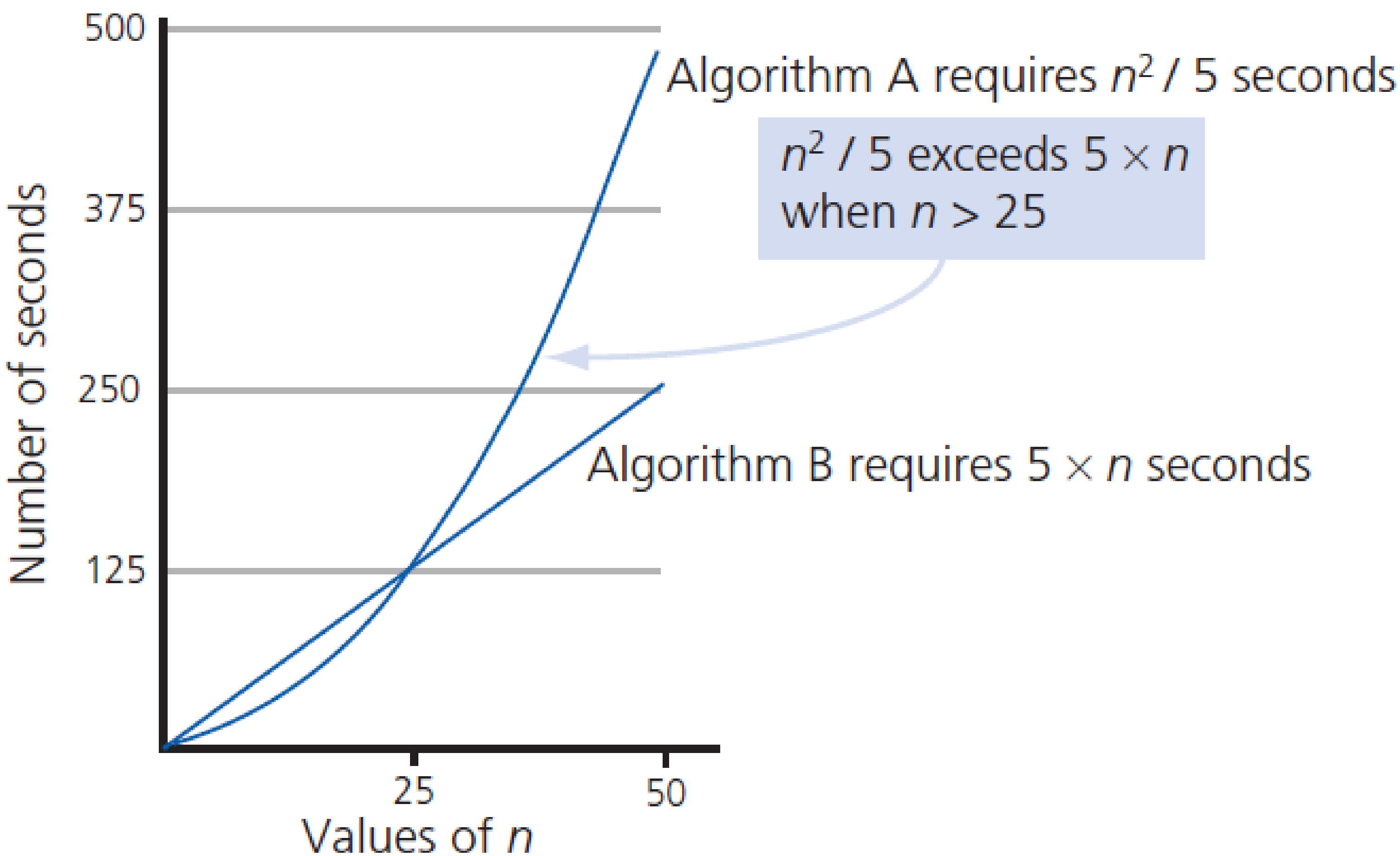
Algorithm *A* requires time proportional to n^2

Algorithm *B* requires time proportional to n

- Demonstrates contrast in growth rates.

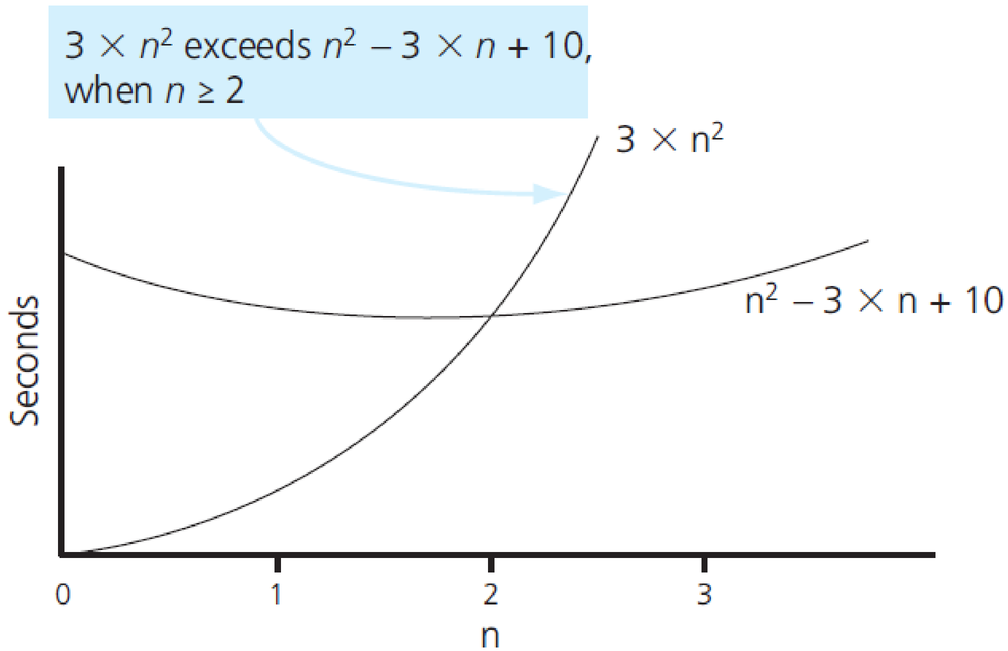
Algorithm Growth Rates (2 of 3)

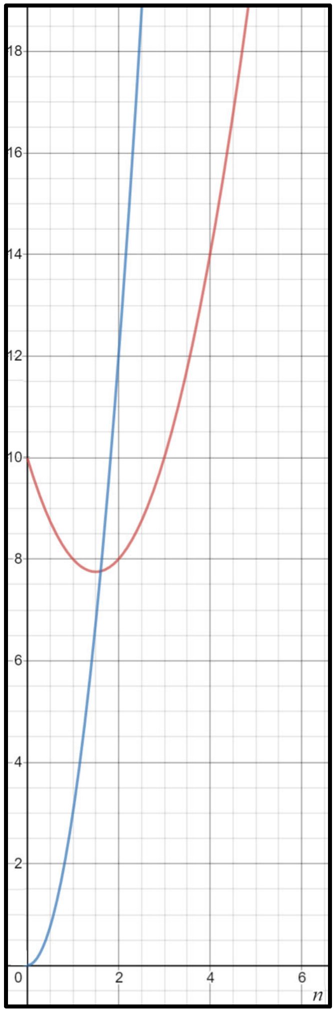
Time requirements as a function of problem size *n* :



Algorithm Growth Rates (3 of 3)

- The graphs of $f_1 = 3 \times n^2$ and $f_2 = n^2 - 3 \times n + 10$





Analysis and Big O Notation (1 of 4)

- Algorithm *A* is said to be order $f(n)$ (denoted as $O(f(n))$)
  - This is one of the ***Asymptotic Notations*** used to describe the running times of algorithms
  - Function $f(n)$ called algorithm's growth rate function

- Algorithm *A* of order denoted $O(f(n))$
  - Constants k and n_0 exist such that
    - *A* requires no more than $k \times f(n)$ time units for problem of size $n \geq n_0$

Analysis and Big O Notation (2 of 4)

- Order of growth of some common functions:

$O(1) < O(\log_2 n) < O(n) < O(n \times \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

- Big O Rules:
  - Constant factors are ignored:

    $\forall\ C > 0,\ Cn = O(n)$
  - Smaller exponents are Big O of larger exponents:

    $\forall (a < b),\ n^a = O(n^b)$
  - Any logarithm is Big O of any polynomial:

    $\forall (a, b > 0),\ \log_a n = O(n^b)$
  - Any polynomial is Big O of any exponential:

    $\forall\ (a>0,\ b>1)\ n^a = O(b^n)$
  - Lower order terms can be dropped: $n^3 + n^2 + n = O(n^3)$

Analysis and Big O Notation (3 of 4)

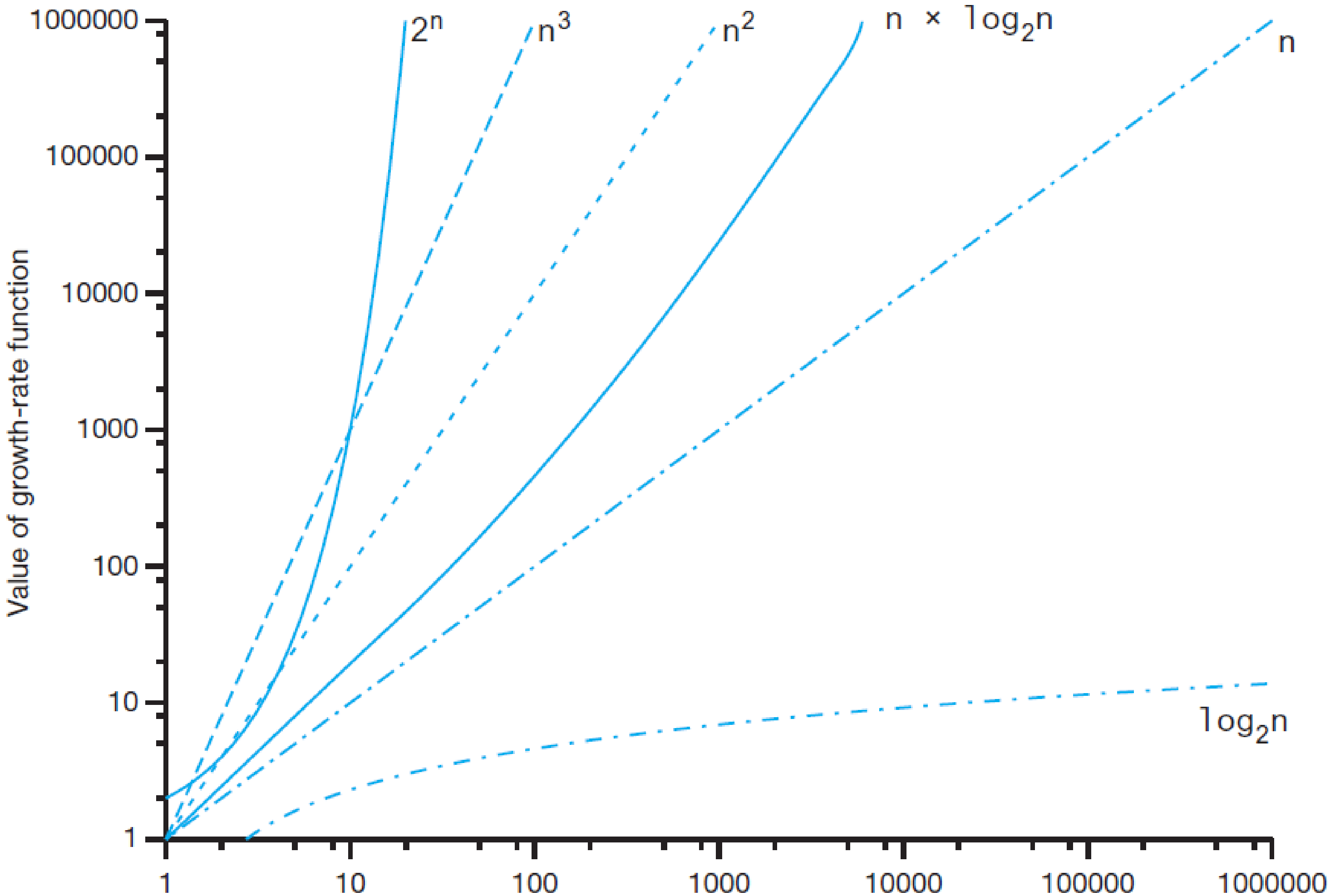
A comparison of growth-rate functions

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n \times \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

For a computer that executes 100 billion instructions per second (double the speed of an i7), a growth of 10^{20} will require more than 30 years running time.

Analysis and Big O Notation (4 of 4)

A comparison of growth-rate functions



Worst, Average, and Best Cases

Best-case analysis: (rarely)

- $T(n)$ = minimum time of algorithm on any input of size n.
- Cheat with a slow algorithm that works fast on some input!

Average-case analysis: (sometimes)

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

Worst-case analysis: (usually)

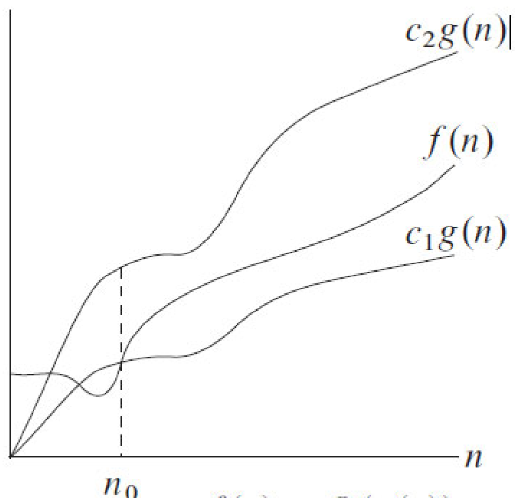
- $T(n)$ = maximum time of algorithm on any input of size n.
- Easier to calculate, thus more common

Comparing Asymptotic Notations

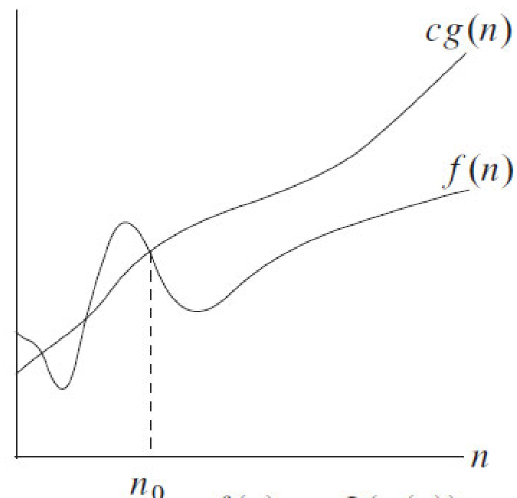
$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

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

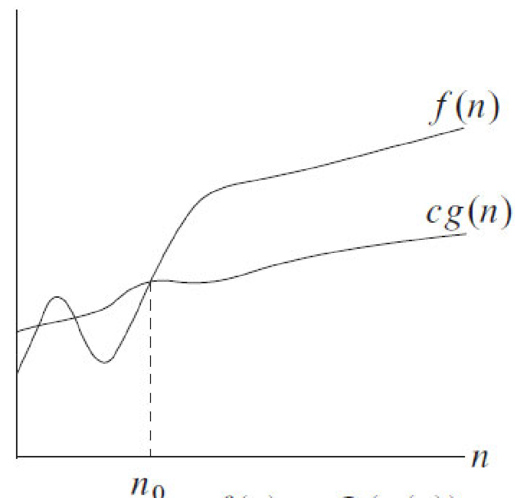
$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



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



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



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

ADT and Algorithms Efficiency

- The used ADT makes a difference
  - Array-based **getEntry** is O(1)
  - Link-based **getEntry** is O(n)

- Choosing implementation of ADT
  - Consider how frequently certain operations will occur
  - Seldom used but critical operations must also be efficient

Keeping Your Perspective

- If problem size is always small
  - Possible to ignore algorithm's efficiency

- Weigh trade-offs between
  - Algorithm's time and memory requirements

- Compare algorithms for style and efficiency

Efficiency of Searching Algorithms

- Sequential search
  - Worst case: O(*n*)
  - Average case: O(*n*)
  - Best case: O(1)

- Binary search
  - $O(\log_2 n)$ in worst case
  - At same time, maintaining array in sorted order requires overhead cost … can be substantial

Analyzing Merge Sort

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

Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter for worst case analysis.

It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting c be the larger of these times.

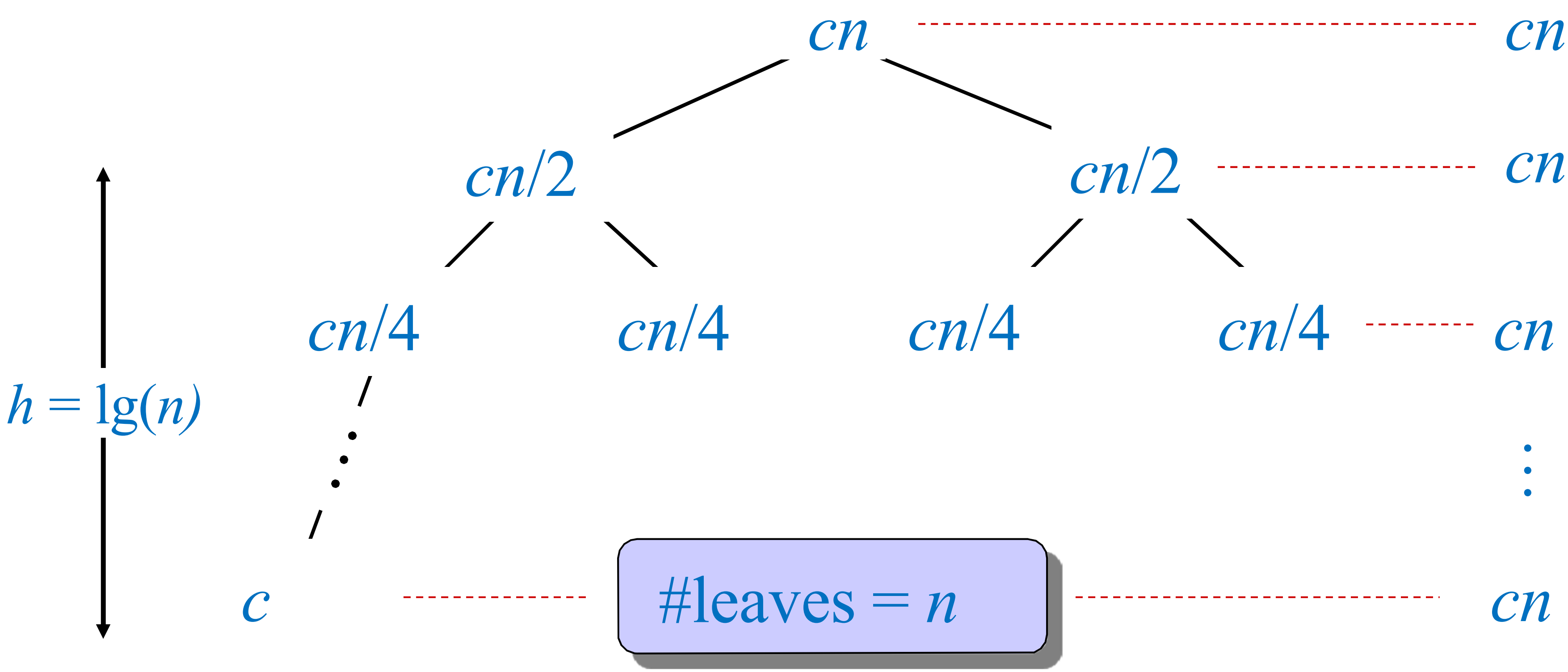
Recurrence for Merge Sort

- When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***.
- The recurrence equation for merge sort is:

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

Merge Sort Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant and assuming n is power of 2



- Total: $cn \lg n + cn$ → Worst-case running time of merge sort is $O(n \lg n)$
- *Note:* All logarithms are within constant factors of each other: $\log_b n = (\log_c n) / (\log_c b)$, which is a constant times $\log_c n$, for any base b & c
- So we can use $O(\log n)$ without specifying a base such as 2 in $\lg(n)$ or e in $\ln(n)$

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- "Programming" in this context refers to a *tabular method*, not to writing computer code.
- Dynamic programming applies when the subproblems overlap—that is, when subproblems share sub-subproblems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-subproblems.

Dynamic Programming (Cont'd)

- A dynamic-programming algorithm solves each sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time it solves each sub-subproblem.

- Dynamic Programming is a time-space tradeoff.

Dynamic Programming Example (1 of 3)

- **Problem**: Write an algorithm to calculate the number of combinations "*n* choose *k*", `C(n, k)`
- *Solution 1* - using factorials:

$$C(n, k) = \frac{n!}{k!\,(n-k)!}$$

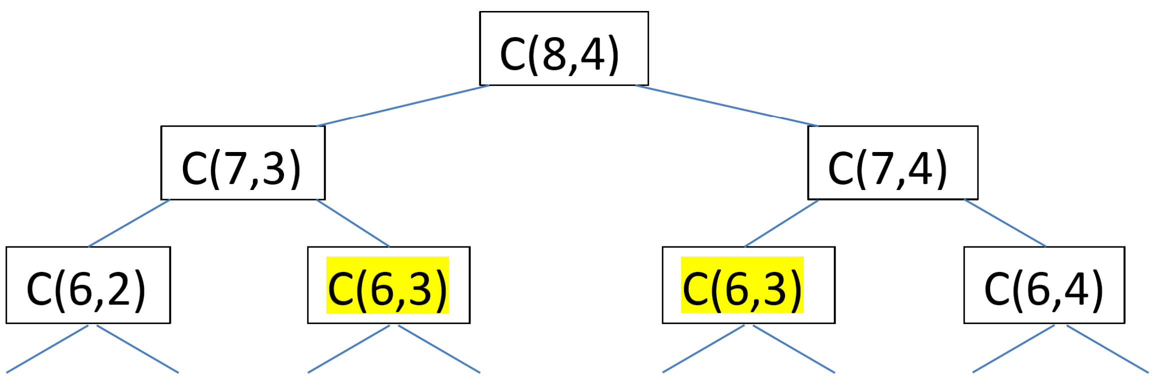
- The problem with this approach is that the factorial of a number grows very rapidly and will exceed the range of even long integer variables.

Dynamic Programming Example (2 of 3)

- *Solution 2* - using Pascal's recursive formula for combinations:

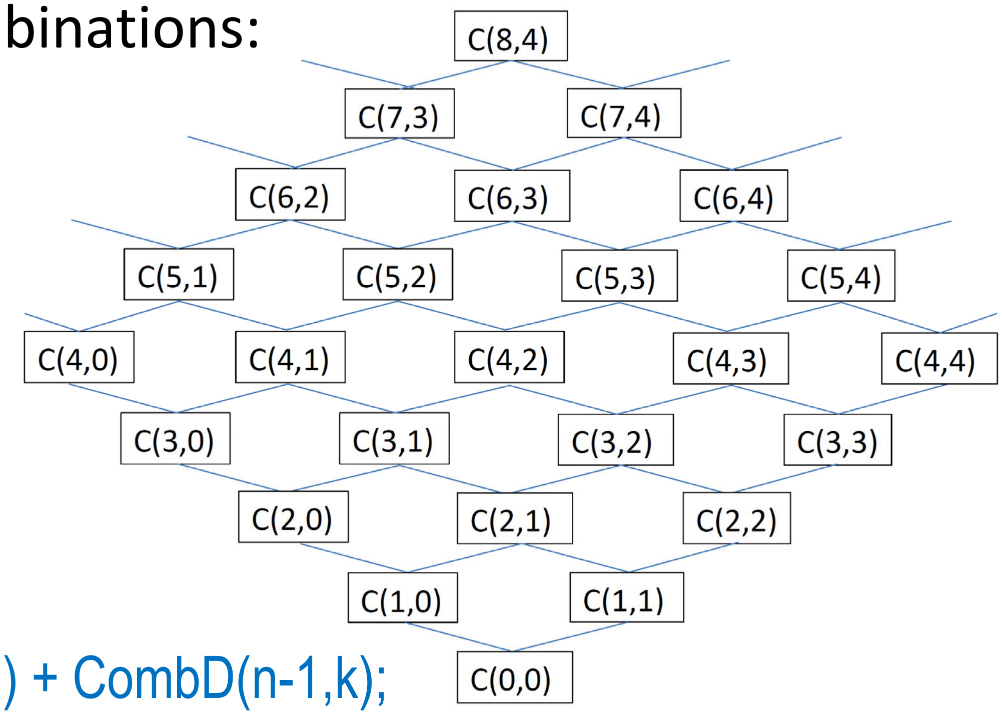
```
C(n, 0) = 1
C(n, n) = 1
C(n, k) = C(n-1, k-1) + C(n-1, k), when 0 < k < n
```

- The problem with this approach is repeating the calculations of the same subproblems resulting in an inefficient *exponential* running time as shown:



Dynamic Programming Example (3 of 3)

- *Solution 3* – using dynamic programming implementation of Pascal's triangle for combinations:




```
allocate array C[0…n][0…k]
and initialize its contents to -1

int CombD(int n, int k) {
  if (C[n][k] != -1) return C[n][k];
  if (k == 0 || k == n) C[n][k] = 1;
    else C[n][k] = CombD(n-1,k-1) + CombD(n-1,k);
  return C[n][k];
}
```

- Pros: Running time: $\Theta(nk)$ and $O(n^2)$ as $k \leq n$
- Cons: Extra space needed also of $\Theta(nk)$

Analysis of Algorithms Formulas (1 of 5)

- Properties of Logarithms
  - $\lg x = \log_2 x$
  - $\ln x = \log_e x$ (e = 2.71828)
  - $\log_a 1 = 0$
  - $\log_a a = 1$
  - $\log_a x^y = y \log_a x$
  - $\log_a xy = \log_a x + \log_a y$
  - $\log_a (x/y) = \log_a x - \log_a y$
  - $\log_a x = \log_b x / \log_b a$

Analysis of Algorithms Formulas (2 of 5)

- Combinatorics
  - Number of permutations of an *n*-element set: $P(n) = n!$
  - Number of *k*-combinations of an *n*-element set:
    $C(n, k) = n! / (k! (n-k)!)$ $(0 \le k \le n)$
  - Number of subsets of an *n*-element set: 2^n
- $n!$ is worse than exponential 2^n but it is not worse than n^n.

Analysis of Algorithms Formulas (3 of 5)

- Arithmetic Progression (Sequence):
  - $a_1, a_2, \ldots a_n$ where $a_i = a_{i-1} + d$
  - $a_n = a_1 + (n-1)\, d$
  - $a_1 + a_2 + \ldots + a_n = \dfrac{n(a_1 + a_n)}{2} = \dfrac{n}{2}[2a_1 + (n-1)d].$

- Geometric Progression (Sequence):
  - $a_1, a_2, \ldots a_n$ where $a_i = a_{i-1} \times r$
  - $a_n = a_1 \times r^{n-1}$
  - $a_1 + a_2 + \ldots + a_n = \dfrac{a_1(1 - r^n)}{1 - r}$

Analysis of Algorithms Formulas (4 of 5)

- Important Summation Formulas:

1. $\sum_{i=\ell}^{u} 1 = 1 + 1 + \ldots + 1 = u - \ell + 1.\ \sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \dfrac{n(n+1)}{2} \approx n^2 / 2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \dfrac{n(n+1)(2n+1)}{6} \approx n^3 / 3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \ldots + n^k = \dfrac{n^{k+1}}{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \ldots + a^n = \dfrac{a^{n+1} - 1}{a - 1}\ (a \neq 1);\ \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1\times 2 + 2\times 2^2 + \ldots\ n\times 2^n = (n - 1)2^{n+1} + 2;\ \sum_{i=1}^{n} i2^{i-1} = (n - 1)2^n + 1$

Analysis of Algorithms Formulas (5 of 5)

- Floor and Ceiling Formulas:
  - The floor of a real number *x*, denoted $\lfloor x \rfloor$, is defined as the greatest integer less than or equal *x*
    - (e.g., $\lfloor 3.8 \rfloor = 3$, $\lfloor -3.8 \rfloor = -4$, $\lfloor 3 \rfloor = 3$).
  - The ceiling of a real number *x*, denoted $\lceil x \rceil$, is defined as the smallest integer greater than or equal *x*
    - (e.g. $\lceil 3.8 \rceil = 4$, $\lceil -3.8 \rceil = -3$, $\lceil 3 \rceil = 3$).