Fu-Yin Cherng

National Taiwan University

Recursion and Algorithm Complexity

Outline

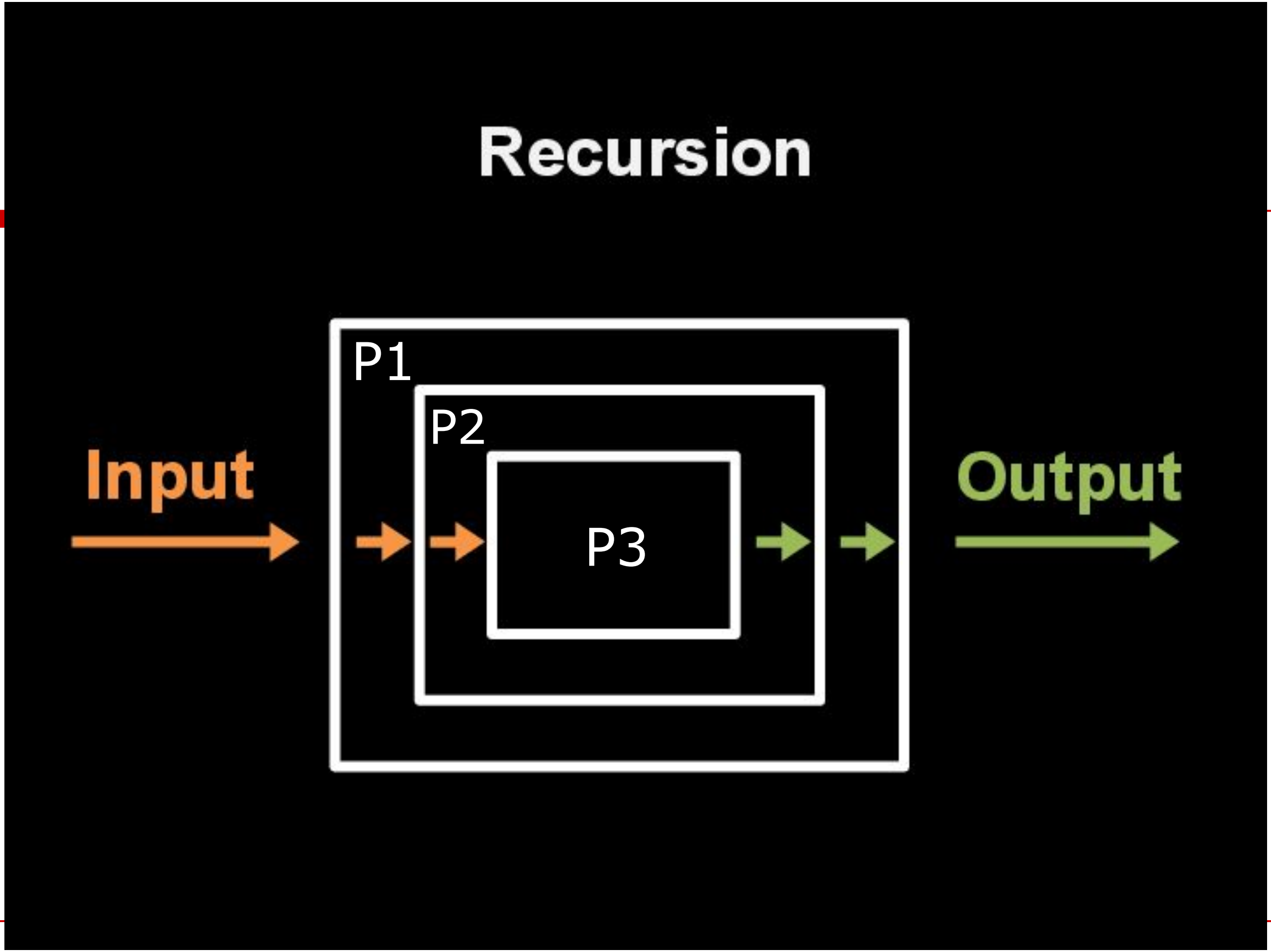
- **Recursion**
  - What is recursion?
  - Examples
  - Recursion & Efficiency
- Algorithm Complexity

Recursion

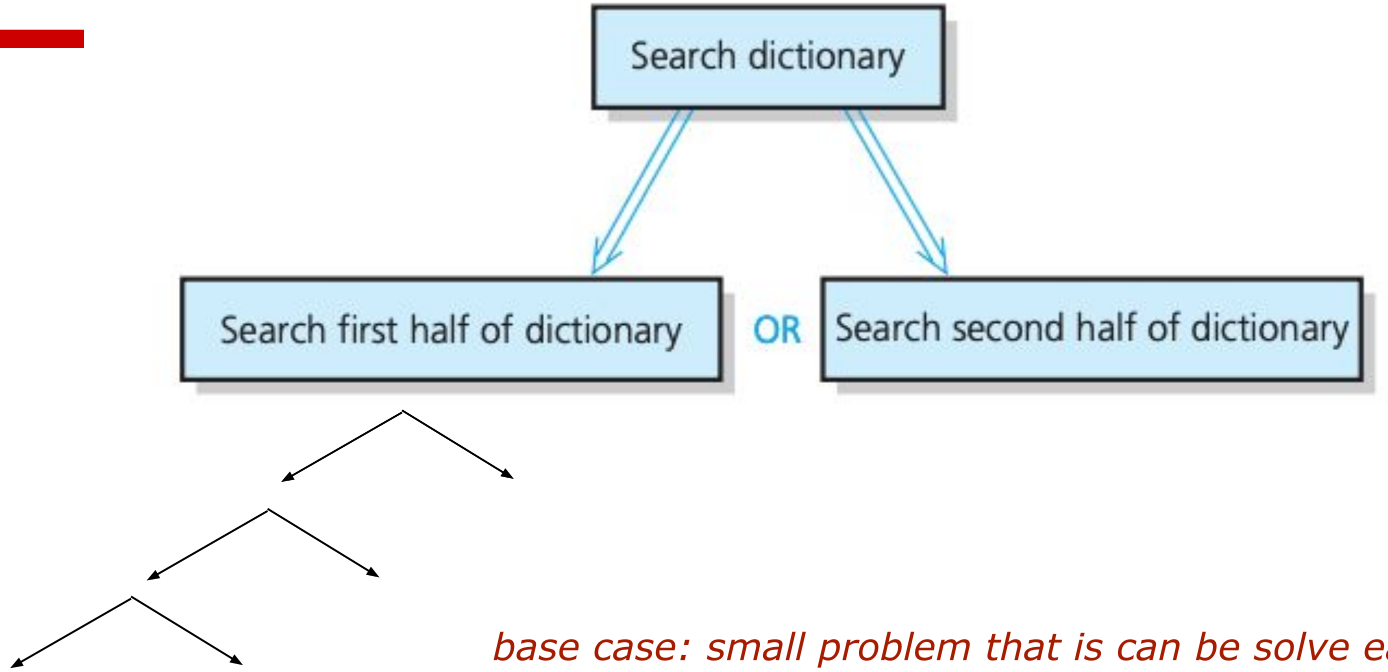
- most powewrful techniques for computer scientist
- using examples to show the thought porcesses that lead to recursive solution
- Some recursive solutions are more elegant and concise than their non-recursive counterparts, but some are not.

What is recursion?

- Recursion breaks a problem into smaller identical problems
- like mirror images, recursive solution solves a problem by solving a smaller instance of the same problem


Recursion
P1
P2
Input
P3
Output

Binary Search



base case: small problem that is can be solve easily

search a word in 1 page

Binary Search

- divide and conquer
  - *solve the problem by first dividing the dictionary into two halves and then conquering the appropriate half*
  - *solve the smaller problem by using the same divide-and-conquer strategy*
  - *dividing continues until you reach the base case*
- divide-and-conquer strategy use in many recursive solutions

Binary Search

```
search(aDictionary: Dictionary, word: string)
if (aDictionary is one page in size) //base case
    Scan the page for word
else{
    Open aDictionary to a point near the middle
    Determine which half of aDictionary contains word

    if (word is in the first half of aDictionary)
        search(first half of aDictionary, word)
    else
        search(second half of aDictionary, word)
} //end of else
```

4 questions for constructing recursive solutions

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

Examples

- Recursive Valued Function
  - The Factorial of n
    - The Box Trace: method to help you understand and debug recursive functions
- Recursive Void Function
  - Backward String

Example - The Factorial of n

- fit the mold mentioned earlier
- simple and efficient in iterative solution (loop), use recursive solution only for demonstration

$$factorial(n) = n \times (n-1) \times (n-2) \times \cdots \times 1 \quad \text{for an integer } n > 0$$
$$factorial(0) = 1$$

$$\begin{aligned} factorial(n) &= n \times [(n-1) \times (n-2) \times \cdots \times 1] \\ &= n \times factorial(n-1) \end{aligned}$$

Example - The Factorial of n

- define the base case: $\text{factorial}(0) = 1$
- always reach the base case
- complete recursive definition of factorial():

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times factorial(n-1) & \text{if } n > 0 \end{cases}$$

smaller problem/size

Example - The Factorial of n

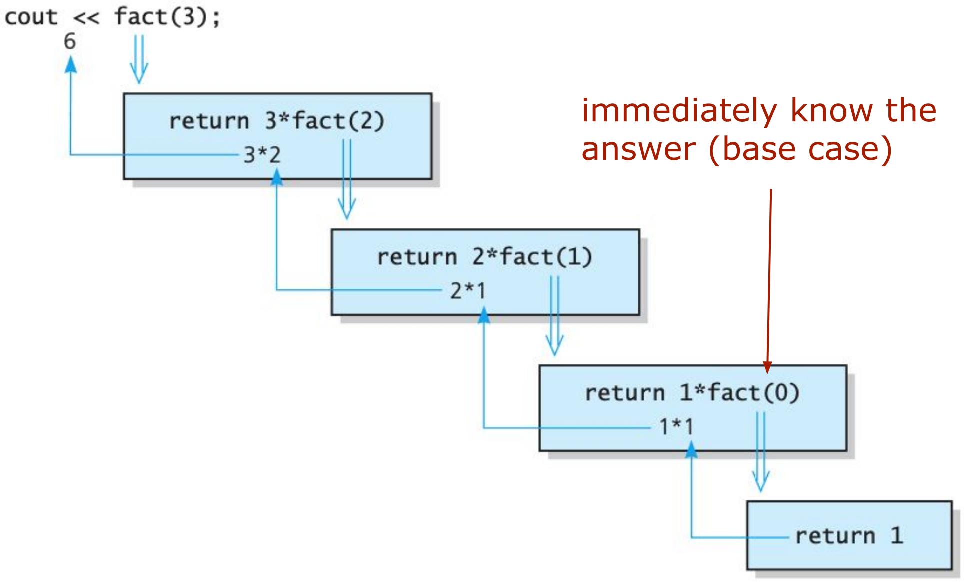
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120

immediately know the answer (base case)

Example - The Factorial of n

```
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
        // n * (n-1)! is n!
}

int main(){
    cout << fact(3);
    return 0;
}
```



The Box Trace

- systematic way to trace actions of a recursive function
- how a compiler usually implements recursion
- use it to help you understand recursion and to debug a recursive function.

The Box Trace of fact()

1. Label each recursive call in the body of the recursive function

- Keep track of the place to which you return after a function/recursive call

```
int fact(int n) {
    if (n == 0)
        return 1;
    else                        A
        return n * fact(n - 1);
        // n * (n-1)! is n!
}
```

The Box Trace of fact()

2. Represent each call to the function by a new box
   - note the local environmnet of the function

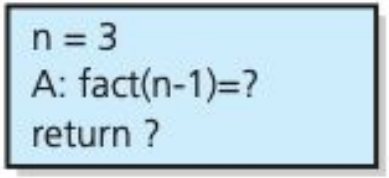
value of function arguments

draw arrow from statement

value of function itself

The Box Trace of fact()

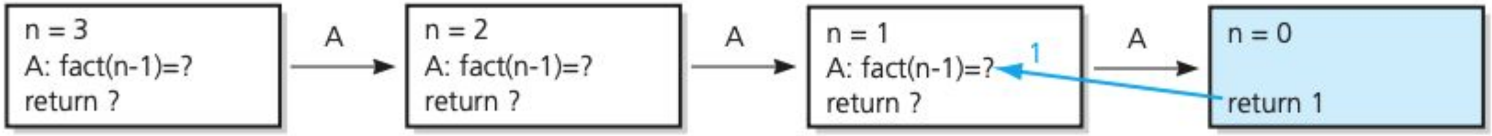
The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

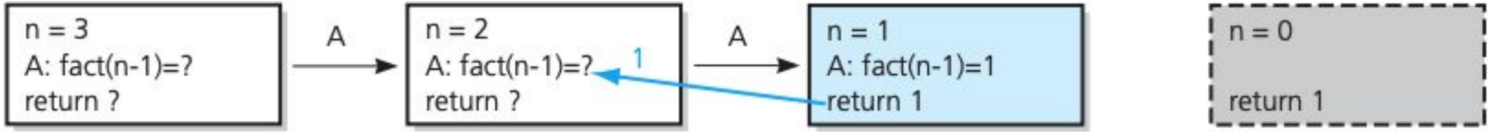
The Box Trace of fact()

This is the base case, so this invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:

The current invocation of `fact` completes and returns a value to the caller:



Example - Backward String

- recursion void function
  - recursive functin don’t need to return a value
- problem: write a given string in reverse order
  - “cat” as “tac”
- solution of the smaller problem can be used in the solution to the original problem
- recursive solution
  - writing a string of length n backward in terms of the problem of writing a string of length n – 1 backward

Example - Backward String

- define recursive solution
  - diminishes problem (string length) by 1 for each call
  - base case: writing a empty string backwared (string length = 0) = do nothing
  - immediate answer of base case: do nothing
  - always reach the base case
    - if not, the algorithm will not terminate

Example - Backward String

```
writeBackward(s: string)
if (the string is empty)//base case
    Do nothing—this is the base case
else{
    Write the last character of s
    writeBackward(s minus its last character)
}
```

cat →

cat -> t
ca -> a
c -> c
empty string -> no nothing

→ **tac**

Example - Backward String

```
void writeBackward(string s) {
int length = s.size(); // Length of string if (length > 0)
    if(length > 0){
        // Write the last character
        cout << s.substr(length - 1, 1);

        // Write the rest of the string backward
        writeBackward(s.substr(0, length - 1));
    }
    // length == 0 is the base case - do nothing
}// end writeBackward
```

Example - Backward String

```
void writeBackward(string s) {
int length = s.size(); // Length of string if (length > 0)
    if(length > 0){
        cout <<"To write last character of string:"<< s << endl;
        cout << s.substr(length - 1, 1);

        // Write the rest of the string backward
        writeBackward(s.substr(0, length - 1));
    }
    // length == 0 is the base case - do nothing
    cout << "Leave writeBackward with string: " << s << endl;
}// end writeBackward
```

More examples

- Recursion with Arrays
  - The Binary Search

Example - The Binary Search

- high-level example of find a word in a dictionary
- change to find a target value in an sorted array

```
binarySearch(anArray: ArrayType, target: ValueType)
if (anArray is of size 1)//base case
    Determine if anArray’s value is equal to target
else{
    Find the midpoint of anArray
    Determine which half of anArray contains target
    if (target is in the first half of anArray)
        binarySearch(first half of anArray, target)
    else
        binarySearch(second half of anArray, target)
}
```

Example - The Binary Search

- How to pass half of an array?
  - pass the entire array but only search the range of given index (e.g., array[first...last])
  - binarySearch(anArray, first, last, target)
  - change index in each recursion
- new midpoint = (first + last) / 2
  - 1st half: binarySearch(anArray, first, mid-1, target)
  - 2nd half: binarySearch(anArray, mid+1, last, target)

Example - The Binary Search

- How to determine which half of the array contains target?
  - because the array is sorted, so we only need to determine `if (target < anArray[mid])`
  - What if target == anArray[mid]?
- Base case(s): termination condition
  - `first > last` (target is not in the array)
  - `target==anArray[mid]`
- Return the index of the array value == target
  - index = -1 if doesn't find targe in the array

Example - The Binary Search

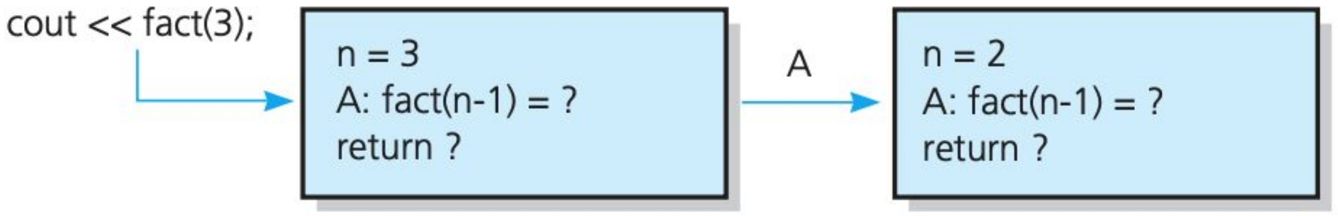
```
int binarySearch(const int anArray[], int first, int last, int target)
{
    int index;
    if (first > last) //base case
        index = -1; // target not in original array
    else{
        int mid = (first + last) / 2;
        if (target == anArray[mid]) //base case
            index = mid; // target found at anArray[mid]
        else if (target < anArray[mid])
            index = binarySearch(anArray, first, mid - 1, target);
        else
            index = binarySearch(anArray, mid + 1, last, target);
    }
    return index;
}
```

Recursion & Efficiency

- Using recursion can produce clean, simple, and short solutions.
- some recursive solutions are efficient but some are not
- 2 factors contribute to the inefficiency of some recursive solutions
  - overhead associated with function calls
  - inherent inefficiency of some recursive algorithms

Overhead associated with function calls

- each function call produces an activation record
  - a box in the box trace
- recursive function generate large number of recursive calls from an initial call



Overhead associated with function calls

- For example, factorial(n) generate n recursive (function) calls
- but the iterative version of factorial() is almost as clear as the recursive one and is more efficient!
- No reason for suffer the cost of overhead
- **Recursion is truly valuable when a problem has no simple iterative solutions.**

Inherent inefficiency of algorithms

- related to the technique that the algorithm employs
- inefficient algorithm will be recursively executed over and over again
- One solution is converting it into an iterative solution

Inherent inefficiency of algorithms

- For tail recursion, iterative solution often more efficient than recursive counterparts
  - solitary recursive call is the last action
  - the conversion is intuitive

```
void writeBackward(string s) {
     int length = s.size();
     if (length > 0){
          cout << s.substr(length - 1, 1);
          writeBackward(s.substr(0, length
     - 1));
     }
}
```

```
void writeBackward(string s){
     int length = s.size();
     while (length > 0){
          cout << s.substr(length - 1, 1);
          length--;
     }
}
```

Determine algorithm efficiency

- How to know whether this recursive algorithm is efficient or not?
- Analysis of algorithms

Outline

- Recursion
- **Algorithm Complexity**
  - What is a Good Solution/Algorithms?
  - Measuring the Efficiency of Algorithms?
    - Execution Time
    - Growth Rates
    - Big O Notation

Algorithm Complexity

- What’s algorithm?
  - 演算法
  - a process or set of rules to be followed in calculations or problem-solving operations
- central and advanced topics in computer science

What is a Good Solution?

- Executing program as solution generates cost
  - e.g., computing time and memory
- These kinds of cost only exist in execution
- Need to include multidimensional view of cost for a solution
  - e.g., designing, analyzing, debugging, and maintaining...

What is a Good Solution?

A solution is good if the total cost it incurs over all phases of its life is minimal.

- including the cost of human time and program execution (efficiency)
- The faster one is not necessarily better
- Good structure and documentation are important
  - don't create a program that only you can understand and maintain
  - `int i` vs `int user_id`

What is a Good Solution?

- solution's execution time is also important
  - Efficiency is only one aspect of a solution's cost
- select the solution with significantly better efficiency
- if two solutions have similar efficiency, consider other aspects
  - documentation, maintenance, structure
- Don't let efficiency overshadow all others

Measuring the Efficiency of Algorithms

- Analysis of algorithm: focus on the efficiency of different algorithms
- algorithms != programs
  - analyze the efficiency of superior algorithms
  - instead of coding implementation
- efficiency of algorithms dominate overall cost of a solution
- **time efficiency** and space efficiency

Execution Time of Algorithms

- related to number of required operations
- counting the number of an algorithm's operations
- For example, print data in a linked list with `n` node

```
Node<ItemType>* curPtr = headPtr;
while (curPtr != nullptr){

    cout << curPtr->getItem();
    curPtr = curPtr->getNext();
}
```

1 assignment (**a** time units

n+1 comparisons (**c** time units

n writes (**w** time units

n assignments (**a** time units

total required time units: (n + 1) X (a + c) + n X w

Execution Time of Algorithms

- if task T need t time units, how many time units for the following nested loops?

```
for(i = 1 through n)
    for(j = 1 through i)
        for(k = 1 through 5)
            Task T
```

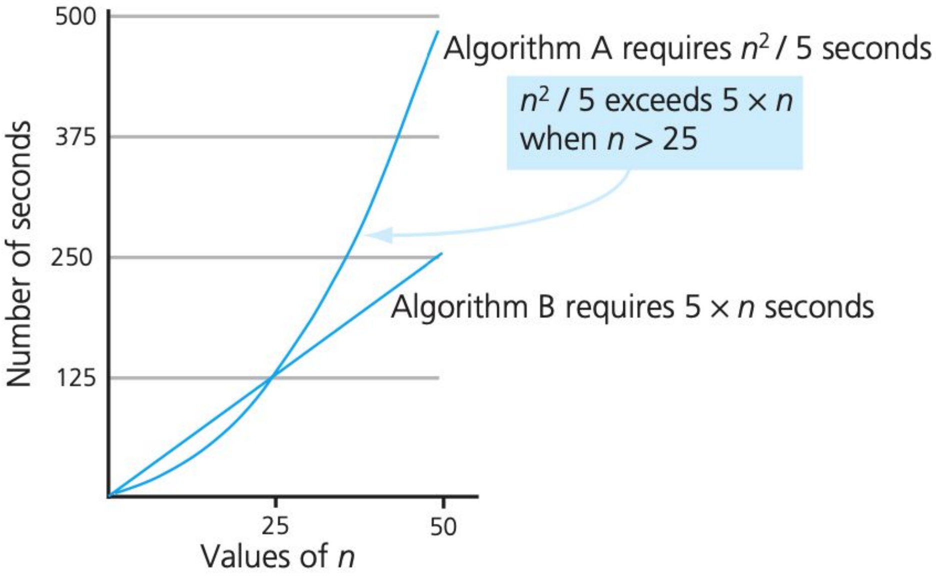
$$\boxed{5 \times t}$$

$$\boxed{5 \times t \times i}$$

$$\boxed{\sum_{i=1}^{n} (5 \times t \times i)}$$

Algorithm Growth Rates

- how quickly the algorithm's time requirement grows as a function of the problem size = growth rate



Algorithm Growth Rates

Algorithm A requires $n^2 / 5$ time units to solve a problem of size n
Algorithm B requires $5 \times n$ time units to solve a problem of size n

⬇

Algorithm A requires time proportional to n^2
Algorithm B requires time proportional to n

use this statment to describe efficiency of an algorithm

Algorithm Growth Rates

- algorithm efficiency is typically a concern for large problems
  - The time requirements for small problems are generally not large enough to matter.
- assume large values of n (large problem)

Big O Notation

Algorithm A requires time proportional to n^2
Algorithm B requires time proportional to n

- Algorithm A is order f(n) = O(f(n))
  - f(n) = growth-rate function
  - Algo A: O(n^2)
  - Algo B: O(n)
- Algo A (O(n^2)) is slower than Algo B (O(n))

Get growth-rate functions

- print data in a linked list with n node
  - time requirement: $(n + 1) \times (a + c) + n \times w$
  - (2 X n) X (a + c) + n X w **>=** (n + 1) X (a + c) + n X w
    - when n >= 1
    - (2 X n) >= (n+1)
  - (2 X a + 2 X C + w) X **n** >= (n + 1) X (a + c) + n X w
    - (2 X a + 2 X C + w) is contant
- The growth-rate function of this algo is **O(n)**

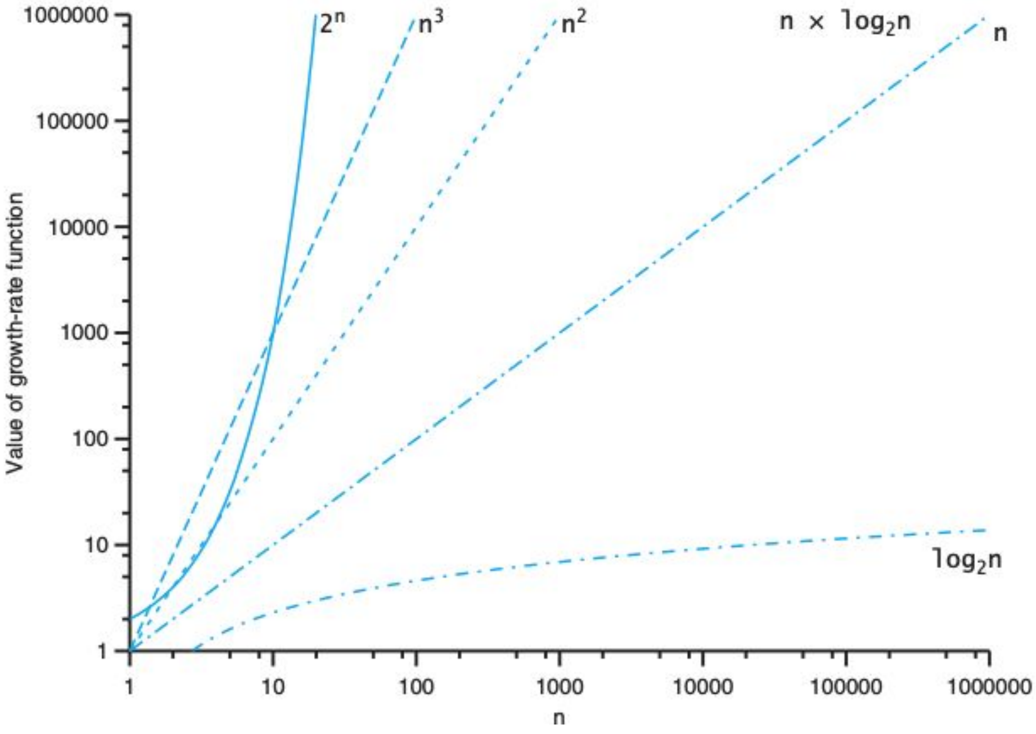
Common growth-rate functions

- Order of growth of some common functions

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

Function	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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Common growth-rate function



Interpretation common growth-rate functions

$O(1)$ time requirement is constant, independent of problem size n

$O(\log_2 n)$ time requirement slowly grow with problem size n

$O(n)$ linear increase of time requirement with problem size n

$O(n^2)$ quadratic increase of time requirement with n, often see in algo with two-nested loops

Mathematical properties of Big O notation

- help to simplify the analysis of an algorithm
- O(f (n)) means “is of order f(n)”
  - *Algo requires time proportional to f(n)*
- **O is not a function**

Mathematical properties of Big O notation

- gnore low-order terms in an algorithm's growth-rate function.

$$O(n^3 + 4 \times n^2 + 3 \times n) \longrightarrow O(n^3)$$

Mathematical properties of Big O notation

- ignore a multiplicative constant in the high-order term
- $O(5 \times n^3)$ = $O(n^3)$

- combine growth-rate functions
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Do not need an exact statement of an algorithm's time requirement

Summary

- What is Recursion
- Examples of Recursion
- Algorithm Efficiency & How to measure it
- Big O Notation
  - practice computing O() for previous examples
  - p.300 to see example of comparing efficiency of search algorithms