VE281

Data Structures and Algorithms

Asymptotic Algorithm Analysis and Arrays

Review

- Best, Worst, Average Cases

- Asymptotic Analysis: Big-Oh
  - Deal with the performance of algorithms for **large** input sizes
  - Upper bound

Outline

- Asymptotic Algorithm Analysis
- Recap of Arrays and Pointers

Big-Oh Notation

- Strictly speaking, we say that $T(n)$ is **in** $O(f(n))$, i.e.,
  $T(n) \in O(f(n))$
- However, for convenience, people also write
  $T(n) = O(f(n))$

A Sufficient Condition of Big-Oh

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, then $f(n)$ is $O(g(n))$.

- With this theorem, we can easily prove that
  $T(n) = c_1 n^2 + c_2 n$ is $O(n^2)$
  - Proof: $\lim_{n\to\infty} \frac{c_1 n^2 + c_2 n}{n^2} = c_1 < \infty$

Rules of Big-Oh

- **Rule 1**: If $f(n) = O\big(g(n)\big)$, then $cf(n) = O(g(n))$.
  - Example: $3n^2 = O(n^2)$
- **Rule 2**: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$
  - Example: $n^3 + 2n^2 = O(\max\{n^3, n^2\}) = O(n^3)$

Rules of Big-Oh

- **Rule 3**: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

- **Rule 4**: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

Common Mistakes of Big-Oh

- **Mistake 1**: $f(n) = O\big(g(n)\big) \Rightarrow f(n) = g(n)$.
  - Wrong!

- **Mistake 2**: If $f(n) \leq cg(n)$, where $c = h(n)$, then $f(n) = O(g(n))$.
  - Wrong!

Common Functions and Their Growth Rates

constant: 1

logarithmic: $\log n$

refers to $\log_2 n$

square root: $\sqrt{n}$

linear: n

loglinear: $n \log n$

quadratic: n^2

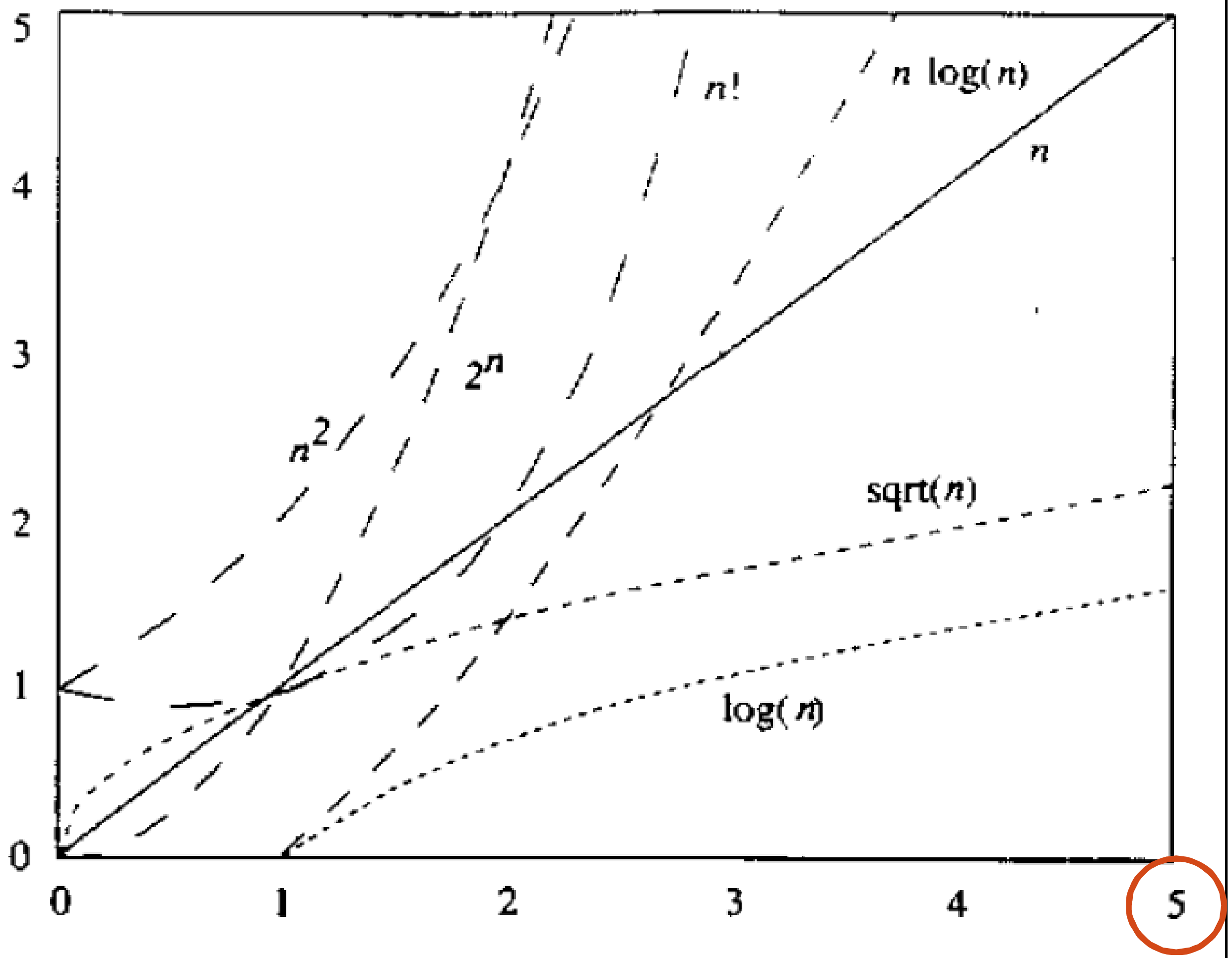
cubic: n^3

general polynomial: n^k

$k \geq 1$

exponential: $a^n, a > 1$

factorial: $n!$



Common Functions and Their Growth Rates

constant: 1

logarithmic: $\log n$

refers to $\log_2 n$

square root: $\sqrt{n}$

linear: n

loglinear: $n \log n$

quadratic: n^2

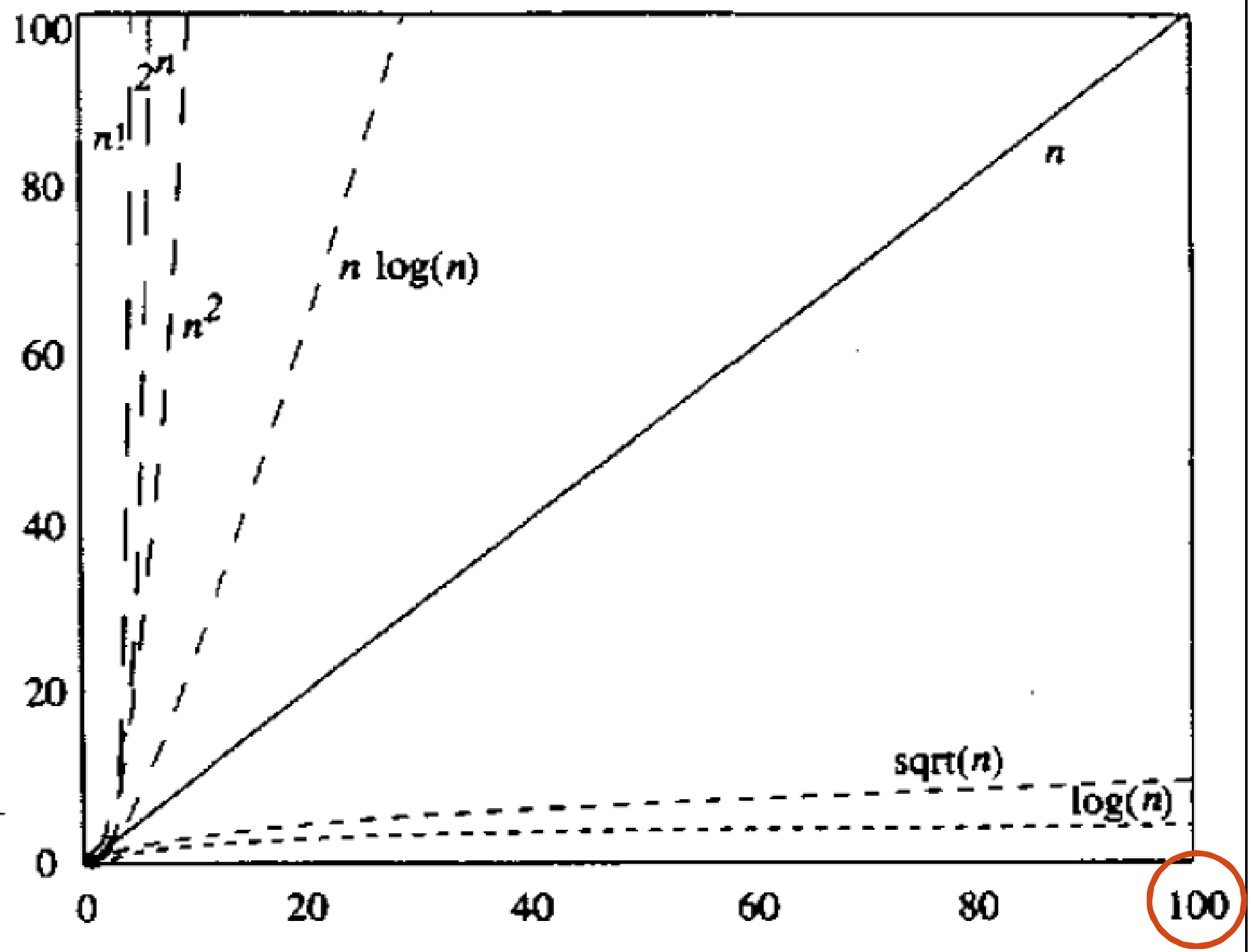
cubic: n^3

general polynomial: n^k

$k \geq 1$

exponential: $a^n, a > 1$

factorial: $n!$



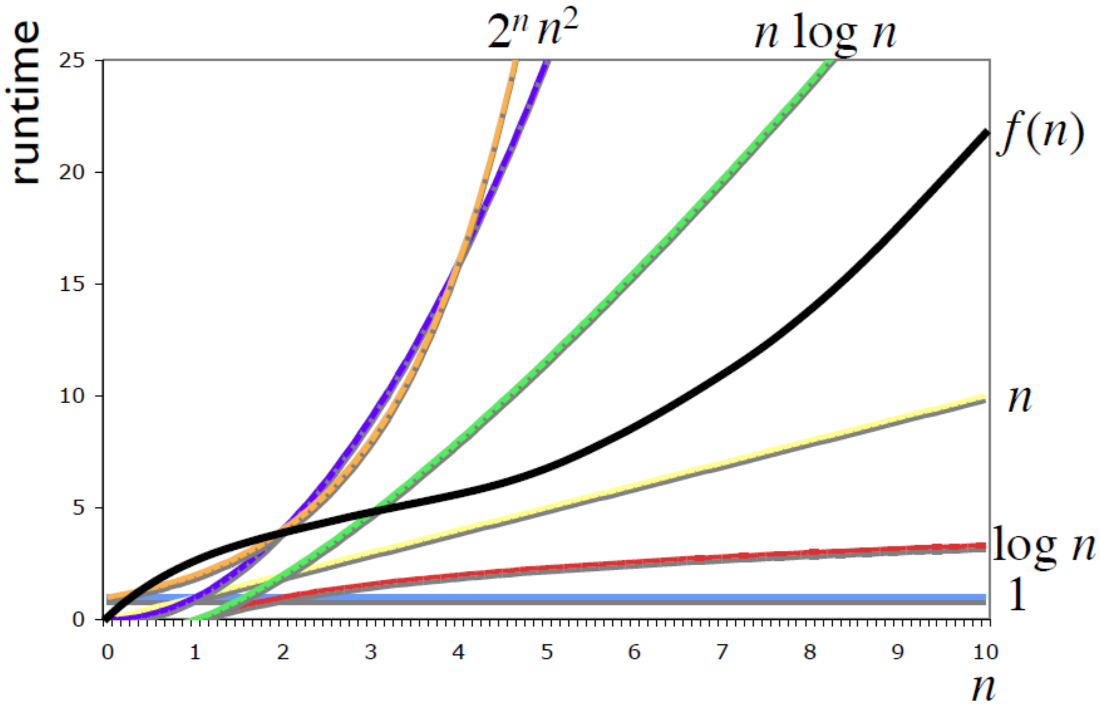
A Few Results about Common Functions

- For a polynomial in n of the form
  $$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$$
  where $a_m > 0$, we have $f(n) = O(n^m)$.

- For every integer $k \geq 1$, $log^k n = O(n)$.

- For every integer $k \geq 1$, $n^k = O(2^n)$.

How Fast is Your Code?



Let $f(n)$ be the complexity of your code, how fast would you advertise it as?

$f(n) = O(g(n))$; You want to pick a $g(n)$ that is as close to $f(n)$ as possible.

Relative of Big-Oh: Big-Omega

- Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the **set** $\Omega(g(n))$ if there **exist** two positive constants c and n_0 such that $T(n) \geq cg(n)$ for all $n > n_0$.

- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always requires **more than** $cg(n)$ steps.

- Big-omega gives a lower bound.
- We usually want the greatest lower bound.

Big-Omega Example

- Consider $T(n) = c_1n^2 + c_2n$, where c_1 and c_2 are positive.
- What is the big-omega notation for $T(n)$?

- Solution:
  - $c_1n^2 + c_2n \geq c_1n^2$ for all $n > 1$.
  - $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.
  - Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

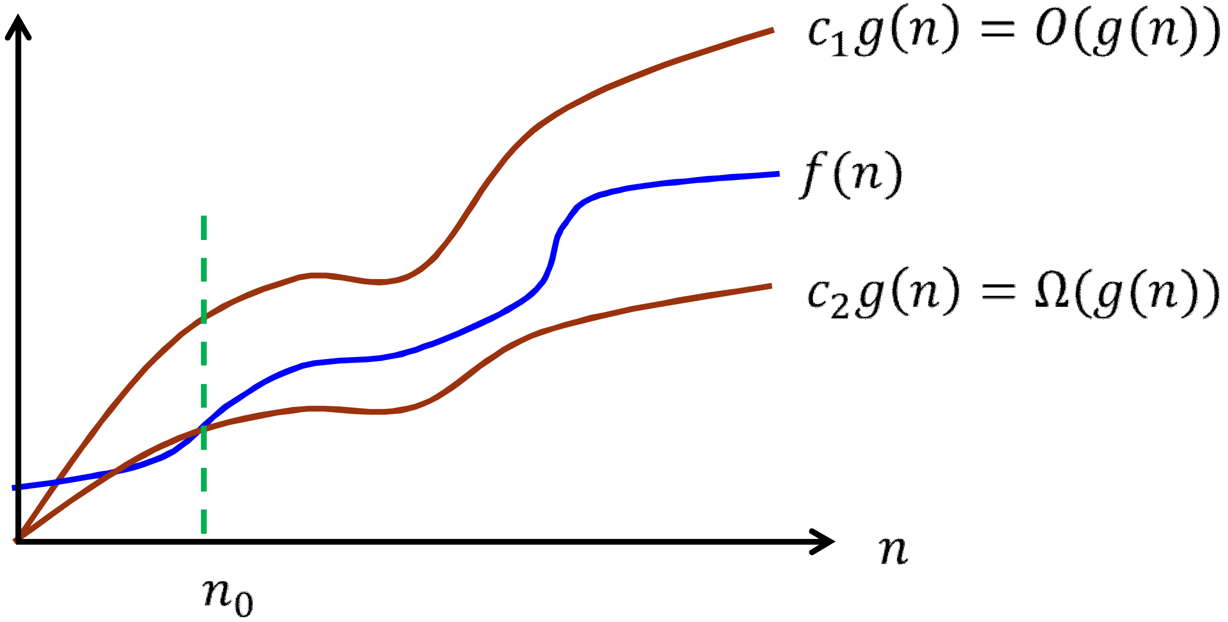
Theta Notation

- When big-oh and big-omega coincide, we indicate this by using big-theta (Θ) notation.

- Definition: $T(n)$ is said to be in the set $\Theta(g(n))$ if it is in $O(g(n))$ and it is in $\Omega(g(n))$.
  - In other words, there **exist** three positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for all $n > n_0$.

Theta Notation

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



- Question: Does $f(n) = \Theta\big(g(n)\big)$ indicate $g(n) = \Theta(f(n))$?

Analyzing Time Complexity of Programs

- For atomic statement, such as assignment, its complexity is $\Theta(1)$.

- For branch statement, such as if-else statement and switch statement, its complexity is that of the most expensive branch.

```
if(Boolean_Expression_1) {Statement_1}
else if (Boolean_Expression_2) {Statement_2}
…
else if (Boolean_Expression_n) {Statement _n}
else {Statement_For_All_Other_Possibilities}
```

Analyzing Time Complexity of Programs

- For subroutine call, its complexity is that of the subroutine.

- For loops, such as while and for loop, its complexity is related the number of operations required in the loop.

Time Complexity Example One

- What is the time complexity of the following code?

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

$\Theta(1)$ (sum = 0;)

$\Theta(n)$ (sum += i;)

$\Theta(1)$ $\Theta(n)$ $\Theta(n)$

- The entire time complexity is $\Theta(n)$.

Rule of Theta: If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$

Time Complexity Example Two

- What is the time complexity of the following code?

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

- Note that the statements

```
j <= i;
j++;
sum++;
```

  all occur (roughly) $1 + 2 + \cdots + n = n(n+1)/2$ times.
- The time complexity is $\Theta(n^2)$.

Time Complexity Example Three

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i *= 2)
   for(j = 1; j <= n; j++)
      sum++;
```

- The outer loop occurs $\log n$ times.
- The statements `sum++ / j<=n / j++` occur $n \log n$ times.
- The time complexity is $\Theta(n \log n)$.

Time Complexity Example Four

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i *= 2)
   for(j = 1; j <= i; j++)
      sum++;
```

- The number of times that the statements **sum++ / j<=i / j++** occur is

$$1 + 2 + 4 + 8 + \cdots 2^{\log n} \approx 2n - 1$$

- The time complexity is $\Theta(n)$.

Multiple Parameters

- Example: Compute the rank ordering for all C (i.e., 256) pixel values in a picture of P (i.e., 64×64) pixels.

```
for(i=0; i<C; i++)    // Initialize count
    count[i] = 0;

for(i=0; i<P; i++)    // Look at all pixels
    count[value[i]]++; // Increment count

sort(count);          // Sort pixel counts
```

$\Theta(C)$

$\Theta(P)$

$\Theta(C \log C)$

- The time complexity is $\Theta(P + C \log C)$.

Space/Time Trade-off Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.

- Example: factorial
  - Iterative method: Get "n!" using a for-loop.
  - This requires $\Theta(1)$ memory space and $\Theta(n)$ runtime.

  - Table lookup method: Pre-compute the factorials for $1,2,\cdots,N$ and store all the results in an array.
  - This requires $\Theta(n)$ memory space and $\Theta(1)$ runtime (fetching from an array).

Outline

- Asymptotic Algorithm Analysis
- Recap of Arrays and Pointers

Foundational Data Structures

- Many abstract data types (ADTs), such as stacks, queues, trees, priority queues, and graphs, can be implemented either using an array or some kind of linked data structure.

- We call array and linked list as the foundational data structures.

A Recap of Arrays

- An array is a fixed-sized, indexed collection of items, all of the same type.
- To declare and define an array of four integers, we would say the following:

```
int array[4];
```

- You can also initialize the contents of an array when declaring it:

```
int array[4] = { 1, 2, 3, 4 };
```

A Recap of Arrays

- You can access the contents of an array using an "index", such as

```
array[i]
```

- The index of the first array element is zero, the next is one, and so on. The last index of an array of size L is $L - 1$.

A Recap of Pointers

- Declaration: `int *bar;`
- Assigning address: `bar = &foo;`
  - The environment we get when we do this is:

0x804240c0	**foo:**	**1**
0x804240c4	**bar:**	**0x804240c0**

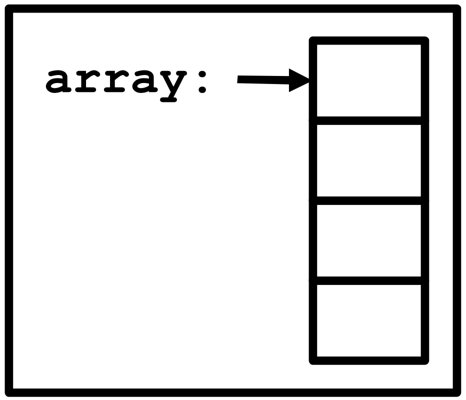
- Dereference: `*bar = 2;`
- Pointers as function arguments

```
void add_one(int *x) {
    *x = *x + 1;
}
```

Pointers and Arrays

- If you were to look at the value of the variable array (not array[0]) you'd find that it was exactly the same as the address of array[0].
- In other words,

(array==&array[0]) → True



Pointer Arithmetic

Enabling Array Traversal

```
int strlen(char *s)
  // REQUIRES: s is a NULL-terminated C-string
  // EFFECTS: returns the length of s, not
  // counting the NULL.
```

- We can implement **strlen** using only pointers and pointer arithmetic.

```
int strlen(char *s) {
    char *p = s;
    while (*p) {
        p++;
    }
    return (p - s);
}
```

Pointer Arithmetic

Enabling Array Traversal

```
int strlen(char *s) {
    char *p = s;
    while (*p) {
        p++;
    }
    return (p - s);
}
```

- Detailed explanation:
  - *p evaluates to "false" if p points to a NULL, true otherwise.
  - p++ advances by "one character".
  - p-s computes the "number of characters" between p and s, which happens to be the

Common Bugs of Arrays

- Out-of-bound access, including
  - index variable not initialized
  - off-by-one error

What's the bug?

```
int y[4]={0,1,2,3};
int i;
cout << y[i] << endl;
```

Index variable not initia

```
const int size = 5;
int x[size];
for(int i=0; i<=size; i++)
  x[i] = i*2;
```

Off-by-one error