Performance analysis of algorithms

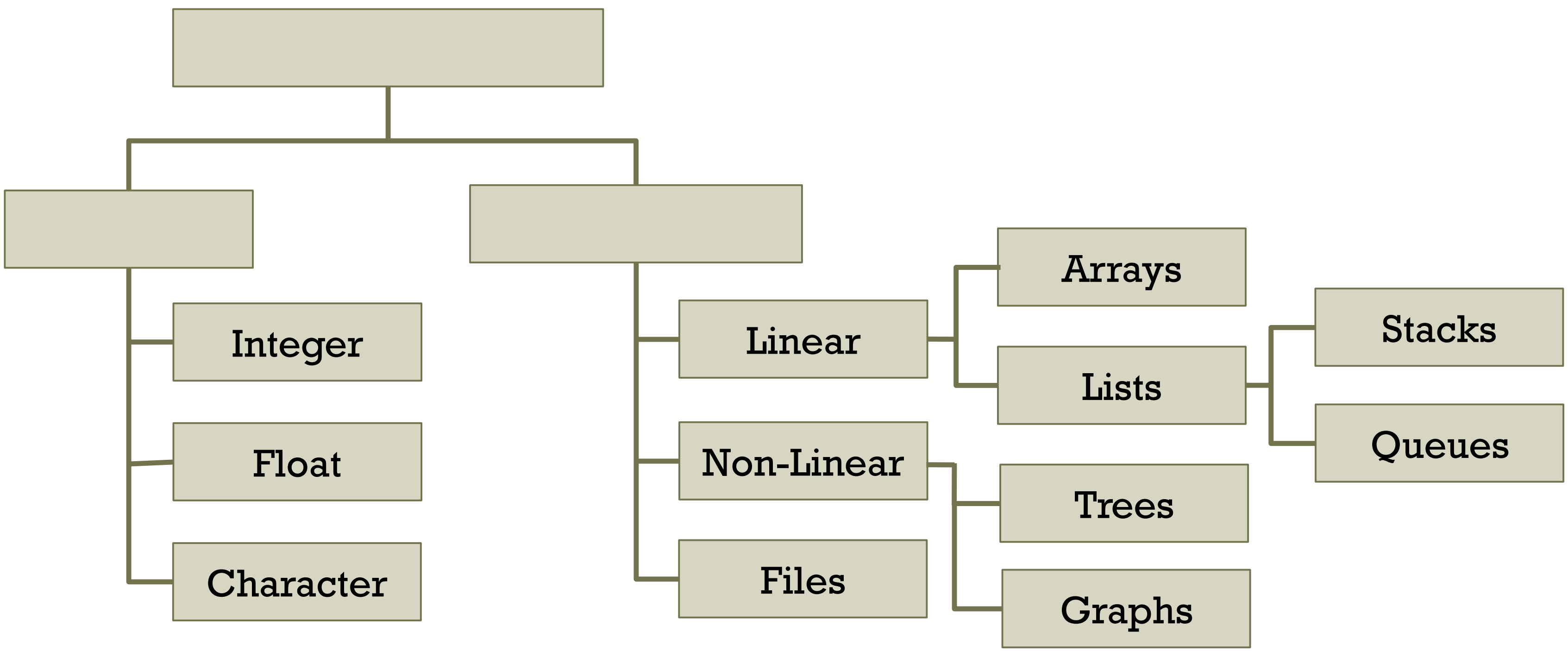
What is Programming?

Programming is to represent data and

solve the problem using the data.

What is Data Structure?

- Definition
  - It is a way of collecting and organizing data in a computer.
  - An aggregation of atomic and composite data into a set with defined relationships.



What is Algorithm?

- Definition: a finite set of instructions that should satisfy the following:
  - **Input**: zero or more inputs
  - **Output**: at least one output
  - **Definiteness**: clear and unambiguous instructions
  - **Finiteness**: terminating after a finite number of steps
  - **Effectiveness** (Machine-executable): enough to be carried out
- In computational theory, algorithm and program are different
  - Program does not satisfy 4): eg. OS

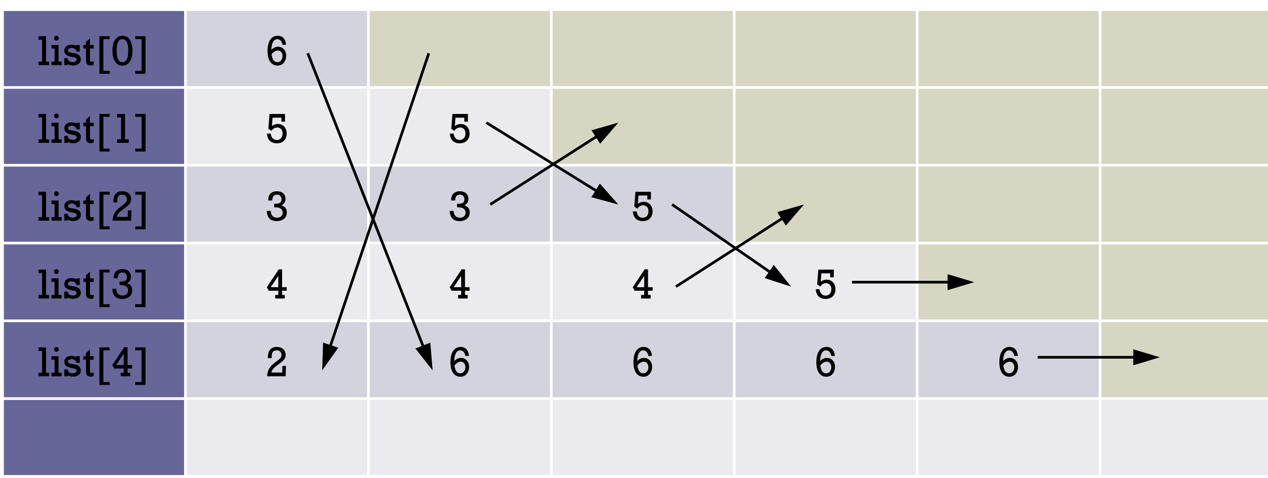
Algorithm Specification

- How to express algorithms
  - **High-level description**
    - Natural language
    - Graphic representation, e.g., flowcharts
    - Pseudocode: informal language-dependent description
  - **Implementation description**
    - C, C++, Java, and etc.

Natural Language vs. Graphic Chart

- Example: Selection Sort

From those integers that are currently unsorted, find the smallest value.
Place it next in the sorted list.



Pseudocode (C-like Language)

- Example: Selection Sort

```
for (i=0; i<n; i++) {
    Examine numbers in list[i] to list[n-1].
    Suppose that the smallest integer is at list[min].
    Interchange list[i] and list[min].
}
```

Implementation in C

- Example: Selection Sort

```c
void sort(int list[], int n)
{
   int i, j, min;
   for (i = 0; i < n - 1; i++) {
         min = i;
         for (j = i + 1; j < n; j++)
              if (list[j] < list[min])
                   min = j;
         SWAP(list[i], list[min]);
   }
}
```

Algorithm Analysis

- How do we evaluate algorithms?
  - Does the algorithm use the storage efficiently?
  - Is the running time of the algorithm acceptable for the task?

```
void search(int arr[], int len, int target) {
    for (int i = 0; i < len; i++) {
        if (arr[i] == target)
            return i;
    }
    return -1
}
```

- Performance analysis
  - Estimating **machine-independent** time and space

Space complexity: an amount of memory needed
Time complexity: an amount of time taken for an algorithm to finish

Space Complexity

- Definition
  - (machine-independent) space required by an algorithm

- Example

```c
int abc(int a, int b, int c)
{
    return a + b + b*c + 4.0;
}
```

```c
char* give_me_memory(int n)
{
    char *p = malloc(n);
    return p;
}
```

Space Complexity

- What is better for space complexity?

```
float sum(float *list, int n)
{
    float tempsum = 0;
    for (int i = 0; i < n; i++)
        tempsum += *(list + i);
    return tempsum;
}
```

```
float rsum(float *list, int n)
{
    if (n > 0)
        return rsum(list, n - 1) + list[n - 1];
    else
        return 0;
}
```

Time Complexity

- Definition
  - (machine-independent) time required by an algorithm
  - **Time is not easy to estimate!**

10 Additions, 10 subtractions, 10 multiplications

$$10C_a+10C_s+10C_m$$

C_a: time for one addition
C_s: time for one subtraction
C_m: time for one multiplication

- Alternative
  - Count the number of program steps instead of time.

10 Additions, 10 subtractions, 10 multiplications $\Rightarrow$

Time Complexity

- Program steps
  - Syntactically or semantically meaningful program segment whose execution time is independent of the number of inputs
  - Any one basic operation $\Rightarrow$ one step
    - +, -, *, /, assignment, jump, comparison, etc.
  - Any combination of basic operations $\Rightarrow$ one step
    - +=, *=, /=, (a+c*d), etc.

Time Complexity

- Example

```
int abc(int a, int b, int c)
{
    return a + b + b*c + 4.0;
}
```

```
void add(int a[][MAX_SIZE], ...)
{
    int i, j;
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

Time Complexity

- What is better for time complexity?

```c
float sum(float *list, int n)
{
    float tempsum = 0;
    for (int i = 0; i < n; i++)
        tempsum += *(list + i);
    return tempsum;
}
```

```c
float rsum(float *list, int n)
{
    if (n > 0)
        return rsum(list, n - 1) + list[n - 1];
    else
        return 0;
}
```

Asymptotic Notation

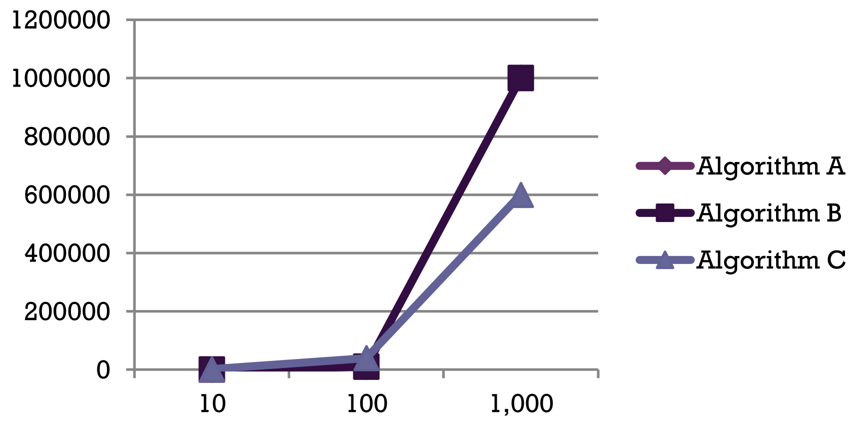
- Do we need to calculate exact numbers?

We have three algorithms, A, B, and C for the same problem.

- The time complexity of A: n^2+n+1
- The time complexity of B: n^2
- The time complexity of C: $200*n*\log(n)$

- What is a important factor? INCREASING SPEED!!
  - The highest term is enough to represent the complexity.
  - Constants is not important.

	10	100	1,000	10,000
A	111	10,101	1,001,001	???
B	100	10,000	1,000,000	???
C	2000	40,000	600,000	???



Asymptotic Notation

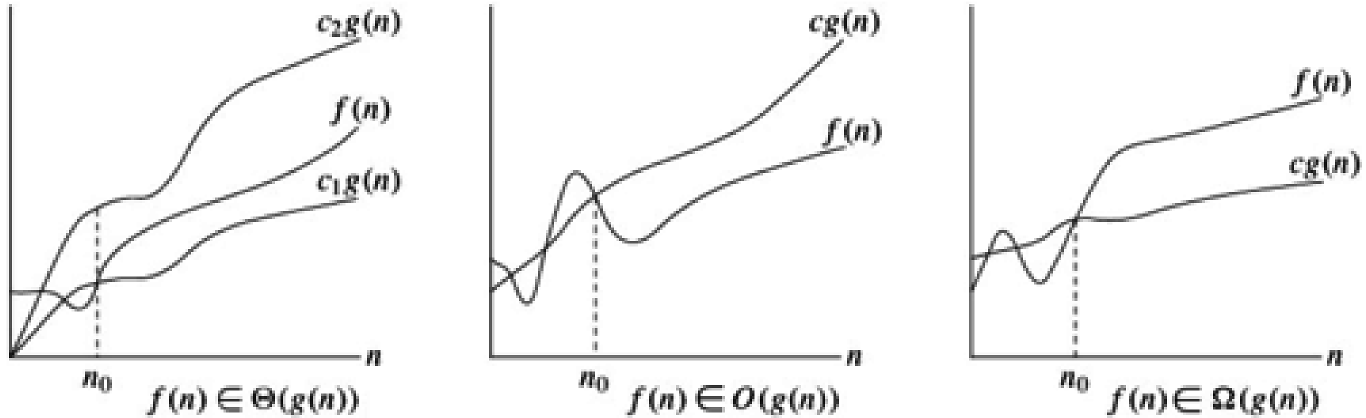
- What is better?
  - $(10n+10)$ vs. $(0.01n^2+10)$
  - $(2000n+3)$ vs. $(n\log n+1000)$
  - (n^3) vs. $(10n^2+1000000n)$

Simple rule:

1. Ignore any constants.
2. Compare only the term of the highest order.

Asymptotic Notation

- Three notations for complexity
  - **Big-O notation : O(*f*(*n*))**
    - The complexity is not increasing faster than *f*(*n*).
    - *f*(*n*) is an upper bound of the complexity.
  - **Big-Ω notation : Ω(*f*(*n*))**
    - The complexity is not increasing slower than *f*(*n*).
    - *f*(*n*) is a lower bound of the complexity.
  - **Big- Θ notation : Θ(*f*(n))**
    - The complexity is equally increasing to *f*(*n*).



Big-O Notation

- Definition

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

⇔ f(n) is not increasing faster than g(n)

⇔ For a large number n_o, c*g(n) will be larger than f(n)

⇔ There exist positive constants c and n_o such that

$$f(n) \leq c*g(n) \text{ for all } n > n_o$$

- Example
  - $3n + \log n + 2 = O(n)$, because $3n + \log n + 2 \leq 5n$ for $n \geq 2$
  - $10n^2 + 4n + 2 = O(n^4)$, because $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$

Example: Asymptotic Notation

- Three asymptotic notations for space complexity

```c
float sum(float *list, int n)
{
    float tempsum = 0;
    for (int i = 0; i < n; i++)
        tempsum += *(list + i);
    return tempsum;
}
```

→ 16 bytes

$\Theta(1)$

$O(1)$

$\Omega(1)$

```c
float rsum(float *list, int n)
{
    if (n > 0)
        return rsum(list, n - 1) + list[n - 1];
    else
        return 0;
}
```

→ 8*n bytes

$\Theta(n)$

$O(n)$

$\Omega(n)$

Example: Asymptotic Notation

- Three asymptotic notations for time complexity

```
float sum(float *list, int n)
{
    float tempsum = 0;
    for (int i = 0; i < n; i++)
        tempsum += *(list + i);
    return tempsum;
}
```

→ 2n+3

$\Theta(n)$

$O(n)$

$\Omega(n)$

```
void add(int a[][MAX_SIZE], ...)
{
    int i, j;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

→ 2*r*c + 2*r + 1

$\Theta(r*c)$

$O(r*c)$

$\Omega(r*c)$

Discussion: Asymptotic Notation

- Big-O notation is widely used.
  - Big-Θ notation is the most informative, but the exact value is hard to know.
  - Big-Ω notation is the least informative. Why?
  - Big-O notation is good for rough description.

There is algorithm *A*. The exact complexity is very hard to evaluate. However, we know that

$$n^2 \leq \text{complexity} \leq n^3.$$

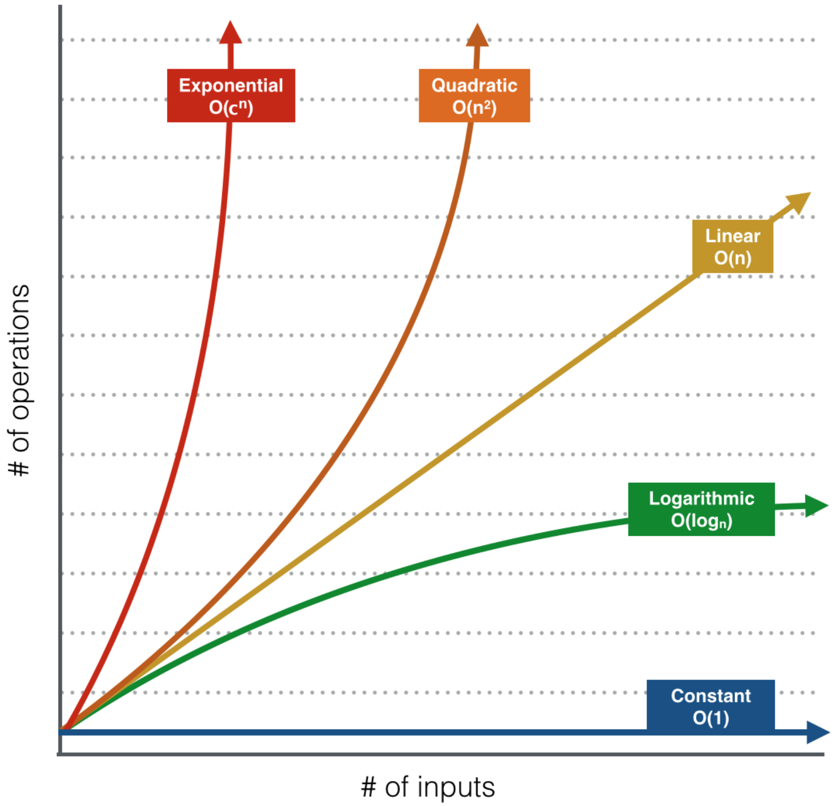
⬇

Then, we can say the complexity is $O(n^3)$.

Comparison: Asymptotic Notation

- Which is more costly?
  - O(1), O(log *n*), O(*n* log *n*), O(n^2), O(n^3), O(2^n), O(n!), etc..

O(1): constant
O($\log_2 n$): logarithmic
O(n): linear
O($n \cdot \log_2 n$): log-linear
O(n^2): quadratic
O(n^3): cubic
O(2^n): exponential
O(n!): factorial



Guideline for Asymptotic Analysis

- Loops
  - The number of iterations * the running time of the statements inside the loop

```
// executes n times
for (int i = 0; i < n; i++)
    m = m + 1; // constant time, c
// Total time = c * n = O(n)
```

```
// outer loop executed n times
for (int i = 0; i < n; i++)
    // inner loop executed n times
    for (int j = 0; j < n; j++)
        m = m + 1; // constant time, c
// Total time = c * n * n = O(n²)
```

Guideline for Asymptotic Analysis

- Consecutive statements
  - Add the time complexities of each statement.

```
// executes n times
for (int i = 0; i < n; i++)
    m = m + 1; // constant time, c1

// outer loop executed n times
for (int i = 0; i < n; i++)
    // inner loop executed n times
    for (int j = 0; j < n; j++)
        k = k + 1; // constant time, c2
// Total time = c1 * n + c2 * n^2 = O(n^2)
```

Guideline for Asymptotic Analysis

- If-then-else statements
  - Consider the worst-case running time among the if, else-if, or else part (whichever is the larger).

```
// executes n times
if (len > 0)
    for (int i = 0; i < n; i++)
        m = m + 1; // constant time, c1
else {
    // outer loop executed n times
    for (int i = 0; i < n; i++)
        if (i > 0)
            k = k + 2 // constant time, c2
        else
            // inner loop executed n times
            for (int j = 0; j < n; j++)
                k = k + 1; // constant time, c3
}
// Total time = n * n * c3 = O(n^2)
```

Guideline for Asymptotic Analysis

- Logarithmic complexity
  - An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
// At kth step, 2k = n and come out of loop.
for (int i = 1; i < n; i*=2)
    m = m + 1; // constant time, c
// Because k = log2n, total time = O(log n)
```

```
// The same condition holds for decreasing sequence.
for (int i = n; i > 0; i/=2)
    m = m + 1; // constant time, c
// Because k = log2n, total time = O(log n)
```