Algorithm Efficiency

1

Contents

- A review on algorithm
- Analysis and Big-O notation
- Algorithm efficiency

2

A review on algorithm

What is Algorithm?

- An algorithm is
  - a strictly defined **finite** sequence of **well-defined** steps (statements, often called instructions or commands)
  - that provides the solution to a problem.

Algorithm

- Give some examples of algorithms.

An Example

- `Input: No`
- `Output:` what do you think about the output?

- `Step 1. Assign sum = 0. Assign i = 0.`
- `Step 2.`
  - `Assign i = i + 1`
  - `Assign sum = sum + i`
- `Step 3. Compare i with 10`
  - `if i  < 10, back to step 2.`
  - `otherwise, if i ≥ 10, go to step 4.`
- `Step 4. return sum`

fit@hcmus

Characteristics of Algorithms

- **Finiteness**

  For any input, the algorithm must terminate after a finite number of steps.
- **Correctness**

  Always correct. Give the same result for different run time.
- **Definiteness**

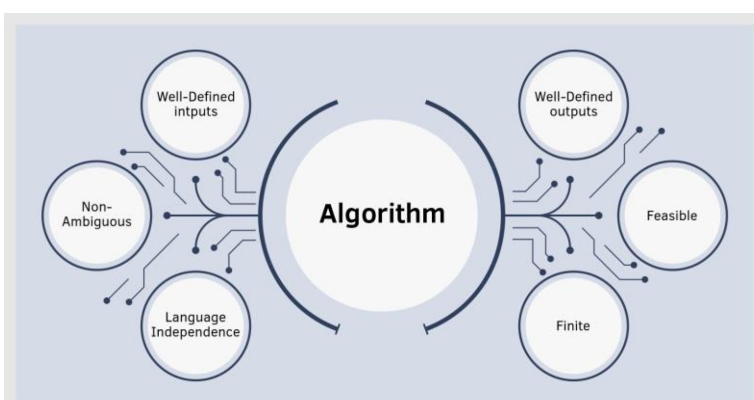
  All steps of the algorithm must be precisely defined.
- **Effectiveness**

  It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

7

fit@hcmus

Characteristics of Algorithms



8

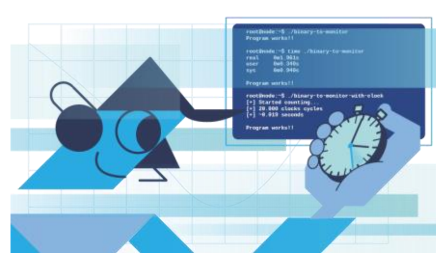
Algorithm Efficiency

- The two factors of Algorithm Efficiency are:
  - **Time Factor**: Time is measured by counting the number of key operations.
  - **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm.

Measuring Efficiency of Algorithms

- Can we compare two algorithms (in time factor) like this?
  - Implement those algorithms (into programs)
  - Calculate the execution time of those programs
  - Compare those two values of time measurement.

Is it fair in this measuring process?



Measuring Efficiency of Algorithms

- Difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?

Measuring Efficiency of Algorithms

- Comparison of algorithms should focus on **significant differences** in efficiency

- Employ mathematical techniques that analyze algorithms **independently** of specific implementations, computers, or data.

Execution Time of Algorithm

- Time complexity is measured by counting the **primitive operations** for the computation that the algorithm needs to perform.
  - Comparisons
  - Assignments
- Derive an algorithm's time requirement as a function of **the problem size**
  - Algorithm *A* requires $n^2/5$ time unit to solve a problem of size *n*.
  - Algorithm *B* requires 5 x n time unit to solve a problem of size *n*.

13

Execution Time of Algorithm

- Traversal of linked nodes – example:

```
Node<ItemType>* curPtr = headPtr;          ← 1 assignment
while (curPtr != nullptr)                  ← n + 1 comparisons
{
   cout << curPtr->getItem() < endl;       ← n writes
   curPtr = curPtr->getNext();             ← n assignments
}  // end while
```

  - Assignment: *a* time units.
  - Comparison: *c* time units.
  - Write: *w* time units.
- Displaying data in linked chain of *n* nodes requires time proportional to *n*

14

Execution Time of Algorithm

fit@hcmus

- Nested loops

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

  - Task *T* requires *t* time units.

Previous Example

fit@hcmus

- **Step 1.** `Assign sum = 0. Assign i = 0.`
- **Step 2.**
  - `Assign i = i + 1`
  - `Assign sum = sum + i`
- **Step 3.** `Compare` ***i*** `with` ***10***
  - `if i  < 10, back to step 2.`
  - `otherwise, if i ≥ 10, go to step 4.`
- **Step 4.** `Return sum`

How many
- Assignments?
- Comparisons?

Another Example

- **Step 1.** Assign sum = 0. Assign i = 0.
- **Step 2.**
  - Assign i = i + 1
  - Assign sum = sum + i
- **Step 3.** Compare *i* with ***n***
  - if i < ***n***, back to step 2.
  - otherwise, if i ≥ ***n***, go to step 4.
- **Step 4.** Return sum

How many
- Assignments?
- Comparisons?

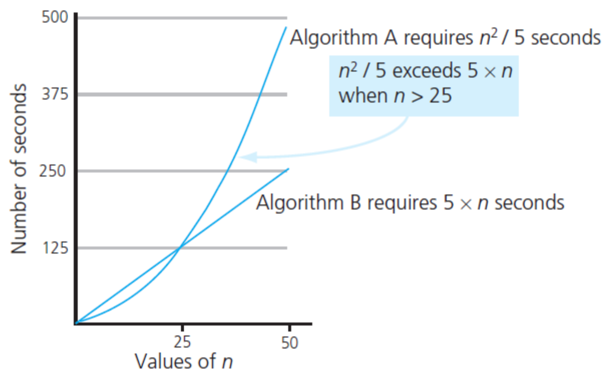
Algorithm Growth Rates

- Measure algorithm's time requirement **as a function** of problem size
- Compare algorithm efficiencies for **large problems**
- Look only at **significant differences**.

fit@hcmus

Algorithm Growth Rates

- Time requirements as a function of the problem size *n*



fit@hcmus

Analysis and Big O Notation

fit@hcmus

Big O Notation

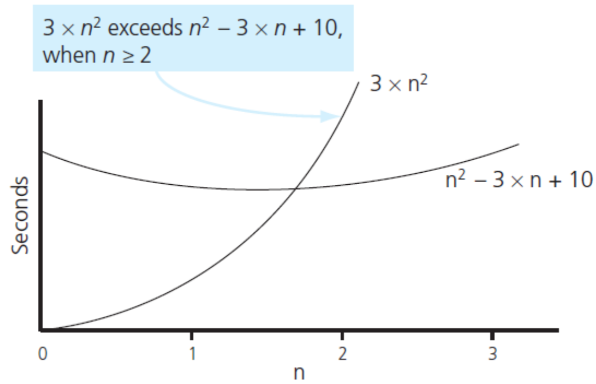
- Definition:
  - Algorithm A is order f (n)
    - Denoted **O(f (n))**
  - If constants **k** and $\mathbf{n_0}$ exist
  - Such that A requires **no more** than **k** × **f (n)** time units to solve a problem of size $\mathbf{n \geq n_0}$.

fit@hcmus

Example

- An algorithm requires $\mathbf{n^2 - 3 \times n + 10}$ (time units). What is the order of algorithm?
  - Hint: Find the values ***k*** va $\boldsymbol{n_0}$.

Example



The graphs of $\mathbf{3 \times n^2}$ and $\mathbf{n^2 - 3 \times n + 10}$

23

Another Example

- How about the order of an algorithm requiring

$$(n + 1) \times (a + c) + n \times w$$

time units?

24

Another Example

- Another algorithm requires $\mathbf{n^2 + 3 \times n + 2}$ time units. What is the order of this algorithm?

Common Growth-Rate Functions

- f(n) =
  - 1: Constant
  - $\log_2 n$: Logarithmic
  - n: Linear
  - $n \times \log_2 n$: Linearithmic
  - n^2: Quadratic
  - n^3: Cubic
  - 2^n: Exponential

fit@hcmus

Common Growth-Rate Functions

- Order of growth of some common functions

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

fit@hcmus

Common Growth-Rate Functions

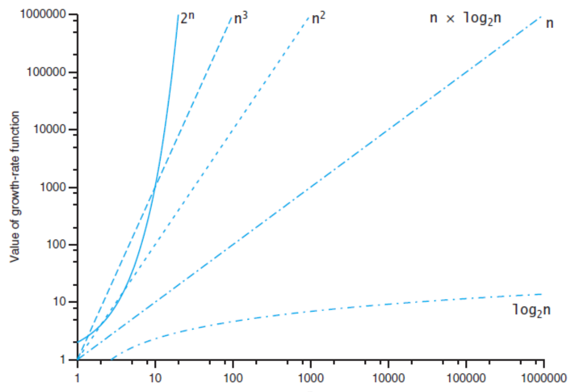
- A comparison of growth-rate functions in **tabular form**

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

fit@hcmus

Common Growth-Rate Functions

- A comparison of growth-rate functions in **graphical form**



fit@hcmus

Properties of Growth-Rate Functions

- Ignore low-order terms
- Ignore a multiplicative constant in the high-order term
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Some Useful Results

- **Constant Multiplication**:
  - If f(n) is O(g(n)) then **c**.f(n) is O(g(n)), where **c** is a constant.
- **Polynomial Function:**
  - $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$.

Some Useful Results

- **Summation Function:**
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - Then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- **Multiplication Function:**
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - Then $f_1(n) \times f_2(n)$ is $O(g_1(n) \times g_2(n))$

Quiz

fit@hcmus

Are these functions of order O(x)?

a) f(x) = 10
b) f(x) = 3x + 7
c) f(x) = $2x^2 + 2$

Quiz

fit@hcmus

What are the order of the following functions?

- $f(n) = (2 + n)(3 + \log_2 n)$
- $f(n) = 11 \log_2 n + \frac{n}{2} - 3542$
- $f(n) = n(3 + n) - 7n$
- $f(n) = \log_2(n^2) + n$

fit@hcmus

Quiz

- What are the order of the following functions?
  - $f(n) = 3^n + n^5$
  - $f(n) = 5n^3 + 100n^2 + 200$
  - $f(n) = n \log n + 500n$
  - $f(n) = \sqrt{n} + log\, n$
  - $f(n) = n! + 2^n$

fit@hcmus

Notes

- Use like this:
  - f(x) is O(g(x)), or
  - f(x) is of order g(x), or
  - f(x) has order g(x)

fit@hcmus

Algorithm Efficiency

fit@hcmus

Algorithm Efficiency

- Best case scenario
- Worst case scenario
- Average case scenario

An Algorithm to Analyze

fit@hcmus

- Input:
- Output:

- **Step 1.** Set the first integer the temporary maximum value (temp_max).
- **Step 2.** Compare the current value with the temp_max.
  - If it is greater than, assign the current value to temp_max.
- **Step 3.** If there is other integer in the list, move to next value. Back to step 2.
- **Step 4.** If there is no more integer in the list, stop.
- **Step 5.** return temp_max (the maximum value of the list).

Another Algorithm to Analyze

fit@hcmus

- Input:
- Output:

- Step 1. Assign $\boldsymbol{i} = 0$
- Step 2. While $\boldsymbol{i} < \boldsymbol{n}$ and $\boldsymbol{x} \neq \boldsymbol{a_i}$, increase $\boldsymbol{i}$ by 1.

```
while (i < n and x ≠ a_i)
        i = i + 1
```

- Step 3.
  - If $\boldsymbol{i} < \boldsymbol{n}$, return $\boldsymbol{i}$.
  - Otherwise (i >= n), return **-1** to tell that $\boldsymbol{x}$ does not exist in list $\boldsymbol{a}$.

Another Algorithm to Analyze

- Use comparisons for counting.
- Worst case:
  - When it occurs?
  - How many operations?
- Best case:
  - When it occurs?
  - How many operations?

Another Algorithm to Analyze

- Use comparisons for counting.
- Average case:
  - If x is found at position i^{th}, the number of comparisons is $2i + 1$.
  - The average number of comparisons is:

$$\frac{3+5+7+..+(2n+1)}{n} = \frac{2(1+2+3+...+n)+n}{n} = \frac{2\frac{n(n+1)}{2}+n}{n} = n+2$$

Analysis of Algorithms

fit@hcmus

- Decide **n** – the input size
- Identify the algorithm's **basic operation** (as a rule, it is in the innermost loop)
- Check whether the number of times the basic operation is executed depends only on n
  - If it depends on some additional property, specify the **worst-case** for Big-Oh
- Set up a **sum** expressing the number of times the algorithm's basic operation is executed.
- Find a closed-form formula for the count and **establish its order of growth.**

Analysis of Algorithms

fit@hcmus

- **Example:** Check whether all the elements in a given array of *n* elements are distinct.

```
UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
 for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
       if A[i] = A[j]
          return false
 return true
```

Basic operation

Analysis of Algorithms

- Worst-case:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

Keeping Your Perspective

- If problem size always small, ignore an algorithm's efficiency
- Weigh trade-offs between algorithm's time and memory requirements
- Compare algorithms for both style and efficiency

fit@hcmus

Exercises

fit@hcmus

Exercise

- Propose an algorithm to calculate the value of *S* defined below. What order does the algorithm have?

$$S = 1 + \frac{1}{2} + \frac{1}{6} + \ldots + \frac{1}{n!}$$

- How many comparisons, assignments are there in the following code fragment with the size *n?*

```
sum = 0;
for (i = 0; i < n; i++)
{
    std::cin >> x;
    sum = sum + x;
}
```

Exercise

fit@hcmus

How many assignments are there in the following code fragment with the size *n?*

```
for (i = 0; i < n ; i++)
      for (j = 0; j < n; j++)
      {
            C[i][j] = 0;
            for (k = 0; k < n; k++)
                  C[i][j] = C[i][j] + A[i][k]*B[k][j];
      }
```


Exercise

fit@hcmus

- Give the order of growth (as a function of N) of the running time of the following code fragment:

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
    for (int i = 0; i < n; i++)
        sum++;
```

Exercise

- Give the order of growth (as a function of N) of the running time of the following code fragment:

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


Exercise

fit@hcmus

- Give the order of growth (as a function of N) of the running time of the following code fragment:

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

fit@hcmus

Questions and Answers