CMPSC 465: LECTURE I

Introduction to Algorithm Analysis

Ke Chen

Algorithm analysis

Three questions will drive us:

1. Is the algorithm/data structure correct?
2. Is it efficient? (time and space)
3. Can we do better?

Other important considerations: robustness, energy efficiency, cache locality, modularity, maintenance time, etc. are beyond the scope of this course.

How to measure running time?

Issue 1: Running time may depend on input size.

Idea: Parametrize running time by the size of the input.

Worst-case analysis: (usually)

- $T(n) =$ maximum time of algorithm on ANY input of size n.

Average-case: (sometimes)

- $T(n) =$ expected time of algorithm over all inputs of size n.
- Requires assumption on input distribution.

Amortized: (sometimes)

- $T(m) =$ total time over m calls / m.

Best-case: (never)

- Cheat with a slow algorithm that works well on some inputs.

An example: linear search

```
linear-search(A, k)

Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found

i = n - 1
while i >= 0:
    if A[i] == k:
        return i
    i = i - 1
return -1
```

An example: linear search

<u>linear-search(A, k)</u>

```
Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found
```

	Cost	Rounds
`i = n - 1`	c_1	1
`while i >= 0:`		
`    if A[i] == k:`		
`        return i`		
`    i = i - 1`		
`return -1`		

An example: linear search

<u>`linear-search(A, k)`</u>

```
Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found
```

	Cost	Rounds	
`i = n - 1`	c_1	1	
`while i >= 0:`	c_2	n	(worst-case)
`    if A[i] == k:`			
`        return i`			
`    i = i - 1`			
`return -1`			

An example: linear search

```
linear-search(A, k)

Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found
```

	Cost	Rounds	
`i = n - 1`	c_1	1	
`while i >= 0:`	c_2	n	(worst-case)
`    if A[i] == k:`	c_3	n	(worst-case)
`        return i`	c_4	1	
`    i = i - 1`	c_5	n	(worst-case)
`return -1`	c_4	1	

An example: linear search

```
linear-search(A, k)

Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found
```

	Cost	Rounds	
`i = n - 1`	c_1	1	
`while i >= 0:`	c_2	n	(worst-case)
`    if A[i] == k:`	c_3	n	(worst-case)
`        return i`	c_4	1	
`    i = i - 1`	c_5	n	(worst-case)
`return -1`	c_4	1	

Worst-case running time:
$T(n) = c_1 + c_2 n + c_3 n + c_4 + c_5 n = (c_2 + c_3 + c_5)n + c_1 + c_4.$

An example: linear search

```
linear-search(A, k)

Input: an array A of size n, a key k to search for
Output: index of k in A, -1 if not found
```

	Cost	Rounds	
`i = n - 1`	c_1	1	
`while i >= 0:`	c_2	n	(worst-case)
`    if A[i] == k:`	c_3	n	(worst-case)
`        return i`	c_4	1	
`    i = i - 1`	c_5	n	(worst-case)
`return -1`	c_4	1	

Worst-case running time:

$$\underbrace{T(n)}_{\text{Linear}} = c_1 + c_2 n + c_3 n + c_4 + c_5 n = \underbrace{\boxed{(c_2 + c_3 + c_5)}}_{a} n + \underbrace{\boxed{c_1 + c_4}}_{b}.$$

How to measure running time?

Issue 2: Running time may depend on implementation/hardware.

Idea: Give up on gauging the actual time and focus on scalability by considering a simplified abstract computing model:

- single processor, sequential execution
- elementary operations take constant time
  - addition, subtraction
  - multiplication, division
  - assignment, branching
  - subroutine call
  - etc.

How to measure running time?

Issue 2: Running time may depend on implementation/hardware.

Idea: Give up on gauging the actual time and focus on scalability by considering a simplified abstract computing model:

- single processor, sequential execution
- elementary operations take constant time
  - addition, subtraction
  - multiplication, division
  - assignment, branching
  - subroutine call
  - etc.

Warning: we assume operands are of constant size irrelevant to the input size n (see book chapter 1).

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to$
- 100x time? E.g., $T(n) = 0.3n^2 \to$
- 1000x time? E.g., $T(n) = 82n^3 \to$
- much much longer? E.g., $T(n) = 2^n \to$

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to T(10n) = 150n$ (linear)
- 100x time? E.g., $T(n) = 0.3n^2 \to$
- 1000x time? E.g., $T(n) = 82n^3 \to$
- much much longer? E.g., $T(n) = 2^n \to$

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to T(10n) = 150n$ (linear)
- 100x time? E.g., $T(n) = 0.3n^2 \to T(10n) = 30n^2$ (quadratic)
- 1000x time? E.g., $T(n) = 82n^3 \to$
- much much longer? E.g., $T(n) = 2^n \to$

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to T(10n) = 150n$ (linear)
- 100x time? E.g., $T(n) = 0.3n^2 \to T(10n) = 30n^2$ (quadratic)
- 1000x time? E.g., $T(n) = 82n^3 \to T(10n) = 82000n^3$ (cubic)
- much much longer? E.g., $T(n) = 2^n \to$

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to T(10n) = 150n$ (linear)
- 100x time? E.g., $T(n) = 0.3n^2 \to T(10n) = 30n^2$ (quadratic)
- 1000x time? E.g., $T(n) = 82n^3 \to T(10n) = 82000n^3$ (cubic)
- much much longer? E.g., $T(n) = 2^n \to T(10n) = 2^{10n} = 1024^n$ (exponential)

How to measure running time?

Scalability: If the input size increases from n to $10n$, will the algorithm take

- the same time? E.g., $T(n) = 42 \to T(10n) = 42$ (constant)
- 10x time? E.g., $T(n) = 15n \to T(10n) = 150n$ (linear)
- 100x time? E.g., $T(n) = 0.3n^2 \to T(10n) = 30n^2$ (quadratic)
- 1000x time? E.g., $T(n) = 82n^3 \to T(10n) = 82000n^3$ (cubic)
- much much longer? E.g., $T(n) = 2^n \to T(10n) = 2^{10n} = 1024^n$ (exponential)

How about the running time for linear search $T(n) = an + b$?

Asymptotic notation

For $T(n) = an + b$, when n approaches infinity, we have

$$\lim_{n\to\infty} \frac{T(10n)}{T(n)} = \lim_{n\to\infty} \frac{10an + b}{an + b} = 10.$$

So, asymptotically, it has the same scaling behavior as $f(n) = 15n$.

Asymptotic notation

For $T(n) = an + b$, when n approaches infinity, we have

$$\lim_{n\to\infty} \frac{T(10n)}{T(n)} = \lim_{n\to\infty} \frac{10an+b}{an+b} = 10.$$

So, asymptotically, it has the same scaling behavior as $f(n) = 15n$.

And they both grow much slower than $g(n) = 0.3n^2$, even though $f(1) = 15 > g(1) = 0.3$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Intuitively, the constant c allows us to ignore multiplicative factors: $15n = O(n)$ because we can choose $c = 20$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Intuitively, the constant c allows us to ignore multiplicative factors: $15n = O(n)$ because we can choose $c = 20$.
- It also helps us to focus on asymptotic growth rate, if f grows no faster than g, then we only need to pick c such that $f(n_0) \leq c \cdot g(n_0)$ for some small initial value n_0.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Intuitively, the constant c allows us to ignore multiplicative factors: $15n = O(n)$ because we can choose $c = 20$.
- It also helps us to focus on asymptotic growth rate, if f grows no faster than g, then we only need to pick c such that $f(n_0) \leq c \cdot g(n_0)$ for some small initial value n_0.
- On the other hand, if f grows faster than g, then no matter how large c is, $f(n)$ will eventually exceed $g(n)$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- $2^{n+1} = O(2^n)$;

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- ◇ $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.
- ◇ $2^{2n} = O(2^n)$;

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.
- $2^{2n} = O(2^n)$; FALSE
- $(n + 10)^3 = O(n^3)$

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- ◇ $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.
- ◇ $2^{2n} = O(2^n)$; FALSE
- ◇ $(n + 10)^3 = O(n^3)$

Proof 1: $(n + 10)^3 \leq (11n)^3 = 11^3 n^3$, for $n_0 = 1$.
Proof 2: $(n + 10)^3 \leq (2n)^3 = 8n^3$, for $n_0 = 10$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- ◇ $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.
- ◇ $2^{2n} = O(2^n)$; FALSE
- ◇ $(n + 10)^3 = O(n^3)$

Proof 1: $(n + 10)^3 \le (11n)^3 = 11^3 n^3$, for $n_0 = 1$.
Proof 2: $(n + 10)^3 \le (2n)^3 = 8n^3$, for $n_0 = 10$.

- ◇ $\log(7n^5) = O(\log n)$

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

◇ $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.

◇ $2^{2n} = O(2^n)$; FALSE

◇ $(n + 10)^3 = O(n^3)$
Proof 1: $(n + 10)^3 \leq (11n)^3 = 11^3 n^3$, for $n_0 = 1$.
Proof 2: $(n + 10)^3 \leq (2n)^3 = 8n^3$, for $n_0 = 10$.

◇ $\log(7n^5) = O(\log n)$
Proof: $\log(7n^5) \leq \log 7 + 5 \log n \leq 12 \log n$, for $n_0 = 2$.

Asymptotic notation: big-O

For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f = O(g)$ if there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.

Examples: (All logs are in base 2 unless another base is specified.)

- $2^{n+1} = O(2^n)$; *Proof.* $2^{n+1} = 2 \cdot 2^n$; can take $c = 2$, $n_0 = 1$.
- $2^{2n} = O(2^n)$; FALSE
- $(n+10)^3 = O(n^3)$

Proof 1: $(n+10)^3 \le (11n)^3 = 11^3 n^3$, for $n_0 = 1$.
Proof 2: $(n+10)^3 \le (2n)^3 = 8n^3$, for $n_0 = 10$.

- $\log(7n^5) = O(\log n)$

Proof: $\log(7n^5) \le \log 7 + 5 \log n \le 12 \log n$, for $n_0 = 2$.

- $n^5 + 1888n^3 + n \log n = O(n^5)$
- $n^5 + 1888n^3 + n \log n = O(n^6)$

Asymptotic notation: big-O

Fact: For any real number $r > 0$, $\log n = O(n^r)$.

Example: $r = 0.01$, $\log n = O(n^{0.01})$.

Proof. Since $\log x \leq x$ for all $x \geq 1$, we have

$$\log n = \frac{1}{r} \log n^r \leq \frac{1}{r} n^r .$$

The equality above holds by the property of logarithms.

Hence one can take $c = 1/r$ and $n_0 = 1$ and the inequality in the definition of big-O is satisfied: we have shown that there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $\log n \leq c\, n^r$ for $n \geq n_0$. $\square$

Question: What are c and n_0 in our example: $\log n = O(n^{0.01})$?

Asymptotic notation: big-O

Fact: For any constants $\alpha, \beta > 0$, $\log^\alpha n = O(n^\beta)$.

Example: $\alpha = 100$, $\beta = 1/10$, $\log^{100} n = O(n^{0.1})$.

Proof. Let $r = \beta/\alpha$; clearly r is a positive constant.
Use the inequality from previous slide:

$\log n = O(n^r)$, or

$\log n \le c_1 n^r = c_1 n^{\beta/\alpha}$, for some (constant) $c_1 > 0$ and $n \ge n_0$.

Raise to the power α:

$$\log^\alpha n \le c_1^\alpha\, n^\beta, \quad \text{for } n \ge n_0, \text{ that is,}$$
$$\log^\alpha n \le c\, n^\beta, \quad \text{for } n \ge n_0,$$

where $c = c_1^\alpha$. □

Asymptotic notation: big-O

Fact: For any fixed $k \geq 0$, $n^k = O(2^n)$.

Example: $k = 1000$, $n^{1000} = O(2^n)$.

Proof. Since $\lim_{x\to\infty} \frac{x}{\log x} = \infty$, there exists $n_0 \in \mathbb{N}$ such that $k \leq \frac{n_0}{\log n_0}$. Hence we can write:

$$n^k \leq n^{\frac{n_0}{\log n_0}} \leq n^{\frac{n}{\log n}} = \left(2^{\log n}\right)^{\frac{n}{\log n}} = 2^n, \text{ for } n \geq n_0.$$

Thus one can take $c = 1$ and n_0 as above and the inequality in the definition of O is satisfied: we have shown that there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that $n^k \leq c\,2^n$ for $n \geq n_0$. $\square$

Question: What are c and n_0 in our example: $n^{1000} = O(2^n)$?