Data Structures and Algorithms (資料結構與演算法)

Lecture 3: Analysis Tools

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

Roadmap

1. the one where it all began

Lecture 2: Data Structures

scheme of purposefully organizing data with access/maintenance algorithms, such as ordered array for faster search

Lecture 3: Analysis Tools

- motivation
- cases of complexity analysis
- asymptotic notation
- usage of asymptotic notation

2. the data structures awaken
3. fantastic trees and where to find them
4. the search revolutions
5. sorting: the final frontier

motivation

Recall: Properties of Good Program

good program: proper use of resources

Space Resources

- memory
- disk(s)
- transmission bandwidth

—space complexity

Computation Resources

- CPU(s)
- GPU(s)
- computation power

—time complexity

need: language for describing complexity

(Extra-)Space Complexity of Get-Min

Get-Min(A)

```
m = 1 // store current min. index
for i = 2 to A.length
    // update if i-th element smaller
    if A[m] > A[i]
        m = i
return A[m]
```

- array A: pointer size s_0 and $n = A.length$ elements
  —extra-space complexity: not counting the input data
- integer m: size s_1
- integer i: size s_1

total space $2s_1$: constant to n

Space Complexity of Get-Min-Waste

Get-Min-Waste(A)

1 $B = \text{Copy}(A, 1, A.\mathit{length})$
2 Insertion-Sort(B)
3 return $B[1]$

- ~~array A: pointer size s_0 and $n = A.\mathit{length}$ elements~~
- array B:
  - pointer size s_0
  - n integers with total size $s_1 \cdot n$, where $n = A.\mathit{length}$
- any space that Insertion-Sort uses: $\square$

total space $s_0 + s_1 n + \square$: (at least) linear to n

Time Complexity of Insertion Sort

	Insertion-Sort(A)	cost	number of times
1	for $m = 2$ to $A.length$	d_1	n
2	$key = A[m]$	d_2	$n-1$
3	// insert $A[m]$ into the sorted sequence $A[1 \,..\, m-1]$	0	$n-1$
4	$i = m-1$	d_4	$n-1$
5	while $i > 0$ and $A[i] > key$	d_5	$\sum_{m=2}^{n} t_m$
6	$A[i+1] = A[i]$	d_6	$\sum_{m=2}^{n}(t_m - 1)$
7	$i = i-1$	d_7	$\sum_{m=2}^{n}(t_m - 1)$
8	$A[i+1] = key$	d_8	$n-1$

(from Introduction to Algorithms Third Edition, Cormen at al.)

total time $T(n)$

$$= d_1 n + d_2(n-1) + d_4(n-1) + d_5 \sum_{m=2}^{n} t_m + d_6 \sum_{m=2}^{n}(t_m - 1) + d_7 \sum_{m=2}^{n}(t_m - 1) + d_8(n-1)$$

actual time $d_\bullet$ depends on machine type;
total $T(n)$ depends on n and t_m, number of while checks

Fun Time

Consider running Get-Min on an array A of length n. If line i takes a time cost of d_i, and the inequality in line 4 is TRUE for t times, what is the time complexity of Get-Min?

```
Get-Min(A)
1  m = 1 // store current min. index
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return A[m]
```

1. $d_1 + d_2 + d_4 + d_5 + d_6$
2. $d_1 + td_2 + td_4 + td_5 + d_6$
3. $d_1 + nd_2 + td_4 + td_5 + d_6$
4. $d_1 + nd_2 + (n-1)d_4 + td_5 + d_6$

Fun Time

Consider running Get-Min on an array A of length n. If line i takes a time cost of d_i, and the inequality in line 4 is TRUE for t times, what is the time complexity of Get-Min?

```
Get-Min(A)
1  m = 1 // store current min. index
2  for i = 2 to A.length
3       // update if i-th element smaller
4       if A[m] > A[i]
5            m = i
6  return A[m]
```

1. $d_1 + d_2 + d_4 + d_5 + d_6$
2. $d_1 + td_2 + td_4 + td_5 + d_6$
3. $d_1 + nd_2 + td_4 + td_5 + d_6$
4. $d_1 + nd_2 + (n-1)d_4 + td_5 + d_6$

Reference Answer: 4

The loop (including ending check) in line 2 is run n times; the condition in line 4 is checked $n-1$ times, and t of those result in execution of line 5.

cases of complexity analysis

Best-case Time Complexity of Insertion Sort

Insertion-Sort(A)	cost	number of times
for $m = 2$ to $A.length$	d_1	n
$key = A[m]$	d_2	$n-1$
// insert $A[m]$ into the sorted sequence $A[1\,..\,m-1]$	0	$n-1$
$i = m-1$	d_4	$n-1$
while $i > 0$ and $A[i] > key$	d_5	$\sum_{m=2}^{n} t_m$
$A[i+1] = A[i]$	d_6	$\sum_{m=2}^{n} (t_m - 1)$
$i = i-1$	d_7	$\sum_{m=2}^{n} (t_m - 1)$
$A[i+1] = key$	d_8	$n-1$

(from Introduction to Algorithms Third Edition, Cormen at al.)

sorted $A \Longrightarrow t_m = 1$

$$
\begin{aligned}
& T(n) \\
= \;& d_1 n + d_2(n-1) + d_4(n-1) + d_5 \sum_{m=2}^{n} t_m + d_6 \sum_{m=2}^{n} (t_m - 1) + d_7 \sum_{m=2}^{n} (t_m - 1) + d_8(n-1) \\
= \;& d_1 n + d_2(n-1) + d_4(n-1) + d_5(n-1) + d_6(0) + d_7(0) + d_8(n-1)
\end{aligned}
$$

best case: $T(n) = \blacksquare \cdot n + \blacklozenge$ (linear to n)

Worst-case Time Complexity of Insertion Sort

Insertion-Sort(A)	cost	number of times
for $m = 2$ to $A.length$	d_1	n
$key = A[m]$	d_2	$n-1$
// insert $A[m]$ into the sorted sequence $A[1\,..\,m-1]$	0	$n-1$
$i = m-1$	d_4	$n-1$
while $i > 0$ and $A[i] > key$	d_5	$\sum_{m=2}^{n} t_m$
$A[i+1] = A[i]$	d_6	$\sum_{m=2}^{n}(t_m - 1)$
$i = i-1$	d_7	$\sum_{m=2}^{n}(t_m - 1)$
$A[i+1] = key$	d_8	$n-1$

(from Introduction to Algorithms Third Edition, Cormen at al.)

reverse-sorted $A \Longrightarrow t_m = m$

$$
\begin{aligned}
& T(n) \\
= \;& d_1 n + d_2(n-1) + d_4(n-1) + d_5 \sum_{m=2}^{n} t_m + d_6 \sum_{m=2}^{n}(t_m - 1) + d_7 \sum_{m=2}^{n}(t_m - 1) + d_8(n-1) \\
= \;& d_1 n + d_2(n-1) + d_4(n-1) + d_5\left(\tfrac{(n+2)(n-1)}{2}\right) + d_6\left(\tfrac{n(n-1)}{2}\right) + d_7\left(\tfrac{n(n-1)}{2}\right) + d_8(n-1)
\end{aligned}
$$

worst case: $T(n) = \bigstar \cdot n^2 + \blacksquare \cdot n + \blacklozenge$ (quadratic to n)

Average-case Time Complexity of Insertion Sort

average case

other cases

$A = [1, 2, 4, 3]$

other cases

$A = [1, 4, 2, 3]$

best cases

$A = [1, 2, 3, 4]$

worst cases

$A = [4, 3, 2, 1]$

$\cdots$

other cases

$A = [4, 3, 1, 2]$

best case $\leq$ average case $\leq$ worst case

Time Complexity Analysis in Practice

Common Focus

worst-case time complexity

- physically meaningful: longest wait time/ max. power consumption
- often ≈ average: when enough near-worst-cases

Common Language

rough time needed w.r.t. input size n

$$T(n) = \star \cdot n^2 + \blacksquare \cdot n + \blacklozenge$$

- care more about
  - larger n
  - leading term of n
- care less about
  - constants
  - other terms of n

next: language of rough notation

Fun Time

Which of the following describes the best-case time complexity of $\mathrm{Get\text{-}Min}$ on an array *A* of length *n*?

```
Get-Min(A)
m = 1 // store current min. index
for i = 2 to A.length
    // update if i-th element smaller
    if A[m] > A[i]
        m = i
return A[m]
```

1. constant to *n*
2. linear to *n*
3. quadratic to *n*
4. none of the other choices

Fun Time

Which of the following describes the best-case time complexity of Get-Min on an array A of length n?

```
Get-Min(A)
1  m = 1 // store current min. index
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return A[m]
```

1. constant to n
2. linear to n
3. quadratic to n
4. none of the other choices

Reference Answer: 2

Even in the best case, where line 5 is executed 0 times, the loop (including ending check) in line 2 still needs to be run n times, and the condition in line 4 still needs to be checked $n - 1$ times.

asymptotic notation

'Rough' Notation

goal

$$\bigstar \cdot n^2 + \blacksquare \cdot n + \blacklozenge \overset{\text{roughly}}{\sim} n^2$$

- care more about
  - larger n
  - leading term of n
- care less about
  - constants
  - other terms of n

notation

$$\underbrace{\bigstar \cdot n^2 + \blacksquare \cdot n + \blacklozenge}_{f(n)} = \Theta(\underbrace{n^2}_{g(n)})$$

for positive $f(n)$ and $g(n)$ [when $n \in \mathbb{R}$ with $n \geq 1$]

extracting the similarity: consider $\frac{f(n)}{g(n)}$

Modeling Roughly with Asymptotic Behavior

goal

$$\underbrace{\bigstar \cdot n^2 + \blacksquare \cdot n + \blacklozenge}_{f(n)} = \Theta(\underbrace{n^2}_{g(n)})$$

- growth of $\blacksquare \cdot n + \blacklozenge$ slower than $g(n) = n^2$:
  for large n, removable by dividing $g(n)$
- asymptotically, two functions only differ by $c > 0$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

—why needing $c > 0$?

'rough' definition ver. 0 (to be changed):

for positive $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$
if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0$

Asymptotic Notation

$$f(n) = \Theta(g(n)) \Longleftarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$$

Θ: $f(n)$ grows roughly the same as $g(n)$

- definition meets criteria:
  - care about larger n: yes, $n \to \infty$
  - leading term more important than other terms: yes, $n + \sqrt{n} + \log n = \Theta(n)$
  - insensitive to constants: yes, $1126n = \Theta(n)$
- "$= \Theta(\cdot)$" actually "$\in$"

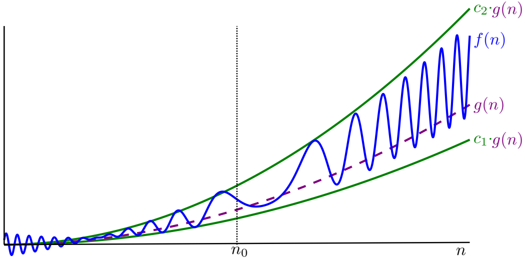
	$\sqrt{n}$	$0.1126n + 6.211$	n	$112.6n$	$n^{1.1}$	$\exp(n)$
is $\Theta(n)$?	×	○	○	○	×	×

asymptotic notation: 'language' for time/space complexity

Issue about the Convergence Definition

$$f(n) = \Theta(g(n)) \Longleftarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$$

- consider a hypothetical algorithm:
  - $T(n) = n$ for even n
  - $T(n) = 2n$ for odd n
- want: $T(n) = \Theta(n)$
- but $\lim_{n\to\infty} \frac{T(n)}{n}$ does not exist!



fixed (formal) definition ver. 1:

for asymptotically non-negative $f(n)$ and $g(n)$,

$f(n) = \Theta(g(n))$ if and only if there exists positive (n_0, c_1, c_2) such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

Convergence Condition $\Rightarrow$ Formal Definition

Theorem: For asymptotically non-negative functions $f(n)$ and $g(n)$,

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ (convergence condition),

then $f(n) = \Theta(g(n))$ (formal definition).

Proof

- definition of $\lim_{n\to\infty}$: for all $\epsilon > 0$, there exists $n_\epsilon > 0$ such that for all $n > n_\epsilon$, $\left|\frac{f(n)}{g(n)} - c\right| < \epsilon$
- choose any $\epsilon' > 0$, and let $n_0' = n_{\epsilon'} + 1$, $c_1' = c - \epsilon'$, $c_2' = c + \epsilon'$
- then for all $n \geq n_0'$, $\underbrace{c - \epsilon'}_{c_1'} < \frac{f(n)}{g(n)} < \underbrace{c + \epsilon'}_{c_2'}$, that is,

$$c_1' \cdot g(n) \leq f(n) \leq c_2' \cdot g(n)$$

- witness (n_0', c_1', c_2') proves $f(n) = \Theta(g(n))$

often suffices to use convergence condition in practice

Fun Time

For asymptotically non-negative functions $f(n)$ and $g(n)$, which of the following condition is sufficient for stating $f(n) = \Theta(g(n))$?

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c > 0$
2. there is (n_0, c_1, c_2) such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$
3. $g(n) = \Theta(f(n))$
4. all of the other choices

Fun Time

For asymptotically non-negative functions $f(n)$ and $g(n)$, which of the following condition is sufficient for stating $f(n) = \Theta(g(n))$?

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c > 0$
2. there is (n_0, c_1, c_2) such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$
3. $g(n) = \Theta(f(n))$
4. all of the other choices

Reference Answer: 4

1 is the convergence condition;

2 is the formal definition of Θ;

3 can be proved by converting the witness (n_0, c_1, c_2) for $g(n) = \Theta(f(n))$ to the witness $(n_0, \frac{1}{c_2}, \frac{1}{c_1})$ for $f(n) = \Theta(g(n))$.

usage of asymptotic notation

The Seven Functions as $g(n)$

popular choices

- $g(n) = 1$: constant
  —meaning $c_1 \leq f(n) \leq c_2$ for $n \geq n_0$
- $g(n) = \log n$: logarithmic
  —does base matter?
- $g(n) = n$: linear
- $g(n) = n \log n$
- $g(n) = n^2$: square
- $g(n) = n^3$: cubic
- $g(n) = 2^n$: exponential
  —does base matter?

will often encounter them in future classes

Logarithmic Function in Asymptotic Notation

Claim: base does not matter for logarithmic function

For any $a > 1$, $b > 1$, if $f(n) = \Theta(\log_a n)$, then $f(n) = \Theta(\log_b n)$.

Proof

- $f(n) = \Theta(\log_a n)$: $\exists(c_1 > 0, c_2 > 0, n_0 > 0)$ such that $c_1 \log_a n \leq f(n) \leq c_2 \log_a n$ for $n \geq n_0$.
- Then, $c_1 \cdot \underbrace{\log_a b \cdot \log_b n}_{\log_a n} \leq f(n) \leq c_2 \cdot \log_a b \cdot \log_b n$ for $n \geq n_0$.
- Note that $\log_a b > 0$ because $a > 1$ and $b > 1$.
- Let $c_1' = c_1 \log_a b > 0$, $c_2' = c_2 \log_a b > 0$, $n_0' = n_0 > 0$. Then, (n_0', c_1', c_2') witnesses

$$c_1' \log_b n \leq f(n) \leq c_2' \log_b n$$

for $n \geq n_0'$, thus proving $f(n) = \Theta(\log_b n)$.

base does not matter in $\Theta(\log n)$

Exponential Function in Asymptotic Notation

Claim: base does not matter for logarithmic function

For any $a > b > 1$ with , if $f(n) = \Theta(a^n)$, then $f(n) \neq \Theta(b^n)$.

Proof

- (prove by contradiction)
  First, assume that $f(n) = \Theta(a^n)$ AND $f(n) = \Theta(b^n)$.
- Then, by definition,
  - $\exists(c_1 > 0, c_2 > 0, n_0 > 0)$ such that $\boxed{c_1 a^n \leq f(n)} \leq c_2 a^n$ for $n \geq n_0$.
  - $\exists(c_1' > 0, c_2' > 0, n_0' > 0)$ such that $c_1' b^n \leq \boxed{f(n) \leq c_2' b^n}$ for $n \geq n_0'$.
- Thus, for arbitrarily big $n \geq \max(n_0, n_0')$, $\boxed{c_1 a^n \leq f(n) \leq c_2' b^n}$
- Take log on both sides: $\log c_1 + n \log a \leq \log c_2' + n \log b$, which implies that $n \leq \frac{\log c_2' - \log c_1}{\log a - \log b}$ because $a > b$.
- That is, n cannot be arbitrarily big. CONTRADICTION!

base matters in $\Theta(a^n)$

Analysis of Sequential Search

```
Seq-Search(A, key)
1   n = A.length
2   for i = 1 to n
3       // return when found
4       if A[i] equals key
5           return i
6   return nil
```

- best case (i.e. *key* at $A[1]$): $T(n) = \Theta(1)$
  —lines 1-5 executed once with constant time d_1 to d_5, remember? :-)
- worst case (i.e. return nil): $T(n) = \Theta(n)$
  —lines 2 for $n+1$ times, lines 3-4 for n times, others constant

often # of loop iterations dominates!

Analysis of Binary Search

```
Bin-Search(A, key, ℓ, r)
while ℓ ≤ r
    m = floor((ℓ + r)/2)
    if A[m] equals key
        return m
    elseif A[m] > key
        r = m − 1 // cut out end
    elseif A[m] < key
        ℓ = m + 1 // cut out begin
return nil
```

for $n = r - \ell + 1$

- best case (i.e. *key* at first *m*): $T(n) = \Theta(1)$
- worst case (i.e. return nil): $T(n) = \Theta(\log_2 n)$ because

$$T(n) = T(\lceil \tfrac{n-1}{2} \rceil) + \text{‘constant’}$$

often care more about worst case, as mentioned

Fun Time

What is the time complexity of Get-Min on an array A of length n?

```
Get-Min(A)
m = 1 // store current min. index
for i = 2 to A.length
    // update if i-th element smaller
    if A[m] > A[i]
        m = i
return A[m]
```

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n^2)$

Fun Time

What is the time complexity of Get-Min on an array A of length n?

```
Get-Min(A)
1  m = 1 // store current min. index
2  for i = 2 to A.length
3      // update if i-th element smaller
4      if A[m] > A[i]
5          m = i
6  return A[m]
```

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n^2)$

Reference Answer: 3

The loop (including ending check) in line 2 is run n times (regardless of the best case or the worst case), remember? :-)

Summary

Lecture 3: Analysis Tools

- motivation
  roughly quantify time/space complexity (efficiency)
- cases of complexity analysis
  often focus on worst-case with ‘rough’ notations
- asymptotic notation
  rough comparison of function for large n
- usage of asymptotic notation
  describe $f(n)$ for time or space by simpler $g(n)$

- next: more asymptotic notations for realistic use