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

Lecture 5: 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)

motivation

Properties of Good Programs

Good Program

- meet requirements, correctness: basic
- clear usage document (external), readability (internal), etc.

Proper Resource Usage

- efficient use of computation resources (CPU, GPU, etc.)?
  time complexity
- efficient use of storage resources (memory, disk, etc.)?
  space complexity

need: language for describing complexity

Space Complexity of GET-MIN-INDEX

GET-MIN-INDEX(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 m
```

- array A: pointer size d_1 (not counting the actual input elements)
- integer m: size d_2
- integer i: size d_2

total space $d_1 + 2d_2$ (constant)
within algorithm execution: not dependent to $n = A.length$

Space Complexity of GET-MIN-INDEX-WASTE

GET-MIN-INDEX-WASTE(A)

```
B = COPY(A, 1, A.length)
INSERTION-SORT(B)
return B[1]
```

- array A: pointer size d_1 (not counting the actual input elements)
- array B: pointer size d_1 and $n = A.length$ integers $n \cdot d_2$

total space $2d_1 + d_2 n$ (linear): dependent to n

Time Complexity of Insertion Sort

INSERTION-SORT(A)	cost	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	$n-1$

total time $T(n)$

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

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

cases of complexity analysis

Best-case Time Complexity of Insertion Sort

INSERTION-SORT(A)	cost	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	$n-1$

sorted $A \Longrightarrow t_m = 1$

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

$T(n) = bn + a$ (linear) for some a, b in best case

Worst-case Time Complexity of Insertion Sort

INSERTION-SORT(A)	cost	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	$n-1$

reverse-sorted $A \Longrightarrow t_m = m$

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

$T(n) = cn^2 + bn + a$ (quadratic)
for some c, b, a in worst case

Time Complexity Analysis in Reality

Common Focus

worst-case time complexity

- meaningful in practice (waiting time)
- similar to average-case when near-worst-case often enough

Common Language

'rough' time needed w.r.t. n (input size)

- care about larger n
- leading (bigger) term more important
- insensitive to constants

next: introduce the language of 'rough' notation

asymptotic notation

'Rough' Notation

want:

$$cn^2 + bn + a \overset{\text{roughly}}{\sim} n^2$$

- care about larger n
- leading term more important
- insensitive to constants

will have

$$\underbrace{cn^2 + bn + a}_{f(n)} = \Theta(\underbrace{n^2}_{g(n)})$$

for positive $f(n)$ and $g(n)$ [when $n \geq 1$]

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

Representing 'Rough' by Asymptotic Behavior

want:

$$\underbrace{cn^2 + bn + a}_{f(n)} = \Theta(\underbrace{n^2}_{g(n)})$$

- growth of $bn + a$ slower than $g(n) = n^2$: removed by dividing $g(n)$ for large 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 version 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: Modeling Rough Growth

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

big-Θ: roughly the same

- definition meets criteria:
  - care about larger n: yes, $n \to \infty$
  - leading term more important: yes, $n + \sqrt{n} + \log n = \Theta(n)$
  - insensitive to constants: yes, $1126n = \Theta(n)$
- meaning: $f(n)$ grows roughly the same as $g(n)$
- "$= \Theta(\cdot)$" actually "$\in$"

	$\sqrt{n}$	$0.1126n$	n	$112.6n$	$n^{1.1}$	$\exp(n)$
$\Theta(n)$?	N	Y	Y	Y	N	N

asymptotic notation:
the most used '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!

fix (formal): for asymptotically non-negative f(n) & g(n)

$$f(n) = \Theta(g(n)) \iff \text{exists positive } (n_0, c_1, c_2) \text{ such that } c_1 g(n) \le f(n) \le c_2 g(n) \text{ for } n \ge n_0$$

Convergence 'Definition' $\Rightarrow$ Formal Definition

For asymptotically non-negative functions $f(n)$ and $g(n)$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, then $f(n) = \Theta(g(n))$.

- with definition of limit, there exists $\epsilon > 0, n_0 > 0$ such that for all $n \geq n_0$, $|\frac{f(n)}{g(n)} - c| < \epsilon$.
- i.e. for all $n \geq n_0$, $c - \epsilon < \frac{f(n)}{g(n)} < c + \epsilon$.
- Let $c_1' = c - \epsilon$, $c_2' = c + \epsilon$, $n_0' = n_0$, formal definition satisfied with (c_1', c_2', n_0'). QED

often suffices to use convergence 'definition' in practice

usage of asymptotic notation

The Seven Functions as $g(n)$

$g(n) = ?$

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

will often encounter them in future classes

Logarithmic Function in Asymptotic Notation

Claim

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

Proof

- $f(n) = \Theta(\log_a n) \iff \exists(c_1, c_2, n_0)$ such that $c_1 \log_a n \leq f(n) \leq c_2 \log_a n$ for $n \geq n_0$
- Then, $c_1 \log_a b \log_b n \leq f(n) \leq c_2 \log_a b \log_b n$ for $n \geq n_0$
- Let $c_1' = c_1 \log_a b$, $c_2' = c_2 \log_a b$, $n_0' = n_0$, we get $f(n) = \Theta(\log_b n)$

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

Analysis of Sequential Search

```
SEQ-SEARCH(A, key)
for i = 1 to A.length
    // return when found
    if A[i] equals key
        return i
return NIL
```

- best case (i.e. *key* at 1): $T(n) = \Theta(1)$
- worst case (i.e. return NIL): $T(n) = \Theta(n)$
- average case with respect to uniform $key \in A$: $\mathbb{E}(T(n)) = \Theta(n)$

iterations in loop: dominating often

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
```

- best case (i.e. *key* at first m): $T(n) = \Theta(1)$
- worst case (i.e. return NIL): because range $(r - \ell + 1)$ roughly halved in each **while** , # iterations roughly $\log_2 n$: $T(n) = \Theta(\log n)$

often care more about worst case, as mentioned

other asymptotic notations

Big-O Notation

binary search: best case $\Theta(1)$, worst case $\Theta(\log n)$
—for 'any' binary search task, needed time roughly no more than $\log n$

Big-O Notation

$f(n)$ grows slower than or similar to $g(n)$: ("$\leq$")

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

iff exist *positive*(c_2, n_0) such that $f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$

—one side of $\Theta(\cdot)$ definition

usage: binary search is $O(\log n)$-time

Re-prove 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
```

- worst case (i.e. return NIL): because range $(r - \ell)$ more than halved in each **while** , # iterations to make $(r - \ell) < 1$ less than $\log_2 n + 1$: $T(n) = O(\log n)$

big-O analysis on worst case often done in practice

Three Big Asymptotic Notations

- $f(n)$ grows slower than or similar to $g(n)$: ("$\leq$")

  $f(n) = O(g(n))$, iff exist c, n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

- $f(n)$ grows faster than or similar to $g(n)$: ("$\geq$")

  $f(n) = \Omega(g(n))$, iff exist c, n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

- $f(n)$ grows similar to $g(n)$: ("$\approx$")

$$f(n) = \Theta(g(n)), \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

usually only need big-O

Sequential and Binary Search

- Input: **any** integer array A with size n, an integer *key*
- Output: if *key* not within A, YES; otherwise, NIL

```
DIRECT-SEQ-SEARCH(A, key)

if SEQ-SEARCH(A, key) = NIL
        return NIL
else
        return YES
```

```
SORT-THEN-BIN-SEARCH(A, key)
B = SEL-SORT(A)
if BIN-SEARCH(A, key) = NIL
        return NIL
else
        return YES
```

- DIRECT-SEQ-SEARCH: $O(n)$ time
- SORT-THEN-BIN-SEARCH: $O(n^2)$ time for SEL-SORT and $O(\log n)$ time for BIN-SEARCH

next: operations on asymptotic complexity

operations on asymptotic complexity

Transitivity of Big-O

Theorem

if $f(n) = O(g(n))$, $g(n) = O(h(n))$ *then* $f(n) = O(h(n))$

Proof

- When $n \geq n_1$, $f(n) \leq c_1 g(n)$
- When $n \geq n_2$, $g(n) \leq c_2 h(n)$
- So, when $n \geq \max(n_1, n_2)$, $f(n) \leq c_1 c_2 h(n)$

similar for Ω and Θ

Closure of Big-O

Theorem

if $f(n) = O(h(n))$, $g(n) = O(h(n))$ *then* $f(n) + g(n) = O(h(n))$

Proof

- When $n \geq n_1$, $f(n) \leq c_1 h(n)$
- When $n \geq n_2$, $g(n) \leq c_2 h(n)$
- So, when $n \geq \max(n_1, n_2)$, $f(n) + g(n) \leq (c_1 + c_2)h(n)$

again, similar for Ω and Θ

Sort-Then-Bin-Search Revisited

```
SORT-THEN-BIN-SEARCH(A, key)
B = SEL-SORT(A)
if BIN-SEARCH(A, key) = NIL
        return NIL
else
        return YES
```

- SEL-SORT: $O(n^2)$ time
- BIN-SEARCH: $O(\log n)$ time, hence $O(n^2)$ by transitivity
- SORT-THEN-BIN-SEARCH: overall $O(n^2)$ time, by closure

operations: allow dividing pseudo code to smaller pieces to analyze

practical complexity

A Bit More on Big-O

is $O(\log n)$ time complexity

- by transitivity, time-complexity also $O(n)$
- time also $O(n \log n)$
- time also $O(n^2)$
- also $O(2^n)$
- ...

prefer the tightest Big-O!

Comparison of Complexity

	(consecutive) array	linked list
index access	$O(1)$	$O(n)$
head insertion	$O(n)$	$O(1)$
tail insertion	$O(1)$	$O(1)$ after getting *tail*
'middle' insertion	$O(n)$	$O(1)$ after getting node
wasted space	$O(1)$ for $A.length$	$O(n)$ for *next*

be familiar with (and don't be afraid to) Big-O

Practical Complexity

some input sizes are time-wise **infeasible** for some algorithms

when 1-billion-steps-per-second

n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	$0.01\mu s$	$0.03\mu s$	$0.1\mu s$	$1\mu s$	$10\mu s$	$10s$	$1\mu s$
20	$0.02\mu s$	$0.09\mu s$	$0.4\mu s$	$8\mu s$	$160\mu s$	$2.84h$	$1ms$
30	$0.03\mu s$	$0.15\mu s$	$0.9\mu s$	$27\mu s$	$810\mu s$	$6.83d$	$1s$
40	$0.04\mu s$	$0.21\mu s$	$1.6\mu s$	$64\mu s$	$2.56ms$	$121d$	$18m$
50	$0.05\mu s$	$0.28\mu s$	$2.5\mu s$	$125\mu s$	$6.25ms$	$3.1y$	$13d$
100	$0.10\mu s$	$0.66\mu s$	$10\mu s$	$1ms$	$100ms$	$3171y$	$4\cdot 10^{13}y$
10^3	$1\mu s$	$9.96\mu s$	$1ms$	$1s$	$16.67m$	$3\cdot 10^{13}y$	$3\cdot 10^{284}y$
10^4	$10\mu s$	$130\mu s$	$100ms$	$1000s$	$115.7d$	$3\cdot 10^{23}y$	
10^5	$100\mu s$	$1.66ms$	$10s$	$11.57d$	$3171y$	$3\cdot 10^{33}y$	
10^6	$1ms$	$19.92ms$	$16.67m$	$32y$	$3\cdot 10^7y$	$3\cdot 10^{43}y$	

note: similar for space complexity,
e.g. store an N by N double matrix when $N = 50000$?

Summary

Lecture 5: Analysis Tools

- motivation

quantify time/space complexity to measure 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 $f(n)$ (time, space) by simpler $g(n)$

- other asymptotic notations

big-O more popular than Θ

- operations on asymptotic complexity

analyze pseudo code by pieces

- practical complexity

how much more resource needed?