More on Basic Concepts

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 08–09, 2011

What We Have Done

- Historical Note
- From Programming to Coding
- What is Algorithm? (Five Criteria)
- What is Data Structure?
- Why Learn Data Structure and Algorithm?
- The SEL-SORT Algorithm and Correctness Proof
- Reading Assignment:
  Pointer, Dynamic Memory Allocation, Recursive Algorithm

Why Correctness Proof?

這是好問題 pass 下

確定基礎正確性

以理服人

Correctness: Shortest Path versus Longest Path

- Input: a directed graph G; vertices S and T; positive distance d
- Output: the shortest/longest simple path from S to T

SHORTEST(S, T)

if $S == T$
  return empty path
else
  $\min_V |\text{SHORTEST}(S, V)| + d(V, T)$
  return SHORTEST(S, V) $\rightarrow d(V, T)$
end if

LONGEST(S, T)

if $S == T$
  return empty path
else
  $\max_V |\text{LONGEST}(S, V)| + d(V, T)$
  return LONGEST(S, V) $\rightarrow d(V, T)$
end if

intuition:
  sub-path of shortest/longest path is a shortest/longest path.
formal proof:
  SHORTEST is correct;
  LONGEST is not always correct.

More on Sequential and Binary Search

- Input: a **sorted** integer array *list* with size *n*, an integer *searchnum*
- Output: if *searchnum* is within *list*, its index; otherwise −1

SEQ-SEARCH (*list*, *n*, *searchnum*)

```
for i ← 0 to n − 1 do
   if list[i] == searchnum
      return i
   end if
end for
return −1
```

BIN-SEARCH (*list*, *n*, *searchnum*)

```
left ← 0, right ← n − 1
while left ≤ right do
   middle ← floor((left + right)/2)
   if list[middle] > searchnum
      right ← middle − 1
   else if list[middle] < searchnum
      left ← middle + 1
   else        /* list[middle] == searchnum */
      return middle
   end if
end while
return −1
```

SEQ-SEARCH(*list*, *n*, *searchnum*)

```
for i ← 0 to n − 1 do
    if list[i] == searchnum
        return i
    end if
end for
return −1
```

$l[0]$	$l[1]$	$l[2]$	$l[3]$	$l[4]$	$l[5]$	$l[6]$
1	3	4	9	9	10	13

- search for 9
- search for 15

Binary Search: Eliminate at Least Half Each Time

BIN-SEARCH(*list*, *n*, *searchnum*)

```
left ← 0, right ← n − 1
while left ≤ right do
    middle ← floor((left + right)/2)
    if list[middle] > searchnum
        right ← middle − 1
    else if
    list[middle] < searchnum
        left ← middle + 1
    else
        return middle
    end if
end while
return −1
```

$l[0]$	$l[1]$	$l[2]$	$l[3]$	$l[4]$	$l[5]$	$l[6]$
1	3	4	9	9	10	13

- search for 9
- search for 15

Another Version of Binary Search

BIN-SEARCH ($list, n, searchnum$)

```
left ← 0, right ← n − 1
while left ≤ right do
   middle ← floor((left + right)/2)
   if list[middle] > searchnum
      right ← middle − 1
   else if list[middle] < searchnum
      left ← middle + 1
   else
      return middle
   end if
end while
return −1
```

BIN-SEARCH ($list, l, r, s$)

```
if l ≤ r
   m ← floor((l + r)/2)
   if list[m] > s
      return BIN-SEARCH(list, l, m − 1, s)
   else if list[m] < s
      return BIN-SEARCH(list, m + 1, r, s)
   else
      return m
   end if
end if
return −1
```

iterative versus recursive

Properties of Good Programs

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

Resource Usage (Performance)

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

Space Complexity of List Summing

LIST-SUM(float array *list*, integer length *n*)

$tempsum \leftarrow 0$
for $i \leftarrow 0$ to $n - 1$ **do**
  $tempsum \leftarrow tempsum + list[i]$
end for
return *tempsum*

- array *list*: size of pointer, commonly 4
- integer *n*: commonly 4
- float *tempsum*: 4
- integer *i*: commonly 4
- float return place: 4

total space 20 (constant), does not depend on *n*

Space Complexity of Recursive List Summing

RECURSIVE-LIST-SUM(float array *list*, integer length *n*)

```
if n = 0
    return 0
else
    return list[n] + RECURSIVE-LIST-SUM(list, n − 1)
end if
```

- array *list*: size of pointer, commonly 4
- integer *n*: commonly 4
- float return place: 4

only 12, better than previous one? (NO, why?)

Space Complexity of Recursive List Summing

RECURSIVE-LIST-SUM(float array *list*, integer length *n*)

if $n = 0$
  return 0
else
  return $list[n]+$ RECURSIVE-LIST-SUM($list, n - 1$)
end if

Calling RECURSIVE-LIST-SUM($\ell, 3$)

list	*n*	return place	total bytes

$12(n + 1)$ actually

Time Complexity of Matrix Addition

MATRIX-ADD
(integer matrix *a*, *b*, result integer matrix *c*, integer *rows*, *cols*)

for $i \leftarrow 0$ to $rows - 1$ **do**
  for $j \leftarrow 0$ to $cols - 1$ **do**
    $c[i][j] \leftarrow a[i][j] + b[i][j]$
  end for
end for

- inner for: $R = P \cdot cols + Q$
- total: $(S + R) \cdot rows + T$

$$P \cdot rows \cdot cols + (Q + S) \cdot rows + T$$

(see textbook for a tabular way of counting)

Rough Time Complexity of Matrix Addition

$$P \cdot rows \cdot cols + (Q + S) \cdot rows + T$$

P, Q, R, S, T hard to keep track and not matter much

MATRIX-ADD
(integer matrix a, b, result integer matrix c, integer $rows$, $cols$)

for $i \leftarrow 0$ to $rows - 1$ **do**
  for $j \leftarrow 0$ to $cols - 1$ **do**
    $c[i][j] \leftarrow a[i][j] + b[i][j]$
  end for
end for

- inner for: $R = P \cdot cols + Q = \Theta(cols)$
- total: $(S + R) \cdot rows + T = \Theta(\Theta(cols) \cdot rows)$

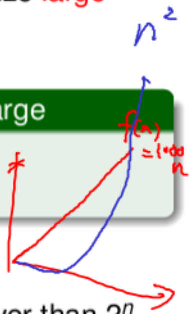
rough total: $\Theta(rows \cdot cols)$

Asymptotic Notations: One Way for Rough Total

- goal: rough total rather than exact steps when input size large
- why rough total? constant not matter much

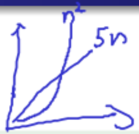
compare two complexity functions $f(n)$ and $g(n)$ when n large

growth of functions matters
—n^3 would eventually be bigger than $1000n$



- n^2 grows much faster than n
- n grows much slower than n^2, which grows much slower than 2^n
- $3n$ grows “slightly faster” than n
  —when constant not matter, $3n$ grows similarly to n

Asymptotic Notations: Symbols



- $f(n)$ grows slower than or similar to $g(n)$: $f(n) = O(g(n))$
- $f(n)$ grows faster than or similar to $g(n)$: $f(n) = \Omega(g(n))$
- $f(n)$ grows similar to $g(n)$: $f(n) = \Theta(g(n))$

- $n = O(n)$; $n = O(10n)$; $n = O(0.3n)$; $n = O(n^2)$; $n = O(n^5)$; $\cdots$ (note: $=$ more like "$\in$")
- $n = \Omega(n)$; $n = \Omega(0.2n)$; $n = \Omega(5n)$; $n = \Omega(\log n)$; $n = \Omega(\sqrt{n})$; $\cdots$
- $n = \Theta(n)$; $n = \Theta(0.1n + 4)$; $n = \Theta(7n)$; $n \neq \Theta(5^n)$

$n^2 = O(n^2 + n + 1)$ $\quad$ $n^2 = \Omega(5n)$

$n^3 = \Omega(n^3)$ $\quad$ $n^3 = \Omega(n^2)$

Asymptotic Notations: Definitions

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

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

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

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

- $f(n)$ grows similar to $g(n)$:

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

Analysis of Sequential Search

Sequential Search

for $i \leftarrow 0$ to $n-1$ **do**
  if $list[i] == searchnum$
    return i
  end if
end for
return -1

- best case (e.g. *searchnum* at 0): time $\Theta(1)$
- worst case (e.g. *searchnum* at last or not found): time $\Theta(n)$
- in general: time $\Omega(1)$ and $O(n)$

Analysis of Binary Search

Binary Search

```
left ← 0, right ← n − 1
while left ≤ right do
    middle ← floor((left + right)/2)
    if list[middle] > searchnum
        left ← middle + 1
    else if
    list[middle] < searchnum
        right ← middle − 1
    else
        return middle
    end if
end while
return −1
```

- best case (e.g. *searchnum* at *middle*): time $\Theta(1)$
- worst case (e.g. *searchnum* not found):
  because $(right - left)$ is halved in each WHILE iteration, needs time $\Theta(\log n)$ iterations if not found
- in general:
  time $\Omega(1)$ and $O(\log n)$

$\log_2 n = (\log_2 10) \log_{10} n$

often care about the worst case (and thus see $O(\cdot)$ often)

More Examples in the Textbook

- PERMUTATION (Example 1.20)
- MAGIC-SQUARE (Example 1.21)

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Sequential and Binary Search

- Input: any integer array *list* with size *n*, an integer *searchnum*
- Output: if *searchnum* is not within *list*, -1; otherwise, othernum

DIRECT-SEQ-SEARCH (*list*, *n*, *searchnum*)

```
for i ← 0 to n − 1 do
    if list[i] == searchnum
        return i
    end if
end for
return −1
```

SORT-AND-BIN-SEARCH (*list*, *n*, *searchnum*)

```
SEL-SORT(list, n)
return BIN-SEARCH(list, n, searchnum)
```

- DIRECT-SEQ-SEARCH is $O(n)$ time
- SORT-AND-BIN-SEARCH is $O(n^2)$ time for SEL-SORT (Why?) and $O(\log n)$ time for BIN-SEARCH

want: show asymptotic complexity of SORT-AND-BIN-SEARCH as its bottleneck

Some Properties of Big-Oh I

Theorem (封閉律)

if $f_1(n) = O(g_2(n))$, $f_2(n) = O(g_2(n))$ *then* $f_1(n) + f_2(n) = O(g_2(n))$

- When $n \geq n_1$, $f_1(n) \leq c_1 g_2(n)$
- When $n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$
- So, when $n \geq \max(n_1, n_2)$, $f_1(n) + f_2(n) \leq (c_1 + c_2) g_2(n)$

Theorem (遞移律)

if $f_1(n) = O(g_1(n))$, $g_1(n) = O(g_2(n))$ *then* $f_1(n) = O(g_2(n))$

- When $n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$
- When $n \geq n_2$, $g_1(n) \leq c_2 g_2(n)$
- So, when $n \geq \max(n_1, n_2)$, $f_1(n) \leq c_1 c_2 g_2(n)$

Some Properties of Big-Oh II

Theorem (併吞律)

if $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$ *and* $g_1(n) = O(g_2(n))$ *then* $f_1(n) + f_2(n) = O(g_2(n))$

Proof: use two theorems above.

Theorem (Theorem 1.2 of Textbook)

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, *then* $f(n) = O(n^m)$

Proof: use the theorem above.

similar proof for Ω and Θ

Some More on Big-Oh

RECURSIVE-BIN-SEARCH is $O(\log n)$ time and $O(\log n)$ space

- by 遞移律, time also $O(n)$
- time also $O(n \log n)$
- time also $O(n^2)$
- also $O(2^n)$
- $\cdots$

prefer the tightest Big-Oh!

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 $N = 50000$?

Performance Measurement (Sec. 1.6)

Suggested Reading
(but NOT Assignment)

if you want to learn more, read it!