Performance Analysis: Asymptotic Complexity

Data Structures

Fall 2018

Insertion Sort

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   int t = a[i];
   int j;
   for (j = i - 1; j >= 0 && t < a[j]; j--)
       a[j + 1] = a[j];
   a[j + 1] = t;
}
```

Worst-Case Comparison Count

```
for (int i = 1; i < n; i++)
  for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

#comparisons = 1 + 2 + 3 + … + (n-1)

= (n-1)n/2

Step Count

A step is an amount of computation that does not depend on the instance characteristic n

For example, 100 adds, 10 subtractions, 100 multiplications can all be counted as a single step

n adds cannot be counted as 1 step

Step Count

	s/e
`for (int i = 1; i < a.length; i++)`	1
`{// insert a[i] into a[0:i-1]`	0
`    int t = a[i];`	1
`    int j;`	0
`    for (j = i - 1; j >= 0 && t < a[j]; j--)`	1
`        a[j + 1] = a[j];`	1
`    a[j + 1] = t;`	1
`}`	0

s/e – steps per execution

Step Count

s/e isn't always 0 or 1

```
x = MyMath.sum(a, n);
// returns the sum of all the elements in a[0,n-1]
```

where n is the instance characteristic

has an s/e count of n

Step Count

Code	s/e	steps
`for (int i = 1; i < a.length; i++)`		
`{// insert a[i] into a[0:i-1]`	0	
`    int t = a[i];`	1	1
`    int j;`	0	
`    for (j = i - 1; j >= 0 && t < a[j]; j--)`	1	i+ 1
`        a[j + 1] = a[j];`	1	i
`    a[j + 1] = t;`	1	1
`}`	0	

Total: 2i+3

Step Count

```
for (int i = 1; i < a.length; i++)
{ 2i + 3 }
```

Suppose a.length = n

Step count for body of for loop is

$2(1+2+3+\ldots+n-1) + 3(n-1)$

$= (n-1)n + 3(n-1)$

$= (n-1)(n+3)$

Exercise: Prefix sums

- Given an array a[0,n-1], write an efficient procedure that constructs a new array b[0,n-1] such that

  b[i] = a[0]+a[1]+…+a[i].

  What is the step count of the procedure?

Asymptotic Complexity

[Finding the exact step count or operation count is cumbersome and time consuming.]

- Describes the behavior of the time (or space) complexity for *large* instance characteristics.

- Useful to compare the growth of different functions (i.e., time/space complexities of different procedures).

Big Oh Notation

- $f(n) = O(g(n))$ (read as "f(n) is big oh of g(n)") iff positive constants c and k exist such that $f(n) \leq c.g(n)$ for all $n \geq k$.

- $f(n)$ is $O(g(n))$ means $f(n)$ grows asymptotically slower than or at the same rate as $g(n)$.

- That is, $g(n)$ is an upper bound for $f(n)$.

  [Note: "$O(g(n)) = f(n)$" is meaningless]

Asymptotic Complexity of Insertion Sort

- Step count = (n-1)(n+3) = $n^2 + 2n - 3$
- Asymptotic complexity is **$O(n^2)$**
- What does this mean?

Complexity of Insertion Sort

- Time or number of operations does not exceed $\mathbf{c.n^2}$ on any input of size **n** (**n** suitably large).

- [$n^2+2n-3 \leq 2\,n^2$ for all positive integers n (i.e., c = 2 and k = 1)]

- So, the worst-case time at most quadruples each time **n** is doubled

Complexity of Insertion Sort

- Is **$O(n^2)$** too much time?
- Is the algorithm practical?

Practical Complexities

10^9 instructions/second

n	***n***	***nlogn***	***n^2***	***n^3***
1000	1mic	10mic	1milli	1sec
10000	10mic	130mic	100milli	17min
10^6	1milli	20milli	17min	32years

Impractical Complexities

10^9 instructions/second

n	n^4	n^{10}	2^n
1000	17min	3.2×10^{13} years	3.2×10^{283} years
10000	116 days	???	???
10^6	3×10^7 years	??????	??????

Faster Computer Vs Better Algorithm

Algorithmic improvement more useful than hardware improvement.

E.g. 2^n to n^3

Fibonacci numbers

- $F(0) = F(1) = 1$; $F(n) = F(n-1) + F(n-2)$
- Write a program to compute the n-th Fibonacci number.

- Alg 1:
```
int fib(n) { if ((n==0)||(n==1)) return 1;
                    else return fib(n-1)+fib(n-2);}
```

- Complexity: $O(F(n))$

Fibonacci numbers

- Alg 2:
```
int fib2(n) { int [] F = new int [n];
              F[0] = 1; F[1] = 1;
              for(i=2; i<=n; i++)
                  F[i] = F[i-1]+F[i-2];
              return F[n];
            }
```
- Complexity: O(n)

More asymptotic notation

- $f(n) = \Omega(g(n))$ means $f(n)$ is asymptotically bigger than or equal to $g(n)$

  i.e., $g(n)$ is a lower bound for $f(n)$
- $f(n) = \Theta(g(n))$ means $f(n)$ is asymptotically equal to $g(n)$, i.e., $g(n)$ is both an upper and a lower bound for $f(n)$
- Also $o()$ (little-oh) and $\omega()$ (little omega) for describing *strict* upper and lower bounds.

Binary search

- Input: A sorted array of n distinct numbers and another number x
- Output: The index i such that $a[i] \le x < a[i+1]$

- Algorithm: Repeatedly bisect the range [0,n-1] till the index i is found.
- What is the complexity of binary search?

O(log n)

Exercise

- What is the complexity of the number of comparisons (between the elements of the input array) performed by “binary insertion sort”?

Binary insertion sort

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   int x = a[i];
   j = BinSearch(x,a[0:i-1]);
   // insert x into a[0:i-1] at position j
}
```

Binary insertion sort

#comparisons

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   int x = a[i];                                    0
   j = BinSearch(x,a[0:i-1]);                 O(log i+1)
   // insert x into a[0:i-1] at position j         0
}
```

Binary insertion sort

- Total number of comparisons =

  O(log 2 + log 3 + … + log n+1) =

  O(log (2*3*…*n+1) =

  O(log ((n+1)!)) ≈ O(n log n).

Exercise: Show that log (n!) = Θ(n log n).