CS 401: Computer Algorithm I

Running Time Analysis

Xiaorui Sun

Last Lecture: O and Ω Notations

Given two positive functions **f** and **g**

- **f(N)** is **O(g(N))** iff there is a constant $c>0$ s.t.,
  f(N) is eventually always $\leq c\, g(N)$
- **f(N)** is **Ω(g(N))** iff there is a constant $c>0$ s.t.,
  f(N) is eventually always $\geq c\, g(N)$

Question: If f_1 is $\Omega(g_1)$ and f_2 is $\Omega(g_2)$, Is $f_1 + f_2$ $\Omega(g_1 + g_2)$?

Answer: Yes

Θ-Notation

Given two positive functions **f** and **g**

- **f(N)** is **Θ(g(N))** iff there are **$c_0 > 0$, $c_1 > 0$** and **$N_0 \geq 0$** s.t.
  $c_0 \cdot g(N) \leq f(N) \leq c_1 \cdot g(N)$ for all **$N \geq N_0$**

E.g. **$f(N)=32N^2+17N+1$**

- **f(N)** is **$\Theta(N^2)$.** ⟵ Choose $c_0=32$, $c_1=50$, $N_0=1$
- **f(N)** is neither **$\Theta(N)$** nor **$\Theta(N^3)$.**

Typical usage: Gale-Sharpley makes $\Theta(n^2)$ proposals in the worst case.

Summary

Given two positive functions **f** and **g**

- **f(N)** is **O(g(N))** iff there is a constant **c>0** s.t.,
  f(N) is eventually always $\leq$ **c g(N)**

- **f(N)** is **Ω(g(N))** iff there is a constant **c >0** s.t.,
  f(N) is eventually always $\geq$ **c g(N)**

- **f(N)** is **Θ(g(N))** iff there are constants c_1, c_2>0 so that
  eventually always $c_1 g(N) \leq f(N) \leq c_2 g(N)$

Practice 1

Suppose $f(n) = n!, g(n) = 2^n$

Is $f = O(g)$?

Definition: f(N) is **O(g(N))** iff there is a constant **c>0** and $\mathbf{N_0 \geq 0}$ s.t., $\mathbf{0 \leq f(N) \leq c \cdot g(N)}$ for all $\mathbf{N \geq N_0}$

=> If $\frac{f(n)}{g(n)} \leq c$ for all large enough n, then f is $O(g)$

But if as n increases, f/g also increases (sometimes can be verified by your calculator), then f is not $O(g)$

Practice 1

Suppose $f(n) = n!, g(n) = 2^n$

Is $f = O(g)$?

Definition: f(N) is **O(g(N))** iff there is a constant **c>0** and **$N_0 \geq 0$** s.t., **$0 \leq f(N) \leq c \cdot g(N)$** for all **$N \geq N_0$**

=> If $\frac{f(n)}{g(n)} \leq c$ for all large enough n, then f is $O(g)$

$$\frac{f(n)}{g(n)} = \underbrace{\left(\frac{1}{2}\right)\left(\frac{2}{2}\right) \dots \left(\frac{n}{2}\right)}_{n \text{ terms}} \geq \underbrace{\left(\frac{n}{4}\right) \dots \left(\frac{n}{2}\right)}_{n/2 \text{ terms}} \geq \left(\frac{n}{4}\right)^{\frac{n}{2}}$$

Which is bigger than any constant c for large enough n.

So, f is not $O(g)$.

Practice 2

Question: $f = n, g = 2^{(\log_2 n)^2}$, Is $f\ O(g)$?

Approach 1: As n increases, f/g approaches 0 (can be verified by your calculator)

So, $f = O(g)$

Practice 2

Question: $f = n, g = 2^{(\log_2 n)^2}$, Is f $O(g)$?

Property: For two functions f and g, if $\log f$ is $O(\log g)$, but $\log g$ is not $O(\log f)$ then f is $O(g)$.

Approach 2: $\log_2 f = \log_2 n, \log_2 g = (\log_2 n)^2,$ and thus $\log f$ is $O(\log g)$, $\log g$ is not $O(\log f)$, so f is $O(g)$

Question: $f = n, g = 2^{0.9 \log_2 n}$, Is f $O(g)$?

$\log_2 f = \log_2 n, \log_2 g = 0.9 \log_2 n$, $\log f$ is $O(\log g)$, $\log g$ is also $O(\log f)$, we cannot conclude f is $O(g)$

A Survey of Common Running Times

Linear Time: O(n)

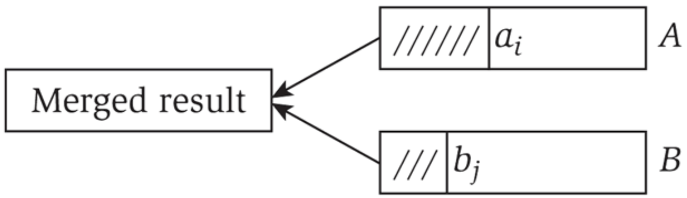
Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else(ai > bj)append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

don't need to take square roots

see chapter 5

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Polynomial Time: O(n^k) Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

O(n^k) solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = O(k^2).
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- O(k^2 n^k / k!) = O(n^k).

poly-time for k=17,
but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2\ 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Efficiency

An algorithm runs in polynomial time if $T(n) = n^{O(1)}$.

Equivalently, $T(n) = O(n^d)$ for some constant d.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest or largest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
quasilinear time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort; Fast Fourier transform.
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort; Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\epsilon})$ for all $\epsilon > 0$	$O(2^{\log n^{\log \log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization ~~and graph isomorphism~~
exponential time (with linear exponent)	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Why it matters?

Suppose we can do 1 million operations per second.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

not only get very big, but do so *abruptly*, which likely yields erratic performance on small instances

Outdated: Nvidia announced a “computer” this Tue that do 2 quadrillion (2×10^{15}) operations/sec.
It brings down the 31,710 years to 500 sec.
However, 2^{100} operations still takes millions of years.

Why “Polynomial”?

Point is not that n^{2000} is a practical bound, or that the differences among n and 2n and n^2 are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

- “My problem is in P” is a starting point for a more detailed analysis
- “My problem is not in P” may suggest that you need to shift to a more tractable variant

Summary

Asymptotic notations: **O, Ω, Θ**

Efficient algorithm: polynomial running time