Data Structures and Algorithms

Lecture 4: Analysis of Algorithms

Introduction

- What is Algorithm?
  - a clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code
- Data structures
  - Methods of organizing data
- Program = algorithms + data structures

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Example: Selection Problem

- Given a list of N numbers, determine the *k*th largest, where $k \leq N$.
- Algorithm 1:

  (1) Read N numbers into an array

  (2) Sort the array in decreasing order by some simple algorithm

  (3) Return the element in position k

Example: Selection Problem…

- Algorithm 2:
  (1) Read the first k elements into an array and sort them in decreasing order
  (2) Each remaining element is read one by one
    - If smaller than the kth element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
  (3) The element in the kth position is returned as the answer.

Example: Selection Problem…

- Which algorithm is better when
  - N =100 and k = 100?
  - N =100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

Algorithm Analysis

- We only analyze *correct* algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

Algorithm Analysis…

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem $\Rightarrow$ the number of items to be sorted
      - E.g. multiply two matrices together $\Rightarrow$ the total number of elements in the two matrices
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations $\Rightarrow$ Not parallel computers

Example

- Calculate

$$\sum_{i=1}^{N} i^3$$

```
  int sum(int n)
  {
        int partialSum;

1       partialSum=0;                 1
2       for (int i=1;i<=n;i++)        2N+2
3          partialSum += i*i*i;       4N
4       return partialSum;            1
  }
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 $\Rightarrow$ O(N)

Worst- / average- / best-case

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input

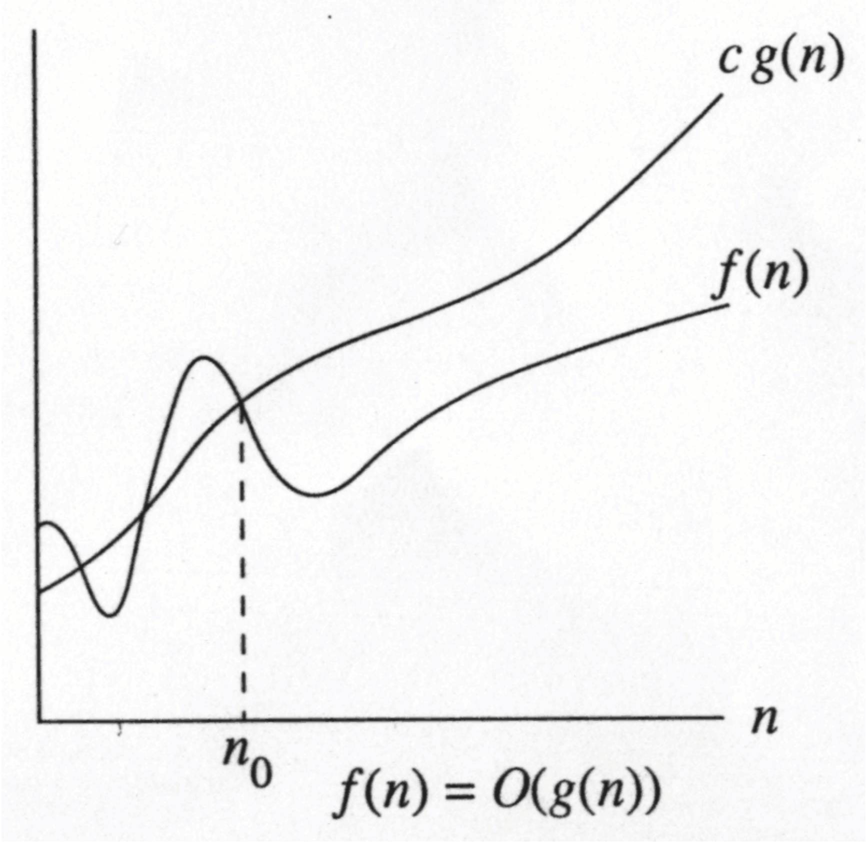
    ⇒ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

Running-time of algorithms

- Bounds are for the algorithms, rather than programs
  - programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds

- Bounds are for algorithms, rather than problems
  - A problem can be solved with several algorithms, some are more efficient than others

Growth Rate



- The idea is to establish a relative order among functions for large n
- $\exists$ c , n_0 > 0 such that $f(N) \le c\, g(N)$ when $N \ge n_0$
- f(N) grows no faster than g(N) for "large" N

Asymptotic notation: Big-Oh

- $f(N) = O(g(N))$
- There are positive constants c and n_0 such that

$$f(N) \leq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of f(N) is *less than or equal to* the growth rate of g(N)
- g(N) is an upper bound on f(N)

Big-Oh: example

- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”

Big Oh: more examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1)$; $10 = O(1)$, $10^{10} = O(1)$
- $$\sum_{i=1}^{N} i \le N \cdot N = O(N^2)$$
  $$\sum_{i=1}^{N} i^2 \le N \cdot N^2 = O(N^3)$$
- $\log N + N = O(N)$
- $\log^k N = O(N)$ for any constant k
- $N = O(2^N)$, but 2^N is not $O(N)$
- 2^{10N} is not $O(2^N)$

Math Review: logarithmic functions

$$x^a = b \quad iff \quad \log_x b = a$$

$$\log ab = \log a + \log b$$

$$\log_a b = \frac{\log_m b}{\log_m a}$$

$$\log a^b = b \log a$$

$$a^{\log n} = n^{\log a}$$

$$\log^b a = (\log a)^b \neq \log a^b$$

$$\frac{d \log_e x}{dx} = \frac{1}{x}$$

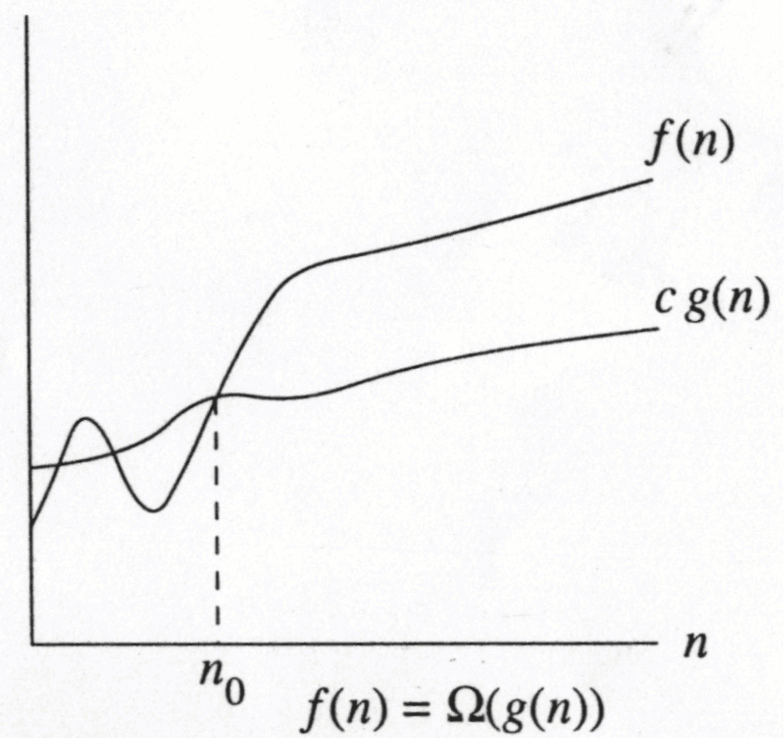
Some rules

When considering the growth rate of a function using Big-Oh

- Ignore the lower order terms and the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If $T_1(N) = O(f(N)$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)),\ O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Big-Omega



- $\exists$ c , n_0 > 0 such that f(N) $\geq$ c g(N) when N $\geq$ n_0
- f(N) grows no slower than g(N) for "large" N

Big-Omega

- $f(N) = \Omega(g(N))$
- There are positive constants c and n_0 such that

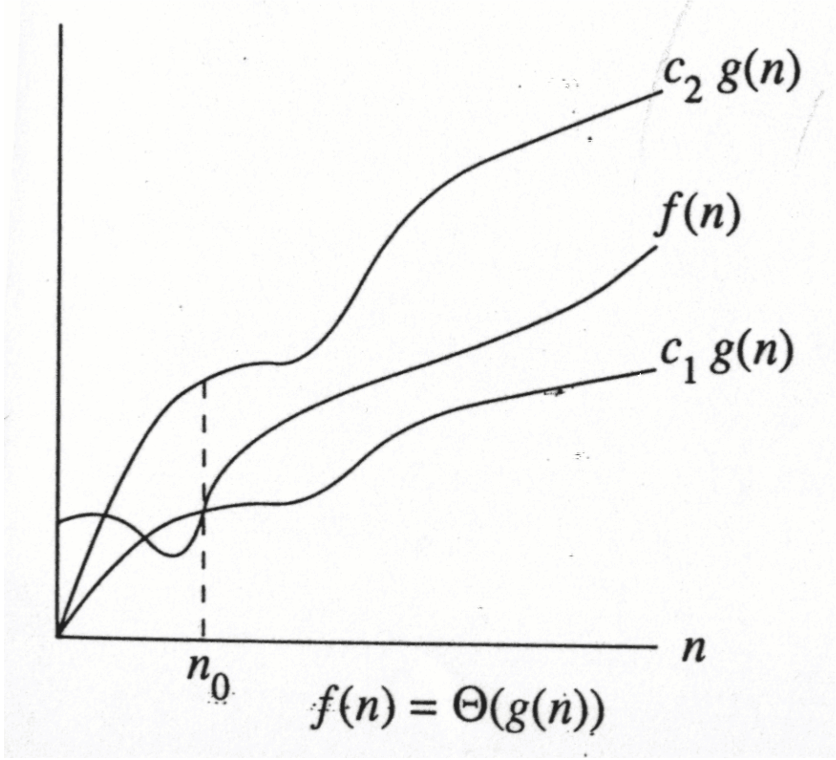
$$f(N) \geq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of f(N) is *greater than or equal to* the growth rate of g(N).

Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

$f(N) = \Theta(g(N))$



- the growth rate of f(N) *is the same as* the growth rate of g(N)

Big-Theta

- $f(N) = \Theta(g(N))$ iff

  $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
- The growth rate of f(N) *equals* the growth rate of g(N)
- Example: Let $f(N)=N^2$, $g(N)=2N^2$
  - Since $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$,

    thus $f(N) = \Theta(g(N))$.
- Big-Theta means the bound is the tightest possible.

Some rules

- If $T(N)$ is a polynomial of degree k, then

  $T(N) = \Theta(N^k)$.

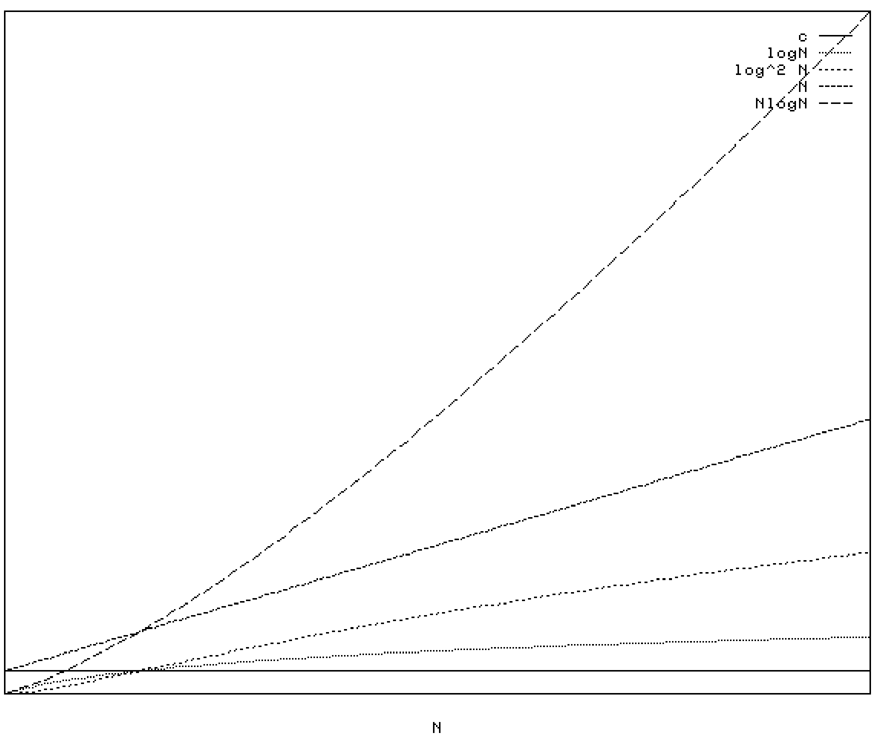
- For logarithmic functions,

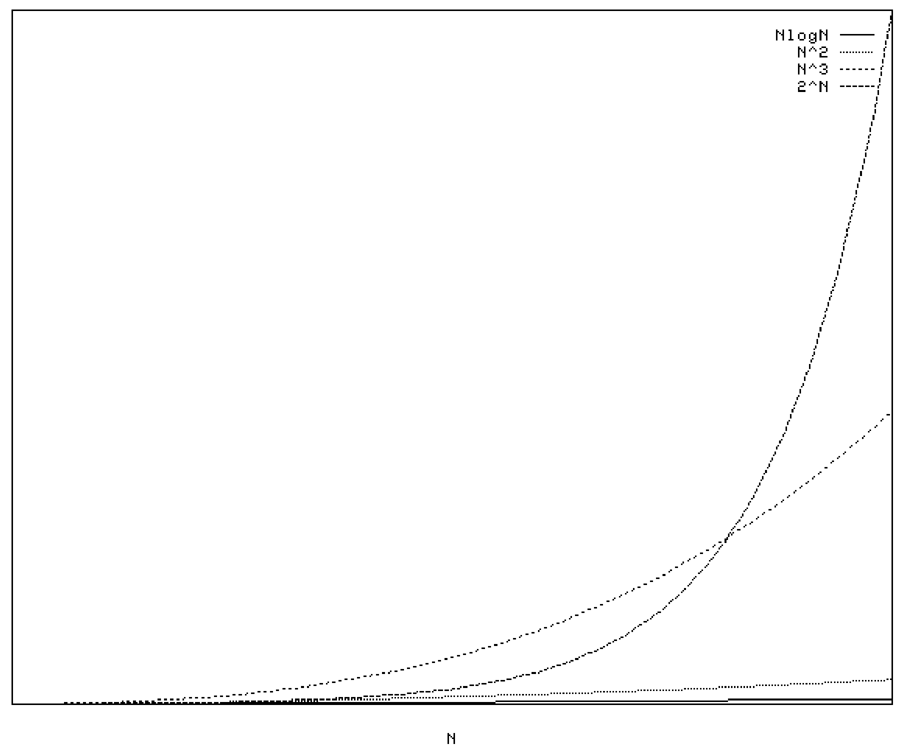
  $T(\log_m N) = \Theta(\log N)$.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates





Growth rates ...

- Doubling the input size
  - $f(N) = c \Rightarrow f(2N) = f(N) = c$
  - $f(N) = \log N \Rightarrow f(2N) = f(N) + \log 2$
  - $f(N) = N \Rightarrow f(2N) = 2\, f(N)$
  - $f(N) = N^2 \Rightarrow f(2N) = 4\, f(N)$
  - $f(N) = N^3 \Rightarrow f(2N) = 8\, f(N)$
  - $f(N) = 2^N \Rightarrow f(2N) = f^2(N)$
- Advantages of algorithm analysis
  - To eliminate bad algorithms early
  - pinpoints the bottlenecks, which are worth coding carefully

Using L' Hopital's rule

- L' Hopital's rule
  - If $\lim_{n\to\infty} f(N) = \infty$ and $\lim_{n\to\infty} g(N) = \infty$

    then $$\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$$

- Determine the relative growth rates (using L' Hopital's rule if necessary)
  - compute $$\lim_{n\to\infty} \frac{f(N)}{g(N)}$$
  - if 0: f(N) = O(g(N)) and f(N) is not Θ(g(N))
  - if constant ≠ 0: f(N) = Θ(g(N))
  - if ∞: f(N) = Ω(f(N)) and f(N) is not Θ(g(N))
  - limit oscillates: no relation

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General rules (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
    a[i]=0;
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        a[i] += a[j]+i+j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$
- `If S1`

  `Else S2`
  - never more than the running time of the test plus the larger of the running times of S1 and S2.

Another Example

- Maximum Subsequence Sum Problem
- Given (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum value of $$\sum_{k=i}^{j} A_k$$
  - For convenience, the maximum subsequence sum is 0 if all the integers are negative
- E.g. for input –2, 11, -4, 13, -5, -2
  - Answer: 20 (A_2 through A_4)

Algorithm 1: Simple

- Exhaustively tries all possibilities (brute force)

```
int maxSubSum1 (const vector<int> &a)
{
        int maxSum=0;

        for (int i=0;i<a.size();i++)
                for (int j=i;j<a.size();j++)
                {
                        int thisSum=0;

                        for (int k=i;k<=j;k++)
                                thisSum += a[k];

                        if (thisSum > maxSum)
                                maxSum = thisSum;
                }
        return maxSum;
}
```

- $O(N^3)$

Algorithm 2: Divide-and-conquer

- Divide-and-conquer
  - split the problem into two roughly equal subproblems, which are then solved **recursively**
  - patch together the two solutions of the subproblems to arrive at a solution for the whole problem

First half				Second half			
4	-3	5	-2	-1	2	6	-2

- The maximum subsequence sum can be
  - Entirely in the left half of the input
  - Entirely in the right half of the input
  - It crosses the middle and is in both halves

Algorithm 2 (cont'd)

The first two cases can be solved recursively

- For the last case:
  - find the largest sum in the first half that includes the last element in the first half
  - the largest sum in the second half that includes the first element in the second half
  - add these two sums together

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

1	-5	4	2	-7	3	6	-1	‖	2	-4	7	-10	2	6	1	-3

Algorithm 2 …

// Input : $A[i \dots j]$ with $i \leq j$

// Output : the MCS of $A[i \dots j]$

$MCS(A, i, j)$

1. If $i == j$ return $A[i]$ **O(1)**
2. Else
3. Find $MCS(A, i, \lfloor \frac{i+j}{2} \rfloor)$; **T(m/2)**
4. Find $MCS(A, \lfloor \frac{i+j}{2} \rfloor + 1, j)$; **T(m/2)**
5. Find MCS that contains **O(m)**
   both $A\left[\lfloor \frac{i+j}{2} \rfloor\right]$ and $A\left[\lfloor \frac{i+j}{2} \rfloor + 1\right]$;
6. Return Maximum of the three sequences found **O(1)**

Algorithm 2 (cont'd)

- Recurrence equation

$$T(1) = 1$$

$$T(N) = 2T(\frac{N}{2}) + N$$

  - 2 T(N/2): two subproblems, each of size N/2
  - N: for "patching" two solutions to find solution to whole problem

Algorithm 2 (cont'd)

- Solving the recurrence:

$$\begin{aligned} T(N) &= 2T(\frac{N}{2}) + N \\ &= 4T(\frac{N}{4}) + 2N \\ &= 8T(\frac{N}{8}) + 3N \\ &= \cdots \\ &= 2^k T(\frac{N}{2^k}) + kN \end{aligned}$$

- With k=log N (i.e. 2^k = N), we have

$$\begin{aligned} T(N) &= N\,T(1) + N \log N \\ &= N \log N + N \end{aligned}$$

- Thus, the running time is O(N log N)
  - faster than Algorithm 1 for large data sets