Mathematical Background and Running Time Analysis

EECS 233

Previous Lecture: Mathematical Definitions

- $T(N) = \boldsymbol{O}(f(N))$
  - if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ for all $N \geq n_0$
- $T(N) = \boldsymbol{\Omega}(f(N))$
  - if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ for all $N \geq n_0$
- $T(N) = \boldsymbol{\Theta}(f(N))$
  - iff $T(N) = \mathbf{O}(f(N))$ and $T(N) = \boldsymbol{\Omega}(f(N))$
- $T(N) = \boldsymbol{o}(f(N))$
  - if for *all* constants c there exists an n_0 such that $T(N) < cf(N)$ for all $N > n_0$
  - OR iff $T(N) = \mathbf{O}(f(N))$ and $T(N) \neq \boldsymbol{\Omega}(f(N))$

How to Determine the Relative Growth Rate?

- If $\lim_{N\to\infty} T(N)/f(N)$
  - $= 0$: $T(N) = \boldsymbol{o}(f(N))$ (and $O(f(N))$ for sure)
  - $= c$!= 0: $T(N) = \boldsymbol{\Theta}(f(N))$
  - = infinity: $f(N) = \boldsymbol{o}(T(N))$

Big-O Toolbox

- Constants do not matter!
  - If $T(N) = O(f(N) + c))$ then $T(N) = O(f(N))$
  - If $T(N) = O(c*f(N))$ then $T(N) = O(f(N))$
  - If $T(N) = g(N) + c$ and $g(N) = O(f(N))$ then $T(N) = O(f(N))$
  - If $T(N) = c*g(N)$ and $g(N) = O(f(N))$ then $T(N) = O(f(N))$
- Algebraic properties:
  - If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N)+T2(N) = O(f(N) + g(N))$
  - If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N)*T2(N) = O(f(N) * g(N))$
- Dominated terms do not matter:
  - If $T(N) = O(f(N) + g(N))$ and $g(N) = o(f(N))$ then $T(N) = O(f(N))$

Logarithm Properties

- Non-negative, monotonic function for N ≥ 1
- Sub linear growth: $\log(X) = o(X)$
- Basic equivalencies:

$$\log(cd) = \log(c) + \log(d)$$
$$\log(c/d) = \log(c) - \log(d)$$
$$\log(c^d) = d\log(c)$$
$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

Corollary: logarithm grows slower than *any* polynomial!

$$\log(N) = o(N^c) \Leftrightarrow \log(\log(N) = o(\log(N^c) = o(c\log(N))$$
$$\Leftrightarrow \log Y = o(cY) = o(Y)$$

Relative Growth Rates

- Examples (which grows faster?)
  - 1000000 versus 0.01*sqrt(N)
  - log(N) versus sqrt(N)
  - $N\log(N)$ versus $N^{1.001}$
  - N^3 versus 10000*N^2
  - $\log^2(N)$ versus 10*log(N^5)
  - 2*$\log_2(N)$ versus $\log_3(N)$
  - N*2^N versus 3^N

$N^{1.001} = N * N^{0.001}$

$10*\log(N^5) = 50*\log(N)$

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

$3^N = 1.5^N * 2^N$

Algorithm Analysis

- Models and Assumptions
  - Consider rather abstract algorithm (a procedure or method)
  - Ignore the details/specifics of a computer
  - Assume sequential process (a sequence of instructions)
  - Ignore small constant factors (e.g., different instructions)
  - Ignore language differences (e.g., C++ versus Java)
  - Running time is more important than memory space

- Average-case versus worst-case performance
  - Example: finding a number in an array using sequential search
    - ✓ Best-case?
    - ✓ Worst-case?
    - ✓ Average-case?

How to Calculate Running Time?

- A simple example: compute $f(N) = \sum_{i=1}^{N} i^3$

```
public static int sum(int n)
{
   int partialSum;
   partialSum = 0;
   for (int i = 1; i <= n; i++)
      partialSum += i*i*i;
   return partialSum;
}
```

- Assume each operation takes unit time, total time is 6N+4
- *Time complexity* $O(N)$
- More accurately $\Theta(N)$ in this example

General Rules

- Simple statement: constant
  i++; i<n; etc.

- Simple loops: # of iterations times the cost of the loop body
  i = 0; While (i < n) { … i++; …}

- Nested loops: take the product of the # of iterations time the cost of the inner loop body

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

- Consecutive statements: count the more expensive one

```
i = 0;
while (i < n) { … i++; …}
for (i=0; i<n; i++)
    for(j=0; j<n; j++)
                k++;
```

- If/else statement: count the more expensive branch (for worst-case analysis)

Recursion – Factorial Example

- Recursion often makes analysis more difficult.
- A simple one:

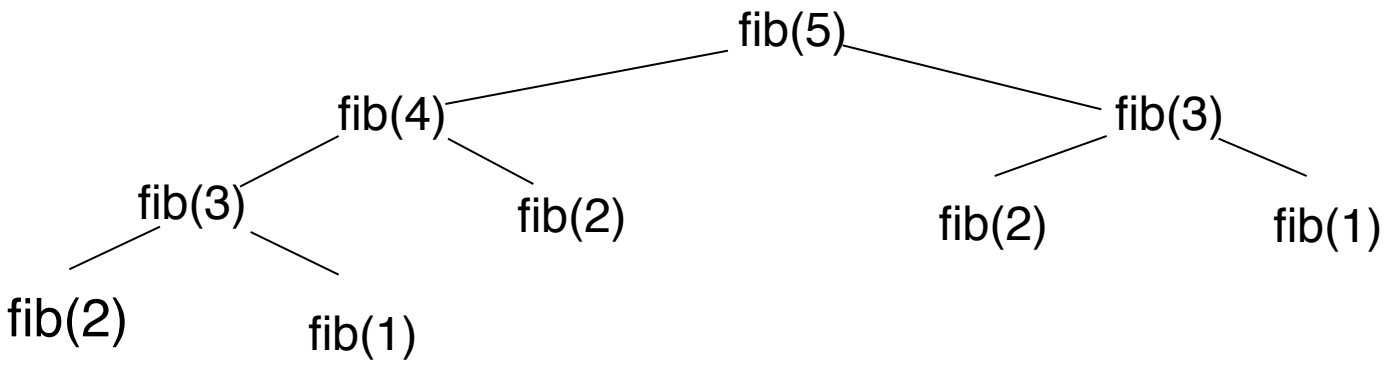
```
public static long factorial (int n)
{
   if (n <= 1 )
      return 1;
   else
      return n * factorial ( n - 1 );
}
```

- Count the number of recursive calls (and the number of multiplications)
- Simple loop achieves the same running time (in fact less expensive by a constant factor)

Recursion – Fibonacci Example

```
public static long fib( int n )
{
   if (n <= 1)
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
}
```

- Redundant calculation of the same Fibonacci numbers, results in exponential running time
- To analyze it, notice T(N) = T(N-1)+T(N-2)+Constant
- It increases exponentially (between $(3/2)^N$ and $(5/3)^N$)



An Example: Maximum Subsequence Sum

- Given an array of (possibly negative) integers A[1…N], find the maximum value sum of sub-array A[i…j].

  example: 4 5 -6 -6 2 3 5 -8 -1 4 3 7 -2

Algorithm 1

- Cubic running time: exhaustively try all possibilities

```
public static int maxSubSum1( int [ ] a )
{
   int maxSum = 0;
   for (int i = 0; i < a.length; i++)
      for( int j = i; j < a.length; j++)
      {
         int thisSum 0;
         for ( int k = i; k <= j; k++)
            thisSum += a[k];
         if ( thisSum > maxSum )
            maxSum = thisSum;
      }
}
```

4 5 -6 -6 2 3 5 -8 -1 4 3 7 -2

- Running time $O(N^3)$

Algorithm 2

- Quadratic running time

```
public static int maxSubSum2( int [] a)
{
  int maxSum = 0;
  for( int i = 0; i < a.length; i++ )
  {
    int thisSum = 0;
    for ( int j = i; j < a.length; j++)
    {
      thisSum += a[j];  // thisSum has sum of a[i to j]
      if(thisSum > maxSum )
        maxSum = thisSum;
    }
  }
}
```

4 5 -6 -6 2 3 5 -8 -1 4 3 7 -2

- Eliminate the inner loop and reduce the redundant additions
- Running time $O(N^2)$

Algorithm 3

- Linear running time (lowest possible, why?)

```
int maxSubSum4( int [ ]  a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.length ); j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 ) // The first negative subsequence
            thisSum = 0;
    }

    return maxSum;
}
```

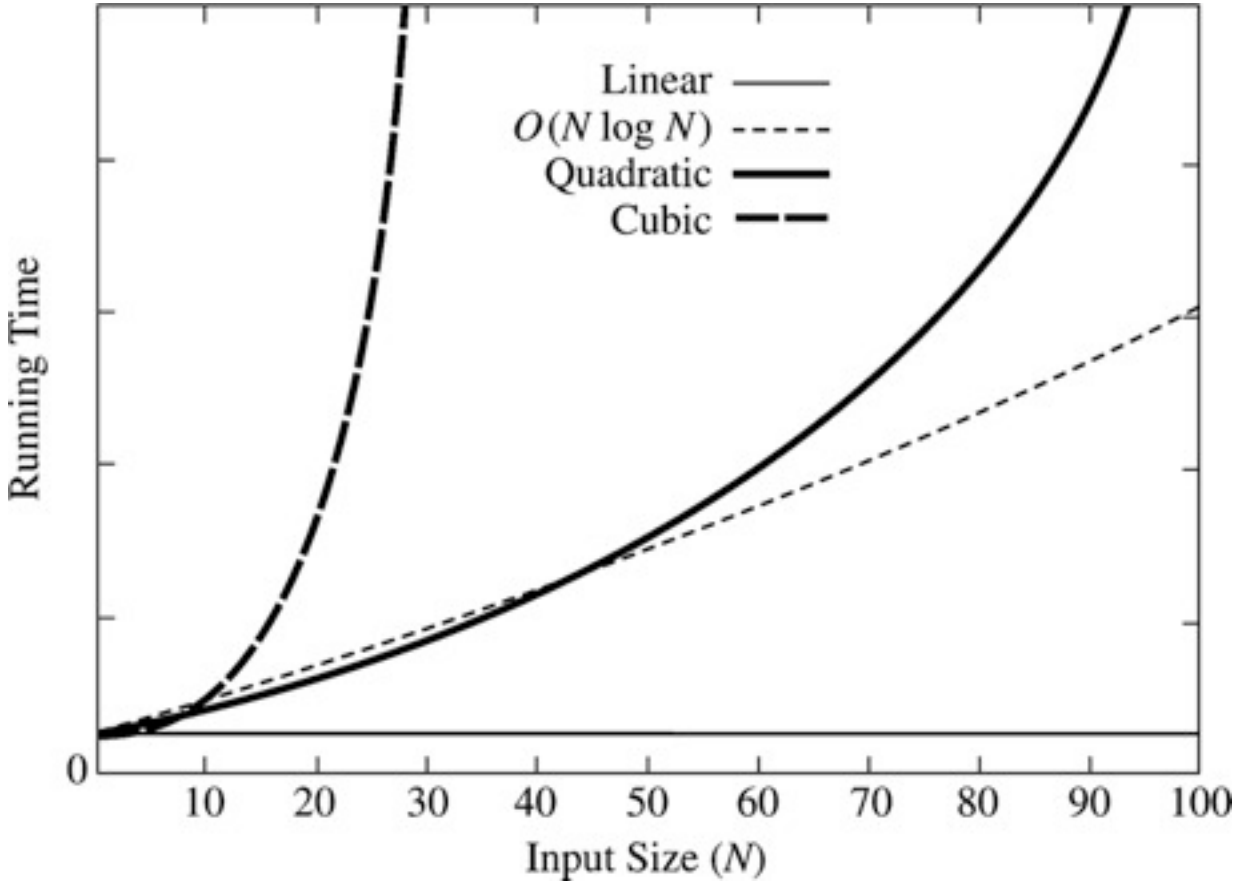
- No need to identify the subsequence – only to find its sum!
- The moment we find a negative subsequence – dump it and start anew from this point
  - It can't start the max subsequence
  - Neither can any of its suffixes

4 5 -6 -6 2 3 5 -8 -1 4 3 7 -2

- Why does it produce the correct result?

Comparison of Running Time

- Four algorithms



Another Example: Euclid's Algorithm

- Find the Greatest Common Divisor of two positive integers (m and n)

```
int gcd( int m, int n )
{
    while( n != 0 )
    {
        int rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

- Example: m=1989 and n=1590; remainders are 399, 393, 6, 3, 0
- Logarithmic running time. Why?
- Theorem 2.1: If M > N, then M mod N < M/2
  - If N <= M/2, then since the remainder is smaller than N, the theorem is true.
  - If N > M/2, then N goes into M once with a remainder M-N < M/2, proving the theorem.

Is Your Analysis Correct?

- Analysis can be complex (e.g., for recursive algorithms)
  - Mistakes possible
- How to verify your analysis?
  - Implement the algorithm
  - Measure running time
- Problems:
  - Constants may skew results for small inputs
  - Need to test asymptotic behavior (hence need large inputs)
  - Measurements usually test average time complexity; analysis usually provides worst-time complexity

Example: the Probability of Two Integers (<= n) Being Relatively Prime

```
public static double probRelPrim( int n )
  {
     int rel = 0, tot = 0;

     for( int i = 1; i <= n; i++ )
        for( int j = i + 1; j <= n; j++ )
        {
           tot++;
           if ( gcd( i, j ) == 1 )
              rel++;
        }

     return (double) rel / tot;
  }
```

Time complexity: $N^2 \log N$

N	CPU time (T)	T/N^2	T/N^3	$T/(N^2 \log N)$
100	022	.002200	.000022000	.0004777
200	056	.001400	.000007000	.0002642
300	118	.001311	.000004370	.0002299
400	207	.001294	.000003234	.0002159
500	318	.001272	.000002544	.0002047
600	466	.001294	.000002157	.0002024
700	644	.001314	.000001877	.0002006
800	846	.001322	.000001652	.0001977
900	1,086	.001341	.000001490	.0001971
1,000	1,362	.001362	.000001362	.0001972
1,500	3,240	.001440	.000000960	.0001969
2,000	5,949	.001482	.000000740	.0001947
4,000	25,720	.001608	.000000402	.0001938

N	T (μsecs)	T/N^2	T/N^3	T/(N^2 log N)
10	7.959e+01	0.7959	0.0796	0.3457
20	1.315e+02	0.3287	0.0164	0.1097
50	7.083e+02	0.2833	0.0057	0.0724
100	3.117e+03	0.3117	0.0031	0.0677
200	1.208e+04	0.3019	0.0015	0.0570
500	7.757e+04	0.3103	0.0006	0.0499
1000	3.233e+05	0.3233	0.0003	0.0468
2000	1.352e+06	0.3379	0.0002	0.0445
5000	8.796e+06	0.3518	0.0001	0.0413
10000	3.649e+07	0.3649	0.0000	0.0396
20000	1.611e+08	0.4028	0.0000	0.0407

Etc.

- Assignment #1
  - Basics of object-oriented programming, algorithm analysis, recursion
  - Please submit hard copies in class
  - Blackboard's digital dropbox is OK for late assignments or when absent
    - ✓ File name format: EECS233_msl88_W1_YourLastName_YourCaseID