1.204 Lecture 5

Algorithms: analysis, complexity

Algorithms

- **Algorithm:**
  - Finite set of instructions that solves a given problem.
  - Characteristics:
    - Input. Zero or more quantities are supplied.
    - Output. At least one quantity is computed.
    - Definiteness. Each instruction is computable.
    - Finiteness. The algorithm terminates with the answer or by telling us no answer exists.
- **We will study common algorithms in engineering design and decision-making**
  - We focus on problem modeling and algorithm usage
  - Variations in problem formulation lead to greatly different algorithms
    - E.g., capital budgeting can be greedy (simple) or mixed integer programming (complex)

Algorithms: forms of analysis

- How to devise an algorithm
- How to validate the algorithm is correct
  - Correctness proofs
- How to analyze running time and space of algorithm
  - Complexity analysis: asymptotic, empirical, others
- How to choose or modify an algorithm to solve a problem
- How to implement and test an algorithm in a program
  - Keep program code short and correspond closely to algorithm steps

Analysis of algorithms

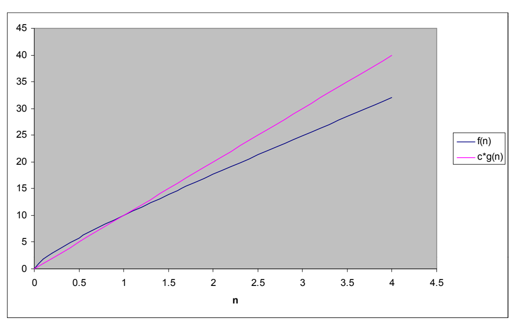
- Time complexity of a given algorithm
  - How does time depend on problem size?
  - Does time depend on problem instance or details?
  - Is this the fastest algorithm?
  - How much does speed matter for this problem?
- Space complexity
  - How much memory is required for a given problem size?
- Assumptions on computer word size, processor
  - Fixed word/register size
  - Single or multi (grid, hypercube) processor
- Solution quality
  - Exact or approximate/bounded
  - Guaranteed optimal or heuristic

Methods of complexity analysis

- Asymptotic analysis
  - Create recurrence relation and solve
    - This relates problem size of original problem to number and size of sub-problems solved
  - Different performance measures are of interest
    - Worst case (often easiest to analyze; need one ‘bad’ example)
    - Best case (often easy for same reason)
    - Data-specific case (usually difficult, but most useful)
- Write implementation of algorithm (on paper)
  - Create table (on paper) of frequency and cost of steps
  - Sum up the steps; relate them to problem size
- Implement algorithm in Java
  - Count steps executed with counter variables, or use timer
  - Vary problem size and analyze the performance
- These methods are all used
  - They vary in accuracy, generality, usefulness and ‘correctness’
  - Similar approaches for probabilistic algorithms, parallel, etc.

Asymptotic notation: upper bound O(..)

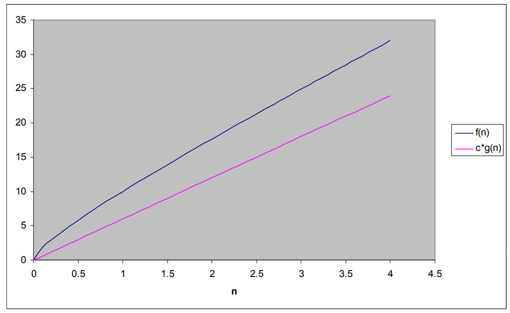
- $f(n)= O(g(n))$ if and only if
  - $f(n) \leq c * g(n)$
  - where $c > 0$
  - for all $n > n_0$
- Example:
  - $f(n)= 6n + 4\sqrt{n}$
  - $g(n)= n$
  - $c= 10$ (not unique)
  - $f(n)= c * g(n)$ when $n= 1$
  - $f(n) < g(n)$ when $n > 1$
  - Thus, $f(n)= O(n)$



- O(..) is worst case (upper bound) notation for an algorithm’s complexity (running time)

Asymptotic notation: lower bound Ω(..)

- f(n)= Ω(g(n)) if and only if
  - f(n) ≥ c * g(n)
  - where c > 0
  - for all n > n_0
- Example:
  - f(n)= 6n + 4√n
  - g(n)= n
  - c= 6 (again, not unique)
  - f(n)= c * g(n) when n= 0
  - f(n) > g(n) when n > 0
  - Thus, f(n)= Ω(n)



- Ω(..) is best case (lower bound) notation for an algorithm's complexity (running time)

Asymptotic notation

- Worst case or upper bound: O(..)
  - f(n)= O(g(n)) if f(n) ≤ c* g(n)
- Best case or lower bound: Ω(..)
  - f(n)= Ω(g(n)) if f(n) ≥ c* g(n)
- Composite bound: Θ(..)
  - f(n)= Θ(g(n)) if c_1* g(n) ≤ f(n) ≤ c_2* g(n)
- Average or typical case notation is less formal
  - We generally say "average case is O(n)", for example

Example performance of some common algorithms

Algorithm	Worst case	Typical case
Simple greedy	O(n)	O(n)
Sorting	$O(n^2)$	O(n lg n)
Shortest paths	$O(2^n)$	O(n)
Linear programming	$O(2^n)$	O(n)
Dynamic programming	$O(2^n)$	$O(2^n)$
Branch-and-bound	$O(2^n)$	$O(2^n)$

Linear programming simplex is $O(2^n)$, though these cases are pathological
Linear programming interior point is $O(Ln^{3.5})$, where L= bits in coefficients
Shortest path label correcting algorithm is $O(2^n)$, though these cases are pathological
Shortest path label setting algorithm is O(a lg n), where a= number of arcs. Slow in practice.

Running times on 1 GHz computer

n	O(n)	O(n lg n)	$O(n^2)$	$O(n^3)$	$O(n^{10})$	$O(2^n)$
10	.01 µs	.03 µs	.10 µs	1 µs	10 s	1 µs
50	.05 µs	.28 µs	2.5 µs	125 µs	3.1 y	13 d
100	.10 µs	.66 µs	10 µs	1 ms	3171 y	10^{13} y
1,000	1 µs	10 µs	1 ms	1 s	10^{13} y	10^{283} y
10,000	10 µs	130 µs	100 ms	16.7 min	10^{23} y	
100,000	100 µs	1.7 ms	10 s	11.6 d	10^{33} y	
1,000,000	1 ms	20 ms	16.7 min	31.7 y	10^{43} y	

Assumes one clock step per operation, which is optimistic

Complexity analysis: recursive sum

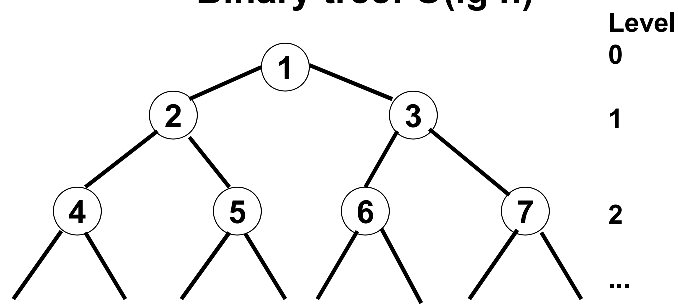
```
public class SumCountRec {
   static int count;

   public static double rSum(double[] a, int n) {
        count++;
        if (n <= 0) {
                count++;
                return 0.0;
        }
        else {
                count++;
                return rSum(a, n-1) + a[n-1];
        }
   }

   public static void main(String[] args) {
        count = 0;
        double[] a = { 1, 2, 3, 4, 5};
        System.out.println("Sum is " + rSum(a, a.length));
        System.out.println("Count is " + count);
   }
} // We can convert any iterative program to recursive
```

Complexity analysis: recurrence relations

- **For recursive sum:**
  - **T(n)= 2 if n= 0**
  - **T(n)= 2 + T(n-1) if n> 0**
- **To solve for T(n)**
  - **T(n)= 2 + T(n-1)**
    = 2 + 2 + T(n-2)
    = 2*2 + T(n-2)
    = n*2 + T(0)
    = 2n + 2

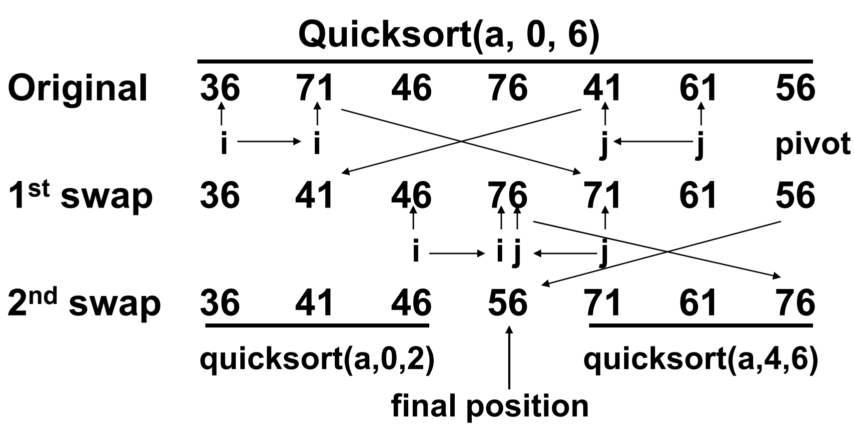
  Thus, T(n) = Θ(n)
- **Solving recurrence relations is a typical way to obtain asymptotic complexity results for algorithms**
  - **There is a master method that offers a cookbook approach to recurrence**

Binary tree: O(lg n)



- Max nodes on level i= 2^i
- Max nodes in tree of depth k= 2^{k+1}-1
    - This is <u>full</u> tree of depth k
- Each item in left subtree is smaller than parent
- Each item in right subtree is larger than parent
- It thus takes one step per level to search for an item
- In a tree of n nodes, how may steps does it take to find an item?
    - Answer: O (lg n)
    - Approximately 2^k nodes in k levels
- Remember that logarithmic is the "inverse" of exponential

Quicksort: O (n lg n)



Complexity analysis: count steps on paper

```
public class MatrixCount {
   static int count;

   public static double[][] add( double[][] a, double[][] b) {
        int m= a.length;
        int n= a[0].length;
        double[][] c = new double[m][n];
        for (int i = 0; i < m; i++) {
                count++;              //`for i‘:     Θ(m)
                for (int j = 0; j < n; j++) {
                        count++;      //`for j‘:     Θ(mn)
                        c[i][j] = a[i][j] + b[i][j];
                        count++;      // assgt  :    Θ(mn)
                }
                count++;              // loop init:  Θ(1)
        }
        count++;                      // loop init:  Θ(1)
        return c;
   }                                  // Total(max): Θ(mn)
   public static void main(String[] args) {
        count = 0;
        double[][] a = { {1, 2}, {3, 4} };
        double[][] b = { {1, 2}, {3, 4} };
        double[][] c = add(a, b);
        System.out.println("Count is: "+ count); }  }
```

Complexity: exponentiation, steps on paper

```
public class Expon {
   public static int count;
   public static long exponentiate(long x, long n) {
     count= 0;
     long answer = 1;
     while (n > 0) {
       while (n % 2 == 0) {
         n /= 2;       // Since n is halved,
         x *= x;       // loop called Θ(log n) times
         count++;
       }
       n--;            // Executed at most once per loop
       answer *= x;
       count++;
     }
     return answer;
   }

   public static void main(String[] args) {
     long myX = 5;
     for (long myN= 1; myN <= 25; myN++) {
       System.out.println(exponentiate(myX, myN)+ “ “+ count);
   }
}
```

Timing: sequential search

```
public class SimpleSearch {
   public static int seqSearch(int[] a, int x, int n) {
        int i= n;
        a[0] = x;
        while (a[i] != x)
               i--;
        return i;
   }

   public static void main(String[] args) {
        // Slot 0 is a placeholder; search value copied there
        int[] a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        System.out.println("SeqSearch location is " +
                seqSearch(a, 7, a.length-1));
        System.out.println("SeqSearch location is " +
                seqSearch(a, 11, a.length-1));
   }
}  // This algorithm is O(n): avg n/2 for steps successful
   // search, and n steps for unsuccessful search
```

Java timing

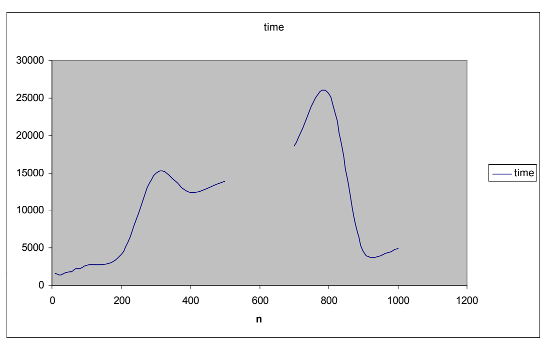
- **Java has method System.*nanoTime*(). This is the best we can do. From Javadoc:**
  - **This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time.**
  - **The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative).**
  - **This method provides nanosecond precision, but not necessarily nanosecond accuracy.**
  - **No guarantees are made about how frequently values change.**

A poor timing program

```
public class SearchTime1 {
   public static void timeSearch() {
        int a[] = new int[1001];
        int n[] = new int[21];
        for (int j = 1; j <= 1000; j++)
            a[j] = j;
        for (int j = 1; j <= 10; j++) {
            n[j] = 10 * (j - 1);
            n[j + 10] = 100 * j;
        }
        System.out.println("     n  time");
        for (int j = 1; j <= 20; j++) {
            long h = System.nanoTime();
            SimpleSearch.seqSearch(a, 0, n[j]);
            long h1 = System.nanoTime();
            long t = h1 - h;
            System.out.println("   " + n[j] + "   " + t);
        }
        System.out.println("Times are in nanoseconds");
   }

   public static void main(String[] args) {
        timeSearch();
}  }
```

SearchTime1 sample output

n	time
0	1572954
10	2013
20	2237
30	2520
40	3288
50	3871
60	3439
70	6520
80	5774
90	6260
100	4615
200	7587
300	9999
400	12696
500	15607
600	29191
700	18299
800	21851
900	5026
1000	5399

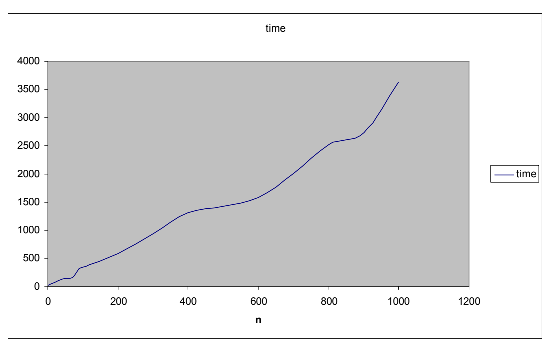


An adequate timing program

```
public class SearchTime2 {
   public static void timeSearch() { // Repetition factors
     int[] r = { 0, 20000000, 20000000, 15000000, 10000000,
        10000000, 10000000, 5000000, 5000000, 5000000, 5000000,
        5000000, 5000000, 5000000, 5000000, 5000000, 5000000,
        2500000, 2500000, 2500000, 2500000 };
     int a[] = new int[1001];
     int n[] = new int[21];
     for (int j = 1; j <= 1000; j++)
        a[j] = j;
     for (int j = 1; j <= 10; j++) {
        n[j] = 10 * (j - 1);
        n[j + 10] = 100 * j;  }
     System.out.println("     n       t1            t\n");
     for (int j = 1; j <= 20; j++) {
        long h = System.nanoTime();
        for (int i = 1; i <= r[j]; i++) {
          SimpleSearch.seqSearch(a, 0, n[j]);  }
        long h1 = System.nanoTime();
        long t1 = h1 - h;
        double t = t1;
        t /= r[j];
        System.out.println("  " + n[j] + "  " + t1 + "    " + t); }
     System.out.println("Times are in nanoseconds");
   }
   public static void main(String[] args) {
        timeSearch();  }  }
```

SearchTime2 sample output

n	time
0	18.06976
10	48.875175
20	69.04334
30	96.90906
40	131.7094
50	146.09915
60	141.81258
70	160.09126
80	232.35527
90	307.9214
100	340.1613
200	590.4388
300	941.6273
400	1305.8167
500	1416.4121
600	1574.6318
700	2004.8795
800	2525.205
900	2734.0051
1000	3634.6343



Summary

- Algorithm complexity varies greatly, from O(1) to $O(2^n)$
- Many algorithms can be chosen to solve a given problem
  - Some fit the problem formulation tightly, some less so
  - Some are faster, some are slower
  - Some are optimal, some approximate
- Complexity is known for most algorithms we're likely to use
  - Analyze variations (or new algorithms) you create
  - Many algorithms of interest are $O(2^n)$:
    - Use or formulate special cases for your problem
    - Limit problem size (decomposition, aggregation, approximation)
    - Implement good code
  - If necessary, reformulate your problem (you often can):
    - Reverse inputs and outputs
    - Change decision variables
    - Develop analytic results to limit computational space to be searched

1.204 Computer Algorithms in Systems Engineering
Spring 2010