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 $O(\cdot)$

- $f(n) = O(g(n))$ if and only if
  - $f(n) \leq c \cdot g(n)$
  - where $c > 0$
  - for all $n > n_0$
- Example:
  - $f(n) = 6n + 4\sqrt{n}$
  - $g(n) = n$
  - $c = 10$ (not unique)
  - $f(n) = c \cdot g(n)$ when $n = 1$
  - $f(n) < g(n)$ when $n > 1$
  - Thus, $f(n) = O(n)$
- $O(\cdot)$ is worst case (upper bound) notation for an algorithm’s complexity (running time)
Asymptotic notation: lower bound $\Omega(\cdot)$

- $f(n) = \Omega(g(n))$ if and only if
  - $f(n) \geq c \cdot g(n)$
  - where $c > 0$
  - for all $n > n_0$
- **Example:**
  - $f(n) = 6n + 4\sqrt{n}$
  - $g(n) = n$
  - $c = 6$ (again, not unique)
  - $f(n) = c \cdot g(n)$ when $n = 0$
  - $f(n) > g(n)$ when $n > 0$
  - Thus, $f(n) = \Omega(n)$

- $\Omega(\cdot)$ is best case (lower bound) notation for an algorithm’s complexity (running time)

Asymptotic notation

- **Worst case or upper bound:** $O(\cdot)$
  - $f(n) = O(g(n))$ if $f(n) \leq c \cdot g(n)$
- **Best case or lower bound:** $\Omega(\cdot)$
  - $f(n) = \Omega(g(n))$ if $f(n) \geq c \cdot g(n)$
- **Composite bound:** $\Theta(\cdot)$
  - $f(n) = \Theta(g(n))$ if $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- **Average or typical case notation is less formal**
  - We generally say “average case is $O(n)$”, for example
Example performance of some common algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Typical case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple greedy</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorting</td>
<td>O(n^2)</td>
<td>O(n lg n)</td>
</tr>
<tr>
<td>Shortest paths</td>
<td>O(2^n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linear programming</td>
<td>O(2^n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>O(2^n)</td>
<td>O(2^n)</td>
</tr>
<tr>
<td>Branch-and-bound</td>
<td>O(2^n)</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>

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

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

Assumes one clock step per operation, which is optimistic
Complexity analysis: recursive sum

```java
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, \( T(n) = \Theta(n) \)

- 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
• Max nodes on level $i = 2^i$
• Max nodes in tree of depth $k = 2^{k-1}$
  • This is full 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 many steps does it take to find an item?
  • Answer: $O(\log n)$
  • Approximately $2^k$ nodes in $k$ levels
• Remember that logarithmic is the “inverse” of exponential

Quicksort: $O(n \log n)$

Original

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>71</th>
<th>46</th>
<th>76</th>
<th>41</th>
<th>61</th>
<th>56</th>
</tr>
</thead>
</table>

1st swap

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>41</th>
<th>46</th>
<th>76</th>
<th>71</th>
<th>61</th>
<th>56</th>
</tr>
</thead>
</table>

2nd swap

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th>41</th>
<th>46</th>
<th>56</th>
<th>71</th>
<th>61</th>
<th>76</th>
</tr>
</thead>
</table>

pivot

quicksort(a,0,2) quicksort(a,4,6)

final position
Complexity analysis: count steps on paper

```java
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 (int j = 0; j < n; j++) {
                count++;
                c[i][j] = a[i][j] + b[i][j];
            }
        }
        count++;
        return c;
    }

    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

```java
public class Expon {
    public static int count;
    public static long exponentiate(long x, long n) {
        long answer = 1;
        while (n > 0) {
            while (n % 2 == 0) {
                n /= 2;
                x *= x;
                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

```java
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) {
        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

```java
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

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1572954</td>
</tr>
<tr>
<td>10</td>
<td>2013</td>
</tr>
<tr>
<td>20</td>
<td>2237</td>
</tr>
<tr>
<td>30</td>
<td>2520</td>
</tr>
<tr>
<td>40</td>
<td>3288</td>
</tr>
<tr>
<td>50</td>
<td>3871</td>
</tr>
<tr>
<td>60</td>
<td>4339</td>
</tr>
<tr>
<td>70</td>
<td>6520</td>
</tr>
<tr>
<td>80</td>
<td>5774</td>
</tr>
<tr>
<td>90</td>
<td>6260</td>
</tr>
<tr>
<td>100</td>
<td>4615</td>
</tr>
<tr>
<td>200</td>
<td>7587</td>
</tr>
<tr>
<td>300</td>
<td>9999</td>
</tr>
<tr>
<td>400</td>
<td>12696</td>
</tr>
<tr>
<td>500</td>
<td>15607</td>
</tr>
<tr>
<td>600</td>
<td>29191</td>
</tr>
<tr>
<td>700</td>
<td>18299</td>
</tr>
<tr>
<td>800</td>
<td>21851</td>
</tr>
<tr>
<td>900</td>
<td>5026</td>
</tr>
<tr>
<td>1000</td>
<td>5399</td>
</tr>
</tbody>
</table>
```
public class SearchTime2 {  
    public static void timeSearch() { // Repetition factors
        int[] r = {0, 20000000, 20000000, 15000000, 30000000, 10000000, 10000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 50000000, 25000000, 25000000, 25000000, 25000000};
        int[] a = new int[1001];
        int[] n = new int[21];
        for (int j = 1; j <= 100; j++)
            for (int j = 1; j <= 10; j++)
                a[j] = j;
        for (int j = 1; j <= 20; j++)
            for (int j = 1; j <= 10; j++)
                n[j] = 10 * (j - 1);
        System.out.println(" n   t1   t\n");
        System.out.println(" + n[j] + " + t1 + "+ t);
        System.out.println("Times are in nanoseconds");
    }
    public static void main(String[] args) {
        timeSearch();
    }
}

SearchTime2 sample output

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