Algorithm Analysis

Readings: Chapter 1.6-1.7.

How can we determine if we have an efficient algorithm?
Criteria:
  Does it meet specification/work correctly?
  Is it understandable/maintainable/simple?
  How much storage (memory & disk) does it use?
  How much time does it take to execute?

Example:
  \[ \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2), \text{Fib}(1) = 0, \text{Fib}(2) = 1 \]

\[
\text{FibSolve}(N) \{
  \text{If (N==1) return 0;}
  \text{if (N==2) return 1;}
  \text{return FibSolve(N-1)+FibSolve(N-2);}
\}

Performance Measurement

The amount of time a program takes to execute depends on:
  Algorithm
  Computer
  Compiler
  Input size
  Input organization

Approach
  For representative inputs {
    \text{Start timer}
    \text{Run algorithm (multiple times?)}
    \text{Stop timer}
  }

What about different computers, compilers, etc.?
Complexity Analysis

Time complexity of an algorithm:
amount of work done by an algorithm as a function of the size of the input

Count all operations ("operation" = semantically meaningful program segment
whose runtime independent of input or output size/complexity)

Operations:
  Integer addition
  for (i=1; i < 100; i++) { A[i]=0; }

Not Operations:
  for (i=1; i < INPUT_VAL; i++) { A[i]=0; }
  count(list);

Result: Compiler & Computer-independent metrics

Example - Sequential Search

for (i=1; i <= N; i++) { if (A[i]==search_element) return TRUE; }
return FALSE;

Assume Q=Probability element isn’t in the list

Worst-case:

Average-case:

Note: we typically focus primarily on worst-case.
Question: How meaningful is the exact number of steps when individual steps can take different amounts of time?

Since duration of operations is variable, we ignore constant factors, and just consider orders of magnitude.

For large enough problems, constant factors irrelevant for comparisons

Example:
\[
\begin{align*}
f(n) &= 3n^2 & g(n) &= 25n & n > 8 & g(n) < f(n) \\
f(n) &= 3n^2 & g(n) &= 300n & n > 100 & g(n) < f(n)
\end{align*}
\]

"Big-Oh":
\[
\begin{align*}
C_1 f(n) &= O(f(n)) \\
f(n) + C_1 &= O(f(n)) \\
\text{ith degree polynomials } C_1 n^i + C_{i-1} n^{i-1} + \cdots + C_1 n + C_0 &= O(n^i) \\
\lg(n) &\text{ is greater than a constant, but less than } n \\
N + \lg(n) &= O(N) \\
N + N\lg(n) &= O(N\lg(n)) \\
N^2 + N\lg(n) &= O(N^2)
\end{align*}
\]

Note:
"Big-Oh" is normally worst-case.
Example Asymptotic Complexity: Bubble Sort

BubbleSort(array A, length N) {
    int j, k, temp;
    for (j = 1; j <= N; j++) {
        for (k = 1; k < N; k++) {
            if (A[k] > A[k+1]) {
                temp = A[k];
                A[k] = A[k+1];
                A[k+1] = temp;
            }
        }
    }
}

Example Asymptotic Complexity: Binary Search


int Search(A) {
    if (A[0] == search_element) return 0;
    if (A[N] == search_element) return N;
    low = 0; high = N;
    while (low < high - 1) {
        middle = (low+high)/2;
        if (A[middle] < search_element) low = middle;
        else if (A[middle] > search_element) high = middle;
        else return middle; /* Found the location */
    }
    return ERROR; /* Element should be in the list */
}

In each step, half of list is eliminated from consideration. N steps can handle a list of ~2^N length.
Example Asymptotic Complexity: Searching lists

Algorithm 1:
for (i=1; i <= N; i++) { if (A[i]==search_element) return TRUE; }
return FALSE;

Algorithm 2:
BubbleSort(array, N);
BinarySearch(array);

Algorithm 3:
UltraSort(array, N)  //assume UltraSort is O(lgN)
BinarySearch(array);

Practical Complexity

Assume 1GHz machine (1 billion instructions per second)

<table>
<thead>
<tr>
<th>n</th>
<th>log₂n</th>
<th>n</th>
<th>nlog₂n</th>
<th>n²</th>
<th>n³</th>
<th>n⁴</th>
<th>n¹⁰</th>
<th>2ⁿ</th>
<th>nⁿ</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003 us</td>
<td>0.01 us</td>
<td>0.033 us</td>
<td>0.1 us</td>
<td>1. us</td>
<td>10. us</td>
<td>10. sec</td>
<td>1.02 us</td>
<td>3.63 ms</td>
</tr>
<tr>
<td>20</td>
<td>0.004 us</td>
<td>0.02 us</td>
<td>0.086 us</td>
<td>0.4 us</td>
<td>8. us</td>
<td>80. us</td>
<td>800. ms</td>
<td>8.41E+01 yr</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0.005 us</td>
<td>0.03 us</td>
<td>0.15 us</td>
<td>0.9 us</td>
<td>27. us</td>
<td>810. us</td>
<td>6.83 day</td>
<td>1.07 sec</td>
<td>8.41E+01 yr</td>
</tr>
<tr>
<td>40</td>
<td>0.005 us</td>
<td>0.04 us</td>
<td>0.21 us</td>
<td>1.6 us</td>
<td>64. us</td>
<td>2.56 ms</td>
<td>121 day</td>
<td>18.3 min</td>
<td>2.99E+01 yr</td>
</tr>
<tr>
<td>50</td>
<td>0.006 us</td>
<td>0.05 us</td>
<td>0.28 us</td>
<td>2.5 us</td>
<td>125. us</td>
<td>6.25 ms</td>
<td>3.10 yr</td>
<td>13.0 day</td>
<td>9.64E+01 yr</td>
</tr>
<tr>
<td>100</td>
<td>0.007 us</td>
<td>0.1 us</td>
<td>0.664 us</td>
<td>10. us</td>
<td>100. ms</td>
<td>3.17E+03 yr</td>
<td>3.40E+13 yr</td>
<td>2.99E+14 yr</td>
<td></td>
</tr>
<tr>
<td>1k</td>
<td>0.01 us</td>
<td>1. us</td>
<td>9.97 us</td>
<td>1. sec</td>
<td>16.7 min</td>
<td>3.17E+13 yr</td>
<td>3.40E+284 yr</td>
<td>3.17E+33 yr</td>
<td></td>
</tr>
<tr>
<td>10k</td>
<td>0.013 us</td>
<td>10. us</td>
<td>133 us</td>
<td>100. ms</td>
<td>16.7 min</td>
<td>3.17E+23 yr</td>
<td>3.17E+33 yr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100k</td>
<td>0.017 us</td>
<td>100. us</td>
<td>1.66 ms</td>
<td>10. sec</td>
<td>11.6 day</td>
<td>3.17E+31 yr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

us = microsecond (10⁻⁶), ms = millisecond (10⁻³)

Given twice as much time, how much can you do:

O(log n): n², O(n): 2n, O(n²): n²¹², O(n³): n³¹³, O(2ⁿ): n+1
Problem Complexity

Do all problems have efficient solutions?

When should I apply heuristics (rules of thumb) instead of looking for an exact algorithm?

What the heck are P, NP, and NP-complete?

P: algorithms that can be solved efficiently

NP: algorithms that probably can be solved heuristically

NP-complete/NP-hard: Problems that almost definitely cannot be solved both exactly & efficiently.
**P = Polynomial Time Algorithms**

“Never met a problem I couldn’t solve”

P is the class of problems for which there is a polynomial time solution

\[ O(n^C) \] for some constant C

Sort is \( O(n \log_2 n) \)

Search is \( O(\log_2 n) \) for sorted lists, \( O(n) \) for unsorted lists

Fibonacci is \( O(n) \)

Note: \( O(n^{57}) \) may not be efficient, but most problems in P are relatively efficiently solved

---

**NP = Nondeterministic Polynomial Algorithms**

“I’ll know it when I see it”

Assume you are given a devious Oracle or Genie

- Knows the answer to all problems
- Will lie to you unless you thoroughly check its work
- Some similarity to Quantum Computing

Any problem who’s answer can be checked in polynomial time is in NP

Why NP?

- Many interesting problems are easy to check, but hard to compute
- Some heuristic techniques (Simulated Annealing, Simulated Evolution, etc.) can help guess the solution given a check on correctness.
NP Example: Travelling Salesman Problem

A salesman must visit a set of \( N \) cities in any order. Given current airline rates, find a set of flights that visits all of these cities for less than \( X \) dollars.

Polynomial Algorithm?
NP Algorithm?
Heuristic?

Seattle
San Francisco
Denver
Chicago
Los Angeles
Dallas
New Orleans
Miami
New York
D.C.
Boston

P vs. NP

Are all algorithms in P also in NP?
NP algorithm for a P problem: Don’t use oracle, just create a solution.

Are any algorithms in NP not in P?
If you can prove it, there’s a Full Professorship at M.I.T. waiting for you.

What do we know?
It is likely that \( P \neq NP \)
There are “NP-Complete” problems which are the least likely to be in P

NP-hard: a problem which, if it were in P would mean all problems in NP are in P

NP-complete: an NP-hard problem which is in NP
NP-Complete

A problem is NP-Complete if:
   - It is in NP.
   - It is NP-hard - The creation of a polynomial-time algorithm for it provides a polynomial-time algorithm for all problems in NP.

The first NP-Complete algorithm: Boolean Satisfiability (SAT)
   - Is there an assignment of values to variables that makes a given Boolean Equation true?

   In NP: Take the solution from the Oracle and check that the Boolean Equation becomes true (Can be done in O(n)).

   Is NP-hard: We can show that any problem in NP can be converted into a Boolean equation in polynomial time. If there was a polynomial time solution to SAT, we would thus have a polynomial time solution to all NP problems. Details in CSE 431 or CSE 531.

NP-Hard

Given that we have at least one NP-Complete problem, showing others are NP-hard becomes easier:
   - For a problem under consideration X
     1.) Take any NP-complete problem Y (pick wisely!)
     2.) Show that there is a polynomial time method for changing any instance of problem Y into an instance of problem X

   Now, a polynomial solution to X means there is a polynomial solution to Y. A polynomial solution to Y implies a polynomial solution to all of NP.
Example NP-Completeness Proof

Vertex Cover:
Given an undirected graph \( G(V, E) \), where \( V \) are the vertices and \( E \) are the edges, and a size \( S \). Find a set of vertices \( C, C \subseteq V \), such that for every edge at least one of its endpoints is in \( C \), and \( C \) has at most \( S \) elements.

This graph has a cover of 3 vertices \((A, C, E)\) and others, but not of 2.

Example NP-Completeness Proof (cont.)

Vertex cover is in NP:

Vertex cover is NP-hard: Pick the NP-complete problem 3-SAT to reduce to vertex cover

3-SAT: Given a Boolean equation in Product of Sums form, where each Sum term has 3 variables, find an assignment of values to variables that makes the equation true.

\[
F = (A + B + C) \cdot (\overline{A} + C + C) \cdot (A + \overline{B} + B) \cdot (\overline{A} + \overline{B} + \overline{C})
\]

Note: 3-SAT was shown to be NP-Complete because any SAT equation can be converted efficiently to 3-SAT form.
Example NP-Completeness Proof (cont.)

Mapping 3-SAT to Vertex cover:
Each literal becomes a vertex. If the vertex ISN’T in the cover, it is true.
Must ensure:
1.) At least one literal per term is true/uncircled

2.) Both X and not(X) can’t both be true

\[ F = (A + B + C) \cdot (\bar{A} + C + C) \cdot (A + \bar{B} + B) \cdot (\bar{A} + \bar{B} + \bar{C}) \]

Why this Works (Intuitive)

We can map Y to X in poly time:
Y ≤ X (Y is no harder than X, or X is no easier than Y)

Y is NP-hard:
Z ≤ Y  Forall Z in NP

Thus, X is NP-hard:
Z ≤ Y ≤ X  Forall Z in NP

Z ≤ X  Forall Z in NP
Why This Works (Mathematical)

X is NP-Hard because we show that there is a polynomial time method for changing any instance of problem Y into an instance of problem X, and Y is NP-Hard.

NP-Hard means that if the problem is in P, then all NP problems are in P.

Why is X NP-Hard? What happens if X is in P?

Algorithm for Y:
- Convert problem to an instance of problem X Poly Time
- Solve problem X Poly Time

Thus, if X is in P, Y is in P.

Y is NP-hard, which means if Y is in P, then all NP problems are in P

Thus, if X is in P, all NP problems are in P. Therefore, X is NP-hard.

How Not to Show NP-Hardness

Common mistake: show that the problem can be transformed to an NP-complete problem. Proves only that the problem is in NP, not that it is NP-complete.

X is NP-hard + Y can be mapped to X:
- Z ≤ X For all Z in NP, and Y ≤ X, does not mean Z ≤ Y

Example - transform 1-SAT to 3-SAT:

Problem: Just because 3-SAT is capable of solving 1-SAT, doesn’t mean 1-SAT is as hard as 3-SAT.

In fact 1-SAT is in P, 3-SAT is NP-Complete
Some NP-Complete Problems

Travelling Salesman Problem: Find the cheapest path that visits all vertices in a graph.

2-level Logic Minimization: Find the simplest Sum-of-Products implementation for a Boolean Equation.

Integer Linear Programming: Given an mxn matrix of integers A and a column vector B of n integers, does there exist a column vector of integers X such that \( A \cdot X \geq B \) ?

Graph Coloring: Given a graph and an integer K, can the graph be colored with K colors so that no two adjacent vertices have the same color?

Clique: Does the specified graph have a clique of K vertices, where each of these vertices is adjacent to all other members of the clique?

Numerous other NP-hard problems exist, including most of Physical Design!

Who Cares About NP-Hardness?

NP-Hardness means that an efficient, exact algorithm is unlikely

Heuristic algorithms are justified

Some problems believed hard aren't, and you're "scooped" with exact algorithm

Routing two-terminal nets in a crossbar system
Heuristic Algorithms

When a problem proves to be too complex to solve exactly, we instead come up with a heuristic algorithm.

Heuristics = rules of thumb, ways of getting reasonable solutions, but which may:
1.) Not get the exact answer
2.) Not get an answer for some problems

Examples: Think of heuristics for the following problems:
- Find the minimum coloring for a graph.
- The travelling salesman problem.

CAD example
- Espresso for 2-level logic minimization