CSE 373: Algorithm Classification

Miscellaneous (Chapters 9 & 10)

Algorithm Types

Implementations:
- recursive vs. iterative

Algorithm Methodology:
- divide-and-conquer (e.g., Binary Search, Quicksort)
- greedy (e.g., Dijkstra’s & Kruskal’s Algorithms)
- dynamic programming (e.g., efficient fibonacci)
Algorithm Requirements

Space and Time:
- asymptotic analysis for primary effects
- evaluation of secondary effects
  - by inspection
  - by experimentation

Q: How fast/space-efficient is “good enough?”
  (e.g., \( \mathcal{O}(n) \) was bad for \texttt{Delete()} \textcolor{red}{\text{, but } \mathcal{O}(n\log n) \text{ was great}} for \texttt{Sort()} \textcolor{red}{\text{\ldots}})

A:

Criteria for Good Running Time

Your resources
- how much time/memory can you afford?

Nature of the problem
- some problems are just harder than others
  (sorting is harder than deletion)

Characteristics of your application
- what problem sizes/input sets will you typically be running on? (be sure to plan for the future)

Maintainability/Elegance
- this tends to dominate software development costs
Evaluating Running Time/Space

- **O(1)** – ideal
- **O(log n)** – generally as good as ideal
- **O(n)** – could be better, could be worse
- **O(n log n)** – could be better, could be worse
- **O(n^2)** – could be better, could be worse
- **O(2^n)** – unusable

Games Theoreticians Play

Prove that an algorithm is Ω(f(n)) by nature

- e.g., sorting using only comparison (<, >, =) cannot be done in less than n log n time (Chapter 7)

What’s wrong with this claim:

“I wrote a **FindMin()** operation that runs in O(log n) time on an unsorted list of integers”
More Games Theoreticians Play

Classify problems based on their running times

- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $P$ – the set of problems that can be solved in polynomial time

What isn’t in $P$?

- All problems that require more than polynomial time (e.g., $O(2^n)$)
- All problems that cannot be solved with a computer (intractable problems)

*And maybe…*

- A set of problems (NP) that nobody knows whether or not they’re in $P$
Sample Intractable Problem

The Halting Problem: Can you write a function that takes a C program as input and determines whether or not the program will halt?

Answer: No, it’s intractable (impossible to solve on a computer)

Proof

Assume that we have such a procedure: halt()

Write the following program (testhalt.c):

```c
void main() {
    int willhalt;
    willhalt = halt("testhalt.c");
    if (willhalt) {
        while (1) {
        }
    } else {
        exit(0);
    }
}
```
**Intractability Summary**

- Lots of intractable problems are of the form “we can’t detect property \( x \) about a program”
- Proofs are similar to that of halting problem
- However, *heuristics* can be used to detect and warn about some simple cases:

  ```
  void main() {
  main();
  }
  ```

**NP Problems**

**NP:**
- A class of problems for which exponential algorithms have been developed
- Nobody has found a polynomial-time algorithm for any of them
- *Yet,* nobody has proven that any of them could not have polynomial time algorithms
- This leads to the BIG question:

  \[ P = NP? \]
Sample NP Problem

**Graph 3-Colorability Problem:** Given a graph $G = (V,E)$, is it possible to color its vertices using 3 colors so that no two adjacent nodes are the same color?

![Graph](image)

Another Sample NP Problem

**Traveling Salesperson Problem:** Given a graph $G = (V,E)$, find a path that starts and ends at the same vertex, visits each vertex exactly once, and has minimal cost?

![Graph](image)
Solving NP Problems

- Exponential solution is too slow for any interesting problem size
- The best we can do is come up with heuristics that give us an approximate solution in polynomial time

Good Running Times (Again)

- Whew, our problem is not intractable!
- Whew, our problem is not in \textit{NP}!
- Now let’s get the asymptotic requirements as low as possible*
- Then let’s get the secondary effects as good as possible*

* Always keeping code maintenance in mind