Reduction, P and NP

Reduction (1)

- HeapSort:
  - Recall that building a Priority Queue using a binary heap and then repeatedly calling *deleteMax/Min* gives us HeapSort.
- This is just one example of reduction:
  - We say that heapSort *reduces to* building a priority queue.
  - Or, in more general terms, we could say that *sort* reduces to building a priority queue.
- Reduction—formal definition:
  - We say that problem A *reduces to* another problem B if we can use an algorithm that solves B to develop an algorithm that solves A.

Reduction (2)

- Cost model for reduction
  - Cost of recasting problem A as problem B
  - Cost of solving problem B
    - Note this cost alone might be significantly greater than the most efficient solution of A
  - Cost of transforming solution of B to domain of A
- Examples of problems that reduce to sorting:
  - Finding the maximum
    - The minimum cost of finding the maximum is actually linear but the cost of sorting and taking the head is linearithmic so the reduction really isn't efficient
  - Distinct
  - Scheduling to minimize average completion time

Intractability

- How hard is a problem?
  - Polynomial time: $c N^k$. Practical as long as $k < 2$ or N small, (“poly-time”).
  - Exponential time: k^{cN}. Intractrable!
- Turing machine:
  - a finite-state machine that reads inputs, moves from state to state, and writes outputs.
  - *Universality*: all physically realizable computing devices can be simulated by a Turing machine (Church-Turing thesis). Cannot be proven but *can* be falsified (but, so far, hasn’t).
  - *Computability*: There exist problems that cannot be solved by a Turing machine (“the halting problem”).
  - *Corollary*: the order of growth of the running time of a program to solve a problem on any computing device is within a polynomial factor of some program to solve the problem on a Turing machine (or any other computing device).

P and NP

- A “search” problem:
  - is a problem having solutions with the property that the time needed to *certify* that any solution is a polynomial function of the size of the problem (“poly-time”).
- **P** is the set of all search problems that can be solved in polynomial time.
- **NP** is the set of *all* search problems:
  - It includes all of the P problems (see above) and all those that can’t.
  - The “N” in NP refers to non-deterministic. If we can guess (randomly or semi-randomly—as in GA) a solution, we can then certify it to be correct in poly-time.

P & NP (2)

- Examples of problems in **P**:
  - sorting;
  - shortest path.
- Examples of problems in **NP**:
  - Hamiltonian path;
  - Factoring;
  - Any problem in **P**.

P & NP (3)

- **P** = **NP**?
  - Nobody knows for sure but all assume that **P** ∈ **NP**.
- Poly-time reduction:
  - If problem A reduces to problem B in no worse than polynomial time, then A *poly-time reduces* to B.
- NP-complete:
  - A search problem A is said to be *NP-complete* if all problems in **NP** poly-time reduce to A.

Wrap-up

- What we covered:
  - We've covered a lot of ground in this class. We've looked at some things in excruciating detail, other things less so. Some things we've skipped over altogether because we ran out of time.
  - We've covered the essential *patterns* of algorithms & data structures, e.g. divide-and-conquer, parallelization, reduction.
  - We spent a lot of time doing quizzes which I hope will be useful to you in the future. Don't forget the advice I posted about interviews.
- What you should take away and remember:
  - the basics of complexity and how to analyze an algorithm;
  - how to choose the best sort algorithm;
  - the different types of symbol table;
  - the basics of graphs;
  - the definition of entropy—which gives us the theoretical minimum number of decisions we have to make in order to solve a problem.
  - treasure the text book if you have a hard copy: you will refer to again many times.

Thank you for having me as your professor :)