Complexity theory

- Linear Programming is viewed as easy and Integer Programming is viewed as hard
- Next, we address some theoretical ways of characterizing easy vs. hard problems
- Often referred to as the theory of NP-completeness or NP-hardness

Overview of complexity

- How can we show that a problem is efficiently solvable?
  - We provide an algorithm and show that it solves the problem efficiently.
- How can we show that a problem is not efficiently solvable?
  - How do you prove a negative?
  - This is the aim of complexity theory, which is the topic of this lecture.
  - The approach here is non-standard in that it covers half of the usual definitions of an introduction to complexity
- Complexity of algorithms & complexity of problems

1. Instances versus problem

- This is an "*instance*" of linear programming.

$$\begin{array}{ll} \text{maximize} & 3x + 4y \\ \text{subject to} & 4x + 5y \leq 23 \\ & x \geq 0\,,\ \ y \geq 0 \end{array}$$

- When we say the linear programming *problem*, we refer to the collection of all instances.
- Similar, the traveling salesman problem refers to all instances of the traveling salesman problem, etc.

- Complexity theory addresses the following problem: When is a problem hard?
- Note: It does not deal with the question of whether any instance is hard.

Size of problems (instances)

- For any instance I of a problem, let *S(I)* be the number of inputs.
  - For an integer programming instance, $S \approx m \times n$
  - For TSP, $S \approx n$.
- Let *M(I)* be the largest integer in the data.
  - We assume that integers are expressed in binary.
  - Consider the problem of determining whether a number M is prime. It's size grows as $\log(M)$.
- *Size(I)* is the number of digits to represent I.

$$S(I) + \log M(I) \leq Size(I) \leq S(I) \times \log M(I)$$

- Interesting fact: everyone in complexity talks about the size of the problem, but almost no one cares about measuring it precisely.

2. Complexity of algorithms

- As problem instances get larger, the time to solve the problem grows.
- But how fast?
- Bounded by polynomial time: the time to solve a problem of size n is at most $p(n)$.

Example: Sorting a list of *n* items by using a greedy approach

4	3	9	12	14	11	7	10	8	24	5
3	4	9	12	14	11	7	10	8	24	5
3	4	5	9	12	14	11	7	10	8	24
3	4	5	7	9	12	14	11	10	8	24
3	4	5	7	8	9	12	14	11	10	24
3	4	5	7	8	9	10	12	14	11	24
3	4	5	7	8	9	10	11	12	14	24

Greedy sorting: for $i = 1$ to n, choose the ith least item on the list and put it in position i.

A quick analysis of greedy sorting

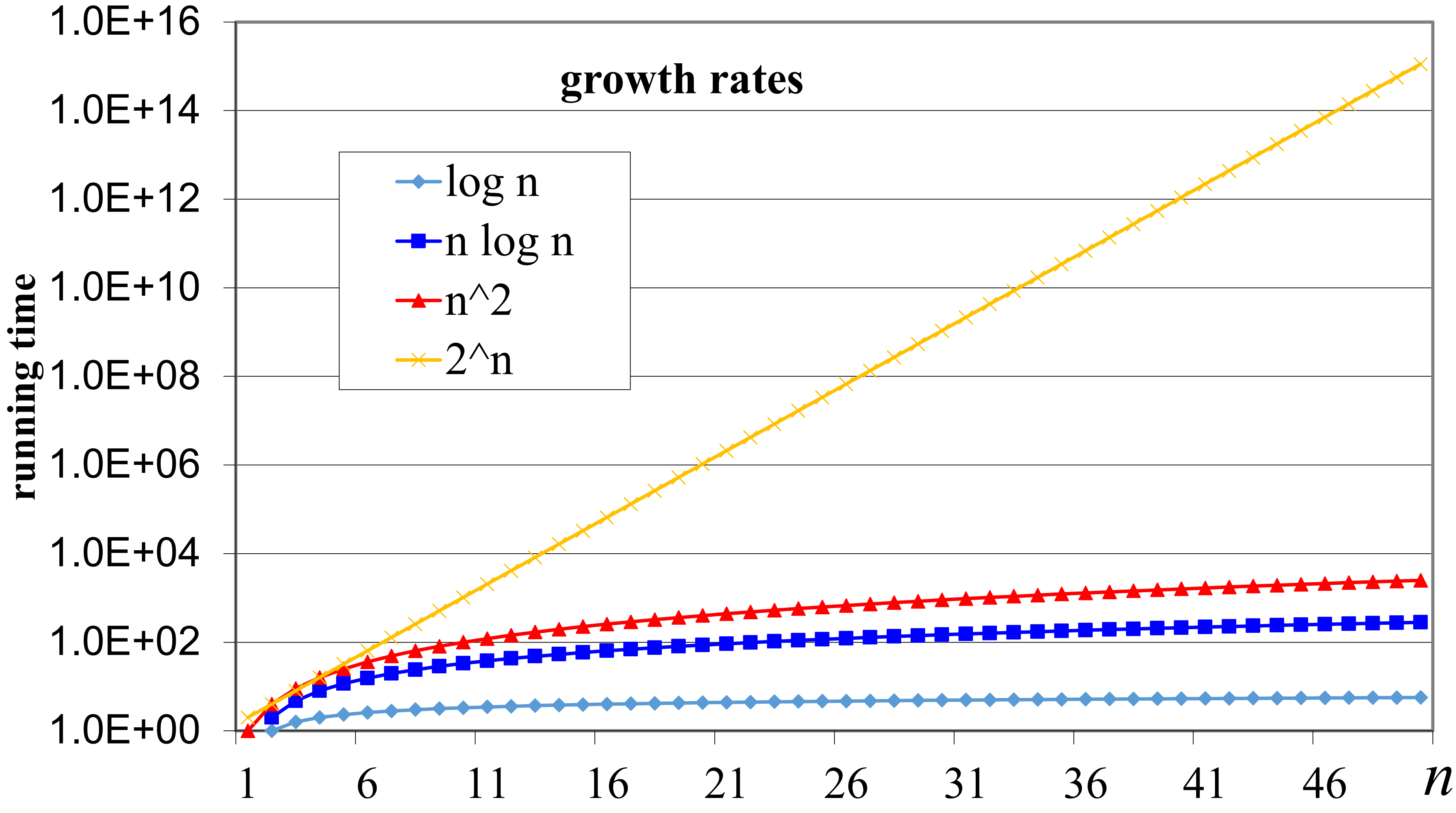
3	4	5	7	9	12	14	11	10	8	24

- Suppose that there are *n* items. How many items have to be scanned to find the next largest item?
- How much time does it take to insert the next largest item in the correct place?
- Claim: The number of comparisons and insertions is at most (n^2-n). This is a polynomial time algorithm.
- The best possible time for sorting is around *n* log(*n*).

Examples

- Finding a word in a dictionary with n entries.
  time $\approx$ log(n), depending on assumptions.
  Polynomial time
- Sorting n items
  time $\approx$ n log(n)
  Polynomial time
- Finding the shortest path from s to t
  time $\approx$ n^2.
  Polynomial time
- Complete enumeration of a binary integer program on n variables: time > 2^n.
  Exponential time (not polynomial time)
- Others: e.g., n! which grows faster than the exponential time

Running times as *n* grows

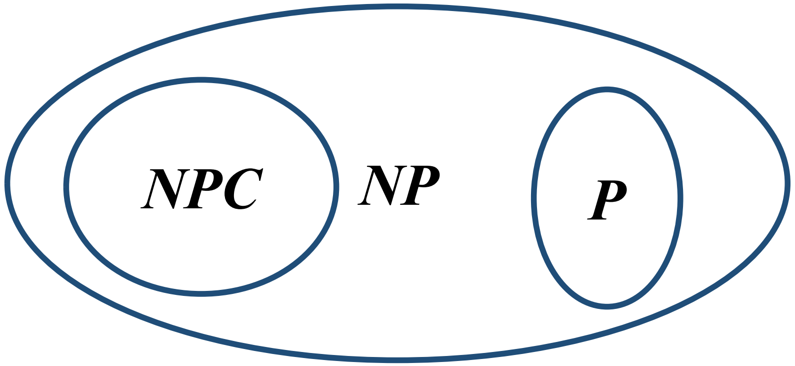


Polynomial time algorithms

- Big $\mathcal{O}$ (asymptotically upper bound): Rather than attempting to get a precise expression for the function of running time (worst case), it will suffice to approximate it from above. Function $f(n) = O(g(n))$ as $n \to \infty$ whenever here exists a positive constant c and a positive integer n' such that $|f(n)| \leq c|g(n)|$ for all integer $n \geq n'$. This means that only the asymptotic behavior of the function as n *approaches infinite (asymptotically)* is being considered.
- The algorithm A for problem X runs in *polynomial time* if the number of steps taken by A on any instance I is bounded by a polynomial in size(I), equivalently, by p(size(I)) for some polynomial p. That is bounded by $\mathcal{O}(p(size(I)))$.

3. Complexity of problems

- Feasibility problems vs. optimization problems
- *NP*: class of feasibility problems for which any given solution (certificate) can be verified in polynomial time. But there may not be known efficient way to locate a solution (i.e., solve the problem)



P: class of polynomial solvable problems

- P: class of (feasibility) problems solvable in polynomial time
  - A problem is solvable in polynomial time (or polynomially solvable) if there is a polynomial function such that the time to solve a problem of size n is bounded by $O(p(n))$.
  - We consider a problem X to be "easy" or efficiently solvable, if there is a polynomial time algorithm (even better) for solving X.
  - If a problem is easier than a polynomial-solvable problem, it is also polynomial-solvable.
  - Some examples in P: linear programming, assignment problem, transportation problem, minimum cost network flow problem
  - If problem B is polynomial time solvable, and problem A is polynomially reducible to B, then A is polynomial time solvable.

NPC: NP-completeness

- The main consequence of "A reduces to B" is "if we can solve B then we can solve A.", in a sense that "A is at most as hard as B is".

- *NPC*: the class of hardest feasibility problems called NP-complete problems.
  - A problem X in NP is NP-complete if every problem in NP can be reduced to X in polynomial time.
  - If A is NP-complete and A is polynomially reducible to B in NP, then B is NP-complete.
  - $NPC \subset NP$; that is, if there exists X in $NPC \cap P$, then every problem in NP is in P, that is, $NP=P$.

Polynomial time reduction

- Polynomial time reduction is a special type of reduction in which the reduction step could be carried out in polynomial time (within some model of computation.)
- A consequence of the reduction step being carried out polynomial time (i.e., “A reduces to B in polynomial time”) is that if we can solve B in polynomial time then one can also solve A in polynomial time. Hence, A is at most as hard as B is, when polynomial time computation is concerned.
- That is, B is at least as hard as A. If we already know that A is at least as hard as any problem in NP, then B is at least as hard as any problem in NP, i.e., it's NP-complete.

NP-hardness

- We call a problem (a feasibility problem or an optimization problem) NP-hard if there is an NP-complete problem that can be polynomially reduced to it.
- While the feasibility problem is NP-complete, the optimization problem is NP-hard. Its resolution is at least as difficult as the feasibility problem.
- If problem X is NP-hard, and if X is a special case of Y, then Y is NP-hard.
  - Example: 0-1 integer programming is NP-hard. 0-1 integer programming is a special case of integer programming. Therefore, integer programming is NP-hard. In fact, every algorithm that has ever been developed for integer programming takes exponential time. It is generally believed that there is no polynomial time algorithm for integer programming.
- If a problem is harder than an NP-hard problem, it is NP-hard.
- Example: travelling salesman problem (no worse than exponential time)

References

- *Integer and Combinatorial Optimization*, G.L. Nemhauser& L. A. Wolsey, Wiley, Chapter 15 Computational Complexity.
- *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Mc-Graw Hill, which is good source for learning the analysis of running time of an algorithm, Chapter 36 NP-completeness. It is good source for learning the analysis of running time of an algorithm.
- *Combinatorial Optimization*, W.J. Cook, W.H. Cunningham, W. R. Pulleyblank, and A. Schrijver, Wiley, Chapter 9 NP and NP-completeness.
- *Scheduling: Theory, Algorithms, and Systems,* Michael L. Pinedo, Springer, Appendix D Complexity Theory.