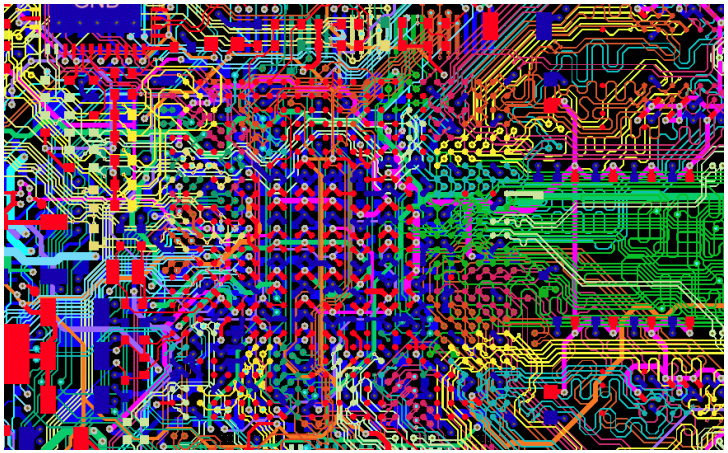
Computational Complexity



Theory of Computation

CISC 603, Spring 2020, Daqing Yun

Recall

- A decision problem is decidable if there is an algorithm that can answer it in principle
- A Turing machine deciding a language $L \subseteq \Sigma^*$ solves a decision problem: given $x \in \Sigma^*$, is $x \in L$?
  - A measure of the *size* of the problem is the length of the input string x

Computational Complexity

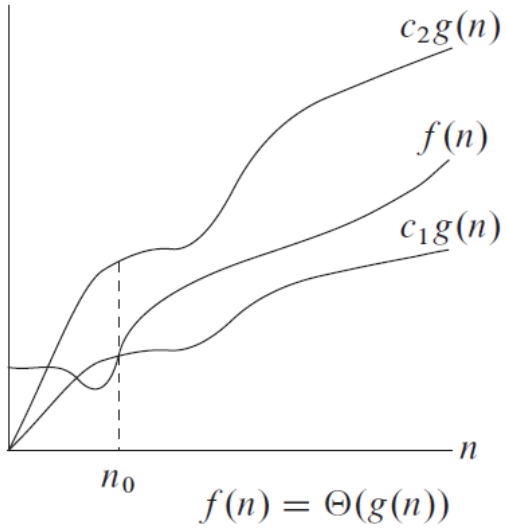
- We try to identify the problems for which there are *practical* algorithms
  - Ones that can answer *reasonable*-size instances in a reasonable amount of time

Growth of Functions

- If f and g are partial functions from $\mathbb{N}$ to $\mathbb{R}^+$; that is, both functions have values that are nonnegative real numbers wherever they are defined
  - We say that $f = O(g)$, or $f(n) = O(g(n))$ (which we read "f is big-oh of g" or "$f(n)$ is big-oh of $g(n)$") if, for some positive numbers C and N, $f(n) \leq C\, g(n)$ for every $n \geq N$
  - For example, every polynomial of degree k with positive leading coefficient is $O(n^k)$

Growth of Functions

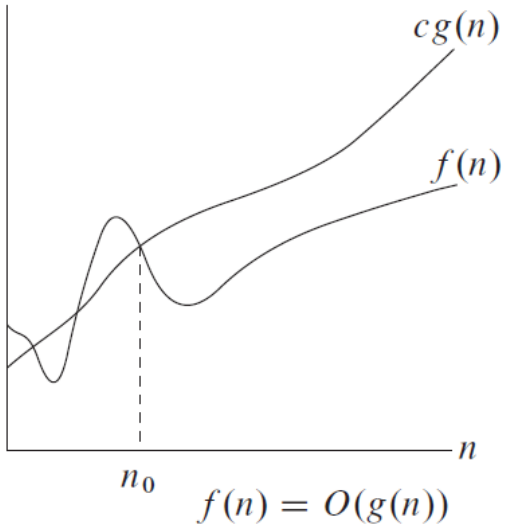
- If f and g are partial functions from $\mathbb{N}$ to $\mathbb{R}^+$; that is, both functions have values that are nonnegative real numbers wherever they are defined



$f(n) = \Theta(g(n))$

(a)

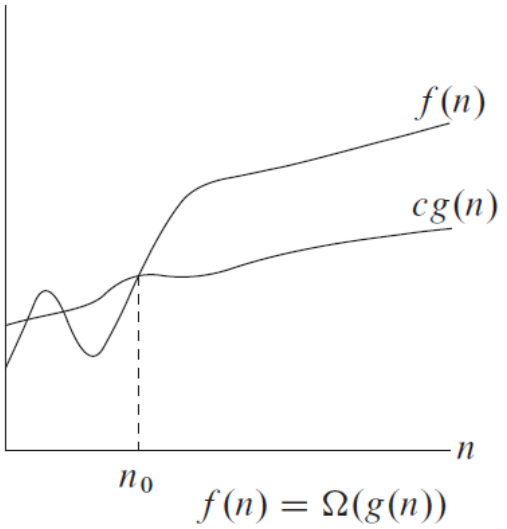
$\frac{1}{2}n^2 - 3n = \Theta(n^2)$



$f(n) = O(g(n))$

(b)

$an^2 + bn + c = O(n^2)$



$f(n) = \Omega(g(n))$

(c)

$an^2 + bn + c = \Omega(n^2)$

Time Complexity of a Turing Machine

- Suppose T is a TM with input alphabet Σ that eventually halts on every input string
  - The *time complexity* of T is the function $\tau_T(n)$ defined by considering, for every input string of length n in Σ^*, the *number of moves* T makes on that string before halting, and letting $\tau_T(n)$ be the *maximum* of these numbers
  - When we refer to a TM with a certain time complexity, it will be understood that it halts on every input

Time Complexity of a Turing Machine

- What constitutes a *tractable* problem?
  - The most common answer is those that can be solved in *polynomial* time on a TM or other computer
  - One reason for this characterization is that it is relatively robust, as problems that can be solved in polynomial time on any computer can be solved in polynomial time on a TM as well, and vice-versa

The *P* Problem

- Definition: *P* is the set of languages *L* such that for some TM *T* deciding *L* and some $k \in \mathbb{N}$, $\tau_T(n) = O(n^k)$
- The set *P* is the set of problems that can be decided by a TM in *polynomial time*, as a function of the instance size. (Brute-force algorithms tend to be exponential)

Time Complexity

- In practice, we "measure" the time complexity of an algorithm by counting the "basic computer steps"

Algorithm 3 TraverseArray(A)

```
1: s = 0
2: for i = 1 to A.length do
3:     j = i
4:     while j ≥ 1 do
5:         s = i + j
6:         j = j/2
7: return s
```

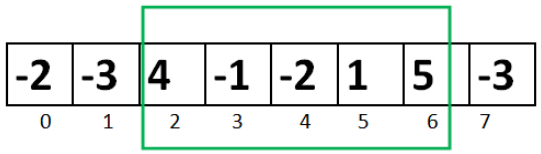
1:	$s = 0$	c_1	1
2:	**for** $i = 1$ to $A.length$ **do**	c_2	n
3:	$j = i$	c_3	n
4:	**while** $j \geq i$ **do**	c_4	$\sum_{j=1}^{n} t_j$
5:	$s = i + j$	c_5	$\sum_{j=1}^{n} (t_j - 1)$
6:	$j = j/2$	c_6	$\sum_{j=1}^{n} (t_j - 1)$
7:	**return** s	c_7	1

$$T(n) = c_1 + c_2 n + c_3 n + c_4 \sum_{j=1}^{n} t_j + c_5 \sum_{j=1}^{n} (t_j - 1) + c_6 \sum_{j=1}^{n} (t_j - 1) + c_7$$
$$= (c_1 + c_7) + (c_2 + c_3 + c_4)n + (c_4 + c_5 + c_6) \sum_{j=1}^{n} \log_2 n.$$

$$T(n) = \mathcal{O}(n \log_2 n).$$

Maximal Continuous Sub-array Sum Problem

Given: an integer array $A = a_1,\ a_2,\ \ldots,\ a_n$.
Output: the sub-array of A starts from index i and ends at index j that has the maximal sum.



Algorithm 1 BruteForceMaxSum(A)

1: $maxSum = A[1]$

$O(n^3)$

Algorithm 2 BruteForceMaxSumFaster(A)

1: $maxSum = A[1]$

$O(n^2)$

Algorithm 3 DivideConquerMaxSum(A)

1: **procedure** MAXSUM(A, L, R)

$O(n\log_2 n)$

Algorithm 4 DPMaxSum(A)

1: $max = a_1$
2: $\mathcal{P}_1 = a_1$
3: **for** $i = 2$ **to** n **do**
4: **if** $\mathcal{P}_{i-1} > 0$ **then**
5: $\mathcal{P}_i = \mathcal{P}_{i-1} + a_i$
6: **else**
7: $\mathcal{P}_i = a_i$
8: **if** $max < \mathcal{P}_i$
9: $max = \mathcal{P}_i$
10: **return** max

$O(n)$

SAT

- An instance of the *satisfiability problem* is a Boolean expression $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$
  - It involves Boolean variables $x_1, x_2, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$
  - It is in conjunctive normal form (the conjunction $\wedge$ of several clauses, each of which is a disjunction $\vee$)
- Question: is there an assignment of truth values to the variables that satisfies the expression (i.e., makes it true)?
  - This problem is clearly decidable – we could simply try every possible assignment of values to variables

TSP

- The *traveling salesman problem* considers *n* cities that a salesman must visit, with a distance specified for every pair of cities
  - It is simplest to formulate this as an optimization problem – determine the order that minimizes the total distance traveled
  - We can turn this into a decision problem by introducing a variable *k* and asking whether there is an order in which the cities could all be visited by traveling *no more than* distance *k*

In Practice …

- There is a brute-force solution to this problem too – consider all $n!$ possible permutations of the cities
- With current hardware we can solve very large problems, if the problems require time $O(n)$
- We can still solve largish problems if they take time $O(n^2)$ or even $O(n^3)$
- Exponential problems are another story
  - If the problem really requires time proportional to 2^n, then doubling the speed of the machine only allows us to increase the size of the problem by 1

However …

- Showing that a brute-force approach takes a long time does not necessarily mean that the problem is complex
- The satisfiability problem and the traveling salesman problem are assumed to be hard, *not* because the brute-force approach takes exponential time, but because no one has found a way of solving either problem that *does not* take at least exponential time

The Set *NP* and Polynomial Verifiability

- The satisfiability problem seems like a hard problem
  - Testing a potential answer is easy, but there are an exponential number of potential answers
- We can approach this problem nondeterministically
  - We "*guess*" an answer (a particular truth assignment) and then test it deterministically
  - This can be done in polynomial time

The Set *NP* and Polynomial Verifiability

- Definition: If T is an NTM with input alphabet Σ such that, for every $x \in \Sigma^*$, every possible sequence of moves of T on input x eventually halts, the time complexity $\tau_T(n)$ is defined as follows:
  - Let $\tau_T(n)$ be the *maximum* number of moves T can possibly make on any input string of length n before halting
  - As before, if we speak of an NTM as having a time complexity, we are assuming implicitly that no input string can cause it to loop forever

The Set *NP* and Polynomial Verifiability

- Definition: *NP* is the set of languages L such that for some NTM T that cannot loop forever on any input, and some integer k, T accepts L and $\tau_T(n) = O(n^k)$
  - We say that a language in *NP* can be accepted in *nondeterministic polynomial time*
  - It is clear that $P \subseteq NP$
  - The *SAT* problem is in *NP* (the "guess-and-test" strategy is typical of problems in *NP*, and we can formalize this by constructing an appropriate NTM)

The Set *NP* and Polynomial Verifiability

- Definition: If $L \subseteq \Sigma^*$, we say that a TM T is a *verifier* for L if:
  - T accepts a language $L_1 \subseteq \Sigma^*\{\$\}\Sigma^*$, T halts on every input, and
  - $L = \{x \in \Sigma^* \mid \text{for some } a \in \Sigma^*, x\$a \in L_1\}$ (we will call such a value a a *certificate* for x)
- A verifier T is a *polynomial-time verifier* if:
  - There is a polynomial p such that for every x and every a in Σ^*, the number of moves T makes on the input string $x\$a$ is no more than $p(|x|)$

The Set *NP* and Polynomial Verifiability

- Theorem: For every language $L \in \Sigma^*$, $L \in NP$ if and only if L is polynomially verifiable
  - i.e., there is a polynomial-time verifier for L
- A verifier for the satisfiability problem could take a specific truth assignment as a certificate; the traveling salesman problem could take a permutation of the cities as a certificate

Polynomial-Time Reductions and *NP*-Completeness

- Just as we can show that a problem is decidable by reducing it to another one that is, we can show that a language is in *P* by reducing it to another that is
  - In the case of decidability, we only needed the reduction to be computable
  - Here we need the reduction function to be computable in polynomial time

Polynomial-Time Reductions and *NP*-Completeness

- Definition: If L_1 and L_2 are languages over respective alphabets Σ_1 and Σ_2, a *polynomial-time reduction* from L_1 to L_2 is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ satisfying two conditions
  - First: for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$
  - Second: f can be computed in polynomial time, i.e., there is a TM with polynomial time complexity that computes f
- If there is a polynomial-time reduction from L_1 to L_2, we write $L_1 \leq_p L_2$ and say that L_1 is polynomial-time reducible to L_2

Polynomial-Time Reductions and *NP*-Completeness

- Polynomial-time reducibility is transitive
- If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$
- If $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$

Polynomial-Time Reductions and *NP*-Completeness

- A language L is *NP-hard* if $L_1 \leq_p L$ for each $L_1 \in NP$
- L is *NP-complete* if $L \in NP$ and L is *NP*-hard
- If L and L_1 are languages such that L is *NP*-hard and $L \leq_p L_1$, then L_1 is also *NP*-hard
- If L is any *NP*-complete language, then $L \in P$ if and only if $P = NP$
- We do not know if $P = NP$, most people assume that *NP* is a larger set, but no one has been able to demonstrate that $P \neq NP$

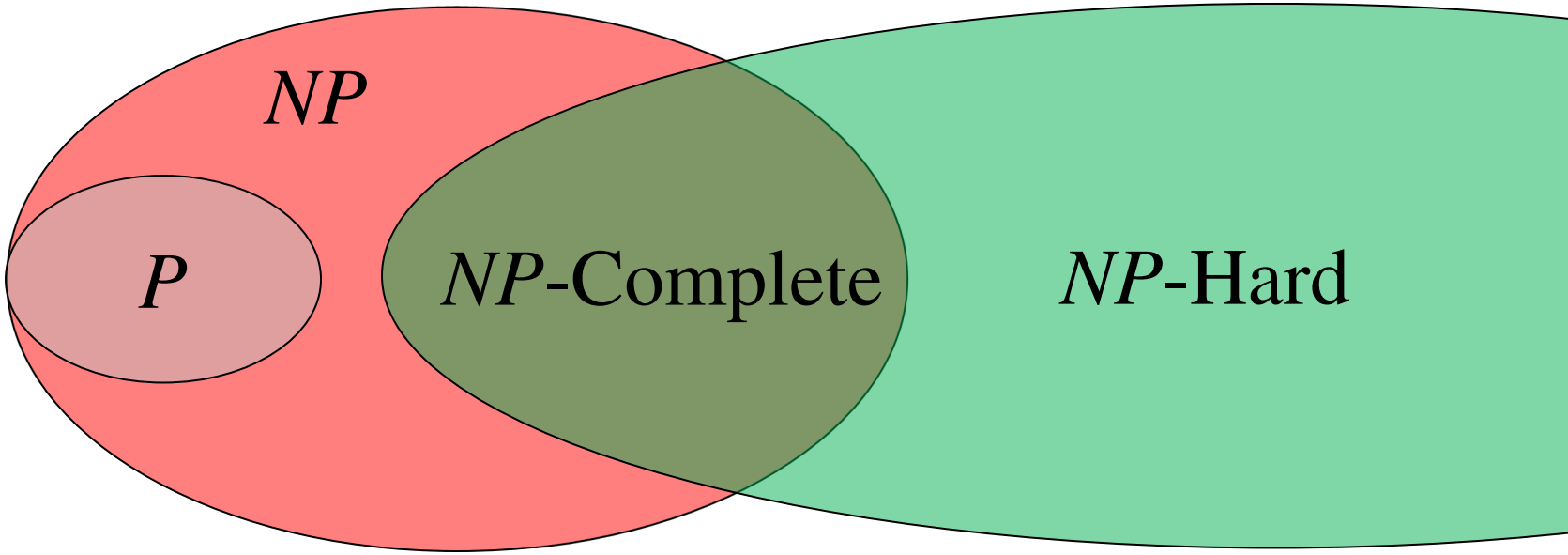
In “Non-Mathematical” Words

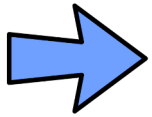
- *P* problems can be solved efficiently (i.e., in polynomial time)
- *NP* problems can be verified efficiently (i.e., in polynomial time)
- *NP-hard* problems are all harder than *NP* problems
- *NP*-Complete problems are the hardest ones in *NP*

In “Visual” Words

If $P \neq NP$

(Many believe this)

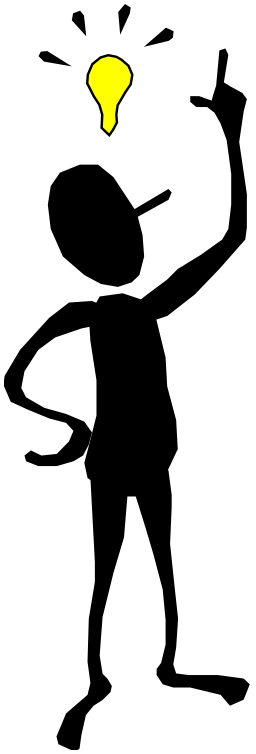


Next 

- How to Prove *NP*-Completeness?

Other Reading Materials

- https://rjlipton.wordpress.com/2009/07/03/is-pnp-an-ill-posed-problem/
- http://sites.math.rutgers.edu/~zeilberg/Opinion98.html
- https://www.bbc.com/news/technology-18327261



Thanks ! ☺

Questions ?