Review: **P,** NP, NPH, NPC

Recall that for decision problems, the algorithm is a search for a solution, if found, answer yes

1

Complexity Classes **P,** NP, NPH, NPC

Only applies to decision problems so we have to convert optimization problems to a corresponding decision problem

2

Example Conversions of Optimization to Decision

- Minimum Spanning Tree *Optimization* Problem:
  - Given a Weighted Graph *G*, find a spanning tree of *G* with the minimum total weight?
- What to do: convert to a decision problem by adding another parameter to the optimization problem, i.e., a *max value* if we are searching for a minimum or a *min value* if we are searching for a maximum.
- Minimum Spanning Tree *Decision* Problem:
  - Given a pair (*G, max*), where G is a graph. Does there exist a spanning tree of G whose total weight is at most *max*?

3

Quiz

- Prove that Subset Sum is a member of NP:

  Subset Sum: Given a triple (S, *max*, *min*), where S is a set of positive integers and *max* and *min* are positive integers. Is there a subset of S such that the sum of the integers in that subset is at most *max* and at least *min*?

  Step 1: Randomly pick a subset of the elements from S and put them in Sequence T

  Step 2: **Algorithm** verifySS (S, max, min, T)

```
sum <- 0
for each e in T do
    sum <- sum + e           // O(n)
if min ≤ sum /\ sum ≤ max then      // O(1)
    return yes
    else return NOT_A_Solution
```


4

Easy to prove members of P are also members of NP

- Three ways
  - Generate a solution using its polynomial time algorithm; if solution matches the guess, then check whether it satisfies the decision criteria
    - Non-deterministic
    - (the reason all members of P are members of NP)
  - Ignore the guess, generate the solution, then check whether solution satisfies decision criteria
    - Deterministic (always returns yes or no in $O(n^k)$ time)
  - Only use the randomly generated guess and check whether guess satisfies decision criteria
    - Non-deterministic
    - (all NP proofs can be done in this way)

5

Prove: $P \subseteq NP$

Claim:

Any problem that can be solved in polynomial time is a member of NP

Proof:

Non-deterministic Polynomial Algorithm:

1. Non-deterministically output a proposed solution (a guess)
2. Compute the correct solution in polynomial time ($O(n^k)$ time)
3. Check whether the proposed solution matches the correct solution in polynomial time (always p(n)=size of w time, why?)
4. Verify that the generated solution satisfies all decision criteria

6

MST-Matching Solution Non-deterministic

- Problem: Does graph G=(V, E) have a spanning tree of weight at most K?
- Non-deterministic Algorithm (**full detail**):

Phase 1. Non-deterministically choose a set of edges from E and insert them into a Sequence T (could specifically choose n-1 edges)

Phase 2.

```
Algorithm checkMST(G , max, T) // check if T contains same edges as in MST
  if T.numEdges() ≠ n-1 then return NOT_A_Solution
  Buruvka-MST(G) // recall that the edges in the MST are labelled
  W ← 0
  for all e ∈ T.elements()  do  // compare edges in T to edges in MST
      If getMSTLabel(e) ≠ IN_MST then return NOT_A_Solution
      W ← W + weight(e)          // add up the edge weights
  if  W > max
      then return NOT_A_Solution
      else return yes
```


MST-Ignore Solution Deterministic

- Problem: Does graph G=(V, E) have a spanning tree of weight at most K?
- Non-deterministic Algorithm (**full detail using Prim-Jarnik**):

Phase 1. Non-deterministically choose a set of edges from E and insert them into a Sequence T

Phase 2. // can only be done like this if MST(G) is a member of P

```
Algorithm checkMST(G, max, T)
  Prim-Jarnik-MST(G) // ignore T; okay since MST runs in O(m log n)
  W ← 0
  for all v ∈ G.vertices()  do
      e ← getParent(v)              // Prim-Jarnik stores MST edges at vertices
      if e ≠ null then
          W ← W + weight(e)         // add up the edge weights in the MST
  if  W > max
      then return no                // deterministically answers in O(n^k) time
      else return yes
```

- The **only** time a verifier can return no is when the problem is a member of P, i.e., a solution can be generated in polynomial time

MST-Non-deterministic (only checks guess T)

Problem: Does graph G=(V, E) have a spanning tree with total weight at most max?

NP Algorithm (even more details using BFS):

1. Non-deterministically choose a Sequence T of edges from E
2. **Algorithm** verifyMST(G,max,T)

```
    if T.size() != G.numVertices()-1 then
        return NOT_A_Solution
    for each e in G.edges() do
        setEdgeOfT(e, NO)
    total ← 0
    for each e in T.elements() do
        setEdgeOfT(e, YES)
        total ← total + weight(e)
    isConnected ← BFS(G)
    if isConnected /\ total ≤ max then
        return yes
    else return NOT_A_Solution
```

```
Algorithm initResult(G)
    components ← 0
Algorithm preComponentVisit(G, v)
    components ← components + 1
Algorithm result(G)
    return (components = 1)
Algorithm postInitEdge(e)
    if getEdgeOfT(e) = NO then
        setLabel(e, SKIP)
```


Template Version of BFS

```
Algorithm BFS(G) {all components}
  Input graph G
  Output labeling of the edges of G as
    discovery edges and cross edges

    initResult(G)
    for all  u ∈ G.vertices() do
        setLabel(u, UNEXPLORED)
        postInitVertex(u)
    for all  e ∈ G.edges() do
        setLabel(e, UNEXPLORED)
        postInitEdge(e)
    for all  v ∈ G.vertices() do
        if  isNextComponent(G, v)
            preComponentVisit(G, v)
            BFScomponent(G, v)
            postComponentVisit(G, v)

    return result(G)

Algorithm isNextComponent(G, v)
    return getLabel(v) = UNEXPLORED
```

```
Algorithm BFScomponent(G, s) {1 component}
    setLabel(s, VISITED)
    Q ← new empty Queue
    Q.enqueue(s)
    startBFScomponent(G, s)
    while  ¬Q.isEmpty()  do
        v ← Q.dequeue()
        preVertexVisit(G, v)
        for all  e ∈ G.incidentEdges(v)  do
            preEdgeVisit(G, v, e)
            if  getLabel(e) = UNEXPLORED
                w ← opposite(v, e)
                edgeVisit(G, v, e, w)
                if getLabel(w) = UNEXPLORED
                    preDiscEdgeVisit(G, v, e, w)
                    setLabel(e, DISCOVERY)
                    setLabel(w, VISITED)
                    Q.enqueue(w)
                    postDiscEdgeVisit(G, v, e, w)
                else
                    setLabel(e, CROSS)
                    crossEdgeVisit(G, v, e, w)
        postVertexVisit(G, v)

    finishBFScomponent(G, s)
```


Generic Non-deterministic Algorithm

- We create a non-deterministic algorithm using verifier V
- We again assume that V returns NOT_A_Solution if the guess is not a **valid** solution

```
Algorithm isMemberOfL(x)
        result ← NOT_A_Solution
        while result = NOT_A_Solution do
1.              w ← randomly guess at a solution from search space
2.              result ← V(x,w)  // must run in polynomial time
        return result // allows returning no from A(x,w) when L ∈ P
```

In a proof that a language is a member of NP, our verifier has to run in polynomial time and has to be substitutable in place of V above.

Reduction of Sorting to Subset Sum

The transformation would use the following algorithm where we only need two instances of Subset Sum:
(S,C) → (R, min, max)

```
Algorithm reduceSortToSS(S, C)
  Input: a Sequence S of elements and a comparator C for possibly sorting
    elements of S
  Output: a Sequence R of integers and the values of max and min that is
    an instance of the Subset Sum problem
    R ← new empty Sequence
    R.insertLast( 5 )
    for  i ← 0 to S.size()-1 do
        if  ¬ C.isComparable(S.elemAtRank(i))
            then return (R, 1, 1)  {integers, max, min}
    return (R, 5, 5)          {integers, max, min}
```

Example reduction

- Consider the following decision problems:
  - **Subset Sum**: Given a triple (S, *min*, *max*), where S is a set of positive sizes and *min* and *max* are positive numbers. Is there a subset of S whose sum is at least *min*, but no larger than *max*?
  - **0-1 Knapsack**: Given a triple (S, *W*, *minB*), where S is a set of (benefit, weight) pairs, *W* is a positive weight, and *minB* is a positive benefit. Is there a subset of S such that the total weight is at most *W* with total benefit at least *minB*?

13

Reduction of Subset Sum to 0-1 Knapsack

Let the (S, min, max) be an instance of Subset Sum. The transformation would use the following algorithm:

(S,min,max) → (P, W, minB)

Algorithm reduceSSto0-1K(S, min, max)

Input: a Sequence S of numbers and the limits min and max from Subset Sum

Output: a Sequence P of pairs (representing benefit and weight) and the values of *W* and *minB* for 0-1 Knapsack

```
P ← new empty Sequence
for i ← 0 to S.size()-1 do
    val ← S.elemAtRank(i)
    P.insertLast( (val, val) )
return (P, max, min)  {pairs, maximum weight, minimum benefit}
```


14

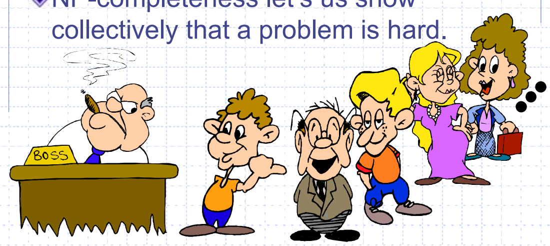
Homework

- Define a polynomial-time reduction from Hamiltonian Path to Longest Path
- First formulate the two problems as decision problems
  - **Hamiltonian Path**: Given a (non-weighted) graph G=(V, E) and two vertices *u*, *v* ∈ V. Is there a simple path from u to v that visits every vertex in V?
  - **Longest Path**: Given a weighted graph G=(V,E), two vertices *u*, *v* ∈ V, and a positive number *min*. Is there a simple path between *u* and *v* with total weight at least *min*?

15

An Approach When Dealing with Hard Problems

- NP-completeness let's us show collectively that a problem is hard.



I couldn't find a polynomial-time algorithm, but neither could all these other smart people.

 (cartoon inspired by [Garey-Johnson, 79])

16

Suppose problem A can be reduced to B in polynomial time

If A $\rightarrow_p$ B,

- then B cannot be easier than A
  - Because A can be solved using the algorithm for B
- If A is NP-hard, then B is NP-hard
  - Since all problems in NP can be reduced to A
- If A is not computable, then B is not computable

Conclusions (review):

- **An easier problem can be reduced to a harder problem (or to one equally as hard)**
  - **This is why many textbooks use $\leq_p$ to indicate reduction in polynomial time (instead of $\rightarrow_p$)**
- NP-hard means at least as hard as any problem in NP, but not necessarily in NP
  - Thus not all NP-hard problems are NP-complete
- If there is a polynomial algorithm for any NP-hard problem, then all NP-complete problems can be solved in polynomial time, i.e., P=NP

17

Review: **P** and NP

- What do we mean when we say a problem is in **P**?
  - A: A solution can be found and verified in polynomial time
- *What do we mean when we say a problem is in **NP**?*
  - A: A non-deterministically proposed solution can be verified in polynomial time
- *What is the relation between **P** and **NP**?*
  - A: **P** ⊆ **NP**, but no one knows whether **P = NP**

18

Review: NP-Complete

- *What, intuitively, does it mean if we can reduce problem A to problem Q?*
  - A is "no harder than" Q or Q is "at least as hard as" A
- *How do we reduce A to Q?*
  - Create algorithm R that transforms, in polynomial time, an arbitrary instance, a, of language A into an instance, R(a), of language Q such that $a \in A$ iff $R(a) \in Q$
- *What does it mean if Q is NP-Hard?*
  - Every problem $A \in$ **NP** can be reduced to Q in polynomial-time
- *What does it mean if Q is NP-Complete?*
  - Q is NP-Hard and $Q \in$ **NP**

19

19

Review: Proving Problems NP-Complete

- *How do we usually prove that a problem Q is NP-Complete?*
  - A: Show $Q \in$ **NP**, and reduce a known NP-Complete problem *A* to *Q*

20

20

Suppose A is a member of P and B is a member of NPC.

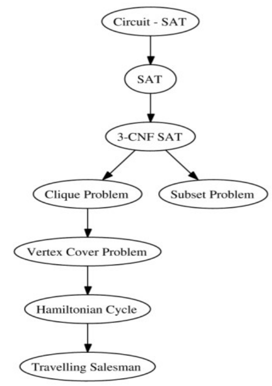
- *Is it possible to reduce problem A to problem B?*
  - Yes. Any member of P can be reduced to any other problem including problems in NPC. All we need is two instances of B. Also, by definition all members of NP ($P \subseteq NP$) can be reduced to members of NPH ($NPC \subseteq NPH$).
- *Is it possible to reduce B to A?*
  - Not unless P=NP (so highly unlikely).
- *Why does a reduction of B to A imply that P=NP?*
  - Because $B \rightarrow_p A$ implies $B \in$ **P** since $A \in$ **P** and every problem $Q \in$ **NP** can be reduced to B by definition of NPC; thus every $Q \in$ **NP** would also be a member of **P**.
- *If P=NP, then would problem A be NP-Complete?*
  - Yes every problem in **P** would be **NPC** and vice versa.

21

21

NP-Complete

- Reducibility of some NP-complete problems
- Therefore, these decision problems are NP-hard (NP-complete since ...)
- Decision problems are reducible, but not the optimization problem
- All NP-complete problems are reducible to each other, by definition



NP-Complete 22

22

Review: Decision Problems Tractable vs. Intractable

- All problems are a decision about whether or not a valid solution exists
  - **Tractable (feasible) problems**:
    - a valid guess can be deterministically generated in polynomial time, then checked in polynomial time, i.e., the problems in complexity class P.
    - OR there exists a polynomial time algorithm that can determine whether or not a solution exists without actually finding it (like sorting or primality).
  - **Intractable (infeasible) problems**:
    - no polynomial time algorithm to deterministically generate a valid guess (or find a solution) has yet been found
    - NP-Complete and NP-Hard problems are considered intractable, but we are not sure
    - Includes problems in NP and others not in NP (such as Halting, Permutations)
  - **Undecidable problems**:
    - there can be no algorithm to validate a guess or decide yes or no
    - must be proven mathematically (e.g., the halting problem)
- Thus there are three categories:
  - Easy (P, tractable), hard (NPH, NPC, intractable), and undecidable (NPH, non-computable)

23

23

General Comments

- Literally many hundreds of problems have been shown to be NP-Complete
  - Some reductions are profound,
  - Some are comparatively easy,
  - Many are easy once the key insight is known
- You can expect a simple reduction or NP-Completeness proof on the final

24

24

Some NP-Complete Problems

- *0-1 knapsack*: when weights are not just integers
- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some target *T*?
- *Hamiltonian path*: Given a graph G, is there a path that visits each vertex exactly once?
- *Hamiltonian circuit*: Given a graph G, is there a cycle that visits each vertex exactly once?
- *TSP*: Given a list of cities and their pair-wise distances, is there a tour that visits each city exactly once with total distance at most D?
- *Graph coloring*: Can a given graph be colored with *k* colors such that no adjacent vertices are the same color?
- *Vertex cover*: Given a graph G, is there a set C of vertices with size K such that each edge is incident to at least one vertex in C?
- Register allocation in compilers, type inference in programming languages,
  - *n* is small enough that brute force or approximation algorithms are useful for solving most practical instances of these problems (i.e., most practical programs)
- Etc…

25

More Graph Problems Which are in NPC?

- Longest Path
  - Given a weighted graph G=(V, E), two vertices *u*, *v* ∈ V, and a positive number K. Is there a simple path between *u* and *v* with total weight at least K?
- Minimum Degree Spanning Tree
  - Given graph G=(V,E) and positive integer K. Is there a spanning tree T =(V,E') such that the maximum degree of any vertex in T is at most K?
- Shortest Total Path Length Spanning Tree
  - Given graph G=(V,E) and positive integer K. Is there a spanning tree T =(V,E') such that the length of the path in T between every pair of vertices u,v∈V is at most K?
- K-minimum Spanning Tree
  - Given graph G=(V,E), positive integer K ≤ |V|, and positive weight W. Is there a tree that spans K vertices with total weight ≤ W?

26

What about?

- Euler Tour
  - Given a graph G=(V, E). Is there a path that visits each edge exactly once?
  - This problem is in P. Yes if exactly zero or two vertices have a degree that is an odd number, all the other vertices must have even degree.

27

How to deal with hard optimization problems?

- Look for ways to reduce the number of computations that have to be done
  - Dynamic programming
  - Branch-and-Bound
- Look for NP-complete problems with a similar structure
  - Approximation

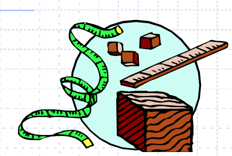
28

How to deal with NP-complete optimization problems?

- Apply an approximation algorithm.
  - Typically faster than an exact solution.
  - Assuming the problem has a large number of feasible solutions.
    - Also, has a cost function for the solutions.
    - Want to find a solution with minimum cost in a reasonable time (i.e. polynomial time).
- Apply Heuristic solution
  - Looking for “good enough” solutions.

29

Approximation Algorithms



30

Outline and Reading

- Approximation Algorithms for NP-Complete Problems (§13.4)
  - Approximation ratios
  - Polynomial-Time Approximation Schemes (§13.4.1)
  - 2-Approximation for TSP special case (§13.4.3)
  - 2-Approximation for Vertex Cover (§13.4.2)
  - Log n-Approximation for Set Cover (§13.4.4)

NP_Complete 31

31

Approximation Ratios

- Optimization Problems
  - We have some problem instance x that has many feasible “solutions”.
  - We are trying to minimize (or maximize) some cost function c(S) for a “solution” S to x. For example,
    - Finding a minimum spanning tree of a graph
    - Finding a smallest vertex cover of a graph
    - Finding a smallest traveling salesperson tour in a graph
- An approximation produces a solution T
  - T is a **k-approximation** to the optimal solution OPT if c(T)/c(OPT) $\leq$ k (assuming a min. prob.)
  - a maximization approximation would be the reverse ($\geq$)

NP_Complete 32

32

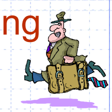
Polynomial-Time Approximation Schemes

- A problem L has a **polynomial-time approximation scheme (PTAS)** if it has a polynomial-time $(1+\varepsilon)$-approximation algorithm, for any fixed $\varepsilon > 0$ (this value can appear in the running time).
- 0/1 Knapsack has a PTAS, with a running time that is $O(n^3/\varepsilon)$. See §13.4.1 in Goodrich-Tamassia for details.

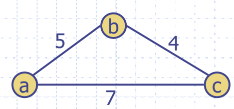
NP_Complete 33

33

Special Case of the Traveling Salesperson Problem



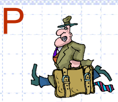
- OPT-TSP: Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex.
  - OPT-TSP is NP-hard
  - Special case: edge weights satisfy the triangle inequality (which is common in many applications):
    - w(a,b) + w(b,c) $\geq$ w(a,c)

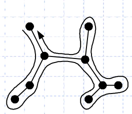


NP_Complete 34

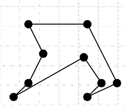
34

A 2-Approximation for TSP Special Case





Euler tour *P* of MST *M*



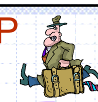
Output tour *T*

Algorithm *TSPApprox*(*G*)
  Input weighted complete graph *G*, satisfying the triangle inequality
  Output a TSP tour *T* for *G*
  M ← a minimum spanning tree for *G*
  P ← an Euler tour traversal of *M*, starting at some vertex *s*
  T ← empty list
  for each vertex *v* in *P* (in traversal order)
    if this is *v*’s first appearance in *P* **then**
      T.insertLast(*v*)
  T.insertLast(*s*)
  return *T*

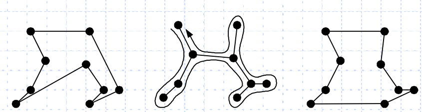
NP_Complete 35

35

A 2-Approximation for TSP Special Case - Proof



- The optimal tour is a spanning tour; hence |M|$\leq$|OPT|.
- The Euler tour P visits each edge of M twice; hence |P|=2|M|
- Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by triangle inequality (w(a,b) + w(b,c) $\geq$ w(a,c)); hence, |T|$\leq$|P|.
- Therefore, |T|$\leq$|P|=2|M|$\leq$2|OPT|

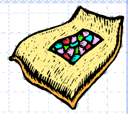


Output tour *T* (at most the cost of *P*) | Euler tour *P* of MST *M* (twice the cost of *M*) | Optimal tour *OPT* (at least the cost of MST *M*)

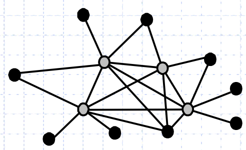
NP_Complete 36

36

Vertex Cover

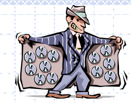


- A **vertex cover** of graph G=(V,E) is a subset W of V, such that, for every (a,b) in E, a is in W or b is in W.
- OPT-VERTEX-COVER: Given an graph G, find a vertex cover of G with smallest size.
- OPT-VERTEX-COVER is NP-hard.
- There is 2-approximation scheme for Vertex Cover (see text)



37

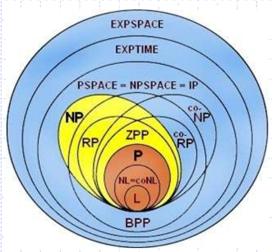
Set Cover



- OPT-SET-COVER: Given a collection of m sets, find the smallest number of them whose union is the same as the whole collection of m sets?
  - OPT-SET-COVER is NP-hard
- Greedy approach produces an O(log n)-approximation algorithm. See §13.4.4 for details.

38

Additional Complexity Classes of Computable Problems



From MIT Web Page for Theory of Computation Course

39

Complexity classes L and NL

- L is the class of decision problems that can be solved using logarithmic space
- NL is the class of decision problems that can be solved non-deterministically using logarithmic space
- $L \subseteq NL \subseteq P$
- Open question: Is L=NL=P?

40

Probabilistic (Randomized) Algorithms

- Algorithms that use some degree of randomness as part of their logical structure
- Examples:
  - Quicksort, Quickselect, Skip List
  - Non-deterministic Algorithms

41

Verifier

- Definition:
  - A *verifier* for a language L is an algorithm V such that
    - If $x \in L$, then there exists a string w such that V(x,w)=yes
    - If $x \notin L$, then for all strings w, V(x,w)=no
  - If V(x,w)=yes, then w is called a *witness* or a *certificate* (or guess) that verifies that $x \in L$
  - Note that the no answer is based on the collection of all strings, whereas the yes answer is based on the existence of one string w
    - This is what helped me understand the difference between NP and Co-NP
- In the complexity classes of interest, all of the verifiers must run in polynomial time

42

Language Verification Algorithms (Notes)

- A *language verifier* is a two-argument algorithm V(x, w), where x is an instance of the problem and w is a string called a certificate or witness (guess).
- The language verified by algorithm V is
  $L=\{x\in\{0,1\}^* \mid \exists\, w\in\{0,1\}^* \text{ such that } V(x,w)=yes\}$
- Several points to note about this definition:
  - String w corresponds to the non-deterministic guess
  - No claims are made about using V to verify/determine when an arbitrary string x is NOT in L!!!!
  - No claims are made about what V(x,w) returns or how long it takes when w is not a **valid** solution

43

43

More Observations

- We are dealing with decision problems that answer yes/no
- A yes answer means that we found a solution (there exists)
- A no answer only happens after searching through every possible solution in the search space without finding a solution (for all)
- The language verified by algorithm V is
  $L=\{x\in\{0,1\}^* \mid \exists\, w\in\{0,1\}^* \text{ such that } V(x,w)=yes\}$

  A mathematician would say that this definition is deterministic. Why?

  No guessing is taking place in the definition!!!

  How would we narrow the structure of w so V(x,w) runs in polynomial time whether or not w is a valid solution?

  How would we implement this deterministically?

44

44

Deterministic implementation of the mathematical definition

- We assume that V(x,w) returns NOT_A_Solution if w is not a **valid** solution to help clarify what we are doing

```
Algorithm isMemberOfL(x)
    solutionSpace ← generate all the possible solutions & put in an iterator
        // note that the solution space could be exponential (power set)
        // or factorial (permutations), etc. may not terminate if infinite size
    result ← NOT_A_Solution
    while result = NOT_A_Solution Λ solutionSpace.hasNext() do
        w ← solutionSpace.nextObject()
        result ← V(x,w)  // L ∈ NP, only requires that V run in O(n^k) time

    if result = NOT_A_Solution then
        return no
    else
        return result  // this allows us to return no from V(x,w) when L ∈ P
```

45

45

Complexity Class NP

- A $\in$ NP: Non-deterministic polynomial time
  - If there exists a verifier V for language A that runs in polynomial time
    - w is randomly (non-deterministically) generated
    - If x $\notin$ A, then V(x,w) always returns no
    - If x $\in$ A, then V(x,w) eventually returns yes
      - V keeps returning no until a certificate/witness w is found
      - i.e., if there exists a string w (guess) that verifies that the answer is yes, then w will eventually be the guess
  - Note that the string w (guess) could be a proof that the answer is yes; but the length of the proof must be polynomial in the input string size |x| (a requirement of guesses)

46

46

Non-deterministic Algorithm

- We create a non-deterministic algorithm using verifier V
- We again assume that V returns NOT_A_Solution if the guess is not a **valid** solution

```
Algorithm isMemberOfL(x)
        result ← NOT_A_Solution
        while result = NOT_A_Solution do
1.              w ← randomly guess at a solution from search space
2.              result ← V(x,w)  // must run in polynomial time
        return result // allows returning no from V(x,w) when L ∈ P
```

In a proof that a language is a member of NP, our verifier has to run in polynomial time and has to be substitutable in place of V above.

47

47

Complexity Classes

- Co-NP
  - Problem A$\in$Co-NP if and only if the complement of A$\in$NP
    - There exists a verifier V that runs in polynomial time
    - w is randomly (non-deterministically) generated
    - If x $\in$ A, then V(x,w) always returns no
    - If x $\notin$ A, then V(x,w) eventually returns yes if there exists a string w (guess) that verifies that x $\notin$ A
      - (i.e., V keeps returning no until a certificate/witness w is found that verifies that x is in the complement of L)
  - Note that the guess (or proof) w verifies that the instance x is <u>not</u> a member of language A, i.e., that x is in the complement of A
  - Thus w could be thought of as a counter example showing that x cannot be in A

48

48

Complexity Classes

- Clearly $P \subseteq NP$ and $P \subseteq$ Co-NP
  - Since we can verify whether the answer is either yes or no in polynomial time no matter what the string w is
    - (e.g., for problems in P, we can ignore w and always compute the correct yes/no answer in polynomial time)

49

49

Complexity Classes

- RP: Randomized polynomial time
  - Verifier V runs in polynomial time
    - If answer is no, V(x,w) always returns no
    - If answer is yes, V(x,w) returns yes with probability 1/2 (if run *m* times, then probability of getting at least one yes is $1-1/2^m$)
  - Intuitively: If the answer is yes, then the algorithm answers yes half the time (or better) on average
    - (perhaps through some polynomial time algorithm that can produce better guesses than required by an NP algorithm)
- ZPP: Zero-error probabilistic polynomial time
  - Verifier runs in polynomial time
    - Returns yes / no / do-not-know
    - If answer is yes, returns yes with probability $\geq$ 1/2 (or returns do-not-know)
    - If answer is no, returns no with probability $\geq$ 1/2 (or returns do-not-know)
  - ZPP = RP $\cap$ Co-RP

50

50

Complexity Classes

- BPP: Bounded-error probabilistic polynomial time
  - Verifier runs in polynomial time
    - If answer is yes, returns yes with probability $\geq$ 2/3
    - If answer is no, returns no with probability $\geq$ 2/3
  - These algorithms can be practical in the sense that we can run them several times, then the answer will be (with high probability) the answer we got the majority of times (the more runs, the higher the probability we have the right answer)
  - Thus this is the largest class that has an efficient probabilistic algorithm (one that would be practical to run on a computer), so BPP is one of the largest practical complexity classes
  - Clearly **BPP=Co-BPP**
  - **Unsolved problem:** Is P = BPP?

51

51

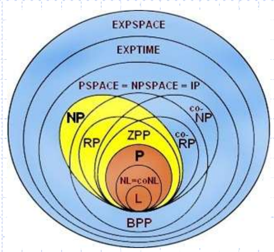
Summary

- NP only requires the existence of a witness/certificate/guess that verifies membership
  - Which could take exponential time to find
- RP and ZPP require that there be lots of witnesses (over half of the guesses produce a witness)
- BPP does not require witnesses, although a witness is sufficient to prove membership
  - Instead, the verification algorithm only has to return the right answer more often than the wrong answer (2/3 of the time)

52

52

Additional Complexity Classes of Computable Problems



From MIT Web Page for Theory of Computation Course

53

53