Problems and Solutions

- Let *P* be a problem and *S* be the set of all solutions to the problem.
- Decision Problem: Is *S* empty?
- Counting Problem: What is the size of *S*?
- Search Problem: find an element of *S*
- Enumeration Problem: find *S*

Problem Populations and Searching

- Let *P* be a problem that can be described by an population of *n* things, and *S* be the set of all solutions to the problem.

- Linear Search: find those elements of the population that satisfy some property.
  e.g. find all the students in the class whose first name is “John”

- Subset Search: find those subsets of the population that satisfy some property.
  e.g. find all the subsets of students in the class who have been in a vehicle together at the same time.

- Combination Search: find those orderings of the population that satisfy some property.
  e.g. find all the orderings of students in the class such that no two adjacent students have the same first name.

Exhaustive Searches

- Problems on populations of size **n** can usually be solved with an exhaustive search.

- **Linear Search:** find those elements of the population that satisfy some property.
  e.g. find all the students in the class whose first name is “John”

  There are **n** combinations to search (linear search)

- **Subset Search**: find those subsets of the population that satisfy some property.
  e.g. find all the subsets of students in the class who have been in a vehicle together at the same time.

  There are $\mathbf{2^n}$ combinations to search

- **Combination Search**: find those orderings of the population that satisfy some property.
  e.g. find all the orderings of students in the class such that no two adjacent students have the same first name.

  There are **n!** combinations to search

Efficiency

- Here are three possible definitions of solving a problem efficiently
- Solving the problem without un-necessary waste
- Solving the problem faster than an exhaustive search could.
- Solving the problem in polynomial-time: an amount of time that can be bounded by some polynomial of the populations size n.
  e.g. *T(n)* the time to solve a problem of size in is at most *p(n)* for some polynomial in $p(n) = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \ldots + a_0$
  (Significance is that this would be faster than exhaustive search of subsets (2^n) or orderings (n!)

Big Oh Notation

- Let *T(n)* be the time it takes an algorithm to solve a problem of size *n*.

- *T(n)* is *O(f(n))* if there exists constants *k* and n_0 such that

$$T(n) \leq kf(n)$$

for $n > n_0$.

In other words: *kf(n)* is an upper bound on the time required to solve problems of size greater than n_0.

Big Omega Notation

- Let *T(n)* be the time it takes an algorithm to solve a problem of size *n*.

- *T(n)* is $\Omega(f(n))$ if there exists constants *k* and n_0 such that

$$T(n) \geq kf(n)$$

for $n > n_0$.

In other words: *kf(n)* is an lower bound on the time required to solve problems of size greater than n_0.

- Lower bounds are much harder to prove than upper bounds

Big Oh Notation -Goals

- We want to get the tightest upper bound we can.

- We would not say $T(n)$ is $O(n^2)$ if we can say, or prove, that $T(n)$ is $O(nlog(n))$ or even better $T(n)$ is $O(n)$.

- We want to always give the simplest bound we can.
  We would not say T(n) is $O(n^2 + n)$ because $n^2 + n$ *is* $O(n^2)$.

  Proof: $n^2 + n < 2n^2$ *so* $n^2 + n < kn^2$ for some k, n>1.
  hence $n^2 + n$ *is* $O(n^2)$

- When we say $T(n)$ *is* $O(f(n))$ we mean the worst case time to solve any problem of size n, even though there may be some problems of size n that are easier to solve. (Oh-notation gives a pessimistic upper bound.)

…Big Oh Notation -Goals

- We are usually interested in determing if *T(n)* is one of the following.

O(1)	constant time
O(log(n))	logarithmic time
O(n)	linear time
O(nlog(n))	super linear time
$O(n^2)$	quadratic time
$O(n^k)$	polynomial time
$O(2^n)$	exponential time
O(n!)	super-exponential time

Big Omega Notation -Goals

- We want to get the tightest lower bound we can.

- We would not say $T(n)$ is $\Omega(n)$ if we can say, or prove, that $T(n)$ is $\Omega(n\log(n))$ or even better $T(n)$ is $\Omega(n^2)$.

- We want to always give the simplest bound we can.
  We would not say T(n) is $\Omega(n^2 + n)$ because $n^2 + n$ *is* $\Omega(n^2)$.

  Proof: $n^2 + n > n^2$ *so* $n^2 + n > kn^2$ for some k, n>1.
  hence $n^2 + n$ *is* $\Omega(n^2)$

- When we say $T(n)$ *is* $\Omega(f(n))$ we mean that some problems of size n will require at least $kf(n)$ time to solve, though there might be some problems of size n that can be solved more quickly.

Mathematical Rules for Big Oh notation

- $f(n)$ is $O(af(n))$ for any constant $a > 0$.
- If $f(n) = g(n)$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
- $f(n) + g(n)$ is $O(\max(f(n),g(n))$.
- If $g(n)$ is $O(h(n))$, then $f(n)+g(n)$ is $O(f(n) + h(n))$.
- If $g(n)$ is $O(h(n))$, then $f(n)g(n)$ is $O(f(n)h(n))$.
- If $f(n)$ is a polynomial of degree d
  (i.e. $f(n) = a_0 + a_1 n + ... + a_d n^d$), then $f(n)$ is $O(n^d)$.
- n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.
- $\log(n^x)$ is $O(\log n)$ for any fixed $x > 0$.
- $\log(x^n)$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.

Useful Log and Exponentiation Identities

$$\log_b a/c = \log_b a - \log_b c$$

$$\log_b a = c \Leftrightarrow a = b^c$$

$$\log_b ac = \log_b a + \log_b c$$

$$\log_b a = (\log_c a)/(\log_c b)$$

$$\log_b(a^c) = c\log_b a$$

$$b^{\log a} = a^{\log b}$$

$$(b^a)^c = b^{ac}$$

$$b^a b^c = b^{a+c}$$

$$b^a / b^c = b^{a-c}$$

Useful Summation Series

$$\sum_{i=1}^{n} i = n(n+1)/2$$

$$\sum_{i=1}^{n} 1/2^i = 1 - 1/2^n$$

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6$$

$$\sum_{i=0}^{n} a^i = (a^{n+1} - 1)/(a-1) \quad \text{for } a > 1$$

$$\sum_{i=1}^{\log n} n = n \log n$$

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1$$

$$\sum_{i=0}^{\infty} a^i = 1/(1-a) \quad \text{for } 0 < a < 1$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Useful Growth Functions:

$\log_2(n!)$ is $O(n\log(n))$ and $\Omega(n\log(n))$

Bubble Sort (Java)

```
int numbers[] = { 43, 5, 67, 3, 12, 54, 89, 345, 45, 4, 67, 45, 457, 0};

    //Display the input
     for(int i=0; i<numbers.length; i++)
       System.out.print(numbers[i] + ", ");
     System.out.println("");

     //Bubble Sort
     for(int i=0; i<numbers.length-1; i++)
       for(int j = 0; j< numbers.length-1-i; j++)
         if(numbers[j] > numbers[j+1]){
                 int temp = numbers[j];
                 numbers[j] = numbers[j+1];
                 numbers[j+1] = temp;
         }

    //Display the result
    for(int i=0; i<numbers.length; i++)
       System.out.print(numbers[i] + ", ");
     System.out.println("");
```

43, 5, 67, 3, 12, 54, 89, 345, 45, 4, 67, 45, 457, 0,
0, 3, 4, 5, 12, 43, 45, 45, 54, 67, 67, 89, 345, 457,
Press any key to continue...

Bubble Sort (Analysis)

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} k$$

$$= k\sum_{i=0}^{n-1} n - i \quad = \quad k(\sum_{i=0}^{n-1} n \; - \; \sum_{i=0}^{n-1} i)$$

$$= k(nn - n(n-1)/2) \quad = \quad \frac{k}{2}(n^2 + n)$$

$$\leq \frac{k}{2}(n^2 + n^2) \quad = \quad kn^2 \qquad \text{For n >= 1}$$

$$So \quad T(n) \quad is \quad O(n^2)$$

Big Oh of a Polynomial Function

Let $T(n) = an^3 + bn^2 + cn + d$ For integer values a,b,c,d

Claim: $T(n)$ *is* $O(n^3)$

Proof: $T(n) <= |a|n^3 + |b|n^2 + |c|n + |d|$

$<= |a|n^3 + |b|n^3 + |c|n^3 + |d|n^3$

$= (|a| + |b| + |c| + |d|)n^3$

$= kn^3$ for n>=1

which is, by definition $O(n^3)$

This can be generalized to a polynomial of any degree

Mystery Method (Java)

```
public class Mystery {

   public static void mystery(int[][][] a){

     for(int i=0; i<a.length-1; i++)
      for(int j = i; j< a.length; j++)
        for(int k = 0; k < j; k++){
          a[i][j][k] = i*j*k;
          System.out.println(i + "," + j + "," + k + ": " + a[i][j][k]);
        }
   }

   public static void main(String args[]) {

    int numbers[][][];
    int n = 10; //n is problem size. 10 is just an example
    numbers = new int[n][n][n];

    mystery(numbers);

   } //end main
}
```

Analysis of Mystery (Java)

$$T(n) = \sum_{i=0}^{n-2}\sum_{j=i}^{n-1}\sum_{k=0}^{j-1} C$$ `//Based on indices in java code`

$$T(n) = C\sum_{i=0}^{n-2}\sum_{j=i}^{n-1} j \qquad \text{since} \qquad \sum_{k=0}^{j-1} 1 = \sum_{k=1}^{j} 1 = j$$

$$T(n) = C\sum_{i=1}^{n-1}(\sum_{j=1}^{n-1} j - \sum_{j=1}^{i-1} j) = C\sum_{i=1}^{n-1}(\frac{n(n-1)}{2} - \frac{i(i-1)}{2})$$

$$T(n) = \frac{C}{2}[(n-1)n(n-1) - \sum_{i=1}^{n-1}(i^2 - i)$$

$$T(n) = \frac{C}{2}[(n-1)n(n-1) - \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i$$

$$T(n) = \frac{C}{2}[(n-1)n(n-1) - (\frac{2(n-1)^3 + 3(n-1)^2 + (n-1)}{6}) + \frac{n(n-1)}{2}]$$

…Analysis of Mystery (Java)

$$T(n) = \frac{C}{2}[(n-1)n(n-1) - (\frac{2(n-1)^3 + 3(n-1)^2 + (n-1)}{6}) + \frac{n(n-1)}{2}]$$

$$= \frac{C}{2}[(n-1)^3 + (n-1)^2 - \frac{(n-1)^3}{3} - \frac{(n-1)^2}{2} - \frac{(n-1)}{6} + \frac{(n-1)^2}{2} + \frac{(n-1)}{2}]$$

$$= \frac{C}{2}[\frac{2}{3}(n-1)^3 + (n-1)^2 + \frac{(n-1)}{3}]$$

$$= \frac{C}{6}[2n^3 - 6n^2 + 6n - 2 + 3n^2 - 6n + 3 + n - 1]$$

$$= \frac{C}{6}[2n^3 - 3n^2 + n] \quad \text{which is } O(n^3)$$

Binary Search (Java)

```
private static boolean search(int[] array, int element, int start, int
   end){
//search for element in sorted array a between start index and end index

    //basis case:
       if(start > end) return false;
       if(start == end) return array[start] == element;
    //recursive case
       int midpoint = (start + end)/2;

       if(array[midpoint] == element) return true;

       if(array[midpoint] < element)
          return search(array, element, midpoint+1, end);
       else return search(array,element, start, midpoint-1);

    }

public static boolean search(int [] array, int element){
    //search for element in the sorted array a
       return search(array, element, 0, array.length-1);

    }
```

Analysis of Binary Search

$$T(n) = k + T(\frac{n}{2})$$ Fixed amount of work plus new problem half as big

$$= k + k + T(\frac{n}{4}) \quad = k + k + T(\frac{n}{2^2})$$

$$= k + k + k + T(\frac{n}{8}) \quad = k + k + k + T(\frac{n}{2^3})$$

$$= k + k + k + \ldots + T(\frac{n}{n}) \quad = k + k + k + \ldots + T(\frac{n}{2^{\log_2(n)}})$$

$$T(n) = \sum_{i=0}^{\log_2(n)} k \quad = k(\log_2(n) + 1) \quad \leq 2k \log_2(n)$$ for n > 1

$$So \quad T(n) \quad is \quad O(\log(n))$$

Linear Recursion (Java)

```
private static int sum(int[] array, int start, int end){
   //sum the elements from start to end

   //basis case:
       if(start >= end) return 0;

   //recursive case

       return array[start] + sum(array, start+1, end);


   }
   public static int sum(int [] array){
  //sum the elements of array
       return sum(array, 0, array.length);

   }
```

Analysis of Linear Recursion

$T(n) = k + T(n-1)$ Fixed amount of work plus new problem one smaller

$T(n) = k + k + T(n-2)$

$T(n) = k + k + k + T(n-3)$

$T(n) = k + k + k + \ldots + T(0)$ Basis case is problem of size 0

$T(n) = \sum_{i=0}^{n} k = k(n+1) \leq 2kn$ for n >= 1

$So \quad T(n) \quad is \quad O(n)$

Palindrome Test

```
public static boolean isPalindrome(String s) {
//answer whether String s is a palindrome
//i.e. reads the same backwards and forwards: "level", "civic"

        //BASIS CASES
        if (s.length() <= 1)
            return true;

        //RECURSIVE STEP
        if (s.charAt(0) == s.charAt(s.length() - 1) ) {
            return isPalindrome(s.substring(1, (s.length() - 1)));
        }
        else
            return false;
 }

 public static void main(String args[]) {

    String s1 ="civic";
    String s2 = "toyota";
    System.out.println(s1 + " is palindrome: " + isPalindrome(s1));
    System.out.println(s2 + " is palindrome: " + isPalindrome(s2));
 } //end main
```

Analysis of isPalindrome()

$$T(n) = k + c(n-2) + T(n-2); \quad T(1) = k, T(0) = k$$
$$= k + c(n-2) + k + c(n-4) + T(n-4)$$
$$= k + c(n-2) + k + c(n-4) + k + c(n-6) + T(n-6)$$

$$T(n) = \sum_{i=1}^{n/2}(k + c(n-2i)) = \sum_{i=1}^{n/2} k + c\sum_{i=1}^{n/2}(n-2i)$$

$$= \sum_{i=1}^{n/2} k + c\sum_{i=1}^{n/2} n - 2c\sum_{i=1}^{n/2} i = \frac{kn}{2} + \frac{cn^2}{2} - 2c\left(\frac{\frac{n}{2}(\frac{n}{2}+1)}{2}\right)$$

$$= \frac{kn}{2} + \frac{cn^2}{2} - \frac{cn^2}{4} - \frac{cn}{2} = \frac{cn^2}{4} + \frac{(k-c)n}{2}$$

So $T(n)$ *is* $O(n^2)$

A Better Palindrome Test

```
private static boolean isPalindrome(String s, int start, int end) {
//answer whether String s is a palindrome
//i.e. read the same backwards and forwards: "level", "civic"

       //BASIS CASES
       if (start >= end)
           return true;

       //RECURSIVE STEP
       if (s.charAt(start) == s.charAt(end) ) {
           return isPalindrome(s, start+1, end-1);
       }
       else
           return false;
}

public static boolean isPalindrome(String s){
      return isPalindrome(s, 0, s.length()-1);
}
```

Analysis of a better isPalindrome()

$$T(n) = k + T(n-2); \quad T(1) = k, T(0) = k$$
$$= k + k + T(n-4)$$
$$= k + k + k + T(n-6)$$

$$T(n) = \sum_{i=1}^{n/2} k = \frac{kn}{2}$$

$$So \quad T(n) \quad is \quad O(n)$$

Exercise: Show this generalizes for any situation where a problem is made smaller by a fixed amount. That is, for any constant k and c, the recurrence relation

$$T(n) = k + T(n-c) \quad is \quad O(n)$$

Solving Two Half Problems

```
private static int sum(int[] array, int start, int end){
//sum the elements from start to end

    //basis case:
      if(start == end) return array[start];
      if(start > end) return 0;

    //recursive case
      int midpoint = (start+end)/2;
      return sum(array, start, midpoint) + sum(array, midpoint+1, end);

 }


 public static int sum(int [] array){

        return sum(array, 0, array.length-1);
 }
```

Analysis of Linear Recursion

$$T(n) = k + 2T(n/2); \quad T(0) = T(1) = c$$

$$= k + 2(k + 2T(\frac{n}{4})) \quad = \quad k + 2(k + 2T(\frac{n}{2^2}))$$

$$= k + 2(k + 2(k + 2T(\frac{n}{8}))) \quad = \quad k + 2(k + 2(k + 2T(\frac{n}{2^3})))$$

$$= k + 2(k + 2(k + ... + 2(k + 2T(\frac{n}{2^{.\log_2 n}})...)))$$

$$= k + 2k + 2^2 k + .2^3 k + ... + 2^{\log_n(n)} k$$

$$= k \sum_{i=0}^{\log_2(n)} 2^i \quad = k(2^{\log_2(n)+1} - 1) \qquad = k(2n - 1)$$

$$So \quad T(n) \quad is \quad O(n)$$

Exercise: Show this generalizes for any situation where you solve d problems of size n/d

Fibonacci Numbers (Java)

$$Fib(n) = Fib(n-1) + Fib(n-2); \quad Fib(0) = Fib(1) = 1$$

```
public static long fibonacci(int n) {
//answer the n'th fibonnaci number

 //BASIS CASES
 if(n<0){
    System.out.println("ERROR:);
    System.exit(-1);
 }
 if (n==0) return 1;
 if (n==1) return 1;

 //RECURSIVE STEP
 return fibonacci(n-1) + fibonacci(n-2);
}
```

Analysis of Fibonacci

$$T(n) = k + T(n-1) + T(n-2); \quad T(0) = T(1) = k$$

$$= k + k + T(n-2) + T(n-3) + T(n-2) = 2k + 2T(n-2) + T(n-3)$$

$$= 2k + 2(k + T(n-3) + T(n-4)) + T(n-3) = 4k + 3T(n-3) + 2T(n-4)$$

$$= 4k + 3(k + T(n-4) + T(n-5)) + 2T(n-4) = 7k + 5T(n-4) + 3T(n-5)$$

What is the pattern??

Alternative Analysis of Fibonacci

$$T(n) = k + T(n-1) + T(n-2); \quad T(0) = T(1) = k$$

$$k + 2T(n-2) \leq T(n) \leq k + 2T(n-1)$$

$$k + 2(k + 2T(n-4)) \leq T(n) \leq k + 2(k + 2T(n-2))$$

$$k + 2(k + 2(k + 2T(n-6))) \leq T(n) \leq k + 2(k + 2(k + 2T(n-3)))$$

$$k\sum_{i=0}^{n/2} 2^i \leq T(n) \leq k\sum_{i=0}^{n} 2^i$$

$$k(2^{n/2+1} - 1) \leq T(n) \leq k(2^{n+1} - 1)$$

$$?? \leq T(n) \leq C_2 2^n$$

So is $T(n)$ *is* $\Omega(2^n)$ *and* $O(2^n)$*?*

Faster Fibonacci

```
public class LongDuple { //object representing a pair of long's
  public long value1, value2;
  public LongDuple(long l1, long l2){ value1 = l1; value2 = l2;}
}
private static LongDuple fastFibonacci(long n) {

    //BASIS CASES
    if (n == 0) return new LongDuple(1,0);
    if (n == 1) return new LongDuple(1,1);
    if (n < -1)
      {System.out.println("ERROR "); System.exit(0);}

     //RECURSIVE STEP
     LongDuple previous = fastFibonacci(n-1);
     return new LongDuple(previous.value1 + previous.value2, previous.value1);
 }

public static long fibonacci(int n) {

        LongDuple result = fastFibonacci(n);
        return result.value1;
}
```

Analysis of Faster Fibonacci

$$\begin{aligned} T(n) &= k + T(n-1); \quad T(1) = k, T(0) = k \\ &= k + k + T(n-2) \\ &= k + k + k + \ldots + T(1) \end{aligned}$$

$$T(n) = \sum_{i=1}^{n} k = kn$$

$$So \quad T(n) \quad is \quad O(n)$$

You definitely do not want to come up with an exponential time algorithm to solve a problem that can be solved in linear time!!

Difficult Problems

Suppose your boss asks you to write an efficient algorithm to solve the Traveling Salesman problem.

Traveling Salesman Problem:

- **INSTANCE: A finite set of "cities", a positive integer "distance" between each pair of cities.**
- **OBJECTIVE: Find a "tour" of all the cities such that the distance travelled is minimum.**

- **More formally,**
  Instance:

$$C = \{c_1, c_2, ..., c_n\} \quad \text{distances} \quad d(c_i, c_j) > 0 \quad i, j = 1...n$$

Objective:
find an ordering:

$$< c_{k1}, c_{k2}, ..., c_{kn} > \quad \text{that minimizes} \quad \sum_{i=1}^{n-1} d(c_{ki}, c_{ki+1}) + d(c_{kn} + c_{k1})$$

What does efficient mean?

Two different definitions of efficiency

- Practical: An efficient algorithm is one that does not use (much) more time and memory than is necessary.

  If your algorithm running time *T(n)* is *O(f(n))* and the problem is *W(f(n))* we can call the algorithm “optimal”.

- Theoretical: An efficient algorithm is one that can solve a problem of size *n* in polynomial time. (*T(n*) is *O(f(n))* for some polynomial *f(n))* For example:

$$T(n) = an^3 + bn^2 + cn + d$$

  That is, it does not take exponential time, or worse to solve the problem

The Problems All Admit an Exponential Time Solution

You can solve the traveling salesman problem, or any other problem you encounter in this course, in exponential, or worse, time as follows.

- Generate all possible orderings of the cities. Add the distances for each possible tour and remember the best one.

  This algorithm would be *O(n!)* which is worse than exponential time and so is not theoretically efficient.

- Suppose you cannot come up with a better algorithm, what are you going to tell your boss?

What to tell your boss?

Option 1) “I found an efficient algorithm” (Yeah!)

Option 2) Prove the problem is intractable (no efficient algorithm can exist). That is, the problem has an exponential time lower bound.

Option 3) Say to your boss: “I guess I’m just to dumb.”

Option 4) Give evidence that the problem might be intractable. (Show that if you could solve this problem efficiently, you could then solve other problems that people have thus far found very difficult.)

…Traveling Salesman Problem

To date nobody has come up with an efficient algorithm to solve the traveling salesman problem.

To date nobody has been able to prove that the problem is intractable (has an exponential lower bound)

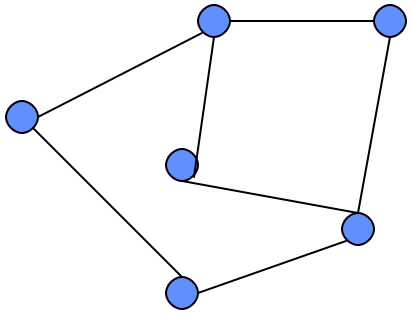
They have given evidence that the problem might be hard by mapping it to the famous hamiltonian cycle problem (which is considered to be hard).

Hamiltonian Cycle Problem

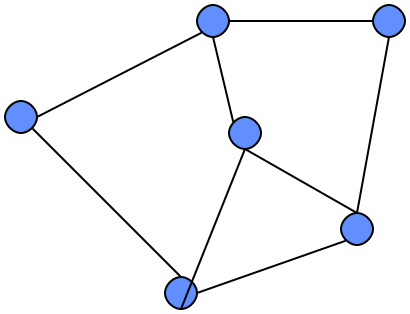
Instance: Given a Graph *G=(V,E)* with set of vertices *V* and set of edges *E*.

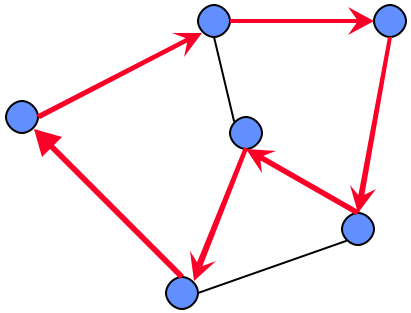
Question: Do the edges of the graph allow a tour of the graph?

NO



YES





Hamiltonian Cycle Problem (Formal)

Instance:

$$G = (V, E) \quad V = \{v_1, v_2, ..., v_n\}, E = \{e_1, e_2, ..., e_m\}$$

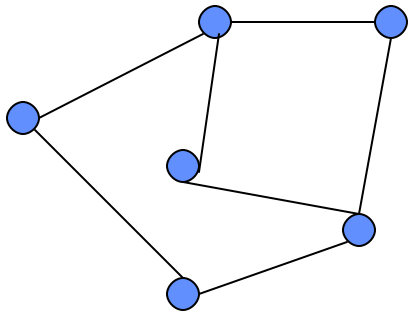
Question:

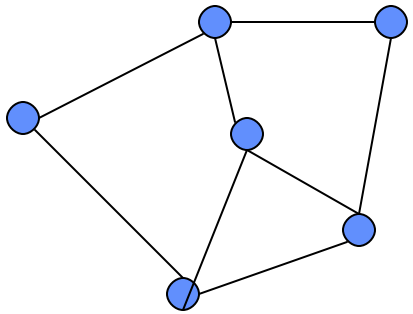
Is there an ordering

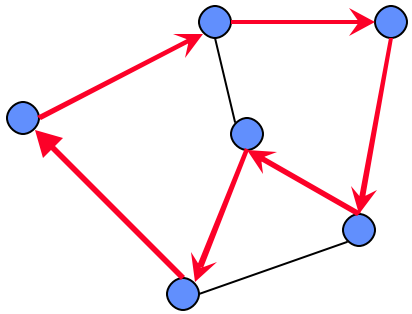
$$< v_{k1}, v_{k2}, ..., v_{kn} >$$

such that

$$(v_{ki}, v_{ki+1}) \in E, \text{ for } i = 1...n-1 \quad \text{and } (v_{k1}, v_{kn}) \in E$$







Claim: TSP is at least has hard at HC

Claim: Traveling salesman problem is at least as hard as the Hamilton Cycle Problem.

Proof: We will show that if you could somehow solve the Traveling Salesman problem efficiently, then you would also be able to solve the Hamilton Cycle Problem efficiently

Proof Idea: Build an instance of the Traveling Salesman problem out of a Hamilton Cycle problem and show that the answer to the Traveling Salesman problem then provides an answer for the Hamilton Cycle problem

But: we must show that the mapping is efficient (we get to use at most a polynomial amount to time to do the mapping of the problem and the answer).

A Traveling Salesman Algm. could solve Hamilton Cycle

HC Instance:

$$G = (V, E) \quad V = \{v_1, v_2, ..., v_n\}, E = \{e_1, e_2, ..., e_m\}$$

Is there an ordering

$$< v_{k1}, v_{k2}, ..., v_{kn} >$$

such that

$$(v_{ki}, v_{ki+1}) \in E, \text{ for } i = 1...n-1 \quad \text{and } (v_{k1}, v_{kn}) \in E$$

New TS Problem

$$C = \{v_1, v_2, ..., v_n\}$$

$$d(v_i, v_j) = 1 \text{ if } (v_i, v_j) \in E \text{ and } d(v_i, v_j) = 2 \text{ if } (v_i, v_j) \notin E$$

Claim: If the minimum length tour found for the TS problem is <= *n*, then G has a Hamilton Cycle, othewise it does not.

A Traveling Salesman Algm. could solve Hamilton Cycle

Justification:

The only way for a tour to have a length of *n* is if it uses only "distances" that correspond to edges of the graph G. Otherwise the length of the tour will be greater than n. If G has no Hamiltonian cycle there can be no tour of length n.

Importance:

If you found an efficient algorithm to solve the Traveling Salesman problem, you could use it to solve the Hamilton Cycle Problem. So the Traveling Salesman problem is at least as hard as the Hamilton Cycle Problem (which many believe is hard.)