Complexity Theory

Time and space

- To analyze an algorithm means:
  - developing a formula for predicting *how fast* an algorithm is, based on the *size of the input* (time complexity), and/or
  - developing a formula for predicting *how much memory* an algorithm requires, based on the *size of the input* (space complexity)
- Usually **time** is our biggest concern
  - Most algorithms require a fixed amount of space

What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be a parameter that determines the actual time (or space) required
  - It is *usually* obvious what this parameter is
  - Sometimes we need two or more parameters

Characteristic operations

- In computing time complexity, one good approach is to count characteristic operations
  - What a "characteristic operation" is depends on the particular problem
  - If searching, it might be comparing two values
  - If sorting an array, it might be:
    - comparing two values
    - swapping the contents of two array locations
    - both of the above
  - Sometimes we just look at how many times the *innermost loop* is executed

Exact values

- It is sometimes possible, *in assembly language*, to compute *exact* time and space requirements
  - We know exactly how many bytes and how many cycles each machine instruction takes
  - For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required
- However, often the exact sequence of steps cannot be known in advance
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)

Higher-level languages

- In a higher-level language (such as Java), we *do not know* how long each operation takes
  - Which is faster, `x < 10` or `x <= 9` ?
  - We don't know exactly what the compiler does with this
  - The compiler almost certainly optimizes the test anyway (replacing the slower version with the faster one)
- In a higher-level language we *cannot* do an exact analysis
  - Our timing analyses will use *major* oversimplifications
  - Nevertheless, we can get some very useful results

Average, best, and worst cases

- Usually we would like to find the *average* time to perform an algorithm
- However,
  - Sometimes the "average" isn't well defined
    - Example: Sorting an "average" array
      - Time typically depends on how out of order the array is
      - How out of order is the "average" unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the *worst* (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest

Constant time

- *Constant time* means there is some constant k such that this operation always takes k nanoseconds
- A Java statement takes constant time if:
  - It does not include a loop
  - It does not include calling a method whose time is unknown or is not a constant
- If a statement involves a choice (if or switch) among operations, each of which takes constant time, we consider the statement to take constant time
  - This is consistent with *worst-case analysis*

Linear time

- We may not be able to predict to the nanosecond how long a Java program will take, but do know *some* things about timing:

```
for (i = 0, j = 1; i < n; i++) {
    j = j * i;
}
```

  - This loop takes time k*n + c, for some constants k and c

    k : How long it takes to go through the loop once
    (the time for j = j * i, plus loop overhead)

    n : The number of times through the loop
    (we can use this as the "size" of the problem)

    c : The time it takes to initialize the loop

  - The total time k*n + c is *linear in n*

Constant time is (usually) better than linear time

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes 5000 time units
  - Algorithm B takes 100*n time units
- Which is better?
  - Clearly, algorithm B is better if our problem size is small, that is, if $n < 50$
  - Algorithm A is better for larger problems, with $n > 50$
  - So B is better on small problems that are quick anyway
  - But A is better for large problems, *where it matters more*
- We usually care most about very large problems
  - But not always!

The array subset problem

- Suppose you have two sets, represented as unsorted arrays:

```
int[] sub = { 7, 1, 3, 2, 5 };
int[] super = { 8, 4, 7, 1, 2, 3, 9 };
```

  and you want to test whether every element of the first set (sub) also occurs in the second set (super):

```
System.out.println(subset(sub, super));
```

- (The answer in this case should be false, because sub contains the integer 5, and super doesn't)
- We are going to write method subset and compute its time complexity (how fast it is)
- Let's start with a helper function, member, to test whether *one* number is in an array

member

```
static boolean member(int x, int[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        if (x == a[i]) return true;
    }
    return false;
}
```

- If x is *not* in a, the loop executes n times, where n = a.length
  - This is the **worst case**
- If x *is* in a, the loop executes n/2 times *on average*
- Either way, linear time is required: k*n+c

subset

```
static boolean subset(int[] sub, int[] super) {
   int m = sub.length;
   for (int i = 0; i < m; i++)
      if (!member(sub[i], super) return false;
   return true;
}
```

- The loop (and the call to member) will execute:
  - m = sub.length times, if sub is a subset of super
    - This is the **worst case**, and therefore the one we are most interested in
  - Fewer than sub.length times (but we don't know how many)
    - We would need to figure this out in order to compute *average* time complexity
- The worst case is a linear number of times through the loop
- But the loop body doesn't take constant time, since it calls member, which takes linear time

Analysis of array subset algorithm

- We've seen that the loop in subset executes m = sub.length times (in the worst case)
- Also, the loop in subset calls member, which executes in time linear in n = super.length
- Hence, the execution time of the array subset method is m*n, along with assorted constants
  - We go through the loop in subset m times, calling member each time
  - We go through the loop in member n times
  - If m and n are similar, this is roughly **quadratic**, i.e., n^2

What about the constants?

- An added constant, f(n)+c, becomes less and less important as n gets larger
- A constant multiplier, k*f(n), does *not* get less important, but...
  - Improving k gives a *linear* speedup (cutting k in half cuts the time required in half)
  - Improving k is usually accomplished by careful code optimization, not by better algorithms
  - We aren't that concerned with *only* linear speedups!
- Bottom line: ***Forget the constants!***

Simplifying the formulae

- Throwing out the constants is one of *two* things we do in analysis of algorithms
  - By throwing out constants, we simplify $12n^2 + 35$ to just n^2
- Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)
  - We usually discard all but the *highest-order* term
    - We simplify $n^2 + 3n + 5$ to just n^2

Asymptotic function notation:

O, o, Ω, Θ

Asymptotic function notation

- Definition: O (big-O)
  - Let f, g be two functions: $N \to R^{\geq 0}$.
  - We write $f(n) = O(g(n))$, and say "f(n) is big-O of g(n)" if the following holds:
    - There is a positive real c, and a positive integer n_0, such that $f(n) \leq c\, g(n)$ for every $n \geq n_0$.
    - That is, f(n) is bounded from above by a constant times g(n), for all sufficiently large n.
- Often used for complexity upper bounds.
- Example: $n + 2 = O(n)$; can use $c = 2$, $n_0 = 2$.
- Example: $3n^2 + n = O(n^2)$; can use $c = 4$, $n_0 = 1$.
- Example: Any degree-k polynomial with nonnegative coefficients, $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$
  - Thus, $3n^4 + 6n^2 + 17 = O(n^4)$.

More big-O examples

- Definition:
  - Let f, g: $N \to R^{\geq 0}$
  - $f(n) = O(g(n))$ means that there is a positive real c, and a positive integer n_0, such that $f(n) \leq c\, g(n)$ for every $n \geq n_0$.
- Example: $3n^4 = O(n^7)$, though this is not the tightest possible statement.
- Example: $3n^7 \neq O(n^4)$.
- Example: $\log_2(n) = O(\log_e(n))$; $\log_a(n) = O(\log_b(n))$ for any a and b
  - Because logs to different bases differ by a constant factor.
- Example: $2^{3+n} = O(2^n)$, because $2^{3+n} = 8 \times 2^n$
- Example: $3^n \neq O(2^n)$

Big O for subset algorithm

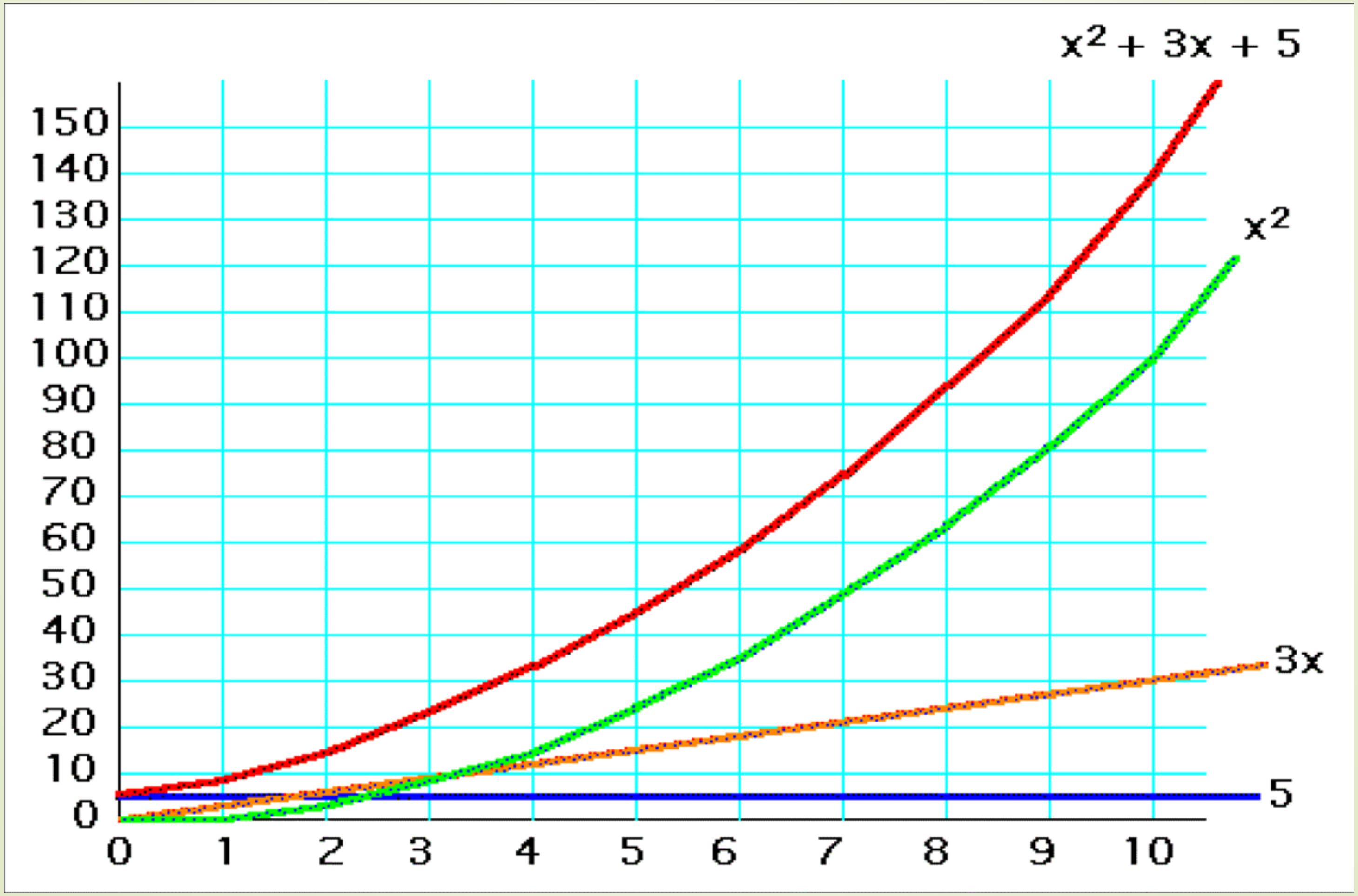
- Recall that, if n is the size of the set, and m is the size of the (possible) subset:
  - We go through the loop in subset m times, calling member each time
  - We go through the loop in member n times
- Hence, the actual running time should be k*(m*n) + c, for some constants k and c
- We say that subset takes O(m*n) time

Can we justify Big O notation?

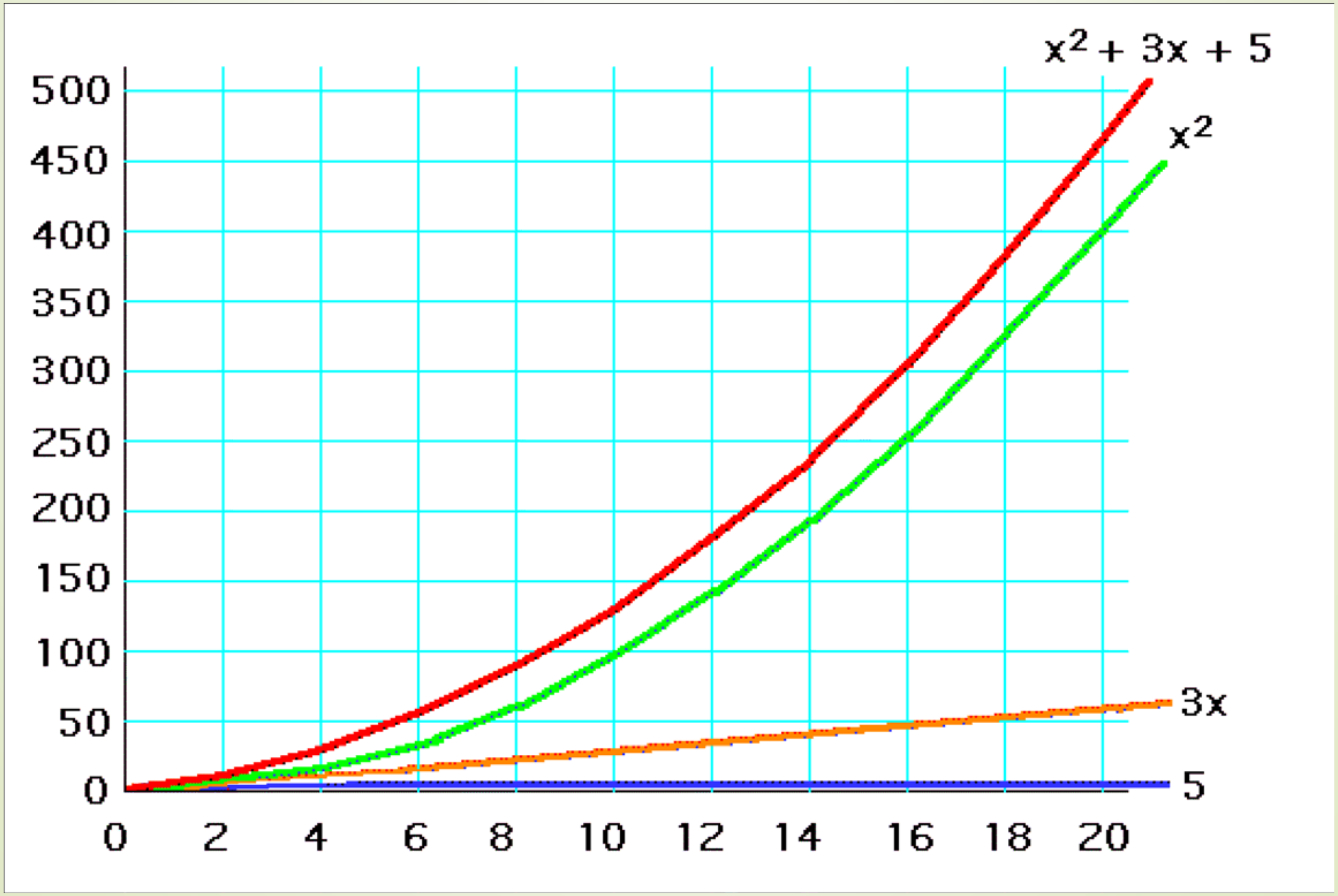
- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - **For sufficiently large problem sizes, the highest-order term swamps all the rest!**
- Consider $R = x^2 + 3x + 5$ as x varies:

$x = 0$	$x^2 = 0$	$3x = 0$	$5 = 5$	$R = 5$
$x = 10$	$x^2 = 100$	$3x = 30$	$5 = 5$	$R = 135$
$x = 100$	$x^2 = 10000$	$3x = 300$	$5 = 5$	$R = 10{,}305$
$x = 1000$	$x^2 = 1000000$	$3x = 3000$	$5 = 5$	$R = 1{,}003{,}005$
$x = 10{,}000$	$x^2 = 10^8$	$3x = 3*10^4$	$5 = 5$	$R = 100{,}030{,}005$
$x = 100{,}000$	$x^2 = 10^{10}$	$3x = 3*10^5$	$5 = 5$	$R = 10{,}000{,}300{,}005$

$y = x^2 + 3x + 5$, for $x = 1..10$



$y = x^2 + 3x + 5$, for $x=1..20$



Common time complexities

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Other notation

- Definition: Ω (big-Omega)
  - Let f, g be two functions: $N \rightarrow R^{\geq 0}$
  - We write $f(n) = \Omega(g(n))$, and say "f(n) is big-Omega of g(n)" if the following holds:
    - There is a positive real c, and a positive integer n_0, such that $f(n) \geq c\, g(n)$ for every $n \geq n_0$.
    - That is, f(n) is bounded from below by a positive constant times g(n), for all sufficiently large n.
- Used for complexity lower bounds.
- Example: $3n^2 + 4n \log(n) = \Omega(n^2)$
- Example: $3n^7 = \Omega(n^4)$.
- Example: $\log_e(n) = \Omega(\log_2(n))$
- Example: $3^n = \Omega(2^n)$

Other notation

- Definition: Θ (Theta)
  - Let f, g be two functions: $N \rightarrow R^{\geq 0}$
  - We write $f(n) = \Theta(g(n))$, and say "f(n) is Theta of g(n)" if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Equivalently, there exist positive reals c_1, c_2, and positive integer n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for every $n \geq n_0$.
- Example: $3n^2 + 4n \log(n) = \Theta(n^2)$
- Example: $3n^4 = \Theta(n^4)$.
- Example: $3n^7 \neq \Theta(n^4)$.
- Example: $\log_e(n) = \Theta(\log_2(n))$
- Example: $3^n \neq \Theta(2^n)$

Other notation

- Definition: o (Little-o)
  - Let f, g be two functions: $N \rightarrow R^{\geq 0}$
  - We write f(n) = o(g(n)), and say "f(n) is little-o of g(n)" if for every positive real c, there is some positive integer n_0, such that f(n) < c g(n) for every $n \geq n_0$.
  - In other words, no matter what constant c we choose, for sufficiently large n, f(n) is less than g(n).
  - In other words, f(n) grows at a slower rate than any constant times g(n).
  - In other words, $\lim_{n \to \infty} f(n)/g(n) = 0$.
- Example: $3n^4 = o(n^7)$
- Example: $\sqrt{n} = o(n)$
- Example: $n \log n = o(n^2)$
- Example: $2^n = o(3^n)$

Exercise

- Give the time complexity of the following algorithm using big-O notation:

There is a 2D (n,n) array **A** and the elements in the array can be traversed using the following algorithm:

```
loop i from 0 to n
    loop j from 0 to n
        print A[i,j]
    end loop
End loop
```

Exercise

- Give the time complexity of the following algorithm using big-Omega notation:

There is a 2D (**m**,n) array **A** and the elements in the array can be traversed using the following algorithm:

```
loop i from 0 to m
    loop j from 0 to n
        print A[i,j]
    end loop
End loop
```

Exercise

1. Give the time complexity of the algorithms with following number of execution steps using big-O and big-Omega notations:
   1. $3n^2 + n$
   2. $3n^3 + 15660n$
   3. $90n^5 + \log(n)$
   4. $4\log_2(n) + 2n$
   5. $n\log_2(n) + 2n$

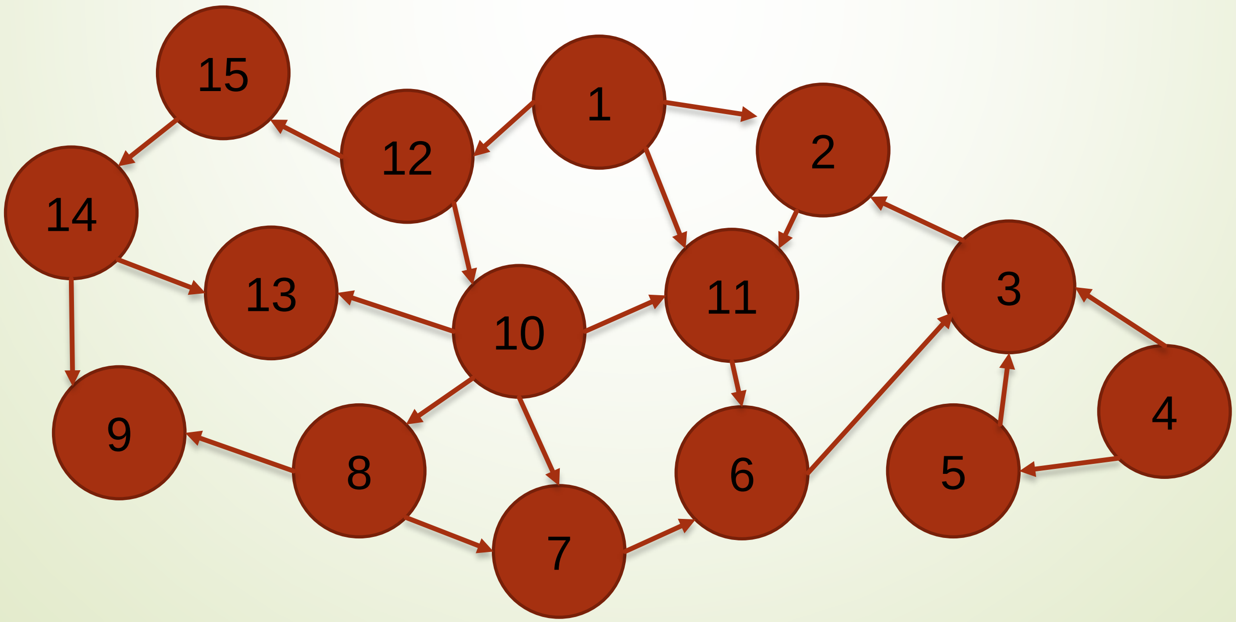
P, Polynomial Time

- Polynomial Time Complexity Class:
  - If the complexity of an algorithm has a polynomial relationship with its input size, it is identified as a polynomial time algorithm.
  - Ex: $2n^k$ where k is a constant and n is the size of input

- Non-Polynomial Time Complexity Class:
  - If the complexity of an algorithm cannot be given as a polynomial relationship with its input size:
  - Ex: $2n^{kn}$ where k is a constant and n is the size of input

Exercise

- Is it possible classify the following problem to Polynomial Time Complexity Class?
  - Input: a directed graph G and two nodes s and t.
  - Is there a path from s to t?



Exercise

- Is it possible classify the following **Hamiltonian Path** problem to Polynomial Time Complexity Class?
  - Input: a directed graph G and two nodes s and t.
  - Is there a path from s to t which goes through the every node only once ?
  - HAMPATH = {<G,s,t> | G is a directed graph, there is a Hamiltonian path between s to t}