EPITECH.
PARIS GRADUATE SCHOOL OF DIGITAL INNOVATION

Introduction to Algorithms

Part II. Problems.

B9 - Introduction to Algorithms

M-ALG-100

Overview of the module

Day 1 Concept of algorithm, Cryptography, recursion, Knapsack, Shortest Path

Day 2 Complexity, Graph problems, Theory

Organisation

- The exercices will be in python
- Clone the following repository : `https://github.com/nlehir/ALGO1.git`
- For day 2 : Please install **matplotlib**, **numpy**
- Optional but useful : **ipdb** (python debug) or your favorite debugger

Day 2

- The problem of complexity
  - Measuring complexities
  - Profiling
  - Computing complexities
- Famous graph problems
  - Random graphs
  - Dominating set
  - Coloring
  - Independent Set
- Theoretical problems

Complexity

- Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question, as a function of the size of the input ? Is it possible to **compute** an answer with a computer ?

Complexity

- Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question ? Is it possible to **compute** an answer with a computer ?
- The answer is that **it depends on the problem**. For some problems, it is very probable that there exists **no exact fast** solution (for instance the NP-hard problems)

Measuring complexities

- Let us start by measuring the complexity of some simple programs. How ?

Measuring complexities

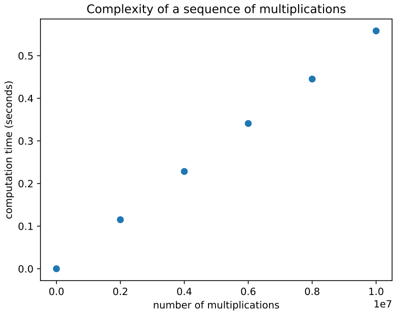
- Let us start by measuring the complexity of some simple programs.
- We can first measure the computing time.

Exercise 1

- **cd complexity** and use **linear_complexity** to verify that the complexity of a sequence of multiplications is proportionnal to its length.

Exercise 1

- **cd complexity** and use **linear_complexity** to verify that the complexity of a sequence of multiplications is proportionnal to its length.
- It should look like this :

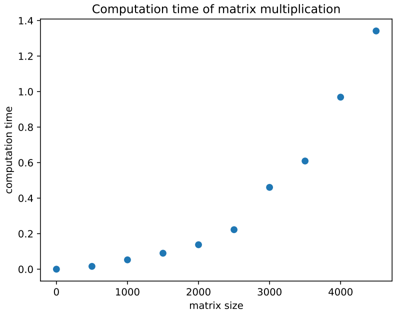


Exercise 2

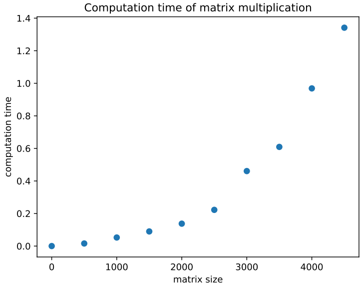
- What happens with matrix multiplication ? modify **matrix_multiplication** to estimate the computing time as a function of the size of the matrix.

Exercise 2

- What happens with matrix multiplication ? modify **matrix_multiplication** to estimate the computing time as a function of the size of the matrix.
- It should look like this :

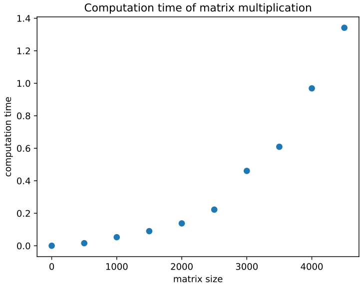


Matrix multiplication

- 

- Let's give a rough approximation of the number of operations as a function of the size n of the matrix.

Matrix multiplication

- 

- Let's give a rough approximation of the number of operations as a function of the size n of the matrix.
- It should then bo of order $\mathcal{O}(n^3)$. However, some **sub-cubic** algorithms exists : faster than n^3

Measuring the time ?

- Why is **time** maybe not the best tool to evaluate the complexity of an algorithm ?

Measuring the time ?

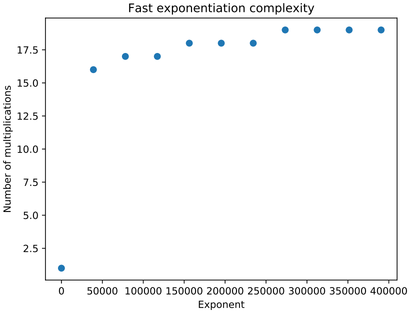
- Why is **time** maybe not the best tool to evaluate the complexity of an algorithm ?
- It depends on the system
- Instead we could count the number of elementary operations

Exercise 3

- Use a variable in **exponentiation_complexity** to compute the number of operations in fast exponentiation and normal exponentiation.

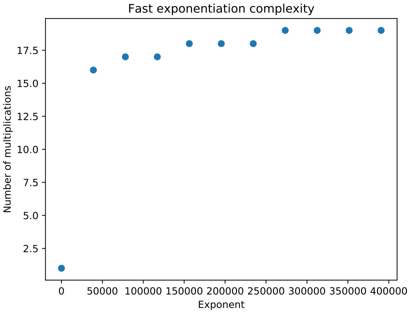
Exercise 3

- Use a variable in **exponentiation_complexity** to compute the number of operations in fast exponentiation and normal exponentiation.
- It should look like :



Exercise 3 : fast exponentiation

- We note the **logarithmic complexity** $\mathcal{O}(\log n)$



Asymptotic behavior

- What matters is the **asymptotic** behavior, when $n \to \infty$

Asymptotic behavior

- What matters is the **asymptotic** behavior, when $n \to \infty$
- This tells if the algorithm **scales** (still works when the instance of the problem is larger)

Asymtptic behavior : $\mathcal{O}$ notation

- stricty speaking, we say that $f = \}$ if the ratio $\frac{|f|}{|g|}$ is **bounded**
- $||$ means "absolute value"
- intruitively, this means that f is not bigger than g

Asymptotic behavior : examples

- $$n^2 + n = \mathcal{O}(n^2) \tag{1}$$
- $$5 \times n^4 + 2178 \times n^3 + \log 3n = \mathcal{O}(?) \tag{2}$$

Examples of algorithms

- Fast exponentiation
- Naive exponentiation
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- Enumeration of permutations $\mathcal{O}(n!)$

Orders of magnitude

Orders of magnitude

Taille	$n \log n$	n^3	2^n
$n = 20$	60	8000	1048576
$n = 50$	196	125000	1125899907000000
$n = 100$	461	1000000	1267650600000000000000000000000

$\Longrightarrow$ Hence the idea of a border between polynomial and exponential algorithms.

Profiling

- Another useful tool to monitor the execution of a program is **profiling**
- From the python docs : "A profile is a set of statistics that describes how often and for how long various parts of the program executed"
- `https://docs.python.org/3.6/library/profile.html`

Profiling

- **cd profiling** and profile some programs that we used before

Profiling

- **cd profiling** and profile some programs that we used before
- Note that when profiling **profiling_demo.py**, the elementary multiplications are not taken into account in the profiling output.

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- For a sequence of blocks :
- For a loop :

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- For a sequence of blocks : complexities sum up
- For a loop :

Computing complexities

We now want to compute some complexities with paper and pen. Let us focus on some intuitive rules :

- For a sequence of blocks : complexities sum up
- For a loop : complexities of all iterations sum up
- If a loop consists in similar iterations, its complexity is the product of the compexity of one iteration by the size of the loop.

Exercise 4

Compute the running time of the following algorithm.

```
p = 100
for i in range(p):
    for j in range(i):
        l = [i+j+k for k in range(p)]
```

Some mathematical concepts

- Mathematical induction
- Applications : prime factors decomposition, $\sum_{k=1}^{n} k$

Exercise 5

Compute the running time of the following algorithm.

```
from math import log
def dec2bin(n):
    m = int(log(n)/log(2))
    liste = [0 for i in range(m+1)]
    for i in range(m+1):
        liste[i] = n%(2**(i+1))//(2**i)
    return liste
def enum_bin(p):
    for n in range(1,p):
        print(dec2bin(n))
```

Horner Algorithm

- Let us consider the case of evaluating polynoms
- A polynom is a function of the form
  $f : x \to a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
- How many multiplications are involved with the naive method ?

Horner Algorithm

- Let us consider the case of evaluating polynoms
- A polynom is a function of the form
  $f : x \to a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$
- How many multiplications are involved with the naive method ?
- We look fot an algorithm that is faster than the naive solution.

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{3}$$

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{4}$$

- How many multiplications are now involved ?

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{5}$$

- How many multiplications are now involved ? $\mathcal{O}(n)$.
- So we went from quadratic to linear.

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{6}$$

- We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \ldots, a_0]$

Horner Exercise

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{7}$$

- We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \ldots, a_0]$
- Please modify **complexity/horner.py** so that it performs the horner algorithm.
- In order to test that our method is correct, we will test it against the method **polyval** from **numpy**

Horner

- What do you see if you write **help(numpy.polyval)** inside python ?

Horner

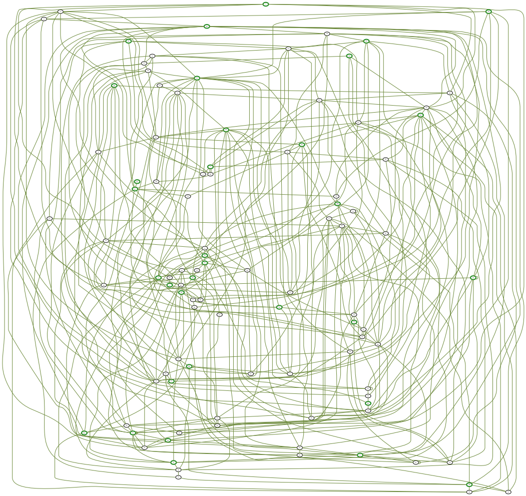
- What do you see if you write **help(numpy.polyval)** inside python ?

```
Horner's scheme [1]_ is used to evaluate the polynomial. Even so,
for polynomials of high degree the values may be inaccurate due to
rounding errors. Use carefully.

References
----------
.. [1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng.
   trans. Ed.), *Handbook of Mathematics*, New York, Van Nostrand
   Reinhold Co., 1985, pg. 720.
```

Figure: Horner is actually the method used by numpy

Graph problems



Graph problems

We will look at famous graph problems, typically of the form :

- "what is the largest subset of nodes of the graph, verifying some property ?"
- "what is the largest subset of edges of the graph, such that some property is verified ?"

Graphviz

We will use graphviz to visualize graphs.

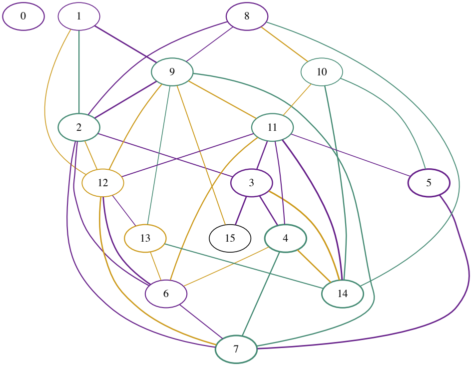


Figure: Undirected random graph generated with python

Warm up question

Given an **unoriented** graph with n nodes, how many edges can we build ?

Notation of a graph : $G(V, E)$

- V : set of n vertices
- E : set of edges

Warm up question

Given an **unoriented** graph with n nodes, how many edges can we build ?

Notation of a graph : $G(V, E)$

- V : set of n vertices
- E : set of edges, maximum size : $\frac{n(n-1)}{2} = \binom{n}{2} = \frac{n!}{2!(n-2)!}$

Exercise 6

cd graphs and use **random_graph** to generate a random graph with 25 nodes and 100 edges

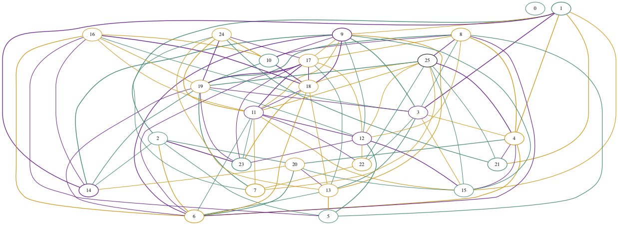
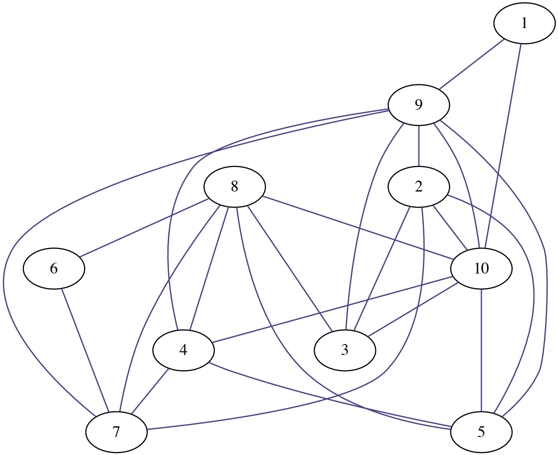


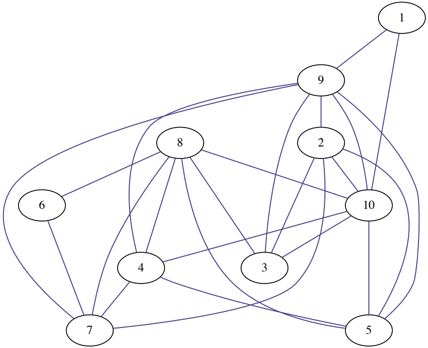
Figure: Random graph with 25 nodes, 100 edges

The dominating set problem



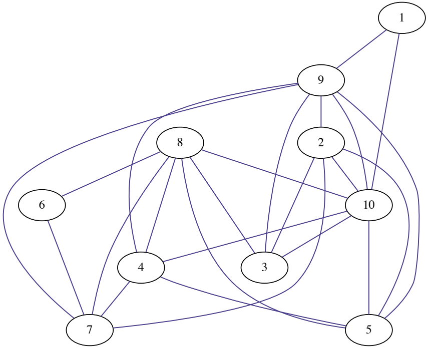
The dominating set problem

For instance : we want to use the smallest possible number of emitters to cover a network.



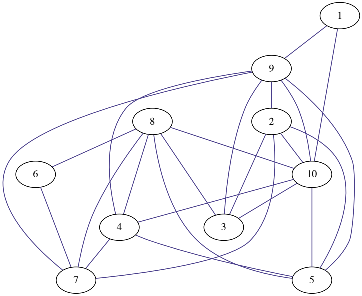
The dominating set problem

Mathematically speaking : if $G(V, E)$ is the graph. We look for a **subset of nodes** D such that **all nodes in the graph** are the neighbor of **at least one node** in D.



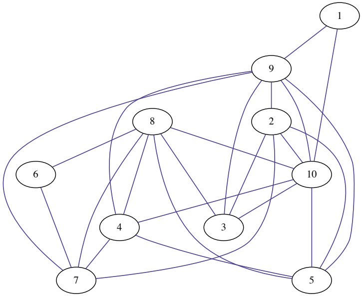
The dominating set problem

Mathematically speaking : if $G(V,E)$ is the graph. We look for a **subset of nodes** D such that **all nodes in the graph** are the neighbor of **at least one node** in D. And we want to pick the **smallest D** that works.



The dominating set problem

What is the most trivial dominant subset ?



Dominating set : example 1

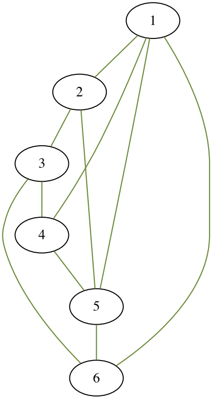


Figure: Some simple graph

Dominating set : example 1

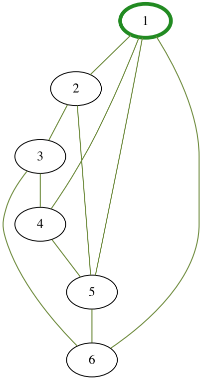


Figure: Is this a dominating subset ?

Dominating set : example 1

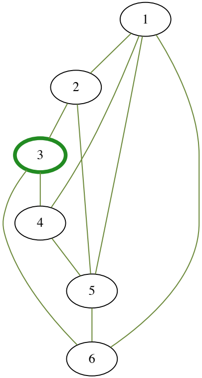


Figure: Is this a dominating subset ?

Dominating set : example 1

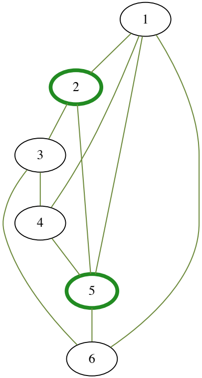


Figure: Is this a dominating subset ?

Dominating set : example 1

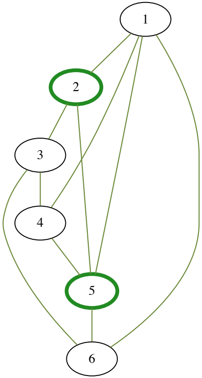
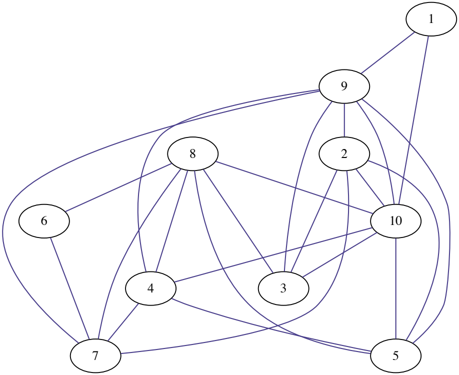


Figure: Is this a dominating subset ? Yes. Is it minimal ?

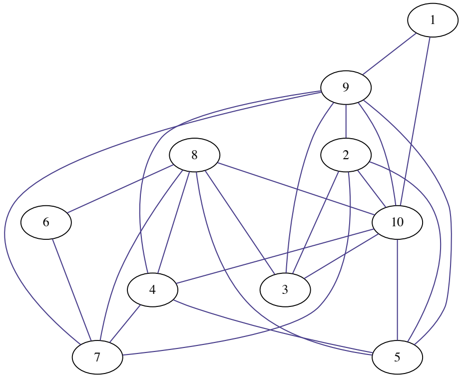
Dominating set : example 2

Please find a dominating set in this graph.



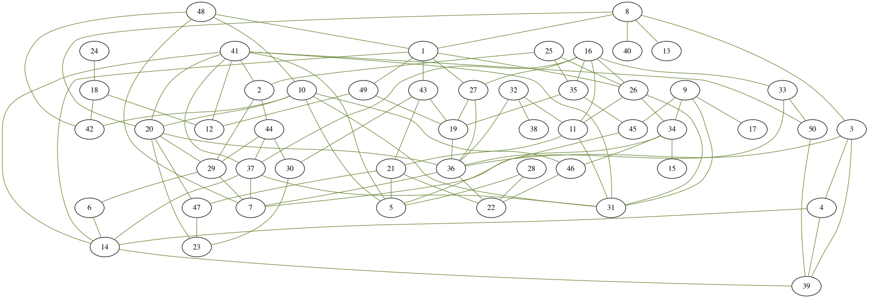
Dominating set : example 2

Please find a **minimal** dominating set in this graph.



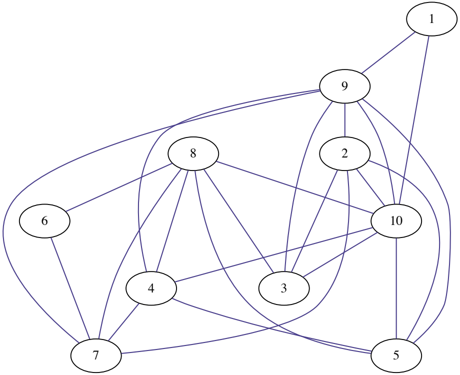
Dominating set : example 2

Please find a **minimal** dominating set in this graph.



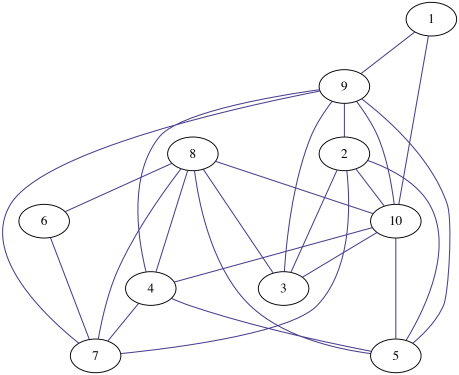
Dominating set : exhaustive search

What would be the **exhaustive search** in the case of the Dominating set problem ?



Dominating set : exhaustive search

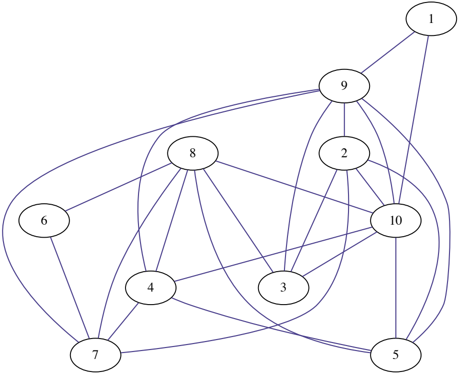
How many possibilities do have to try as a function of n ?



Dominating set : exhaustive search

How many possibilities do have to try as a function of n ?

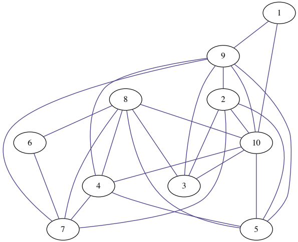
The number of subsets in $[1 : n]$ is :



Dominating set : exhaustive search

How many possibilities do have to try as a function of n ?
The number of subsets in $[1 : n]$ is :

$$2^n = \sum_{k=0}^{n} \binom{n}{k} \tag{8}$$



Heuristic

Ok so the exhaustive search is no possible. So what method should we use ?

Heuristic

Ok so the exhaustive search is no possible. So what method should we use ?
Let's build a **greedy algorithm**.

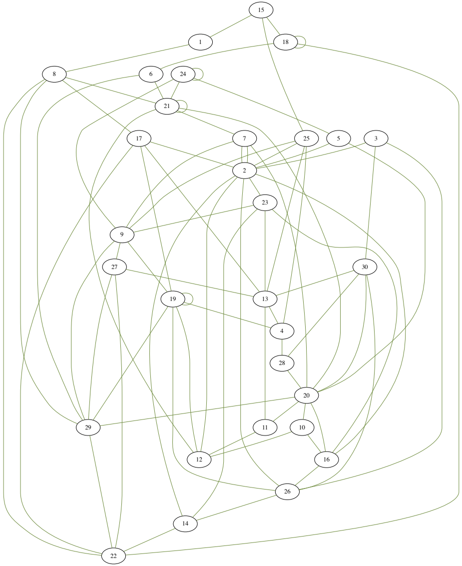
Greedy algorithm

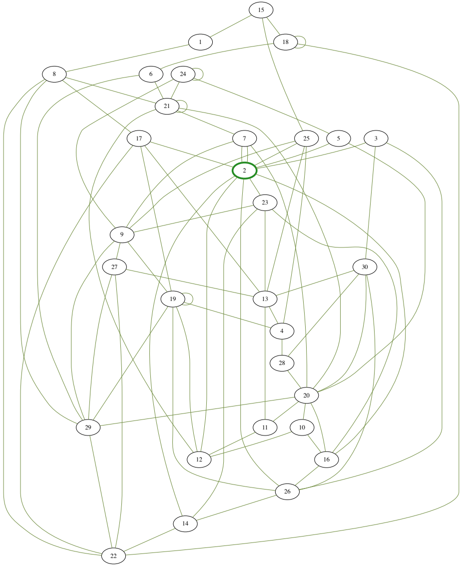
In a graph (unweighted), the **degree of a node** is its number of neighbors.

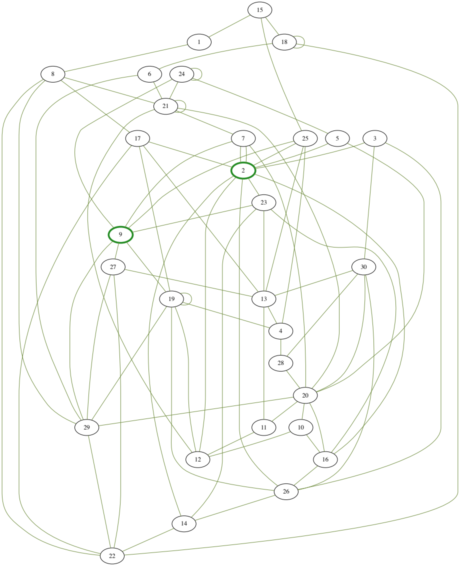
Greedy algorithm

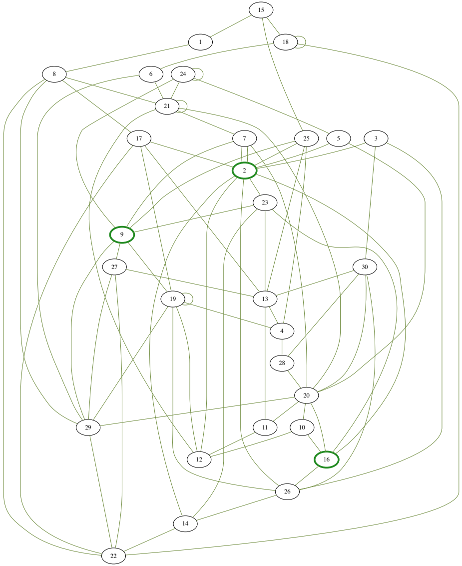
Modify **dominant_greedy_1** to apply the greedy algorithm :

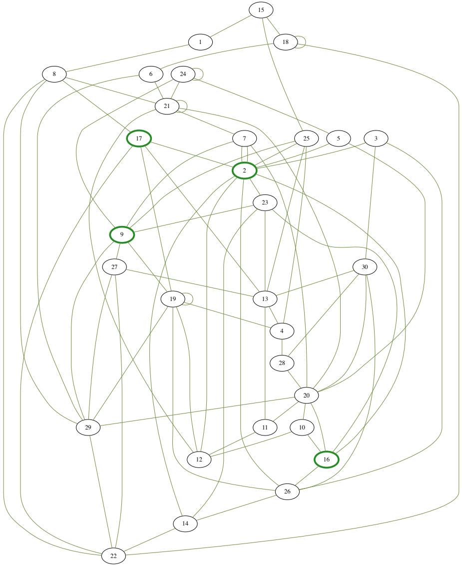
- sort nodes by degree
- progressively add the to the set until it's dominant

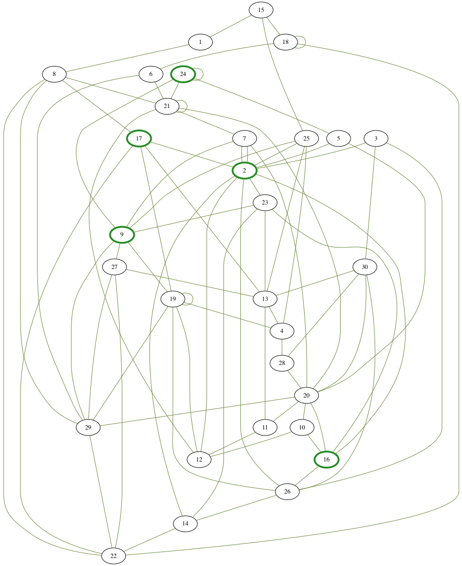

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

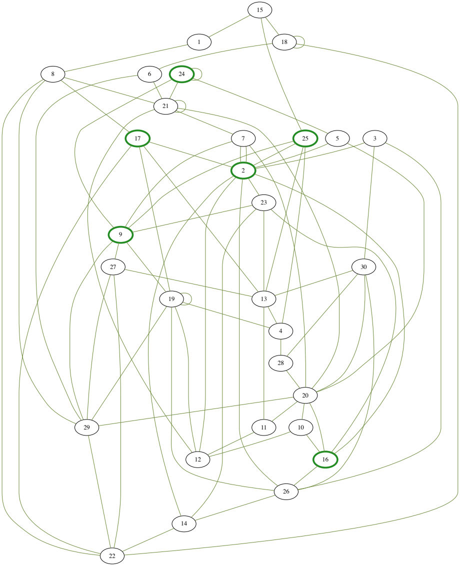

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

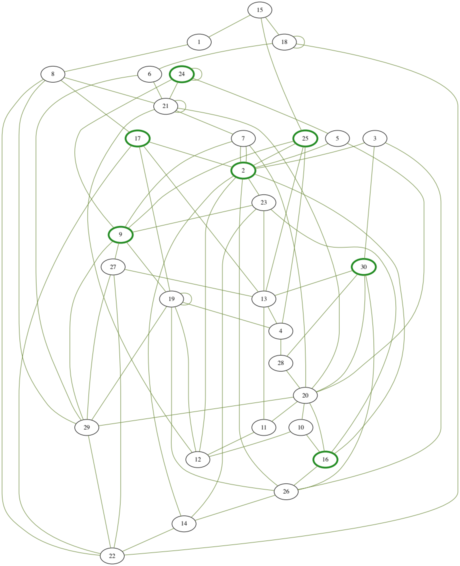

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

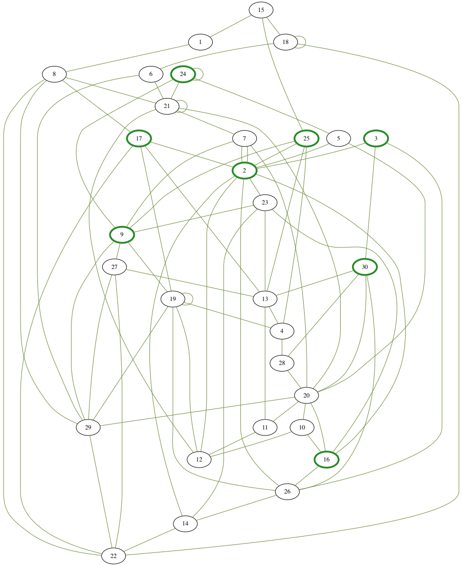

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

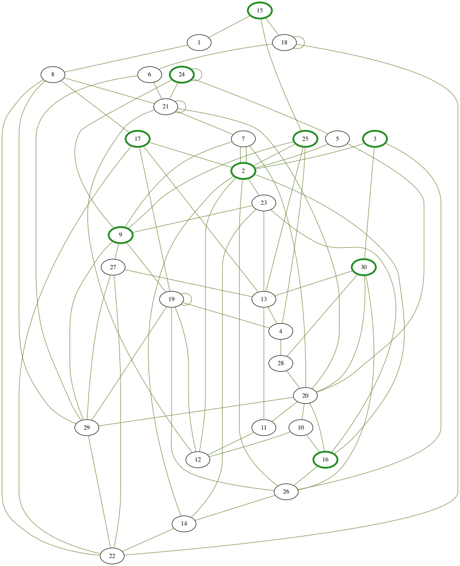

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

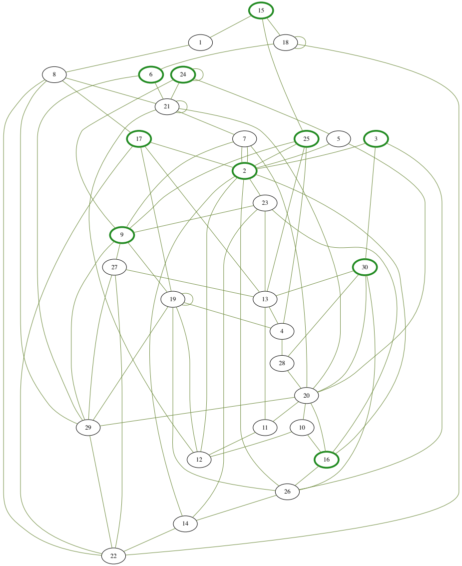

15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22


15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22


15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22


15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22


15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22


15
1
18
8
6
24
21
17
7
25
5
3
2
23
9
27
30
19
13
4
28
20
29
11
10
12
16
26
14
22

Non optimal greedy algorithm

Let us find an example where the greedy algorithm is clearly not optimal.

Non optimal greedy algorithm

Let us find an example where the greedy algorithm is clearly not optimal.

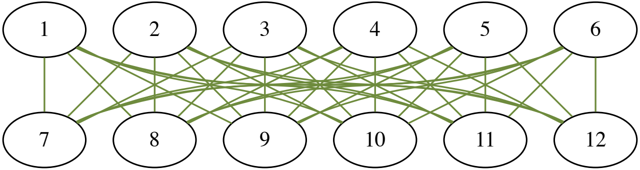


Figure: Complete bipartie graph

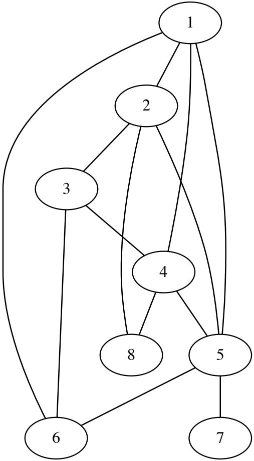
The coloring problem

We want to find the smallest number **fully disconnected subgraph** in a graph.

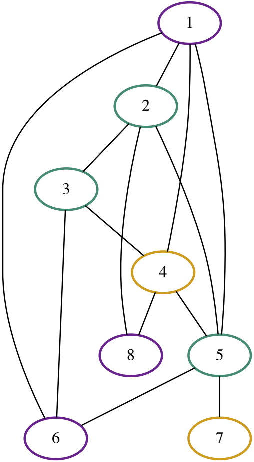
The coloring problem

We want to find the smallest number **fully disconnected subgraph** in a graph.
Each subgraph will be associated with a color.

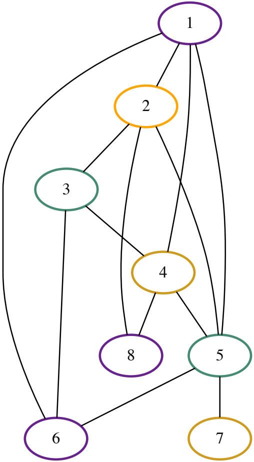
Coloring



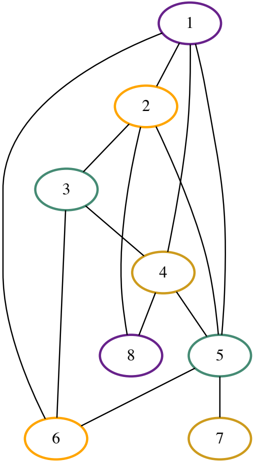
Is this a coloring ?



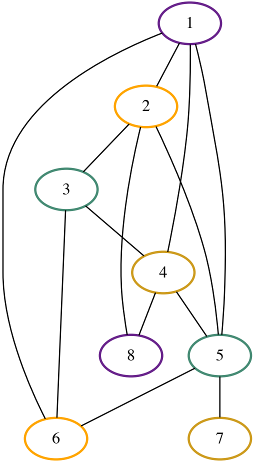
Is this a coloring ?



Is this a coloring ? yes



Could we have used only 3 colors ?



Coloring

- What would be a trivial coloring ?

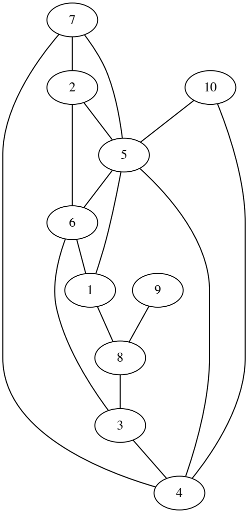
Coloring

- What would be a trivial coloring ? assign a color to each node (very bad solution)
- Could you think of a heuristic ?

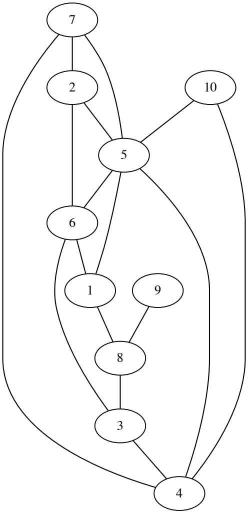
Independent Set

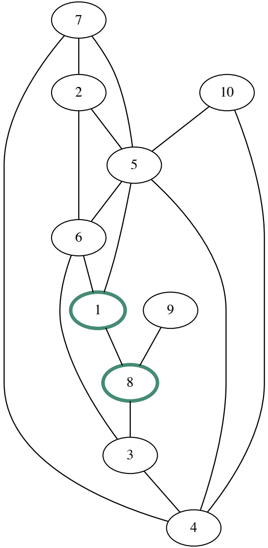
We want to find **the largest disconnected subgraph**. For instance : building the biggest possible team of people that can work together (a edge between them meaning that they can't.)

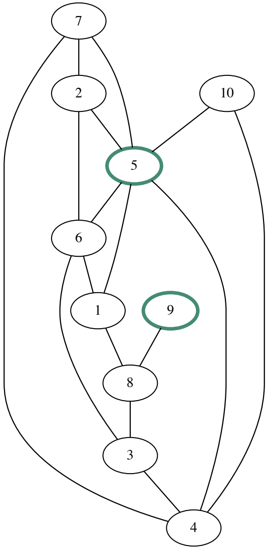
Independent set

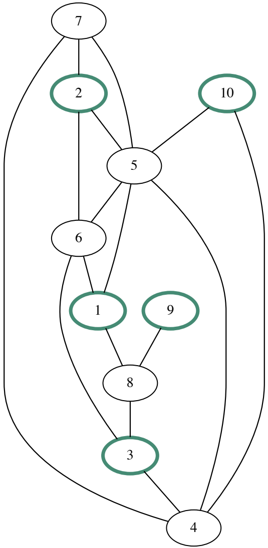


Independent set : what is a trivial independent set ?

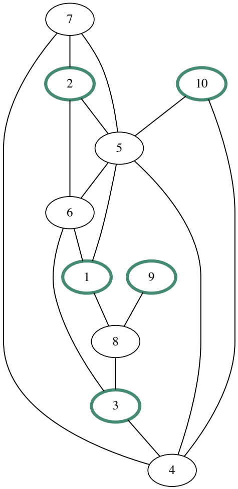



7
2
10
5
6
1
9
8
3
4


7
2
10
5
6
1
9
8
3
4


7
2
10
5
6
1
9
8
3
4

Maximal vs maximum independent set



Complexity

- Running time of an algorithm is its running time on the worst possible input (instance I) it can get (for a given size)
- Complexity of a problem is the running time of the best possible algorithm for that problem.

$$T(P) = \min_{A} \max_{I} T(P, A, I) \tag{9}$$

Equivalence between problems

- Some problems have the same difficulty because they are equivalent
- Some are strictly more complex than others
- Hard problems : Maximum independent set, minimum coloring, smallest dominating set, TSP, etc.
- Easier : Shortest Path

Equivalence between problems

- Dominating set and maximum clique

Problems that are not equivalent

- Eulerian paths and hamiltonian paths

Classes of complexity

- Problems have been gathered under **classes of complexity**
- **P** : we can obtain a solution with polynomial complexity
- **NP** : we can verify a solution in polynomial time (doesn't mean we can find a solution)
- **NP hard** : if it is in P, all NP problems are in P.
- **NP complete** : NP and NP hard

P=NP ?