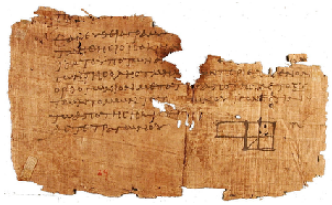
HE VD
IG



Introduction

ASD - Algorithmes et Structures de Données - Printemps 2021

ASD - Algorithmes et Structures de Données - Printemps 2021

Un bon code ?

- Mais au fond, quelle différence y a-t-il entre le bon et le mauvais code(ur) ?
- Un bon logiciel est fiable, robuste, extensible, réutilisable, compatible et efficace

Qualités d'un logiciel

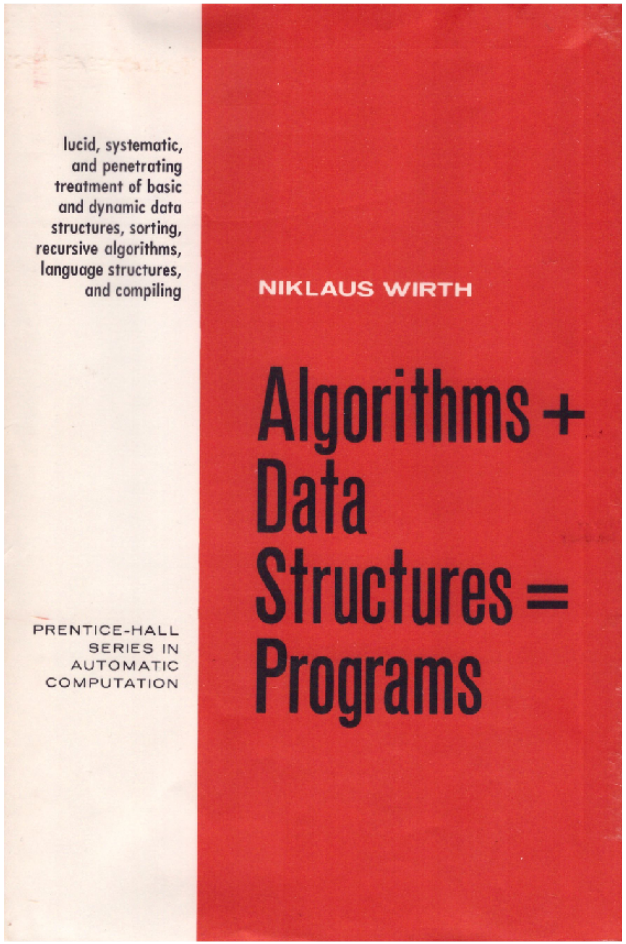
- Fiable : s'exécute sans erreur pour tous les jeux de données dans le domaine concerné
- Robuste : résiste et réagit aux erreurs, prévisibles ou non
- Extensible : peut s'adapter aux changements de spécifications
- Réutilisable : composants peuvent être réutilisés dans d'autres applications
- Compatible : peut être utilisé avec d'autres composants
- Efficace : s'exécute rapidement sans mobiliser trop de ressources du système.

Loi de Wirth


Les logiciels
ralentissent plus vite que
le matériel n'accélère

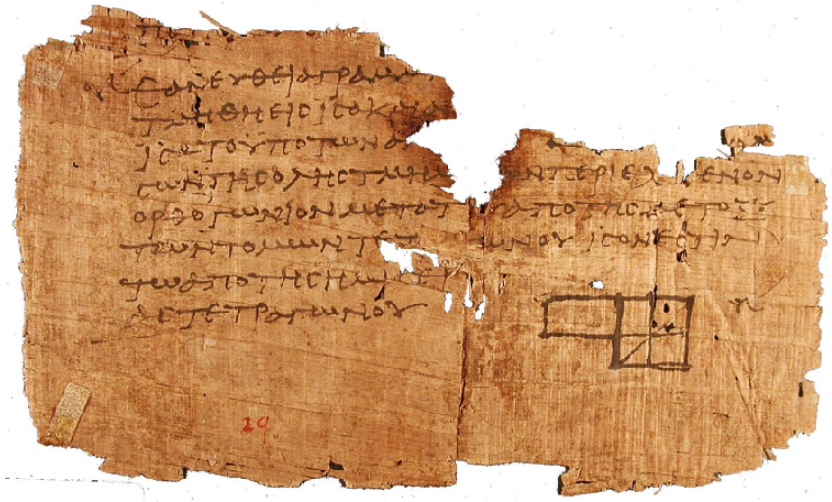
Nicklaus Wirth

- Né en 1934 à Winterthour.
- Ingénieur ETHZ, Master Univ. Laval, Doctorat UC Berkeley, Prof. Assistant Stanford, Professeur ETHZ
- On lui doit les langages Euler, Algol W, Pascal, Modula et Oberon.

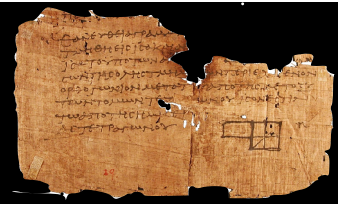


HEIG VD

1.1 Algorithmes



HEIG VD



Mais au fond, qu’est-ce qu’un algorithme ?

Algorithmes ?

- Mais au fond, qu’est-ce qu’un algorithme ?

- Qu’est-ce que l’algorithmique ?

HE VD
IG

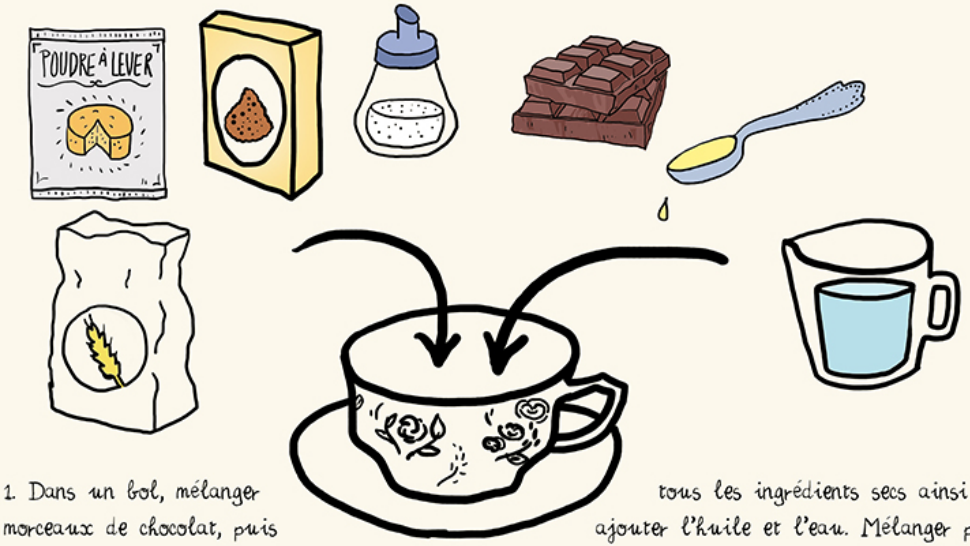
Une recette de cuisine

Teacup chocolate cake

Proportions : 1 tasse de thé
Temps de préparation : 5min

Difficulté :
Temps de cuisson : 1min20

Ingrédient : 30g farine (1,5 cuil. à soupe), 7g cacao en poudre (1 cuil. à soupe), 20g sucre (1cuil. à soupe rase), 20g morceaux de chocolat noir haché en morceaux (1 cuil. à soupe), 1 pincée de poudre à lever, 1,5 cuil. à soupe huile (20ml), 40ml d'eau (2,5 cuil. à soupe), 1 cuil. à café de beurre de cacahuète.



1. Dans un bol, mélanger tous les ingrédients secs ainsi que les morceaux de chocolat, puis ajouter l'huile et l'eau. Mélanger pour obtenir une texture homogène.
2. Verser dans une tasse de thé et faire tomber au centre une cuillère à café de beurre de cacahuète. Enfoncer un peu avec le doigt si nécessaire pour qu'un peu de pâte chocolatée le recouvre.
3. Placer dans le micro-onde sur un bord de l'assiette tournante (et non au centre car la cuisson ne serait pas homogène) et cuire 1min20 secondes à 650 watts. Attention à la sur-cuisson, pour éviter la texture étouffe-chrétien, surveiller à mi-cuisson : au centre, il doit être à peine cuit.

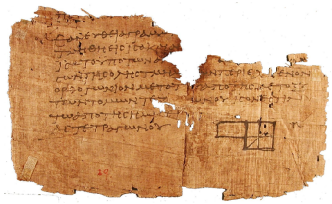
Recette & illustrations de Mathilda Motte

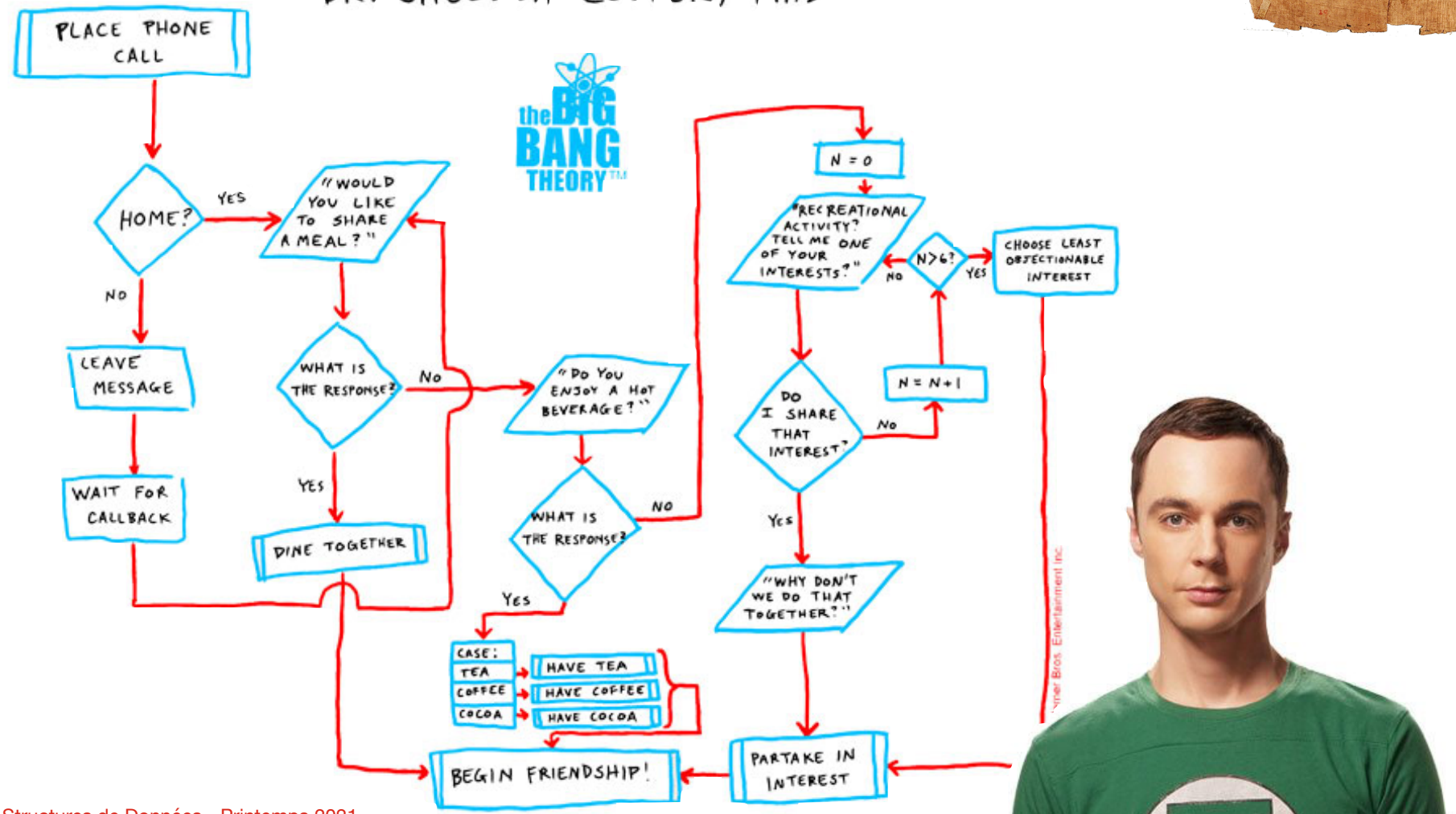

HOW I MET YOUR MOTHER
NEXT


HOW I MET YOUR MOTHER
NEXT

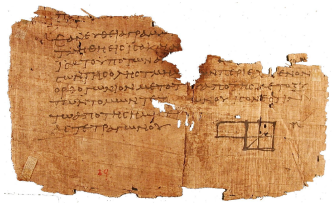
THE FRIENDSHIP ALGORITHM

DR. SHELDON COOPER, Ph.D

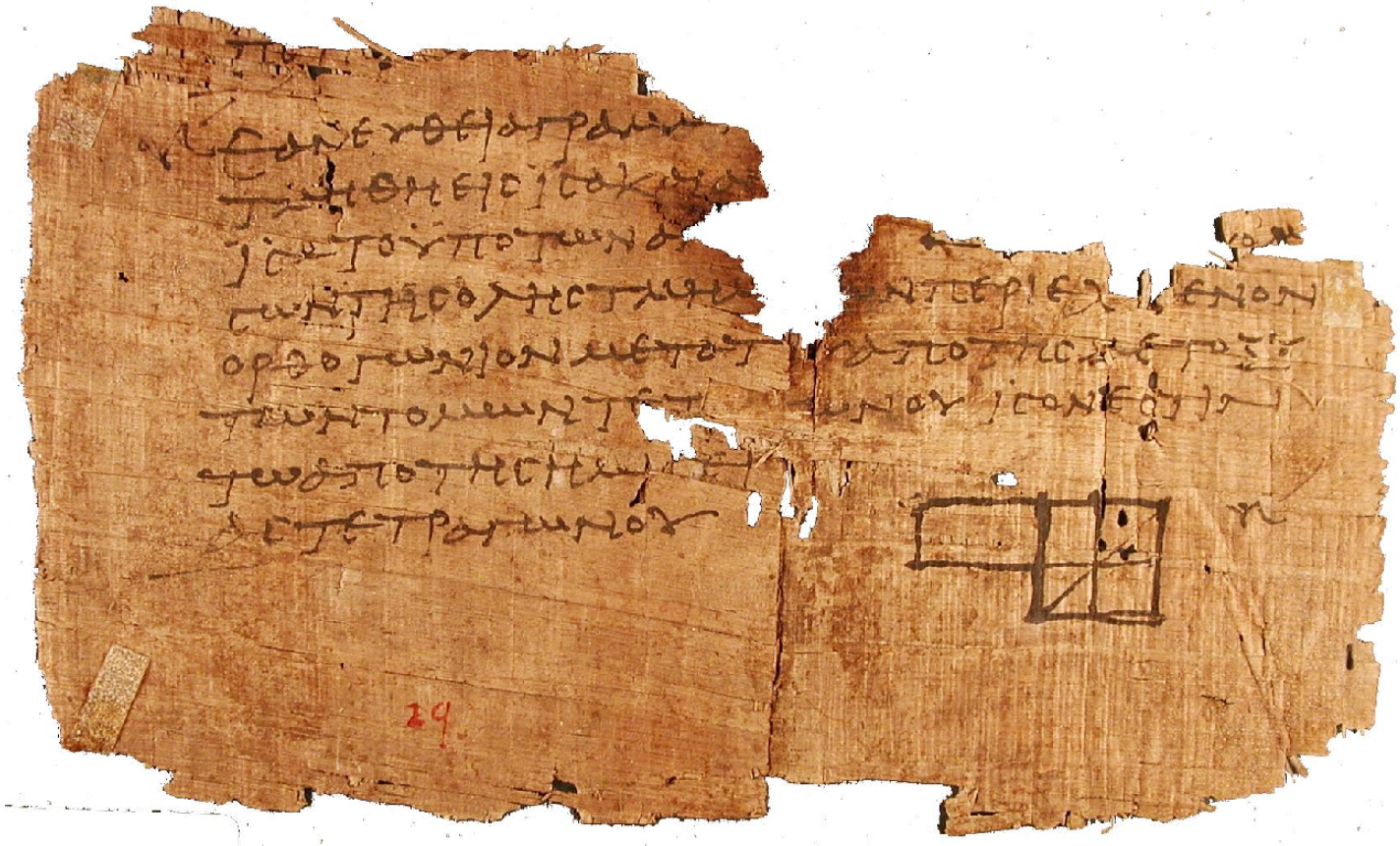




Algorithme d'Euclide



Un des plus anciens fragment des Éléments d'Euclide qui nous soit parvenu, découvert à Oxyrhynque et qui daterait d'entre 75 et 125 AC



Algorithme d'Euclide

- Pour calculer le Plus Grand Commun Diviseur (PGCD) des entiers positifs a et b,
  - Calculer r, le reste de la division entière de a par b
  - Tant que r est non nul, a prend la valeur de b, b celle de r, et on recalcule r comme précédemment
  - Le PGCD est b.

Algorithme (déf.)

- un ensemble fini d’étapes formées d’un
- ensemble fini d’opérations dont chacune est
- définie de façon non ambiguë et peut être
- réalisée par une machine

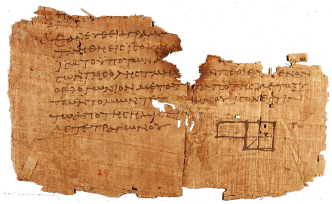
(voire même dans les cas extrêmes par Sheldon Cooper)

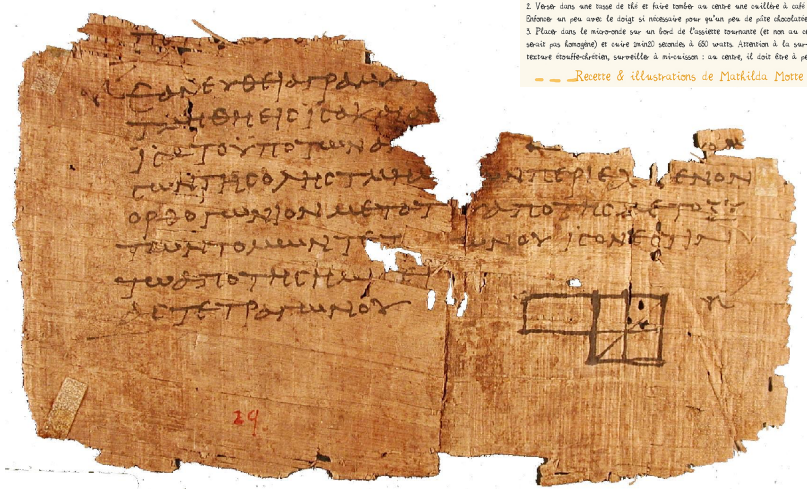
HE VD
IG

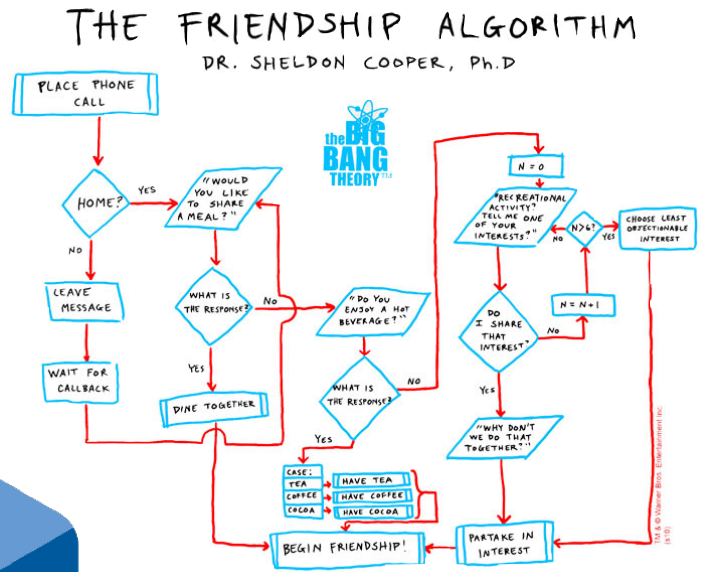
Algorithmique
(Science des algorithmes)

- Connaitre les algorithmes classiques
- Concevoir de nouveau algorithmes
- Estimer la complexité d'un algorithme
- Choisir entre plusieurs algorithmes
- Appliquer les algorithmes à des cas précis
- Mettre en oeuvre en C++, Java, …

Avec quel langage décrire nos algorithmes?







Pseudo-code

```
si vous êtes cuisinier
    Utilisez le style des recettes
sinon si vous parlez grec ancien
    Utilisez le style d'Euclide
sinon si vous êtes Sheldon Cooper
    Que faites-vous à mon cours ?
    signaler une erreur
sinon
    tant que vous êtes en ASD1 boucler
        Utilisez du pseudo code
    fin boucler
fin si
```

Pseudo-code

```
si vous êtes cuisinier
    Utilisez le style des recettes
sinon si vous parlez grec ancien
    Utilisez le style d'Euclide
sinon si vous êtes Sheldon Cooper
    Que faites-vous à mon cours ?
    signaler une erreur
sinon
    tant que vous êtes en ASD1 boucler
        Utilisez du pseudo code
    fin boucler
fin si
```

```
si …
  …
sinon si …
  …
sinon
  …
fin si
```

```
pour … boucler
  …
fin boucler
```

```
tant que … boucler
   …
fin boucler
```

```
terminer / retourner
```

```
sortir boucle
```

```
signaler une erreur
```

Somme $\sum_{i=1}^{N} i$

- Ecrivez un algorithme qui permette à cet enfant de calculer `s`, la somme des `n` premiers entiers positifs

```
s ← 0
pour i allant de 1 à n boucler
    s ← s + i
fin boucler
retourner s
```

HEIG VD

Carl Friedrich Gauss


Carl Friedrich Gauss
Je pouvais faire mieux à l'âge de 5 ans ...
$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

HEIG VD

Pourquoi la solution de Monsieur Gauss est-elle meilleure ?

Pourquoi la solution de Monsieur Gauss est-elle meilleure ?

- Elle est aussi juste

$$\frac{n}{2}\text{ termes}\begin{cases} 1 & + & n & = & n+1 \\ 2 & + & n-1 & = & n+1 \\ 3 & + & n-2 & = & n+1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{n}{2}-1 & + & \frac{n}{2}+2 & = & n+1 \\ \frac{n}{2} & + & \frac{n}{2}+1 & = & n+1 \end{cases}$$

$$S(n) = \frac{n(n+1)}{2}.$$

Pourquoi la solution de Monsieur Gauss est-elle meilleure ?

- Elle est aussi juste

$$\underbrace{\frac{n}{2}\text{ termes}}\left\{\begin{array}{ccccc} 1 & + & n & = & n+1 \\ 2 & + & n-1 & = & n+1 \\ 3 & + & n-2 & = & n+1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{n}{2}-1 & + & \frac{n}{2}+2 & = & n+1 \\ \frac{n}{2} & + & \frac{n}{2}+1 & = & n+1 \end{array}\right.$$

$$S(n) = \frac{n(n+1)}{2}.$$

- Elle est plus efficace

Plus formellement

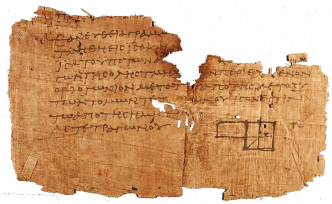
Un problème computationnel est spécifié par un 3 ensembles

- `I`, l'ensemble d'entrées, non vide
- `O`, l'ensemble de sorties, non vide
- `R`, la dépendance relationnelle, où `R ⊆ I × O` définit la sortie correspondant à chaque entrée possible

Par exemple, pour la somme des n premiers entiers,

$$
\begin{aligned}
I &= \mathbb{N} \\
O &= \mathbb{N} \\
R &= \left\{ (n, m) \,\middle|\, m = \sum_{k=1}^{n} k \right\} \subseteq I \times O.
\end{aligned}
$$

Pile de dossier

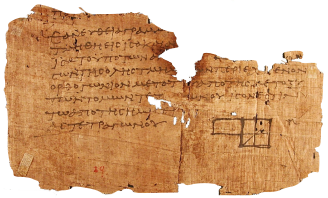


- Plusieurs dossiers, classés par ordre alphabétique sur le nom, sont empilés sur une table.
- On veut déterminer si une personne de nom X a un dossier à son nom dans la pile.
- Quel algorithme choisir pour résoudre ce problème ?

Recherche séquentielle

```
début
    tant que la pile de dossiers P n'est pas vide
    et le nom du dossier au sommet de P est inférieur à X faire
        Prendre le dossier au sommet de P
        Le poser à coté de P
    fintq
    si P est vide ou le nom du dossier au sommet de P est différent de X alors
        R ← non
    sinon
        R ← oui
    finsi
fin
```

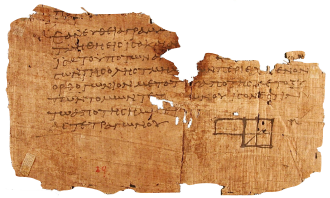
Recherche dichotomique



début

  tant que *la pile de dossiers* P *n'est pas vide*
  et *le nom du dossier au sommet de* P *est différent de* X **faire**
    Couper la pile de dossiers P en 2 parties
    (On appellera $Psup$ la moitié supérieure, et $Pinf$ la moitié inférieure)
    si *le nom du dossier au sommet de* $Pinf = X$ **alors**
      Ne garder que le premier dossier de $Pinf$ dans P
    sinon si *le nom du dossier au sommet de* $Pinf > X$ **alors**
      Enlever de P tous les dossiers de $Pinf$
    sinon
      Enlever de P le premier dossier de $Pinf$
      Enlever de P tous les dossiers de $Psup$
    finsi
  fintq

Recherche dichotomique



```
        |   Ne garder que le premier dossier de Pinf dans P
        sinon si le nom du dossier au sommet de Pinf > X alors
        |   Enlever de P tous les dossiers de Pinf
        sinon
        |   Enlever de P le premier dossier de Pinf
        |   Enlever de P tous les dossiers de Psup
        finsi
    fintq
    si P est vide alors
    |   R ← non
    sinon
    |   R ← oui
    finsi
fin
```

Quel algorithme choisir?

Séquentielle

- Grand nombre de comparaisons, au pire n pour n dossiers.
- Comparaison par égalité, `operator==` en C++
- Pas de pré-requis d'ordre
- Opérations simples : accéder au dossier suivant

Dichotomique

- Petit nombre de comparaisons, au pire log2(n) pour n dossiers.
- Comparaison par inégalité , `operator<` en C++
- Pré-requis d'ordre
- Opérations plus complexes : accéder au milieu de la pile

Recherches en C++

Séquentielle

`std::find`

`std::find_if`

`std::find_if_not`

`std::find_first_of`

`std::adjacent_find`

`std::search`

`std::search_n`

`std::find_end`

Dichotomique

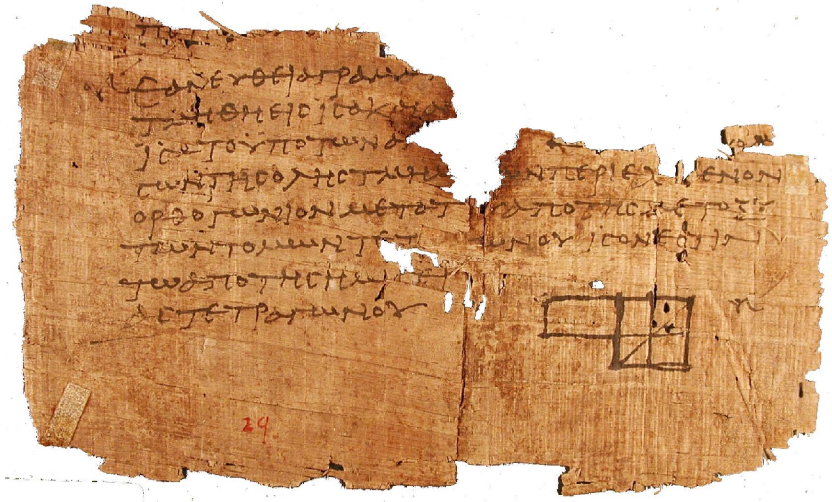
`std::lower_bound`

`std::upper_bound`

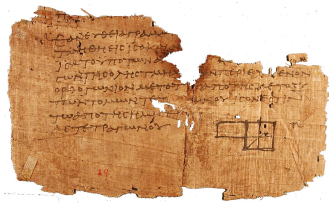
`std::equal_range`

`std::binary_search`

1.2. Complexités



Estimer la complexité



- Quelle est la complexité de ce programme ?
- Quelle est la relation entre la valeur de n et le temps d'exécution / le nombre d'instructions effectuées ?

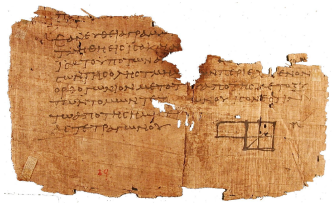
```
int main() {

  int n;
  cin >> n;

  int k = 0;

  ++k;

  for(int i = 1; i <= n; ++i) {
    ++k;
  }

  for(int i = 0; i <= n; ++i) {
    for(int j = 0; j <= n; ++j) {
      ++k;
    }
  }

  cout << k;
}
```

Estimer la complexité

- Quelle est
  complexit
  programm
- Quelle est
  entre la va
  le temps
  d'exécuti
  nombre d
  effectuées

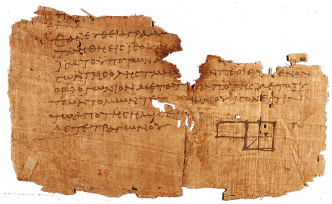
```
++k;

for(int i = 1; i <= n; ++i) {
  ++k;
}

for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

Estimer la complexité



```
++k;

for(int i = 1; i <= n; ++i) {
  ++k;
}

for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

Estimer la complexité

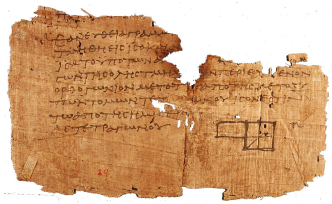


```
++k;
```

O(1)

```
for(int i = 1; i <= n; ++i) {
  ++k;
}

for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

Estimer la complexité



```
++k;
```

O(1)

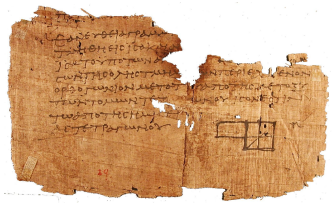
```
for(int i = 1; i <= n; ++i) {
  ++k;
}
```

O(n)

```
for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

HE VD
IG

Estimer la complexité



```
++k;
```

O(1)

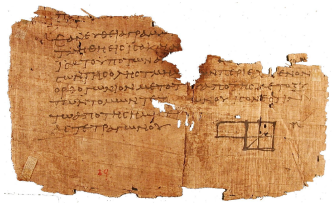
```
for(int i = 1; i <= n; ++i) {
  ++k;
}
```

O(n)

```
for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

$O(n^2)$

Estimer la complexité



$O(n^2)$

```
++k;
```

$O(1)$

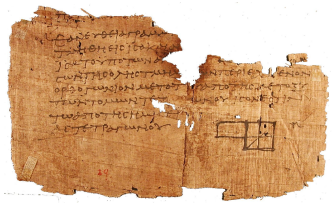
```
for(int i = 1; i <= n; ++i) {
  ++k;
}
```

$O(n)$

```
for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

$O(n^2)$

Boucles particulières



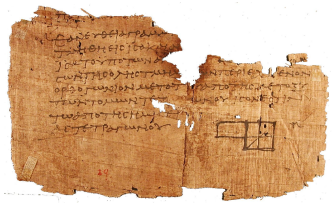
```
for(int i = 0; i < n; ++i) {
   for(int j = 0; j < m; ++j) {
     k++;
   }
 }
```

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < i; ++j) {
    k++;
  }
}
```

```
for(int i = 1; i <= n; i *= 2) {
  k++;
}
```

```
for(int i = n; i > 0; i /= 3) {
  k++;
}
```

Enchainement alternatif



```
if( rand() % n == 0 ) {
  for(int i = 0; i < n; ++i) {
      ++k;
  }
} else {
  ++k;
}
```

- Quelle est la complexité dans le pire des cas ?
- Quelle est la complexité dans le meilleur cas ?
- Quelle est la complexité en moyenne ?

Exercice 1.1

Quelle est la complexité dans le **pire** des cas ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
vector<int> v(n);
generate(v.begin(), v.end(), rand);

for(size_t i = 1; i < v.size(); ++i)
{
  size_t j = i;
  while( j > 0 and v[j-1] > v[j] )
  {
    swap(v[j-1], v[j]);
    --j;
  }
}
```

Exercice 1.2

Quelle est la complexité dans le **meilleur** cas ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
vector<int> v(n);
generate(v.begin(), v.end(), rand);

for(size_t i = 1; i < v.size(); ++i)
{
  size_t j = i;
  while( j > 0 and v[j-1] > v[j] )
  {
    swap(v[j-1], v[j]);
    --j;
  }
}
```

Exercice 1.3

Quelle est la complexité dans le cas **moyen** ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
vector<int> v(n);
generate(v.begin(), v.end(), rand);

for(size_t i = 1; i < v.size(); ++i)
{
  size_t j = i;
  while( j > 0 and v[j-1] > v[j] )
  {
    swap(v[j-1], v[j]);
    --j;
  }
}
```

Exercice 2

Que vaut k à 5% près ?

- 10
- 500
- 1000
- 250000
- 1000000
- Autre ?

```
int k = 0, p2 = 1, n = 1000;
for(int i = 0; i < n; ++i) {
  if(i == p2) {
    for(int j = 0; j < i; ++j) {
      ++k;
    }
    p2 *= 2;
} }
```

Exercice 3.1

Quelle est la complexité dans le **pire** des cas ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
size_t rechercheLineaire(
       const vector<string>& v,
       const string& s )
{
  const size_t n = v.size();
  for(size_t i = 0; i < n; ++i)
  {
    if(v.at(i) == s)
      return i;
  }
   return -1;
}
```

Exercice 3.2

Quelle est la complexité dans le **meilleur** cas ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
size_t rechercheLineaire(
       const vector<string>& v,
       const string& s )
{
  const size_t n = v.size();
  for(size_t i = 0; i < n; ++i)
  {
    if(v.at(i) == s)
      return i;
  }
   return -1;
}
```

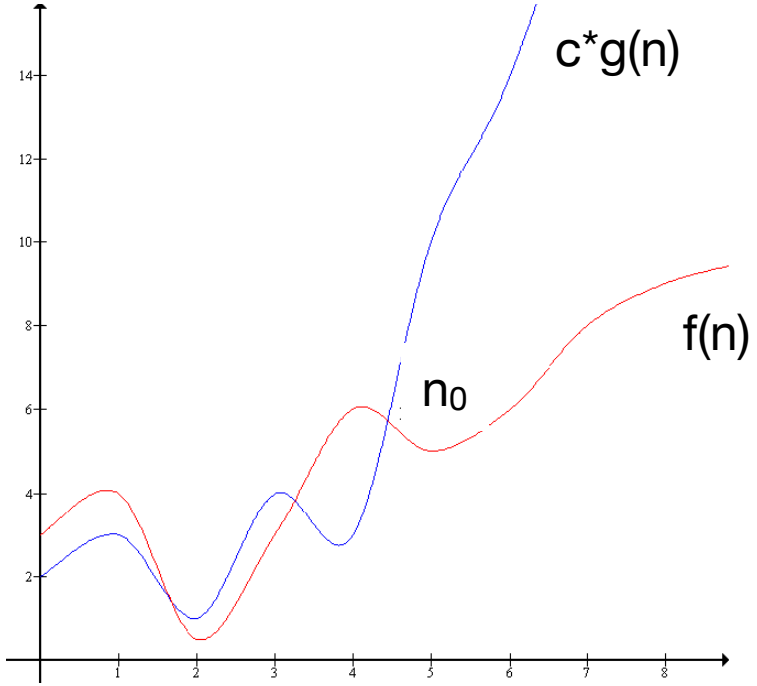
Exercice 3.3

Quelle est la complexité dans le cas **moyen** ?

- O(1)
- O(log(n))
- O(n)
- O(n*log(n)
- O(n^2)
- Autre ?

```
size_t rechercheLineaire(
       const vector<string>& v,
       const string& s )
{
  const size_t n = v.size();
  for(size_t i = 0; i < n; ++i)
  {
    if(v.at(i) == s)
      return i;
  }
   return -1;
}
```

Notation de Landau



$$f(n) = O(g(n)) \quad \textbf{si}$$

$$\exists c > 0 \; \exists n_0 > 0$$

$$\forall n \geq n_0 : \quad f(n) \leq cg(n).$$

Edmund Landau

(Allemagne, 1877 – 1938)

O(…) en pratique

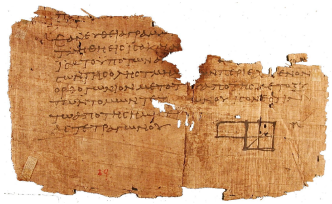
- Somme de termes - ignorer les termes croissant moins vite que les autres

$$1 + n^2 + n^5 + n^8 = O(n^8)$$

- Produits de facteurs - ignorer les facteurs constants

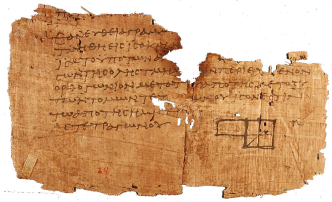
$$3.1415 * n^2 = O(n^2)$$

Notations complémentaires



grand O	f = O(g)	Notation de Landau	f croit au plus aussi vite que g
petit o	f = o(g)	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$	f croit strictement plus lentement que g
grand Oméga	f = Ω(g)	g = O(f)	f croit au moins aussi vite que g
grand Thêta	f = Θ(g)	f = O(g) et f = Ω(g)	f et g ont le même ordre de grandeur

Exemple



Soit $f(n) = 1 + 3 \cdot n^2 + 4 \cdot n^3$

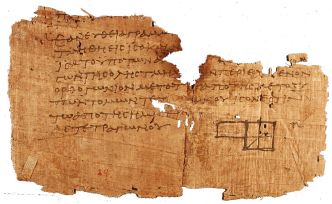
$f(n) = O(f(n)), O(n^3), O(25 \cdot n^3), O(n^6), O(2^n), \ldots$

$f(n) = o(n^6), o(2^n),$ mais pas $o(f(n))$ ou $o(n^3)$

$f(n) = \Omega(f(n)), \Omega(n^3), \Omega(n^2), \Omega(\log_2(n)), \ldots$

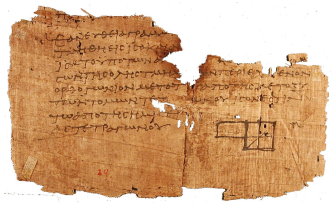
$f(n) = \Theta(f(n)), \Theta(n^3), \Theta(1000 \cdot n^3), \Theta(0.00023 \cdot n^3)$

Pour n allant de 10 à 10 millions…



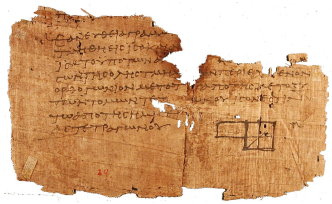
n	log(n)	n	n log(n)	n^2	n^3	2^n
10	1	10	10	100	1000	1024
100	2	100	200	10000	1000000	1.26E+30
1000	3	1000	3000	1000000	1E+09	1.07E+301
10000	4	10000	40000	1E+08	1E+12	
100000	5	100000	500000	1E+10	1E+15	
1000000	6	1000000	6000000	1E+12	1E+18	
10000000	7	10000000	70000000	1E+14	1E+21	

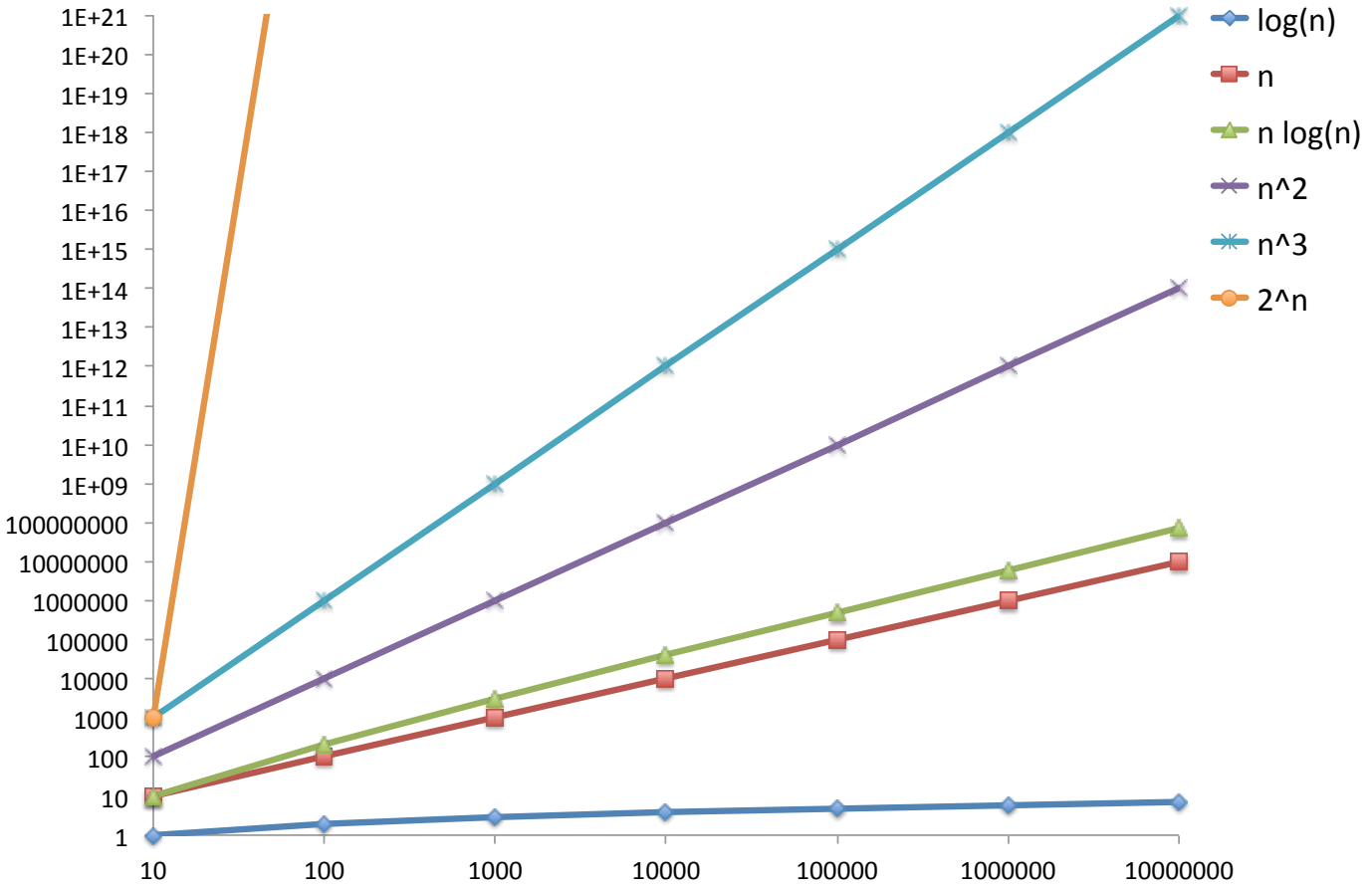
Si une opération dure 1 nanoseconde…



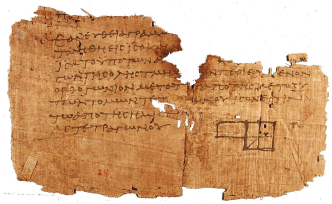
n	log(n)	n	n log(n)	n^2	n^3	2^n
10	1 ns	10 ns	10 ns	100 ns	1 µs	1 µs
100	2 ns	100 ns	200 ns	10 µs	1 ms	3000 fois l'âge de l'univers
1000	3 ns	1 µs	3 µs	1 ms	1 s	
10000	4 ns	10 µs	40 µs	100 ms	17 min	
100000	5 ns	100 µs	500 µs	10 s	11 jours	
1000000	6 ns	1 ms	6 ms	17 min	32 ans	
10000000	7 ns	10 ms	70 ms	1 jour	31700 ans	

Graphique log-log



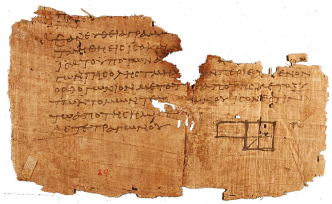


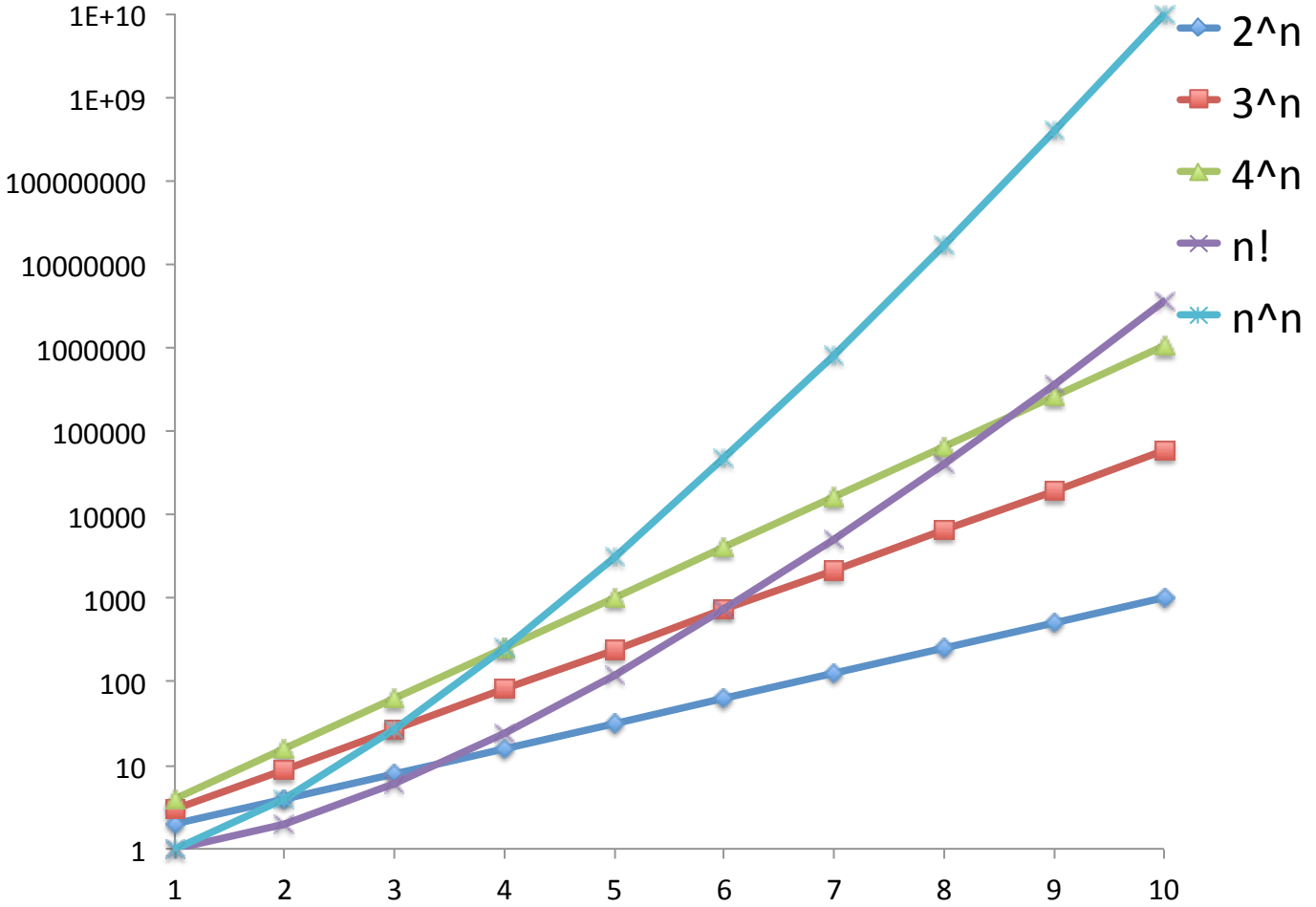
Pour n allant de 1 à 10…



n	2^n	3^n	4^n	n!	n^n
1	2	3	4	1	1
2	4	9	16	2	4
3	8	27	64	6	27
4	16	81	256	24	256
5	32	243	1024	120	3125
6	64	729	4096	720	46656
7	128	2187	16384	5040	823543
8	256	6561	65536	40320	16777216
9	512	19683	262144	362880	387420489
10	1024	59049	1048576	3628800	1E+10

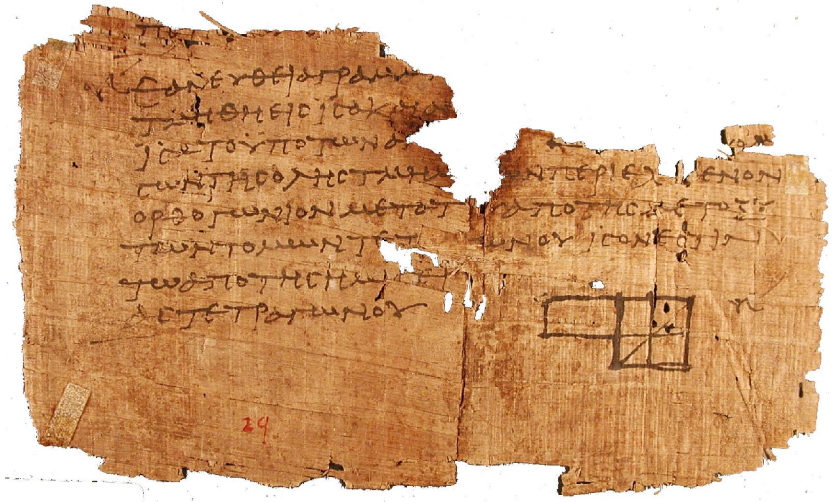
Graphique linéaire-log



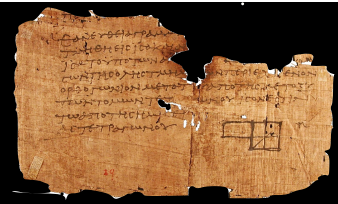


HE VD
IG

1.3. Structures de Données



HEIG VD

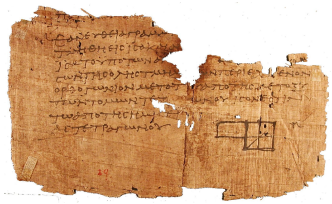


Mais au fond, ~~qu'est-ce qu'un algorithme~~ ?

Qu'est-ce qu'une structure de donnée ?

Qu'est-ce qu'un type ?

Le type `int` en C++



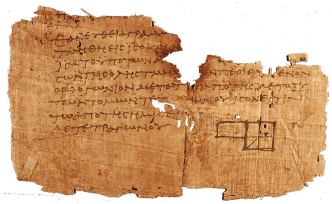
Données

32 bits représentant des entiers de -2147483648 à 2147483647

Opérations

addition, soustraction, multiplication, division euclidienne, modulo

Une pile…



Données

Un ensemble d'éléments stockés en mémoire sous une forme ou une autre

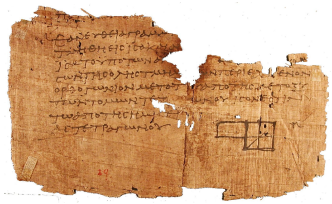
Opérations

- push - empiler
- pop - dépiler
- top - sommet
- empty - vide

Type de données abstrait

- Données
- Opérations pour gérer ces données
- Pas de détails sur la mise en oeuvre

std::vector



std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vector

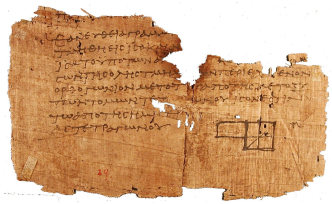
Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

std::vector



std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vector

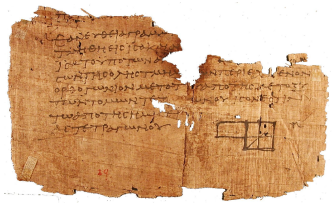
Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

std::vector



std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vector

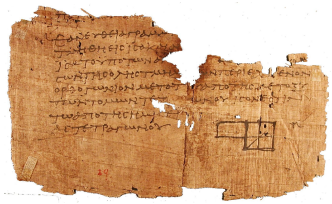
Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

std::vector



std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vector

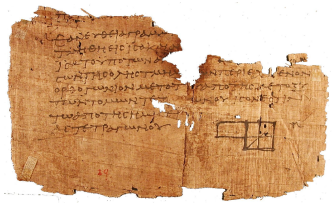
Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

std::vector



std::vector

```
template < class T, class Alloc = allocator<T> > class vector;
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

Paragidmes du TDA

- Abstraction : Identifier les caractéristiques importantes vis-à vis des applications envisagées
- Encapsulation : intégrer données et les opérations permet de masque les détails de mise en oeuvre
- Modularité : Division du problème en éléments réutilisables

Opérations d'un TDA

- Constructeurs : créent les instances
- Modificateurs : modifient les données stockées
- Sélecteurs : interrogent la structure sans la modifier
- Itérateurs : parcourent les données

Exercice 4

string::push_back(…)

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

string::string(const string&)

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

string::string(const string&)

string::begin(…)

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

string::string(const string&)

string::begin(…)

string::resize(…)

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

string::string(const string&)

string::begin(…)

string::resize(…)

string::capacity(…)

Exercice 4

De quel type d'opération d'agit-il ?

- Constructeur
- Modificateur
- Sélecteur
- Itérateur

string::push_back(…)

string::find_first_of(…)

string::string(const string&)

string::begin(…)

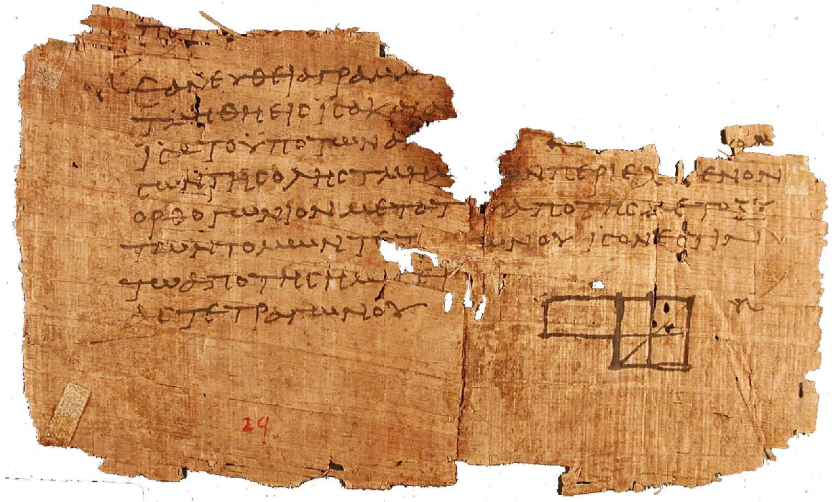
string::resize(…)

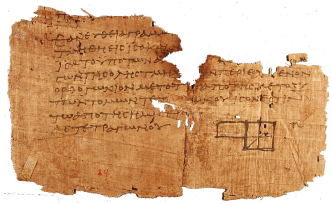
string::capacity(…)

string::back(…)

HE VD
IG

1.4.
Mise en oeuvre d'une pile en C++





Data

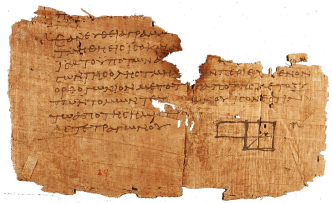
```
const size_t CAPACITY = 10;
int data[CAPACITY];
size_t nbElements = 0;
```

Modificateurs

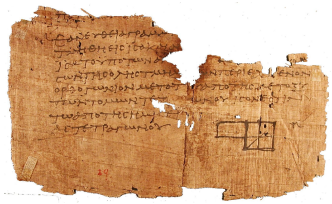
```
void push(int i) {
 data[nbElements++] = i;
}

void pop() {
  nbElements--;
}
```

Sélecteurs

```
int top() {
  return data[nbElements - 1];
}

bool empty() {
  return (nbElements == 0);
}
```

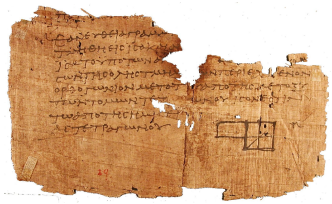


```
struct Stack // structure Pile
{
  static const size_t CAPACITY = 10;
  int data[CAPACITY];
  size_t nbElements = 0;

  void push(int i) {
    data[nbElements++] = i;
  }

  void pop(){
    nbElements--;
  }

  int top(){
    return data[nbElements - 1];
  }

  bool empty() {
    return (nbElements == 0);
  }
};
```

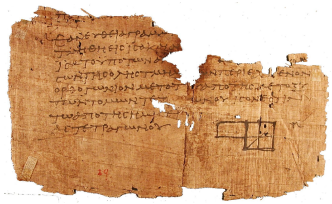


```
struct Stack // structure Pile
{
  static const size_t CAPACITY = 10;
  int data[CAPACITY];
  size_t nbElements = 0;

  void push(int i) {
    data[nbElements++] = i;
  }

  void pop(){
    nbElements--;
  }

  int top(){
    return data[nbElements - 1];
  }

  bool empty() {
    return (nbElements == 0);
  }
};
```

```
Stack s;
for(int i = 0; i < 5; ++i) {
  s.push(i);
}

while(not s.empty()) {
  cout << s.top() << ' ';
  s.pop();
}
```

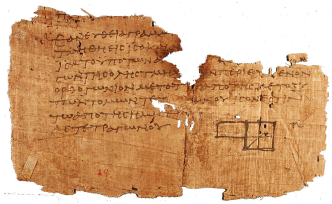


```
class Stack  // classe Pile
{
private:     // attributs privés
  static const size_t CAPACITY = 10;
  int data[CAPACITY];
  size_t nbElements;

public:      // méthodes publiques
  Stack();   // constructeur
  void push(int i);
  void pop();
  int top();
  bool empty();
};

Stack::Stack() {
  nbElements = 0;
}

bool Stack::empty() {
  return (nbElements == 0);
}
```



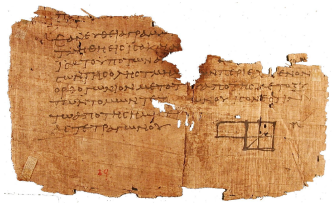
```
class Stack  // classe Pile
{
private:     // attributs privés
  static const size_t CAPACITY = 10;
  int data[CAPACITY];
  size_t nbElements;

public:      // méthodes publiques
  Stack();   // constructeur
  void push(int i);
  void pop();
  int top();
  bool empty();
};

Stack::Stack() {
  nbElements = 0;
}

bool Stack::empty() {
  return (nbElements == 0);
}
```

```
Stack s;
for(int i = 0; i < 5; ++i) {
  s.push(i);
}

while(not s.empty()) {
  cout << s.top() << ' ';
  s.pop();
}
```

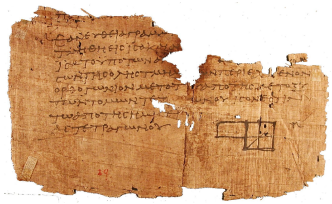


```
class ErrorStackEmpty { };

void Stack::pop() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  nbElements--;
}

int Stack::top() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  return data[nbElements - 1];
}


class ErrorStackFull { };

void Stack::push(int i) {
  if(nbElements == CAPACITY)
    throw ErrorStackFull();
  data[nbElements++] = i;
}
```

```
int main() {
  // on crée une pile
  Stack unePile;

  try {
    // on empile 1,2,3
    unePile.push(1);
    unePile.push(2);
    unePile.push(3);

    // on affiche 3 2 1
    while (not unePile.empty()) {
      cout << unePile.top() << " ";
      unePile.pop();
    }
  }
  catch (ErrorStackEmpty) {
    cout << "Erreur pile vide";
  }
  catch (ErrorStackFull) {
    cout << "Erreur pile pleine";
  }
}
```

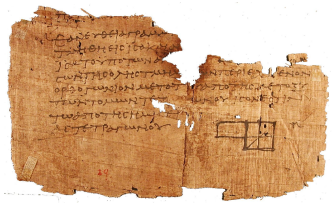


```
template <typename T> class Stack
{
private:
  static const size_t CAPACITY = 10;
  T data[CAPACITY]; size_t nbElements;
public:
  Stack(); void push(T i); void pop();
  T top(); bool empty();
};

template <typename T>
void Stack<T>::pop() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  nbElements--;
}

template <typename T>
T Stack<T>::top() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  return data[nbElements - 1];
}
```



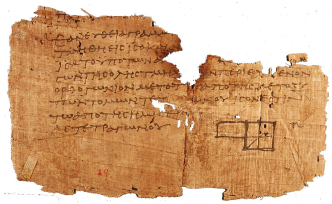
```
template <typename T> class Stack
{
private:
  static const size_t CAPACITY = 10;
  T data[CAPACITY]; size_t nbElements;
public:
  Stack(); void push(T i); void pop();
  T top(); bool empty();
};

template <typename T>
void Stack<T>::pop() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  nbElements--;
}

template <typename T>
T Stack<T>::top() {
  if(nbElements == 0)
    throw ErrorStackEmpty();
  return data[nbElements - 1];
}
```

```
Stack<int> s;
for(int i = 0; i < 5; ++i) {
  s.push(i);
}

while(not s.empty()) {
  cout << s.top() << ' ';
  s.pop();
}
```



```
template <typename T, size_t N> class Stack
{
private:
  T data[N]; size_t nbElements;
public:
  Stack(); void push(T i); void pop();
  T top(); bool empty();
};

template <typename T, size_t N>
void Stack<T,N>::push(T i) {
  if(nbElements == N) throw ErrorStackFull();
  data[nbElements++] = i;
}

template <typename T, size_t N>
void Stack<T,N>::pop() {
  if(nbElements == 0) throw ErrorStackEmpty();
  nbElements--;
}

template <typename T, size_t N>
T Stack<T,N>::top() {
  if(nbElements == 0) throw ErrorStackEmpty();
  return data[nbElements - 1];
}
```

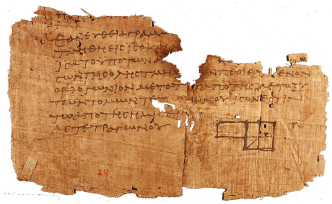
HE VD
IG



```
template <typename T, size_t N> class Stack
{
private:
  T data[N]; size_t nbElements;
public:
  Stack(); void push(T i); void pop();
  T top(); bool empty();
};

template <typename T, size_t N>
void Stack<T,N>::push(T i) {
  if(nbElements == N) throw ErrorStackFull();
  data[nbElements++] = i;
}

template <typename T, size_t N>
void Stack<T,N>::pop() {
  if(nbElements == 0) throw ErrorStackEmpty();
  nbElements--;
}

template <typename T, size_t N>
T Stack<T,N>::top() {
  if(nbElements == 0) throw ErrorStackEmpty();
  return data[nbElements - 1];
}
```

```
Stack<int,10> s;
for(int i = 0; i < 5; ++i) {
  s.push(i);
}

while(not s.empty()) {
  cout << s.top() << ' ';
  s.pop();
}
```

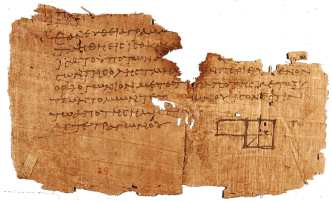
std::stack



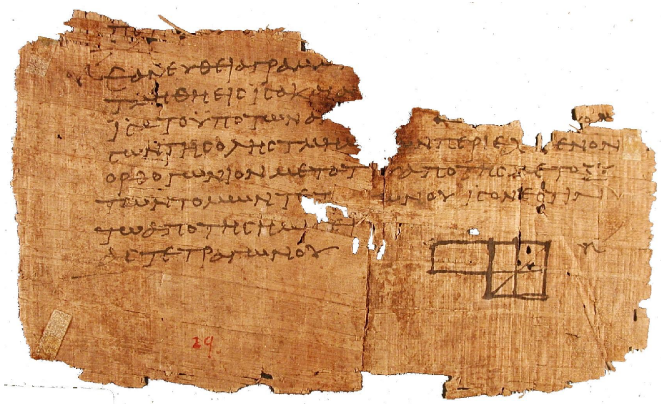
```
template <class T, class Container = deque<T> > class stack;
```

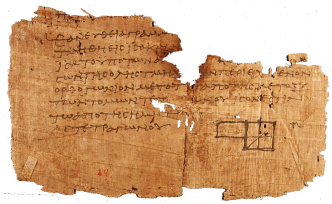
```
#include <iostream>  // std::cout
#include <stack>     // std::stack

int main ()
{
  std::stack<int> mystack;

  for (int i=0; i<5; ++i) mystack.push(i);

  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }

  return 0;          // ce programme affiche
}                    // 4 3 2 1 0
```

En résumé

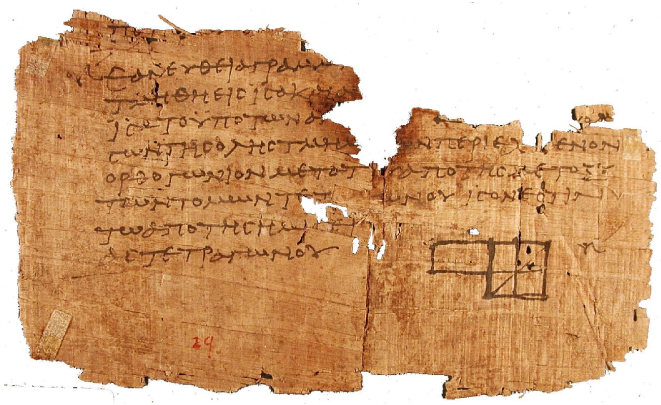


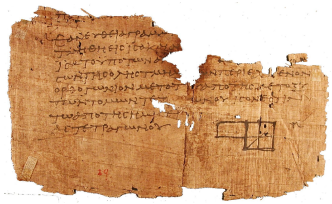
En résumé

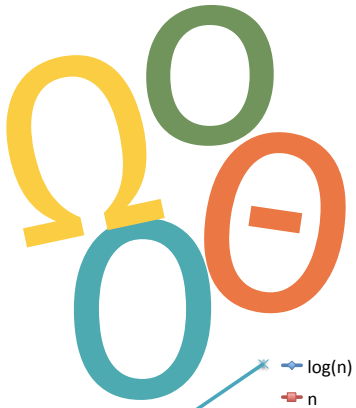


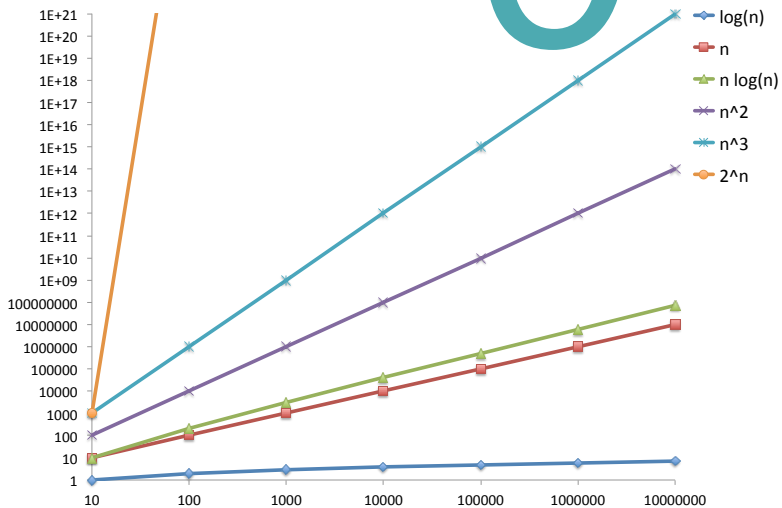


En résumé

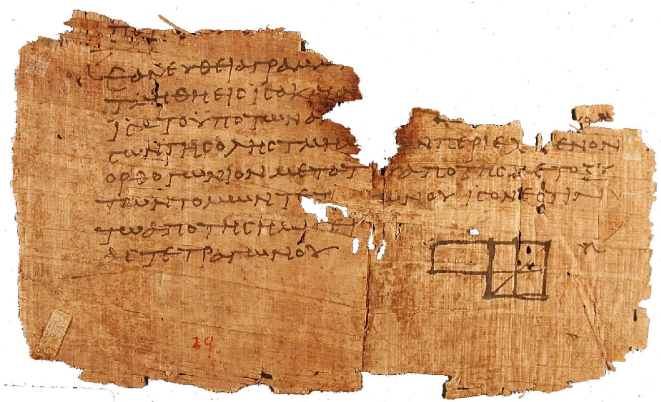


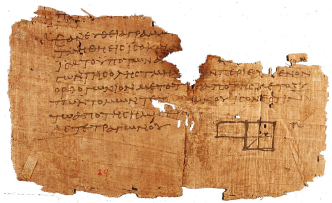


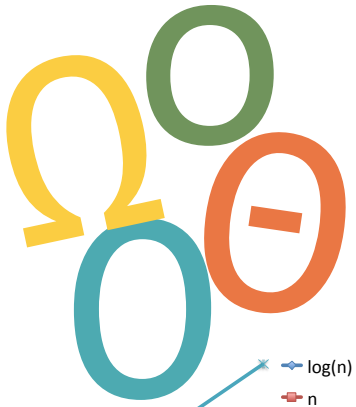


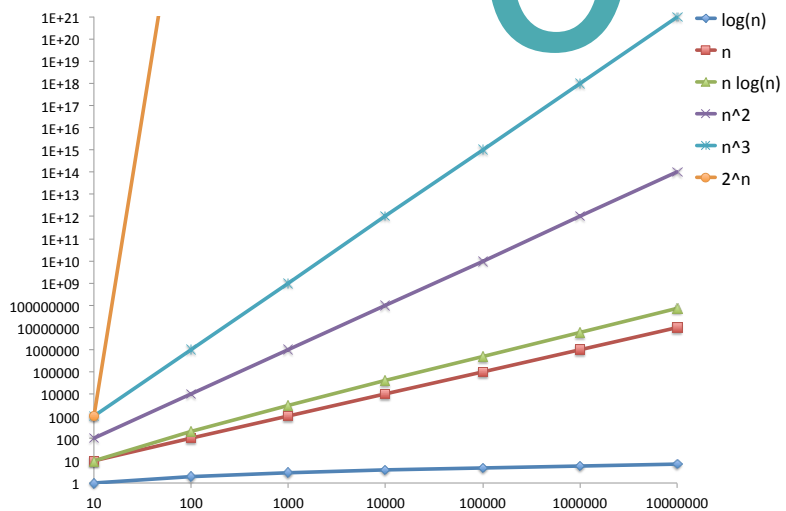

1E+21
1E+20
1E+19
1E+18
1E+17
1E+16
1E+15
1E+14
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
10
100
1000
10000
100000
1000000
10000000
log(n)
n
n log(n)
n^2
n^3
2^n

En résumé








1E+21
1E+20
1E+19
1E+18
1E+17
1E+16
1E+15
1E+14
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
10
100
1000
10000
100000
1000000
10000000
log(n)
n
n log(n)
n^2
n^3
2^n