Projet de Programmation 2

P.A. Wacrenier@labri.fr
Bât A29 bis (INRIA)

Organisation

- 2 cours
  - Présentation
  - Problème du sac à dos
- 6 TD
  - 1 TD échauffement
  - 4 TD évaluation-formative
  - 1 TD Rapport
- DS (5 avril)
- Soutenance + Rapport

Organisation

- 2 cours
  - Présentation
  - Problème du sac à dos
- 6 TD
  - 1 TD échauffement
  - 4 TD évaluation-formative
  - 1 TD Rapport
- DS (5 avril)
- Soutenance + Rapport

Évaluation-formative

- Objectifs :
  - Mettre au centre le travail réalisé par l'étudiant
  - Montrer aux étudiants ce que l'on attend d'eux sur leur travail
  - Amener le trinôme à améliorer son projet

- Mise en œuvre :
  - Pour chaque TD un ensemble d'objectifs et un barème sont fixés
  - Les étudiants transmettent leur travail à l'enseignant au moins 24h avant le TD
  - L'enseignant évalue le travail en dehors du TD
  - Lors du TD
    - Restitution par l'enseignant de son évaluation
    - Discussion pour améliorer la qualité du travail
    - Préparation des objectifs prochains
- Tout manque de sincérité sera lourdement sanctionné

Nos objectifs

- TD1
  - 1 respect des consignes
  - 2 compte rendu
  - 3 algorithmes gloutons
  - 2 étude récursif
  - 2 qualité du code
- TD2
  - ½ respect des consignes
  - 2 compte rendu
  - 2 algo récursif
  - 2 étude PSE
  - 1 modularité
  - 2 ½ qualité du code
- TD3
  - ½ respect des consignes
  - 2 compte rendu
  - 3 algo PSE
  - 1 ½ étude mémorisation
  - 3 Qualité du code
- TD4
  - ½ respect des consignes
  - 2 compte rendu
  - 2 mémorisation
  - 2 étude valorisation
  - 3 ½ Qualité du code

Objectifs

- Consolider votre culture algorithmique et votre savoir faire en matière de programmation

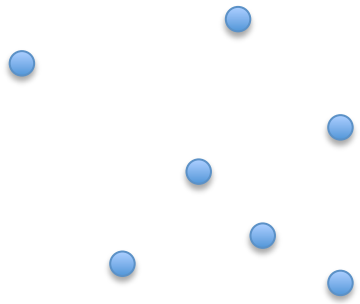
- Mettre en œuvre
  - Algorithme 1
  - Programmation 1
  - Environnement de Développement

- Étudier un problème standard
  - Problème complexe du point de vue temps de résolution
  - Méthodes approchées (en temps polynomial)
  - Méthodes exacts (en temps exponentiel)

  → Ressentir la complexité : confronter théorie et pratique

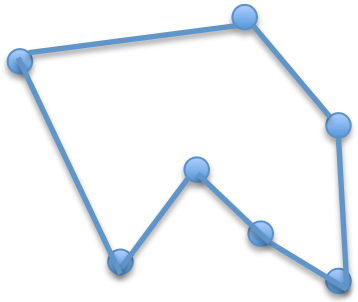
Exemple de problèmes complexes

- Le problème du voyageur de commerce
  - « Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur ».



Exemple de problèmes complexes

- Le problème du voyageur de commerce
  - « Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur ».



Longueur=112

Exemple de problèmes complexes

- Le problème du sac à dos :
  - *« Étant donné plusieurs objets possédant chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ? »*
  - *Max = 3000*

Fer-a-repasser	1250	35
Bague	12.1	3400
Tournevis	270	7
Ordinateur	2500	1234
Baguette	250	0.95
Foie-gras	400	34
Foie-gras	300	20.45
Television	1700	1150
Livre	412	8.50
Chaussures	900	99
Pull-over	800	45

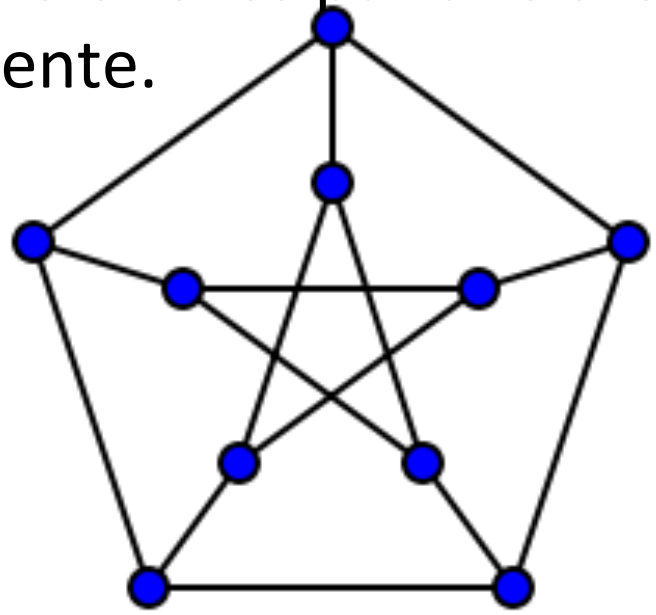
Exemple de problèmes complexes

- Le problème du *bin packing* :
  - *« Étant donné plusieurs objets possédant chacun une dimension et combien faut-il de sacs de capacité C pour empaqueter tous les objets ? »*
  - *C=3000*

Fer-a-repasser	1250	35
Bague	12.1	3400
Tournevis	270	7
Ordinateur	2500	1234
Baguette	250	0.95
Foie-gras	400	34
Foie-gras	300	20.45
Television	1700	1150
Livre	412	8.50
Chaussures	900	99
Pull-over	800	45

Exemple de problèmes complexes

- K-coloration de graphe
  - Étant donné un graphe et k couleurs, peut-on colorier ce graphe de telle façon à ce que deux sommets reliés par une arête soient de couleur différente.



Exemple de problèmes complexes

- Satisfiabilité d'une formule booléenne
  - Étant donnée une formule booléenne $f(a_1,a_2,\ldots,a_n)$ décider si f peut être vérifiée.

$(a_1$ *et* a_2 *et non* $(a_3))$ *ou* $(a_3$ *et non*$(a_2))$

Problèmes (co) NP

- Classe des problèmes faciles à résoudre avec de la chance.
  - Facile = en temps polynomial
  - Avec de la chance = machine non déterministe
  - On a toujours de la chance

- NP :
  - Avec de la chance on tombe sur une solution candidate et on vérifie *facilement* que la solution est bonne.
- CoNP :
  - Avec de la chance on tombe sur un contre-exemple *facile* à vérifier.
- P = NP ?
  - P inclus dans NP
  - La réciproque n'est pas démontrée ni infirmée à ce jour

NP ou Co-NP ?

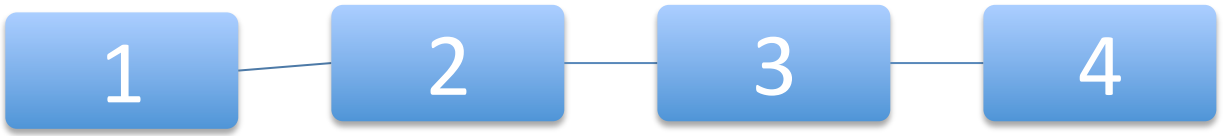
- Voyageur de commerce
- Satisfiabilité d'une formule booléenne
- K-coloration d'un graphe
- Bin-packing
- Sac à dos

NP ou Co-NP ?

- Voyageur de commerce
- Satisfiabilité d'une formule booléenne
- K-coloration d'un graphe
- Bin-packing
- Sac à dos

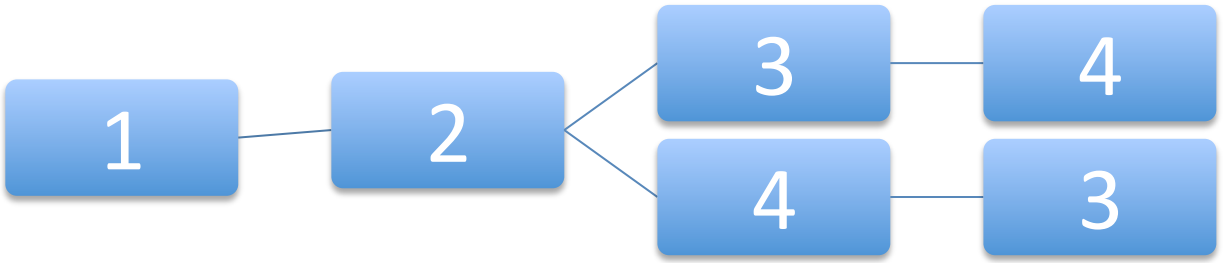
Voyageur de commerce

- On numérote les villes 1,2,3 et 4
- Nombre de parcours possibles :



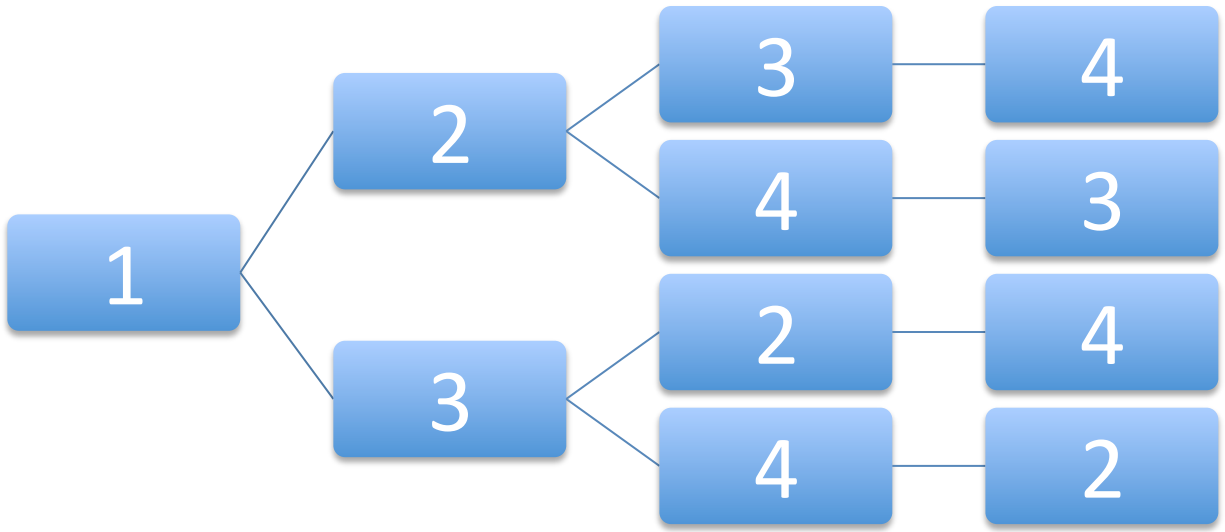
Voyageur de commerce

- On numérote les villes 1,2,3 et 4
- Nombre de parcours possibles :



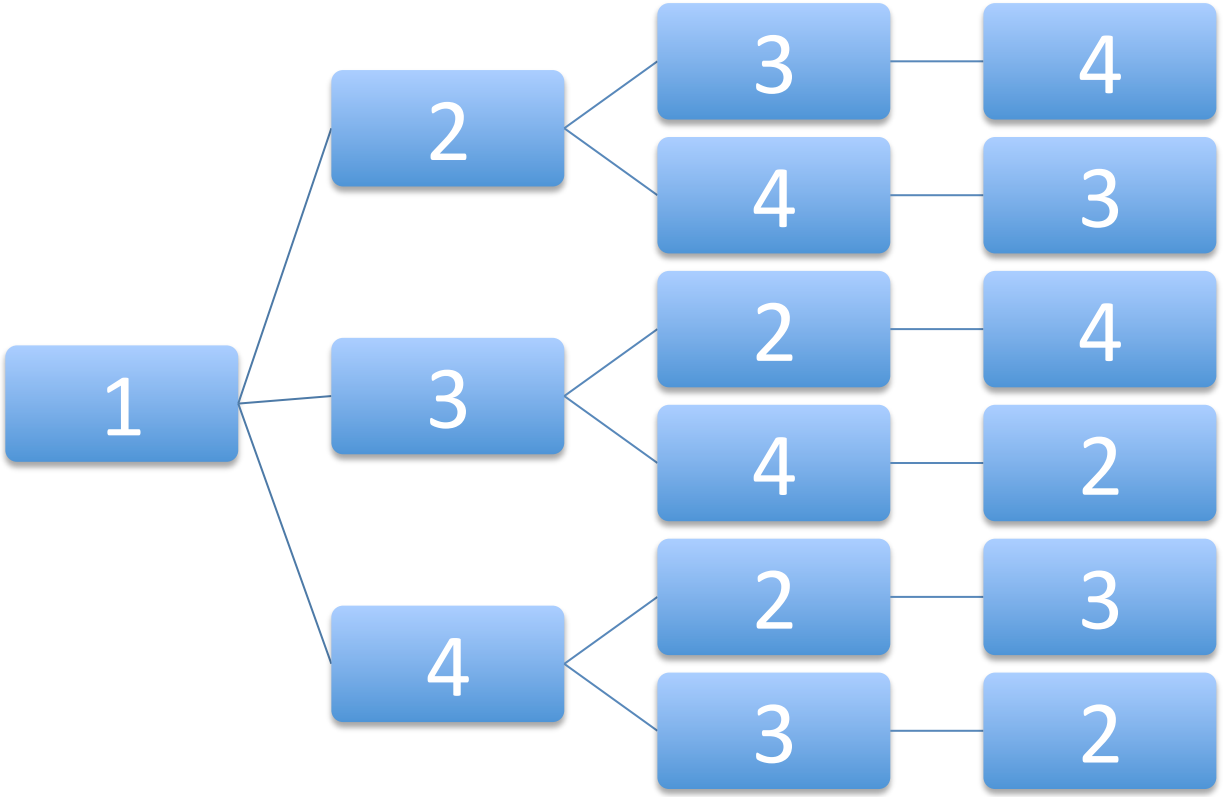
Voyageur de commerce

- On numérote les villes 1,2,3 et 4
- Nombre de parcours possibles :



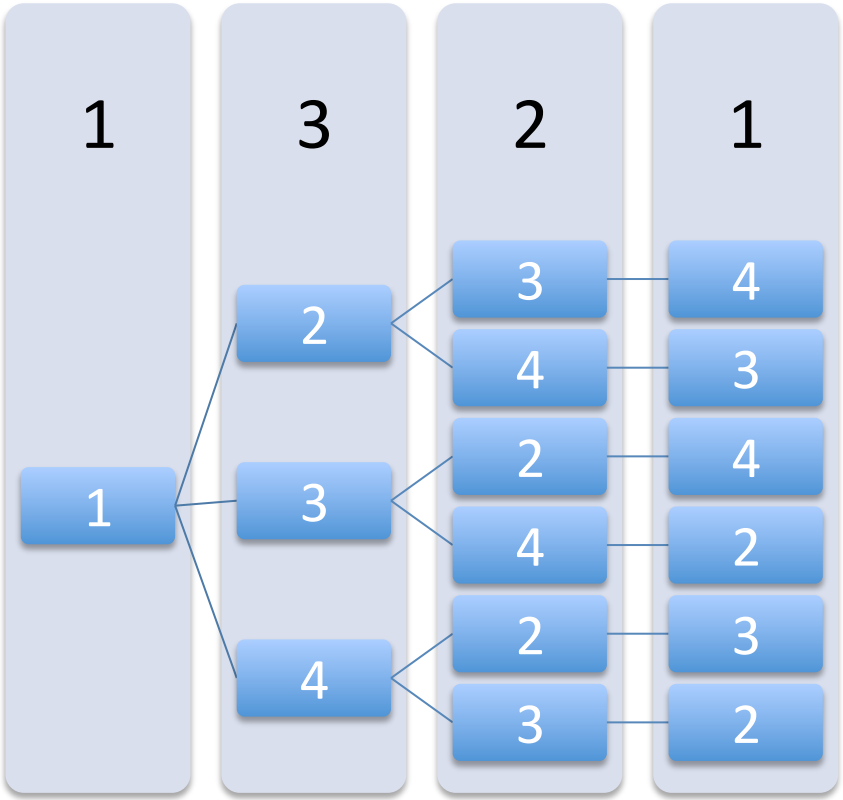
Voyageur de commerce

- On numérote les villes 1,2,3 et 4
- Nombre de parcours possibles :



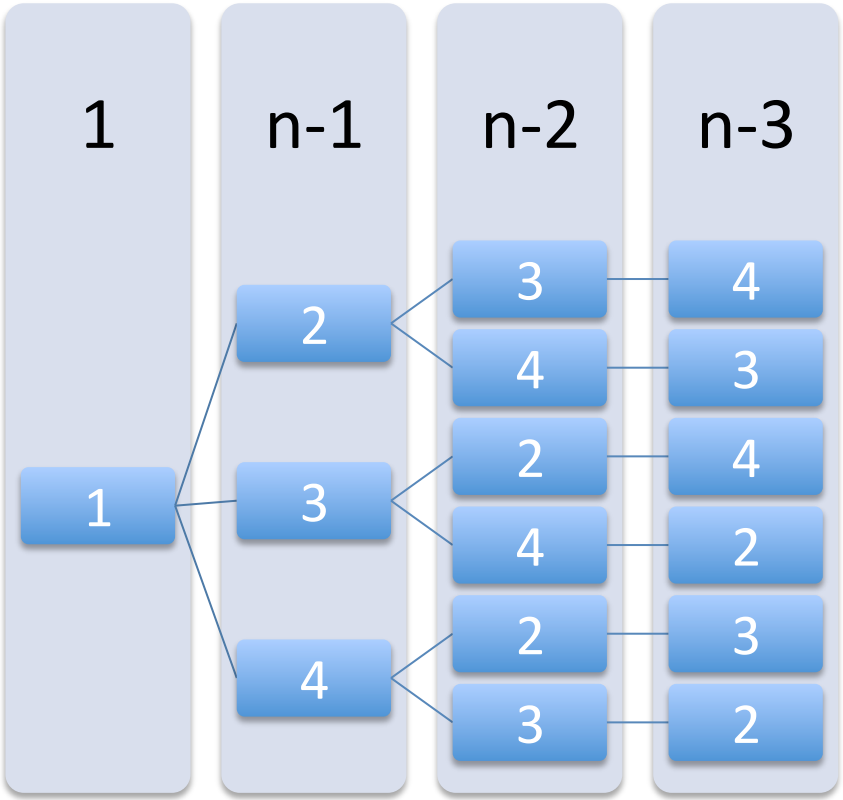
Voyageur de commerce

- On numérote les villes 1,2,3 et 4
  - Nombre de parcours possibles : 6



Voyageur de commerce

- Généralisation à n villes
- Nombre de parcours possibles : (n-1)!



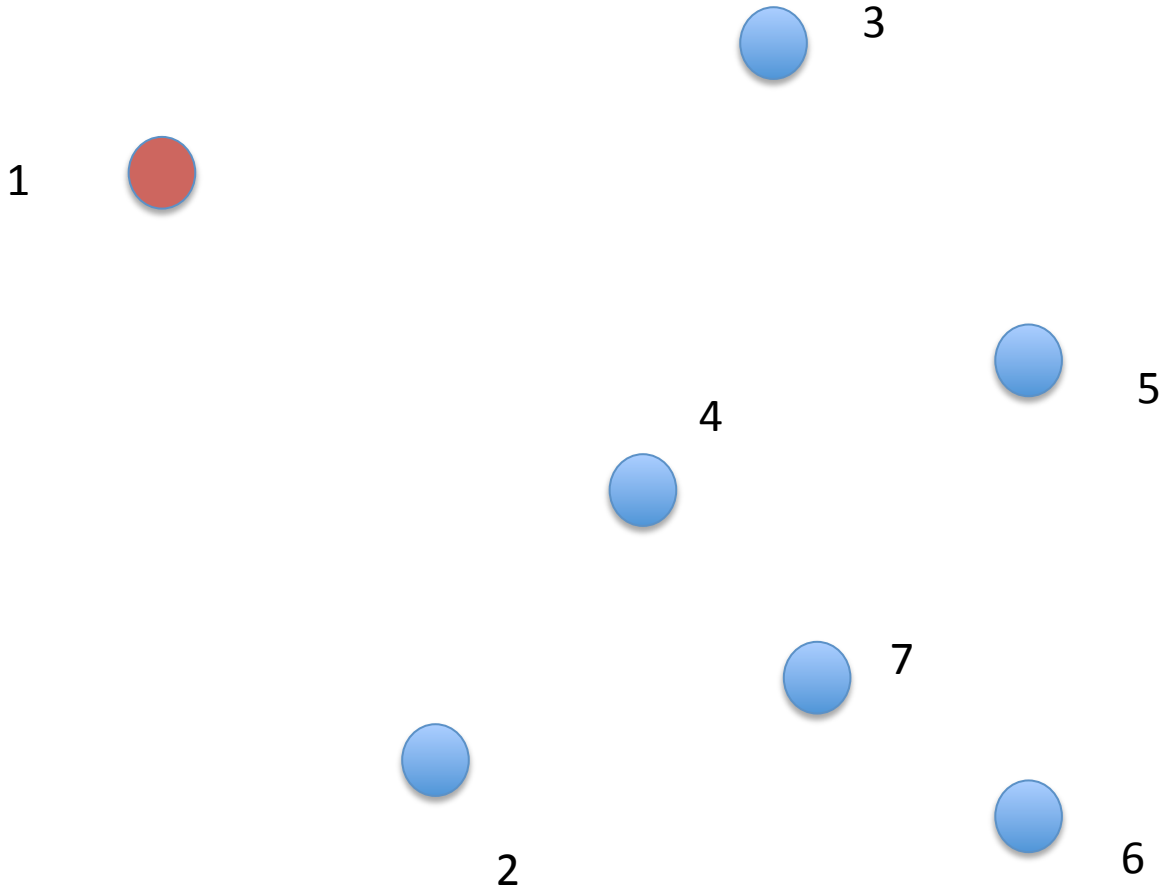
Méthodes exactes

- Méthodes exhaustives
  - Tester tous les parcours possibles
    - Explorer l'arbre de décision
    - Complexité pour n villes : (n-1)!
  - Tester un sous ensemble suffisant de parcours
    - éviter les calculs inutiles
      - Détecter les parcours non optimaux le plus tôt possible
- Éviter les calculs redondants
  - Mémoriser des calculs
    - « programmation dynamique »

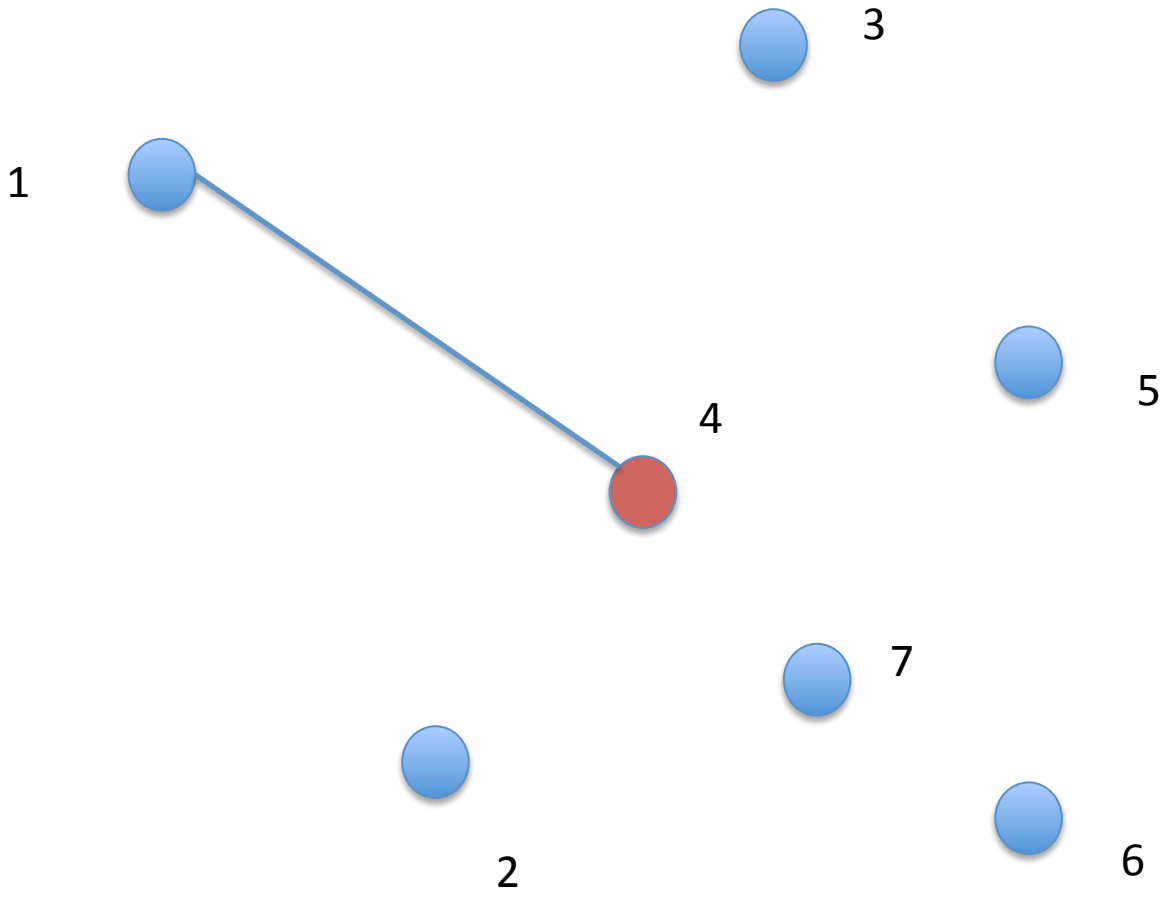
Le voyageur de commerce

- Méthodes approchées
  - Le résultat obtenu peut ne pas être optimal
- Algorithmes gloutons
  - Procéder étape par étape
    - Ne pas remettre en question les décisions prises
  - Faire le meilleur choix possible selon un critère simple (heuristique)
    - Chercher à optimiser localement la solution en construction
- Exemples
  - Se rendre à la plus proche ville non visitée
  - Insérer successivement les villes les unes après les autres

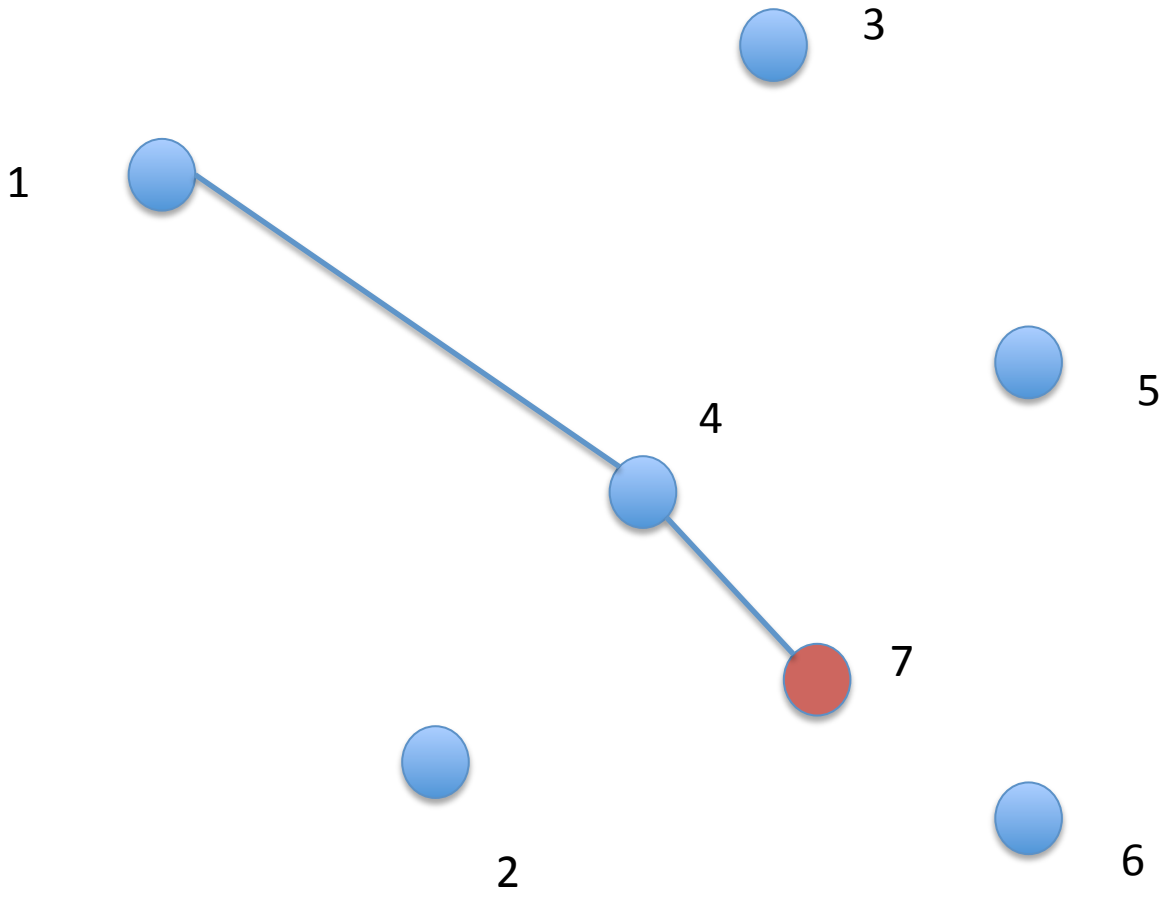
Plus proche voisin



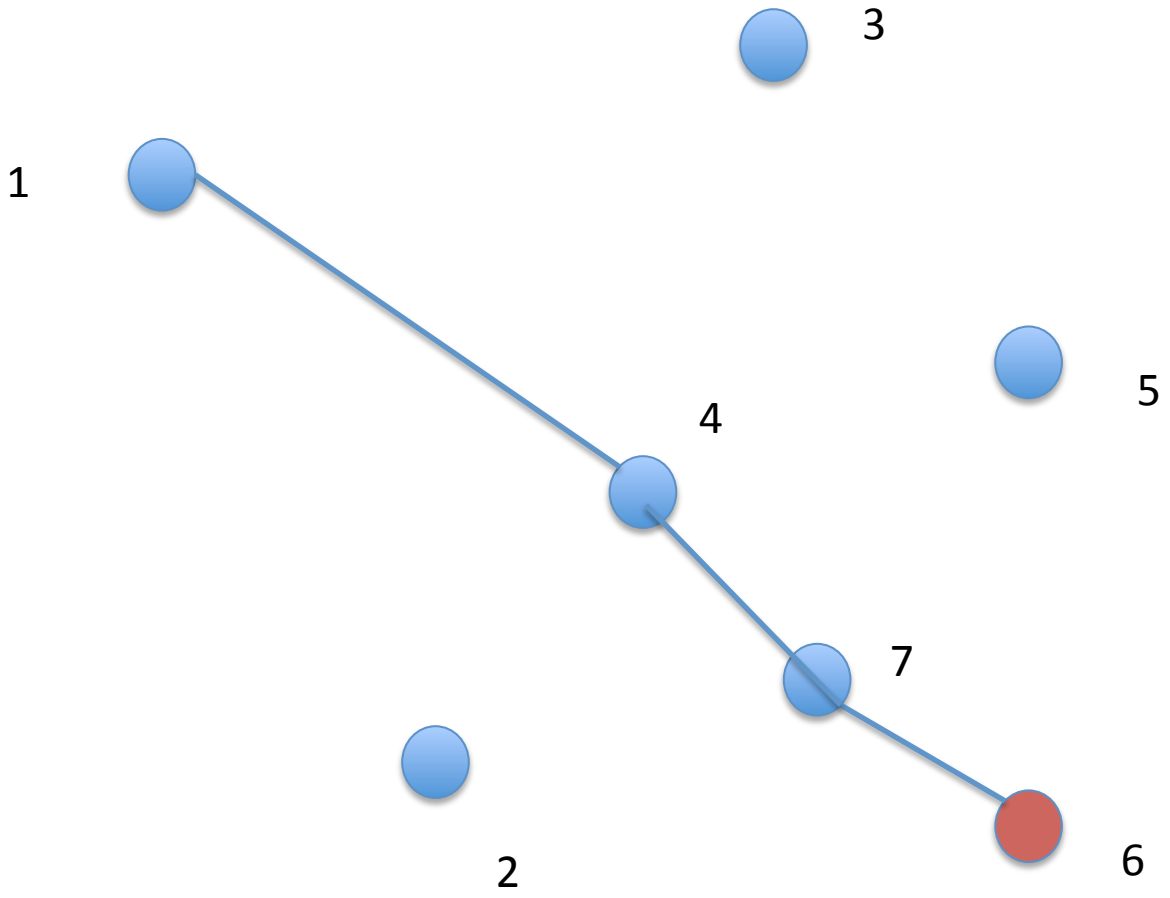
Plus proche voisin



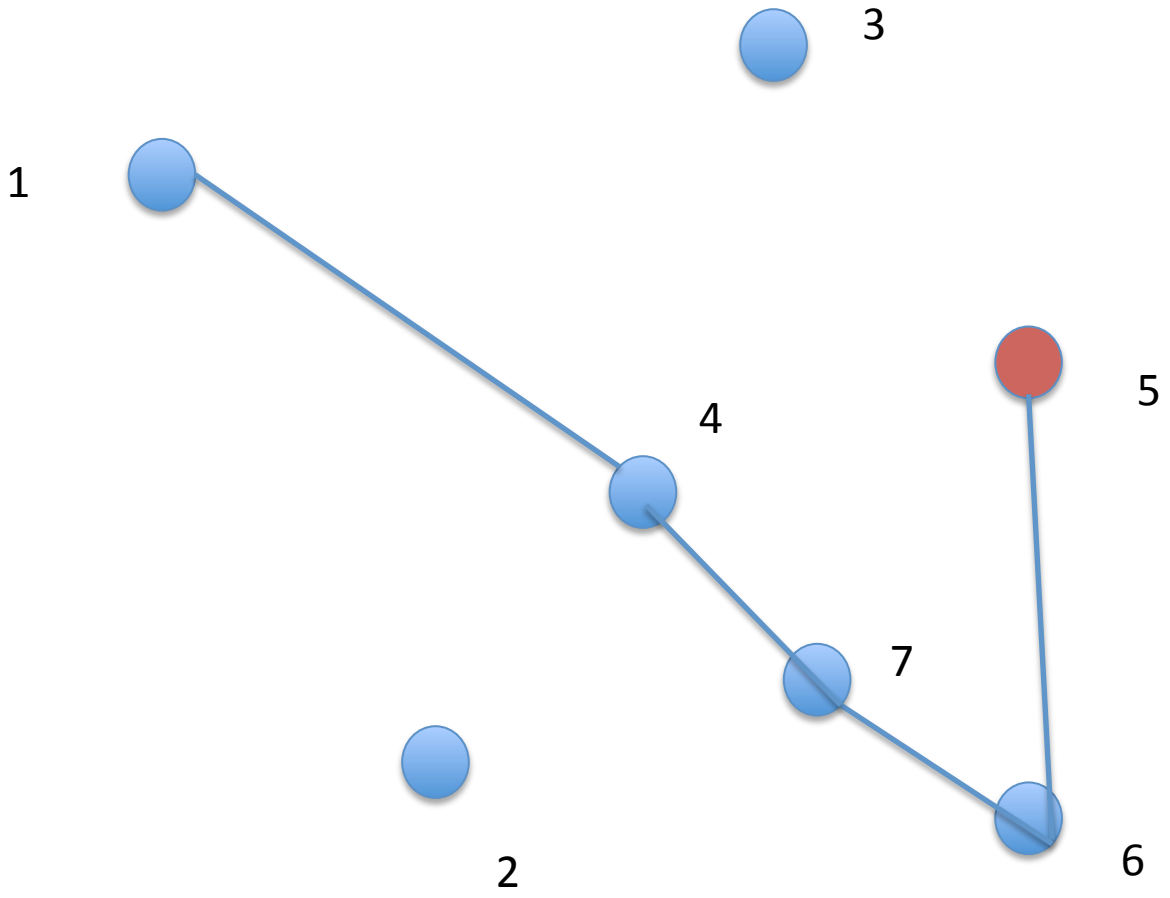
Plus proche voisin



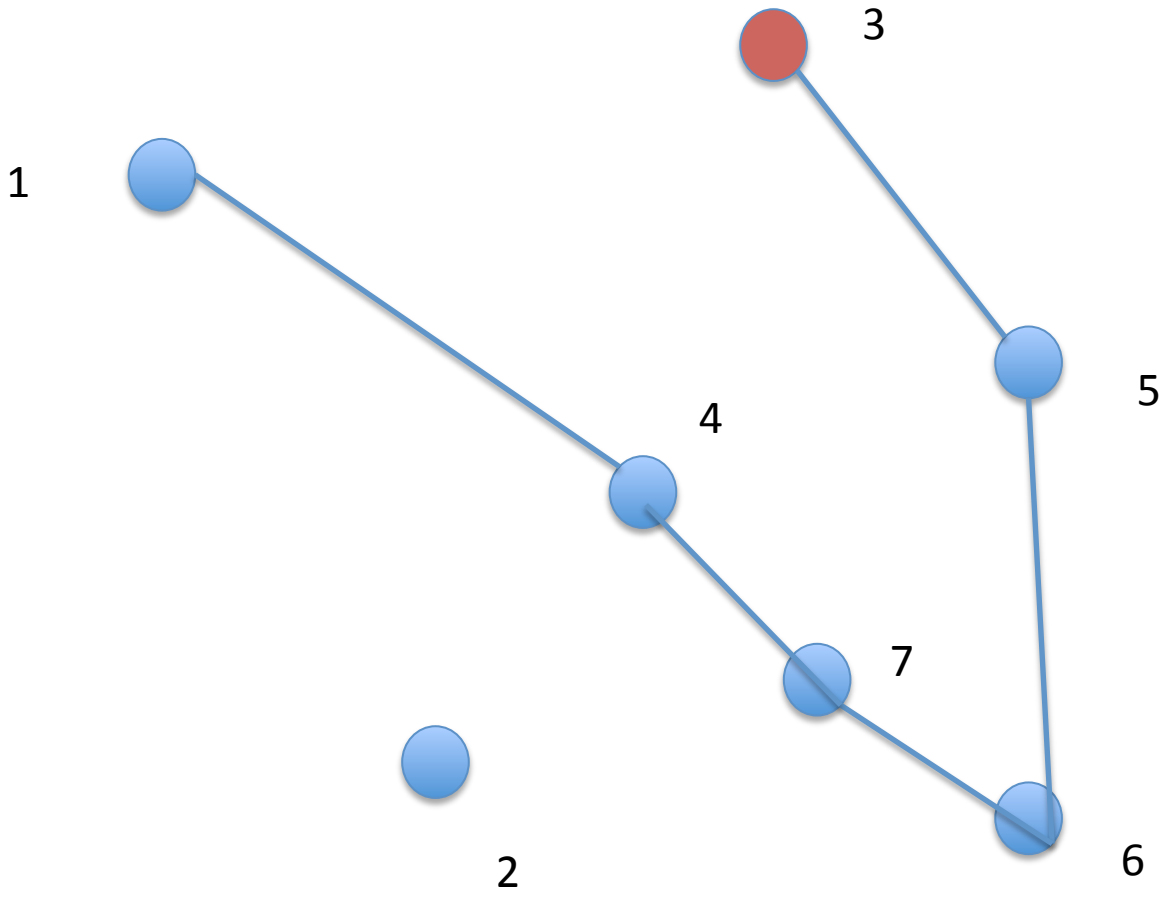
Plus proche voisin



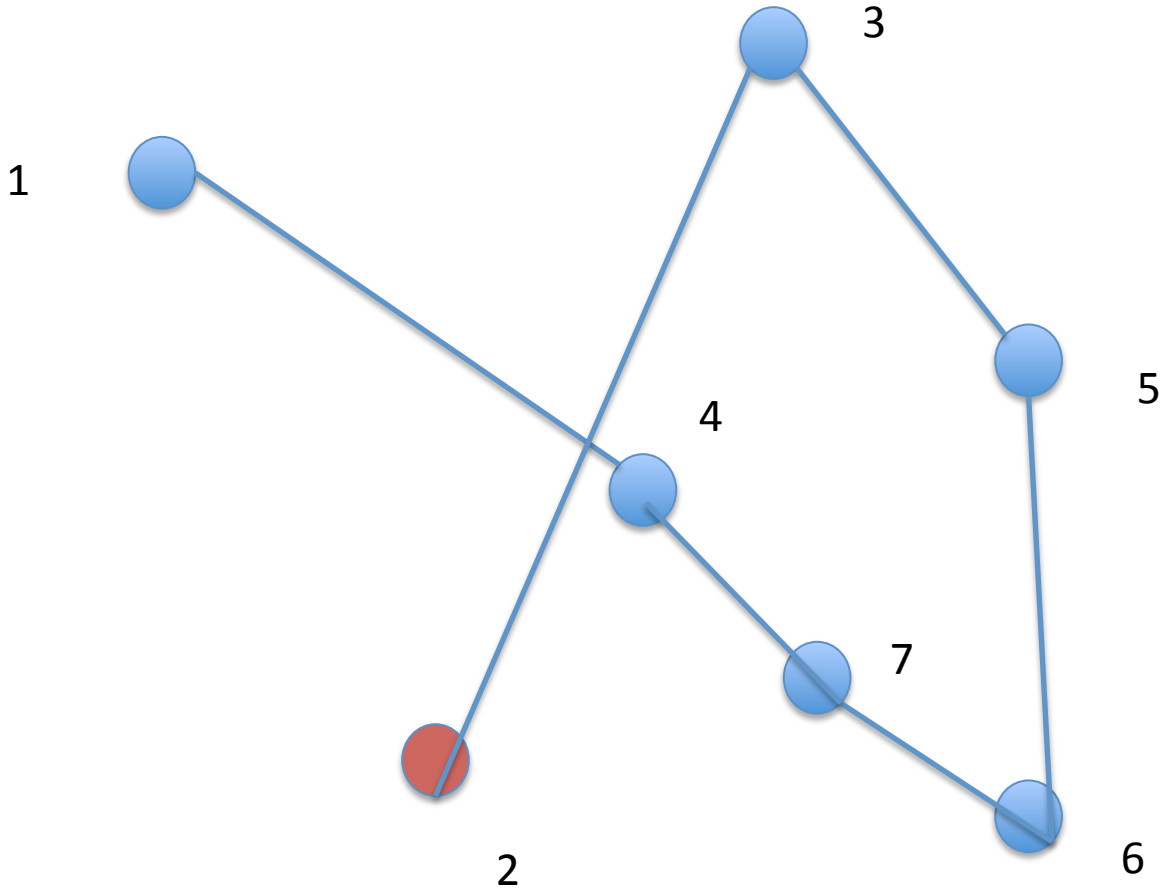
Plus proche voisin



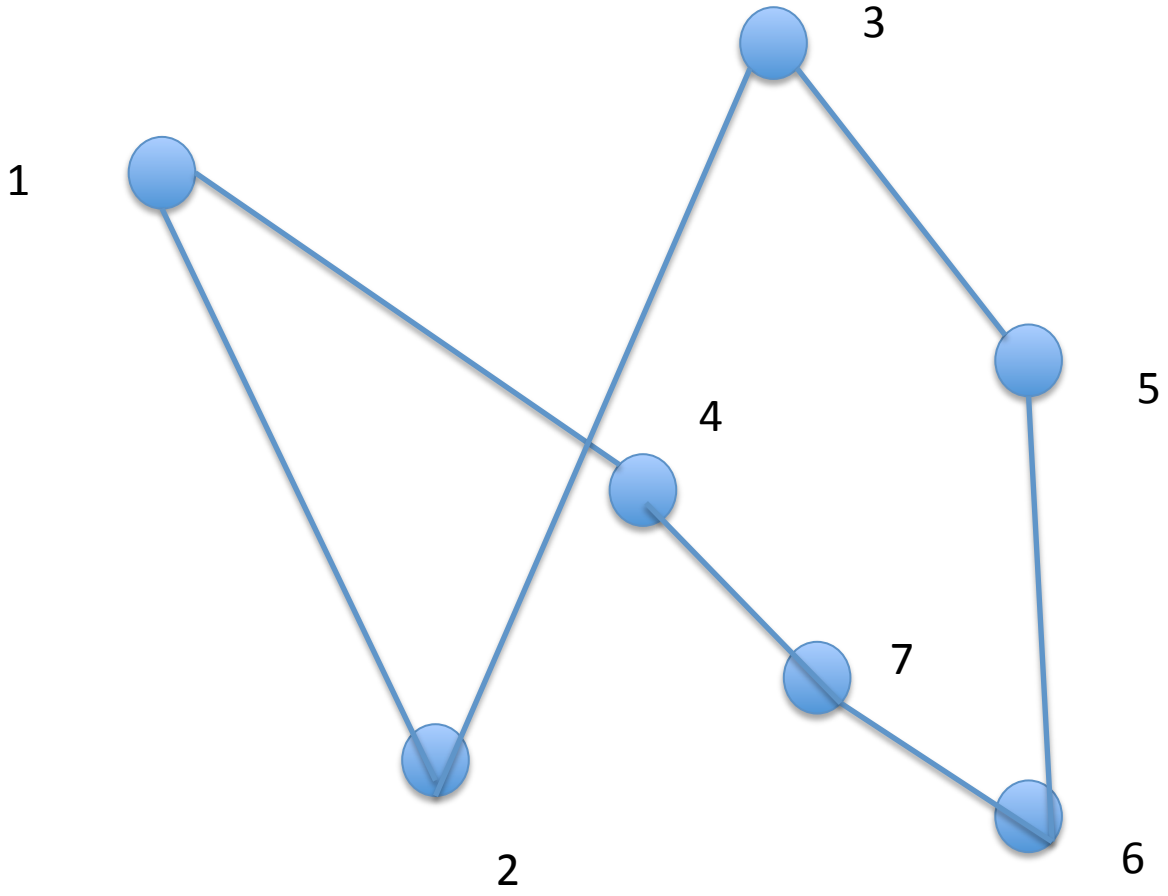
Plus proche voisin



Plus proche voisin

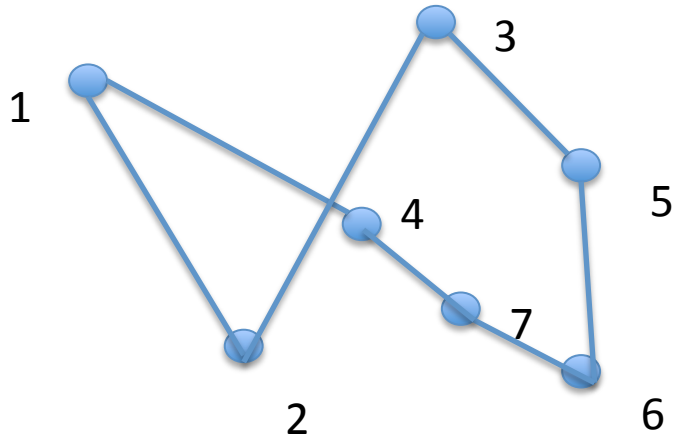


Plus proche voisin



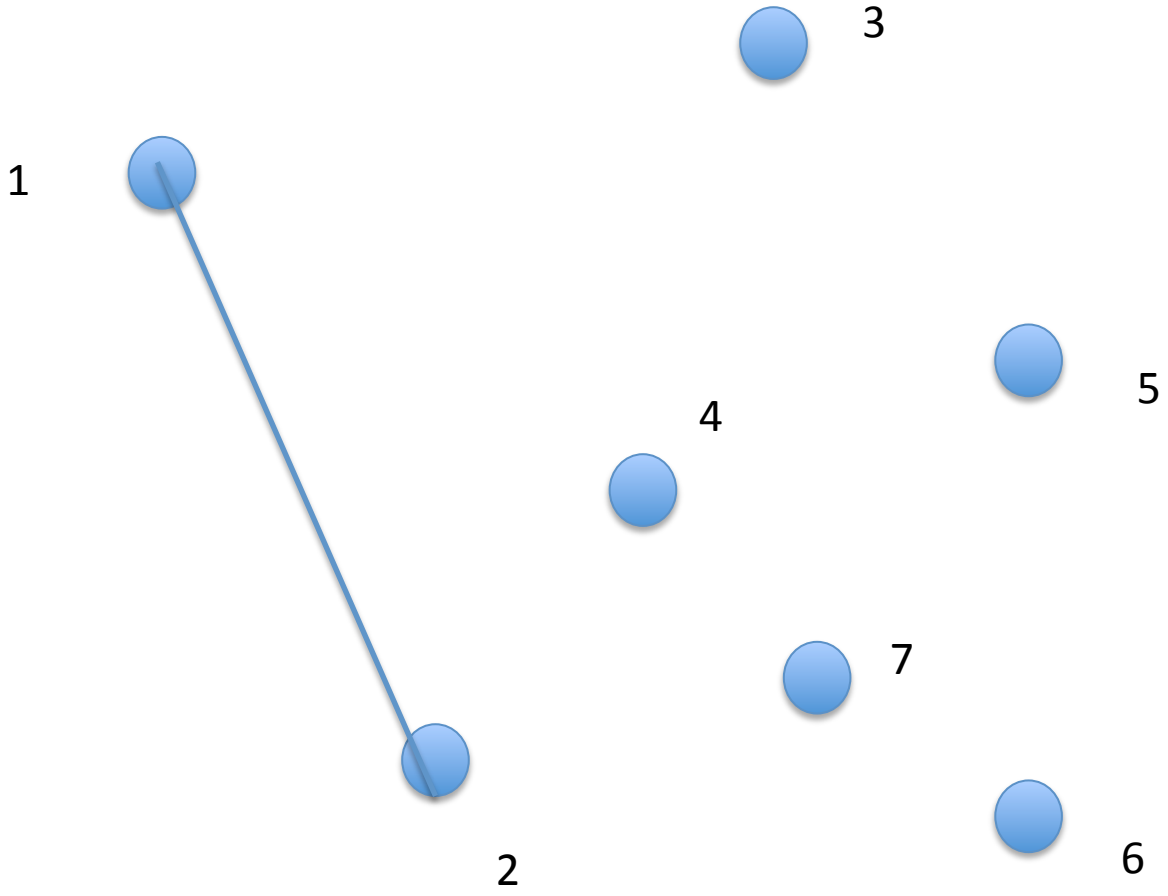
Plus proche voisin

- Pas optimal
  - Possibilité de supprimer les croisements
    - Lorsque l'inégalité triangulaire est respectée

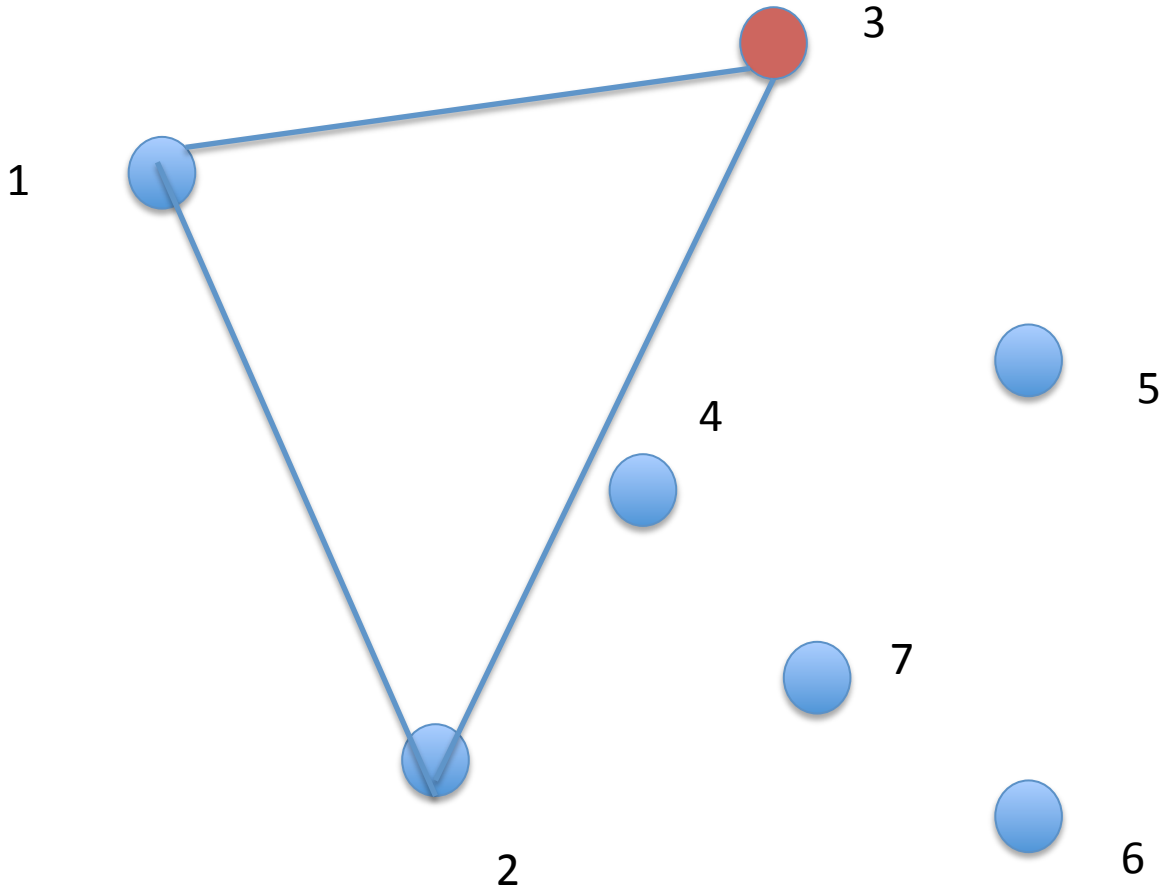


    - D(1,4) + D(3,2) > D(2,4) + D(1,3)
  - Il faut vérifier toutes les paires d'arêtes

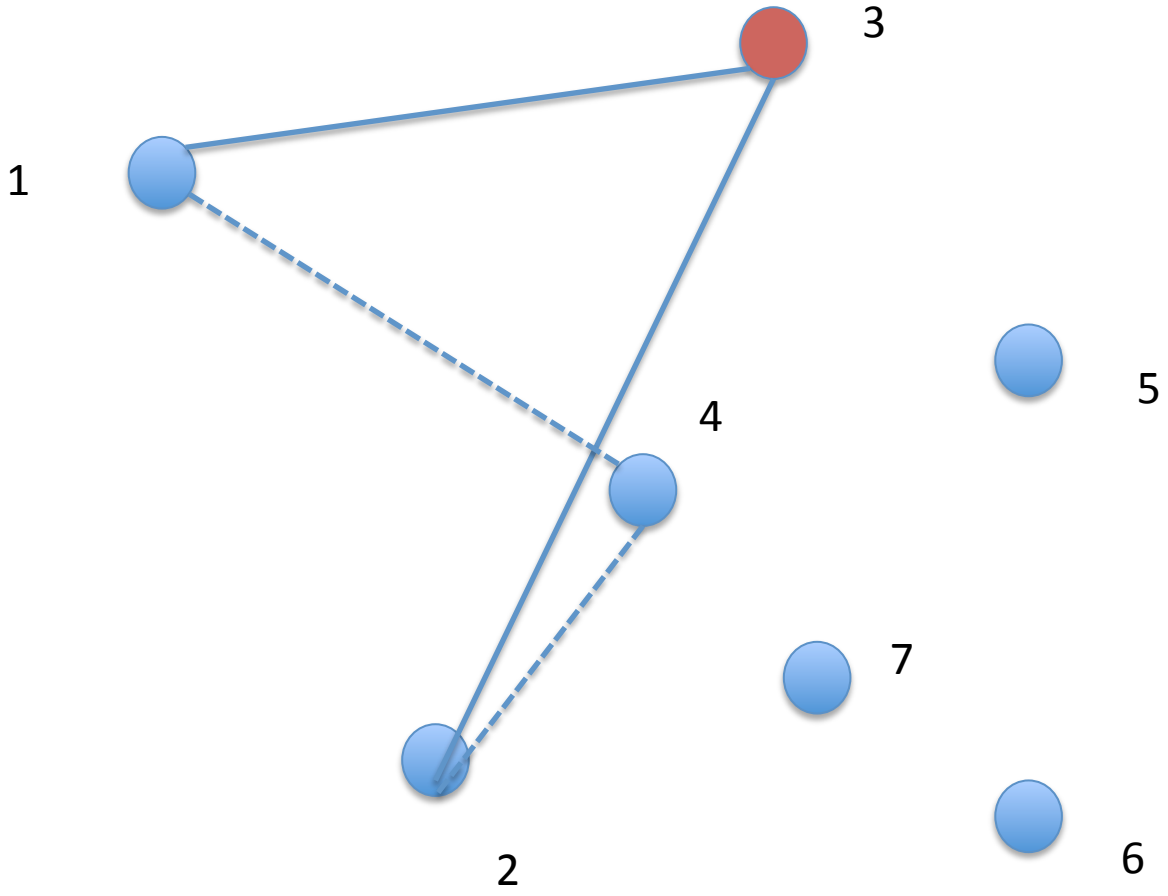
Insertion



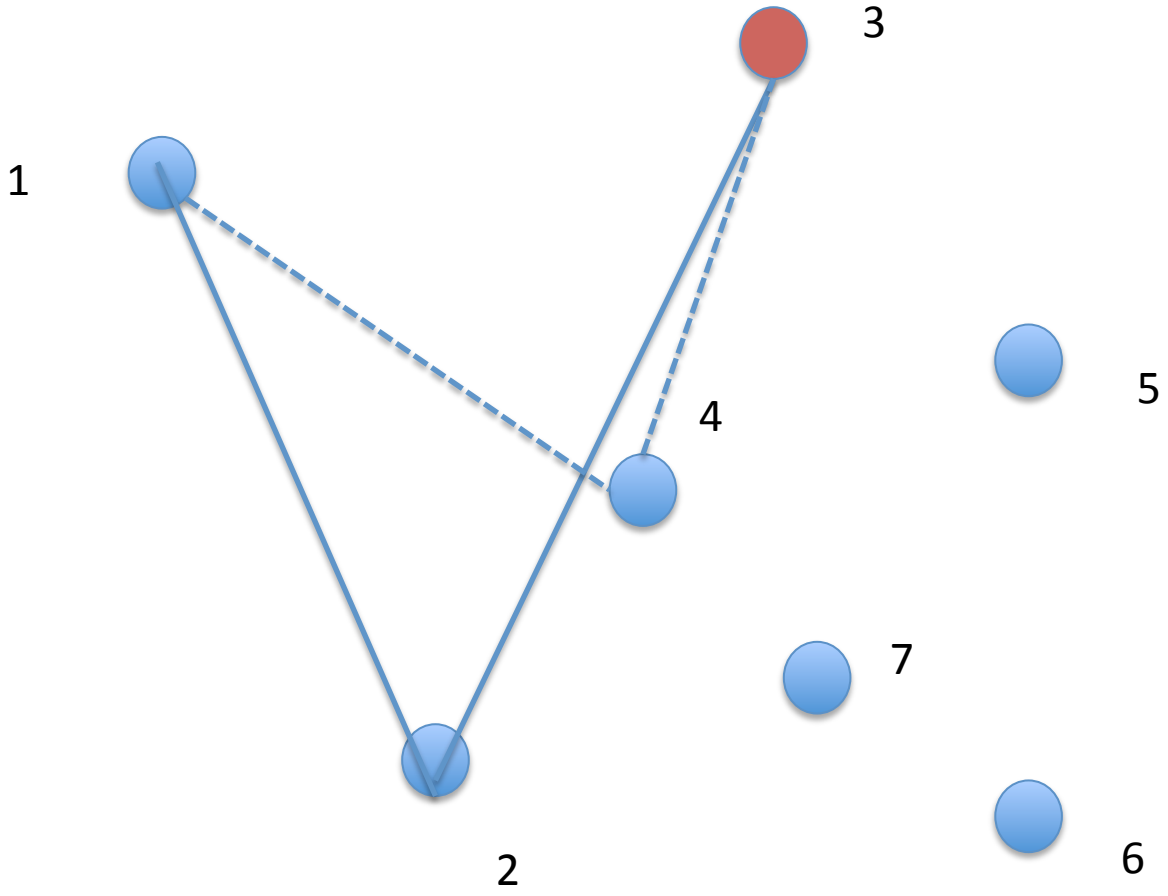
Insertion



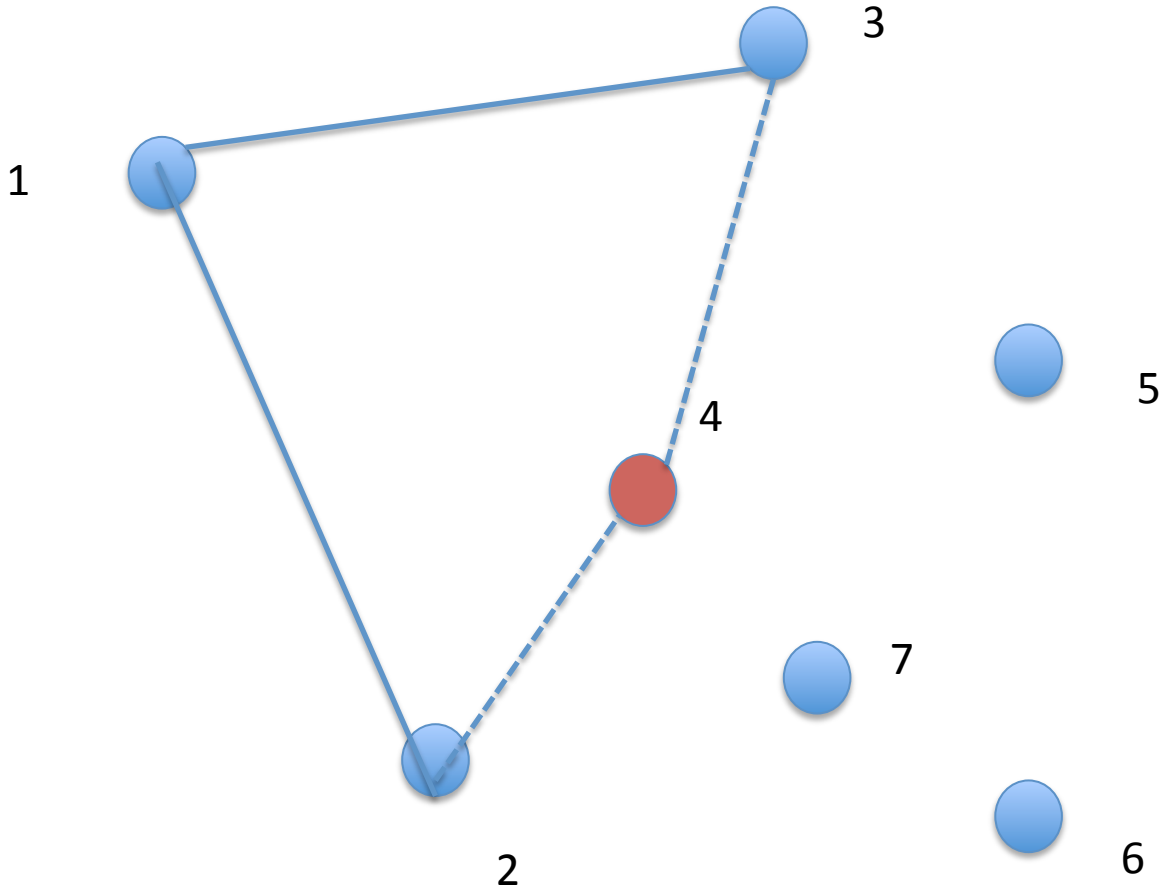
Insertion



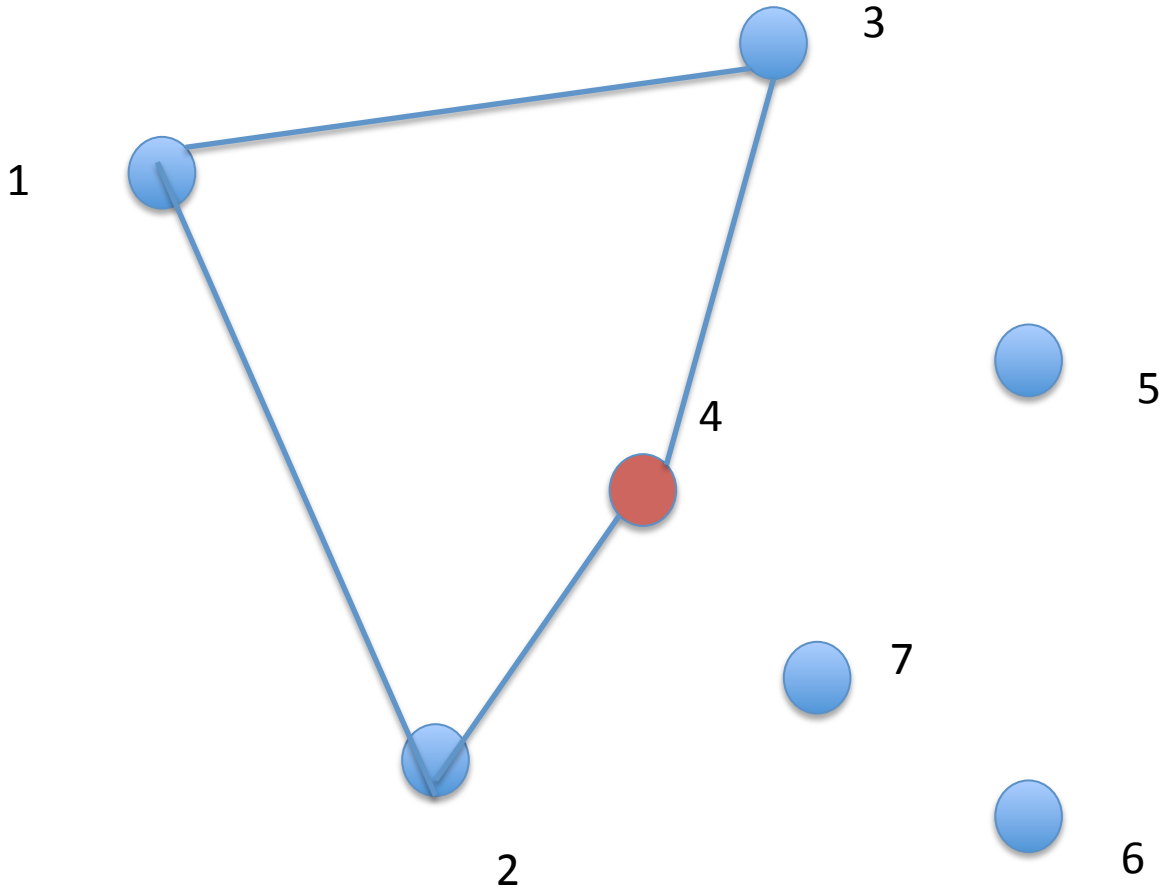
Insertion



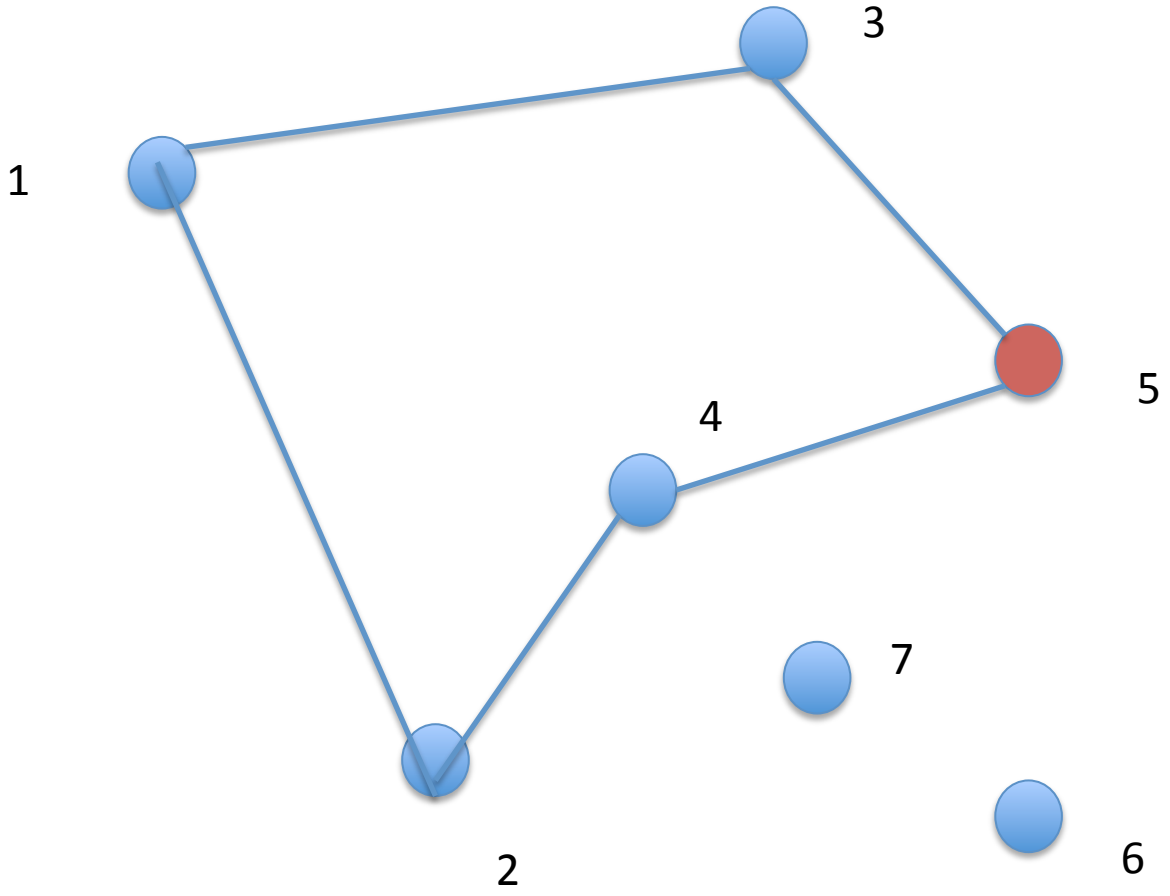
Insertion



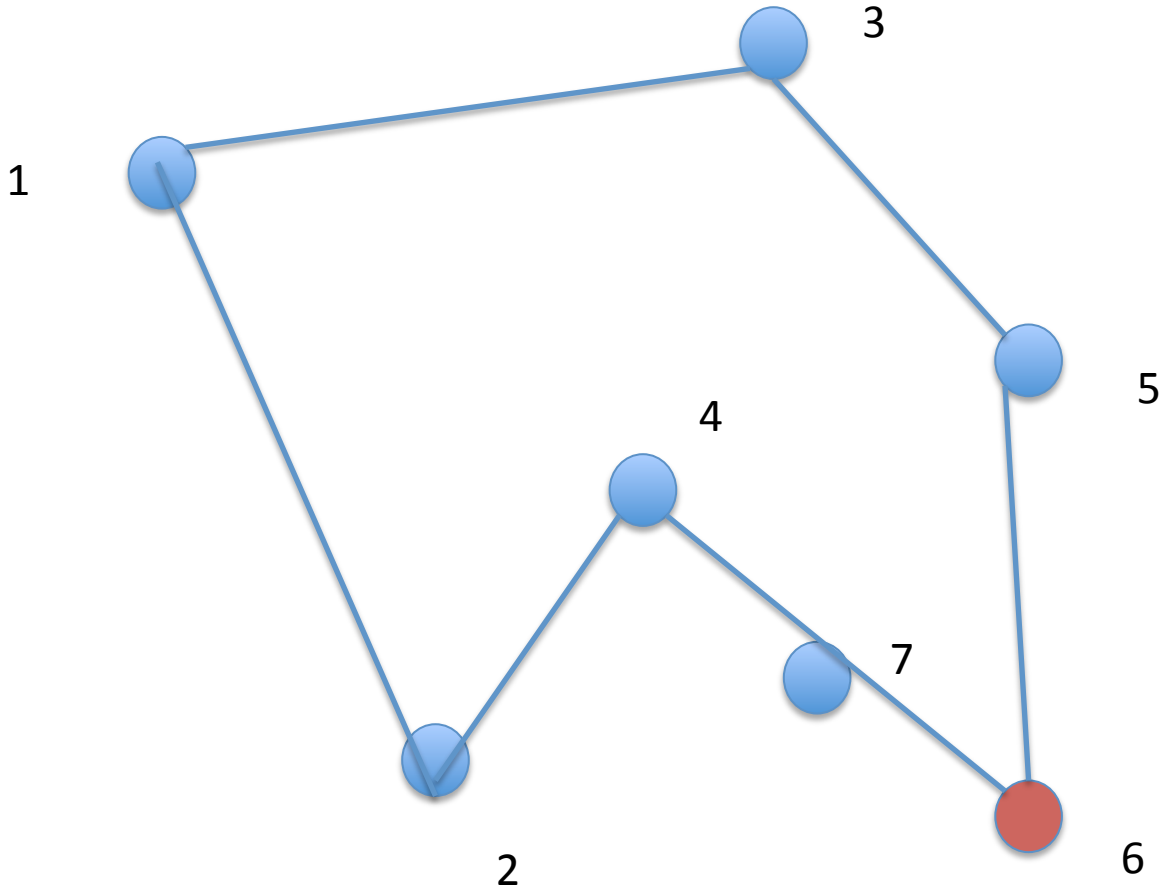
Insertion



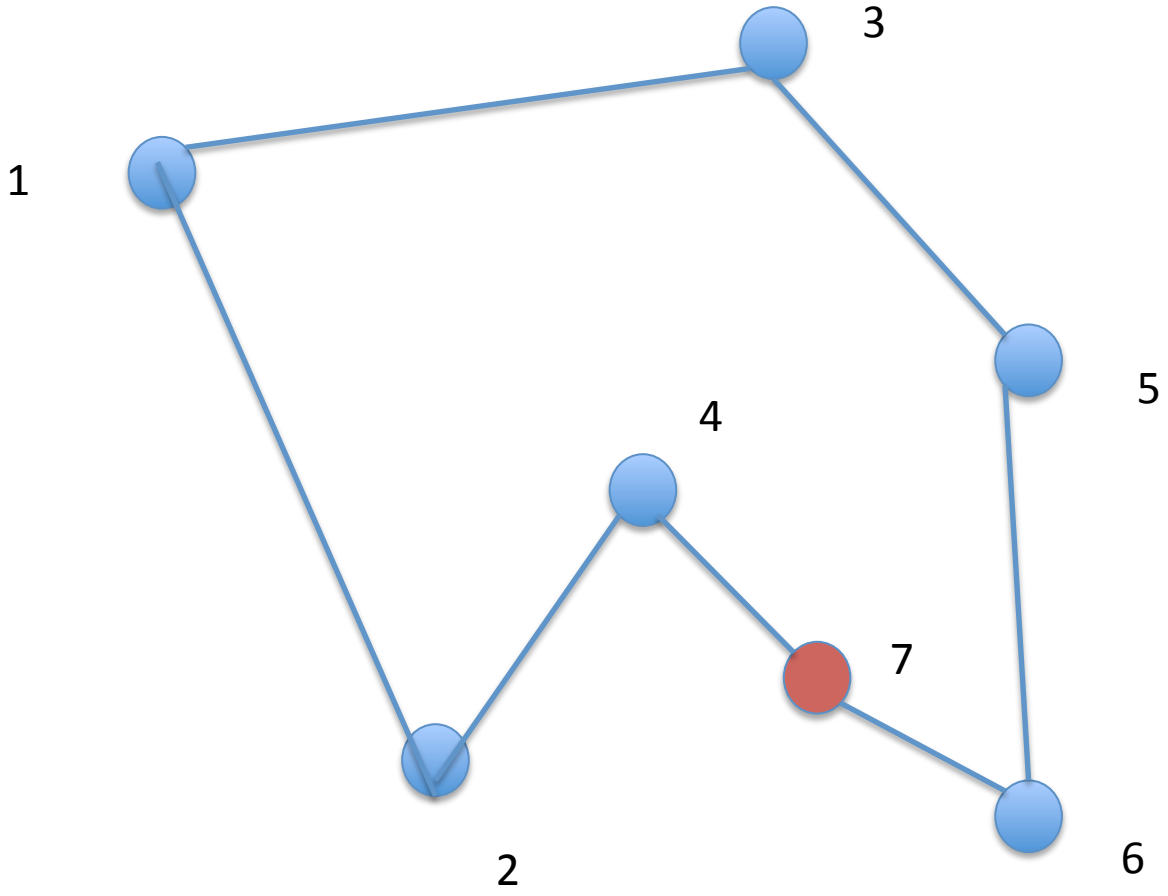
Insertion



Insertion



Insertion



Insertion

- Ne produit pas de croisement
- Possibilité de combiner les deux méthodes :
  - Insertion du plus proche voisin

- Recuit simulé (voir interstice)
  - Recuit : réchauffement suivi d'un refroidissement lent
  - On chauffe en provoquant des croisements aléatoirement
  - On refroidit en « réparant » les croisements

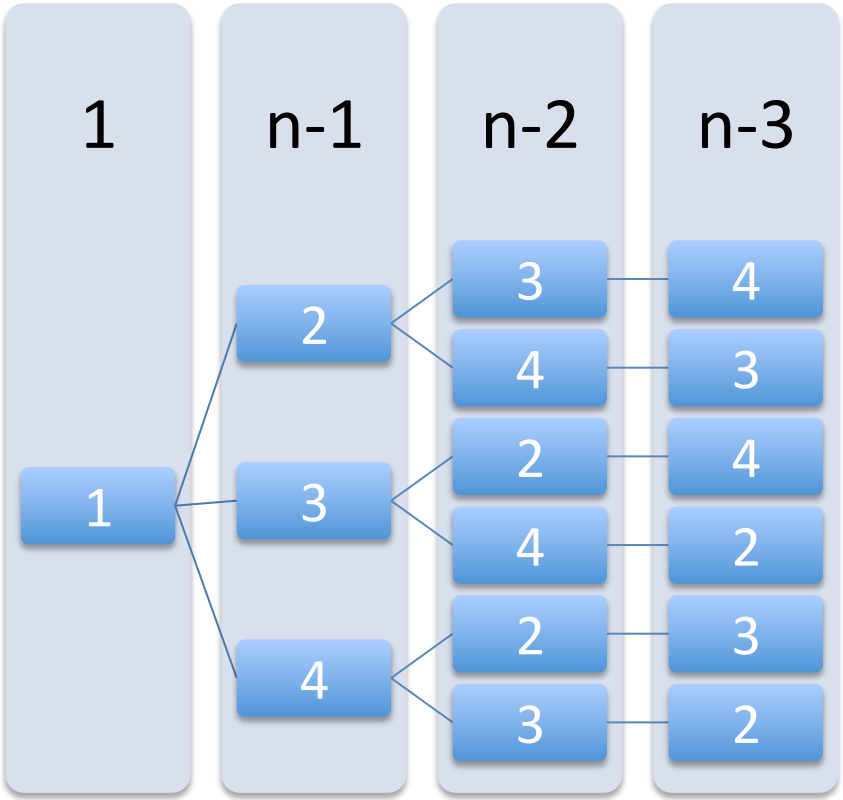
Objectifs 1ère étape

- TD1
  - 1 respect des consignes
  - 2 compte rendu
  - 3 algorithmes gloutons
  - 2 étude récursif
  - 2 qualité du code

- Pas de point pour la modularité, ni pour les entrées / sorties
- Partir de quelques matrices de distances données en dur dans le code.
- Bien préparer l'étude sur le récursif

Approche exhaustive

- La récursivité une technique pour parcourir les arborescences



Approche exhaustive

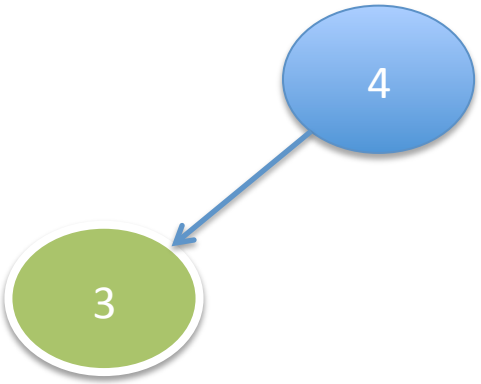
- La récursivité une technique pour parcourir les arborescences

```
int fibo(int n)
{
    If (n < 2) return 1;
    return fib(n-1)+fibo(n-2);
}
```

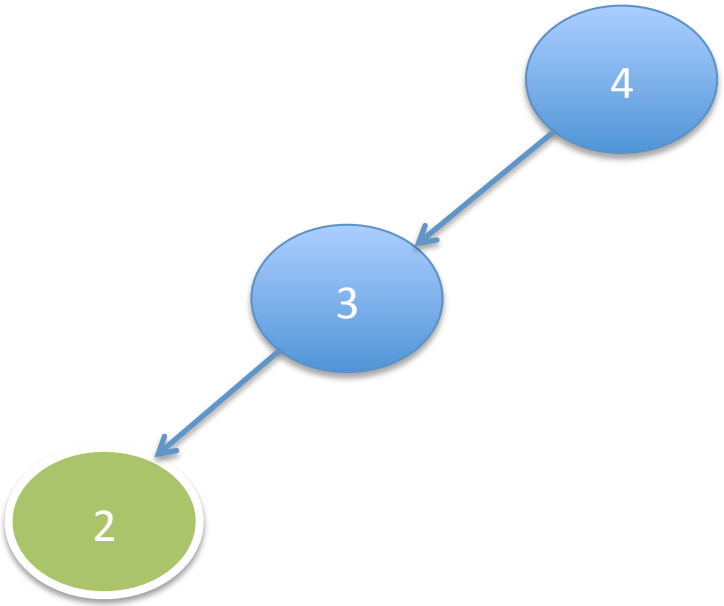
Fibo(4)

Fibo(4)

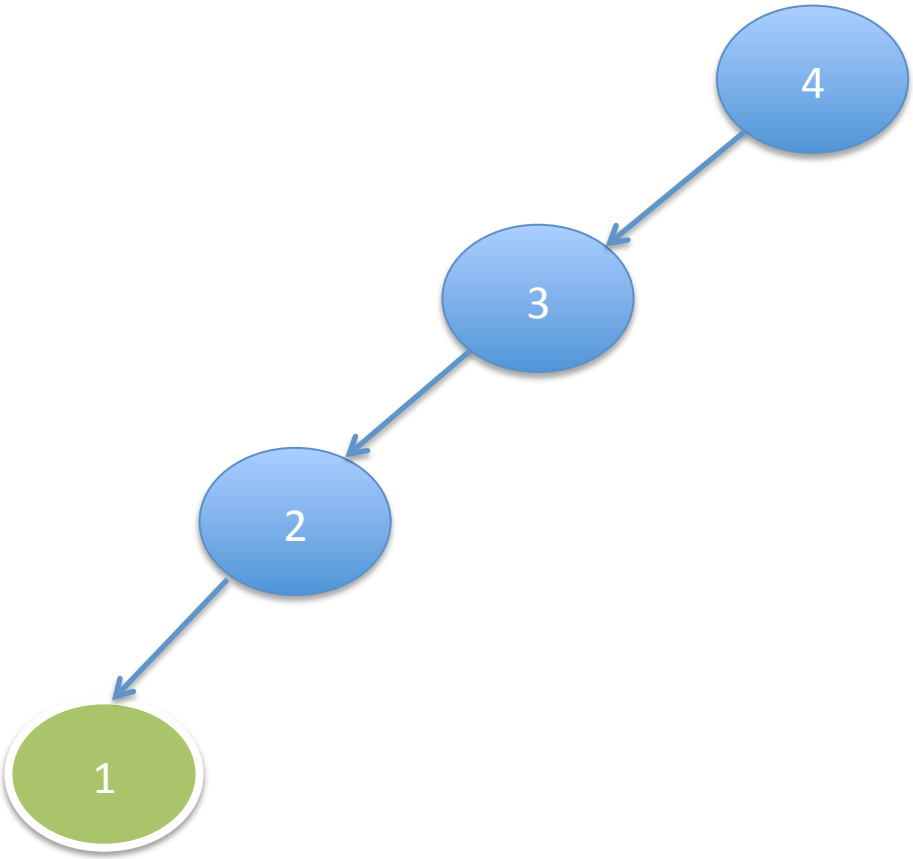
Fibo(4)



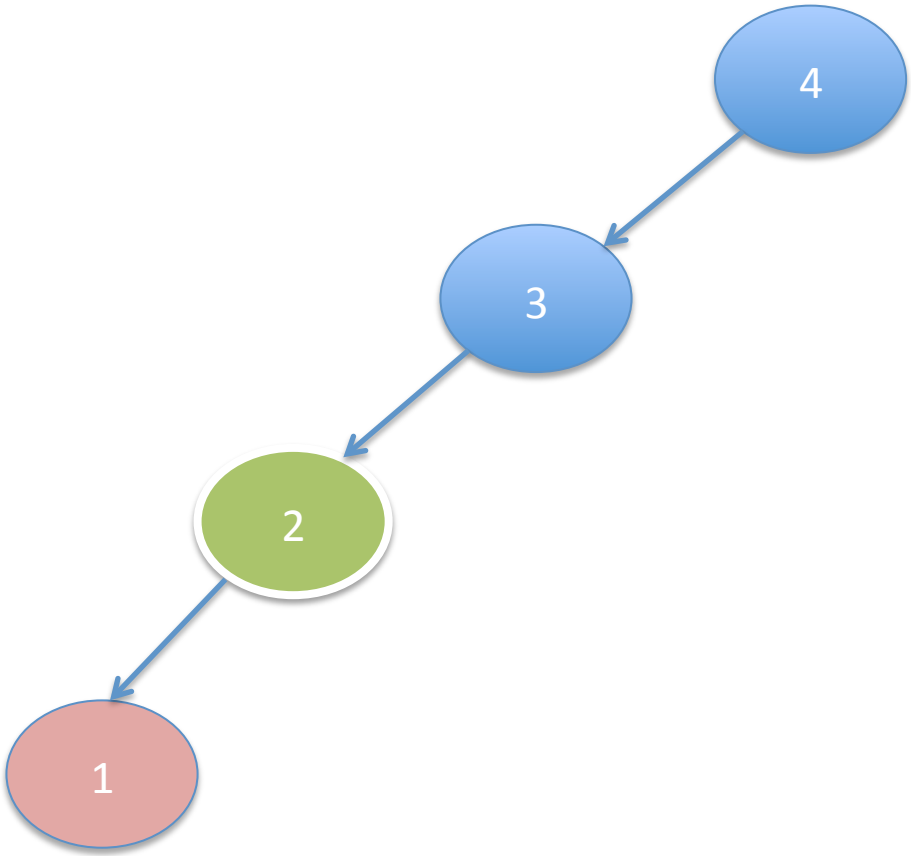
Fibo(4)



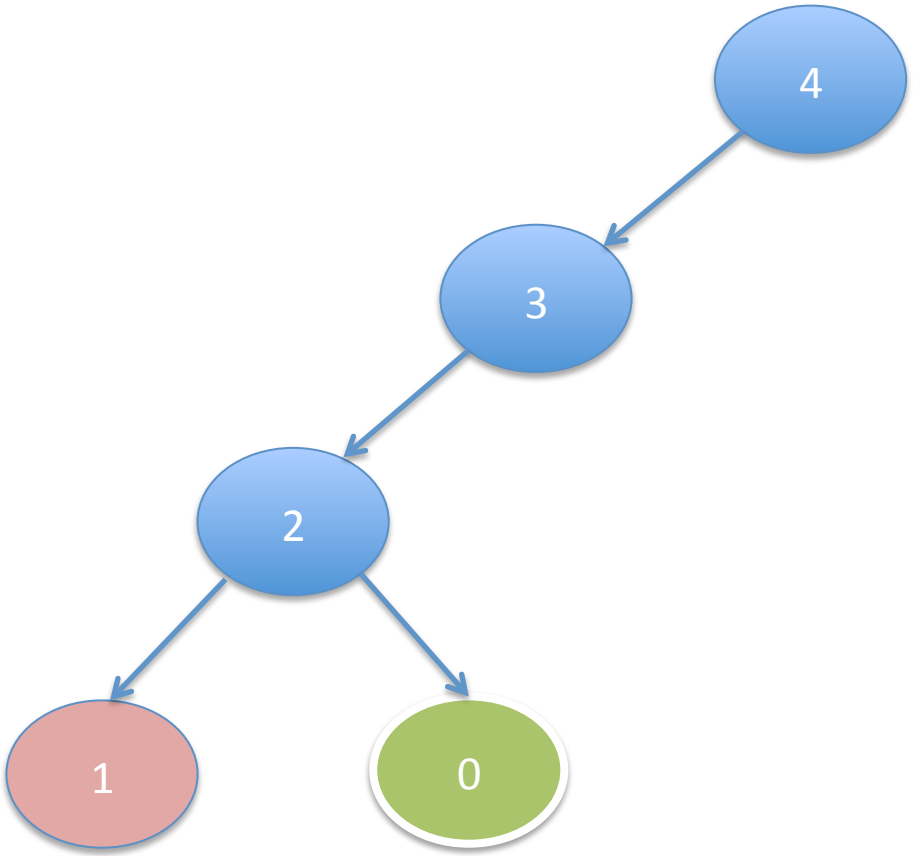
Fibo(4)



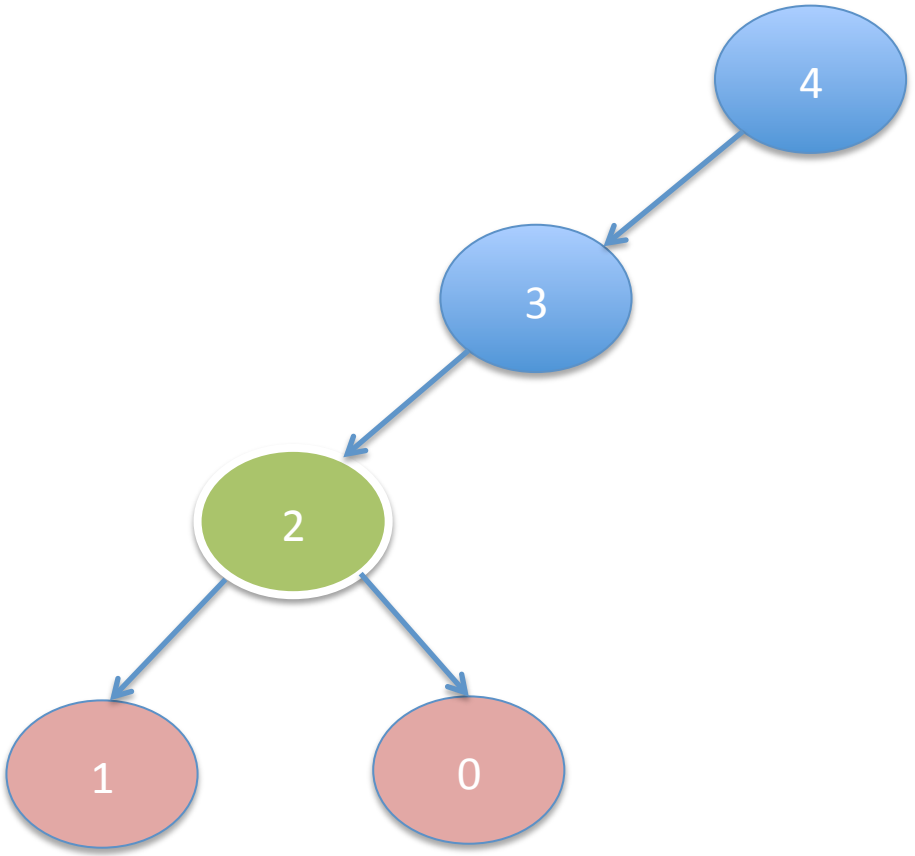
Fibo(4)



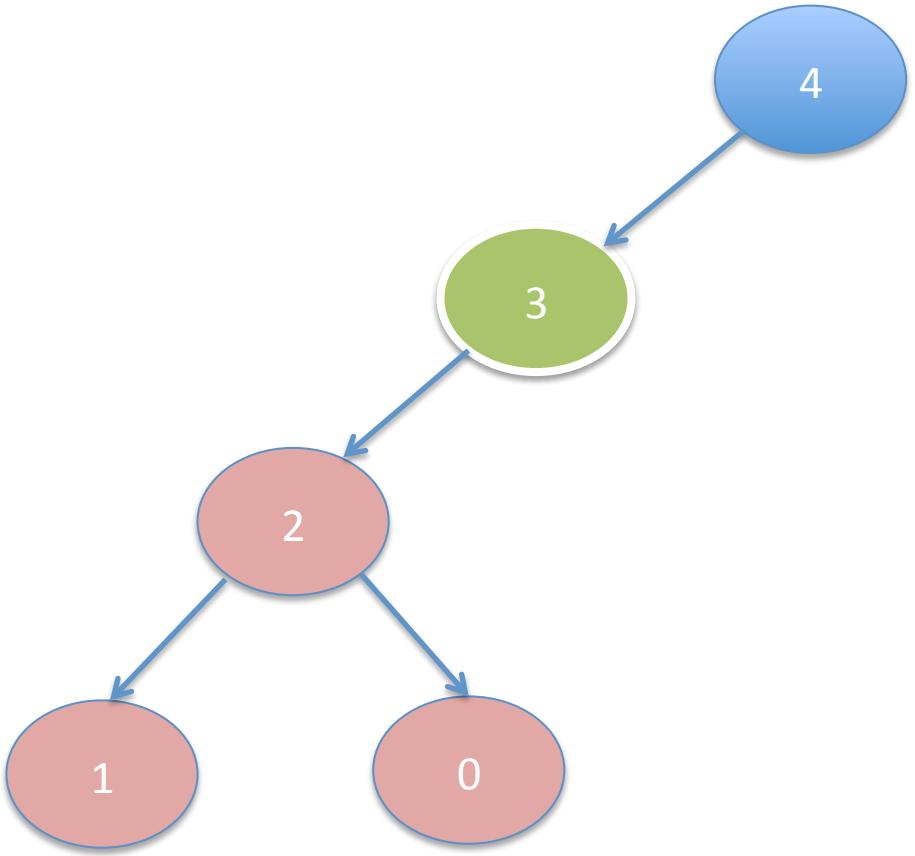
Fibo(4)



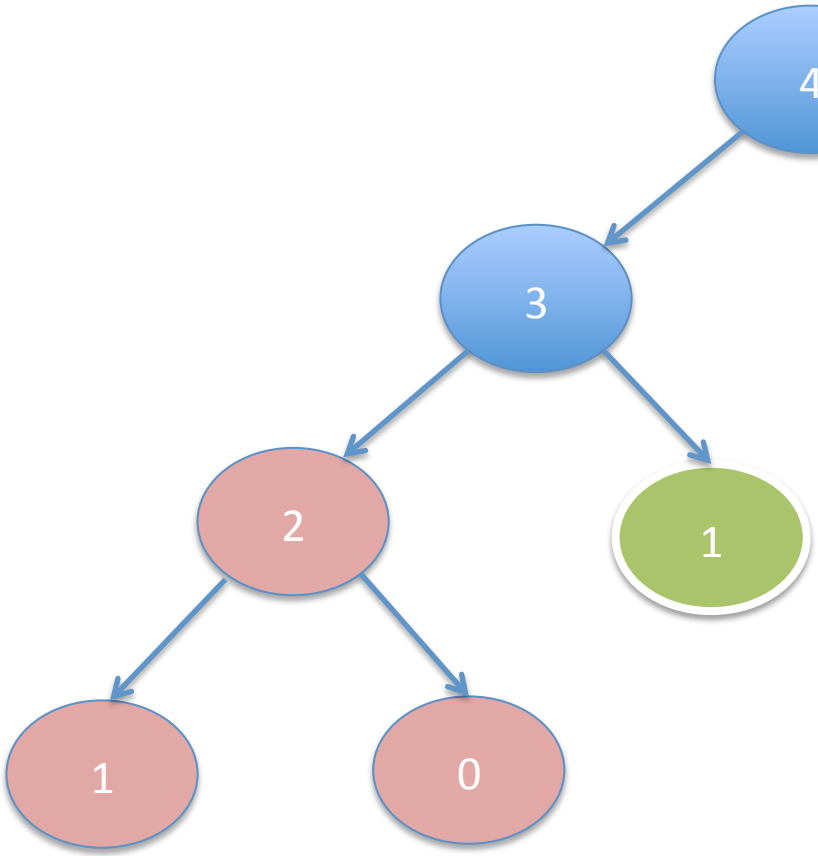
Fibo(4)



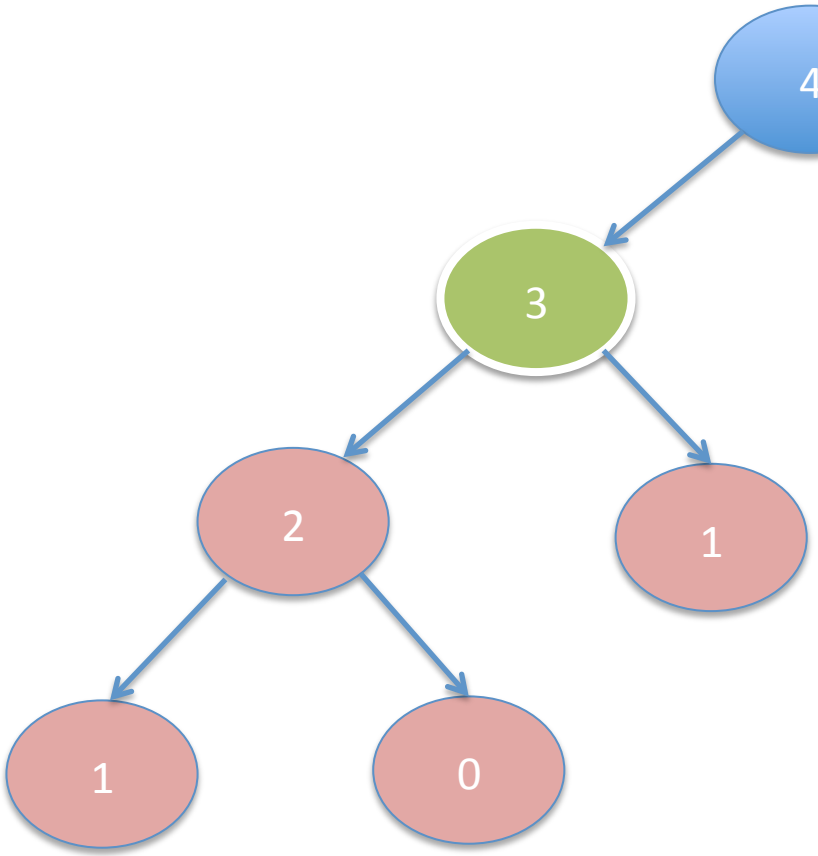
Fibo(4)



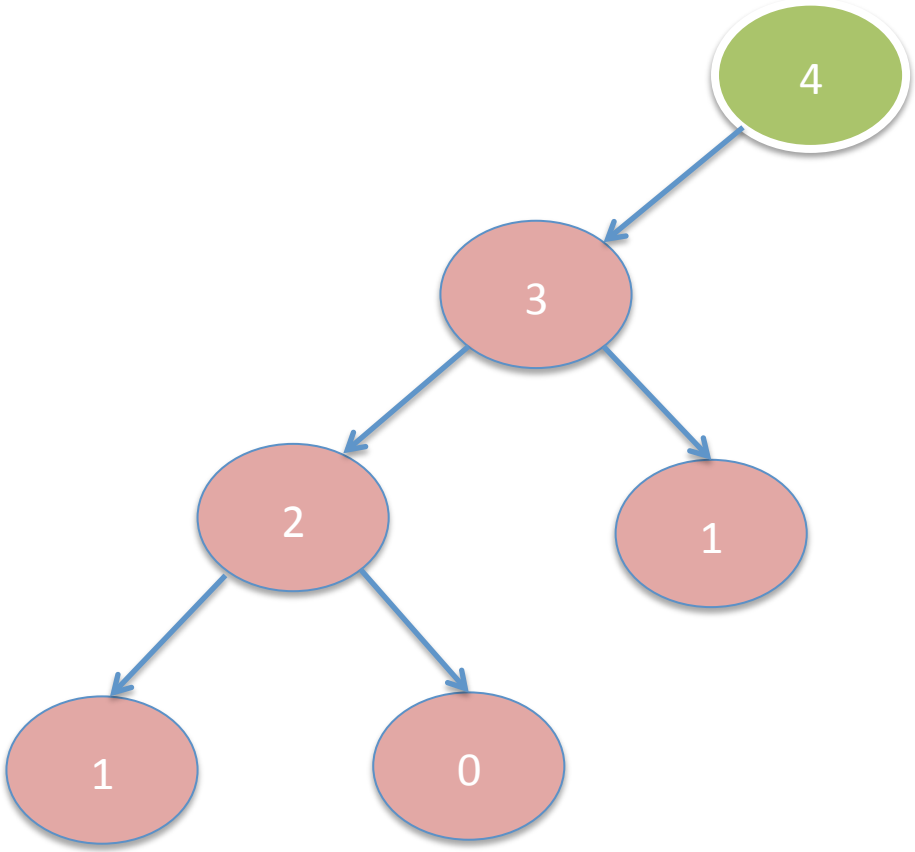
Fibo(4)



Fibo(4)

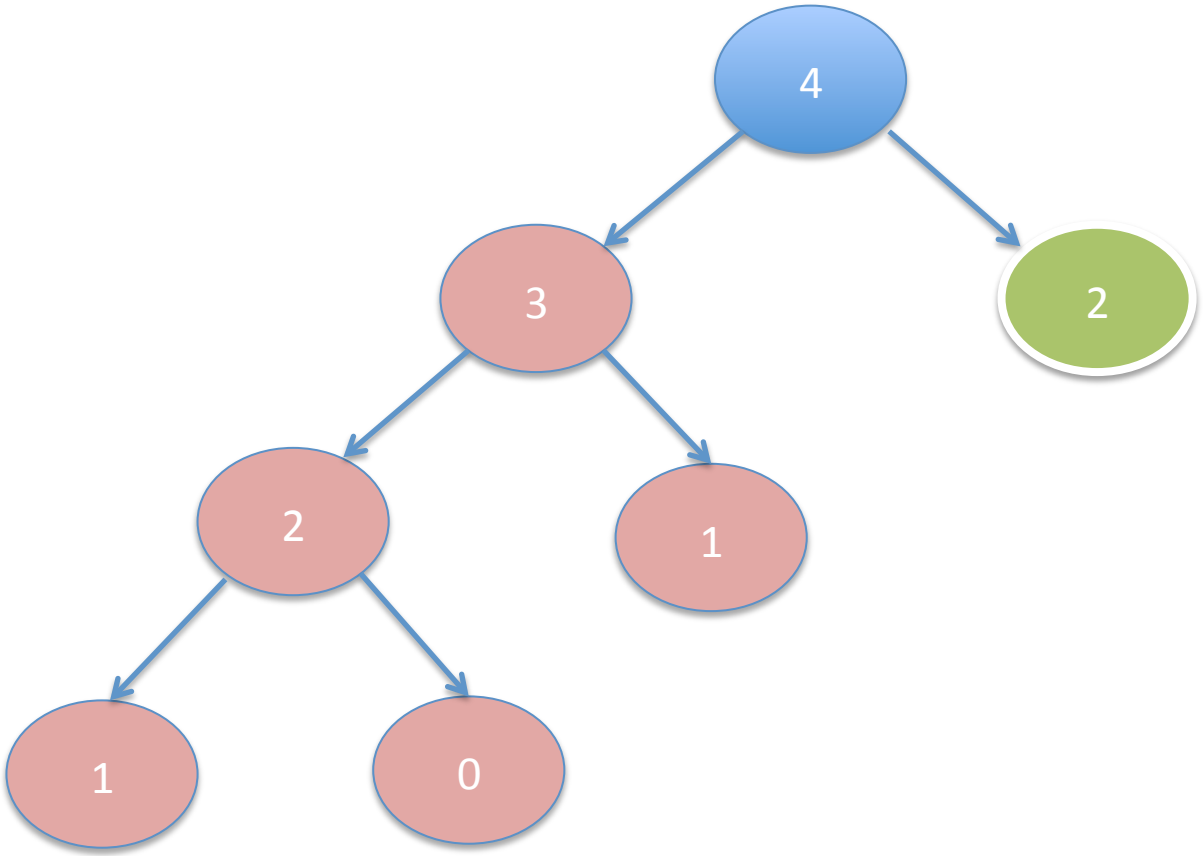


Fibo(4)

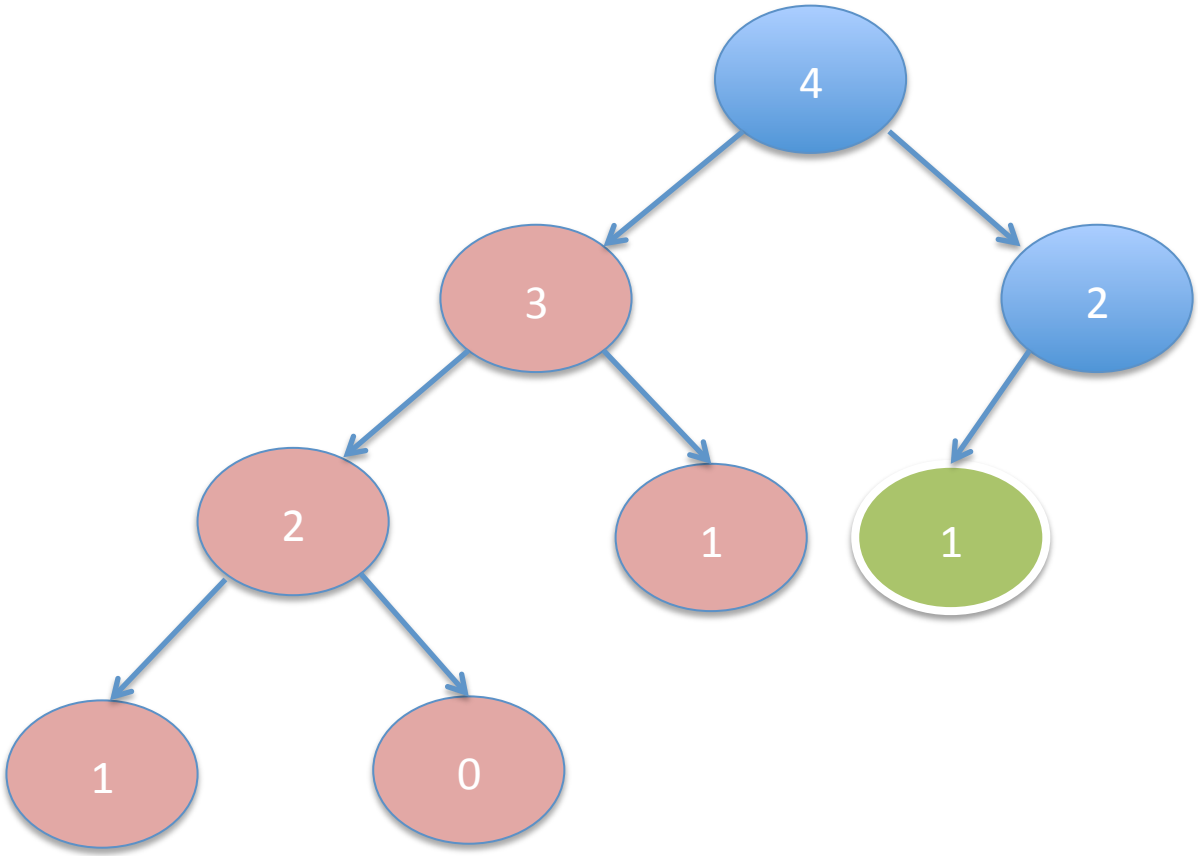


Fibo(4)

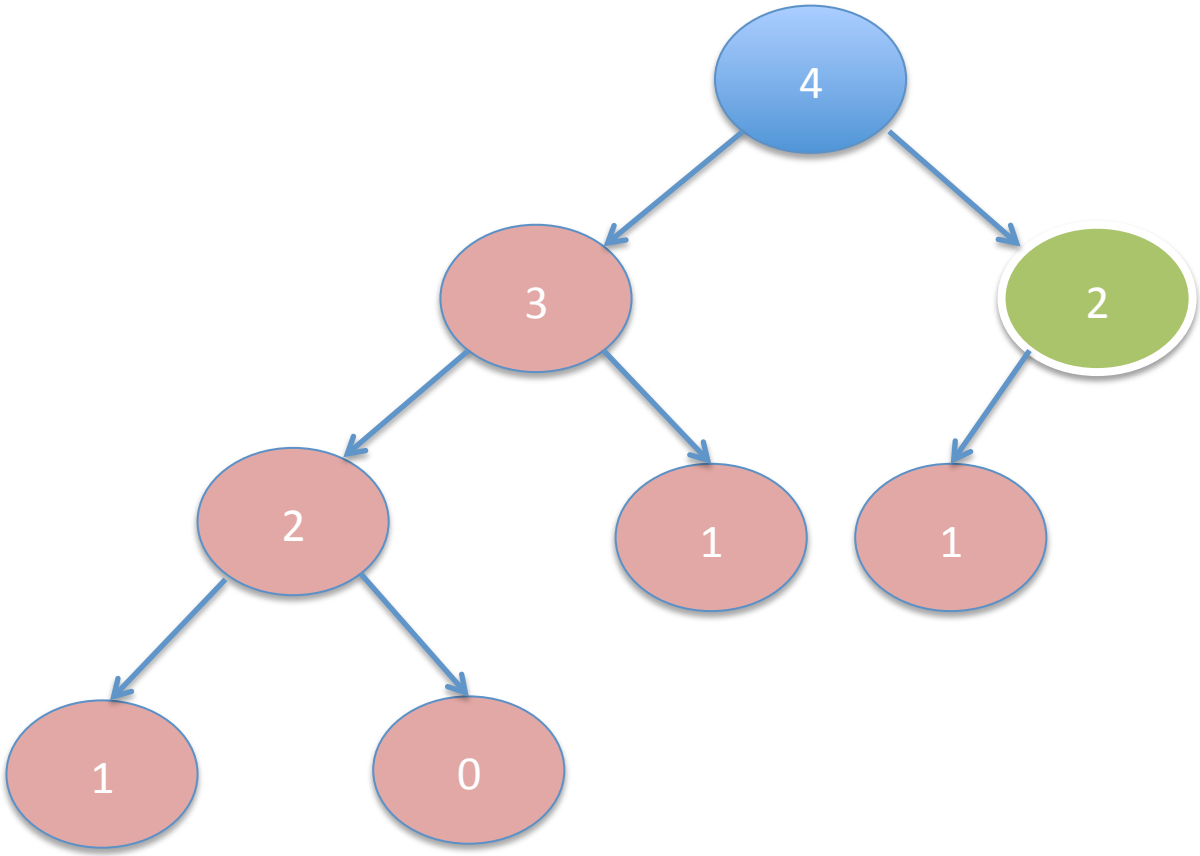
Fibo(4)



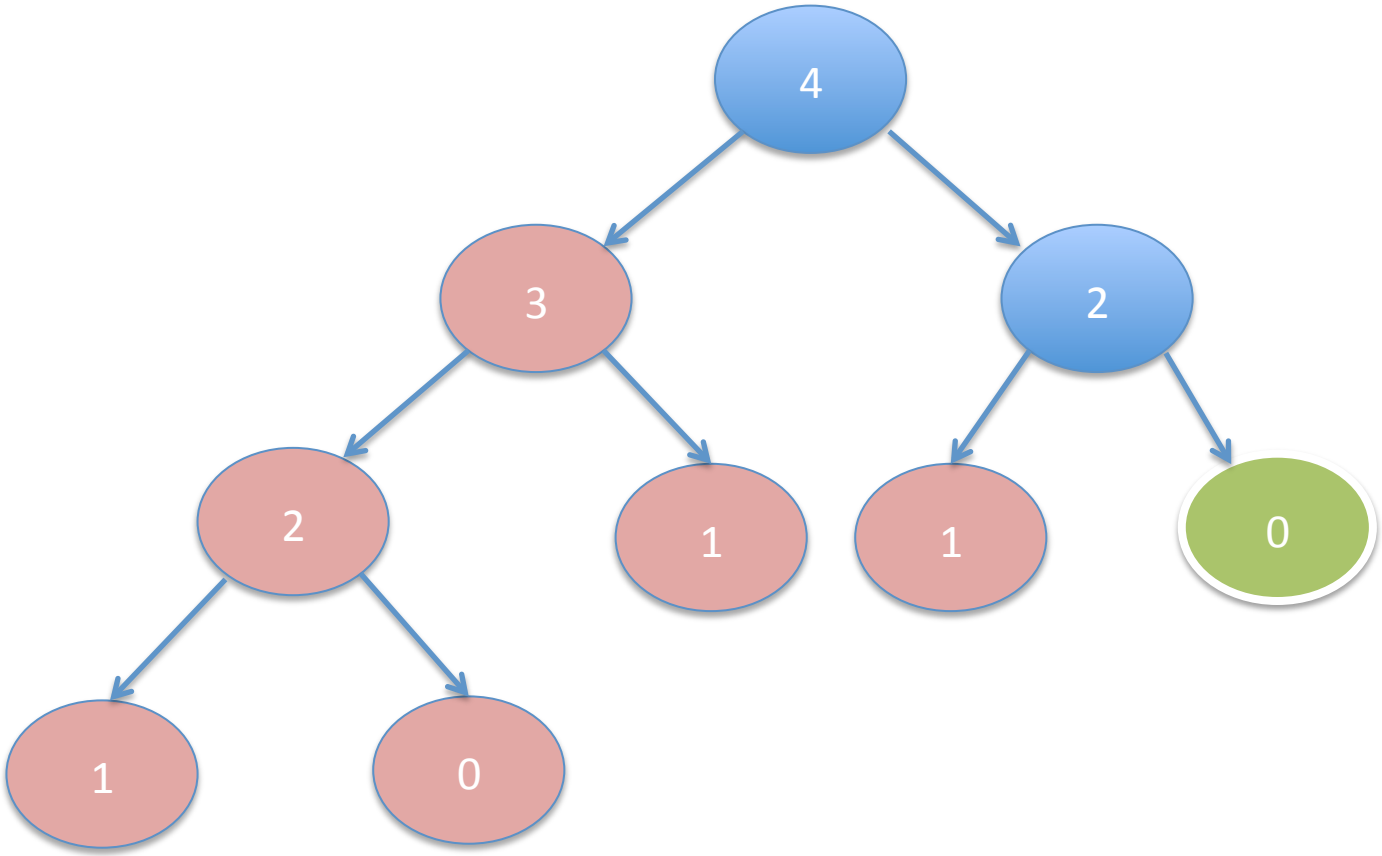
Fibo(4)



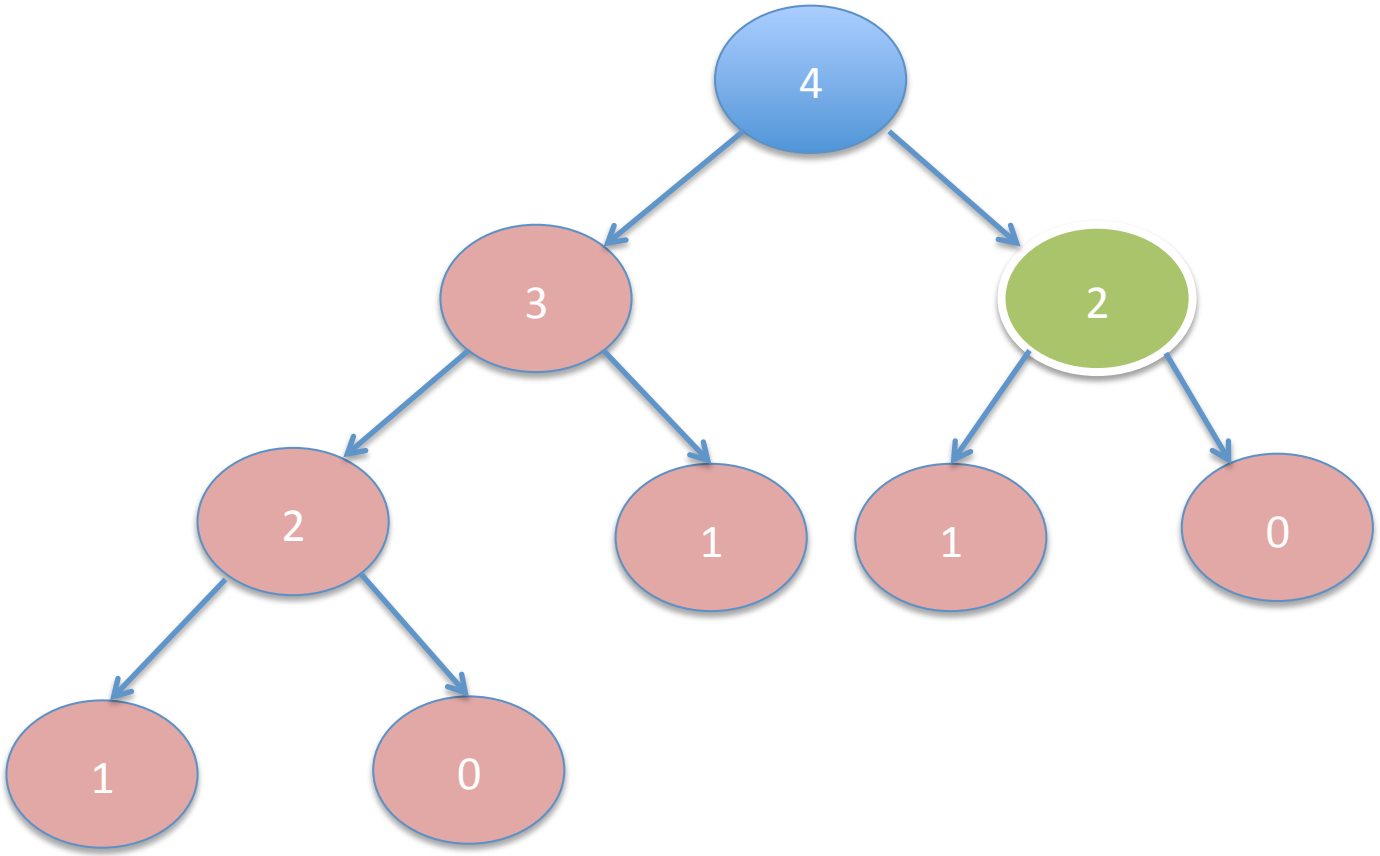
Fibo(4)



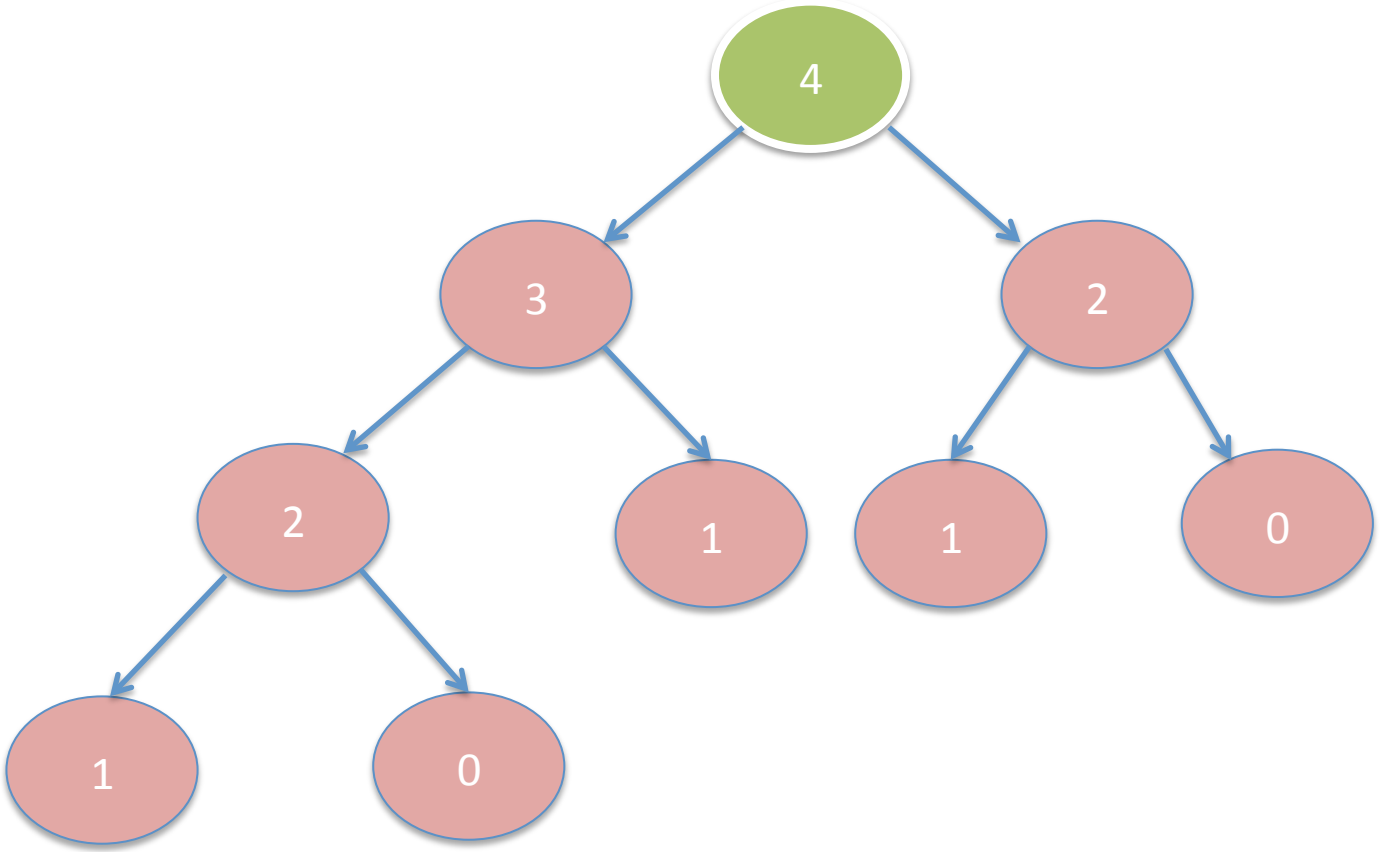
Fibo(4)



Fibo(4)

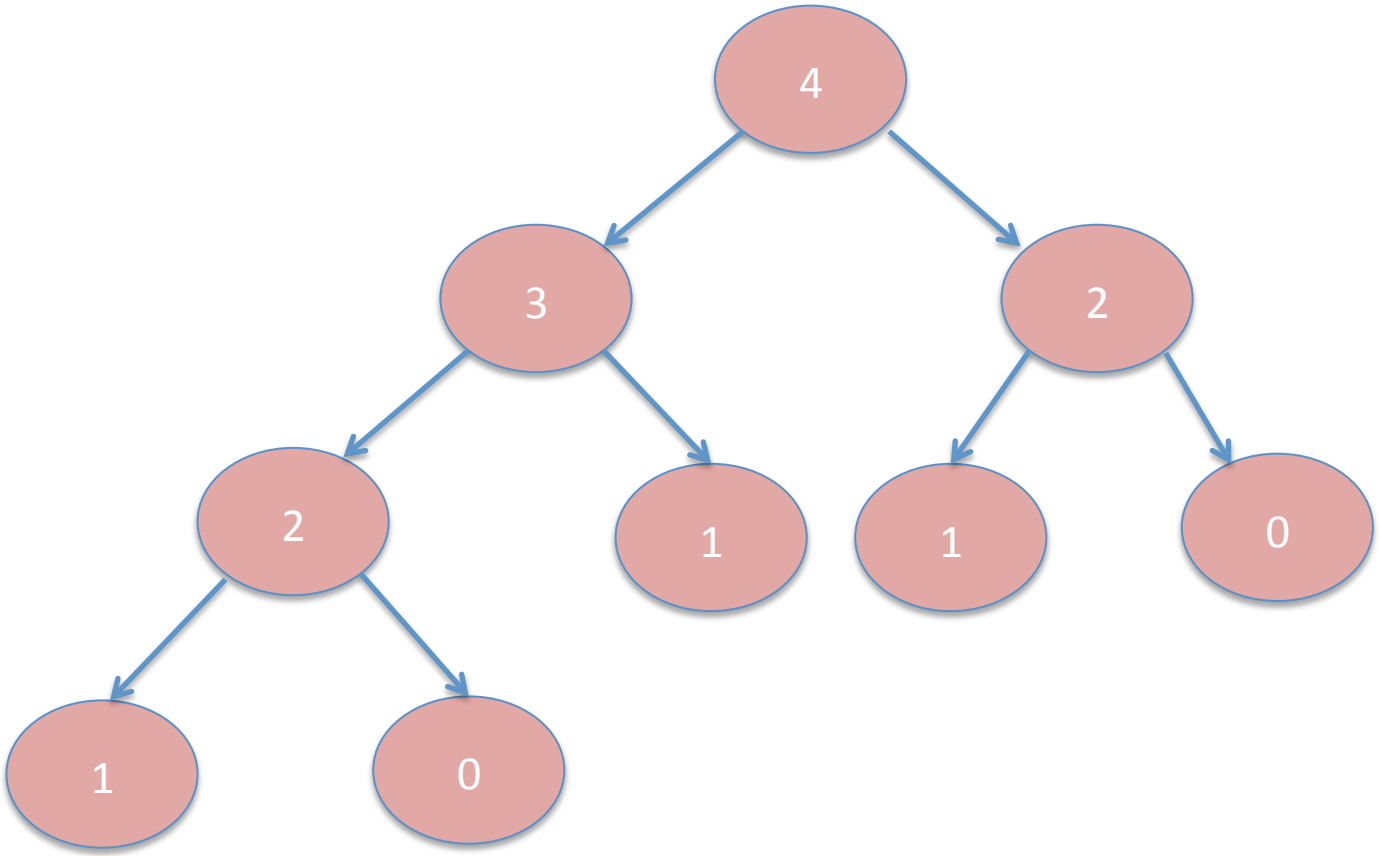


Fibo(4)



Fibo(4)

Fibo(4)



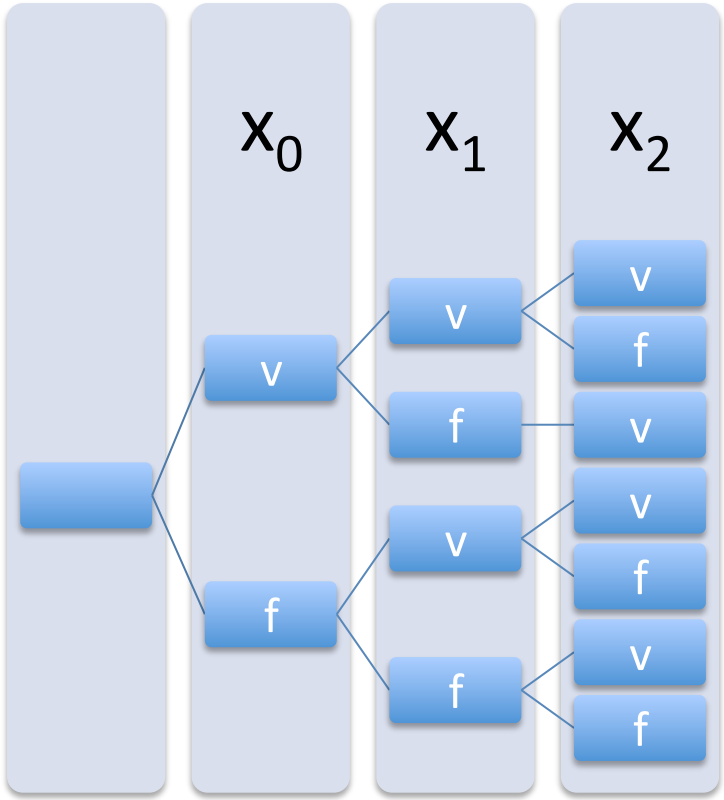
Exploration arborescente

- La récursivité permet de traverser une structure arborescente sans la construire explicitement
- Mettre en œuvre cette technique pour visiter des arbres de décisions
- Permet de résoudre de façon exhaustive des problèmes NP-Difficiles.

Satisfiablité

- Soit $f(x_0,x_1,x_2)$, il faut tester tous les cas pour affirmer que f n'est pas satisfiable.

Arbre de décision



Algorithme

```
parcourir(vecteur v, int arite, int profondeur)
{
    Si profondeur == arité
        retour;
    v[profondeur] = vrai;
    parcourir(v,arite,profondeur+1);
    v[profondeur] = faux;
    parcourir(v,arite,profondeur+1);
}
```

Mise en œuvre en C

```
bool formule_1(    bool *v)
{
return v[0] && !v[1] && v[3] ;
}

bool formule_2(bool *v)
{
return false ;
}
```

```
bool formule_3(bool *v)
{
 for(int i =1; i < 12 ; i++)
       if (  v[i-1] && v[i])
             return false;
 return true;
}
```

Mise en œuvre en C

```
bool satisfiable(  bool (*f)(bool *vecteur),
                   int arite)
{
    bool v[arite];
    memset(v,sizeof(bool)*arite,0);
    return verifier(f,v,arité,0);
}
```

Mise en œuvre en C

```
bool vérifier(  bool (*f)(bool *vecteur), bool *v,
                              int arite, int profondeur)
{
     if (profondeur == arite)
          return f(v);
     v[profondeur] = true;
     if (verifier(f,v,arite,profondeur+1))
          return true;
     v[profondeur] = false;
     return verifier(f,v,arite,profondeur+1);
}
```

matrice2d

```
matrice2d matrice2d_creer(int d1, int d2)
{
  matrice2d m = memoire_allouer (d1 * sizeof (*m));
  m[0]= memoire_allouer (d1 * d2 * sizeof (**m));
  for(int i=1; i < d1; i++)
    m[i]=m[0]+i*d2;
  return m;
}

void
matrice2d_liberer(matrice2d self)
{
  memoire_liberer(self[0]);
  memoire_liberer(self);
}
```

```
#ifndef MATRICE2D_H
#define MATRICE2D_H
typedef double **matrice2d;
matrice2d matrice2d_creer(int d1, int d2);
void matrice2d_liberer(matrice2d self);
#endif
```

Tableau 2 dimensions

- Simuler un tableau 2d avec un tableau 1d

```
inline double *elem(double *m, int i, int j, int ncol)
{
	return elem + i * ncol + j;
}

inline double *allouer(int nlignes, int ncolonnes)
{
	return malloc(sizeof(double) * nlignes * ncolonnes) ;
}

inline void liberer(double *m)
{
	free(m);
}
```

Le Problème du Sac à dos

- *« Étant donné plusieurs objets possédant chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ? »*
  - *Max = 3000*

Fer-a-repasser	1250	35
Bague	12.1	3400
Tournevis	270	7
Ordinateur	2500	1234
Baguette	250	0.95
Foie-gras	400	34
Foie-gras	300	20.45
Television	1700	1150
Livre	412	8.50
Chaussures	900	99
Pull-over	800	45

Le Problème du Sac à dos

- Méthode approchée
  - Algorithme glouton
- Méthode exhaustive
  - Algorithme récursif
- Méthode *Branch and Bound*
  - Procédure de séparation et évaluation
  - Énumérer de façon intelligente les meilleures solutions possibles
    - Éliminer les pistes non prometteuses
    - Organiser le calcul pour le minimiser

Le Problème du Sac à dos

- Énoncé des données

MAX = 30

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

  - Abstraction nécessaire
    - Ne pas être pénalisé par des choix non algorithmiques.
    - Choisir la meilleure structure de données pour l'algorithme considéré.
    - Savoir passer d'une structuration à une autre

Le Problème du Sac à dos
Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant l'ordre du tableau

Max = 30

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

Le Problème du Sac à dos
Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant l'ordre du tableau

Max = 30

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

  - Sac = {1} reste 17, Valeur = 7

Le Problème du Sac à dos Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant l'ordre du tableau

Max = 30

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

  - Sac = {1} Max = 17, Valeur = 7
  - Sac = {1,2} Max = 5, Valeur = 11

Le Problème du Sac à dos
Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant le rapport Prix/Poids

Max = 30

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

Objets	1	2	3	4
p_i / w_i	0,54	0,33	0,37	0,30

Le Problème du Sac à dos Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant le rapport Valeur / Poids

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

Max = 30

Organiser les données

Objets	1	3	2	4
pi	7	3	4	3
wi	13	8	12	10
p_i / w_i	0,54	0,37	0,33	0,30

Le Problème du Sac à dos Glouton

- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant le rapport Valeur/ Poids

Objets	1	3	2	4
pi	7	3	4	3
wi	13	8	12	10
p_i / w_i	0,54	0,37	0,33	0,30

Max = 30

  - Sac = {1} Max = 17, Valeur = 7
  - Sac = {1,3} Max = 9, Valeur = 10

Le Problème du Sac à dos Glouton

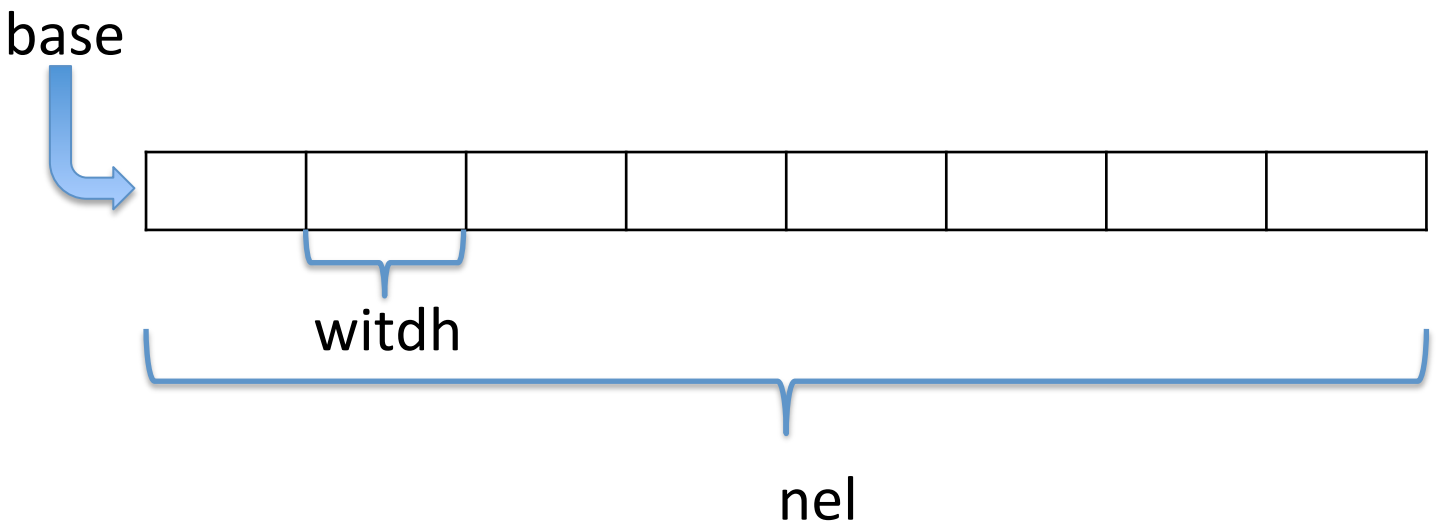
- Mettre le plus d'objets possible dans le sac
  - Considérer les objets suivant l'ordre
    - du rapport Valeur/Poids
    - de valeur
    - du poids
- Implémentation en C
  - Nécessite de pouvoir trier les donnée
    - Utilisation de qsort()

Le Problème du Sac à dos Glouton

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *, const void *));
```

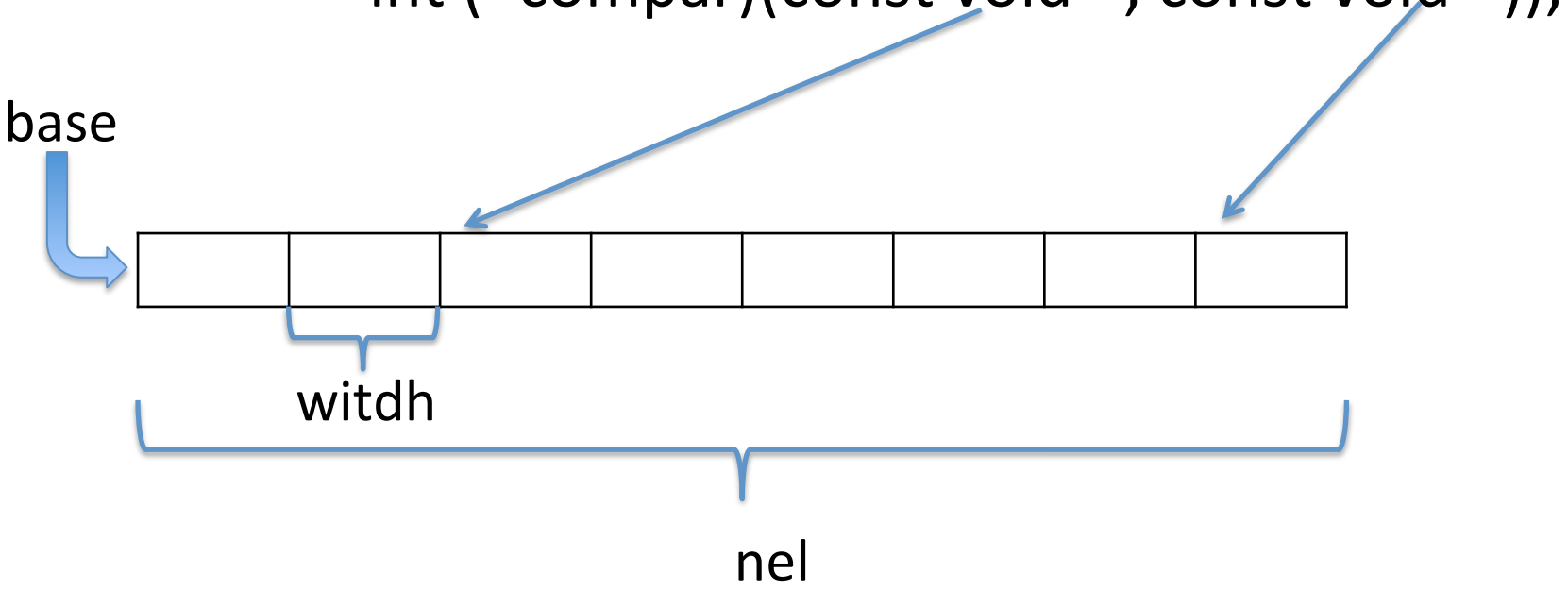
Le Problème du Sac à dos Glouton

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *, const void *));
```



Le Problème du Sac à dos Glouton

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compar)(const void *, const void *));
```



Le Problème du Sac à dos
Glouton

```
void qsort(void *base,
                size_t nel,
                size_t width,
                int (*compar)(const void *, const void *));

int comparer_entier(const void *a, const void *b){
    int va = *(int*)a;
    int vb = *(int*)b;
    return (va>vb) – (va<vb);
}
```

Le Problème du Sac à dos Glouton

```
qsort( tableau,
        sizeof(int),
        8,
        comparer_entier);

int comparer_entier(const void *a, const void *b){
    int va = *(int*)a;
    int vb = *(int*)b;
    return (va>vb) – (va<vb);
}
```

Le Problème du Sac à dos Glouton

```
typedef struct {int num; float valeur; float poids} objet;
int comparer_rapport(const void *a, const void *b){
    objet x = (objet*)a;
    objet y = (objet*)b;
    float rx = x->valeur / x->poids ;
    float ry= y->valeur / y->poids ;
    return (rx>ry) – (rx<ry);
}
qsort(tableau, sizeof(objet), 8, comparer_rapport);
```

Le Problème du Sac à dos interface

- Ne pas s'embarrasser avec de détails
  - Proposer une interface simple et efficace

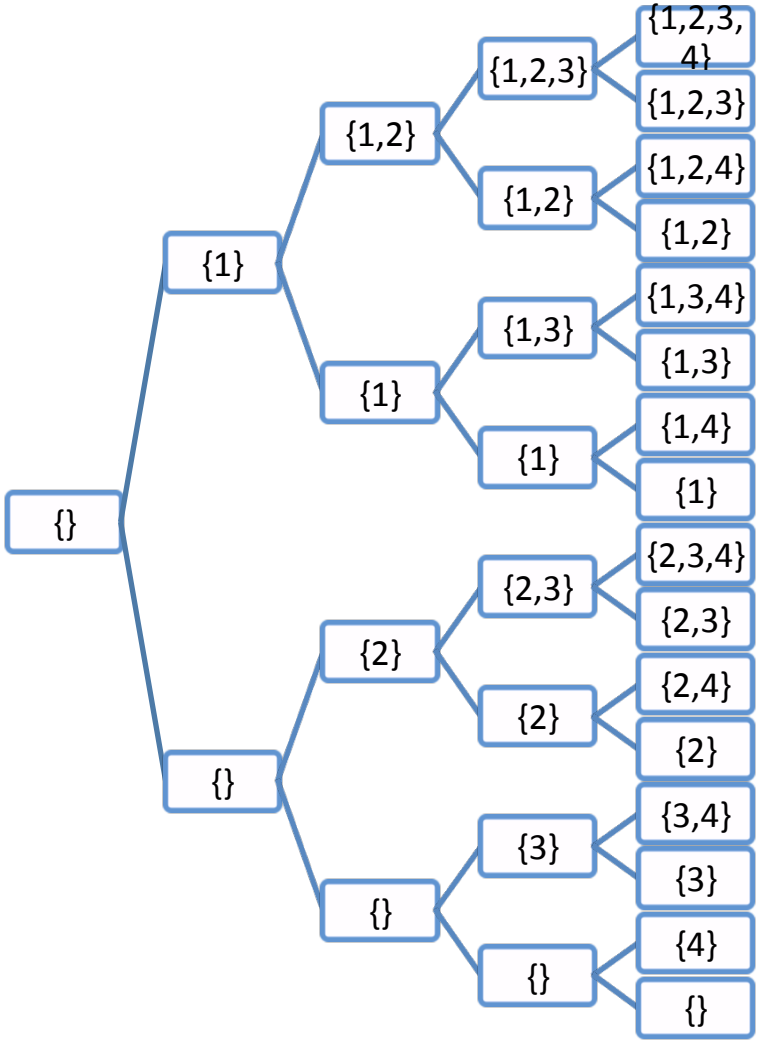
Le Problème du Sac à dos interface

- Ne pas s'embarrasser avec de détails
  - Proposer une interface simple et efficace

```
#ifndef sac_a_dos_h
#define sac_a_dos_h
typedef struct {float valeur; float poids} objet;
void sad_glouton_rapport(objet *les_objets, int nb_objets,
                                        int *solution, int *longueur_solution);
/* place dans solution les indices des objets retenus selon la méthode
gloutonne privilégiant le rapport valeur / poids
NB. l'allocation est à la charge de l'utilisateur */
#endif
```

Le Problème du Sac à dos
Approche exhaustive

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10



Le Problème du Sac à dos
Approche exhaustive

```
void sac_a_dos_rec(
                        sac données,
                        sac solution,
                        sac en_cours,
                        int p){

if (p==données->nb){
        if(en_cours->valeur > solution->valeur &&
                en_cours->poids <= données->poids)
        copier(en_cours,solution);
        return ;
}
 ajouter(en_cours, données->o[p]);
 sac_a_dos_rec(données, solution, en_cours, p+1);
 retirer_dernier(en_cours);
 sac_a_dos_rec(données, solution, en_cours, p+1);
}
```

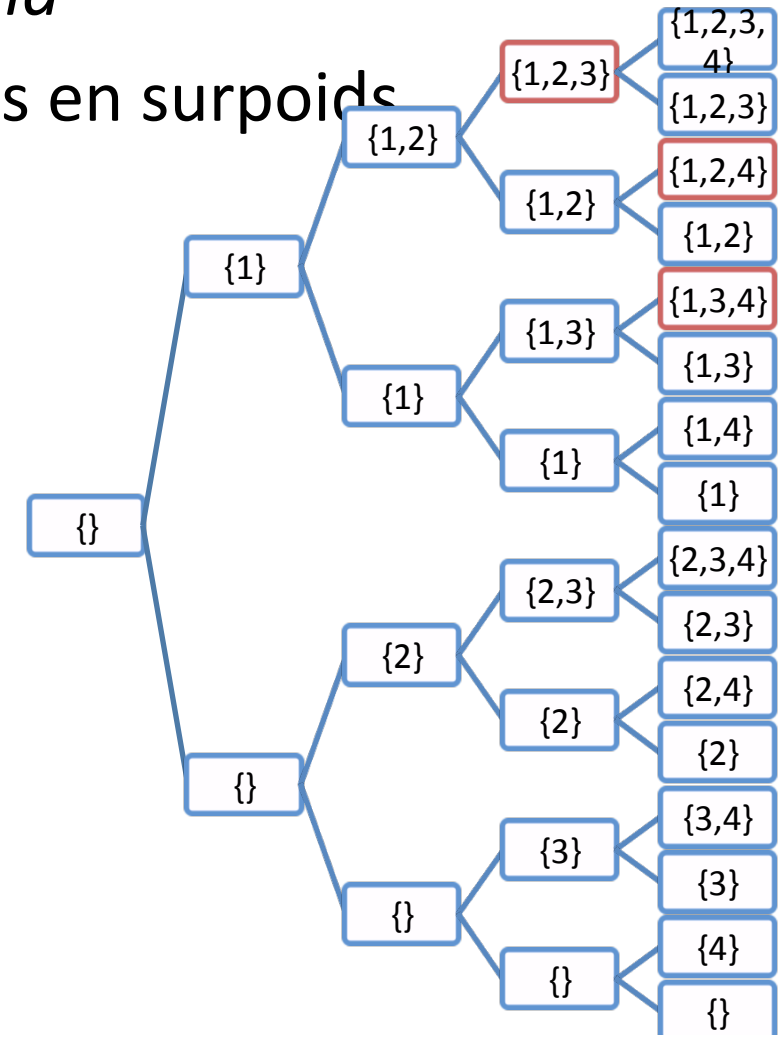
```
typedef struct {
  objet *o;
  int nb;
  float poids;
  float valeur;
  } *sac;
```

Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

ne pas visiter les nœuds en surpoids

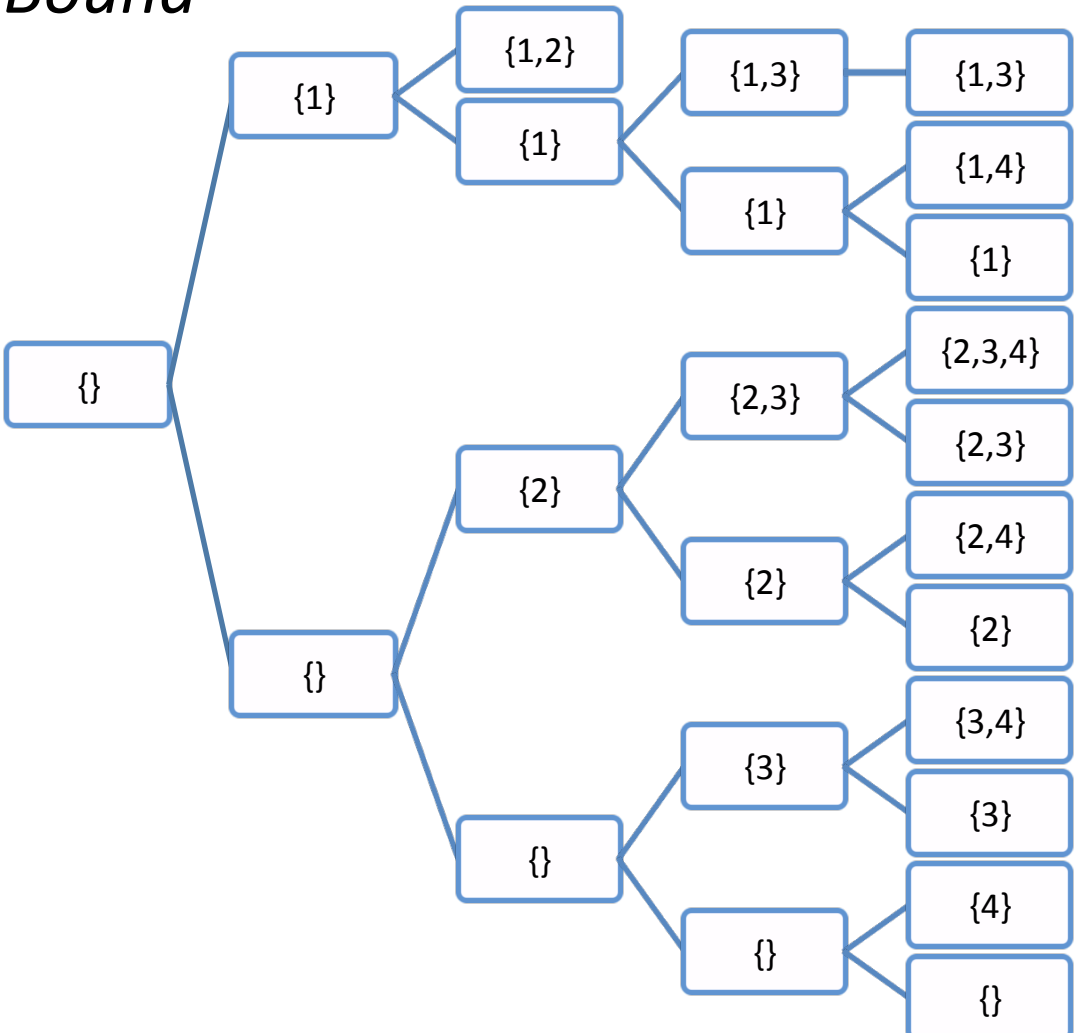
Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10



Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10



Le Problème du Sac à dos
Approche exhaustive

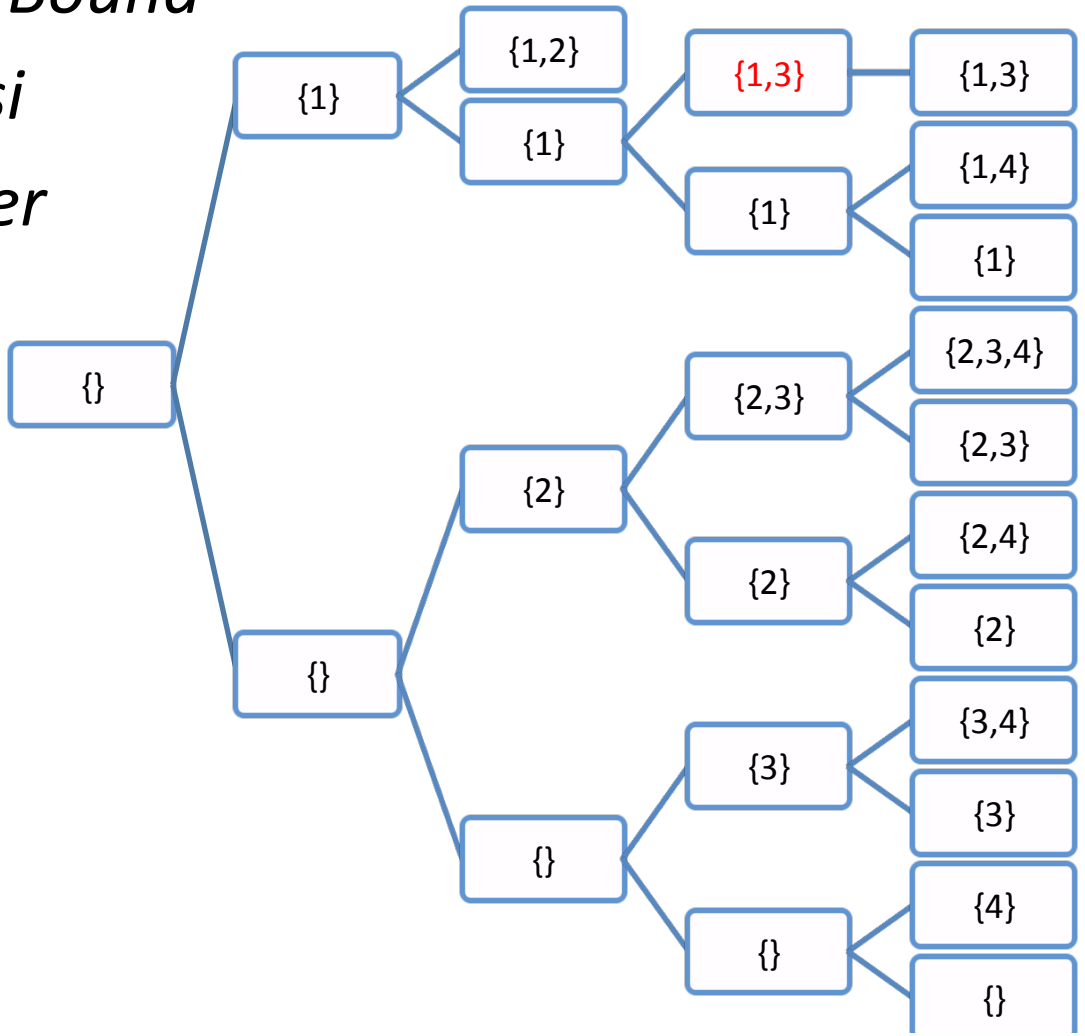
```
void sac_a_dos_rec(…){

if (p==données->nb){
        if(en_cours->valeur > solution->valeur &&
                en_cours->poids <= données->poids)
        copier(en_cours,solution);
        return ;
}
 if (données->o[p]+en_cours->poids <= données->poids) {
        ajouter(en_cours, données->o[p]);
        sac_a_dos_rec(données, solution, en_cours, p+1);
        retirer_dernier(en_cours);
}
 sac_a_dos_rec(données, solution, en_cours, p+1);
}
```

Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Arrêter l'exploration si on ne peut plus ajouter d'objet assez petit

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10



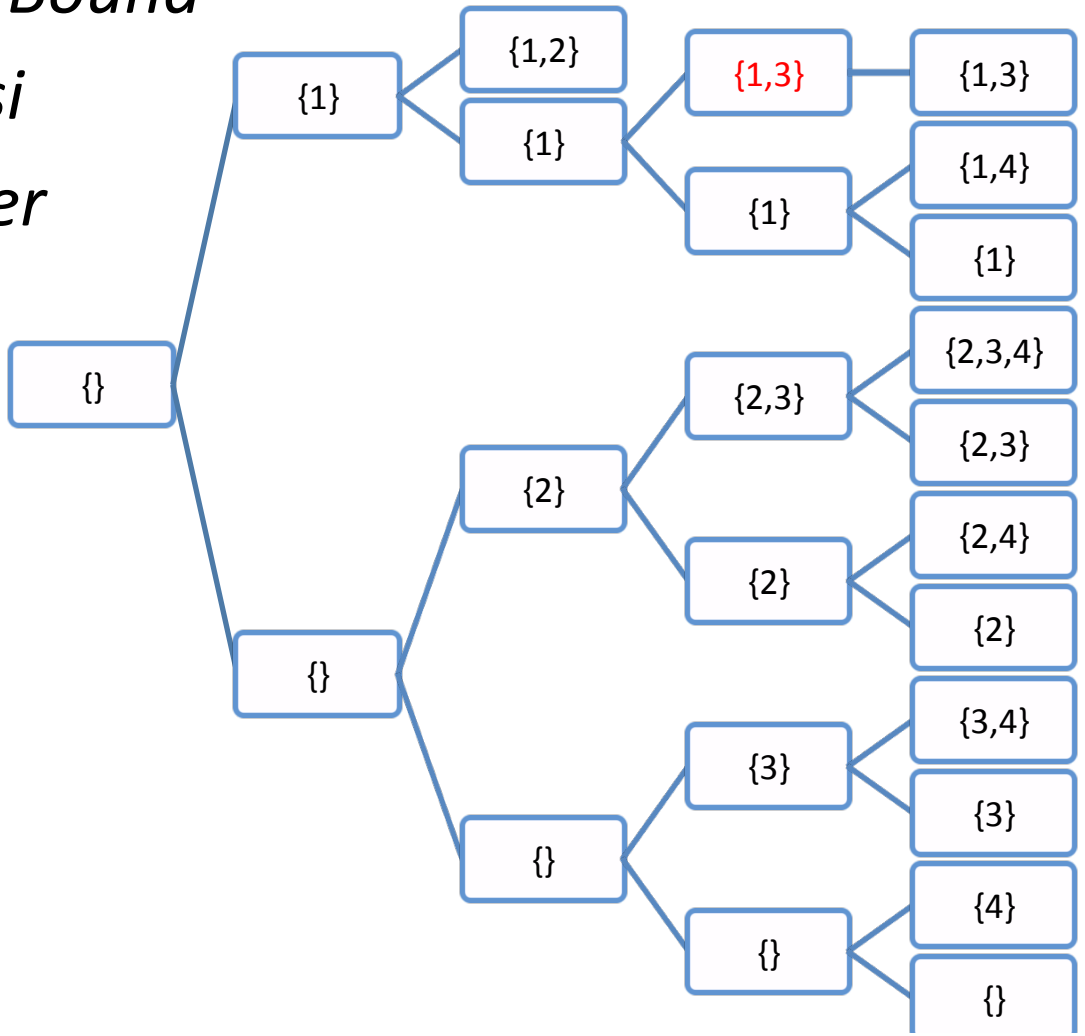
Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Arrêter l'exploration si on ne peut plus ajouter d'objet assez petit

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

8 8 8 10



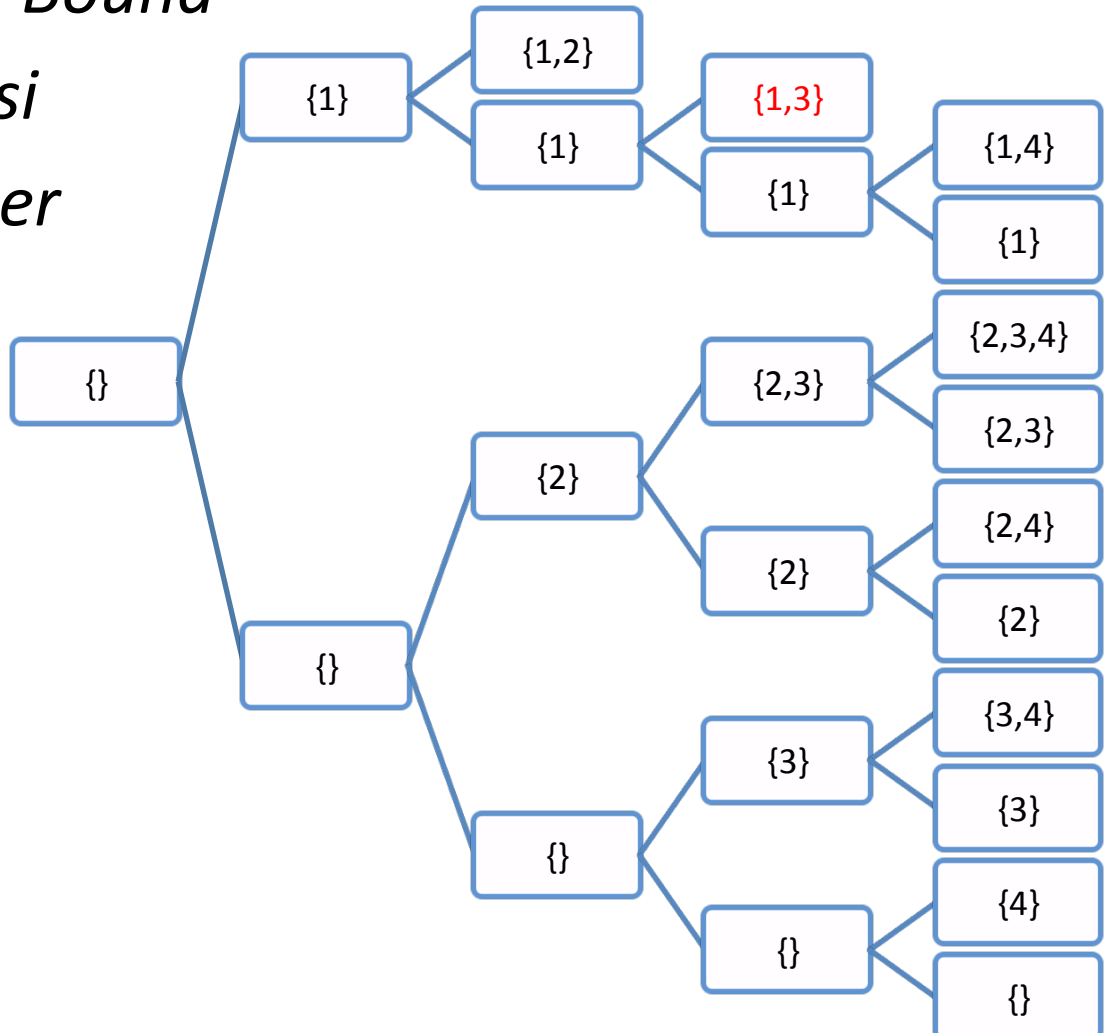
Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Arrêter l'exploration si on ne peut plus ajouter d'objet assez petit

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

8	8	8	10



Le Problème du Sac à dos
Approche exhaustive

```
void sac_a_dos_rec(…){

if (p==données->nb || poids_min[p] + en_cours->poids[p] <= données->poids) {
        if(en_cours->valeur > solution->valeur &&
                en_cours->poids <= données->poids)
        copier(en_cours,solution);
        return ;
}
 if (données->o[p]+en_cours->poids[p] <= données->poids) {
        ajouter(en_cours, données->o[p]);
        sac_a_dos_rec(données, solution, en_cours, p+1);
        retirer_dernier(en_cours);
}
 sac_a_dos_rec(données, solution, en_cours, p+1);
}
```

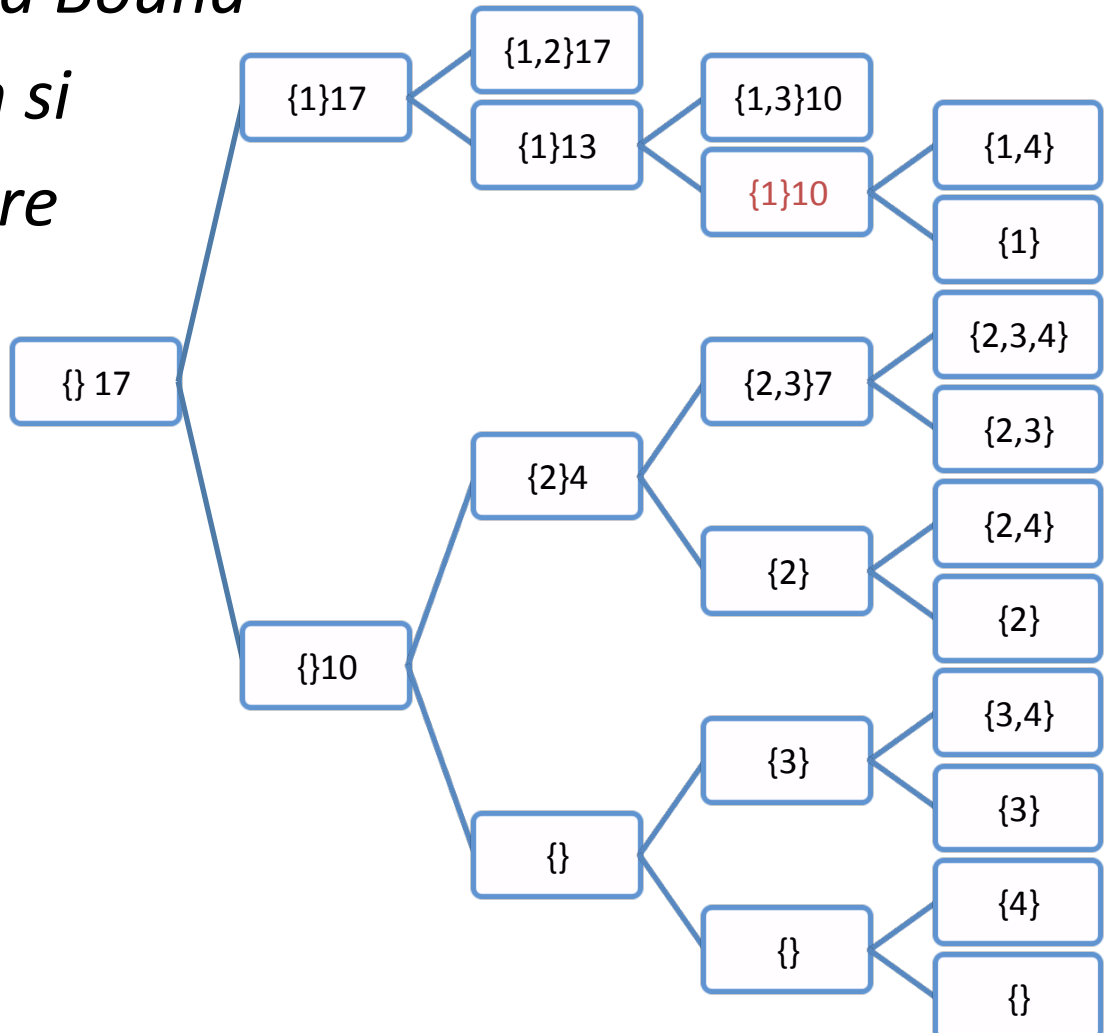
Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Arrêter l'exploration si on ne peut plus battre la solution actuelle

Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

espérance



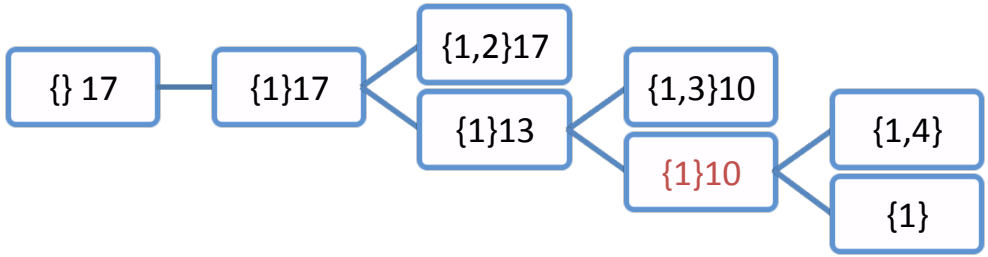
Le Problème du Sac à dos
Approche exhaustive

Méthode *Branch and Bound*

Arrêter l'exploration si

on ne peut plus battre

la solution actuelle



Objets	1	2	3	4
p_i	7	4	3	3
w_i	13	12	8	10

espérance

Le Problème du Sac à dos
Approche exhaustive

```
void sac_a_dos_rec(…){
if(espérance[p] + en_cours->valeur <= solution->valeur) return;
if (p==données->nb || poids_min[p] + en_cours->poids <= données->poids) {
        if(en_cours->valeur > solution->valeur &&
                en_cours->poids <= données->poids)
        copier(en_cours,solution);
        return ;
}
 if (données->o[p]+en_cours->poids <= données->poids) {
        ajouter(en_cours, données->o[p]);
        sac_a_dos_rec(données, solution, en_cours, p+1);
        retirer_dernier(en_cours);
}
 sac_a_dos_rec(données, solution, en_cours, p+1);
}
```

Le Problème du Sac à dos
Approche exhaustive

- Branch and bound permet de réduire significativement le temps d'exploration
  - Sauf si le surcoût du calcul des bornes est trop important
    - Calcul incrémental des bornes
  - Sauf si l'ordre d'exploration n'est pas bon
    - Singer un algorithme glouton
      - Trier le tableau suivant le rapport poids / performance

Branch and Bound
voyageur de commerce

- Borne maximum = solution proposée par algorithme glouton
- Borne minimum = distance parcourue + distance minimale à parcourir
  - Distance de la ville courante à la ville de départ;
  - Mieux : distance à la ville restant à visiter la plus lointaine de la ville initiale plus le retour à cette même ville initiale;
  - Somme des la somme des plus petites arêtes des villes à visiter;
  - …
- Ordre de parcours :
  - Plus proche voisin non visité

Optimisation voyageur de commerce

- éviter les croisements
  - Complexité du test
    - *k* arêtes → *k-2* tests de croisement
    - Construction d'un chemin de longueur n → $O(n^2)$ tests
  - Le coût du test augmente avec la profondeur
  - L'impact du test diminue avec la profondeur

  → Désactiver le test à partir d'un seuil