USTHB
Université des Sciences et de la Technologie
Houari Boumediene

Projet

Conception analyse et complexité des algorithmes

ABDOU Maria hind 161631069767
FERRADJI Tarek 181931058767
YOUSFI Zakaria 17173202650
MOUAZ Rayane Hamza 191931030386

Master 1 SII 2022/2023
Groupe 03

Introduction générale

En informatique, la complexité d'un algorithme est la quantité de ressources nécessaires à son exécution. Une attention particulière est accordée aux besoins en temps (complexité temporelle) et en mémoire (complexité spatiale). De nos jours les tailles de mémoire ont augmenté c'est pourquoi les informaticiens sont plus intéressés par la complexité temporelle. La complexité est un moyen de mesurer l'efficacité d'un tel algorithme.

1) Environnent Expérimentale :

RAM: 8Go
CPU: i5
6300U
Windows 10
Langage de programmation : C
Environnement de développement : Notepad++

Objectif :

L'objectif de ce projet est d'écrire six algorithmes différents pour déterminer si un nombre entier est premier ou composé et l'évaluation de la complexité pour chacun des six algorithmes.

Partie I : Développement de l'algorithme et du programme correspondant

Explication de la fonction BOOLP5:

La fonction BOOLP5 prends en paramètre le nombre « nmbr » et elle retourne un booléen « t », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai.
-En suite en boucle de 1 jusqu'à le nombre divisé par 2.
-on met le nombre modulo i si égale a 0 on incrément la variable n sinon on fait rien
-À la fin on teste si n > 1 la fonction retourne faux « nombre pas premier »
Sinon elle retourne vrai « nombre premier »

Pseudo code :

```
Fonction BOOLP5:
Entrée : nmbr
Sortie :t
Var : int A, i,n=0;
      bool t=vrai;

DEBUT :

Pour i=1 à nmbr/2
faire
  A=nmbr % i ;
  Si (A==0) :
      n++ ;
  Fsi
fait
Si n>1 :
     t=faux ;
Sinon
     t=vrai ;
FIN .
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute la boucle « Pour » avec ces 4 instructions jusqu'à le nombre sur 2 ($\frac{n}{2}$ fois) plus le nombre d'instruction restant (6 instructions d'initialisation et le return = $\frac{4}{2}$ n +6.

Ce qui nous donne donc une complexité théorique de O(n).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Pour » avec ces 4 instructions jusqu'à le nombre sur 2 ($\frac{n}{2}$ fois) et le nombre d'instruction dans la boucle (6 instructions) plus les deux instructions (Si n>1 et t=faux) = $\frac{4}{2}$ n +8

Ce qui nous donne donc une complexité théorique de O(n).

Explication de la fonction BOOLP4:

La fonction BOOLP4 prends en paramètre le nombre « nmbr » et elle retourne un booléen « t », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai.
-En suite en boucle de 1 jusqu'à le nombre divisé par x (pour optimiser le code,) .
- on met le nombre modulo i si égale a 0 on incrément la variable n est x==n, sinon on fait rien
-À la fin on teste si n > 1 la fonction retourne faux « nombre pas premier »
Sinon elle retourne vrai « nombre premier »

Pseudo code :

```
Fonction BOOLP4:
Entrée : nmbr
Sortie :t
Var : int A, i,n=1,x=2;
      bool t=vrai;

DEBUT :

Pour i=2 à nmbr/x
faire
  A=nmbr % i ;
  Si (A==0) :
     x=i;
     n++;
  Fsi
fait
Si n>1 :
      t=faux ;
Fsi
FIN.
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute la boucle « Pour » jusqu'à le nombre sur «x» ($\frac{n}{x}$ fois) ➡ $\frac{3}{x}$ n, plus le nombre d'instruction dans la boucle (6 instructions) = $\frac{3}{x}$ n +6.
Ce qui nous donne donc une complexité théorique de O(n).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Pour » avec ces 5 instructions jusqu'à le nombre sur « 2 » ($\frac{n}{2}$ fois) ➡ $\frac{5}{2}$ n, et le nombre d'instruction restant (6 instructions d'initialisation et le return plus les deux instructions (Si n>1 et t=faux) = $\frac{5}{2}$ n +8
Ce qui nous donne donc une complexité théorique de O(n).

Explication de la fonction BOOLP6:

La fonction BOOLP6 prends en paramètre le nombre « nmbr » et elle retourne un booléen « t », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai.
-En suite en boucle de 1 jusqu'à le nombre moins 1.
- on met le nombre modulo i si égale a 0 on incrément la variable n sinon on fait rien
-À la fin on teste si n > 1 la fonction retourne faux « nombre pas premier »
Sinon elle retourne vrai « nombre premier »

Pseudo code :

```
Fonction BOOLP6:
Entrée : nmbr
Sortie :t
Var : int A, i,n=1;
      bool t=vrai;

DEBUT :

Pour i=1 à nmbr-1
faire
  A=nmbr % i ;
  Si (A==0) :
      n++ ;
  Fsi
fait
Si n>1 :
     t=faux ;
Sinon
      t=vrai ;
FIN.
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute la boucle « Pour » avec ces 4 instructions jusqu'à le nombre moins 1(n fois) ➡ 4n, plus le nombre d'instruction restant (6 instructions) = 4n +6.
Ce qui nous donne donc une complexité théorique de O(n).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Pour » avec ces 4 instructions jusqu'à moins 1 (n fois) ➡ 4n, plus le nombre d'instruction restant (6 instructions d'initialisation et le return et les deux instructions (Si n>1 et t=faux) = 4n +8
Ce qui nous donne donc une complexité théorique de O(n).

Explication de la fonction BOOLP2:

La fonction BOOLP2 prends en paramètre le nombre « nmbr » et elle retourne un booléen « n », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai et tester si nombre modulo i défirent de 0.

-si le nombre est impaire en boucle tant que « i » inférieur à nombre sur 2 et n est vrai si le modulo égale pas 0 sinon n==faux est on sort de la boucle .

-sinon on teste si nombre égale à 2 , la fonction retourne vrai « nombre premier »
Sinon elle retourne faux « nombre pas premier »

Pseudo code :

```
Fonction BOOLP2:
Entrée : nmbr
Sortie :n
Var : int A, i=2;
      bool n=vrai;


DEBUT :
SI ((nmbr % i)!=0): i=3;
Tant que (i<nmbr/2 && n==vrai)
faire
  A=nmbr % i ;
  Si (A==0) :
      n=faux ;
  Fsi
  i=i+2;
 fait
Sinon
   Si (nmbr==2) :
        n=vrai ;
   Sinon
       n=faux ;
    Fsi
FSI
FIN.
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute que 6 instructions ➡6
Ce qui nous donne donc une complexité théorique de O(6).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Tant que » avec ces 5 instructions jusqu'à nombre sur 2 ($\frac{n}{2}$ fois)➡ = $\frac{4}{2}n$.
plus le nombre d'instruction restant (8 instructions d'initialisation et le return) et les deux instructions (Si n>1 et t=faux) = 5n +10
Ce qui nous donne donc une complexité théorique de O(n).

Explication de la fonction BOOLP1:

La fonction BOOLP1 prends en paramètre le nombre « nmbr » et elle retourne un booléen « n », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai.
-si le nombre il est paire en boucle tant que « i » inférieur à la racine de nombre et n est vrai.
-sinon on test si le nombre egale a 2 on met t==vrai sinon on met t==vrai
-si le nombre modulo i egale pas 0 on continue sinon on met t==faux et on sort
-À la fin on teste si nombre modulo i = 0 la fonction retourne faux « nombre pas premier »
Sinon elle retourne vrai « nombre premier »

Pseudo code :

```
Fonction BOOLP1:
Entrée : nmbr
Sortie :n
Var : int A, i=2;
      bool n=vrai;

DEBUT :
SI ((nmbr % i) !=0):
    i=3;
    Tant que (i<=sqrt(nmbr) && n==vrai)
    faire
    A=nmbr % i ;
    Si (A==0) :
        n=faux ;
     Fsi
     i=i+2;
     fait
Sinon
    Si (nmbr==2) :
         n=vrai ;
    Sinon
        n=faux ;
     Fsi
FSI
FIN.
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute que 6 instructions ➡6

Ce qui nous donne donc une complexité théorique de O(6).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Tant que » avec ces 5 instructions jusqu'à la racine du nombre ($\frac{\sqrt{n}}{2}$ fois)= = ($\frac{5\sqrt{n}}{2}$.
plus le nombre d'instruction restant (10 instructions d'initialisation et le return) = 3n +10
Ce qui nous donne donc une complexité théorique de O($\sqrt{n}$).

Explication de la fonction BOOLP3:

La fonction BOOLP3 prends en paramètre le nombre « nmbr » et elle retourne un booléen « n », (« vrai »pour le nombre est premier et « faux »pour le nombre n'est pas premier) .

Le traitement qu'elle effectue possède comme suite :

-On commence par initialiser booléen à vrai et tester si nombre modulo i défirent de 0.
-si le nombre est impaire en boucle tant que « i » inférieur à nombre et n est vrai si le modulo egale pas 0 sinon n==faux est on sort de la boucle .
-sinon on teste si nombre égale à 2 , la fonction retourne vrai « nombre premier »
Sinon elle retourne faux « nombre pas premier »

.

Pseudo code :

```
Fonction BOOLP3:
Entrée : nmbr
Sortie :n
Var : int A, i=1;
      bool n=vrai;

DEBUT :

SI ((nmbr%i)!=0):
   i=3;
   Tant que (i<nmbr && n==vrai)
   faire

        A=nmbr% i;
        Si (A==0) :
            n=faux ;
         Fsi
I=i+2 ;
   fait
Sinon
       Si (nmbr==2) : n=vrai ;
Sinon
       n=faux ;
Fsi
```

Evaluation de la complexité :

Le meilleur cas : représente le cas au le nombre est premier .Dans ce cas, l'algorithme exécute que 8 instructions si le nombre égale à é ou le nombre est paire ➡ 8
Ce qui nous donne donc une complexité théorique de O(8).

Le pire cas : correspond au cas au le nombre n'est pas premier .Dans ce cas, l'algorithme exécute la boucle « Tant que » avec ces 5 instructions jusqu'à le nombre (n fois)= = $5n$.
plus le nombre d'instruction restant (6 instructions d'initialisation et le return) = $5n+10$
Ce qui nous donne donc une complexité théorique de O(n).

PARTIE2 :

1) Tests sur des nombres premiers ayant au plus 12 chiffres :

Algorithme \ Nombre	105929	1300367	15495737	2124749677
P2	0.000 S	0.003 S	0.028 S	5.15 S
P4	0.000 S	0.005 S	0.045 S	7.19 S
P5	0.000 S	0.007 S	0.059 S	7.771 S
P3	0.000 S	0.003 S	0.038 S	6.26 S
P1	0.000 S	0.002 S	0.01 S	2.65 S
P6	0.000 S	0.006 S	0.078 S	13.689 S

Tableau 1 : Temps d'exécution des six algorithmes pour des nombres premiers ayant au plus 12 chiffres .

2) Tests sur 20 nombres par longueur :

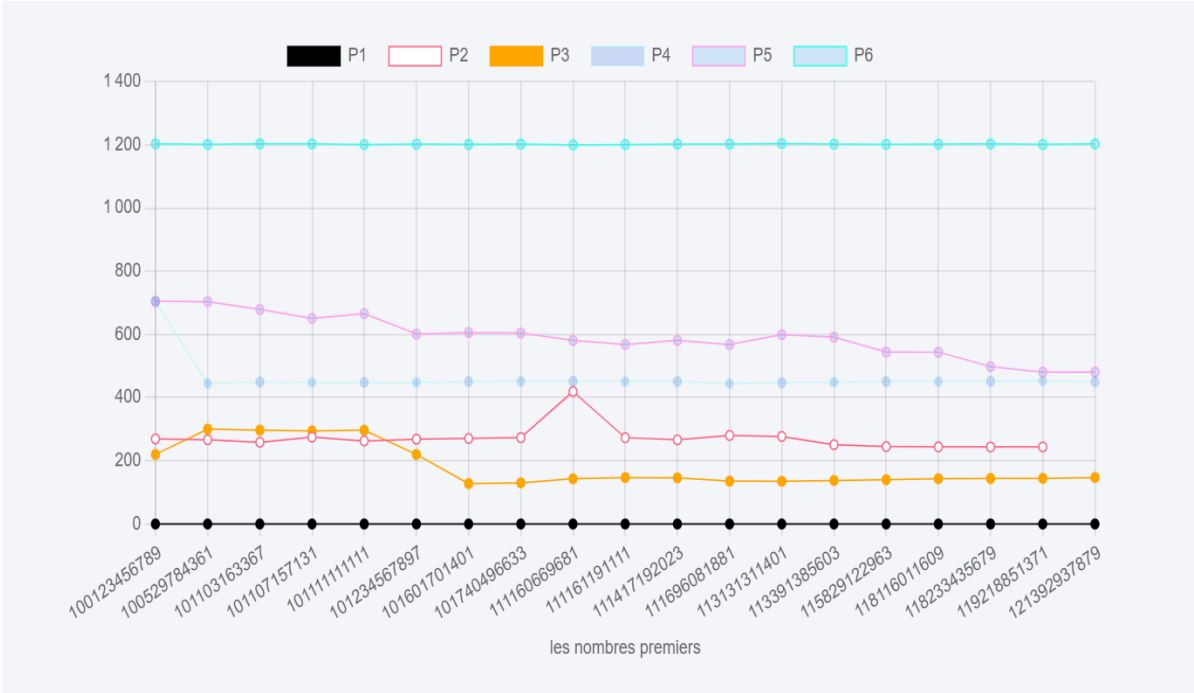
Dans cette partie nous allons observer le comportement de temps d'exécution (seconde) après l'exécution des six algorithmes sur 20 nombre de même longueur.

Algorithme \ Nombre	1001234567899	1005297843611	1011033633716	1011071571311	1011111111111	1012345678967	1016017014001	1017404966633	1111606696831	1111611911111
P1	0.003	0.002	0.003	0.004	0.001	0.001	0.001	0.004	0.003	0.001
P2	220.187	300.345	296.943	294.61	296.713	220.063	127.906	130.295	143.211	146.86
P3	269.212	266.661	258.628	274.730	262.819	268.414	270.581	273.199	419.281	420.298
P4	703.606	444.604	449.379	447.781	447.837	447.861	450.376	451.562	451.60	451.321
P5	704.879	703.65	678.87	650.44	666.06	600.78	605.76	604.32	580.76	567.980
P6	1202.85	1201.39	1202.987	1202.8	1200.87	1201.89	1201.34	1201.876	1999.76	1200.876

Nombre / Algorithme	1114171 92023	11169608 1881	1131313 11401	1133913 85603	1158291 22963	11811601 1609	118233 435679	119218 851371	1213929 37879
P1	0.001	0.002	0.001	0.001	0.001	0.001	0.001	0 .004	0.002
P2	145.978	135.544	135.134	137.801	140.149	143.157	144.013	144.422	147.205
P3	272.980	266.324	280.123	277.009	250.98	244.87	243.98	243.763	243.890
P4	450.879	444.76	446.87	448.765	450.76	450.76	451.54	452.54	450.12
P5	580.87	567.80	599.09	590.97	543.987	543.7	498.06	480.65	480.88
P6	1202.20	1202.45	1203.87	1201.98	1201.21	1202.001	1202.66	1202.1	1202.88

Tableau 2 : Temps d'exécution des six algorithmes pour 20 nombre premiers de même longueur (12 chiffres).

Le graphe : Représente temps d'exécution des six algorithmes en fonction de 20 nombre premiers de même longueur (12 chiffres).



Observation et conclusion:

A partir de ces résultats, on peut observer que chaque fois que la complexité augmente, le temps d'exécution augmente, par exemple nous avons p1 avec une complexité très faible et donc un temps d'exécution faible, tandis que P6 a l'inverse.

3) Tests sur des nombres premiers de 6 à 12 chiffres 50 fois :

Pour des longueurs différentes de nombres, allant de 6 à 12, on exécute les 6 programmes 50 fois (si possible) et on reporte la moyenne du temps d'exécution.

Dresser la table des résultats numériques puis le graphique correspondant. Que pouvez-vous conclure :

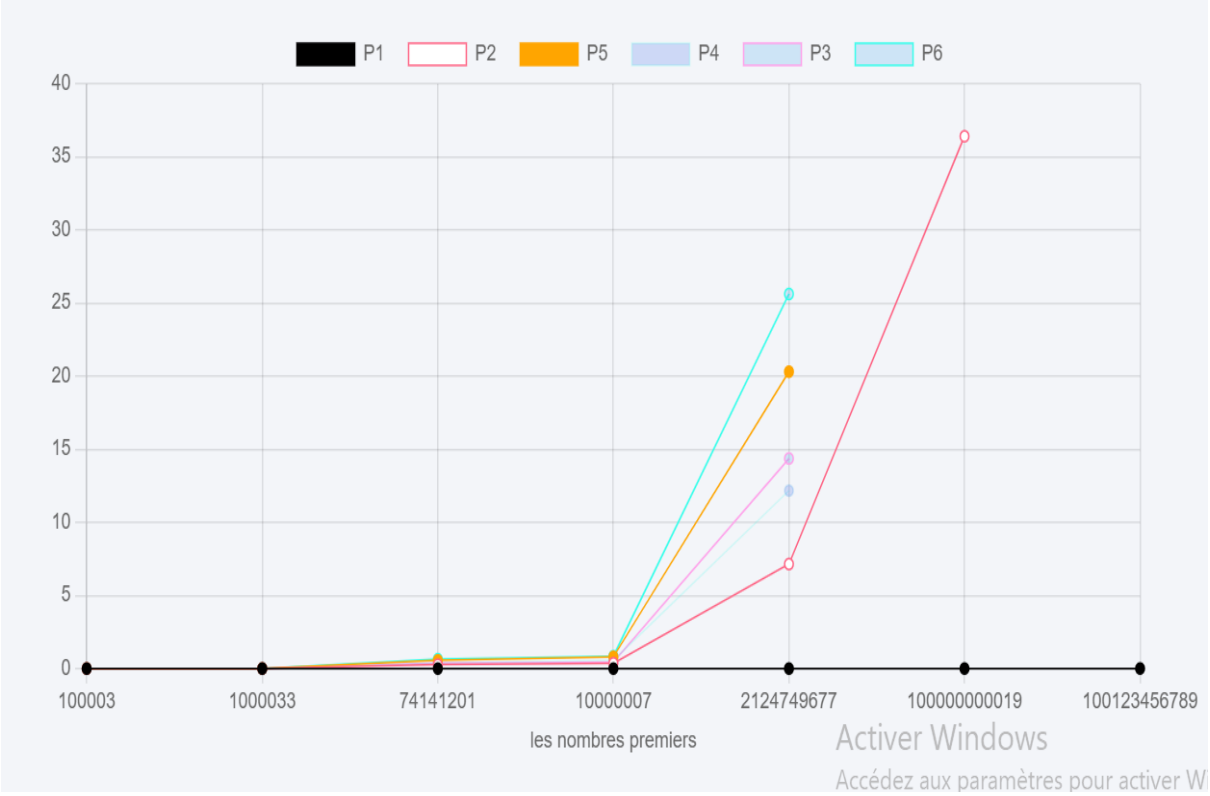
```
int main()
{
bool a ;
double moy, delta;
clock_t  t1,t2;
ong long  int  t[19];
moy=0;
t[0]=100003;
t[1]=1000033;
t[2]=74141201;
t[3]=100000007  ;
t[4]=2124749677;
t[5]=10000000019;
t[6]=100123456789;
int i ,j;

for(i=0;i<=6;i++){
moy=0;
for(j=0;j<50;j++){
 t1=clock();
a=p1(t[i]);//choisit le programme
t2=clock();
delta = (double) (t2-t1)/CLOCKS_PER_SEC ;
moy=moy+delta;

}
if(a==true){
          printf("le nombre %lli il est premier et le temps moyen
  d'execution est %f \n",t[i], moy/50);
  }
  else{
          printf("le nombre %lli  est composé  et le temps d'execution est
  %f \n",t[i], moy/50);
  }
}

  }
```

Table des résultats numériques :

Nombre / Algorithme	100003	1000033	74141201	10000007	2124749677	100000000019	100123456789
P1	0	0	0.000120	0.000120	0.000560	0.001440	0.005620
P2	0.00340	0.00332	0.2683	0.3697	7.1509	36.410540	Aucune résultat
P5	0.0044	0.00464	0.56344	0.80626	20.309040	Aucune résultat	Aucune résultat
P3	0.003	0.00544	0.39	0.561040	12.1718	Aucune résultat	Aucune résultat
P4	0.0004	0.005820	0.37318	0.44072	14.368960	Aucune résultat	Aucune résultat
P6	0.00096	0.00838	0.6505	0.84888	25.62316	Aucune résultat	Aucune résultat

Tableau 3 : Temps d'exécution des six algorithmes pour des nombres premiers de 6 à 12 chiffres 50 fois .

Le graphe : Représente temps d'exécution des six algorithmes en fonction des nombres premiers de 6 à 12 chiffres 50 fois .



Observation et conclusion :

D'après ces résultats nous remarquons que plus les valeurs en entrée grandissent plus le temps d'exécution devient important, et que l'utilisant de la même valeur avec les six différents algorithmes montre

que plus l'algorithme a une complexité petite plus le temps d'exécution est faible.

Analyse détaillée :
- Le temps d'exécution de p6 représente le temps d'exécution le plus important. Ceci est dû au fait qu'on teste tous les diviseurs, on fait donc N-1 itérations.

- Le temps d'exécution de p5 a été réduit approximativement à la moitié du temps d'exécution P6. Ceci correspond au passage de n-1 itérations à n-1/2.
- Aussi, on constate que le temps d'exécution p2 équivaut approximativement à la moitié du temps d'exécution p3. ce résultat correspond bien au résultat obtenu en théorie : nb_itérations (p2)= [n/4] – 1 et nb_itérations (p3) = n/2 -1.
- Enfin, pour p1, on remarque clairement que les temps d'exécution diminuent considérablement (le temps d'exécution est très proche de 0).

Conclusion :
. Le lien entre temps d'exécution et la complexité est l'ordre de grandeur du nombre d'opération arithmétique ou logique, du nombre d'accès en mémoire et d'affectation qu'on doit effectuer lors de l'exécution d'un algorithme.
c'est ce qu'on a pu observer d'après les test faite pour les six algorithmes précédemment .
(plus l'algorithme est optimisé plus le temps devient petit).
ce qui montre l'importance de la complexité.

distribution des tâches :

ABDOU MARIA HIND : le rapport +3algorithme
FERRADJI TAREK : question 1PARTIE 2 +3 algorithme
YOUSFI ZAKARIA :question 2 PARTIE 2
MOUAZ RAYANE HAMZA :question 3 PARTIE 2