1. Concurrent api

interface Runnable (java.lang)

Depuis java 8 , l'interface fondamentale Runnable est devenue une interface (uni-)fonctionnelle.

```
public interface Runnable {
        public void run();
}
```

```
Runnable r = () -> { /*code de la tâche*/ }
```

interface Callable<T> (java.util.concurrent)

```
public interface Callable<T> {
        public T call();
}

Callable<T> c = () -> { /*corps de la tâche*/ return t; }
```

Callable<T> existe depuis le jdk 1.5 et est depuis java 1.8 vue comme une interface (uni-)fonctionnelle .

interface Future<T> (java.util.concurrent)

Disponible depuis le jdk 1.5, un objet **Future<T>** en **java** correspond partiellement à la notion de ***Promise*** en *javascript* .

Un objet technique de ce type (*récupéré immédiatement lors d'un lancement de traitement long*) **permettra de récupérer un résultat en différé** (dans le futur) .

boolean **isDone()**	teste la terminaison du thread pour savoir si la donnée résultante de son exécution est disponible. Retourne true si ce thread s'est bien terminé ou s'il a malheureusement levé une exception pendant son exécution, ou enfin s'il a été suspendu.
T **get()**	retourne le résultat (instance de T) de la tâche exécutée par le thread. Si le thread n'a pas terminé son exécution, alors l'appel de cette méthode est bloqué en attente active jusqu'à ce qu'il termine.
boolean **cancel**(true)	Demande à arrêter la tâche si elle n'est pas finie , le paramètre d'entrée "mayInterruptIfRunning" est souvent fixé à true . La valeur de retour (souvent ignorée) est à true si tâche interrompue ou false si tâche déjà terminée .
*boolean **isCancelled()***	renvoi true si tâche "interrompue" avant la fin .
T get(*long **timeout, TimeUnit** unit**)***	*variante avec **timeout** de la méthode **.get()*** ***Si par exemple après un timeout de 1500 TimeUnit.MILLISECONDS*** la tâche n'est toujours pas finie , cette méthode remonte une exception de type ***TimeoutException*** à rattraper via un try/catch pour nous signaler que cette tâche n'est pas terminée .

ExecutorService

Au coeur du package java.util.concurrent l'interface **ExecutorService** comporte les principales méthodes suivantes :

***void* execute(*Runnable* command)**	lance l'exécution (via un thread créé ou disponible) d'une tâche de type Runnable, ne retourne rien.
***Future<T>* submit(*Callable<T>* task)**	lance l'exécution d'une tâche de type Callable<T> , retourne un objet de type Future<T> permettant de récupérer ultérieurement (en différé) un résultat de type T.
***Future<?>* submit(Runnable task)**	Comme *execute*() mais retournant Future<?> pour attente du résultat ou de la fin
T **invokeAny**(Collection<? extends Callable<T> > tasks) ou bien T **invokeAny**(... tasks,long **timeout**, ...)	Lance (via des threads) une liste de tâches et attend en mode bloquant la première réponse , les autres tâches moins rapides sont automatiquement annulées/stoppées.
List<Future<T>> **invokeAll**(Collection<? extends Callable<T>> tasks) *existe en version avec timeout*	lance en mode bloquant plein de tâches et récupère une liste de résultats encapsulés dans des Future<T> quand tout est prêt/fini .
Autres méthodes	*shutdown*() , *awaitTermination*(timeout,...) , ...

Une instance d'une classe implémentant l'interface ExecutorService pourra être créée via

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

ou bien

```
ExecutorService executor = Executors.newFixedThreadPool(3/*nThreads*/)
```

ou bien

```
ExecutorService executor = Executors.newCachedThreadPool();
```

ou bien d'autres façon encore .

newSingleThreadExecutor() ;	Un seul nouveau thread pouvant lancer plusieurs tâches alors exécutées séquentiellement les unes après les autres .
***newFixedThreadPool*(3/*nThreads*/)**	via un pool de threads (en //) de taille maxi à paramétrer (si plus de tâches à exécuter que de threads dispos --> attente automatique via queue) . chaque thread ne sera arrêté que si appel explicite à .shutdown() .
***newCachedThreadPool*();**	via un pool de threads dont la taille est automatiquement ajustée en fonction des besoins (à la hausse ou à la baisse si rien à faire durant 60s)
autres	voir javadoc Executors

Exemple "EssaiExecutors"

MyRunnableCode.java

```
package tp.langage.thread;

public class MyRunnableCode implements Runnable {
	private String prefix;

	public MyRunnableCode() {super(); this.prefix = "";	}
	public MyRunnableCode(String prefix) {	super(); this.prefix = prefix;	}

	@Override
	public void run() {
		try {
			Thread.sleep(500);
		} catch (InterruptedException e) { }
		System.out.println(prefix + Thread.currentThread().getName());
	}
}
```

EssaiExecutors.java

```
package tp.langage.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

//ExecutorService depuis jdk 1.5
public class EssaisExecutors {

	// Un "ExecutorService" (à fabriquer via Executors.new....Executor())  démarre automatiquement des
	// Threads (rangés dans des "pools") pour exécuter des instances de Callabale<T> ou de Runnable
  public static void main(String[] args) {

	MyRunnableCode myRunnableCode = new MyRunnableCode("*");

	ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
						//exécution séquentielle en background
	singleThreadExecutor.submit(myRunnableCode);
	singleThreadExecutor.submit(myRunnableCode);
	singleThreadExecutor.submit(myRunnableCode);
	singleThreadExecutor.shutdown();//automatiquement différé
```

```
        MyRunnableCode myRunnableCode2 = new MyRunnableCode("#");

        ExecutorService multiThreadExecutor = Executors.newFixedThreadPool(3);
                                                    //exécutions multiples (en //) en background
        multiThreadExecutor.submit(myRunnableCode2);
        multiThreadExecutor.submit(myRunnableCode2);
        multiThreadExecutor.submit(myRunnableCode2);
        multiThreadExecutor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat

        MyRunnableCode myRunnableCode3 = new MyRunnableCode("@");

        ScheduledExecutorService scheduleExecutor =
                                    Executors.newSingleThreadScheduledExecutor();
        System.out.println("Lancement d'un thread/tâche (@) en différé (2000ms)");
        scheduleExecutor.schedule(myRunnableCode3, 2000, TimeUnit.MILLISECONDS);
        scheduleExecutor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat
    }
}
```

Résultats :

```
Lancement d'un thread (@) en différé (2000ms)
#pool-2-thread-3
#pool-2-thread-2
*pool-1-thread-1
#pool-2-thread-1
*pool-1-thread-1
*pool-1-thread-1
@pool-3-thread-1
```

Exemple "TestFuture" (avec Executors et Callable<T>)

LongTask.java

```
package tp.langage.thread;

public class LongTask {

  public static void printThread(){
          System.out.println(Thread.currentThread().getName() );
  }

  public static void simulateLongTask(String msg,long nbMs){
          try {
                  System.out.println(">>(begin)" + msg + " / by " + Thread.currentThread().getName());
                  Thread.sleep(nbMs);
                  System.out.println("<<(end)" + msg + " / by " + Thread.currentThread().getName());
          } catch (InterruptedException e) {
                  //e.printStackTrace();
                  System.out.println("** interrupted **");
          }
  }
}
```

CallableComputing.java

```
package tp.langage.thread;
import java.util.concurrent.Callable;

public class CallableComputing implements Callable<String>{
          private double x;

          @Override
          public String call() throws Exception {
                  LongTask.simulateLongTask("long computing task (in background) ...", 5000);
                  return String.valueOf(Math.sqrt(x));
          }

          public CallableComputing() {super(); this.x = 0;    }
          public CallableComputing(double x) {super();this.x = x;     }

          public double getX() {     return x;          }
          public void setX(double x) {this.x = x;      }
}
```

EssaiFuture.java

```
package tp.langage.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class EssaisFuture {

  public static void main(String[] args) {
        //NB: Callable<T>/call() ressemble un peu à l'interface Running/run()
        //mais permet de récupérer (ultérieurement) un résultat via Future<T> .
        Callable<String> c = new CallableComputing(9);
        String result=null;

        ExecutorService executor = Executors.newSingleThreadExecutor();

      Future<String> futureRes = executor.submit(c);
      while(result==null){
            LongTask.simulateLongTask("other works ...",2000);
            if(futureRes.isDone()){
                try {
                        result = futureRes.get();
                } catch (InterruptedException e) {
                            e.printStackTrace();
                } catch (ExecutionException e) {
                            e.printStackTrace();
                }
            }
      }
      System.out.println("result=" + result);

      System.out.println("----------------------");
      Future<String> futureRes2 = executor.submit(c);
      LongTask.simulateLongTask("other works ...",2000);
      if(futureRes2.isDone()){
          try {
                result = futureRes2.get();
                System.out.println("result2=" + result);
          } catch (InterruptedException e) {
                e.printStackTrace();
          } catch (ExecutionException e) {
                e.printStackTrace();
          }
      }
      else {
```

```
            futureRes2.cancel(true);
            if(futureRes2.isCancelled()){
              System.out.println("background computing was cancelled");
            }
           }
       System.out.println("--------------------");
       Future<String> futureRes3 = executor.submit(c);
       LongTask.simulateLongTask("other works ...",2000);
       try {
           result = futureRes3.get(1500,TimeUnit.MILLISECONDS);
           System.out.println("result3=" + result);
           } catch (InterruptedException e) {
                    e.printStackTrace();
           } catch (ExecutionException e) {
                    e.printStackTrace();
           } catch (TimeoutException e) {
                    System.err.println("tâche 3 toujours pas terminée au bout de 1500ms");
                    System.out.println("after 1500ms,futureRes3.isDone()="+futureRes3.isDone());
                    System.out.println("after 1500ms,futureRes3.isCancelled()="+futureRes3.isCancelled());
                    //e.printStackTrace();      }
           }
          executor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat
}
```

Résultats :

```
>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
>>(begin)other works ... / by main
<<(end)other works ... / by main
>>(begin)other works ... / by main
<<(end)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
result=3.0
--------------------
>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
** interrupted **
background computing was cancelled
--------------------
>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
after 1500ms,futureRes3.isDone()=false
after 1500ms,futureRes3.isCancelled()=false
tâche 3 toujours pas terminée au bout de 1500ms
<<(end)long computing task (in background) ... / by pool-1-thread-1
```

Semaphore (Synchronisation de Threads)

Un **Semaphore** est (en java depuis le jdk 1.5) un objet technique de synchronisation entre différents threads .

Un **sémaphore** correspond conceptuellement à un **ensemble de jetons disponible** .

Un thread doit acquérir un jeton disponible (via un **appel bloquant** à semaphore.**acquire**() ou bien semaphore.**tryAcquire**(*timeout...*) ***pour pouvoir ensuite travailler seul sur une ressource partagée*** .

En appelant la méthode symétrique semaphore.**release()** , un thread peut rendre un jeton dans le semaphore (souvent après avoir terminé un certain travail en mode "exclusivité") . Cette action va immédiatement débloquer d'éventuelles attentes exprimées via semaphore.acquire() .

Plus précisément, un **sémaphore** encapsule un entier, avec une contrainte de positivité, et deux opérations atomiques d'incrémentation et de décrémentation :

- via le constructeur : variable entière (toujours positive ou nulle) ;
- opération -- (**acquire()**) : décrémente le compteur s'il est strictement positif ; bloque s'il est nul en attendant de pouvoir le décrémenter ;
- opération ++ (**release()**) : incrémente le compteur.

NB : Depuis java 1.5 , les **Semaphores** constituent un mécanisme de synchronisation **plus souple et plus fiable que** le mécanisme **.wait()/.notify()** disponible dès java 1.0 sur la classe Object .

EssaiSemaphore.java

```
package tp.langage.thread;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class EssaiSemaphore {
  public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(0);

        Thread t = new Thread(()-> { LongTask.simulateLongTask("background thread work ..." , 3000);
                                     semaphore.release();});
        t.start();
        System.out.println("... faire autre chose ...");

        //attendre la disponibilité du sémaphore:
        try {
                //semaphore.acquire();
                if(semaphore.tryAcquire(5, TimeUnit.MINUTES)){
                        System.out.println("semaphore acquis");
                }else{
                        System.out.println("after 5mn (semaphore toujours pas disponible)");
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
 }
```

```
}
```

Résultats :

```
... faire autre chose ...
>>(begin)background thread work ... / by Thread-0
<<(end)background thread work ... / by Thread-0
semaphore acquis
```

La notion de "***Mutex***" (mutuelle exclusion) peut s'implémenter en java avec **un sémaphore à un seul jeton** .

CompletionService (attendre et consommer résultats produits)

EssaiCompletionService.java

```
package tp.langage.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

//CompletionService depuis jdk1.6 avec méthode take() retournant un Future
//logique producteur/consommateur

public class EssaiCompletionService {
   public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(4); //jusqu'à 4 threads en //
        CompletionService<String> completionService =
                        new ExecutorCompletionService<String>(executor);

        double[] tabVal = { 4, 9 , 16 , 25, 36, 49 , 64, 81, 100 };
        int taille = tabVal.length;
        for(int i=0;i<taille;i++){
                Callable<String> c = new CallableComputing(tabVal[i]);
                //CompletionService<String> encapsule l'executor et est typé comme Future<T>
                completionService.submit(c); //lancement asynchrone d'un "producteur"
        }
        for(int i=0;i<taille;i++){
          try {
                Future<String> futureRes = completionService.take(); //attente du PREMIER TERMINE
                System.out.println(futureRes.get()); //consomateur simple
          } catch (Exception e) {
```

```
                        e.printStackTrace();
                        }
                }
        System.out.println("fin-main");
        //arrêter l'executor (si besoin en différé) :
        executor.shutdown();
    }
}
```

Résultats :

```
>>(begin)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-3
>>(begin)long computing task (in background) ... / by pool-1-thread-4
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)long computing task (in background) ... / by pool-1-thread-2
<<(end)long computing task (in background) ... / by pool-1-thread-4
<<(end)long computing task (in background) ... / by pool-1-thread-3
<<(end)long computing task (in background) ... / by pool-1-thread-1
>>(begin)long computing task (in background) ... / by pool-1-thread-3
3.0
4.0
2.0
>>(begin)long computing task (in background) ... / by pool-1-thread-1
>>(begin)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-4
5.0
<<(end)long computing task (in background) ... / by pool-1-thread-3
<<(end)long computing task (in background) ... / by pool-1-thread-1
<<(end)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-1
6.0
<<(end)long computing task (in background) ... / by pool-1-thread-4
8.0
7.0
9.0
<<(end)long computing task (in background) ... / by pool-1-thread-1
10.0
fin-main
```

2. ForkJoin

basic fork/join

```
public class EssaiBasicForkJoin {
	public static void main(String[] args) {
		System.out.println("debut - main");
		Thread t = new Thread(new MyRunnableCode());
		t.start(); // basic sort of fork()
		System.out.println(("suite - main / avant join"));
		try {
			t.join(); //attente de la fin de l'exécution du thread
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("fin main - apres join");
	}
}
-->
```

```
debut - main
suite - main / avant join
Thread-0
fin main - apres join
```

fork/join (depuis java 1.7)

Le framework fork / join en Java (depuis le jdk 1.7) est idéal pour un problème qui peut être divisé en parties plus petites et résolu en parallèle.

Les étapes fondamentales d'un problème fork / join sont les suivantes:

- **Diviser le problème en plusieurs morceaux**
- **Résoudre chacune des pièces en parallèle**
- **Combinez chacune des sous-solutions en une solution globale**

Une ForkJoinTask est l'interface qui définit un tel problème. On s'attend généralement à ce que vous sous-classiez l'une de ses implémentations abstraites (généralement la RecursiveTask) plutôt que d'implémenter l'interface directement.

Détail technique :

Le *ForkJoinPool* est le coeur du framework. Il s'agit d'une implémentation de *ExecutorService* qui gère les threads de travail et nous fournit des outils pour obtenir des informations sur l'état et les performances du pool de threads.

Les threads de travail ne peuvent exécuter qu'une tâche à la fois, mais *ForkJoinPool* ne crée pas de thread séparé pour chaque sous-tâche. Au lieu de cela, chaque thread du pool a sa propre file d'attente à deux extrémités (double ended *queue[deque]*) qui stocke les tâches.

Cette architecture est essentielle pour équilibrer la charge de travail du thread à l'aide de **l'algorithme "work-stealing" .**

fork/join appliqué sur un quickSort en mode "multi-processeurs"

MyQuickSortAlgo.java (version sans fork/join)

```
package tp.langage.thread;
public class MyQuickSortAlgo {

 static void echanger(double[] tableau ,int indice1 ,int indice2){
                double temp = tableau[indice1];
                        tableau[indice1] = tableau[indice2];
                        tableau[indice2] = temp;
            }

 static int  partition(double[] tableau,int deb,int fin){
     int indicePivot=deb; //au sens indice initial qui va évoluer
     double valeurPivot=tableau[deb];//valeur du pivot (= arbitrairement valeur en première position du tableau)
                                     //via une future permutation , cette valeur sera à une future autre position

     for(int i=deb+1;i<=fin;i++){
        if (tableau[i]<valeurPivot){
           indicePivot++; //nouvelle valeur pour le futur indice du pivot (qui peut encore évoluer selon boucle en cours)
           echanger(tableau,indicePivot,i); //pour placer à la "future gauche" de l'indice provisoire du pivot
                                            //tous les éléments plus petits que le pivot
         }
     }
     echanger(tableau,deb,indicePivot); //permutation pour que la valeur du pivot soit rangée à sa place (précédemment calculée)
                                        //et pour qu'un des éléments plus petits soit placé au début (à gauche )
     return indicePivot;
 }

  //version ordinaire (sans optimisation multi-proc):
    static void tri_rapide(double[] tableau,int deb,int fin){
      if(deb<fin){
           //partitionner le tableau en 2 parties partiellement ré-arrangées .
           //d'un coté tous les éléments plus petits que le pivot , de l'autre coté tous les éléments plus grands:
        int positionPivot=partition(tableau,deb,fin);
        tri_rapide(tableau,deb,positionPivot-1); //trier le sous tableau des plus petits éléments que le pivot
        tri_rapide(tableau,positionPivot+1,fin); //trier le sous tableau des plus grands éléments que le pivot
        }
     }

    static void quick_sort(double[] tableau){
                        tri_rapide(tableau, 0, tableau.length - 1 );
            }
}
```

MyQuickSortMultiProc.java (avec fork/join)

```
package tp.langage.thread;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
```

```
//ForkJoin (RecursiveAction) / divide & conquer since jdk 1.7
public class MyQuickSortMultiProc extends RecursiveAction {

        private static final long serialVersionUID = 1L;
        private static final int FORK_JOIN_MIN_SIZE=1024;

        private double[] tab;
        private int start,end;

        public MyQuickSortMultiProc(double[] tableau,int deb,int fin){
                this.tab = tableau;
                this.start = deb;
                this.end = fin;
        }

  static void quick_sort_multiProc(double[] tableau){
         MyQuickSortMultiProc myQuickSortMultiProc=
                        new MyQuickSortMultiProc(tableau, 0, tableau.length - 1);
         ForkJoinPool threadPool = new ForkJoinPool();
         threadPool.invoke(myQuickSortMultiProc);
    }

 @Override //RecursiveAction
 protected void compute() {
        // pas de paramètre et donc les tab et indices
        // doivent être renseignés en tant qu'attributs + constructeurs
        MyQuickSortMultiProc sousTriGaucheViaForkJoin=null;
        MyQuickSortMultiProc sousTriDroitViaForkJoin = null;
        //System.out.println("MyQuickSortMultiProc.compute() executé par "+Thread.currentThread().getName() );

        if(start<end){
           //partitionner le tableau en 2 parties partiellement ré-arrangées .
           //d'un coté tous les éléments plus petits que le pivot , de l'autre coté tous les éléments plus grands:
          int positionPivot=MyQuickSortAlgo.partition(tab,start,end);

       //NB: étant donné que la version forkJoin ajoute une complexité au niveau du code
       // (instance à créer , thread à gérer) , cette version ne sera activée/utilisée
       //que pour trier des sous tableaux dont la taille minimum est supérieure à
       //FORK_JOIN_MIN_SIZE=1024

       if(positionPivot - start > FORK_JOIN_MIN_SIZE){
          sousTriGaucheViaForkJoin= new MyQuickSortMultiProc(tab, start, positionPivot-1);
          sousTriGaucheViaForkJoin.fork(); //déléguer le tri du sous tableau des plus petits éléments que le pivot
       } else MyQuickSortAlgo.tri_rapide(tab,  start, positionPivot-1);

       if(end - positionPivot > FORK_JOIN_MIN_SIZE){
          sousTriDroitViaForkJoin= new MyQuickSortMultiProc(tab, positionPivot+1,end);
          sousTriDroitViaForkJoin.fork(); //déléguer le tri du tableau des plus grands éléments que le pivot
       } else MyQuickSortAlgo.tri_rapide(tab, positionPivot+1, end);
```

```
        if(sousTriGaucheViaForkJoin!=null) sousTriGaucheViaForkJoin.join(); //attendre
        if(sousTriDroitViaForkJoin!=null) sousTriDroitViaForkJoin.join(); //attendre

        //NB: il existe invokeAll(recursiveAction1 , recursiveAction2) qui declenche en // .fork() et .join()
        }
    }
}
```

TestMultiProcesseurs.java (tests avec rapidités comparées)

```
package tp.langage.thread;
public class TestMultiProcesseurs {
    public static void main(String[] args) {
            System.out.println("nb processors:" + Runtime.getRuntime().availableProcessors());
            double[] t1 = produce_init_tab();
            double[] copyOfT1 = t1.clone();
            display_tab(t1);   //display_tab(copyOfT1);
            System.out.println("tri ordinaire (quick-sort) ");
            test_tri(t1);
            System.out.println("tri (quick-sort) optimisé pour machine multi-processeurs ");
            test_tri_multiProc(copyOfT1);
    }

    static double[] produce_init_tab(){
            //double[] t = { 5,2,1,9,3,4,12,8,16,6 };
            //double[] t = { 26,7,5,2,1,9,3,4,34,12,8,16,6,78,10,89,33,23,90,123,72,3,48 };
            //final int taille=10;
             final int taille=1024*1024*8;
            double[] t = new double[taille];
                for(int i=0;i<taille;i++){
                            t[i]=Math.random()*taille;
                    }
            return t;
     }

     static void display_tab(double[] tab){
                    if(tab.length <= 30) {
```

```
                for(double x : tab)
                        System.out.print(x + " ");
                System.out.print("\n");
              } else{   System.out.println("tableau de taille = " + tab.length);
                }
        }

        static void test_tri(double[] tab){
                long td = System.nanoTime();
                MyQuickSortAlgo.quick_sort(tab);
                long tf = System.nanoTime();      display_tab(tab);
                System.out.println("## " + (tf-td)/ 1000000 + " ms");
         }

        static void test_tri_multiProc(double[] tab){
                long td = System.nanoTime();
                MyQuickSortMultiProc.quick_sort_multiProc(tab);
                long tf = System.nanoTime();          display_tab(tab);
                System.out.println("** " + (tf-td) / 1000000 + " ms");
         }
}
```

Résultats (avec **i7**):

```
nb processors:4
tableau de taille = 8388608
## tri ordinaire (quick-sort)
tableau de taille = 8388608
## 872 ms
** tri (quick-sort) optimisé pour machine multi-processeurs
tableau de taille = 8388608
** 538 ms
```

Autre exemple de fork/join (calcul de moyenne ou d'écartType) .

SequentialComputing.java

```
package tp.thread.sam;
//interface d'une reférence de fonction pour calcul ordinaire de somme ou moyenne ou  ...
public interface SequentialComputing {
        //start et end sont les indices sur la plage du tableau à manipuler .
        //arg = null ou éventuel argument nécessaire à un  calcul (ex: arg=moyenne pour calcul de variance)
        double basicCompute(double[] numbers, int start, int end,Double arg); //sum or average or ....
}
```

ResultAggregate.java

```
package tp.thread.sam;
//interface d'une reférence de fonction pour recombiner 2 sous sommes ou 2 sous moyennes
public interface ResultAggregate {
        Double composeTotalRes(double res_tab1, int taille_tab1,double res_tab2, int taille_tab2);
}
```

MyStatsAlgo.java

```
package tp.thread;

import tp.thread.sam.ResultAggregate;
import tp.thread.sam.SequentialComputing;

//calcul de somme , moyenne ou variance sur grands tableaux
public class MyStatsAlgo {
        public static final int DECOMP_MIN_SIZE=1024*50;

        public static final Double composeMoyenneTotale(double moyenne_tab1, int taille_tab1,
                                                        double moyenne_tab2, int taille_tab2) {
                return (moyenne_tab1*taille_tab1 + moyenne_tab2*taille_tab2) / (taille_tab1 + taille_tab2);
        }


         //versions ordinaires (sans decomposition en 2 sous parties):

        static Double moyenne_ordinaire_subPart(double[] tableau,int deb,int fin,Double notUsedArg){
            Double somme = 0.0;
            for(int i=deb; i<=fin ;i++) {
                    somme += tableau[i];
            }
            int sizeSubPart = (fin - deb)+1;
            return somme / sizeSubPart;
         }




         static Double variance_ordinaire_subPart(double[] tableau,int deb,int fin,Double moyenne){
            Double varianceFoisN = 0.0;
             for(int i=deb; i<=fin ;i++) {
```

```
                    varianceFoisN += (tableau[i] - moyenne) * (tableau[i] - moyenne);
                }
                int sizeSubPart = (fin - deb)+1;
                return varianceFoisN/sizeSubPart;
            }

  //versions avec decomposition en 2 sous parties:

    static double moyenne_decomp_subPart(double[] tableau,int deb,int fin){
            return compute_decomp_subPart(tableau,deb,fin,null,
                                    MyStatsAlgo::moyenne_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
    }

    static double variance_decomp_subPart(double[] tableau,int deb,int fin,double moyenne){
            return compute_decomp_subPart(tableau,deb,fin,moyenne,
                                    MyStatsAlgo::variance_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
    }

   static double compute_decomp_subPart(double[] tableau,int deb,int fin,Double arg,
                                SequentialComputing seqComptuting , ResultAggregate resAggregate){
      double resSp;
      int sizeSubPart = (fin - deb)+1;
      if(sizeSubPart >= DECOMP_MIN_SIZE){
            int indiceMilieu = deb + (sizeSubPart / 2);
            double resCalculSousPartie1 = compute_decomp_subPart(tableau,deb,indiceMilieu-1,arg,seqComptuting,resAggregate);
            double resCalculSousPartie2 = compute_decomp_subPart(tableau,indiceMilieu,fin,arg,seqComptuting,resAggregate);
                        resSp = resAggregate.composeTotalRes(resCalculSousPartie1 , (indiceMilieu - deb) ,
                                                                resCalculSousPartie2 , (fin - indiceMilieu +1) );
         }
      else
              resSp = seqComptuting.basicCompute(tableau,deb,fin,arg);
      return resSp;
      }

   //fonctions de niveau principal (appels simples , niveau global):
    static double moyenne_ordinaire(double[] tableau){
            return moyenne_ordinaire_subPart(tableau, 0, tableau.length - 1 , null );
             }

    static double ecartType_ordinaire(double[] tableau){
            double moyenne = moyenne_ordinaire_subPart(tableau, 0, tableau.length - 1 , null );
            double variance =  variance_ordinaire_subPart(tableau, 0, tableau.length - 1 , moyenne );
               return Math.sqrt(variance);
             }


    static double moyenne_decomp(double[] tableau){
            return moyenne_decomp_subPart(tableau, 0, tableau.length - 1 );
             }

    static double ecartType_decomp(double[] tableau){
            double moyenne = moyenne_decomp_subPart(tableau, 0, tableau.length - 1 );
            double variance =  variance_decomp_subPart(tableau, 0, tableau.length - 1 , moyenne);
               return Math.sqrt(variance);
             }
}
```

MyStatsMultiProc.java

```
package tp.thread;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

import tp.thread.sam.ResultAggregate;
import tp.thread.sam.SequentialComputing;

public class MyStatsMultiProc extends RecursiveTask<Double> {
        private static final int FORK_JOIN_MIN_SIZE=1024*50;
                        // THRESHOLD/SEUIL à éventuellement ajuster selon complexité du calcul
        private double[] tab;
        private int start,end;
        private SequentialComputing seqComputing; //référence de fonction pour calcul ordinaire de somme ou moyenne ou  ...
        private ResultAggregate resAggregate; //référence de fonction pour recombiner 2 sous sommes ou 2 sous moyennes
        private Double arg; //null ou éventuel argument nécessaire à un  calcul (ex: arg=moyenne pour calcul de variance)


        public MyStatsMultiProc(double[] tableau,int deb,int fin , Double arg,
                        SequentialComputing seqComputing , ResultAggregate resAggregate){
                this.tab = tableau;   this.start = deb;      this.end = fin;         this.arg = arg;
                this.seqComputing = seqComputing;     this.resAggregate = resAggregate;
        }

    static double moyenne_multiProc(double[] tableau){
        MyStatsMultiProc myStatsMultiProc= new MyStatsMultiProc(tableau, 0, tableau.length – 1,
                        null,MyStatsAlgo::moyenne_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
        ForkJoinPool threadPool = new ForkJoinPool();
        return threadPool.invoke(myStatsMultiProc);
         }

    static double ecartType_multiProc(double[] tableau){
        double moyenne = moyenne_multiProc(tableau);
        MyStatsMultiProc myStatsMultiProc= new MyStatsMultiProc(tableau, 0, tableau.length – 1,
                        moyenne,MyStatsAlgo::variance_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
        ForkJoinPool threadPool = new ForkJoinPool();
        double variance =  threadPool.invoke(myStatsMultiProc);
        return Math.sqrt(variance);
         }

        @Override
        protected Double compute() {
                Double res =0.0;
                //System.out.println("MyStatsMultiProc.compute() executé par "+Thread.currentThread().getName() );

                //NB: etant donné que la version forkJoin ajoute une complexité au niveau du code
                // (instance à créer , thread à gérer) , cette version ne sera activée/utilisée
                //que pour traiter des sous tableaux dont la taille minimum est supérieure à FORK_JOIN_MIN_SIZE

                int sizeSubPart = (end - start)+1;

          if(sizeSubPart >= FORK_JOIN_MIN_SIZE) {
                  int indiceMilieu = this.start + (sizeSubPart / 2);
                  // pas de parametre dans .compute() et donc les tab et indices doivent être renseignés
                  //en tant qu'attributs + constructeurs
                  MyStatsMultiProc sousCalculGaucheViaForkJoin=
                          new MyStatsMultiProc(tab,start,indiceMilieu-1,arg,seqComputing,resAggregate);
                  MyStatsMultiProc sousCalculDroitViaForkJoin=
                          new MyStatsMultiProc(tab,indiceMilieu,end,arg,seqComputing,resAggregate);
```

```
                    sousCalculGaucheViaForkJoin.fork();//déléguer (via potentiel autre thread)

                    //sous solution A (.compute() ) :
                    //Double resCalculSousPartie2 = sousCalculDroitViaForkJoin.compute();//faire soit même (via meme thread)

                    //sous solution B (.fork/join ) :
                    sousCalculDroitViaForkJoin.fork();//déléguer (via potentiel autre thread)
                    Double resCalculSousPartie2 = sousCalculDroitViaForkJoin.join(); //attendre

                    Double resCalculSousPartie1 = sousCalculGaucheViaForkJoin.join(); //attendre

                    res = resAggregate.composeTotalRes(resCalculSousPartie1 , (indiceMilieu - start) ,
                                                       resCalculSousPartie2 , (end - indiceMilieu +1) );
            }else {
                    res = seqComputing.basicCompute(this.tab,this.start,this.end,this.arg);
            }
            return res;
        }
}
```

Exemple de performances comparées (sur un petit i7) :

nb processors:4
tableau de taille = 134217728
moyenne ordinaire (sans decomposition)
##MOYENNE=500.0102357180581
601 ms
$$ moyenne avec decomposition
$$MOYENNE=500.01023571807764
$$ **174 ms**
** moyenne avec decomposition optimise pour machine multi-processeurs (fork/join)
**MOYENNE=500.01023571807764
** **64 ms**

3. CompletableFuture , streams asynchrones

CompletableFuture<T> disponible *depuis le jdk 1.8* est une version améliorée de Future<T> pour **enchaîner certains traitements asynchrones** et qui comporte *quelques similitudes avec les "callback" et "Promise.then()" de javascript* .

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

dans le package **java.util.concurrent** .

Dans les grandes lignes , **la principale valeur ajoutée de CompletableFuture<T>** vis à vis de Future<T> réside dans l'**implémentation de l'interface CompletionStage<T>** de manière à **bien resynchroniser / ordonner différentes tâches asynchrones qui seront par nature exécutées en différé** (dans le futur) .

Autrement dit , un bloc d'instructions de ce type :

```
CompletableFuture.supplyAsync(traitementAsynchrone1 )
                 .thenApply( traitementAsynchrone2 )
                 .thenApply(traitementAsynchrone3)
                 .thenAccept( traitementAsynchroneFinal );
```

correspond à enregistrement de tâches asynchrones à effectuer en différé dès que possible .
Le future résultat du *traitementAsynchrone1* constituera l'entrée de la tâche *traitementAsynchrone2* qui ne pourra alors être démarrée que lorsque la tâche *traitementAsynchrone1* sera terminée et ainsi de suite via cet enchaînement de .thenApply() .then... ()

Rappel :

- l'interface (uni-)fonctionnelle **Supplier<T>** comporte l'unique méthode **T .get()** et correspond à un producteur ou *fournisseur de valeur de Type T* .
- l'interface (uni-)fonctionnelle **Function<T,R>** comporte la méthode **R .apply(T t)** et correspond à une *fonction appliquée à un unique paramètre de type T et retournant R* .
- l'interface (uni-)fonctionnelle **Consumer<T>** comporte la méthode **void .accept(T t)** et correspond à un *consommateur d'élément de type T (sans valeur produite)* .

Principales méthodes de CompletableFuture<T> et CompletionStage<T> :

NB1 : l'abréviation **CF<T>** est à comprendre comme CompletableFuture<T>

NB2 : Sans précision de l'executor (souvent en tant que dernier paramètre facultatif des méthodes surchargées) , la classe CompletableFuture lance en interne les tâches asynchrones via **ForkJoinPool.commonPool()** par défaut .

NB3 : Au sein de ce tableau , les méthodes en ***italiques*** seront à considérées comme "**static**" .

et <T,R> est généralement à interpréter comme < ? super T , ? extends R>

CF<Void> ***runAsync***(Runnable r)	retourne un CF<T> correspondant à la future exécution asynchrone de la tâche r (Runnable)
CF<U> ***supplyAsync***(Supplier<U> s)	retourne un CF<U> correspondant à la future exécution asynchrone de la tâche s (fournissant une valeur de type U)
CF<R> **thenApply**(Function<T,R> f)	Souvent en milieu d'enchaînement, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<R> correspondant à la future exécution asynchrone de la tâche fonctionnelle f
CF<R> **thenCompose**(Function<T,**CF<R>**> f)	Un peu comme thenApply() mais en précisant une fonction qui (immédiatement, sans attente) construit elle même une instance de CF <R> qui sera consommée dans la suite de l'enchaînement
CF<T> **exceptionally**(Function<Throwable ex,T> f)	Souvent en milieu d'enchaînement, transforme une exception potentielle en élément de type T pour le bon déroulement de la suite des enchaînements, retransmet la valeur inchangée si pas d'exception .
CF<R> **handle**(BiFunction<T, Throwable,R> f)	Combine le comportement de thenApply() et exceptionally() via une tâche asynchrone prenant 2 argument en entrée : value et exception .
CF<Void> **thenAccept**(Consumer<T> c)	En terminaison, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<Void> correspondant à la future exécution asynchrone de la tâche terminale c (Consumer avec argument en entrée)
CF<Void> **thenRun**(Runnable r)	Souvent en terminaison, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<Void> correspondant à la future exécution asynchrone de la tâche terminale r (Runnable sans argument en entrée)
boolean **complete**(T value) et boolean **completeExceptionally**(Throwable ex)	si tâche pas terminée , fixe le résultat de cette tâche qui sera ultérieurement récupérée par .get().

Remarque : si cf.***thenApplyAsync***(...); à la place de cf.*thenApply*(...); alors tâche quelquefois exécutée par encore un autre thread.

Premiers exemples de CompletableFuture

EssaiAsyncJava8.java

```
package tp.langage.thread;

import java.util.concurrent.CompletableFuture;   import java.util.function.Supplier;

public class EssaiAsyncJava8 {
    public static void main(String[] args) {
        System.out.println("debut main / interpreted by " + Thread.currentThread().getName());

        //initialisation asynchrone:
        //CompletableFuture<Void> cf = CompletableFuture.runAsync(aRunnableObject); //with no return result !!!
        CompletableFuture<Double> completableFuture1 =
                /*CompletableFuture.supplyAsync( new Supplier<Double>(){
                        public Double get(){ LongTask.simulateLongTask("long computing - p1" ,  2000);
                                return 2.0; }
                        });*/
        CompletableFuture.supplyAsync( ()-> { LongTask.simulateLongTask("long ...- p1" ,  2000);
                        /*throw new RuntimeException("exceptionXY");*/ return 2.0; } );

        //En cas d'exception en asynchrone/tâche de fond:
        CompletableFuture<Double> safecompletableFuture1 =
            completableFuture1.exceptionally(ex -> { System.out.println("problem: " +
                                        ex.getMessage());     return 0.0; } );
        /*CompletableFuture<Double> safecompletableFuture1 =
                completableFuture1.handle((resOk,ex) -> {
                        if(resOk!=null) return resOk;
                        else { System.out.println("problem: " + ex.getMessage()); return 0.0;} } );*/

        System.out.println("suite main A / interpreted by " + Thread.currentThread().getName());

        // continuations asynchrones (avec Function<T1,T2>):
        CompletableFuture<Double> completableFuture2 =
          safecompletableFuture1.thenApply((x) -> { LongTask.simulateLongTask("long computing -
                                        p2" ,  2000); return x*x; } );

        CompletableFuture<String> completableFuture3 =
          completableFuture2.thenApply((x) -> { LongTask.simulateLongTask("long computing - p3" ,
                                        2000); return String.valueOf(x); } );

        System.out.println("suite main B / interpreted by " + Thread.currentThread().getName());
        //fin/terminaison asynchrone:
        completableFuture3.thenAccept((x) -> System.out.println(x) );
        //computableFuture.thenRun(()->System.out.println("ok"));//with no input !!!
```

```
        LongTask.simulateLongTask("pause pour eviter arrêt complet du programme " +
                                  " avant la fin des taches de fond" ,  8000);
        System.out.println("fin main / interpreted by " + Thread.currentThread().getName());
        }
}
```

Résultats :

```
debut main / interpreted by main
suite main A / interpreted by main
>>(begin)long computing - p1 / by ForkJoinPool.commonPool-worker-3
suite main B / interpreted by main
>>(begin)pause pour eviter arrêt complet du programme avant la fin des taches de fond / by main
<<(end)long computing - p1 / by ForkJoinPool.commonPool-worker-3
>>(begin)long computing - p2 / by ForkJoinPool.commonPool-worker-3
<<(end)long computing - p2 / by ForkJoinPool.commonPool-worker-3
>>(begin)long computing - p3 / by ForkJoinPool.commonPool-worker-3
<<(end)long computing - p3 / by ForkJoinPool.commonPool-worker-3
4.0
<<(end)pause pour eviter arrêt complet du programme avant la fin des taches de fond / by main
fin main / interpreted by main
```

Variation syntaxique (générant le même résultat) :

EssaiAsyncJava8V2.java

```
package tp.langage.thread;

import java.util.concurrent.CompletableFuture;

public class EssaiAsyncJava8V2 {

        public static Double extractInitValue(){
                LongTask.simulateLongTask("long computing - p1" ,  2000);
                /*throw new RuntimeException("exceptionXY");*/ return 2.0;
        }

        public static Double auCarre(Double x){
                 LongTask.simulateLongTask("long computing - p2" ,  2000); return x*x;
        }

        public static String convertAsString(Double x){
                LongTask.simulateLongTask("long computing - p3" ,  2000); return String.valueOf(x);
        }

        public static void displayString(String s){
                 System.out.println(s) ;
        }
```

```
        public static void main(String[] args) {
                System.out.println("debut main / interpreted by " + Thread.currentThread().getName());

                CompletableFuture.supplyAsync(EssaiAsyncJava8V2::extractInitValue )
                                 .exceptionally(ex -> { System.out.println("problem: " +
                                                        ex.getMessage()); return 0.0; } )
                                .thenApply(EssaiAsyncJava8V2::auCarre )
                                .thenApply(EssaiAsyncJava8V2::convertAsString)
                                .thenAccept(EssaiAsyncJava8V2::displayString );

                System.out.println("suite main / interpreted by " + Thread.currentThread().getName());
                LongTask.simulateLongTask("pause pour eviter arrêt complet du programme" +
                                          " avant la fin des taches de fond" ,  8000);
                System.out.println("fin main / interpreted by " + Thread.currentThread().getName());
        }
}
```

Autre variation syntaxique avec this et lambda expressions :

EssaiAsyncJava8V2Bis.java

```
package tp.langage.thread;
import java.util.concurrent.CompletableFuture;
public class EssaiAsyncJava8V2Bis {
        public static void simuLong(String number,long nbMs) {
                LongTask.simulateLongTask("long computing - p"+number ,  nbMs);
        }
        private double initialXValue;
        private String result;

   public void noStaticMethodWithLambda() {
         this.initialXValue=2;
        //code am"liorable avec synchronized(this){ this... }
        CompletableFuture.supplyAsync(()->{ simuLong("p1",2000);
                                            return this.initialXValue; } )
                         .thenApply(  (x)->{ simuLong("p2",2000); return x*x; } )
                         .thenApply(  (x)->{ simuLong("p3",2000); return String.valueOf(x); })
                         .thenAccept( (s)->{System.out.println(s); this.result=s;});
        LongTask.simulateLongTask("pause pour eviter arrêt complet du programme"
                                  + " avant la fin des taches de fond" ,  8000);
        System.out.println("result="+this.result);
        }
```

```
        public static void main(String[] args) {
                EssaiAsyncJava8V2Bis thisApp = new EssaiAsyncJava8V2Bis();
                thisApp.noStaticMethodWithLambda();
        }
}
```

result=4

Combinaisons avec CompletableFuture

Exemple partiel : ***EssaiAsyncJava8V3.java***

```
package tp.langage.thread;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

public class EssaiAsyncJava8V3 {

        //plein de code commum avec  EssaiAsyncJava8V2 (pas répété ici)

        public static Double plus(Double x,Double y){
                LongTask.simulateLongTask("long computing - plus" ,  2000); return x+y;
        }

       public static CompletionStage<Double> cfInitVal(){
                return CompletableFuture.supplyAsync(EssaiAsyncJava8V3::extractInitValue );
        }

        //méthode static pour pour then.compose(...) :
        public static CompletionStage<String> cfAsString(Double x){
                CompletableFuture<Double> cfDouble = new CompletableFuture<Double>();
                cfDouble.complete(x);
                System.out.println("**** tout le debut de cfAsString sans attente ****");
                return cfDouble.thenApplyAsync(EssaiAsyncJava8V3::convertAsString);
        }

   public static void main(String[] args) {
        System.out.println("debut main de EssaiAsyncJava8V3");
        //thenCompose register another future to apply/compose to thisFuture (without waiting)
        // to produce a new future :
        cfInitVal().thenCompose(EssaiAsyncJava8V3::cfAsString)
                   .thenAccept(EssaiAsyncJava8V3::displayString );

        CompletableFuture<Double> cfD1 = new CompletableFuture<Double>();
        cfD1.complete(3.0); //COMPLETE BY MAIN --> NEED .thenApplyAsync() to be continued by other thread
```

```
        CompletableFuture<Double> cfD2 = new CompletableFuture<Double>();
        cfD2.complete(2.0); //COMPLETE BY MAIN --> NEED .thenApplyAsync() to be continued by other thread
        System.out.println("suite du main de EssaiAsyncJava8V3");
        CompletableFuture<Double> cfD3 = cfD1.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//3*3=9
        CompletableFuture<Double> cfD4 = cfD2.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//2*2=4

        //thenCombine apply a biFunction from 2 futures (this & other) to produce a new future

         CompletableFuture<Double> cfD5 =
                        cfD3.thenCombine(cfD4, EssaiAsyncJava8V3::plus);//9+4=13
        cfD5.thenAccept((x)->System.out.println(x));


        CompletableFuture<Double> cfD6 = cfD1.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//3*3=9
        CompletableFuture<Double> cfD7 = cfD2.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//2*2=4
        cfD6.thenAcceptBoth(cfD7, (x,y) -> System.out.println(x*y));//9*4 = 36

        LongTask.simulateLongTask("pause pour éviter arrêt complet du programme "
                                        + "avant la fin des taches de fond" ,  8000);
        System.out.println("fin main de EssaiAsyncJava8V3 / interpreted by " + Thread.currentThread().getName());
        }
}
```

Autres combinaisons :

Variantes proches de thenCombine :

.thenAcceptBoth(otherFuture, biConsumer) --> action terminale (sans valeur calculée ni retournée) avec valeurs en entrées lorsque les 2 futures sont terminés

.runAfterBoth(otherFuture, runnable) --> action sans valeur en entrée lorsque les 2 futures sont terminés

Variantes proches de thenAcceptBoth/runAfterBoth :

.thenAcceptEither(otherFuture, consumer) --> action avec valeur en entrées lorsque le premier des 2 futures est terminé

.runAfterEither(otherFuture, runnable) --> action sans valeur en entrée lorsque le premier des 2 futures est terminé

Variante proche de thenAcceptEither :

thenApplyEither(otherFuture , function) --> génère un nouveau future (avec valeur) pour poursuivre un enchaînement avec thenApply() ou autre.

Variantes avec nombres d'arguments variables:

.allOf(...) pour attendre la fin de n(3 ou plus) futures

.anyOf(...) pour attendre la fin du plus rapide parmi n