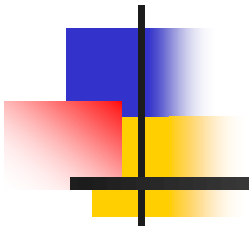


INF2440 Uke 9, v2014 :

Arne Maus
OMS,
Inst. for informatikk

Hva har vi sett på i uke 8:

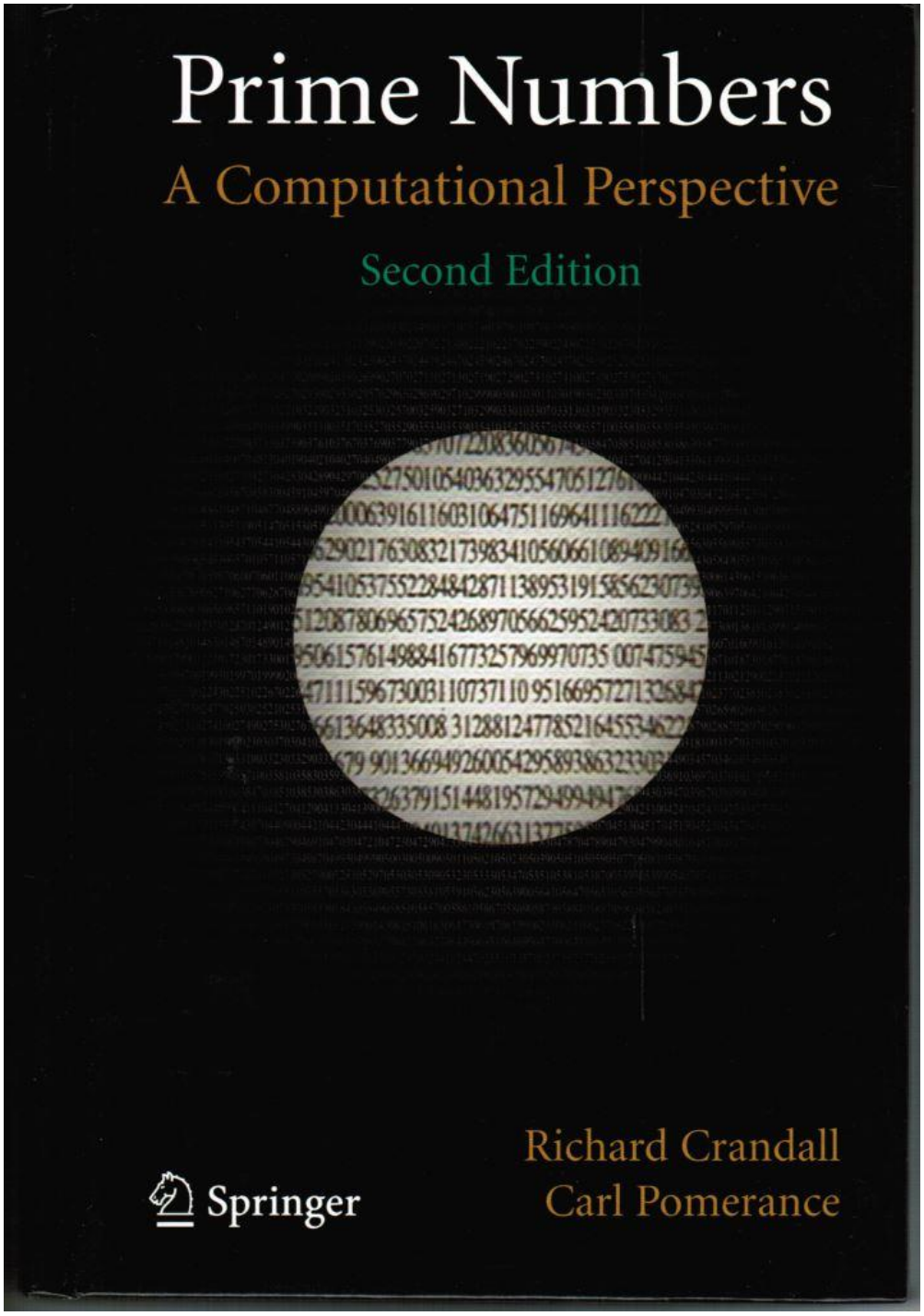
1. En effektiv Threadpool ?
   - Executors.newFixedThreadPool
2. Mer om effektivitet og JIT-kompilering !
3. Om et problem mellom long og int
4. Presisering av det å faktorisere ethvert tall M
5. Om parallellisering av
   - Laging og lagring av Eratosthenes Sil
     - To alternativer
   - Faktorisering av ethvert tall M < N*N
     (finne de primtallene < N som ganget sammen gir M).
     - To alternativer
6. Utsettelse av innleveringsfristen for Oblig2 en uke.

Hva skal vi se på i uke 9

- Et sitat om tidsforbruk ved faktorisering
- En presisering av Oblig2.
- Om en feil i Java ved tidtaking
- Hvor lang tid tar de ulike mekanismene vi har i Java ?
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger

Sitat fra denne boka om hvor lang tid det tar å finne ut om et 19-sifret tall er primtall ved divisjon.
Denne boka hevder 1 døgn, vi skriver program som gjør det på ca. 1-3 sek – eller : 24*60*60 = 86 400 x fortere

Og selv om boka deler med all oddetall $<\sqrt{N}$ og ikke bare primtallene $<\sqrt{N}$, skulle det bare gå ca. 10x langsommere (fordi ca 10% av alle oddetall $<\sqrt{N}$, er primtall).



3.1.3 Practical considerations

It is perfectly reasonable to use trial division as a primality test when n is not too large. Of course, "too large" is a subjective quality; such judgment depends on the speed of the computing equipment and how much time you are willing to allow a computer to run. It also makes a difference whether there is just the occasional number you are interested in, as opposed to the possibility of calling trial division repeatedly as a subroutine in another algorithm. On a modern workstation, and very roughly speaking, numbers that can be proved prime via trial division in one minute do not exceed 13 decimal digits. In one day of current workstation time, perhaps a 19-digit number can be resolved. (Although these sorts of rules of thumb scale, naturally, according to machine performance in any given era.)

Krav til løsning på Oblig2

- Hvis du ikke får speedup > 1 på noen av algoritmene, skal du selvsagt levere og kommentere hvorfor du tror det gikk slik. Koden skal kompilere og kjøre.
- Siden oblig2 består i å parallelliser faktoriseringen av 100 tall, så kunne man tenke seg at man:
  - Parallelliserte dannelsen av Eratosthenes Sil og så:
  - kjøre den sekvensielle faktoriseringen av de 100 tallene i parallell, slik at tråd-0 tok de 100/k første tallene, tråd-1 de neste 100/k tallene,..osv
  - Da har man ikke parallellisert de to algoritmene, men den første + Oblig2. Det er ikke det var ikke meningen (eller teksten) i oppgaven.
  - En slik parallellisering av Oblig2 vil ikke bli godkjent selv om den tar like kort (eller kortere) tid.
  - Hvorfor det ?

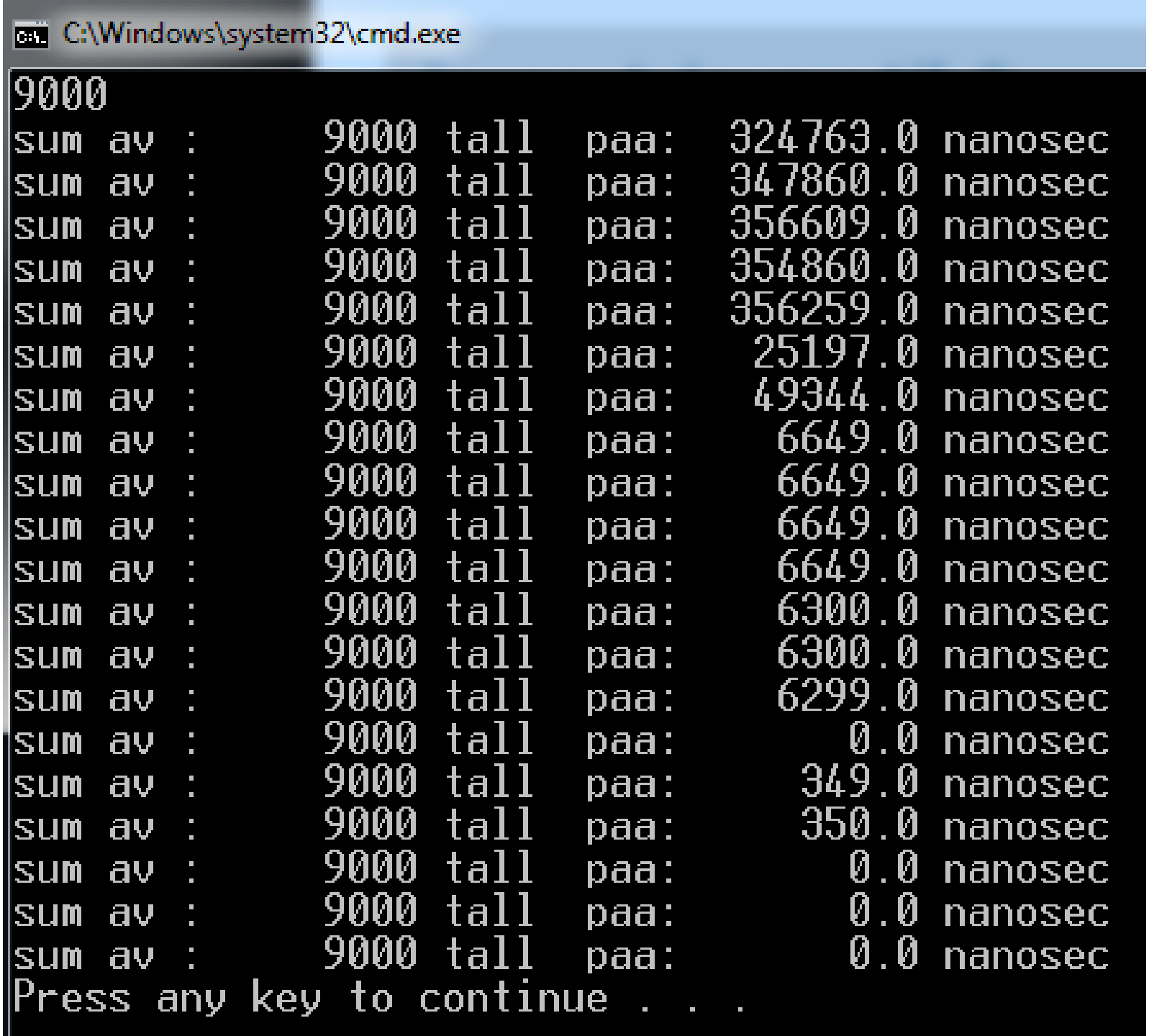
2) Merkelig feil (?) i Java tidtaking ?

- Av og til blir kjøretidene == 0 – hvorfor ?
- Eller veldig få nanoSekunder – hvorfor ?

```
import java.util.*;
import easyIO.*;
class FinnSum{
  public static void main(String[] args){
        int len = new In().inInt();
        FinnSum fs =new FinnSum();

        for(int k = 0; k < 20; k++){
              int[] arr = new int[len];
              Random r = new Random();
              for(int i = 0; i < arr.length; i++){
                    arr[i] = r.nextInt(len-1);
              }
             long start  = System.nanoTime();
             long sum = fs.summer(arr);
             long timeTaken = System.nanoTime() - start ;

            System.out.println( //"SUM:"+ sum +
               Format.align(k+1,2)+ ") kjoring; sum av:"+
                Format.align(len,9)+" tall  paa:" +
                Format.align(timeTaken,10) + " nanosec");
   }} // end main

  long summer(int [] arr){
              long sum = 0;
              for(int i = 0; i < arr.length; i++)
                          sum += arr[i];
             return sum;
  } } // end summer
```

kjøres:
>java FinnSumFeil

og kjører man dette med n=9000
ser du de ulike skrittene i JIT-kompilering og en feil.


C:\Windows\system32\cmd.exe
9000
sum av : 9000 tall paa: 324763.0 nanosec
sum av : 9000 tall paa: 347860.0 nanosec
sum av : 9000 tall paa: 356609.0 nanosec
sum av : 9000 tall paa: 354860.0 nanosec
sum av : 9000 tall paa: 356259.0 nanosec
sum av : 9000 tall paa: 25197.0 nanosec
sum av : 9000 tall paa: 49344.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6300.0 nanosec
sum av : 9000 tall paa: 6300.0 nanosec
sum av : 9000 tall paa: 6299.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 349.0 nanosec
sum av : 9000 tall paa: 350.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
Press any key to continue . . .

Dear Java Developer,

Thank you for reporting this issue.

We have determined that this report is a new bug and have entered the bug into our bug tracking system under Bug Id: 9006037. You can look for related issues on the Java Bug Database at http://bugs.sun.com.

We will try to process all newly posted bugs in a timely manner, but we make no promises about the amount of time in which a bug will be fixed. If you just reported a bug that could have a major impact on your project, consider using one of the technical support offerings available at Oracle Support.

Thanks again for your submission!

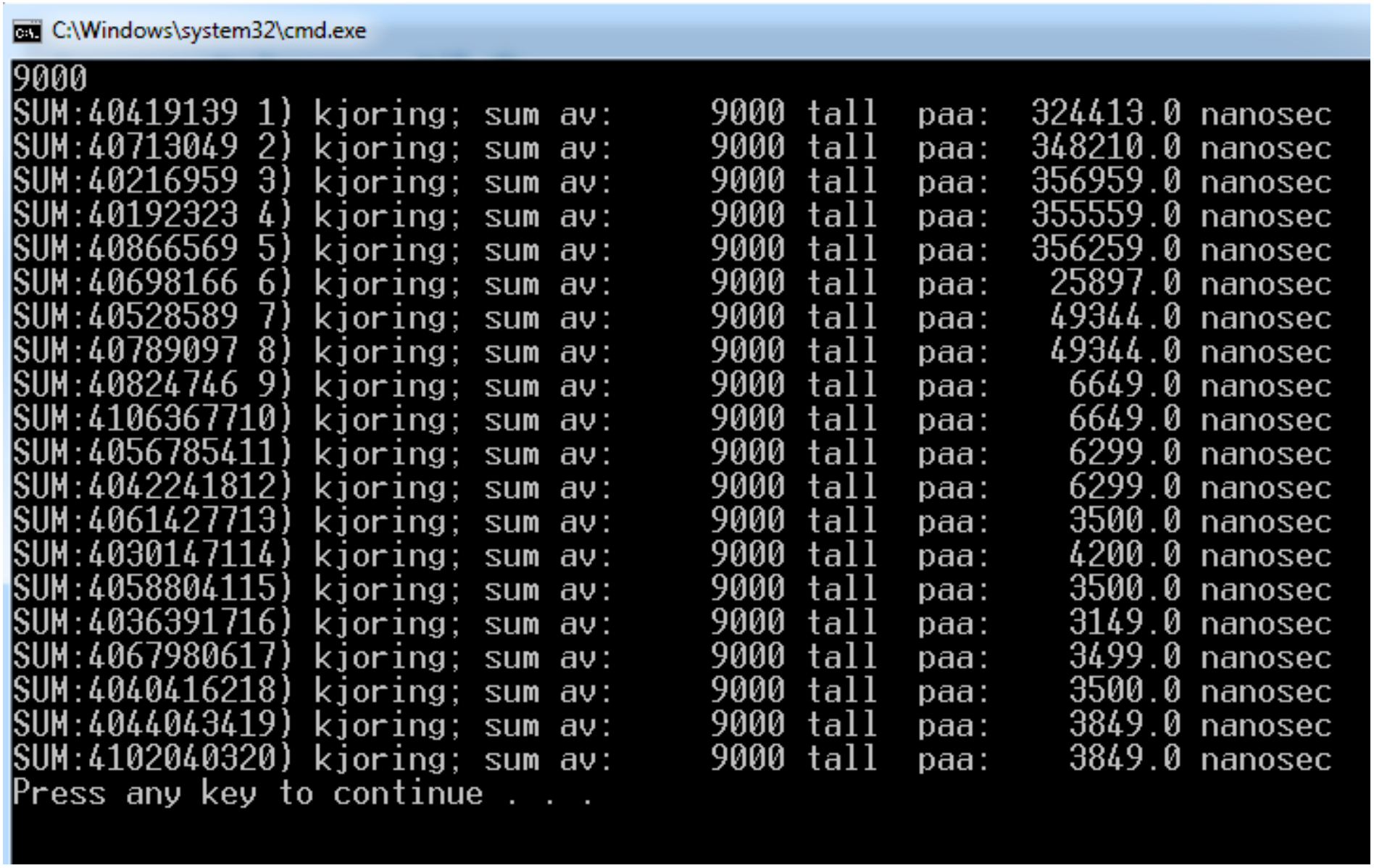
Regards,
Java Developer Support

```
import java.util.*;
import easyIO.*;
class FinnSum{
  public static void main(String[] args){
        int len = new In().inInt();
        FinnSum fs =new FinnSum();

        for(int k = 0; k < 20; k++){
              int[] arr = new int[len];
              Random r = new Random();
              for(int i = 0; i < arr.length; i++){
                    arr[i] = r.nextInt(len-1);
              }
              long start  = System.nanoTime();
              long sum = fs.summer(arr);
              long timeTaken = System.nanoTime() - start ;

             System.out.println( "SUM:"+ sum +
                Format.align(k+1,2)+ ") kjoring; sum av:"+
                 Format.align(len,9)+" tall  paa:" +
                 Format.align(timeTaken,10) + " nanosec");
   }} // end main

  long summer(int [] arr){
              long sum = 0;
              for(int i = 0; i < arr.length; i++)
                          sum += arr[i];
             return sum;
  } } // end summer
```

Eneste måten jeg har funnet å få dette riktig er å bruke det tallet (resultatet fra metoden vi skulle måle tiden på) inne i en utskriftssetning:

>java FinnSumOK

og man ser de ulike skrittene i JIT-kompilering og en feil.


C:\Windows\system32\cmd.exe
9000
SUM:40419139 1) kjoring; sum av: 9000 tall paa: 324413.0 nanosec
SUM:40713049 2) kjoring; sum av: 9000 tall paa: 348210.0 nanosec
SUM:40216959 3) kjoring; sum av: 9000 tall paa: 356959.0 nanosec
SUM:40192323 4) kjoring; sum av: 9000 tall paa: 355559.0 nanosec
SUM:40866569 5) kjoring; sum av: 9000 tall paa: 356259.0 nanosec
SUM:40698166 6) kjoring; sum av: 9000 tall paa: 25897.0 nanosec
SUM:40528589 7) kjoring; sum av: 9000 tall paa: 49344.0 nanosec
SUM:40789097 8) kjoring; sum av: 9000 tall paa: 49344.0 nanosec
SUM:40824746 9) kjoring; sum av: 9000 tall paa: 6649.0 nanosec
SUM:4106367710) kjoring; sum av: 9000 tall paa: 6649.0 nanosec
SUM:4056785411) kjoring; sum av: 9000 tall paa: 6299.0 nanosec
SUM:4042241812) kjoring; sum av: 9000 tall paa: 6299.0 nanosec
SUM:4061427713) kjoring; sum av: 9000 tall paa: 3500.0 nanosec
SUM:4030147114) kjoring; sum av: 9000 tall paa: 4200.0 nanosec
SUM:4058804115) kjoring; sum av: 9000 tall paa: 3500.0 nanosec
SUM:4036391716) kjoring; sum av: 9000 tall paa: 3149.0 nanosec
SUM:4067980617) kjoring; sum av: 9000 tall paa: 3499.0 nanosec
SUM:4040416218) kjoring; sum av: 9000 tall paa: 3500.0 nanosec
SUM:4044043419) kjoring; sum av: 9000 tall paa: 3849.0 nanosec
SUM:4102040320) kjoring; sum av: 9000 tall paa: 3849.0 nanosec
Press any key to continue . . .

Konklusjon om optimaliseringsfeil

- Sannsynligvis gjøres følgende feil:
  - sum trenges ikke/brukes ikke, og fjernes
  - For å optimaliser mer flyttes kallet på summer()-metoden over kallet på de to kallene på System.nanoTime()
- Riktig går det når 'sum' (og tiden) brukes i utskrift, og da byttes det ikke om på kallene.
  - Fjern kommentaren i utskriften, dvs:

```
long start  = System.nanoTime();
long sum = fs.summer(arr);
long timeTaken = System.nanoTime() - start ;

System.out.println( "SUM:"+ sum +
   Format.align(k+1,2)+ ") kjoring; sum av:"+
```

- Dette er 'eneste' grepet jeg har funnet hittil for å unngå denne feilen i tillegg til :
  - >java –Xint Mittprogram….

Hvor lang tid tar egentlig ulike mekanismer i Java ? (fra Petter A. Busteruds master-oppgave: Invesigating Different Concurrency Mechanisms in Java)

- Dette er målinger på en relativt gammel CPU :
  - Intel Pentium M 760@ 2GHz – single core
  - 2 GB DDR2 RAM @ 533 MHz
- Windows 7 32bit
- Linux Mint 13, 32bit
- Det er de relative hastighetene som teller
- Petter målte bl.a. :
  - kalle en metode
  - lage et nytt objekt (new C()) + kalle en metode
  - lage en Tråd (new Thread + kalle run())
  - Lage en ExecutorService + legge inn tråder
  - Tider på int[] og trådsikre AtomicIntegerArray

Generelt er Linux 15-20% raskere enn Windows (ikke optimalisert)

Hvor lang tid tar egentlig ulike mekanismer i Java ? (fra Petter A. Busteruds master-oppgave: Investigating Different Concurrency Mechanisms in Java)

A) Metodekall på Windows og Linux (ikke optimalisert)

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.75	4.75	4.75	4.75	4.75	4.75
Additional Calls	-	2.24	2.05	1.96	1.96	1.96
Total Run Time	4.75	6.71	10.7	18.5	34.1	65.4

Table 6.2: Overhead using Methods on Windows (in μs)

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.54	4.54	4.55	4.54	4.54	4.54
Additional Calls	-	1.61	1.49	1.45	1.43	1.42
Total Run Time	4.54	6.08	8.95	14.6	26.0	48.5

Table 6.3: Overhead using Methods on Linux (in μs)

Generelt er kan vi gjøre mer enn 500-700 metodekall per millisek.

Å lage et nytt objekt av en klasse (new) og kalle en metode i objektet (μs)

ClassCall	Number of Classes					
	1	2	4	8	16	32
Create the Object	759	768	774	774	773	777
First Method Call	11	11	11	11	11	11
Additional Calls	-	4	3	2	2	2
Total Run Time	770	788	802	817	848	914

Table 6.4: Overhead using Classes on Windows (in μs)

Konklusjon:

- Verken det å kalle en metode eller å lage et objekt tar særlig lang tid
- Mye blir optimalisert videre
- Forskjellen mellom første og andre kall er at det sannsynligvis er blitt JIT-kompilert til maskinkode, men ikke optimalisert

Lage en tråd (new Thread()) og kalle run() - µs

- **First `Run()` Call**: Executing *`thread`*`.start()`, starting the object's `Run` method to be called in that separately executing thread.
- **Additional Threads**: Identical to "Create the First Thread", for the rest of the specified number of threads.
- **Additional `Run()` Calls**: Identical to "First `Run()` Call", but for the additional threads.
- **Total Thread Create**: Total cost of creating all Threads, First Thread + All Additional Threads.
- **Total `Run()` Call**: Total cost of the First `Run()` Call + All Additional `Run()` Calls.

ThreadCall	Number of Threads					
	1	2	4	8	16	32
Create the First Thread	759	758	764	767	766	772
First `Run()` Call	839	835	836	815	820	846
Additional Threads	-	37	27	24	22	22
Additional `Run()` Calls	-	188	178	180	170	168
Total Thread Create	-	797	850	935	1101	1456
Total `Run()` Call	-	1039	1388	2118	3425	6050
Total Run Time	1614	1841	2244	3062	4554	7533

Table 6.6: Overhead using Threads on Windows (in μs)

Lage en ExecutorService

- **Create ExecutorService**: Is the set-up time for the ExecutorService, here we create a fixed `ThreadPool` with the specified number of threads as a parameter. In addition create a `Future` to provide the ability to check when *computations* (calls) are completed.
- **First Submit**: The very first submit done to the `ThreadPool`, with the additional overhead for the system to assign the submitted work to an available Thread in the Pool.
- **Additional Submits**: Every additional submit done to the `ThreadPool`.
- **Total Submit**: Total cost of the First Submit + All Additional Submits.

ExecutorCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ExecutorService	4881	4861	4874	4870	4835	4957
First Submit	2026	2045	2030	2061	2042	2067
Additional Submits	-	231	238	212	203	214
Total Submit	-	2282	2756	3565	5142	8699
Total Run Time	6915	7169	7694	8534	10123	13734

Table 6.8: Overhead using ExecutorService on Windows (in μs)

Lage og bruke en vanlig int [] a (IKKE trådsikker) mot AtomicIntegerArray (trådsikre heltall)

`int[]`	Number: N					
	10	100	1000	10^4	10^5	10^6
GetAndSet N-times	0.119	0.288	1.96	22.6	225	2357
Increment N-times	0.339	0.540	2.22	21.2	208	2182
Decrement N-times	0.347	0.547	2.24	21.1	208	2177

Table 6.13: Overhead using Array of `int` on Windows (in μs)

`AtomicIntegerArray`	Number: N					
	10	100	1000	10^4	10^5	10^6
Create Array[N]	138	139	138	140	173	251
Get N-times	1.37	3.03	19.2	163	688	2951
Set N-times	4.37	5.92	19.2	167	1173	3860
Increment N-times	8.57	11.8	42.4	341	2323	5768
Decrement N-times	8.60	11.9	42.3	360	2365	5805

Table 6.15: Overhead using AtomicIntegerArray on Windows (in μs)

Oppsummering om kjøretider:

- Metodekall tar svært liten tid: **2-5** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage et objekt av en klasse (og kalle en metode i det) tar liten tid: ca.**785** μs
- Å lage en tråd og starte den opp tar en del tid: ca.**1500** μs, men lite ca. **180** μs for de neste trådene (med start())
- Å lage en en tråd-samling og legge tråder (og Futures) opp i den tar ca. **7000** μs for første tråd og ca. **210** μs for neste tråd.
- Å bruke trådsikre heltall (AtomicInteger og AtomicIntegerArray) mot vanlige int og int[] tar vanligvis **10-20 x** så lang tid , men for mye bruk (> 10^6) tar det bare ca. 2x tid (det siste er sannynligvis en cache-effekt)

Sequential Quicksort

Number of Elements	Sorting Time
1 000	104 μs
2 500	288 μs
5 000	607 μs
7 500	973 μs
10 000	1318 μs
25 000	3632 μs
50 000	7980 μs
75 000	12214 μs
100 000	16980 μs

Figure 6.2: Quicksort Sorting Times

En parallell løsning må alltid sammenlignes med hvor mye vi kunne ha sortert sekvensielt på denne tida!

Hvordan ikke gå i den rekursive fella!

- Vi skal nå gå gjennom hvordan vi behandler rekursjon og parallellisering av den
- Først en 'teoretisk' PRAM-lignende parallell løsning
  - som går ca. 1000x langsommere enn en sekvensiell versjon.
- Så skal vi se på to forbedringer som gjør at den uhyre treige løsningen vår tilslutt har en speedup på ca. 4
- Hele problemet bunner i de tallene vi fant i forrige seksjon:
  - Å starte en ny tråd: **180** μs
  - Å gjøre et metodekall: **2-5** μs (og vil bli sterkt optimalisert)

Om rekursiv oppdeling av et problem

- Svært mange problemer kan gis en (sekvensiell og parallell) rekursiv løsning:
  - De fleste søkeproblemer
    - Del søkebunken rekursivt opp i disjunkte deler og søk (i parallell) i hver bunke.
  - Mange sorteings-algoritmer som QuickSort, Flettesortering, og venstreRadix-sortering er definert rekursivt
- Skal nå bruke en ny formulering av quicksort som eksempel og gi den 3 ulike løsninger:
  A. Ren oversettelse av rekursjonen til tråder
  B. Med to tråder for hvert nivå inntil vi bruker InnstikkSortering
  C. Med en ny tråd for hver nivå og avslutting av tråder når lengden < LIMIT (si 50 000) – deretter vanlig rekursjon

Generelt om rekursiv oppdeling av a[] i to deler

```
void Rek (int [] a, int left, int right) {
        <del opp omradet a[left..right] >
        int deling = partition (a, left,right);

        if (deling - left  > LIMIT ) Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) Rek (a,deling, right);
        else  <enkel løsning>
}
```

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2= null;

        if (deling - left > LIMIT )  t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) t2 = new Thread (a,deling, right);
        else  <enkel løsning>
        try{ if (t1!=null)t1.join();
              if (t2!=null)t2.join();} catch(Exception e){};
}
```

A

```
void Rek (int [] a, int left, int right) {
            <del opp omradet a[left..right] >
            int deling = partition (a, left,right);

            if (deling - left  > LIMIT ) Rek (a,left, deling -1);
            else <enkel løsning>;
            if (right - deling > LIMIT) Rek (a, deling , right);
            else  <enkel løsning>
}
```

B

```
void Rek(int [] a, int left, int right) {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            if (deling- left > LIMIT )
                 (t1 = new Thread (a,left, deling -1)).start();
            else <enkel løsning>;
            if (right - deling> LIMIT)
                 (t2 = new Thread (a, deling , right)).start();
            else  <enkel løsning>
            try{ if (t1!=null)t1.join();
                 if (t2!=null)t2.join();} catch(Exception e){};
}
```

Oppdeling med **to** tråder per nivå i treet:

- Når ventes det i den reukursive løsningen
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i den parallelle løsningen A?
  - Har rekkefølgen på venting på t1 og t2 betydning?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar A og B
- Hvilken er raskest ?

Oppdeling med **en tråd** per nivå i treet:

- Hvorfor virker dette ?
- To alternativ løsning med 1 tråd
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i C-løsningen?
  - Har rekkefølgen på venting på t1 betydning?
- Når ventes det i D-løsningen?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar C
- Hvor lang tid tar D

Hvilken er klart raskest:
C eller D?

C

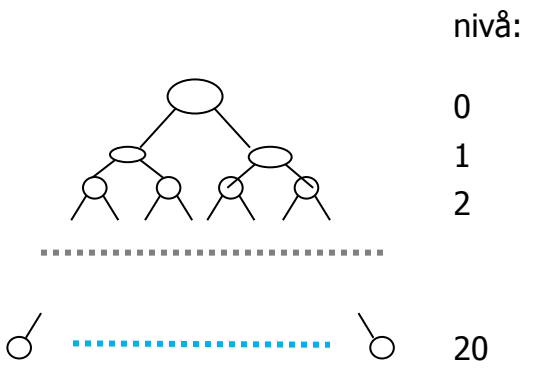
```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                t1 = new Thread (a,left,deling-1);
        else  <enkel løsning>
         try{t1.join();} catch(Exception e){};}
```

D

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                Rek (a,deling, right);
        else  <enkel løsning>
        try{t1.join();} catch(Exception e){};}
```

Hvor mange kall gjør vi i en rekursiv løsning?

- Anta Quicksort av n $=2^k$ tall
  (k= 10 $\Rightarrow$ n = 1000, k= 20 $\Rightarrow$ n= 1 mill)
- Kalltreet vil på første nivå ha 2 lengder av $2^{19,}$ på neste: 4 = 2^2 hver med 2^{18} og helt ned til nivå 20, hvor vi vil ha 2^{20} kall hver med 1 = 2^0 element.
- I hele kalltreet gjør vi altså 2 millioner -1 kall for å sortere 1 mill tall !
- Bruker vi innstikksortering for n < 32 = 2^5 så får vi 'bare' $2^{20-5} = 2^{15}$ = 32 768 kall.
- Metodekall tar : **2-5** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage en tråd og starte den opp tar: ca.**1500** μs, men ca. **180** μs for de neste trådene (med start())



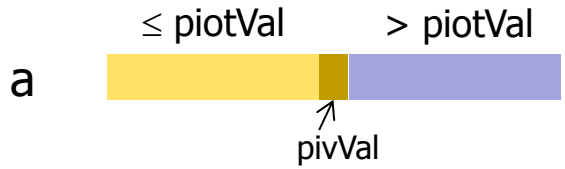
Vi kan IKKE bare erstatte rekursive kall med nye tråder i en rekursiv løsning !

A) Sekvensiell kvikksort – ny og enklere kode

```
// sekvensiell Kvikksort
void quicksortSek(int[] a, int left, int right) {
        int piv = partition (a, left, right); // del i to
        int piv2 = piv-1, pivotVal = a[piv];
        while (piv2 > left  && a[piv2] == pivotVal) {
             piv2--; // skip like elementer i midten
         }

        if ( piv2-left >0) quicksortSek(a, left,  piv2);
           else insertSort(a,left, piv2);
        if ( right-piv >1) quicksortSek(a, piv + 1, right);
          else  insertSort(a, piv+1, right);
} // end quicksort
```

```
// del opp a[] i to: smaa og storre
int partition (int [] a, int left, int right) {
    int pivVal = a[(left + right) / 2];
    int index = left;
   // plasser pivot-element helt til høyre
   swap(a, (left + right) / 2, right);

    for (int i = left; i < right; i++) {
        if (a[i] <= pivVal) {
            swap(a, i, index);
            index++;
        }
     }
     swap(a, index, right); // sett pivot tilbake
     return index;
 } // end partition

 void swap(int [] a, int left, int right) {
      int temp = a[left];
      a[left] = a[right];
      a[right] = temp;
} // end swap
```

Etter: partition(a,left,right)



B) En parallell kode (modellert etter A)

```
void Rek(int [] a, int left, int right) {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            else insertSort(a,left, deling -1);
            if (right - deling> LIMIT)
                (t2 = new Thread (a, deling , right)).start();
            else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
                if (t2!=null)t2.join();} catch(Exception e){};
}
```

B) Ren kopi av rekursiv løsning: Katastrofe

```
M:>java QuickSort 100 10 100000 1 uke9.txt
Test av  TEST AV QuickSort
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:2


      n      sekv.tid(ms)   para.tid(ms)    Speedup
 100000        34.813      41310.276        0.0008
  10000         0.772        735.838        0.0010
   1000         0.078         66.007        0.0012
    100         0.009          3.491        0.0026
```

Konklusjon:

- For store n speeddown på ca. 1000
- Kunne ikke kjøre for n > 100 000 pga. trådene tok for stor plass

Hva med en passe LIMIT = 32 ?

```
>java QuickSort 100 10 1000000 1 uke9.

Test av  TEST AV QuickSort
med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32


     n       sekv.tid(ms)    para.tid(ms)     Speedup
  1000000       89.181       41789.076        0.0021
   100000        8.118         823.432        0.0099
    10000        3.021          55.845        0.0541
     1000        0.060           3.463        0.0173
      100        0.006           0.302        0.0185
```

Konklusjon:

- Mye bedre, men fortsatt 100 x langsommere enn sekvensiell(N = 100 000)
- Greide nå n= 1 mill (fordi færre tråder)
- Fortsatt håpløst dårlig pga. for mange tråder)
- Trenger ny ide : Bruk sekvensiell løsning når n < 50 000 ? BIG_LIMIT

Skisse av ny løsning

```
void Rek(int [] a, int left, int right) {
    if ( right – left  < BIG_LIMIT) quicksort(a, left,right);
    else {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            //if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            //else insertSort(a,left, deling -1);
            //if (right - deling> LIMIT)
                (t2 = new Thread (a, deling , right)).start();
            //else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
                if (t2!=null)t2.join();} catch(Exception e){};
    }
}
```

Generere nye tråder bare i toppen av rekusjonstreet

- Kan da også stryke kode om LIMIT (vil ikke bli utført)
- Bruken av insertSort gjøres i (sekv) quicksort(..)

Kjøreeksempel med BIG_LIMIT og LIMIT

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner,
                   LIMIT:32, BIG_LIMIT:50000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 100000000    12042.708        3128.675        3.8491
  10000000     1090.252         277.264        3.9322
   1000000       92.958          32.640        2.8480
    100000        7.682           5.198        1.4777
     10000        0.616           0.737        0.8356
      1000        0.051           0.117        0.4354
       100        0.013           0.015        0.8636
```

BIG_LIMIT = 100 000

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 100000000    12110.344        2967.764        4.0806
  10000000     1084.587         277.454        3.9091
   1000000       93.428          32.078        2.9125
    100000        7.894           7.828        1.0085
     10000        0.815           0.617        1.3224
      1000        0.072           0.101        0.7153
       100        0.013           0.015        0.8410
```

Konklusjon om å parallellisere rekursjon

- Antall tråder må begrenses !
- I toppen av treet brukes tråder (til vi ikke har flere og kanskje litt mer)
- I resten av treet bruker vi sekvensiell løsning i hver tråd!
- Viktig også å kutte av nedre del av treet (her med insertSort) som å redusere treets størrelse drastisk (i antall noder)
- Vi har for n = 100 00 gått fra:

n	sekv.tid(ms)	para.tid(ms)	Speedup	
100000	34.813	41310.276	0.0008	Ren trådbasert
100000	8.118	823.432	0.0099	Med insertSort
100000	7.682	5.198	1.4777	+ Avkutting i toppen

- Speedup > 1 og ca. 10 000x fortere enn ren oversettelse.

Hva så vi på i uke 9

- Et sitat om tidsforbruk ved faktorisering
- En presisering av Oblig2.
- Om en feil i Java ved tidtaking (tid == 0 ??)
- Hvor lang tid de ulike mekanismene vi har i Java tar.
- Hvordan parallellisere rekursive algoritmer
- Gå IKKE i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger

En enklere kode (og ganske effektiv) for parallell kvikksort.

```
void quicksort(int[] a, int left, int right) {
    if (left < right) {
        if (right-left < LIMIT) insertSort(a,left,right) ;
        else{
            int pivotIndex = partition(a, left, right);
            int pivotIndex2 = pivotIndex-1,
                pivotVal = a[pivotIndex];
                while (pivotIndex2 > left
                        && a[pivotIndex2] == pivotVal){
                            pivotIndex2--;}
                Thread t1 = null ;
                if (right-left <= BIG_LIMIT) {
                    quicksort(a, left,  pivotIndex2);
                }
                else {  t1 = new Thread(new
                      QuickPara(a, left,  pivotIndex2));
                      t1.start();
                }
                // use same thread on right branch
                quicksort(a, pivotIndex + 1, right);
                if (t1 != null) {
                    try { t1.join();
                    } catch (InterruptedException e) { return;
                }
      } // end else
    } // end if
} // end quicksort
```

```
int partition (int[] a, int left, int right) {
    int pivotValue = a[(left + right) / 2];
    swap(a, (left + right) / 2, right);
    int index = left;

    for (int i = left; i < right; i++) {
            if (a[i] <= pivotValue) {
                swap(a, i, index);
                index++;
            }
     }
     swap(a, index, right);
     return index;
 } // end partition

 void swap(int[] a, int left, int right) {
      int temp = a[left];
      a[left] = a[right];
      a[right] = temp;
 } // end swap
```