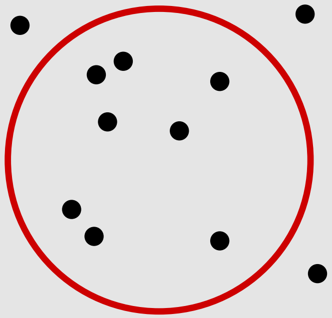
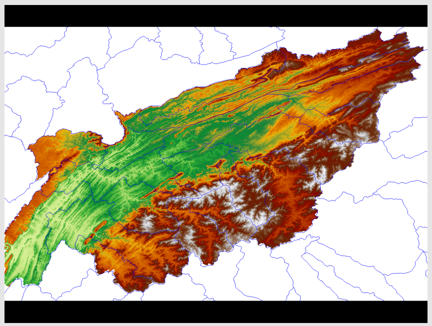
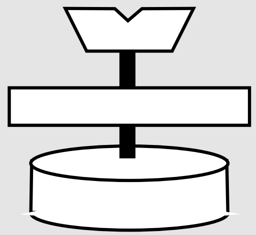
I/O- and Cache-Efficient Algorithms for Spatial Data

Mark de Berg

TU Eindhoven





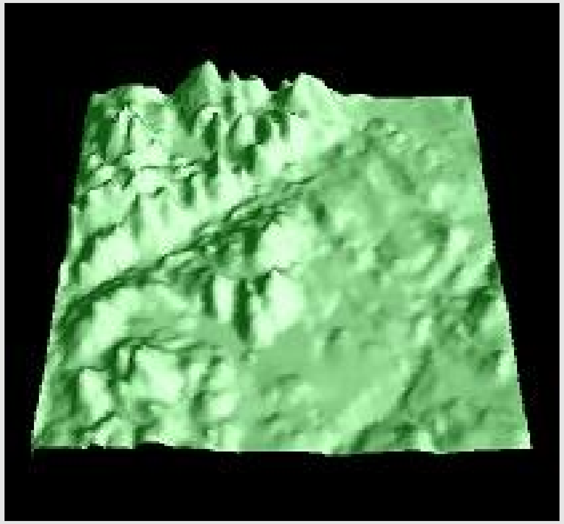


Massive data sets are becoming more and more common

- AT&T: data base of phone calls: 20 TB
- Wal-Mart: data base of buying patterns: 70 TB
- geographic data: NASA satellites collect more than 1 TB / day

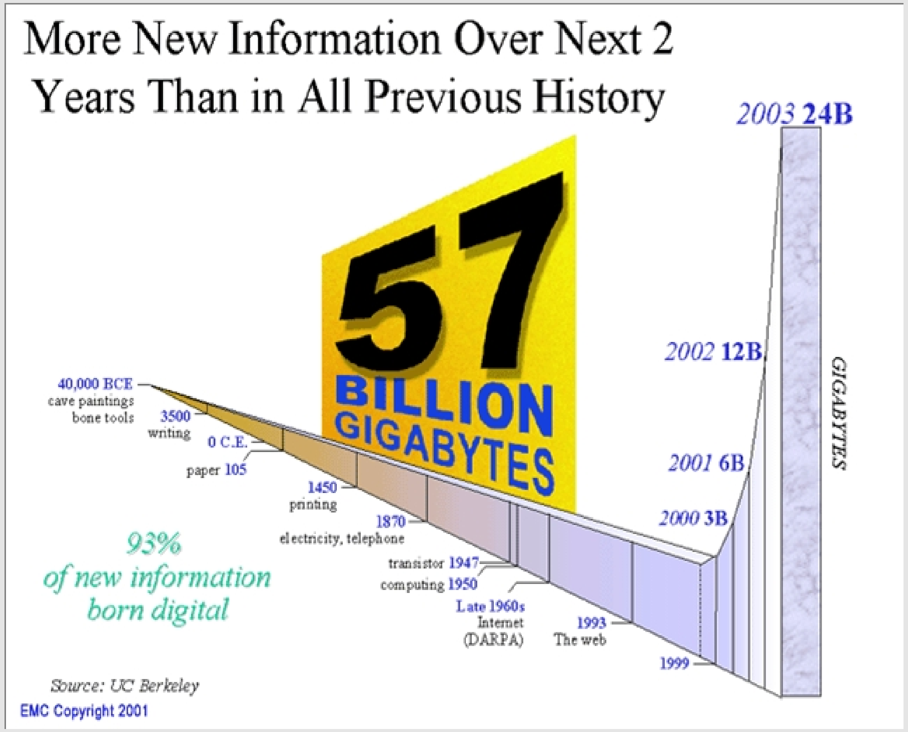
Massive data sets are becoming more and more common

- AT&T: data base of phone calls: 20 TB
- Wal-Mart: data base of buying patterns: 70 TB
- geographic data: NASA satellites collect more than 1 TB / day



Apalachian mountains (800 x 800 km^2)

- 100 m resolution: 500 MB
- 30 m resolution: 5.5 GB
  (available for 80% of earth)
- 1 m resolution: 5 TB



UC Berkeley study (2000)

Two sorting algorithms: *InsertionSort* and *MergeSort*

```
InsertionSort (A)
for j := 2 to N
  do key := A[j]; i := j − 1
     while i > 0 and A[i] > key
       do A[i + 1] := A[i]; i := i − 1
A[i + 1] := key
```

```
MergeSort (A, p, r)
if p < r
then q := (p + r)/2
     MergeSort (A, p, q)
     MergeSort (A, q + 1, r)
     Merge (A, p, q, r)
```

```
Merge (A, p, q, r)
N1 := q − p + 1; N2 := r − q
Create arrays L[1..N1 + 1] and R[1..N1 + 1]
for i := 1 to N1 do L[i] := A[p + i − 1]
for i := 1 to N2 do R[i] := A[q + j]
L[N1 + 1] := ∞; R[N2 + 1] := ∞
for k := p to r
  do if L[i] ≤ R[j]
     then A[k] := L[i]; i := i + 1
     else A[k] := R[j]; j := j + 1
```

Which one is faster ?

running time depends on input size

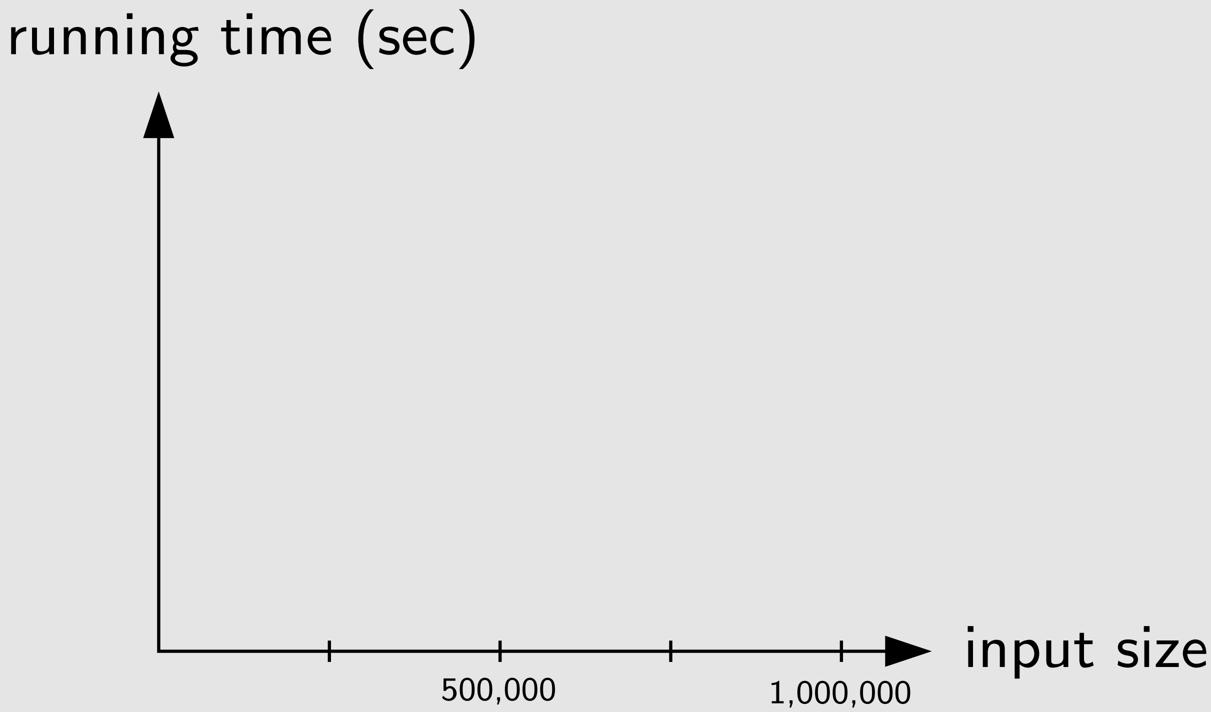
$\Downarrow$

analyze running time as a function of the input size

running time depends on input size

$\Downarrow$

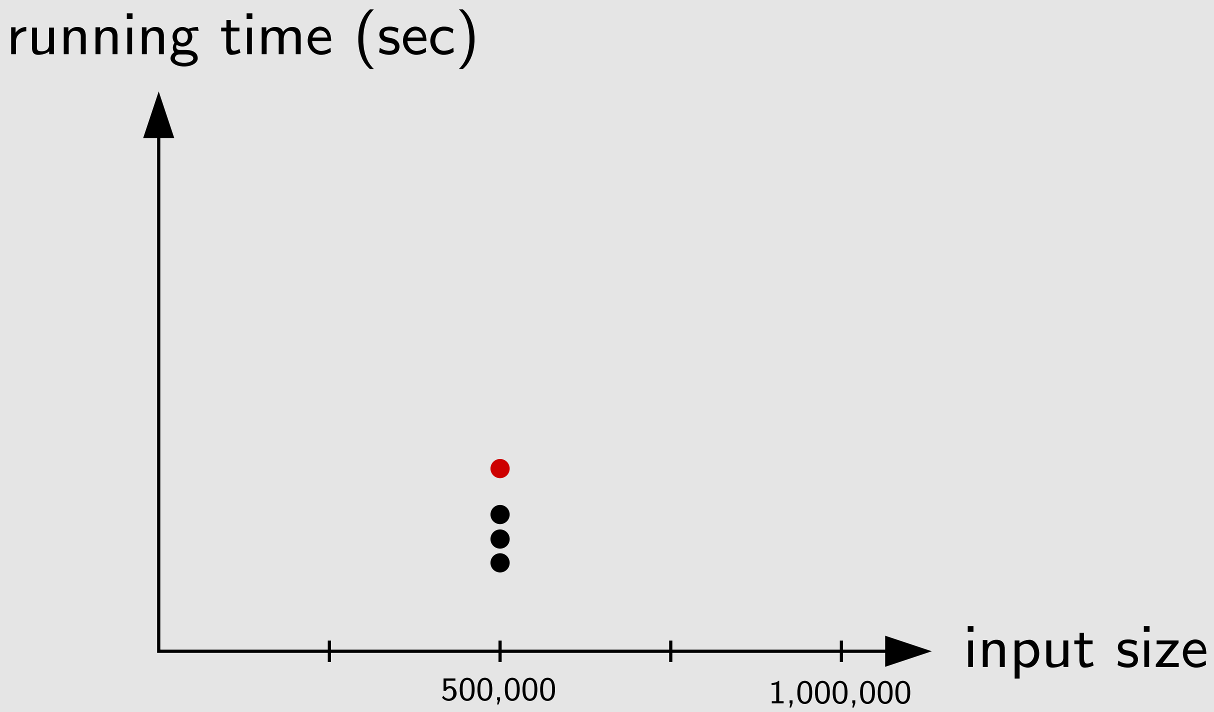
analyze running time as a function of the input size



running time depends on input size

⇓

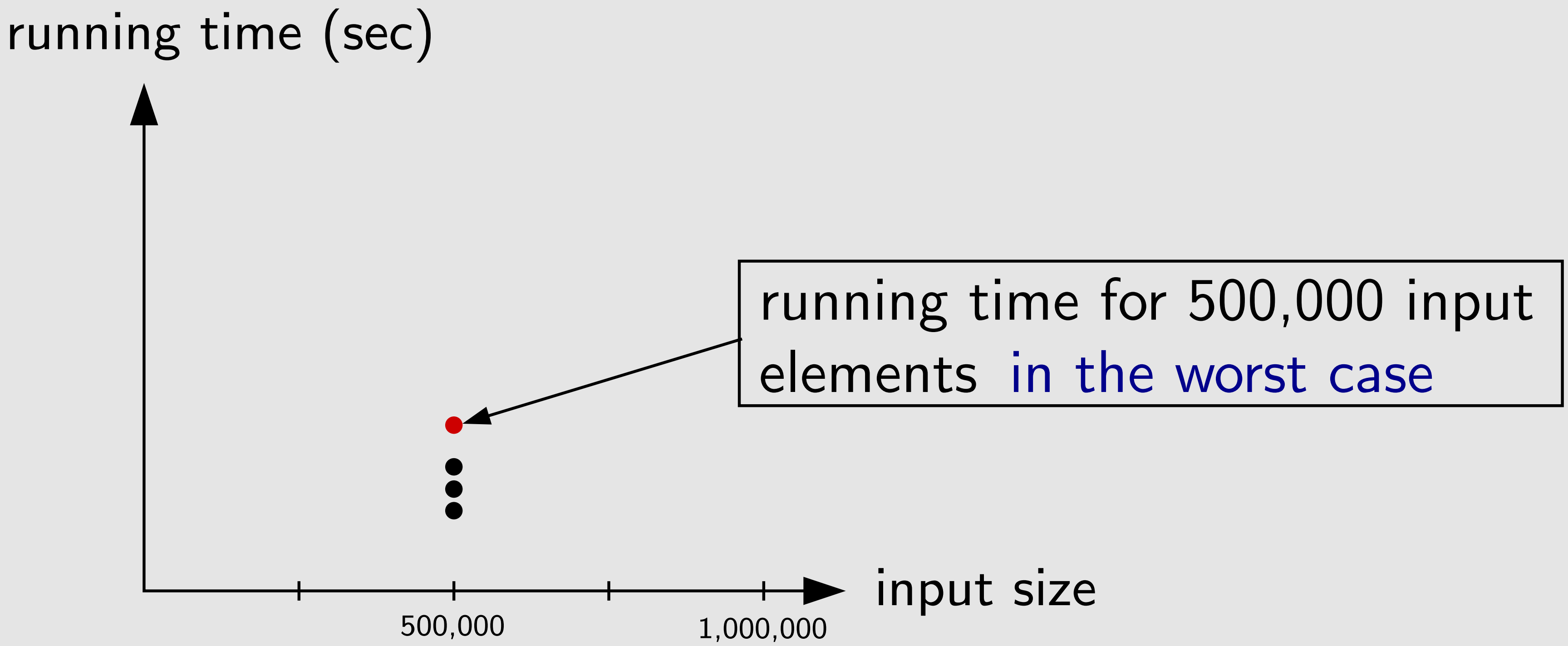
analyze running time as a function of the input size



running time depends on input size

⇓

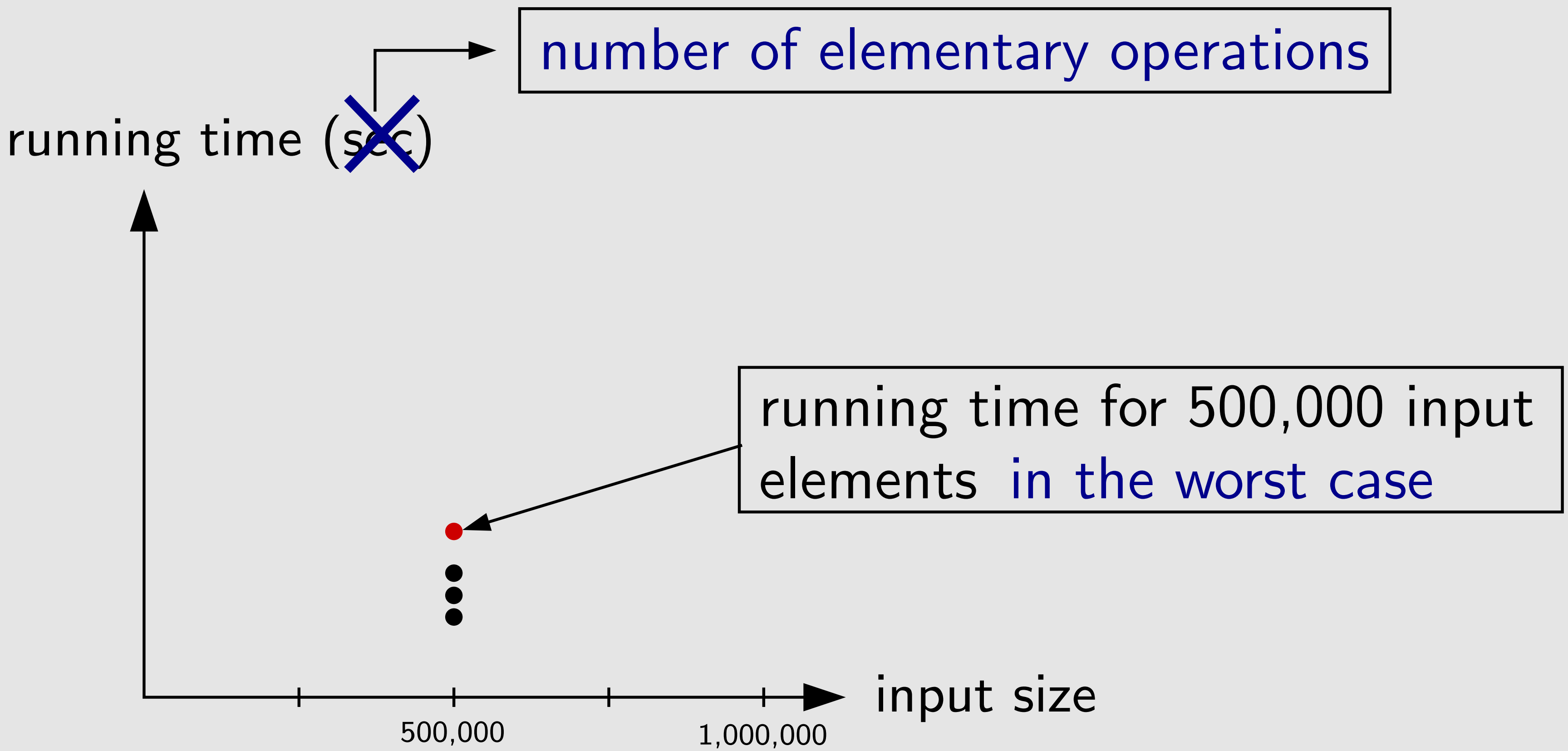
analyze running time as a function of the input size



running time depends on input size

⇓

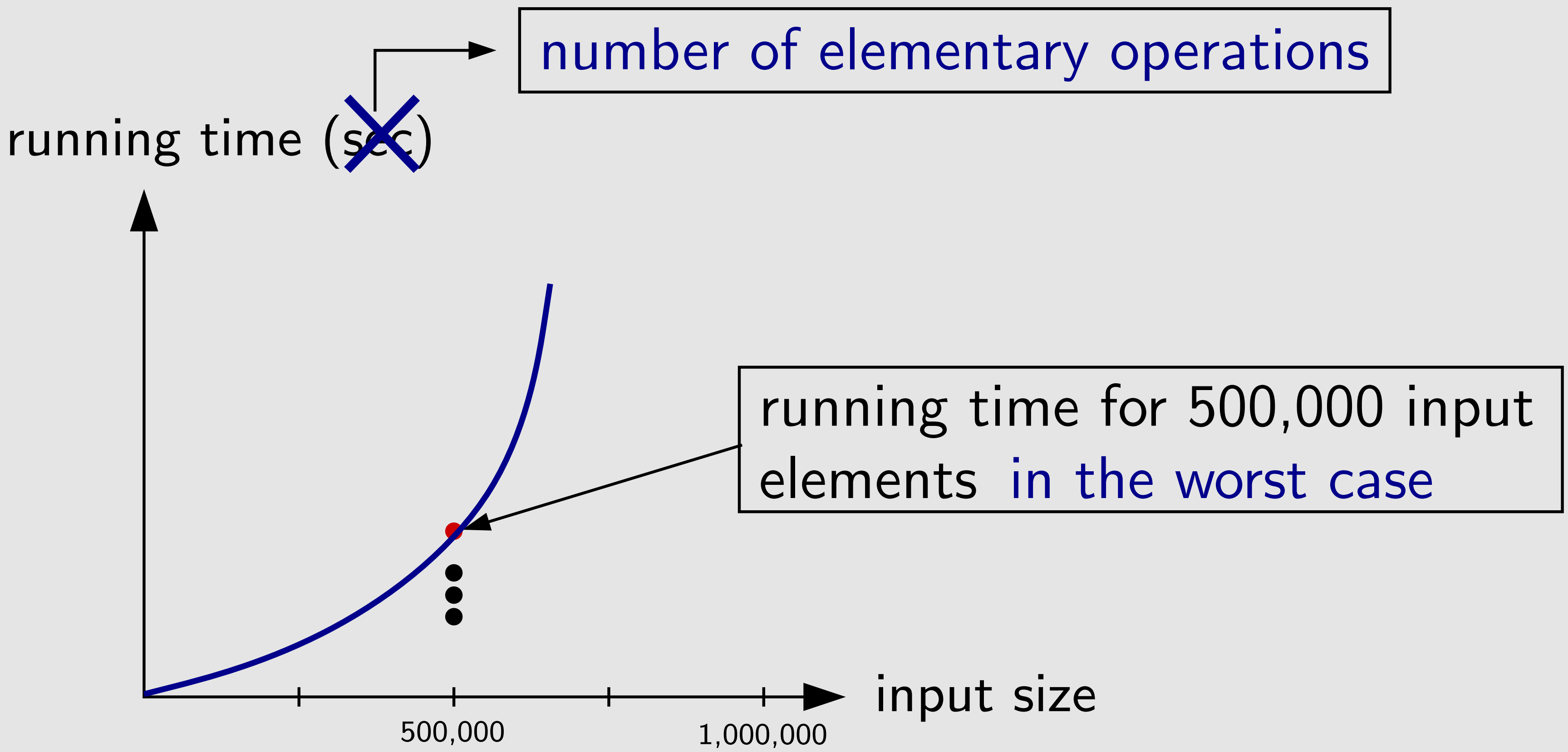
analyze running time as a function of the input size

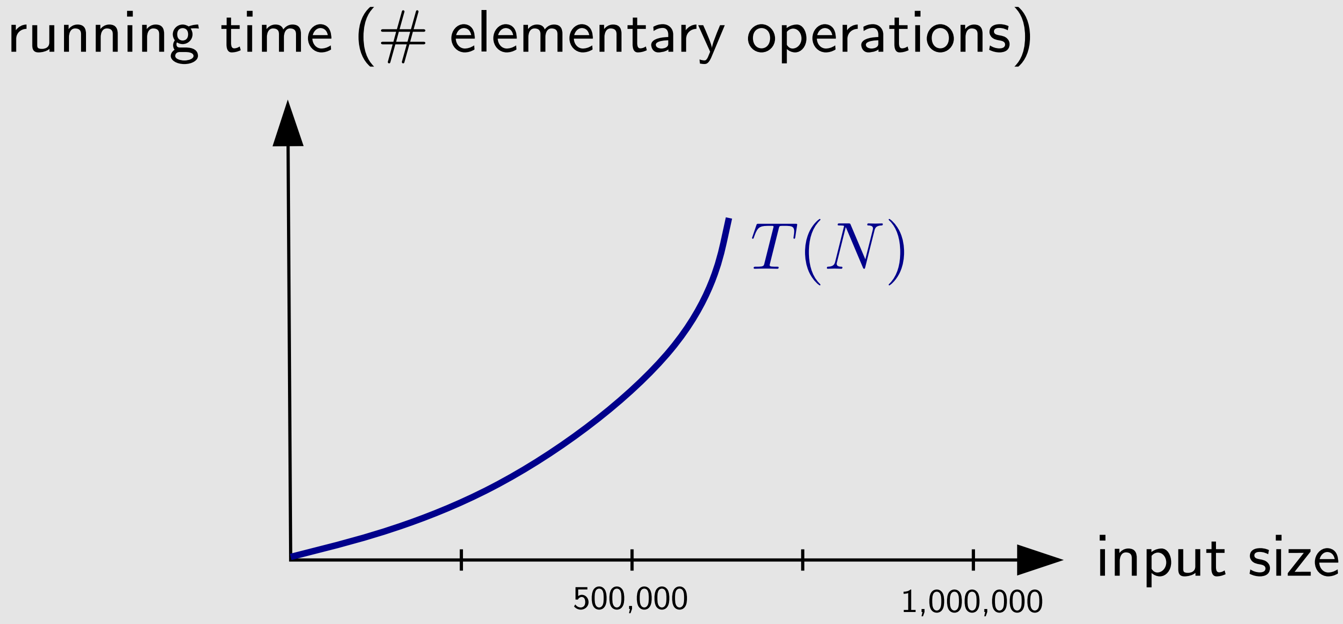


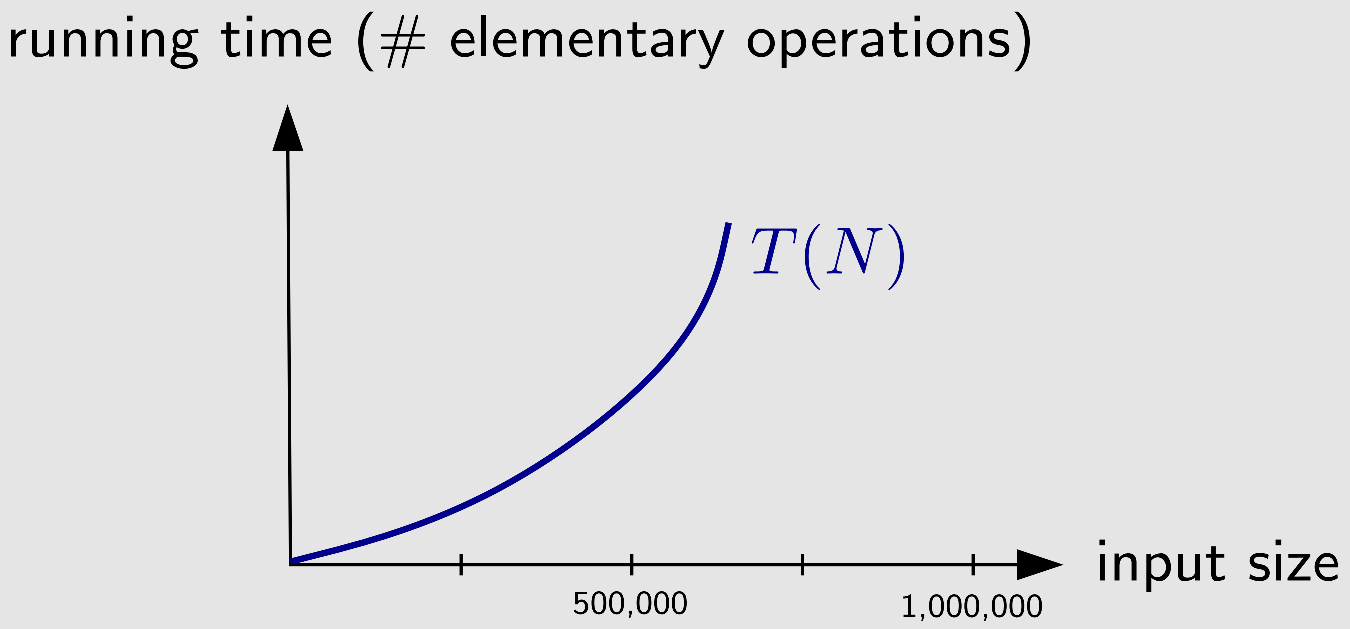
running time depends on input size

$\Downarrow$

analyze running time as a function of the input size




running time (# elementary operations)
$T(N)$
input size
500,000
1,000,000



$T(N) =$ # elementary operations the algorithm performs in the worst case as function of N, the number of input elements

we analyze asymptotic behavior of $T(N)$: is it $O(N)$, $O(N^2)$, etc.

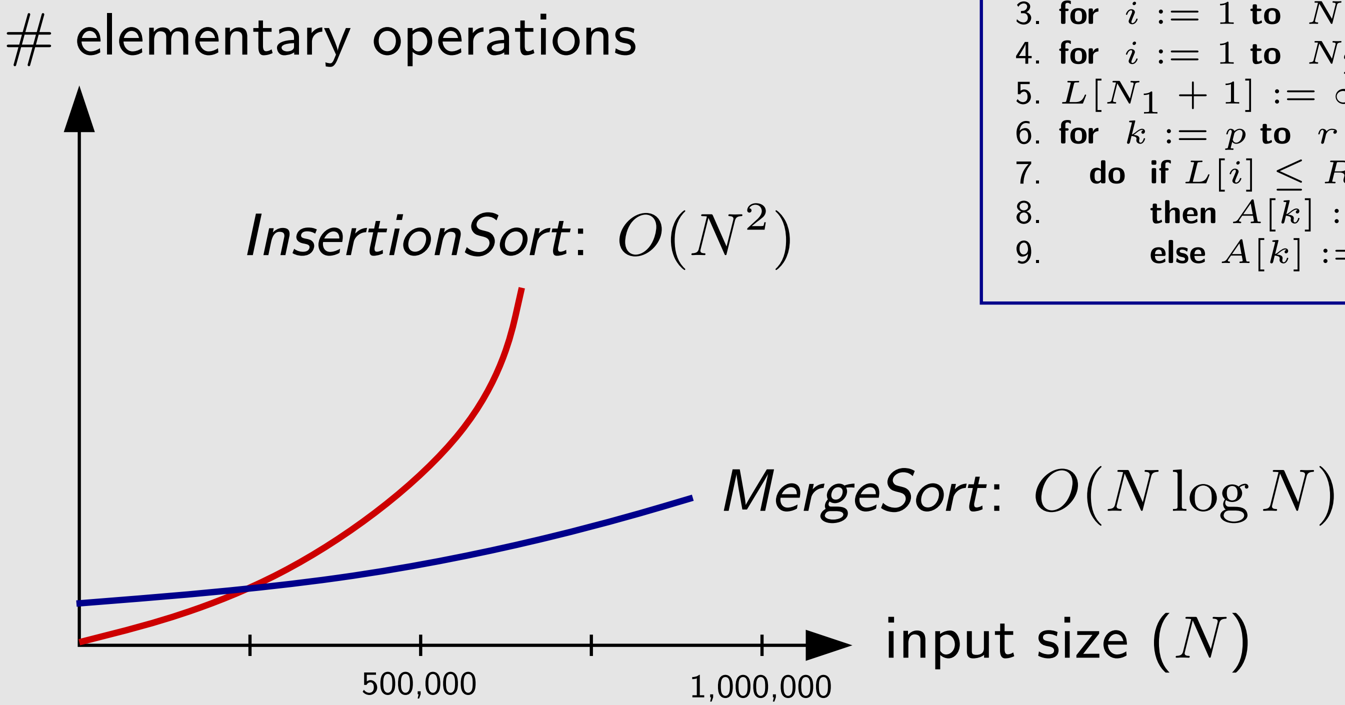
$\Longrightarrow$ relevant for large data sets!

Analysis of algorithms: *InsertionSort* vs. *MergeSort*

InsertionSort (A)
for $j := 2$ **to** N
  do *key*$:= A[j]$; $i := j - 1$
    while $i > 0$ **and** $A[i] >$ *key*
      do $A[i+1] := A[i]$; $i := i - 1$
    $A[i+1] :=$ *key*

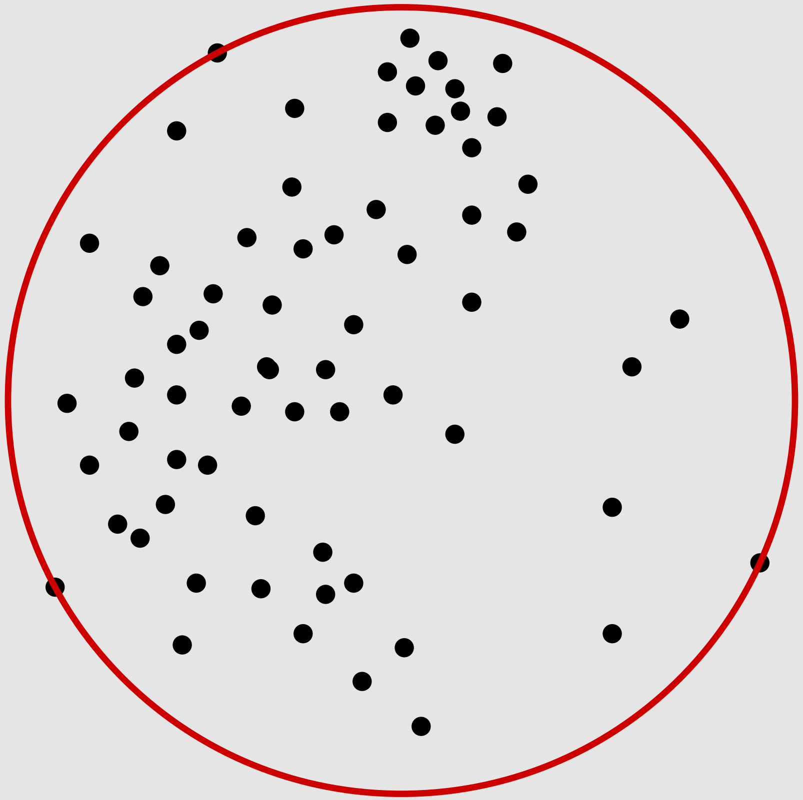
MergeSort (A, p, r)
if $p < r$
then $q := (p+r)/2$
    MergeSort (A, p, q)
    MergeSort $(A, q+1, r)$
    Merge (A, p, q, r)

Merge (A, p, q, r)
$N_1 := q - p + 1$; $N_2 := r - q$
Create arrays $L[1..N_1+1]$ and $R[1..N_1+1]$
for $i := 1$ **to** N_1 **do** $L[i] := A[p+i-1]$
for $i := 1$ **to** N_2 **do** $R[i] := A[q+j]$
$L[N_1+1] := \infty$; $R[N_2+1] := \infty$
for $k := p$ **to** r
  do if $L[i] \leq R[j]$
    then $A[k] := L[i]$; $i := i + 1$
    else $A[k] := R[j]$; $j := j + 1$



Compute smallest enclosing disk of set P of N points in the plane.

Compute smallest enclosing disk of set P of N points in the plane.



SmallestDisk(P)

1. *RandomPermute*(P)
2. $D :=$ smallest disk for $P[1], P[2], P[3]$
3. **for** $i := 4$ **to** N
4. **do** **if** $P[i] \in D$
5. **then skip**
6. **else** $D :=$ smallest disk for $\{P[1], \ldots, P[i]\}$ where $P[i]$ is on the boundary

SmallestDisk(P)
1. *RandomPermute*(P)
2. D := smallest disk for $P[1], P[2], P[3]$
3. **for** $i := 4$ **to** N
4. **do** **if** $P[i] \in D$
5. **then skip**
6. **else** D := smallest disk for $\{P[1], \ldots, P[i]\}$
   where $P[i]$ is on the boundary

"recursive" call

SmallestDisk(P)
1. *RandomPermute*(P)
2. D := smallest disk for $P[1], P[2], P[3]$
3. **for** $i := 4$ **to** N
4. **do** **if** $P[i] \in D$
5. **then skip**
6. **else** D := smallest disk for $\{P[1], \ldots, P[i]\}$ where $P[i]$ is on the boundary

"recursive" call

RandomPermute(P)
1. **for** $i := 1$ **to** $N - 1$
2. **do** r := random integer in range $i \ldots N$
3. swap $P[i]$ and $P[r]$

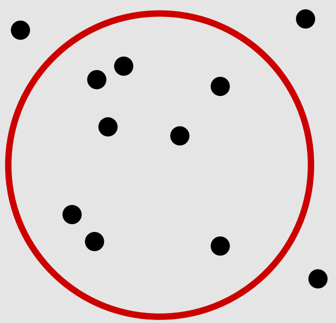
Smallest enclosing disk: analysis (1)

RandomPermute(P)
1. **for** $i := 1$ **to** $N-1$
2. **do** $r :=$ random integer in range $i \ldots N$
3. swap $P[i]$ and $P[r]$

running time is $O(N)$

Smallest enclosing disk: analysis

```
SmallestDisk(P)
RandomPermute(P)
D := smallest disk for P[1], P[2], P[3]
for i := 4 to N
  do if P[i] ∈ D
       then skip
       else D := smallest disk for {P[1], . . . , P[i]}
                 where P[i] is on the boundary
```

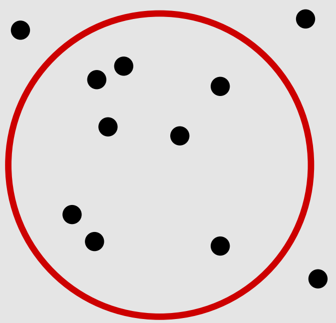


- $P[1]$ t/m $P[i]$
- $P[i+1]$ t/m $P[N]$

TU/e

Smallest enclosing disk: analysis

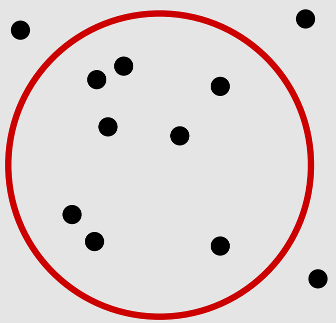
```
SmallestDisk(P)
RandomPermute(P)
D := smallest disk for P[1], P[2], P[3]
for i := 4 to N
  do if P[i] ∈ D
       then skip
       else D := smallest disk for {P[1], . . . , P[i]}
                 where P[i] is on the boundary
```



Pr [$P[i] \notin D$] $\leq 3/i$

- $P[1]$ t/m $P[i]$
- $P[i+1]$ t/m $P[N]$

SmallestDisk(P)
1. *RandomPermute*(P)
2. $D :=$ smallest disk for $P[1], P[2], P[3]$
3. **for** $i := 4$ **to** N
4. **do** **if** $P[i] \in D$
5. **then skip**
6. **else** $D :=$ smallest disk for $\{P[1], \ldots, P[i]\}$ where $P[i]$ is on the boundary

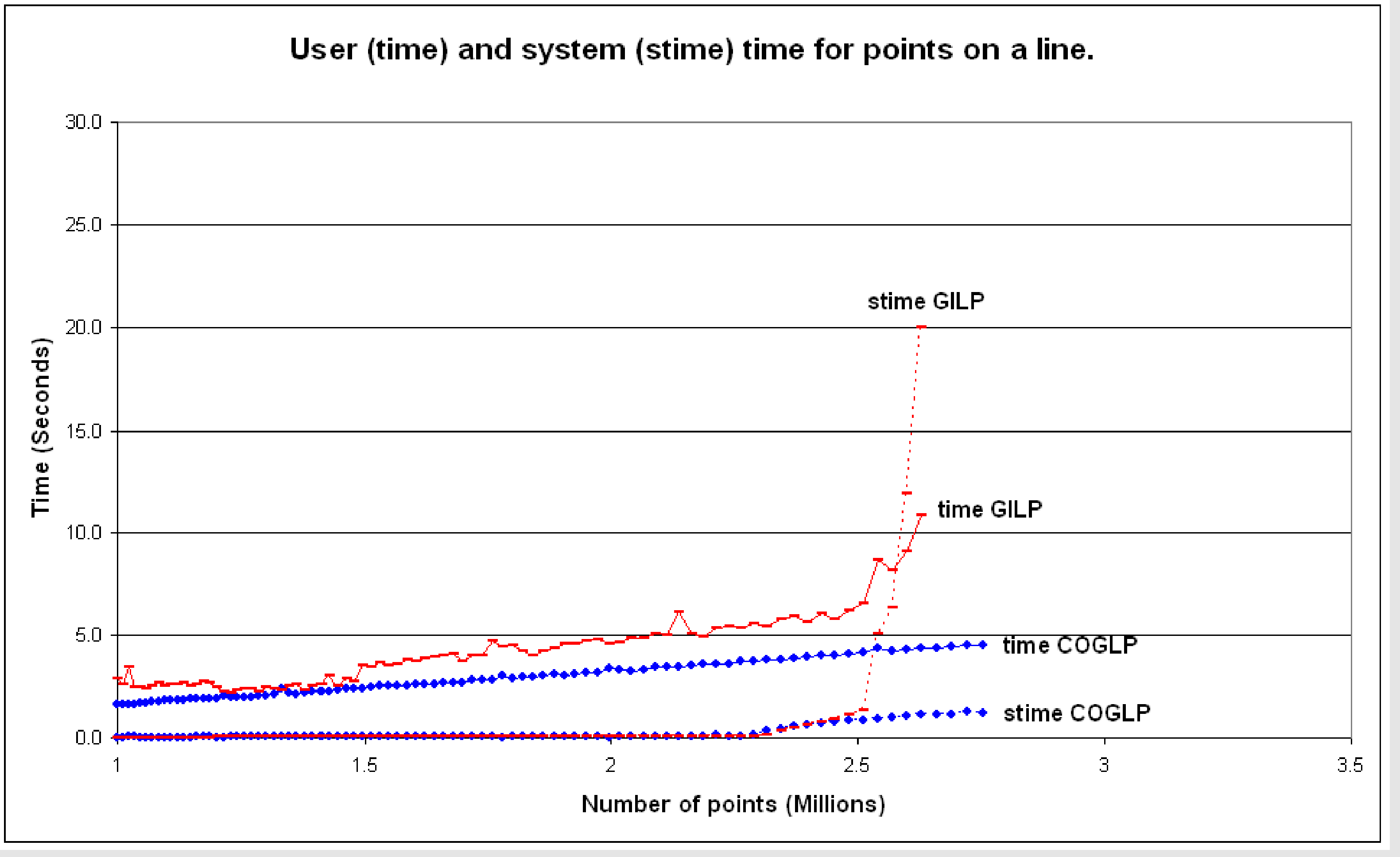


- $P[1]$ t/m $P[i]$
- $P[i+1]$ t/m $P[N]$

Pr [$P[i] \notin D$] $\leq 3/i$

$\Longrightarrow$ expected running time is $O(N)$

Smallest enclosing disk: experiments



Pentium 4, 2.60GHz

$\approx$ 89 MB main memory available to the program

$T(n) =$ # elementary operations the algorithm performs in the worst case as function of N, the number of input elements

additions, multiplications, comparisons, reading a value from memory, etc.

Hmmm ... is this justified?

$T(n) =$ # elementary operations the algorithm performs in the worst case as function of N, the number of input elements

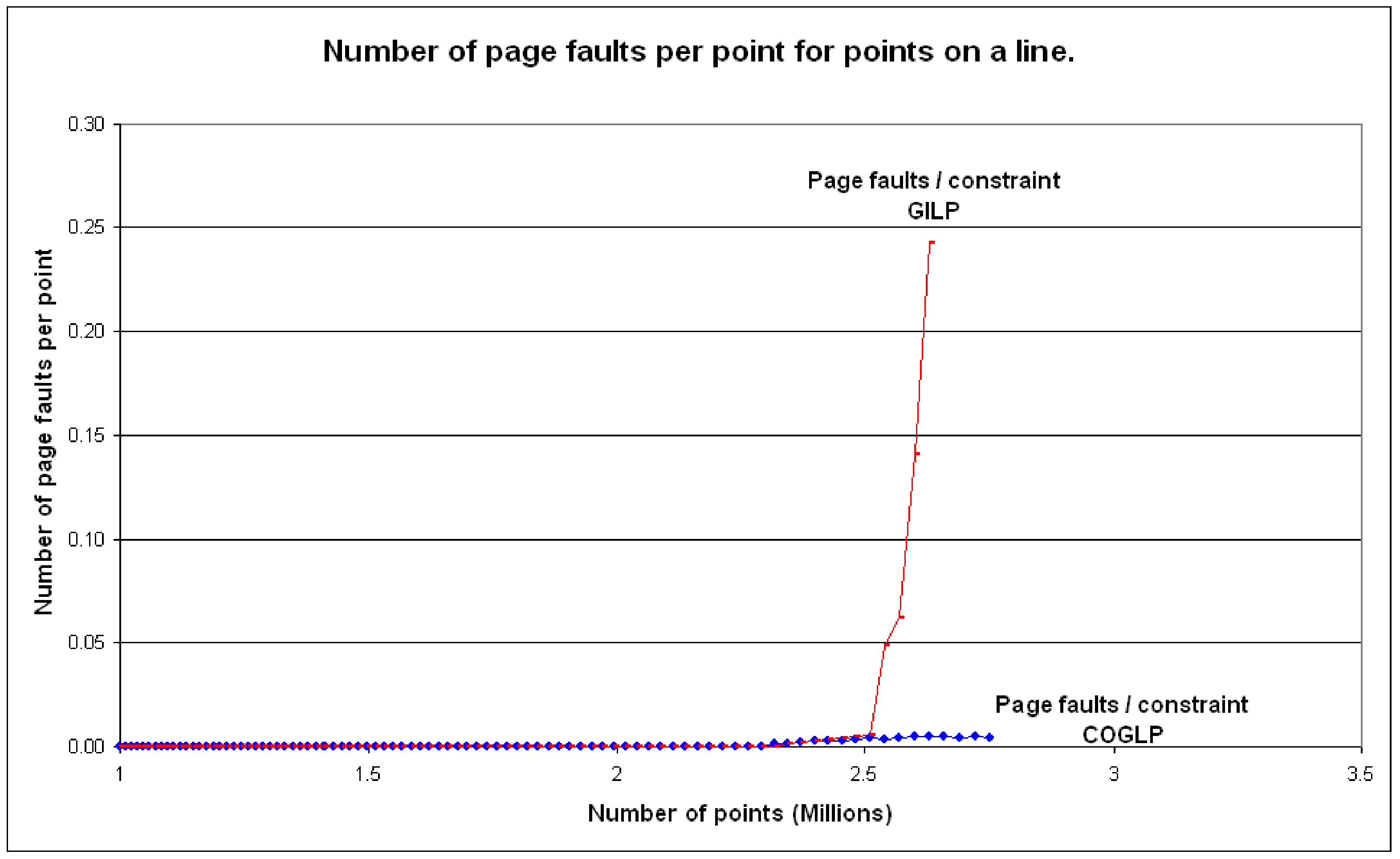
additions, multiplications, comparisons, reading a value from memory, etc.

Hmmm ... is this justified?

NO!

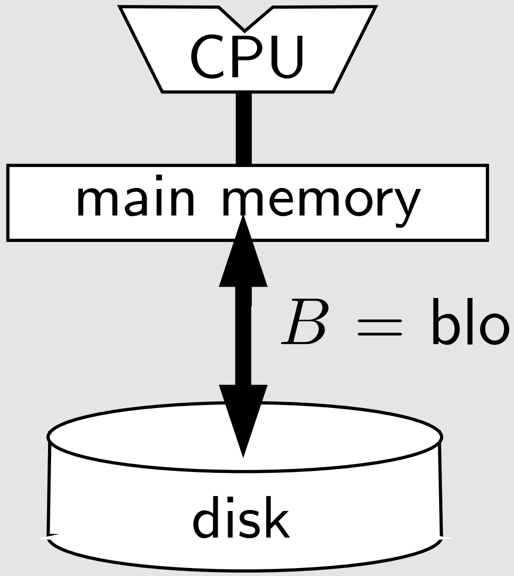
operations on data in main memory: tens of nanoseconds
disk operations: several milliseconds

Smallest enclosing disk: experiments (2)



Pentium 4, 2.60GHz

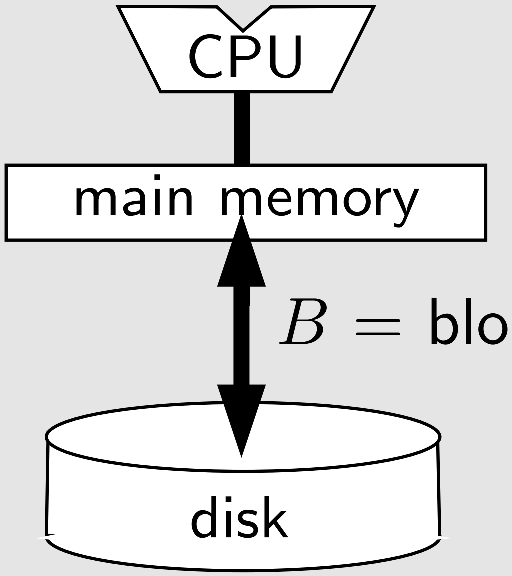
$\approx$ 89 MB main memory available to the program



M = size of main memory

B = block size for data transport

"typical value" 8KB



M = size of main memory

B = block size for data transport
"typical value" 8KB

- let algorithm handle data placement and transport
  - which data are placed together in a block
  - which blocks are kept in main memory
- analyze number of disk operations

RandomPermute(P)
1. **for** $i := 1$ **to** $N-1$
2. **do** $r :=$ random integer in range $i \dots N$
3. swap $P[i]$ and $P[r]$

analysis of (expected) number of disk operations

- $N \leq M$: 0

  0 disk operations

- $N > M$:

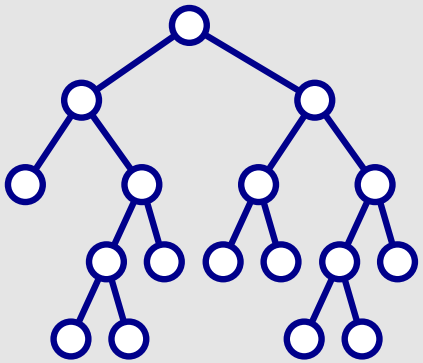
  $(N-1) \cdot (1 - \frac{M}{N})$ disk operations

  (e.g. $(N-1)/2$) disk operations when $N = 2M$)

TU/e

I/O-efficient search structures: a well known example

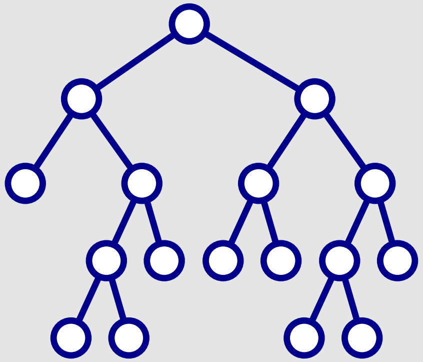
binary search tree: search structure for internal memory



- nodes contain one key, have degree 2
- depth is $O(\log N)$

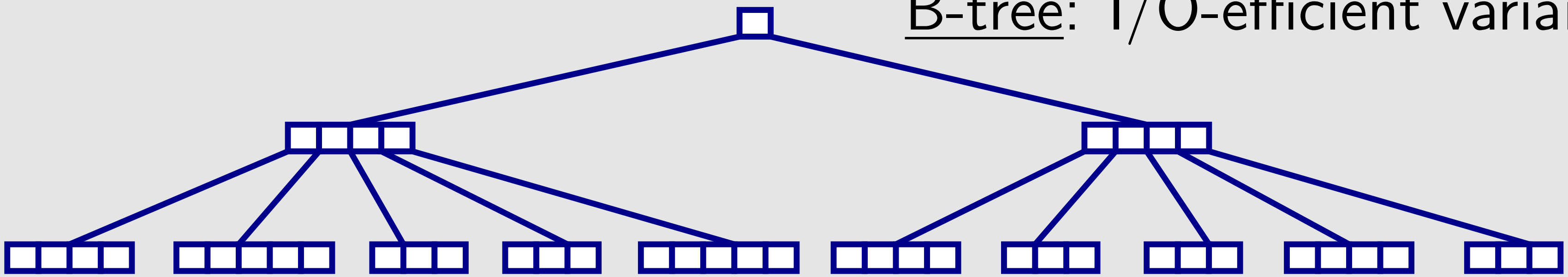
I/O-efficient search structures: a well known example

binary search tree: search structure for internal memory



- nodes contain one key, have degree 2
- depth is $O(\log N)$

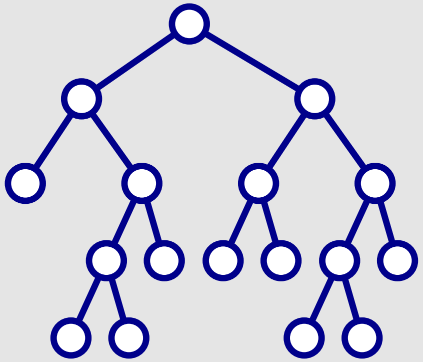
B-tree: I/O-efficient variant



- nodes contain many keys, have high degree
- put each node into one block on disk
- depth is $O(\log N / \log B)$

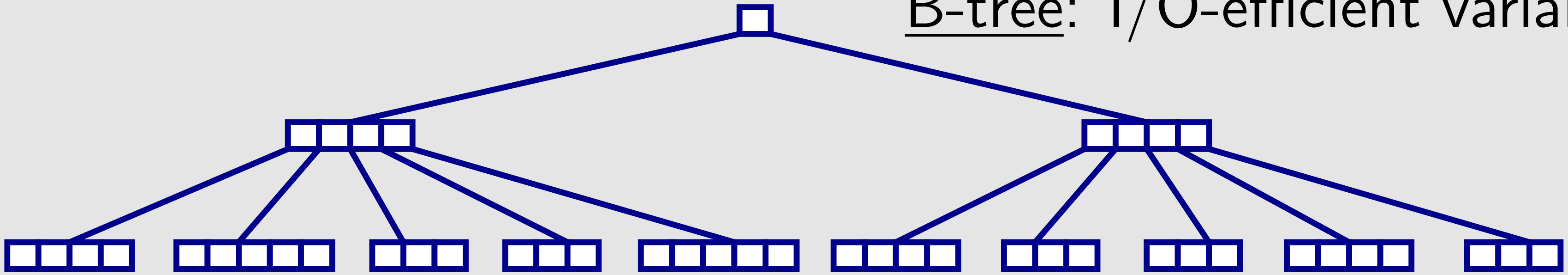
I/O-efficient search structures: a well known example

binary search tree: search structure for internal memory



- nodes contain one key, have degree 2
- depth is $O(\log N)$

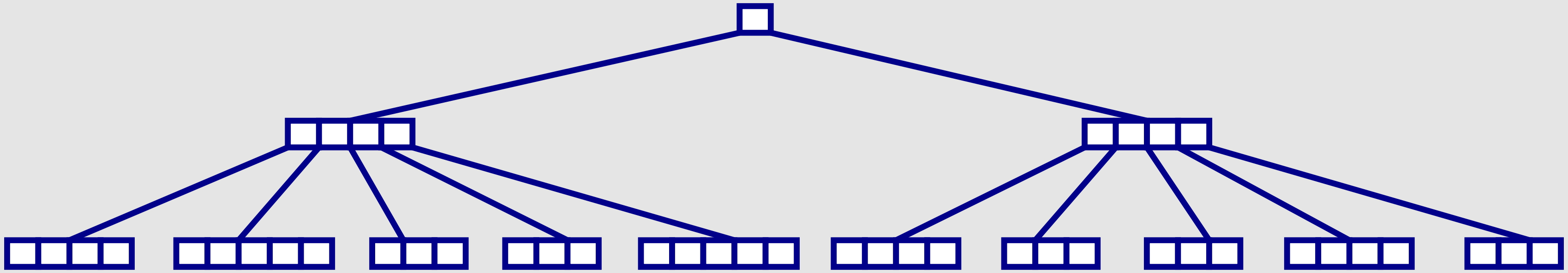
B-tree: I/O-efficient variant



- nodes contain many keys, have high degree
- put each node into one block on disk
- depth is $O(\log N / \log B)$

in practice, degree is 250 – 2000
and depth is at most 4

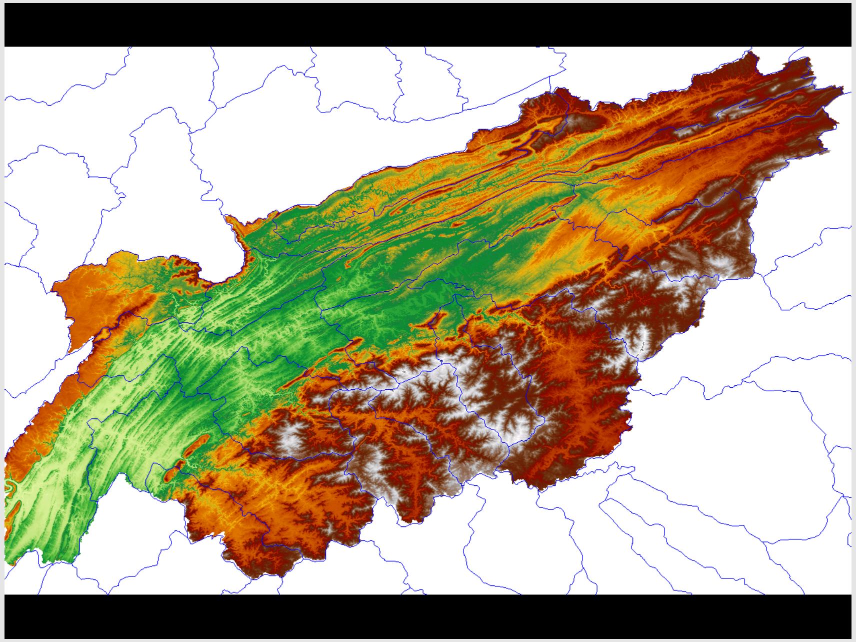
R-tree: search structure (index) for spatial data



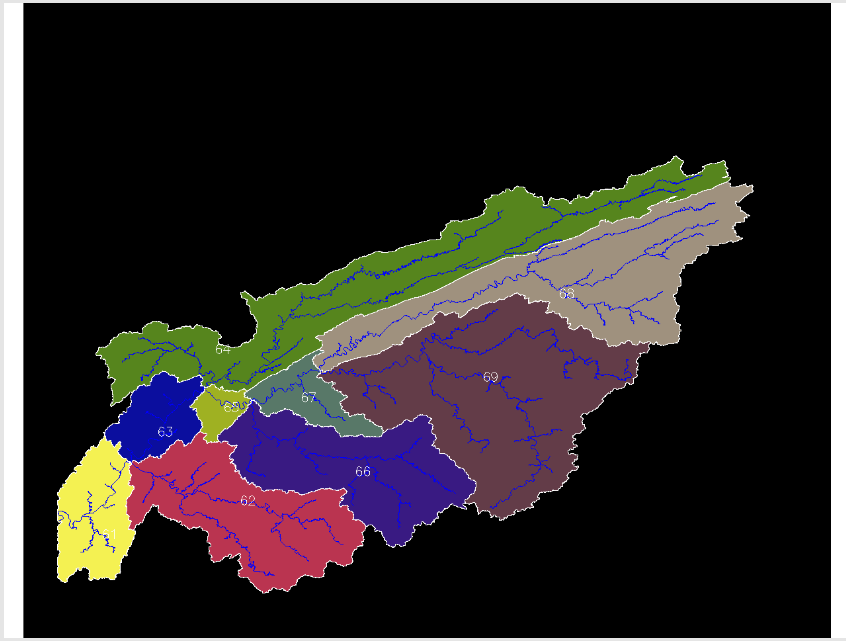
I/O-efficient variant of bounding-volume hierarchy:

- nodes contain many bounding boxes, have high degree
- put each node into one block on disk
- depth is $O(\log N / \log B)$

Another example: TerraFlow

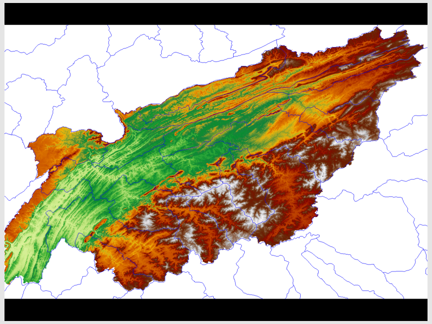


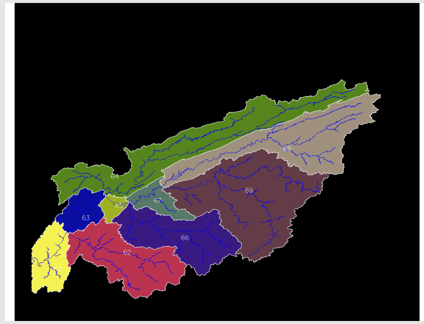
digital elevation model (DEM)



- flow routing, flow accumulation
- watersheds, Pfaffstetter labeling

TerraFlow: P. Agarwal, L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. Vitter, R. Wickremesinghe. Pfaffstetter: + H. Haverkort





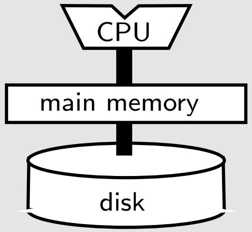
comparison with existing commercial software (ArcInfo) and open source software (GRASS):

speed-up factor between 2 and 1000

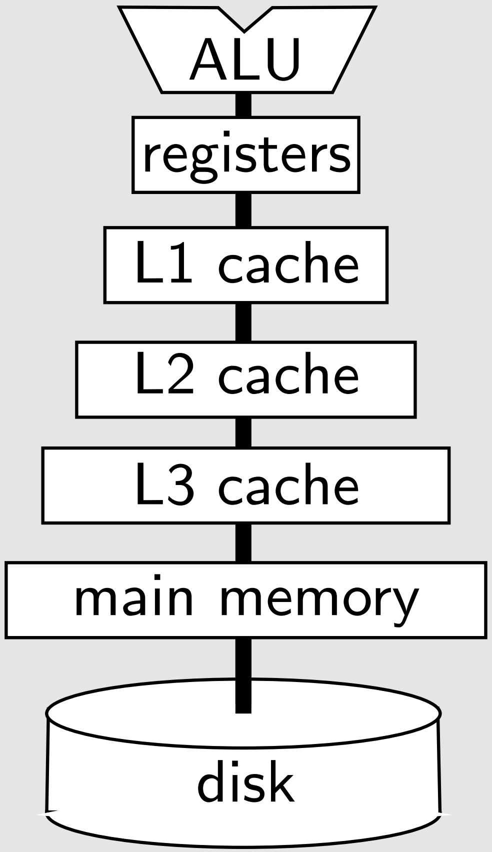
implemented using TPIE, a library for I/O-efficient algorithms

Designing I/O-efficient algorithms?

- M and B depend on platform
- even on fixed machine values of M and B may vary
  - main memory may have to be shared with other processes
  - disk-cache "changes" block size
- two-level I/O-model too simplistic

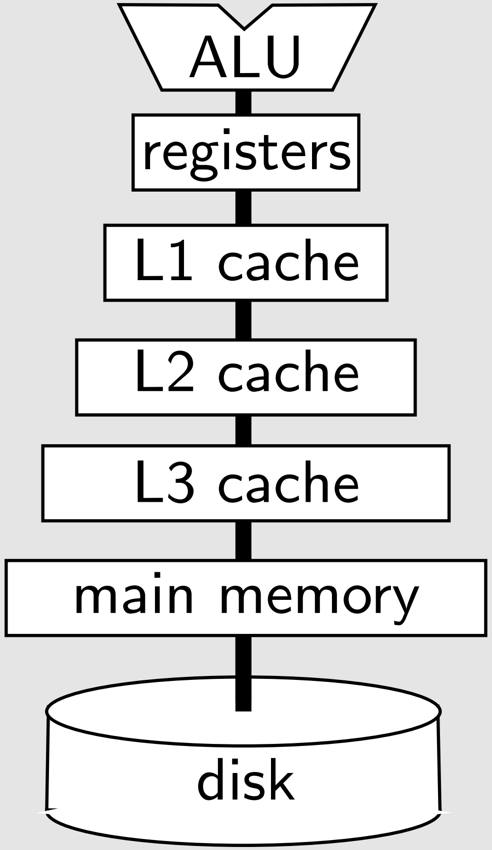


Intel Itanium2 memory hierarchy



	capacity	latency
L1 cache	16 KB	1 cycle
L2 cache	256 KB	5+ cycles
L3 cache	6MB	12+ cycles
main memory	2 GB	$>$ 150 cycles
disk		can be 10^6 cycles

Intel Itanium2 memory hierarchy



	capacity	latency
registers		
L1 cache	16 KB	1 cycle
L2 cache	256 KB	5+ cycles
L3 cache	6MB	12+ cycles
main memory	2 GB	$>$ 150 cycles
disk		can be 10^6 cycles

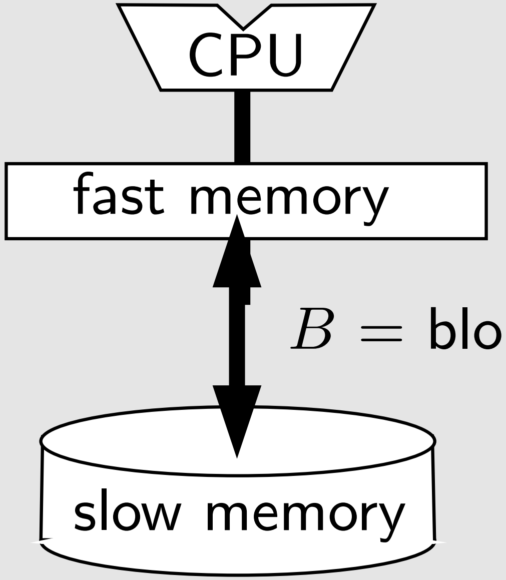
Conclusion: caching behavior can also make a large difference.

Ideal: algorithm that is effcient w.r.t. disk and all cache levels

- caches are not under control of algorithm
- algorithms taking all cache-levels into account quite complicated

So what can we do ??

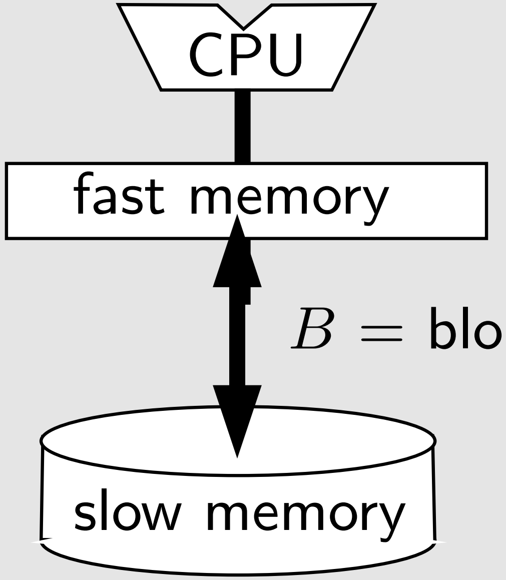
Algorithm designed for simple two-level memory model



M = size of fast memory

B = block size for data transport

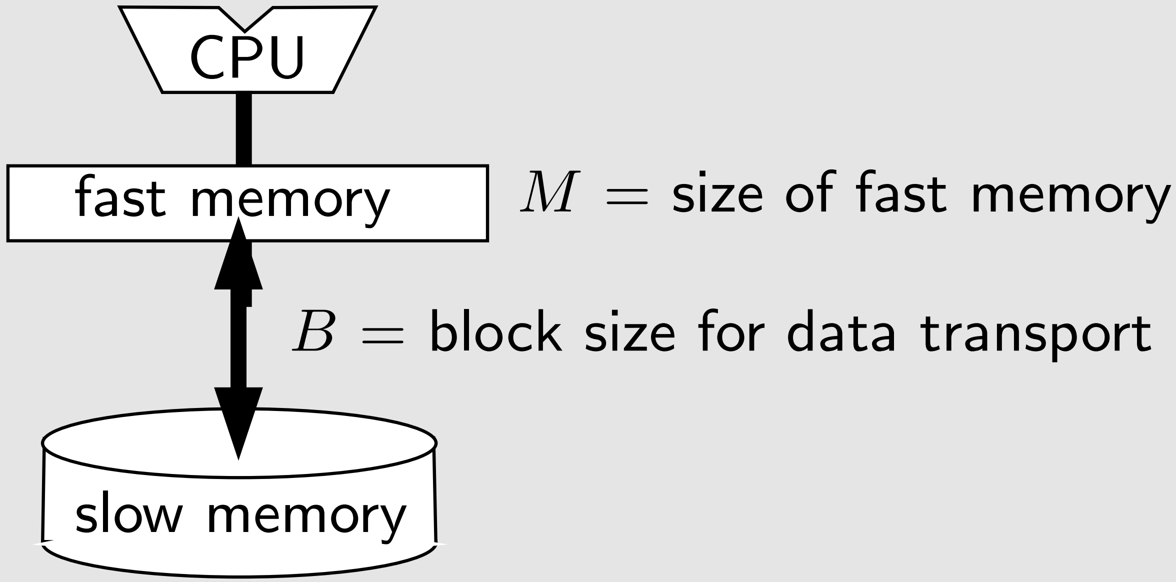
But: algorithm is not allowed to use the value of B and M !



M = size of fast memory

B = block size for data transport

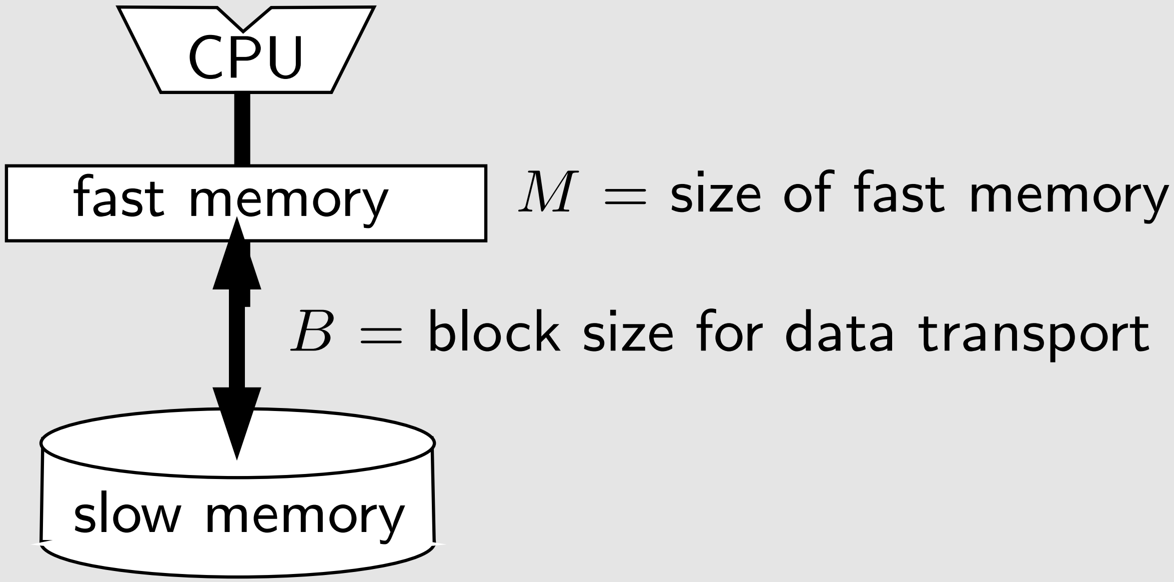
Assumptions:



Assumptions:

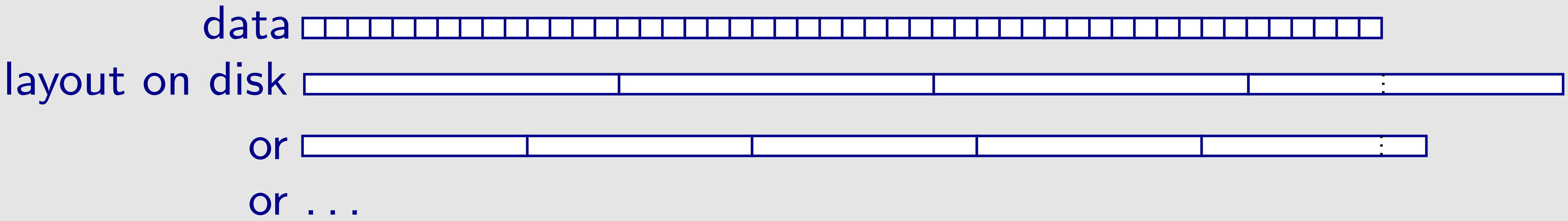
- Data written to "disk" consecutively is stored consecutively on disk

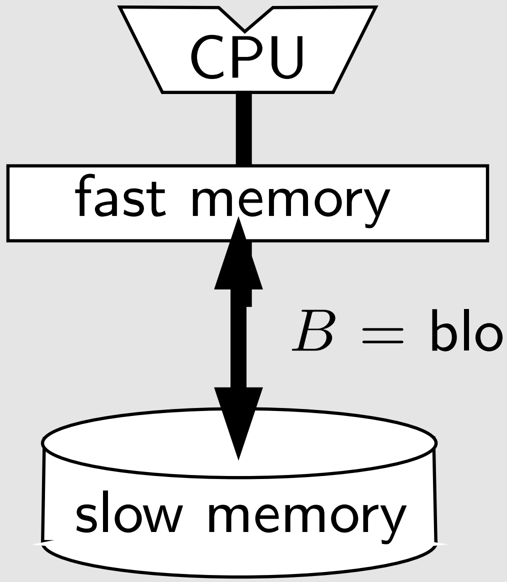
data 



Assumptions:

- Data written to "disk" consecutively is stored consecutively on disk





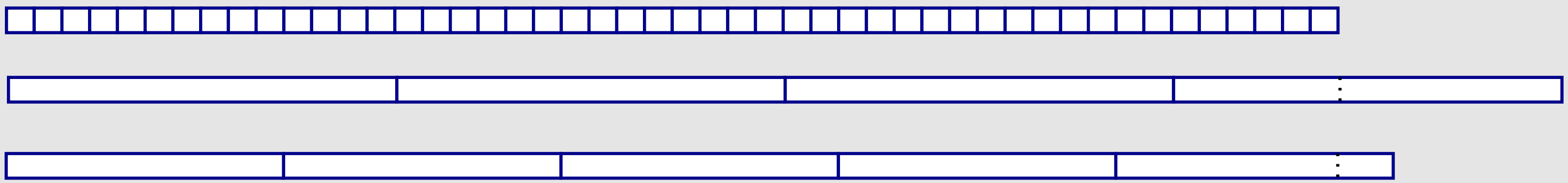
M = size of fast memory

B = block size for data transport

Assumptions:

- Data written to "disk" consecutively is stored consecutively on disk

data
layout on disk
or
or . . .



- Operating system uses optimal replacement strategy

A cache-oblivious algorithm for smallest enclosing disk

CacheObliviousSmallestDisk(P)

if (# points in P) ≤ 3
then **return** smallest disk for P
else $(P_1, P_2) :=$ *RandomSplit* (P)
    $D :=$ *CacheObliviousSmallestDisk*(P_1)
    for all $P[i] \in G_2$
      do if $P[i] \notin D$
          then $D_i :=$ smallest disk for P with $P[i]$ on its boundary
    return best of all computed disks

With: S. Cabello, X. Goaoc, M. Schroders

CacheObliviousSmallestDisk(P)

if (# points in P) ≤ 3
then **return** smallest disk for P
else $(P_1, P_2) :=$ *RandomSplit* (P)
    $D :=$ *CacheObliviousSmallestDisk*(P_1)
    for all $P[i] \in G_2$
      do if $P[i] \notin D$
        then $D_i :=$ smallest disk for P with $P[i]$ on its boundary
    return best of all computed disks

RandomSplit(P)

for $i := 1$ **to** N
  do $r :=$ random number in range $[0, 1]$
    if $r < 1/2$
    then Put $P[i]$ into P_1
    else Put $P[i]$ into P_2
return (P_1, P_2)

With: S. Cabello, X. Goaoc, M. Schroders old algorithm:

RandomPermute(P)
1. **for** $i := 1$ **to** $N-1$
2. **do** $r :=$ random integer in range $i \ldots N$
3. swap $P[i]$ and $P[r]$

E[#disk operations] $= (N-1)\cdot(1-\frac{M}{N})$

TU/e

A cache-oblivious algorithm for smallest disk: analysis

old algorithm:

RandomPermute(P)
for $i := 1$ **to** $N - 1$
    do $r :=$ random integer in range $i \dots N$
        swap $P[i]$ and $P[r]$

E[#disk operations] $= (N-1) \cdot (1 - \frac{M}{N})$

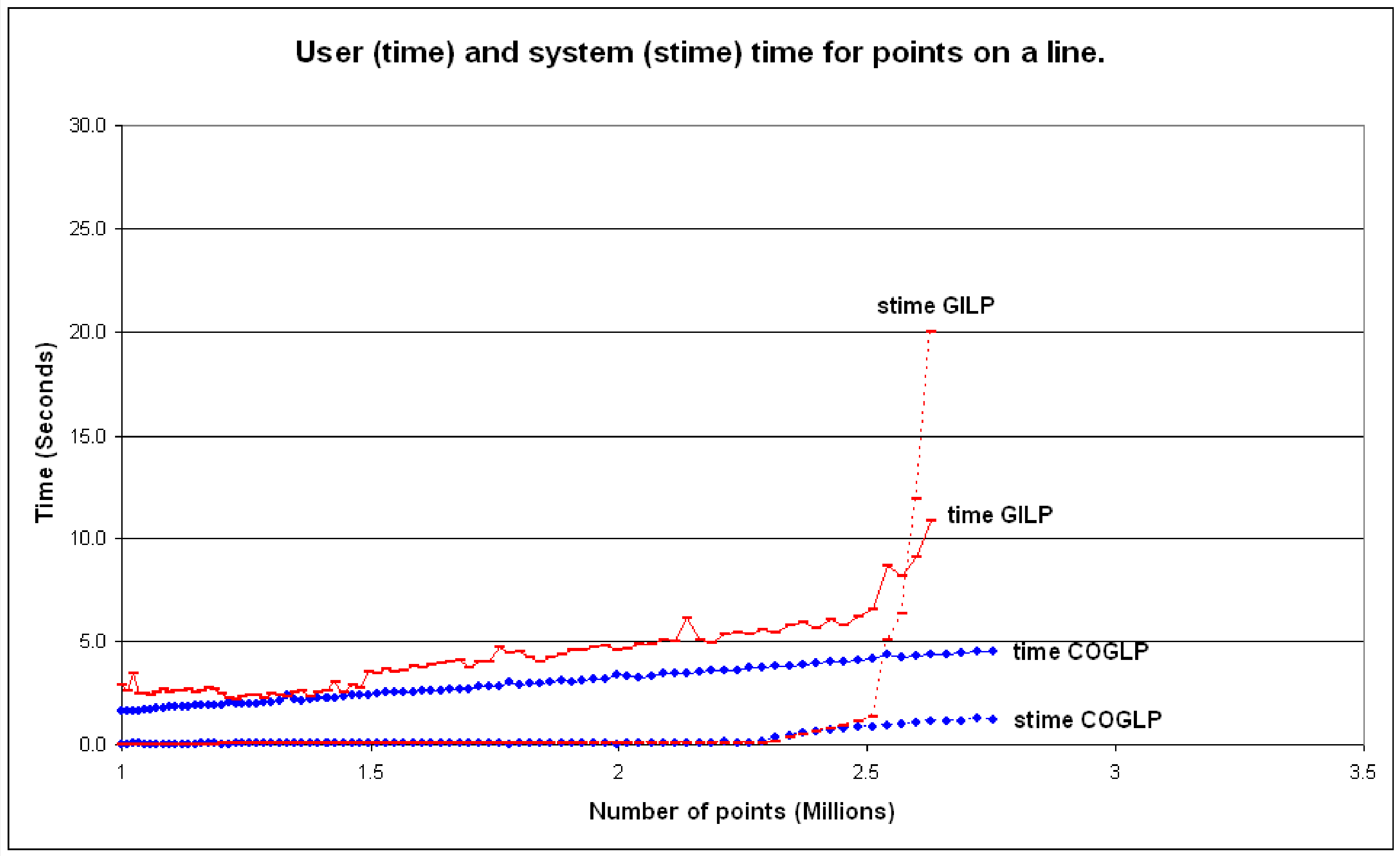
new algorithm:

RandomSplit(P)
for $i := 1$ **to** N
    do $r :=$ random number in range $[0, 1]$
        if $r < 1/2$ **then** Put $P[i]$ into P_1 **else** Put $P[i]$ into P_2
return (P_1, P_2)

layout on disk 

E[#disk operations] $\leq N/B$

Smallest enclosing disk: experiments

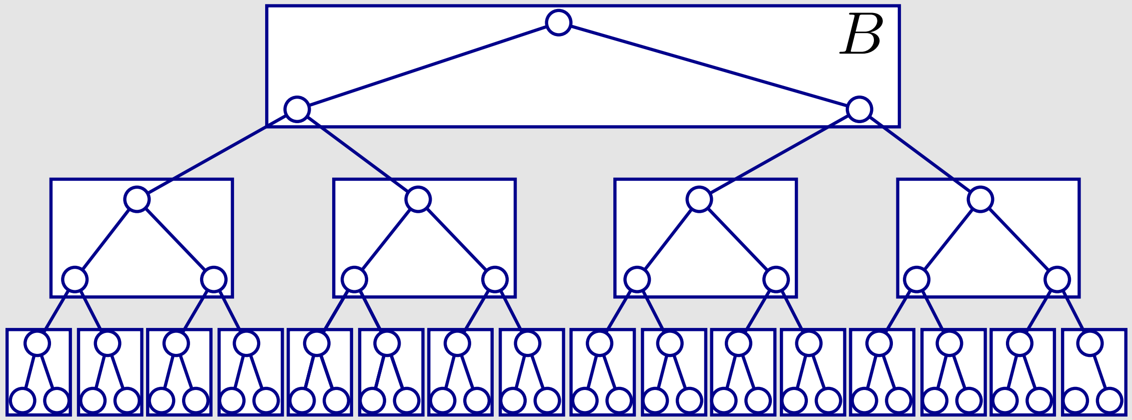


Pentium 4, 2.60GHz

$\approx$ 89 MB main memory available to the program

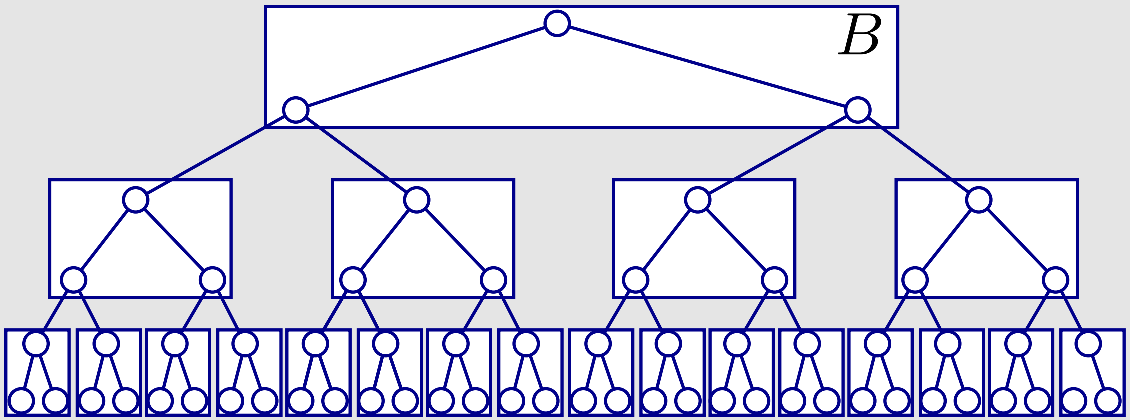
regular (cache-aware) B-tree:

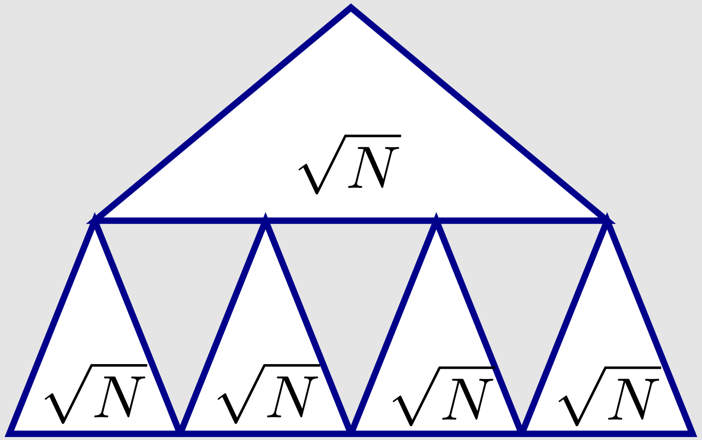
- blocks: subtrees of size B



regular (cache-aware) B-tree:

- blocks: subtrees of size B

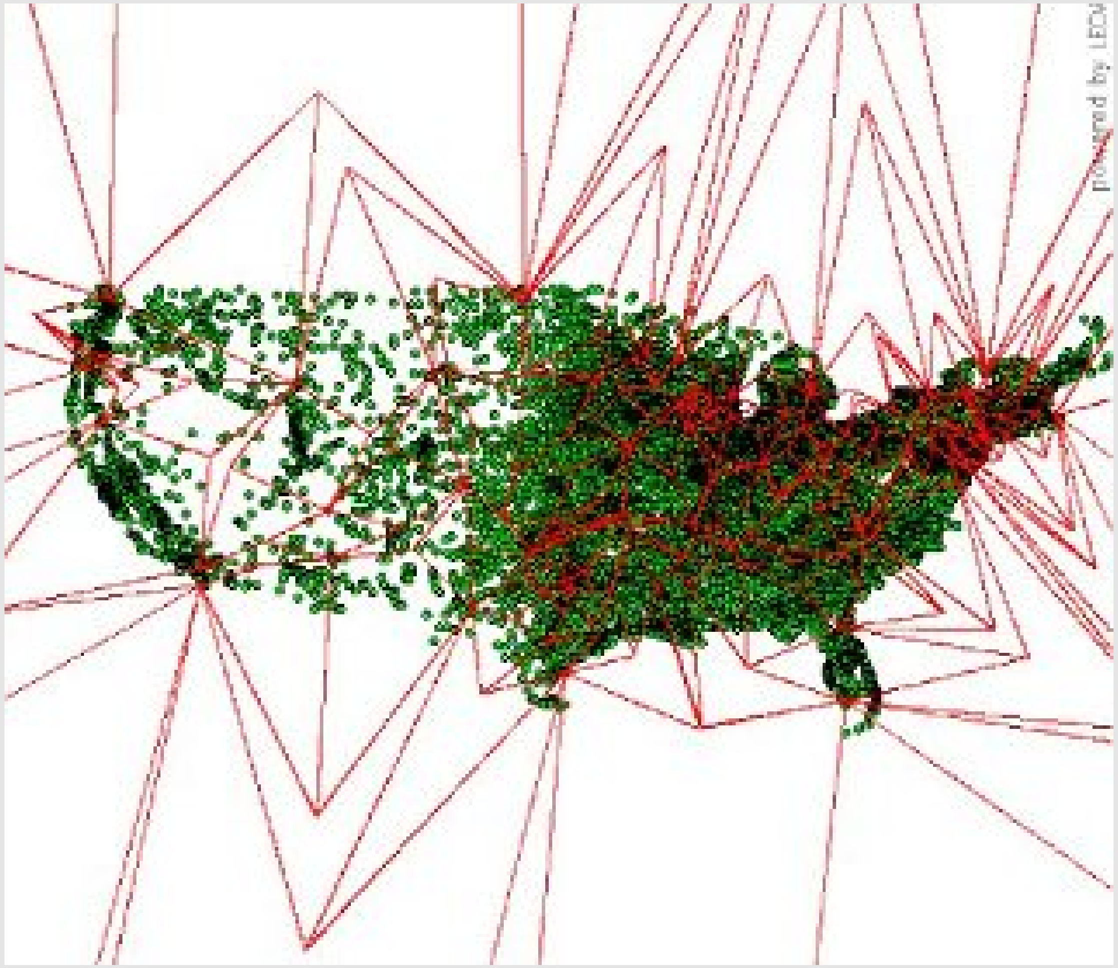




cache-oblivious B-tree:

- cut tree into subtree at middle level; gives 1 top tree, $\sqrt{N}$ lower trees
- first, write top to disk recursively
- next, write lower trees to disk recursively

search visits $O(\log N / \log B)$ blocks



Kumar: Voronoi diagrams of up to 300 M points

Conclusions

- I/O- and caching behavior crucial for massive data sets
- algorithms community is now addressing these issues
- I/O-efficient algorithms
  - have proven their value for various practical problems
  - need tuning for hardware, do not optimize caching behavior
- cache-oblivious algorithms:
  - ideal in theory: no tuning, good on all cache-levels
  - practical relevance needs further investigation