ad clicks   gigabytes   digital elevation models

chain store purchases   hyperlinks   terabytes

cache-efficient algorithms and data structures for data that does not fit in memory

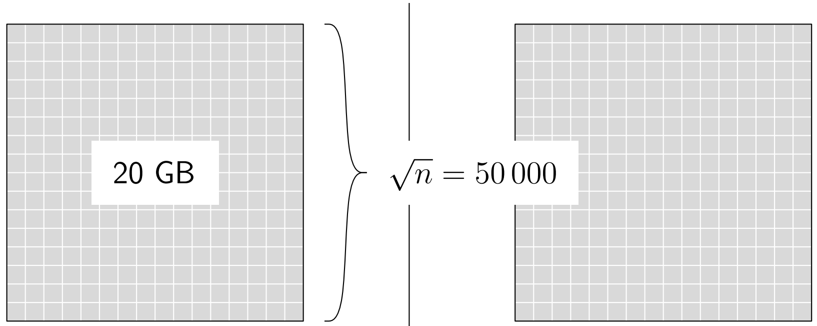
cell phone tracking data

Earth surface at 30m resolution, 4 bytes/sample = 600 GB

text indexing

phone calls   remote sensing data

$\sqrt{n} \times \sqrt{n}$ matrix of n cells. To do: set $M[i,j]$ to $i+j$.



Algorithm 1:

for $row \leftarrow 1$ **to** $\sqrt{n}$
    for $col \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

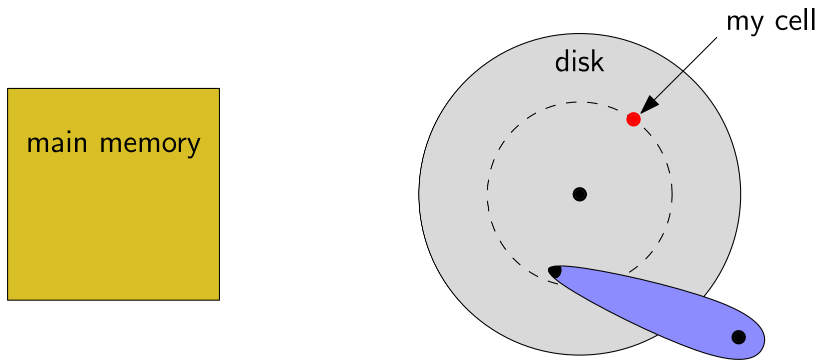
Running time: $\Theta(n)$

Algorithm 2:

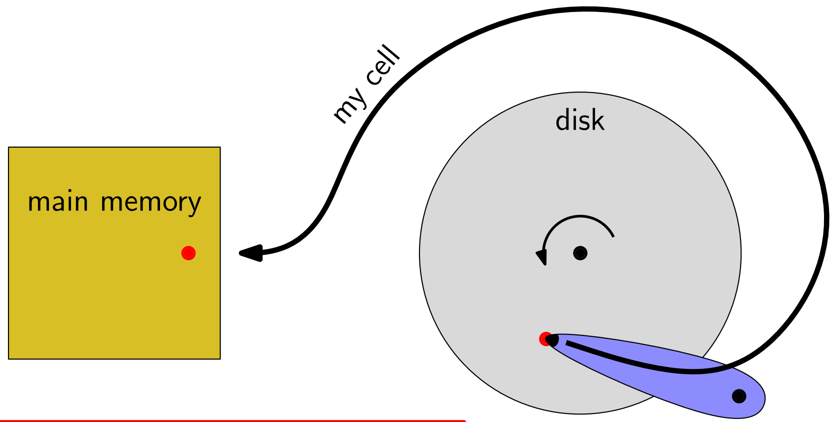
for $col \leftarrow 1$ **to** $\sqrt{n}$
    for $row \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

Running time: $\Theta(n)$

Getting cells from disk into memory

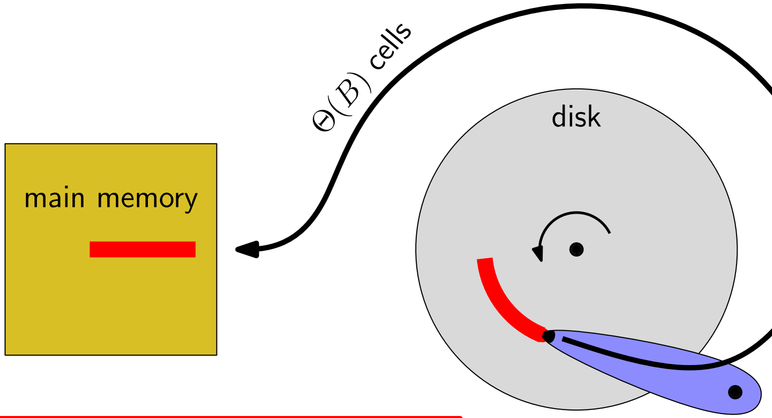


Getting cells from disk into memory



after 10 milliseconds: 1 cell

Getting cells from disk into memory

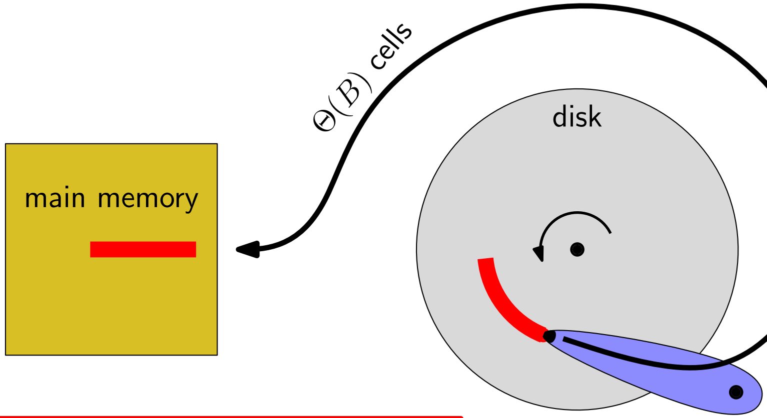


after 10 milliseconds: 1 cell

after 11 milliseconds: 10 000 cells

B = #bytes in one I/O

Getting cells from disk into memory

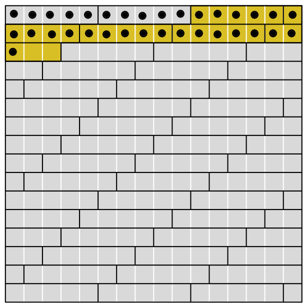


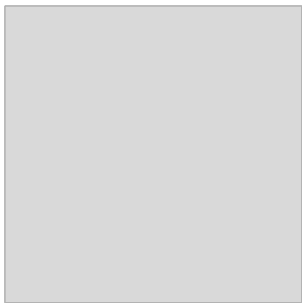
after 10 milliseconds: 1 cell

after 20 milliseconds: 100 000 cells

B = #bytes in one I/O

$\sqrt{n} \times \sqrt{n}$ matrix of n cells. To do: set $M[i, j]$ to $i + j$.





Algorithm 1:

for $row \leftarrow 1$ **to** $\sqrt{n}$
    for $col \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

I/O's: $\Theta(n/B) \approx 5$ minutes

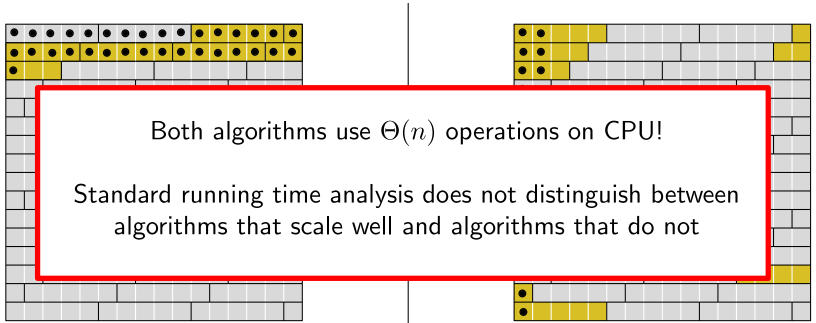
Algorithm 2:

for $col \leftarrow 1$ **to** $\sqrt{n}$
    for $row \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

Running time: $\Theta(n)$

B = #bytes in one I/O   main memory

$\sqrt{n} \times \sqrt{n}$ matrix of n cells. To do: set $M[i, j]$ to $i + j$.



Both algorithms use $\Theta(n)$ operations on CPU!

Standard running time analysis does not distinguish between algorithms that scale well and algorithms that do not

Algorithm 1:

for $row \leftarrow 1$ **to** $\sqrt{n}$
    for $col \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

I/O's: $\Theta(n/B) \approx 5$ minutes

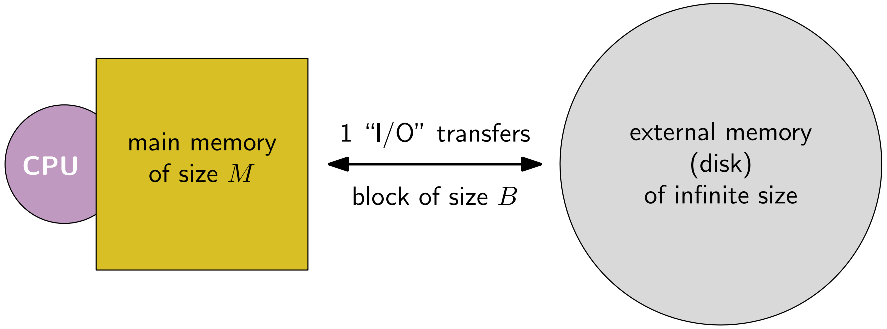
Algorithm 2:

for $col \leftarrow 1$ **to** $\sqrt{n}$
    for $row \leftarrow 1$ **to** $\sqrt{n}$
        $A[row, col] \leftarrow row + col$

I/O's: $\Theta(n) \approx 10$ months

B = #bytes in one I/O   main memory

Analysing I/O-efficiency: model of computation

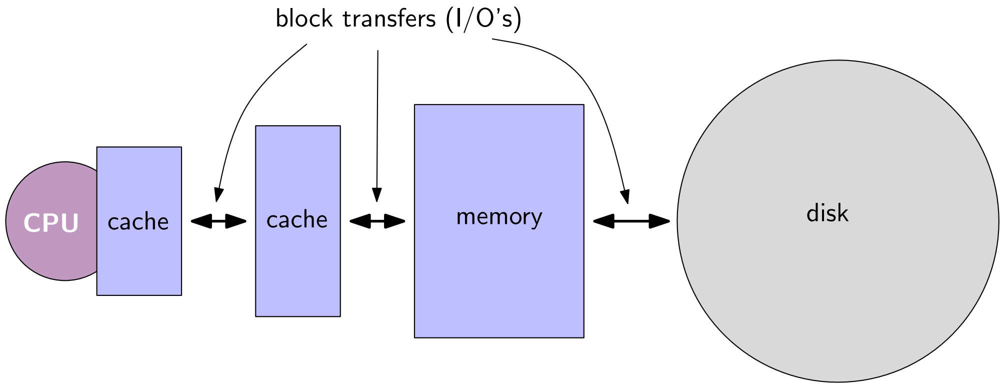


CPU only operates on data in main memory (for free)

I/O-efficiency $=$ number of I/O's as function of M, B, and input parameters (for example n)

$B = \#$bytes in one I/O $\quad M = \#$bytes of main memory

Analysing I/O-efficiency: model of computation

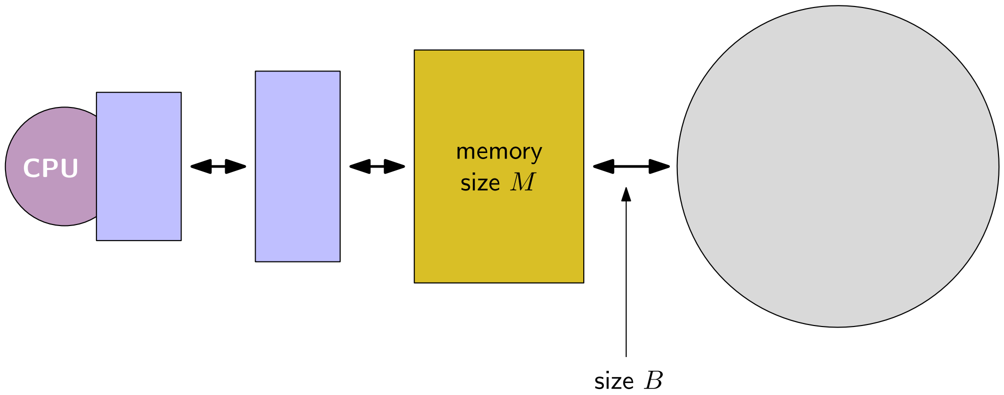


CPU only operates on data in main memory (for free)

I/O-efficiency $=$ number of I/O's as function of M, B, and input parameters (for example n)

$B = \#$bytes in one I/O $\quad M = \#$bytes of main memory

Analysing I/O-efficiency: model of computation

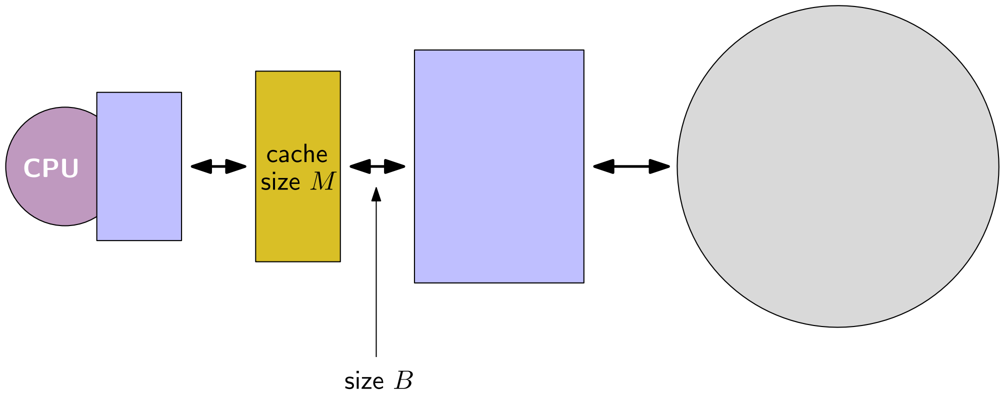


CPU only operates on data in main memory (for free)

I/O-efficiency $=$ number of I/O's as function of M, B, and input parameters (for example n)

$B = \#$bytes in one I/O $\quad M = \#$bytes of main memory or cache

Analysing I/O-efficiency: model of computation

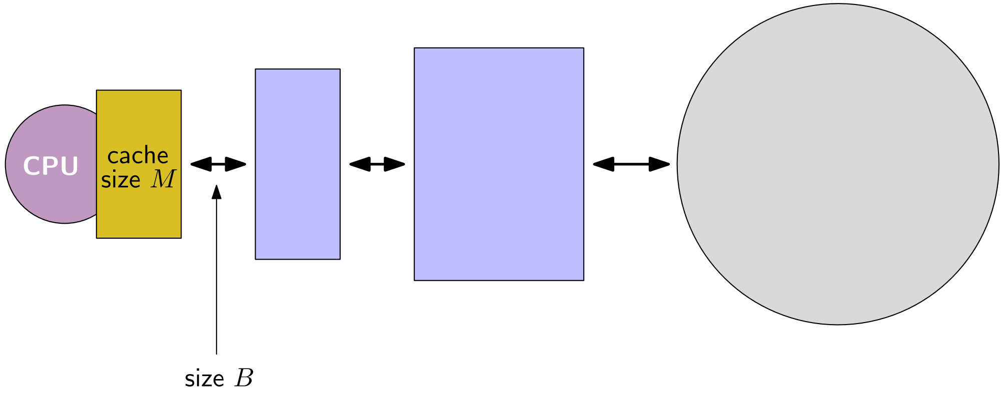


CPU only operates on data in main memory (for free)

I/O-efficiency $=$ number of I/O's as function of M, B, and input parameters (for example n)

$B = \#$bytes in one I/O $\quad M = \#$bytes of main memory or cache

Analysing I/O-efficiency: model of computation

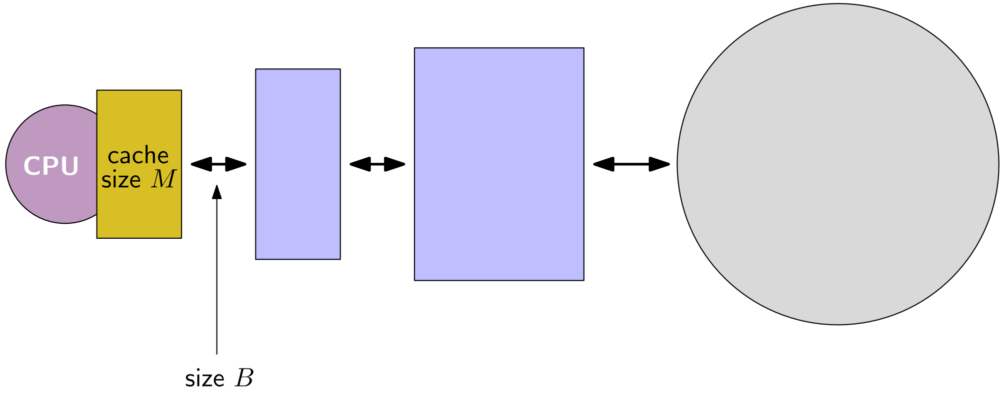


CPU only operates on data in main memory (for free)

I/O-efficiency = number of I/O's as function of M, B, and input parameters (for example n)

B = #bytes in one I/O M = #bytes of main memory or cache

Analysing I/O-efficiency: model of computation



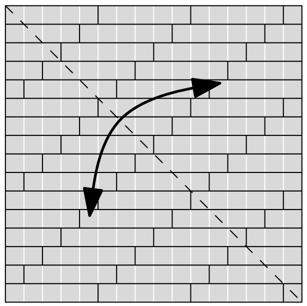
CPU only operates on data in main memory (for free)

I/O-efficiency = number of I/O's as function of M, B, and input parameters (for example n)

frequent assumption

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Transposing a matrix

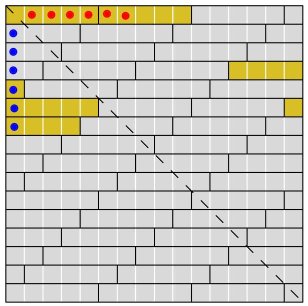


First attempt:

for $i \leftarrow 1$ **to** $\sqrt{n}$
  for $j \leftarrow i + 1$ **to** $\sqrt{n}$
    swap($A[i,j]$, $A[j,i]$)

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Transposing a matrix



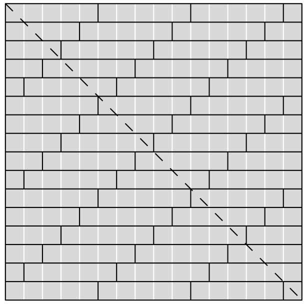
First attempt:

for $i \leftarrow 1$ **to** $\sqrt{n}$
  for $j \leftarrow i + 1$ **to** $\sqrt{n}$
    swap($A[i,j]$, $A[j,i]$)

I/O's: $\Theta(n) \approx$ many months

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Transposing a matrix



Algorithm $\mathrm{T}(starti, startj, endi, endj)$:

if $endj \le starti$ **then return**

if $starti = endi$ **then**
    swap$(A[endi, endj], A[endj, endi])$

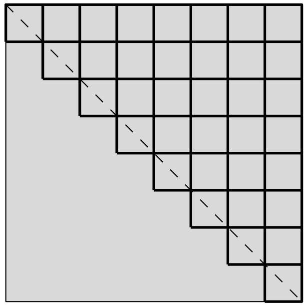
else
    $midi \leftarrow \lfloor (starti + endi)/2 \rfloor$
    $midj \leftarrow \lfloor (startj + endj)/2 \rfloor$

    $\mathrm{T}(starti, startj, midi, midj)$
    $\mathrm{T}(starti, midj + 1, midi, endj)$
    $\mathrm{T}(midi + 1, startj, endi, midj)$
    $\mathrm{T}(midi + 1, midj + 1, endi, endj)$

return

Initial call: $\mathrm{T}(1, 1, \sqrt{n}, \sqrt{n})$

Transposing a matrix



$O(n/M)$ calls on submatrices of size $\ge M/100$, $\le M/25$

Algorithm $\mathrm{T}(starti, startj, endi, endj)$:

if $endj \le starti$ **then return**

if $starti = endi$ **then**
    swap$(A[endi, endj], A[endj, endi])$

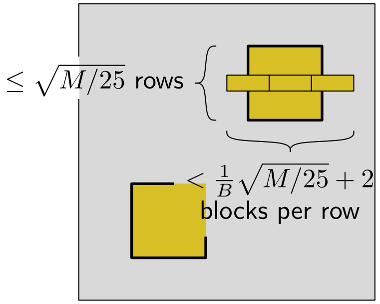
else
    $midi \leftarrow \lfloor (starti + endi)/2 \rfloor$
    $midj \leftarrow \lfloor (startj + endj)/2 \rfloor$

    $\mathrm{T}(starti, startj, midi, midj)$
    $\mathrm{T}(starti, midj + 1, midi, endj)$
    $\mathrm{T}(midi + 1, startj, endi, midj)$
    $\mathrm{T}(midi + 1, midj + 1, endi, endj)$

return

Initial call: $\mathrm{T}(1, 1, \sqrt{n}, \sqrt{n})$

Transposing a matrix



$O(n/M)$ calls on submatrices of size $\ge M/100$, $\le M/25$

#blocks in submatrix plus mirror:
less than $2 \cdot \frac{1}{5}\sqrt{M} \cdot (\frac{1}{B} \cdot \frac{1}{5}\sqrt{M} + 2) \le$
$\frac{2}{5}(M/B)(\frac{1}{5} + 2\cancel{B/\sqrt{M}}) \le M/B$

fits in memory!

Total I/O: $\Theta(n/M) \cdot M/B = \Theta(n/B)$

Algorithm $\mathrm{T}(starti, startj, endi, endj)$:

if $endj \le starti$ **then return**

if $starti = endi$ **then**
    swap$(A[endi, endj], A[endj, endi])$

else
    $midi \leftarrow \lfloor (starti + endi)/2 \rfloor$
    $midj \leftarrow \lfloor (startj + endj)/2 \rfloor$

    $\mathrm{T}(starti, startj, midi, midj)$
    $\mathrm{T}(starti, midj + 1, midi, endj)$
    $\mathrm{T}(midi + 1, startj, endi, midj)$
    $\mathrm{T}(midi + 1, midj + 1, endi, endj)$

return

Our goals

CPU operations	I/O operations
$\Theta(1)$	$\Theta(1/B)$ amortized
$\Theta(n)$	$\Theta(n/B)$
$\Theta(\log n)$	at most $\Theta(\log_B \frac{n}{B})$, but preferably: $\Theta(\frac{1}{B}\log_{M/B}\frac{n}{B})$ amortized
$\Theta(n \log n)$	$\Theta(\frac{n}{B}\log_{M/B}\frac{n}{B})$

$$\boxed{\text{Division by } B \text{ is crucial!}}$$

Changing the logarithm: merge sort

Algorithm MergeSort(array A):

if length$(A) \leq 1$ **then**
  return *(array is already sorted)*
else
  divide A into arrays A_1, A_2 of equal size
  MergeSort(A_1); MergeSort(A_2)
  merge A_1 and A_2 into one sorted array B
  replace A by B

Running time:

$\Theta(\log_2 n)$ levels of recursion;

merge takes $\Theta(n)$ per level:

total $\Theta(n \log_2 n)$ time

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Changing the logarithm: merge sort

Algorithm still does the same: merge-sort recursively down to arrays of size 1.
The change is only to clarify how much I/O is done.

Algorithm MergeSort(array A):

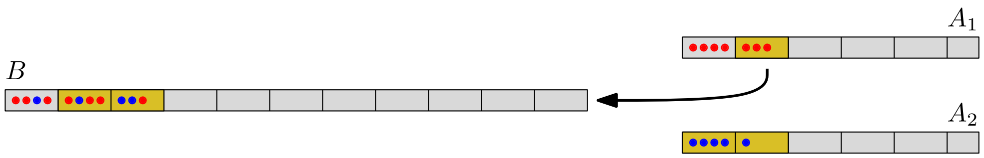
if length$(A) \leq M/2$ **then**
  merge-sort A (loading A into memory once)
  write result to disk
else
  divide A into arrays A_1, A_2 of equal size
  MergeSort(A_1); MergeSort(A_2)
  merge A_1 and A_2 into one sorted array B
  replace A by B

Number of I/O's:

$\Theta(\log_2 (n/M))$ levels of recursion;

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Changing the logarithm: merge sort



Algorithm MergeSort(array A):

if length$(A) \leq M/2$ **then**
  merge-sort A (loading A into memory once)
  write result to disk
else
  divide A into arrays A_1, A_2 of equal size
  MergeSort(A_1); MergeSort(A_2)
  merge A_1 and A_2 into one sorted array B
  replace A by B

Number of I/O's:

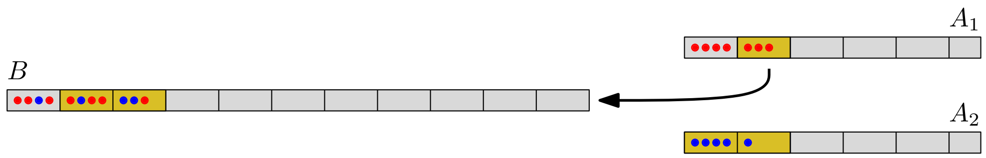
$\Theta(\log_2 (n/M))$ levels of recursion;

merge takes $\Theta(n/B)$ per level:

total $\Theta(\frac{n}{B} \log_2 \frac{n}{M})$ I/O's

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Changing the logarithm: merge sort



Algorithm MergeSort(array A):

if length$(A) \leq 1$ **then**
  return *(array is already sorted)*
else
  divide A into arrays A_1, A_2 of equal size
  MergeSort(A_1); MergeSort(A_2)
  merge A_1 and A_2 into one sorted array B
  replace A by B

Number of I/O's:

$\Theta(\log_2 (n/M))$ levels of recursion;

merge takes $\Theta(n/B)$ per level:

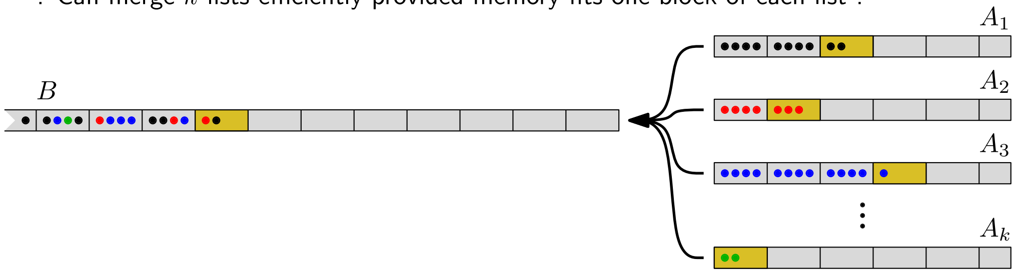
total $\Theta(\frac{n}{B} \log_2 \frac{n}{M})$ I/O's

analysis also applies to unmodified algorithm

B = #bytes in one I/O M = #bytes of main memory or cache $M \geq B^2$

Changing the logarithm: merge sort

! Can merge k lists efficiently provided memory fits one block of each list !



Algorithm MergeSort(array A):

if length$(A) \leq M/2$ **then**
  merge-sort A (loading A into memory once)
  write result to disk
else
  divide A into arrays $A_1, ..., A_k$ of equal size
  for $i \leftarrow 1$ **to** k **do** MergeSort(A_i)
  merge $A_1, ..., A_k$ into one sorted array B
  replace A by B

Number of I/O's:

$\Theta(\log_k(n/M))$ levels of recursion;

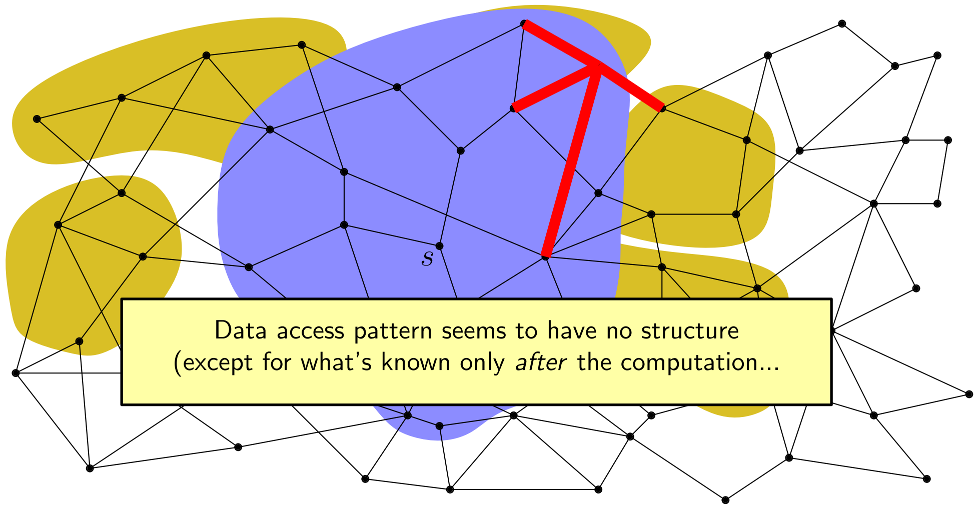
merge takes $\Theta(n/B)$ per level:

total $\Theta(\frac{n}{B} \log_k \frac{n}{M})$ I/O's

$k = \frac{M}{B} - 1$: get $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$

Is it always this simple?

Try Dijkstra's single-source shortest paths algorithm:
visiting vertices by increasing distance from source node s



What makes algorithms I/O-(in)efficient?

1 I/O per operation is too much! You want $\Theta(1/B)$ amortized.

What makes algorithms I/O-efficient?

- spatial locality:
  when algorithm accesses data item, it accesses nearby data around the same time;
  example: scanning in arrays
- temporal locality:
  the moments of access to a data item are clustered in time.

I/O-efficient randomised algorithms are still very well possible and useful!

What makes algorithms I/O-inefficient?

- random/unpredictable/unstructured jumps to memory locations:
  pointer-based data structures are often horribly inefficient with data on disk.
- (accidentally) sabotaging spatial locality:
  for example traversing a matrix orthogonally to its lay-out in memory

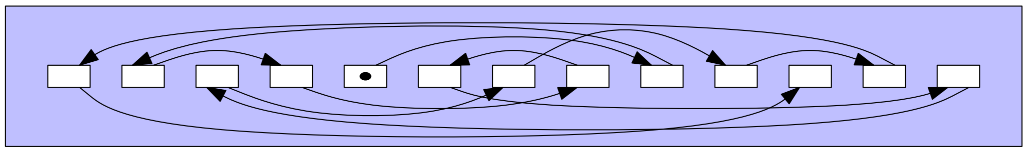
Some examples

I/O-Efficient:

Array-based implementations of stacks and queues:
$\Theta(1)$ time, $\Theta(1/B)$ amortized I/O's per operation,
thanks to spatial locality.

Not I/O-efficient:

Linked-list-based stacks and queues (with dynamic memory allocation):
$\Theta(1)$ time, $\Theta(1)$ I/O's per operation,
traversing a linked list may cause a jump to a block that is currently not in cache *every time*:



Some examples

I/O-Efficient:

Smart B-trees
(trees in which each node is a little subtree of size $\Theta(B)$, stored in one block on disk):
$\Theta(\log n)$ time, $\Theta(\log_B n)$ I/O's per operation,
thanks to spatial locality.

Still not great:
especially for priority
queues we would like
$\Theta(\frac{1}{B} \log_{M/B} \frac{n}{B})$
amortized.

Not I/O-efficient:

Red-black trees:
$\Theta(\log n)$ time, $\Theta(\log n)$ I/O's per operation,
due to following pointers

Array-based heaps:
$\Theta(\log n)$ time, $\Theta(\log n)$ I/O's per operation,
due to the unpredictable access pattern of HEAPIFY

Some examples

I/O-Efficient:

$\Theta(M/B)$-way mergesort, $\Theta(M/B)$-way quicksort:
$\Theta(n \log n)$ time, $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/O's,
thanks to spatial and temporal locality

Medium:

2-way mergesort, 2-way quicksort:
$\Theta(n \log n)$ time, $\Theta(\frac{n}{B} \log \frac{n}{M})$ I/O's,
good spatial locality but poor temporal locality:
on average, every time a data item is read from disk, it is compared to only two others

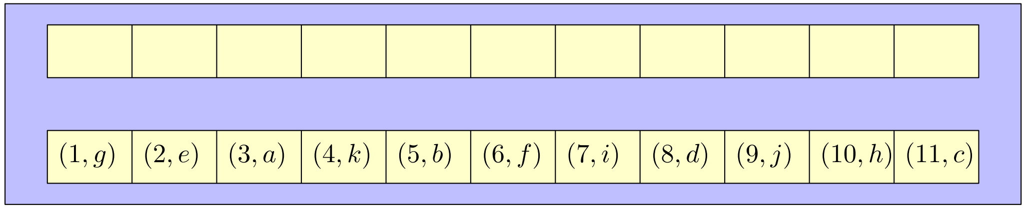
Not I/O-efficient:

Heap sort with array-based heaps: $\Theta(n \log n)$ I/O's,
counting sort: $\Theta(n)$ I/O's.

Some examples

Some things that are easily done in linear time in main memory,
cannot be done I/O-efficiently (with $\Theta(1/B)$ I/O's per operation),
and need completely different algorithms.

Example: permuting. Trivial linear-time algorithm is horribly I/O-inefficient.
Theory fact: I/O-efficient permutation is as difficult as I/O-efficient sorting



Some examples

Some things that are easily done in linear time in main memory,
cannot be done I/O-efficiently (with $\Theta(1/B)$ I/O's per operation),
and need completely different algorithms.

Example: permuting. Trivial linear-time algorithm is horribly I/O-inefficient.
Theory fact: I/O-efficient permutation is as difficult as I/O-efficient sorting

Example: breadth-first search on graph $G = (V, E)$ with $|E| = O(|V|)$.
In memory: $O(V)$ time. On disk: best known algo needs $\Omega(V/\sqrt{B})$ I/O's.

Example: depth-first search on graph $G = (V, E)$ with $|E| = O(|V|)$.
In memory: $O(V)$ time. On disk: best known algorithm needs $\Theta(V)$ I/O's.

Theory question: can we do BFS and DFS as fast as sorting?
(up to a constant factor)