2IL35 I/O-efficient algorithms (edition 2012)
Herman Haverkort

cache-efficient algorithms and data structures for data that does not fit in memory
2IL35 I/O-efficient algorithms
(special edition 2010)
Herman Haverkort

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

Ad clicks
Hyperlinks
digital elevation models
Chain store purchases
terabytes
gigabytes
Cell phone tracking data
3D medical images
terabytes
Petabytes
Text indexing
Phone calls
Remote sensing data
Earth surface at 30m resolution, 4 bytes/sample = 600 GB

2IL35 I/O-efficient algorithms (special edition 2010)
Herman Haverkort

cache-efficient algorithms and data structures for data that does not fit in memory
\(\sqrt{n} \times \sqrt{n}\) matrix of \(n\) cells. To do: set \(M[i, j]\) to \(i + j\).

Algorithm 1:

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

Running time: \(\Theta(n)\)
Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

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

$\sqrt{n} = 50000$

20 GB
Getting cells from disk into memory

- Disk
- Read/write head
- Main memory
- My cell
Getting cells from disk into memory
Getting cells from disk into memory

main memory

disk

my cell
Getting cells from disk into memory

main memory

disk

my cell
Getting cells from disk into memory

main memory

disk

my cell
Getting cells from disk into memory

main memory

disk

my cell
Getting cells from disk into memory
Getting cells from disk into memory

- Disk
- Read/write head
- My cell
- Main memory
Getting cells from disk into memory

- main memory
- disk
- read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk
read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

- main memory
- disk
- read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

- Disk
- Read/write head
- Main memory
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

- Main memory
- Disk
- Read/write head
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

- Disk
- Read/write head
- Main memory
Getting cells from disk into memory

main memory

disk

read/write head
Getting cells from disk into memory

main memory

read/write head
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
Getting cells from disk into memory

\[ \Theta(B) \text{ cells} \]

After 10 milliseconds: 1 cell

After 11 milliseconds: 10,000 cells

\[ B = \#\text{bytes in one I/O} \]
Getting cells from disk into memory

\[ \Theta(B) \text{ cells} \]

main memory

\begin{align*}
\text{after 10 milliseconds: 1 cell} \\
\text{after 20 milliseconds: 100,000 cells}
\end{align*}

\[ B = \# \text{bytes in one I/O} \]
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

```plaintext
for row ← 1 to \( \sqrt{n} \)
  for col ← 1 to \( \sqrt{n} \)
    \( A[row, col] \) ← row + col
```

Running time: \( \Theta(n) \)

Algorithm 2:

```plaintext
for col ← 1 to \( \sqrt{n} \)
  for row ← 1 to \( \sqrt{n} \)
    \( A[row, col] \) ← row + col
```

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$
Algorithm 1:

for row ← 1 to \( \sqrt{n} \)
    for col ← 1 to \( \sqrt{n} \)
        \( A[row, col] \leftarrow row + col \)

Running time: \( \Theta(n) \)

\( B = \text{\#bytes in one I/O} \)

main memory

Algorithm 2:

for col ← 1 to \( \sqrt{n} \)
    for row ← 1 to \( \sqrt{n} \)
        \( A[row, col] \leftarrow row + col \)

Running time: \( \Theta(n) \)
Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

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

\( B = \# \text{bytes in one I/O} \) main memory
Algorithm 1:

for row ← 1 to \(\sqrt{n}\)
  for col ← 1 to \(\sqrt{n}\)
    \(A[row, col] \leftarrow row + col\)

Running time: \(\Theta(n)\)

Algorithm 2:

for col ← 1 to \(\sqrt{n}\)
  for row ← 1 to \(\sqrt{n}\)
    \(A[row, col] \leftarrow row + col\)

Running time: \(\Theta(n)\)

\(B = \#\text{bytes in one I/O}\)
Algorithm 1:

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

Running time: $\Theta(n)$

\[
B = \#\text{bytes in one I/O}
\]

Algorithm 2:

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

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

Algorithm 1:

\begin{verbatim}
for row ← 1 to $\sqrt{n}$
    for col ← 1 to $\sqrt{n}$
        $A[row, col] ← row + col$
\end{verbatim}

Running time: $\Theta(n)$

Algorithm 2:

\begin{verbatim}
for col ← 1 to $\sqrt{n}$
    for row ← 1 to $\sqrt{n}$
        $A[row, col] ← row + col$
\end{verbatim}

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

\[
\begin{align*}
&\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \( \Theta(n) \)

Algorithm 2:

\[
\begin{align*}
&\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
Algorithm 1:

\begin{verbatim}
for row ← 1 to \sqrt{n}
    for col ← 1 to \sqrt{n}
        A[row, col] ← row + col
\end{verbatim}

Running time: $\Theta(n)$

Algorithm 2:

\begin{verbatim}
for col ← 1 to \sqrt{n}
    for row ← 1 to \sqrt{n}
        A[row, col] ← row + col
\end{verbatim}

Running time: $\Theta(n)$

\[ B = \# \text{bytes in one I/O} \]
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B =$ #bytes in one I/O
Algorithm 1:

for row ← 1 to \(\sqrt{n}\) 
  for col ← 1 to \(\sqrt{n}\)
    \(A[row, col] \leftarrow row + col\)

Running time: \(\Theta(n)\)

Algorithm 2:

for col ← 1 to \(\sqrt{n}\) 
  for row ← 1 to \(\sqrt{n}\)
    \(A[row, col] \leftarrow row + col\)

Running time: \(\Theta(n)\)

\(B = \#\text{bytes in one I/O}\) main memory

\(\sqrt{n} \times \sqrt{n}\) matrix of \(n\) cells. To do: set \(M[i, j]\) to \(i + j\).
Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

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

\( B \) = \#bytes in one I/O main memory
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)  

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

Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\[
B = \# \text{bytes in one I/O}
\]

main memory
Algorithm 1:

```
for row ← 1 to √n
  for col ← 1 to √n
    A[row, col] ← row + col
```

Running time: $\Theta(n)$

Algorithm 2:

```
for col ← 1 to √n
  for row ← 1 to √n
    A[row, col] ← row + col
```

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$

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

Algorithm 1:

\[
\begin{align*}
\text{for } \text{row} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{for } \text{col} & \leftarrow 1 \text{ to } \sqrt{n} \\
A[\text{row}, \text{col}] & \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \(\Theta(n)\)

Algorithm 2:

\[
\begin{align*}
\text{for } \text{col} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{for } \text{row} & \leftarrow 1 \text{ to } \sqrt{n} \\
A[\text{row}, \text{col}] & \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \(\Theta(n)\)

\[B = \#\text{bytes in one I/O} \quad \text{main memory}\]
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

\begin{verbatim}
for row ← 1 to $\sqrt{n}$
    for col ← 1 to $\sqrt{n}$
        $A[row, col] ← row + col$
\end{verbatim}

Running time: $\Theta(n)$

Algorithm 2:

\begin{verbatim}
for col ← 1 to $\sqrt{n}$
    for row ← 1 to $\sqrt{n}$
        $A[row, col] ← row + col$
\end{verbatim}

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

\[
\begin{align*}
\text{for } & \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{for } \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{A}[\text{row, col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \( \Theta(n) \)

Algorithm 2:

\[
\begin{align*}
\text{for } & \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{for } \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{A}[\text{row, col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$
Algorithm 1:

\[
\begin{align*}
\text{for } & \text{ \textbf{row} } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{\hspace{1em} \text{for } \textbf{col} \leftarrow 1 \text{ to } \sqrt{n} \\
& \quad \quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: $\Theta(n)$

Algorithm 2:

\[
\begin{align*}
\text{for } & \text{ \textbf{col} } \leftarrow 1 \text{ to } \sqrt{n} \\
& \text{\hspace{1em} \text{for } \textbf{row} \leftarrow 1 \text{ to } \sqrt{n} \\
& \quad \quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$

\[\sqrt{n} \times \sqrt{n} \text{ matrix of } n \text{ cells. To do: set } M[i, j] \text{ to } i + j.\]
Algorithm 1:

```plaintext
for row ← 1 to \( \sqrt{n} \)
    for col ← 1 to \( \sqrt{n} \)
        \( A[row, col] \leftarrow row + col \)
```

Running time: \( \Theta(n) \)

Algorithm 2:

```plaintext
for col ← 1 to \( \sqrt{n} \)
    for row ← 1 to \( \sqrt{n} \)
        \( A[row, col] \leftarrow row + col \)
```

Running time: \( \Theta(n) \)

\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$

main memory

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

Algorithm 1:

```plaintext
for row ← 1 to \( \sqrt{n} 
  for col ← 1 to \( \sqrt{n} 
    A[row, col] ← row + col
```

Running time: \( \Theta(n) \)

Algorithm 2:

```plaintext
for col ← 1 to \( \sqrt{n} 
  for row ← 1 to \( \sqrt{n} 
    A[row, col] ← row + col
```

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

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

Running time: \( \Theta(n) \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\( B = \text{#bytes in one I/O} \)
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \# \text{bytes in one I/O}$
Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

for col ← 1 to $\sqrt{n}$
  for row ← 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 $i + j$. 

$B =$ #bytes in one I/O

main memory
Algorithm 1:

\[
\begin{align*}
&\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: $\Theta(n)$

Algorithm 2:

\[
\begin{align*}
&\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: $\Theta(n)$

\[B = \#\text{bytes in one I/O}\]

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

Algorithm 1:

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

Running time: $\Theta(n)$

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
\[
\sqrt{n} \times \sqrt{n} \text{ matrix of } n \text{ cells. To do: set } M[i, j] \text{ to } i + j.
\]

<table>
<thead>
<tr>
<th>Algorithm 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for row ← 1 to (\sqrt{n})</td>
</tr>
<tr>
<td>for col ← 1 to (\sqrt{n})</td>
</tr>
<tr>
<td>[A[row, col] \leftarrow row + col]</td>
</tr>
<tr>
<td>Running time: (\Theta(n))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for col ← 1 to (\sqrt{n})</td>
</tr>
<tr>
<td>for row ← 1 to (\sqrt{n})</td>
</tr>
<tr>
<td>[A[row, col] \leftarrow row + col]</td>
</tr>
<tr>
<td>Running time: (\Theta(n))</td>
</tr>
</tbody>
</table>

\[B = \#\text{bytes in one I/O}\] main memory
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

**Algorithm 1:**

```
for row ← 1 to \( \sqrt{n} \)
    for col ← 1 to \( \sqrt{n} \)
        \( A[\text{row, col}] \) ← row + col
```

Running time: \( \Theta(n) \)

**Algorithm 2:**

```
for col ← 1 to \( \sqrt{n} \)
    for row ← 1 to \( \sqrt{n} \)
        \( A[\text{row, col}] \) ← row + col
```

Running time: \( \Theta(n) \)

\( B = \) #bytes in one I/O

main memory
Algorithm 1:

```plaintext
for row ← 1 to √n
  for col ← 1 to √n
    A[row, col] ← row + col
```

Running time: Θ(√n × √n)

Algorithm 2:

```plaintext
for col ← 1 to √n
  for row ← 1 to √n
    A[row, col] ← row + col
```

Running time: Θ(√n × √n)

√n × √n matrix of n cells. To do: set \( M[i, j] \) to \( i + j \).

---

\[ B = \# \text{bytes in one I/O} \] main memory
Algorithm 1:

for row ← 1 to √n
  for col ← 1 to √n
    \( A[row, col] \) ← row + col

Running time: \( \Theta(n) \)

Algorithm 2:

for col ← 1 to √n
  for row ← 1 to √n
    \( A[row, col] \) ← row + col

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
Algorithm 1:

\[
\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
\quad \text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col} \\
\]

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

Algorithm 2:

\[
\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
\quad \text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col} \\
\]

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

**Algorithm 1:**

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

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

**Algorithm 2:**

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

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

Algorithm 2:

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

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 $i + j$.

---

**Algorithm 1:**

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

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

---

**Algorithm 2:**

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

Running time: $\Theta(n)$

---

$B = \#\text{bytes in one I/O}$
Algorithm 1:

$$\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n}$$

$$\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n}$$

$$A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}$$

I/O’s: $\Theta(n/B) \approx 5 \text{ minutes}$

Algorithm 2:

$$\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n}$$

$$\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n}$$

$$A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}$$

Running time: $\Theta(n)$

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

Algorithm 1:

```
for row ← 1 to \( \sqrt{n} \)
    for col ← 1 to \( \sqrt{n} \)
        \( A[row, col] \) ← row + col
```

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

Algorithm 2:

```
for col ← 1 to \( \sqrt{n} \)
    for row ← 1 to \( \sqrt{n} \)
        \( A[row, col] \) ← row + col
```

Running time: \( \Theta(n) \)

\( B = \# \) 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:

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

I/O's: \( \Theta(n/B) \approx 5 \text{ minutes} \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
\[
\sqrt{n} \times \sqrt{n} \text{ matrix of } n \text{ cells. To do: set } M[i, j] \text{ to } i + j.
\]

**Algorithm 1:**

\begin{align*}
\text{for } \text{row} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{for } \text{col} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{A}[\text{row}, \text{col}] & \leftarrow \text{row} + \text{col}
\end{align*}

\text{I/O's: } \Theta(n/B) \approx 5 \text{ minutes}

**Algorithm 2:**

\begin{align*}
\text{for } \text{col} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{for } \text{row} & \leftarrow 1 \text{ to } \sqrt{n} \\
\text{A}[\text{row}, \text{col}] & \leftarrow \text{row} + \text{col}
\end{align*}

\text{Running time: } \Theta(n)

\[B = \#\text{bytes in one I/O}\] main memory
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O} \quad \text{main memory}$
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i,j]$ to $i+j$.

Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$

main memory
Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

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

$B =$ #bytes in one I/O — main memory
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
\[ \sqrt{n} \times \sqrt{n} \] matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

Algorithm 1:

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

I/O’s: \( \Theta(n/B) \approx 5 \text{ minutes} \)

Algorithm 2:

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

Running time: \( \Theta(n) \)

\( B = \#\text{bytes in one I/O} \)

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

Algorithm 1:

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

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

Algorithm 2:

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

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$
\( \sqrt{n} \times \sqrt{n} \) matrix of \( n \) cells. To do: set \( M[i, j] \) to \( i + j \).

**Algorithm 1:**

```plaintext
for row ← 1 to \( \sqrt{n} 
    for col ← 1 to \( \sqrt{n} 
        A[row, col] ← row + col
```

I/O's: \( \Theta(n/B) \approx 5 \text{ minutes} \)

**Algorithm 2:**

```plaintext
for col ← 1 to \( \sqrt{n} 
    for row ← 1 to \( \sqrt{n} 
        A[row, col] ← row + col
```

Running time: \( \Theta(n) \)

\( B = \# \text{bytes in one I/O} \)
Algorithm 1:

\[
\text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
\hspace{1cm} \text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
\hspace{2cm} A[\text{row, col}] \leftarrow \text{row} + \text{col}
\]

1/O’s: $\Theta(n/B) \approx 5$ minutes

Algorithm 2:

\[
\text{for } \text{col} \leftarrow 1 \text{ to } \sqrt{n} \\
\hspace{1cm} \text{for } \text{row} \leftarrow 1 \text{ to } \sqrt{n} \\
\hspace{2cm} A[\text{row, col}] \leftarrow \text{row} + \text{col}
\]

Running time: $\Theta(n)$

$B = \#\text{bytes in one I/O}$

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

Algorithm 1:

\[
\begin{align*}
&\text{for row } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for col } \leftarrow 1 \text{ to } \sqrt{n} \\
&\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

I/O's: \(\Theta(n/B) \approx 5\text{ minutes}\)

Algorithm 2:

\[
\begin{align*}
&\text{for col } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for row } \leftarrow 1 \text{ to } \sqrt{n} \\
&\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\end{align*}
\]

Running time: \(\Theta(n)\)

\(B = \#\text{bytes in one I/O}\)
\[ \sqrt{n} \times \sqrt{n} \text{ matrix of } n \text{ cells. To do: set } M[i, j] \text{ to } i + j. \]

Algorithm 1:

\[
\text{for } \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
\quad \text{for } \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\]

I/O's: \( \Theta(n/B) \approx 5 \text{ minutes} \)

Algorithm 2:

\[
\text{for } \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
\quad \text{for } \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
\qquad A[\text{row}, \text{col}] \leftarrow \text{row} + \text{col}
\]

Running time: \( \Theta(n) \)

\[ B = \# \text{bytes in one I/O} \quad \text{main memory} \]
$\sqrt{n} \times \sqrt{n}$ matrix of $n$ cells. To do: set $M[i, j]$ to $i + j$.

---

**Algorithm 1:**

```latex
\begin{align*}
&\text{for } \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row, col}] \leftarrow \text{row } + \text{col} \\
\end{align*}
```

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

---

**Algorithm 2:**

```latex
\begin{align*}
&\text{for } \text{col } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \text{for } \text{row } \leftarrow 1 \text{ to } \sqrt{n} \\
&\quad \quad A[\text{row, col}] \leftarrow \text{row } + \text{col} \\
\end{align*}
```

Running time: $\Theta(n)$

---

$B = \#\text{bytes in one I/O}$

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

**Algorithm 1:**

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

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

**Algorithm 2:**

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

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

\( B = \#\text{bytes in one I/O} \)
$\sqrt{n} \times \sqrt{n}$ 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 ← 1 to $\sqrt{n}$
    for col ← 1 to $\sqrt{n}$
        $A[row, col] \leftarrow row + col$
```

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

**Algorithm 2:**

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

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

---

$B = \#\text{bytes in one I/O}$

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 $n$)

$B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory}$
Analysing I/O-efficiency: model of computation

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

\[ \text{I/O-efficiency} = \text{number of I/O's as function of } M, B, \text{ and input parameters (for example } n) \]

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory} \]
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 $n$)

$B = \#\text{bytes in one I/O}$  \hspace{1cm} $M = \#\text{bytes of main memory or cache}$
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 $n$)

$B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache}$
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 $n$)

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$
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 $n$)

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

$$M \geq B^2$$
Transposing a matrix

First attempt:

\[
\begin{align*}
\text{for } i & \leftarrow 1 \text{ to } \sqrt{n} \\
& \quad \text{for } j \leftarrow i + 1 \text{ to } \sqrt{n} \\
& \quad \quad \text{swap}(A[i, j], A[j, i])
\end{align*}
\]

\[B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

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

\[B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

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

\[B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

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

\[B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

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

\[B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

\[
\begin{align*}
\text{for } i & \leftarrow 1 \text{ to } \sqrt{n} \\
\quad & \text{for } j \leftarrow i + 1 \text{ to } \sqrt{n} \\
\quad & \text{swap}(A[i, j], A[j, i])
\end{align*}
\]

\[B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2\]
Transposing a matrix

First attempt:

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

I/O's: \( \Theta(n) \approx \text{many months} \)

\[B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2\]
Algorithm $T(start_i, start_j, end_i, end_j)$:

- if $end_j \leq start_i$ then return
- if $start_i = end_i$ then
  - swap($A[end_i, end_j], A[end_j, end_i]$)
- else
  - $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
  - $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
  - $T(start_i, start_j, midi, midj)$
  - $T(start_i, midj + 1, midi, end_j)$
  - $T(midi + 1, start_j, end_i, midj)$
  - $T(midi + 1, midj + 1, end_i, end_j)$

return

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

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

if $end_j \leq start_i$ then return

if $start_i = end_i$ then
    swap($A[end_i, end_j], A[end_j, end_i]$)

else
    $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
    $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$

    $T(start_i, start_j, midi, midj)$
    $T(start_i, midj + 1, midi, end_j)$
    $T(midi + 1, start_j, end_i, midj)$
    $T(midi + 1, midj + 1, end_i, end_j)$

return

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

$B = \#\text{bytes in one I/O}$
$M = \#\text{bytes of main memory or cache}$
$M \geq B^2$
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

1. If $end_j \leq start_i$ then return
2. If $start_i = end_i$ then
   a. Swap $(A[end_i, end_j], A[end_j, end_i])$
3. Else
   a. $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
   b. $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
   c. $T(start_i, start_j, midi, midj)$
   d. $T(start_i, midj + 1, midi, end_j)$
   e. $T(midi + 1, start_j, end_i, midj)$
   f. $T(midi + 1, midj + 1, end_i, end_j)$
4. Return

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

$B = \# \text{bytes in one I/O}$ $M = \# \text{bytes of main memory or cache}$ $M \geq B^2$
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

1. **if** $end_j \leq start_i$ **then** **return**
2. **if** $start_i = end_i$ **then**
   a. `swap(A[end_i, end_j], A[end_j, end_i])`
3. **else**
   a. $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
   b. $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
   c. $T(start_i, start_j, midi, midj)$
   d. $T(start_i, midj + 1, midi, end_j)$
   e. $T(midi + 1, start_j, end_i, midj)$
   f. $T(midi + 1, midj + 1, end_i, end_j)$

**return**

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

$B = \text{#bytes in one I/O}$  \hspace{1cm} $M = \text{#bytes of main memory or cache}$  \hspace{1cm} $M \geq B^2$
Transposing a matrix

Algorithm $T(starti, startj, endi, endj)$:

1. **if** $endj \leq starti$ **then** return

2. **if** $starti = endi$ **then**

3. **else**
   - $midi \leftarrow \lfloor (starti + endi)/2 \rfloor$
   - $midj \leftarrow \lfloor (startj + endj)/2 \rfloor$
   - $T(starti, startj, midi, midj)$
   - $T(starti, midj + 1, midi, endj)$
   - $T(midi + 1, startj, endi, midj)$
   - $T(midi + 1, midj + 1, endi, endj)$

4. **return**

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

\[ B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2 \]
Transposing a matrix

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$

Algorithm $T(start_i, start_j, end_i, end_j)$:

- **if** $end_j \leq start_i$ **then** return
- **if** $start_i = end_i$ **then**
  - swap($A[end_i, end_j], A[end_j, end_i]$)
  - else
    - $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
    - $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
    - $T(start_i, start_j, midi, midj)$
    - $T(start_i, midj + 1, midi, end_j)$
    - $T(midi + 1, start_j, end_i, midj)$
    - $T(midi + 1, midj + 1, end_i, end_j)$
- return

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

$O(n/M)$ calls on submatrices of size $\geq M/100$, $\leq M/25$
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

if $end_j \leq start_i$ then return

if $start_i = end_i$ then
    swap($A[end_i, end_j], A[end_j, end_i]$)

else
    $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
    $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
    $T(start_i, start_j, midi, midj)$
    $T(start_i, midj + 1, midi, end_j)$
    $T(midi + 1, start_j, end_i, midj)$
    $T(midi + 1, midj + 1, end_i, end_j)$

return

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

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

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

$M \geq B^2$
Transposing a matrix

\[ \begin{align*}
\leq \sqrt{\frac{M}{25}} & \text{ rows} \\
\leq \frac{1}{B} \sqrt{\frac{M}{25}} + 2 & \text{ blocks per row}
\end{align*} \]

\(O(n/M)\) calls on submatrices of size
\[\geq \frac{M}{100}, \leq \frac{M}{25}\]

Algorithm \(T(starti, startj, endi, endj):\)

\[\begin{align*}
\text{if } endj \leq starti \text{ then return} \\
\text{if } starti = endi \text{ then} \\
& \text{ swap}(A[endi, endj], A[endj, endi]) \\
\text{else} \\
& midi \leftarrow \lfloor (starti + endi)/2 \rfloor \\
& midj \leftarrow \lfloor (startj + endj)/2 \rfloor \\
& T(starti, startj, midi, midj) \\
& T(starti, midj + 1, midi, endj) \\
& T(midi + 1, startj, endi, midj) \\
& T(midi + 1, midj + 1, endi, endj) \\
\text{return}
\end{align*} \]

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

\(B = \#\text{bytes in one I/O}\) \hspace{1cm} \(M = \#\text{bytes of main memory or cache}\) \hspace{1cm} \(M \geq B^2\)
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

- if $end_j \leq start_i$ then return
- if $start_i = end_i$ then
  - swap($A[end_i, end_j], A[end_j, end_i]$)
- else
  - $midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor$
  - $midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor$
  - $T(start_i, start_j, midi, midj)$
  - $T(start_i, midj + 1, midi, end_j)$
  - $T(midi + 1, start_j, end_i, midj)$
  - $T(midi + 1, midj + 1, end_i, end_j)$

return

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

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

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

$B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2$
Transposing a matrix

Algorithm $T(\text{start}_i, \text{start}_j, \text{end}_i, \text{end}_j)$:

if $\text{end}_j \leq \text{start}_i$ then return

if $\text{start}_i = \text{end}_i$ then
    swap($A[\text{end}_i, \text{end}_j]$, $A[\text{end}_j, \text{end}_i]$)
else
    $\text{midi} \leftarrow \lfloor (\text{start}_i + \text{end}_i)/2 \rfloor$
    $\text{mid}_j \leftarrow \lfloor (\text{start}_j + \text{end}_j)/2 \rfloor$
    $T(\text{start}_i, \text{start}_j, \text{midi}, \text{mid}_j)$
    $T(\text{start}_i, \text{mid}_j + 1, \text{midi}, \text{end}_j)$
    $T(\text{midi} + 1, \text{start}_j, \text{end}_i, \text{mid}_j)$
    $T(\text{midi} + 1, \text{mid}_j + 1, \text{end}_i, \text{end}_j)$

return

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

$B = \text{#bytes in one I/O}$  
$M = \text{#bytes of main memory or cache}$  
$M \geq B^2$
Transposing a matrix

\[
\begin{align*}
\leq \sqrt{M/25} \text{ rows} & \quad \leq \frac{1}{B} \sqrt{M/25} + 2 \\
\text{blocks per row} & \quad < \frac{1}{5} \sqrt{M/25} + 2
\end{align*}
\]

\(O(n/M)\) calls on submatrices of size 
\(\geq M/100, \leq M/25\)

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

Algorithm \(T(\text{start}_i, \text{start}_j, \text{end}_i, \text{end}_j)\):

\[
\begin{align*}
& \text{if } \text{end}_j \leq \text{start}_i \text{ then return} \\
& \text{if } \text{start}_i = \text{end}_i \text{ then} \\
& \quad \text{swap}(A[\text{end}_i, \text{end}_j], A[\text{end}_j, \text{end}_i]) \\
& \text{else} \\
& \quad \text{midi} \leftarrow \left\lfloor (\text{start}_i + \text{end}_i)/2 \right\rfloor \\
& \quad \text{midj} \leftarrow \left\lfloor (\text{start}_j + \text{end}_j)/2 \right\rfloor \\
& \quad T(\text{start}_i, \text{start}_j, \text{midi}, \text{midj}) \\
& \quad T(\text{start}_i, \text{midj} + 1, \text{midi}, \text{end}_j) \\
& \quad T(\text{midi} + 1, \text{start}_j, \text{end}_i, \text{midj}) \\
& \quad T(\text{midi} + 1, \text{midj} + 1, \text{end}_i, \text{end}_j) \\
& \text{return}
\end{align*}
\]

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

\(B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2\)
Transposing a matrix

Algorithm $T(start_i, start_j, end_i, end_j)$:

If $end_j \leq start_i$ then return

If $start_i = end_i$ then

\[ \text{swap}(A[end_i, end_j], A[end_j, end_i]) \]

else

\[ midi \leftarrow \lfloor (start_i + end_i) / 2 \rfloor \]
\[ midj \leftarrow \lfloor (start_j + end_j) / 2 \rfloor \]

$T(start_i, start_j, midi, midj)$
$T(start_i, midj + 1, midi, end_j)$
$T(midi + 1, start_j, end_i, midj)$
$T(midi + 1, midj + 1, end_i, end_j)$

return

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

\[ \text{blocks per row} < \frac{1}{B} \sqrt{M/25} + 2 \]

$\leq \sqrt{M/25}$ rows

$\leq \frac{1}{5} \sqrt{M} \cdot \left( \frac{1}{B} \cdot \frac{1}{5} \sqrt{M} + 2 \right) \leq \frac{2}{5} (M/B) (\frac{1}{5} + 2B/\sqrt{M}) \leq M/B$

fits in memory!

$B = \text{#bytes in one I/O}$  \hspace{1cm}  $M = \text{#bytes of main memory or cache}$  \hspace{1cm}  $M \geq B^2$
Algorithm \( T(start_i, start_j, end_i, end_j) \):

- **if** \( end_j \leq start_i \) **then return**
- **if** \( start_i = end_i \) **then**
  - swap(\( A[end_i, end_j] \), \( A[end_j, end_i] \))
- **else**
  - \( midi \leftarrow \lfloor (start_i + end_i)/2 \rfloor \)
  - \( midj \leftarrow \lfloor (start_j + end_j)/2 \rfloor \)
  - \( T(start_i, start_j, midi, midj) \)
  - \( T(start_i, midj + 1, midi, end_j) \)
  - \( T(midi + 1, start_j, end_i, midj) \)
  - \( T(midi + 1, midj + 1, end_i, end_j) \)

**return**

\( O(n/M) \) calls on submatrices of size 
\( \geq M/100, \leq M/25 \)

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

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

\( B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2 \)
Our goals

<table>
<thead>
<tr>
<th>CPU operations</th>
<th>I/O operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>$\Theta(1/B)$ amortized</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n/B)$</td>
</tr>
<tr>
<td>$\Theta(\log n)$</td>
<td>at most $\Theta(\log_B \frac{n}{B})$, but preferably: $\Theta(\frac{1}{B} \log_{M/B} \frac{n}{B})$ amortized</td>
</tr>
<tr>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CPU operations</th>
<th>I/O operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(1))</td>
<td>(\Theta(1/B)) amortized</td>
</tr>
<tr>
<td>(\Theta(n))</td>
<td>(\Theta(n/B))</td>
</tr>
<tr>
<td>(\Theta(\log n))</td>
<td>at most (\Theta(\log_B \frac{n}{B})), but preferably: (\Theta(\frac{1}{B} \log_{M/B} \frac{n}{B})) amortized</td>
</tr>
<tr>
<td>(\Theta(n \log n))</td>
<td>(\Theta(\frac{n}{B} \log_{M/B} \frac{n}{B}))</td>
</tr>
</tbody>
</table>

Division by \(B\) is crucial!

\[ B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array \ A):

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

\[ B = \# \text{bytes in one I/O} \quad \quad M = \# \text{bytes of main memory or cache} \quad \quad M \geq B^2 \]
Changing the logarithm: merge sort

Algorithm $\text{MergeSort}(\text{array } A)$:

if $\text{length}(A) \leq 1$ then
    return (array is already sorted)
else
    divide $A$ into arrays $A_1$, $A_2$ of equal size
    $\text{MergeSort}(A_1)$; $\text{MergeSort}(A_2)$
    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

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

$M \geq B^2$
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array }A\text{):

\begin{itemize}
  \item [if] \text{length}(A) \leq 1 \text{ then}
  \begin{itemize}
    \item [return] (array is already sorted)
  \end{itemize}
  \item [else]
    \begin{itemize}
      \item divide }A\text{ into arrays }A_1, A_2\text{ of equal size}
      \item \textsc{MergeSort}(A_1); \textsc{MergeSort}(A_2)
      \item merge }A_1\text{ and }A_2\text{ into one sorted array }B
      \item replace }A\text{ by }B
    \end{itemize}
\end{itemize}

Number of I/O's:

\[ \Theta(\log_2 n) \text{ levels of recursion}; \]

\begin{align*}
B &= \#\text{bytes in one I/O} & M &= \#\text{bytes of } \text{main memory or cache} & M \geq B^2
\end{align*}
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 \texttt{MergeSort}(array \: A):  

if \: \text{length}(A) \leq \frac{M}{2} \text{ then}  
merge-sort \: A \text{ (loading \: A \: into \: memory \: once)}  
write result to disk  
else  
divide \: A \: into \: arrays \: A_1, \: A_2 \: of \: equal \: size \n\texttt{MergeSort}(A_1); \: \texttt{MergeSort}(A_2) \nmerge \: A_1 \: and \: A_2 \: into \: one \: sorted \: array \: B \nreplace \: A \: by \: B

Number of I/O's:  
$\Theta(\log_2 (n/M))$ \: levels \: of \: recursion;

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

Algorithm `MergeSort(array A)`:  

if $\text{length}(A) \leq M/2$ 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 ...

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

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array } A): 

\textbf{if} \ length(A) \leq M/2 \hspace{1em} \textbf{then} \\
\hspace{1em} merge-sort } A \hspace{1em} \text{(loading } A \text{ into memory once)} \\
\hspace{1em} write result to disk \\
\textbf{else} \\
\hspace{1em} divide } A \hspace{1em} \text{into arrays } A_1, A_2 \hspace{1em} \text{of equal size} \\
\hspace{1em} \textsc{MergeSort}(A_1); \textsc{MergeSort}(A_2) \\
\hspace{1em} merge } A_1 \hspace{1em} \text{and } A_2 \hspace{1em} \text{into one sorted array } B \\
\hspace{1em} replace } A \hspace{1em} \text{by } B \\

Number of I/O's: \\
\Theta(\log_2(n/M)) \hspace{1em} \text{levels of recursion;} \\
merge takes ... \\

\begin{align*}
B &= \# \text{bytes in one I/O} \hspace{2em} M &= \# \text{bytes of main memory or cache} \hspace{2em} M \geq B^2
\end{align*}
Changing the logarithm: merge sort

Algorithm $\text{MERGE\textsc{Sort}}(\text{array } A)$:

\begin{itemize}
  \item if $\text{length}(A) \leq M/2$ then
    \begin{itemize}
      \item merge-sort $A$ (loading $A$ into memory once)
      \item write result to disk
    \end{itemize}
  \item else
    \begin{itemize}
      \item divide $A$ into arrays $A_1, A_2$ of equal size
      \item $\text{MERGE\textsc{Sort}}(A_1); \text{MERGE\textsc{Sort}}(A_2)$
      \item merge $A_1$ and $A_2$ into one sorted array $B$
      \item replace $A$ by $B$
    \end{itemize}
\end{itemize}

Number of I/O's:

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

$B = \#\text{bytes in one I/O}$ \hspace{0.5cm} $M = \#\text{bytes of main memory or cache}$ \hspace{0.5cm} $M \geq B^2$
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array $A$):

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

Number of I/O's:

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

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

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array } A \text{):

\textbf{if} \ \text{length}(A) \leq M/2 \ \textbf{then}
\begin{itemize}
\item merge-sort } A \text{ (loading } A \text{ into memory once)}
\item write result to disk
\end{itemize}
\textbf{else}
\begin{itemize}
\item divide } A \text{ into arrays } A_1, A_2 \text{ of equal size}
\item \textsc{MergeSort}(A_1); \textsc{MergeSort}(A_2)
\item merge } A_1 \text{ and } A_2 \text{ into one sorted array } B
\item replace } A \text{ by } B
\end{itemize}

Number of I/O's:
\[ \Theta(\log_2(n/M)) \] levels of recursion; merge takes ...

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

Algorithm $\text{MergeSort}(\text{array } A)$:

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

Number of I/O's:
\[ \Theta(\log_2(n/M)) \text{ levels of recursion; merge takes ...} \]

$B = \#\text{bytes in one I/O}$ \quad $M = \#\text{bytes of main memory or cache}$ \quad $M \geq B^2$
Changing the logarithm: merge sort

Algorithm **MergeSort**(array $A$):

if $\text{length}(A) \leq \frac{M}{2}$ 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 ...

$B = \#\text{bytes in one I/O}$   $M = \#\text{bytes of main memory or cache}$  $M \geq B^2$
Changing the logarithm: merge sort

Algorithm \textsc{MergeSort}(array }A\text{):

\begin{align*}
\text{if } \text{length}(A) \leq M/2 & \text{ then } \quad \Theta(\log_2(n/M)) \text{ levels of recursion; } \\
\text{merge-sort } A \text{ (loading } A \text{ into memory once)} \quad \text{merge takes } \ldots \\
\text{write result to disk} \quad \text{Number of I/O's:}
\end{align*}

\text{else}

\begin{align*}
\text{divide } A \text{ into arrays } A_1, A_2 \text{ of equal size} \quad M \geq B^2 \\
\text{MergeSort}(A_1); \text{ MergeSort}(A_2) \\
\text{merge } A_1 \text{ and } A_2 \text{ into one sorted array } B \\
\text{replace } A \text{ by } B
\end{align*}

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \]
Changing the logarithm: merge sort

Algorithm $\text{MergeSort}(\text{array } A)$:

if $\text{length}(A) \leq M/2$ 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
  $\text{MergeSort}(A_1)$; $\text{MergeSort}(A_2)$
  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 ...

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
Changing the logarithm: merge sort

Algorithm \textbf{MergeSort}(array } A \textbf{):}

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

Number of I/O’s:
\[\Theta(\log_2(n/M))\) levels of recursion; merge takes ...

\[B = \#\text{bytes in one I/O}\quad M = \#\text{bytes of main memory or cache}\quad M \geq B^2\]
Changing the logarithm: merge sort

Algorithm MERGE_SORT(array $A$):

if length($A$) ≤ $M/2$ 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
MERGE_SORT($A_1$); MERGE_SORT($A_2$)
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 $\Theta\left(\frac{n}{B} \log_2 \frac{n}{M}\right)$ I/O’s

$B = \#\text{bytes in one I/O}$  \quad $M = \#\text{bytes of main memory or cache}$  \quad $M \geq B^2$
Changing the logarithm: merge sort

Algorithm \texttt{MERGE}\texttt{SORT}(array \textit{A}):

\textbf{if} \ \text{length}(\textit{A}) \leq 1 \ \textbf{then}
\begin{itemize}
  \item \textbf{return} \ (array is already sorted)
\end{itemize}
\textbf{else}
\begin{itemize}
  \item divide \textit{A} into arrays \textit{A}_1, \textit{A}_2 of equal size
  \item \texttt{MERGE}\texttt{SORT}(\textit{A}_1); \texttt{MERGE}\texttt{SORT}(\textit{A}_2)
  \item merge \textit{A}_1 and \textit{A}_2 into one sorted array \textit{B}
  \item replace \textit{A} by \textit{B}
\end{itemize}

\textbf{Number of I/O's:}

\[ \Theta(\log_2(n/M)) \] levels of recursion;
merge takes \( \Theta(n/B) \) per level:

\[ \text{total } \Theta\left(\frac{n}{B} \log_2 \frac{n}{M}\right) \] I/O's

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of } \text{main memory or cache} \quad M \geq B^2 \]

\textbf{analysis also applies to unmodified algorithm}
Changing the logarithm: merge sort

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

Algorithm \textsc{MergeSort}(array \( A \)):

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

Number of I/O's:

\( \Theta(\log_2(n/M)) \) levels of recursion;
merge takes \( \Theta(n/B) \) per level:

\[ \Theta\left(\frac{n}{B} \log_2 \frac{n}{M}\right) \] I/O's

\[ B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of } \text{main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

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

Algorithm \textsc{MergeSort}(array $A$):

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

Number of I/O's:

$\Theta(\log_k (n/M))$ levels of recursion;
merge takes $\Theta(n/B)$ per level:
total $\Theta\left(\frac{n}{B} \log_k \left(\frac{n}{M}\right)\right)$ I/O's

\[ B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2 \]
Changing the logarithm: merge sort

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

Algorithm **MergeSort**(array $A$):

if length($A$) ≤ $M/2$ 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 $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$

$B = \#\text{bytes in one I/O}$  $M = \#\text{bytes of main memory or cache}$  $M \geq B^2$
Is it always this simple?

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

$$B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2$$
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$ $M = \#\text{bytes of main memory or cache}$ $M \geq B^2$
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$  
$M = \#\text{bytes of main memory or cache}$  
$M \geq B^2$
Is it always this simple?

Try Dijkstra’s single-source shortest paths algorithm:
visiting vertices by increasing distance from source node \( s \)

\[
B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2
\]
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$

$M = \#\text{bytes of main memory or cache}$

$M \geq B^2$
Is it always this simple?

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

$B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2$
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
Is it always this simple?

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

\[ B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2 \]
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$

$M = \#\text{bytes of main memory or cache}$

$M \geq B^2$
Is it always this simple?

Try Dijkstra’s single-source shortest paths algorithm:
visiting vertices by increasing distance from source node \( s \)

\[
B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2
\]
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$

$M = \#\text{bytes of main memory or cache}$

$M \geq B^2$
Is it always this simple?

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

$B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2$
Is it always this simple?

Try Dijkstra’s single-source shortest paths algorithm:
visiting vertices by increasing distance from source node \( s \)

\[
B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2
\]
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
Is it always this simple?

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

$B = \#\text{bytes in one I/O}$  $M = \#\text{bytes of main memory or cache}$  $M \geq B^2$
Is it always this simple?

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

$B = \text{#bytes in one } I/O$  $M = \text{#bytes of main memory or cache}$  $M \geq B^2$
Is it always this simple?

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

Data access pattern seems to have no structure
(except for what’s known only after the computation...)

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
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.

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

\[ B = \# \text{bytes in one I/O} \quad M = \# \text{bytes of main memory or cache} \quad M \geq B^2 \]
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.

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

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

\[
B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2
\]
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.

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 $\text{Heapify}$

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

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

I/O-Efficient:

$\Theta(M/B)$-way mergesort, $\Theta(M/B)$-way quicksort:

$\Theta(n \log n)$ time, $\Theta(\frac{n}{B} \log \frac{M}{B} \frac{n}{B})$ 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.

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

\[
\begin{align*}
(3, a) & \quad (5, b) & \quad (11, c) & \quad (8, d) & \quad (2, e) & \quad (6, f) & \quad (1, g) & \quad (10, h) & \quad (7, i) & \quad (9, j) & \quad (4, k)
\end{align*}
\]
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.
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.

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

\[ B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2 \]
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.

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

\[
B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2
\]
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.

\[
\begin{array}{cccccccc}
(6, f) & (1, g) & (10, h) & (7, i) & (9, j) & (4, k) \\
(2, e) & (3, a) & (5, b) & (8, d) & & & (11, c) \\
\end{array}
\]

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

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

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

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

<table>
<thead>
<tr>
<th>1, g</th>
<th>2, e</th>
<th>3, a</th>
<th>5, b</th>
<th>6, f</th>
<th>7, i</th>
<th>8, d</th>
<th>9, j</th>
<th>4, k</th>
</tr>
</thead>
</table>

$B = \#\text{bytes in one I/O}$ \hspace{1cm} $M = \#\text{bytes of main memory or cache}$ \hspace{1cm} $M \geq B^2$
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.

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

\[ B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2 \]
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)

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

There are a number of tools and special techniques for designing I/O-efficient / external-memory / out-of-core / cache-oblivious algorithms and data structures, including:

- cache-oblivious search trees with queries in $\Theta(\log_B n)$ I/O’s
- priority queues with $\Theta\left(\frac{1}{B} \log_{M/B} \frac{n}{B}\right)$ amortized I/O’s per operation
- putting linked lists in order in $\Theta\left(\frac{n}{B} \log_{M/B} \frac{n}{B}\right)$ I/O’s
- BFS and SSSP on planar graphs in $\Theta\left(\frac{V}{B} \log_{M/B} \frac{V}{B}\right)$ I/O’s
- general techniques for I/O-efficient algorithms on trees and DAGs
- general techniques for I/O-efficient algorithms on planar graphs
- general techniques for I/O-efficient matrix operations
- general techniques for building I/O-efficient geometric data structures

$B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2$
Course organisation

1. first weeks: lectures by the instructor
   2. then: lectures by you (in pairs or triples);
      case studies with experiments → reports (in pairs or triples);
      small assignments (individual)
   3. then: case study reviews (individual) and discussion

Everything is mandatory (including attendance).
What you say in class counts towards your grade.
Unable to fulfil your obligations? Discuss it with me a.s.a.p.!

Schedule depends on number of students, so...
we’ll make it in the coming week (everybody reads mail every day?)
not taking the course? please unregister!

Some students do early lectures, late case studies; others do early case studies, late lectures;
late lectures/studies depend on early lectures/studies; every team gets one week on computer.

\[ B = \#\text{bytes in one I/O} \quad M = \#\text{bytes of main memory or cache} \quad M \geq B^2 \]
Course organisation

Prerequisites:

- if applicable, you must have completed 2IL05 (Data Structures) and 2IL15 (Algorithms);

- prerequisites test (see website) required, deadline Sunday 12 February

Results do not count for anything; they are used only to help you.

\[ B = \text{#bytes in one I/O} \quad M = \text{#bytes of main memory or cache} \quad M \geq B^2 \]