CSE 548 / AMS 542: Analysis of Algorithms

Lecture 13
(Analyzing I/O and Cache Performance)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook

Fall 2023

Iterative Matrix-Multiply Variants

double $Z[\,n\,][\,n\,],\ X[\,n\,][\,n\,],\ Y[\,n\,][\,n\,];$

I-J-K

```
for ( int i = 0; i < n; i++ )
  for ( int j = 0; j < n; j++ )
    for ( int k = 0; k < n; k++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

I-K-J

```
for ( int i = 0; i < n; i++ )
  for ( int k = 0; k < n; k++ )
    for ( int j = 0; j < n; j++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

J-I-K

```
for ( int j = 0; j < n; j++ )
  for ( int i = 0; i < n; i++ )
    for ( int k = 0; k < n; k++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

J-K-I

```
for ( int j = 0; j < n; j++ )
  for ( int k = 0; k < n; k++ )
    for ( int i = 0; i < n; i++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

K-I-J

```
for ( int k = 0; k < n; k++ )
  for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n; j++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

K-J-I

```
for ( int k = 0; k < n; k++ )
  for ( int j = 0; j < n; j++ )
    for ( int i = 0; i < n; i++ )
      Z[ i ][ j ] += X[ i ][ k ] * Y[ k ][ j ];
```

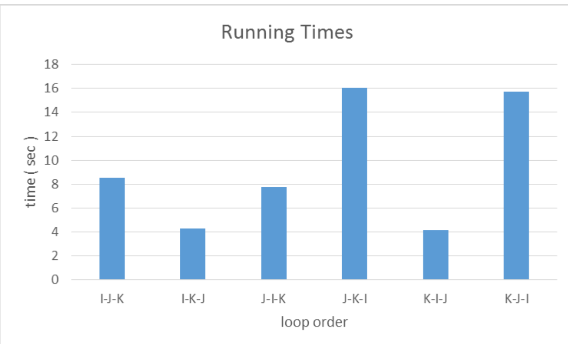
Performance of Iterative Matrix-Multiply Variants

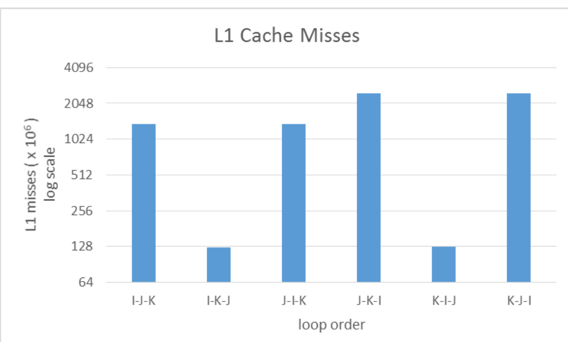
Processor: 2.7 GHz Intel Xeon E5-2680 (used only one core)

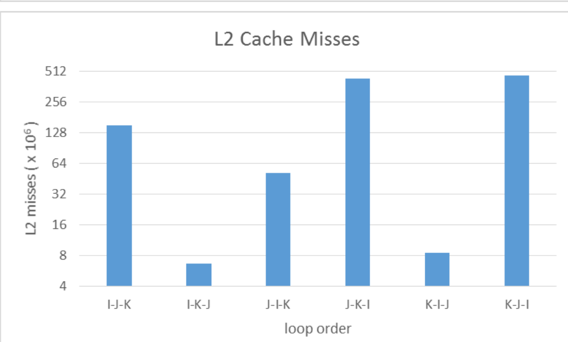
Caches & RAM: private 32KB L1, private 256KB L2, shared 20MB L3, 32 GB RAM

Optimizations: none (icc 13.0 with –O0)

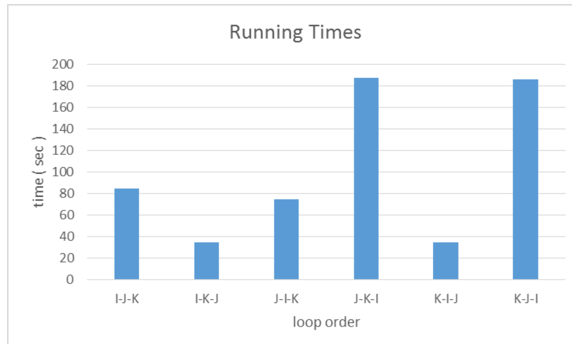
$n = 1000$

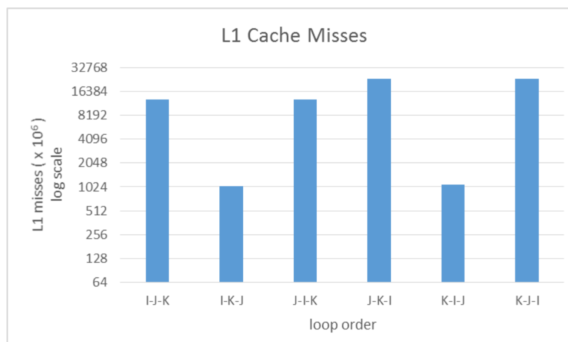


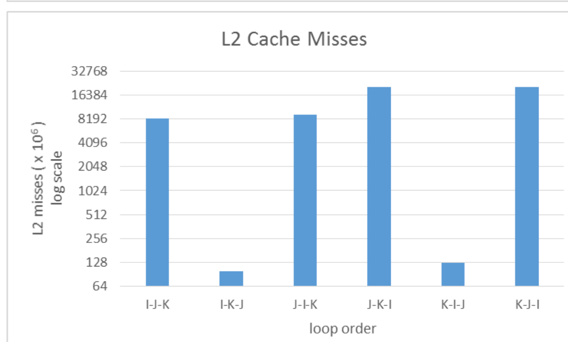




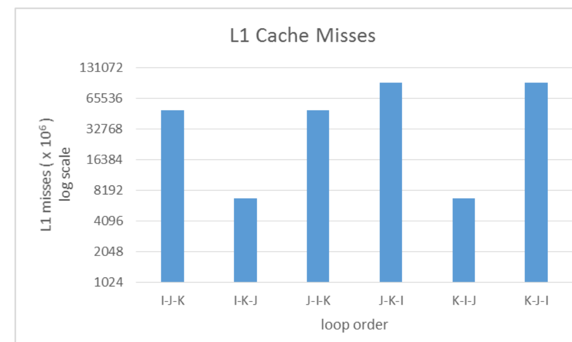
$n = 2000$

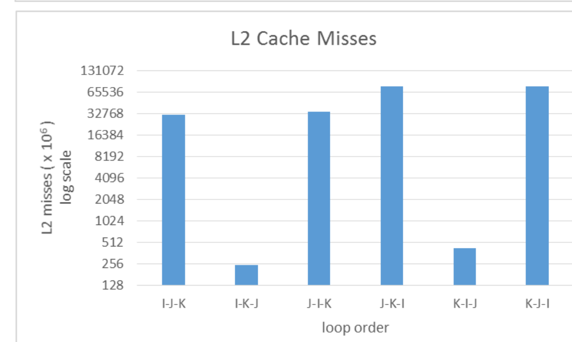






$n = 3000$





Memory: Fast, Large & Cheap!

For efficient computation we need

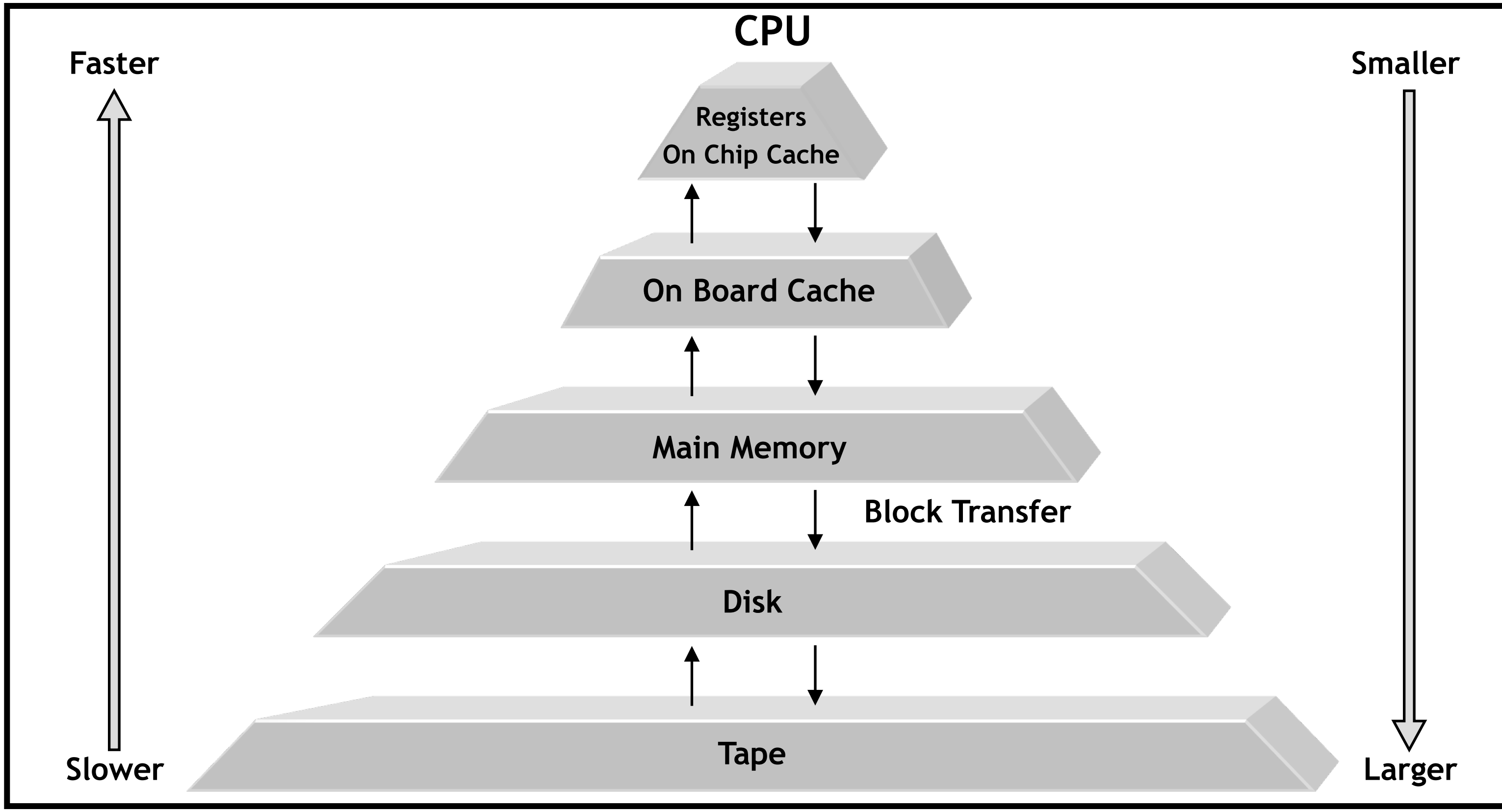
- fast processors
- fast and large (but not so expensive) memory

But memory cannot be cheap, large and fast at the same time, because of

- finite signal speed
- lack of space to put enough connecting wires
- capacitance of long connecting wires, etc.

A reasonable compromise is to use a *memory hierarchy*.

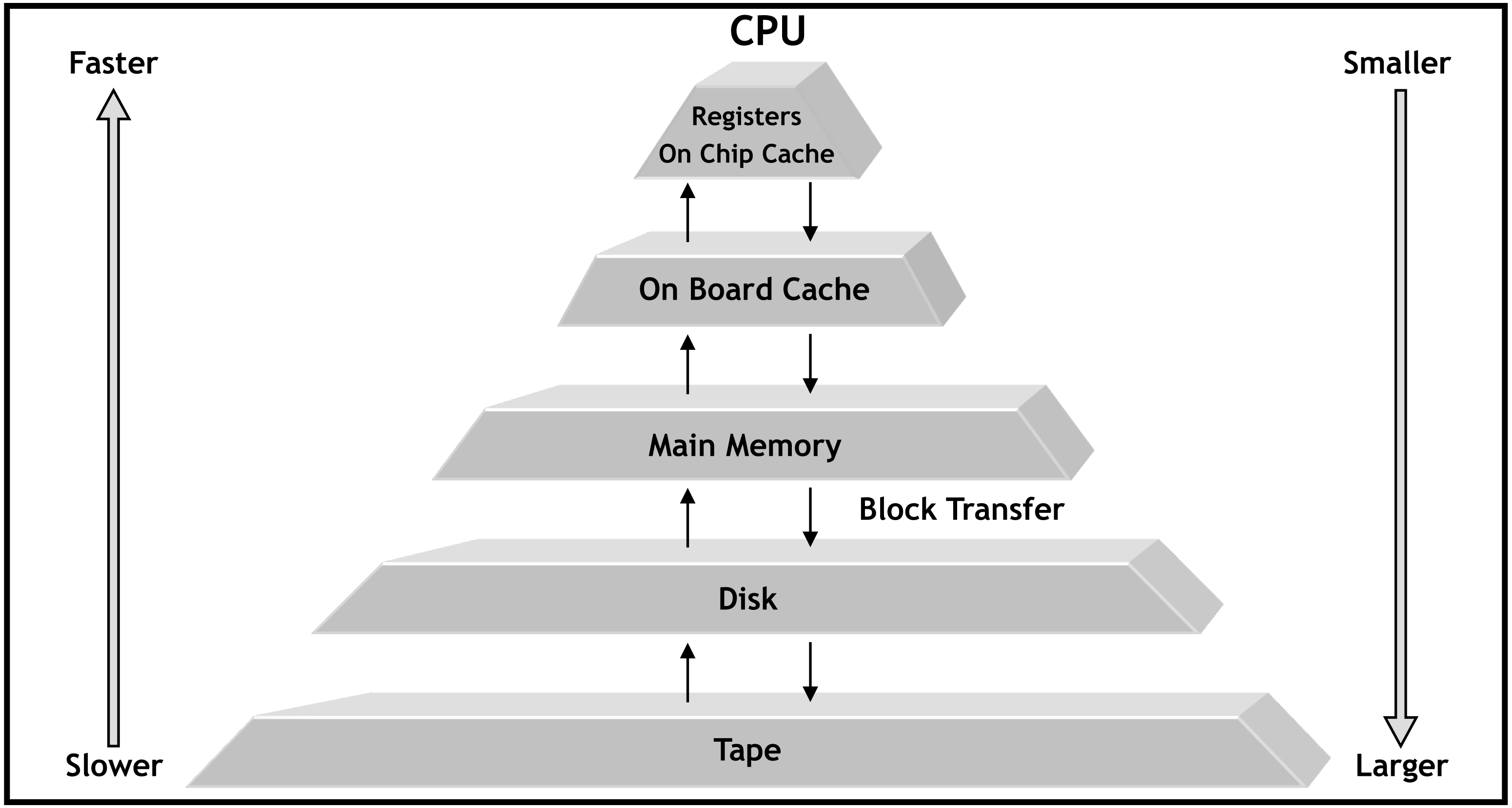
The Memory Hierarchy



A *memory hierarchy* is intended to be

- almost as fast as its fastest level
- almost as large as its largest level
- inexpensive

The Memory Hierarchy



To perform well on a memory hierarchy algorithms must have high locality in their memory access patterns.

Locality of Reference

Spatial Locality: When a block of data is brought into the cache it should contain as much useful data as possible.

Temporal Locality: Once a data point is in the cache as much useful work as possible should be done on it before evicting it from the cache.

CPU-bound vs. Memory-bound Algorithms

The Op-Space Ratio: Ratio of the number of operations performed by an algorithm to the amount of space it uses.

Intuitively, this gives an upper bound on the average number of operations performed for every memory location accessed.

CPU-bound Algorithm:

- high op-space ratio
- more time spent in computing than transferring data
- a faster CPU results in a lower running time

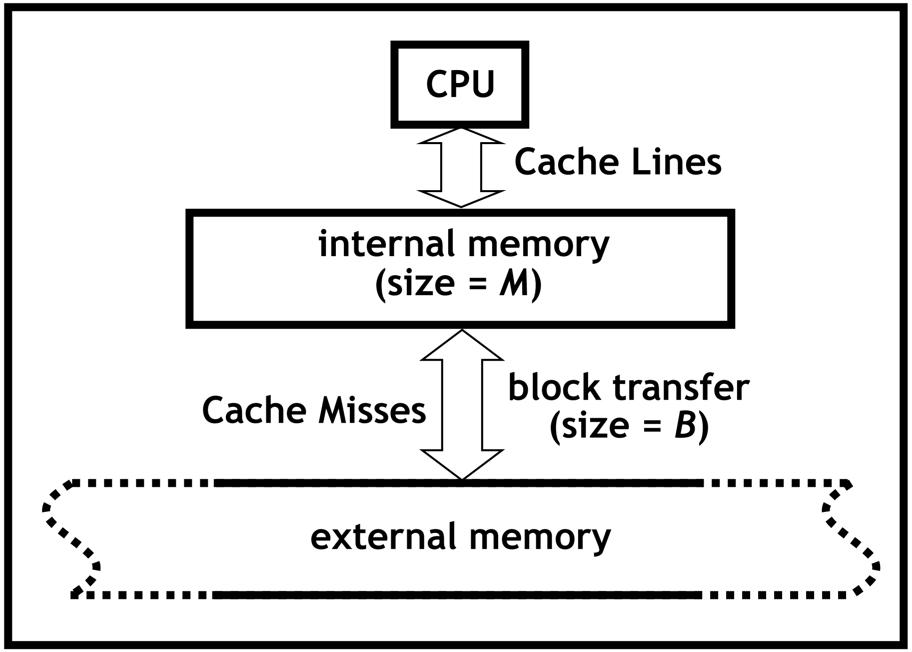
Memory-bound Algorithm:

- low op-space ratio
- more time spent in transferring data than computing
- a faster memory system leads to a lower running time

The Two-level I/O Model

The *two-level I/O model* [Aggarwal & Vitter, CACM'88] consists of:

- an *internal memory* of size *M*
- an arbitrarily large *external memory* partitioned into blocks of size *B*.



I/O complexity of an algorithm

= number of blocks transferred between these two levels

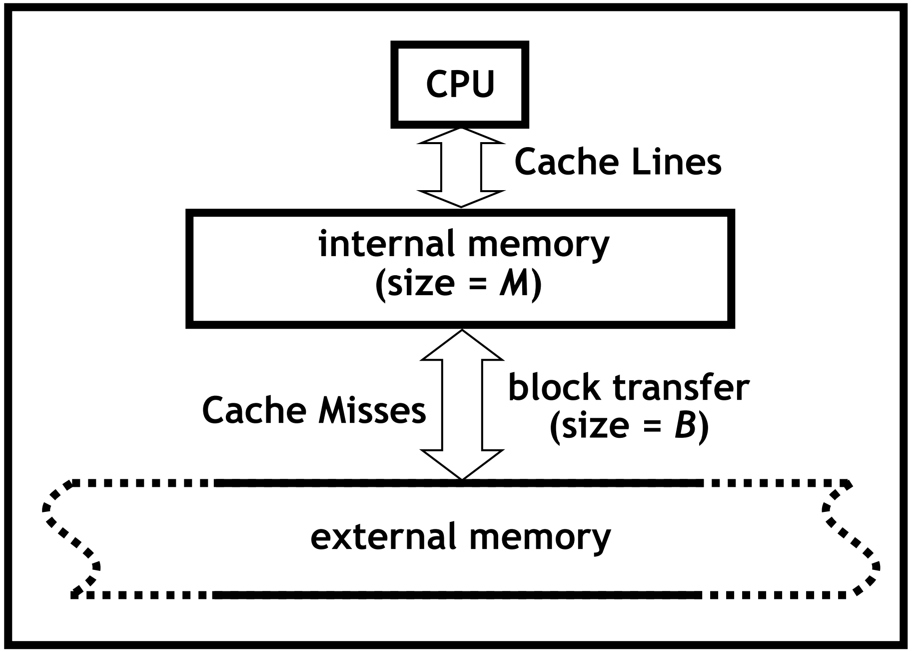
Basic I/O complexities: $scan(N) = \Theta\left(\frac{N}{B}\right)$ and $sort(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$

Algorithms often crucially depend on the knowledge of *M* and *B*

$\Rightarrow$ algorithms do not adapt well when *M* or *B* changes

The Ideal-Cache Model

The *ideal-cache model* [Frigo et al., FOCS'99] is an extension of the I/O model with the following constraint:



> algorithms are not allowed to use knowledge of M and B.

Consequences of this extension

- algorithms can simultaneously adapt to all levels of a multi-level memory hierarchy
- algorithms become more flexible and portable

Algorithms for this model are known as *cache-oblivious algorithms*.

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
  - LRU & FIFO allow for a constant factor approximation of optimal [Sleator & Tarjan, JACM'85]
- Exactly two levels of memory
- Automatic replacement & full associativity

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
  - can be effectively removed by making several reasonable assumptions about the memory hierarchy [Frigo et al., FOCS'99]
- Automatic replacement & full associativity

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- ❑ Optimal offline cache replacement policy
- ❑ Exactly two levels of memory
- ❑ Automatic replacement & full associativity
  - in practice, cache replacement is automatic (by OS or hardware)
  - fully associative LRU caches can be simulated in software with only a constant factor loss in expected performance [Frigo et al., FOCS’99]

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity

Often makes the following assumption, too:

- $M = \Omega\left(B^2\right)$, i.e., the cache is *tall*

The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity

Often makes the following assumption, too:

- $M = \Omega\left(B^2\right)$, i.e., the cache is *tall*
  - most practical caches are tall

The Ideal-Cache Model: I/O Bounds

Cache-oblivious vs. cache-aware bounds:

- Basic I/O bounds (same as the cache-aware bounds):
  - $scan(N) = \Theta\left(\frac{N}{B}\right)$
  - $sort(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$
- Most cache-oblivious results match the I/O bounds of their cache-aware counterparts
- There are few exceptions; e.g., no cache-oblivious solution to the *permutation* problem can match cache-aware I/O bounds [Brodal & Fagerberg, STOC'03]

Some Known Cache Aware / Oblivious Results

Problem	Cache-Aware Results	Cache-Oblivious Results
Array Scanning (*scan(N)*)	$O\left(\frac{N}{B}\right)$	$O\left(\frac{N}{B}\right)$
Sorting (*sort(N)*)	$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$	$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$
Selection	$O(scan(N))$	$O(scan(N))$
B-Trees [Am] (*Insert*, *Delete*)	$O\left(\log_B\frac{N}{B}\right)$	$O\left(\log_B\frac{N}{B}\right)$
Priority Queue [Am] (*Insert*, *Weak Delete*, *Delete-Min*)	$O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$	$O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$
Matrix Multiplication	$O\left(\frac{N^3}{B\sqrt{M}}\right)$	$O\left(\frac{N^3}{B\sqrt{M}}\right)$
Sequence Alignment	$O\left(\frac{N^2}{BM}\right)$	$O\left(\frac{N^2}{BM}\right)$
Single Source Shortest Paths	$O\left(\left(V+\frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$	$O\left(\left(V+\frac{E}{B}\right)\cdot\log_2\frac{V}{B}\right)$
Minimum Spanning Forest	$O(\min(sort(E)\log_2\log_2 V,\ V+sort(E)))$	$O\left(\min\left(sort(E)\log_2\log_2\frac{VB}{E},\ V+sort(E)\right)\right)$

Table 1: *N* = #elements, *V* = #vertices, *E* = #edges, Am = Amortized.

Matrix Multiplication

Iterative Matrix Multiplication

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Iter-MM (X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n *do*
2.   *for* $j \leftarrow 1$ *to* n *do*
3.   *for* $k \leftarrow 1$ *to* n *do*
4. $z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj}$

Iterative Matrix Multiplication

```
Iter-MM ( X, Y, Z, n )
1. for i ← 1 to n do
2.     for j ← 1 to n do
3.         for k ← 1 to n do
4.             z_ij ← z_ij + x_ik × y_kj
```

store in **row-major** order | store in **row-major** order

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Each iteration of the *for* loop in line 3 incurs $O(n)$ cache misses.

I/O-complexity of *Iter-MM*, $Q(n) = O(n^3)$

Iterative Matrix Multiplication

```
Iter-MM ( X, Y, Z, n )
1.  for i ← 1 to n do
2.      for j ← 1 to n do
3.          for k ← 1 to n do
4.              z_ij ← z_ij + x_ik × y_kj
```

store in **row-major** order

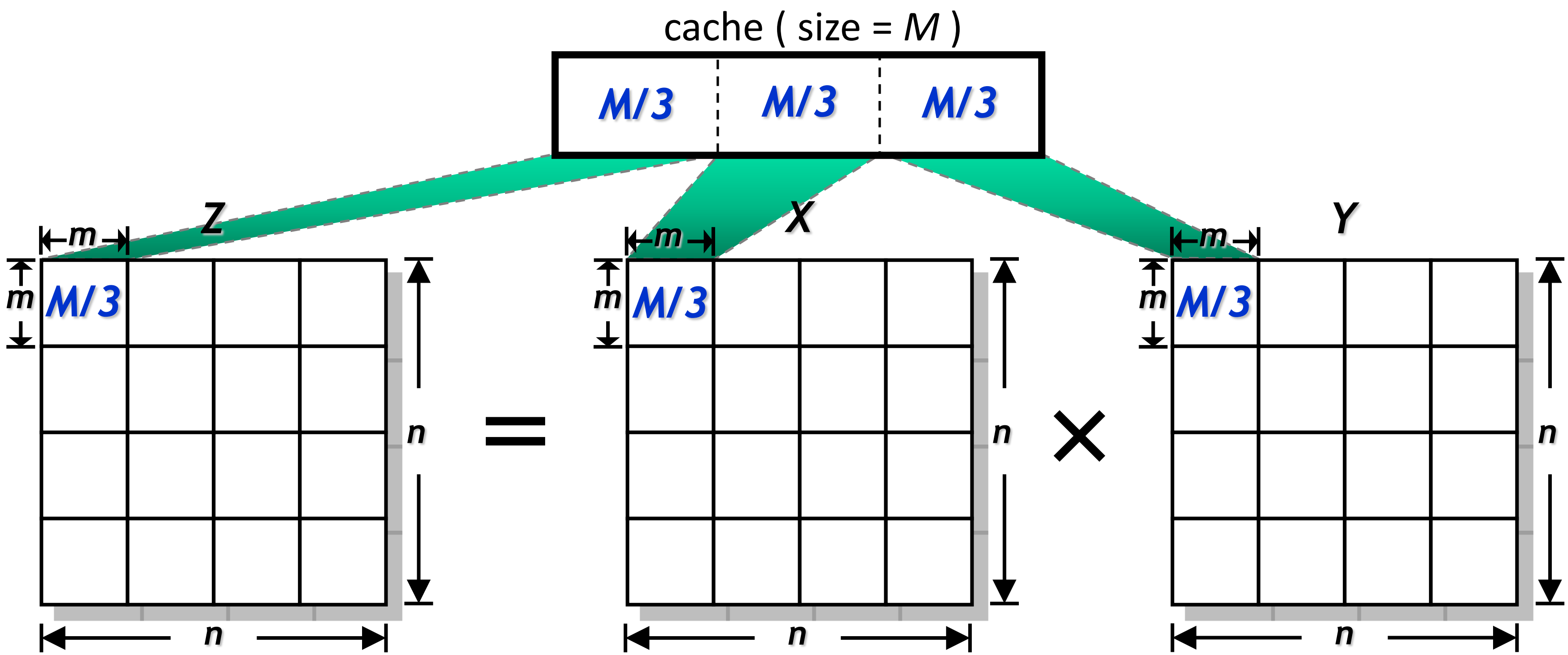
store in **column-major** order

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Each iteration of the *for* loop in line 3 incurs $O\left(1+\frac{n}{B}\right)$ cache misses.

I/O-complexity of *Iter-MM*, $Q(n) = O\left(n^2\left(1+\frac{n}{B}\right)\right) = O\left(\frac{n^3}{B}+n^2\right)$

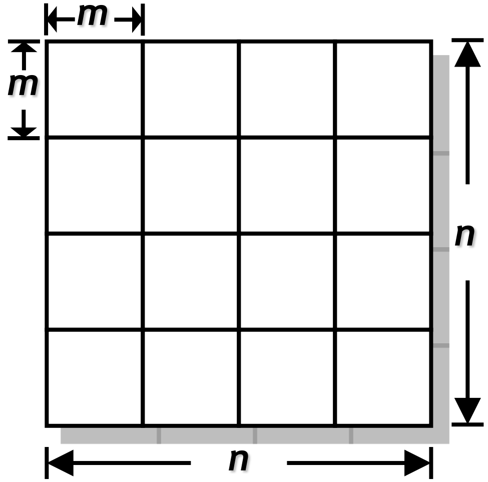
Block Matrix Multiplication



Block-MM (X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n / m *do*
2. *for* $j \leftarrow 1$ *to* n / m *do*
3. *for* $k \leftarrow 1$ *to* n / m *do*
4. *Iter-MM* (X_{ik}, Y_{kj}, Z_{ij})

Block Matrix Multiplication



Block-MM (X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n / m *do*
2. *for* $j \leftarrow 1$ *to* n / m *do*
3. *for* $k \leftarrow 1$ *to* n / m *do*
4. *Iter-MM* (X_{ik}, Y_{kj}, Z_{ij})

Choose $m = \sqrt{M/3}$, so that X_{ik}, Y_{kj} and Z_{ij} just fit into the cache.

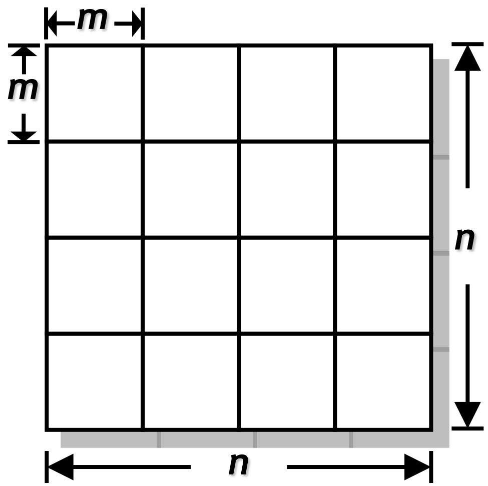
Then line 4 incurs $\Theta\left(m\left(1+\frac{m}{B}\right)\right)$ cache misses.

I/O-complexity of *Block-MM* [assuming a *tall cache*, i.e., $M = \Omega\left(B^2\right)$]

$$= \Theta\left(\left(\frac{n}{m}\right)^3\left(m+\frac{m^2}{B}\right)\right) = \Theta\left(\frac{n^3}{m^2}+\frac{n^3}{Bm}\right) = \Theta\left(\frac{n^3}{M}+\frac{n^3}{B\sqrt{M}}\right) = \Theta\left(\frac{n^3}{B\sqrt{M}}\right)$$

(Optimal: Hong & Kung, STOC'81)

Block Matrix Multiplication



Block-MM (X, Y, Z, n)

1. *for* i ← 1 *to* n / m *do*
2. *for* j ← 1 *to* n / m *do*
3. *for* k ← 1 *to* n / m *do*
4. *Iter-MM* (X_{ik}, Y_{kj}, Z_{ij})

Choose $m = \sqrt{M/3}$ so that X, Y and Z just fit into the cache.

The [illegible] ses.

I/O-[illegible] *cache*, i.e., $M = \Omega\left(B^2\right)$]

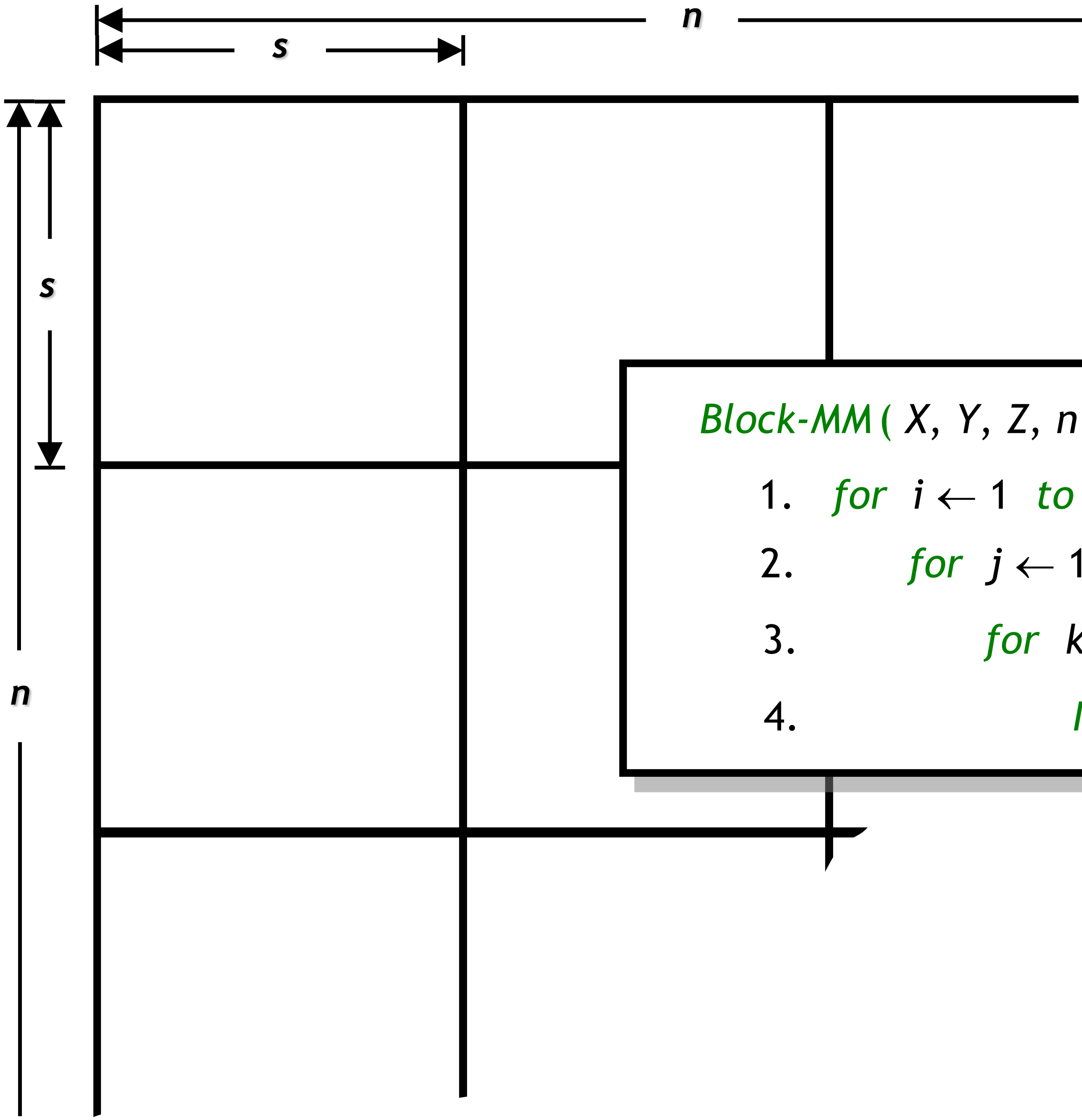
$= \Theta\left(\left(\frac{n}{m}\right)^3\left(m + \frac{m^2}{B}\right)\right) = \Theta\left(\frac{n^3}{m^2} + \frac{n^3}{Bm}\right) = \Theta\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \Theta\left(\frac{n^3}{B\sqrt{M}}\right)$

Optimal for any algorithm that performs the operations given by the following definition of matrix multiplication:

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

(Optimal: Hong & Kung, STOC'81)

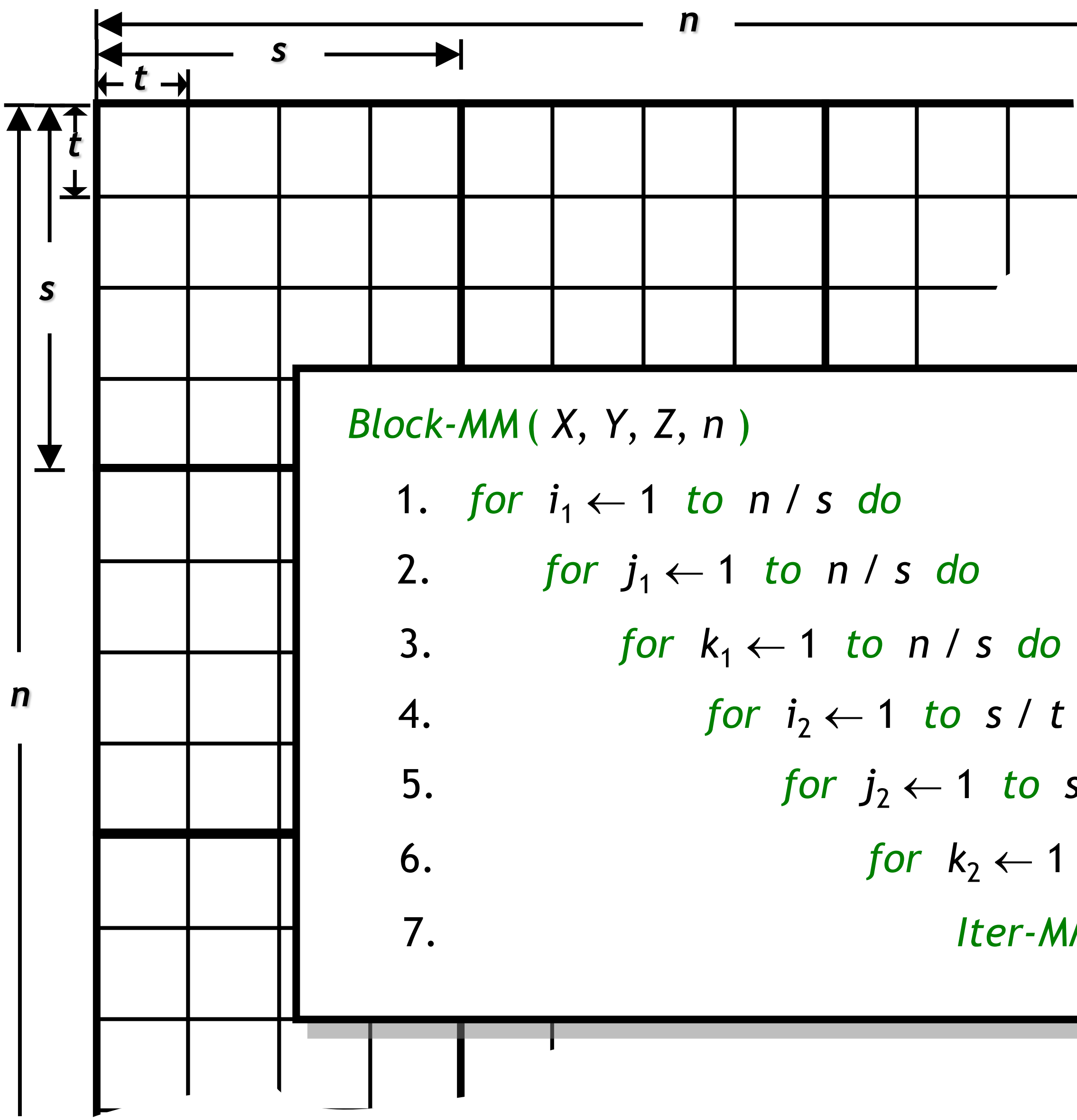
Multiple Levels of Cache



Block-MM (X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n / s *do*
2. *for* $j \leftarrow 1$ *to* n / s *do*
3. *for* $k \leftarrow 1$ *to* n / s *do*
4. *Iter-MM* ($X_{ik}, Y_{kj}, Z_{ij}, s$)

Multiple Levels of Cache



Block-MM (X, Y, Z, n)

$\quad$ **for** $i_1 \leftarrow 1$ **to** n / s **do**

$\qquad$ **for** $j_1 \leftarrow 1$ **to** n / s **do**

$\qquad\quad$ **for** $k_1 \leftarrow 1$ **to** n / s **do**

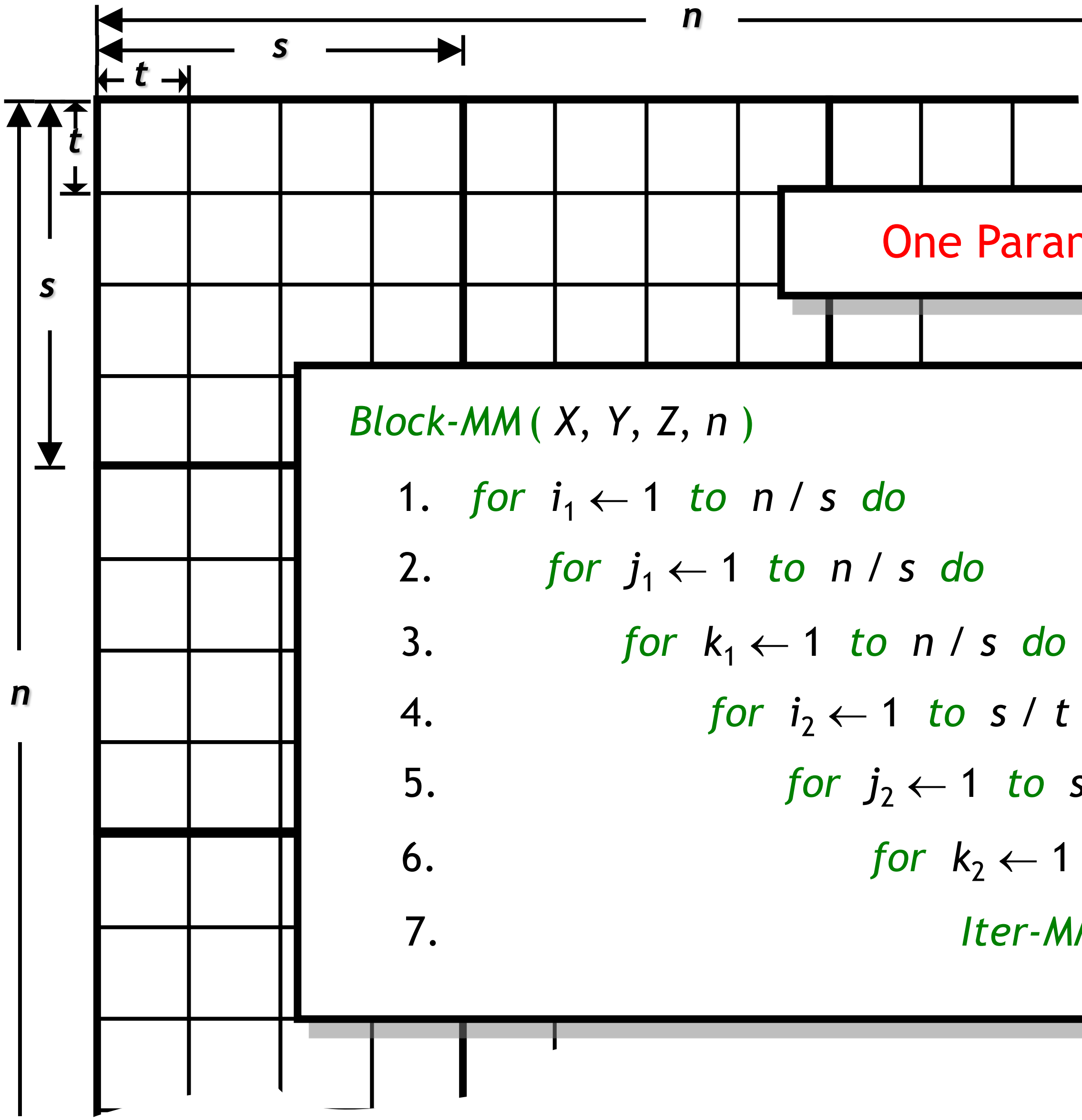
$\qquad\qquad$ **for** $i_2 \leftarrow 1$ **to** s / t **do**

$\qquad\qquad\quad$ **for** $j_2 \leftarrow 1$ **to** s / t **do**

$\qquad\qquad\qquad$ **for** $k_2 \leftarrow 1$ **to** s / t **do**

$\qquad\qquad\qquad\quad$ *Iter-MM* $((X_{i_1 k_1})_{i_2 k_2}, (Y_{k_1 j_1})_{k_2 j_2}, (X_{i_1 j_1})_{i_2 j_2}, t)$

Multiple Levels of Cache



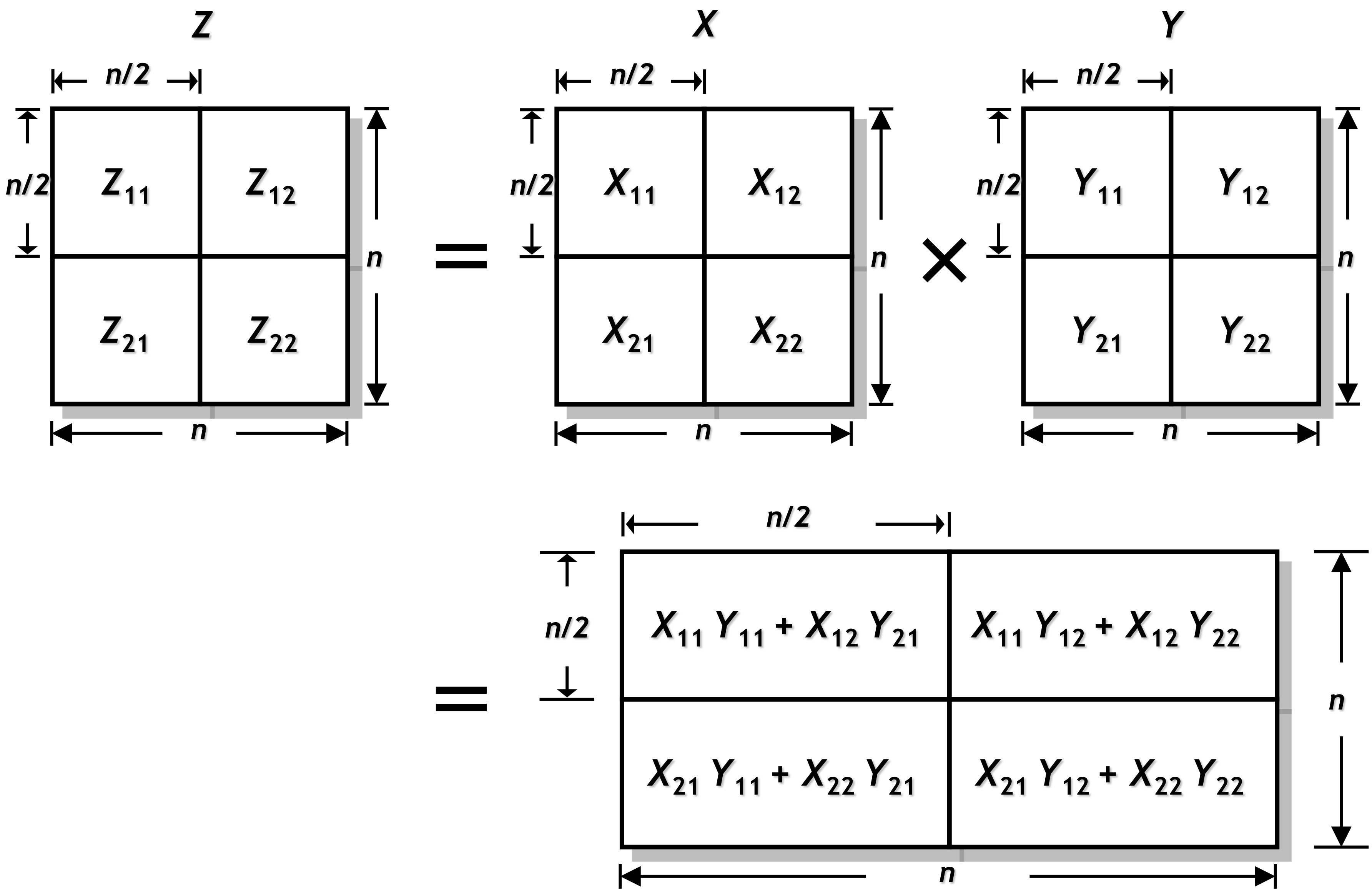
One Parameter Per Caching Level

Block-MM (X, Y, Z, n)

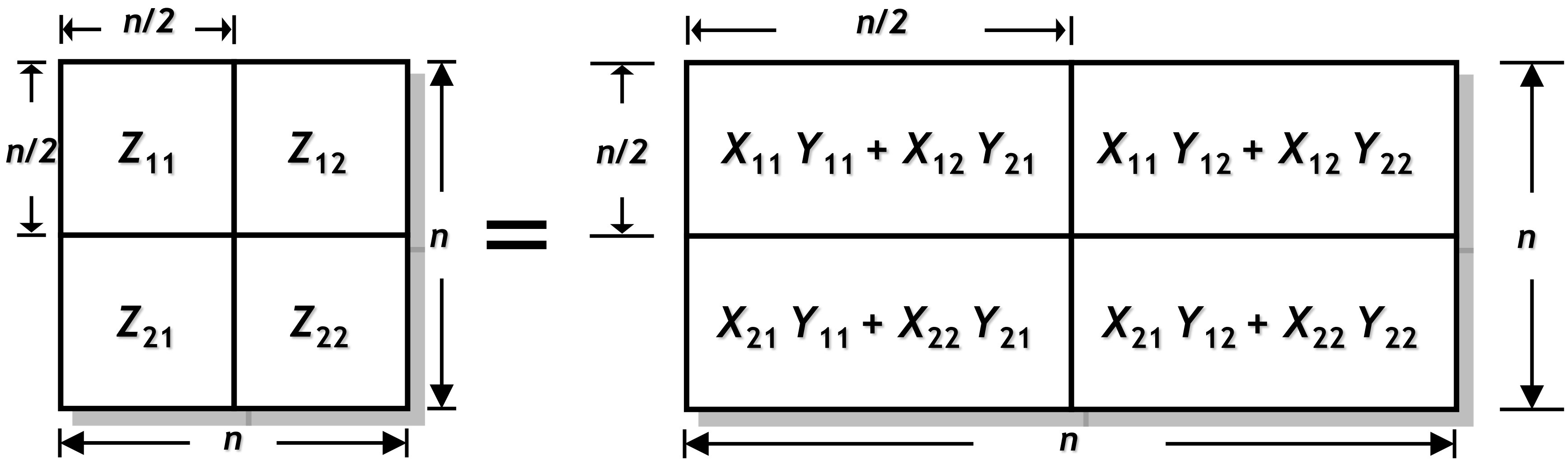
```
for i₁ ← 1 to n / s do
    for j₁ ← 1 to n / s do
        for k₁ ← 1 to n / s do
            for i₂ ← 1 to s / t do
                for j₂ ← 1 to s / t do
                    for k₂ ← 1 to s / t do
                        Iter-MM( (X_{i₁k₁})_{i₂k₂}, (Y_{k₁j₁})_{k₂j₂}, (X_{i₁j₁})_{i₂j₂}, t )
```

$$\text{Iter-MM}\left((X_{i_1k_1})_{i_2k_2},\ (Y_{k_1j_1})_{k_2j_2},\ (X_{i_1j_1})_{i_2j_2},\ t \right)$$

Recursive Matrix Multiplication



Recursive Matrix Multiplication



Rec-MM (Z, X, Y)

1. *if* $Z \equiv 1 \times 1$ matrix *then* $Z \leftarrow Z + X \cdot Y$
2. *else*
3. *Rec-MM* (Z_{11}, X_{11}, Y_{11}), *Rec-MM* (Z_{11}, X_{12}, Y_{21})
4. *Rec-MM* (Z_{12}, X_{12}, Y_{12}), *Rec-MM* (Z_{12}, X_{12}, Y_{22})
5. *Rec-MM* (Z_{21}, X_{21}, Y_{11}), *Rec-MM* (Z_{21}, X_{22}, Y_{21})
6. *Rec-MM* (Z_{22}, X_{21}, Y_{12}), *Rec-MM* (Z_{22}, X_{22}, Y_{22})

Recursive Matrix Multiplication

Rec-MM (Z, X, Y)

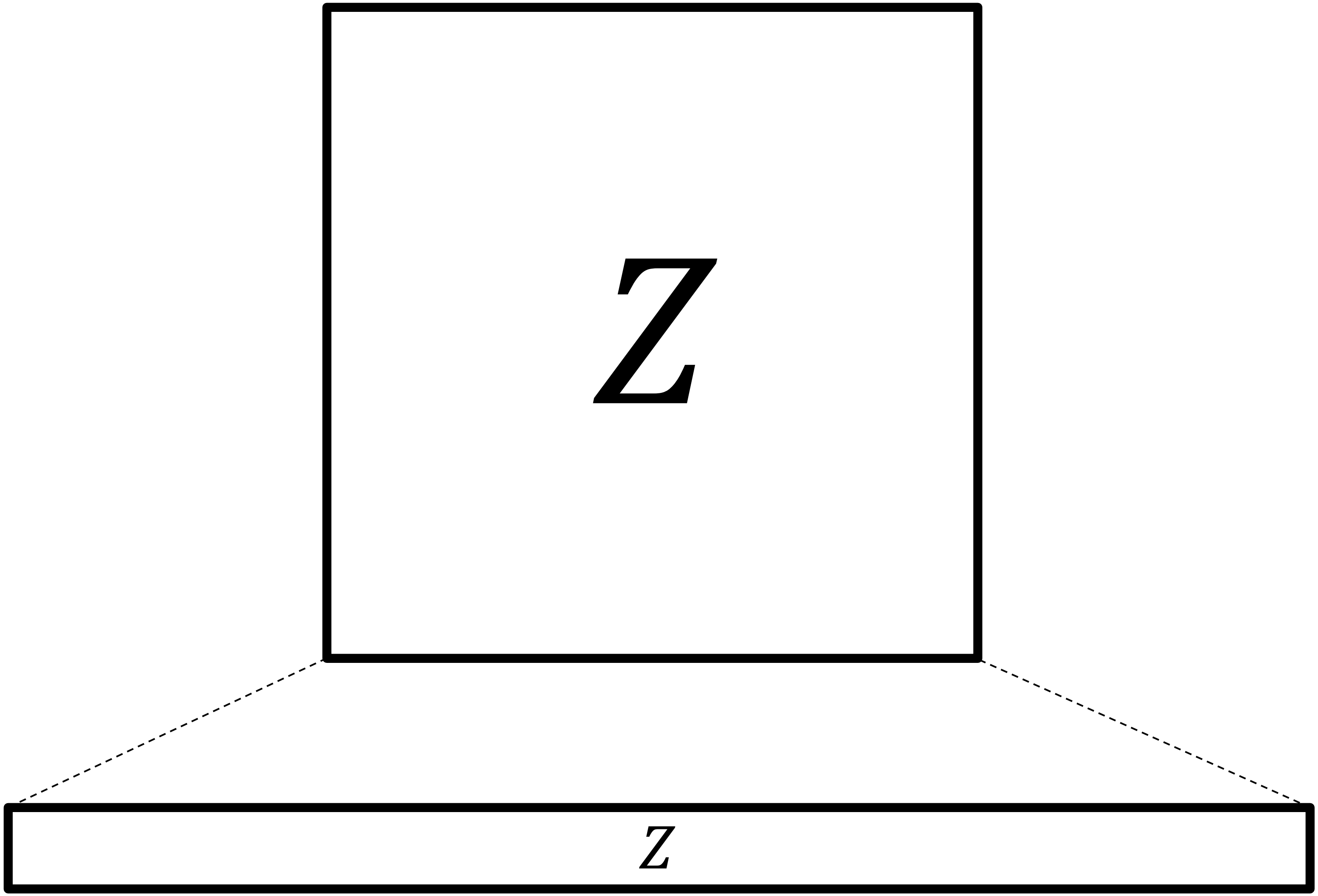
1. *if* $Z \equiv 1 \times 1$ matrix *then* $Z \leftarrow Z + X \cdot Y$
2. *else*
3. *Rec-MM* (Z_{11}, X_{11}, Y_{11}), *Rec-MM* (Z_{11}, X_{12}, Y_{21})
4. *Rec-MM* (Z_{12}, X_{12}, Y_{12}), *Rec-MM* (Z_{12}, X_{12}, Y_{22})
5. *Rec-MM* (Z_{21}, X_{21}, Y_{11}), *Rec-MM* (Z_{21}, X_{22}, Y_{21})
6. *Rec-MM* (Z_{22}, X_{21}, Y_{12}), *Rec-MM* (Z_{22}, X_{22}, Y_{22})

I/O-complexity, $Q(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right), & if\ n^2 \le \alpha M \\ 8Q\left(\frac{n}{2}\right) + O(1), & otherwise \end{cases}$

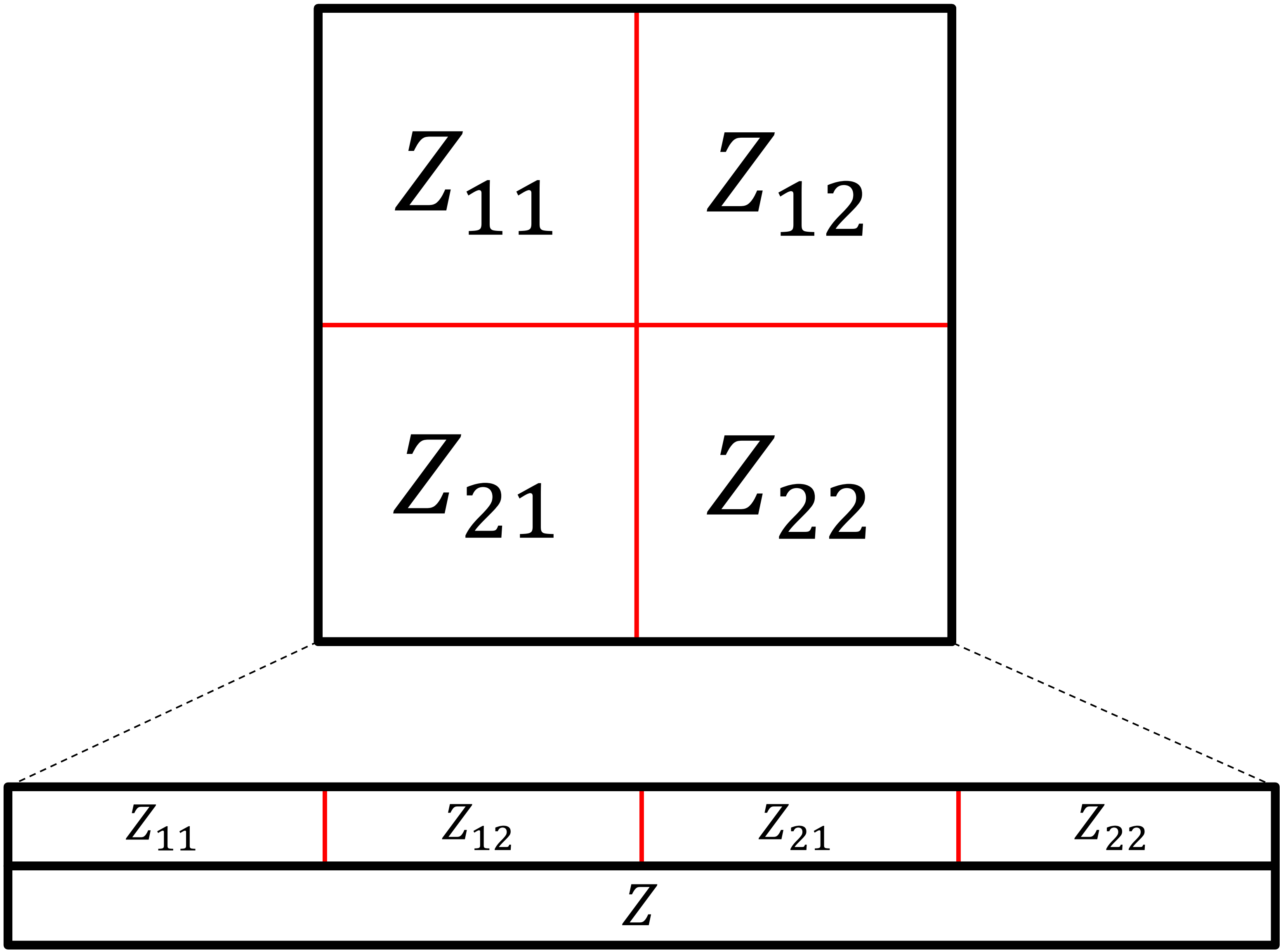
$$= O\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = O\left(\frac{n^3}{B\sqrt{M}}\right), when\ M = \Omega\left(B^2\right)$$

I/O-complexity (for all n) $= O\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1\right)$ (why?)

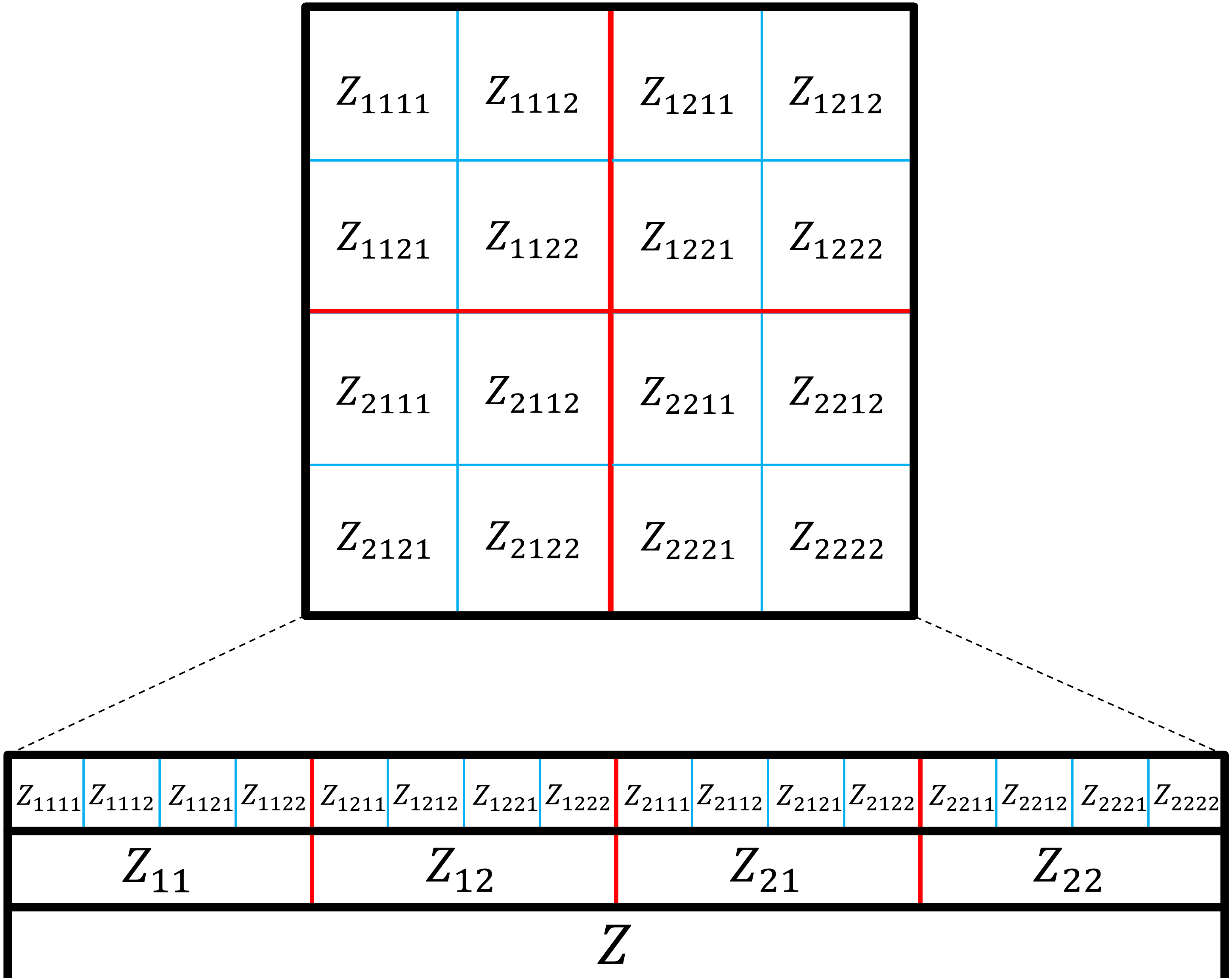
Recursive Matrix Multiplication with Z-Morton Layout



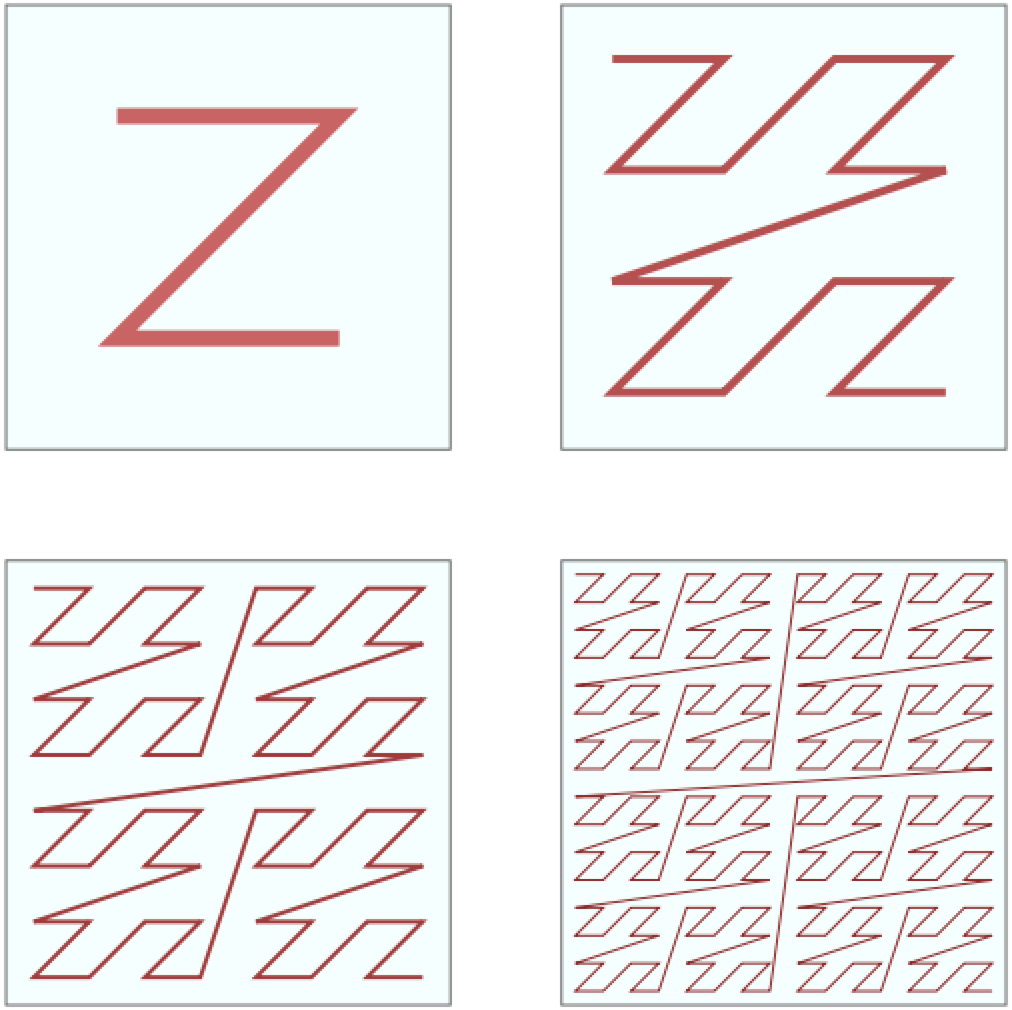
Recursive Matrix Multiplication with Z-Morton Layout



Recursive Matrix Multiplication with Z-Morton Layout



Recursive Matrix Multiplication with Z-Morton Layout



Source: wikipedia

Recursive Matrix Multiplication with Z-Morton Layout

Rec-MM (Z, X, Y)

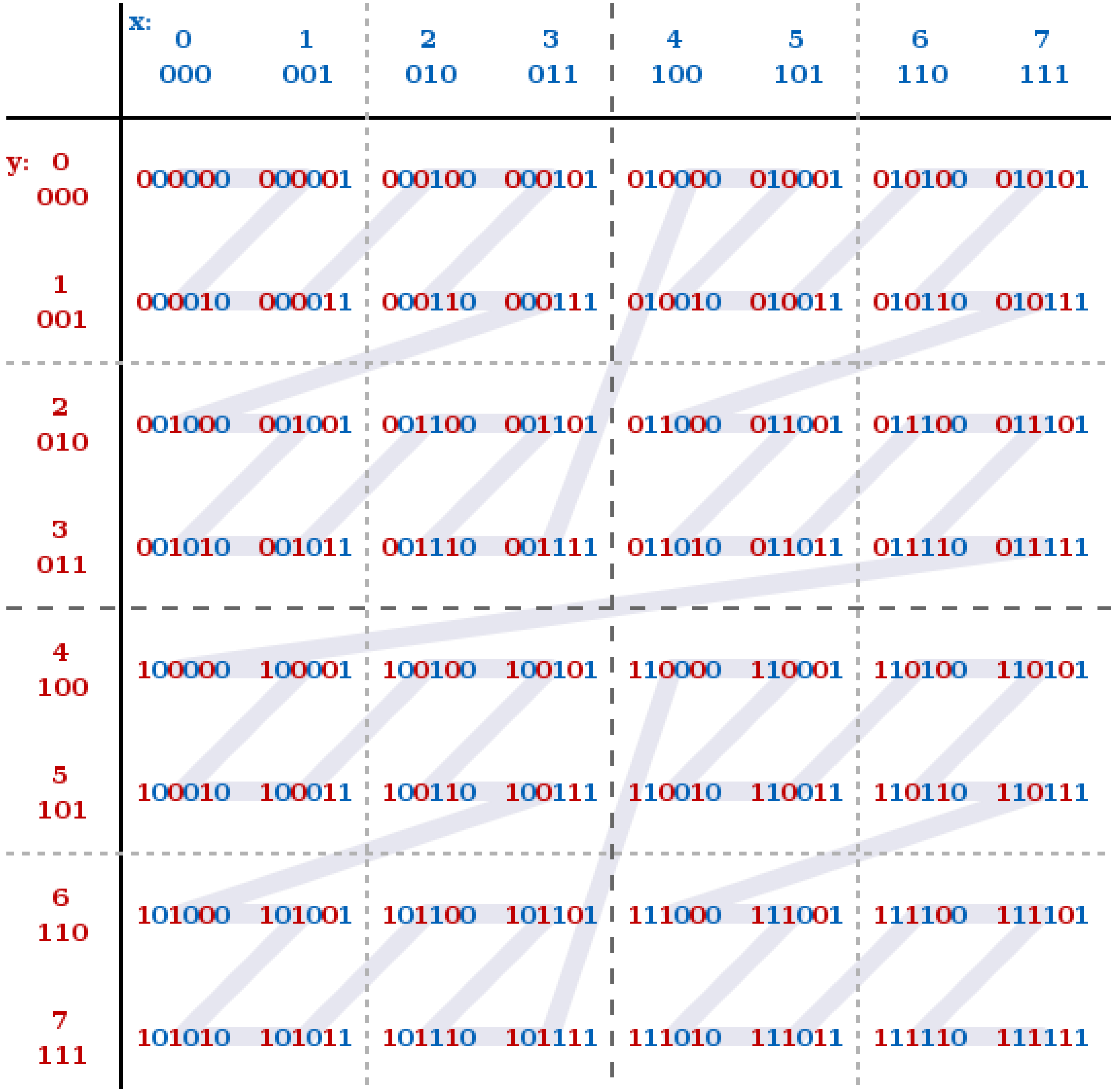
1. *if* $Z \equiv 1 \times 1$ matrix *then* $Z \leftarrow Z + X \cdot Y$
2. *else*
3. *Rec-MM* (Z_{11}, X_{11}, Y_{11}), *Rec-MM* (Z_{11}, X_{12}, Y_{21})
4. *Rec-MM* (Z_{12}, X_{12}, Y_{12}), *Rec-MM* (Z_{12}, X_{12}, Y_{22})
5. *Rec-MM* (Z_{21}, X_{21}, Y_{11}), *Rec-MM* (Z_{21}, X_{22}, Y_{21})
6. *Rec-MM* (Z_{22}, X_{21}, Y_{12}), *Rec-MM* (Z_{22}, X_{22}, Y_{22})

I/O-complexity, $Q(n) = \begin{cases} O\left(1 + \frac{n^2}{B}\right), & if\ n^2 \leq \alpha M \\ 8Q\left(\frac{n}{2}\right) + O(1), & otherwise \end{cases}$

$$= O\left(\frac{n^3}{M\sqrt{M}} + \frac{n^3}{B\sqrt{M}}\right) = O\left(\frac{n^3}{B\sqrt{M}}\right), when\ M = \Omega(B)$$

I/O-complexity (for all n) $= O\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1\right)$

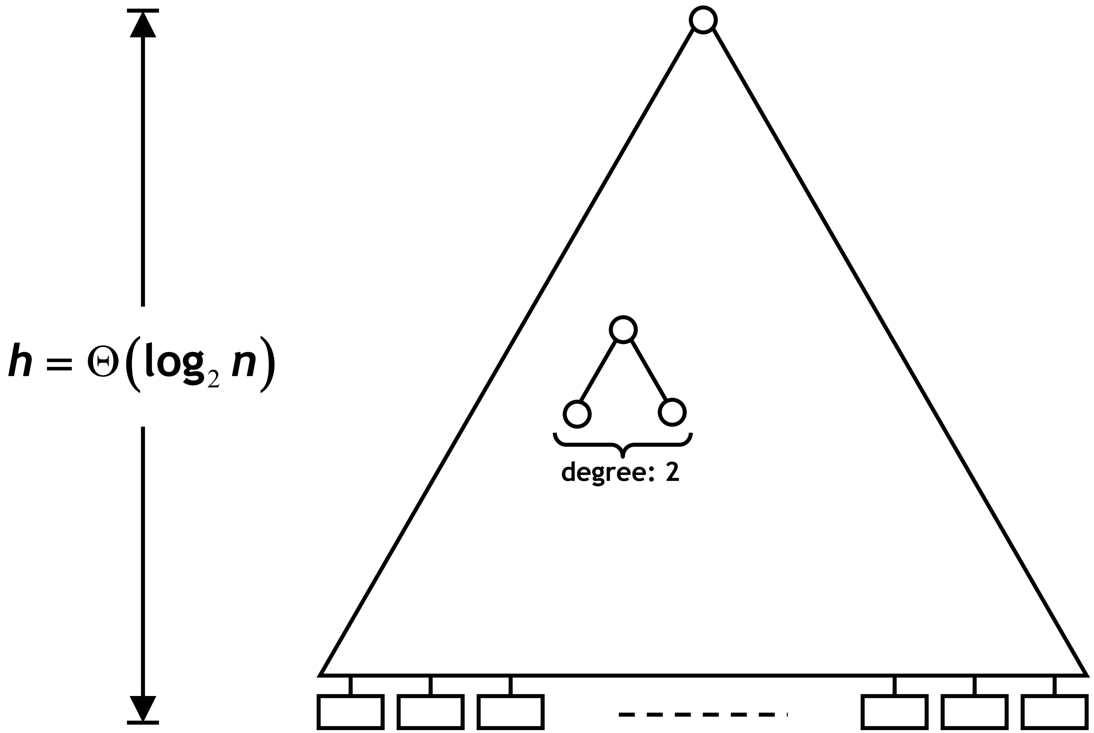
Recursive Matrix Multiplication with Z-Morton Layout



Source: wikipedia

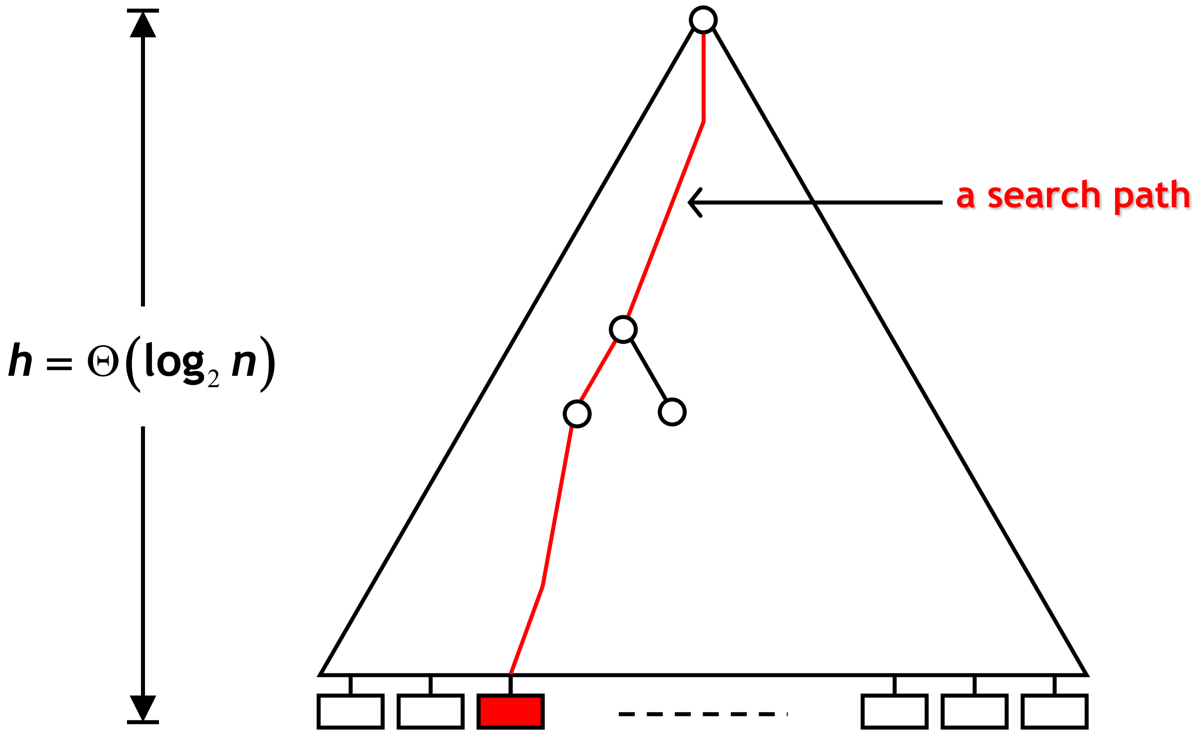
Searching
(Static B-Trees)

A Static Search Tree



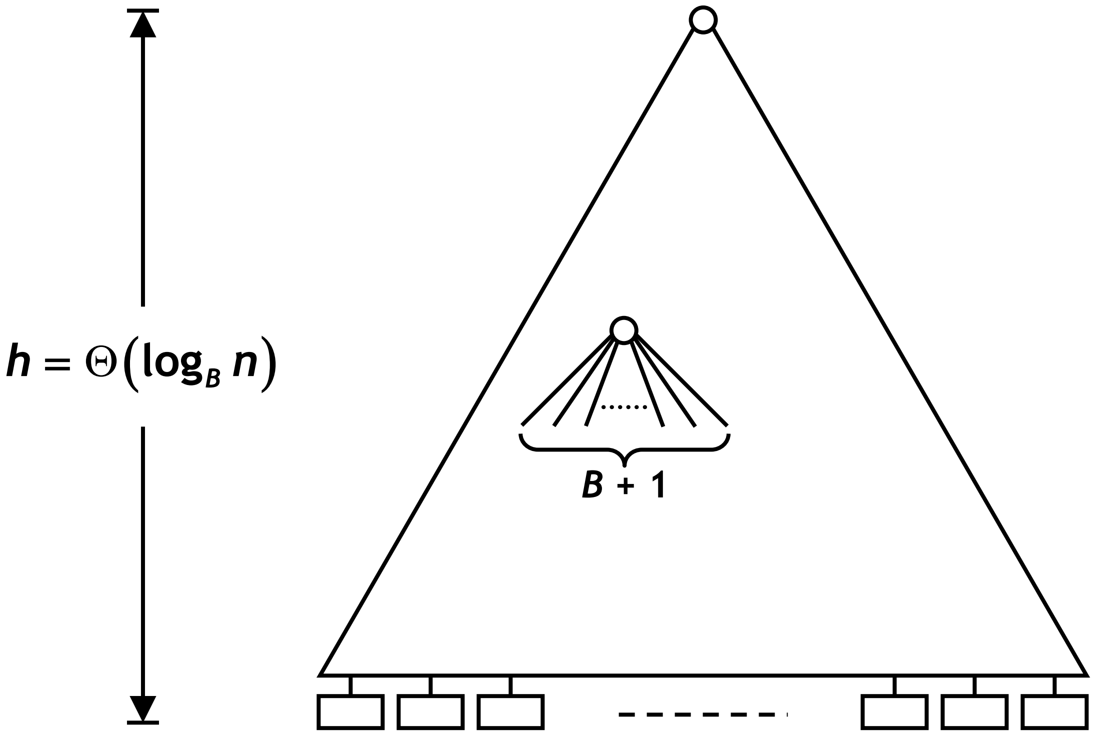
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$

A Static Search Tree



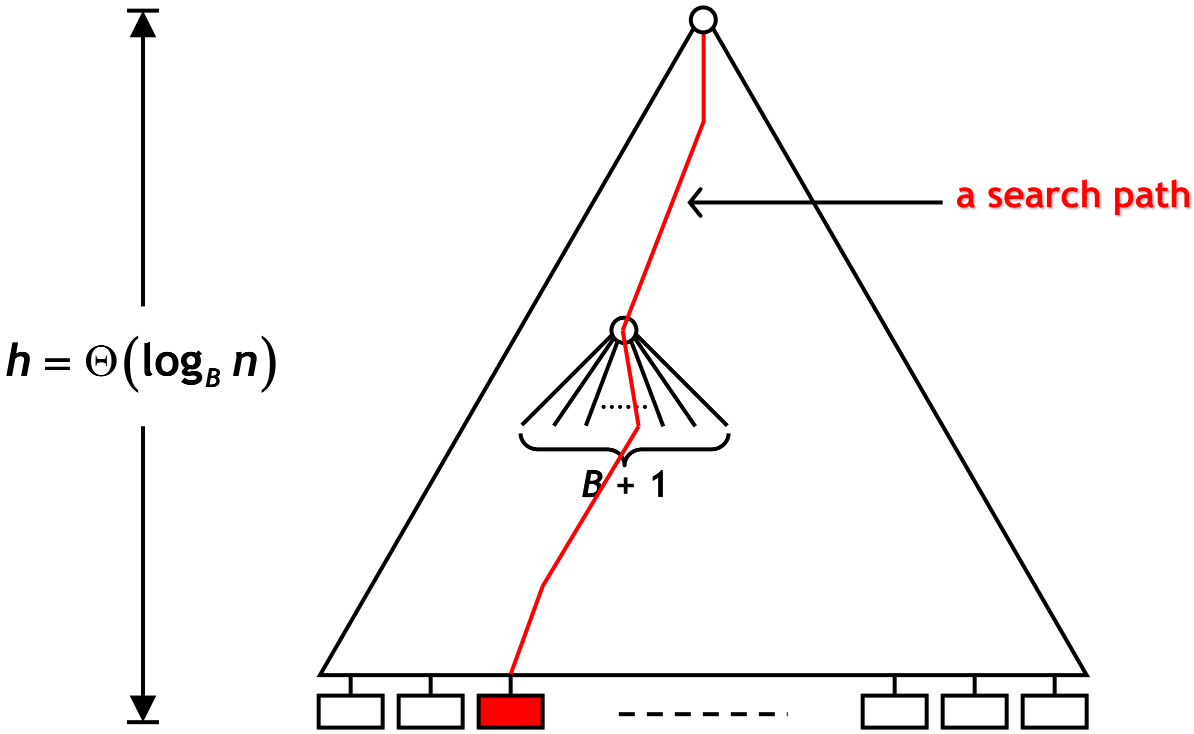
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$
- A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_2 n)$ I/Os

I/O-Efficient Static B-Trees



- Each node stores B keys, and has degree $B + 1$
- Height of the tree, $h = \Theta(\log_B n)$

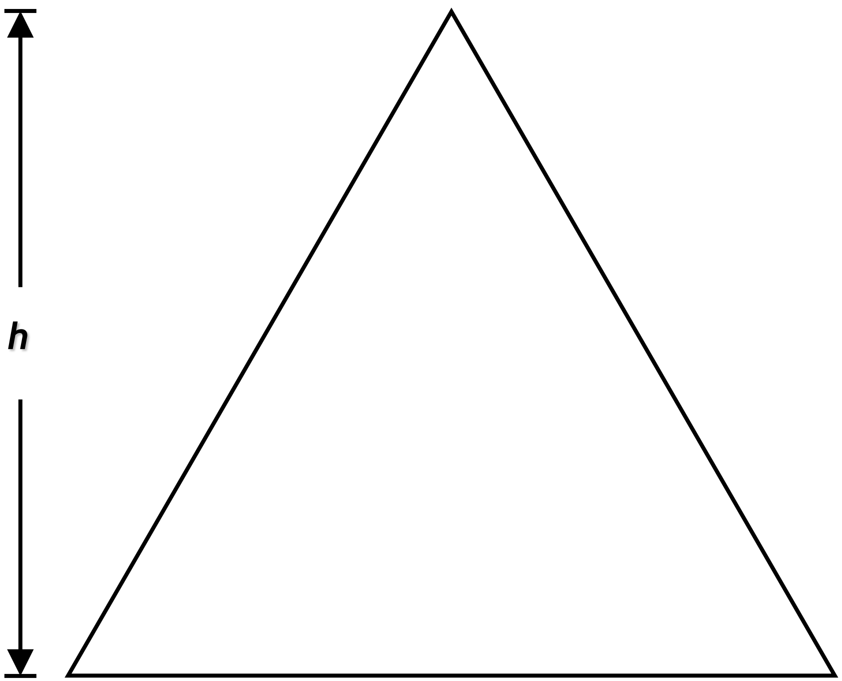
I/O-Efficient Static B-Trees



- Each node stores B keys, and has degree $B + 1$
- Height of the tree, $h = \Theta(\log_B n)$
- A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_B n)$ I/Os

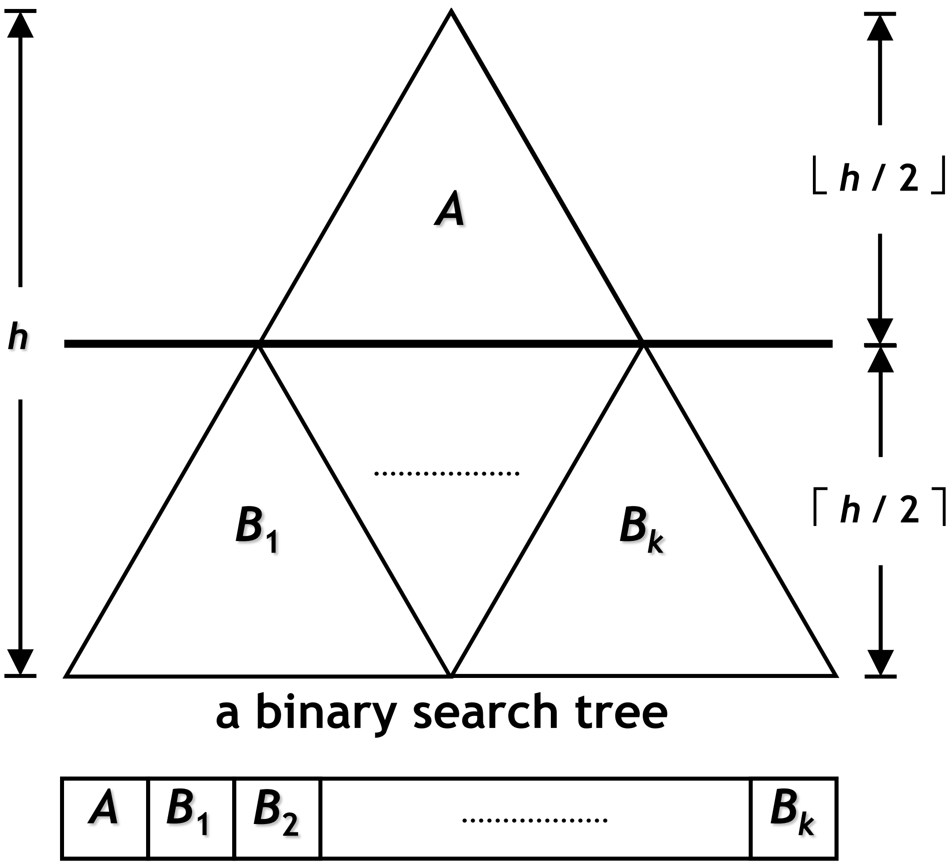
Cache-Oblivious Static B-Trees?

van Emde Boas Layout



a binary search tree

van Emde Boas Layout

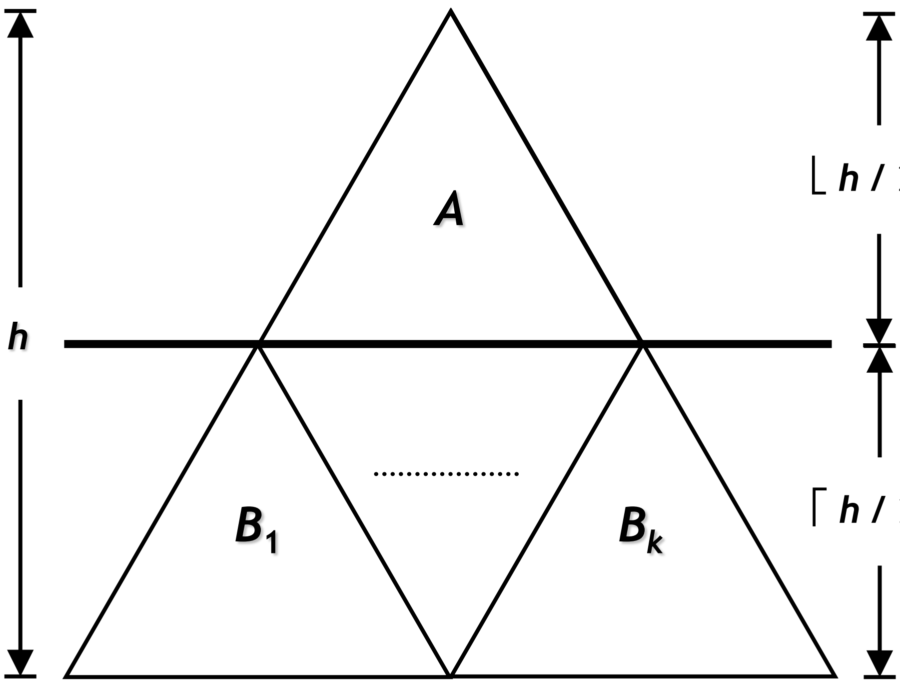


If the tree contains n nodes,

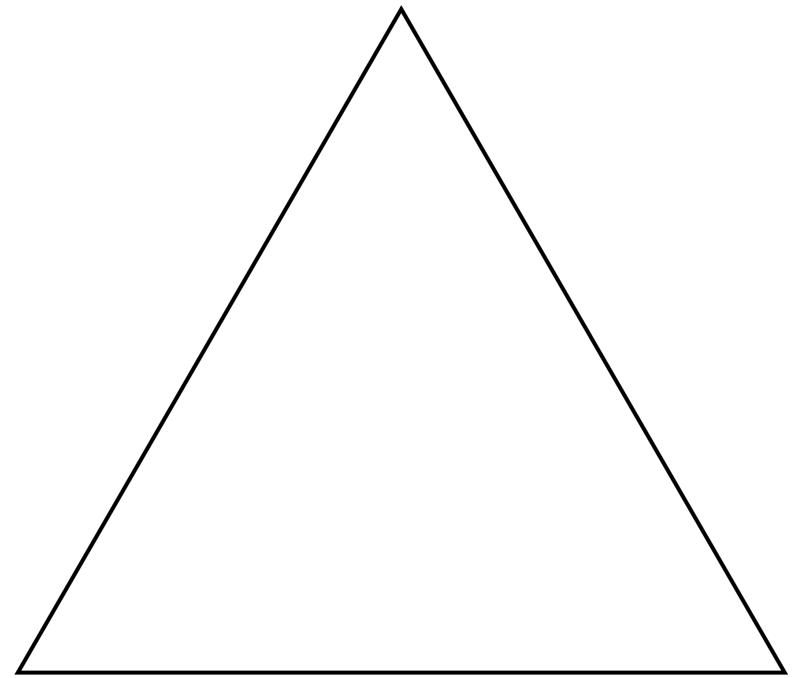
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

and $k = \Theta\left(\sqrt{n}\right)$.

van Emde Boas Layout



a binary search tree



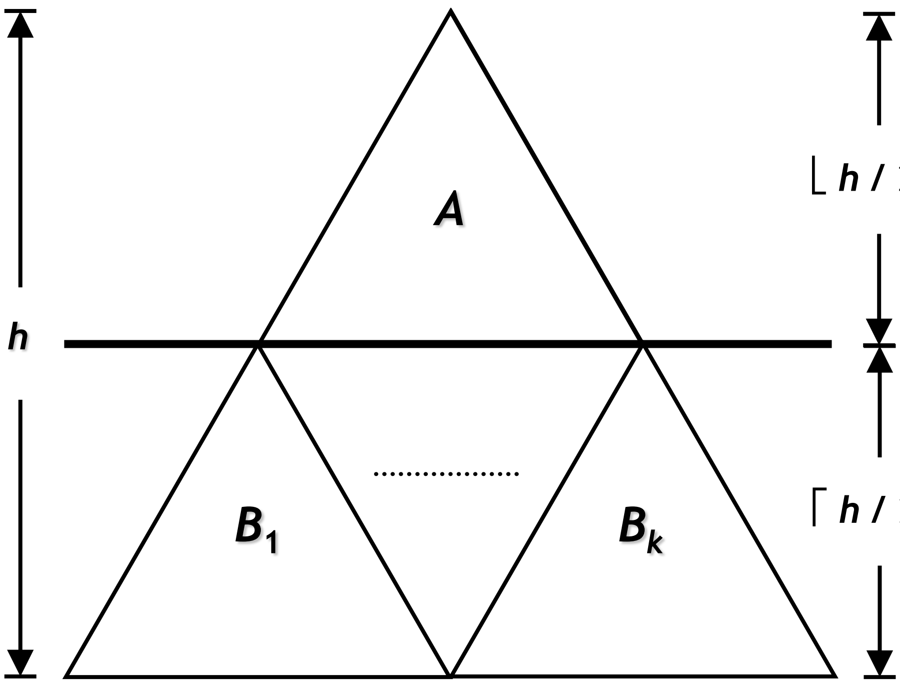
Recursive Subdivision

If the tree contains n nodes,

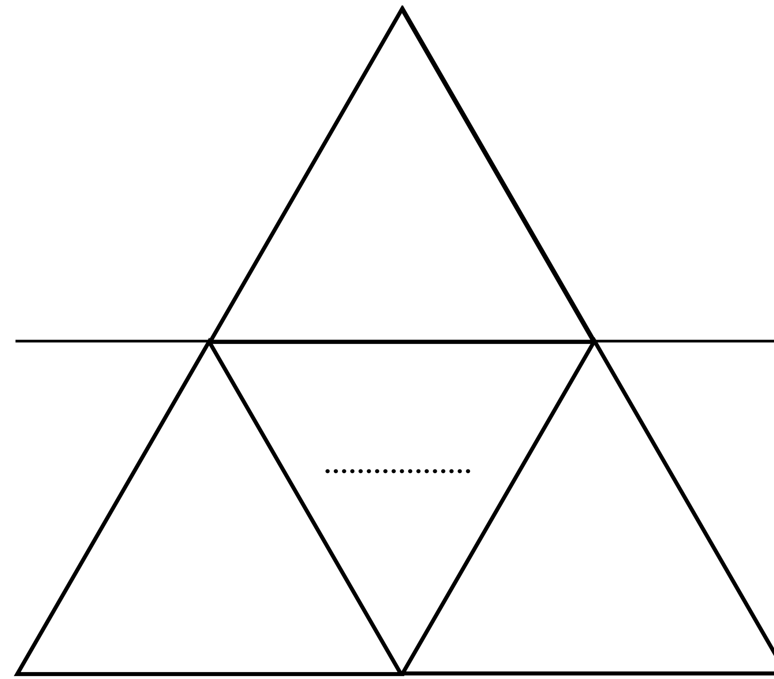
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

and $k = \Theta\left(\sqrt{n}\right)$.

van Emde Boas Layout



a binary search tree



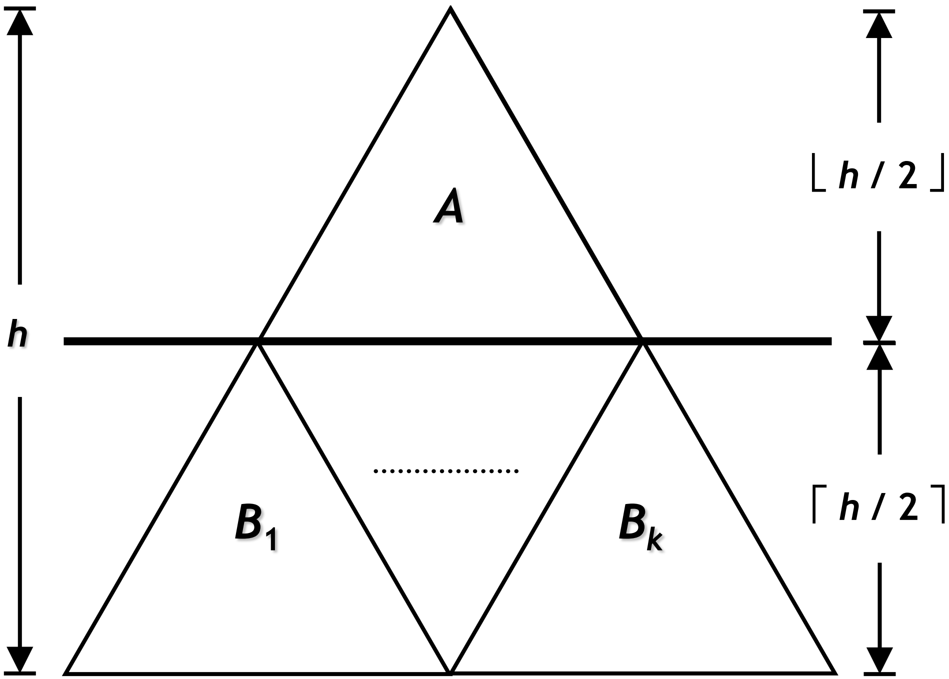
Recursive Subdivision

If the tree contains n nodes,

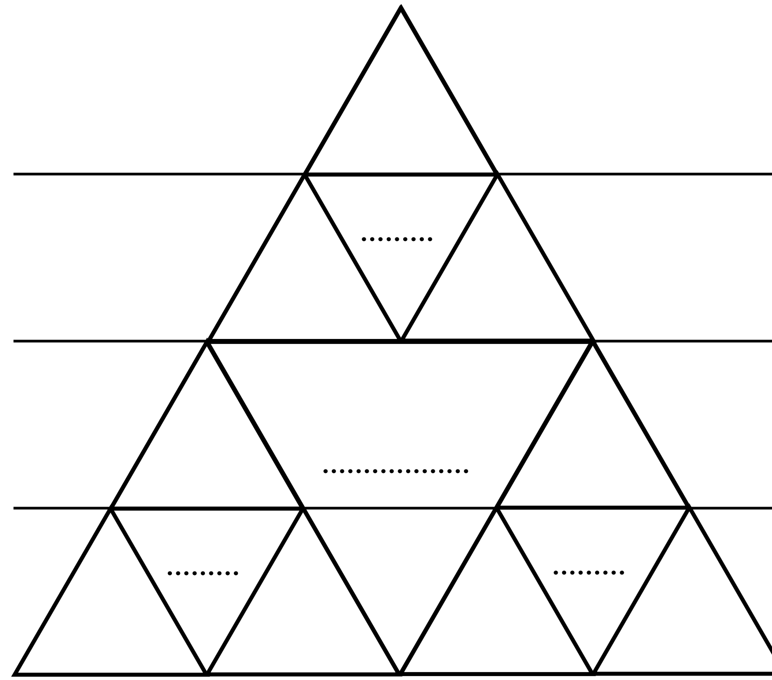
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

and $k = \Theta\left(\sqrt{n}\right)$.

van Emde Boas Layout



a binary search tree



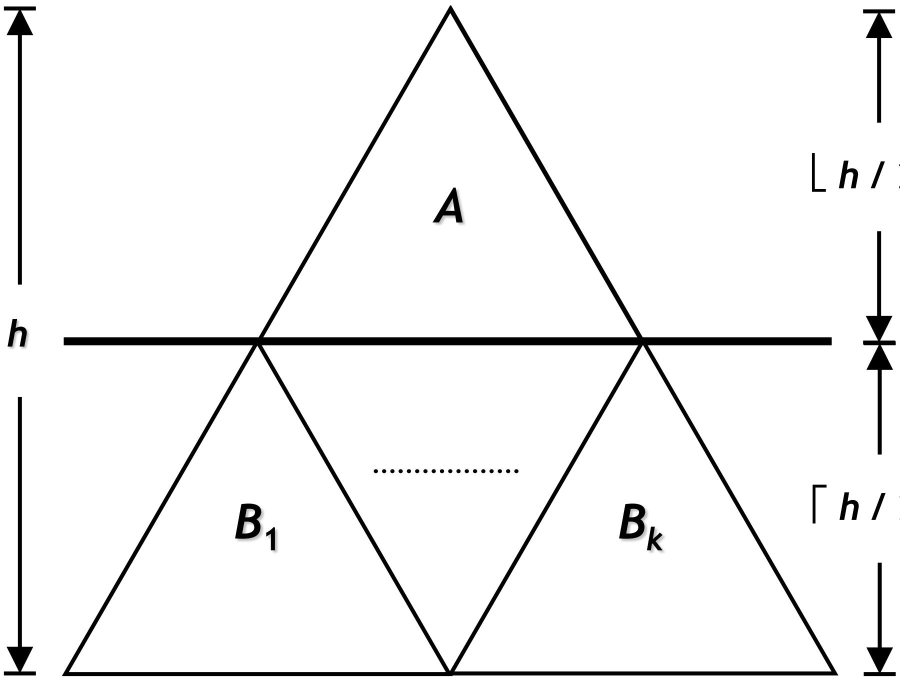
Recursive Subdivision

If the tree contains n nodes,

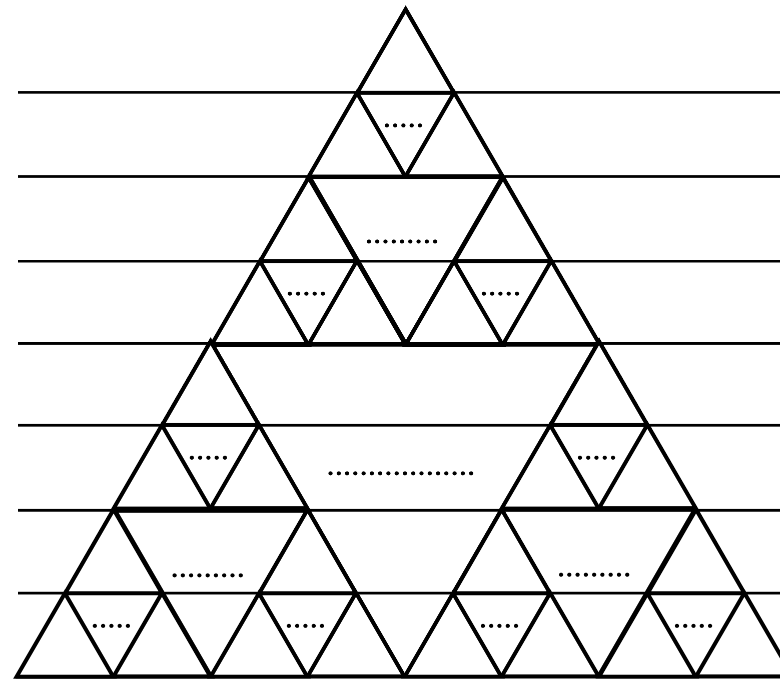
each subtree contains $\Theta(2^{h/2}) = \Theta(\sqrt{n})$ nodes,

and $k = \Theta(\sqrt{n})$.

van Emde Boas Layout



a binary search tree



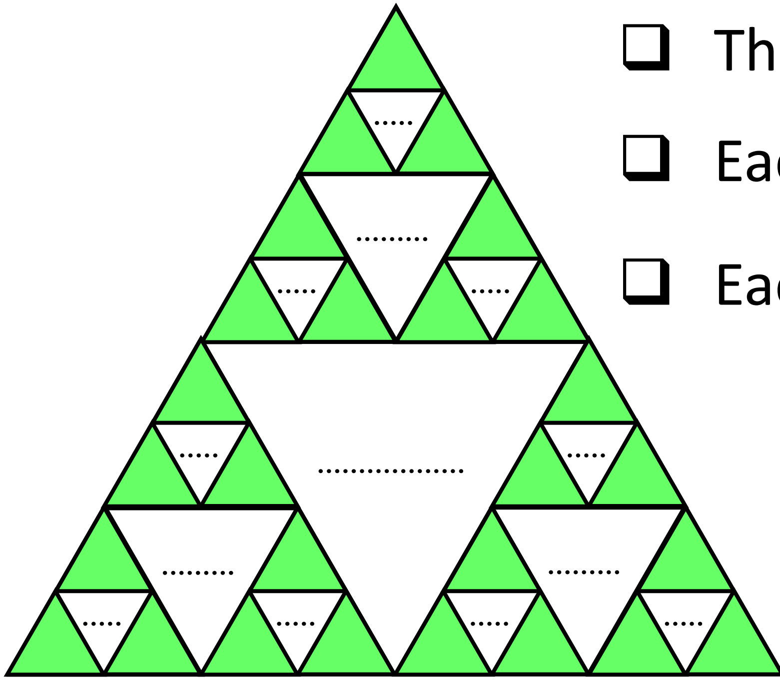
Recursive Subdivision

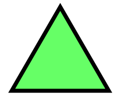
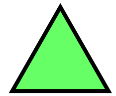
If the tree contains n nodes,

each subtree contains $\Theta\left(2^{h/2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

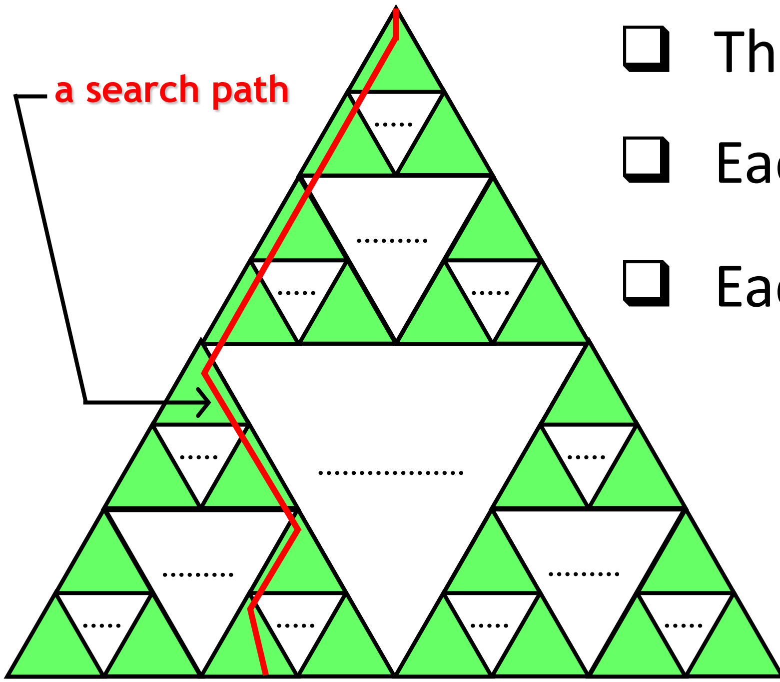
and $k = \Theta\left(\sqrt{n}\right)$.

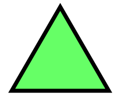
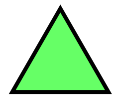
I/O-Complexity of a Search

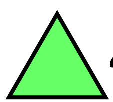


- The height of the tree is $\log n$
- Each  has height between $\frac{1}{2}\log B$ & $\log B$.
- Each  spans at most 2 blocks of size B.

I/O-Complexity of a Search



- The height of the tree is $\log n$
- Each  has height between $\frac{1}{2}\log B$ & $\log B$.
- Each  spans at most 2 blocks of size B.

- *p* = number of 's visited by a search path
- Then $p \geq \frac{\log n}{\log B} = \log_B n$, and $p \leq \frac{\log n}{\frac{1}{2}\log B} = 2\log_B n$
- The number of blocks transferred is $\leq 2 \times 2\log_B n = 4\log_B n$

Sorting
(Mergesort)

Merge Sort

Merge-Sort (A, p, r) { *sort the elements in* $A[\, p \ldots r \,]$ }

1. *if* $p < r$ *then*
2. $q \leftarrow \lfloor (p + r) / 2 \rfloor$
3. *Merge-Sort* (A, p, q)
4. *Merge-Sort* (A, $q + 1$, r)
5. *Merge* (A, p, q, r)

Merging *k* Sorted Sequences

- $k \geq 2$ sorted sequences $S_1, S_2, \ldots, S_k$ stored in external memory
- $|S_i| = n_i$ for $1 \leq i \leq k$
- $n = n_1 + n_2 + \cdots + n_k$ is the length of the merged sequence S
- S (initially empty) will be stored in external memory
- Cache must be large enough to store
    - one block from each S_i
    - one block from S

  Thus $M \geq (k + 1)B$

Merging k Sorted Sequences

- Let $\mathcal{B}_i$ be the cache block associated with S_i, and let $\mathcal{B}$ be the block associated with S (initially all empty)
- Whenever a $\mathcal{B}_i$ is empty fill it up with the next block from S_i
- Keep transferring the next smallest element among all $\mathcal{B}_i$s to $\mathcal{B}$
- Whenever $\mathcal{B}$ becomes full, empty it by appending it to S
- In the *Ideal Cache Model* the block emptying and replacements will happen automatically $\Rightarrow$ cache-oblivious merging

I/O Complexity

- Reading S_i: #block transfers $\leq 2 + \frac{n_i}{B}$
- Writing S: #block transfers $\leq 1 + \frac{n}{B}$
- Total #block transfers $\leq 1 + \frac{n}{B} + \sum_{1 \leq i \leq k} \left(2 + \frac{n_i}{B} \right) = O\left(k + \frac{n}{B} \right)$

Cache-Oblivious 2-Way Merge Sort

Merge-Sort (A, p, r) { *sort the elements in* $A[\, p \ldots r \,]$ }

1. *if* $p < r$ *then*
2. $q \leftarrow \lfloor (p + r) / 2 \rfloor$
3. *Merge-Sort* (A, p, q)
4. *Merge-Sort* ($A, q + 1, r$)
5. *Merge* (A, p, q, r)

I/O Complexity: $$Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & if\ n \le M, \\ 2Q\left(\frac{n}{2}\right) + O\left(1 + \frac{n}{B}\right), & otherwise. \end{cases}$$

$$= O\left(\frac{n}{B} \log \frac{n}{M}\right)$$

How to improve this bound?

Cache-Oblivious *k*-Way Merge Sort

I/O Complexity: $Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & if\ n \leq M, \\ k \cdot Q\left(\frac{n}{k}\right) + O\left(k + \frac{n}{B}\right), & otherwise. \end{cases}$

$$= O\left(k \cdot \frac{n}{M} + \frac{n}{B} \log_k \frac{n}{M}\right)$$

How large can k be?

Recall that for k-way merging, we must ensure

$$M \geq (k+1)B \Rightarrow k \leq \frac{M}{B} - 1$$

Cache-Aware $\left(\frac{M}{B}-1\right)$-Way Merge Sort

I/O Complexity: $Q(n) = \begin{cases} O\left(1+\frac{n}{B}\right), & if\ n \leq M, \\ k \cdot Q\left(\frac{n}{k}\right) + O\left(k+\frac{n}{B}\right), & otherwise. \end{cases}$

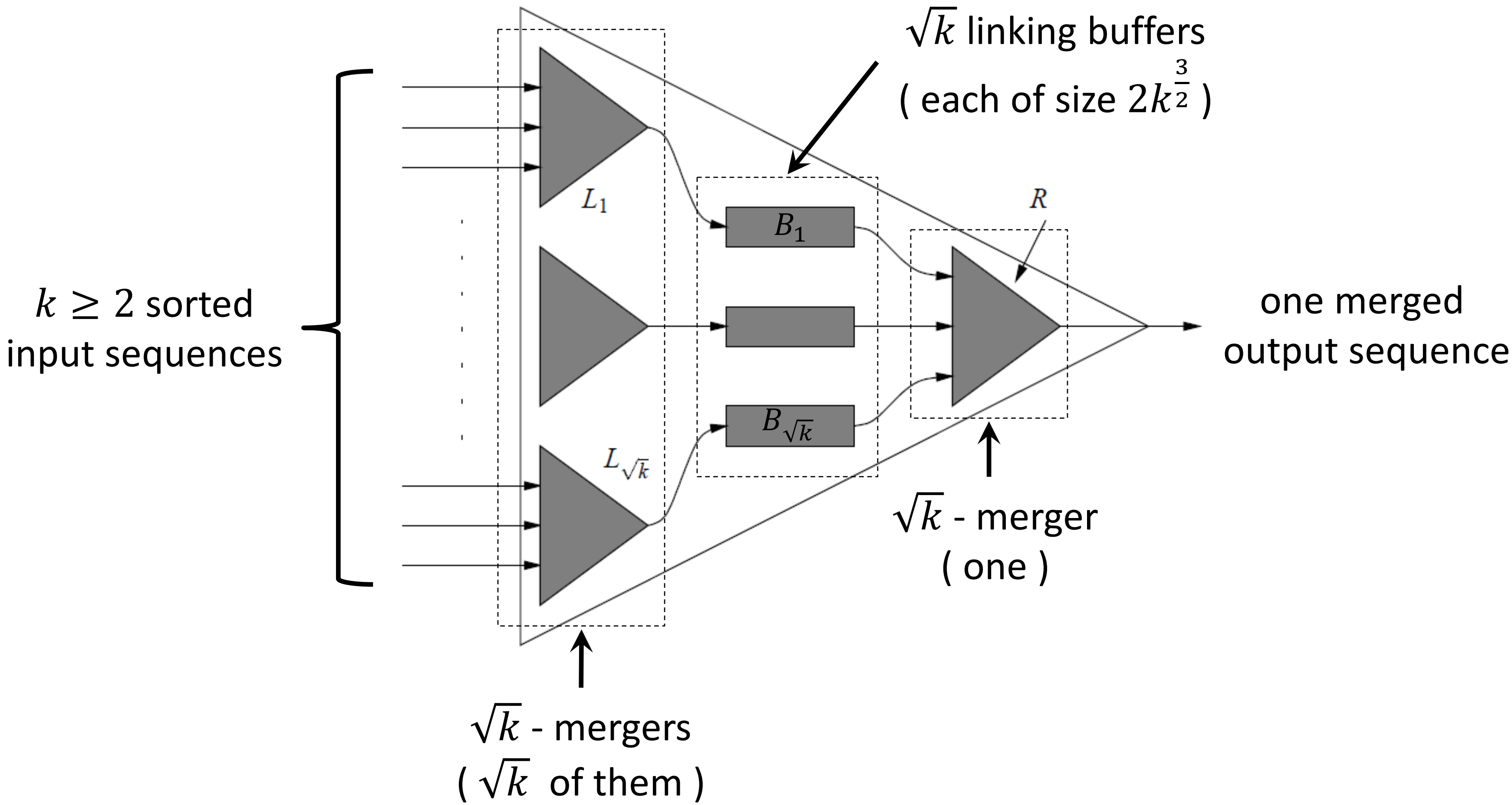
$$= O\left(k \cdot \frac{n}{M} + \frac{n}{B}\log_k \frac{n}{M}\right)$$

Using $k = \frac{M}{B} - 1$, we get:

$$Q(n) = O\left(\left(\frac{M}{B}-1\right)\frac{n}{M} + \frac{n}{B}\log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right) = O\left(\frac{n}{B}\log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right)$$

Sorting
(Funnelsort)

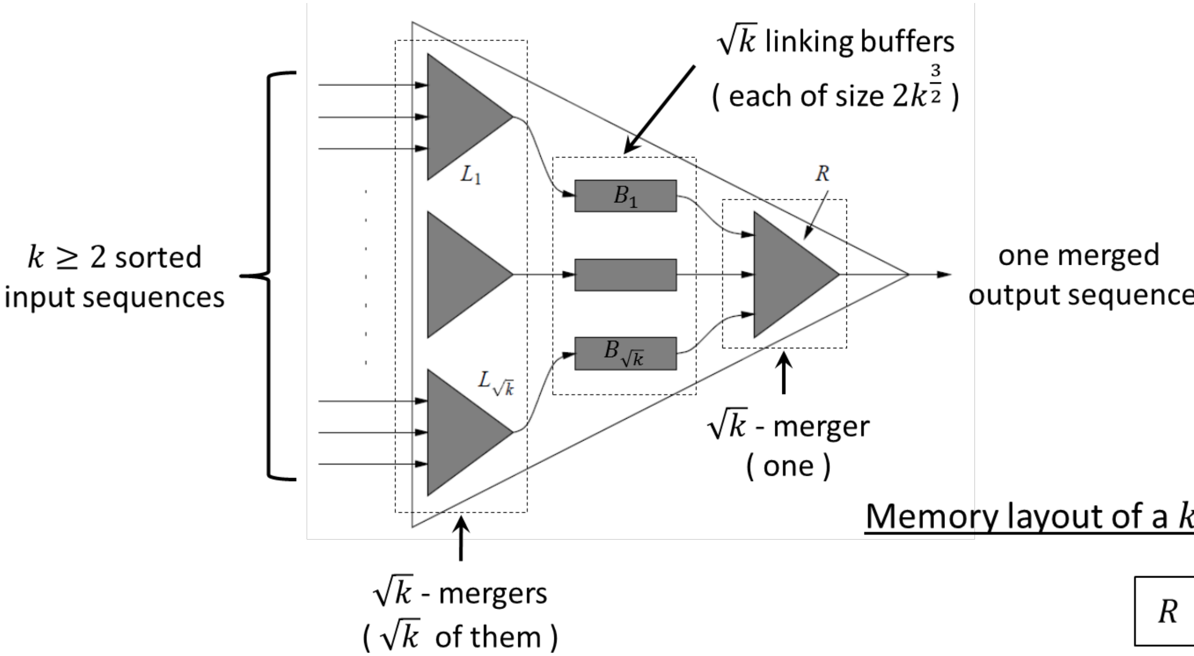
***k*-Merger (*k*-Funnel)**



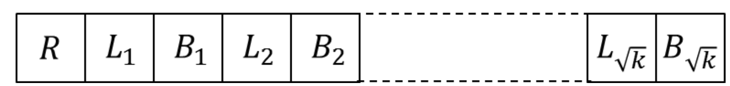
Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2	…	$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

***k*-Merger (*k*-Funnel)**



Memory layout of a k-merger:

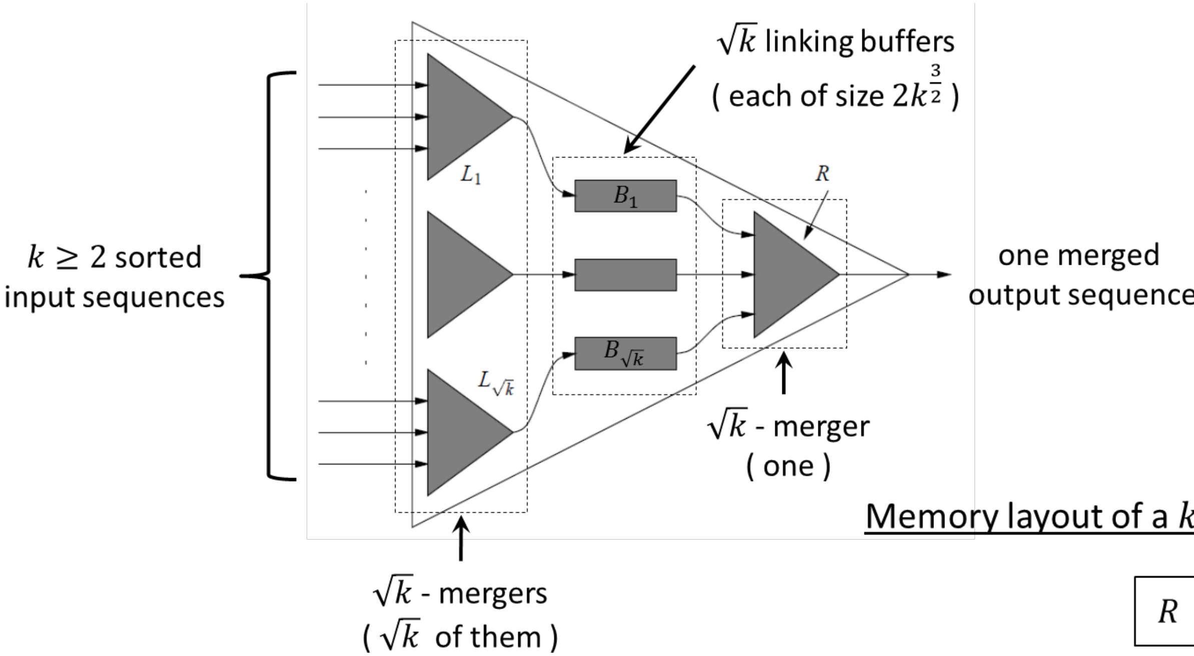


Space usage of a k-merger: $S(k) = \begin{cases} \Theta(1), & if\ k \leq 2, \\ \left(\sqrt{k}+1\right) S\left(\sqrt{k}\right) + \Theta\left(k^2\right), & otherwise. \end{cases}$

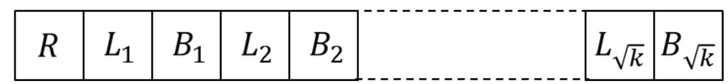
$$= \Theta\left(k^2\right)$$

A k-merger occupies $\Theta\left(k^2\right)$ contiguous locations.

k-Merger (k-Funnel)



Memory layout of a k-merger:

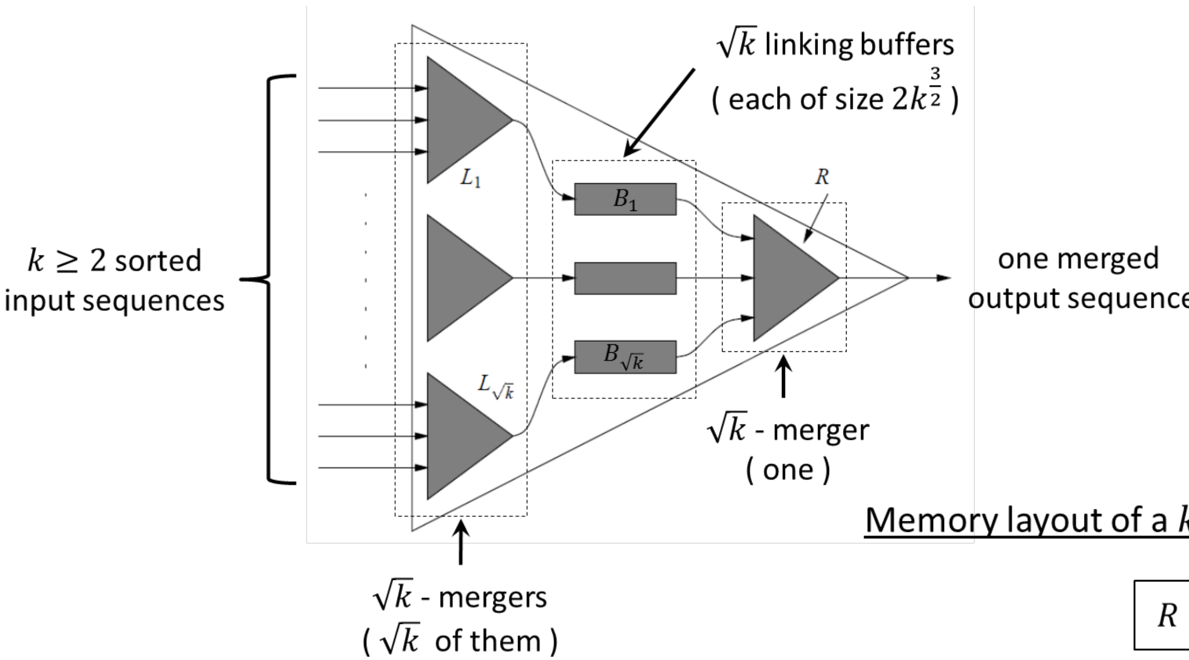


Each invocation of a k-merger

– produces a sorted sequence of length k^3

– incurs $O\left(1 + k + \frac{k^3}{B} + \frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right)$ cache misses provided $M = \Omega\left(B^2\right)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

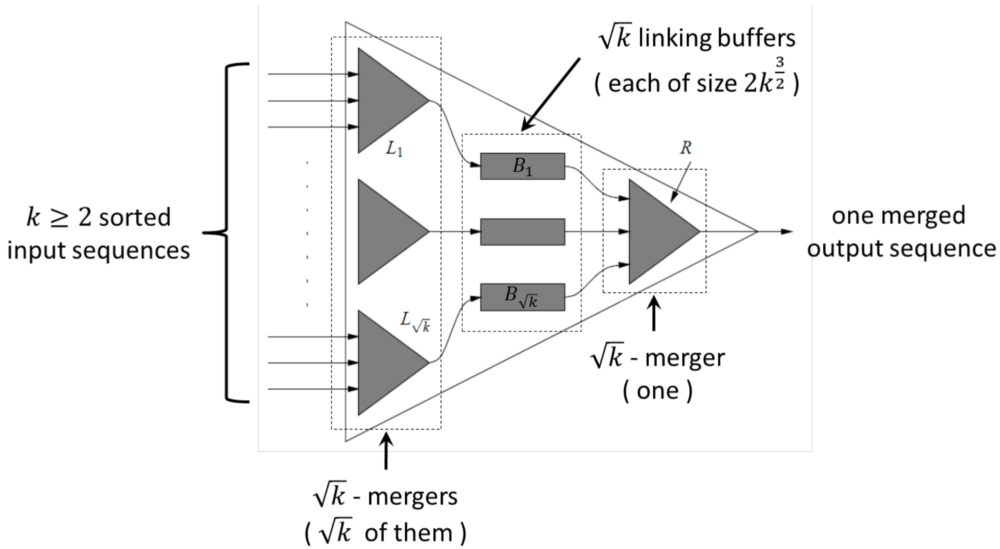
R	L_1	B_1	L_2	B_2	…	$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right) Q'\left(\sqrt{k}\right) + \Theta(k^2), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2	…	$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'\left(\sqrt{k}\right) + \Theta(k^2), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

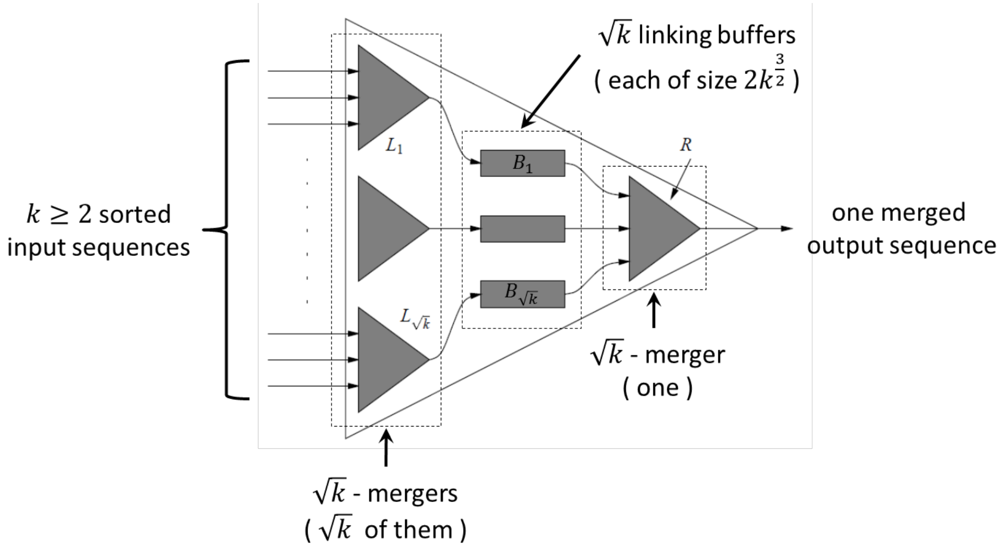
$$k < \alpha\sqrt{M}\text{: } Q'(k) = O\left(1 + k + \frac{k^3}{B}\right)$$

– Let r_i be #items extracted the i-th input queue. Then $\sum_{i=1}^{k} r_i = O\left(k^3\right)$.

– Since $k < \alpha\sqrt{M}$ and $M = \Omega\left(B^2\right)$, at least $\frac{M}{B} = \Omega(k)$ cache blocks are available for the input buffers.

– Hence, #cache-misses for accessing the input queues (assuming circular buffers) $= \sum_{i=1}^{k} O\left(1 + \frac{r_i}{B}\right) = O\left(k + \frac{k^3}{B}\right)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2	...	$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

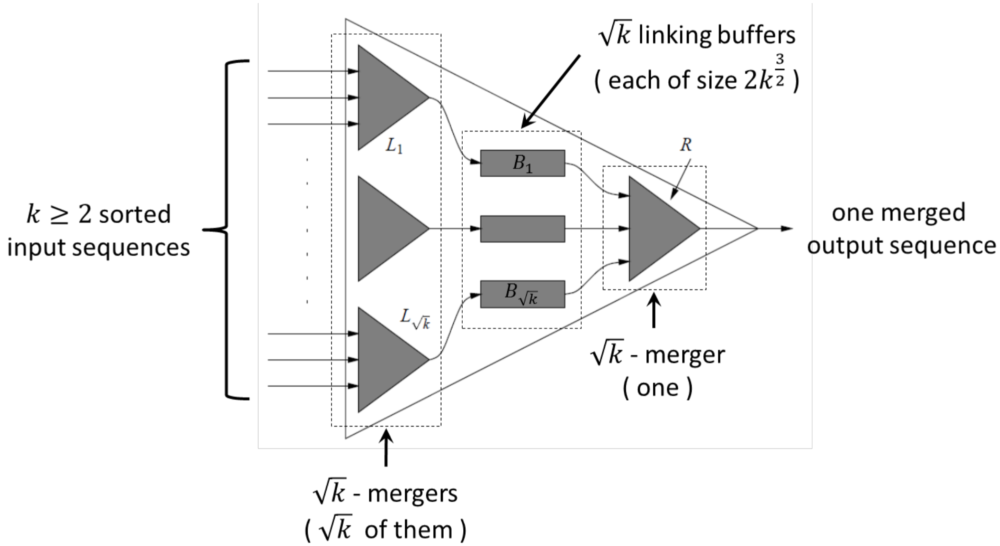
$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'\left(\sqrt{k}\right) + \Theta(k^2), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

$$k < \alpha\sqrt{M}\colon\ Q'(k) = O\left(1 + k + \frac{k^3}{B}\right)$$

- #cache-misses for accessing the input queues $= O\left(k + \frac{k^3}{B}\right)$
- #cache-misses for writing the output queue $= O\left(1 + \frac{k^3}{B}\right)$
- #cache-misses for touching the internal data structures $= O\left(1 + \frac{k^2}{B}\right)$
- Hence, total #cache-misses $= O\left(1 + k + \frac{k^3}{B}\right)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'(\sqrt{k}) + \Theta(k^2), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \quad \text{provided } M = \Omega(B^2)$$

$$k \geq \alpha\sqrt{M}:\ Q'^{(k)} = \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'\left(\sqrt{k}\right) + \Theta\left(k^2\right)$$

– Each call to R outputs $k^{\frac{3}{2}}$ items. So, #times merger R is called $= \frac{k^3}{k^{\frac{3}{2}}} = k^{\frac{3}{2}}$

– Each call to an L_i puts $k^{\frac{3}{2}}$ items into B_i. Since k^3 items are output, and the buffer space is $\sqrt{k} \times 2k^{\frac{3}{2}} = 2k^2$, #times the L_i's are called $\leq k^{\frac{3}{2}} + 2\sqrt{k}$

– Before each call to R, the merger must check each L_i for emptiness, and thus incurring $O\left(\sqrt{k}\right)$ cache-misses. So, #such cache-misses $= k^{\frac{3}{2}} \times O\left(\sqrt{k}\right) = O\left(k^2\right)$

Funnelsort

- Split the input sequence A of length n into $n^{\frac{1}{3}}$ contiguous subsequences $A_1, A_2, \dots, A_{n^{\frac{1}{3}}}$ of length $n^{\frac{2}{3}}$ each
- Recursively sort each subsequence
- Merge the $n^{\frac{1}{3}}$ sorted subsequences using a $n^{\frac{1}{3}}$-merger

Cache-complexity:

$$Q(n) = \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le M, \\ n^{\frac{1}{3}} Q\left(n^{\frac{2}{3}}\right) + Q'\left(n^{\frac{1}{3}}\right), & otherwise. \end{cases}$$

$$= \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le M, \\ n^{\frac{1}{3}} Q\left(n^{\frac{2}{3}}\right) + \mathrm{O}\left(\frac{n}{B} \log_M \left(\frac{n}{B}\right)\right), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(1 + \frac{n}{B} \log_M n\right)$$