CSE 548: Analysis of Algorithms

Lecture 24
(Analyzing I/O and Cache Performance)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2012

Memory: Fast, Large & Cheap!

For efficient computation we need

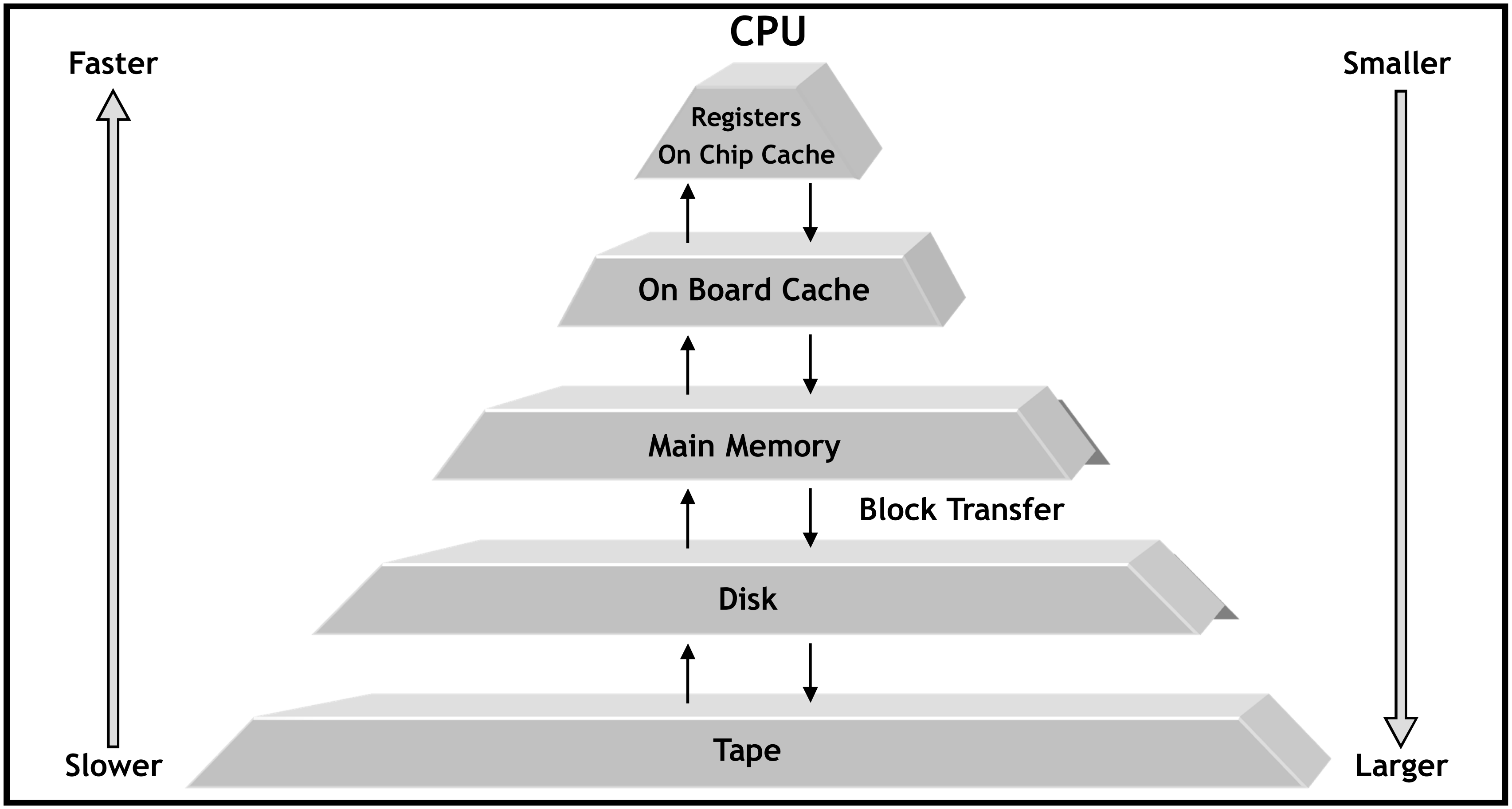
- fast processors
- fast and large (but not so expensive) memory

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

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

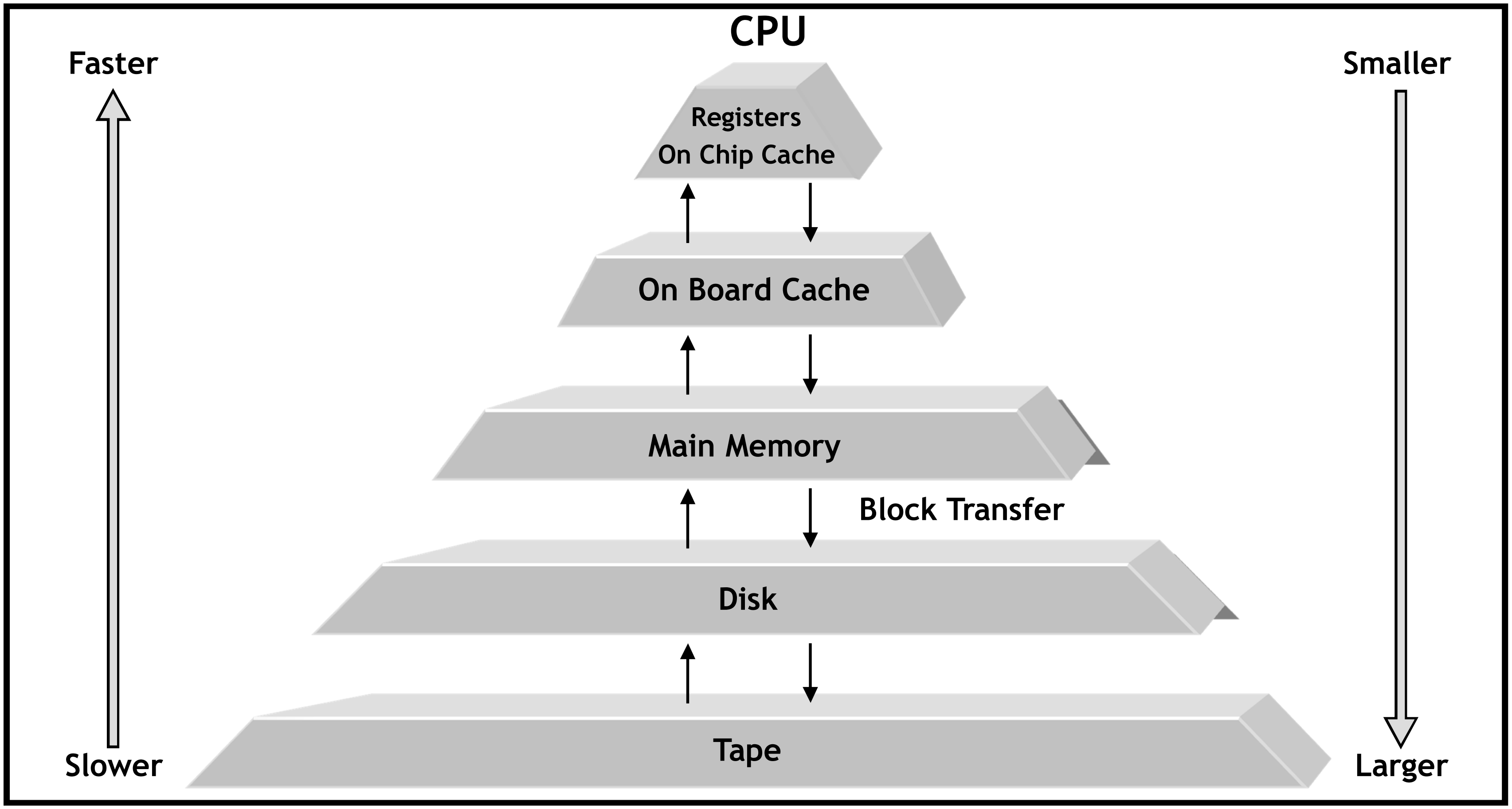
The Memory Hierarchy



A *memory hierarchy* is

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

The Memory Hierarchy

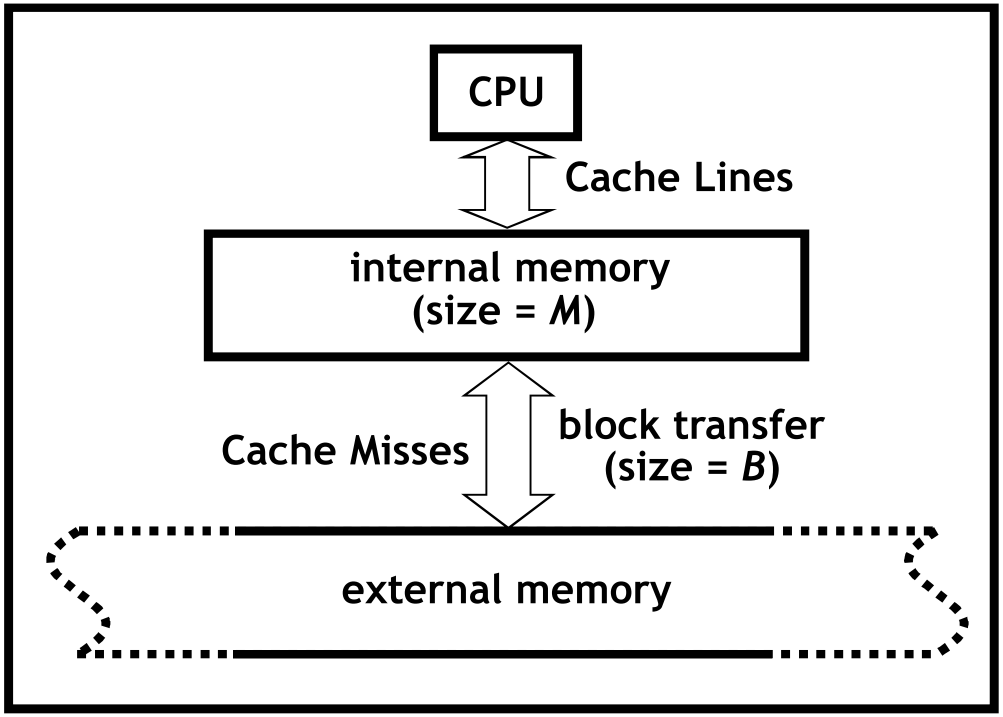


To perform well on a memory hierarchy algorithms must have <u>high locality</u> in their memory access patterns.

The Two-level I/O Model

The *two-level I/O (or cache-aware) 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

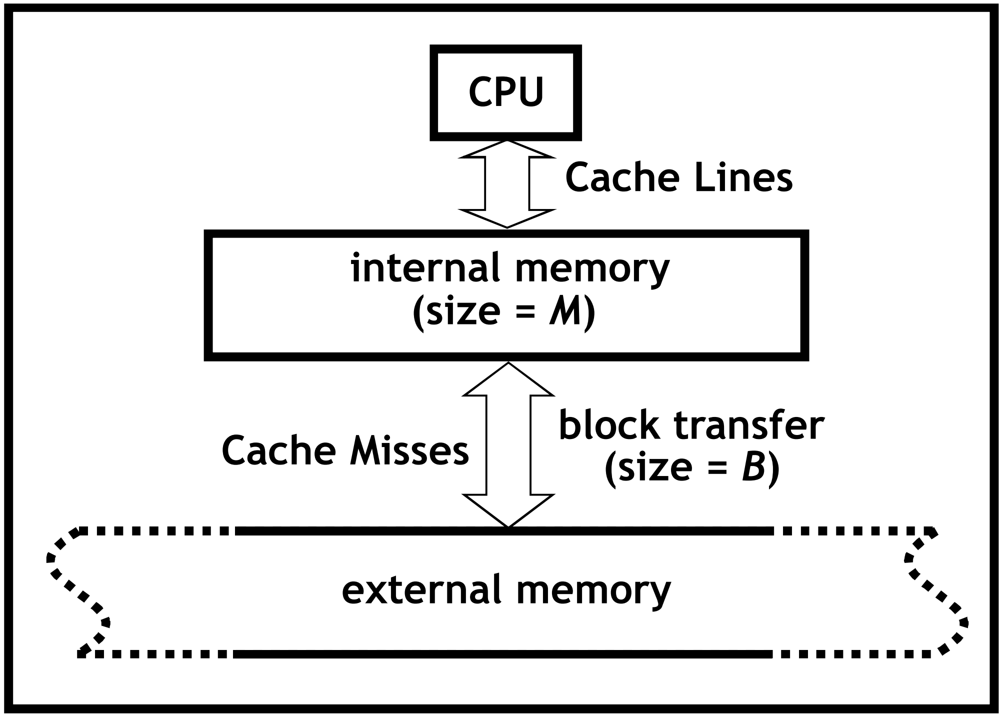
Basic I/O complexities: $scan(N) = \Theta\left(\frac{N}{B}\right)$ and $sort(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$

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 additional feature:



algorithms for this model 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:

- $M = \Omega(B^2)$, 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	$O\left(\min\left(sort(E)\log_2\log_2 V,\ V+sort(E)\right)\right)$	$O\left(\min\left(sort(E)\log_2\log_2\frac{VB}{E},\ V+sort(E)\right)\right)$

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

Matrix Multiplication

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. $z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj}$

I/O-Complexity: Iter-MM

```
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

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

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

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

I/O-Complexity: Iter-MM

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. $z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj}$

store in **row-major** order

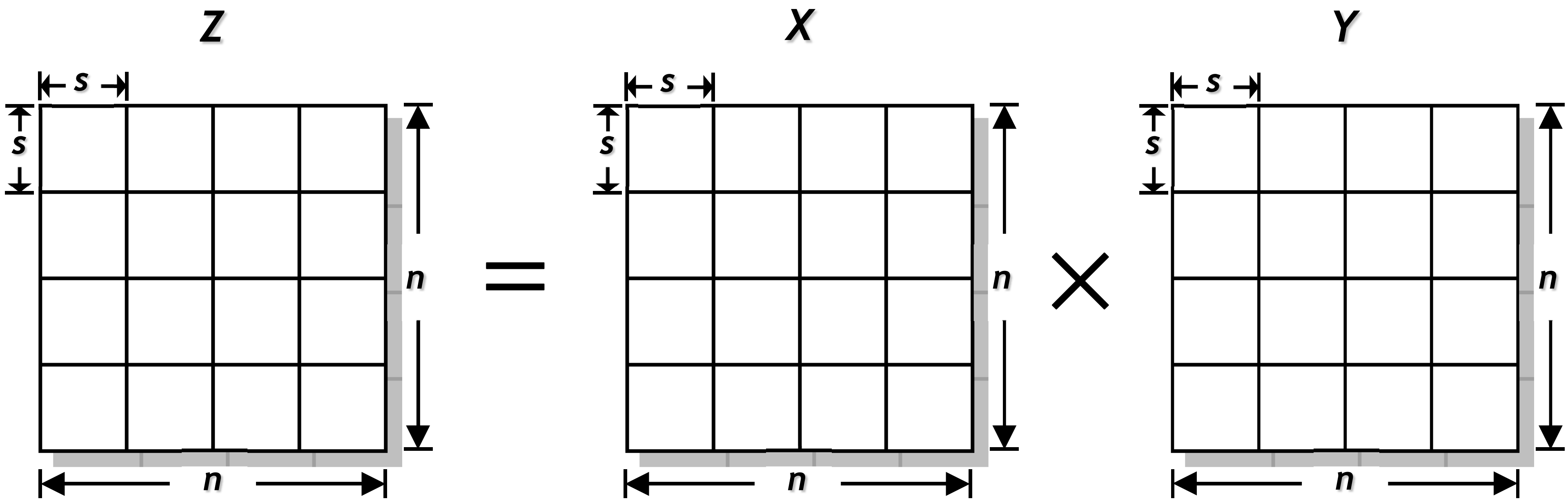
store in **column-major** order

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

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* $= O\left(n^2\left(1 + \frac{n}{B}\right)\right) = O\left(n^2 + \frac{n^3}{B}\right) = O\left(\frac{n^3}{B}\right)$

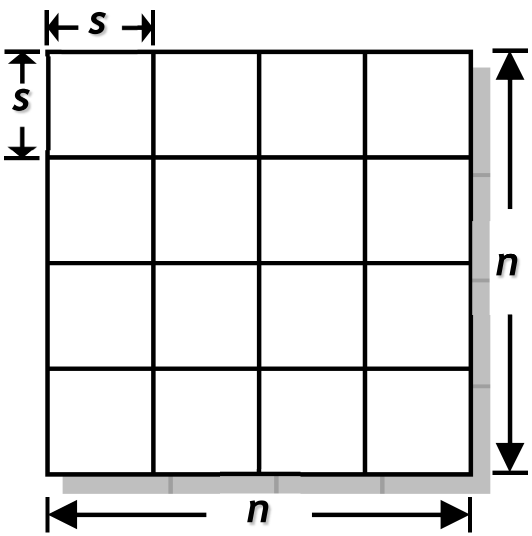
Block Matrix Multiplication



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)

I/O-Complexity: Block-MM



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)

Choose $s = \Theta\left(\sqrt{M}\right)$, so that X_{ik}, Y_{kj} and Z_{ij} just fit into the cache.

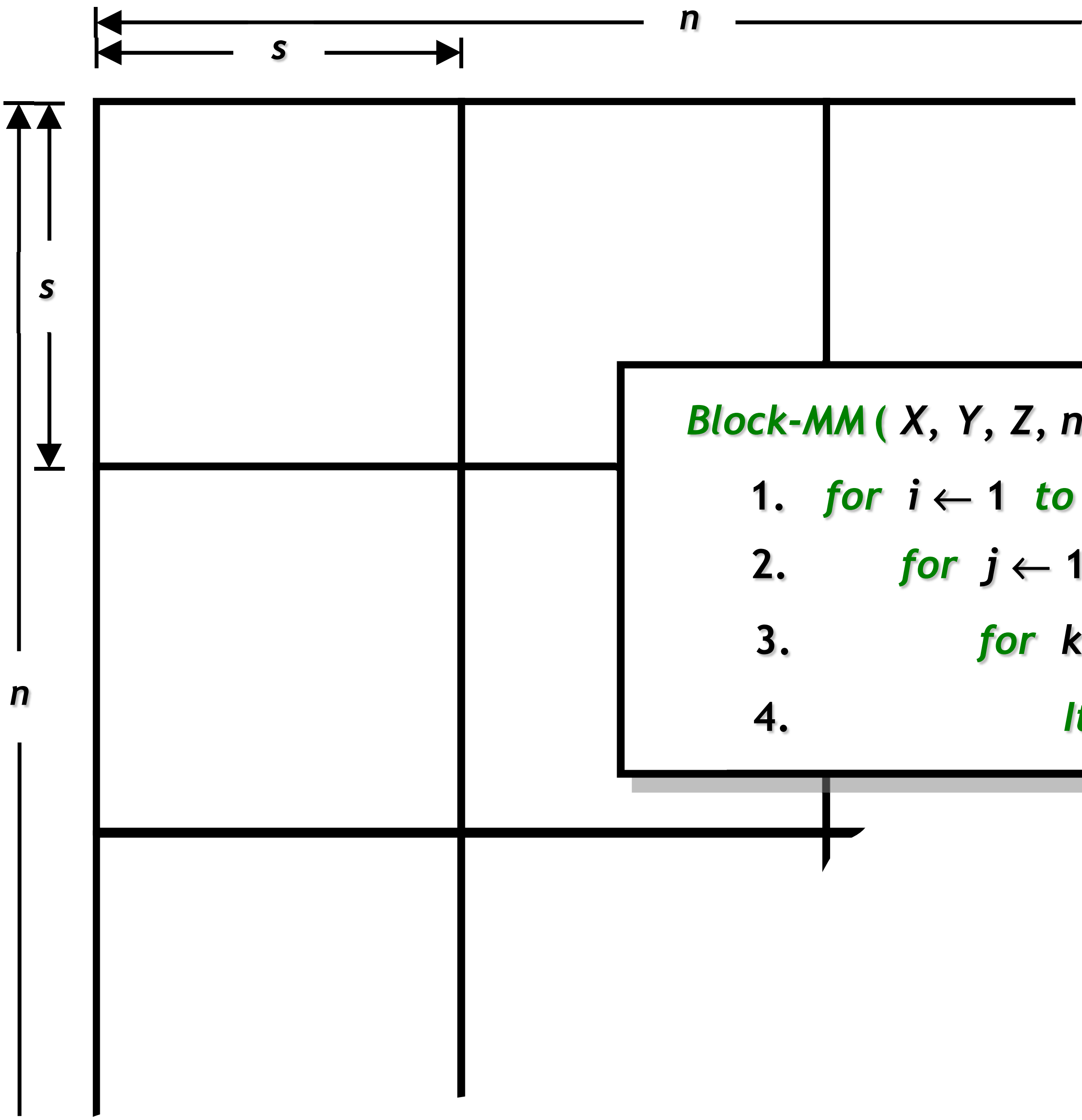
Then line 4 incurs $\Theta\left(s\left(1+\frac{s}{B}\right)\right)$ cache misses.

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

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

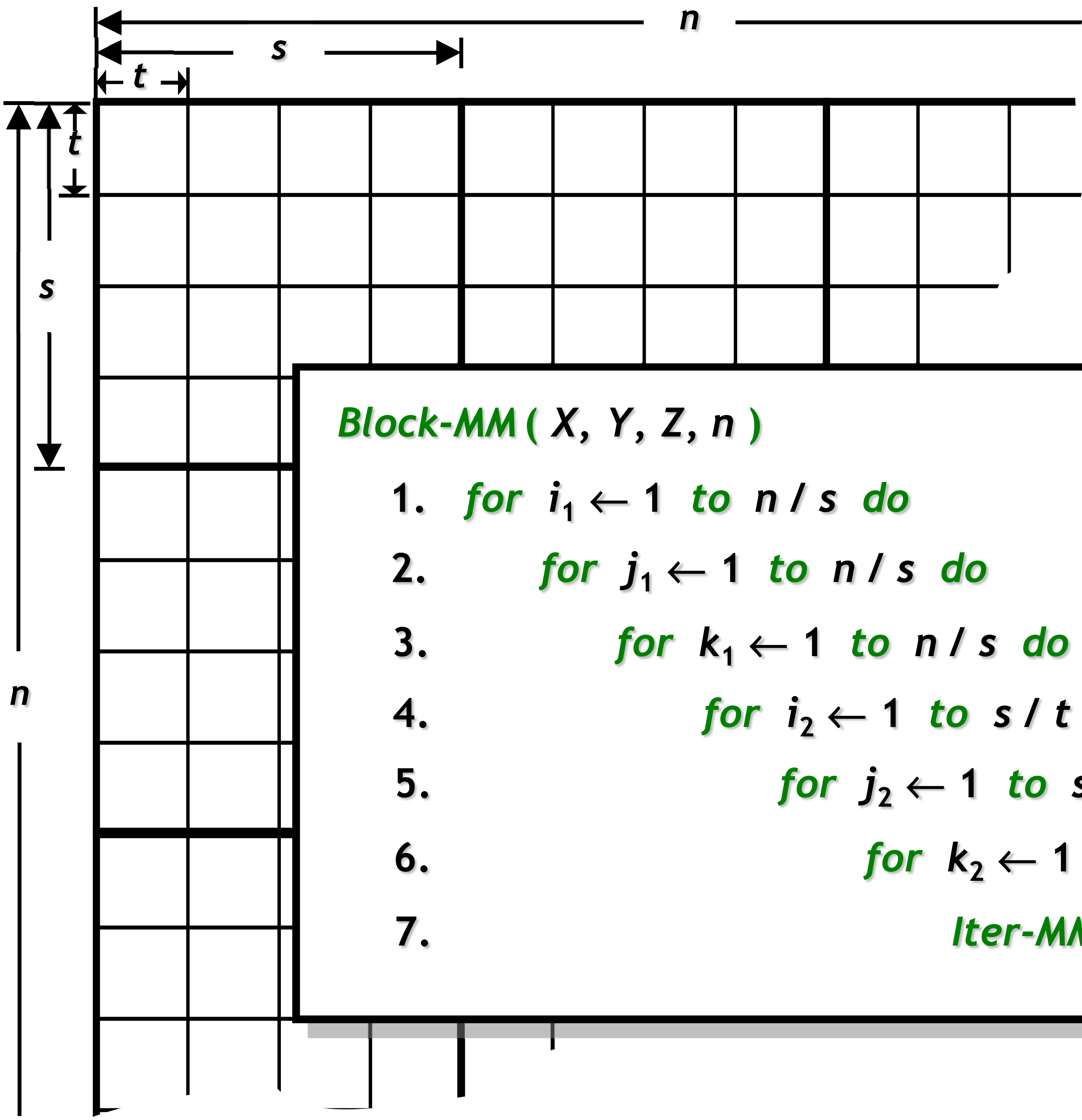
(Optimal: Hong & Kung, STOC'81)

Multiple Levels of Cache




```
Block-MM ( X, Y, Z, n )
    for i ← 1 to n / s do
        for j ← 1 to n / s do
            for k ← 1 to n / s do
                Iter-MM ( X_ik, Y_kj, Z_ij, s )
```

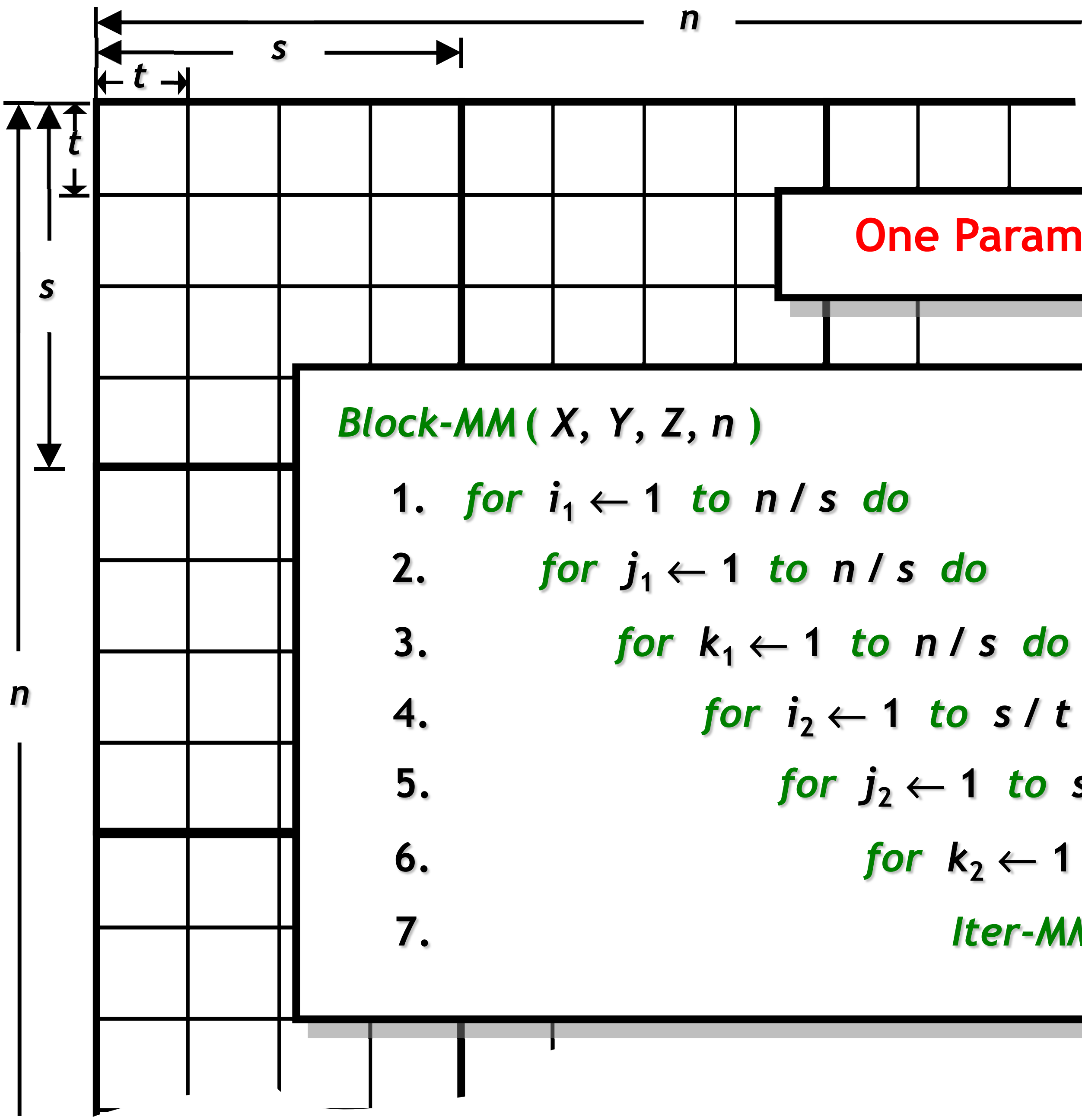
Multiple Levels of Cache



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

1. *for* $i_1 \leftarrow 1$ *to* n / s *do*
2. *for* $j_1 \leftarrow 1$ *to* n / s *do*
3. *for* $k_1 \leftarrow 1$ *to* n / s *do*
4. *for* $i_2 \leftarrow 1$ *to* s / t *do*
5. *for* $j_2 \leftarrow 1$ *to* s / t *do*
6. *for* $k_2 \leftarrow 1$ *to* s / t *do*
7. *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 )
```

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

Recursive Matrix Multiplication

$$\underbrace{\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}}_{Z} = \underbrace{\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}}_{X} \times \underbrace{\begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}}_{Y}$$

$$= \begin{bmatrix} X_{11}\,Y_{11} + X_{12}\,Y_{21} & X_{11}\,Y_{12} + X_{12}\,Y_{22} \\ X_{21}\,Y_{11} + X_{22}\,Y_{21} & X_{21}\,Y_{12} + X_{22}\,Y_{22} \end{bmatrix}$$

Recursive Matrix Multiplication

$$\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} = \begin{bmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{bmatrix}$$

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

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

I/O-Complexity: Rec-MM

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

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

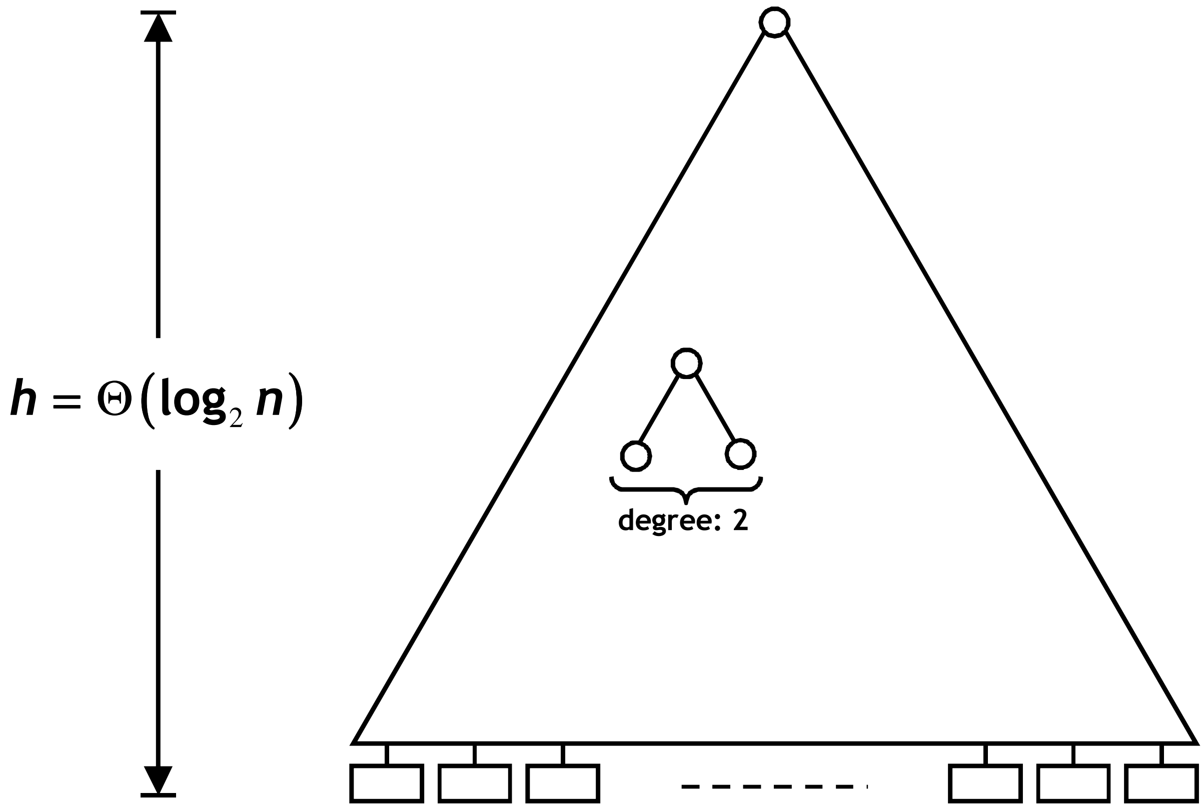
I/O-complexity of *Rec-MM*, $I(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right), & \text{if } n^2 \le \alpha M \\ 8I\left(\frac{n}{2}\right) + O(1), & \text{otherwise} \end{cases}$

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

(Optimal: Hong & Kung, STOC'81)

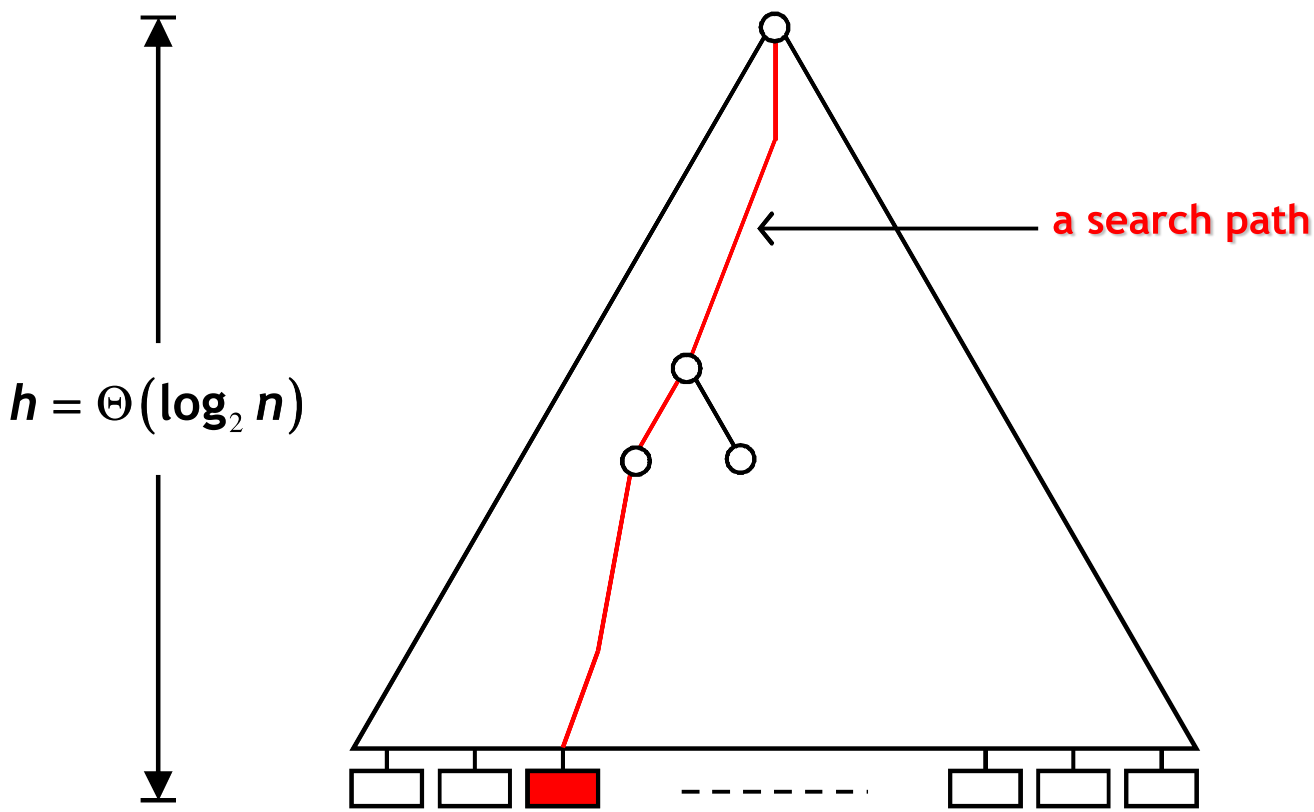
Searching
(Static B-Trees)

A Static Search Tree



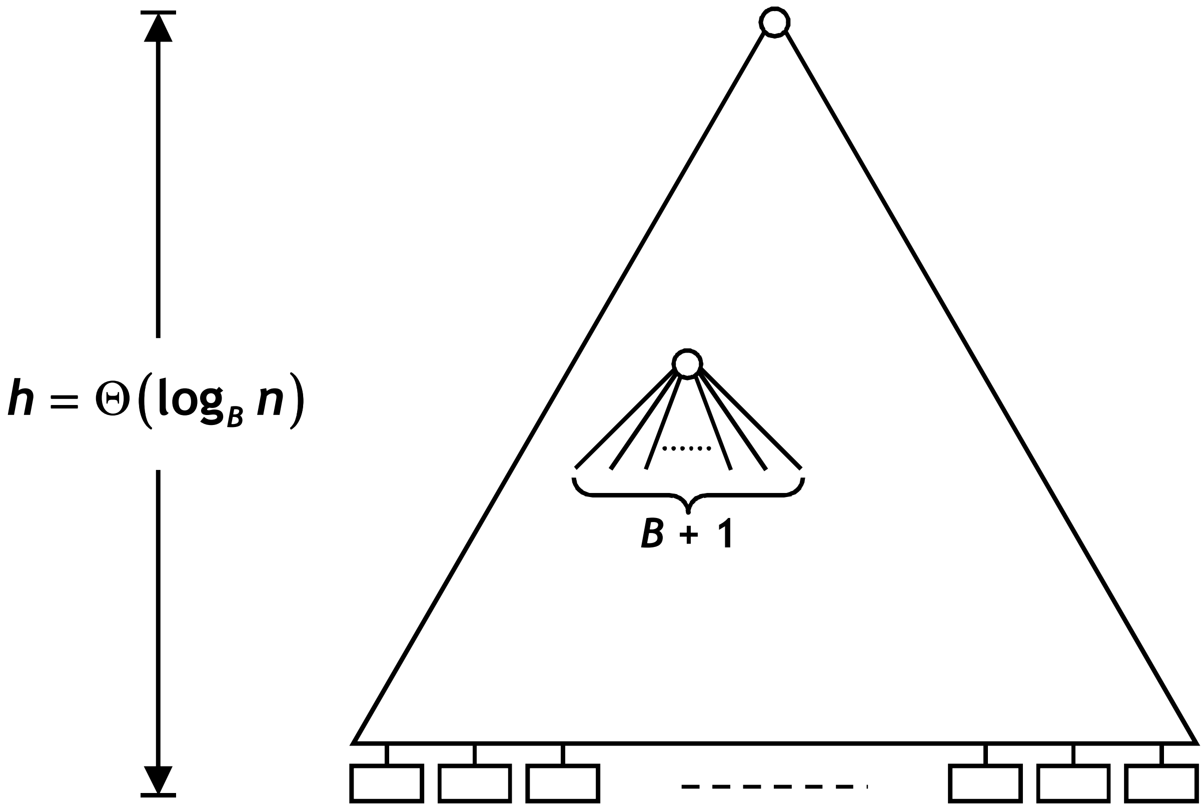
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$

A Static Search Tree



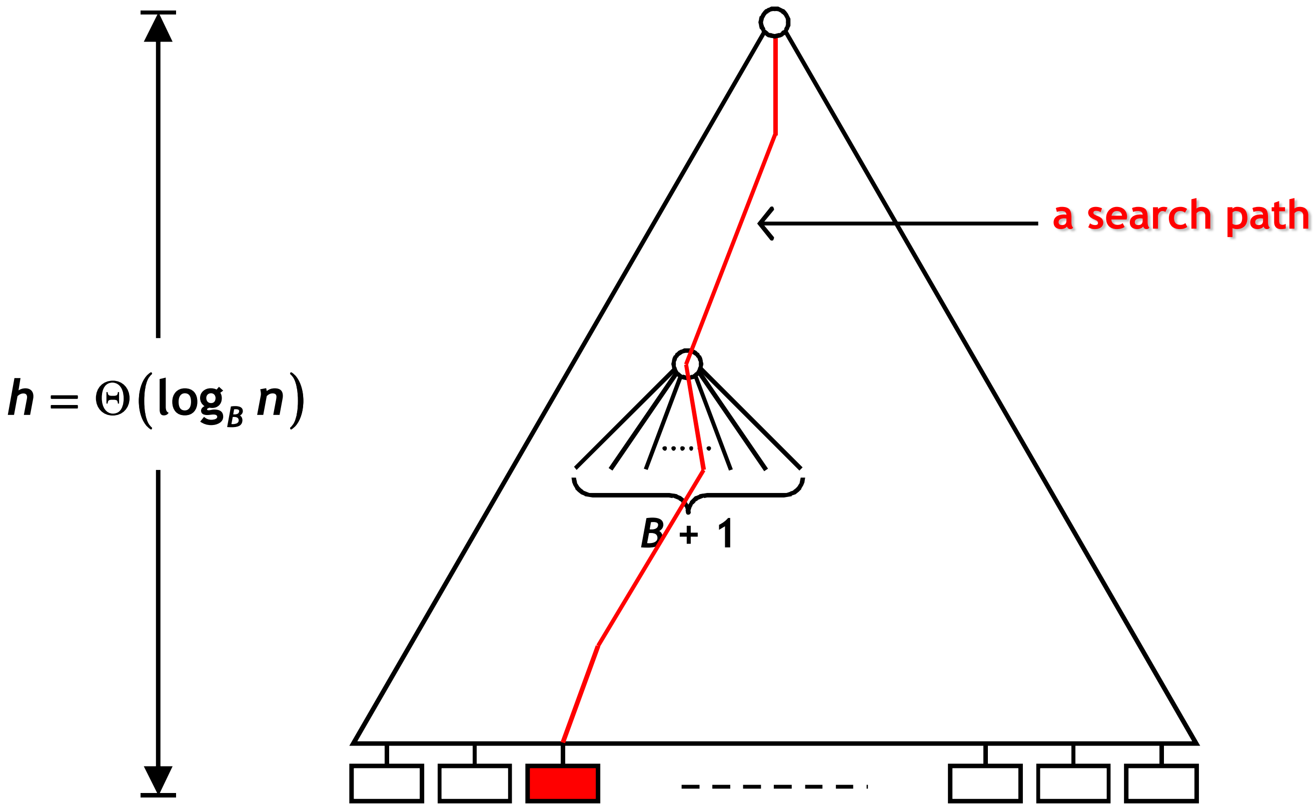
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$
- A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_2 n)$ I/Os

I/O-Efficient Static B-Trees



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

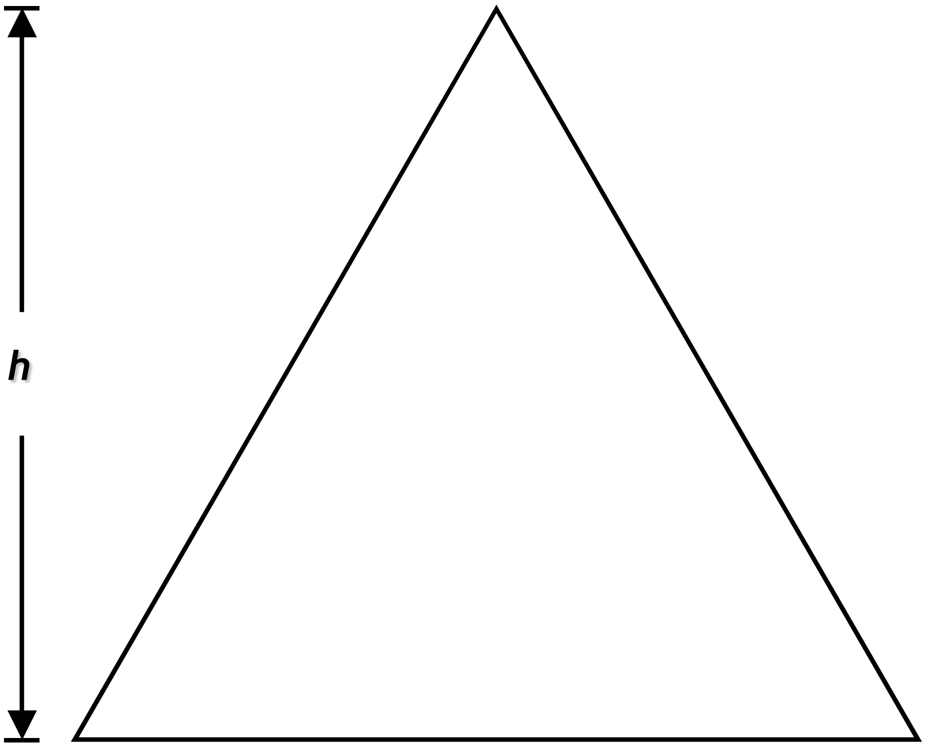
I/O-Efficient Static B-Trees



- 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

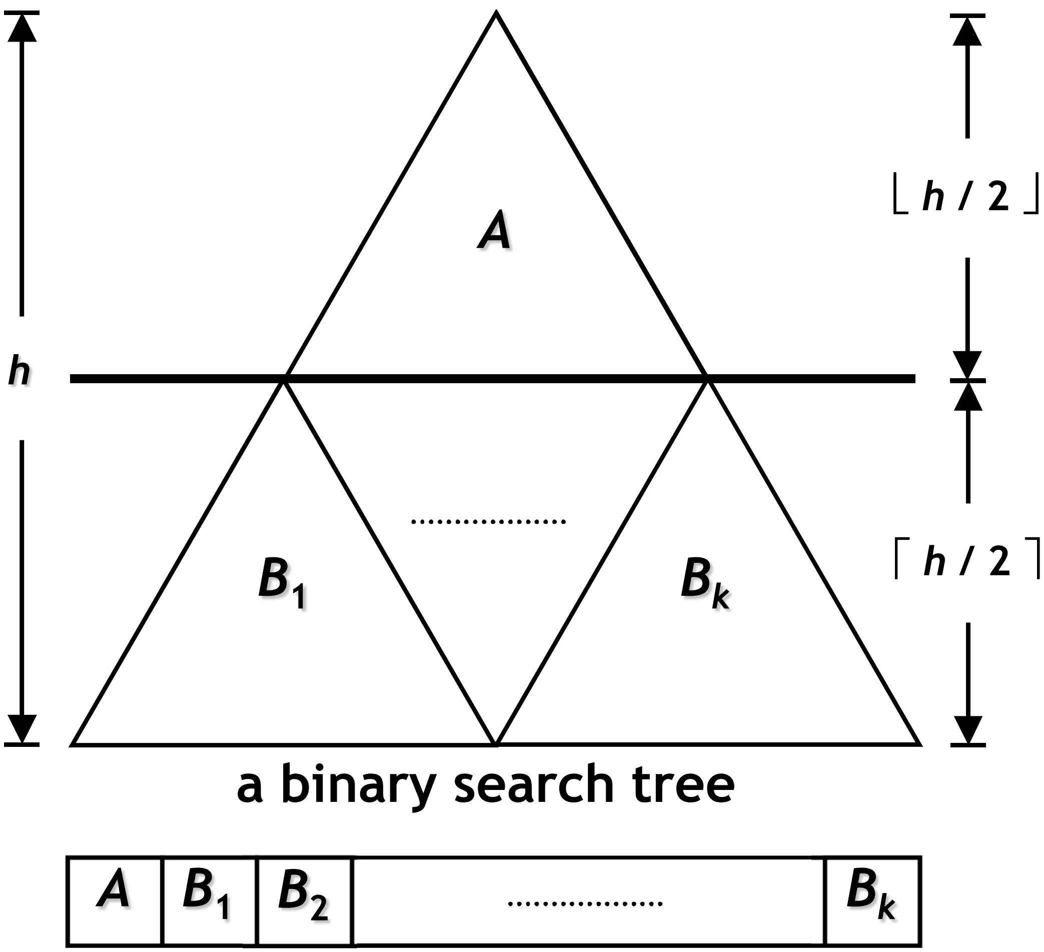
Cache-Oblivious Static B-Trees?

van Emde Boas Layout



a binary search tree

van Emde Boas Layout

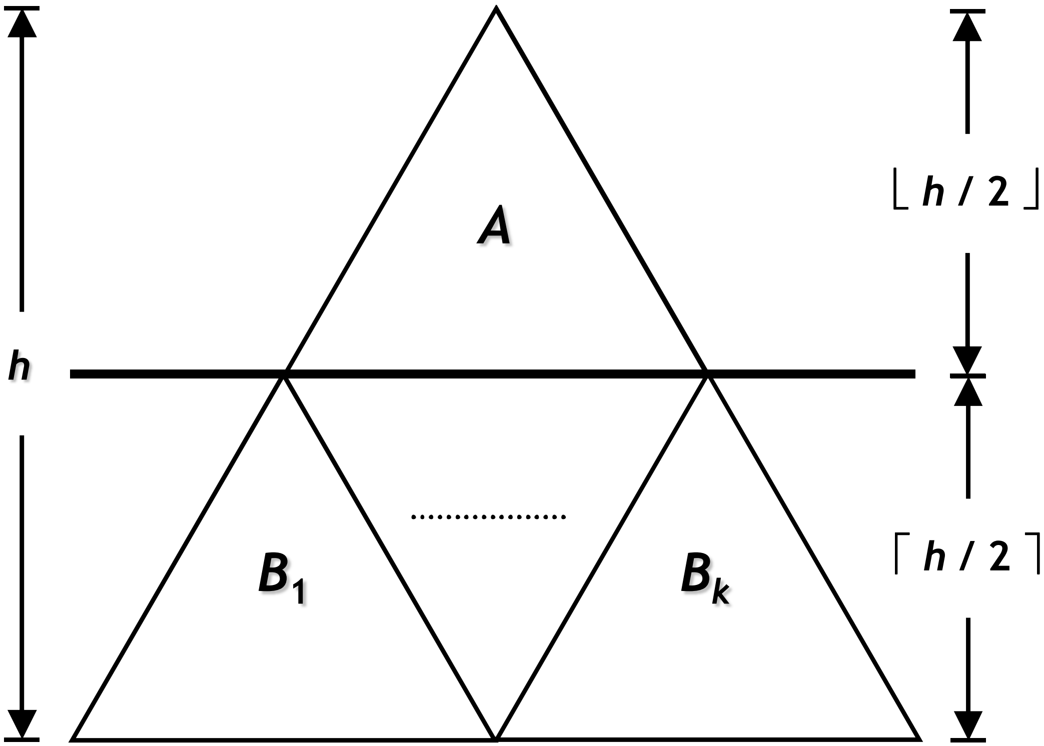


If the tree contains n nodes,

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

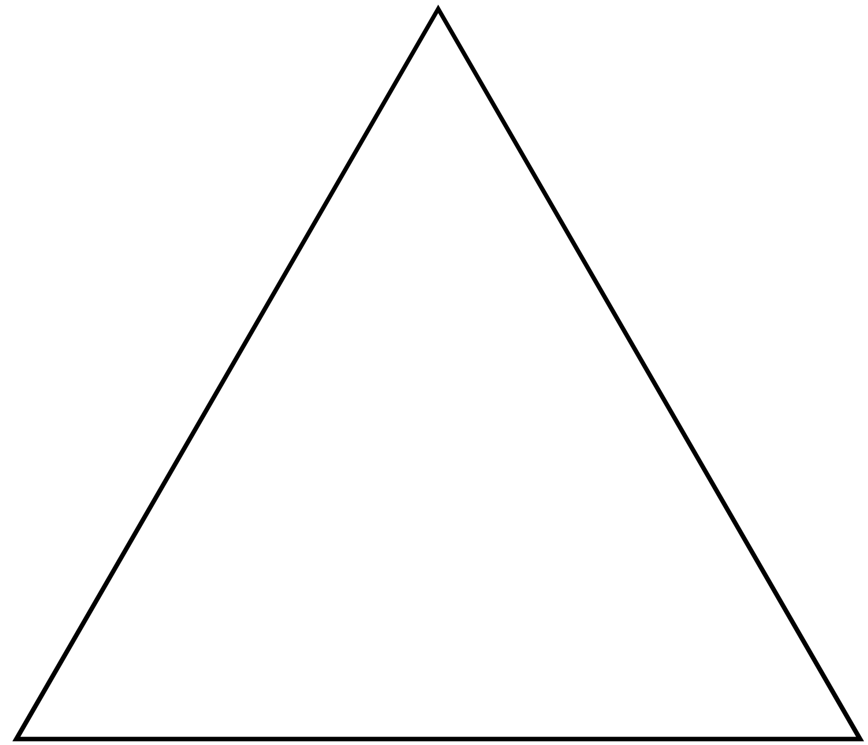
and $k = \Theta\left(\sqrt{n}\right)$

van Emde Boas Layout



a binary search tree

A	B_1	B_2	....	B_k



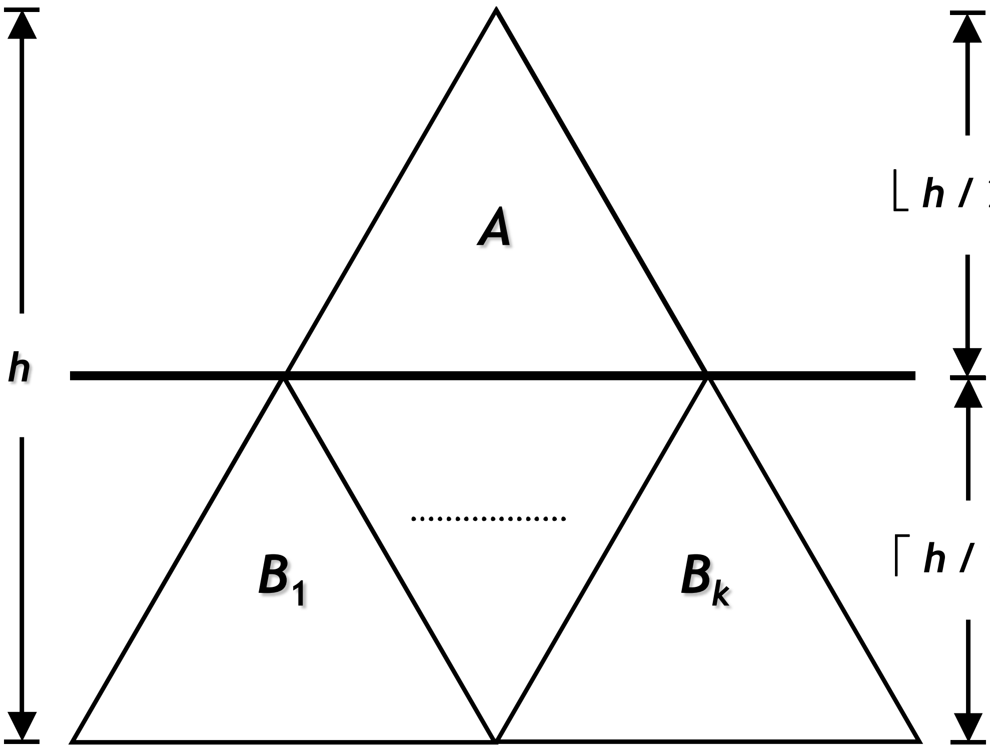
Recursive Subdivision

If the tree contains n nodes,

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

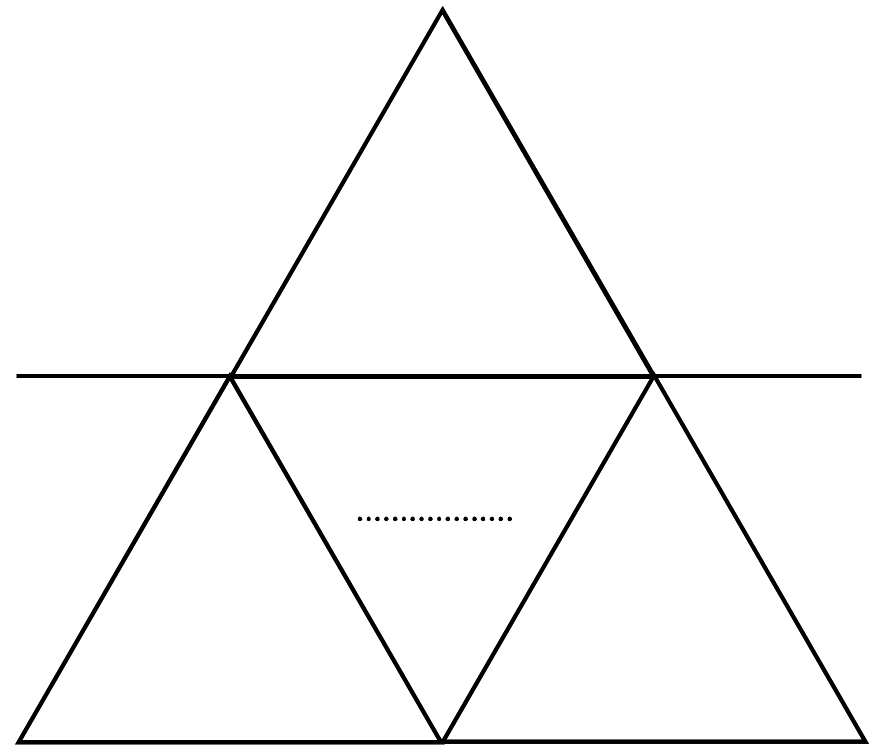
and $k = \Theta\left(\sqrt{n}\right)$

van Emde Boas Layout



a binary search tree

A	B_1	B_2	....	B_k



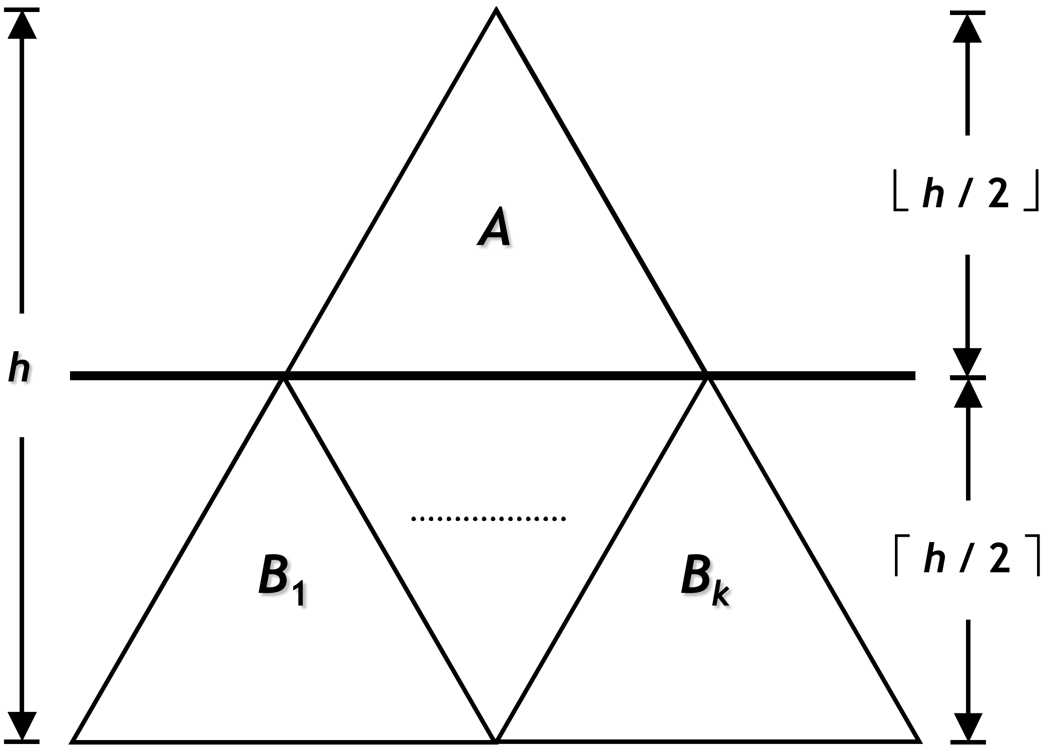
Recursive Subdivision

If the tree contains n nodes,

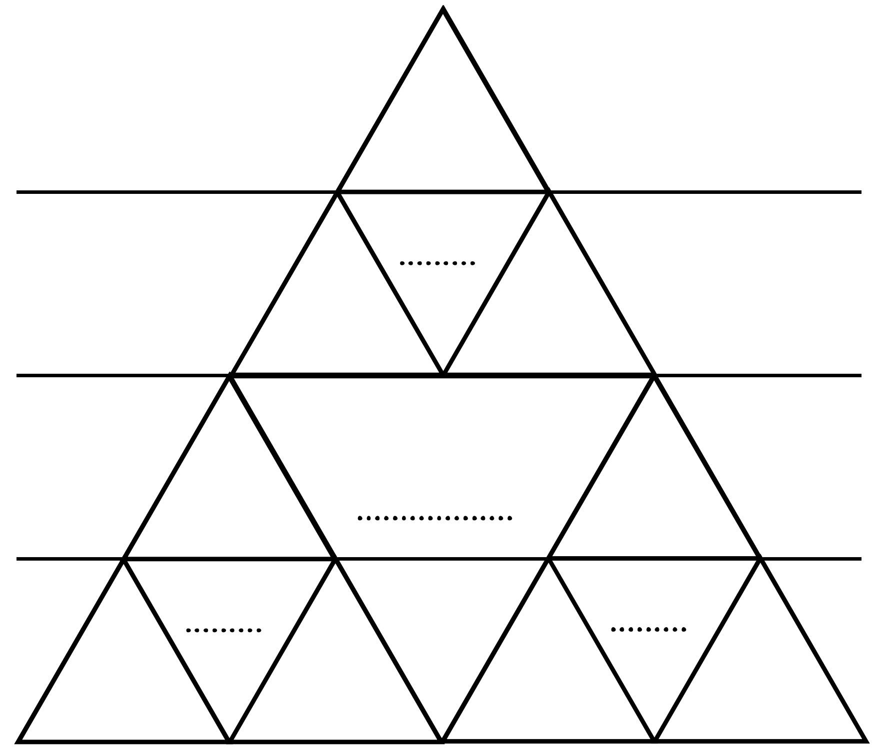
each subtree contains $\Theta\left(2^{\frac{h}{2}}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

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

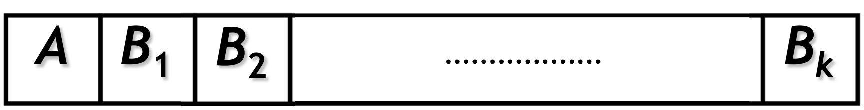
van Emde Boas Layout



a binary search tree



Recursive Subdivision

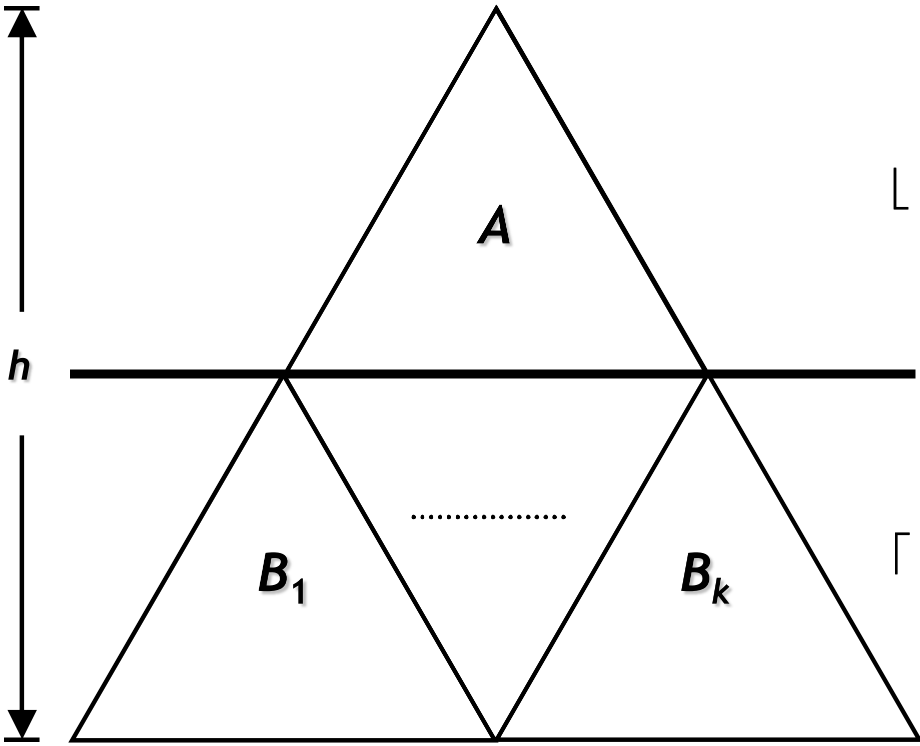


If the tree contains n nodes,

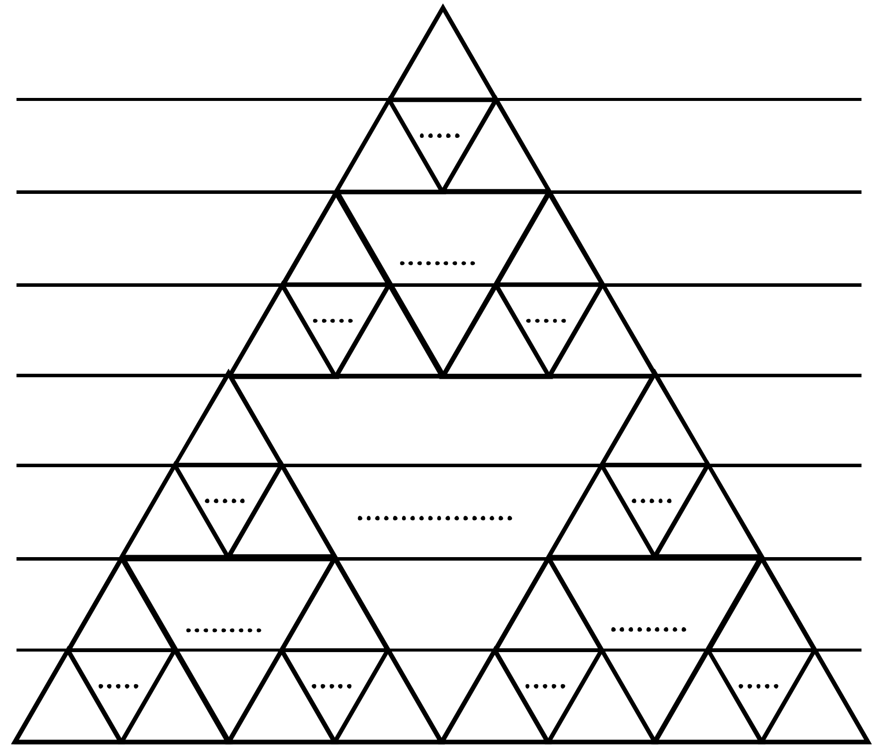
each subtree contains $\Theta\left(2^{\frac{h}{2}}\right) = \Theta\left(\sqrt{n}\right)$ nodes,

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

van Emde Boas Layout



a binary search tree



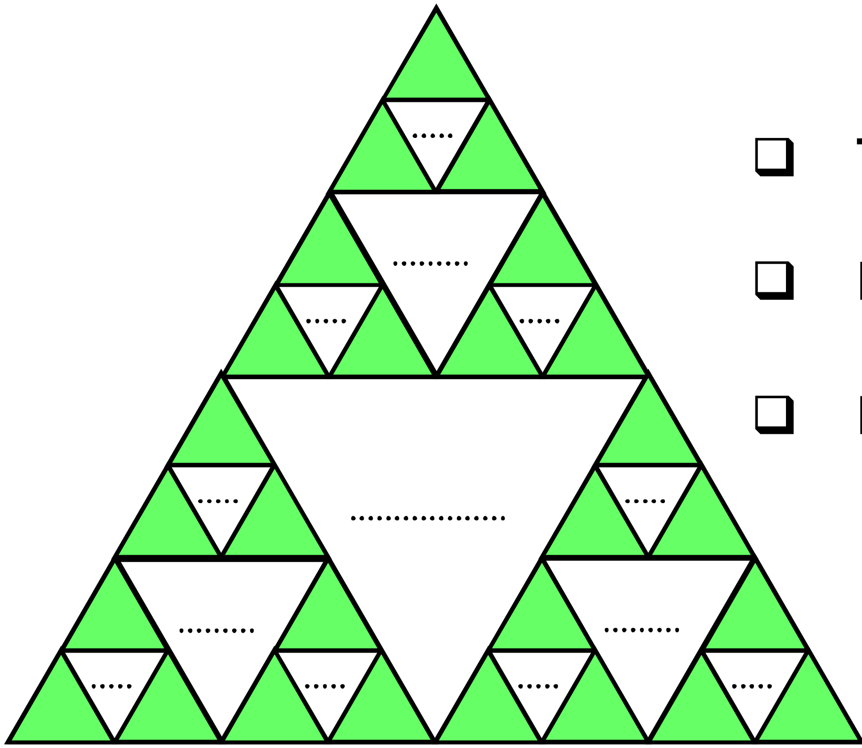
Recursive Subdivision

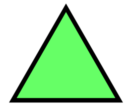
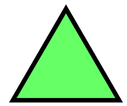
If the tree contains n nodes,

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

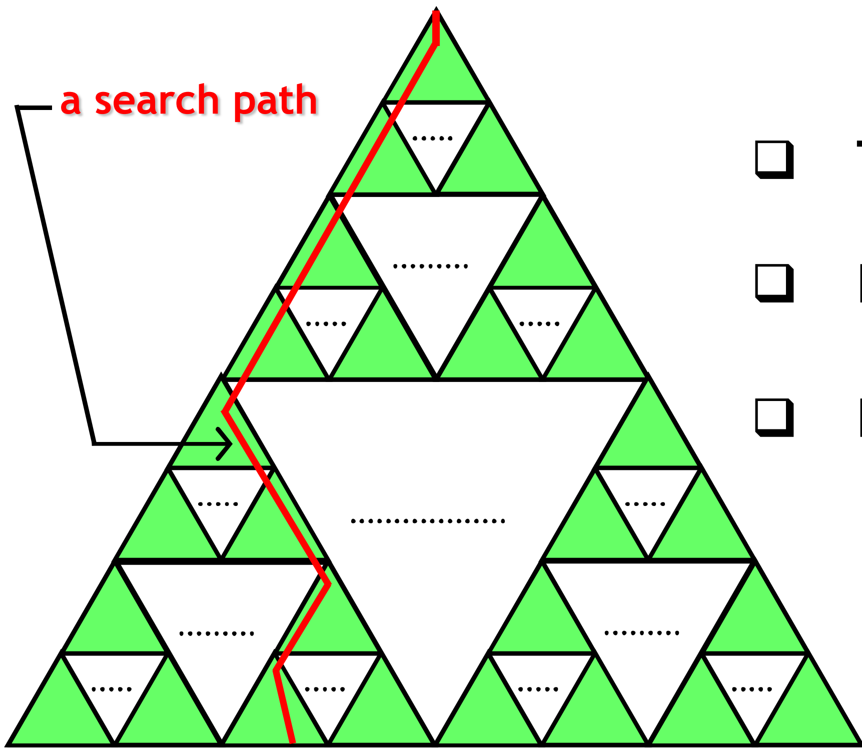
and $k = \Theta\left(\sqrt{n}\right)$

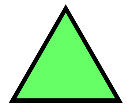
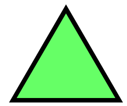
I/O-Complexity of a Search

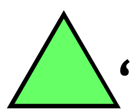


- The height of the tree is $= \log n$
- Each  has height between $\frac{1}{2}\log B$ and $\log B$.
- Each  spans at most 2 blocks of size B.

I/O-Complexity of a Search



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

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

Sorting
(Distribution Sort)

Cache-Complexity of Sorting

Algorithm	Cache-Complexity
Traditional (e.g., mergesort and heapsort)	$\mathrm{O}\left(N \log N\right)$
Cache-Aware (e.g., external-memory versions of mergesort and distribution sort)	$\mathrm{O}\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$
Cache-Oblivious (e.g. funnelsort, cache-oblivious distribution sort and proximity mergesort)	$\mathrm{O}\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$

Cache-Complexity of Sorting

Algorithm	Cache-Complexity
Traditional (e.g., mergesort and heapsort)	$O\left(\frac{N}{B}\log_2 N\right)$
Cache-Aware (e.g., external-memory versions of mergesort and distribution sort)	$\boxed{O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)}$
Cache-Oblivious (e.g. funnelsort, cache-oblivious distribution sort and proximity mergesort)	$\boxed{O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)}$

optimal

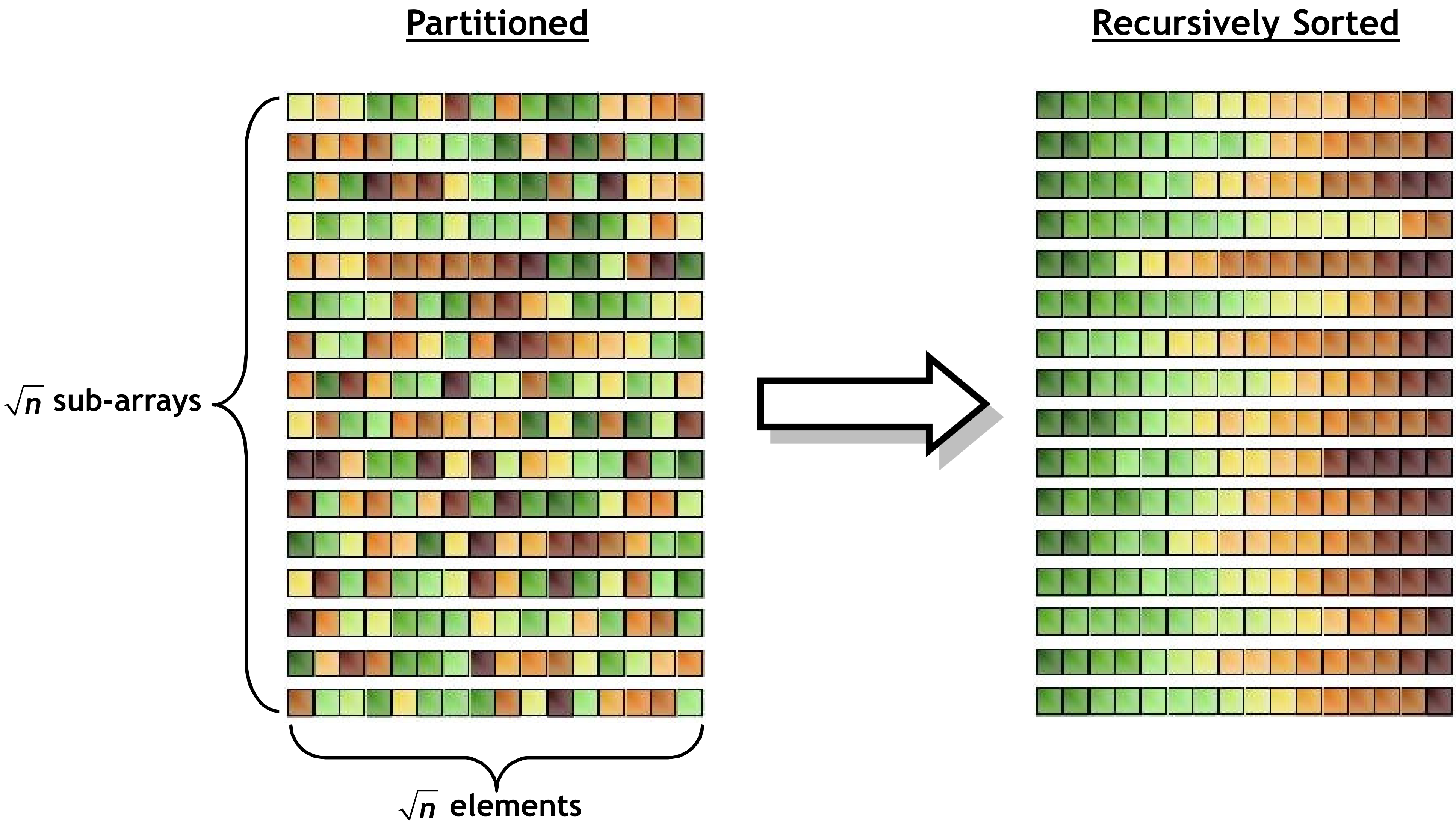
Cache-Oblivious Distribution Sort

Step 1: Partition, and recursively sort partitions.

Step 2: Distribute partitions into buckets.

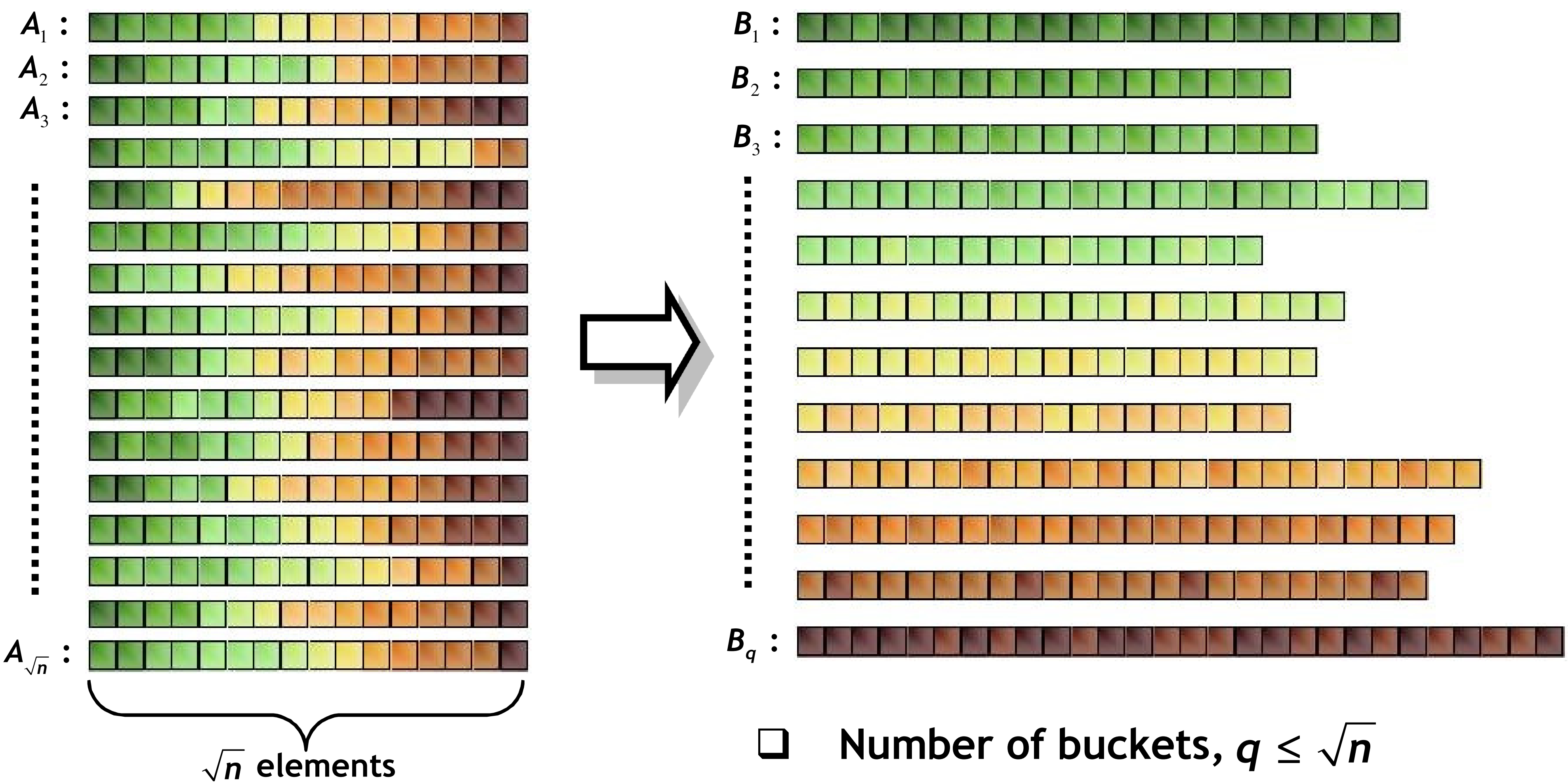
Step 3: Recursively sort buckets.

Step 1: Partition & Recursively Sort Partitions



Step 2: Distribute to Buckets

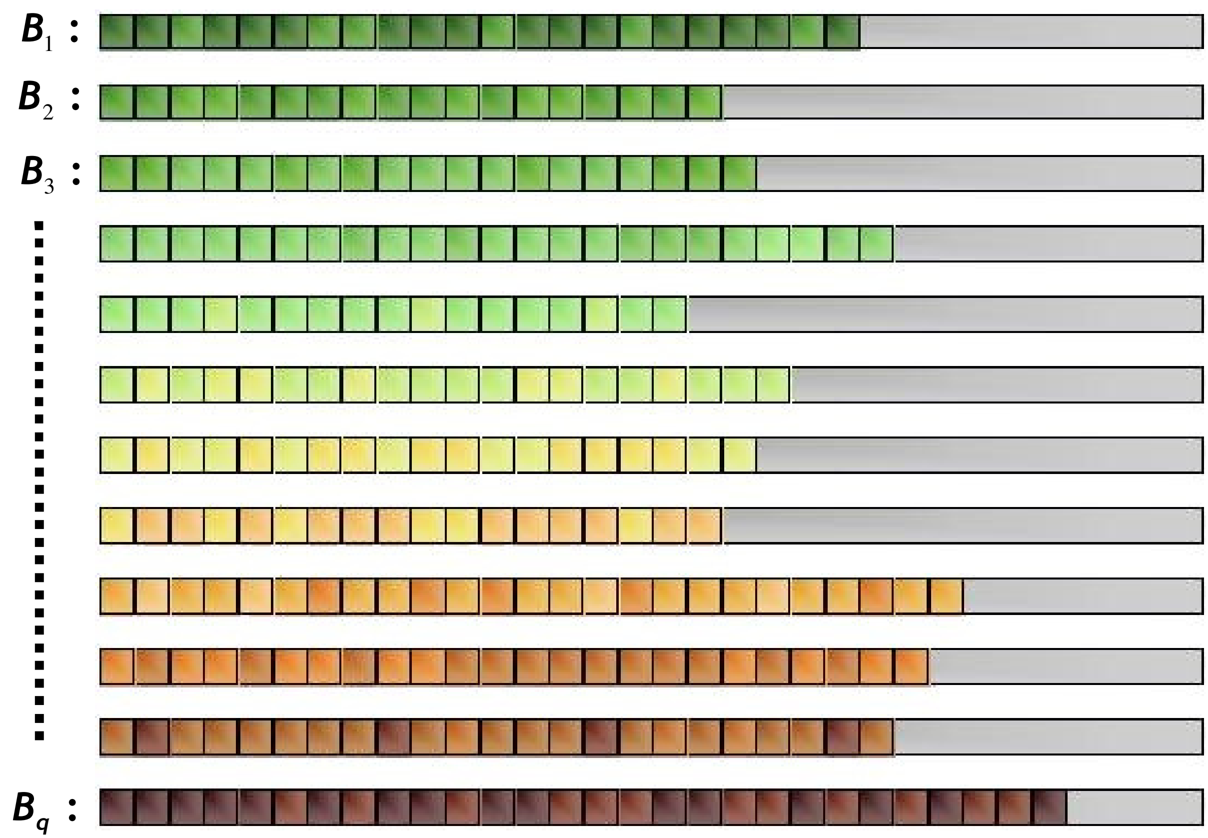
Recursively Sorted | **Distributed to Buckets**



- Number of buckets, $q \le \sqrt{n}$
- Number of elements in $B_i = n_i \le 2\sqrt{n}$
- $\max\{x \mid x \in B_i\} \le \min\{x \mid x \in B_{i+1}\}$

Step 3: Recursively Sort Buckets

Recursively Sort Each Bucket



Done!

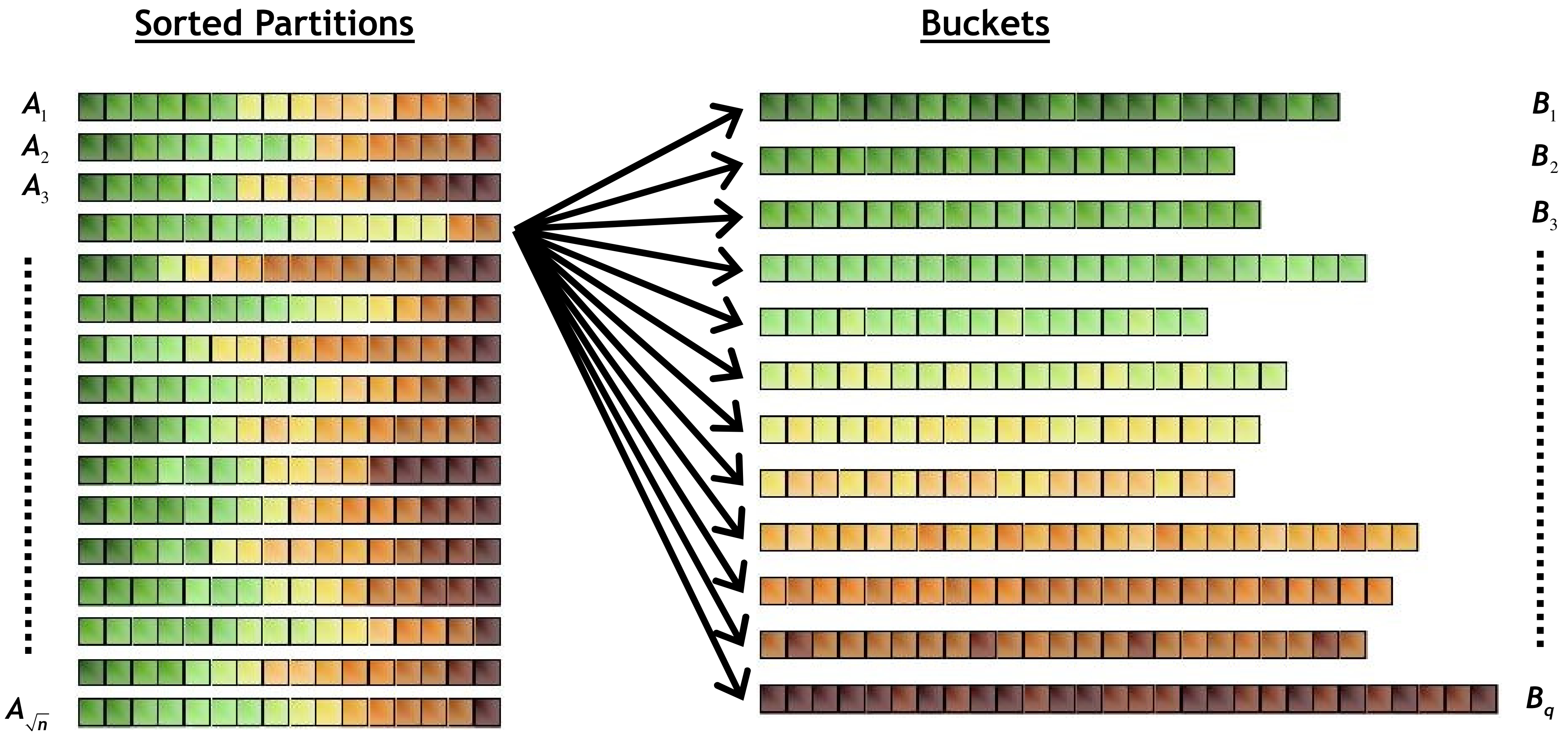
Distribution Sort

Step 1: Partition, and recursively sort partitions.

Step 2: Distribute partitions into buckets.

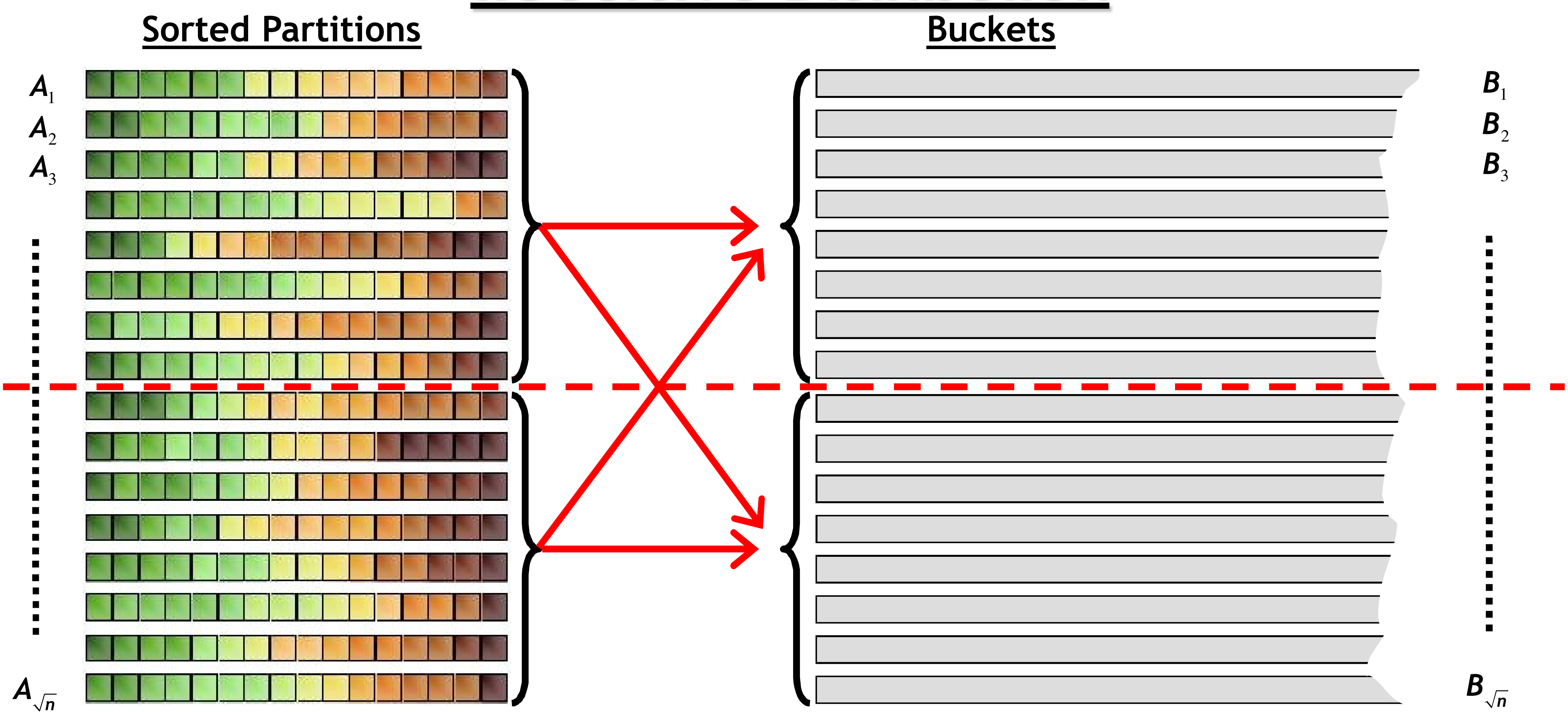
Step 3: Recursively sort buckets.

The Distribution Step



- ❑ We can take the partitions one by one, and distribute all elements of current partition to buckets
- ❑ Has very poor cache performance: $\Theta\left(\sqrt{n} \times \sqrt{n}\right) = \Theta(n)$ I/Os

Recursive Distribution



$[A_i, \ldots, A_{i+m-1}]$

$\Downarrow$

$[B_j, \ldots, B_{j+m-1}]$

Distribute (i, j, m)

1. *if* $m = 1$ *then* copy elements from A_i to B_j
2. *else*
3. *Distribute* (i, j, $m / 2$)
4. *Distribute* ($i + m / 2$, j, $m / 2$)
5. *Distribute* (i, $j + m / 2$, $m / 2$)
6. *Distribute* ($i + m / 2$, $j + m / 2$, $m / 2$)

may need to split B_j to maintain $B_j \le 2\sqrt{n}$

Recursive Distribution

```
Distribute ( i, j, m )
  1. if  m = 1  then copy elements from A_i to B_j
  2. else
  3.      Distribute (         i,          j,    m / 2 )
  4.      Distribute ( i + m / 2,          j,    m / 2 )
  5.      Distribute (         i,  j + m / 2,    m / 2 )
  6.      Distribute ( i + m / 2,  j + m / 2,    m / 2 )
```

ignore
the cost of splits
for the time being

Let $R(m, d)$ denote the cache misses incurred by *Distribute* (*i*, *j*, *m*) that copies *d* elements from *m* partitions to *m* buckets. Then

$$R(m,d) = \begin{cases} O\left(B + \frac{d}{B}\right), & \text{if } m \le \alpha B, \\ \sum_{1 \le i \le 4} R\left(\frac{m}{2}, d_i\right), & \text{otherwise, where } d = \sum_{1 \le i \le 4} d_i \end{cases}$$

$$= O\left(B + \frac{m^2}{B} + \frac{d}{B}\right)$$

$$\therefore R\left(\sqrt{n}, n\right) = O\left(\frac{n}{B}\right)$$

Recursive Distribution

Distribute (i, j, m)

1. *if* $m = 1$ *then* $\boxed{\textit{copy elements from } A_i \textit{ to } B_j}$
2. *else*
3. *Distribute* (i, j, $m / 2$)
4. *Distribute* ($i + m / 2$, j, $m / 2$)
5. *Distribute* (i, $j + m / 2$, $m / 2$)
6. *Distribute* ($i + m / 2$, $j + m / 2$, $m / 2$)

ignore
the cost of splits
for the time being

Recursive Distribution

```
Distribute ( i, j, m )
  1. if  m = 1  then copy elements from A_i to B_j
  2. else
  3.      Distribute (          i,            j,  m / 2 )
  4.      Distribute ( i + m / 2,             j,  m / 2 )
  5.      Distribute (          i,  j + m / 2,  m / 2 )
  6.      Distribute ( i + m / 2,  j + m / 2,  m / 2 )
```

total cache misses incurred by all splits

$$= \sqrt{n} \times O\left(\frac{\sqrt{n}}{B}\right) = O\left(\frac{n}{B}\right)$$

I/O-complexity of *Distribute* (1, 1, $\sqrt{n}$) is

$$= R\left(\sqrt{n}, n\right) + O\left(\frac{n}{B}\right) = O\left(\frac{n}{B}\right)$$

I/O-Complexity of Distribution Sort

Step 1: Partition into $\sqrt{n}$ sub-arrays containing $\sqrt{n}$ elements each and sort the sub-arrays recursively.

Step 2: Distribute sub-arrays into buckets $B_1, B_2, \ldots, B_q$.

Step 3: Recursively sort the buckets.

I/O-complexity of Distribution Sort:

$$Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & \text{if } n \le \alpha' M \\ \sqrt{n}Q\left(\sqrt{n}\right) + \sum_{i=1}^{q} Q(n_i) + O\left(1 + \frac{n}{B}\right), & \text{otherwise} \end{cases}$$

$$= O\left(\frac{n}{B}\log_M n\right), \text{ when } M = \Omega\left(B^2\right)$$