Cache-Oblivious Algorithms

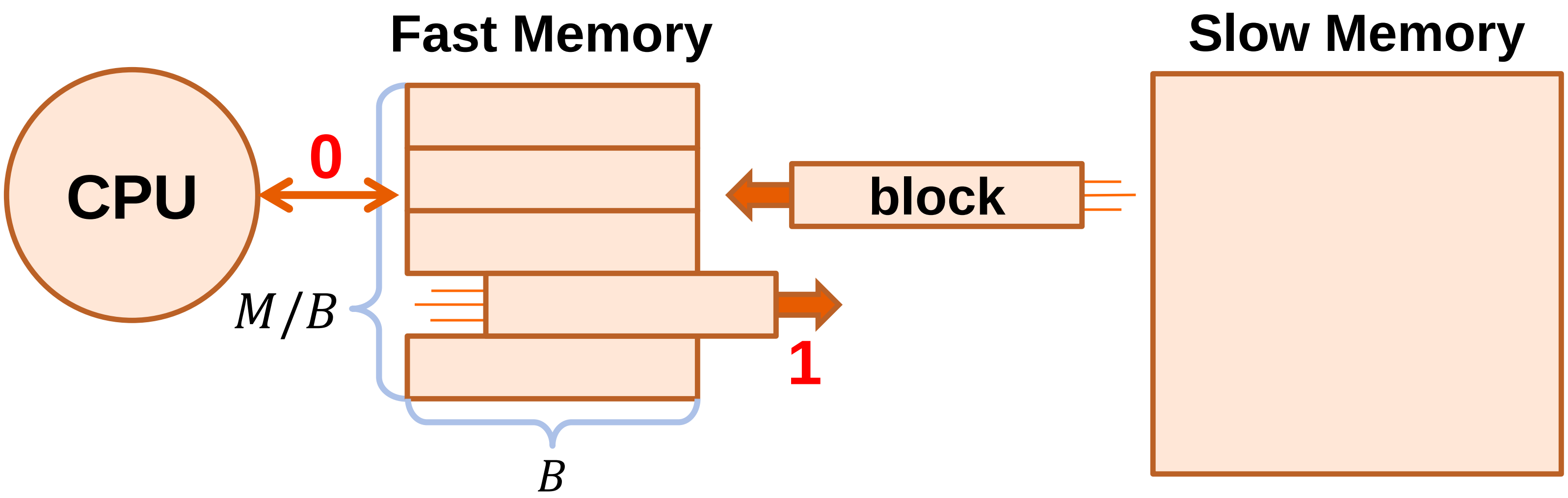
- Matrix Transpose
- Matrix Multiplication
- Binary Search
- Sorting

15-853: Algorithms in the Real World

I/O model (External-memory model)

Abstracts a single level of the memory hierarchy

- Fast memory (cache) of size M
- Accessing fast memory is free, but moving data from slow memory is expensive
- Memory is grouped into size-B ***blocks*** of contiguous data

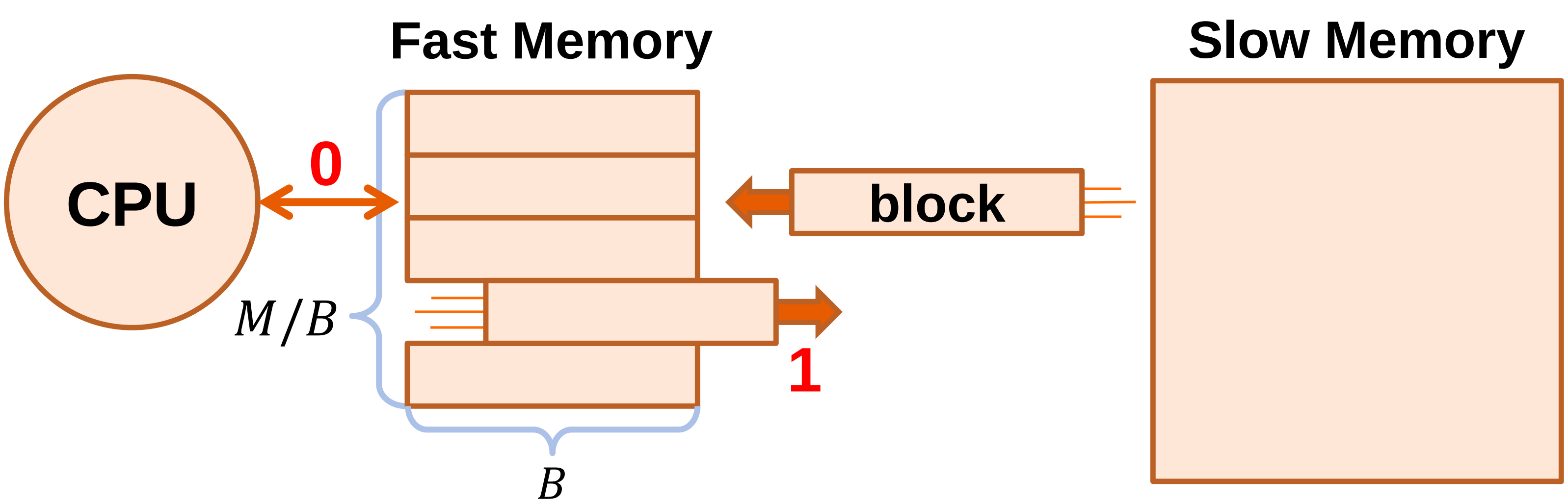


Cost: the number of **block transfers** (or **I/O**s) from slow memory to fast memory

I/O model (External-memory model)

Challenges:

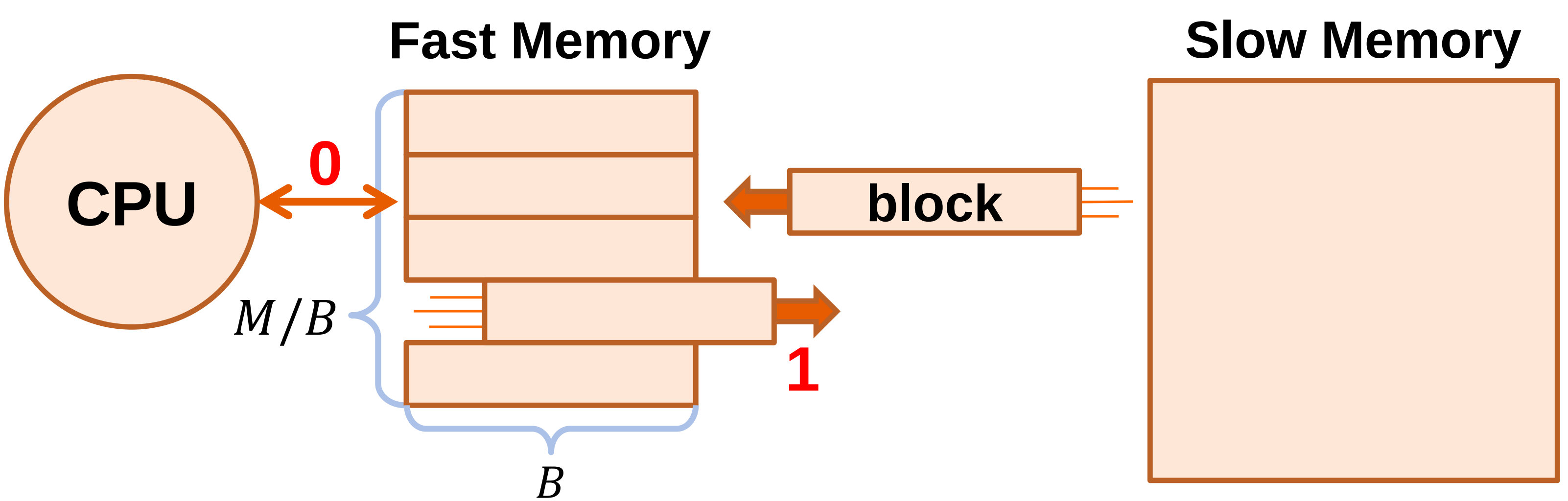
- Assume explicit control of the cache and main memory
- The algorithms are based on M and B



Cost: the number of **block transfers** (or **I/O**s) from slow memory to fast memory

Cache-Oblivious Algorithms

- Algorithms not parameterized by *B* or *M*
  - These algorithms are unaware of the parameters of the memory hierarchy
- Analyze in the ***ideal cache*** model — same as the I/O model except optimal replacement is assumed



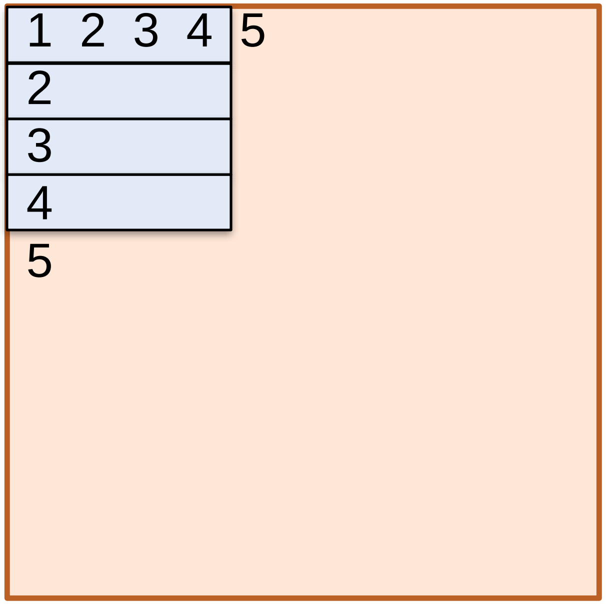
Question: how do I know or analyze based on ideal cache-replacement policy?

Cache-Oblivious Algorithms

- Algorithms not parameterized by B or M.
  - These algorithms are unaware of the parameters of the memory hierarchy
- Analyze in the ***ideal cache*** model — same as the I/O model except optimal replacement is assumed

- Use a specific cache sequence to analyze the I/O cost
- An ideal cache will do no worse than this specific load/evict sequence
- In practice, real caches based on LRU policy perform similarly to the optimal cache

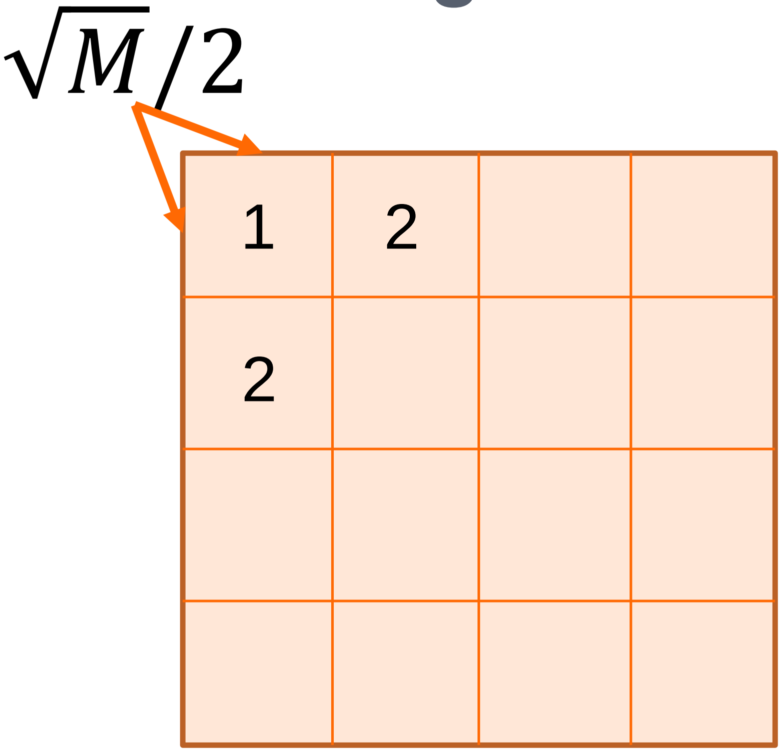
Toy example: matrix transpose




```
for i = 1 to N do
    for j = i+1 to N do
        swap(A[i][j], A[j][i])
```

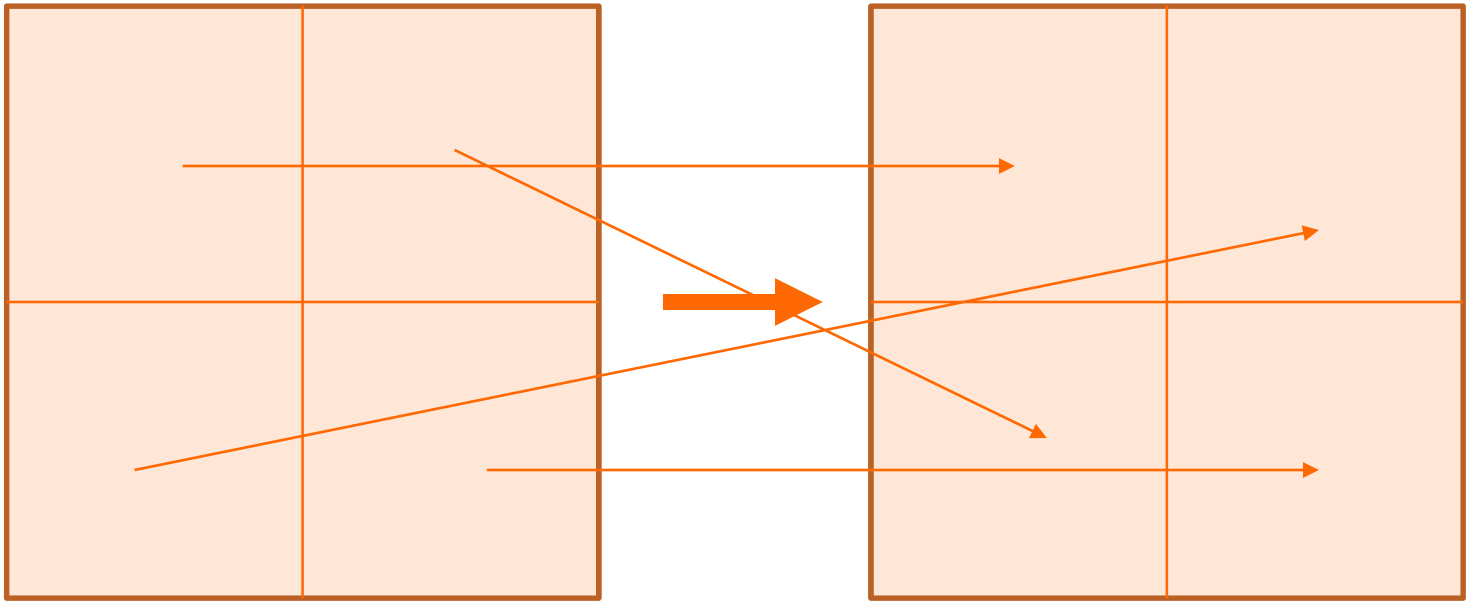
- The simplest implementation is not I/O efficient assuming the matrix is stored in row-major

Toy example: matrix transpose —— I/O algorithm



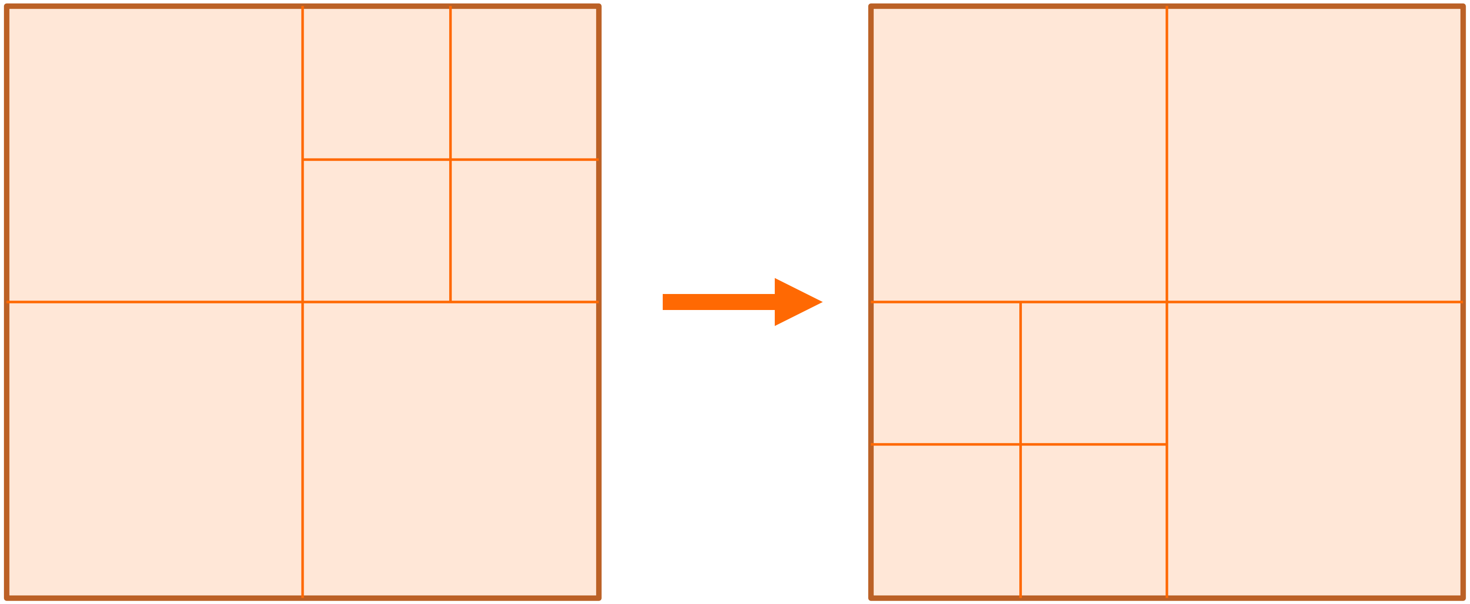
- The simplest implementation is not I/O efficient
- The I/O algorithm has a cost of $\Theta(n^2/B)$

Toy example: matrix transpose —— Cache-oblivious algorithm



- The simplest implementation is not I/O efficient
- The I/O algorithm has a cost of $\Theta(n^2/B)$

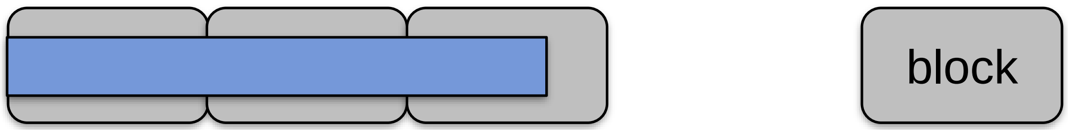
Toy example: matrix transpose —— Cache-oblivious algorithm



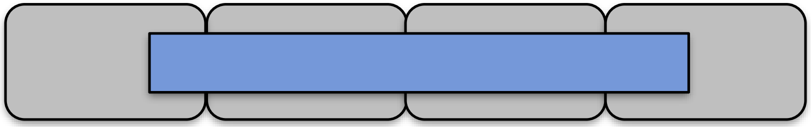
- The simplest implementation is not I/O efficient
- The I/O algorithm has a cost of $\Theta(n^2/B)$

Array storage

- How many blocks does a size-N array occupy?
  - If it's aligned on a block (usually true for cache-aware), it takes exactly $\lceil N/B \rceil$ blocks

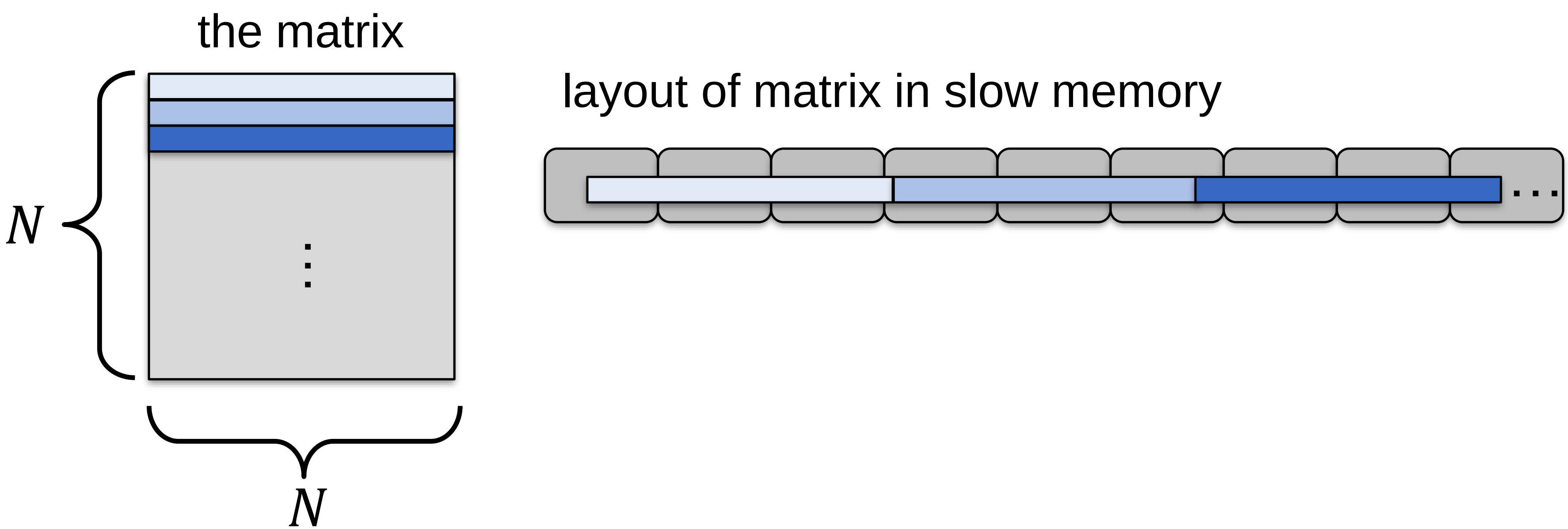


  - If you're unlucky, it's $\lceil N/B \rceil + 1$ blocks. This is generally what you need to assume for cache-oblivious algorithms as you can't force alignment



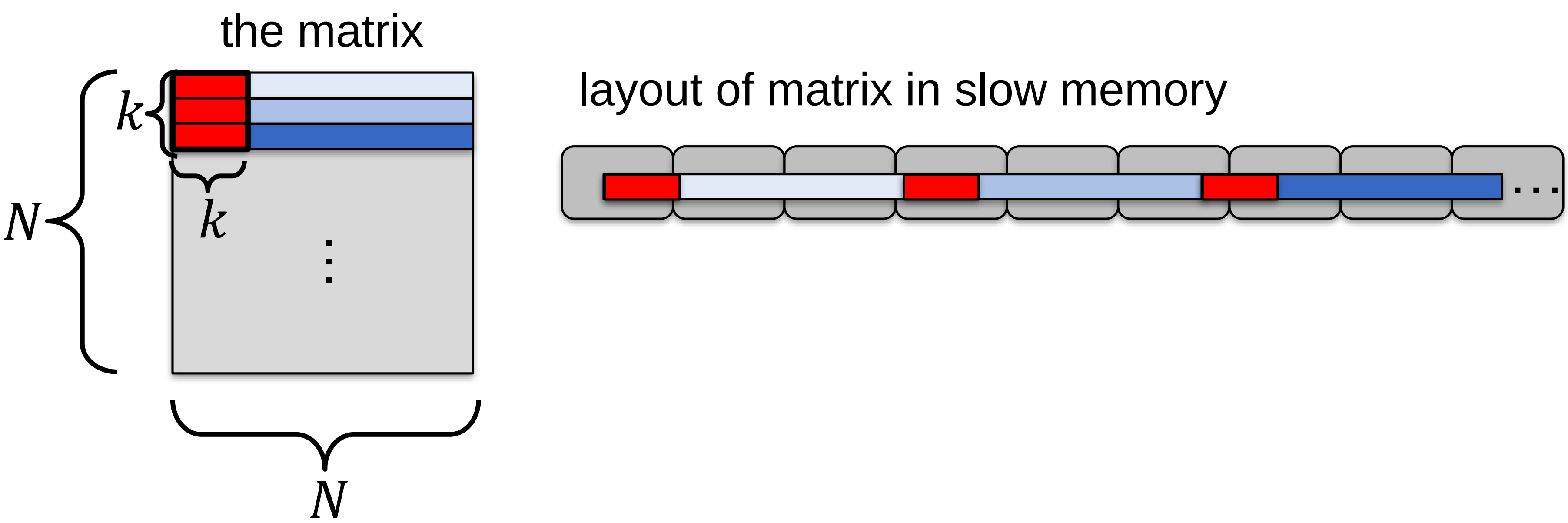
A size-N array occupies at most $\lceil N/B \rceil + 1 = \Theta(1 + N/B)$ blocks, since you cannot control cache alignment

Toy example: matrix transpose —— Cache-oblivious algorithm



A size-N array occupies at most $\lceil N/B \rceil + 1 = \Theta(1 + N/B)$ blocks, since you cannot control cache alignment

Toy example: matrix transpose —— Cache-oblivious algorithm



The total number of blocks to store submatrix is $k(\lceil k/B \rceil + 1) = \Theta(k + k^2/B)$ blocks

Toy example: matrix transpose —— Cache-oblivious algorithm

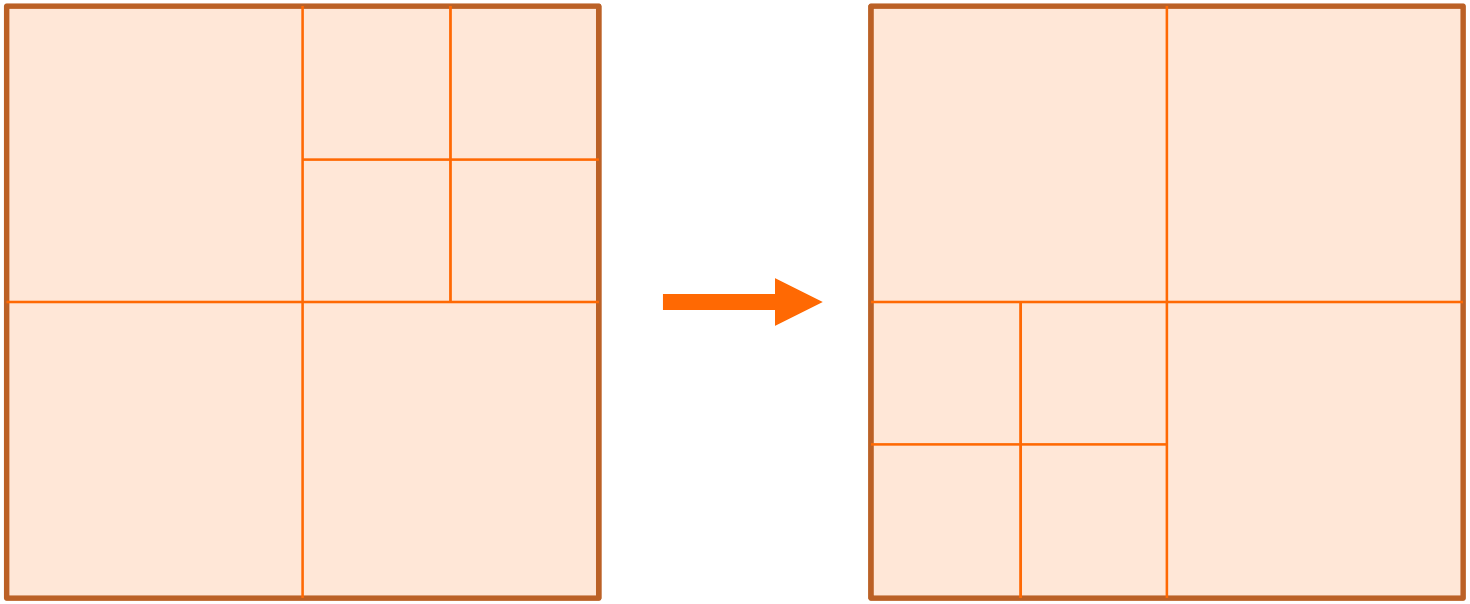
Want: $k(\lceil k/B \rceil + 1) \leq M/B$

If assuming $M \geq B^2$, then $k \leq c\sqrt{M}$ for some c like ¼

This is called the **Tall-Cache Assumption**

The total number of blocks to store submatrix is $k(\lceil k/B \rceil + 1) = \Theta(k + k^2/B)$ blocks

Toy example: matrix transpose —— Cache-oblivious algorithm



- The simplest implementation is not I/O efficient
- The I/O algorithm has a cost of $\Theta(n^2/B)$
- The cache-oblivious algorithm also has a cost of $\Theta(n^2/B)$

Cache-Oblivious Algorithms

- Algorithms not parameterized by B or M
  - These algorithms are unaware of the parameters of the memory hierarchy
- Analyze in the ***ideal cache*** model — same as the I/O model except optimal replacement is assumed

- Use a specific cache sequence to analyze the I/O cost (similar to the I/O algorithms)
- An ideal cache will do no worse than this specific load/evict sequence
- In practice, real caches based on LRU policy perform similarly to the optimal cache

Advantages of Cache-Oblivious Algorithms

- Since CO algorithms do not depend on memory parameters, bounds generalize to multilevel hierarchies

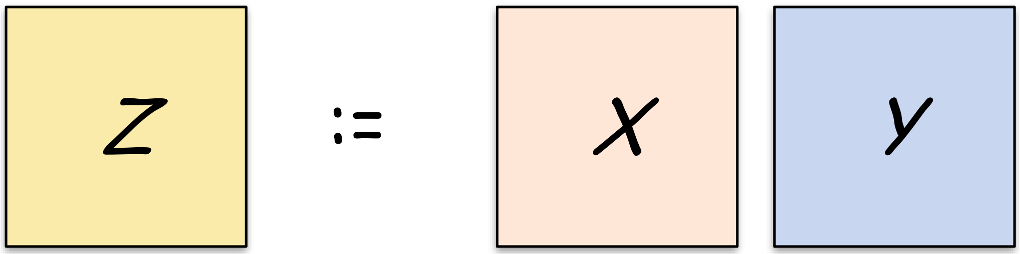
- Algorithms are platform independent

- Algorithms should be effective even when B and M are not static

- In this specific case and many other algorithms in this lecture, CO algorithms are based on divide-and-conquer, which can be parallelized naturally

Matrix Multiplication

Consider standard iterative matrix-multiplication algorithm



- Where X, Y, and Z are $N \times N$ matrices

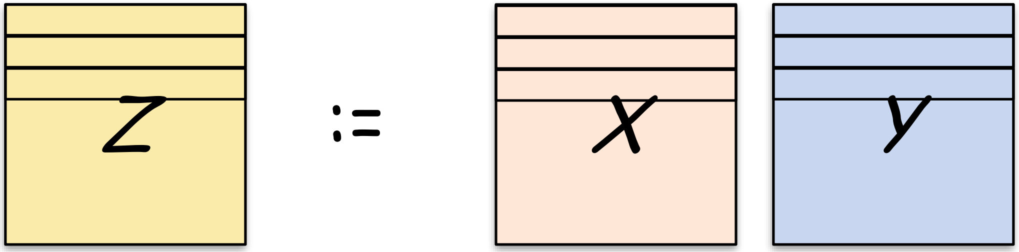
```
for i = 1 to N do
    for j = 1 to N do
        for k = 1 to N do
            Z[i][j] += X[i][k] * Y[k][j]
```

- $\Theta(N^3)$ computation in RAM model. What about I/O?

How Are Matrices Stored?

How data is arranged in memory affects I/O performance

- Suppose *X*, *Y*, and *Z* are in row-major order




```
for i = 1 to N do
   for j = 1 to N do
      for k = 1 to N do
         Z[i][j] += X[i][k] * Y[k][j]
```

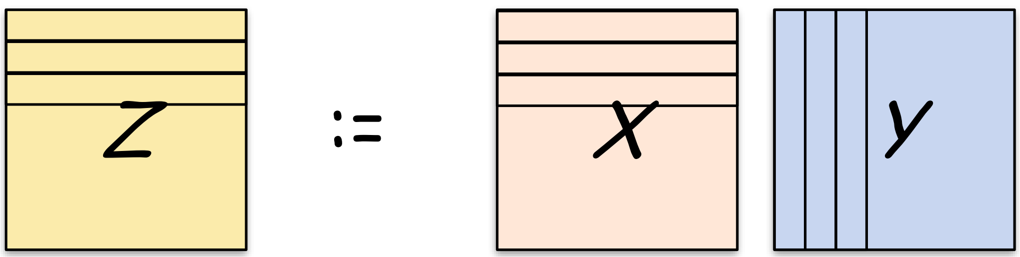
If $N \geq B$, reading a column of *Y* is expensive ⇒ $\Theta(N)$ I/Os

If N is larger than M, no locality across iterations for *X* and *Y* ⇒ $\Theta(N^3)$ I/Os

How Are Matrices Stored?

Suppose X and Z are in row-major order but Y is in column-major order

- Not too inconvenient. Transposing Y has $\Theta(N^2/B)$ cost




```
for i = 1 to N do
    for j = 1 to N do
        for k = 1 to N do
            Z[i][j] += X[i][k] * Y[k][j]
```

Scan row of X and column of Y $\Rightarrow \Theta(N/B)$ I/Os

No locality across iterations for Y $\Rightarrow \Theta(N^3/B)$ I/Os

We can do much better than $\Theta(N^3/B)$ I/Os, even if all matrices are row-major

Recursive Matrix Multiplication

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

Compute 8 submatrix products recursively

$$Z_{11} := X_{11}Y_{11} + X_{12}Y_{21}$$
$$Z_{12} := X_{11}Y_{12} + X_{12}Y_{22}$$
$$Z_{21} := X_{21}Y_{11} + X_{22}Y_{21}$$
$$Z_{22} := X_{21}Y_{12} + X_{22}Y_{21}$$

- When all three submatrices X', Y' and Z' fit into cache, the algorithm loads them all from the slow memory, apply the multiply, and update Z' back to the main memory, with the cost of $\Theta(M/B)$

Recursive Matrix Multiplication

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

Compute 8 submatrix products recursively

$Z_{11} := X_{11}Y_{11} + X_{12}Y_{21}$
$Z_{12} := X_{11}Y_{12} + X_{12}Y_{22}$
$Z_{21} := X_{21}Y_{11} + X_{22}Y_{21}$
$Z_{22} := X_{21}Y_{12} + X_{22}Y_{21}$

$$Q_{MM}(n) = 8Q_{MM}(n/2)$$

Base case:

$$Q_{MM}\left(\sqrt{M}\right) = \Theta(M/B)$$

Solving it gives $Q_{MM}(n) = \Theta\left(n^3/B\sqrt{M} + n^2/B\right)$

Recursive Matrix Multiplication

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

Compute 8 submatrix products recursively

$Z_{11} := X_{11}Y_{11} + X_{12}Y_{21}$
$Z_{12} := X_{11}Y_{12} + X_{12}Y_{22}$
$Z_{21} := X_{21}Y_{11} + X_{22}Y_{21}$
$Z_{22} := X_{21}Y_{12} + X_{22}Y_{21}$

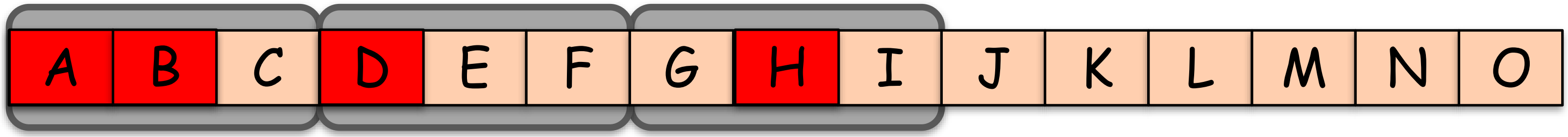
Solving it gives $Q_{MM}(n) = \Theta\left(n^3/B\sqrt{M} + n^2/B\right)$

With a careful implementation, the parallel depth can be $O(\log^2 n)$

Algorithms Similar to Matrix Multiplication

- Linear algebra:
  - Gaussian elimination, LU decomposition, triangular system solver, QR decomposition
- Dynamic Programming:
  - Floyd-Warshall for APSP, edit distance (longest common sequences), 1D clustering, and many applications in molecular biology and geology (e.g. protein folding/ editing, many structure problems)

Searching: binary search is bad



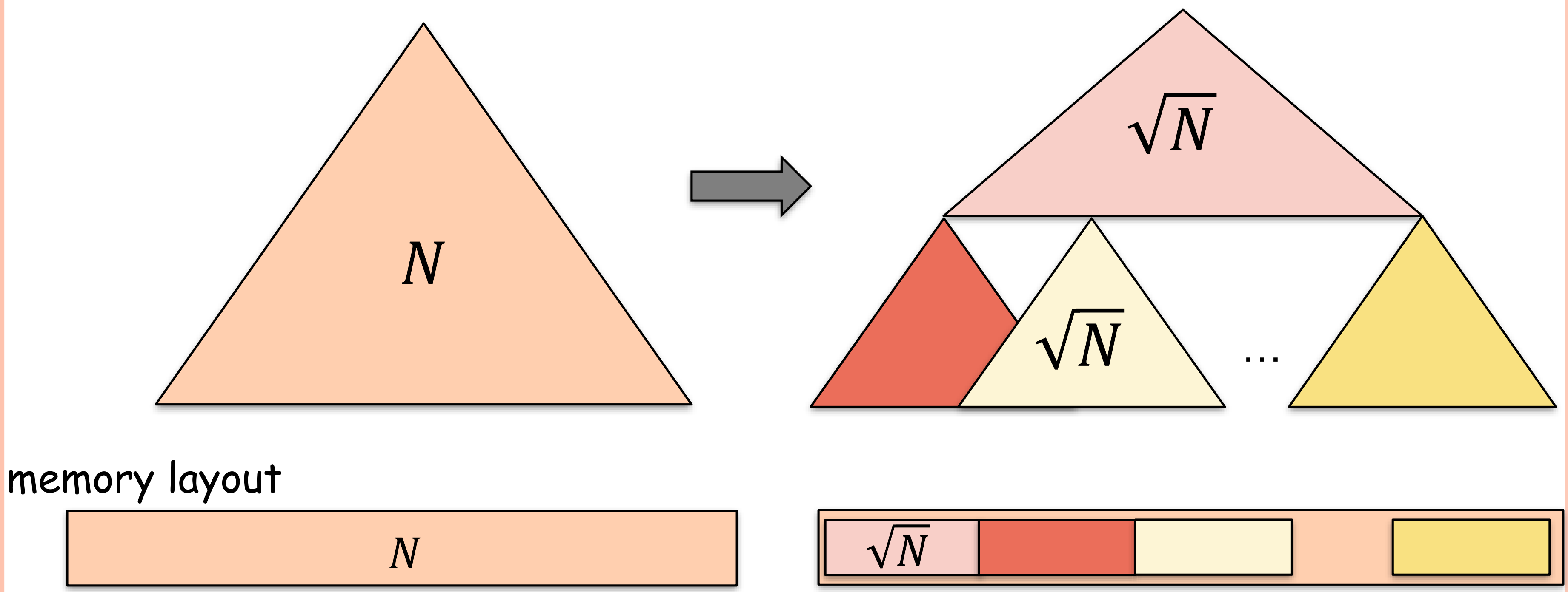
Example: binary search for element A
with block size $B = 3$

- Search hits a a different block until reducing keyspace to size $\Theta(B)$
- Thus, total cost is $\log_2 N - \Theta(\log_2 B) = \Theta(\log_2(N/B)) \approx \Theta(\log_2 N)$ for $N \gg B$

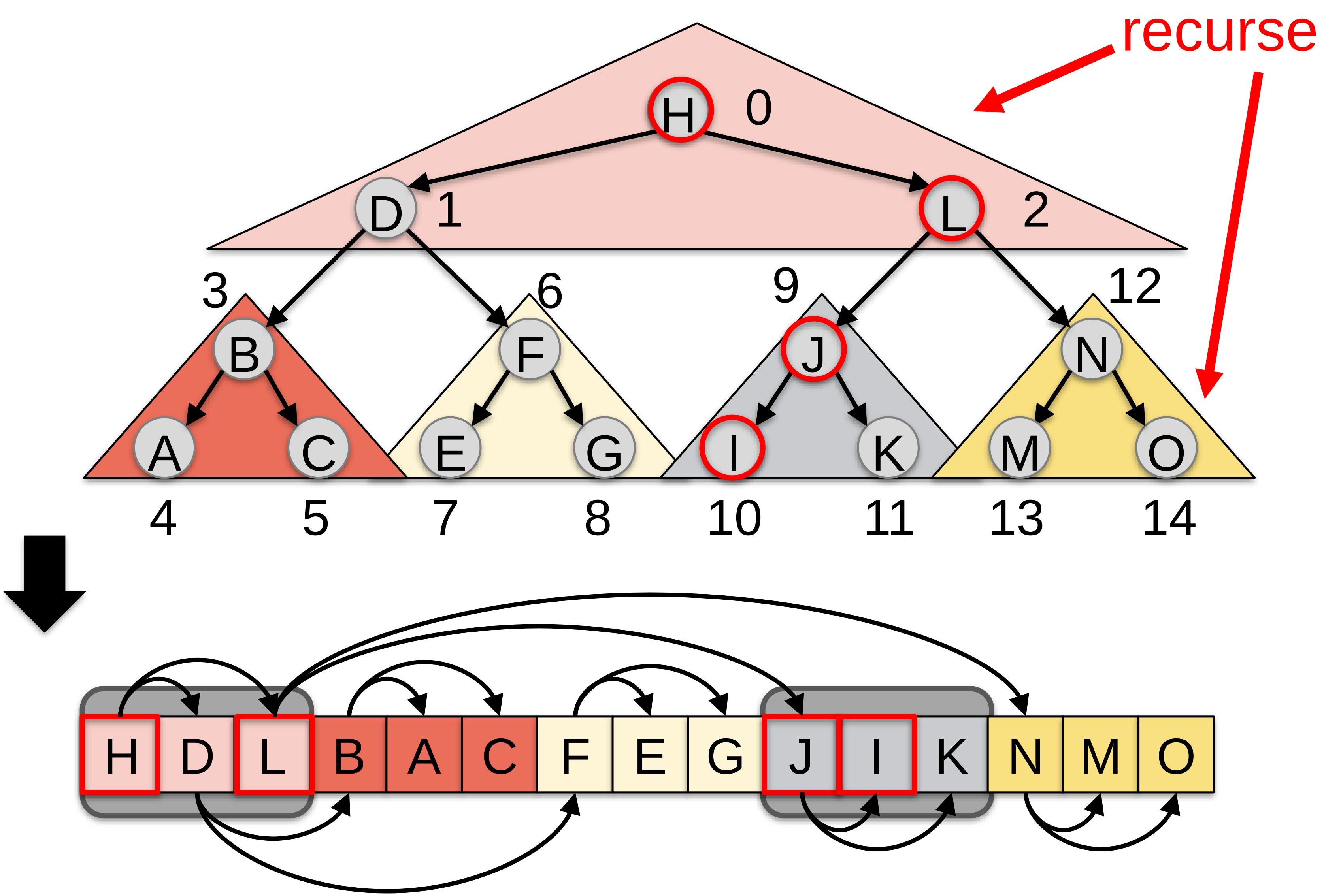
Static cache-oblivious searching

Goal: organize N keys in memory to facilitate efficient searching. (van Emde Boas layout)

1. build a balanced binary tree on the keys
2. layout the tree recursively in memory, splitting the tree at half the height

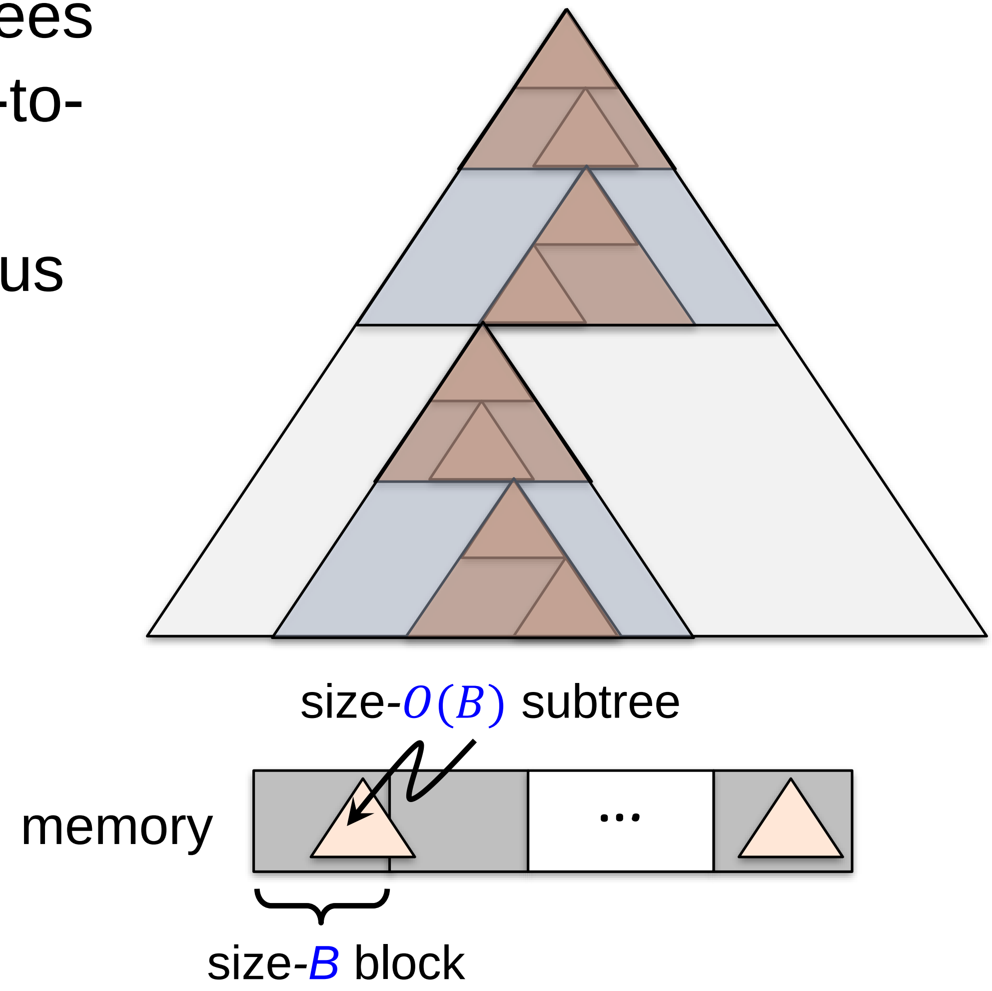


Static layout example



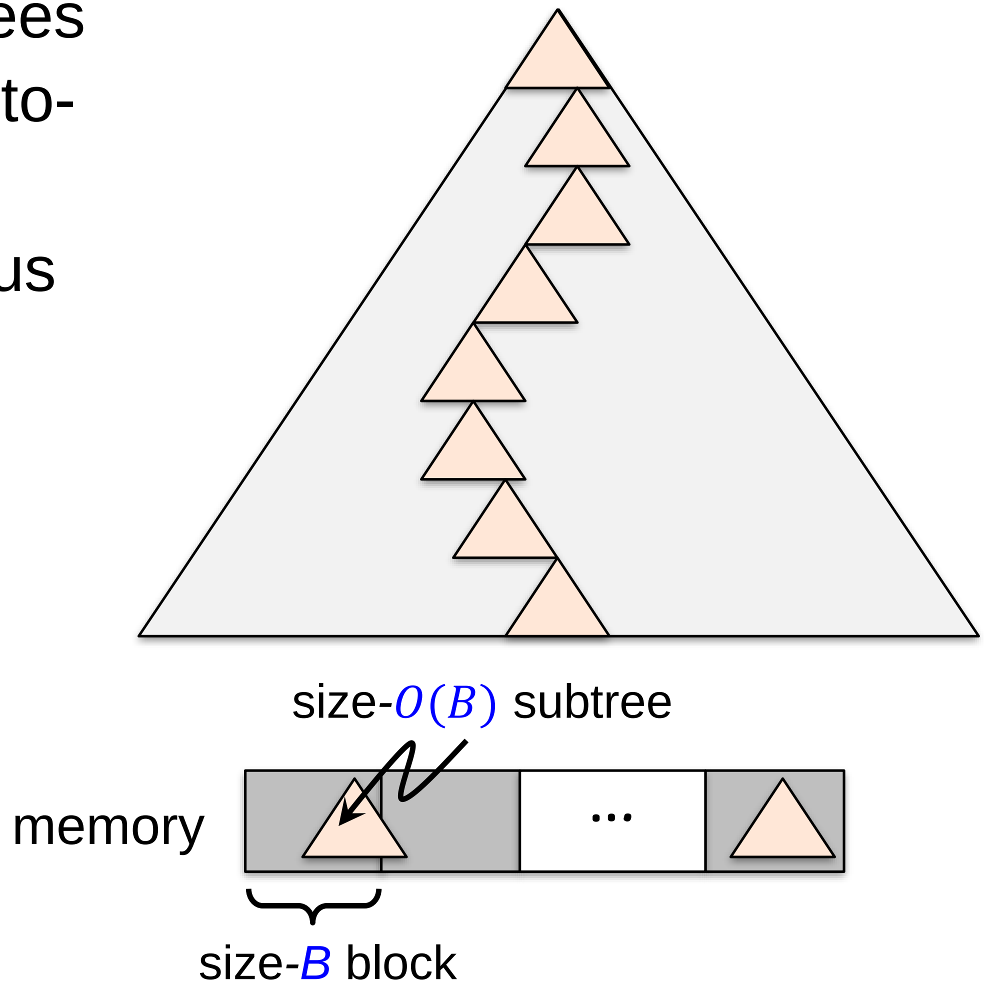
Cache-oblivious searching: Analysis I

- Consider recursive subtrees of size $\sqrt{B}$ to B on a root-to-leaf search path.
- Each subtree is contiguous and fits in $O(1)$ blocks.
- Each subtree has height $\Theta(\log_2 B)$, so there are $\Theta(\log_B N)$ of them.

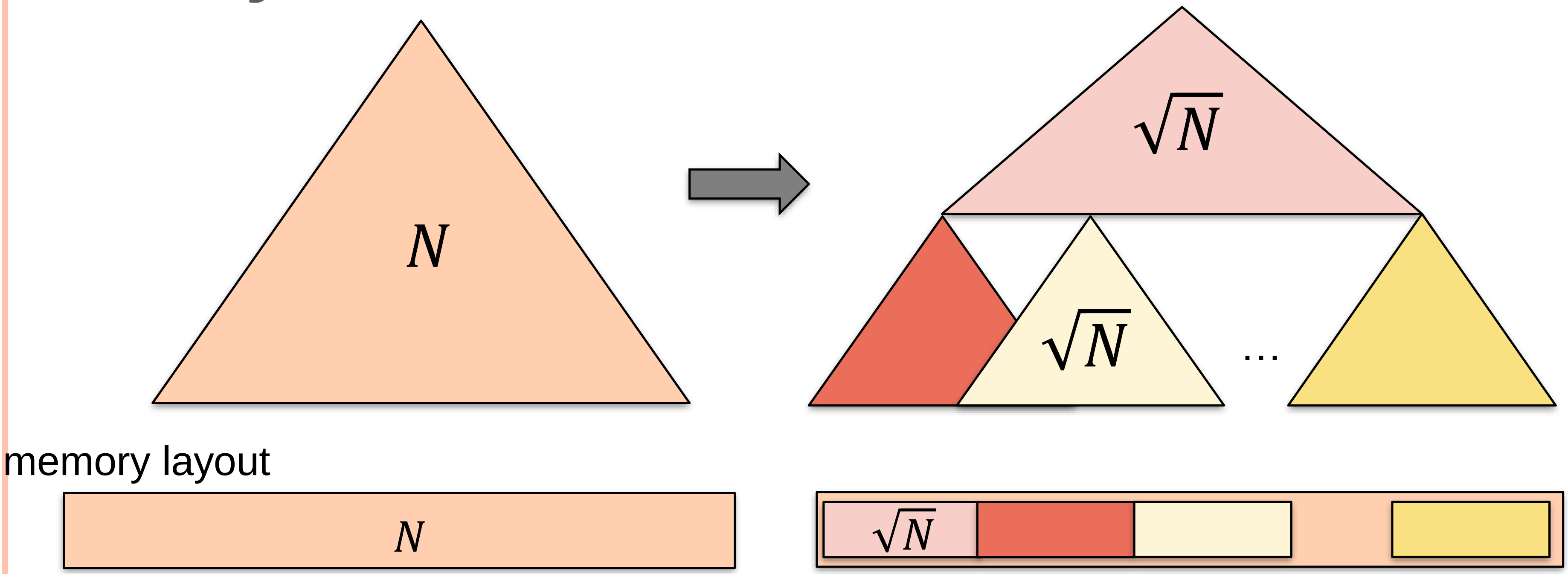


Cache-oblivious searching: Analysis I

- Consider recursive subtrees of size $\sqrt{B}$ to B on a root-to-leaf search path.
- Each subtree is contiguous and fits in $O(1)$ blocks.
- Each subtree has height $\Theta(\log_2 B)$, so there are $\Theta(\log_B N)$ of them.



Cache-oblivious searching: Analysis II



Analyze using a recurrence

$$S(N) = 2S(\sqrt{N})$$

Base case: $S(B) = 1$

Solves to $O(\log_B N)$

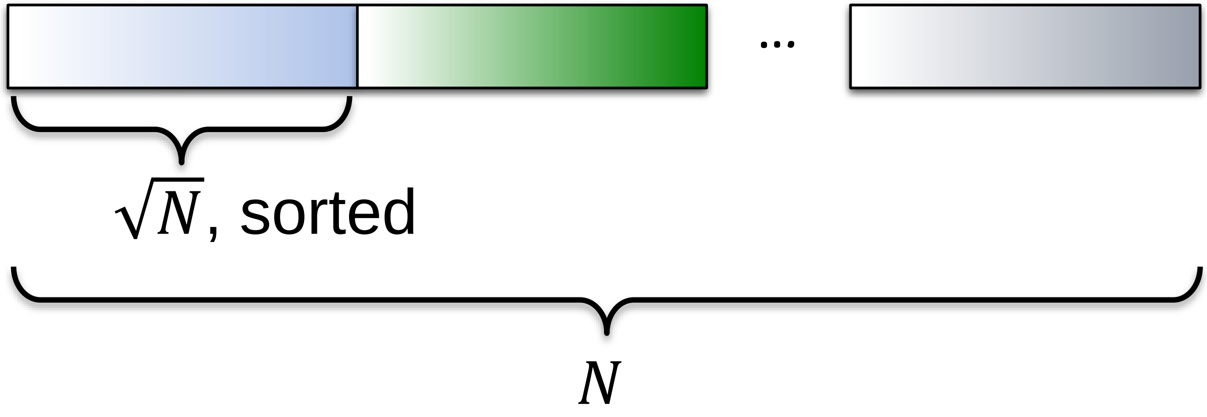
Summary for Cache-Oblivious searching

- Use a special (van Emde Boas) layout to reduce the memory footprint from $O(\log_2 n)$ blocks to $O(\log_B n)$ blocks
- The similar idea can be applied to more complicated scenarios like the dynamic setting, and such a search data structure can be used to implement other algorithms

Sample-sort outline

Analogous to multiway quicksort

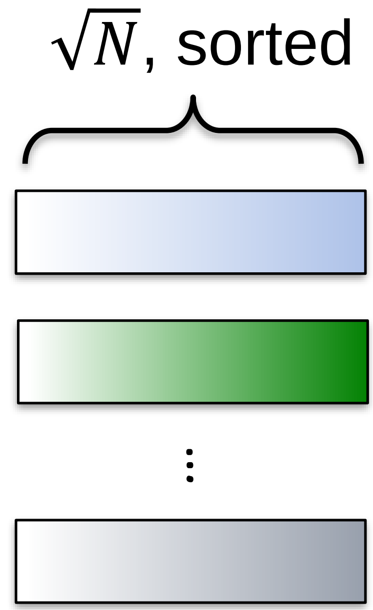
1. Split input array into $\sqrt{N}$ contiguous ***subarrays*** of size $\sqrt{N}$. Sort subarrays recursively



Sample-sort outline

Analogous to multiway quicksort

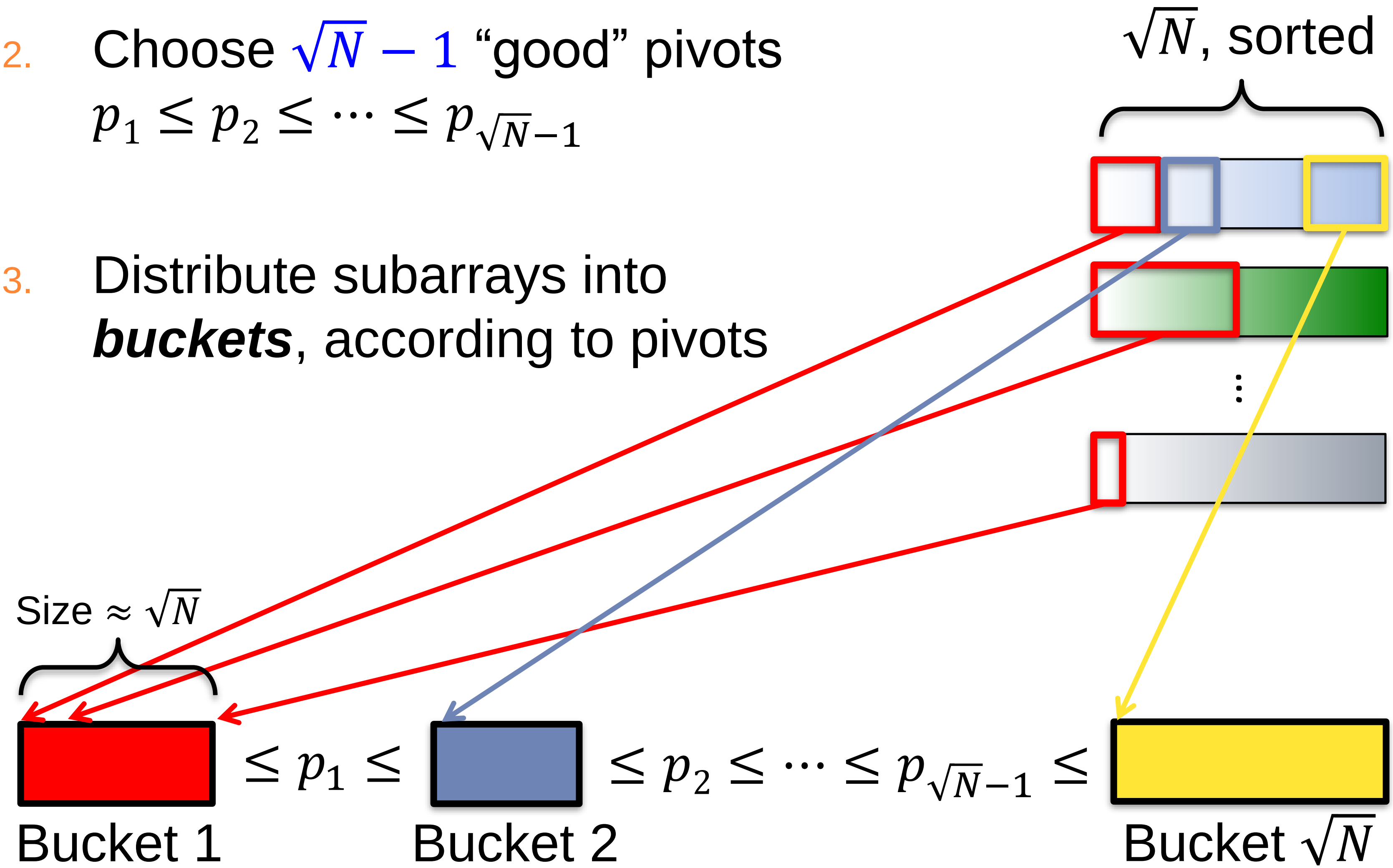
1. Split input array into $\sqrt{N}$ contiguous ***subarrays*** of size $\sqrt{N}$. Sort subarrays recursively



Sample-sort outline

2. Choose $\sqrt{N}-1$ "good" pivots
   $p_1 \le p_2 \le \cdots \le p_{\sqrt{N}-1}$

3. Distribute subarrays into ***buckets***, according to pivots



Sample-sort outline

4. Recursively sort the buckets

$$\text{Bucket } 1 \le p_1 \le \text{Bucket } 2 \le p_2 \le \cdots \le p_{\sqrt{N}-1} \le \text{Bucket } \sqrt{N}$$

Bucket 1

Bucket 2

Bucket $\sqrt{N}$

5. Copy concatenated buckets back to input array

sorted

Sample-sort analysis sketch

- Step 1 (implicitly) divides array and sorts $\sqrt{N}$ size-$\sqrt{N}$ subproblems
- Step 4 sorts $\sqrt{N}$ buckets of size $n_i \approx \sqrt{N}$, with total size N
- Claim: Step 2, 3 and 5 uses $\Theta(N/B)$ work

$$
\begin{aligned}
Q(N) &= \sqrt{N} \cdot Q\left(\sqrt{N}\right) + \sum Q(n_i) + \Theta(N/B) \\
&\approx 2\sqrt{N} \cdot Q\left(\sqrt{N}\right) + \Theta(N/B) \\
&= \Theta\left(\frac{N}{B} \log_M N\right)
\end{aligned}
$$

Sample-sort outline

2. Choose $\sqrt{N}-1$ "good" pivots $p_1 \le p_2 \le \cdots \le p_{\sqrt{N}-1}$

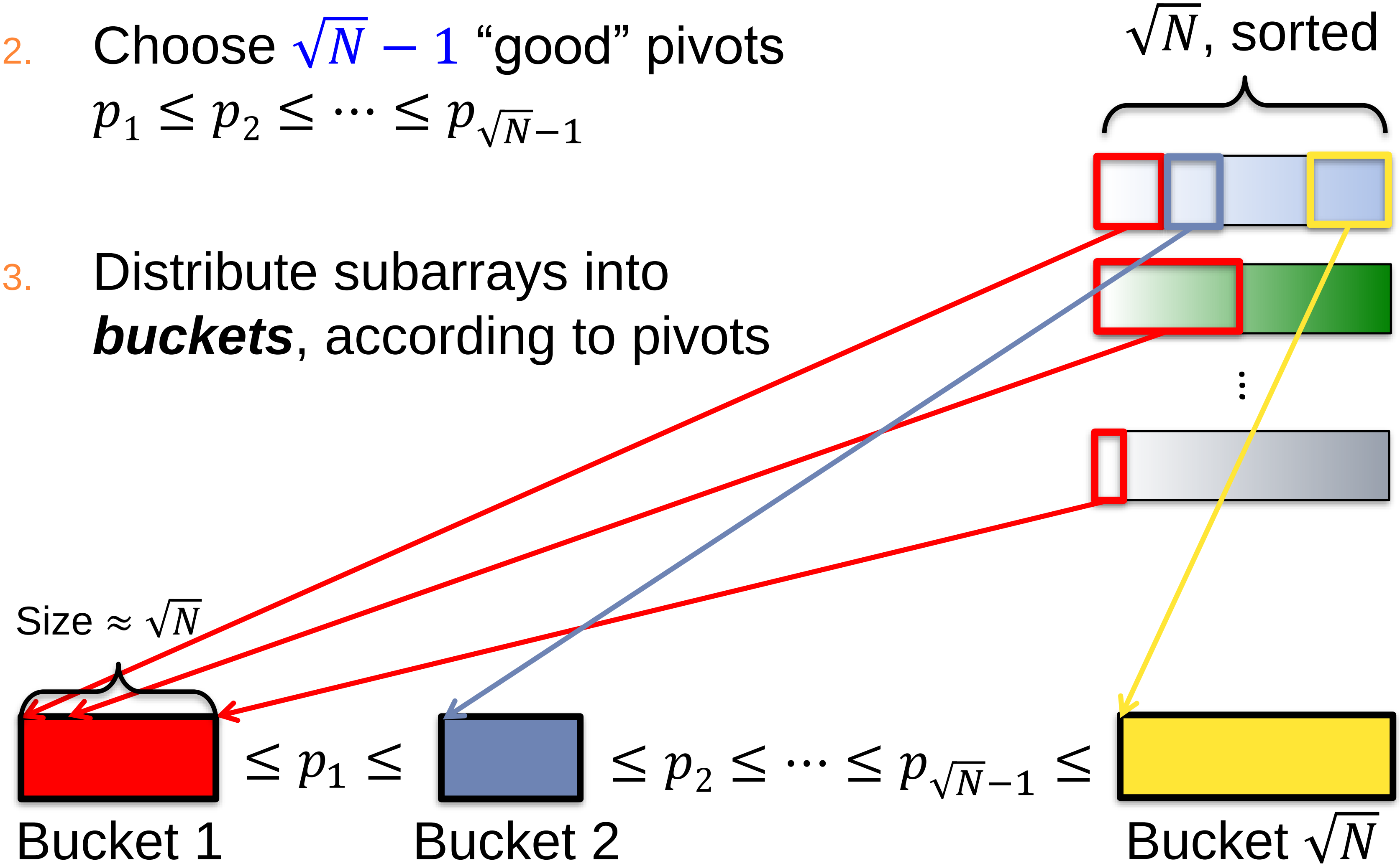
Can be achieved by randomly pick $c\sqrt{N}\log N$ random samples, sort them and pick the every $(c\log N)$-th element

This step requires $O(N/B)$ operations

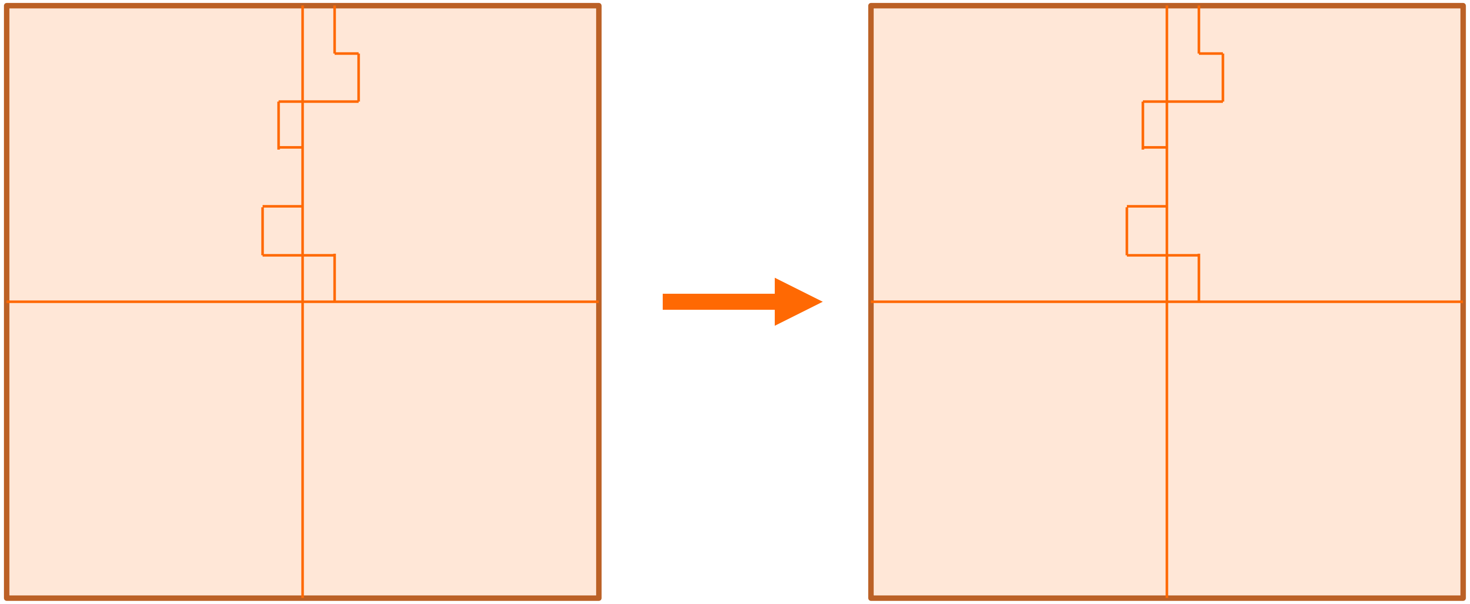
Sample-sort outline

2. Choose $\sqrt{N}-1$ "good" pivots $p_1 \le p_2 \le \cdots \le p_{\sqrt{N}-1}$

3. Distribute subarrays into ***buckets***, according to pivots



Transposing elements to buckets



Given subarrays $s_1, \dots, s_k$ and buckets $b_1, \dots, b_k$

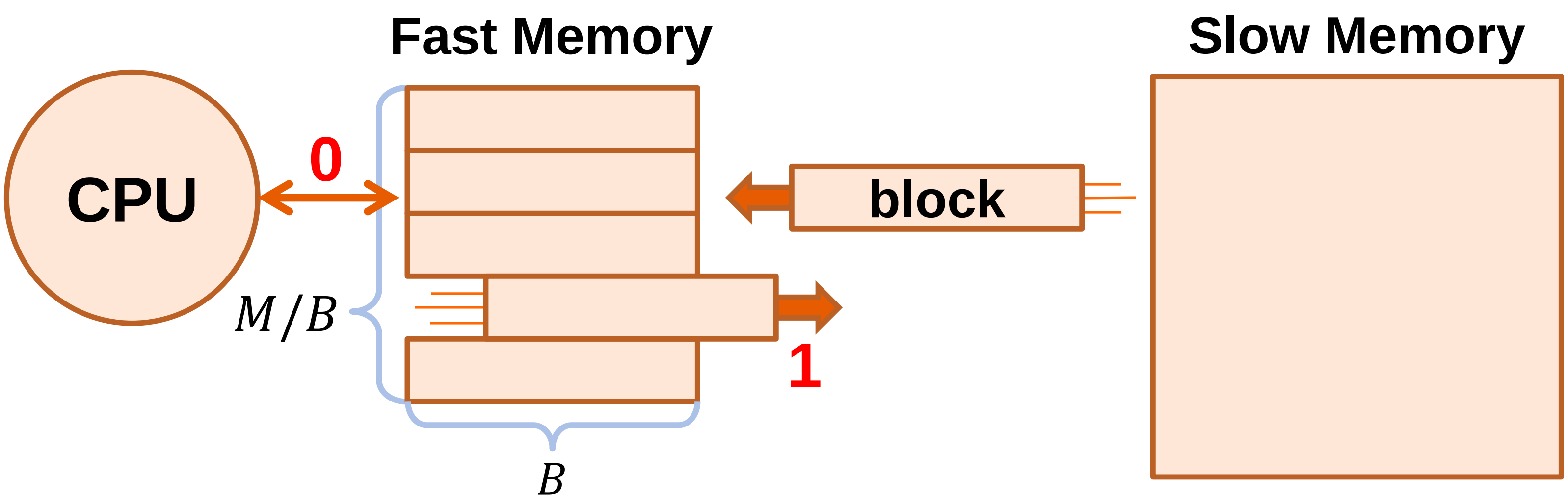
1. Recursively distribute $s_1, \dots, s_{k/2}$ to $b_1, \dots, b_{k/2}$
2. Recursively distribute $s_1, \dots, s_{k/2}$ to $b_{k/2+1}, \dots, b_k$
3. Recursively distribute $s_{k/2+1}, \dots, s_k$ to $b_1, \dots, b_{k/2}$
4. Recursively distribute $s_{k/2+1}, \dots, s_k$ to $b_{k/2+1}, \dots, b_k$

Summary for Cache-Oblivious Sorting

- Has I/O cost $\Theta\left(\frac{N}{B}\log_M N\right)$, matching the bound on the cache-aware model
- The implementation is based on $\sqrt{N}$-way divide-and-conquer, with a recursion depth of $\log_2 \log_2 n$
- This algorithm can be highly parallelized, with $O(\log^2 n)$ depth
- The most efficient general-purpose implementation of parallel sorting

Cache-Oblivious Algorithms

- Cache-Oblivious algorithms are unware of B or M
- Analyze in the ***ideal cache*** model — same as the I/O model except optimal replacement is assumed
  - Can analyze the algorithm assuming an arbitrary replacement, and the ideal cache is always no worse



Advantages of Cache-Oblivious Algorithms

- Since CO algorithms do not depend on memory parameters, bounds generalize to multilevel hierarchies
- Algorithms are platform independent
- Algorithms should be effective even when B and M are not static
- Many of the CO algorithms are highly parallelized because of their recursive approaches