Paper Presentation

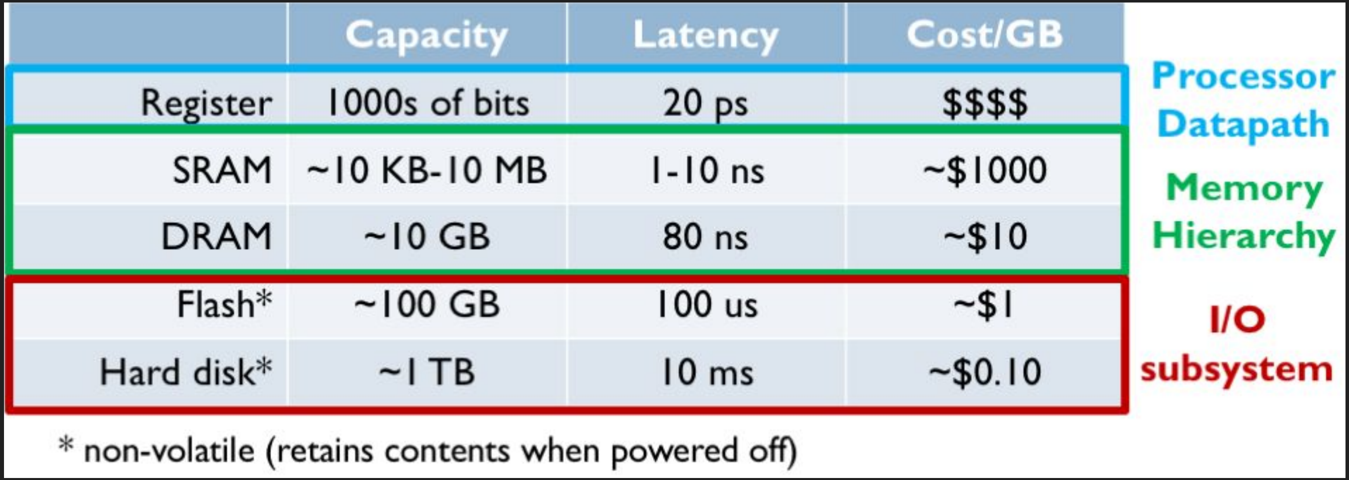
Frigo, Leiserson, Prokop, Ramachandran

# Cache Oblivious Algorithms

Presentation by Viktor Urvantsev

# Why Are Cache Oblivious Algorithms Important?

- Fast access memory is limited in space
- Not using that fast access memory results in large lookup times
- Not as performant as if one was able to keep everything in cache

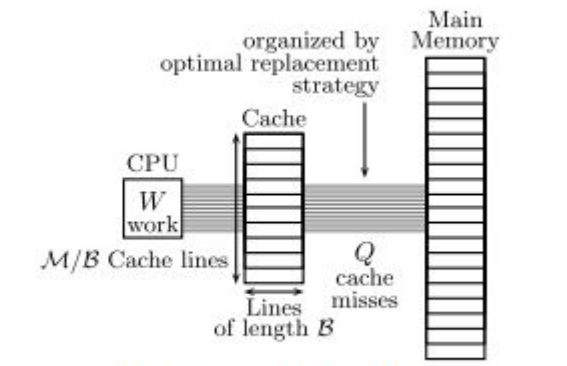


| | Capacity | Latency | Cost/GB | |
|---|---|---|---|---|
| Register | 1000s of bits | 20 ps | $$$$ | Processor Datapath |
| SRAM | ~10 KB-10 MB | 1-10 ns | ~$1000 | Memory Hierarchy |
| DRAM | ~10 GB | 80 ns | ~$10 | |
| Flash* | ~100 GB | 100 us | ~$1 | I/O subsystem |
| Hard disk* | ~1 TB | 10 ms | ~$0.10 | |

* non-volatile (retains contents when powered off)

## (M, B) Ideal Cache Model

- Important Foundation
  - M cache memory
  - B cache line length
  - M/B cache lines
  - Effectively infinite main memory
- Tall Cache Model
  - $M = \Omega(B^2)$

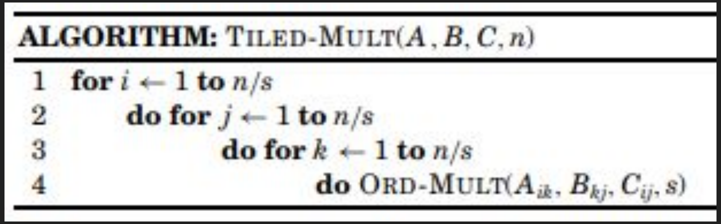


Main Memory
organized by optimal replacement strategy
Cache
CPU
$W$ work
$\mathcal{M}/\mathcal{B}$ Cache lines
$Q$ cache misses
Lines of length $\mathcal{B}$

Fig. 1. The ideal-cache model.

## Cache Aware Algorithms

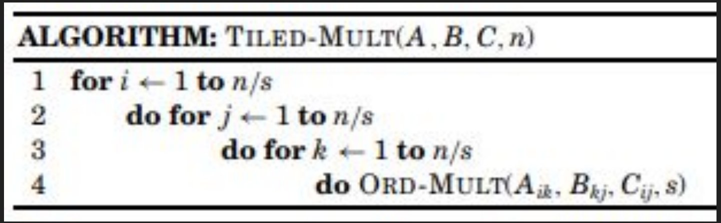
- These algorithms perform well because of tuning
- Programmer has to modify one or more variables in order to maximize performance for a specific input in a specific environment



**ALGORITHM:** TILED-MULT($A$, $B$, $C$, $n$)

```
1  for $i \leftarrow 1$ to $n/s$
2      do for $j \leftarrow 1$ to $n/s$
3          do for $k \leftarrow 1$ to $n/s$
4              do ORD-MULT($A_{ik}$, $B_{kj}$, $C_{ij}$, $s$)
```

## Tiled-Mult Asymptotics Breakdown: Cache Line Number



**ALGORITHM:** TILED-MULT$(A, B, C, n)$

```
1 for $i \leftarrow 1$ to $n/s$
2     do for $j \leftarrow 1$ to $n/s$
3         do for $k \leftarrow 1$ to $n/s$
4             do ORD-MULT$(A_{ik}, B_{kj}, C_{ij}, s)$
```

- Number of cache lines occupied: $\Theta(s + s^2/B)$
  - At least s cache lines occupied by an s x s submatrix in row major order
  - At sizes where row width >= B, $s^2/B$ cache lines occupied
  - In asymptotic notation, maximums and additions are equivalent

## Tiled-Mult Asymptotics Breakdown: Total Cache Complexity

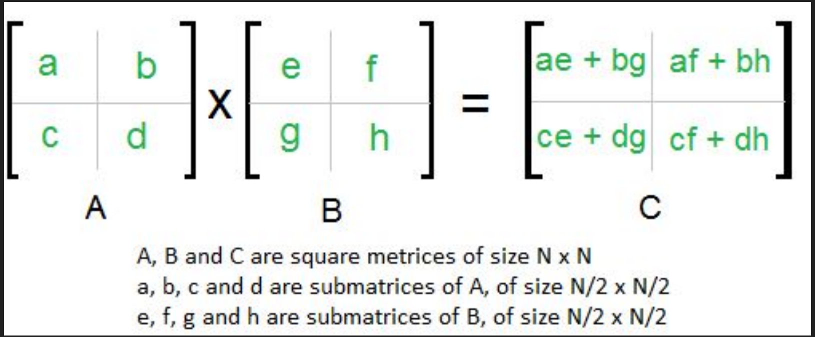
- From tall cache assumption, we can say $s = \Theta(M^{0.5})$
- From the previous slide, we know the submatrix calculation runs with at most $\Theta(s^2/B) = \Theta(M/B)$ cache misses to bring the three matrices into cache
- Total Cache Complexity: $\Theta(1 + n^2/B + (n/(M^{0.5}))^3(M/B))$
  - First term: constant factor data alongside the algorithm necessary for operation
  - Second term: if the whole input matrix can fit in cache
  - Third term:
    - M/B cache lines per iteration of the innermost loop, if the matrix is larger than cache
    - Each loop has $n/s = n/M^{0.5}$ iterations,
    - Triply nested loops yield $(n/M^{0.5})^3$ iterations overall
    - Final complexity of $(n/M^{0.5})^3(M/B)$
- Also written as: $\Theta(1 + n^2/B + (n^3/BM^{0.5}))$

## Cache Oblivious Algorithms

- Can achieve the same bound as before
- Without the tuning to cache!
- Can be placed in various systems and achieve best performance without tweaking

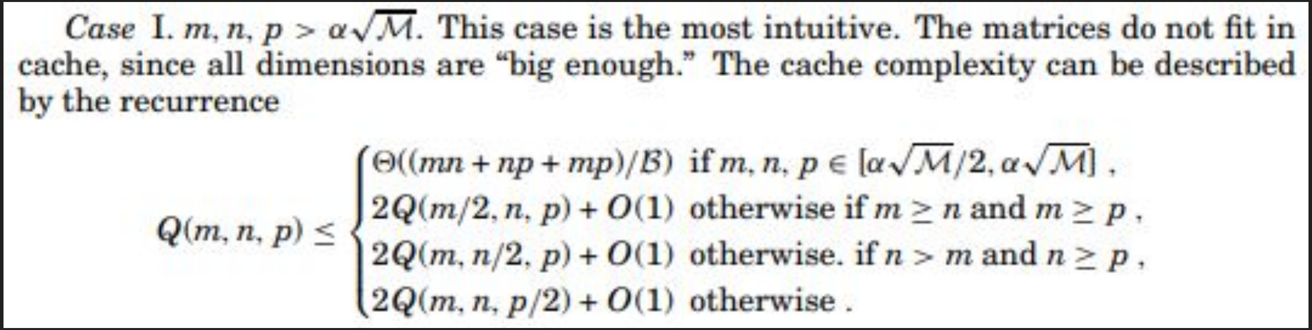
## Cache Oblivious Algorithm: Rec-Mult

- Basic idea
  - Split matrix into quarters
  - Multiply quarters as necessary
  - Sum relevant pieces
  - See picture
  - Calculate multiplication of quarters recursively
  - Base case is when the size of each matrix is 1 x 1
  - Then use integer multiplication and addition to populate C
  - Assuming A = m x n, B = n x p:
    - Work Complexity: O(mnp)
    - Cache Complexity: $O(m + n + p + (mn + np + mp)/2 + mnp/BM^{0.5})$


$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

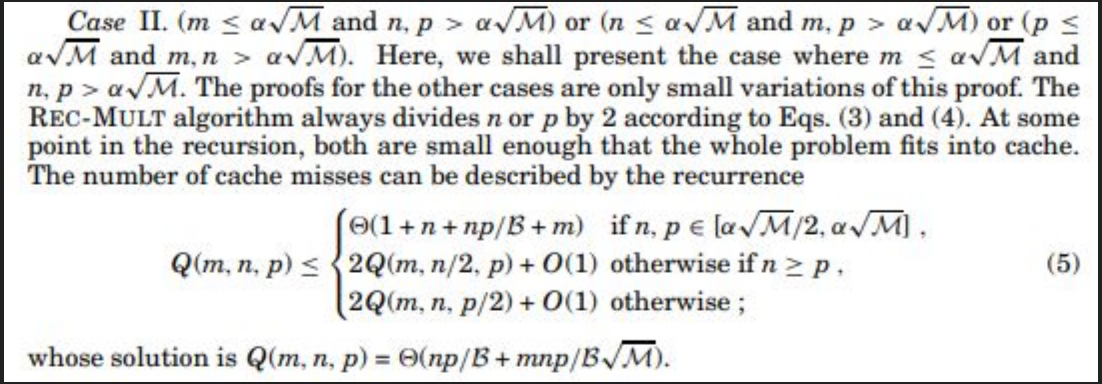
# Rec-Mult Cache Complexity Analysis: Case 1



*Case* I. $m, n, p > \alpha\sqrt{\mathcal{M}}$. This case is the most intuitive. The matrices do not fit in cache, since all dimensions are "big enough." The cache complexity can be described by the recurrence

$$Q(m,n,p) \le \begin{cases} \Theta((mn+np+mp)/\mathcal{B}) & \text{if } m,n,p \in [\alpha\sqrt{\mathcal{M}}/2, \alpha\sqrt{\mathcal{M}}]\ , \\ 2Q(m/2,n,p)+O(1) & \text{otherwise if } m \ge n \text{ and } m \ge p\ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise. if } n > m \text{ and } n \ge p\ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise}\ . \end{cases}$$

## Rec-Mult Cache Complexity Analysis: Case 2

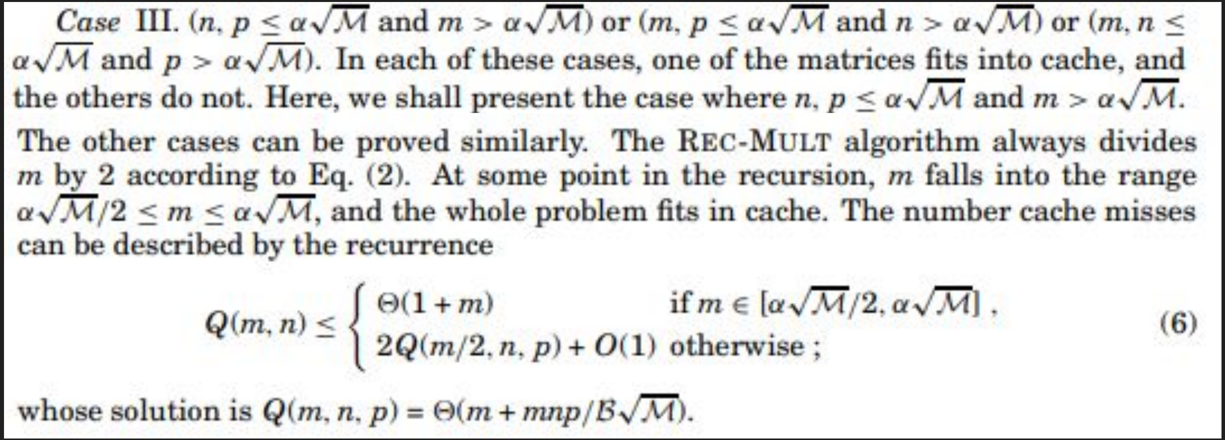


*Case* II. ($m \le \alpha\sqrt{\mathcal{M}}$ and $n, p > \alpha\sqrt{\mathcal{M}}$) or ($n \le \alpha\sqrt{\mathcal{M}}$ and $m, p > \alpha\sqrt{\mathcal{M}}$) or ($p \le \alpha\sqrt{\mathcal{M}}$ and $m, n > \alpha\sqrt{\mathcal{M}}$). Here, we shall present the case where $m \le \alpha\sqrt{\mathcal{M}}$ and $n, p > \alpha\sqrt{\mathcal{M}}$. The proofs for the other cases are only small variations of this proof. The REC-MULT algorithm always divides $n$ or $p$ by 2 according to Eqs. (3) and (4). At some point in the recursion, both are small enough that the whole problem fits into cache. The number of cache misses can be described by the recurrence

$$Q(m, n, p) \le \begin{cases} \Theta(1 + n + np/\mathcal{B} + m) & \text{if } n, p \in [\alpha\sqrt{\mathcal{M}}/2, \alpha\sqrt{\mathcal{M}}] , \\ 2Q(m, n/2, p) + O(1) & \text{otherwise if } n \ge p , \\ 2Q(m, n, p/2) + O(1) & \text{otherwise ;} \end{cases} \tag{5}$$

whose solution is $Q(m, n, p) = \Theta(np/\mathcal{B} + mnp/\mathcal{B}\sqrt{\mathcal{M}})$.

# Rec-Mult Cache Complexity Analysis: Case 3

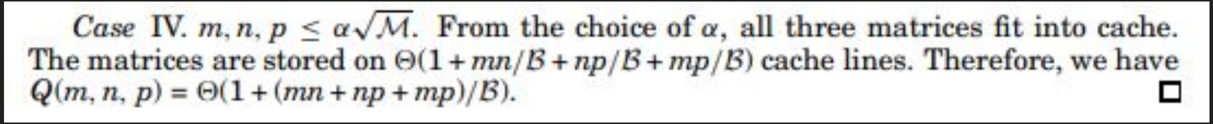


*Case* III. ($n, p \leq \alpha\sqrt{\mathcal{M}}$ and $m > \alpha\sqrt{\mathcal{M}}$) or ($m, p \leq \alpha\sqrt{\mathcal{M}}$ and $n > \alpha\sqrt{\mathcal{M}}$) or ($m, n \leq \alpha\sqrt{\mathcal{M}}$ and $p > \alpha\sqrt{\mathcal{M}}$). In each of these cases, one of the matrices fits into cache, and the others do not. Here, we shall present the case where $n, p \leq \alpha\sqrt{\mathcal{M}}$ and $m > \alpha\sqrt{\mathcal{M}}$. The other cases can be proved similarly. The REC-MULT algorithm always divides $m$ by 2 according to Eq. (2). At some point in the recursion, $m$ falls into the range $\alpha\sqrt{\mathcal{M}}/2 \leq m \leq \alpha\sqrt{\mathcal{M}}$, and the whole problem fits in cache. The number cache misses can be described by the recurrence

$$Q(m, n) \leq \begin{cases} \Theta(1 + m) & \text{if } m \in [\alpha\sqrt{\mathcal{M}}/2, \alpha\sqrt{\mathcal{M}}] \,, \\ 2Q(m/2, n, p) + O(1) & \text{otherwise ;} \end{cases} \tag{6}$$

whose solution is $Q(m, n, p) = \Theta(m + mnp/\mathcal{B}\sqrt{\mathcal{M}})$.

## Rec-Mult Cache Complexity Analysis: Case 4



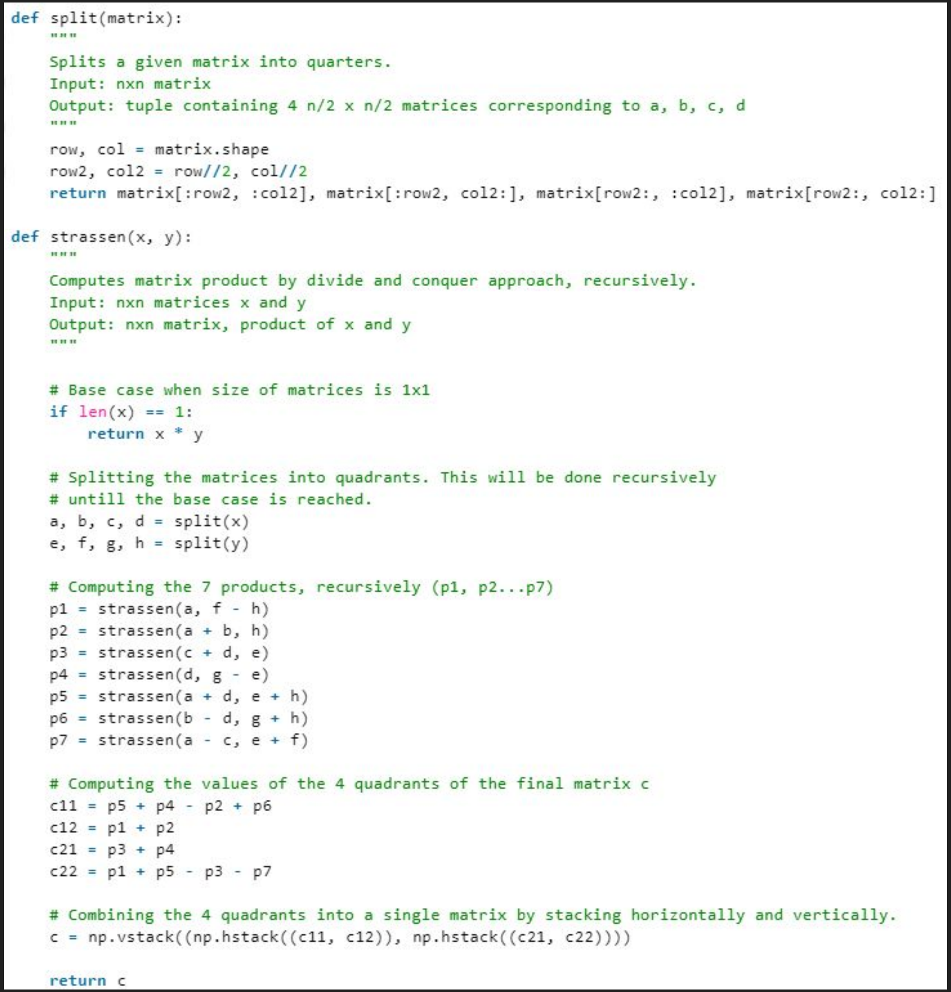
*Case* IV. $m, n, p \leq \alpha\sqrt{\mathcal{M}}$. From the choice of $\alpha$, all three matrices fit into cache. The matrices are stored on $\Theta(1 + mn/\mathcal{B} + np/\mathcal{B} + mp/\mathcal{B})$ cache lines. Therefore, we have $Q(m, n, p) = \Theta(1 + (mn + np + mp)/\mathcal{B})$. $\square$

# Strassen's Algorithm

- Runtime: $\Theta(n^{\log 7})$
- Cache Complexity Recurrence:

$$Q(n) \le \begin{cases} \Theta(1 + n + n^2/\mathcal{B}) & \text{if } n^2 \le \alpha\mathcal{M}\,, \\ 7Q(n/2) + O(n^2/\mathcal{B}) & \text{otherwise}\,; \end{cases}$$

- Cache Complexity:
    - $\Theta(n + n^2/B + n^{\log 7}/BM^{(\log 7/2)-1})$

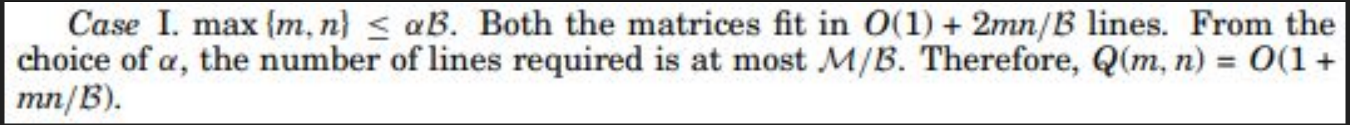


```
def split(matrix):
    """
    Splits a given matrix into quarters.
    Input: nxn matrix
    Output: tuple containing 4 n/2 x n/2 matrices corresponding to a, b, c, d
    """
    row, col = matrix.shape
    row2, col2 = row//2, col//2
    return matrix[:row2, :col2], matrix[:row2, col2:], matrix[row2:, :col2], matrix[row2:, col2:]

def strassen(x, y):
    """
    Computes matrix product by divide and conquer approach, recursively.
    Input: nxn matrices x and y
    Output: nxn matrix, product of x and y
    """

    # Base case when size of matrices is 1x1
    if len(x) == 1:
        return x * y

    # Splitting the matrices into quadrants. This will be done recursively
    # untill the base case is reached.
    a, b, c, d = split(x)
    e, f, g, h = split(y)

    # Computing the 7 products, recursively (p1, p2...p7)
    p1 = strassen(a, f - h)
    p2 = strassen(a + b, h)
    p3 = strassen(c + d, e)
    p4 = strassen(d, g - e)
    p5 = strassen(a + d, e + h)
    p6 = strassen(b - d, g + h)
    p7 = strassen(a - c, e + f)

    # Computing the values of the 4 quadrants of the final matrix c
    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7

    # Combining the 4 quadrants into a single matrix by stacking horizontally and vertically.
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

    return c
```

## Matrix Transposition

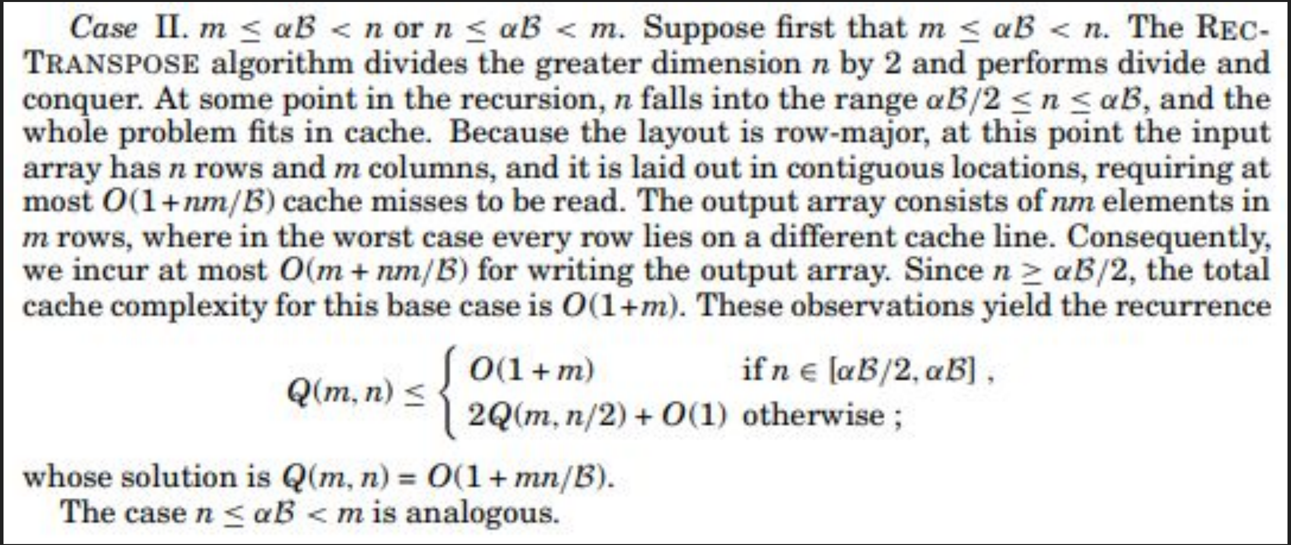
- Problem
  - Turn a m x n matrix A into $A^T$ (or B), an n x m matrix, both stored in row major order
- Basic algorithm
  - Doubly nested loops
  - O(mn) cache misses, where the matrix is m x n
- Recursive algorithm
  - Split matrices in half at each level, dividing (A horizontally/B vertically) or (B horizontally/A vertically)
  - Goes down until base case (n = 1, m = 1)
  - Writes $A_{ij}$ to $B_{ji}$
  - O(mn/B) cache misses
    - Will prove in following slides

## Recursive Matrix Transposition: Case 1



*Case* I. $\max\{m, n\} \leq \alpha\mathcal{B}$. Both the matrices fit in $O(1) + 2mn/\mathcal{B}$ lines. From the choice of $\alpha$, the number of lines required is at most $\mathcal{M}/\mathcal{B}$. Therefore, $Q(m, n) = O(1 + mn/\mathcal{B})$.

# Recursive Matrix Transposition: Case 2



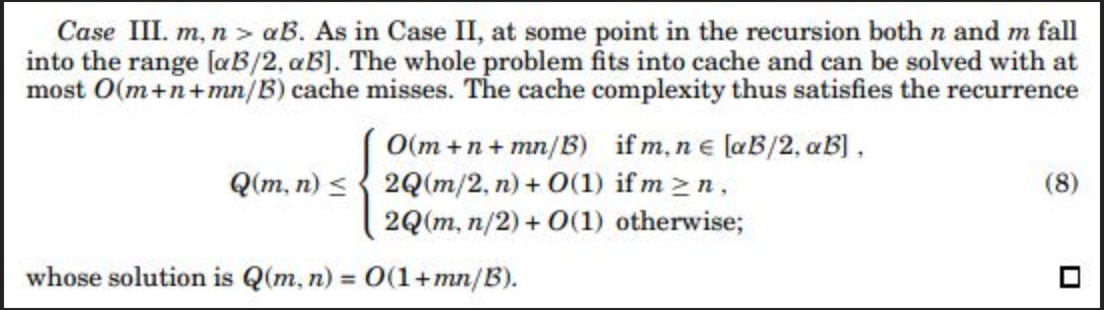
*Case* II. $m \le \alpha\mathcal{B} < n$ or $n \le \alpha\mathcal{B} < m$. Suppose first that $m \le \alpha\mathcal{B} < n$. The REC-TRANSPOSE algorithm divides the greater dimension $n$ by 2 and performs divide and conquer. At some point in the recursion, $n$ falls into the range $\alpha\mathcal{B}/2 \le n \le \alpha\mathcal{B}$, and the whole problem fits in cache. Because the layout is row-major, at this point the input array has $n$ rows and $m$ columns, and it is laid out in contiguous locations, requiring at most $O(1+nm/\mathcal{B})$ cache misses to be read. The output array consists of $nm$ elements in $m$ rows, where in the worst case every row lies on a different cache line. Consequently, we incur at most $O(m+nm/\mathcal{B})$ for writing the output array. Since $n \ge \alpha\mathcal{B}/2$, the total cache complexity for this base case is $O(1+m)$. These observations yield the recurrence

$$Q(m,n) \le \begin{cases} O(1+m) & \text{if } n \in [\alpha\mathcal{B}/2, \alpha\mathcal{B}] \, , \\ 2Q(m,n/2)+O(1) & \text{otherwise} \, ; \end{cases}$$

whose solution is $Q(m,n) = O(1+mn/\mathcal{B})$.

The case $n \le \alpha\mathcal{B} < m$ is analogous.

## Recursive Matrix Transposition: Case 3

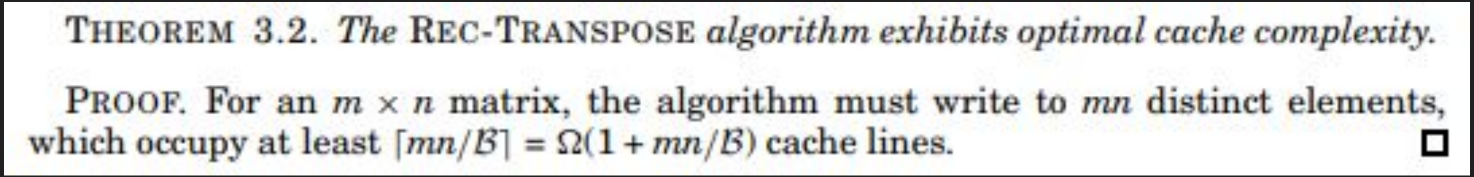


*Case* III. $m, n > \alpha\mathcal{B}$. As in Case II, at some point in the recursion both $n$ and $m$ fall into the range $[\alpha\mathcal{B}/2, \alpha\mathcal{B}]$. The whole problem fits into cache and can be solved with at most $O(m+n+mn/\mathcal{B})$ cache misses. The cache complexity thus satisfies the recurrence

$$Q(m,n) \le \begin{cases} O(m+n+mn/\mathcal{B}) & \text{if } m,n \in [\alpha\mathcal{B}/2, \alpha\mathcal{B}] , \\ 2Q(m/2,n) + O(1) & \text{if } m \ge n , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases} \qquad (8)$$

whose solution is $Q(m,n) = O(1+mn/\mathcal{B})$. $\square$

# Recursive Matrix Transposition: Optimal Cache Complexity



THEOREM 3.2. *The* REC-TRANSPOSE *algorithm exhibits optimal cache complexity.*

PROOF. For an $m \times n$ matrix, the algorithm must write to $mn$ distinct elements, which occupy at least $\lceil mn/\mathcal{B} \rceil = \Omega(1 + mn/\mathcal{B})$ cache lines. $\square$

## Fast Fourier Transform

- Discrete Fourier Transform

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} ,$$

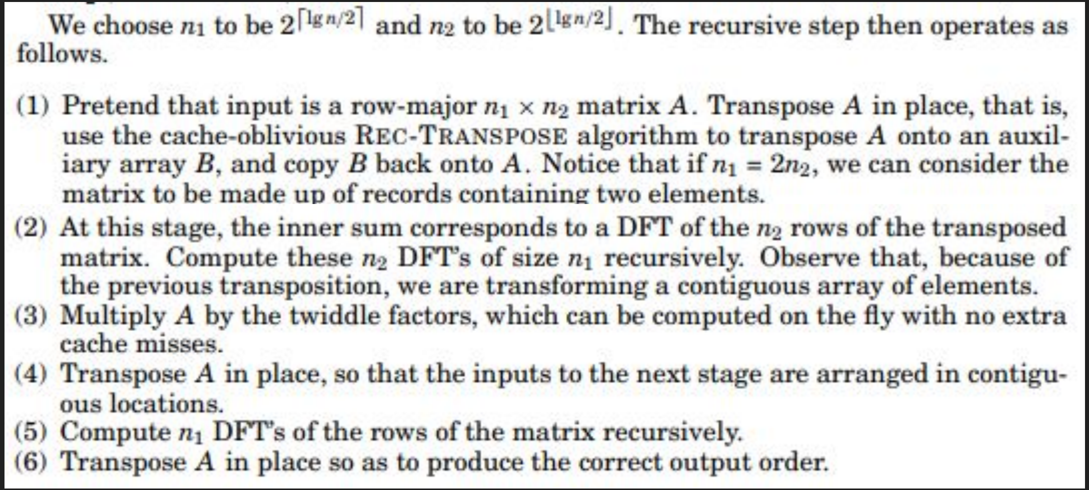
where $\omega_n = e^{2\pi\sqrt{-1}/n}$

- Can also be calculated using:

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left( \left( \sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right) \omega_{n_2}^{-i_2 j_2} . \qquad (10)$$

Where $n_1$, $n_2$ are any two factors of n such that $n_1 n_2 = n$

# Cache Oblivious Algorithm: Fast Fourier Transform



We choose $n_1$ to be $2^{\lceil \lg n/2 \rceil}$ and $n_2$ to be $2^{\lfloor \lg n/2 \rfloor}$. The recursive step then operates as follows.

(1) Pretend that input is a row-major $n_1 \times n_2$ matrix $A$. Transpose $A$ in place, that is, use the cache-oblivious REC-TRANSPOSE algorithm to transpose $A$ onto an auxiliary array $B$, and copy $B$ back onto $A$. Notice that if $n_1 = 2n_2$, we can consider the matrix to be made up of records containing two elements.

(2) At this stage, the inner sum corresponds to a DFT of the $n_2$ rows of the transposed matrix. Compute these $n_2$ DFT's of size $n_1$ recursively. Observe that, because of the previous transposition, we are transforming a contiguous array of elements.

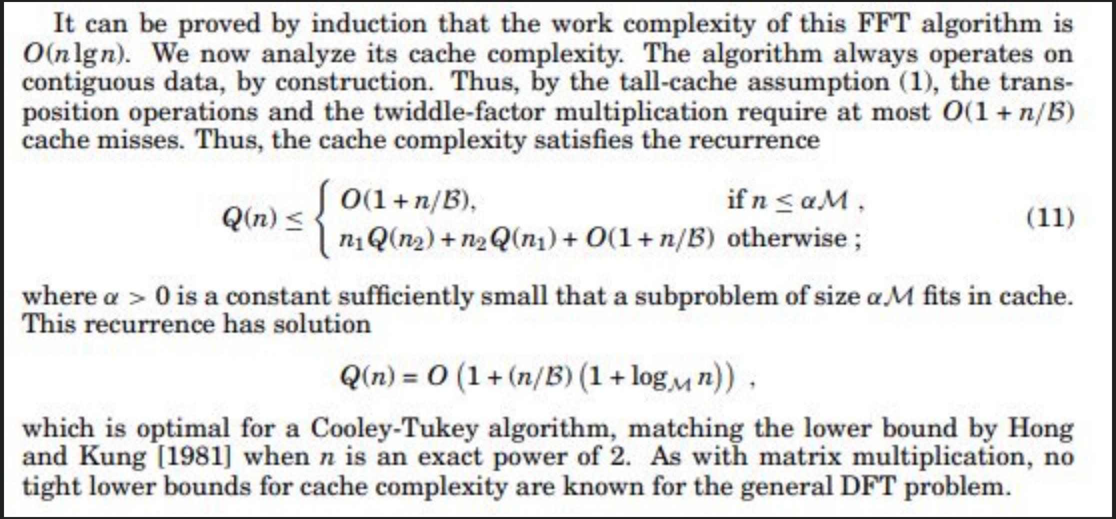
(3) Multiply $A$ by the twiddle factors, which can be computed on the fly with no extra cache misses.

(4) Transpose $A$ in place, so that the inputs to the next stage are arranged in contiguous locations.

(5) Compute $n_1$ DFT's of the rows of the matrix recursively.

(6) Transpose $A$ in place so as to produce the correct output order.

# Recursive Fast Fourier Transform: Cache Complexity



It can be proved by induction that the work complexity of this FFT algorithm is $O(n \lg n)$. We now analyze its cache complexity. The algorithm always operates on contiguous data, by construction. Thus, by the tall-cache assumption (1), the transposition operations and the twiddle-factor multiplication require at most $O(1 + n/\mathcal{B})$ cache misses. Thus, the cache complexity satisfies the recurrence

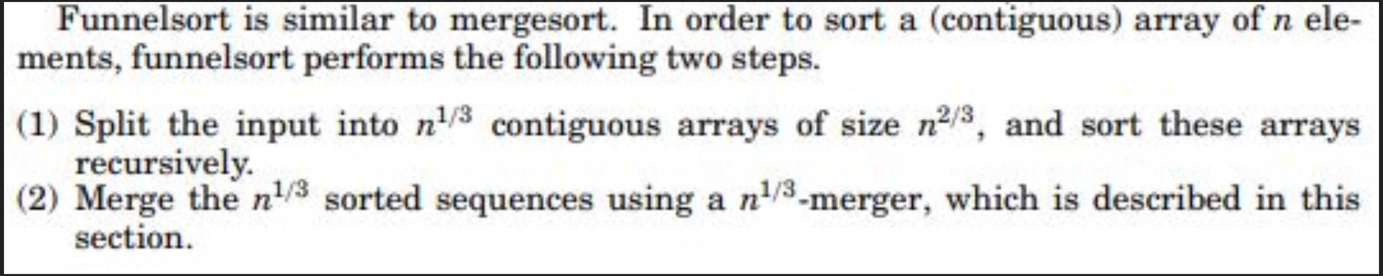
$$Q(n) \le \begin{cases} O(1 + n/\mathcal{B}), & \text{if } n \le \alpha\mathcal{M} , \\ n_1 Q(n_2) + n_2 Q(n_1) + O(1 + n/\mathcal{B}) & \text{otherwise ;} \end{cases} \tag{11}$$

where $\alpha > 0$ is a constant sufficiently small that a subproblem of size $\alpha\mathcal{M}$ fits in cache. This recurrence has solution

$$Q(n) = O\left(1 + (n/\mathcal{B})\left(1 + \log_{\mathcal{M}} n\right)\right) ,$$

which is optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [1981] when $n$ is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general DFT problem.

# Funnelsort



Funnelsort is similar to mergesort. In order to sort a (contiguous) array of $n$ elements, funnelsort performs the following two steps.

(1) Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.
(2) Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described in this section.

- Merge step uses a k-merger
- $Q(n) = \Theta(1 + (n/B)(\log_M n))$

# K-Merger

- Built out of $k^{0.5}$ $k^{0.5}$-mergers on the left and one $k^{0.5}$-merger on the right
- Each buffer can hold $2k^{3/2}$ elements

A $k$-merger operates recursively in the following way. In order to output $k^3$ elements, the $k$-merger invokes $R$ $k^{3/2}$ times. Before each invocation, however, the $k$-merger fills all buffers that are less than half full, that is, all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer $i$, the algorithm invokes the corresponding left merger $L_i$ once. Since $L_i$ outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after $L_i$ finishes.

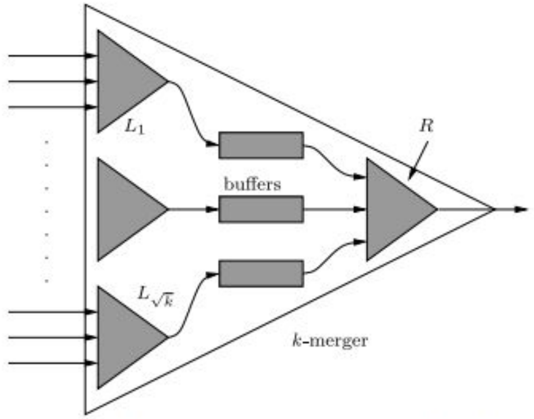
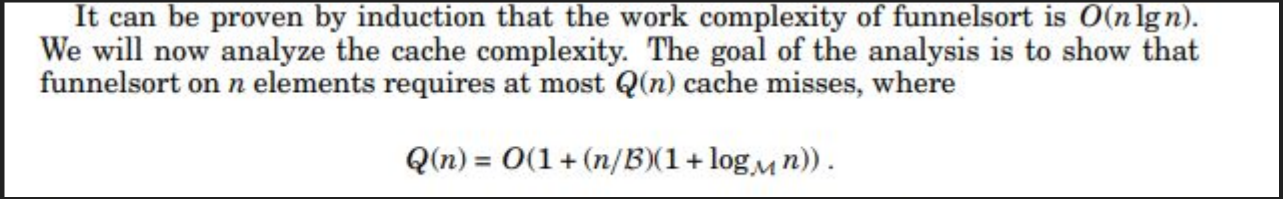


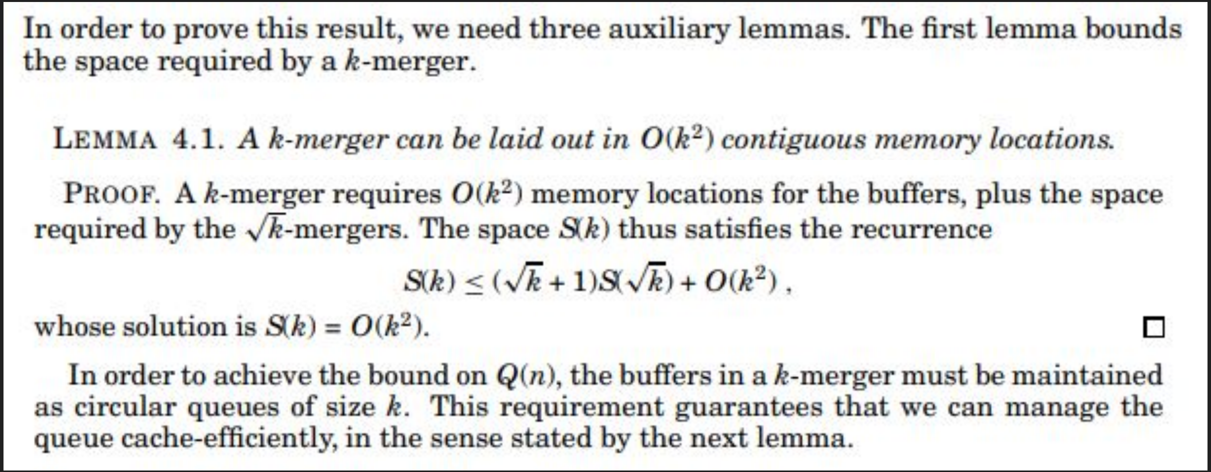
Fig. 3. Illustration of a $k$-merger. A $k$-merger is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger $R$.

## FunnelSort Cache Complexity, Beginning Of Proof


It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$. We will now analyze the cache complexity. The goal of the analysis is to show that funnelsort on $n$ elements requires at most $Q(n)$ cache misses, where

$$Q(n) = O(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n)) .$$

## FunnelSort Cache Complexity, Lemma 1



In order to prove this result, we need three auxiliary lemmas. The first lemma bounds the space required by a $k$-merger.

LEMMA 4.1. *A $k$-merger can be laid out in $O(k^2)$ contiguous memory locations.*

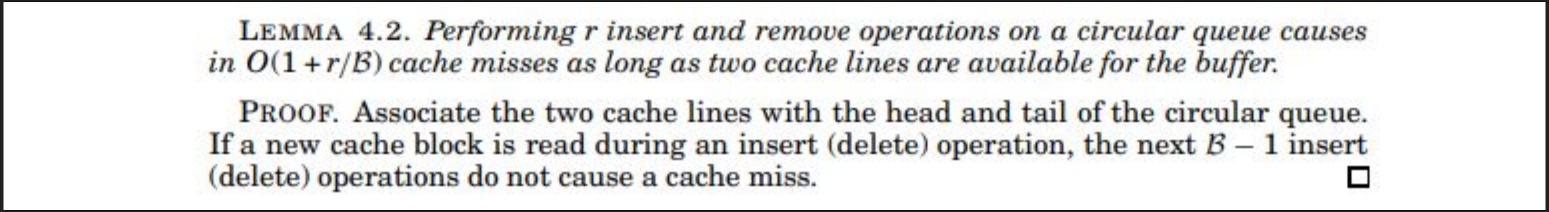
PROOF. A $k$-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \le (\sqrt{k}+1)S(\sqrt{k}) + O(k^2) ,$$

whose solution is $S(k) = O(k^2)$. $\square$

In order to achieve the bound on $Q(n)$, the buffers in a $k$-merger must be maintained as circular queues of size $k$. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

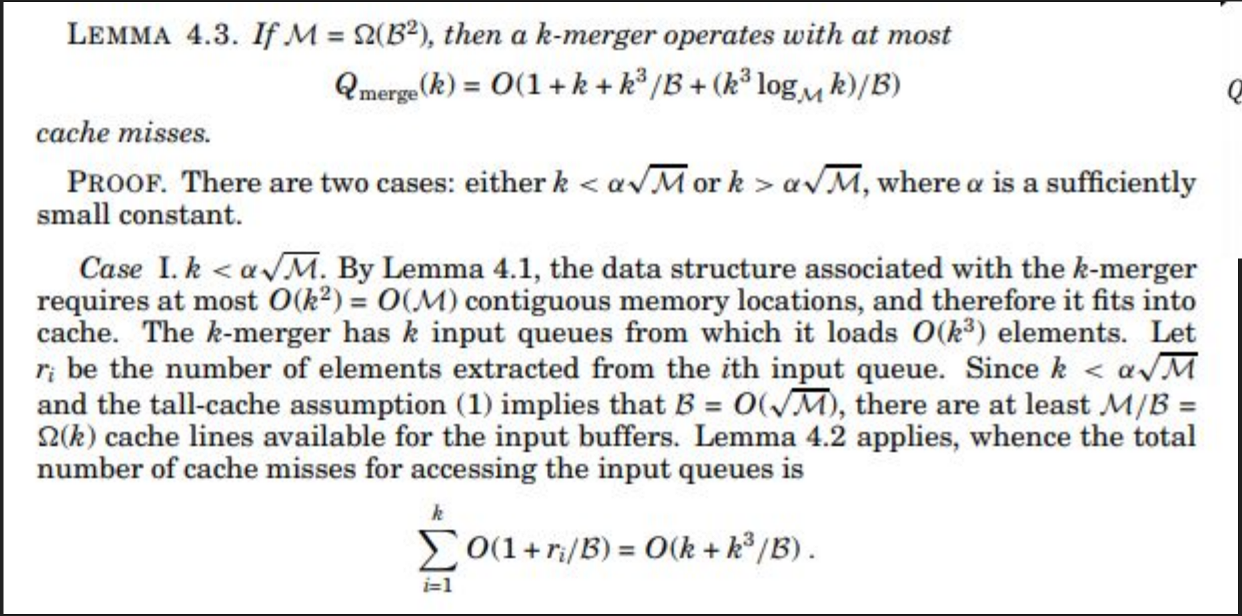
## FunnelSort Cache Complexity, Lemma 2



LEMMA 4.2. *Performing $r$ insert and remove operations on a circular queue causes in $O(1 + r/\mathcal{B})$ cache misses as long as two cache lines are available for the buffer.*

PROOF. Associate the two cache lines with the head and tail of the circular queue. If a new cache block is read during an insert (delete) operation, the next $\mathcal{B} - 1$ insert (delete) operations do not cause a cache miss. $\square$

# FunnelSort Cache Complexity, Lemma 3, Part 1



LEMMA 4.3. *If $\mathcal{M} = \Omega(\mathcal{B}^2)$, then a $k$-merger operates with at most*

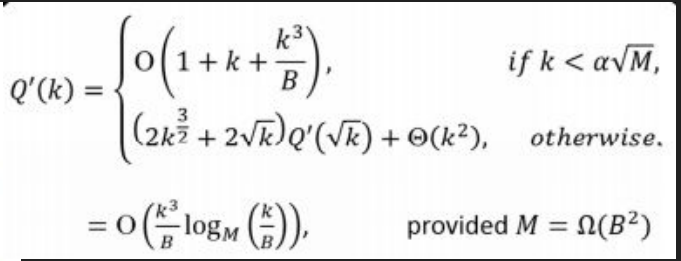
$$Q_{\text{merge}}(k) = O(1 + k + k^3/\mathcal{B} + (k^3 \log_{\mathcal{M}} k)/\mathcal{B})$$

*cache misses.*

PROOF. There are two cases: either $k < \alpha\sqrt{\mathcal{M}}$ or $k > \alpha\sqrt{\mathcal{M}}$, where $\alpha$ is a sufficiently small constant.

*Case* I. $k < \alpha\sqrt{\mathcal{M}}$. By Lemma 4.1, the data structure associated with the $k$-merger requires at most $O(k^2) = O(\mathcal{M})$ contiguous memory locations, and therefore it fits into cache. The $k$-merger has $k$ input queues from which it loads $O(k^3)$ elements. Let $r_i$ be the number of elements extracted from the $i$th input queue. Since $k < \alpha\sqrt{\mathcal{M}}$ and the tall-cache assumption (1) implies that $\mathcal{B} = O(\sqrt{\mathcal{M}})$, there are at least $\mathcal{M}/\mathcal{B} = \Omega(k)$ cache lines available for the input buffers. Lemma 4.2 applies, whence the total number of cache misses for accessing the input queues is

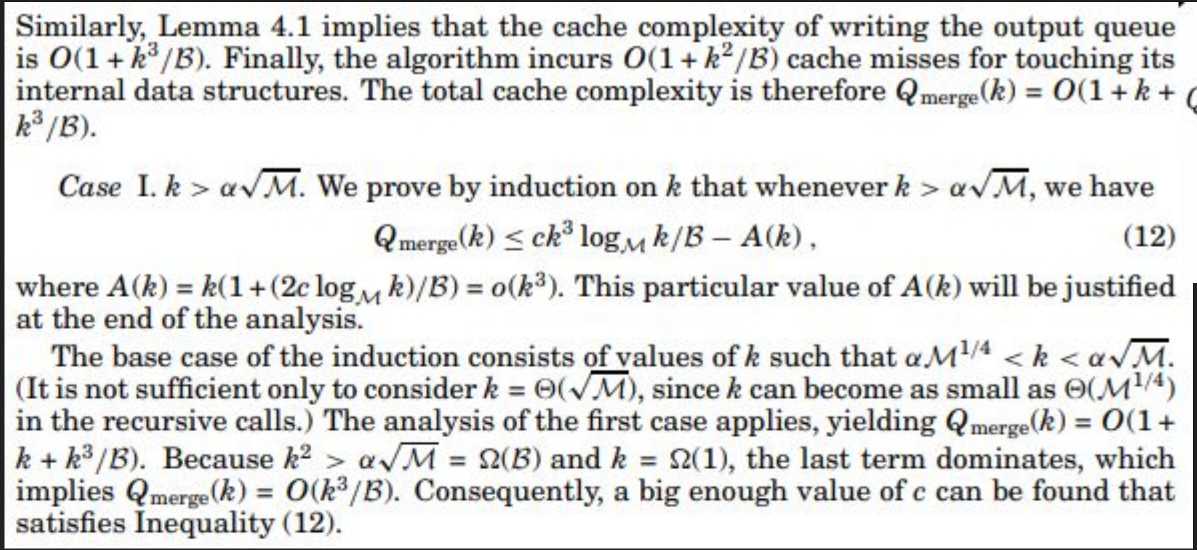
$$\sum_{i=1}^{k} O(1 + r_i/\mathcal{B}) = O(k + k^3/\mathcal{B}) .$$



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

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

# FunnelSort Cache Complexity, Lemma 3, Part 2



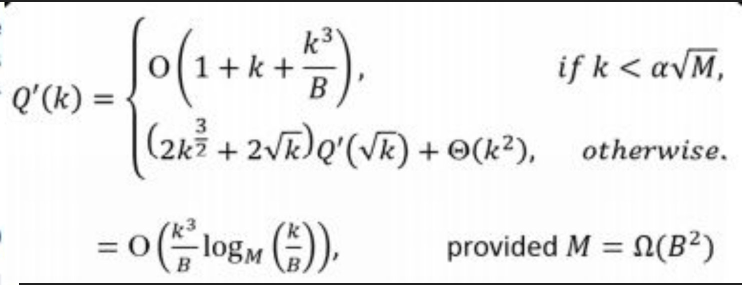
Similarly, Lemma 4.1 implies that the cache complexity of writing the output queue is $O(1 + k^3/\mathcal{B})$. Finally, the algorithm incurs $O(1 + k^2/\mathcal{B})$ cache misses for touching its internal data structures. The total cache complexity is therefore $Q_{\text{merge}}(k) = O(1 + k + k^3/\mathcal{B})$.

*Case* I. $k > \alpha\sqrt{\mathcal{M}}$. We prove by induction on $k$ that whenever $k > \alpha\sqrt{\mathcal{M}}$, we have

$$Q_{\text{merge}}(k) \le ck^3 \log_{\mathcal{M}} k/\mathcal{B} - A(k)\,, \tag{12}$$

where $A(k) = k(1 + (2c \log_{\mathcal{M}} k)/\mathcal{B}) = o(k^3)$. This particular value of $A(k)$ will be justified at the end of the analysis.

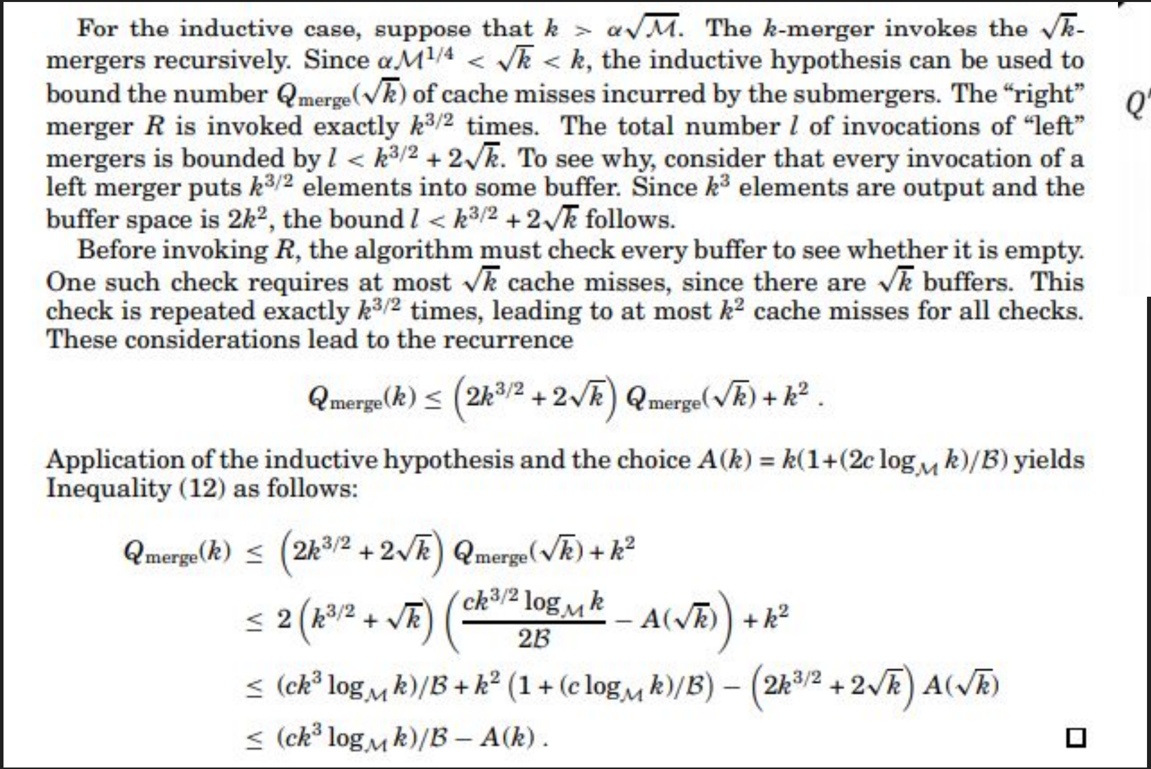
The base case of the induction consists of values of $k$ such that $\alpha\mathcal{M}^{1/4} < k < \alpha\sqrt{\mathcal{M}}$. (It is not sufficient only to consider $k = \Theta(\sqrt{\mathcal{M}})$, since $k$ can become as small as $\Theta(\mathcal{M}^{1/4})$ in the recursive calls.) The analysis of the first case applies, yielding $Q_{\text{merge}}(k) = O(1 + k + k^3/\mathcal{B})$. Because $k^2 > \alpha\sqrt{\mathcal{M}} = \Omega(\mathcal{B})$ and $k = \Omega(1)$, the last term dominates, which implies $Q_{\text{merge}}(k) = O(k^3/\mathcal{B})$. Consequently, a big enough value of $c$ can be found that satisfies Inequality (12).



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

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

# FunnelSort Cache Complexity, Lemma 3, Part 3



For the inductive case, suppose that $k > \alpha\sqrt{\mathcal{M}}$. The $k$-merger invokes the $\sqrt{k}$-mergers recursively. Since $\alpha\mathcal{M}^{1/4} < \sqrt{k} < k$, the inductive hypothesis can be used to bound the number $Q_{\text{merge}}(\sqrt{k})$ of cache misses incurred by the submergers. The "right" merger $R$ is invoked exactly $k^{3/2}$ times. The total number $l$ of invocations of "left" mergers is bounded by $l < k^{3/2} + 2\sqrt{k}$. To see why, consider that every invocation of a left merger puts $k^{3/2}$ elements into some buffer. Since $k^3$ elements are output and the buffer space is $2k^2$, the bound $l < k^{3/2} + 2\sqrt{k}$ follows.

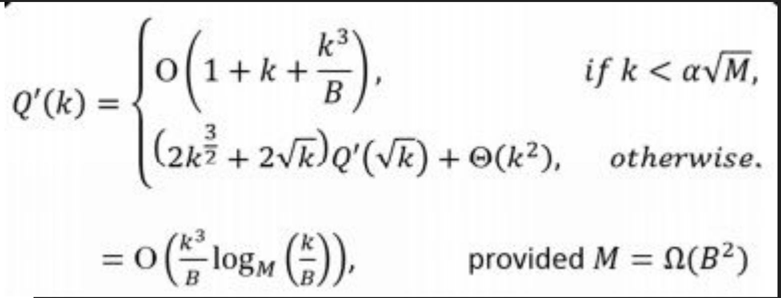
Before invoking $R$, the algorithm must check every buffer to see whether it is empty. One such check requires at most $\sqrt{k}$ cache misses, since there are $\sqrt{k}$ buffers. This check is repeated exactly $k^{3/2}$ times, leading to at most $k^2$ cache misses for all checks. These considerations lead to the recurrence

$$Q_{\text{merge}}(k) \le \left(2k^{3/2} + 2\sqrt{k}\right) Q_{\text{merge}}(\sqrt{k}) + k^2 \ .$$

Application of the inductive hypothesis and the choice $A(k) = k(1+(2c\log_{\mathcal{M}} k)/\mathcal{B})$ yields Inequality (12) as follows:

$$\begin{aligned}
Q_{\text{merge}}(k) &\le \left(2k^{3/2} + 2\sqrt{k}\right) Q_{\text{merge}}(\sqrt{k}) + k^2 \\
&\le 2\left(k^{3/2} + \sqrt{k}\right)\left(\frac{ck^{3/2}\log_{\mathcal{M}} k}{2\mathcal{B}} - A(\sqrt{k})\right) + k^2 \\
&\le (ck^3\log_{\mathcal{M}} k)/\mathcal{B} + k^2\left(1 + (c\log_{\mathcal{M}} k)/\mathcal{B}\right) - \left(2k^{3/2} + 2\sqrt{k}\right) A(\sqrt{k}) \\
&\le (ck^3\log_{\mathcal{M}} k)/\mathcal{B} - A(k) \ .
\end{aligned}$$

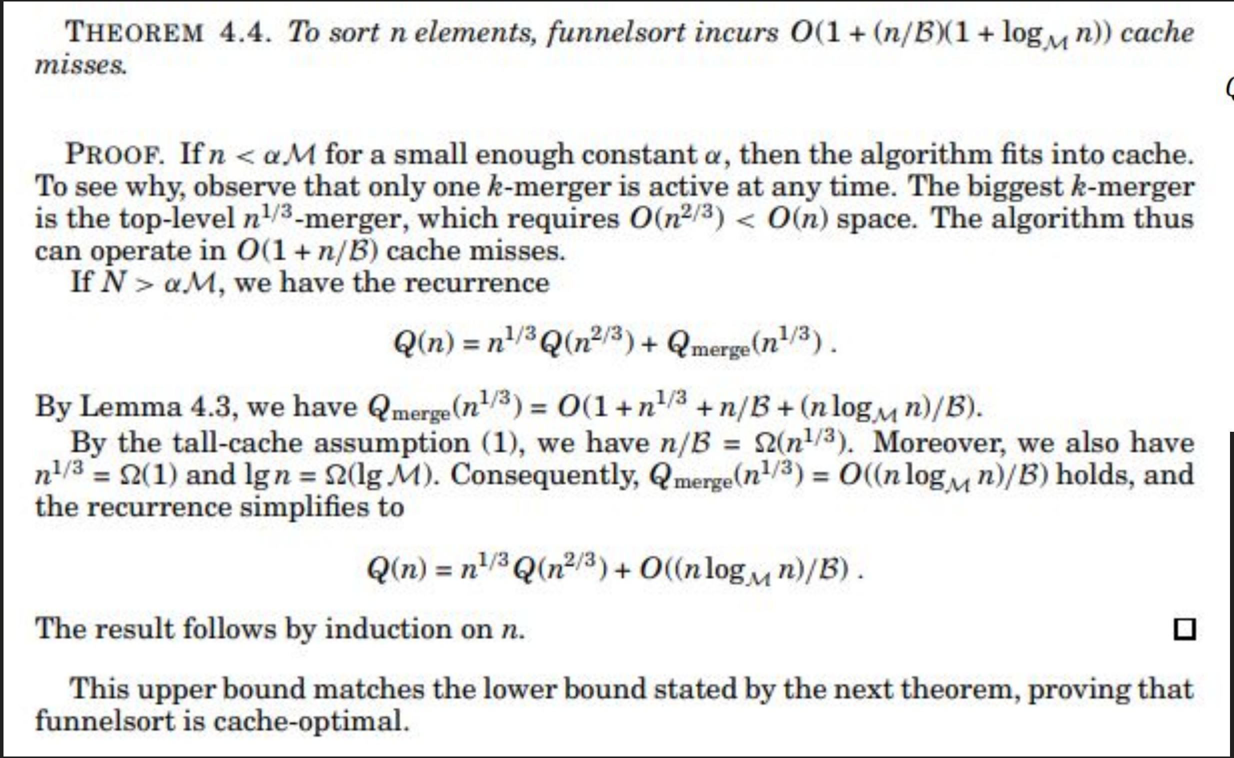
□



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

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

# FunnelSort Cache Complexity, Wrapped Up



THEOREM 4.4. *To sort n elements, funnelsort incurs $O(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$ cache misses.*

PROOF. If $n < \alpha\mathcal{M}$ for a small enough constant $\alpha$, then the algorithm fits into cache. To see why, observe that only one $k$-merger is active at any time. The biggest $k$-merger is the top-level $n^{1/3}$-merger, which requires $O(n^{2/3}) < O(n)$ space. The algorithm thus can operate in $O(1 + n/\mathcal{B})$ cache misses.

If $N > \alpha\mathcal{M}$, we have the recurrence

$$Q(n) = n^{1/3} Q(n^{2/3}) + Q_{\text{merge}}(n^{1/3}) .$$

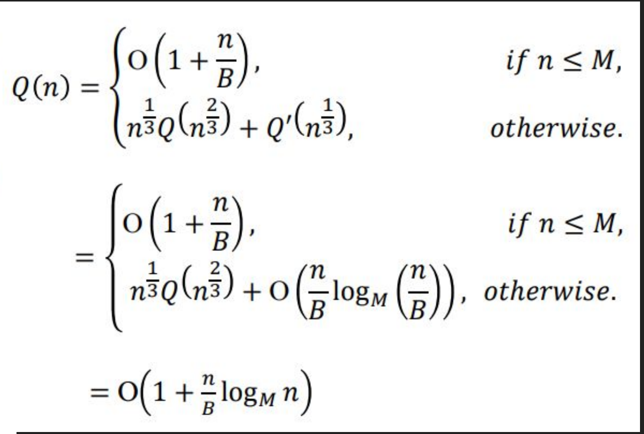
By Lemma 4.3, we have $Q_{\text{merge}}(n^{1/3}) = O(1 + n^{1/3} + n/\mathcal{B} + (n \log_{\mathcal{M}} n)/\mathcal{B})$.

By the tall-cache assumption (1), we have $n/\mathcal{B} = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg \mathcal{M})$. Consequently, $Q_{\text{merge}}(n^{1/3}) = O((n \log_{\mathcal{M}} n)/\mathcal{B})$ holds, and the recurrence simplifies to

$$Q(n) = n^{1/3} Q(n^{2/3}) + O((n \log_{\mathcal{M}} n)/\mathcal{B}) .$$

The result follows by induction on $n$. ☐

This upper bound matches the lower bound stated by the next theorem, proving that funnelsort is cache-optimal.

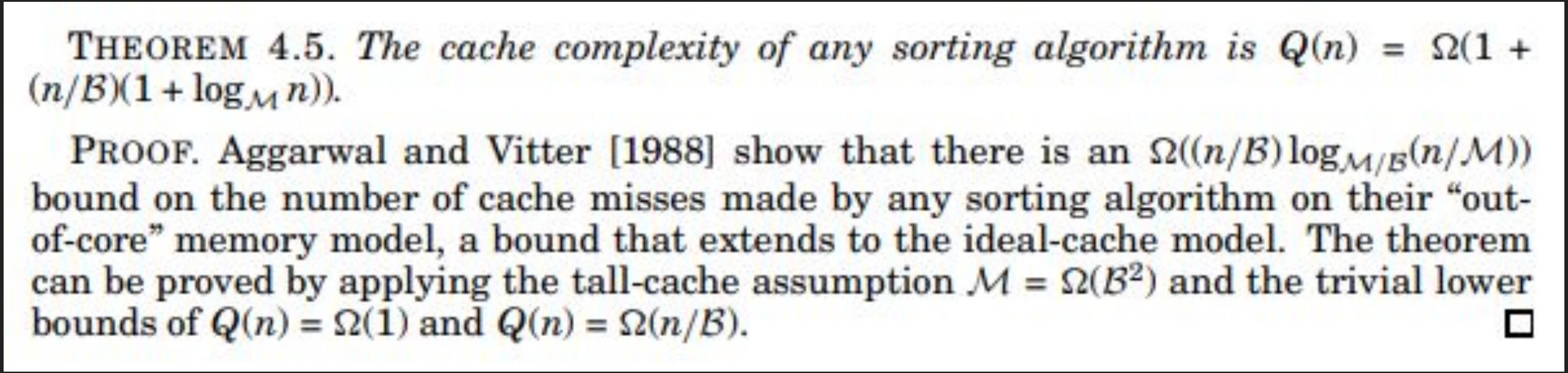


$$Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & \textit{if } n \le M, \\ n^{\frac{1}{3}} Q\left(n^{\frac{2}{3}}\right) + Q'\left(n^{\frac{1}{3}}\right), & \textit{otherwise.} \end{cases}$$

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

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

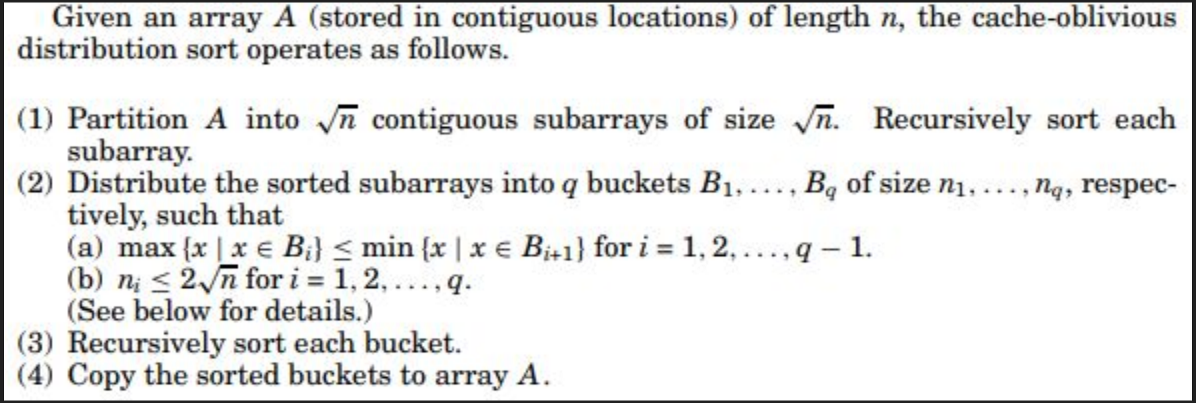
# General Cache Complexity Of Any Sorting Algorithm



THEOREM 4.5. *The cache complexity of any sorting algorithm is* $Q(n) = \Omega(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$.

PROOF. Aggarwal and Vitter [1988] show that there is an $\Omega((n/\mathcal{B})\log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{M}))$ bound on the number of cache misses made by any sorting algorithm on their "out-of-core" memory model, a bound that extends to the ideal-cache model. The theorem can be proved by applying the tall-cache assumption $\mathcal{M} = \Omega(\mathcal{B}^2)$ and the trivial lower bounds of $Q(n) = \Omega(1)$ and $Q(n) = \Omega(n/\mathcal{B})$. $\square$

# Distribution Sort



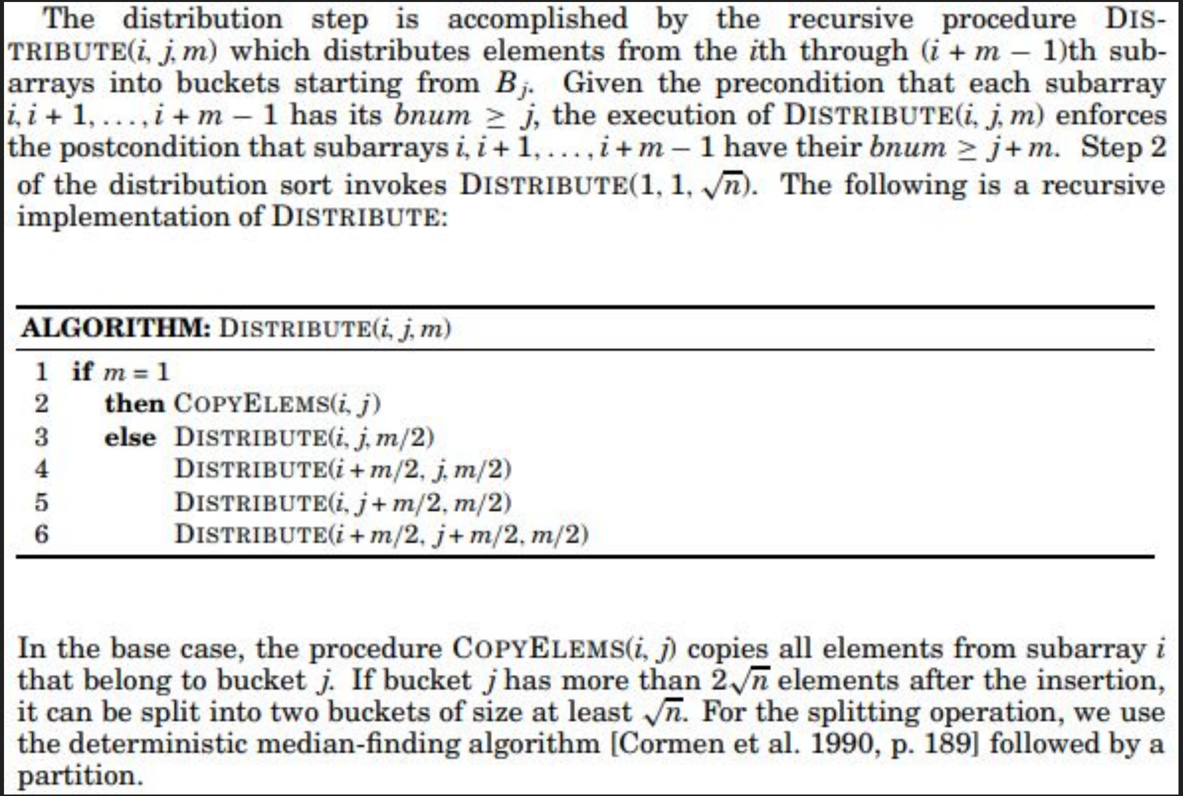
Given an array $A$ (stored in contiguous locations) of length $n$, the cache-oblivious distribution sort operates as follows.

(1) Partition $A$ into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.
(2) Distribute the sorted subarrays into $q$ buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$, respectively, such that
  (a) $\max \{x \mid x \in B_i\} \leq \min \{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.
  (b) $n_i \leq 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.
  (See below for details.)
(3) Recursively sort each bucket.
(4) Copy the sorted buckets to array $A$.

## Buckets

- Q buckets
- Each bucket has <= $2n^{0.5}$ elements
- Any element in bucket $B_i$ is less than every element in bucket $B_{i+1}$
- Every bucket has an associated pivot
- To start, there is 1 bucket with a pivot of $\infty$

## Distribute Component



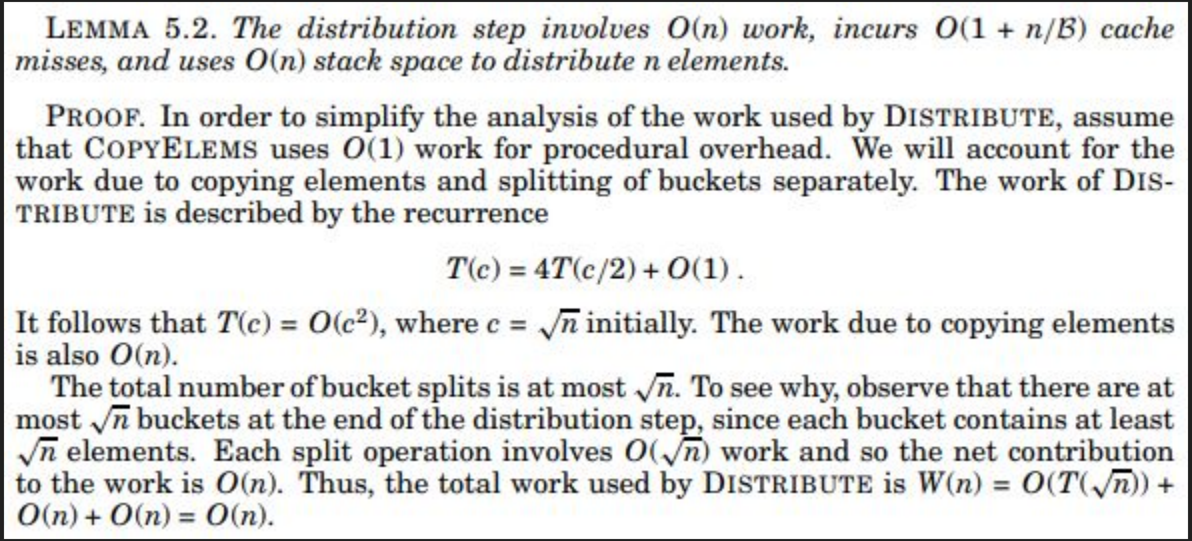
The distribution step is accomplished by the recursive procedure DISTRIBUTE($i$, $j$, $m$) which distributes elements from the $i$th through $(i + m - 1)$th subarrays into buckets starting from $B_j$. Given the precondition that each subarray $i, i+1, \ldots, i+m-1$ has its *bnum* $\geq j$, the execution of DISTRIBUTE($i$, $j$, $m$) enforces the postcondition that subarrays $i, i+1, \ldots, i+m-1$ have their *bnum* $\geq j+m$. Step 2 of the distribution sort invokes DISTRIBUTE($1, 1, \sqrt{n}$). The following is a recursive implementation of DISTRIBUTE:

**ALGORITHM:** DISTRIBUTE($i$, $j$, $m$)

```
1  if $m = 1$
2     then CopyElems($i$, $j$)
3     else Distribute($i$, $j$, $m/2$)
4          Distribute($i + m/2$, $j$, $m/2$)
5          Distribute($i$, $j + m/2$, $m/2$)
6          Distribute($i + m/2$, $j + m/2$, $m/2$)
```

In the base case, the procedure COPYELEMS($i$, $j$) copies all elements from subarray $i$ that belong to bucket $j$. If bucket $j$ has more than $2\sqrt{n}$ elements after the insertion, it can be split into two buckets of size at least $\sqrt{n}$. For the splitting operation, we use the deterministic median-finding algorithm [Cormen et al. 1990, p. 189] followed by a partition.

## Distribute Step Asymptotics, Part 1



LEMMA 5.2. *The distribution step involves $O(n)$ work, incurs $O(1 + n/\mathcal{B})$ cache misses, and uses $O(n)$ stack space to distribute $n$ elements.*

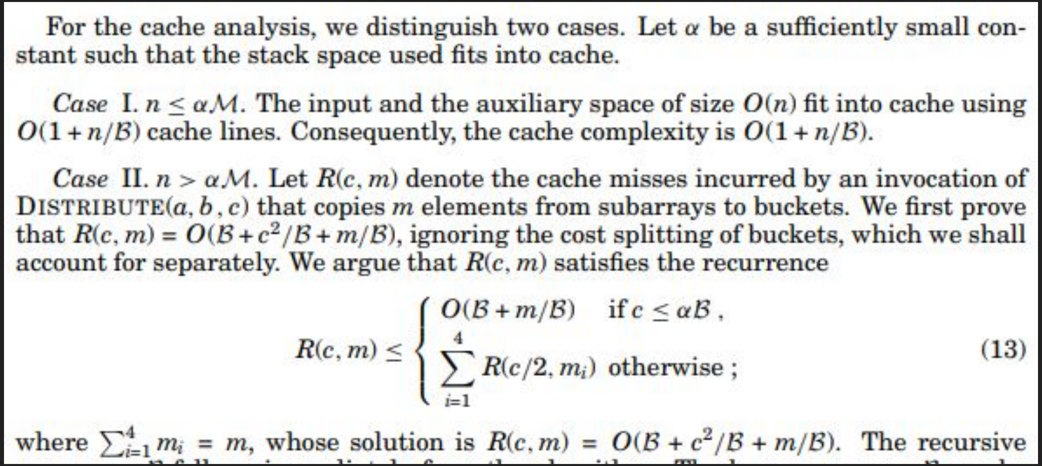
PROOF. In order to simplify the analysis of the work used by DISTRIBUTE, assume that COPYELEMS uses $O(1)$ work for procedural overhead. We will account for the work due to copying elements and splitting of buckets separately. The work of DISTRIBUTE is described by the recurrence

$$T(c) = 4T(c/2) + O(1) .$$

It follows that $T(c) = O(c^2)$, where $c = \sqrt{n}$ initially. The work due to copying elements is also $O(n)$.

The total number of bucket splits is at most $\sqrt{n}$. To see why, observe that there are at most $\sqrt{n}$ buckets at the end of the distribution step, since each bucket contains at least $\sqrt{n}$ elements. Each split operation involves $O(\sqrt{n})$ work and so the net contribution to the work is $O(n)$. Thus, the total work used by DISTRIBUTE is $W(n) = O(T(\sqrt{n})) + O(n) + O(n) = O(n)$.

## Distribute Step Asymptotics, Part 2



For the cache analysis, we distinguish two cases. Let $\alpha$ be a sufficiently small constant such that the stack space used fits into cache.

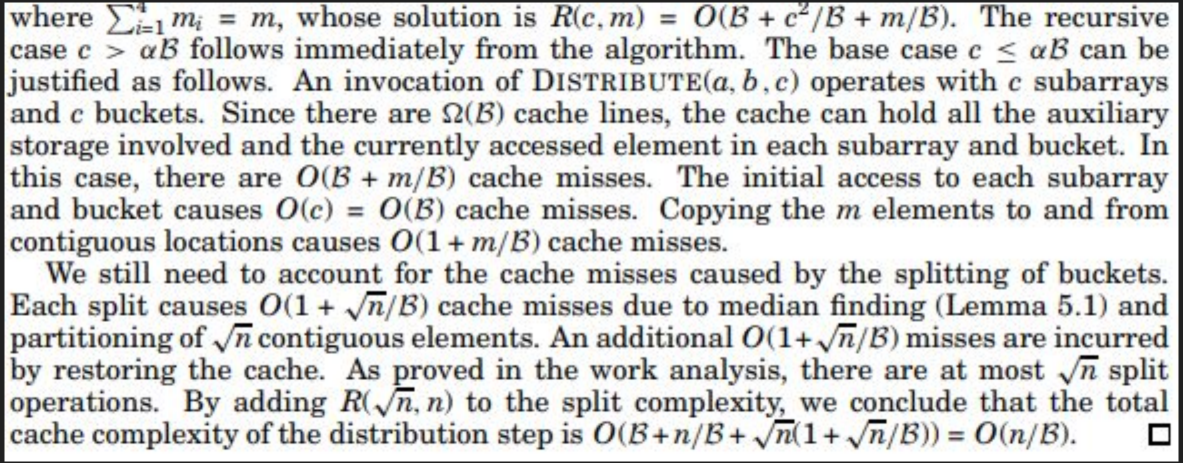
*Case* I. $n \leq \alpha\mathcal{M}$. The input and the auxiliary space of size $O(n)$ fit into cache using $O(1+n/\mathcal{B})$ cache lines. Consequently, the cache complexity is $O(1+n/\mathcal{B})$.

*Case* II. $n > \alpha\mathcal{M}$. Let $R(c,m)$ denote the cache misses incurred by an invocation of DISTRIBUTE$(a,b,c)$ that copies $m$ elements from subarrays to buckets. We first prove that $R(c,m) = O(\mathcal{B}+c^2/\mathcal{B}+m/\mathcal{B})$, ignoring the cost splitting of buckets, which we shall account for separately. We argue that $R(c,m)$ satisfies the recurrence

$$R(c,m) \leq \begin{cases} O(\mathcal{B}+m/\mathcal{B}) & \text{if } c \leq \alpha\mathcal{B}\,, \\ \sum_{i=1}^{4} R(c/2, m_i) & \text{otherwise}\,; \end{cases} \tag{13}$$

where $\sum_{i=1}^{4} m_i = m$, whose solution is $R(c,m) = O(\mathcal{B}+c^2/\mathcal{B}+m/\mathcal{B})$. The recursive

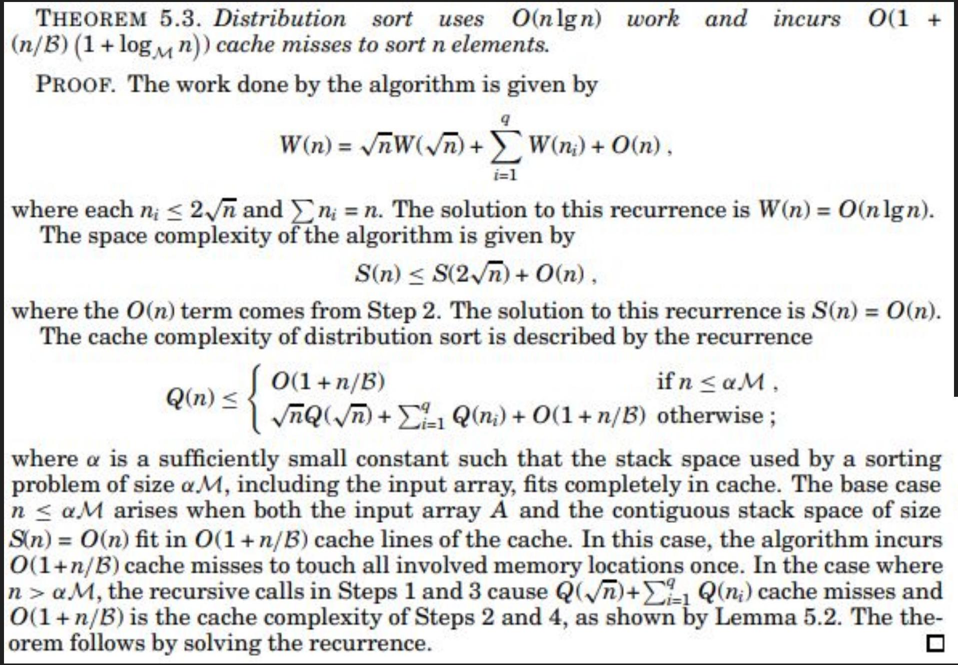
## Distribute Step Asymptotics, Part 3



where $\sum_{i=1}^{4} m_i = m$, whose solution is $R(c,m) = O(\mathcal{B} + c^2/\mathcal{B} + m/\mathcal{B})$. The recursive case $c > \alpha\mathcal{B}$ follows immediately from the algorithm. The base case $c \leq \alpha\mathcal{B}$ can be justified as follows. An invocation of DISTRIBUTE$(a, b, c)$ operates with $c$ subarrays and $c$ buckets. Since there are $\Omega(\mathcal{B})$ cache lines, the cache can hold all the auxiliary storage involved and the currently accessed element in each subarray and bucket. In this case, there are $O(\mathcal{B} + m/\mathcal{B})$ cache misses. The initial access to each subarray and bucket causes $O(c) = O(\mathcal{B})$ cache misses. Copying the $m$ elements to and from contiguous locations causes $O(1 + m/\mathcal{B})$ cache misses.

We still need to account for the cache misses caused by the splitting of buckets. Each split causes $O(1 + \sqrt{n}/\mathcal{B})$ cache misses due to median finding (Lemma 5.1) and partitioning of $\sqrt{n}$ contiguous elements. An additional $O(1+\sqrt{n}/\mathcal{B})$ misses are incurred by restoring the cache. As proved in the work analysis, there are at most $\sqrt{n}$ split operations. By adding $R(\sqrt{n}, n)$ to the split complexity, we conclude that the total cache complexity of the distribution step is $O(\mathcal{B}+n/\mathcal{B}+\sqrt{n}(1+\sqrt{n}/\mathcal{B})) = O(n/\mathcal{B})$. $\square$

# Distribution Sort Asymptotics



THEOREM 5.3. *Distribution sort uses $O(n \lg n)$ work and incurs $O(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$ cache misses to sort $n$ elements.*

PROOF. The work done by the algorithm is given by

$$W(n) = \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{q} W(n_i) + O(n) ,$$

where each $n_i \le 2\sqrt{n}$ and $\sum n_i = n$. The solution to this recurrence is $W(n) = O(n \lg n)$.

The space complexity of the algorithm is given by

$$S(n) \le S(2\sqrt{n}) + O(n) ,$$

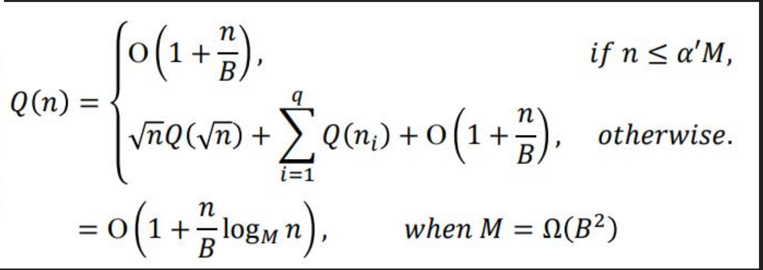
where the $O(n)$ term comes from Step 2. The solution to this recurrence is $S(n) = O(n)$.

The cache complexity of distribution sort is described by the recurrence

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

where $\alpha$ is a sufficiently small constant such that the stack space used by a sorting problem of size $\alpha\mathcal{M}$, including the input array, fits completely in cache. The base case $n \le \alpha\mathcal{M}$ arises when both the input array $A$ and the contiguous stack space of size $S(n) = O(n)$ fit in $O(1 + n/\mathcal{B})$ cache lines of the cache. In this case, the algorithm incurs $O(1 + n/\mathcal{B})$ cache misses to touch all involved memory locations once. In the case where $n > \alpha\mathcal{M}$, the recursive calls in Steps 1 and 3 cause $Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i)$ cache misses and $O(1 + n/\mathcal{B})$ is the cache complexity of Steps 2 and 4, as shown by Lemma 5.2. The theorem follows by solving the recurrence. $\square$

- O(1 + n/B) = sorting cache complexity when the array fits in memory
- $n^{0.5}Q(n^{0.5})$ is the cache of the partition step
- Sum from 1 to q of $Q(n_i)$ is the distribute and recursive sort step



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

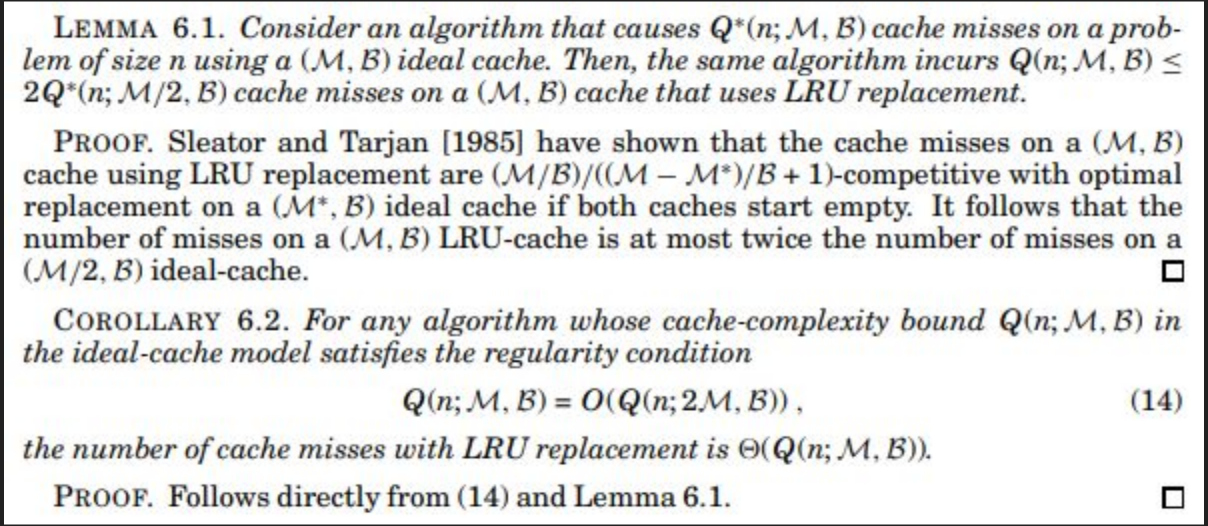
$$= O\left(1 + \frac{n}{B}\log_M n\right), \qquad \text{when } M = \Omega(B^2)$$

## Ideal Cache Model Justifications

- Ideal Cache Model has
  - Optimal Replacement
  - Two Levels of Memory
  - Automatic Replacement
  - Full Associativity
- Too strong of assumptions?

## What Happens Without Optimal Replacement?

- Overall, constant time difference between LRU and Optimal



LEMMA 6.1. *Consider an algorithm that causes $Q^*(n; \mathcal{M}, \mathcal{B})$ cache misses on a problem of size $n$ using a $(\mathcal{M}, \mathcal{B})$ ideal cache. Then, the same algorithm incurs $Q(n; \mathcal{M}, \mathcal{B}) \leq 2Q^*(n; \mathcal{M}/2, \mathcal{B})$ cache misses on a $(\mathcal{M}, \mathcal{B})$ cache that uses LRU replacement.*

PROOF. Sleator and Tarjan [1985] have shown that the cache misses on a $(\mathcal{M}, \mathcal{B})$ cache using LRU replacement are $(\mathcal{M}/\mathcal{B})/((\mathcal{M} - \mathcal{M}^*)/\mathcal{B} + 1)$-competitive with optimal replacement on a $(\mathcal{M}^*, \mathcal{B})$ ideal cache if both caches start empty. It follows that the number of misses on a $(\mathcal{M}, \mathcal{B})$ LRU-cache is at most twice the number of misses on a $(\mathcal{M}/2, \mathcal{B})$ ideal-cache. $\square$

COROLLARY 6.2. *For any algorithm whose cache-complexity bound $Q(n; \mathcal{M}, \mathcal{B})$ in the ideal-cache model satisfies the regularity condition*

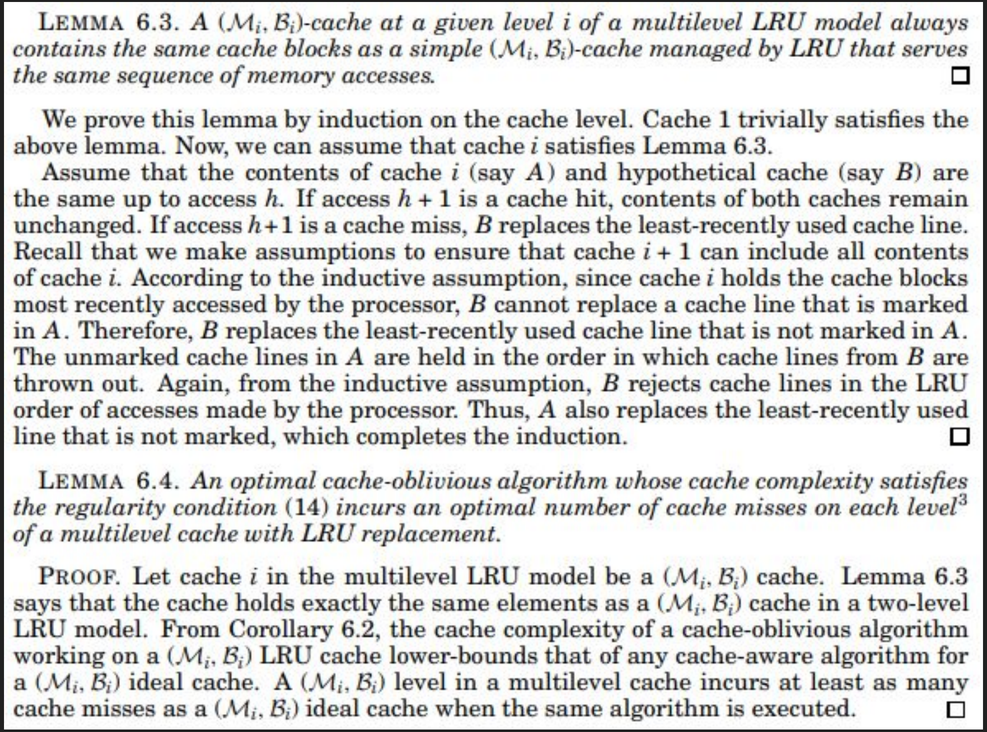
$$Q(n; \mathcal{M}, \mathcal{B}) = O(Q(n; 2\mathcal{M}, \mathcal{B})) , \tag{14}$$

*the number of cache misses with LRU replacement is $\Theta(Q(n; \mathcal{M}, \mathcal{B}))$.*

PROOF. Follows directly from (14) and Lemma 6.1. $\square$

# What Happens Without Two Levels of Memory?

- Overall, optimal cache misses on multilevel as on a two-level



LEMMA 6.3. *A $(\mathcal{M}_i, \mathcal{B}_i)$-cache at a given level $i$ of a multilevel LRU model always contains the same cache blocks as a simple $(\mathcal{M}_i, \mathcal{B}_i)$-cache managed by LRU that serves the same sequence of memory accesses.* □

We prove this lemma by induction on the cache level. Cache 1 trivially satisfies the above lemma. Now, we can assume that cache $i$ satisfies Lemma 6.3.

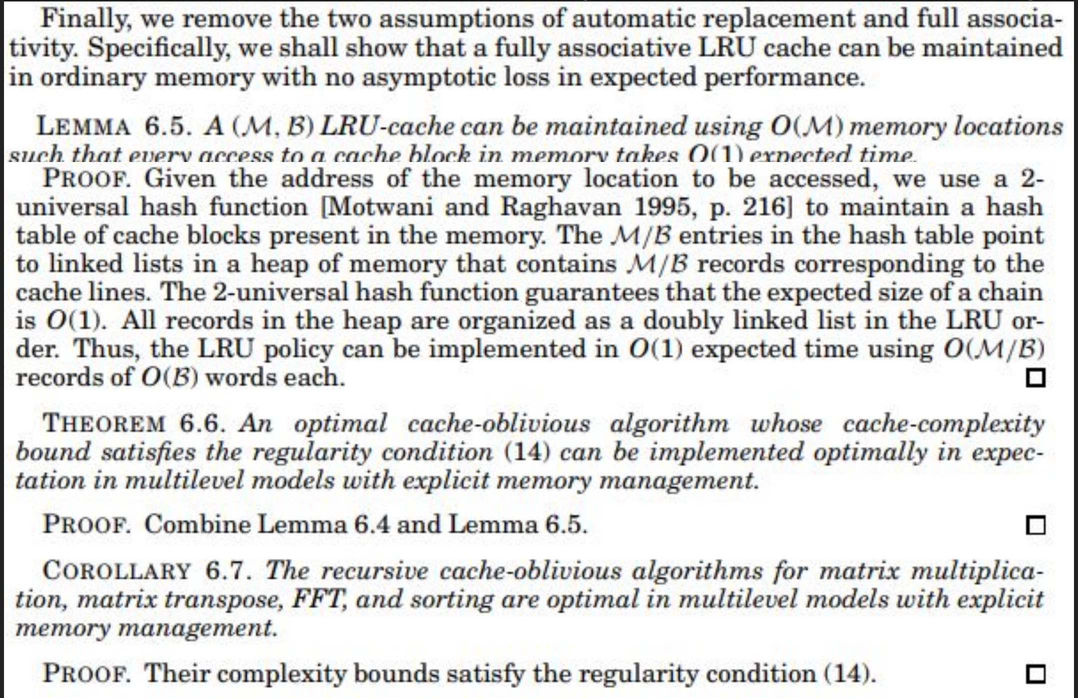
Assume that the contents of cache $i$ (say $A$) and hypothetical cache (say $B$) are the same up to access $h$. If access $h+1$ is a cache hit, contents of both caches remain unchanged. If access $h+1$ is a cache miss, $B$ replaces the least-recently used cache line. Recall that we make assumptions to ensure that cache $i+1$ can include all contents of cache $i$. According to the inductive assumption, since cache $i$ holds the cache blocks most recently accessed by the processor, $B$ cannot replace a cache line that is marked in $A$. Therefore, $B$ replaces the least-recently used cache line that is not marked in $A$. The unmarked cache lines in $A$ are held in the order in which cache lines from $B$ are thrown out. Again, from the inductive assumption, $B$ rejects cache lines in the LRU order of accesses made by the processor. Thus, $A$ also replaces the least-recently used line that is not marked, which completes the induction. □

LEMMA 6.4. *An optimal cache-oblivious algorithm whose cache complexity satisfies the regularity condition* (14) *incurs an optimal number of cache misses on each level*[3] *of a multilevel cache with LRU replacement.*

PROOF. Let cache $i$ in the multilevel LRU model be a $(\mathcal{M}_i, \mathcal{B}_i)$ cache. Lemma 6.3 says that the cache holds exactly the same elements as a $(\mathcal{M}_i, \mathcal{B}_i)$ cache in a two-level LRU model. From Corollary 6.2, the cache complexity of a cache-oblivious algorithm working on a $(\mathcal{M}_i, \mathcal{B}_i)$ LRU cache lower-bounds that of any cache-aware algorithm for a $(\mathcal{M}_i, \mathcal{B}_i)$ ideal cache. A $(\mathcal{M}_i, \mathcal{B}_i)$ level in a multilevel cache incurs at least as many cache misses as a $(\mathcal{M}_i, \mathcal{B}_i)$ ideal cache when the same algorithm is executed. □

## What Happens Without Automatic Replacement/Full Associativity?

- Overall, as long as Q(n; M, B) = O(Q(n; 2M, B)), an optimal cache oblivious algorithm can be implemented optimally with explicit memory management.



Finally, we remove the two assumptions of automatic replacement and full associativity. Specifically, we shall show that a fully associative LRU cache can be maintained in ordinary memory with no asymptotic loss in expected performance.

LEMMA 6.5. *A $(\mathcal{M}, \mathcal{B})$ LRU-cache can be maintained using $O(\mathcal{M})$ memory locations such that every access to a cache block in memory takes $O(1)$ expected time.*

PROOF. Given the address of the memory location to be accessed, we use a 2-universal hash function [Motwani and Raghavan 1995, p. 216] to maintain a hash table of cache blocks present in the memory. The $\mathcal{M}/\mathcal{B}$ entries in the hash table point to linked lists in a heap of memory that contains $\mathcal{M}/\mathcal{B}$ records corresponding to the cache lines. The 2-universal hash function guarantees that the expected size of a chain is $O(1)$. All records in the heap are organized as a doubly linked list in the LRU order. Thus, the LRU policy can be implemented in $O(1)$ expected time using $O(\mathcal{M}/\mathcal{B})$ records of $O(\mathcal{B})$ words each. □

THEOREM 6.6. *An optimal cache-oblivious algorithm whose cache-complexity bound satisfies the regularity condition* (14) *can be implemented optimally in expectation in multilevel models with explicit memory management.*

PROOF. Combine Lemma 6.4 and Lemma 6.5. □

COROLLARY 6.7. *The recursive cache-oblivious algorithms for matrix multiplication, matrix transpose, FFT, and sorting are optimal in multilevel models with explicit memory management.*

PROOF. Their complexity bounds satisfy the regularity condition (14). □

## Empirical Results

- Matrix multiplication tangible and consistent improvement over iterative version

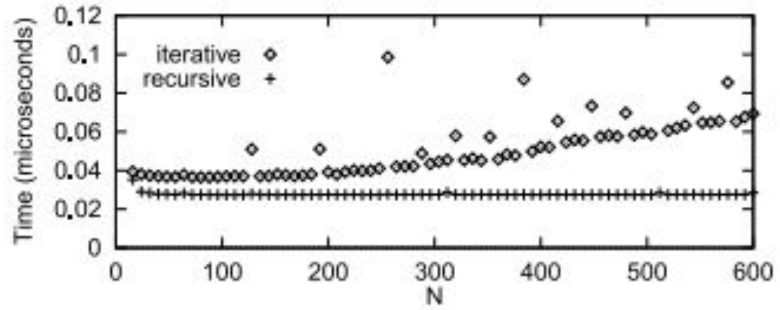


Fig. 5. Average time taken to multiply two $N \times N$ matrices, divided by $N^3$.

## Strengths and Weaknesses

- Strengths
  - Loved the progression of concepts from easier to more difficult, allows the reader to get used to the realm of study before getting into meatier proofs.
  - Thorough explanations
  - Paper is written in proper context
- Weaknesses
  - Not enough testing of all of the various algorithms spoken about, compared to cache aware and unoptimized counterparts
  - Could use a bit more imagery when explaining concepts, like the physical matrix overlaps in cache in some cases

## Thanks For Listening!

Questions?

## Bibliography

Asymptotic Cache Complexity Recurrences and Expressions images on slides 27-30, 38 - SUNY Stony Brook CSE638 Lectures 18 and 19 (https://www3.cs.stonybrook.edu/~rezaul/Spring-2013/CSE638/CSE638-lectures-18-19.pdf)

Strassen’s Algorithm Code image, Slide 13, Rec-Mult Diagram, Slide 8 - GeeksForGeeks (https://www.geeksforgeeks.org/strassens-matrix-multiplication/)

Storage Cost Diagram image, Side 2 - computationstructures.org, (https://computationstructures.org/lectures/caches/caches.html)

All other outside images and text came from the namesake paper: Cache-Oblivious Algorithms by Frigo, Leiserson, Prokop, Ramachandran