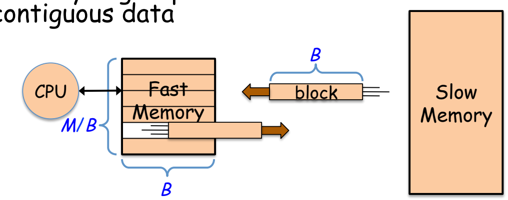
15-853: Algorithms in the Real World

Locality II: Cache-oblivious algorithms
- Matrix multiplication
- Distribution sort
- Static searching

I/O Model

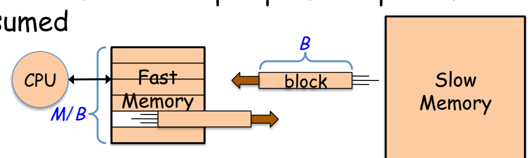
Abstracts a single level of the memory hierarchy
- Fast memory (cache) of size M
- Accessing fast memory is free, but moving data from slow memory is expensive
- Memory is grouped into size-B ***blocks*** of contiguous data



- Cost: the number of ***block transfers*** (or **I/Os**) 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



  - Optimal replacement means proofs may posit an arbitrary replacement policy, even defining an algorithm for selecting which blocks to load/evict.

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

Matrix Multiplication

Consider standard iterative matrix-multiplication algorithm

$$Z := X \; Y$$

- 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

$$Z := X \; Y$$

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

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

If $N \geq M$, no locality across iterations for X and $Y \Rightarrow \Theta(N^3)$ 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 is relatively cheap

$$Z := X \; Y$$

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

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

If $N \geq M$, no locality across iterations for X and $Y \Rightarrow \Theta(N^3/B)$

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

Recursive Matrix Multiplication

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

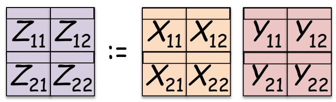
Compute 8 submatrix products recursively

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

Summing two matrices with row-major layout is cheap — just scan the matrices in memory order.

- Cost is $\Theta(N^2/B)$ I/Os to sum two $N \times N$ matrices, assuming $N \geq B$.

Recursive Multiplication Analysis

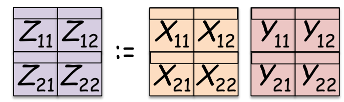


Recursive algorithm:
$Z_{11} := X_{11}Y_{11} + X_{12}Y_{21}$
$Z_{12} := X_{11}Y_{12} + X_{12}Y_{22}$
$Z_{21} := X_{21}Y_{11} + X_{22}Y_{21}$
$Z_{22} := X_{21}Y_{12} + X_{22}Y_{21}$

- $\text{Mult}(n) = 8\text{Mult}(n/2) + \Theta(n^2/B)$
- $\text{Mult}(n_0) = O(M/B)$
  when n_0 for X, Y and Z fit in memory

The big question is the base case n_0

Recursive Multiplication Analysis



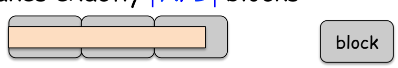
Recursive algorithm:
$Z_{11} := X_{11}Y_{11} + X_{12}Y_{21}$
$Z_{12} := X_{11}Y_{12} + X_{12}Y_{22}$
$Z_{21} := X_{21}Y_{11} + X_{22}Y_{21}$
$Z_{22} := X_{21}Y_{12} + X_{22}Y_{21}$

The big question is the base case:

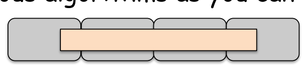
- Suppose an X, Y, and Z submatrices fit in memory at the same time
- Then multiplying them in memory is free after paying $\Theta(M/B)$ to load them into memory

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



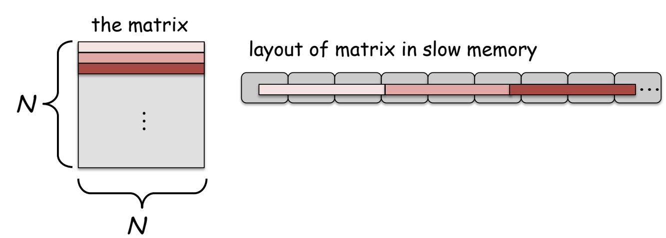
- If you're unlucky, it's $\lceil N/B \rceil+1$ blocks. This is generally what you need to assume for cache-oblivious algorithms as you can't force alignment



- In either case, it's $\Theta(1+N/B)$ blocks

Row-major matrix

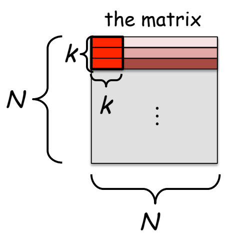
- If you look at the full matrix, it's just a single array, so rows appear one after the other

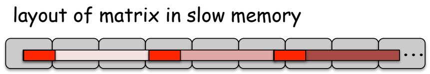


- So entire matrix fits in $\lceil N^2/B \rceil+1=\Theta(1+N^2/B)$ blocks

Row-major submatrix

- In a submatrix, rows are not adjacent in slow memory





- Need to treat this as k arrays,
- so total number of blocks to store submatrix is $k(\lceil k/B \rceil+1) = \Theta(k+k^2/B)$

Row-major submatrix

- Recall we had the recurrence
  $\text{Mult}(N) = 8\ \text{Mult}(N/2) + \Theta(N^2/B)$ (1)
- The question is when does the base case occur here? Specifically, does a $\Theta(\sqrt{M})\times\Theta(\sqrt{M})$ matrix fit in cache, i.e., does it occupy at most M/B different blocks?
- If a $\Theta(\sqrt{M})\times\Theta(\sqrt{M})$ fits in cache, we stop the analysis at a $\Theta(\sqrt{M})$ size — lower levels are free.
  i.e., $\text{Mult}(\Theta(\sqrt{M})) = \Theta(M/B)$ (2)
- Solving (1) with (2) as a base case gives
  $\text{Mult}(N) = \Theta(N^2/B + N^3/B\sqrt{M})$

load full submat in cache

Is that assumption correct?

Does a $\Theta(\sqrt{M})\times\Theta(\sqrt{M})$ matrix occupy at most $\Theta(M/B)$ different blocks?

- We have a formula from before. A $k \times k$ submatrix requires $\Theta(k + k^2/B)$ blocks,
- so a $\Theta(\sqrt{M}) \times \Theta(\sqrt{M})$ submatrix occupies roughly $\sqrt{M} + M/B$ blocks
- The answer is "yes" only if $\Theta(\sqrt{M} + M/B) = \Theta(M)$.
  iff $\sqrt{M} \leq M/B$, or $M \geq B^2$.
- If "no," analysis (base case) is broken — recursing into the submatrix will still require more I/Os.

Fixing the base case

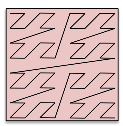
Two fixes:

1. The "<u>tall cache</u>" assumption: $M \geq B^2$.
   Then the base case is correct, completing the analysis.
2. Change the matrix layout.

Without Tall-Cache Assumption

Try a better matrix layout

- The algorithm is recursive. Use a layout that matches the recursive nature of the algorithm
- For example, Z-morton ordering:



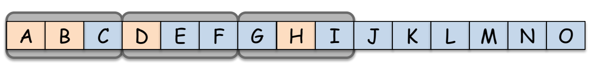
- The line connects elements that are adjacent in memory
- In other words, construct the layout by storing each quadrant of the matrix contiguously, and recurse

Recursive MatMul with Z-Morton

The analysis becomes easier

- Each quadrant of the matrix is contiguous in memory, so a $c\sqrt{M} \times c\sqrt{M}$ submatrix fits in memory
  - The tall-cache assumption is not required to make this base case work
- The rest of the analysis is the same

Searching: binary search is bad



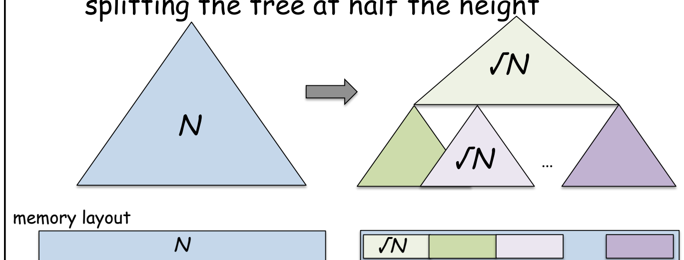
Example: binary search for element A with block size $B = 2$

- 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 $N \gg B$

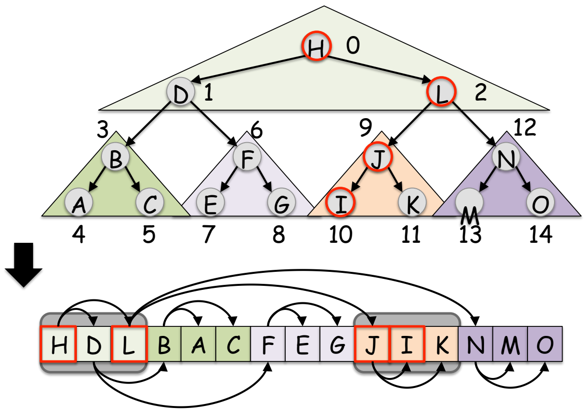
Static cache-oblivious searching

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

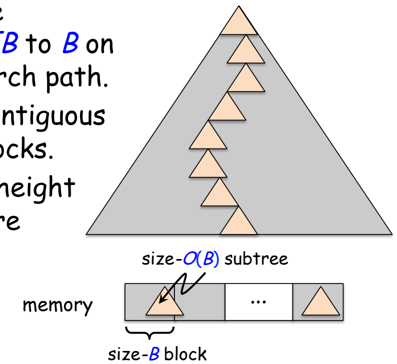


Static layout example

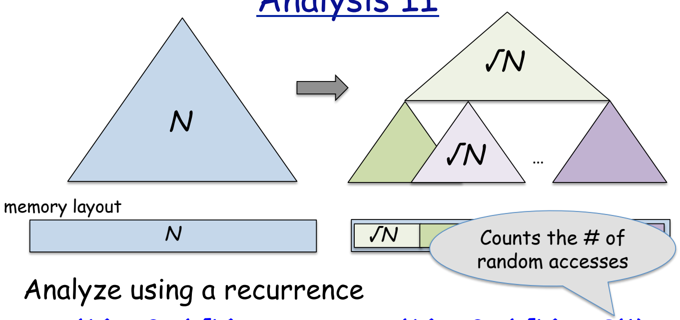


Cache-oblivious searching: Analysis I

- 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 $\Theta(\lg B)$, 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 $S(<B) = 1$.

or

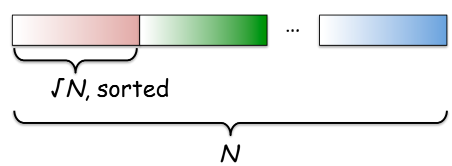
- $S(N) = 2S(\sqrt{N}) + O(1)$
- base case $S(<B) = 0$.

Solves to $O(\log_B N)$

Distribution sort outline

Analogous to multiway quicksort

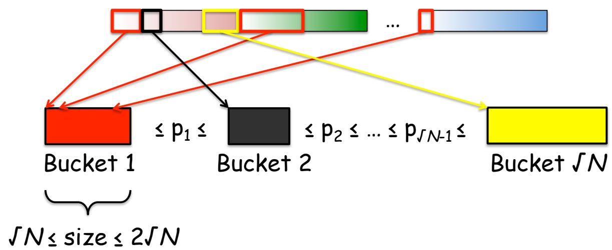
1. Split input array into $\sqrt{N}$ contiguous ***subarrays*** of size $\sqrt{N}$. Sort subarrays recursively



Distribution sort outline

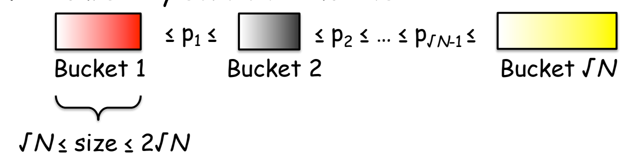
2. Choose $\sqrt{N}$ "good" pivots $p_1 \le p_2 \le \ldots \le p_{\sqrt{N}-1}$.

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



Distribution sort outline

4. Recursively sort the buckets



5. Copy concatenated buckets back to input array

Distribution sort analysis sketch

- Step 1 (implicitly) divides array and sorts $\sqrt{N}$ size-$\sqrt{N}$ subproblems
- Step 4 sorts $\sqrt{N}$ buckets of size $\sqrt{N} \le n_i \le 2\sqrt{N}$, with total size N
- Step 5 copies back the output, with a scan

Gives recurrence:

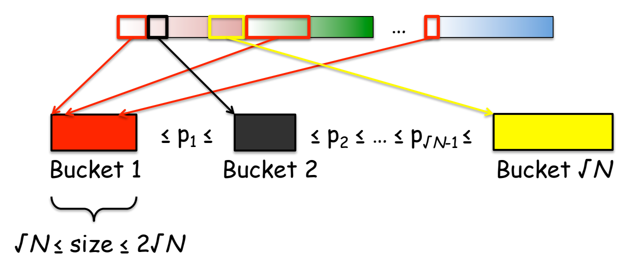
$$T(N) = \sqrt{N}\,T(\sqrt{N}) + \sum T(n_i) + \Theta(N/B) + \text{Step 2\&3}$$
$$\approx 2\sqrt{N}\,T(\sqrt{N}) + \Theta(N/B)$$

Base: $T(<M) = 1$

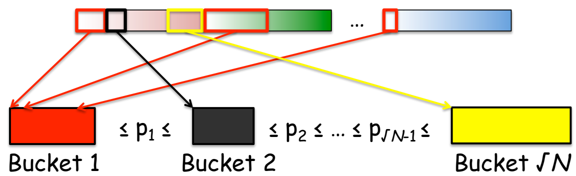
$$= \Theta((N/B)\log_{M/B}(N/B)) \text{ if } M \ge B^2$$

Missing steps

2. Choose $\sqrt{N}$ "good" pivots $p_1 \le p_2 \le \ldots \le p_{\sqrt{N}-1}$.
   (2) Not too hard in $\Theta(N/B)$
3. Distribute subarrays into ***buckets***, according to pivots

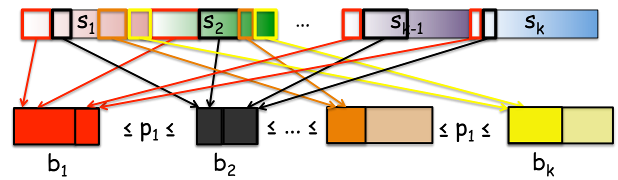


Naïve distribution



- Distribute first subarray, then second, then third, …
- Cost is only $\Theta(N/B)$ to scan input array
- What about writing to the output buckets?
  - Suppose each subarray writes 1 element to each bucket. Cost is 1 I/O per write, for N total!

Better recursive distribution



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

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

Despite crazy order, each subarray operates left to right. So only need to check next pivot.

Distribute analysis

Counting only "random accesses" here

- $D(k) = 4D(k/2) + O(k)$

Base case: when the next block in each of the k buckets/subarrays fits in memory

(this is like an M/B-way merge)

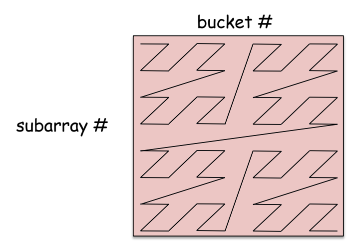
- So we have $D(M/B) = D(B)$ = free

Solves to $D(k) = O(k^2/B)$

⇒ distribute uses $O(N/B)$ random accesses — the rest is scanning at a cost of $O(1/B)$ per element

Note on distribute

If you unroll the recursion, it's going in Z-morton order on this matrix:



- i.e., first distribute s_1 to b_1, then s_1 to b_2, then s_2 to b_1, then s_2 to b_2, and so on.