Applied Algorithms

~~Lecture 6: External memory and code review~~

Lecture 6: Sorting, sorting, and more sorting

Admin

- Examples posted on site, also base cases
  - Hoping to help you avoid tedious edge case debugging!
- Previous slides updated/corrected
- Assignment 2 testing up and running (I think!)
  - detailedScriptFeedback.txt
- Questions/comments?

Today

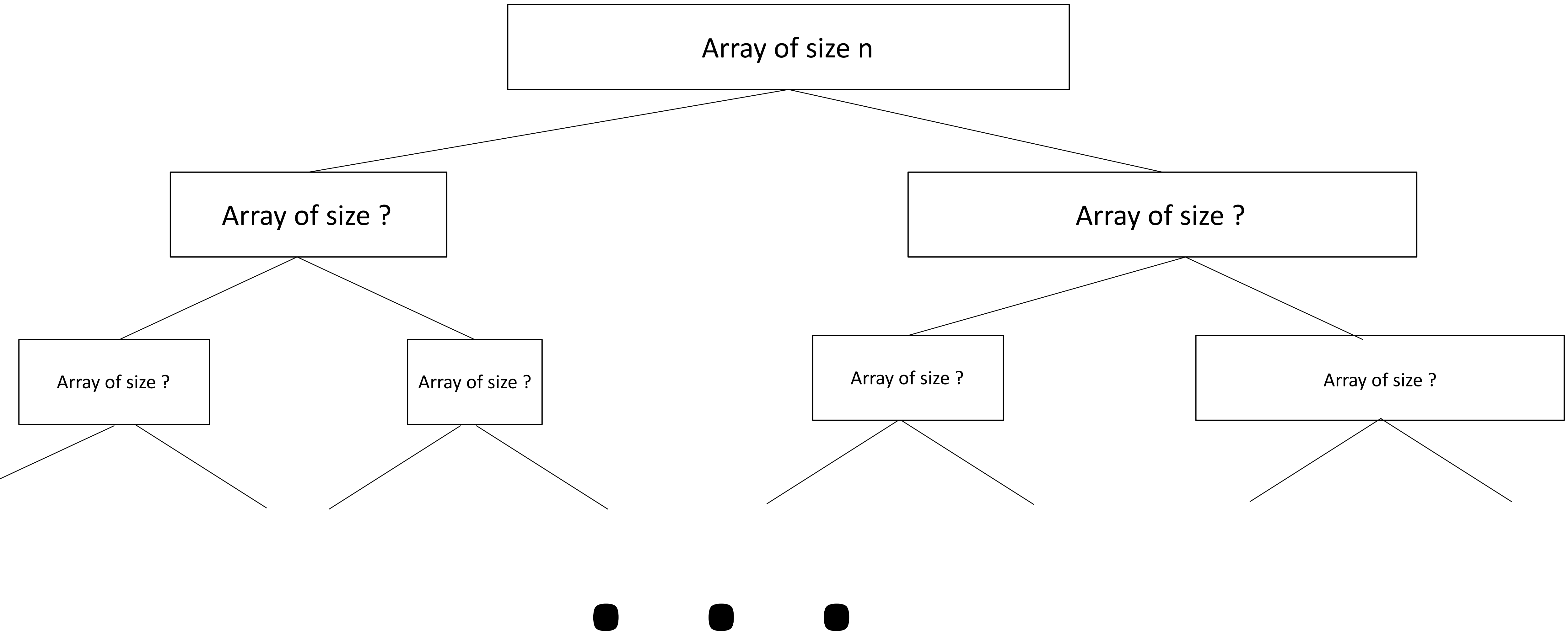
- More external memory model practice, proofs, and examples
- “Code review”: how can you make a super fast two towers executable?
  - Discussion of some really cool ideas
  - Some topics in scope of class—we’ll go into more detail

Quicksort

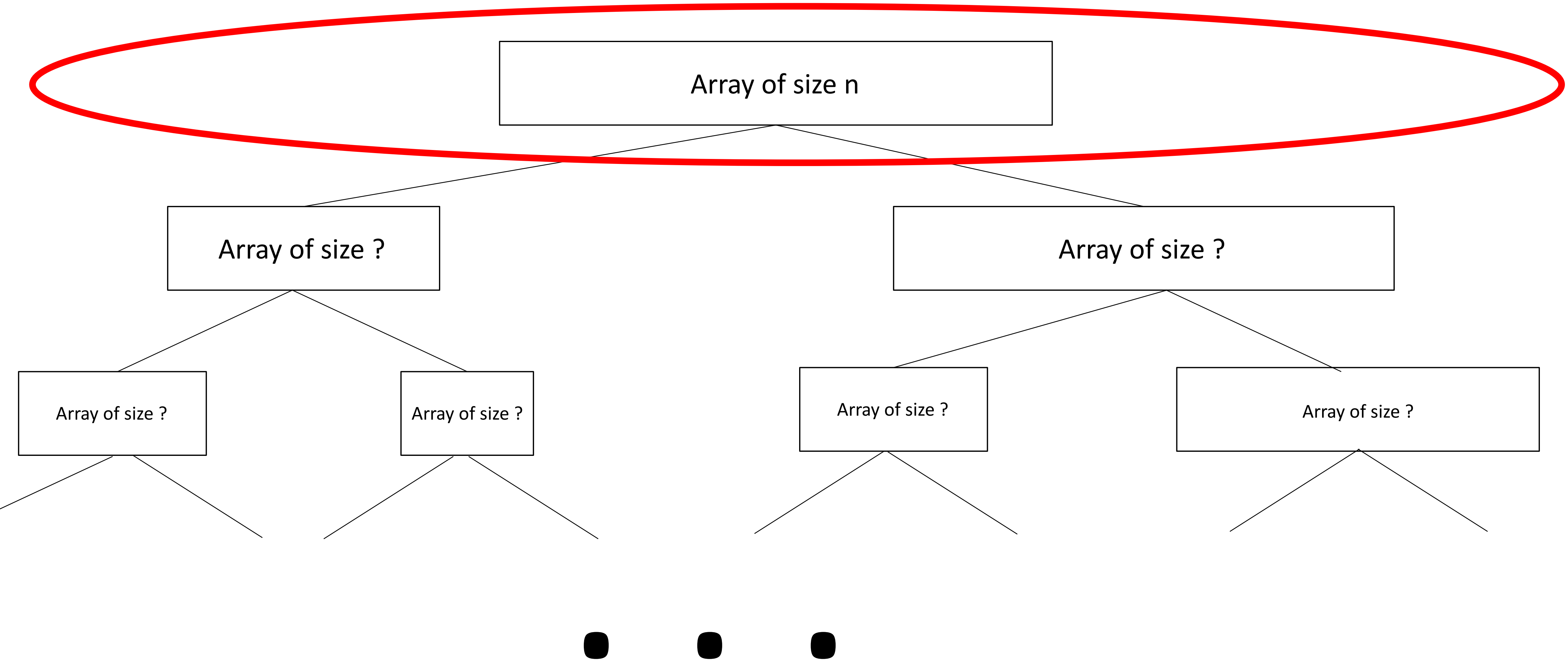
- Let's go through this formally
- **Assume:**
  - after $O(k)$ recursive calls, the input size decreases by 2^k
  - There are $O(n/S)$ calls such that this recursive call is of size $\leq S$, and the parent recursive call is of size $\geq S$
  - (Will "prove" during/after randomized algorithm analysis next week)

- How do we analyze?
  - Let's look at the recursion tree

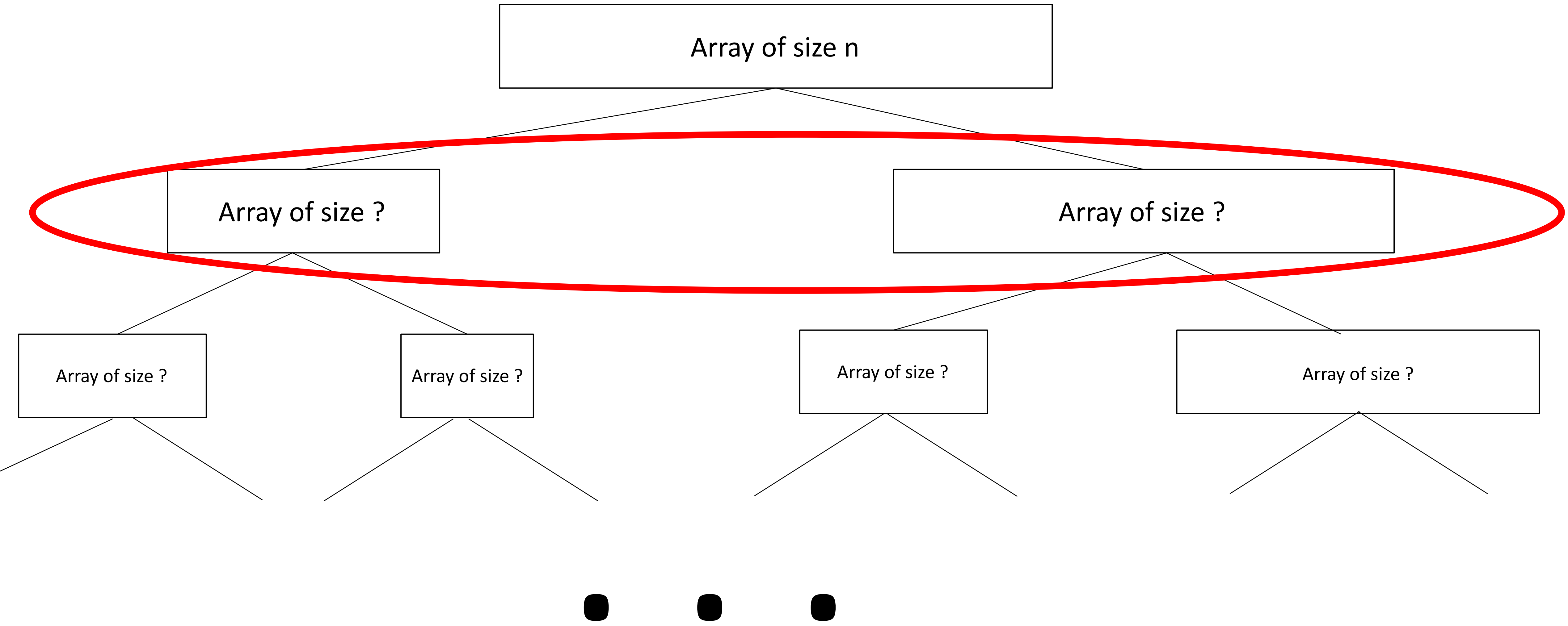
Quicksort


Array of size n
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?
. . .

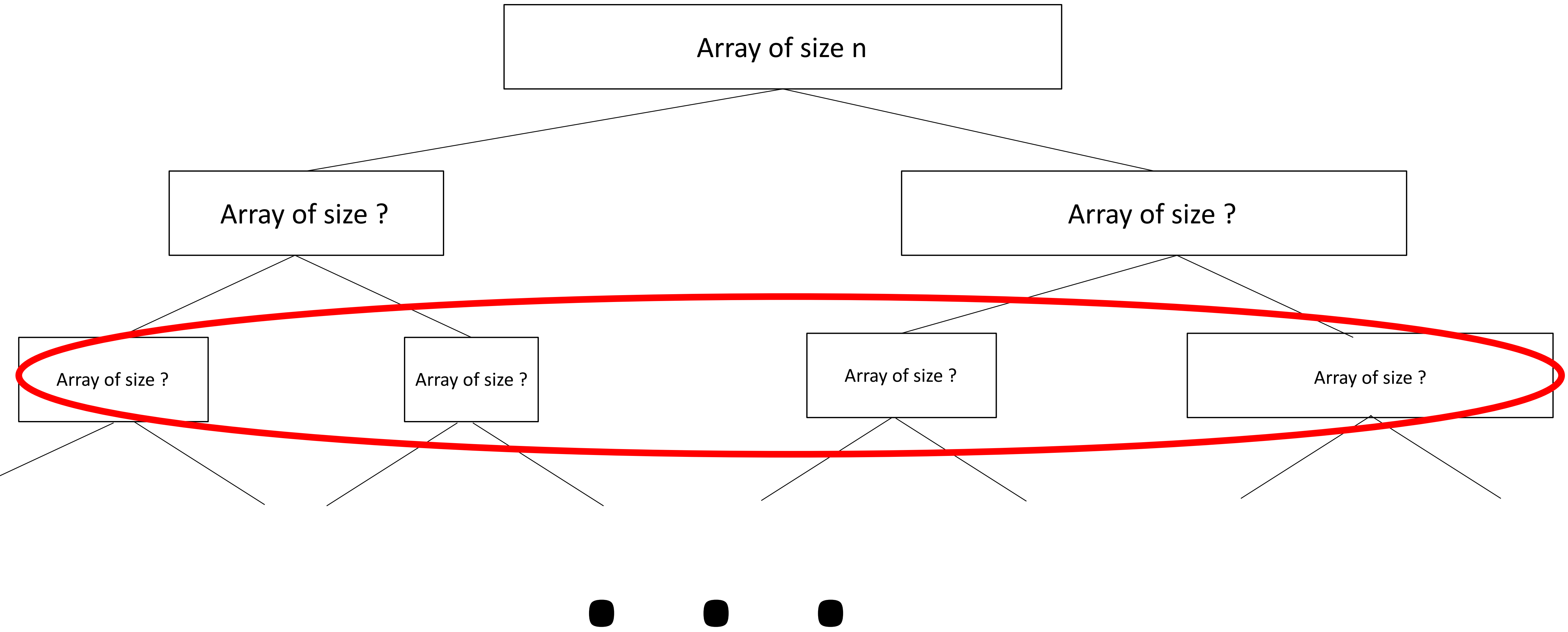
Quicksort


Array of size n
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?

Quicksort


Array of size n
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?

Quicksort



Array of size n
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?
Array of size ?

Strategy

- Sum in parts:
- What is the total cost of all leaf nodes?
- What is the cost of each level of the tree?

Leaf nodes

- If a recursive call is to an array of size $\ell_i \leq M$, then only need $O\left(1 + \frac{\ell_i}{B}\right)$ I/Os
  - Why?
  - All memory regions accessed in this subcall are to the array of size ℓ_i
  - With perfect caching, none of them will get evicted! So worst possible cost is that each block gets brought in once.
  - Why 1?
  - Because if $\ell_i < B$ the equation is not true otherwise

Leaf nodes

- We assumed that there are $O(\frac{n}{M})$ recursive calls that are of size $\leq M$ (with parent of size $> M$)

- Total size of leaf nodes: $\sum_i \ell_i = n$

- Total cost: $\sum_i \left(1 + \frac{\ell_i}{B}\right) = O\left(\frac{n}{M}\right) + O((\sum_i \ell_i)/B) = O(n/B)$

Cost of level i

- Assume that level i has subproblems of size $k_1^i, k_2^i, \ldots, k_s^i$
- Can we bound how many non-leaf subproblems there are?
  - Yes, $\leq N/M$
- What is the cost of a subproblem of size k_j^i?
  - $O(1 + \frac{k_j^i}{B})$
- What is the total size $\sum_j k_j$?
- Summing, cost of level $i = \sum_j O\left(1 + \frac{k_j^i}{B}\right) = O\left(\frac{N}{M}\right) + O\left(\frac{N}{B}\right) = O\left(\frac{N}{B}\right)$

Putting it all together

- By assumption, have $O\left(\log_2 \frac{N}{M}\right)$ levels containing non-leaf nodes
- Total cost:
- $O\left(\frac{N}{B}\right) + O\left(\frac{N}{B}\right) * O\left(\log_2 \frac{N}{M}\right) = O(\frac{N}{B}\log_2 \frac{N}{M})$

Matrix multiplication

The problem

- Given two $n{\times}n$ matrices A, B
- Want to compute their product C:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

- Example:

1	2
8	-1

$\times$

2	3
-2	7

$=$

-2	17
18	17

How do we do this?

```
for i = 1 to n
        for j = 1 to n
                for k = 1 to n
                        C[i][j] = A[i][k] + B[k][j]
```

How many I/Os does it take?

Every addition requires an I/O for B: $O(n^3)$

Can we improve this?

```
for i = 1 to n
        for k = 1 to n
                for j = 1 to n
                        C[i][j] = A[i][k] + B[k][j]
```

How many I/Os does it take?

Inner loop gets B additions per I/O: $O(n^3)/B$

I am given two functions for finding the product of two matrices:

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){
      for (int i = 0; i < n; i++)
          for (int j = 0; j < n; j++)
              for (int k = 0; k < n; k++)
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
 }

void MultiplyMatrices_2(int **a, int **b, int **c, int n){
      for (int i = 0; i < n; i++)
          for (int k = 0; k < n; k++)
              for (int j = 0; j < n; j++)
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

c algorithm matrix matrix-multiplication gprof

share improve this question

edited Sep 13 '11 at 2:47
templatetypedef
295k ● 80 ● 725 ● 933

asked Sep 13 '11 at 0:29
kevlar1818
2,639 ● 4 ● 19 ● 39

add a comment

nswers

active oldest votes

I believe that what you're looking at is the effects of locality of reference in the computer's memory

Can we improve further??

- Our goal is to perform all $O(n^3)$ multiplications with the fewest possible I/Os.

- Restated: our goal is to have each I/O result in the maximum possible number of multiplications.

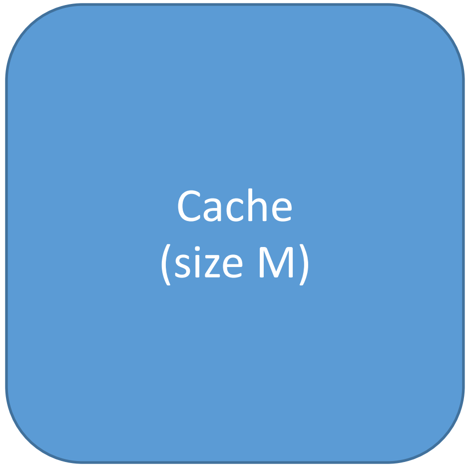
- What is the most efficient way we can use our cache???

Can we improve further??

- If we have three matrices, each of total size < M/3, we can fit them all in cache and multiply them

- How many I/Os does this take?

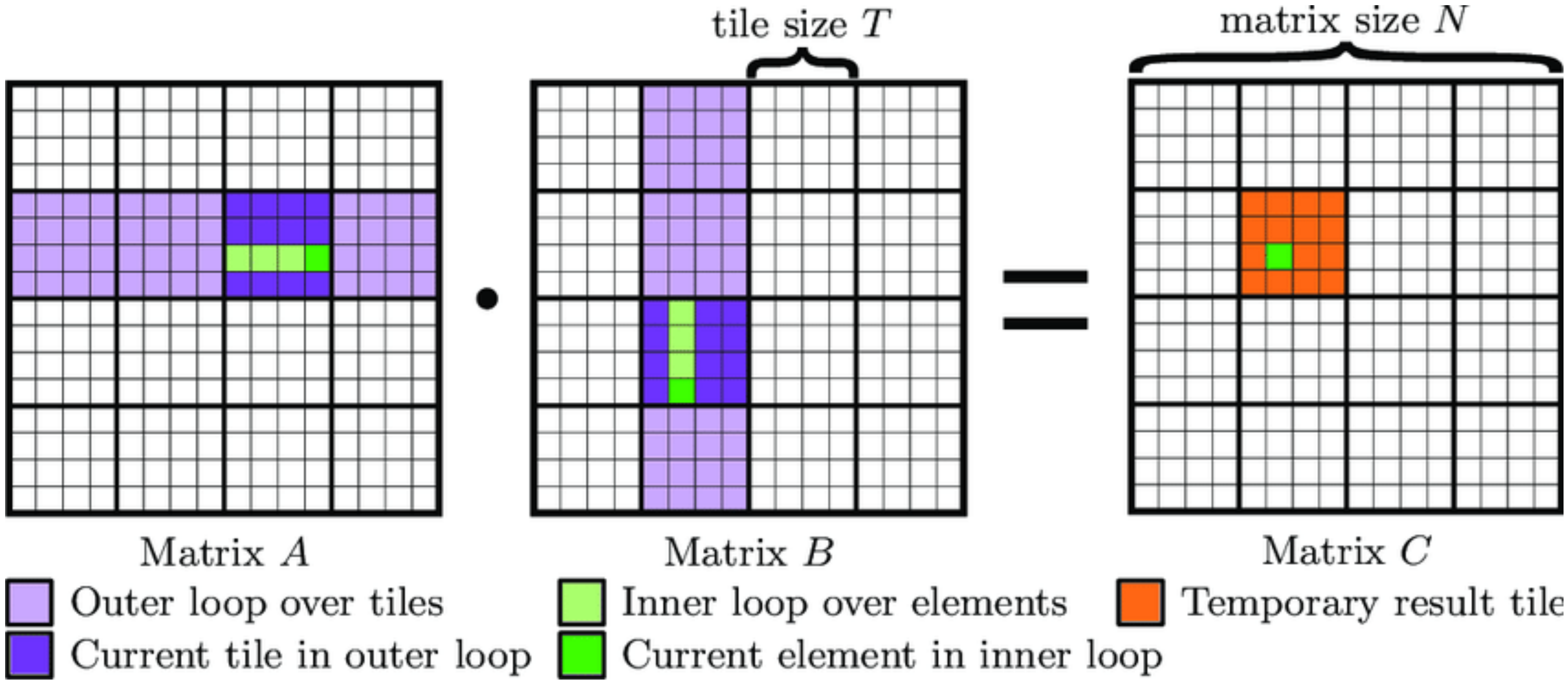
- How many multiplications do we get out of it?



How can we take advantage of this?

- Can we partition matrix multiplication into a series of multiplications of matrices of size at most M/3?

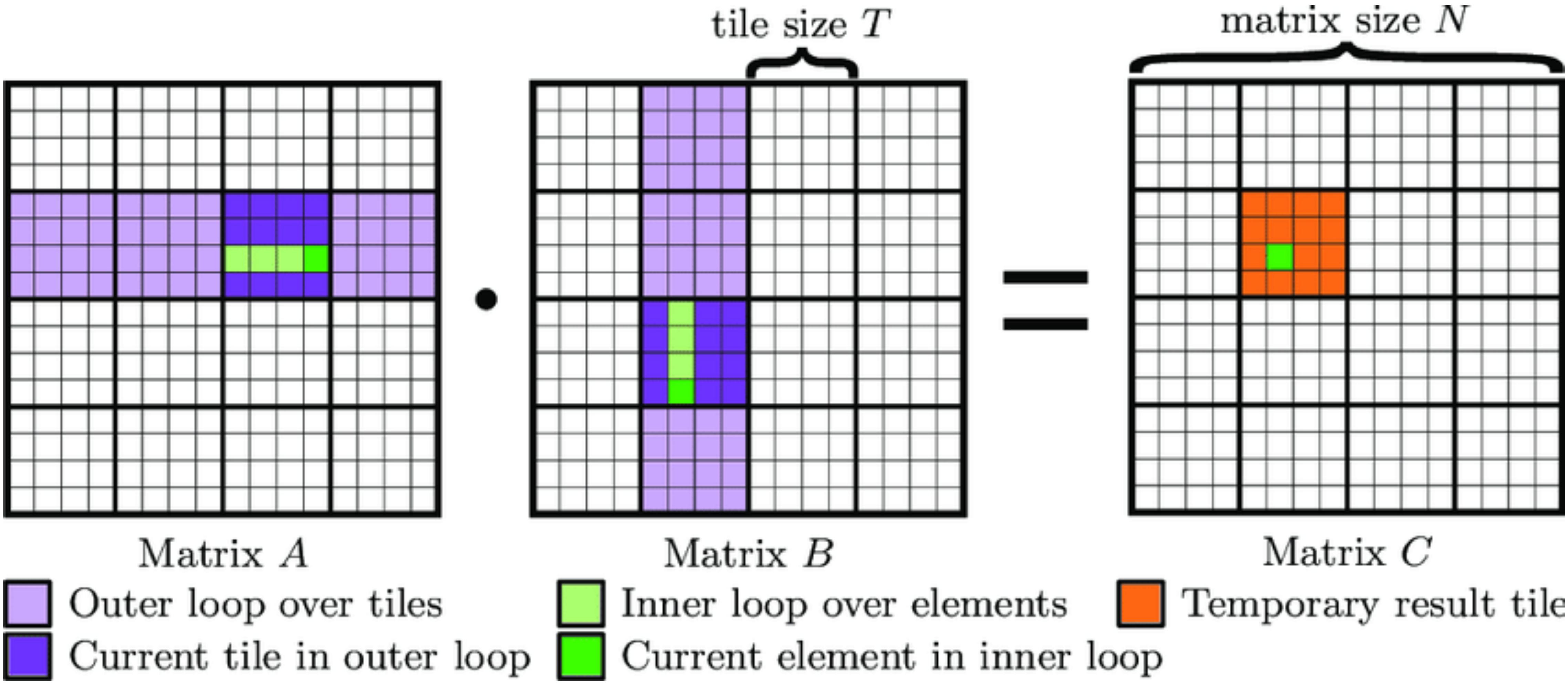
Blocking (tiling)



Blocking (tiling)

- Partition each matrix into ”tiles” (ideally, should fit in memory)
- Outer loop: perform a normal matrix multiplication of two $n/\sqrt{M} \times n/\sqrt{M}$ matrices
- Inner loop: for each tile, multiply the matrices as usual

Blocking (tiling)



Analysis

- How many tiles do we need to multiply?
  - Answer: $\frac{n}{\sqrt{M}} \times \frac{n}{\sqrt{M}} \times \frac{n}{\sqrt{M}} = \frac{n^3}{M^{3/2}}$
- How many I/Os does it take to multiply two tiles and store the result?
  - Answer: let's assume that $\sqrt{M}$ is much larger than B
  - Then we can read in each matrix in $O(\frac{M}{B})$ I/Os
- Total cost: $O(\frac{n^3}{B\sqrt{M}})$ I/Os
- Is this the entire cost?
  - Yes

External memory analysis

- This is the level of analysis I would like on your homework and exams
  - When in doubt: break into easily-digestible problems, sum their costs
  - Should be somewhat familiar expectations (from 256)
- Tiling is likely to be very very very useful in this class!

What about sorting?

- Quicksort: $O((n/B)\log(n/M))$

- Can we do better?

- What does the cache of size M get us?

Sorting really large data

- Stick to merge sort for simplicity
- Intuition: How much cache is used?
  - About 3 blocks

- Can we merge more arrays?
- How many can we merge?
  - $\sim M/B$

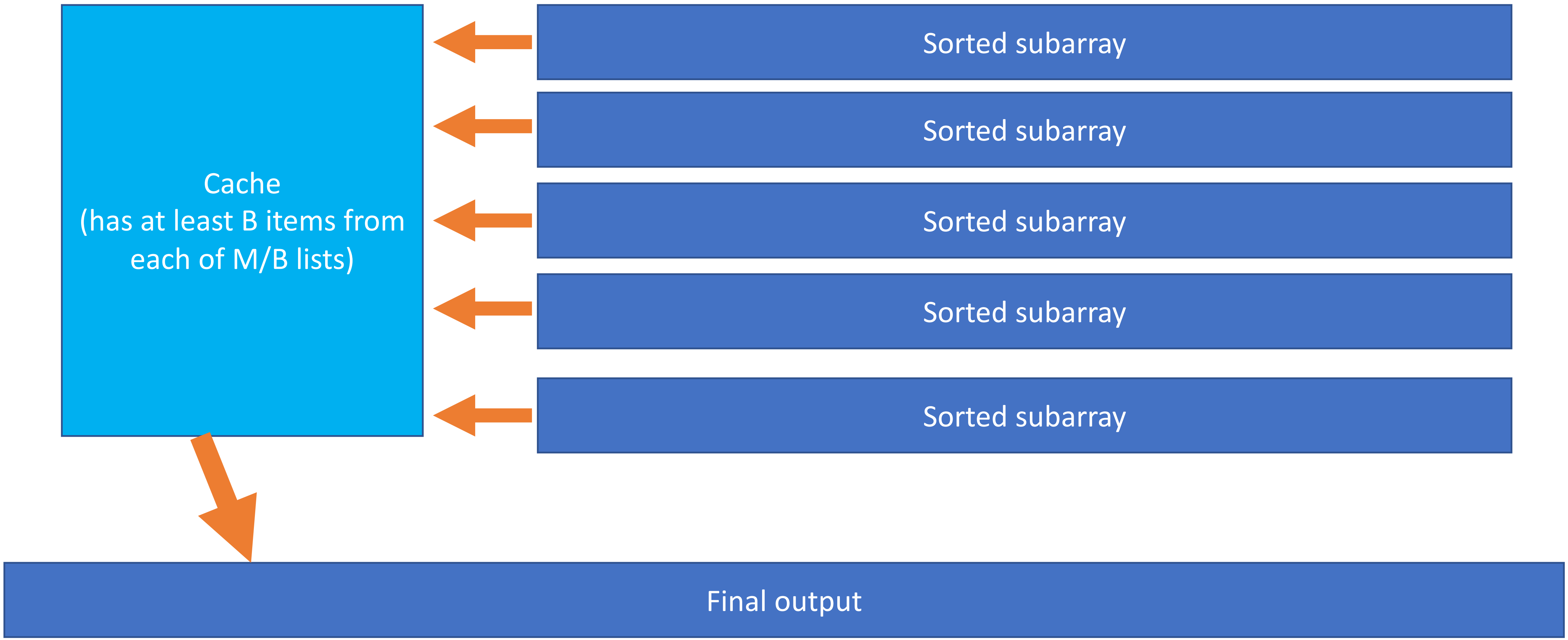
M/B-way merge sort

Algorithm:

- Split array into $M/2B$ equal-sized parts
- Recursively sort each
- Merge all $M/2B$ arrays into a final sorted array

- Cost? (Let's assume n is a power of $M/2B$, and all subarrays have size that's a multiple of B, for simplicity)
- First: how long does this big merge take?

Cost of a merge



How a merge works

- Take smallest B elements from cache, output them
- If any subarray has $\leq B$ elements in the cache, take in another B elements from that subarray.

- Analysis?
    - Total output I/Os to final array?
    - $O(k/B)$ on a subproblem of size k
    - Total input I/Os from subarray?
    - $O(\ell_i/B)$ on a subarray of size ℓ_i, so total $O(k/B)$

M/B-way merge sort

Recurrence:

- $T(n) = \frac{M}{B} T\left(\frac{nB}{M}\right) + O(\frac{n}{B})$

Solve using your favorite method

Final running time: $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$

- (Outside scope of class) Optimal!

Is this a thing?

- Yes-for large enough data
- Usually $\frac{M}{B}$ in the base of the log isn't really worth it until you get to sorting things on the hard drive

- Can we make a quicksort-like algorithm using this?
  - Yes; it's called "distribution sort"

Permutation

- Let’s say I want to shuffle my data to a particular position
  - (Like sorting, but I don’t need comparisons)
  - Think of it as sorting the numbers $1 \ldots n$

- How can I do this?

Permutation

- Computation cost?
  - $O(n \log n)$ to sort, $O(n)$ to place
- I/O cost?
  - $O(n)$ to place, $O(\frac{n}{B} \log_{\mathrm{M/B}} \frac{n}{B})$ to sort (or, $O(\frac{n}{B} \log_2 n/M)$)
- When is this better?
  - When $B > \log n$, and array is large enough that I/Os matter

Basically always true

What about trees?

- What is binary search wasting?

- How can a tree structure resolve this?

B-trees

- Branching factor of B rather than 2
- (Also have some nice balancing rules)

- Cost of searching a B-tree?
  - $O(\log_{\mathrm{B}} n/B)$

  - Is this better?   Turns out: nearly always (even if it’s just a little bigger than 2 to optimize for L1 cache)
  - (But not by too much)

Carrying back to practice

- How do we implement this? Do we plug in our best guess for M and B? What level of the hierarchy do we use?

- Cache improvement: predicted by model

- Constants: experimentation

- What should you be looking for in your code??
  - Opportunities to sort
  - Opportunities to split into moderate-sized “chunks” (both for moving data around (B), and for keeping in cache (M))

Two towers

Today

- I’ll talk about a few cool optimizations
- No one had all of these optimizations!
    - (I don’t think so at least)
- I learned a lot this lab—some of these are very clever
    - Both in idea and implementation

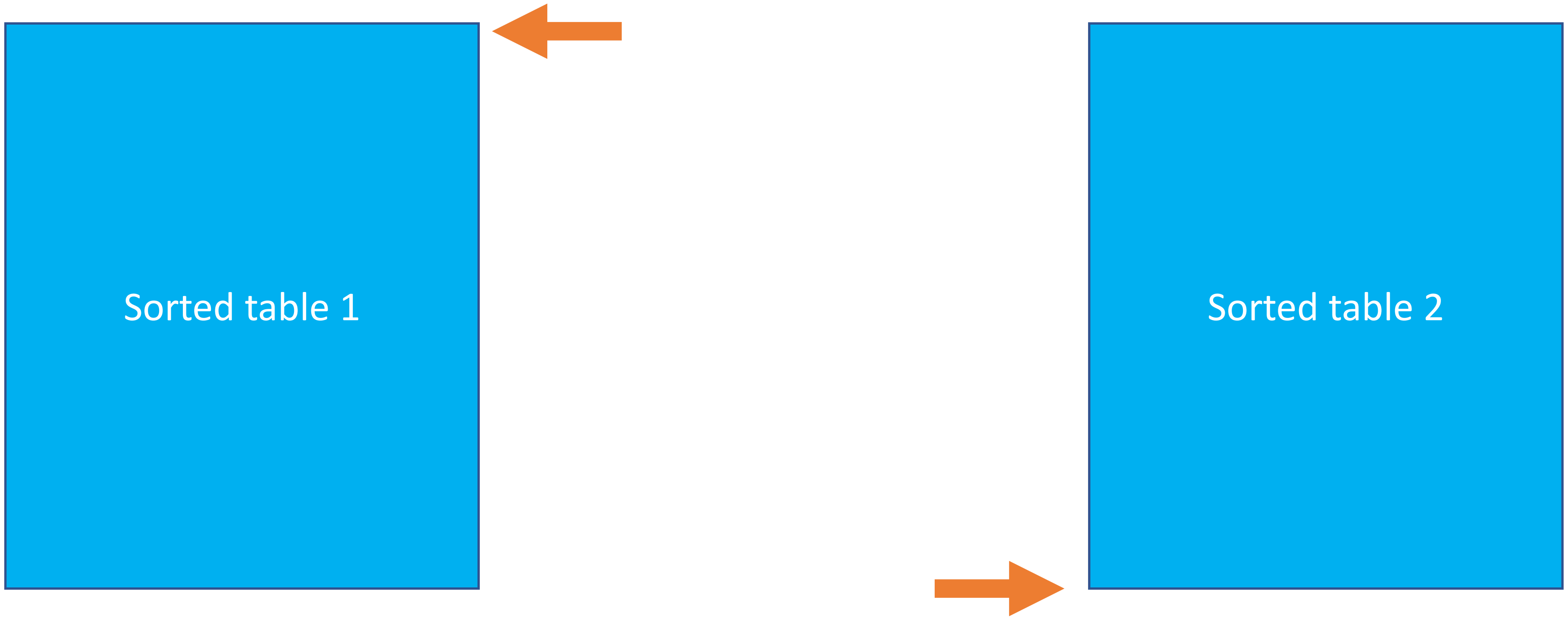
First idea: sort two tables

- We discussed last time: makes binary searches much more efficient

- But can actually improve beyond this!

Improving searches

Improving searches



Will we ever need to look at these again?

Seems like a good place to start…

Improving searches

- Just need to scan through each table once
  - Merge-like operation: move down whichever pointer keeps us under the target
- So what's our algorithm?
  - Generate tables
  - Sort tables
  - Scan through for answer

Generating tables quickly

- If we're not careful, takes $O(n)$ to generate each table entry
- (Need to add up sums)

- How can we avoid this?

Grey codes

(?)

Grey code

- Permute 1…n (n is a power of 2)
- Two successive numbers only differ in one bit position

- Example: (000, 001, 011, 010, 110, 111, 101, 100)
- (0, 1, 3, 2, 6, 7, 5, 4)

Grey codes

- Widely applicable concept!
- Useful when swapping bits has a large cost
  - That's our situation
  - Error-prone hardware
  - Useful for generating binary codes with good locality properties

Grey codes: a simple way to generate

- Simple recursive rule
- Let’s say we have a grey code G of (k-1)-bit numbers, want a grey code of k-bit numbers
- Solution:
  - prepend 0 to all numbers in G
  - Prepend 1 to all numbers in rev(G)
  - Concatenate
  - Why does this work?

Grey codes in two towers

- For each number we want to generate:
  - Find the bit to swap to get to the next number in the grey code
  - Figure out if that bit is going 0-1 or 1-0
  - Add or subtract the correct number

- Challenges?
  - Need to get bit to swap quickly
  - (And figure out which direction it’s going)
  - Floating point issues

Quickly finding bit to swap

- Binary reflective grey code has a nice property:
- To get the ith number, need to swap bit lsb(i)
- (lsb = least set bit. For example, 3 = 11, so lsb(3) = 1; 12 = 1100, so lsb(12) = 3)

- How can we do this quickly?
  - Clean loop; averages 2 iterations
  - OR: ffs()
    - Does it automatically
    - On (very) modern processors, the CPU does this in one operation!!

Comment about library calls

- ffs() tells the CPU to do it in one operation if possible
- sqrt() does this too!
  - You may have noticed that sqrt() is actually absurdly fast on testing machines

- When you have a low-level operation, check to see if C can do some low-level work for you

Optimized grey code method

- Iterate through each i
- Swap the correct bit (lsb(i)), add or subtract the corresponding input value

- Advantages?
  - 2-3 operations per new table entry
  - Definitely way better than summing from scratch each time

- Disadvantages?
  - Final table ordering is a bit arbitrary
  - Have to implement, have to deal with floating-point loss
  - One “if” per entry (? Did anyone get rid of this with a grey code method?)

Generating the table

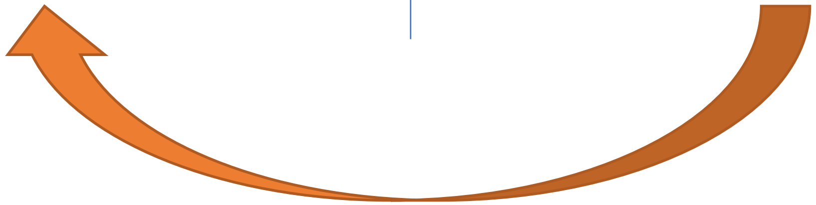
- Do we need this overhead?

0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1

Generating the table

- Do we need this overhead?

0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1



Generating the table

- To incorporate item i:
- For all table slots j from 0 to $2^i - 1$:
  - Copy slot j to slot $2^i - 1 + j$
  - Add item i to it

- (Example on board)
- Basically just a couple linear scans, no floating point problems

Two towers

- So what's our algorithm?
  - Generate tables
  - Sort tables
  - Scan through for answer

Basically just a scan or two

Sorting is now the entire problem

- qsort() is pretty slow
- Why?
  - Backend problem: C cannot inline the comparison function
  - Need to set up a function on the call stack for every comparison

- Three better options:
  - Make your own sort
  - Call C++'s sort
  - Call a library with a good sort (like timsort)

Make your own sort

- Not too hard to beat qsort by ~10-20%
- Quicksort implementation, switch to insertion sort if problem size is small
  - Why?
  - Constants
  - Nearly-sorted data

Calling C++

- C++ has std::sort()
- It’s just a good quicksort implementation
  - Optimized more than you likely have time to do
  - CAN inline comparisons! (due to improved backend)
  - About 10x faster than qsort() for simple types

- Pretty easy to call C++ from C code
  - Do need to compile with g++ though

- (Please don’t go crazy with this; make sure your code is readable in C)

Call a library

- Not many “official” sorting libraries for C
  - I don’t know why

- Only one submission got this working I think

- Some libraries have fancy sorts, like timsort

std::sort()

- Quicksorts large data
- Switches to insertion sort after a certain point

- Also: detects poor pivot performance, switches to another sort if things are going badly

- Pivot selection is implementation-dependent so far as I can tell
  - Often median-of-3

Timsort

- More recent sorting method
- On sufficiently small arrays, timsort does insertion sort

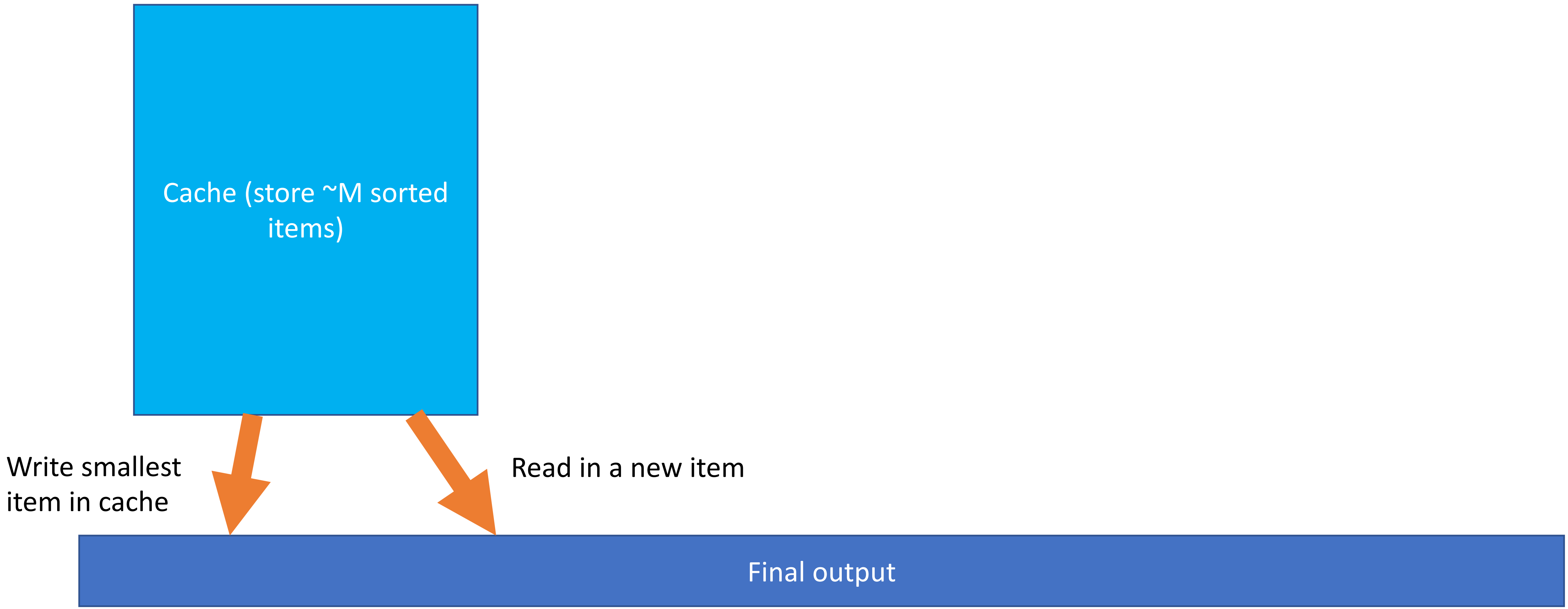
- Let’s talk about what it does otherwise

First pass: run generation

- Before sorting a large array, timsort looks through the array for big “runs” of nearly-sorted data

- What does this look like for your cache?

Run generation: cache perspective



Second step: merge runs

- Timsort then takes these large-ish runs and merges them together
- Carefully selects merges
  - Merging different-sized arrays is not too helpful

- Merging has a first step of binary search
  - Often, one array is strictly bigger

- One more optimization: big step, small step

Big step, small step

- Let’s say we’re merging two arrays, and we’ve passed a large number of items in the smaller array
- One option: binary search for the next place
  - $\log n$ time
- Can we do better?
- Repeatedly double the size of each step, then binary search
  - If we want to skip forward k, this takes $O(\log k)$

Timsort

- Performs much better than quicksort on almost-sorted data

- So if we want to sort really fast:
  - Our starting tables should be somewhat sorted
  - Then perform an “adaptive” sort like timsort

- How can we guarantee somewhat-sorted tables?
  - Sort input

Final optimization

- We are searching for the optimal smaller tower
- Idea: instead, search for the tower closest to the target (above or below) that contains the first item
  - Advantage: one table becomes half as big
  - Disadvantage: binary search needs to be “closest” instead of predecessor