Applied Algorithms Lec 6: External Memory; Optimization

Sam McCauley

September 24, 2024

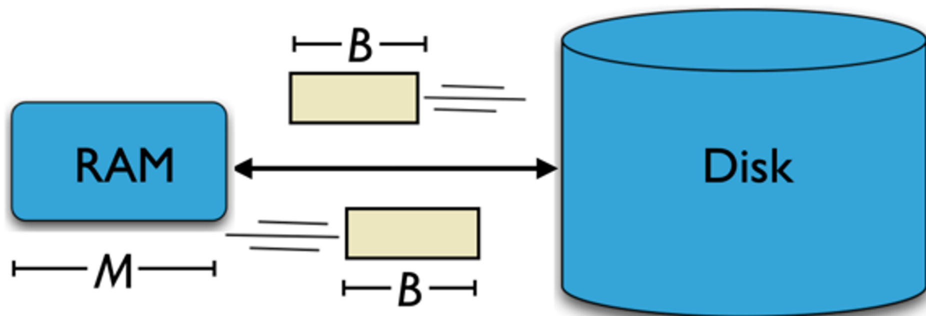
Williams College

Admin

- Any questions about Homework 2?
- Heads up: if your program outputs more than $\approx$ 20 GB of text, I can't run it. So you won't get feedback if that is happening
- Homework 1 should be graded before Thursday
- Today: finish up the external memory model; discuss Homework 1 approaches and some more optimization

Matrix Multiplication in External Memory

External Memory Model Basics



Transferring B *consecutive* items to/from cache costs 1 “cache miss”. Can only store M things in cache. Computation is free.

Matrix Multiplication Reminder(?)

- 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:

$$\begin{bmatrix} 1 & 2 \\ 8 & -1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -2 & 7 \end{bmatrix} = \begin{bmatrix} -2 & 17 \\ 18 & 17 \end{bmatrix}$$

First Idea: Compute Product Directly

```
1  for i = 1 to n:
2    for j = 1 to n:
3      for k = 1 to n:
4        C[i][j] += A[i][k] *
             B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M < n^2$ Then all fits in cache; $O(n^2/B)$ cache misses
  - What if $M > n^2$?
  - Answer: $O(n^3)$ cache misses. Every operation requires a cache miss for B.

Any ideas for how to improve this?

- One idea: transpose *B* (store in column-major order)
  - A good idea; works well! A bit nontrivial, especially if you want the transposition to be cache-efficient
- Another idea: swap the loops! How many cache misses is this?

```
1  for i = 1 to n:
2    for k = 1 to n:
3      for j = 1 to n:
4        C[i][j] += A[i][k] + B[k][j]
```

Any ideas for how to improve this?

```
1  for i = 1 to n:
2    for k = 1 to n:
3      for j = 1 to n:
4        C[i][j] += A[i][k] + B[k][j]
```

- This gives us $O(n^3/B)$ cache misses: (we'll do the math on the board; assume $B < n$ to make things easier)
- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.
- Let's say $C[i][j]$ is a cache miss. No more cache misses until $C[i][j']$ with $j' = j + B$.

Any ideas for how to improve this?

```
1  for i = 1 to n:
2    for k = 1 to n:
3      for j = 1 to n:
4        C[i][j] += A[i][k] + B[k][j]
```

- This gives us $O(n^3/B)$ cache misses
- Question: Is this worth doing?

Yep!

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

Swapping MM Loops

```
1 for i = 1 to n:
2   for k = 1 to n:
3     for j = 1 to n:
4       C[i][j] += A[i][k] + B[k][j]
```

- `-O3` optimization of `gcc` actually tries to do this automatically (Very cool)

Can we do even better?

- Idea: we haven't used the cache yet
- No Ms in any running times—except when the whole problem fits in cache
- Why? All algorithms so far have read the data once and then thrown it away.
- Goal: bring items into cache so that we can perform *many* computations on them before writing them back.
- Note: can't do this with linear scan. $O(n/B)$ is optimal. But we did do this with `smallunsortedlinkedlist.c`

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember: cache efficiency can get WAY better when the problem fits in cache. Let's find subproblems that can fit in cache.
- Idea: break problems into subproblems of size $O(M)$
  - Can solve any such problem in $O(M/B)$ cache misses
  - Efficiently combine them for a cache-efficient solution

Blocked Matrix Multiplication

- Split matrices A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$ blocks
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now so there's no rounding
- Multiply blocks one at a time
- Need some structure to help us make this work

Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

- This idea is used in recursive matrix multiplication
- And Strassen's algorithm for matrix multiplication

Decomposing matrices into blocks

Example: Recall how to multiply $2x2$ matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

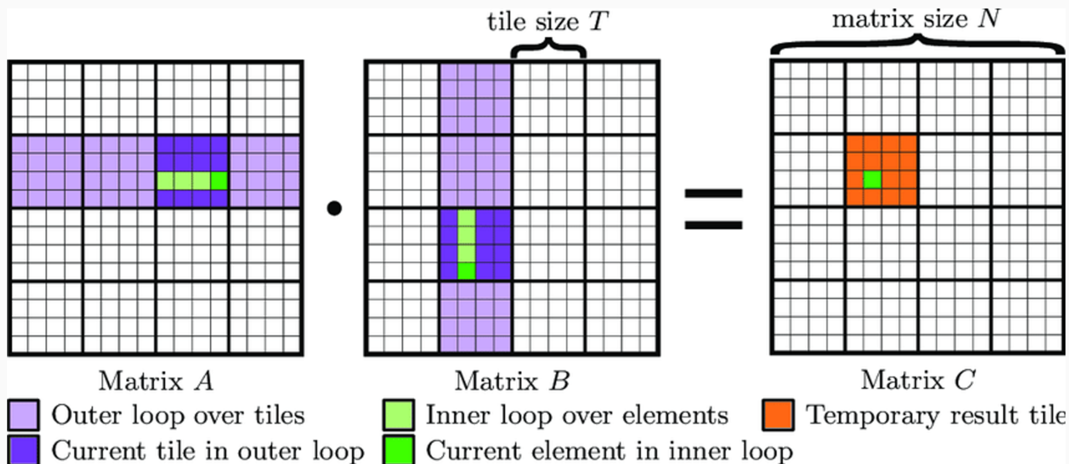
We can use this principle to multiply two larger matrices.

$$\begin{bmatrix} 17 & 15 & 20 & 4 \\ 15 & 3 & 20 & 8 \\ 1 & 10 & 15 & 2 \\ 3 & 19 & 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 & 9 & 1 \\ 4 & 6 & 11 & 2 \\ 13 & 18 & 8 & 20 \\ 3 & 11 & 18 & 9 \end{bmatrix} =$$

$$\begin{bmatrix} \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \end{bmatrix}$$

Blocked Matrix Multiplication

- Decompose matrix into blocks of length T (recall that $T^2 \leq M/3$)
- Do a normal $n/T \times n/T$ matrix multiplication



Blocked Matrix Multiplication Pseudocode

```
1   MatrixMultiply(A, B, C, n, T):
2       for i = 1 to n/T:
3         for j = 1 to n/T:
4           for k = 1 to n/T:
5             A′ = TxT matrix with upper left corner A[Ti][Tk]
6             B′ = TxT matrix with upper left corner B[Tk][Tj]
7             C′ = TxT matrix with upper left corner C[Ti][Tj]
8             BlockMultiply(A′, B′, C′, T)
9
10  BlockMultiply(A, B, C, n):
11      for i = 1 to n:
12          for j = 1 to n:
13              for k = 1 to n:
14                  C[i][j] += A[i][k] + B[k][j]
```

Let’s analyze the cost of this algorithm in the EM model together on the board!

Analysis (for future reference)

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.
- `BlockMultiply` only accesses elements of A', B', C'. Since all three matrices are in cache, it requires zero additional cache misses
- Therefore, our total running time is the number of loop iterations times the cost of a loop. This is $O((n/T)^3 \cdot T^2/B) = O((n/\sqrt{M})^3 \cdot M/B) = O(n^3/B\sqrt{M})$.

Implementation questions!

- What do we do if n is not divisible by T?
  - Easy way to implement: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well
- How do we figure out M? We don't have a two-level cache and we're ignoring that space is used for other programs, other variables, etc.
  - Experiment! Try different values of M and see what's fastest on a particular machine.
- Is blocking actually worthwhile?
  - Yes; it is used all the time to speed up programs with poor cache performance.
  - (Not a panacea; some programs (like linear scan, binary search) can't be blocked.)

More Optimization (and Homework 1 Review)

Plan for this topic

- First, talk about how various techniques can make code more efficient
  - ...or less efficient
- Focus on loops, and on compiler options
- Then, look back a bit at Homework 1. Talk about various strategies, and what some final products looked like
  - May continue this a bit Friday if we run out of time

Taking out expensive operations

```
1       for(int i = 0; i < strlen(str1); i++){
2           str1[i] = 'a';
3       }
```

- What's wrong with this code? How long does it take?
- Does the compiler optimize this out?

- It *can't*:[1] we're changing the array, which could change the location of the first 0.

[1] Of course, we know that we're never setting any values to 0 before checking them, but the compiler doesn't check for that.

More subtle issues

```
int len = strlen(str1);
for(int i=0; i < len; i++){
    str1[i] = str1[0];
}
```

```
int len = strlen(str1);
int start = str1[0];
for(int i=0; i < len; i++){
    str1[i] = start;
}
```

- Version on the right runs 2-3x faster even with optimizations on
- Why is that?
- Don’t need to look up value! (Compiler doesn’t know it doesn’t change after the first iteration)

Theme of user optimizations vs compiler optimizations

- The compiler will do the best optimizations it can that work for *all* code
- Bear in mind: only common optimizations are implemented
- Opportunities for you: what do you know about your data, and about your methodology, that allows for further efficiency?

Loop Unrolling

- Classic technique to improve loop efficiency

- What are the costs of each iteration of a simple for loop?

```
1  for(int x = 0; x < 1000; x++){
2      total += array[x];
3  }
```

Loop Unrolling

```
for(int x = 0; x < 1000; x++){
    total += array[x];
}
```

- Need to do a branch every loop
- Instruction pointer jump every loop (cost of “jumping back” varies; outside scope of course)
- Need to compare every loop
- Need to increment every loop

Unrolled Loop

```
for(int x = 0; x < 1000; x+=5){
    total += array[x];
    total += array[x+1];
    total += array[x+2];
    total += array[x+3];
    total += array[x+4];
}
```

- In short: repeat body of the loop multiple times.
- What does this gain us?

Unrolled Loop

```
for(int x=0; x < 1000; x++){
    total += array[x];
}
```

- Branch every loop
- Instruction pointer jump every loop
- Compare every loop
- Increment every loop

```
for(int x=0; x<1000; x+=5){
    total += array[x];
    total += array[x+1];
    total += array[x+2];
    total += array[x+3];
    total += array[x+4];
}
```

- Branch every 5 loops
- Instruction pointer jump every 5 loops
- Compare every 5 loops
- Increment every 5 loops (?) Need to do some *extra* additions however

What did we need to know to make this substitution?

- Needed array size to be a multiple of 5
- Can get around this with some extra work

Disadvantages of Loop Unrolling?

- Seems like we break even at worst?
- But: Loop unrolling increases code size
- Can hurt performance if important parts of code no longer fit in cache
- *Fetching instructions* can require cache misses!
  - We saw this in the output of `cachegrind`

Automatic loop unrolling?

- Why can't `gcc` unroll our loops?
- It can!
- Need to turn on specifically (not enabled at *any* optimization level)

`-funroll-loops`
Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`. This option makes code larger, and may or may not make it run faster.

`-funroll-all-loops`
Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. `-funroll-all-loops` implies the same options as `-funroll-loops`,

- `-O3` does a specific kind of unrolling of nested loops

Compiler optimizations?

- We've stumbled upon a classic (and thematic) problem in optimization: time vs space of the machine code itself
- Many optimizations of code reduce the number of operations (or their total time), but increase the size of the code itself—potentially leading to cache misses

Revisiting compiler flags

- `-O0`: No optimizations
- `-O1`: Some optimizations; may take longer to compile than `-O0`
- `-O2`: Turns on "nearly all" optimizations *that do not involve a space-time tradeoff*
- `-O3`: More optimizations. May lead to larger final programs
- `-Ofast`: Even more optimizations. Most notable is reordering floating point operations (can lead to correctness issues)

Optimizations and this course

- Our projects generally involve really small programs. This is why the very optimized versions tend to work well for your code.
- Not advised in general
- Example: Gentoo user manual. (Gentoo is a linux distribution in which *all* software is compiled from scratch. So this is advice for people compiling large software like the linux kernel, chromium, libreoffice, etc. (as well as, of course, very small utilities like git))

Gentoo optimization advice

- `-O1`: the most basic optimization level. The compiler will try to produce faster, smaller code without taking much compilation time. It is basic, but it should get the job done all the time.
- `-O2`: A step up from `-O1`. The *recommended* level of optimization unless the system has special needs. `-O2` will activate a few more flags in addition to the ones activated by `-O1`. With `-O2`, the compiler will attempt to increase code performance without compromising on size, and without taking too much compilation time. SSE or AVX may be utilized at this level but no YMM registers will be used unless `-ftree-vectorize` is also enabled.
- `-O3`: the highest level of optimization possible. It enables optimizations that are expensive in terms of compile time and memory usage. Compiling with `-O3` is not a guaranteed way to improve performance, and in fact, in many cases, can slow down a system due to larger binaries and increased memory usage. `-O3` is also known to break several packages. Using `-O3` is not recommended. However, it also enables `-ftree-vectorize` so that loops in the code get vectorized and will use AVX YMM registers.

One more common optimization with a time-space tradeoff

- We’ve talked about how costly it is to call a function
- Well, most of the time, we don’t really *need* function calls at all, do we? If the function doesn’t call another function, can just put the code for the function directly into the code
- Called *function inlining*
- Tradeoff?

Function Inlining

- Can do it yourself. May not be a good idea. (Makes code harder to read.)
- `gcc` will judge each function for you and inline it if `gcc` thinks it's a good idea (flag to get `gcc` to do this is `-finline-functions`; it is turned on with `-O2`)
- Can use `inline` keyword. `gcc` will try particularly hard to inline it for you, and if it can't will tell you if you have `-Winline` flag on
  - Can use `__inline__`; does the same thing. Some compilers may like this better
  - Probably want to always use `static inline`
- Can also use `__attribute__((always_inline))` which really forces it to inline even if optimizations are turned off

One more optimization flag

- `-march=native`
- tells `gcc` to use instructions specific to this processor. May increase speed
- Only disadvantage: your compiled binary may not run on other computers unless they have an identical processor (this is not a problem for us!)

Looking Back at Homework 1

Some comments

- Lots of great submissions!
- It seems that algorithmic improvements are more important than engineering improvements for Homework 1
- (I believe the reverse is the case for Homework 2)
- Obviously most of you did not implement these. But there are some interesting lessons in terms of optimization!

Leaderboard at the end

CSCI 358 - Fall 2024

Applied Algorithms

Home | Lectures | Assignments | Handouts | Leaderboard | CS@Williams

Homework1

Last Updated Sep 19 22:27

1	Best Last Year	0.692553
2	7e59	2.777017
3	91b6	7.732826
4	99bf	10.324679
5	a7ac	11.420267
6	Sam	12.099376
7	9820	12.733552

Where do our costs come from in Homework 1?

- Three $O(n2^{n/2})$ terms:
- Calculating the height of all subsets
- Sorting the table
- Performing a binary search for each first-half-subset

Let's improve *all* of these to $O(2^{n/2})$. The fastest submission is a very clean implementation of all three of these. The second-fastest does the first two, but then does a binary search—but a very cheap binary search?? (We'll come back to this)

Calculating the height in constant time

- What's faster than calculating the height from scratch each time?
- Only adding on "new" items; deleting "old" items
- That's $O(1)$ on average, but it's a pain to implement efficiently (and need to be careful of floating point issues!)
    - Would want to use instructions like `__bultin_clz(x)`: the CPU counts the number of leading zeroes in an integer in a couple clock cycles
- Can we change the order in which we calculate the set height to get improved performance?
- Idea: fill in set *one item at a time*. Only works if we store all subsets. Let's talk about this on the board.

Calculating the height in constant time

```
for(int64_t i = 0; i < f_half_length; i ++) {
        int64_t two_p_i = 1LL << i;
        for(int64_t j = 0; j < two_p_i; j ++){
                s2_heights[j + two_p_i].height = s2_heights[j].height + heights[i];
                s2_heights[j + two_p_i].mask = j + two_p_i;
        }
}
```

Sorting in linear time??

- Not possible in general, but our data has special structure
- Remember how we could more efficiently build up the heights. What would happen if we sorted the array at the same time?

Sorting in linear time (fastest two solns this year both had this)

```
for(int i=0; i<(inputSize-inputSize/2); i++){


        int64_t currID = s2t[i].id;
        double currH = sqrt(s2t[i].value);

        int maxq= pow(2,i);
        struct SubsetItem* newArr = (struct SubsetItem*)calloc(maxq,sizeof(struct SubsetItem));

        for(int q=0; q<maxq; q++){
                newArr[q].height=pArr[q].height+currH;
                newArr[q].key=pArr[q].key+currID;
        }

        sortedMerge(pArr,newArr,currPsize,maxq);
        currPsize+=maxq;
        free(newArr);
}
```

Binary search

- Really costly. Why? Three reasons:

  1. Cache misses: every jump to a new spot in the array is likely to incur a cache miss

  2. Branch mispredictions: every `if` statement is true roughly half the time (that’s the whole idea of binary search!)(that’s the whole idea of binary search!)

  3. Overhead: either from recursive calls, or from updates each time the loop runs

Getting rid of the binary search

- That said, even these highly optimized binary searches are costly. Can we avoid them entirely?
- Hint: the high cost of binary search is that we're jumping all over the table. Can we group entries so that we don't need to jump all over the table?
- Stronger hint: let's say I sort the left table (of yellow blocks), and I find the best possible solution for some entry (the largest entry in the right table that sums to $\leq h/2$). What can I say about the best possible solution for the *next*—a larger—entry?
  - The solution in the right (blue) blocks must be smaller
- How can we use this insight to reimagine our searches?

Use two tables

- Idea: make a table for *both* halves of the input; sort each. $O(2^{n/2})$ time to sort with the optimizations from before.
  - This does double our space usage!
- Now, can do a merge-like operation to determine, for each set in the first half, the optimal set in the second half
- $O(2^{n/2})$ total time.
- Cache efficiency? $O(2^{n/2}/B)$.

(Pretty much) rest of best solution

```
int currPindex = pSize-1;
for(int i=0; i<qSize; i++){

        while(qArr[i].height+pArr[currPindex].height>totalHeight/2){
                currPindex--;
        }
        if(maxSubset.height<qArr[i].height+pArr[currPindex].height){
                maxSubset.height=qArr[i].height+pArr[currPindex].height;
                maxSubset.key=qArr[i].key+pArr[currPindex].key;
        }
}
```

Two sorted tables with a binary search

- What happens if we sort both tables, and use binary search? What can we do to optimize right away?
- Can stop searching when you hit a (yellow block) subset of size $> h/2$. Saves a constant
- Harder question: how costly is the binary search really?

Two sorted tables with a binary search

Recall our costs:

1. Cache misses: every jump to a new spot in the array is likely to incur a cache miss
2. Branch mispredictions: every `if` statement is true roughly half the time (that's the whole idea of binary search!)(that's the whole idea of binary search!)
3. Overhead: either from recursive calls, or from updates each time the loop runs

- Since the solution to our binary search is in sorted order, we are accessing very similar blocks in each successive search—there should be relatively few cache misses!
- Other two costs are still high. Can we reduce them with optimizations?

Inlined, Unrolled binary search

```
 * Binary search for the predecessor in a sorted table
 * The loop is unrolled to enable slightly better performance
 */
__inline__ int unrolled_bin_search(entry* table, uint32_t high, double target) {
  uint32_t low = 0, mid;
  double mid_val;

  while (low < high) {
    mid = (low + high + 1) / 2;
    mid_val = table[mid].val;

    if (target > mid_val) {
      if (mid >= high) return mid;
      mid = (mid + high + 1) / 2;
      mid_val = table[mid].val;
      if (target > mid_val) {
        low = mid;
      } else if (target < mid_val) {
        high = mid - 1;
      } else {
        return mid;
      }

    } else if (target < mid_val) {
      if (low >= mid - 1) return low;
      mid = (low + mid) / 2;
      mid_val = table[mid].val;
      if (target > mid_val) {
        low = mid;
      } else if (target < mid_val) {
        high = mid - 1;
      } else {
        return mid;
      }
    } else {
      return mid;
    }
  }

  return low;
}
```

- From a submission from few years ago; performs quite well if both sides sorted!
- Branch mispredictions still fairly high; not going to be as fast as the much simpler scan

Lessons

- Cache efficiency is king
- In this case, most optimizations depended on the problem itself. `gcc` can't help with that
  - I think Homework 2 is much more optimization-heavy since our goal is fitting in cache anyway

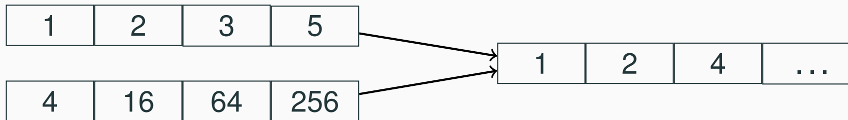
Sorting in External Memory

What about algorithms we know?

- How long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\frac{n}{B} \log_2 \frac{n}{B}$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $n/B \log_2 n/B$.
- Heapsort is $n \log_2 n/B$ unless we're careful...
- Can we do better?

Merge sort reminder

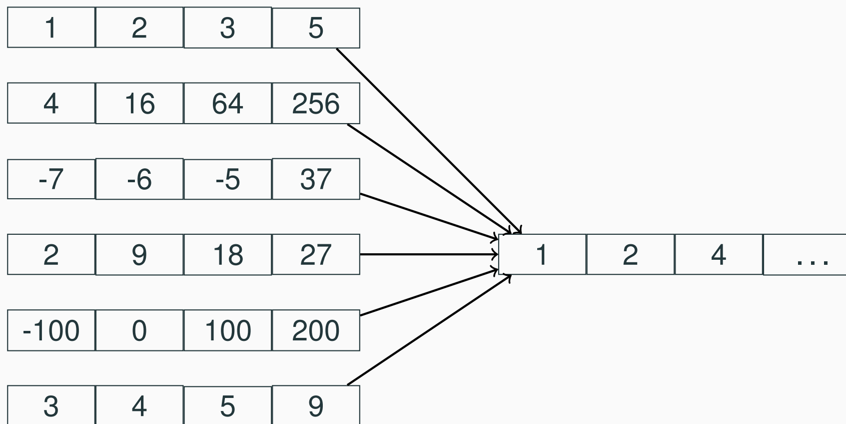
- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



M/B-way merge sort

- Divide array into M/B equal parts
- Recursively sort all M/B parts
- Merge all M/B arrays in $O(n)$ time (and $O(n/B)$ cache misses)

Diagram of M/B-way merge sort



More Detail on merges

- Keep B slots for each array in cache. (M/B arrays so this fits!)
- When all B slots are empty for the array, take B more items from the array in cache.
- Example on board

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = T\left(\frac{N}{M/B}\right) + O\left(\frac{N}{B}\right)$$
$$T(B) = O(1)$$

- Solves to $O(\frac{n}{B}\log_{M/B} n/B)$ cache misses
- Optimal!

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort

Timsort

- Developed to be the sorting method for python
- Now also used in Java, Rust
- Keeps cache in mind, but focuses more on taking advantage of easy patterns in data

Blocking revisited: run generation

- Basic idea: sort all M-sized subarrays. That would give us sorted subarrays of length M to start out with
- This is wasteful, as we empty out cache between each subarray
- Timsort starts with “run generation”: a greedy version of this that uses the same cache for as long as possible. Always outputs sorted runs of length at least M; can be MUCH longer

Timsort after run generation

- First, run generation
- Then, super optimized (2-way) merge sort
- Insertion sort on any very small arrays that are encountered (size < 64)

External Memory Sorting

- M/B way merge sort is most efficient
- Timsort is very popular in practice; uses a simpler blocking approach to stay cache-friendly.