Applied Algorithms Lec 5: Hirshberg's Algorithm

Sam McCauley

October 21, 2021

Williams College

Admin

- Lab: no food or drink! Don't open electrical boxes
- Don't shut down machines
- Talk on Friday!
  - I mentioned last time that it's on streaming algorithms. This seems to not quite be the case. But, it is about randomization, and it appears to be about a learning/data science topic.
- More TA office hours? Maybe Tuesday?
- Assignment released right after class (on website and pushed to repo). I'll send a slack message

Writing code

- Valgrind is your friend!
    - Tells you where seg faults are
- Debuggers (like `gdb`) are best. Print statements are also OK
- Start early! And ask for help (me, TA, other students, slack).
- Sometimes I get a sense from students that they feel they're the only one who finds C/the theory/everything difficult. Just a quick reminder that that's not the case!

Plan for today

- Start with Assignment 2
- (I'd like to continue external memory first... But can't risk not finishing the basics you need to get started on assignment 2)
- We'll finish up external memory at the end
- Monday: mostly focus on reviewing Assignment 1 and going over some gcc features (lighter day in terms of concepts)

Hirschberg’s Algorithm

Time and space

- In Assignment 1, you learned about how to use space to reduce the time required by your algorithm
- In Assignment 2, we're going to do the opposite: we're going to show how a space-efficient approach can actually result in smaller wall clock time
- True even though the space-efficient approach does extra computations!

Edit Distance

- Minimum number of inserts/deletes/replaces to get from one string to another
- Useful in comp bio. Classic dynamic programming solution.

OCURRANCE vs OCCURRENCE:

Delete C

OCCURRENCE

Replace E with A

OCCURRANCE

OCURRANCE

Recursive edit distance (building up to D.P.)

- Base case: if X has length 0, what is the edit distance between X and some string Y?
    - Length of Y

Recursive edit distance (building up to D.P.)

- If the last characters of X and Y match, what is $ED(X, Y)$?
  - If X' and Y' are X and Y respectively with the last character removed, then $ED(X, Y) = ED(X', Y')$

OCCURRA**N**

OCCURRE**N**

Recursive edit distance (building up to D.P.)

- If the last characters of X and Y *don't* match, what is $ED(X, Y)$?
- Let's say we're transforming Y into X
- Min of three options: (X' and Y' are X and Y with one character removed)
    - **Replace:** $1 + ED(X', Y')$
    - **Insert:** $1 + ED(X', Y)$ (Insert the last character of X into Y. The characters of Y must match the remaining characters of X)
    - **Delete:** $1 + ED(X, Y')$ (delete the last character of Y; match the rest to X)

OCCURR**A**

OCCURR**E**

Dynamic programming

- Basically the same idea as the recursion, but we build a table
- Let $m = |X|$, $n = |Y|$.
- Build an $n + 1 \times m + 1$ table
    - (+1s are so we can have 0-length entries)
- Fill out the table row-by-row using our recursive method (doing lookups instead of recursive calls)

Example DP execution

		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	6	6	5	4	4	3	2	3
E	9	8	7	7	7	6	5	4	4	3	2

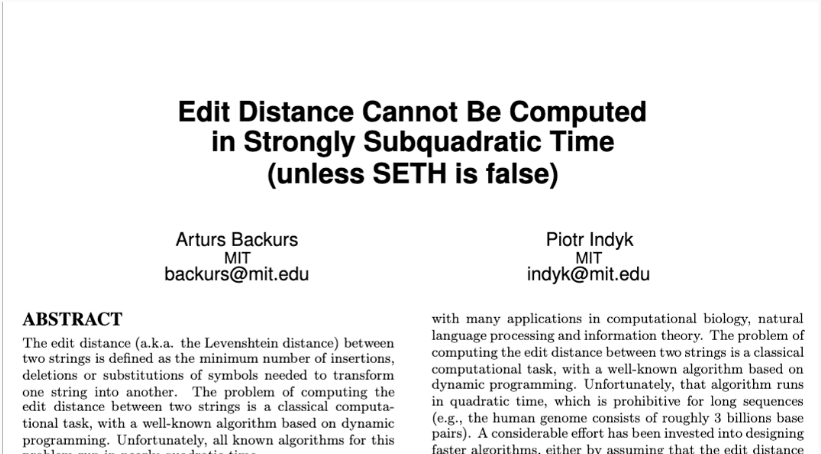
Edit distance analysis

- $O(mn)$ time (to fill out a table entry just need to look in three other table slots)

- $O(mn)$ space

Fun aside: Can we improve on this running time?

- Edit distance is an important problem. Can we do better than quadratic time?
- Probably not by more than log factors



Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false)

Arturs Backurs
MIT
backurs@mit.edu

Piotr Indyk
MIT
indyk@mit.edu

ABSTRACT

The edit distance (a.k.a. the Levenshtein distance) between two strings is defined as the minimum number of insertions, deletions or substitutions of symbols needed to transform one string into another. The problem of computing the edit distance between two strings is a classical computational task, with a well-known algorithm based on dynamic programming. Unfortunately, all known algorithms for this

with many applications in computational biology, natural language processing and information theory. The problem of computing the edit distance between two strings is a classical computational task, with a well-known algorithm based on dynamic programming. Unfortunately, that algorithm runs in quadratic time, which is prohibitive for long sequences (e.g., the human genome consists of roughly 3 billions base pairs). A considerable effort has been invested into designing faster algorithms, either by assuming that the edit distance

Edit distance in external memory

- Number of cache misses? Let's assume $n, m > B$.
- Idea: after bringing $O(1)$ cache lines in, can fill out B table entries
- $O(mn/B)$ cache misses.
- Optimal # cache misses required to fill out that table

Finding the edit distance more efficiently

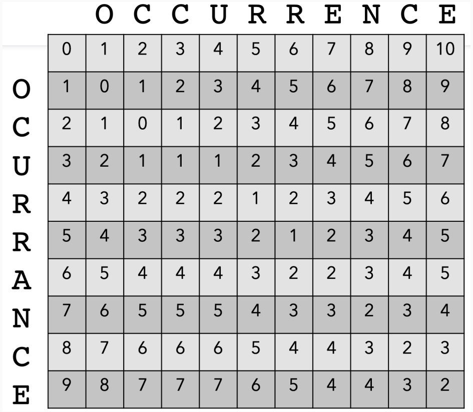
- Can we find the edit distance between two strings in less space?
  - Hint: it's almost exactly the same algorithm
- Yes: only need to store two rows of the DP table (the row we're filling out and the previous row)
- Let's say $n < m$. Then $O(n)$ extra space.
- Quick example: SPOT vs TOPS
- What is the cache efficiency of this algorithm if $3n + B \leq M$?
- $O((m+n)/B)$: can do everything in cache. WAY better than $O(mn/B)$!

Takeaway: Improved Space Can Imply Improved Cache Efficiency

One problem

- In practice, you may want to find the actual (optimal) sequence of edits between the two strings
- How can we do that with the space-inefficient approach?
- Actually not so bad: follow the path back!

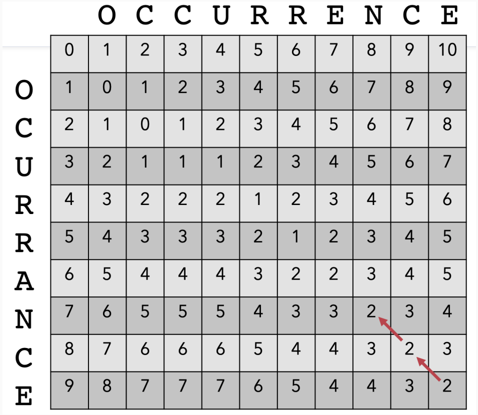
Recovering the edits



		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	6	6	5	4	4	3	2	3
E	9	8	7	7	7	6	5	4	4	3	2

- How can we tell where each entry came from?

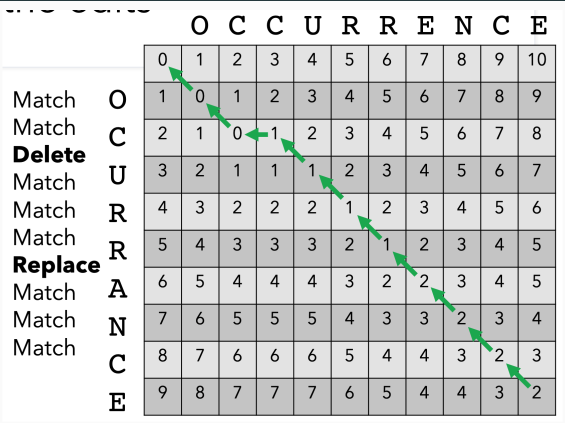
Recovering the edits



		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	6	6	5	4	4	3	2	3
E	9	8	7	7	7	6	5	4	4	3	2

- Redo same min computation from the normal dynamic program. (Break ties arbitrarily—for now.)

Recovering the edits



Match
Match
Delete
Match
Match
Match
Replace
Match
Match
Match

		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	6	6	5	4	4	3	2	3
E	9	8	7	7	7	6	5	4	4	3	2

- Once you have the path back, can essentially read back the edits: a diagonal is a match or replace; right is a delete; down is an insert. (This is if we're putting the target string vertically—if Y is being edited to become X, then X is vertical.)

Recovering the edits

- This method takes a lot of space! (The algorithm may no longer fit in cache.)

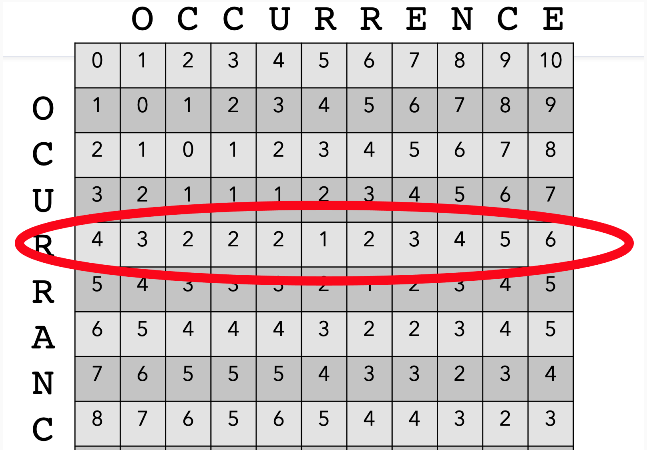
- Can we get the best of both worlds—$O(n)$ space as well as recovering the edits?

Answer: Hirschberg's algorithm!

- Recursive approach that extends the dynamic program to make it space-efficient
- Can find in textbook (woo); I also posted the original paper (a tad old but still a reasonable resource).
- "For strings of length 1,000, the problem might require one second and 1000K byes. The former is easily accommodated, the latter is not so easily obtainable."

(Slightly odd) Thought question

- Can I recover just ONE edit?
- Specifically: the edit in the middle row
- In other words: what square in the middle row is on my solution path?



		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	5	6	5	4	4	3	2	3

Structural Lemma

Lemma 1

Let's say that X and Y have edit distance k. Divide X into two halves X_1 and X_2. Then there is some way to partition Y into two parts Y_1 and Y_2 such that $ED(X_1, Y_1) + ED(X_2, Y_2) = k$.

For example:

ADVICE and VINCENT have edit distance 5.

What parts of VINCENT match up with ADV? ICE?

ED(ADV, V) $= 2$

ED(ICE, INCENT) $= 3$

Structural Lemma

Lemma 2

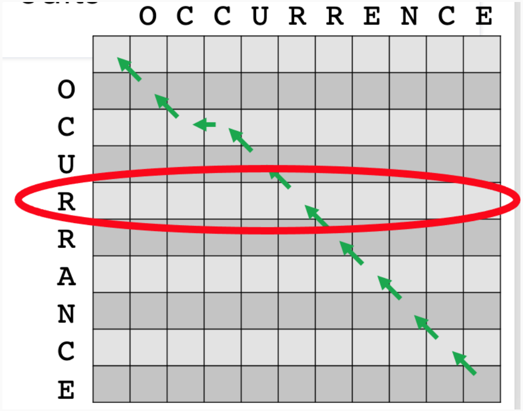
Let's say that X and Y have edit distance k. Divide X into two halves X_1 and X_2. Then there is some way to partition Y into two parts Y_1 and Y_2 such that $ED(X_1, Y_1) + ED(X_2, Y_2) = k$.

Proof idea: there is some sequence of edits applied to Y that obtain X. Let's apply those edits left to right. At some point, after we've applied some number of edits, X_1 will be a prefix of this changed string. Then Y_1 is the characters of Y that those edits were applied to.

Using the Structural Lemma

- Remember: our goal is to find where the optimal sequence crosses the middle row of the table.
- How can we use this lemma to help us out with that?
- Idea: calculate, for every possible Y_1, Y_2, $ED(X_1, Y_1) + ED(X_2, Y_2)$ (not sure how to do this yet! But bear with me)
- By the above lemma, there is at least one of these with sum exactly $ED(X, Y)$. These correspond to optimal paths through the matrix!

Using the Structual Lemma



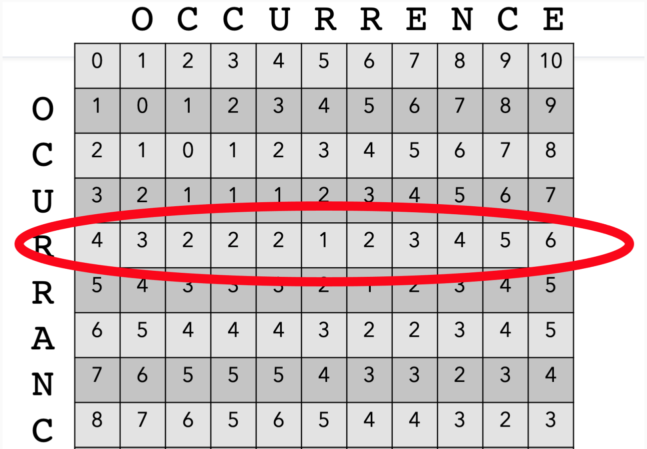
Why are we doing this?

(Just want a reminder of what we're doing. We'll come back to this analysis once we're done.)

- Let's say we can get the place where we cross over the middle in $O(nm)$ time and $O(n)$ space
- Where do we go from there?
- Answer: recurse on both subproblems! Then put the parts back together.
- How much time? We reduce the size by a factor of 2 each time we recurse. So linear time!
- Kind of like $T(X) = T(X/2) + O(X)$.

Calculating the edit distance for all Y_1 and X_1

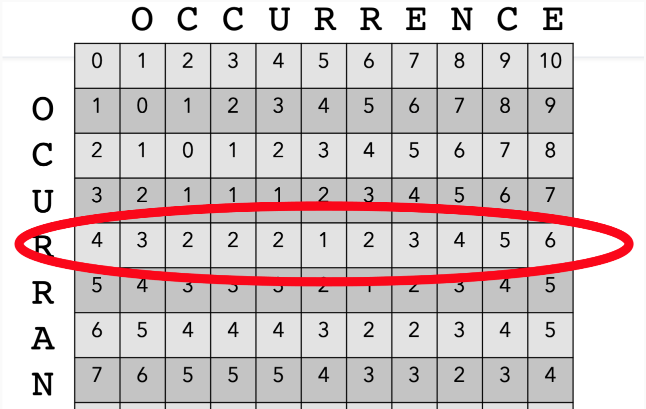
- We want to calculate, for all $i = 0 \ldots n$, the edit distance between the first i characters of Y and the first $m/2$ characters of X.
- How can we do this in $O(nm)$ time?



		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	5	6	5	4	4	3	2	3

Calculating the edit distance for all Y_2 and X_2

- We want to calculate, for all $i = 0, \ldots, n$, the edit distance between the last i characters of Y and the last $m - m/2$ characters of X.
- How can we do *this* in $O(nm)$ time?
- Problem: this doesn't quite correspond to a table row



		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4

Really nice trick

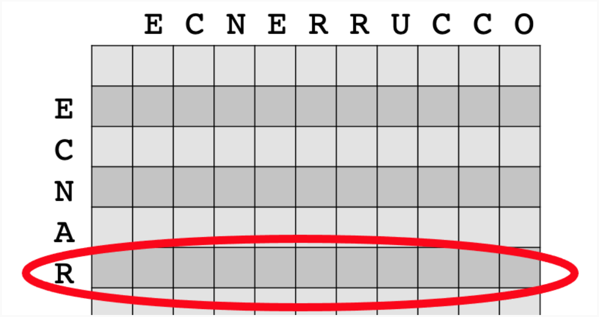
Lemma 3

Let X^R be the reverse of X, and let Y^R be the reverse of Y. Then $ED(X, Y) = ED(X^R, Y^R)$.

(Proof: just apply the same edits in reverse!)

- Let's reverse the two strings.
- "We want to calculate, for all $i = 0, \ldots, n$, the edit distance between the last i characters of Y and the last $m - m/2$ characters of X" becomes...
- We want to calculate, for all $i = 0, \ldots, n$, the edit distance between the first i characters of Y^R and the first $m - m/2$ characters of X^R
- We know how to do this from last slide! It's just the middle row of the DP table between the reversed strings

Calculating the edit distances of the last characters



Putting it all together

Let X_1 be the first half of X, and X_2 be the second half of X. Let Y_i be the first i characters of Y, and Y_i' be the last $n - i$ characters of Y.

Here's how to calculate $ED(X_1, Y_i)$ and $ED(X_2, Y_i')$ for all i, in $O(nm)$ total time and $O(n)$ space:

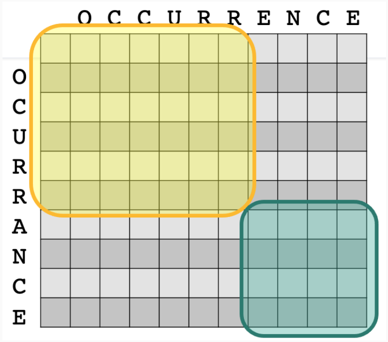
- Perform the space-efficient dynamic program (keeping track of one row at a time) between X_1 and Y (i.e. fill out the middle row of the table).
- Entry $(m/2, i)$ holds $ED(X_1, Y_i)$ by definition!
- Reverse X_2 to get X_2^R. Reverse Y to get Y^R.
- Perform the space-efficient dynamic program between X_2^R and Y^R (i.e. fill out the middle row of the reversed)
- Entry $(m - m/2, n - i)$ holds $ED(X_2, Y_i')$ by definition (and since edit distance is retained through reversal).

Where we are

- For a given X, Y, can calculate where the optimal solution crosses the middle row in $O(nm)$ time and $O(n)$ space. (Let's go back to make sure that this is true.)
- Idea: calculate all of the $X_1, Y_i, X_2, Y\prime_i$ as above. Find the Y_i and Y_i' that minimize $ED(X_1, Y_i) + ED(X_2, Y_i')$.
- If there's a tie, any of them are optimal.

Now: recurse!

- For the i we calculated as the crossing point: find the optimal sequence of edits between X_1 and Y_i. Then, find the optimal sequence of edits between X_2 and Y_i'.



What else does a recursive algorithm need!

- First, base case: if $n \leq 1$ or $m \leq 1$, use the space-inefficient edit distance algorithm.
    - In terms of implementation, base case is a bit up to you: you can use a larger base case, or possibly a smaller one. This worked well for me in practice though.
- Second, need a way to come up with the actual solution. (Remember the lemma we used to allow us to recurse?)
- Just concatenate the two recursive solutions.

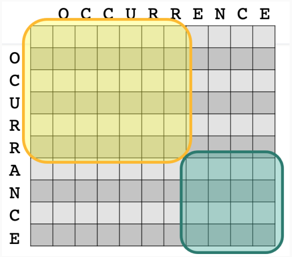
Implementation Tips

- It may be useful to keep a reversed version of both strings handy from the beginning
- When you make your recursive calls, your solutions *almost definitely* should not overlap. (Each character in a string should be a part of exactly one recursive call.)
- Implement the space-inefficient version first. You need it anyway for the base case.

Assignment 2 Review and Tips

Analysis

- How much time does this approach take?
- One recursive call takes $O(nm)$ time and $O(n)$ space.
- We make two recursive calls: one with $(i, m/2)$, and the other with $(n - i, m - m/2)$
- Can prove by induction that the total time is $O(nm)$.
- Basic idea: the total cost of all recursive calls at a given level is the size of the table remaining; this decreases by a factor of 2 each time.



Some discussion about practice

- Hirschberg's algorithm is more space-efficient. How does its time efficiency compare to the space-inefficient approach?
  - Same asymptotics, but much worse constants.
- Hirschberg's is (sometimes, and hopefully in your lab) faster in practice. Why??
- Answer: improved cache efficiency!
- If all work fits into cache, we only have the cache misses to set up the problem
- The space-inefficient approach may incur many cache misses to fill up the table.
- We'll have strings of length $\approx 30,000$. So yes, this will be the difference between fitting in (and not fitting in) L3 cache.

Matrix Multiplication in External Memory

Compute Product Directly

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

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many I/Os does this take?
- Assume matrices are stored in row-major order.
  - First: assume $3M < n^2$
  - After $O(n^2/B)$ I/Os, all three matrices are in memory, and don't have any more I/Os.
  - What if $M > n^2$?
  - Answer: $O(n^3)$ I/Os. Every inner loop operation requires an I/O for B.

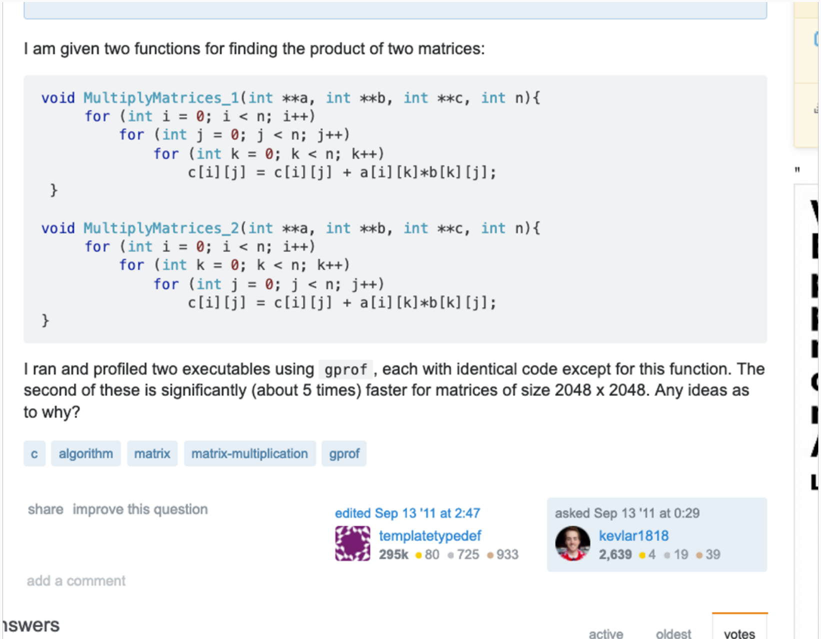
Any ideas for how to improve this?

- One idea: transpose B.
- Another idea: swap the loops!

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

- This gives us $O(n^3/B)$ I/Os: (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 $j' = j + B$.

Improvement in practice



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?

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.

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember from before: 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 $O(M/B)$ I/Os
    - Efficiently combine them for a cache-efficient solution

Blocked Matrix Multiplication

- Split A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$-sized blocks
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now.
- Multiply blocks one at a time

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:

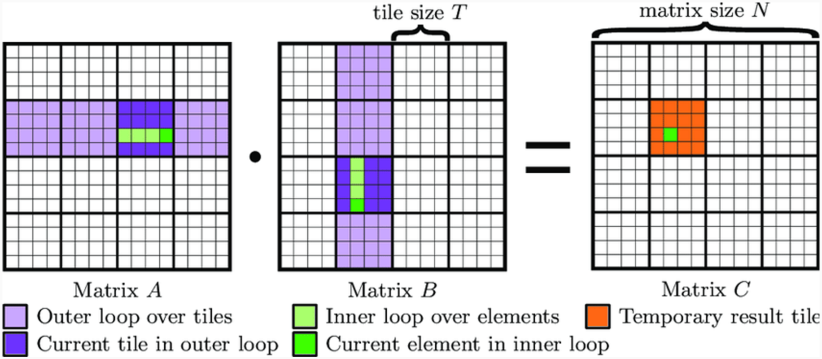
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot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

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

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

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

```
MatrixMultiply(A, B, C, n, T):
   for i = 1 to n/T:
     for j = 1 to n/T:
      for k = 1 to n/T:
        A’ = TxT matrix with upper left corner A[Ti][Tk]
        B’ = TxT matrix with upper left corner B[Tk][Tj]
        C’ = TxT matrix with upper left corner C[Ti][Tj]
        BlockMultiply(A’, B’, C’, T)

BlockMultiply(A, B, C, n):
   for i = 1 to n:
      for j = 1 to n:
         for k = 1 to n:
            C[i][j] += A[i][k] + B[k][j]
```

Let’s analyze the cost of this algorithm in the EM model

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(M^2)$ then we can simplify this to $O(M/B)$. (Called the "tall cache assumption.")
- `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 answer: 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.)

Sorting in External Memory

What about algorithms we know?

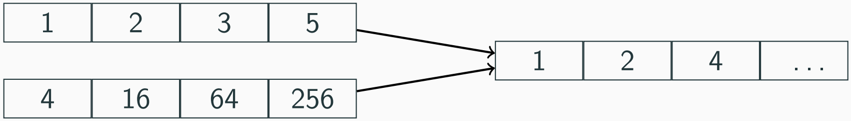
- How long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $n/B \log_2 n/B$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $n/B \log_2 n/B$.
- Heapsort is $n \log_2 n/B$ unless we're careful...
- Can we do better?

Using the cache

- Blocking? A little unclear. (We'll come back to this.)
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?
- Answer: each time you compare two numbers, can only have two outcomes.
- Each time we bring a cache line into cache, how many more things can we compare it to?

Merge sort reminder

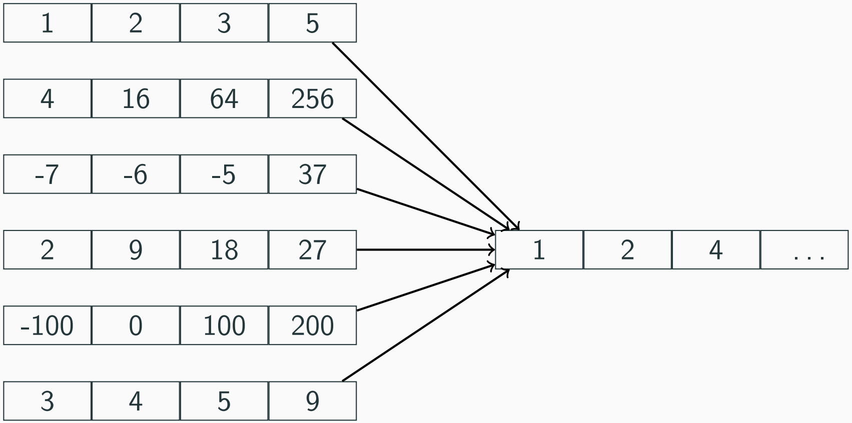
- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



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:

$$T(N) = T(N/(M/B)) + O(N/B)T(B) = O(1)$$

- 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.