Applied Algorithms Lec 5: Hirshberg's Algorithm

Sam McCauley

September 20, 2024

Williams College

Admin: Office Hours

- Unfortunately: mixed poll responses. Some really wanted the Wed 4–5 to stay; some really wanted the new hours Mon 3–4; some wanted office hours at a different time Monday

Admin: Office Hours

- Unfortunately: mixed poll responses. Some really wanted the Wed 4–5 to stay; some really wanted the new hours Mon 3–4; some wanted office hours at a different time Monday
- Let’s do the following: I’ll have drop in hours in my office Monday 9–9:45 and 3–4.

Admin: Office Hours

- Unfortunately: mixed poll responses. Some really wanted the Wed 4–5 to stay; some really wanted the new hours Mon 3–4; some wanted office hours at a different time Monday
- Let’s do the following: I’ll have drop in hours in my office Monday 9–9:45 and 3–4.
- I have to try to get some work done but you can come and work and I can answer questions

Admin: Office Hours

- Unfortunately: mixed poll responses. Some really wanted the Wed 4–5 to stay; some really wanted the new hours Mon 3–4; some wanted office hours at a different time Monday
- Let's do the following: I'll have drop in hours in my office Monday 9–9:45 and 3–4.
- I have to try to get some work done but you can come and work and I can answer questions
- Wednesday will stay 2-5 as before

Admin: Homeworks

- Homework 1 in. How was it?
- Some really cool ideas! We'll talk about some of them next week.
- Homework 2 is out
  - It is probably the most difficult homework this semester (not because it's complicated per se—it's recursive, which makes it hard to debug, and off-by-1s are very consequential)
  - Start early (!)
  - I took out a question from last time the course was taught so it should be a touch shorter
- For what it's worth: Homeworks 3 and 4 are perhaps the easiest; so things will ease up a bit in a couple weeks

Admin: Textbooks

- All course textbooks available in the lab

Admin: Textbooks

- All course textbooks available in the lab

- In the back corner next to a bunch of VHS tapes (?!)

Admin: Textbooks

- All course textbooks available in the lab
- In the back corner next to a bunch of VHS tapes (?!)
- Please don’t take them out of the lab so other students can use them

Plan for today

- Wrap up the example from last time

Plan for today

- Wrap up the example from last time
- Topics for Homework 2

Plan for today

- Wrap up the example from last time
- Topics for Homework 2
- More external memory at the end if we have time

Plan for today

- Wrap up the example from last time
- Topics for Homework 2
- More external memory at the end if we have time
- Monday: mostly focus on reviewing Homework 1 and going over some `gcc` features; perhaps another external memory model example (lighter day in terms of concepts)

External Memory Wrapup

Let’s revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?

Let's revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$

Let's revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B+1$?

Let's revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B+1$?
- Answer: $O(n)$

Let's revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$
- The external memory model predicts the real-world slowdown of this process.

Let's revisit `sortedlinkedlist.c` and `unsortedlinkedlist`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$
- The external memory model predicts the real-world slowdown of this process.
- (Actual performance is *worse* in this case: we get a slowdown of ≈ 30, whereas the number of nodes in a cache line is 8. I imagine that this is due to prefetching; seem to be some further optimizations internally.)

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same
    - Each linked list item is 16 bytes

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same
    - Each linked list item is 16 bytes
    - So total space is $\approx 8000 \cdot 16 = 128000$ byes; 128KB

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same
  - Each linked list item is 16 bytes
  - So total space is $\approx 8000 \cdot 16 = 128000$ byes; 128KB
  - L1 cache is 192KB, so it should fit!

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same
  - Each linked list item is 16 bytes
  - So total space is $\approx 8000 \cdot 16 = 128000$ byes; 128KB
  - L1 cache is 192KB, so it should fit!
- Running time is almost as good as `sortedlinkedlist.c`

What about a *shorter* linked list?

- `smallunsortedlinkedlist.c` is another unsorted linked list
- But it is only 8000 items long rather than 100 million!
- How much space does this linked list take?
- We access the list 12500 times, so the total nodes accessed remains the same
    - Each linked list item is 16 bytes
    - So total space is $\approx 8000 \cdot 16 = 128000$ byes; 128KB
    - L1 cache is 192KB, so it should fit!
- Running time is almost as good as `sortedlinkedlist.c`
- The linked list *stays in cache*. So it is cheap to access!

Homework 2: Hirschberg's Algorithm

Time and space

- In Homework 1, you learned about how to use space to reduce the time required by your algorithm

Time and space

- In Homework 1, you learned about how to use space to reduce the time required by your algorithm
- In Homework 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

Time and space

- In Homework 1, you learned about how to use space to reduce the time required by your algorithm

- In Homework 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:

OCURRANCE

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:

OCCURRENCE

OCURRANCE

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

OCURRANCE

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

OCURRANCE

OCURRANCE

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

OCURRANCE

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?

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 $ED(X, Y)$?

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 $ED(X, Y)$?

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

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)

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:** $1 + ED(X', Y')$

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)

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

Dynamic programming

- Basically the same idea as the recursion, but we build a table
- Let $m = |X|$, $n = |Y|$.

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

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)

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
  - ($+1$s 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)

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?

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

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
- [Backurs Indyk 2014]: if you can solve edit distance in less than $O(nm)$ time, you can solve 3SAT in less than 2^n time

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

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

Edit distance in external memory

- Number of cache misses? Let's assume n, m are much larger than B.

Edit distance in external memory

- Number of cache misses? Let's assume n, m are much larger than B.
- Let's work out the number of cache misses on the board.

Edit distance in external memory

- Number of cache misses? Let's assume n, m are much larger than B.
- Let's work out the number of cache misses on the board.
- Idea: after bringing $O(1)$ cache lines in, can fill out B table entries

Edit distance in external memory

- Number of cache misses? Let's assume n, m are much larger than B.
- Let's work out the number of cache misses on the board.
- Idea: after bringing $O(1)$ cache lines in, can fill out B table entries
- $O(\frac{mn}{B})$ cache misses.

Edit distance in external memory

- Number of cache misses? Let's assume n, m are much larger than B.
- Let's work out the number of cache misses on the board.
- Idea: after bringing $O(1)$ cache lines in, can fill out B table entries
- $O(\frac{mn}{B})$ cache misses.
- Optimal # cache misses required to fill out that table

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

Can we find the edit distance between two strings in less space?

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

Can we find the edit distance between two strings in less space?

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- Yes: only need to store two rows of the DP table (the row we're filling out and the previous row)

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- 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.

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- 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 on board: SPOT vs TOPS

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- 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 on board: SPOT vs TOPS
- What is the cache efficiency of this algorithm if $3n + m \leq M$?

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- 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 on board: SPOT vs TOPS
- What is the cache efficiency of this algorithm if $3n + m \leq M$?
- $O(\frac{n+m}{B})$: the only cache misses are from reading in the strings!

Finding the edit distance more efficiently

- Can we find the edit distance between two strings in less space?
- 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 on board: SPOT vs TOPS
- What is the cache efficiency of this algorithm if $3n + m \leq M$?
- $O(\frac{n+m}{B})$: the only cache misses are from reading in the strings!
- WAY better than $O(\frac{mn}{B})$!

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

One problem

- In practice, you may want to find the actual (optimal) sequence of edits between the two strings
- Warmup: how can we do that with the *space-inefficient* approach?

One problem

- In practice, you may want to find the actual (optimal) sequence of edits between the two strings
- Warmup: 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

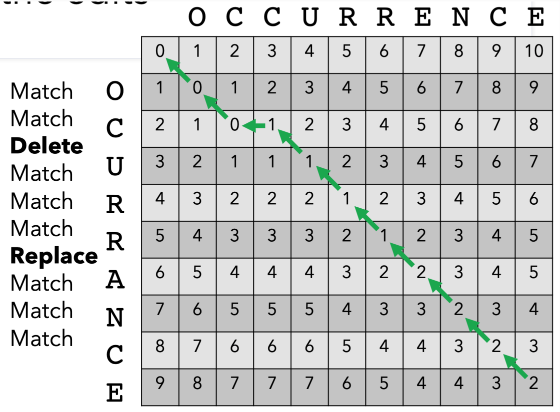
		O	C	C	U	R	R	E	N	C	E
	0	1	2	3	4	5	6	7	8	9	10
O	1	0	1	2	3	4	5	6	7	8	9
C	2	1	0	1	2	3	4	5	6	7	8
U	3	2	1	1	1	2	3	4	5	6	7
R	4	3	2	2	2	1	2	3	4	5	6
R	5	4	3	3	3	2	1	2	3	4	5
A	6	5	4	4	4	3	2	2	3	4	5
N	7	6	5	5	5	4	3	3	2	3	4
C	8	7	6	6	6	5	4	4	3	2	3
E	9	8	7	7	7	6	5	4	4	3	2

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



		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.)

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?

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?
- A note on space vs time:

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?
- A note on space vs time:
  - This problem was originally looked at in 1975 with the goal of limiting space to *fit the problem* on computers at that time

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?
- A note on space vs time:
  - This problem was originally looked at in 1975 with the goal of limiting space to *fit the problem* on computers at that time
  - Now it's still used, but the goal is to fit the problem in cache

Intro to Hirshberg's Original Paper

Introduction

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space [1, 3]. For strings of length 1,000 (assuming coefficients of 1 microsecond and 1 byte) the solution would require 10^6 microseconds (one second) and 10^6 bytes (1000K bytes). The former is easily accommodated, the latter is not so easily obtainable. If the strings were of length 10,000, the problem might not be solvable in main memory for lack of space.

Answer: Hirschberg's algorithm!

- Recursive approach that extends the dynamic program to make it space-efficient

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).

(Slightly odd) Thought question

- Can I recover just ONE edit?

(Slightly odd) Thought question

- Can I recover just ONE edit?
- Specifically: the edit in the middle row

(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
E	9	8	7	6	6	6	5	4	4	3	2

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?

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 optimal sequence of edits applied to Y that obtain X. Let's apply those edits left to right. As we apply those edits, more and more of Y will match X (let's do an example with ADVICE and VINCENT on the board).

At some point, the beginning of Y will match the first half of X (that is to say: will match X_1). We can take that as Y_1, and the remainder of Y as Y_2.

Structural Lemma

Lemma 3

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 $ED(X_1, Y_1) + ED(X_2, Y_2) = k$.

Note: I am not showing you this lemma just to be formal. This is a useful reference for when you're coding so that you know *exactly* how subproblems fit together. Perhaps most importantly: Y_1 and Y_2 do not overlap; nor do X_1 and X_2.

Using the Structural Lemma

- Remember: our goal is to find where the optimal sequence crosses the middle row of the table.

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?

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?
- As before: let's split X into two *equal* sized parts X_1 and X_2 (corresponds to the middle row of the table)

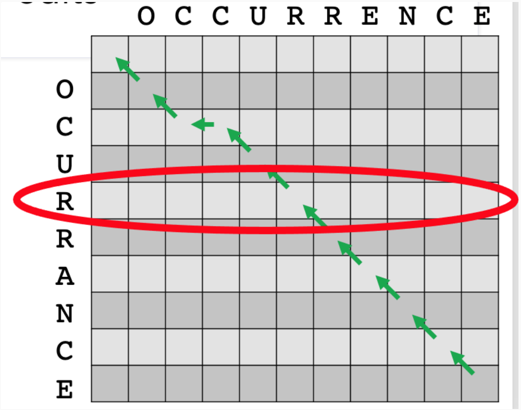
Using the Structural Lemma

- 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?
- As before: let's split X into two *equal* sized parts X_1 and X_2 (corresponds to the middle row of the table)
- Idea: for *every possible* Y_1, Y_2, calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$ (slow for now! But bear with me)

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?
- As before: let's split X into two *equal* sized parts X_1 and X_2 (corresponds to the middle row of the table)
- Idea: for *every possible* Y_1, Y_2, calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$ (slow for now! 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

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?

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.

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!

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 $T(X) = T(X/2) + O(X)$

What we want

- For all Y_1 and Y_2 we want to calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$

What we want

- For all Y_1 and Y_2 we want to calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$

- Let's calculate them separately: let's calculate $ED(X_1, Y_1)$ for all Y_1, and $ED(X_2, Y_2)$ for all Y_2.

Calculating $ED(X_1, Y_1)$ for all Y_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.

Calculating $ED(X_1, Y_1)$ for all Y_1

- We want to calculate, for all $i = 0 \dots n$, 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 and $O(n)$ space?

Calculating $ED(X_1, Y_1)$ for all Y_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 and $O(n)$ space?

		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
E	9	8	7	6	6	6	5	4	4	3	2

Calculating $ED(X_1, Y_1)$ for all Y_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 and $O(n)$ space?

The values we want *are* the entries in row $m/2$ of the DP table! So we already know how to calculate these in $O(nm)$ time and $O(n)$ space

Calculating $ED(X_2, Y_2)$ for all Y_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.

Calculating $ED(X_2, Y_2)$ for all Y_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 and $O(n)$ space?

Calculating $ED(X_2, Y_2)$ for all Y_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 and $O(n)$ space?
- 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
C	8	7	6	5	6	5	4	4	3	2	3

Really nice trick

Lemma 4

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.

Really nice trick

Lemma 4

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...

Really nice trick

Lemma 4

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

Really nice trick

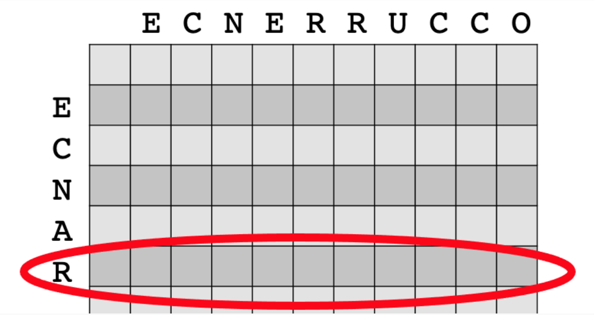
Lemma 4

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 $i = 0, \dots, n$, 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 $i = 0, \dots, n$, 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).

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!

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.

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)

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.

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.

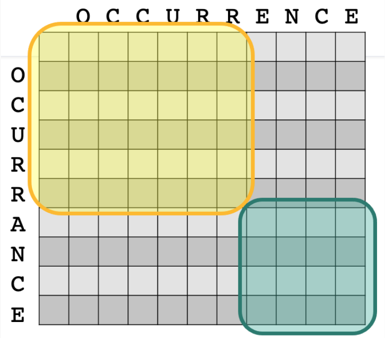
- Idea: calculate all of the X_1, Y_i, X_2, Y_i' as above. Find the Y_i and Y_i' that minimize $ED(X_1, Y_i) + ED(X_2, Y_i')$.

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.
- Idea: calculate all of the X_1, Y_i, X_2, Y_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 will give an optimal solution.

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?

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.

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.
- Second, need a way to come up with the actual solution. (Remember the lemma we used to allow us to recurse?)

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.
- 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.

Analysis

- How much time does this approach take?

Analysis

- How much time does this approach take?
- One recursive call takes $O(nm)$ time and $O(n)$ space.

Analysis

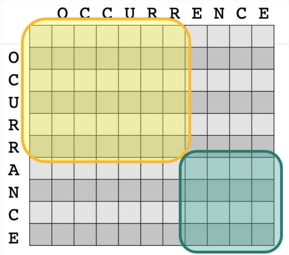
- How much time does this approach take?
- One recursive call takes $O(nm)$ time and $O(n)$ space.
- We make two recursive calls: one with $(i, m/2)$, and the other with $(n - i, m - m/2)$

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 $O(nm)$.

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?

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.

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??

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!

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

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.

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.

Implementation Tips

- It may be useful to keep a reversed version of both strings handy from the beginning

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.)

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.

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.
- Let's look over the homework quickly

Matrix Multiplication in External Memory

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B
- Want to compute their product C:

Matrix Multiplication Reminder

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

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:

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

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: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

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 cache misses does this take?

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 cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M < n^2$

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 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

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 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 $M > n^2$?

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 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?

Any ideas for how to improve this?

- One idea: transpose B (store in column-major order)

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

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?

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

Any ideas for how to 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]
```

Any ideas for how to 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]
```

- This gives us $O(n^3/B)$ cache misses: (assume $B < n$ to make things easier)

Any ideas for how to 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]
```

- This gives us $O(n^3/B)$ cache misses: (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 $k' = k + B$.

Any ideas for how to 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]
```

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

Any ideas for how to 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]
```

- This gives us $O(n^3/B)$ cache misses: (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?

```
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 $O(n^3/B)$ cache misses: (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 $j' = j + B$.
- Sum up each on the board

Any ideas for how to 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]
```

- This gives us $O(n^3/B)$ cache misses: (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 $j' = j + B$.
- Sum up each on the board
- 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

We haven’t used the cache yet

- No *M*s in any running times—except when the whole problem fits in cache

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.

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.

We haven't used the cache yet

- No *M*s 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.

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.

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 $O(M)$

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

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 A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$ matrices
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now so there's no rounding

Blocked Matrix Multiplication

- Split A, B, and C into blocks of size $M/3$
    - $\sqrt{M/3} \times \sqrt{M/3}$ matrices
    - 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

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.

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

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:

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

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.

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.

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

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.

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

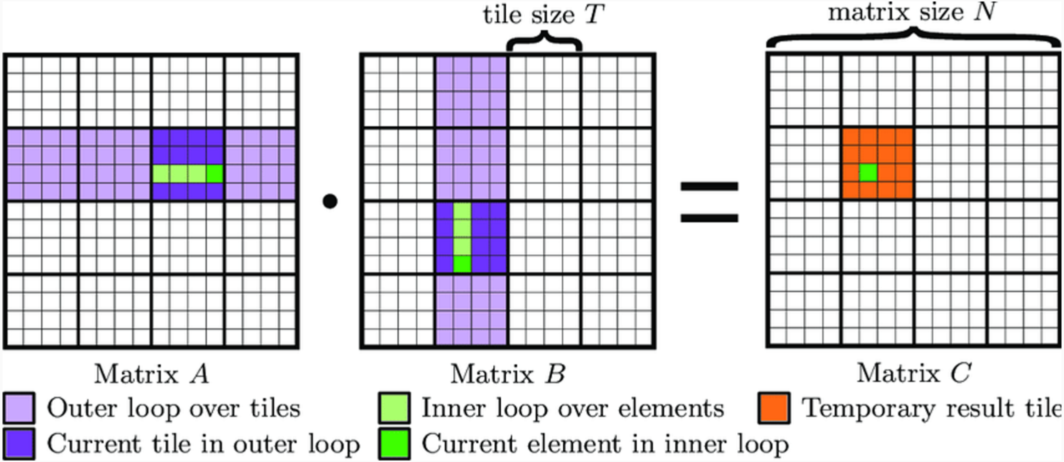
$$\begin{bmatrix} \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\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

Blocked Matrix Multiplication

- Decompose matrix into blocks of length T (recall that $T^2 \leq M/3$)

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]
```

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 together on the board!

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses.

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ 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

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ 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 answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well

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.

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?

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 $n/B \log_2 n/B$.

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?

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 $n/B \log_2 n/B$.

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...

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.)

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?

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.

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

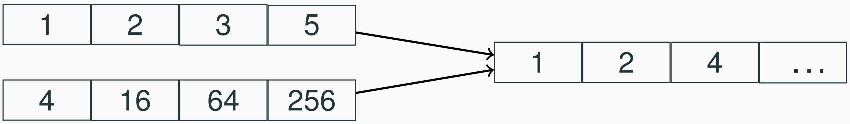
- Divide array into two equal parts

Merge sort reminder

- Divide array into two equal parts
- Recursively sort both parts

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

M/B-way merge sort

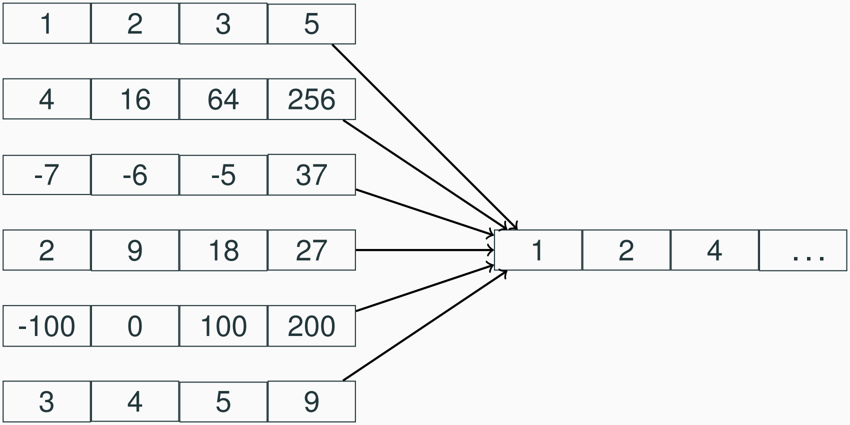
- Divide array into M/B equal parts

- Recursively sort all M/B parts

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!)

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.

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.

Analysis

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

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

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

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

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort

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

Timsort

- Developed to be the sorting method for python
- Now also used in Java, Rust

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

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

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

Timsort after run generation

- First, run generation
- Then, super optimized (2-way) merge sort

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

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.