# Applied Algorithms Lec 5: Hirshberg's Algorithm

Sam McCauley

September 19, 2025

Williams College

# Admin

- Cool talk right after class! On “practical” algorithms
- Assignment 1 handed in
- Assignment 2 out later today. Get started early!
- Recursive algorithm—so getting (my) help with debugging will be particularly useful

# Plan for today

- Topics for Assignment 2
- More external memory at the end if we have time

# Assignment 2: Hirschberg's Algorithm

# Time and space

- In Assignment 1, we used space to reduce the time required for the algorithm

# Time and space

- In Assignment 1, we used space to reduce the time required for the 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 Assignment 1, we used space to reduce the time required for the 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:

# 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

# 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

# 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

OCURRENCE

# 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

OCURRENCE

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
    - ($+1$s 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

# Edit distance analysis

- How much time does this take?

# Edit distance analysis

- How much time does this take?
- $O(mn)$ time (to fill out a table entry just need to look in three other table slots)

# Edit distance analysis

- How much time does this take?
- $O(mn)$ 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 $3n + m \leq M$?

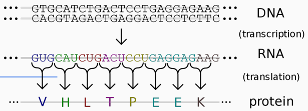
# Finding the edit distance more efficiently

- 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

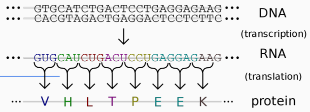
- Can we find the edit distance between two strings in less space?
- Yes: only need to store two rows of the DP table (the row we're filling out and the previous row)
- Let's say $n < m$. Then $O(n)$ extra space.
- Quick example on board: SPOT vs TOPS
- What is the cache efficiency of this algorithm if $3n + m \leq M$?
- $O(\frac{n+m}{B})$: the only cache misses are from reading in the strings!
- WAY better than $O(\frac{mn}{B})$!

# Summary




- Classic edit distance: $O(\frac{mn}{B})$ cache misses

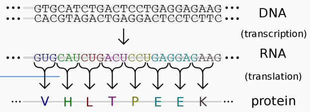
# Summary




- Classic edit distance: $O(\frac{mn}{B})$ cache misses

- Improving space usage: if $3n + m \leq M$, then only $O(\frac{n+m}{B})$ cache misses

# Summary




- Classic edit distance: $O(\frac{mn}{B})$ cache misses
- Improving space usage: if $3n + m \leq M$, then only $O(\frac{n+m}{B})$ cache misses
- Improve by a factor of $\min\{n, m\}$ to compute edit distance of two strings that fit in cache

# 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

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.)

# 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

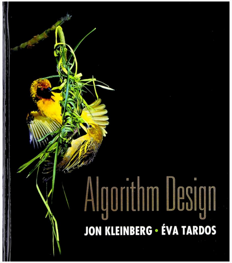
- This method takes a lot of space! (The algorithm may no longer fit in cache.)
- Can we get the best of both worlds—$O(n)$ space as well as recovering the edits?
- A note on space vs time:
  - This problem was originally looked at in 1975 with the goal of limiting space to *fit the problem* on computers 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

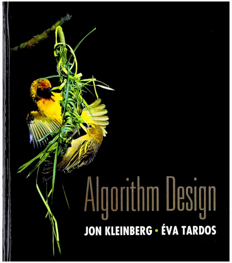
The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space [1, 3]. For strings of length 1,000 (assuming coefficients of 1 microsecond and 1 byte) the solution would require $10^6$ microseconds (one second) and $10^6$ bytes (1000K bytes). The former 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 Kleinberg-Tardos (algorithms) textbook (woo). I'll email you the relevant pages. I also posted the original paper (a tad old but still a reasonable resource).

# (Slightly odd) Thought question

- Can I recover just ONE piece of the optimal path?

# (Slightly odd) Thought question

- Can I recover just ONE piece of the optimal path?
- Specifically: find one square in the middle row that is on the optimal 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**

*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**

*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**

*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**

*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 $X_2$.

**$Y_1$ and $Y_2$ do not overlap; nor do $X_1$ and $X_2$**

**$Y_1$ and $Y_2$ do not overlap; nor do $X_1$ and $X_2$**

**(Check this when you are coding! It will save you time. ☺)**

# 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)

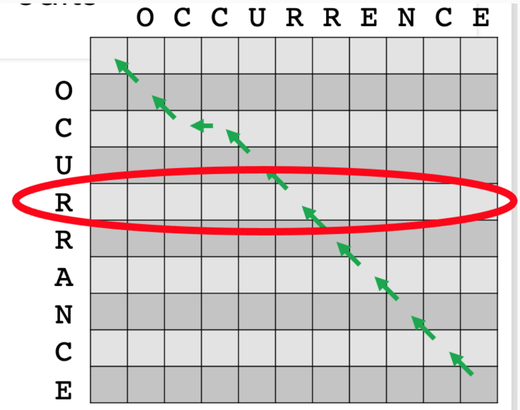
# Using the Structural Lemma

- 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)
- Find the $Y_1$, $Y_2$ that minimizes this sum

# 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)
- Find the $Y_1$, $Y_2$ that minimizes this sum
- By the above lemma, we can optimally edit $X$ into $Y$ by: first editing $X_1$ into $Y_1$, and then editing $X_2$ into $Y_2$

# 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 divide $X$ and $Y$ into two pieces 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 divide $X$ and $Y$ into two pieces 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 divide $X$ and $Y$ into two pieces 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 divide $X$ and $Y$ into two pieces 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? (Rough sketch of argument): We reduce the size of the DP table 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 divide $X$ and $Y$ into two pieces 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? (Rough sketch of argument): We reduce the size of the DP table 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

$\boxed{\text{We want to calculate } ED(X_1, Y_1) + ED(X_2, Y_2) \text{ for all } Y_1}$

# What we want

$\boxed{\text{We want to calculate } ED(X_1, Y_1) + ED(X_2, Y_2) \text{ for all } Y_1}$

- Recall that $X_1$ and $X_2$ divide $X$ in half; $Y_2$ is the rest of $Y$

# What we want

We want to calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$ for all $Y_1$

- Recall that $X_1$ and $X_2$ divide $X$ in half; $Y_2$ is the rest of $Y$
- If we can do this, we can find the $Y_1$ that minimizes this cost; then recurse on $X_1, Y_1$ and $X_2, Y_2$

# What we want

We want to calculate $ED(X_1, Y_1) + ED(X_2, Y_2)$ for all $Y_1$

- Recall that $X_1$ and $X_2$ divide $X$ in half; $Y_2$ is the rest of $Y$
- If we can do this, we can find the $Y_1$ that minimizes this cost; then recurse on $X_1, Y_1$ and $X_2, Y_2$
- 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 |
| E | 9 | 8 | 7 | 6 | 6 | 6 | 5 | 4 | 4 | 3 | 2 |

# Really nice trick

**Lemma**

*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**

*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**

*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

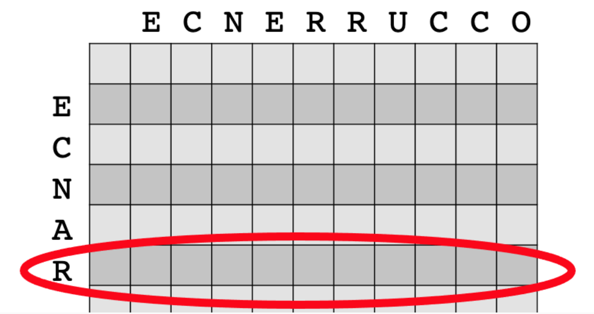
**Lemma**

*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).

# 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

# Where we are

- Can: 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

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

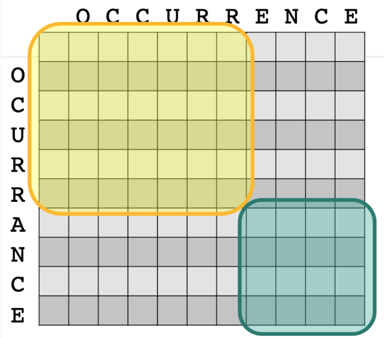
- If there's a tie, *any* of them will give an optimal solution.

# 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 $Y_i'$.

# 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 $Y_i'$.
- Concatenate these two sequences to get the optimal sequence of edits for $X$ and $Y$



# What else does a recursive algorithm need?

# What else does a recursive algorithm need?

- Base case:

# What else does a recursive algorithm need?

- Base case:

- if $n \leq 1$ or $m \leq 1$, use the space-inefficient edit distance algorithm.

# What else does a recursive algorithm need?

- Base case:
- if $n \leq 1$ or $m \leq 1$, use the space-inefficient edit distance algorithm.
- Can be more clever about it to improve the speed

# 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

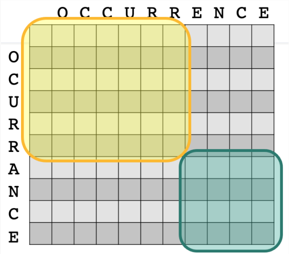
- How much time does this approach take?
- One recursive call takes $O(nm)$ time and $O(n)$ space.
- We make two recursive calls: one with $(i, m/2)$, and the other with $(n - i, m - m/2)$

# 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.

# Implementation Tips

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

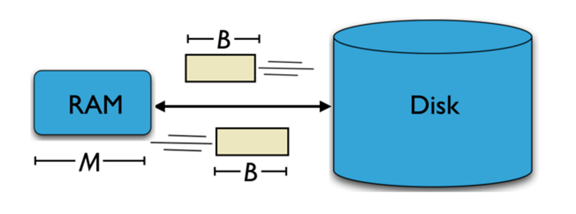
# Implementation Tips

- 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.)

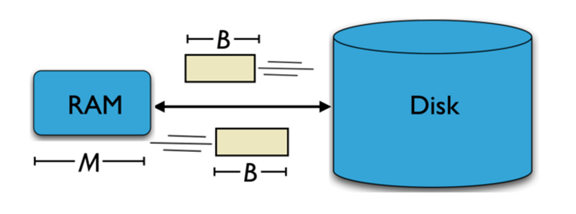
# Sorting in External Memory

# External Memory Model Reminder

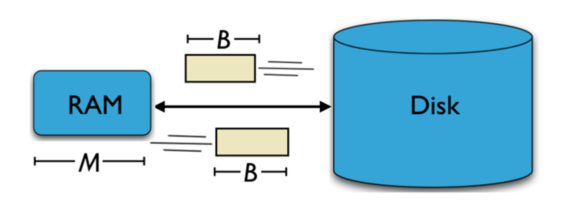



- Cache line of size $B$
- Cache can hold at most $M$ elements
- Computing on elements in cache is free! Only cost is to bring things in and out of cache

# External Memory Model Reminder

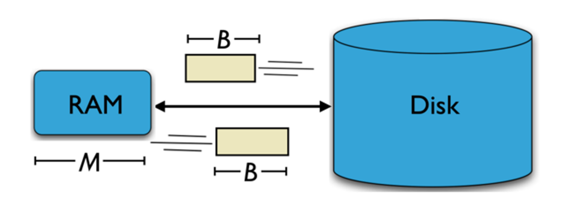


# External Memory Model Reminder




- Let's say I want to find the maximum element in an array using a linear scan. How much time does that take?

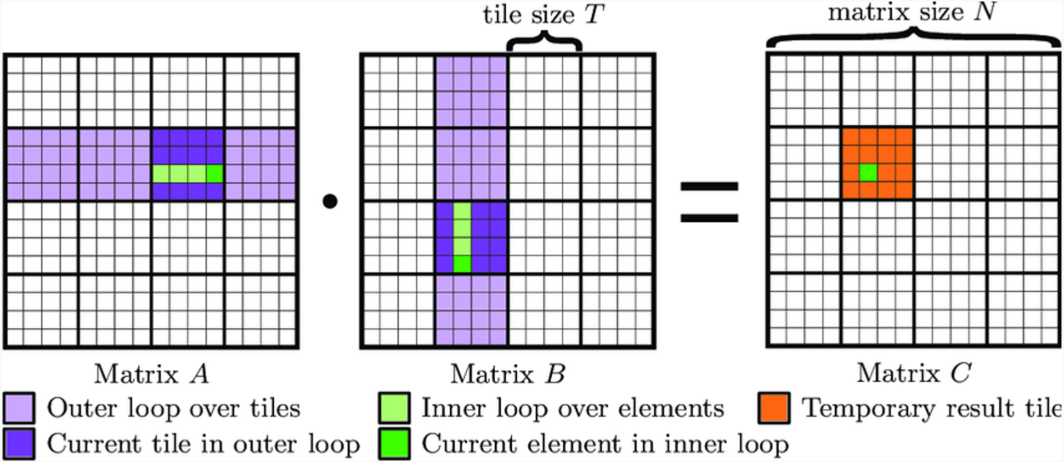
# External Memory Model Reminder



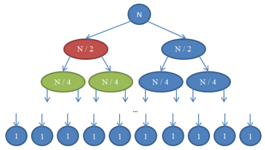

- Let's say I want to find the maximum element in an array using a linear scan. How much time does that take?
    - $O(n/B)$ cache misses: if I have a cache miss accessing $A[i]$, my next cache miss is accessing $A[i + B]$.

# Blocked Matrix Multiplication

- Decompose matrix into blocks of length $T$ (where $T^2 = M/3$)
- Do a normal $n/T \times n/T$ matrix multiplication
- Last class, saw: $O(n^3/B\sqrt{M})$ total cache misses.

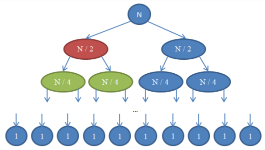


# What about algorithms we know?




- In pairs: how long does Mergesort take in external memory?

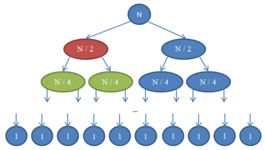
# What about algorithms we know?




- In pairs: how long does Mergesort take in external memory?

- Merge is $O(n/B)$; base case is when $n = B$, so total is $\text{O}(n/B \log_2 n/B)$.

# What about algorithms we know?

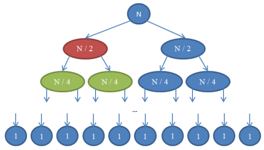



- In pairs: how long does Mergesort take in external memory?

- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.

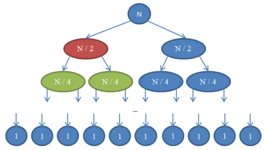
- How about quicksort?

# What about algorithms we know?



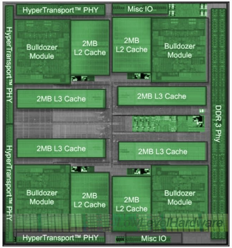

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.

# What about algorithms we know?



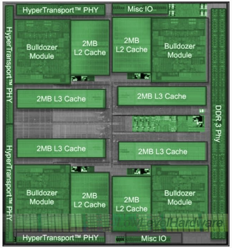

- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $\mathrm{O}(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.
- Seems pretty good! Can we do better?

# Using the cache




- Blocking? A little unclear. (We'll come back to this.)

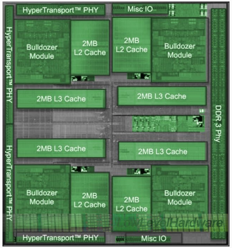
# Using the cache




- Blocking? A little unclear. (We'll come back to this.)

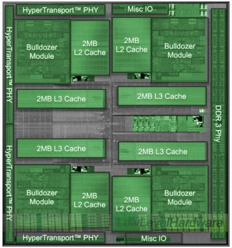
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?

# 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.

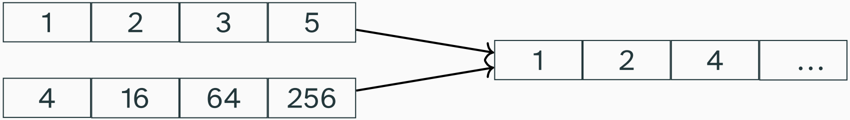
# Using the cache




- 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
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



# $M/B$-way merge sort

# $M/B$-way merge sort

- Divide array into $M/B$ equal parts

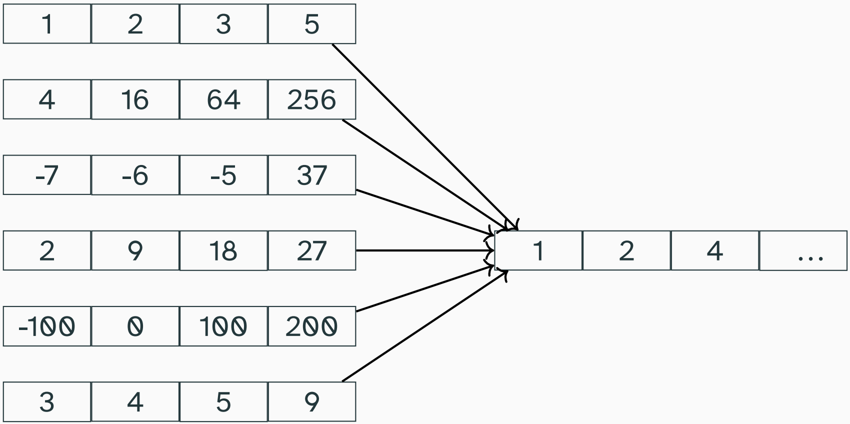
# $M/B$-way merge sort

- Divide array into $M/B$ equal parts
- Recursively sort all $M/B$ parts

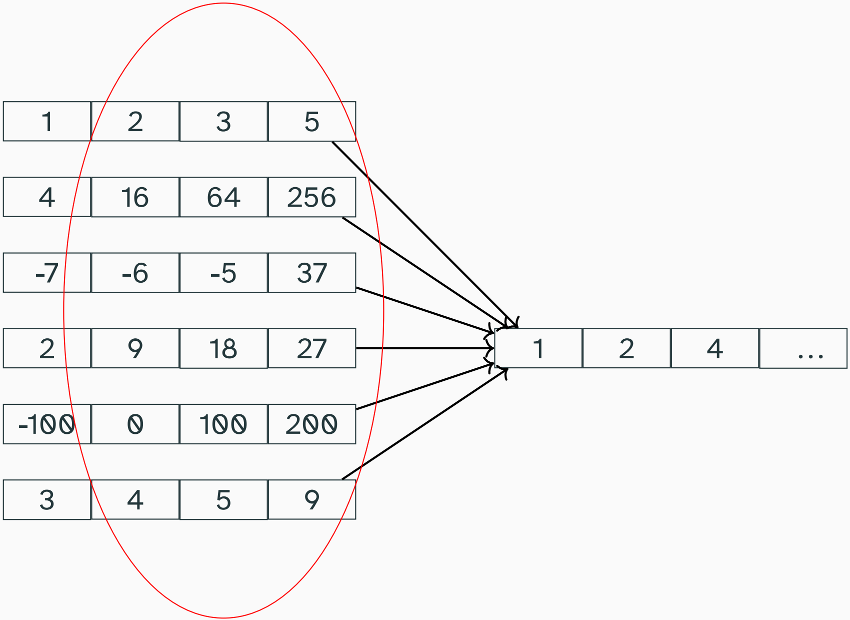
# $M/B$-way merge sort

- 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)

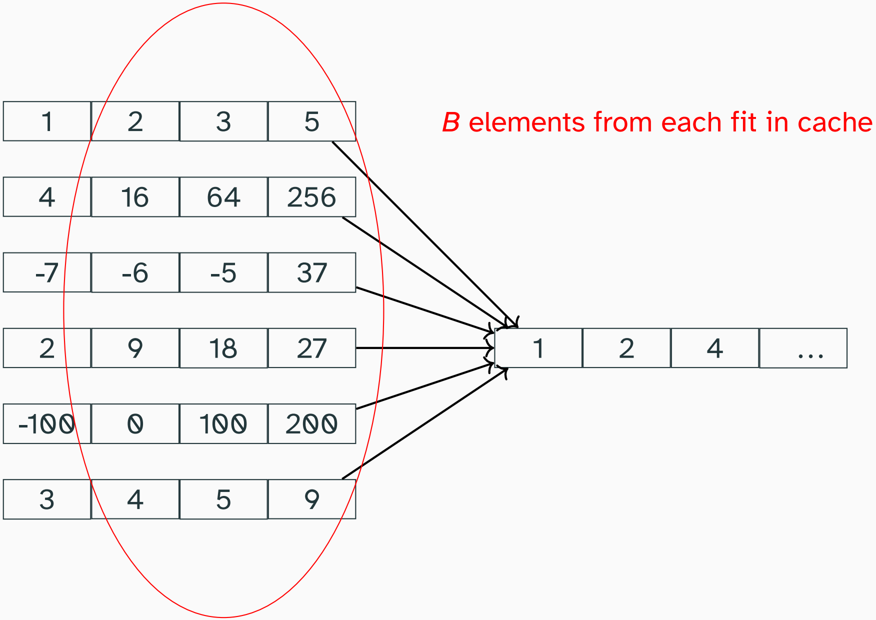
# Diagram of $M/B$-way merge sort



# Diagram of $M/B$-way merge sort



# 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 $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses (use recursion tree method)

# 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 $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cache misses (use recursion tree method)
- 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: Powersort

# 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: Powersort
- New; invented in 2018 to improve on Timsort

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a “stack” of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a “stack” of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.
- Takes advantage of already-sorted portions of the array

# 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: Powersort
- New; invented in 2018 to improve on Timsort
- Merges a "stack" of runs. Somewhat similar to $M/B$-way merge sort, achieves strong cache efficiency in practice.
- Takes advantage of already-sorted portions of the array
- When you call sort in python, it is either Timsort or Powersort