DSC 190

DATA STRUCTURES & ALGORITHMS

Today's Lecture

Dynamic Programming

- We’ve seen that dynamic programming can lead to fast algorithms that find the optimal answer.
- Today, we’ll see one data science application: longest common substring.
- Used to match DNA sequences, fuzzy string comparison, etc.

The Strategy

1. Backtracking solution.

2. A “nice” backtracking solution with overlapping subproblems.

3. Memoization.

DSC 190

Data Structures & Algorithms

Longest Common Subsequence

Fuzzy String Matching

- Suppose you're doing a sentiment analysis of tweets.
- How do people feel about the University of California?
- Search for: `university of california`
- People can't spell: `uivesity of califrbia`
- How do we recognize the match?

DNA String Matching

- Suppose you're analyzing a genome.
- DNA is a sequence of `G,A,T,C`.
- Mutations cause same gene to have slight differences.
- Person 1: `GATTACAGATTACA`
- Person 2: `GATCACAGTTGCA`

Measuring Differences

- Given two strings of (possibly) different lengths.
- Measure how similar they are.
- One approach: **longest common subsequences**.

Common Subsequences

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
u	n	i	v	e	r	s	i	t	y		o	f		c	a	l	i	f	o	r	n	i	a

u	i	v	e	s	i	t	y		o	f		c	a	l	i	f	r	b	i	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Longest Common Subsequences

- We will measure similarity by finding length of the **longest common subsequence** (LCS).
- Now: let’s define the LCS..

Subsequences

```
s a n d i e g o
s a n d i e g o   →   igo
s a n d i e g o   →   sio
s a n d i e g o   →   sadego
s a n d i e g o   →   sandiego
```

Not Subsequences

`s a n d i e g o`

`s a n d i e g o` ↛ **sea**

`s a n d i e g o` ↛ **sooo**

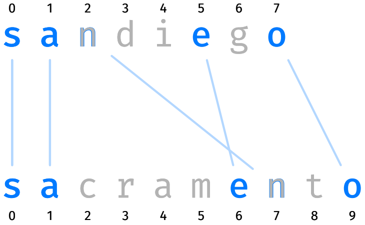
Subsequences

- A **subsequence** of a string `s` of length n is determined by a strictly monotonically increasing sequence of indices with values in $\{0, 1, \ldots, n-1\}$.

0	1	2	3	4	5	6	7		0	1	3	5	6	7
s	a	n	d	i	e	g	o	→	s	a	d	e	g	o

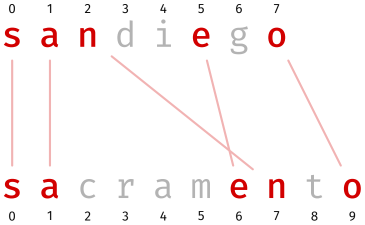
Common Subsequences

- Given two strings, a **common subsequence** is subsequence that appears in both.



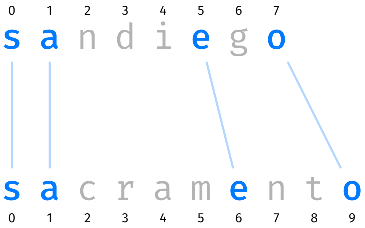
Not Common Subsequences

- The lines cannot overlap.



Longest Common Subsequences

- A **longest common subsequence** (LCS) between two strings is a common subsequence that has the greatest length out of all common subsequences.



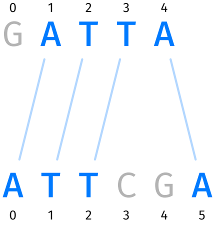
Main Idea

The longer the LCS, the more similar the two strings.

Common Subsequences, Formally

- Our backtracking solution will build a common subsequence piece by piece.
- How can we represent the idea of “lines between letters” more formally?

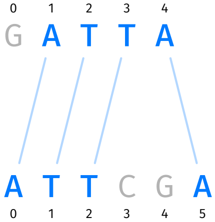
Matching



(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Matching

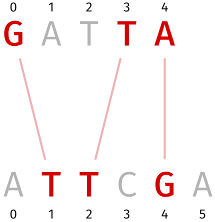
- A **matching** between strings a and b is a set of (i, j) pairs.
- Each (i, j) pair is interpreted as "`a[i]` is paired with `b[j]`".
- Example: $\{(1,0),(2,1),(3,2),(4,5)\}$



Invalid Matchings

- Not all matchings represent common subsequences!

- Example: $\{(0,1),(3,2),(4,4)\}$:



- Example: $\{(4,0),(2,1),(3,2)\}$:

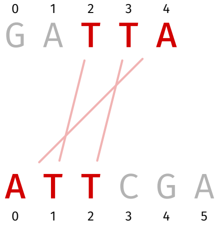
Valid Matchings

- We'll say a matching M is **valid** if:
  - `a[i] == b[j]` for every pair (i, j); and
  - there are no "crossed lines"

“Crossed Lines”

- Suppose (i, j) and (i', j') are in the matching.

- “Crossed lines” occur when either:
  - $i < i'$ but $j > j'$; or
  - $i > i'$ but $j < j'$.

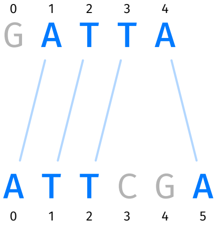


Valid Matchings

- We'll say a matching M is **valid** if:
  - `a[i] == b[j]` for every pair (i, j); and
  - there are no "crossed lines". that is, for every choice of distinct pairs $(i, j), (i', j') \in M$:

$$i < i' \text{ and } j < j' \qquad \text{or} \qquad i > i' \text{ and } j > j'$$

- Example: $\{(1, 0), (2, 1), (3, 2), (4, 5)\}$



DSC 190

Data Structures & Algorithms

Step 01: Backtracking

Road to Dynamic Programming

- We’ll follow same road to a DP solution as last time.
- **Step 01: Backtracking solution.**
- Step 02: A “nice” backtracking solution with overlapping subproblems.
- Step 03: Memoization.

Backtracking

- We'll build up a matching, one pair at a time.
- Choose an arbitrary pair, `(i, j)`.
  - Recursively see what happens if we **do** include `(i, j)`.
  - Recursively see what happens if we **don't** include `(i, j)`.
- This will try **all valid matchings**, keep the best.

Backtracking

```
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    best_with = ...

    # best without
    best_without = ...

    return max(best_with, best_without)
```

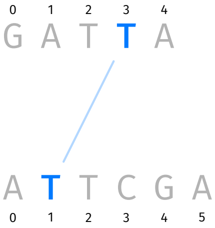
Recursive Subproblems

- What is BEST(`a, b, pairs`) if we assume that `(i, j)` **is** in matching?

- If `a[i] != a[j]`:
  - Your current common substring is **invalid**. Length is zero.
  - Don't build matching further.

- If `a[i] == a[j]`:
  - Your current common substring has length one.
  - Pairs remaining to choose from: those **compatible** with (i, j).
  - You find yourself in a similar situation as before.
  - Answer: 1 + BEST(`activities.compatible_with(x)`)

`pairs.compatible_with(x)`



(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Backtracking

```python
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    if a[i] == b[j]:
        best_with = 1 + lcs_bt(a, b, pairs.compatible_with(i, j))
    else:
        best_with = 0

    # best without
    best_without = ...

    return max(best_with, best_without)
```

Recursive Subproblems

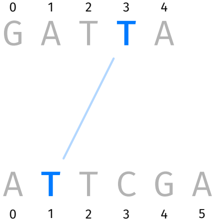
- What is BEST(`a, b, pairs`) if we assume that `(i, j)` **is not** in matching?

- Imagine not choosing x.
    - Your current common substring is empty.
    - Activities left to choose from: all except `(i, j)`.

- You find yourself in a similar situation as before.

- Answer: BEST(a, b, `pairs.without(i, j)`)

`pairs.without(x)`



(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Backtracking

```python
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    # assume (i, j) is in the LCS, but only if a[i] == b[j]
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt(a, b, pairs.compatible_with(i, j))

    # best without
    best_without = lcs_bt(a, b, pairs.without(i, j))

    return max(best_with, best_without)
```

Backtracking

- This will try all **valid** matchings.
- Guaranteed to find optimal answer.
- But takes exponential time in worst case.

DSC 190

DATA STRUCTURES & ALGORITHMS

Step 02: A “Nicer” Backtracking Solution

Arbitrary Sets

- In previous backtracking solution, subproblems are arbitrary sets of pairs.
- Rarely see the same subproblem twice.
- This is not good for memoization!

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)

Nicer Subproblems

- In backtracking, we are building a solution piece-by-piece.
- In last lecture, we saw that a careful choice of next piece led to nice subproblems.
- Let's try choosing the *last* letters from each string as the next piece of the matching.

Last Letters

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
0 1 2 3 4 G A T T A	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
A T T C G A 0 1 2 3 4 5	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Nicer Backtracking

```python
def lcs_bt_nice(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.last_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice(a, b, pairs.compatible_with(i, j))

    # best without
    best_without = lcs_bt_nice(a, b, pairs.without(i, j))

    return max(best_with, best_without)
```

Subproblems

- There are two subproblems: LCS using `pairs.compatible_with(i, j)` and LCS using `pairs.without(i, j)`

- Are they “nicer”?

pairs.compatible_with(i, j)

```
0 1 2 3 4
G A T T A

A T T C G A
0 1 2 3 4 5
```

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Nicer Subproblems

- By taking (i, j) as bottom-right pair, `pairs.compatible_with(i, j)` is again rectangular.
- Easily described by its bottom-right pair, $(i - 1, j - 1)$!
- Instead of keeping set of `pairs`, just need to pass in i and j of last element.

```python
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice_2(a, b, i-1, j-1)

    # best without
    best_without = ...

    return max(best_with, best_without)
```

`pairs.without(i, j)`

0 1 2 3 4
G A T T A

A T T C G A
0 1 2 3 4 5

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Problem

- `pairs.without(i, j)` is **not** rectangular.
- Cannot be described by a single pair.
- But there's a fix.

Observation

- A common substring cannot have pairs both in the last row and the last column. **Crossing lines!**

0	1	2	3	4
G	A	T	T	A

A	T	T	C	G	A
0	1	2	3	4	5

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Consequence

- BEST(`pairs.without(i, j)`) = max {BEST(`pairs.without_row(i)`), BEST(`pairs.without_col(j)`)}

	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
0 1 2 3 4 G A T T A	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
A T T C G A 0 1 2 3 4 5	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

Observation

- `pairs.without_row(i)` represented by subprob. $(i-1, j)$
- `pairs.without_col(j)` represented by subprob. $(i, j-1)$

	0	1	2	3	4	
	G	A	T	T	A	
	A	T	T	C	G	A
	0	1	2	3	4	5

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)

“Nice” Backtracking

```python
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice_2(a, b, i-1, j-1)

    # best without
    best_without = max(
            lcs_bt_nice_2(a, b, i-1, j),
            lcs_bt_nice_2(a, b, i, j-1)
            )

    return max(best_with, best_without)
```

One More Observation

- This is fine, but we can do a little better.
- If `a[i] == b[j]`, we can assume (i, j) is in matching – don't need to consider otherwise![1]

0	1	2	3	4	
G	A	T	T	A	
A	T	T	C	G	A
0	1	2	3	4	5

[1] This is true we chose last pair; not true if choice was arbitrary.

“Nicer” Backtracking

```
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] == b[j]:
        # best with (i, j)
        return 1 + lcs_bt_nice_2(a, b, i-1, j-1)
    else:
        # best without (i, j)
        return max(
                lcs_bt_nice_2(a, b, i-1, j),
                lcs_bt_nice_2(a, b, i, j-1)
                )
```

Overlapping Subproblems

- Suppose a and b are of length m and n.
- There are mn possible subproblems.
- Backtracking tree has exponentially-many nodes.
- We will see many subproblems over and over again!

DSC 190

DATA STRUCTURES & ALGORITHMS

Step 03: Memoization

Backtracking

- The backtracking solutions are slow.
- `a = 'CATCATCATCATCATGAAAAAAAA'`
- `b = 'GATTACAGATTACAGATTACA'`
- “Nice” backtracking solution: 8 seconds.
- Memoized solution: 100 microseconds.

```python
def lcs_dp(a, b, i=None, j=None, cache=None):
    """Solve LCS problem for a[:i], b[:j]."""
    if i is None:
        i = len(a) - 1

    if j is None:
        j = len(b) - 1

    if cache is None:
        cache = {}

    if i < 0 or j < 0:
        return 0

    if (i,j) in cache:
        return cache[(i, j)]

    # best with
    if a[i] == b[j]:
        # best with (i, j)
        best = 1 + lcs_dp(a, b, i-1, j-1, cache)
    else:
        # best without (i, j)
        best = max(
                lcs_dp(a, b, i-1, j, cache),
                lcs_dp(a, b, i, j-1, cache)
                )

    cache[(i, j)] = best
    return best
```

Top-Down vs. Bottom-Up

- This is the top-down dynamic programming solution.
- It takes time $\Theta(mn)$, where m and n are the string lengths.
- To find a bottom-up iterative solution, start with the easiest subproblem.
- What is it?