CS2 RECITATION 5: DYNAMIC PROGRAMMING

1/13/2014

ASSIGNMENT 4

- 7 points of explanation
- 10 points of coding
- 3 "red" points (no coding)
- DNA alignment – Given two strings, find the optimal alignment between them
- Seam Carving - Given an array, find the lowest-weight path from
- top to bottom

USEFUL CLASSES: STD::STRING

- A string classes with helpful functions, similar to string in python
- .substring(int start, int length) : returns a copy of the specified substring, using -1 for length goes to end of string
- .compare(string s) : returns 0 if two strings are equal
- .size() returns the length of the string
- Can be concatenated with + and +=
- string(int n, char c) : constructor that creates a string of c repeated n times

USEFUL CLASSES: STD::UNORDERED_MAP

- Similar to a dictionary in python
- Stores values that can be accessed using keys
- Types must be declared
- Values can be accessed and modified using map[key]
- unordered_map<string, align_result> is aliased to memo_type for your convenience

DYNAMIC PROGRAMMING

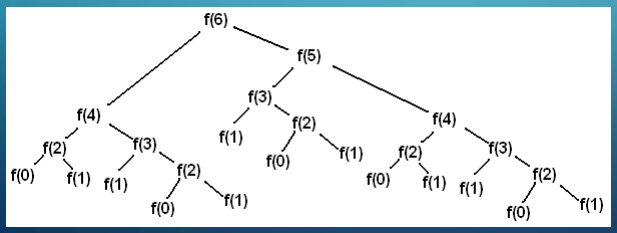
- Basically two things done together
  - Recursion
  - Memoization
- Trades space for time by saving the results of previous function calls
- Makes otherwise unfeasible algorithms possible

SIMPLE EXAMPLE: FIBONACCI SEQUENCE

Fibonacci(n):

If n is 0 or 1 return n

Otherwise return Fibonacci(n-1) + Fibonacci(n-2)



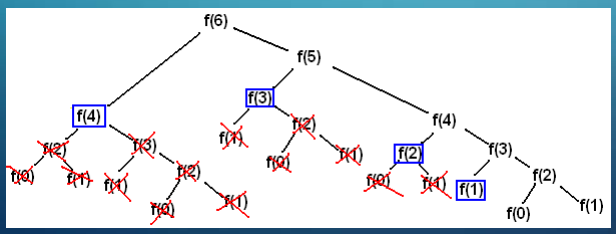
PROGRAMMING O(2^N)

SIMPLE EXAMPLE: FIBONACCI SEQUENCE

Map<int, int> calculated

Fibonacci(n):

If n is in calculated return calculated[n]
If n is 0 or 1 return n
calculated[n] = Fibonacci(n-1) + Fibonacci(n-2) //memoize
Otherwise return Fibonacci(n-1) + Fibonacci(n-2)



DYNAMIC PROGRAMMING O(N)

DNA ALIGNMENT

- Given two strings of DNA find the optimal way to align them
- We do this by inserting gaps
- For each position in the aligned result, 3 possibilities:
  - Top string character, bottom string gap
  - Top string gap, bottom string character
  - Both strings have a character
    - They can either match or not match
- Instruction string containing {s, t, |, *}

DNA ALIGNMENT

- Scoring: 2 points for a match, -1 points for a mismatch, -5 points for a gap
- Ex. ACTGGCCGT vs. TGACGTAA

S:	A	C	T	G	G	C	C	G	T
T:	T	G	A	C	G	T	A	A	_
Score:	-1	-1	-1	-1	2	-1	-1	-1	-5
Inst:	*	*	*	*	|	*	*	*	s

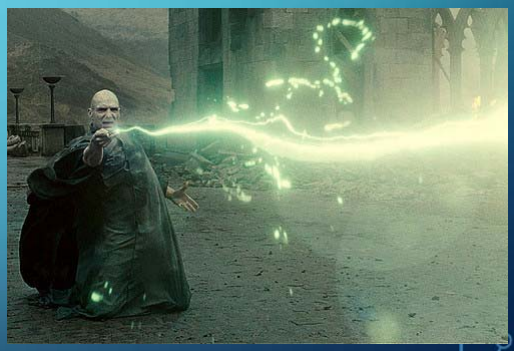
Total score: -10

ANOTHER EXAMPLE

abracadabra vs. avada kedavra

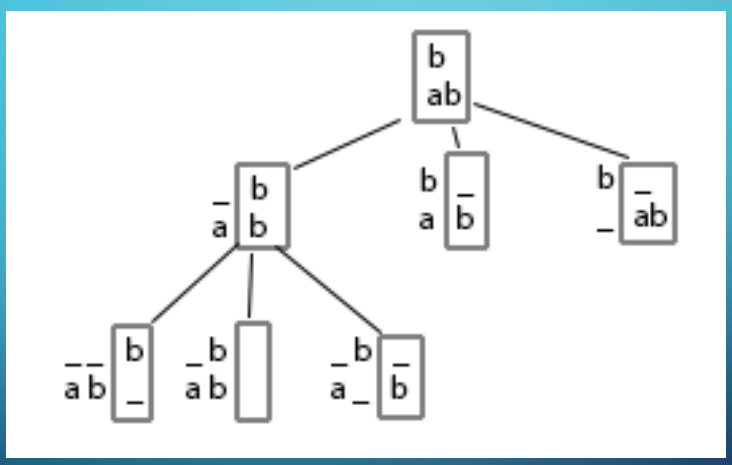
S:	a	b	r	_	a	c	a	_	d	a	b	r	a
T:	a	v	a	d	a	\s	k	e	d	a	v	r	a
Inst:	|	*	*	t	|	*	*	t	|	|	*	|	|

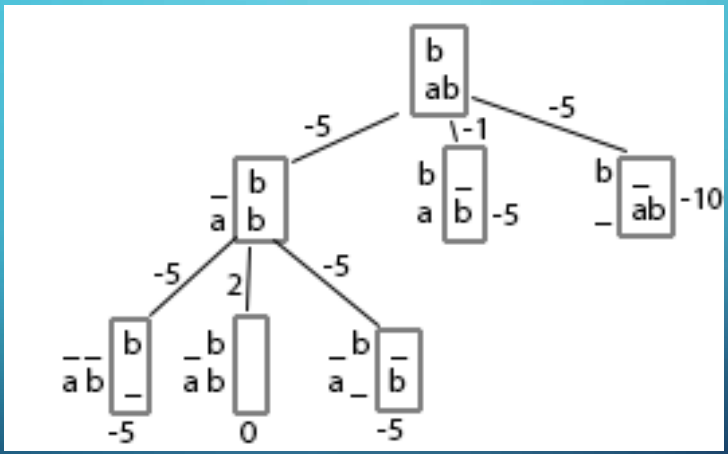
Total Score: -3



ALGORITHM?

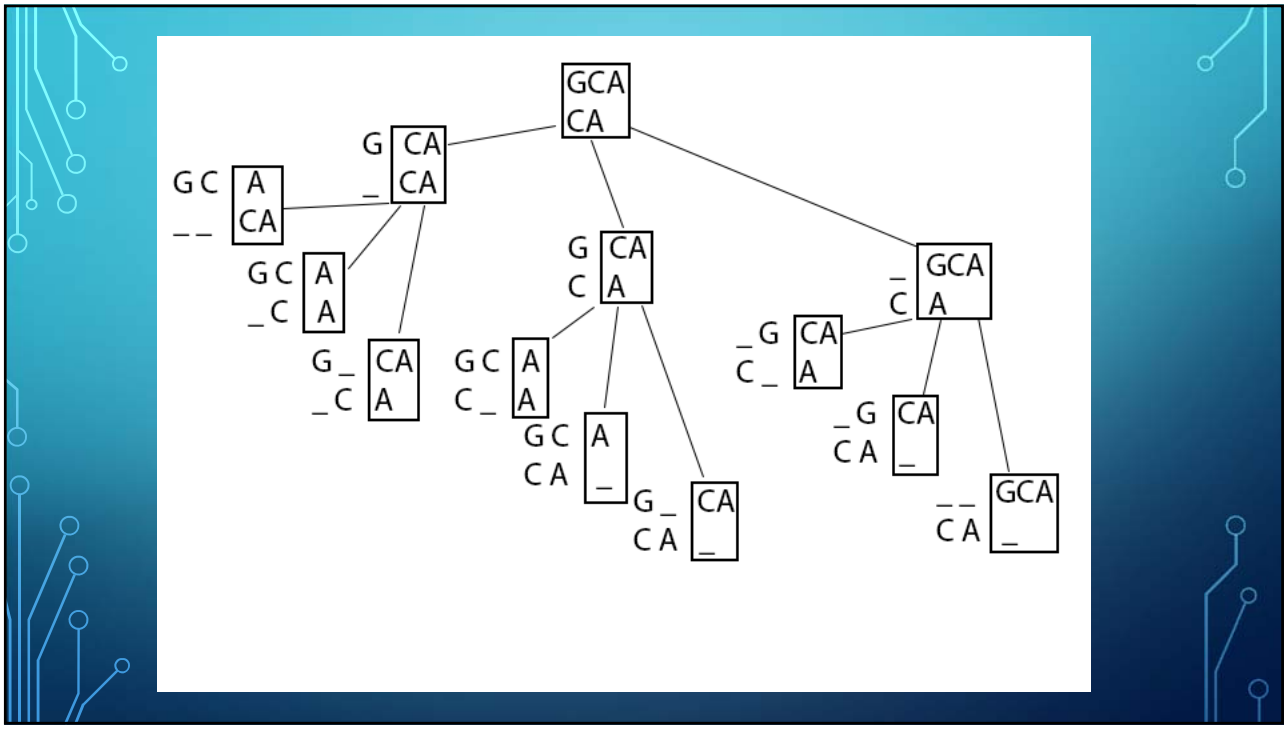
- Reduce to a recursive problem
- Make a function call(s) to compare shorter strings
- For each function call, find scores recursively for:
  - A gap in the second string
  - A gap in the first string
  - No gaps
- Remove the first character of one or both strings
- Add the score for the first position and return the highest of 3 scores
- Base Case: aligning two strings, one is empty

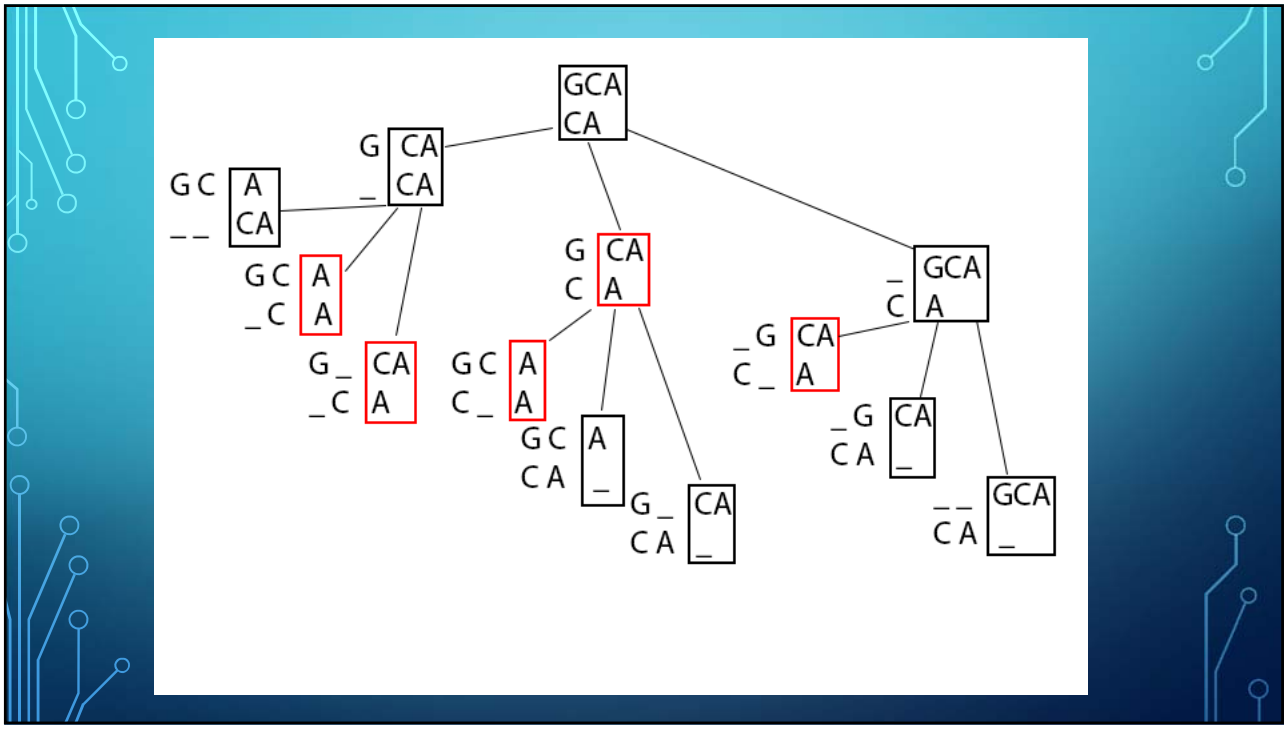




MEMOIZE

- Each function call will generate 3 function calls
- Time complexity at least O(3^N) where N is len of compared result
- Gets VERY slow very quickly
- Solution?
- Dynamic Programming


GCA
CA
G
_
CA
CA
G C
_ _
A
CA
G C
_ C
A
A
G _
_ C
CA
A
G
C
CA
A
G C
C _
A
A
G C
C A
A
_
G _
C A
CA
_
_
C
GCA
A
_ G
C _
CA
A
_ G
C A
CA
_
_ _
C A
GCA
_


GCA
CA
G
_
CA
CA
G C
_ _
A
CA
G C
_ C
A
A
G _
_ C
CA
A
G
C
CA
A
G C
C _
A
A
G C
C A
A
_
G _
C A
CA
_
_
C
GCA
A
_ G
C _
CA
A
_ G
C A
CA
_
_ _
C A
GCA
_

MANY REDUNDANT FUNCTION CALLS

- Memoize!
- Store the results of each comparison in a map
- The key is string1 + "," + string2
- The value is a struct containing an int score and a string for alignment instructions

RESULTS

w/
Dynamic Programming

```
Calling DNA align on strings abracadabra, avada kedavra
abr aca dabra
avada kedavra
|**t|**t||*||
Score for this alignment: -3
Number of calls: 168
```

w/o
Dynamic Programming

```
Calling DNA align on strings abracadabra, avada kedavra
abr aca dabra
avada kedavra
|**t|**t||*||
Score for this alignment: -3
Number of calls: 336447346
```

MORE APPLICATIONS OF DYNAMIC PROGRAMMING

BAD

ALSO BAD

SEAM CARVING AKA CONTENT AWARE IMAGE RESIZING

SEAM CARVING ALGORITHM

- Find the paths of "lowest energy" aka the seams
- "Energy" refers to how noticeable each pixel in the seam is with respect to the rest of the image
- Continuously remove seams to shrink the image a pixel at a time
- Energy found using a saliency map
- For each pixel, assign a value based on how different it is from its neighbors
- Already done for you

MORE ALGORITHM

- Cost map represented as 2D array of unsigned ints
- Must find cheapest path from top to bottom
- Brute force?
- Complexity O(W^L) where w and l are width and length of image

CREATING A COST TABLE

- Records the cheapest cost to each pixel
- First row is identical to first row of saliency map
- Build table row by row by calculating shortest path from above 3 pixels
- Each row memoizes calculations to use in the next row
- At the end, find lowest cost pixel in bottom row and backtrace steps

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7			

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7	10		

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7	10	1	

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7	10	1	6

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7	10	1	6
11	9	8	3

SALIENCY MAP

5	6	1	3
2	9	0	5
4	8	7	2

COST TABLE

5	6	1	3
7	10	1	6
11	9	8	3

NEW IMAGE WITH SEAM REMOVED

5	6	3
2	9	5
4	8	7

5	6	3
7	12	8
11	15	15

NEW IMAGE WITH SEAM REMOVED

5	6	3
2	9	5
4	8	7

5	6	3
7	12	8
11	15	15

FINAL NOTES ON SEAM CARVING

- Dynamic programming algorithm is approx. O(W*L)
- Saliency map represents a 2D array but is stored as 1D array
- We provide a get(array, x, y) to automatically translate 2D bounds
- As image shrinks, iteration bounds should change but array size does not
- Return an array containing the x coordinate of seam pixel starting from top

GOOD LUCK; HAVE FUN

OFFICE HOURS ETC.