CS 2 Recitation 5: Dynamic programming

1/29/2016

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

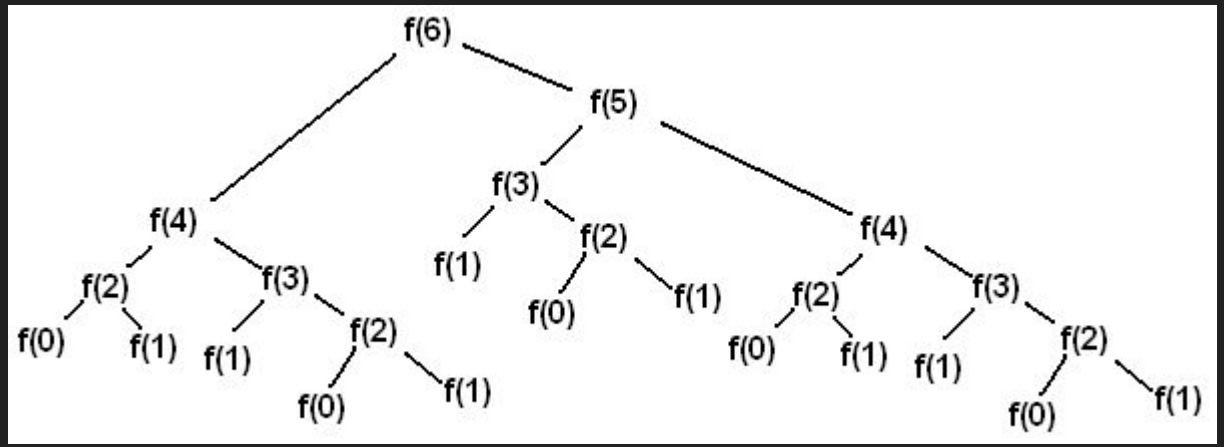
- Trades space for time by saving the results of previous function calls
- Makes otherwise unfeasible algorithms possible
- Two requirements
  - Optimal substructure
  - Overlapping subproblems
- Time complexity = Number of stored elements * time for each element

Simple Example: Fibonacci sequence

Fibonacci(n):

If n is 0 or 1 return n

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

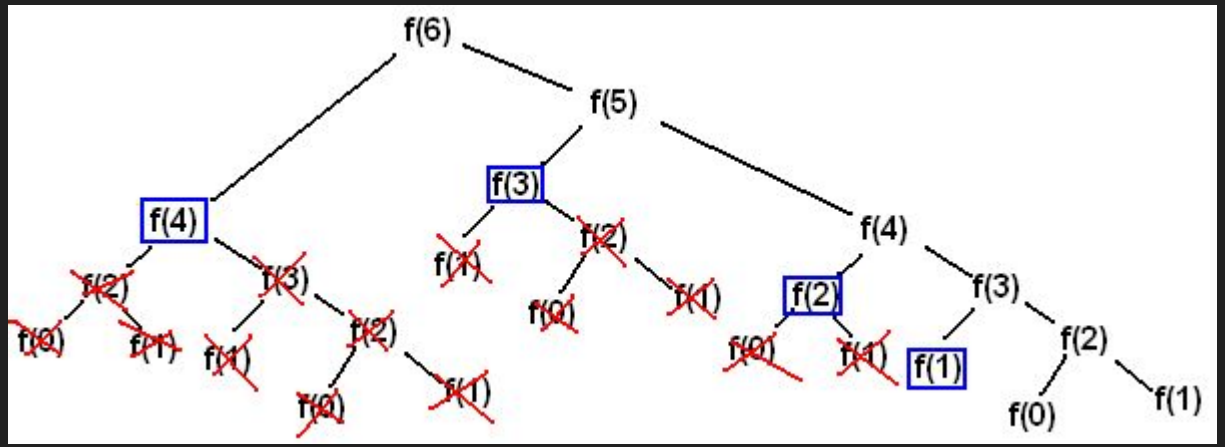


Programming

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
    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. ACAGCGGT

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

Total score: +5

Another Example

abracadabra vs. avada kedavra

S: a b r _ a c a _ d a b r a

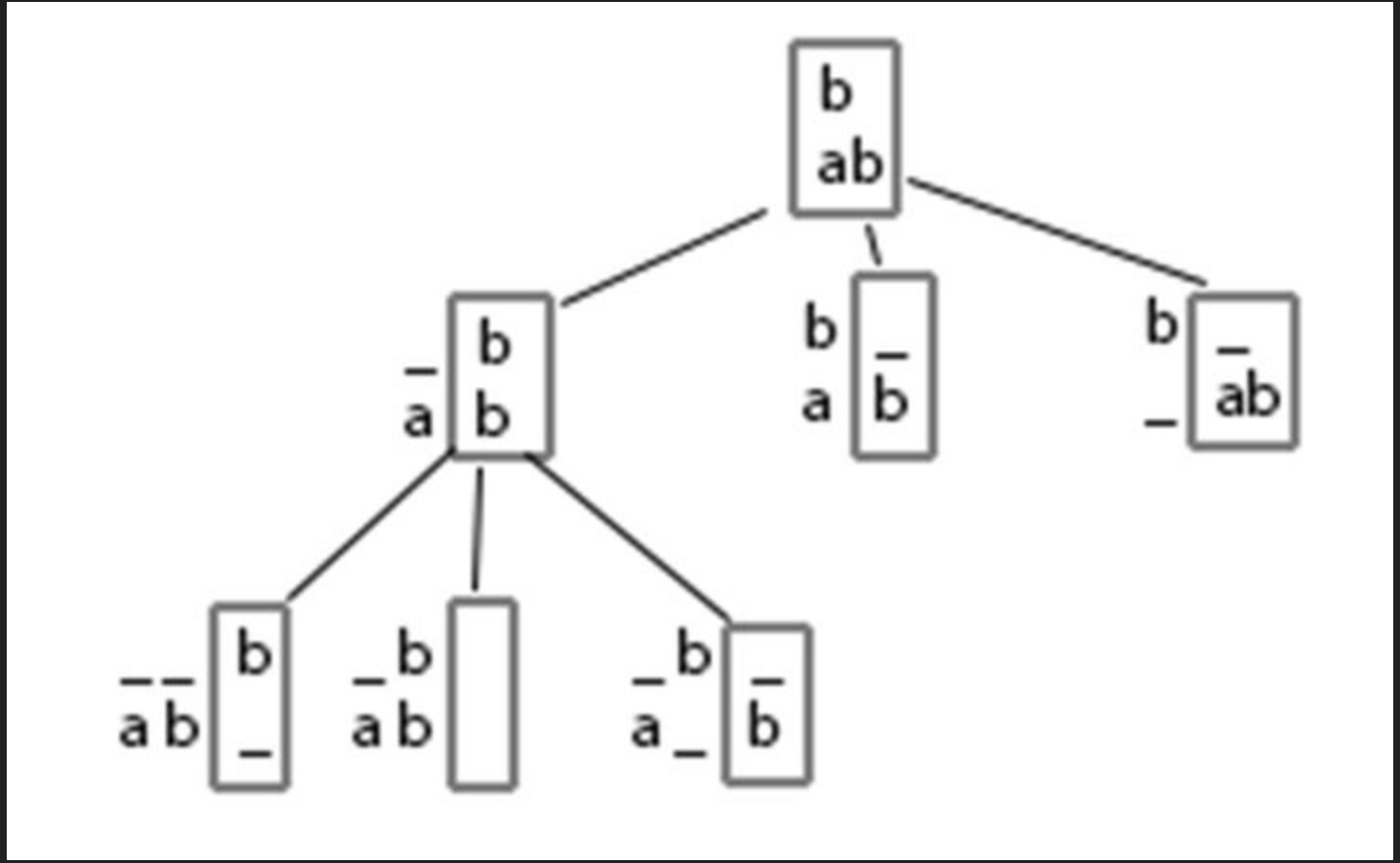
T: a v a d a \s k e d a v r a

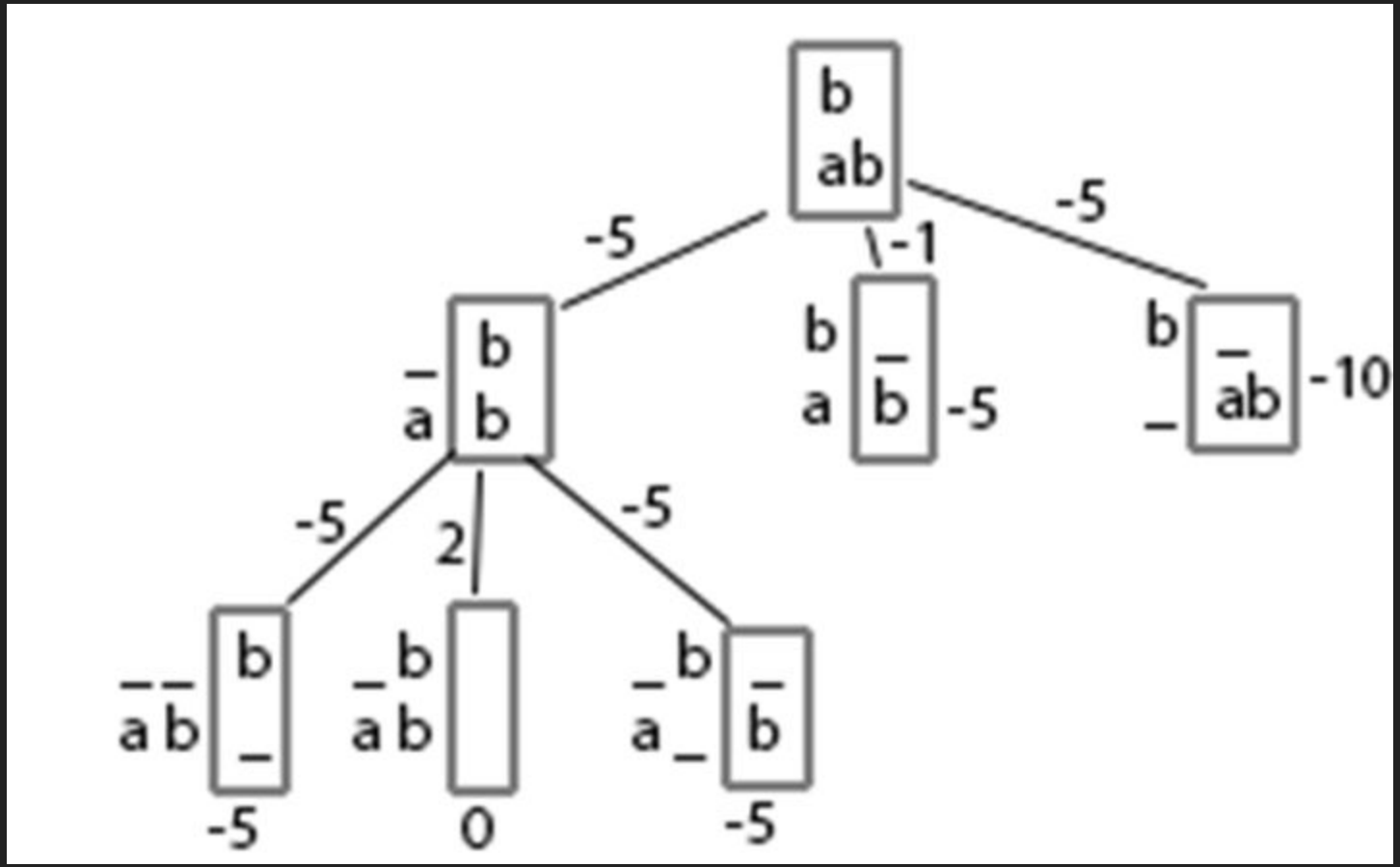
Inst: | * * t | * * t | | * | |

Total Score: -3

Algorithm?

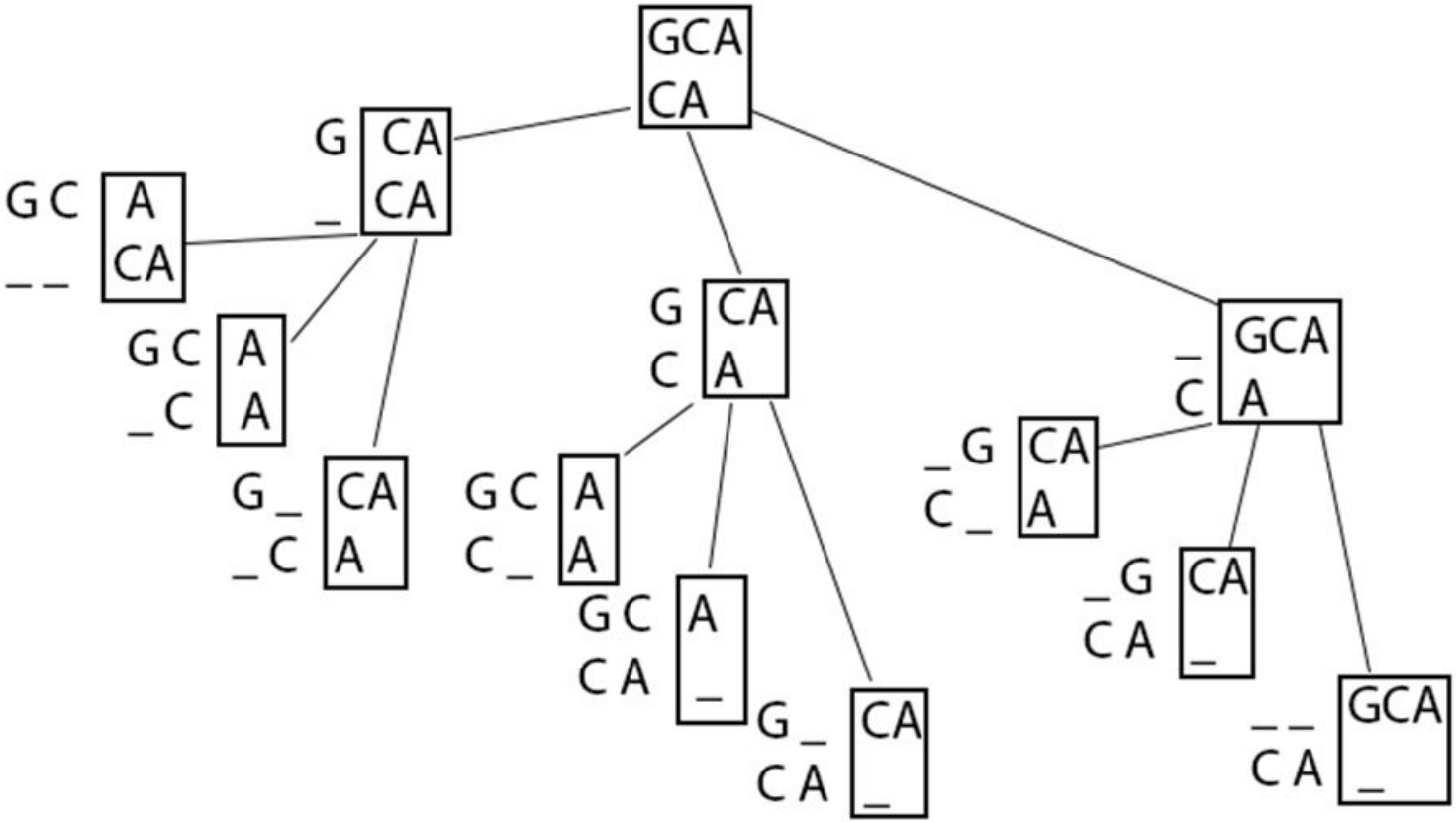
- Given two strings to compare, returns the optimal alignment
- Base Case: aligning two strings, one is empty
- Make recursive call(s) to compare shorter strings:
  - A gap in the second string, remove char from s1
  - A gap in the first string, remove char from s2
  - No gaps, remove char from both
- Get 3 alignments back. Add the score for the removed char(s) to each.
- Optimal alignment is the best alignment of the 3, return it

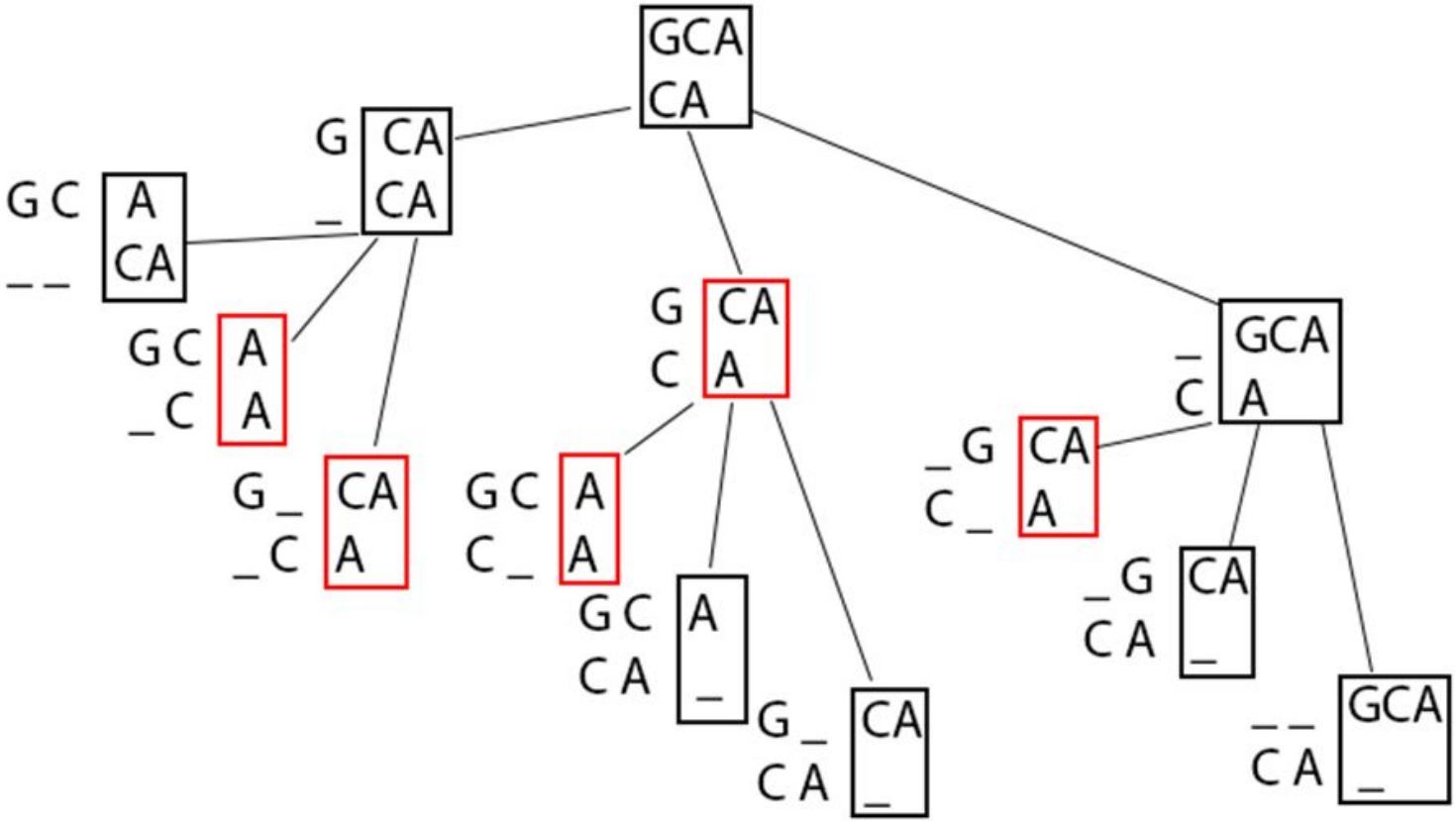

b
ab
_
a
b
b
b
_
a
b
b
_
_
ab
_ _
a b
b
_
_ b
a b
_ b
a _
_
b


b
ab
-5
-1
-5
b
_
a
b
-5
b
_
ab
-10
b
_
a
b
-5
2
-5
b
_
_
a
b
-5
b
_
_
a
b
0
b
_
_
a
_
b
-5

Memoize

- Each function call will generate 3 function calls
- Time complexity is O(3^N) where N is length of the longer input
- Gets VERY slow very quickly
- Solution?
- Dynamic Programming


GCA
CA
G CA
_ CA
GC A
__ CA
GC A
_C A
G_ CA
_C A
G CA
C A
GC A
C_ A
GC A
CA _
G_ CA
CA _
_ GCA
C A
_G CA
C_ A
_G CA
CA _
__ GCA
CA _


GCA
CA
G CA
_ CA
GC A
_ _ CA
GC A
_C A
G_ CA
_C A
G CA
C A
GC A
C_ A
GC A
CA _
G_ CA
CA _
_ GCA
C A
_G CA
C_ A
_G CA
CA _
_ _ GCA
CA _

Many redundant function calls

- 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
```

Complexity

- Number of elements stored * time for each element
- = (N * M) * (1)
- O(N*M) where N and M are the lengths of the two strings

More applications of dynamic programming

Bad

Also bad

Seam carving

Seam carving algorithm

- Seams are the least noticeable paths from top to bottom
- Remove seams to shrink the image a pixel at a time
- Seams found using a saliency map
- For each pixel, assign a value based on how different it is from its neighbors (already done for you)

Seam carving algorithm

- Cost map represented as 2D array of unsigned ints
- Must find cheapest path from top to bottom
- Brute force?
- Complexity O(W3^H) where w and h are width and height of the image

Creating a cost table

- Records the cheapest cost from top 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

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

Saliency Map

5	6	3
2	9	5
4	8	7

Cost Table

5	6	3
7	12	8
11	15	15

Saliency Map

5	6	3
2	9	5
4	8	7

Cost Table

5	6	3
7	12	8
1 1	15	15

Implementation notes

- Dynamic programming complexity - size of table * time per element = O(W*L)
- Return an array containing the x coordinate of seam pixel starting from top

Initial Seams

Other Dynamic Programming Problems

- Given a monetary amount, how many ways can you make it with a set of coins?
- Given a dictionary of words, what is the longest chain of words you can make by removing letters (i.e. price->rice->ice) ?
- Given a capacity W and objects with weights and values, what is the maximum value you can get without going over capacity (aka knapsack problem) ?