Dynamic Programming
Michael Schatz (mschatz@cshl.edu)
==

Motivation:
  Many optimization problems can be naively solved with an exhaustive search
  in O(2^n) time or worse. However, many of these optimization problems have a
  particular form that allows them to be solved much faster (O(N^2)) using
  a bottom-up approach called dynamic programming. This is possible iff
  the problems have (1) overlapping subproblems and (2) optimal substructure.

1. Top-down versus Bottom-up recursion
==

Consider the Fibonacci sequence: F(0) = 0; F(1) = 1; F(n) = F(n-1) + F(n-2)

We can compute it "top-down" using a recursive implementation

```
def fib(n):
 if n == 0 or n == 1:
   return n
 else:
   return fib(n-1) + fib(n-2)
```

Lets draw the recursion tree for it:

```
                             F(6)
                           /      \
                         /          \
                       /              \
                     /                  \
                   F(5)                  F(4)
                 /     \                  /\
               /         \              /    \
             /             \          /        \
           F(4)            F(3)      F(3)      F(2)
         /    \             /\      / \       / \
        /      \           /  \    /   \     /   \
      F(3)      F(2)     F(2) F(1) F(2) F(1) F(1) F(0)
      /  \      / \      / \       / \
   F(2) F(1) F(1) F(0) F(1) F(0) F(1) F(0)
   /  \
F(1) F(0)
```

Solved

```
                              8
                          /       \
                        /           \
                      /               \
                    /                   \
                   5                      3
                 /     \                  /\
               /         \              /    \
             /             \          /        \
           3                2         2         1
         /    \             /\      / \       / \
        /      \           /  \    /   \     /   \
       2         1        1    1   1     1   1     0
      /  \      / \      / \      / \
    1     1    1   0    1   0    1   0
   /  \
  1    0
```

What is the running time?

  Notice that the recursion is creating a tree of height n (leftmost)
  to height n/2 (rightmost). Each node branches by 2, so the overall number
  of steps is O(2^n).

But this is incredibly wasteful, the values of F(X) are recomputed many times:
F(0): 5; F(1): 8; F(2):4; F(3): 3; F(4): 2; F(5): 1; F(6): 1

Instead of computing top-down, lets compute it bottom up:

```
def fib(n):
 table = [0] * (n+1)
 table[0] = 0
 table[1] = 1
 for i in xrange(2,n+1):
   table[i] = table[i-2] + table[i-1]
 return table[n]
```

Initialization (zeros)

idx	0	1	2	3	4	5	6
table	0	0	0	0	0	0	0

Initialization (base case)

idx	0	1	2	3	4	5	6
table	0	1	0	0	0	0	0

For loop

idx	0	1	2	3	4	5	6
table	0	1	1	2	3	5	8

What is the running time?
```
  initialization: O(n)
  for loop:       O(n)

  overall:        O(n)
```

The fast bottom-up approach works because computing the Fibonacci sequence has overlapping subproblems (subproblems of F(x) reused multiple times) with optimal substructure (computing the final solution can be efficiently constructed from optimal solutions to subproblems).

```
                   F(6)
                   /  \
                F(5) - F(4)
                  |  /  |
                F(3) - F(2)
                  |  /  |
                F(1)   F(0)
```

Anti-example: Cheapest flight from NYC to SFO has a stop in ORD, but cheapest flight from NYC to ORD passes through ATL.

Advanced Alternate technique: "Memoization"
Remember the solutions along the way: more general approach, but often slower
----

```
table = {}
def fib(n):
 global table
 if table.has_key(n):
   return table[n]
 if n == 0 or n == 1:
   table[n] = n
   return n
 else:
   value = fib(n-1) + fib(n-2)
   table[n] = value
   return value
```

2. Longest Increasing Subsequence

Problem statement: Given a sequence of N numbers A1, A2, ... An, find the longest monotonically increasing subsequence

Example:

```
    29,  6, 14, 31, 39, 78, 63, 50, 13, 64, 61, 62, 19
```

Greedy approach:

```
    29,   ,   , 31, 39, 78,   ,   ,   ,   ,   ,   ,    =>  4
```

Is that optimal?

No. If you swap in 6, 14 for 29, you can increase the length to 5. There might be some beneficial swaps at the end of the list. We need a systematic method to explore possible swaps

Brute force:

Enumerate through the powerset of all possible subsequences. Check to see if each one is a valid increasing subsequence or not

```
    29,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,       => valid, 1
    29,  6,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,       => invalid
    29,   , 14,   ,   ,   ,   ,   ,   ,   ,   ,   ,       => invalid
    29,   ,   , 31,   ,   ,   ,   ,   ,   ,   ,   ,       => valid, 2
    ...
    29,  6, 14,   ,   ,   ,   ,   ,   ,   ,   ,   ,       => invalid
    29,  6,   , 31,   ,   ,   ,   ,   ,   ,   ,   ,       => invalid
    ...
      ,  6, 14, 31,   ,   ,   ,   ,   ,   ,   ,   ,       => valid, 3
    ...
```

We can turn this into a recursive definition:

```
  LIS(j) = 1 + max (LIS(1), LIS(2), LIS(3), ... LIS(j-1))
```

This works, but requires O(2^n) time to explore every possible subsequence

Pruning invalid searches, and branch-and-bound will help but no guarantees the running time will substantially improve. Unlike quicksort, recursion doesnt help because the subproblem is not much smaller than the original problem.

Dynamic Programming Solution to LIS:

The solution for all N values depends on the solution for the first N-1 values. Look through the previous values to find the longest subsequence ending at X such that Ax < An

```
  Def: LIS[i] is the longest increasing subsequence ending at position i
       Base case: LIS[0] = 0;
       Recurrence: LIS[i] = max_{h<i; A[h] < A[i]} { LIS[h] + 1 }
```

idx	0	1	2	3	4	5	6	7	8	9	10	11	12
val	29,	6,	14,	31,	39,	78,	63,	50,	13,	64,	61,	62,	19
LIS	1	1	2	3	4	5	5	5	2	6	6	7,	3
prev	0	0	2	3	4	5	5	5	2	7	8	11,	3

Solution: After evaluating the dynamic programming algorithm, the solution is the maximum element in the LIS table (7, ending at 62)

To find the sequence, keep track of previous pointers in a parallel array, and backtrack to the beginning
62 (11) -> 61 (8) -> 50 (5) -> 39 (4) -> 31 (3) -> 14 (2) -> 6 (0)

Note there may be more than one increasing subsequence that has maximum length. The particular one selected just depends on how the code is implemented (64 could link up with 50 or 63 to reach a length 6 chain)

```
 Running time:
    Initialization:   O(N)
    LIS Outer loop x Inner loop:    O(N) x O(N) = O(N^2)
    Find LIS Length   O(N)
    Backtracking:     O(N)

Note: There is an even faster DP strategy that can solve it in O(N lg N)

Python Implementation:
--------------------------------------------------------------------------------

def compute_lis(A):
    ## initialize
    LIS = [0] * len(A)
    P   = [0] * len(A)

    ## compute the LIS ending at every position
    for i in xrange(0, len(A)):
        bestlis = 0
        bestidx = -1

        for j in xrange (0, i):
            if ((A[j] < A[i]) and (LIS[j] > bestlis)):
                bestlis = LIS[j]
                bestidx = j
        LIS[i] = bestlis + 1
        P[i] = bestidx

    ## Print the matrices
    print "A:   " + str(A)
    print "LIS: " + str(LIS)
    print "P:   " + str(P)

    ## Compute the LIS length
    lis = 0
    lisidx = -1
    for i in xrange(0, len(A)):
        if (LIS[i] > lis):
            lis = LIS[i]
            lisidx = i

    print "The LIS has length %d ending at pos %d" % (lis, lisidx)

    ## Backtrack to print out the LIS
    while (lisidx != -1):
        l = LIS[lisidx]
        p = P[lisidx]
        a = A[lisidx]

        print "%d: A[%d]=%d (%d)" % (l, lisidx, a, p)

        lisidx = p


A = [29, 6, 14, 31, 39, 78, 63, 50, 13, 64, 61, 62, 19]
compute_lis(A)

Output
--------------------------------------------------------------------------------

A:   [29, 6, 14, 31, 39, 78, 63, 50, 13, 64, 61, 62, 19]
LIS: [1, 1, 2, 3, 4, 5, 5, 5, 2, 6, 6, 7, 3]
P:   [-1, -1, 1, 2, 3, 4, 4, 4, 1, 6, 7, 10, 2]
The LIS has length 7 ending at pos 11
7: A[11]=62 (10)
6: A[10]=61 (7)
5: A[7]=50 (4)
4: A[4]=39 (3)
3: A[3]=31 (2)
2: A[2]=14 (1)
1: A[1]=6 (-1)
```

```
3. Edit distance
================================================================================

Last time we talked extensively about exact matching using an index to
accelerate the search. Given these algorithms, a widely used approach for
in-exact alignment is "seed-and-extend". The basic idea is for there to be a
"good" in-exact alignment there must be some segment that exactly matches. We
can rapidly find the exact matches (the seeds), and then check the flanking
characters to see how "good" the end-to-end match is.


       ---------------------GATTACA-------------------
                            |||x|||
                            GATCACA

Here we could use the short seeds GAT or ACA to anchor and then check the flanking
bases to discover the off-by-one alignment. This simple way to count differences
is called the hamming distance or Manhattan distance, and counts the number
of substitutions to transform one sequence into another.

A more general metric is called the "edit distance" or Levenshtein distance,
that counts the number of substitutions, insertions, or deletions:

       --------------------MICHAELSCHATZ-----------------
                           ||||||||x||||
                           MICHAELS-HATZ


     Has edit distance of 1 versus a hamming distance of 4

       --------------------MICHAELSCHATZ-----------------
                           ||||||||xxxx
                           MICHAELSHATZ


How do we compute the edit distance of AGCACACA and ACACACTA?

One possible alignment:

0. aGcacaca     change G to C
1. acCacaca     delete 2nd C
2. acacacA      change A to T
3. acacacT      insert A after T
4. ACACACTA     done

This implies the edit distance is at most 4. Is this the best we can do?

0. aGcacaca     change G to C
1. acCacaca     delete C
2. acacaCa      insert T after 3rd C
3. ACACACTA     done

This implies the edit distance is at most 3.

Is this the best we can do?  Maybe, we need a systematic way to evaluate
possible edits.



Recursive edit distance
--------------------------------------------------------------------------------

       D(AGCACACA, ACACACTA) = ?

Imagine we have the optimal alignment of the strings, the last column can only
be 1 of 3 options:

         ...M              ...I                ...D
         ...A              ...-                ...A
         ...A              ...A                ...-

          D=0               D=1                 D=1
```

These are the only options because it would be suboptimal to have a gap over gap
From this point of view A/A is the best choice of the partial alignment, adding
cost of 0 to the previous score, while I or D add 1

The optimal alignment of the last two columns is then 1 of 9 possibilities

```
    ...MM ...IM ...DM     ...MI ...II ...DI        ...MD ...ID ...DD
    ...CA ...-A ...CA     ...A- ...-- ...A-        ...CA ...-A ...CA
    ...TA ...TA ...-A     ...TA ...TA ...-A        ...A- ...A- ...--

      D=1   D=1   D=1       D=2   D=2   D=2          D=2   D=2   D=2
```

The optimal alignment of the last 3 columns is then 1 of 27 possibilities

```
         ...M...      ...I...       ...D...
         ...X...      ...-...       ...X...
         ...Y...      ...Y...       ...-...
```

Eventually will spell out every possible optimal sequence of {I,M,D}

For scoring purposes, we will introduce a function s(x,y) that returns 0
if they are the same or 1 if they are different.

With this, we can define the edit distance recursively as:

```
     D(AGCACACA, ACACACTA) = min{ D(AGCACAC, ACACACT) + s(A, A),
                                  D(AGCACACA, ACACACT) + 1,
                                  D(AGCACAC, ACACACTA) + 1 }
```

```
                                  D(8,8)
                                    |
                  ===============================
                 /                  |                \
                /                   |                 \
          +1d  /                    |+s                \ +1i
              /                     |                   \
          D(8,7)                 D(7,7)                 D(7,8)
          /  |  \                /  |  \                /  |  \
         /   |   \              /   |   \              /   |   \
        /    |    \            /    |    \            /    |    \
       /     |     \          /     |     \          /     |     \
    D(8,6) D(7,6) D(7,7)   D(7,6) D(6,6) D(6,7)   D(7,7) D(6,7) D(6,8)
```

Each node branches recursively, considering a deletion, a substitution, or an
insertion. The subproblems get smaller by at least one character in each step,
so it will terminate in at most N levels, but will take O(3^n) steps!

Edit distance by dynamic programming

Instead of recursion, lets try a dynamic programming approach filling in a
M x N matrix bottom up considering all pairs of possible prefixes of the
strings S and T. This will save a considerable amount of time, since the same
subproblems arise over and over again (notice D(7,7) occurs 3 times above)

Initialize:

  Aligning any prefix of length l to an empty string costs l edits

  D(i,0) = i for all i
  D(0,j) = j for all j

```
       0  A  C  A  C  A  C  T  A
     0 0  1  2  3  4  5  6  7  8
     A 1
     G 2
     C 3
     A 4
     C 5
     A 6
     C 7
     A 8
```

```
Recurrence: fill in from top to bottom, left to right
            Each cell only depends on 3 neighbors: left, up, and diagonal

  D(i,j) = min {
                 D(i-1, j) + 1 // align 0 characters of S, 1 from T
                 D(i, j-1) + 1 // align 1 characters of S, 0 from T
                 D(i-1, j-1) + s(S[i], T[i]) // align 1 from S, 1 from T
               }


    D(1,1) = D(A,A) = min{D[A,] + 1, D[,A]+1, D[,] + s(A,A)}
                    = min{1+1,       1+1,     0}
                    = 0

    D(1,2) = D(A,AC) = min(D[A,A]+1, D[,AC]+1, D[,A] + s(A,C)}
                     = min(0 + 1,    2+1,      1+1)
                     = 1

       0  A  C  A  C  A  C  T  A
     0 0  1  2  3  4  5  6  7  8
     A 1  0  1  2  3  4  5  6  7
     G 2
     C 3
     A 4
     C 5
     A 6
     C 7
     A 8

After the first row is done, we know the edit distance of D(ACACACTA, A) = 7
Now compute the second row to compute D(ACACACTA, AG) = 7
Now compute the third row to compute D(ACACACTA, AGC) = 7
...

Complete the matrix:

       0  A  C  A  C  A  C  T  A
     0 0  1  2  3  4  5  6  7  8
     A 1  0  1  2  3  4  5  6  7
     G 2  1  1  2  3  4  5  6  7
     C 3  2  1  2  2  3  4  5  6
     A 4  3  2  1  2  2  3  4  5
     C 5  4  3  2  1  2  2  3  4
     A 6  5  4  3  2  1  2  3  3
     C 7  6  5  4  3  2  1  2  3
     A 8  7  6  5  4  3  2  2  2

The edit distance is the value in the lower right corner: 2


Like LIS, keep a parallel matrix with back pointers to find
the complete alignment. A diagonal move aligns a character on top of a character,
a move to the left aligns a character from the top string to a gap, a move up
aligns a character of the left string to a gap.

       0  A  C  A  C  A  C  T  A
     0 0
     A    0
     G    1
     C       1
     A          1
     C             1
     A                1
     C                   1  2
     A                         2

      AGCACAC-A
      |*|||||*|     D=2
      A-CACACTA

Note there may be multiple possible ways to backtrack to get the same score.
```

```
Edit Distance in Python
-------------------------------------------------------------------------------
## Thanks to http://people.cs.umass.edu/~mccallum/courses/cl2006/lect4-stredit.pdf

import sys

def stredit (S,T):
    len1 = len(S) # vertically
    len2 = len(T) # horizontally

    print "Aligning " + S + " and " + T

    # Allocate the table
    table = [None]*(len2+1)
    for i in xrange(len2+1): table[i] = [0]*(len1+1)

    # Initialize the table
    for i in xrange(1, len2+1): table[i][0] = i
    for i in xrange(1, len1+1): table[0][i] = i

    # Do dynamic programming
    for i in xrange(1,len2+1):
        for j in xrange(1,len1+1):
            d = 1
            if S[j-1] == T[i-1]:
                d = 0
            table[i][j] = min(table[i-1][j-1] + d,
                              table[i-1][j]+1,
                              table[i][j-1]+1)

    sys.stdout.write("      0");
    for j in xrange(0,len1):
        sys.stdout.write("   " + S[j])
    print

    for i in xrange(0,len2+1):
        if (i>0):
            sys.stdout.write("  " + T[i-1])
        else:
            sys.stdout.write("  0")

        for j in xrange(0,len1+1):
            sys.stdout.write("  %2d" % table[i][j])

        print

S="ACACACTA"
T="AGCACACA"

S="MICHAELSCHATZ"
T="MICHELSHATZ"
stredit(S,T)


### See the slides for remaining topics
```