Dynamic Programming

Ryan Filgas

Dynamic Programming Solutions

CS350

```
// MEMOIZATION - DYNAMIC PROGRAMMING STRATEGY
// using memorization reduces this algorithm to T sub m O(n)
// To make brute force recursive algorithms more efficient:

// make a table 't' as big as the input size
// initialize the table to 0, -1, -1, -1, -1.......
// the tabel will hold the results of the f function
// any time the function is called on a number, check the table. If the value
// isn't -1, return the value.

// minimum change "greedy" algorithm
int t[0, 1, 2, 3, 4, 5, 6, 7, 8];

// this becomes a linear time function!!
int f(int n){
  if(n < 0) {return 10000000}; //return infinity
  if(t[n] != -1){ return t[n]}; // return the answer if it's here already

  // do the work if it's not
  a = f(n-1); //call children
  b = f(n-3); // this number subtracted matches up with the coin values!
  c = f(n-4);
  min = a;
  if(b < min)
    min = b;
  if(c < min)
    min = c;
  int r = 1 + min; // count our coin!
  t[n] = r; // record our answer in the table
  return r;
}
```

```
// Tabulation: - no recursion used! - requires recursive thinking, but
// not recursive code. Calculate possible values starting from 0 -> n.
// "Dynamic Programming"
// f(n) = 1 + min(f(n-1), f(n-3), f(n-4));
// if (n < 0) return 9999999;
// if (n == 0) return 0;

//------------------------------------------------------------
// coin problem with tabulation
int table[n];

//initiate table
for(int i = 0; i < n; ++i){
  table[i] = -1;
}

// tabulate by getting the solutions from 0 -> n using each previous solution.
for(int i = 0; i < n; ++i){
  table[i] = f[i]; // f(0) is the base case. f(1) gives the minimum of f(
}
int f(int n, int table){
  if(n < 0)
      return 999999;
  if(n == 0)
      return 0;
  //count ourselves as one coin, then the minimum of our "children".
  int num = 1 + minimum(table(n-4), table(n-3), table(n-1));
  return num;
}
```

```
//Backtracing: (still in the hypothetical. have not implemented.)
// backtrace the solution by finding the minimum of the previous entries
// If there is a tie, record the state in a separate branch of the solution set.
//------------------------------------------------------------

// build a table and calculate values ahead of time.
// f(0) = 0;
// f(1) = 1;
// f(2) = 2;
// f(3) = 1;  //comes from 4 back, pick the minimum of the ones before
// f(4) = 1;  // since we've filled out the ones before we can fill out this
// f(5) = 2;
// f(6) = 2;
// f(7) = 2;
// f(8) = 2;
// f(9) = 3;

// the next entry is 1 plus the minimum of (n-1, n-3, n-4 -> to represent coins)
// traversing backwards through this allows us to find which coins
// the time for the tabular method is T sub T(n)cO(n)
```

Marble collection:

Naive solution: recurse right and down, returning the path with the most marbles.

    if we hit the bottom, go right until the end.

    if we hit the right, go down until the end.

Memoization of pebble problem: keep a table with number of recursive calls mapped to the minimum for that path. If a path is smaller than the max at that iteration it immediately stops. If it is larger, it replaces the value. If it is the same, it continues to the next recursive call. This will control for different calls finishing at different times.

call_array[0,0,0,0,0,0,0,0,0,0]; Each index represents an iteration, and the solution table is the size of n. The number of rows. The answer will be at the final index.

tabulation of pebble problem:

```
for rows in matrix
    for columns in matrix
        Each index becomes the larger of left and upwards indices. Add in my value. It will be a 0 for no pebble, or. 1 for a pebble. By the time we hit the bottom right we should have the max path.
        This will require working from the top row and proceeding down left to right.
```

```
//Minimum Change Problem with n = 1, 3, 5
Initialize:
[-1][-1][-1][-1][-1][-1][-1][-1][-1]

 B. B. B. B. B. B. ---> Base cases & tabulated numbers after.
[0][1][2][1][2][1][2][3][2][3]

int change(int n, int table[n+1]){
    //initiate table
    for(int i = 0; i < n+1; ++i){
        table[i] = -1;
    }
    // tabulate
    for(int i = 0; i < n+1; ++i){
        table[i] = f(i, table);
    }
    return table[n];
}
int f(int n, int table[n+1]){
    if(n < 0)
        return 999999;
    if(n == 0)
        return 0;
    if(n == 1 || n == 3 || n == 5)
        return 1;
    if(n == 2)
        return 2;
    if(n == 4)
        return 2;
    int T1 = table[n-1];
    int T3 = table[n-3];
    int T5 = table[n-5];
    int min = minimum(T1, T3, T5);
    return 1 + min;
}

// Time Complexity: Big_theta(n)   -> n*m num coins * price
// Space Complexity:  Big_theta(n) -> n+1 indices.
```

```
// rod cutting problem. We have a list of sizes, and a list of values for each size.
// We have a single rod that can be cut into a number of pieces and sold.
// rod price max problem for n prices and an n length rod.

-> Example:
f(1) = max(  f(0) + p[1]);

f(2) = max( (f(1) + p[1]),
            (f(0) + p[2]),);

f(3) = max( (f(2) + p[1]),
            (f(1) + p[2]),
            (f(0) + p[3]));
//Initialize
// There are n+1 indices. This could be initialized to 0 too.
[index 0 = -1][-1][-1][-1][-1][-1][-1][-1]....[index n = -1]

int rod_problem(int n){
  int table[n];
  int prices[num_prices];
  for(int i = 0; i < n; ++i){ //initialization
    table[i] = -1;
  }
  for(int i = 0; i < n; ++i){
    table[i] = cut_rod(i, table, prices); //tabulation
  }
  return prices[n];
}

//calculate each in terms of the previous
int cut_rod(int n, int table[], int prices[]){
  if(n < 0)
    return -1;
  if(n == 0)
    return 0;
  int temp = 0;
  int max = 0;
  for(int i = 0, record = n; i < n; ++i, --record)
    temp = table[i] + price[record];
    if(temp > max)
      max = temp;
  }
  return max;
}
n=1: 1 For each input, cut_rod does a constant amount of work proportional to input.
n=2: 2 operations
n=3: 3 operations
sum(0 + 1 + 2 + 3 + .... n - 1 ) -> basic summation 0 to n-1 ~ n^2

// Time Complexity: Big_theta(n^2)
 // Space Complexity:  Big_theta(n) -> table size is as big as input
// N choose K

// Design a tabular solution for the recursive n choose k problem
// k combinations of n
// We have five friends, A, b, c, d, e
// only 3 can come to a birthday party
// A, B, C, D, E
// How many ways can I invite 3 out of the five friends
// ORDER DOESN:T MATTER IN COMBINATIONS. ABC = CBA
// formula: n! / (n-k)! * k!
// ABC
// ABD
// ABE
// ACD
// ACE
// ABE
// BCD
// BCE
// BDE
// CDE
// = 5! / [(5-3)! / 3!] = 10 //factorial formula. But not what we're doing here.
// n choose k = formula

//Recursive solution:
// If I select 'A', we're down to 4 choose 2.
// If I exclude albert isn't coming, we have 4 choose 3.

// make a decreasing pyramid from the top to represent a recursive solution.
// Left child (we've selected someone)
// right child(we've excluded someone)

// if n == k, then n choose k is one. return 1.
// if k == 1, n choose 1 is just k. return n.

// on the return, add the results from the two children up to the root.
// 6 results included an 'A', 4 results did not.
// 5 choose 0 = 1.
-----------------------------------------------------------------
// Tabular Solution
(a+b)^2 = a^2 + 2ab + b^2 ---> this is row 3
Initialized:
  0  1  2  3  4 .....n
0[1][0][0][0][0].....
1[0][0][0][0].....
2[0][0][0].....
3[0][0].....
4[0].....
.
.
n

Completed:
  0  1  2  3  4........n
0[1][1][1][1][1].....
1[1][2][3][4].....
2[1][3][6].....
3[1][4].....
4[1].....
.
.
n

// This is pascals triangle.
    1
  1 2 1
 1 3 3 1
1 4 6 4 1

// This overflows really easily so I used unsigned when testing.
// SIZE is a global variable set to N. I was testing different sizes.
long unsigned int nksolve(long unsigned int n, long unsigned int k, long unsigned int array[SIZE][SIZE]){
  // initialize table t
  for (long unsigned int r = 0 ; r < SIZE; r++) {
    for(long unsigned int c = 0 ; c < SIZE ; c++) {
        array[r][c] = 0;
    }
  }
  array[0][0] = 1 ; // initialize top of table.
  long unsigned int check = 0;

  // tabulate
  for (long unsigned int r = 0; r < SIZE; r++) {
    for(long unsigned int c = 0; c < SIZE; c++) {
        if(c == 0 || r == 0)
            array[r][c] = 1;
        else
            array[r][c] = array[r-1][c] + array[r][c-1];
    //if(c > check) array[r][c] = 0;
    }
    ++check;
  }

  return array[n-k][k];
}

// Time Complexity: Big_theta(n*k)
// Space Complexity:  Big_theta(n*k)
--> can reduce space complexity to K by only remembering row used
// Note: Easy creation of the table results in
//      space complexity: Big_theta(n*n)
```

```
Minimum Path Problem
[0]    [0][0][0]..... // I separated the 0s so
[1][2]    [0][0]..... // the structure would
[3][4][5]    [0]..... // be easier to see.
[6][7][8][9]........ //
.................... //
// The numbers are where array indexes go.

int min_path(int input_array[], int array_size, int tree[SIZE][SIZE]){
    for(int i = 0; i < SIZE; ++i){
        for(int j = 0; j < SIZE; ++j){
            tree[i][j] = 0;
        }
    }
    int x = 0;
    int count = 0;
    int row_max = 1;
    while(count < array_size){
        //for each row import array
        for(int y = 0; y < row_max; ++y){
            if(count >= array_size) break;
            tree[x][y] = input_array[count];
            ++count;
        }
        ++row_max;
        ++x;
    }
    // get tree height for equilateral path tree
    int height = get_eqheight(array_size);
    return get_min(tree, height);
}

int get_min(int tree[SIZE][SIZE], int height){
    int level = height;
    while(level >= 0){
        for(int i = 0; i < level; ++i){
            int left = tree[level][i];
            int right = tree[level][i+1];
            tree[level-1][i] += ((left < right) ? left : right);
        }
        --level;
    }
    return tree[0][0];
}
// Get the height of the tree so we know where to start
// I wasn't in the right mental space to find out the calculation
int get_eqheight(int n){
    int i = 1;
    int height = 1;
    while(i < n){
        i = 2*i + 1;
        ++height;
    }
    return height;
}
------------------- ALWAYS DOUBLE CHECK N. N SHOULD BE 4!!
// Time Complexity: CORRECT-> BIG_THETA(N^2) - we updated each node from previous values
// Space Complexity:  CORRECT -> Big_theta(N^2) -> n is a function of the number of
// entries in the bottom row.
```

```
PROFESSOR SOLUTIONS
Solutions to the dynamic programming homework problems:
-------------------------------------------------------------------
4) Modified Change problem: Answer is 3 total coins. Can be obtained via 1 + 3 + 5 or 3 + 3 + 3.
------------------------------------------------------------------- 6) Rod-cutting problem:
double p[1001], t[1001] ;
// t[0] gets initialized to 0
double tabular_rod(int n)
{
  int k, c, a ;
  double max, v[1001] ;
  for(c=1;c<=n;c++) {
// calculate each answer using the known parts of the table for(k=0;k<c;k++) {
      v[k] = t[k] + p[c-k] ;
    }
    // update new table value
    max = v[0] ;
    for(k=1;k<c;k++) {
      if(v[k] > max) {
      max = v[k] ;
} }
    // update new table value
t[c] = max ; }
  return t[n] ;
}
// Time Complexity: 1 + 2 + 3 + 4 + ... + n = Big_theta(n*n) // Space Complexity: Big_theta(n)
------------------------------------------------------------------- 8) Minimum-sum decent:
int t[101][101] ;
// t gets initialized to hold the data to the problem and then // gets overwritten to hold the answer at each step.
// The final answer is in t[0][0]
int min_sum_descent(int n)
{
int r, c, a, b ;
for(r=n-2;r>=0;r--) {
// calculate each answer using the known parts of the table for(c=0;c<=r;c++) {
      a = t[r+1][c] ;  b = t[r+1][c+1] ;
      if(a<b) { t[r][c] += a ; }
      else    { t[r][c] += b ; }
} }
  return t[0][0] ;
}
// Time Complexity: 1 + 2 + 3 + 4 + ... + (n-1) = Big_theta(n*n) // Space Complexity: Big_theta(n*n) [to store the original problem]
------------------------------------------------------------------- n-choose-k problem:
int tchoose(int n, int k)
{
int r,c ;
// initialize table t
for (r = 0 ; r <= n+1 ; r++) {
    for(c = 0 ; c <= k+1 ; c++) {
      t[r][c] = 0 ;
} }
t[0][0] = 1 ; # initialize top of table.
// tabulate
for (r = 1 ; r <= n+1 ; r++) {
for(c = 1 ; c <= k+1 ; c++) {
t[r][c] = t[r-1][c] + t[r-1][c-1] ;
} }
  return t[n+1][k+1] ;
}

Time Complexity: Big_theta(n*k)
```

```
Space Note:
Complexity: Big_theta(n*k) [although you could just save the row you currently need to compute the
next row, resulting in Big_theta(n). Easy creation of the table as a 2d array results in
space complexity: Big_theta(n*n)
```