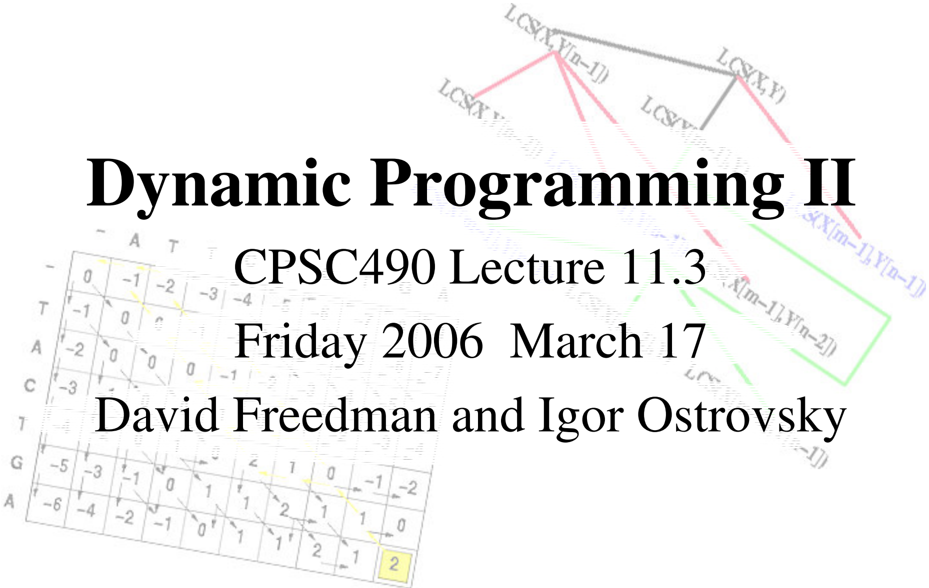


Dynamic Programming II

CPSC490 Lecture 11.3

Friday 2006 March 17

David Freedman and Igor Ostrovsky

Agenda

- Non DP stuff:
  - dUFLP/dHUNG Naming Conventions (xBase)
  - Binary Arithmetic Review
- Today we'll be looking at these DP topics...
  - Memoization
  - The Coin Changer Problem
  - Hamiltonian Paths and Cycles
  - The Travelling Salesman Problem

dUFLP/dHUNG

Naming Conventions

dUFLP/dHUNG Variable Conventions

- Standard for "xBase" family of languages (dBase, Clipper, Paradox, Fox Pro, etc.)
  - dUFLP = dBase Users Functional Library Project, the dBase (later xBase standards group)
  - dHUNG = "Simplified Hungarian," also known as "Short Hungarian" and "Hungarian without the Arian"
- Hallmark feature: The first letter of each variable denotes it's type.
- Key differences between dHUNG and Hungarian notation:
  - type codes correspond to dBase standard type names
  - only 1 letter is used for each type
  - types don't chain (an **a**rray of **in**tegers is just "a" not "an")
- The original goal was that this would be a cross-language standard.
- *bastardizations* of the standard (less standard standards, like the one you're about to see), sometimes find their way into functional/OOP languages.

xBase Types Applied to C/C++/Java/JavaScript/Basic... 5

a **ar**ray (any type/dimensions)

```
int atheArray[];
boolean[][]  atheArray;
var atheArray = { false, 12, { 0, 0 }, "abc" };
```

b code **b**lock

```
/* no Basic, C/C++ or Java equivalent!
   Arguably used in Java for Runnable */
var btheFunc = function() { return "OK"; }
```

c string (**c**haracter field)

```
char[80] ctheString = "abc"; //pre C99!
string ctheString = "abc";
String ctheString = "abc";
char ctheChar = 'a';
```

d **d**ate

```
/*use only 1 of these within a single program!*/
   Date dnow = new Date();
   String dnow = "19991231";
   long dnow = System.currentTimeMillis();
```

f **f**loating point

```
float ftheFloat = 12.34;   double ftheFloat = 12.34;
var famt = 12.34;   Dim famt As Single

/*use only 1 of these within a single program!*/
   Double ftheFloat = new Double(12.34);
   Double FtheFloat = new Double(12.34);
```

l **l**ogical (boolean)

```
int ltheFlag = 0; //pre C99!
bool ltheFlag = false;
boolean ltheFlag = false;
Dim ltheFlag as Integer
```

n **in**teger

```
byte nint = 123;   int nint = 123;   long nint = 123;

/*use only 1 of these within a single program!*/
   Integer nint = new Integer(123);
   Integer Nint = new Integer(123);
```

o **o**bject

```
Scanner oscanner = new Scanner(System.in);
```

v **v**ariant (where the type matters)

```
/* no C/C++ or Java equivalent! */
switch(vtheVar) { case "string": case "number":...
Dim vtheVar As Variant;
```

x variant (where the type is irrelevant)

```
/* no Java equivalent! */
#define IFF(lcond, xa, xb) ( (lcond) ? (xa) : (xb) )
Sub Iff(p_lcond As Integer, p_xa As Any, p_xb As Any)
```

y **cu**rrency (integer cents)

```
long ybalance = 1234; //$12.34
```


Scope Prefix 6

Another prefix, separated by an underscore, denotes scope for all non-local variables (variables declared inside a function or method)

m, g: private/public

- used for both object fields and "file level" variables (top level in an include/header file)

```
class TheClass { private int m_nvar; public int g_nvar = ...

var m_sVERSION = "1.2";
function TheClass() { this.m_nvar = 0; this.g_nvar = ...
```

p: parameter

- parameter passed to function.

```
public static void main(String[] p_aargs) { ...

function doSomething(p_nint, p_cstr) { ...
```

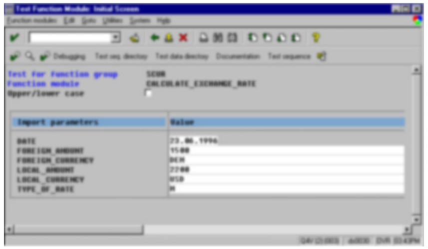
- in JavaScript, P_ is used for constructor parameters to avoid name conflicts with method parameters.

```
function TheClass(P_nparam, P_cparam) {
   this.p_nvar = P_nparam;
   this.method = function(p_nparam, p_cparam) { ...
```


Scope Prefix (cont'd) 7

i, o: input/output parameters

- used instead of "p" for languages where functions return values by changing parameters (usually database languages like ABAP or SQL).
- also used for recordset fields for stored procedures which take table parameters
- **i** = input parameter, which will be read and not be modified. Can pass a variable or a literal.
- **o** = output parameter, which will not be read but will be modified. Must pass a variable, cannot pass a literal.



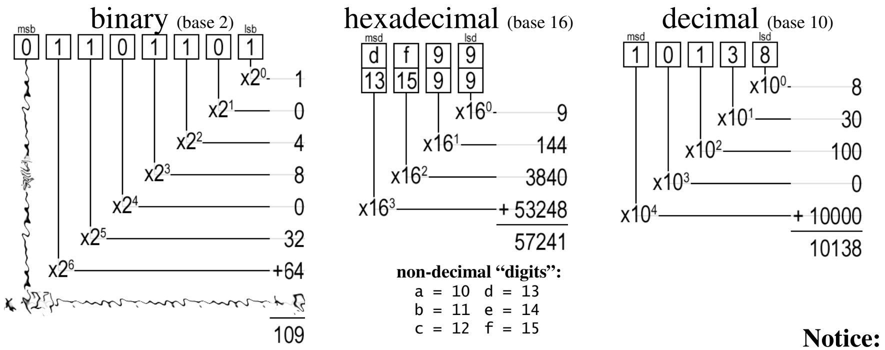
Why would we ever want to do this? 8

- Makes it easier to understand code in *implicitly typed* languages (esp. scripting languages) like JavaScript or VBScript
- Simplifies translating code between languages
- Simplifies collaboration on multilingual projects (and *reduces the amount of documentation required*!)

Binary Arithmetic Review

Positive Number Systems in Decimal

How are numbers in other systems decoded to decimal?

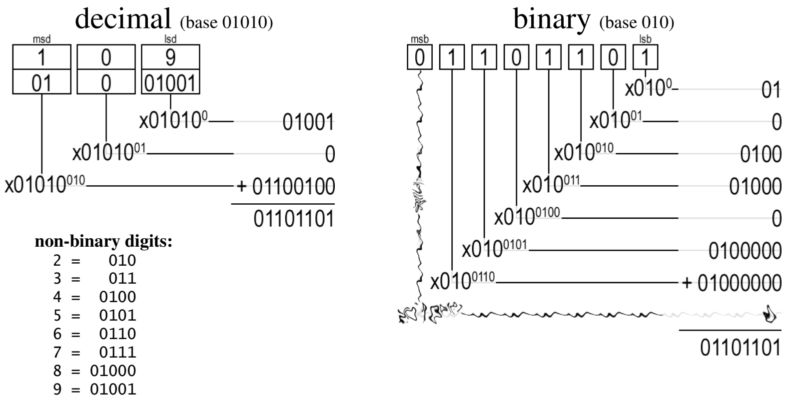


Notice:
binary 10 is decimal 2.
hexadecimal 10 is decimal 16.

Why is this important? (Look over at the decimal chart)

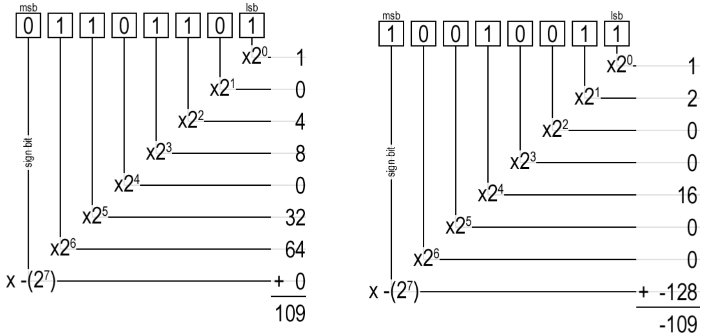
Positive Number Systems in Binary

How are numbers in encoded in binary?



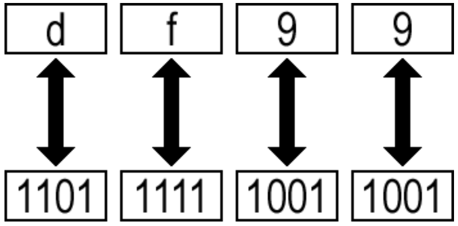
Negative Binary Integers

- For signed integers (all integers in Java), the leftmost bit is called the *sign bit*. It works exactly like all the other bits, except it is subtracted instead of added. This system is called *complement numbers*.
- To change the sign of a number, *flip* every bit, then add 1 to the total
- The first "computer" which worked with *complement numbers* was developed in 1640 by Blaise Pascal (1623-1662).



Binary and Hexadecimal

- Binary numbers and hexadecimal numbers have a special relationship: every 4 digits in a binary number correspond to 1 hexadecimal digit, and vice versa.



0 ↔ 0000	8 ↔ 1000
1 ↔ 0001	9 ↔ 1001
2 ↔ 0010	a ↔ 1010
3 ↔ 0011	b ↔ 1011
4 ↔ 0100	c ↔ 1100
5 ↔ 0101	d ↔ 1101
6 ↔ 0110	e ↔ 1110
7 ↔ 0111	f ↔ 1111

Bitwise Operators

		negation	conjunction	disjunction	exclusive disjunction	equivlance	implication
Basic syntax		NOT x	x AND y	x OR y	x XOR y	x EQV y	x IMP y
Java/C syntax		~x	x & y	x | y	x ^ y	~(x ^ y)	~x | y
x	y						
0	0	1	0	0	0	1	1
0	1		0	1	1	0	1
1	0	0	0	1	1	0	0
1	1		1	1	0	1	1
3 (0011)	9 (0101)	c (1100)	1 (0001)	7 (0111)	6 (0110)	9 (1001)	d (1101)

- Java/C/C++ don't have EQV/IMP operators, but you can get the same result by combining the other operators
- Java/C/C++ also implement assignment versions of each of these operators (~= , |= , &= , ^=). e.g. `x ~= y` is equivalent to `x = x ~ y`
- Common usage (almost every time you see these operators it means one of these):
  - AND
    - keep only these bits
    - test if these bits are set
    - remainder for division by a power of 2: `x % 2`^{y} `== x & (2`^{y} `- 1)`
  - OR
    - turn these bits on
  - XOR
    - "flip/toggle" these bits
    - turn these known-on bits off (use `x & ~y`, or "keep everything except) to turn off bits with unknown states)

Logical Operators (C/C++/JavaScript)

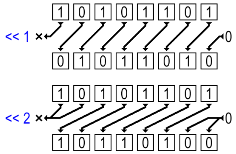
- C, which before the *C99* standard did not have an explicit boolean type, defines special "logical" operators which treat integers as entirely zero or non-zero. This is also true of JavaScript (which I've always thought was closer to C than it is to Java).
- These operators can be thought of in terms of the bitwise operators (though in practice the below definitions are less efficient)

logical and	`x && y`	`(x == 0 ? 0 : 1) & (y == 0 ? 0 : 1)`
logical or	`x || y`	`(x == 0 ? 0 : 1)	(y == 0 ? 0 : 1)`
logical xor	`x ^^ y`	`(x == 0 ? 0 : 1) ^ (y == 0 ? 0 : 1)`
logical not	`! y`	`~ (y == 0 ? 0 : 1)`

- Unlike their bitwise counterparts, there are no assignment operators in either Java or C for any of these operators.
  - *!* = does not have the same relation to *!* as ~= does to ~
- Java also defines these operators, but they can only be used with two boolean variables (likewise, in Java, the bitwise can only be used with two integers)

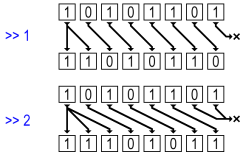
Binary Shifting

Applies to C family languages only (C/C++/Java/JavaScript). Each of the below also has an assignment operator equivalent.



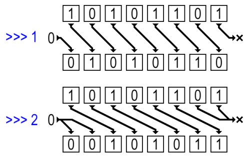
Left shift operator: <<

- right fills with 0
- x << y == x * 2^y



Signed right shift operator: >>

- left fills by copying the leftmost bit
- x >> y == x / 2^y for signed numbers



Unsigned right shift operator: >>>

- left fills with 0 (opposite of <<)
- x >>> y == x / 2^y for unsigned numbers

(even in Java which doesn't explicitly support unsigned numbers!)

Shifting Common Usage

- 2^n: `1 << n`
- Multiply $x * 2^n$: `x << n`
- Set bit n of x: `x |= 1 << n`
- Test bit n of x: `x & (1 << n) == 0`
- Get the n leftmost bits of x (also `x % 2`^{n}):
  `x & ((1 << n) - 1)`
- Binary decompression (get the n bits m bits from the left of x):
  `(x >> m) & ((1 << n) - 1)`

  e.g. if `x == r * 2`^{3} `+ c` where $0 \leq r,c, < 2^3$, then
  ```
  c == x & ((1 << 3) - 1)
  r == (x >> 3) & ((1 << 3) - 1)
  ```


finally... DP!

Iterative DP

- All the examples we've looked at so far have been **iterative** examples.
- Iterative algorithms are those where each step builds forward from the previous step.

Example of an iterative algorithm:

```
//Get the p_ndigit^th digit in the Fibonacci sequence in O(p_ndigit)
static int getFibonacciDigitIteratively(int p_ndigit) {
   int[] aprev = { 1, 1 };
   int nidx = 0;

   for (int i = 2; i < p_ndigit; i++)
        aprev[nidx ^= 1] = aprev[0] + aprev[1];
   return aprev[nidx];
} //end method
```

Psst: we're ignoring the non-DP, O(log(digit)) solution to this problem!

Backwards DP

- Another way to build a DP function is to start from the solution and work backwards, loading subproblems on the fly.
- However, a side effect of working backwards is that we very often end up resolving the same subproblem multiple times.

```
/**Get digit p_ndigit in the Fibonacci sequence in O(1.65^p_ndigit)*/
static int getFibonacciDigitRecursively(int p_ndigit) {
   if (p_ndigit < 3)
      return 1;
   return getFibonacciDigitRecursively(p_ndigit - 1) +
          getFibonacciDigitRecursively(p_ndigit - 2);
} //end method
```

- We can generally solve this problem through...

Memoization

- Memoization is a technique for speeding algorithms where we keep solving the same subproblem over again.
- This is done by recording every subproblem answered, and then reusing that answer if we ever need to solve that problem again.

```
/**Get digit p_ndigit in the Fibonacci sequence in O(p_ndigit)*/
static int getFibonacciDigitMemoized(int p_ndigit) {
   return getFibonacciDigitMemoized(p_ndigit, new int[p_ndigit]);
} //end method

   /**helper method for getFibonacciDigitMemoized(int)*/
   private static int getFibonacciDigitMemoized(int p_ndigit, int[] p_amemo) {
      if (p_ndigit < 3)
          return 1;
      if (p_amemo[p_ndigit] != 0)
          return p_amemo[p_ndigit];
      return p_amemo[p_ndigit] =
             getFibonacciDigitMemoized(p_ndigit - 1, p_amemo) +
             getFibonacciDigitMemoized(p_ndigit - 2, p_amemo);
   } //end method
```


Think About...

For each of the following pairs, which is better and why?

- `getFibonacciDigitRecursively(32)`
- `getFibonacciDigitMemoized(32)`

- `getFibonacciDigitRecursively(1234567890)`
- `getFibonacciDigitMemoized(1234567890)`

- `getFibonacciDigitRecursively(46)`
- `getFibonacciDigitRecursively(48)`

Memoization Implementation

- The key to memoization is to have a set where elements can be accessed randomly using a canonical representation of all the input parameters.
- For functions which use *n* non-negative/unsigned int, short, boolean/bool, char, or byte parameters (no longs... why?), this is most easily implemented by an *n*-dimensional memoization array.
- For functions which use boolean arrays (bool[<= 32] in C/C++, boolean[<= 64] in Java), the array can be **compressed** into a single int, and used as a single dimension in the memoization array.
- For functions using String/string parameters, or one which takes negative integer parameter values, a map must be used in place of an array.
  Note: this decreases the function's efficiency (e.g. replacing the array with a Map in our `getFibonacciDigitMemoized` function changes it from O(n) to O(n*log(n)).
- **IMPORTANT:** Functions using either Object or real number parameters (float/double/etc.) **cannot** be memoized.

Memoization Etymology (what, no 'R'?)

- Coined by Donald Michie in his 1968 paper "Memo Functions and Machine Learning" in *Nature*.
- *Memoization* is derived from the Latin word *memorandum*, meaning *what must be remembered*. In common parlance, a memorandum is abbreviated as memo, and thus "memoization" means "to turn (a function) into a memo".
- The word *memoization* is often confused with *memo**r**ization*, which, although a good description of the process, is not limited to this specific meaning.

Source: Wikipedia contributors. "Memoization." *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/w/index.php?title=Memoization&oldid=42692224

Abnormal Psych 101 (Coin Changer Problem) 25

- Raymond is a *high-functioning autistic savant* with *obsessive compulsive disorder* who can calculate complicated algorithms in his head.
- Right now he's working as a cashier, but every time he make changes he **needs** to tell the customer all the different ways he can do it, otherwise he's going to have an *episode*.

*"**15 cents**... a dime and a nickel... a dime and five pennies... three nickels... two nickels and five pennies... one nickel and ten pennies... fifteen pennies... **6 ways to make 15 cents**... gotta watch Wapner."*

- Unfortunately, even the Rain Man gets a bit slow when he has to change anything more than a few cents, and the manager is getting really close to firing him.
- Raymond's brother Charlie decides to help him count the possibilities faster by showing Raymond a DP algorithm...

Rain Man. Dir. Barry Levinson. Perfs. Dustin Hoffman, Tom Cruise. Film. United Artists, 1988.

Coin Changer Code 26

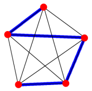
```
static int coinChanger(int p_nsum, int[] p_acoin) {
  return coinChanger(p_nsum, p_acoin.length - 1, p_acoin, new int[p_nsum + 1][p_acoin.length]);
} //end method

  private static int coinChanger(int p_nsum, int p_ncoin, int[] p_acoin, int[][] p_amemo) {
    int nret;

    //recursive base cases
    if (p_nsum == 0)
      return 1;
    else if (p_ncoin < 0)
      return 0;

    //memoization short circuit
    nret = p_amemo[p_nsum][p_ncoin];
    if (nret != 0)
      if (nret < 0)
        return 0;
      else
        return nret;

    //recurse
    for (int nsum = 0; nsum <= p_nsum; nsum += p_acoin[p_ncoin])
      nret += coinChanger(p_nsum - nsum, p_ncoin - 1, p_acoin, p_amemo);

    //memoization
    p_amemo[p_nsum][p_ncoin] = (nret != 0) ? nret : -1;

    return nret;
  } //end method
```

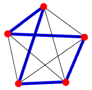
Questions:

- What's goes in the `p_acoin` array?
- What are the subproblems?
- What's special about memoizing 0? Why could we do better with C/C++?
- Why 2 methods? Do we need both?
- What's the efficiency of this algorithm?

Hamiltonian Paths and Cycles 27

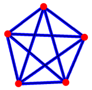
- A *Hamiltonian Path* is a walk within a directed graph which visits each **node** exactly once, and uses no (undirected) edge more than once.

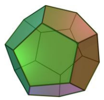




- A *Hamiltonian Cycle* is a *Hamiltonian Path* which ends at the same node where it started.

- These are a close cousins of a *Euler Paths* and *Cycles*, which visit each **edge** at exactly once, but may revisit nodes.





- *Hamiltonian Paths* and *Cycles* are named after mathematician William Rowan Hamilton (1805-1865) who invented a game called *the Icosian Game* which involves finding cycles around dodecahedrons (12-faced 3D shapes).

Hamiltonian Cycle Code

```
static boolean hamiltonianCycle(boolean[][] p_agraph) {
   int nnodes = p_agraph.length;
   int nlastNode = nnodes - 1;
   return hamiltonianCycle(p_agraph,
                           0,
                           (1 << nlastNode) - 2,
                           new boolean[nnodes][1 << nlastNode]);
} //end method

   private static boolean hamiltonianCycle(boolean[][] p_agraph,
                                           int p_nnode,
                                           int p_nseen,
                                           boolean[][] p_amemo) {
      if (p_nseen == 0)
         return p_agraph[p_nnode][0];
      if (p_amemo[p_nnode][p_nseen])
         return false;
      p_amemo[p_nnode][p_nseen] = true;
      for (int i = 0, nbit = 1; i < p_agraph.length; i++, nbit <<= 1)
         if (((p_nseen & nbit) != 0) && p_agraph[p_nnode][i])
            if (hamiltonianCycle(p_agraph, i, p_nseen ^ nbit, p_amemo))
               return true;
      return false;
   } //end method
```

Questions:

- What is the running time of this algorithm?
- What would the running time be without memoization?
- What's going on with the `p_nseen` parameter?
- What's the largest graph we can handle?
- Can this algorithm scale? Why or why not?
- How do we turn this into a Hamiltonian *Path* algorithm?

Traveling Salesman Problem

The *Traveling Salesman Problem* is the problem of finding the shortest possible Hamiltonian cycle:

- There is a salesman who has a list of cities he must visit.
- Every city on his list must be visited exactly once, and the salesman must end up back where he started to file expenses or whatever.
- There is a known travel cost (e.g. airfare) in moving from city to city, but not necessarily connected to geography.
- The total trip must be have the lowest cost possible.

Traveling Salesman History

- This problem was first introduced by Karl Menger (1902-1985) during a 1930 mathematical conference in Vienna.
- This problem first appeared in print in Menger's article "Das botenproblem," published in *Ergebnisse eines Mathematischen Kolloquiums* in 1932.
- Originally, the problem statement translated to "the messenger problem" and involved a postman trying to deliver letters using an optimal route, instead of a traveling salesman.
- *The Travelling Salesman* is also a silent film released in 1916 by Paramount Studios about a travelling salesman who finds romance and has a hilarious set of misadventures on his way home for Christmas.

Traveling Salesman Code

```
static int tsp(int[][] p_agraph) {
    int nnodes = p_agraph.length;
    int nlastNode = nnodes - 1;
    return tsp(p_agraph, 0, 0, (1 << nlastNode) - 2,
               new int[nnodes][1 << nlastNode]);
} //end method

    private static int tsp(int[][] p_agraph, int p_nnode,
                           int p_ndist, int p_nseen,
                           int[][] p_amemo) {
        int nret = Integer.MAX_VALUE;
        if (p_nseen == 0)
            return p_ndist + p_agraph[p_nnode][0];
        if (p_amemo[p_nnode][p_nseen] != 0)
            return p_amemo[p_nnode][p_nseen];
        for (int i = 0, nbit = 1; i < p_agraph.length; i++, nbit <<= 1)
            if (((p_nseen & nbit) != 0) && (p_agraph[p_nnode][i] != 0))
                nret = Math.min(nret, tsp(p_agraph, i, p_ndist + p_agraph[p_nnode][i],
                                          p_nseen ^ nbit, p_amemo));
        return p_amemo[p_nnode][p_nseen] = nret;
    } //end method
```

Questions:

- Look familiar? What's changed?
- What happens if the shortest path has a length of 0?
- Can this algorithm handle negative edge weights? Why or why not?
- What's the efficiency of this algorithm? How can we improve it?

Irish Problem

- 364 practice days.
- 1 St. Patrick's Day.

Questions?

References

Bellman, Richard. *Eye of the Hurricane: An Autobiography*. World Scientific Pub Co Inc, 1984. ISBN: 9971966018.

Frank and Igor. "Dynamic Programming." *University of British Columbia, CS490*. Available: Mike and Dustin.

Michie, Donald. "Memo Functions and Machine Learning." *Nature*, 218:19-22. Macmillan Publishers, 1968.

Rain Man. Dir. Barry Levinson. Perfs. Dustin Hoffman, Tom Cruise. Film. United Artists, 1988.

Tan, Gang. "Chapter 6: Dynamic Programming." *Boston College, CS383*. Fall 2005. Available: http://www.cs.bc.edu/~gtan/teaching/cs383f5/slides/cs383_06dynamic-programming.pdf

Wikipedia contributors. "Dynamic programming" *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Dynamic_programming

Wikipedia contributors. "Hamiltonian path" *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Hamiltonian_path

Wikipedia contributors. "Knapsack problem" *Wikipedia, The Free Encyclopedia*. February 2005. Available: http://en.wikipedia.org/wiki/Knapsack_problem

Wikipedia contributors. "Longest increasing subsequence problem" *Wikipedia, The Free Encyclopedia*. January 2005. Available: http://en.wikipedia.org/wiki/Longest_increasing_subsequence_problem

Wikipedia contributors. "Memoization" *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Memoization

Wikipedia contributors. "Optimal substructure" *Wikipedia, The Free Encyclopedia*. January 2005. Available: http://en.wikipedia.org/wiki/Optimal_substructure

Wikipedia contributors. "Overlapping subproblem" *Wikipedia, The Free Encyclopedia*. February 2005. Available: http://en.wikipedia.org/wiki/Overlapping_subproblem

Wikipedia contributors. "Traveling salesman problem" *Wikipedia, The Free Encyclopedia*. March 2005. Available: http://en.wikipedia.org/wiki/Traveling_salesman_problem