COS 226 Lecture 22: Dynamic Programming

DYNAMIC PROGRAMMING
* old term from operations research
* recursion revisited

Modern definition of dynamic programming:
"Bottom-up implementation of recursive programs with overlapping subproblems"
Top-down implementation also possible

Typically improves running time of an algorithm from exponential time to polynomial time (!)

Many applications

Fibonacci numbers

F(i) = F(i−1) + F(i−2)

0	0	10	55	20	6765
1	1	11	89	21	10946
2	1	12	144	22	17711
3	2	13	233	23	28657
4	3	14	377	24	46368
5	5	15	610	25	75025
6	8	16	987	26	121393
7	13	17	1597	27	196418
8	21	18	2584	28	317811
9	34	19	4181	29	514229

Numbers grow exponentially

F(i)/F(i−1) approaches 1.618... (golden ratio)

F(45) more than one billion

Recursive program to compute F(i)

```
int F(int i)
  {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
  }
```

Problem: program is slow!

Easy proof:

running time satistfies the recurrence

F(i) = F(i−1) + F(i−2)

Overlapping subproblems

Fibonacci computation is slow because subproblems in recursive computation overlap

To compute F(5)

```
    compute F(4)
        compute F(3)
            compute F(2)
                compute F(1) = 1
                compute F(0) = 0
                return 1
            compute F(1) = 1
            return 2
        compute F(2)
            compute F(1) = 1
            compute F(0) = 0
            return 1
        return 3
    compute F(3)
        compute F(2)
            compute F(1) = 1
            compute F(0) = 0
            return 1
        compute F(1) = 1
        return 2
    return 5
```

Fix by remembering solutions already computed

Avoid recomputing known results

Maintain array (indexed by parameter value)
* zero if recursive routine not yet called
  for that value yet
* result to be returned otherwise

First call on procedure for a given value:
  compute as before, save value
Second and subsequent calls:
  use value from first call

Ex: Fibonacci numbers less than 10^9

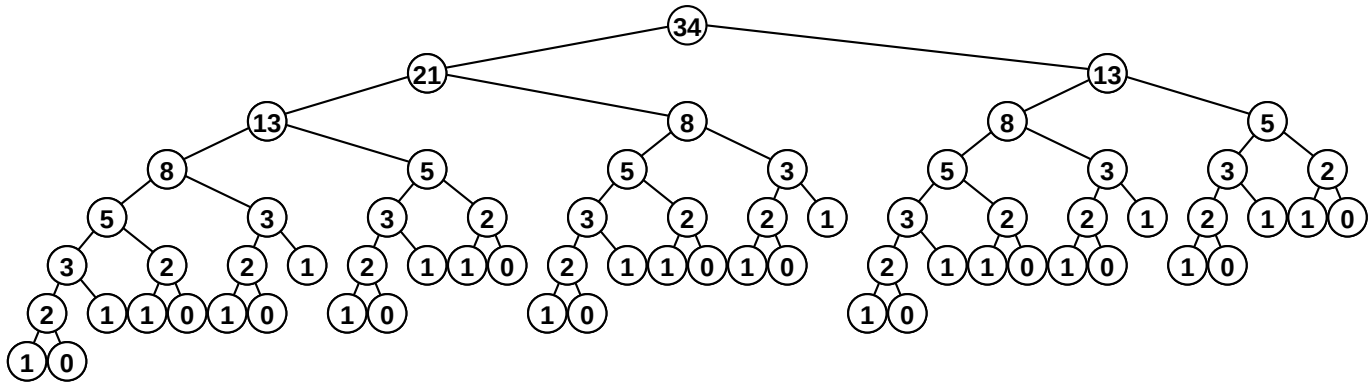
```
for (i = 0; i < 45; i++) known[i] = 0;
int F(int i)
   { int t;
     if (known[i]) return known[i];
     if (i == 0) t = 0;
     if (i == 1) t = 1;
     if (i > 1) t = F(i-1) + F(i-2);
     known[i] = t;
     return t;
   }
```

LINEAR running time

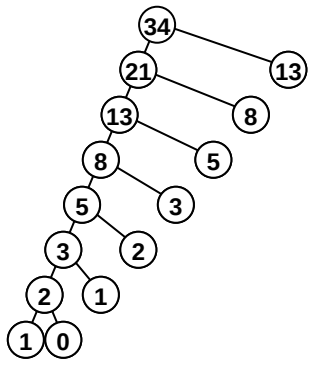
LOGARITHMIC amount of extra space

Recursive call structures

Straight recursive algorithm



Remembering known results



To compute F(5)

compute F(4)

compute F(3)

compute F(2)

compute F(1) = 1

compute F(0) = 0

return 1

use known F(1) = 1

return 2

use known F(2) = 1

return 3

use known F(3) = 2

return 5

Bottom-up approach

DYNAMIC PROGRAMMING

* Tabulate answers to subproblems
* Build table in increasing order of problem size
* Use early entries to compute later ones (no recomputation needed)

Ex: Fibonacci numbers

```
F[0] = 0; F[1] = 1;
for (j = 2; j < i; j++)
   F[j] = F[j-1] + F[j-2];
```

"Dynamic programming solution to the problem of computing Fibonacci numbers"

Perhaps the simplest nontrivial example of dynamic programming

Reduces running time from exponential to linear

Similar savings are available for a wide variety of important practical problems

Applicable to *any* pure recursive algorithm with

* no effects on global variables
* enough space to save solutions to subproblems

No overlap: bottom-up divide-and-conquer

Knapsack problem

Given N types of food items, where each item
* takes a certain amount of space
* provides a certain amount of nutrients

KNAPSACK PROBLEM
Fill a knapsack of capacity M with items
such that total amount of nutrients is maximized

Ex:

	size	**value**
apple	**3**	**4**
banana	**4**	**5**
chocolate	**7**	**10**
donut	**8**	**11**
egg	**9**	**13**

In a knapsack of capacity 17
two apples + banana + chocolate gives 23
donut + egg gives 24 (maximum)

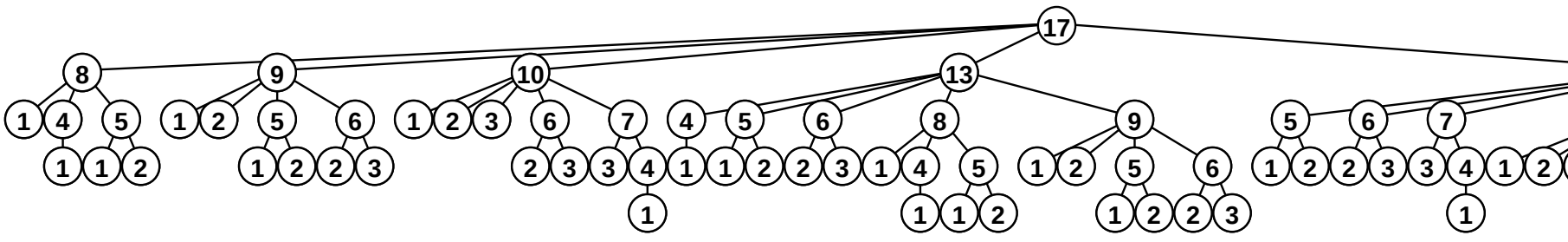
Applications
* shipping cargos
* job scheduling
...

Recursive solution of knapsack problem

For each item type, compute the value obtained
if the last item added is of that type
Pick the maximum
(also could return ID of last item added)

```
int knap(int capacity)
   { int i, j, max, t;
      for (i = 0, max = 0; i < N; i++)
         if ((j = capacity-size[i]) >= 0)
            if ((t = knap(j) + val[i]) > max)
               max = t;
      return max;
   }
```

DON'T USE THIS PROGRAM!
overlapping subproblems
excessive recomputation
exponential time



Use top-down or bottom-up dynamic programming to
avoid recomputation for overlapping subproblems

"Known answers" approach

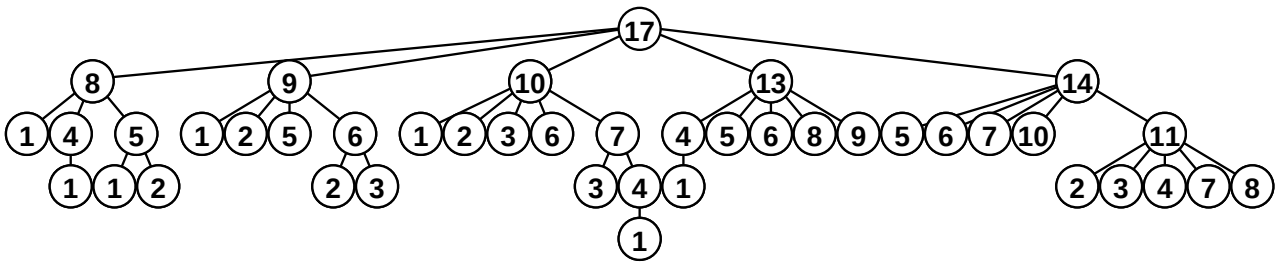
To avoid recomputation in knapsack algorithm

remember, for every knapsack capacity

* total cost for best way to fill knapsack
* last item used in filling it

```
int knap(int C)
   { int i, j, max, maxj, t;
     if (costKnown[C]) return costKnown[C];
     for (i = 0, max = 0, j = 0; i < N; i++)
        if ((j = cap-size[i]) >= 0)
           if ((t = knap(j) + val[i]) > max)
                 { max = t; maxj = j; }
     costKnown[C] = max;
     bestKnown[C] = maxj;
     return max;
   }
```

Use bestKnown to recover set of items



Maintain elegance of recursive formulation

without sacrificing efficiency

Bottom-up approach to knapsack problem

Generalized version of Fibonacci numbers program

Use previous solutions to compute next one

```
for (i = 0; i < C; i++) cost[i] = 0;
for (i = 0; i < C; i++)
   for (j = 1; j <= N; j++)
      if (i >= size[j])
         if (cost[i] < cost[i-size[j]]+val[j])
            {
               cost[i] = cost[i-size[j]]+val[j];
               best[i] = j;
            }
```

3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
4	5	5	5	10	11	13	14	15						
a	b	b	b	c	d	e	a	a						

Total time proportional to N*C

Another approach (in book)

N passes through the cost array

[just interchange "for" loops]

Dynamic programming summary

Many problem solutions are elegantly
described as recursive programs

If subproblems overlap, programs may
require exponential time

KNOWN ANSWERS (top-down) approach
Mechanically modify recursive routine to
* test if called yet for the given parameter value
* if so, return previously computed value
* if not, compute value and save it

DYNAMIC PROGRAMMING (bottom-up) approach
* Tabulate answers to subproblems
* Build table in order of problem size
* Use early entries to compute later ones

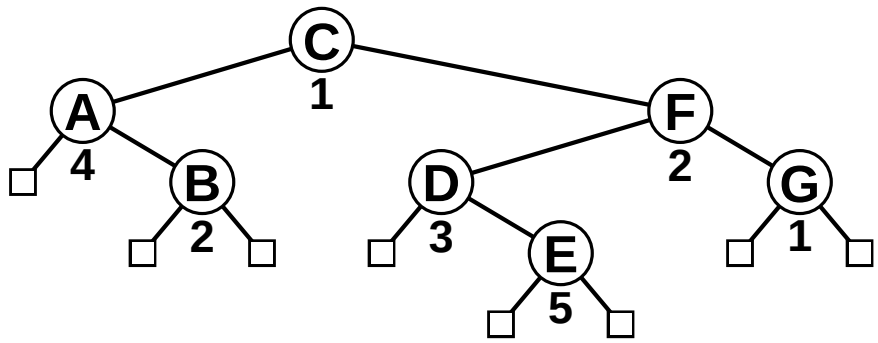
Further improvements often possible
* do computations in particular order
* use specialized knowledge to eliminate some

Dynamic programming with two variables

OPTIMAL BST problem

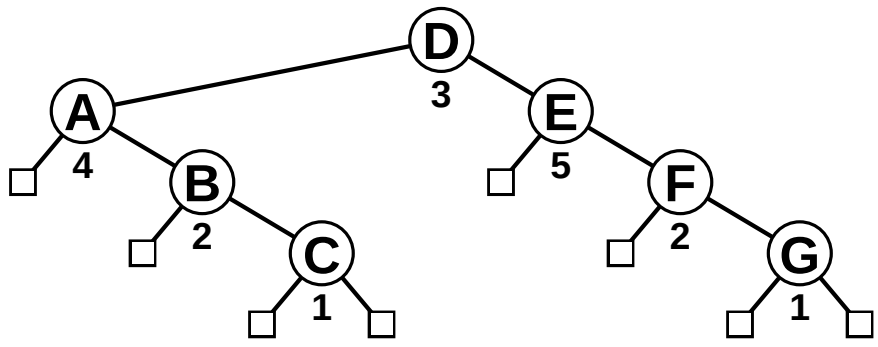
Given access frequencies, minimize average search cost (weighted path length) in BST

Ex:



weighted path length

1*1 + 2*(4+2) + 3*(2+3+1) + 4*5 = 51



weighted path length

1*3 + 2*(4+5) + 3*(2+2) + 4(1+1) = 41

Top-down solution to optimal BST problem

Recursive program with two arguments

best subtree for ith through jth keys

```
int bestBST(int i, int j)
   { int k, min, t;
      for (k = i, min = 0; k <= j; k++)
         if (t = bestBST(i, k-1) + bestBST(k+1, j) < min)
      return min;
   }
```

DON'T USE THIS PROGRAM

overlapping subproblems, exponential time

Known values (top-down) approach

* precisely the same as for one-variable
* eliminate recomputation for overlapping subproblems by saving and recalling known values in 2D array

Bottom-up solution to optimal BST problem

Two-dimensional array
* diagonal has subproblems of size 1
* one up from diagonal has subproblems of size 2
* two up from diagonal has subproblems of size 3
    ...

```
for (i = 1; i <= N; i++)
  for (j = i+1; j <= N+1; j++) cost[i][j] = MAX;
for (i = 1; i <= N; i++) cost[i][i] = f[i];
for (i = 1; i <= N+1; i++) cost[i][i-1] = 0;
```

```
for (j = 1; j <= N-1; j++)
  for (i = 1; i <= N-j; i++)
  {
    for (k = i; k <= i+j; k++)
      {
        t = cost[i][k-1]+cost[k+1][i+j];
        if (t < cost[i][i+j])
          { cost[i][i+j] = t; best[i][i+j] = k; }
      }
    for (k = i; k <= i+j; cost[i][i+j] += f[k++]) ;
  }
```

Optimal BST example

"cost" array

```
.  4  8 11 19 31 37 41
.     2  4 10 20 25 28
.        1  5 14 18 21
.           3 11 15 18
.              5  9 12
.                 2  4
.                    1
```

"best" array

```
.     A  A  B  D  D  D
.        B  D  D  E  E
.           D  E  E  E
.              E  E  E
.                 E  E
.                    F
```

Optimal recursive decomposition problems

Trees model many other computation problems

"Optimal divide-and-conquer"

MATRIX CHAIN PRODUCT problem

Given a list of matrices, find optimal prenthiesization for pairwise multiplication

TRIANGULATION

Divide a convex polygon into triangles using minimal lines of total length

Correspondence between trees and

* parenthesizations
* triangulated polygons
* ...

String processing DP example

LONGEST COMMON SUBSEQUENCE problem
Find the longest common subsequence in two strings
Applications: molecular biology, text editors, etc.

subsequence: subset of string, in order
    2^N different subsequences in string of length N

Ex: LCS in ABCBDAB and BDCAGA is BCBA

Recursive program is a description of a
    recursively defined solution

```
int LCSlen(int i, int j)
  {
    if ((i == -1) || (j == -1)) return 0;
    if (a[i] == b[j]) return 1+LCSlen(i-1,j-1);
    t = LCSlen(i, j-1); u = LCSlen(i-1, j);
    if (t > u) return t; else return u;
  }
```

As usual, DON'T USE THIS PROGRAM
It takes exponential time as is, but it is easy to
    * change to use known values OR
    * compute values bottom-up

"Far too many" subproblems

Dynamic programming cannot be used unless space
is available to hold answers to subproblems

Three important cases illustrated by different
versions of knapsack problem

EXAMPLE 1:

Sizes are reals, not integers

	size	**value**
apple	**30000**	**.0004**
banana	**.0001**	**12345**
chocolate	**.0002**	**21244**
donut	**.1234**	**.2125**
egg	**.0012**	**123**

Table size dicatated by precision, relative sizes
typically much more precision than space

EXAMPLE 2:

Allow at most one of each type of item in knapsack

Different subproblem for each SUBSET of item types

2^N subsets

EXPONENTIAL space requirements

EXAMPLE 3:

Bananas and eggs can’t go on the bottom, etc.

Different subproblem for each PERMUTATION of item types

N! permutations

superEXPONENTIAL space requirements

N! grows like 2^(N lgN)