Dynamic Programming (15.1 - 15.5

Ex.

Consider the problem of computing the Binomial Coefficient $\binom{n}{k}$ using Pascal's identity

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k=0 \text{ or } k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0<k<n \\ 0 & \text{otherwise} \end{cases}$$

The obvious recursive algorithm is

```
BinCoef(n, k)
1.) if k = 0 or 1
2.)     return 1
3.) else
4.)     return BinCoef(n-1, k-1) + BinCoef(n-1, k)
```

Observe that at the bottom level BinCoef always returns 1, so ultimately it just adds a lot of 1's. Thus BinCoef runs in time

$$\Omega\left(\binom{n}{k}\right).$$

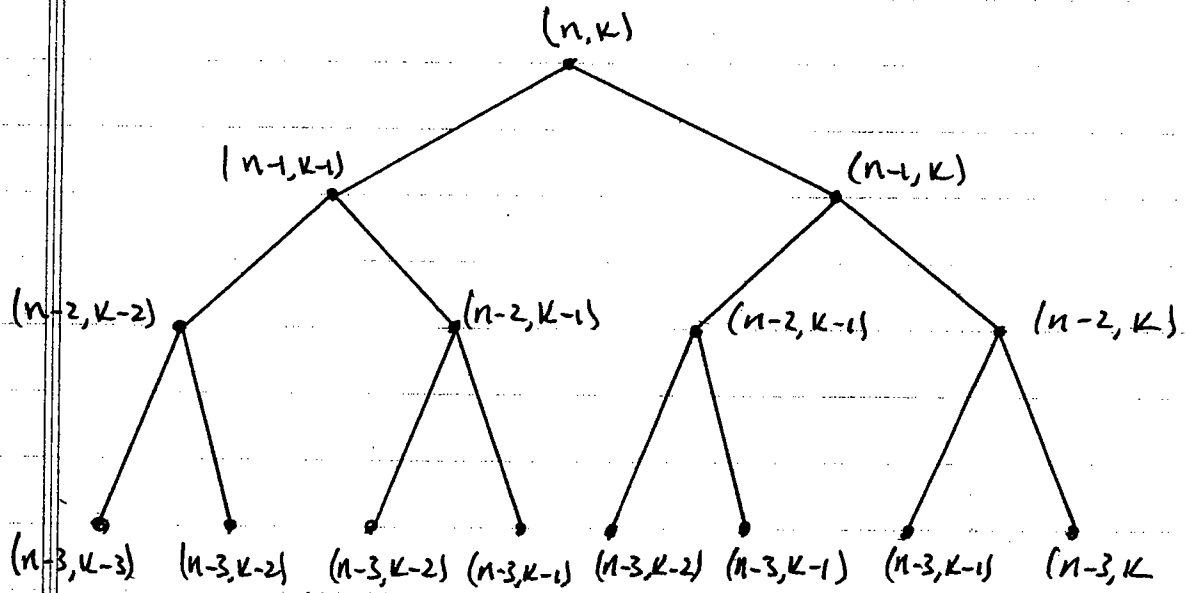
EXERCISE

SHOW THAT IF $n = 2k$, THEN

$$\binom{n}{k} = \Theta\left(\frac{2^n}{\sqrt{n}}\right) = \Theta\left(\frac{4^k}{\sqrt{k}}\right)$$

HINT: USE STIRLING'S FORMULA.

THE RECURSION TREE SHOWS THE PROBLEM



OBSERVE THAT MANY OF THE SAME VALUES ARE COMPUTED MULTIPLE TIMES.

A MORE EFFICIENT APPROACH IS TO MAINTAIN A TABLE OF INTERMEDIATE RESULTS.

	0	1	2	3	. . . .	K-1	K
0	1	0	0	0	. .	0	0
1	1	1	0	0	. .	0	0
2	1	2	1	0	. .	0	0
3	1	3	3	1	. .	0	0
⋮						⋮	⋮
n-1	.	.	.		. .	$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n	.	.	.		. .		$\binom{n}{k}$

IN FACT IT'S NOT NECESSARY TO STORE THE WHOLE TABLE, JUST A SINGLE ROW AT A TIME.

DynBinCoef(n, k)

```
1.) C[0] ← 1
2.) for i ← 1 TO k
3.)     C[i] ← 0
4.) for j ← 1 TO n
5.)     for i ← k TO 1
6.)         C[i] ← C[i-1] + C[i]
7.) return C[k].
```

NOTE: THERE IS STILL SOME EFFICIENCY TO BE GAINED BY DELETING ALL ZEROS, NOT CALCULATING ALL OF BOTTOM ROW, ETC. THIS IS LEFT AS AN EXERCISE.

THE SAME PROBLEM ARISES FREQUENTLY IN THE DIVIDE & CONQUER APPROACH. THE OBVIOUS AND NATURAL WAY OF DIVIDING A PROBLEM INTO SUBINSTANCES LEADS TO OVERLAPING (OR IDENTICAL) SUBINSTANCES WHICH ARE SOLVED MULTIPLE TIMES, LEADING TO AN INEFFICIENT ALGORITHM.

IN DYNAMIC PROGRAMING WE ARRANGE TO SOLVE EACH SUBINSTANCE ONLY ONCE, SAVING THE RESULT FOR LATER USE.

WE TYPICALLY STORE THIS INFORMATION AS A TABLE (e.g. 2-dim ARRAY) OF KNOWN RESULTS.

THE WORD "PROGRAMING" IS USED HERE IN AN ARCHAIC SENSE. "PROGRAM" IS SYNONYMOUS WITH "TABLE", AS IN LINEAR PROGRAMMING.

DYNAMIC PROGRAMMING IS A TECHNIQUE FOR SOLVING OPTIMIZATION PROBLEMS. WE SEEK AN OPTIMAL SOLUTION FROM AMONG A SET OF FEASIBLE SOLUTIONS.

Coin Changing Problem

Suppose we have coins in n denominations $\{d_1, d_2, \ldots, d_n\}$, where each $d_i \geq 1$ is an integer. We wish to pay an amount N using the fewest number of coins possible.

Assumption: There is an unlimited supply of coins in each denomination.

There are two questions:

- What is the least number of coins needed to pay N units.
- Exactly which coins (i.e. which denominations) are to be disbursed.

To answer the first question we create a table $C[1 \ldots n; 0 \ldots N]$, where

$C[i,j]$ = minimum number of coins necessary to pay amount j using coins in denominations $\{d_1, \ldots, d_i\}$, $1 \leq i \leq n$, $0 \leq j \leq N$.

Thus we seek $C[n, N]$.

FIRST OBSERVE THAT $C[i,0] = 0$ FOR ALL $1 \le i \le n$.

NEXT, NOTICE THAT TO PAY THE AMOUNT j USING DENOMINATIONS $\{d_1, \ldots, d_i\}$ WE HAVE IN GENERAL TWO CHOICES

(1) USE NO COINS VALUES d_i (EVEN THOUGH THIS IS PERMITTED). WE CAN DO THIS USING $C[i-1, j]$ COINS.

(2) USE AT LEAST 1 COIN OF VALUE d_i. AFTER HANDING OVER ONE SUCH COIN, THERE ARE $j - d_i$ UNITS LEFT TO PAY FROM DENOMINATIONS $\{d_1, \ldots, d_i\}$. WE CAN DO THIS USING $1 + C[i, j-d_i]$ COINS.

$C[i,j]$ SHOULD BE WHICHEVER ALTERNATIVE USES THE FEWEST COINS. THUS

$$C[i,j] = \min\left(C[i-1, j],\ 1 + C[i, j-d_i]\right)$$

NOTE IF $i = 1$ OR $j < d_i$ THEN ONE OF THE VALUES ON RIGHT FALLS OUTSIDE THE TABLE.

IT IS CONVENIENT TO THINK OF SUCH VALUES AS BEING $+\infty$.

IF BOTH $i = 1$ AND $i < d_i$ THEN WE SET

$$C[1, j] = +\infty$$

INDICATING THAT IT IS IMPOSSIBLE TO PAY AMOUNT j USING ONLY COINS OF TYPE 1.

EX. $n = 4$, $N = 8$

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 3$	0	1	2	1	2	3	2	3	4
$d_3 = 5$	0	1	2	1	2	1	2	3	2
$d_4 = 6$	0	1	2	1	2	1	1	2	2

NOTE THAT IF WE HAVE AN UNLIMITED SUPPLY OF COINS OF VALUE 1 (e.g. $d_1 = 1$) THEN IT IS POSSIBLE TO DISBURSE ANY AMOUNT. OTHERWISE IT MAY BE IMPOSSIBLE TO PAY CERTAIN AMOUNTS. WE INDICATE THIS WITH $C[i, j] = \infty$.

EX. $n = 3$, $N = 8$

	0	1	2	3	4	5	6	7	8
$d_1 = 2$	0	∞	1	∞	2	∞	3	∞	4
$d_2 = 4$	0	∞	1	∞	1	∞	2	∞	2
$d_3 = 5$	0	∞	1	∞	1	1	2	2	2

THE FOLLOWING ALGORITHM TAKES INPUTS $d[1..n]$, N, AND USES A LOCAL ARRAY $C[1..n; 0..N]$

```
CoinChange(d, N)
1.) n ← length[d]
2.) for i ← 1 TO n
3.)     C[i,0] ← 0
4.) for i ← 1 TO n
5.)     for j ← 1 TO N
6.)         if i=1 AND j<d[1]
7.)             C[1,j] ← ∞
8.)         else if i = 1
9.)             C[1,j] ← 1+C[1, j-d[i]]
10.)        else if j<d[i]
11.)            C[i,j] ← C[i-1, j]
12.)        else
13.)            C[i,j] ← min(C[i-1,j], 1+C[i, j-d[i]])
14.) return C[n,N].
```

THE ALGORITHM CAN BE EASILY ALTERED TO RETURN THE ENTIRE TABLE $C[1..n, 0..N]$ OF INTERMEDIATE RESULTS.

THE RUN TIME IS OBVIOUSLY $\Theta(nN)$ SINCE EACH OF $n \cdot (N+1)$ TABLE ENTRIES MUST BE FILLED.