# CS 25 - Algorithms

10/9/95

[Las]t time (chap 9)
- Sorting in linear time

Today (chap 16)
- Dynamic Programming

Announcements
- Homeworks — start early / get clarification / work in groups
- X-hour — late HW2 due
- midterm ① 10/20 - 10/27 ② 10/27 - 11/3

① 36.8% ② 63.2%

## Paradigms

Elements of the D&C Paradigm

① Problem can be split into independent subproblems of the same type.

② Solutions to the subproblems can be combined to obtain a solution to the original.

Structure of a D&C Algorithm

① Divide into subproblems

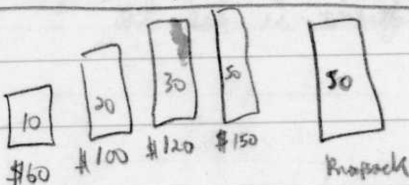
② Recursively solve subproblems

③ Combine solutions

Dynamic Programming : Typically applied to optimization problems

- many "solutions" — find solution with the optimal value

0-1 Knapsack example ?




(over)

## Elements of the DP Paradigm

① Problems exhibits *optimal substructure*.

- optimal solution to problem is combination of optimal solutions to subproblems

② Problem exhibits *overlapping subproblems*.

## DP Methodology (I)

① Characterize structure of an optimal solution

② Recursively define value of optimal solution

③ Compute value of optimal solution in bottom-up fashion

④ Construct optimal solution from computed information.

Consider 1, 10, 25 (no nickel)
make change for 30 ¢

Smallest (?)
1, 3, 4
make change for 6 ¢



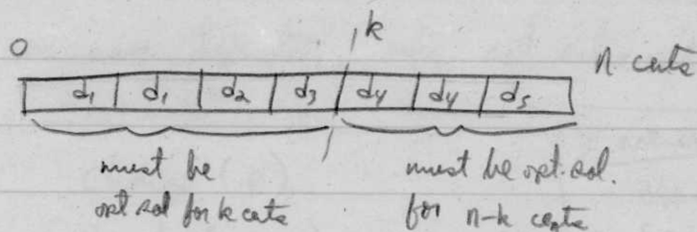
## Coin changing

Make change for n cents using minimum number of coins of denominations $d_1, d_2, \ldots, d_k$ where $d_1 < d_2 < \cdots < d_k$ and $d_1 = 1$.

### Methodology

① Characterize structure of an optimal solution (optimal substructure)

opt. sol.




* – problem exhibits opt. sub.

② Recursively define value of opt solution

- Let $C[p]$ be min. # coins needed to make change for $p$ cents
- opt. sol. might use (last or first) a $d_1$ or $d_2$ or $d_3$ or $d_4$ ...
- By definition $C$ and opt. sub. of problem, value of opt sol. using a

$d_1$ is $C[p-d_1]+1$

$d_2$ is $C[p-d_2]+1$

$d_3$ is $C[p-d_3]+1$

⋮

$d_k$ is $C[p-d_k]+1$ (assuming $d_k \le p$)

- opt sol is therefore $\min_{i:\, d_i \le p} \{ C[p-d_i]+1 \}$

- Recursive definition

$$C[p] = \begin{cases} 0 & \text{if } p=0 \\ 1 & \text{if } p=d_1 \text{ or } d_2 \text{ or} \ldots \text{ or } d_k \\ \min_{i: d_i \le p} \{ C[p-d_i] + 1 \} & \text{otherwise} \end{cases}$$

← don't need this base case...

Digression: Recursive approach (bad)

- We have a recursive definition, why not solve with a recursive alg.?

- Consider: (3-coin scenario)

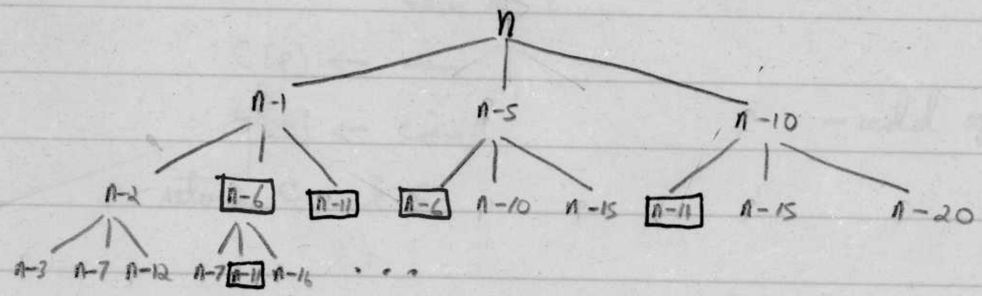
```
change (p)
    if (p<0)
        return ∞
    else if (p==0)
        return 0
    else
        return min ( change(p-10)+1, change(p-5)+1, change(p-1)+1 )
```

memoized

```
C[0] ← 0
For i=1 to n
   C[i] ← ∞
return change(n)

change (p) {
   if C[p] < ∞
      return C[p]
   else
      C[p] ← min( change ... )
      return C[p]
}
```

- Running time? Consider recursion tree of calls to change() (each call is constant work):




- Tree is full for $n/10$ levels, at least $\Omega(3^{n/10})$ work!!!

(actually much worse...)

☆ - Note, however, tons of repeated subproblems

• Idea: Solve each subproblem only once.

end digression

③ Compute value of optimal solution in bottom-up fashion

Idea: Solve problem for all coin changing problems up to $n$;
store results of "smaller" problems for use in solving "larger" problems.

```
change (d[], k, n)              // d array contains denomination values
  C[0] ← 0                      // k is # denominations
  for p ← 1 to n
    min ← ∞
    for i ← 1 to k
    * if p ≥ d[i] then
        if C[p-d[i]] + 1 < min then
          min ← C[p-d[i]] + 1
          coin ← i
    C[p] ← min
    S[p] ← coin
  return C and S
```

– add green lines later

Running time: $\Theta(nk)$

Space : $\Theta(n)$

(4) Construct optimal solution from computed information

- need "green" lines from before to calculate solution array

$S[p]$ – "first" coin in opt. solution to change problem of size $p$.

$$S[p] = \arg\min_{i:\, d_i \le p} \{ C[p - d_i] + 1 \}$$

```
Make-change ( S[], d[], n)
    while (n > 0)
        print S[n]                  // coin # (nickel, dime, penny)
        n ← n - d[S[n]]
```

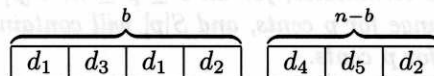
- Additional $\Theta(n)$ time

Total: $\Theta(nk)$ time

$\Theta(n)$ space

# Dynamic Programming Solution to the Coin Changing Problem

**(1) Characterize the Structure of an Optimal Solution.** The Coin Changing problem exhibits optimal substructure in the following manner. Consider any optimal solution to making change for $n$ cents using coins of denominations $d_1, d_2, \ldots, d_k$. Now consider breaking that solution into two different pieces along any coin boundary. Suppose that the "left-half" of the solution amounts to $b$ cents and the "right-half" of the solution amounts to $n - b$ cents, as shown below.



**Claim 1** *The left-half of the solution must be an optimal way to make change for $b$ cents using coins of denominations $d_1, d_2, \ldots, d_k$, and the right-half of the solution must be an optimal way to make change for $n - b$ cents using coins of denominations $d_1, d_2, \ldots, d_k$.*

**Proof:** By contradiction, suppose that there was a better solution to making change for $b$ cents than the "left-half" of the optimal solution shown. Then the left-half of the optimal solution could be replaced with this better solution, yielding a valid solution to making change for $n$ cents with fewer coins than the solution being considered. But this contradicts the supposed optimality of the given solution, $\rightarrow\leftarrow$. An identical argument applies to the "right-half" of the solution. □

Thus, the optimal solution to the coin changing problem is composed of optimal solutions to smaller subproblems.

**(2) Recursively Define the Value of the Optimal Solution.** First, we define in English the quantity we shall later define recursively. Let $C[p]$ be the minimum number of coins of denominations $d_1, d_2, \ldots, d_k$ needed to make change for $p$ cents. In the optimal solution to making change for $p$ cents, there must exist some first coin $d_i$, where $d_i \leq p$. Furthermore, the remaining coins in the optimal solution must themselves be the optimal solution to making change for $p - d_i$ cents, since coin changing exhibits optimal substructure as proven above. Thus, if $d_i$ is the first coin in the optimal solution to making change for $p$ cents, then $C[p] = 1 + C[p - d_i]$; i.e., one $d_i$ coin plus $C[p - d_i]$ coins to optimally make change for $p - d_i$ cents. We don't know which coin $d_i$ is the first coin in the optimal solution to making change for $p$ cents; however, we may check all $k$ such possibilities (subject to the constraint that $d_i \leq p$), and the value of the optimal solution must correspond to the minimum value of $1 + C[p - d_i]$, by definition. Furthermore, when making change for 0 cents, the value of the optimal solution is clearly 0 coins. We thus have the following recurrence.

**Claim 2** $C[p] = \begin{cases} 0 & \textit{if } p = 0 \\ \min_{i:d_i \leq p}\{1 + C[p - d_i]\} & \textit{if } p > 0 \end{cases}$

**Proof:** The correctness of this recursive definition is embodied in the paragraph which precedes it. □

**(3) Compute the Value of the Optimal Solution Bottom-up.** Consider the following piece of pseudocode, where $d$ is the array of denomination values, $k$ is the number of denominations, and $n$ is the amount for which change is to be made.

```
CHANGE(d, k, n)
1  C[0] ← 0
2  for p ← 1 to n
3      min ← ∞
4      for i ← 1 to k
5          if d[i] ≤ p then
6              if 1 + C[p − d[i]] < min then
7                  min ← 1 + C[p − d[i]]
8                  coin ← i
9      C[p] ← min
10     S[p] ← coin
11 return C and S
```

**Claim 3** *When the above procedure terminates, for all $0 \leq p \leq n$, $C[p]$ will contain the correct minimum number of coins needed to make change for p cents, and $S[p]$ will contain (the index of) the first coin in an optimal solution to making change for p cents.*

**Proof:** The correctness of the above procedure is based on the fact that it correctly implements the recursive definition given above. The base case is properly handled in Line 1, and the recursive case is properly handled in Lines 2 to 10, as shown below. Note that since the loop defined in Line 2 goes from 1 to $n$ and since $d_i \geq 1$ for all $i$, no array element $C[\cdot]$ is accessed in either Line 6 or 7 before it has been computed. Lines 3 to 7 correctly compute $\min_{i:d_i \leq p}\{1 + C[p - d_i]\}$, and $C[p]$ is set to this value in Line 9. Similarly, Lines 3 to 8 correctly compute $\arg\min_{i:d_i \leq p}\{1 + C[p - d_i]\}$, and $S[p]$ is set to this value in Line 10. □

**(4) Construct the Optimal Solution from the Computed Information.** Consider the following piece of pseudocode, where $S$ is the array computed above, $d$ is the array of denomination values, and $n$ is the amount for which change is to be made.

```
MAKE-CHANGE(S, d, n)
1  while n > 0
2      Print S[n]
3      n ← n − d[S[n]]
```

**Claim 4** *The above procedure correctly outputs an optimal set of coins for making change for n cents.*

**Proof:** By Claim 3, $S[n]$ will contain (the index of) the first coin in an optimal solution to making change for $n$ cents, and this coin in printed in Line 2 during the first pass through the **while** loop. Since our problem exhibits optimal substructure by Claim 1, it must be the case that the solution to the remaining $n - d_{S[n]}$ cents be optimal as well. By setting $n \leftarrow n - d[S[n]]$ in Line 3, the first coin in such a solution will be printed in the next pass through the **while** loop, and so on. (Properly, one would prove that the sequence of coins output by this procedure is correct by induction, but the above suffices as far as I'm concerned.) □

**(5) Running Time and Space Requirements.** The CHANGE procedure runs in $\Theta(nk)$ due to the nested loops (Lines 2 and 4), and it uses $\Theta(n)$ additional space in the form of the $C[\cdot]$ and $S[\cdot]$ arrays. The MAKE-CHANGE procedure runs in time $O(n)$ since the parameter $n$ is reduced by at least 1 (the minimum coin denomination value) in each pass through the **while** loop. It uses no additional space beyond the inputs given. Thus, the total running time is $\Theta(nk)$ and the total space requirement is $\Theta(n)$.

# CS 25 – Algorithms

10/11/95

**Last time** (chap 16)
- Dynamic Programming

**Today** (chap 16)
- Dynamic Programming

**Announcements**
- Midterm
  - 10/20 – 10/27 36.8 %
  - 10/27 – 11/3 63.2 %
- HW3 hints

---

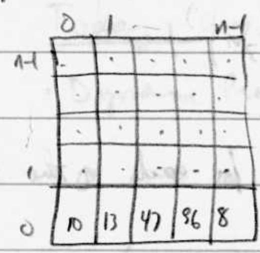
**Elements of DP**
- optimal substructure
- overlapping subproblems

## DP Methodology (I)

✓ ① Characterize structure of an optimal solution
✓ ② Recursively define value of optimal solution
③ Compute value of optimal solution in bottom-up fashion
④ Construct optimal solution from computed information

## Chess board problem

- n×n chessboard :



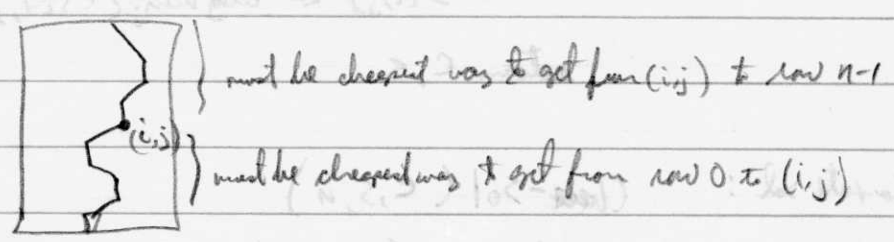

P

- numbers in squares are penalties.

- Goal: start anywhere in row 0, get to anywhere in row n-1, with least total penalty, where only moves can make are ↖↑↗

- Let P[i,j] be penalty matrix

① Characterize structure of opt. sol.



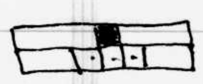

Let C[i,j] be cheapest way to get from row 0 to square (i,j)

② Rec def value of opt sol.

(i,j)
↖↑↗

$$C[i,j] = \begin{cases} P[i,j] & \text{if } i=0 \\ P[i,j] + \min\{ C[i-1, j-1], C[i-1, j], C[i-1, j+1] \} \end{cases}$$

(all j's are mod n)



Note: recursive solution is

$$T(n) = 3T(n-1) + \Theta(1)$$
$$= \Omega(3^n) \ !$$

In fact, must do this for each of the $n$ squares in row $n-1$

$$\Rightarrow \quad \Omega(n3^n)$$

③ Compute cost bottom up:

```
Chess( P[], n )
    for j ← 0 to n-1
        C[0,j] ← P[0,j]
    for i ← 1 to n-1
        for j ← 0 to n-1                                    ↙ all j's (mod n)
            C[i,j] ← P[i,j] + min{ C[i-1,j-1], C[i-1,j], C[i-1,j+1] }
            S[i,j] ← arg min_j { C[i-1,j-1], C[i-1,j], C[i-1,j+1] }
    return C, S
```

④ Compute sol:

```
Chess-Sol ( C, S, n )
    j ← min_k {C[n-1,k]}
    Print-Square (n-1, j)
    for i ← n-1 downto 1
        Print-Square (i-1, S[i,j])
        j ← S[i,j]
```

check this ... → OK

Other DP notes



## Matrix chain multiplication

Input: A sequence $\langle p_0, p_1, p_2, \ldots, p_n \rangle$ of matrix dimensions, where matrix $A_i$ is $p_{i-1} \times p_i$.

Output: A parenthesization of the product $A_1 A_2 \cdots A_n$ that yields the fewest scalar multiplications.

Example:

$A_1$ $10 \times 100$ $\quad p_0 = 10$

$A_2$ $100 \times 5$ $\quad p_1 = 100$

$A_3$ $5 \times 50$ $\quad p_2 = 5$

$\quad p_3 = 50$

Takes $pqr$ scalar mults to multiply $(p \times q) \cdot (q \times r)$.

$((A_1 A_2) A_3) \Rightarrow 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50$
$= 5000 + 2500$
$= 7500$

$(A_1 (A_2 A_3)) \Rightarrow 100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50$
$= 25000 + 50000$
$= 75000$

Second way is 10× worse!

Why not just try all ways to parenthesize?
Let $P(n)$ = # ways to parenthesize $n$ matrices.
Can split seq. bet. kth & (k+1)st for
any $k = 1, 2, \ldots, n-1$, then parenthesize
the 2 resulting subseqs. independently.

$$P(n) = \begin{cases} 1 & \text{if } n=1 \text{ or } n=2, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 3. \end{cases}$$

Show that $P(n) \geq 2^{n-2} = \Omega(2^n)$:
True for $n=1$, $n=2$. Assume true $\forall k < n$.

$$\begin{aligned} P(n) &= \sum_{k=1}^{n-1} P(k)P(n-k) \\ &\geq P(1)P(n-1) + P(n-1)P(1) \\ &= 2P(1)P(n-1) \\ &= 2P(n-1) \\ &\geq 2 \cdot 2^{n-1-2} \\ &= 2^{n-2}. \end{aligned}$$

In fact, $P(n) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$.

## The structure of an optimal parenthesization

Notation: $[i,j] = A_i A_{i+1} \cdots A_j$

Opt. paren of $[1,n]$ splits product at some $1 \le k <$

$\Rightarrow$ compute $[1,k]$, $[k+1,n]$, then combine to compute $[1,n]$.

Cost = cost to compute $[1,k]$ +
" " " $[k+1,n]$ +
cost of multiplying.

Key observation:

(Paren of $[1,k]$ within opt-paren of $[1,n]$ must be an opt paren of $[1,k]$.

Why? Suppose $\exists$ cheaper way. Then use it in opt. paren of $[1,n]$ to get a paren whose cost is < opt cost.

Similarly, paren of $[k+1,n]$ must be opt.

This is an example of optimal substructure: an opt soln to the problem contains within it opt. solns to subproblems.

Recursively defn of value of an opt soln

Let $m[i,j]$ = min cost of computing $[i,n]$.
Want $m[1,n]$.

For $[i,i]$, have just 1 matrix $\Rightarrow m[i,i]=0 \ \forall i$
For $[i,j]$, let opt. split be at position $k \Rightarrow$
$[i,j] = [i,k]\cdot[k+1,j] \Rightarrow$
$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
Note: $[i,k]$ is $p_{i-1} \times p_k$, $[k+1,j]$ is $p_k \times p_j$

But we don't know which is the optimal $k$.
Must check for candidate $k = i, \ldots, j-1$

$$\therefore m[i,j] = \begin{cases} 0 & \text{if } i=j, \\ \min_{k=i}^{j-1} \{ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \} & \text{if } i<j \end{cases}$$

start here.

We'll also need to keep track of the values of $k$:
$s[i,j] = k$ s.t. an opt. paren of $[i,j]$ splits at $k$.

## ② Recursively defn of value of an opt soln

Let $m[i,j]$ = min cost of computing $[1,n]$.
Want $m[1,n]$.

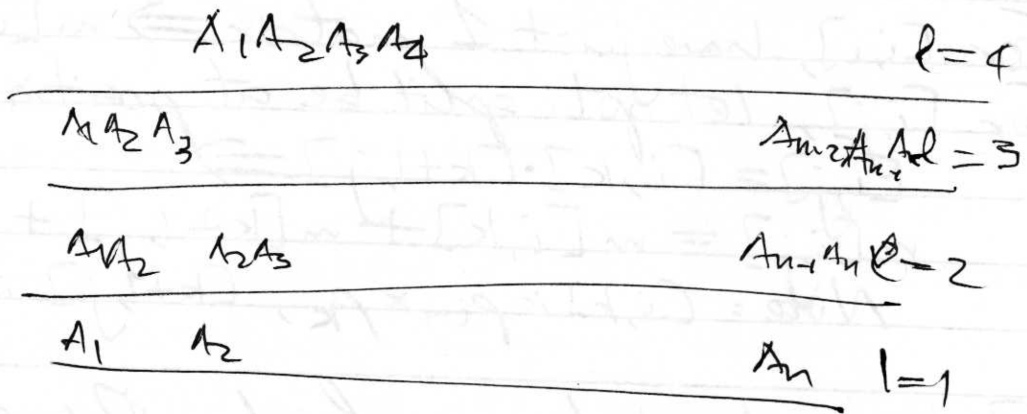
For $[i,i]$, have just 1 matrix $\Rightarrow m[i,i]=0 \; \forall i$.
For $[i,j]$, let opt. split be at position $k \Rightarrow$
$[i,j] = [i,k]\cdot[k+1,j] \Rightarrow$
$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
Note: $[i,k]$ is $p_{i-1}\times p_k$, $[k+1,j]$ is $p_k \times p_j$.

But we don't know which is the optimal k.
Must check for candidate $k = i,\ldots,j-1$.

$$\therefore m[i,j] = \begin{cases} 0 & \text{if } i=j, \\ \min\limits_{k=1}^{j-1} \{ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \} & \text{if } i<j \end{cases}$$

We'll also need to keep track of the values of k:
$s[i,j] = k$ s.t. an opt. paren of $[i,j]$ splits at k.

$$S[i,j] = \underset{k\in\{i,\ldots,j-1\}}{\arg\min} \{ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \} \quad \text{if } i<j$$


A1A2A3A4
l=4
A1A2A3
An-2An-1An l=3
A1A2 A2A3
An-1An l=2
A1 A2
An l=1

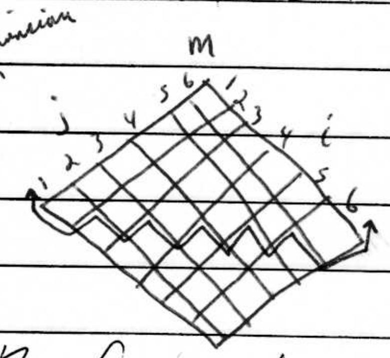
## Computing an opt. soln

Could write a recursive alg based on $m[i,j]$ equation, but this would be exponential tim

Observation: have only 1 problem for each unique pair $(i,j)$.

How many such problems? $\binom{n}{2} + n = \Theta(n^2)$ (i<j, i=j)

A recursive alg might encounter same subproblem over & over — we'll make sure we encounter it once.

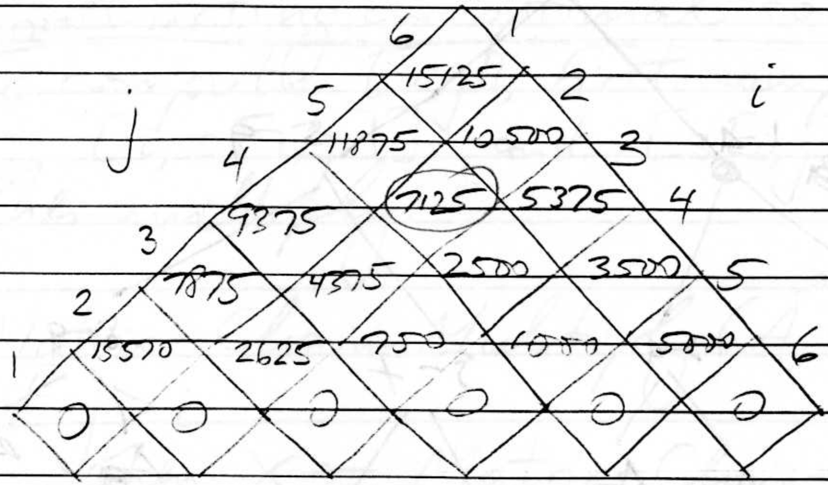



③ compute bottom-up

```
Matrix-Chain-Order(p, n)        ← array of dimension p0 p1 ... pn
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n     ▷ l = length of subchain
    do for i ← 1 to n - l + 1
        do j ← i + l - 1
           m[i, j] ← ∞
           for k ← i to j - 1                      } check all splits
               do q ← m[i, k] + m[k+1, j]          } to find min &
                      + p_{i-1} p_k p_j            } arg min
                  if q < m[i, j]
                     then m[i, j] ← q
                          s[i, j] ← k
return m and s
```

CS25-X94

L7 P7
also L8 P1



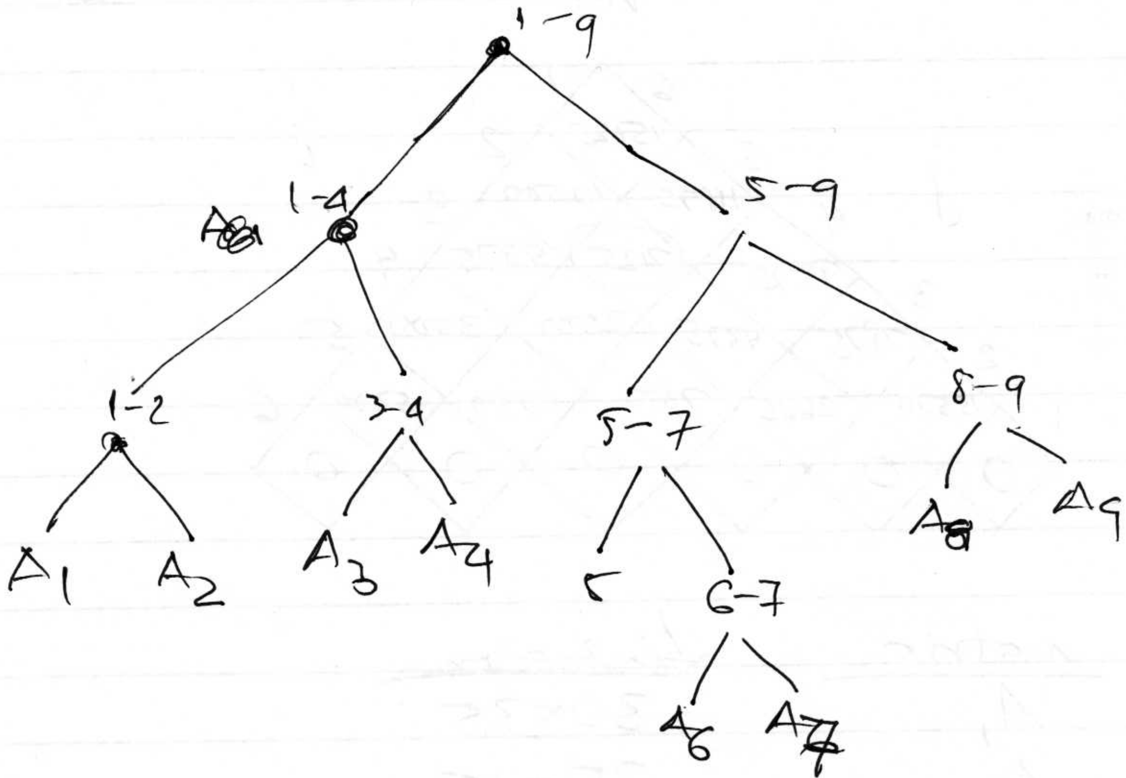

| matrix | dimension |
|---|---|
| $A_1$ | $30 \times 35$ |
| $A_2$ | $35 \times 15$ |
| $A_3$ | $15 \times 5$ |
| $A_4$ | $5 \times 10$ |
| $A_5$ | $10 \times 20$ |
| $A_6$ | $20 \times 25$ |

Computes bottom to top, left to right.

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

Analysis: $O(n^3)$.
Actually $\Theta(n^3)$ (HW problem)

$$((A_1 A_2)(A_3 A_4))((A_5(A_6 A_7))(A_8 A_9))$$

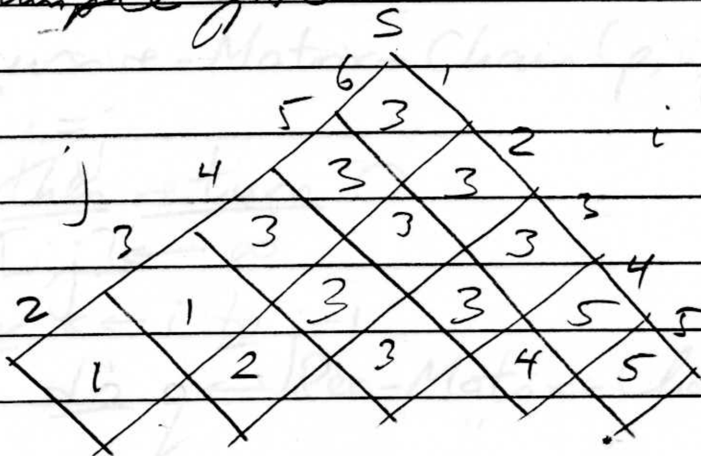


(4) <u>Constructing an optimal soln</u> from computed info

To compute [i,j], first compute
 [i, s[i,j]] and [s[i,j]+1, j]
then multiply.

```
Matrix-Chain-Multiply(A, s, i, j)      -- array of matrices A_1 - A_n
if j > i
   then X <- Matrix-Chain-Multiply(A, s, i, s[i,j])
        Y <-        "                (A, s, s[i,j]+1, j)
        return X x Y
   else return A_i
```

Example gives




$(A_1 A_2 A_3)(A_4 A_5 A_6)$

$(A_1 (A_2 A_3))((A_4 A_5) A_6)$

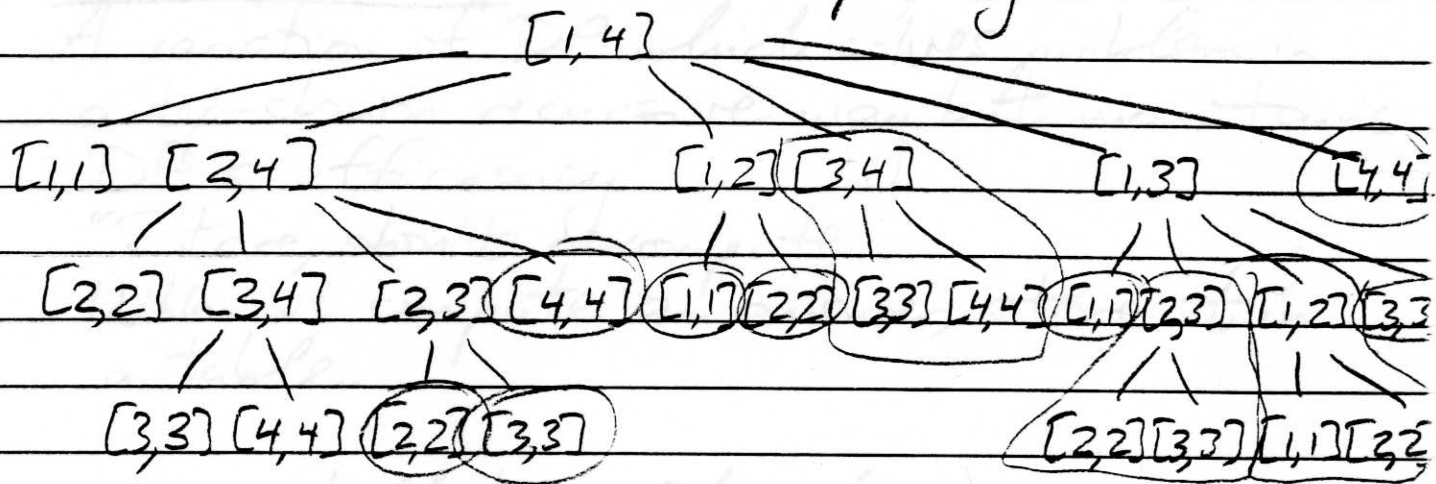
## Elements of DP

- Opt substructure – already saw
- Overlapping subproblems
  Need a small space of subproblems – a recursive alg would keep solving the same ones over & over.
  For DP, # of distinct subprobs should be polynomial in input size.

Consider recursive program based on earlier formula: $m[i,j] = \min_{k=i}^{j-1} \{ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \}$

```
Recursive-Matrix-Chain(p, i, j)
  if i = j
    then return 0
  m[i,j] ← ∞
  for k ← i to j-1
      do q ← Rec-Matrix-Chain(p, i, k) +
                 "     (p, k+1, j) + p_{i-1} p_k p_j
         if q < m[i,j]
            then m[i,j] ← q
  return m[i,j]
```

Recursion tree: values not recomputed by memoization circled




Has many repeated subproblems.

Time is $\Omega(2^n)$:

Assume: $T(k) \geq 2^k \quad \forall k < n$

Prove: $T(n) \geq 2^n$

$T(1) \geq 1$

$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad$ if $n > 1$.

$\geq 2T(n-1)$

$\geq 2 \cdot 2^{n-1}$

$= 2^n$

Compare to DP soln, which solves only $\Theta(n^2)$ subproblems.

## Memoization

A variation of DP which solves problem in a top-down recursive way but maintains DP's efficiency.

"Store, don't recompute"

When we compute an answer, store it in a table.

```
Memoized-Matrix-Chain(p, n)
  for i ← 1 to n
      do for j ← 1 to n
             do m[i,j] ← ∞
  return Lookup-Chain(p, 1, n)

Lookup-Chain(p, i, j)
  if m[i,j] < ∞
      return m[i,j]
  if i = j
      then m[i,j] ← 0
      else for k ← i to j-1
               do q ← Lookup-Chain(p, i, k) +
                         "      (p, k+1, j) + p_{i-1} p_k p_j
                  if q < m[i,j]
                      then m[i,j] ← q
  return m[i,j]
```

Analysis: $\Theta(n^2)$ entries to be filled
$O(n)$ time to fill each entry
$\Rightarrow O(n^3)$ ($\Omega$-analysis same as for DP)

$\therefore$ Memoization has turned an $\Omega(2^n)$ alg into $O(n^3)$ alg.

In practice:

- DP often faster because less overhead (no recursion) & regular access patterns.
- Memoization can be faster if not all subproblems need to be solved.

Longest common subsequence

Problem: Given two sequences $x[1..m]$ & $y[1..n]$, find a longest subsequence common to both.

x A B C B D A B

y B D C A B A

LCS = BCBA

# CS 25 - Algorithms

10/13/95

**Last time** (chap 16)

- Dynamic Programming

**Today** (chap 16, 17)

- DP vs. Greedy: 0-1 & Fractional Knapsack

**Handouts**

- HW 4

**Announcements**

- Group policy
- HW 2 solutions
- HW 2

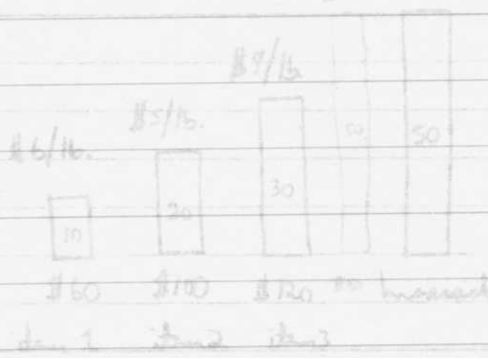
$max = 70$

$\bar{x} = 50.25$

$\sigma = 12.75$

$\frac{2}{3}\sigma = 8.5$

| | |
|---|---|
| 0 – 41.75 | 25% |
| 41.75 – 50.25 | 25% |
| 50.25 – 58.75 | 25% |
| 58.75 – 70 | 25% |



---

## 0-1 Knapsack Problem

- $n$ items
- $\forall i$, item $i$ has weight $w_i$ and value $v_i$
- knapsack which holds $W$ pounds

Goal: optimize value of items taken

Restrictions:
- must either take item or leave it
- total weight cannot exceed $W$

Javed Aslam

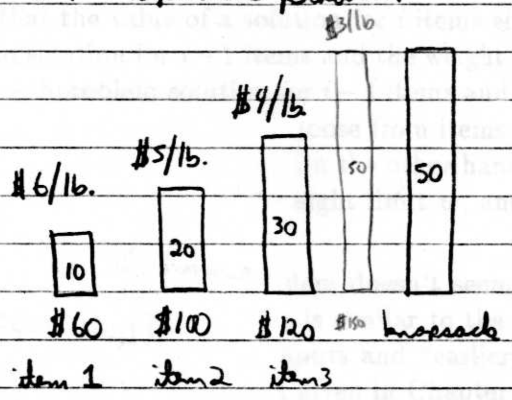


Greedy choice? – take items with highest value per pound.

Does this work?

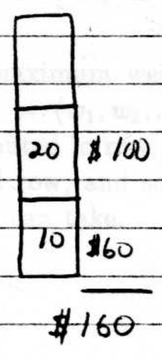
2) Fractional knapsack problem – yes (argue intuitively)

1) 0-1 knapsack problem – no



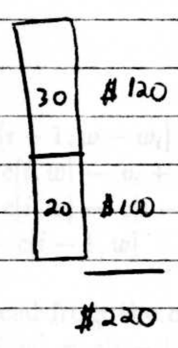

greedy strategies:
① largest → smallest $/lb
② largest → smallest value
③ smallest → largest weight

– greedy alg:




$160

– optimal:




$220

Note: can use dynamic programming to solve this problem...

# 0-1 Knapsack problem

## 6 Exercise 17.2-2

The solution is based on the optimal-substructure observation in the text: Let $i$ be the highest-numbered item in an optimal solution $S$ for $W$ pounds and items $1..n$. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and items $1..i-1$, and the value of the solution $S$ is $v_i$ plus the value of the subproblem solution $S'$.

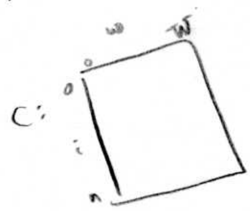
We can express this in the following formula: Define $c[i, w]$ to be the value of the solution for items $1..i$ and maximum weight $w$. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 , \\ c[i-1, w] & \text{if } w_i > w , \\ \max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i . \end{cases}$$

This says that the value of a solution for $i$ items either includes item $i$, in which case it is $v_i$ plus a subproblem solution for $i-1$ items and the weight excluding $w_i$, or doesn't include item $i$, in which case it is a subproblem solution for $i-1$ items and the same weight. That is, if the thief picks item $i$, he takes $v_i$ value, and he can choose from items $1..i-1$ up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item $i$, he can choose from items $1..i-1$ up to the weight limit $w$, and get $c[i-1, w]$ value. The better of these two choices should be made.

Although the 0-1 knapsack problem doesn't seem analogous to the longest-common-subsequence problem, the above formula for $c$ is similar to the LCS formula: boundary values are 0, and other values are computed from the inputs and "earlier" values of $c$. So the 0-1 knapsack algorithm is like the LCS-LENGTH algorithm given in Chapter 16 for finding a longest common subsequence of two sequences.

The algorithm takes as inputs the maximum weight $W$, the number of items $n$, and the two sequences $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order. (That is, the first row of $c$ is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.



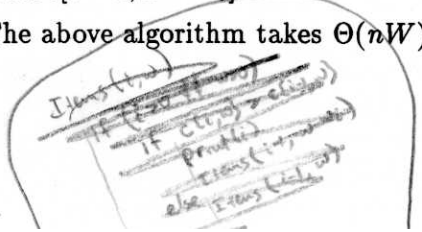

```
DYNAMIC-0-1-KNAPSACK(v, w, n, W)
    for w ← 0 to W
        do c[0, w] ← 0
    for i ← 1 to n
        do c[i, 0] ← 0
           for w ← 1 to W
               do if w_i ≤ w
                     then if v_i + c[i-1, w-w_i] > c[i-1, w]
                             then c[i, w] ← v_i + c[i-1, w-w_i]
                             else c[i, w] ← c[i-1, w]
                     else c[i, w] ← c[i-1, w]
```

```
Items(c[], w[], n, W)
  while n>0 and W>0
    if c[n,W] = c[n-1, W]
      n ← n-1
    else
      Print n
      W ← W - w[n]
      n ← n-1

Simpler  Items( --- )
  while --
    if c[n,W] > c[n-1,W]
      Print n
      W ← W - w[n]
      n ← n-1
```

The set of items to take can be deduced from the $c$ table by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i-1, w]$, item $i$ is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise item $i$ is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

The above algorithm takes $\Theta(nW)$ time total:



Do this first

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ \max\limits_{j \le i\,:\, w_j \le w} \{ v_j + C[j-1, w-w_j] \} \end{cases}$$

Idea: don't know what highest numbered item picked is, so try all possibilities..

- $\Theta(nW)$ to fill in the $c$ table — $(n+1)\cdot(W+1)$ entries each requiring $\Theta(1)$ time to compute.
- $O(n)$ time to trace the solution (since it starts in row $n$ of the table and moves up 1 row at each step).

Do this first

$$c[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ \max\limits_{j \le i:\ w_j \le w} \{ v_j + c[j-1, w-w_j] \} \end{cases}$$

Idea: don't know what highest numbered item picked is, so try all possibilities...