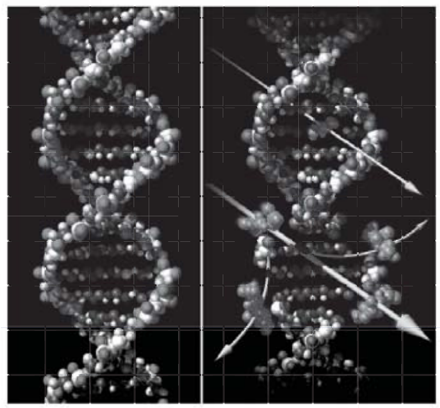
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

# Dynamic Programming



Effects of radiation on DNA's double helix, 2003. U.S. government image. NASA-MSFC.



## Terrible Fibonacci Computation

- Fibonacci sequence:

  $f_1$=1
  $f_2$=1
  $f_3$=2
  $f_4$=3
  $f_5$=5
  $f_6$=8
  $f_7$=13
  ……

- f($n$)
- 1. if ($n$=1) or ($n$=2) then return 1;
- 2. else return f($n$-1)+f($n$-2);

- This algorithm is far from being efficient, as there are many duplicate recursive calls to the procedure.

# Treat Space for Time

The main idea of dynamic programming

```
f(n)
if ((n=1) or (n=2)) return 1;
else {
    fn_1=1;
    fn_2=1;
    for k←3 to n {
        fn=fn_1+fn_2;
        fn_2=fn_1;
        fn_1=fn;
    }
}
return fn;
```

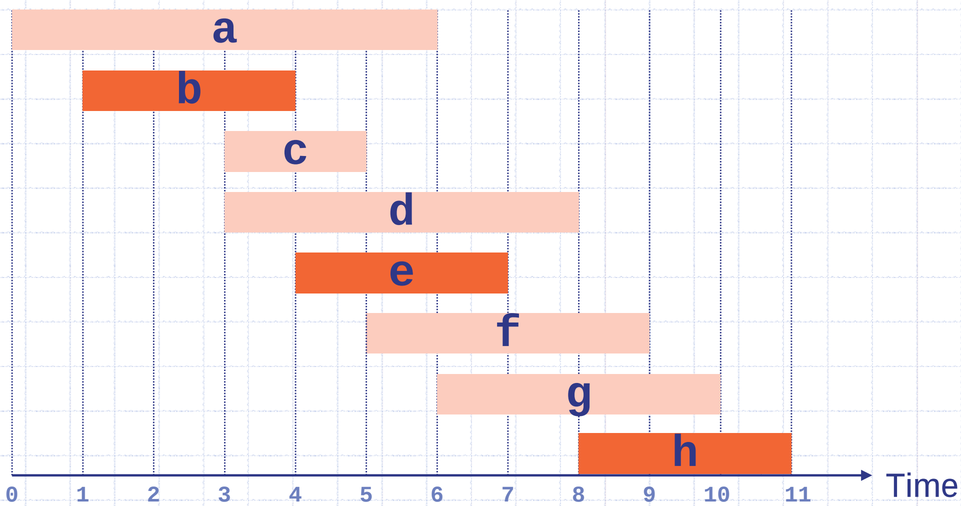
- Time: $n$-2 additions $\Rightarrow \theta(n)$
- Space: $\theta(1)$

# Dynamic Programming

- An algorithm that employs the dynamic programming technique is not necessarily recursive by itself, but the underlying solution of the problem is usually started in the form of a recursive function.
- This technique resorts to evaluating the recurrence in a bottom-up manner, **storing intermediate results** that are used later on to compute the desired solution.
- This technique applies to many **combinatorial optimization problems** to derive efficient algorithms.

# Task Scheduling

- Given: a set T of n tasks, start time, $s_i$ and finish time, $f_i$ (where $s_i < f_i$)
- Goal: Perform a maximum number of compatible jobs on a single machine.






# Task Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job in the order, provided it's compatible with the ones already taken.

**Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.**

set of jobs selected

```
A ← ϕ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

- Implementation: O(n log n).
  - Let job j* denote the job that was added last to A.
  - Job j is compatible with A if $s_j \ge f_{j*}$, i.e., j starts after j* finished.

# Telescope Scheduling Problem

- Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them.
- This high demand for observation times is especially true, for instance, for a space telescope, which could receive thousands of observation requests per month.
- The start and finish times for an observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee.

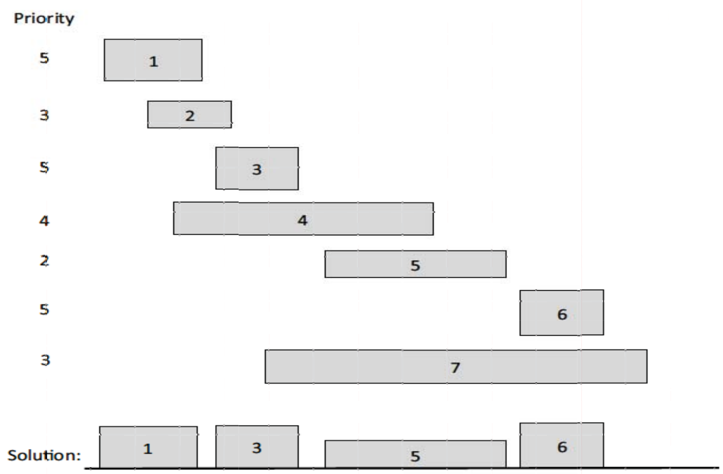


# Telescope Scheduling Problem

- The input to the telescope scheduling problem is a list, **L**, of observation requests, where each request, **i**, consists of the following elements:
  - a **requested start time**, $\mathbf{s_i}$, which is the moment when a requested observation should begin
  - a **finish time**, $\mathbf{f_i}$, which is the moment when the observation should finish.
  - a positive numerical **benefit**, $\mathbf{b_i}$, which is an indicator of the scientific gain expected by performing this observation.
- Task Scheduling is a special case of this problem where every task has the same benefit.

# Goal:

How to maximize the total benefit of the observations that are performed by the schedule?




The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request's benefit (Priority) on the left. The optimal solution has total benefit 17=5+5+2+5.



# False Start 1: Brute Force

- There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of L and choose the one that has the highest total benefit without causing any scheduling conflicts.
- Implementing this brute-force algorithm would take $O(n2^n)$ time, where n is the number of observation requests.
- We can do much better than this, however, by using other programming technique.

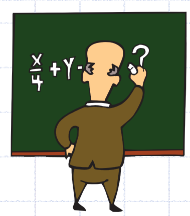
# False Start 2: Greedy Method

- A natural **greedy** strategy would be to consider the observation requests ordered by non-increasing benefits, and include each request that doesn't conflict with any chosen before it.
- This strategy doesn't lead to an optimal solution, however. For instance, suppose we had a list containing just 3 requests — one with benefit 100 that conflicts with two nonconflicting requests with benefit 75 each.
  - The greedy method would choose the observation with benefit 100, whereas we can achieve a total benefit of 150 by taking the two requests with benefit 75 each.
- How about ordering the observations by finish time?

Possible Quiz Question: Find a counter-example.



# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as i, j, k, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal solutions of subproblem.
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

# Defining Simple Subproblems

- A natural way to define general subproblems is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits.
- Unlike Greedy Method, we are allowed to undo our choices, instead of sticking to the greedy choice.
- So let us order observations by finish times.

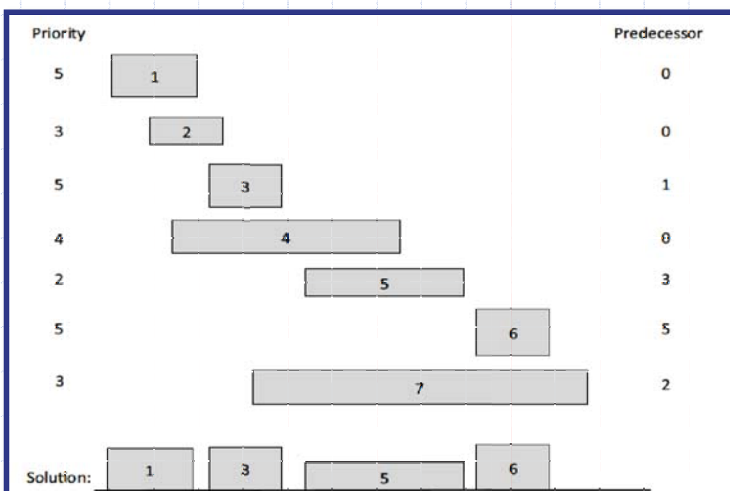
$B_i$ = the maximum benefit that can be achieved with the first $i$ requests in $L$.

So, as a boundary condition, we get that $B_0 = 0$.



# Predecessors

- For any request **i**, the set of other requests that conflict with **i** cannot be in the solution if **i** is in the solution.
- Define the **predecessor**, pred(**i**), for each request, **i**, then, to be the largest index, **j < i**, such that requests **i** and **j** don't conflict. If there is no such index, then define the predecessor of **i** to be 0.



# Subproblem Optimality

- A schedule that achieves the optimal value, $\mathbf{B_i}$, either includes observation **i** or not.

- If the optimal schedule achieving the benefit $B_i$ includes observation $i$, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most pred$(i)$.
- On the other hand, if the optimal schedule achieving the benefit $B_i$ does not include observation $i$, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves $B_{i-1}$.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1}, B_{\text{pred}(i)} + b_i\}.$$



# Subproblem is Overlapping

- $\mathbf{B}_i = \max\{ \mathbf{B}_{i-1}, \mathbf{B}_{\text{pred}(i)} + b_i \}$ gives the final solution when i=n.
- It has subproblem overlap.
- Thus, it is most efficient for us to use memoization when computing $\mathbf{B_i}$ values, by storing them in an array, **B**, which is indexed from 0 to **n**.
- Given the ordering of requests by finish times and an array, **P**, so that **P[i]** = pred(**i**), then we can fill in the array, **B**, using the following simple algorithm:

$B[0] \leftarrow 0$
**for** $i = 1$ to $n$ **do**
  $B[i] \leftarrow \max\{B[i-1], B[P[i]] + b_i\}$

Predecessor of i

After this algorithm completes, the benefit of the optimal solution will be $B[n]$

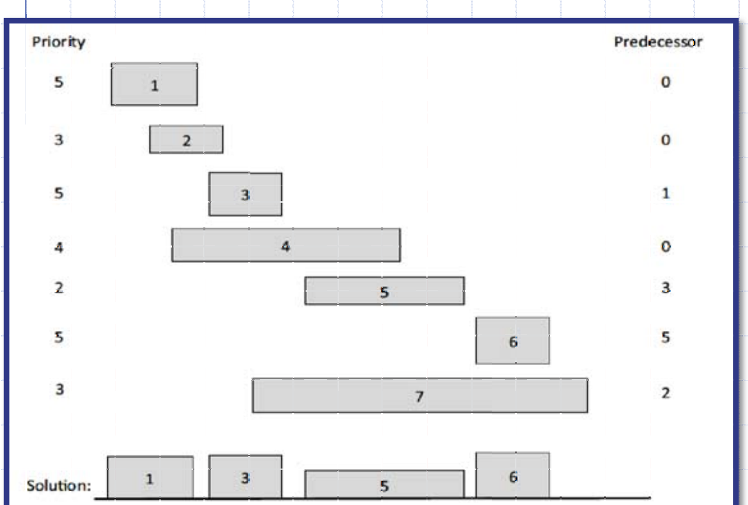
# Analysis of the Algorithm

- It is easy to see that the running time of this algorithm is **O(n)**, assuming the list **L** is ordered by finish times and we are given the predecessor for each request **i**.
- Of course, we can easily sort **L** by finish times if it is not already sorted according to this ordering – **O(n log n).**
- To compute the predecessor of each request **i**, we search $\mathbf{f_i}$ in **L** by binary search – **O(n log n).**



# Compute Predecessor

- To compute the predecessor of each request **i**, we search $\mathbf{f_i}$ in **L** by binary search on finish times – **O(n log n).**

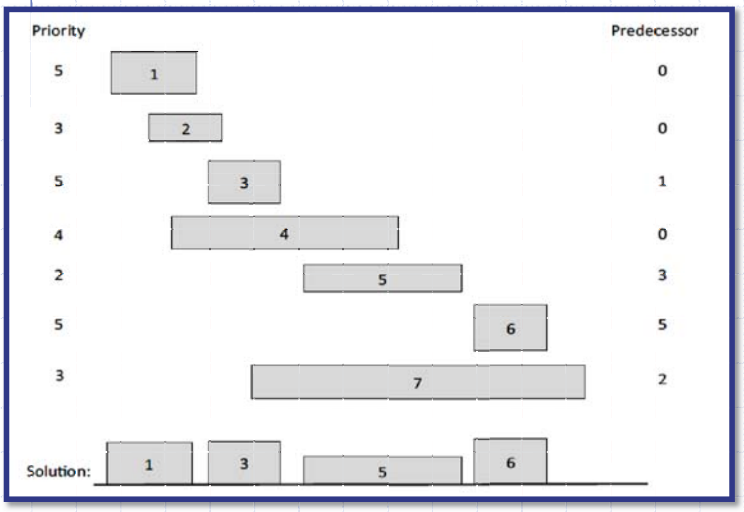



L: (0, 5), (2, 7), (6, 11), (4, 17), (13, 23), (24, 28), (9, 30).

2 is before (0, 5) finishes
6 is after (0, 5) finishes
4 is before (0, 5) finishes
13 is after (6, 11) ...
24 is after (13, 23) ...
9 is after (2, 7) ...

# What are in the optimal solution?

- B[n] gives only the optimal total benefit value, not the actual choices, which can be computed from B[i].
- This is typical for dynamic programming solutions.




How:

For j = n downto 1
if B[j] = B[j-1] then
request j is not
chosen



# Subsequences

- A ***subsequence*** of a character string $x_0x_1x_2 \ldots x_{n-1}$ is a string of the form $x_{i_1}x_{i_2} \ldots x_{i_k}$, where $i_j < i_{j+1}$.
- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGJIK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

# The Longest Common Subsequence (LCS) Problem

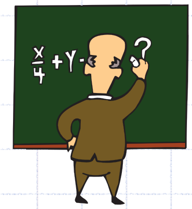
- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y
- Has applications to DNA similarity testing (alphabet is {A,C,G,T})
- Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence



# A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has $2^n$ subsequences
  - This is an exponential-time algorithm!

# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

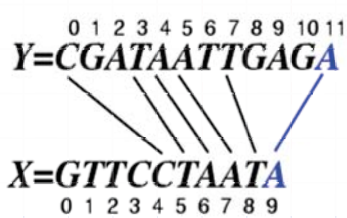


# A Dynamic-Programming Approach to the LCS Problem

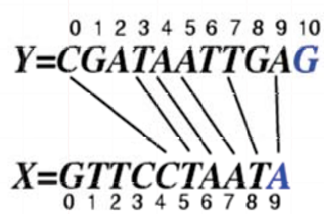
- Define L[i,j] to be the length of the longest common subsequence of X[1..i] and Y[1..j].
- Allow for 0 as an index, so L[0,k] = 0 and L[k,0]=0, to indicate that the null part of X or Y has no match with the other.
- Then we can define L[i,j] in the general case as follows:
  1. If $x_i=y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

Case 1:

$L[8,10]=5$




Case 2:




$L[9,9]=6$

$L[8,10]=5$

# LCS Algorithm

**Algorithm** LCS(X,Y ):

**Input:** Strings X and Y with n and m elements, respectively

**Output:** For i = 1,...,n, j = 1,...,m, the length L[i, j] of a longest string that is a subsequence of both the string X[1..i] = $x_1x_2...x_i$ and the string Y [1.. j] = $y_1y_2...y_j$

**for** i =1 to n **do**
    L[i,0] = 0
**for** j =1 to m **do**
    L[0,j] = 0
**for** i =1 to n **do**
    **for** j =1 to m **do**
        **if** $x_i = y_j$ **then**
            L[i, j] = L[i-1, j-1] + 1
        **else**
            L[i, j] = max{L[i-1, j] , L[i, j-1]}
**return** array L

L is an (n+1)x(m+1) matrix.



# Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates *n* times
  - The inner one iterates *m* times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is O(*nm*)
- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).

# From L[i,j] to actual LCS

**Algorithm** getLCS(X,Y ):
**Input:** Strings X and Y with n and m elements, respectively
**Output:** One of the longest common subsequence of X and Y.

LCS(X,Y) /* Now, for i = 1,...,n, j = 1,...,m, the length L[i, j] of a longest string that is a subsequence of both the string X[1..i] = $x_1x_2...x_i$ and the string Y [1.. j] = $y_0y_1y_2...y_j$ */
i = n; j = m;
S = new stack();
**while** (i > 0 && j > 0) **do**
  **if** $x_i = y_j$ **then**
    push(S, $x_i$); i--; j--;
  **else if** L[i-1, j] > L[i, j-1]
    i--;
  **else**
    j--;
**return** stack S



# LCS Algorithm Example

Example:
A="steal",
B="staple"

What is the longest common subsequence of A and B?

Possible Quiz Question:
What are the content of L[0..7, 0..8] after calling LCS(A, B), where
A="vehicle",
B="vertices"

# Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of **A**, **C**, **G**, and **T** characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.



# Application: Edit Distance

- What is the minimal of steps needed to convert one string to another?
  - **ocurrance**
  - **occurrence**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | u | r | r | a | n | c | e | - |
| o | c | c | u | r | r | e | n | c | e |

6 mismatches, 1 gap

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | a | n | c | e |
| o | c | c | u | r | r | e | n | c | e |

1 mismatch, 1 gap

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | - | a | n | c | e |
| o | c | c | u | r | r | e | - | n | c | e |

0 mismatches, 3 gaps

# Minimal Edit Distance

- Define D[i,j] to be the minimal edit distance of X[1..i] and Y[1..j].
- Allow for 0 as an index, so D[0,j] = j and D[i,0]=i, to indicate that if one string is null, then the length of the other string is the edit distance.
- Then we can define D[i,j] in the general case as follows:
  1. If $x_i=y_j$, then D[i,j] = D[i-1,j-1] (we can add this match)
  2. If $x_i \neq y_j$, then D[i,j] = min{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+1} (we have no match here)

gap mismatch

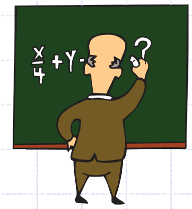
Possible Quiz Question:

Provide a complete algorithm for computing D[i,j] and analyze its complexity.

# Application: Edit Distance

- Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
  - Cost = sum of gap and mismatch penalties.
- Applications.
  - Basis for Unix/Linux diff.
  - Speech recognition.
  - Computational biology.

# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal solutions of subproblem.
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).



# DP Problem Patterns

- Telescope Scheduling Problem:
  - $B_i$ = the max profit from the first i requests
  - $B_i = \max(B_{i-1}, B_{pred(i)} + b_i)$
- Longest Common Subsequence Problem:
  - L[i,j] = the length of the longest common subsequence of X[1..i] and Y[1..j].
  - L[i,j] = L[i-1,j-1]+1 if X[i]=Y[j], max(L[i-1,j], L[i,j-1]) otherwise
- Edit Distance Problem:
  - D[i,j] = the shortest distance of X[1..i] and Y[1..j].
  - D[i,j] = D[i-1,j-1] if X[i]=Y[j], min(D[i-1,j-1], D[i-1,j], D[i,j-1])+1 otherwise

# Coins in a Line

- "Coins in a Line" is a game whose strategy is sometimes asked about during job interviews.
- In this game, an even number, n, of coins, of various denominations, are placed in a line.
- Two players, who we will call Alice and Bob, take turns removing one of the coins from either end of the remaining line of coins.
- The player who removes a set of coins with larger total value than the other player wins and gets to keep the money. The loser gets nothing.
- Game goal: get the most.

If the value of the first coin is $4, how would Alice do?


Alice: (1, $6), (6, $5), (4, $7) Total value = $18

Bob: (2, $5), (5, $3), (3, $2) Total value = $10

**Figure 12.7:** The coins-in-a-line game. In this instance, Alice goes first and ultimately ends up with $18 worth of coins. U.S. government images. Credit: U.S. Mint.

# False Start 1: Greedy Method

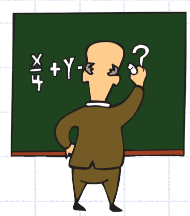
- A natural **greedy** strategy is "always choose the largest-valued available coin."
- But this doesn't always work:
  - [5, 10, 25, 10]: Alice chooses 10
  - [5, 10, 25]: Bob chooses 25
  - [5, 10]: Alice chooses 10
  - [5]: Bob chooses 5
- Alice's total value: 20, Bob's total value: 30. (Bob wins, Alice loses)

# False Start 2: Greedy Method

- Another **greedy** strategy is "choose all odds or all evens, whichever is better."
- Alice can always win with this strategy, but won't necessarily get the most money.
- Example: [1, 3, 6, 3, 1, 3]
- All odds = 1 + 6 + 1 = 8
- All evens = 3 + 3 + 3 = 9
- Alice's total value: $9, Bob's total value: $8.
- Alice wins $9, but could have won $10.
- How?



# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

# Defining Simple Subproblems

- Since Alice and Bob can remove coins from either end of the line, an appropriate way to define subproblems is in terms of a range of indices for the coins, assuming they are initially numbered from 1 to n.
- Thus, let us define the following indexed parameter:

$$M_{i,j} = \begin{cases} \text{the maximum value of coins taken by Alice, for coins} \\ \text{numbered } i \text{ to } j\text{, assuming Bob plays optimally.} \end{cases}$$

Therefore, the optimal value for Alice is determined by $M_{1,n}$.



# Subproblem Optimality

- Let us assume that the values of the coins are stored in an array, V, so that coin 1 is of value V[1], coin 2 is of value V[2], and so on.
- Note that, given the line of coins from coin i to coin j, the choice for Alice at this point is either to take coin i or coin j and thereby gain a coin of value V[i] or V[j].
- Once that choice is made, play turns to Bob, who we are assuming is playing optimally.
  - We should assume that Bob will make the choice among his possibilities that minimizes the total amount that Alice can get from the coins that remain.

# Subproblem Overlap

- Alice should choose based on the following:

> - If $j = i + 1$, then she should pick the larger of $V[i]$ and $V[j]$, and the game is over.
> - Otherwise, if Alice chooses coin $i$, then she gets a total value of
> $$\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i].$$
> - Otherwise, if Alice chooses coin $j$, then she gets a total value of
> $$\min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j].$$

- That is, we have initial conditions, for i=1,2,...,n-1:

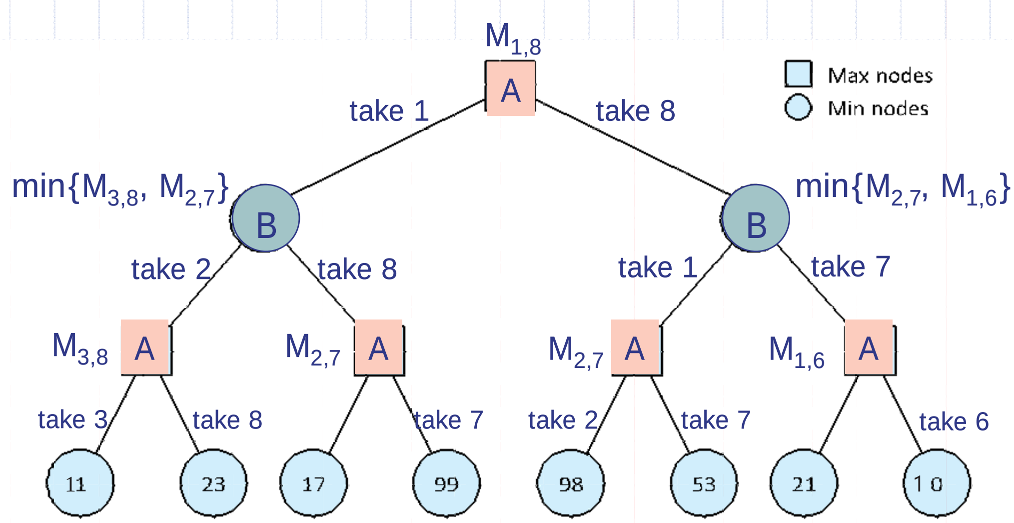
$$\boxed{M_{i,i+1} = \max\{V[i], V[i+1]\}.}$$

- And general equation:

$$\boxed{M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j]\right\}.}$$



# Decision Tree in Games: minimax tree




$$\boxed{M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j]\right\}.}$$

$$\boxed{M_{i,i+1} = \max\{V[i], V[i+1]\}.}$$

$$M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\, M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\, M_{i+1,j-1}\} + V[j]\right\}.$$

$$M_{i,i+1} = \max\{V[i], V[i+1]\}.$$

- Example: [1, 3, 6, 3, 1, 3]
- M:

| i\j | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 3 | | ? | | ? |
| 2 | | 6 | | ? | |
| 3 | | | 6 | | ? |
| 4 | | | | 3 | |
| 5 | | | | | 3 |

M[1,4] = max{
min{M[2,3], M[3,4]}+V[1],
min{M[1,2], M[2,3]}+V[4] }
= max{min{6, 6}+1, min{3,6}+3}
= max{7, 6}
= 7

M[2,5] = max{
min{M[3,4], M[4,5]}+V[2],
min{M[2,3], M[3,4]}+V[5] }
= max{min{6, 3}+3, min{6,6}+1}
= max{6, 7}
= 7

M[3,6] = max{
min{M[4,5], M[5,6]}+V[3],
min{M[3,4], M[4,5]}+V[6] }
= max{min{3, 3}+6, min{6,3}+3}
= max{9, 6}
= 9

# CoinInALine Algorithm

**Algorithm** CoinInALine(X,Y ):
**Input:** a list of n coins with values V[i] for i=1 to n, n is even.
**Output:** For i = 1,...,n-1, j = i+1,...,n, M[i, j] stores the maximal values that Alice can get from coins i to j.

```
for i =1 to n-1 do              // base case
    M[i, i+1] = max(V[i], V[i+1])

for k = 3 to n-1 step 2 do
    for i =1 to n-k do
        j = i+k                 // [i, j] has (k+1) coins
        v1 = min(M[i+1,j-1], M[i+2, j])
        v2 = min(M[i,j-2], M[i+1, j-1])
        M[i, j] = max(v1+V[i], v2+V[j])
return array M
```

$$M_{i,i+1} = \max\{V[i], V[i+1]\}.$$

$$M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\, M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\, M_{i+1,j-1}\} + V[j]\right\}.$$

# Analysis of the Algorithm

- We can compute the $\mathbf{M}_{i,j}$ values, then, using memoization, by starting with the definitions for the above initial conditions and then computing all the $\mathbf{M}_{i,j}$'s where j – i + 1 is 4, then for all such values where j – i + 1 is 6, and so on.
- Since there are **O**(**n**) iterations in this algorithm and each iteration runs in **O**(**n**) time, the total time for this algorithm is **O**($\mathbf{n^2}$).
- To recover the actual game strategy for Alice (and Bob), we simply need to note for each $\mathbf{M}_{i,j}$ whether Alice should choose coin i or coin j.

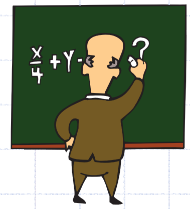


# DP Problem Patterns

- Telescope Scheduling Problem:
  - $B_i$ = the max profit from the first i requests: b[1..i]
  - $B_i = \max(B_{i-1}, B_{pred(i)} + b_i)$
- Longest Common Subsequence Problem:
  - $L_{i,j}$ = the length of the longest common subsequence of X[1..i] and Y[1..j].
  - $L_{i,j} = L_{i-1,j-1}+1$ if X[i]=Y[j], $\max(L_{i-1,j}, L_{i,j-1})$ otherwise
- Coin-in-a-line Problem:
  - $M_{i,j}$ = the max possible value of Alice for coins in $V[i..j]$.

$$M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\, M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\, M_{i+1,j-1}\} + V[j]\right\}.$$

# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).



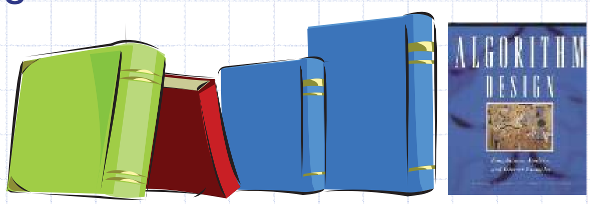
# The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - $w_i$ - a positive weight
  - $b_i$ - a positive value
- Goal: Choose items with maximum total value but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint: $\sum_{i \in T} w_i \leq W$

# Example

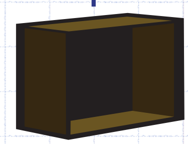
- Given: A set S of n items, with each item i having
  - $b_i$ - a positive “value”
  - $w_i$ - a positive “weight”
- Goal: Choose items with maximum total value but with weight at most W.

Items:




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Weight: | 4 in | 2 in | 2 in | 6 in | 2 in |
| value: | $20 | $3 | $6 | $25 | $80 |

“knapsack”



box of width 9 in

Solution:

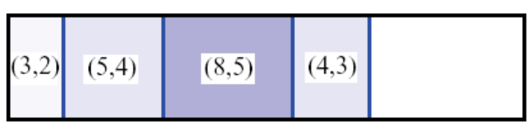
- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)



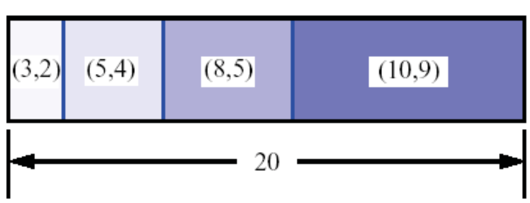
# A 0/1 Knapsack Algorithm, First Attempt

- $S_k$: Set of items numbered 1 to k.
- Define B[k] = best selection from $S_k$.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (value, weight) pairs and total weight W = 20

Best for $S_4$:




Best for $S_5$:



# A 0/1 Knapsack Algorithm, Second (Better) Attempt

- $S_k$: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from $S_k$ with weight at most w
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of $S_k$ with weight at most w is either
  - the best subset of $S_{k-1}$ with weight at most w or
  - the best subset of $S_{k-1}$ with weight at most $w–w_k$ plus item k



# 0/1 Knapsack Example

W + 1 →

n + 1 ↓

| $B[k,w]$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| { 1, 2, 3, 4, 5 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 34 | 40 |

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

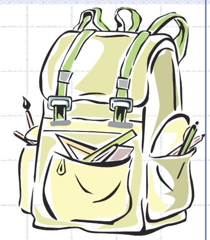
| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

W = 11

We can use two rows for all k in B[k,w].

OPT: { 4, 3 }
value = 22 + 18 = 40

# 0/1 Knapsack Algorithm



$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Recall the definition of B[k,w]
- Running time: O(nW).
- Not a polynomial-time algorithm since W is not the size of the input.
- This is a pseudo-polynomial time algorithm.
- Only two rows of B[k,w] is needed: replace B[k, w] by B[k%2, w] will work.
- So the space is O(W).

**Algorithm** *01Knapsack*(*S, W*):
  **Input:** set ***S*** of ***n*** items with value $b_i$ and weight $w_i$; maximum weight ***W***
  **Output:** value of best subset of ***S*** with weight at most ***W***
  **for** *w* ← **0 to** *W* **do** ***B*[0, *w*]** ← **0**
  **for** *k* ← 1 **to** *n* **do**
    ***B*[*k*, 0]** ← **0**
    **for** *w* ← 1 **to** *W* **do**
      ***B*[*k*, *w*]** ← ***B*[*k*-1, *w*]**
      **if** $w >= w_k$ &&
        $B[k-1, w-w_k] + b_k > B[k, w]$ **then**
        $B[k, w] \leftarrow B[k, w-w_k] + b_k$
  **return** *B*[*n*, *W*]



# 0/1 Knapsack Algorithm

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- How to get the actual set of items?
- First call *01Knapsack*(*S, W*) to get B[k,w].
- Then call

  ***01KS*(*S, B, n, W*)**
- Running time: O(n).

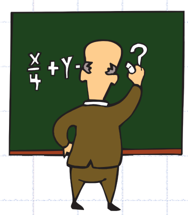
**Algorithm** ***01KS*(*S, B, k, w*)**:
  **Input:** set ***S*** of ***n*** items with value $b_i$ and weight $w_i$; maximum weight ***W***
  **if** $w >= w_k$ &&
    $B[n\%2, w-w_k] + b_k == B[n\%2, w]$
    ***01KS*(*S, B, k*-1, $w-w_k$)**
    **print(*k*)**
  **else**
    ***01KS*(*S, B, k*-1, *w*)**

# DP Problem Patterns

- Longest Common Subsequence Problem:
  - $L_{i,j}$ = the length of the longest common subsequence of X[1..i] and Y[1..j].
  - $L_{i,j} = L_{i-1,j-1}+1$ if X[i]=Y[j], max($L_{i-1,j}$, $L_{i,j-1}$) otherwise
- Coin-in-a-line Problem:
  - $M_{i,j}$ = the max possible value of Alice for coins in $V[i..j]$.
  - $$M_{i,j} = \max\left\{\min\{M_{i+1,j-1},\ M_{i+2,j}\} + V[i],\ \min\{M_{i,j-2},\ M_{i+1,j-1}\} + V[j]\right\}.$$
- 0-1 Knapsack Problem:
  - $B[k, w]$ = max value from the first $k$ items under weight limit $w$.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

# The General Dynamic Programming Technique



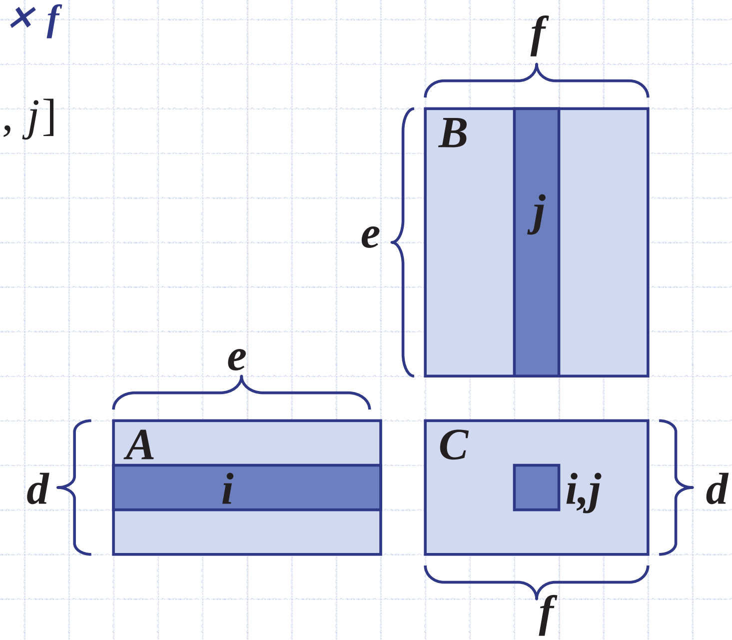

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal solutions of subproblem.
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

# Matrix Multiplication

- Review: Matrix Multiplication.
  - ***C*** = ***A********B***
  - ***A*** is ***d*** ✕ ***e*** and ***B*** is ***e*** ✕ ***f***

$$C[i, j] = \sum_{k=0}^{e-1} A[i,k] * B[k, j]$$

  - ***O***(***def*** ) time




# Matrix multiplication

**MATRIX-MULTIPLY (**A,B**)**
  **if** *columns* [A] ≠ *rows* [B]
    **then error "**incompatible dimensions**"**
    **else for** *i*←1 **to** *rows* [A]
        **for** *j*←1 **to** *columns* [B]
          C[*i*, *j*]←0
          **for** *k*←1 **to** *columns* [A]
            C[ *i*, *j* ]← C[ *i*, *j*] +A[ *i*, *k*]*B[ *k*, *j*]
  **return** C

Time: O(*d·e·f* ) if A is *d x e* and B is *e x f*.
Divide and Conquer can reduce it slightly.

# Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - $A_i$ is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is $3 \times 100$
  - C is $100 \times 5$
  - D is $5 \times 5$
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops



# An Exhaustive Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of paranethesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost $4^n$.
  - This is a terrible algorithm!

# A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 250+500+250 = 1000 ops



# Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 200 × 10
  - B is 10 × 10
  - C is 10 × 100
  - D is 100 × 100
  - Greedy idea #2 gives (A*(B*C))*D which takes 10000+200000+2000000=2,210,000 ops
  - (A*B)*(C*D) takes 20000+100000+200000=320,000 ops
  - A*((B*C)*D)) takes 10000+100000+200000=310,000 ops
- The greedy approach is not giving us the optimal value.

# A "Recursive" Approach

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*\ldots*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiplication.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.



# A Characterizing Equation

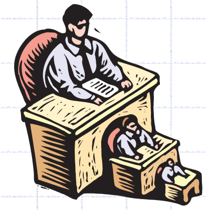
- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that $A_i$ is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,i} = 0$$

$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note that subproblems are not independent -- the **subproblems overlap**.

# A Dynamic Programming Algorithm



- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are 0, so start with them
- Then do length 2,3,... subproblems, and so on.
- The running time is O($n^3$)

**Algorithm** *matrixChain*(*S*):

**Input:** sequence *S* of *n* matrices to be multiplied

**Output:** number of operations in an optimal paranethization of *S*

**for** $i \leftarrow 0$ **to** $n$-1 **do**

  $N_{i,i} \leftarrow 0$

**for** $b \leftarrow 1$ **to** $n$-1 **do**

  **for** $i \leftarrow 0$ **to** $n$-$b$-1 **do**

    $j \leftarrow i+b$

    $N_{i,j} \leftarrow$ +**infinity**

    **for** $k \leftarrow$ i **to** $j$-1 **do**

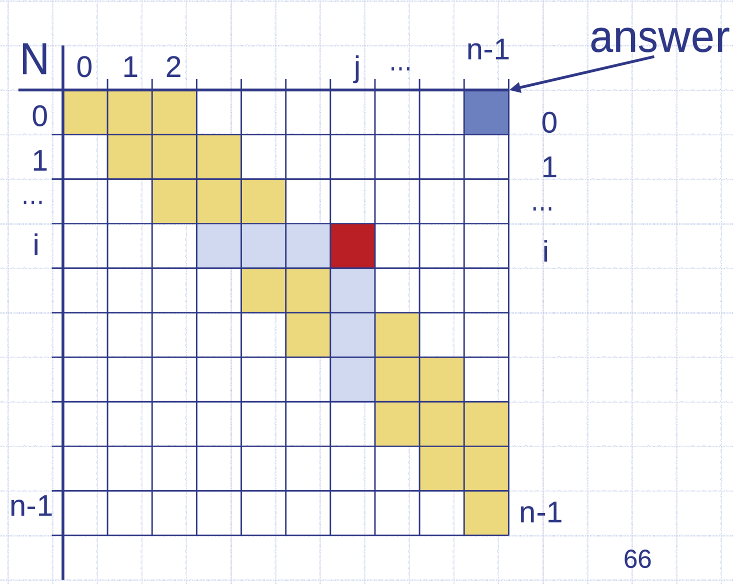
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

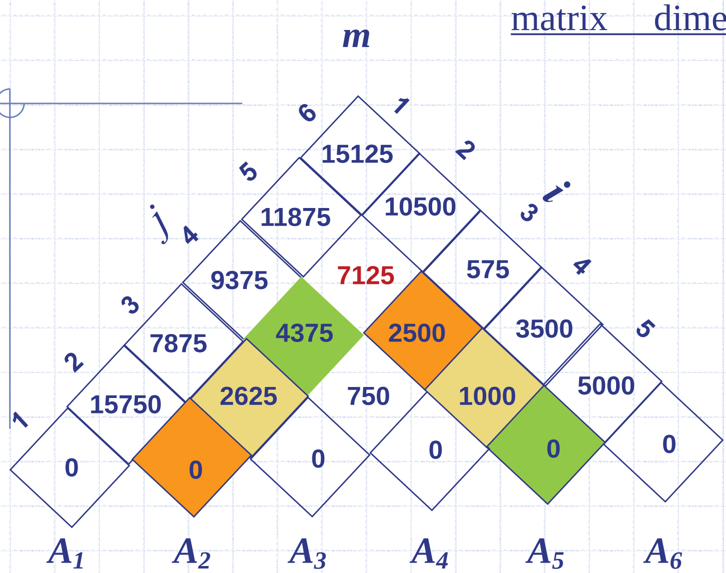


# A Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: O($n^3$)
- Getting actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



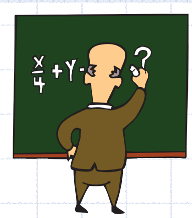



| matrix | dimensions: 30, 35, 15, 5, 10, 20, 25 |
|---|---|
| $A_1$ | 30 x 35 |
| $A_2$ | 35 x 15 |
| $A_3$ | 15 x 5 |
| $A_4$ | 5 x 10 |
| $A_5$ | 10 x 20 |
| $A_6$ | 20 x 25 |

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

# The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal solutions of subproblem.
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).