Dynamic Programming

Dynamic programming is not about filling in tables.
It's about smart recursion!

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson / Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Dynamic Programming

The name “dynamic programming" is due to Bellman, but the paradigm was already used in the 12th century.

- “**dynamic**" refers to the **multistage, time-varying processes**, and
- “**programming**" refers to **planning** or **scheduling**, in many cases by filling in a table.

Dynamic programming is recursion without repetition. DP in two stages:

1. Formulate the problems recursively.
   - Describe the problem that you want to solve,
   - Identify subproblems (on smaller instances), and
   - Define a recursive formula for the problem in terms of the solutions to smaller subproblems.
2. Solve subproblems in order by the recurrence, and store their solutions.
   - Smallest first, using the answers to smaller problems
   - Until the whole lot of them is solved.

Recursion in Fibonacci numbers

The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them.

$$F_0 = 0,\ F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geqslant 2.$$

FIB1(n)
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
return FIB1($n-1$) + FIB1($n-2$)

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + O(1). \quad T(n) \approx \Theta(1.6^n)$$

Memoization. Instead of computing the same Fibonacci numbers over and over again from scatch, remember the results and use them when needed later.

$$T(n) = O(n)$$

FIB2(n)
if $n = 0$ **then**
    return 0
create an array $f[0, \dots, n]$
$f[0] = 0, f[1] = 1$
for $i = 2, \dots, n$ **do**
    $f[i] = f[i-1] + f[i-2]$
return $f[n]$

Shortest Paths in DAGs, Revisited

Dynamic programming is recursion without repetition. DP in two stages:

1. Formulate the problems recursively.
   - Describe the problem that you want to solve,
   - Identify subproblems (on smaller instances), and
   - Define a recursive formula for the problem in terms of the solutions to smaller subproblems.
2. Solve subproblems in order by the recurrence, and store their solutions.
   - Smallest first, using the answers to smaller problems
   - Until the whole lot of them is solved.

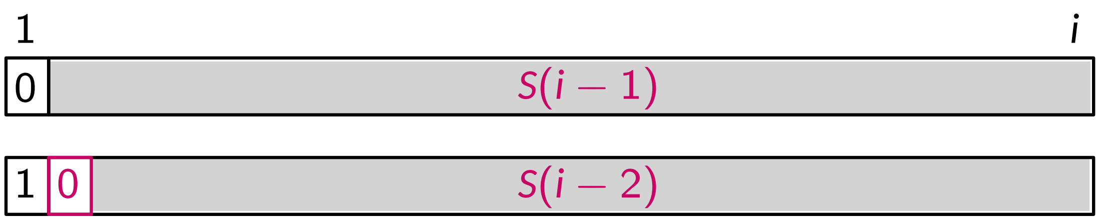
In general, there is

- an **ordering** (dag) on the subproblems, and
- a **relation** that shows how to solve a subproblem *given the answers to smaller subproblems that appear earlier in the ordering*.

Problem : Count the number of bit strings of length n that do not contain two or more consecutive '1's.

- $n = 1$: 0 1
- $n = 2$: 00 01 10
- $n = 3$: 000 001 010 100 101

Let $S(i)$ be the number of such bit strings of length i.
Then $S(1) = 2$, $S(2) = 3$.



$$S(i) = S(i-1) + S(i-2)$$

BitString(n)
if $n = 0$ **then**
  return 1
create an array $T[0, \dots, n]$
$T[0] = 1, T[1] = 2$
for $i = 2, \dots, n$ **do**
  $T[i] = T[i-1] + T[i-2]$
return $T[n]$

Largest Sum Contiguous Subarray

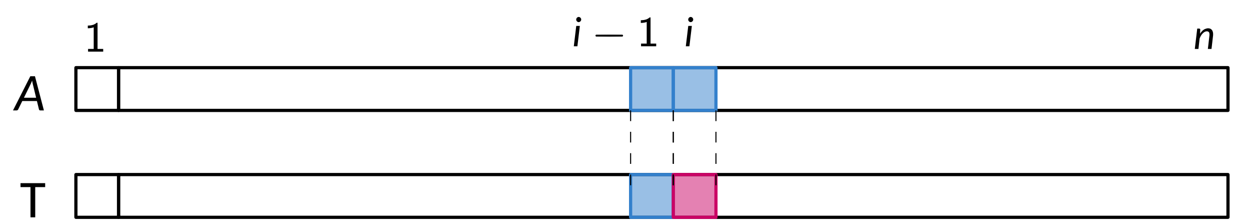
Input : An array A containing n numbers $A[1], \dots, A[n]$.
Goal : A contiguous subarray with largest sum.

-2   -3   4   -1   -2   1   5   -3

Let $S(i)$ be the largest sum among all subarrays that end with $A[i]$ for $i = 1, 2, \dots, n$.
Then $S(1) = A[1]$.

$$S(i) = \max\{S(i-1) + A[i],\ \ A[i]\}$$

idx = 1
for $i \leftarrow 2$ to n
    if $T[i-1] > 0$
        $T[i] = T[i-1] + A[i]$
    else
        $T[i] = A[i]$
    if $T[idx] < T[i]$
        $idx = i$
return idx



Time complexity: $O(n)$
Space complexity: $O(n)$

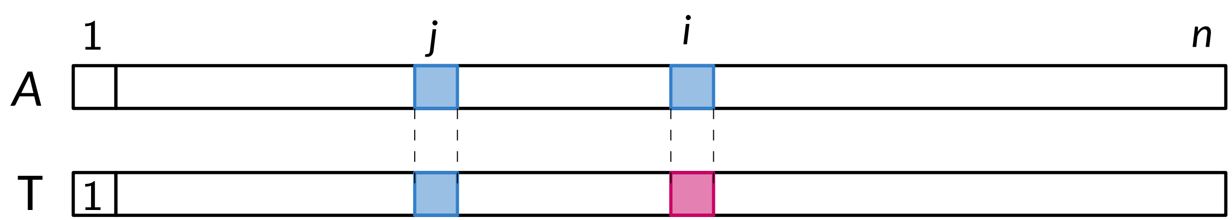
Longest Increasing Subsequences

Input : a sequence S of n numbers $a_1, \dots, a_n$.
Goal : A longest increasing subsequence of S.

5 2 8 6 3 5 6 9 7 8

Let $\mathrm{LIS}(i)$ be the length of LIS that ends with $A[i]$ for $i = 1, 2, \dots, n$.
Then $\mathrm{LIS}(1) = 1$.

$$\mathrm{LIS}(i) = 1 + \max\{\mathrm{LIS}(j) \mid j < i \text{ and } A[j] < A[i]\}$$



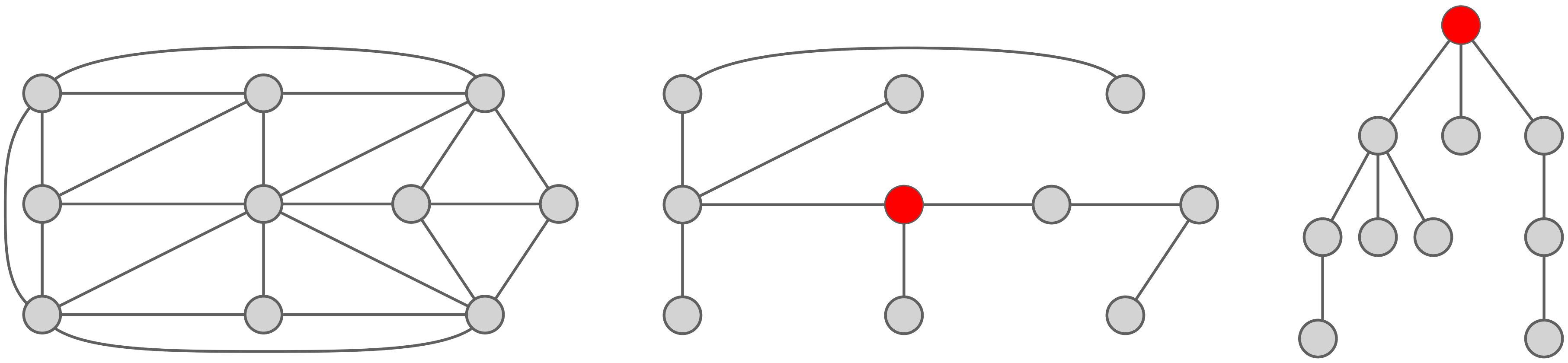
for $i \leftarrow 1$ to n
  $T[i] = 1$
$idx = 1$

for $i \leftarrow 2$ to n
  for $j \leftarrow 1$ to $i - 1$
    if $A[j] < A[i]$ and $T[i] < 1 + T[j]$
      $T[i] = 1 + T[j]$
  if $T[idx] < T[i]$
    $idx = i$
return idx

Time complexity: $O(n^2)$
Space complexity: $O(n)$

Independent Sets in Trees

Independent Set problem. Given a graph, find a largest subset of vertices with no edges between them.



In general, it is highly unlikely to be solvable in polynomial time. But if G is a tree, we can solve it in linear time.

What is the appropriate subproblem when G is a tree?

$$\mathrm{IS}(u) = \begin{cases} 1 & \text{if } u \text{ is a leaf} \\ \max\{\sum_{v=\mathrm{children}(u)} \mathrm{IS}(v), 1 + \sum_{w=\mathrm{grandchildren}(u)} \mathrm{IS}(w)\} & \text{otherwise} \end{cases}$$

Edit Distance

Edit distance between two strings X, Y is the minimum #. insertions, deletions, and substitutions of letters required to transform X to Y.

$\texttt{SNOWY} \xrightarrow{\texttt{insert}} \texttt{SSNOWY} \xrightarrow{\texttt{substitute}} \texttt{SUNOWY} \xrightarrow{\texttt{delete}} \texttt{SUNWY} \xrightarrow{\texttt{delete}} \texttt{SUNY} \xrightarrow{\texttt{insert}} \texttt{SUNNY}$

Distance 5

$\texttt{SNOWY} \xrightarrow{\texttt{insert}} \texttt{SUNOWY} \xrightarrow{\texttt{substitute}} \texttt{SUNNWY} \xrightarrow{\texttt{delete}} \texttt{SUNNY}$

Distance 3

Alignment of X and Y: Insert gaps to X and Y and align them.

- **A gap in** X corresponds to **an insertion of a letter to** X**.**
- **A gap in** Y corresponds to **a deletion of a letter from** X**.**
- **A column with two different letters** corresponds to **a substitution of a letter.**

-	S	N	O	W	-	Y
S	U	N	-	-	N	Y

Cost 5

S	-	N	O	W	Y
S	U	N	N	-	Y

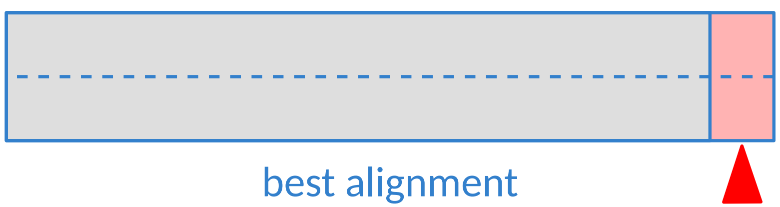
Cost 3

The cost of an alignment is the number of columns in which the letters differ.
The edit distance between two strings = cost of their best possible alignment.

Edit Distance

Given two strings, $X[1\cdots m]$ and $Y[1\cdots n]$,
a good subproblem would be to compute the optimal edit distance between some **prefixes**, $X[1\cdots i]$ and $Y[1\cdots j]$.

The best alignment for prefixes $X[1\cdots i]$ and $Y[1\cdots j]$ has **the last column**. If we remove the last column, the remaining columns must represent the best alignment for the remaining prefixes.



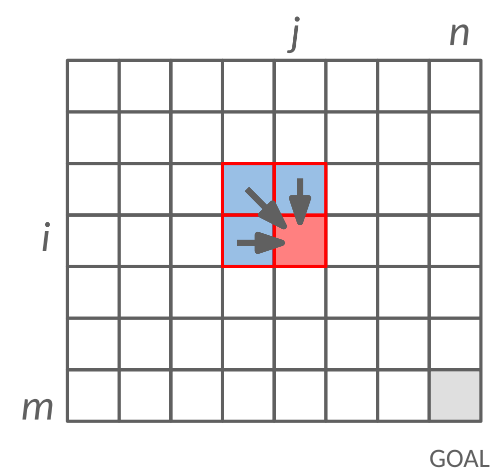
$\boxed{\begin{array}{c} X[i] \\ \text{-} \end{array}}$ or $\boxed{\begin{array}{c} \text{-} \\ Y[j] \end{array}}$ or $\boxed{\begin{array}{c} X[i] \\ Y[j] \end{array}}$

`delete` `insert` `substitute?`

Let $E(i,j)$ be the optimal edit distance between prefixes $X[1\cdots i]$ and $Y[1\cdots j]$.

$E(i,j) = i$ if $j = 0$.
$E(i,j) = j$ if $i = 0$.

$$E(i,j) = \min \begin{cases} E(i-1,j)+1 & \texttt{delete} \\ E(i,j-1)+1 & \texttt{insert} \\ E(i-1,j-1)+\text{diff}(i,j) & \texttt{substitute?} \end{cases}$$



for $i \leftarrow 0$ to m
  $E[i, 0] = i$
for $j \leftarrow 1$ to n
  $E[0, j] = j$

Time complexity: $O(mn)$
Space complexity: $O(mn)$

for $i \leftarrow 1$ to m
  for $j \leftarrow 1$ to n
    $E[i, j] = \min\{E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \text{diff}(i, j)\}$
return $E[m, n]$

E X P O N E N T I A L

P O L Y N O M I A L

Underlying DAG structure

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	6	7	8	9	10
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Once the table is filled, we can reconstruct the optimal alignment in $O(n + m)$ additional time by comparing the numerical values from $E[m, n]$.

Subset Sum

Input : A set X of n positive integers and a target integer T.
Goal : Decide ($\texttt{Yes}/\texttt{No}$) if there is a subset of elements in X that add up to T.

Backtracking. For the input array X, does any subset of $X[1 \dots i]$ sum to T?

$SS(i, T) = \texttt{True}$ iff some subset of $X[1 \dots i]$ sums to T.

```
SubsetSum(X, i, T)
if T = 0 then
    return True
else if T < 0 or i = 0 then
    return False
else
    with ← SubsetSum(X, i − 1, T − X[i])
    wout ← SubsetSum(X, i − 1, T)
    return (with ∨ wout)
```

Time complexity: $O(2^n)$
Space complexity: $O(n)$

Subset Sum

Input : A set X of n positive integers and a target integer T.
Goal : Decide (`Yes`/`No`) if there is a subset of elements in X that add up to T.

Dynamic programming. Memoize the recurrence into a 2D array $S[n, T]$, where $S[i, t]$ stores the value of $\mathrm{SS}(i, t)$.

$$\mathrm{SS}(i,t) = \begin{cases} \texttt{True} & \text{if } t = 0 \\ \texttt{False} & \text{if } i < 1 \\ \mathrm{SS}(i-1, t) & \text{if } t < X[i] \\ \mathrm{SS}(i-1,t) \vee \mathrm{SS}(i-1, t - X[i]) & \text{otherwise} \end{cases}$$

for $i \leftarrow 0$ to n
  $S[i, 0] = \texttt{True}$
for $t \leftarrow 1$ to T
  $S[0, t] = \texttt{False}$

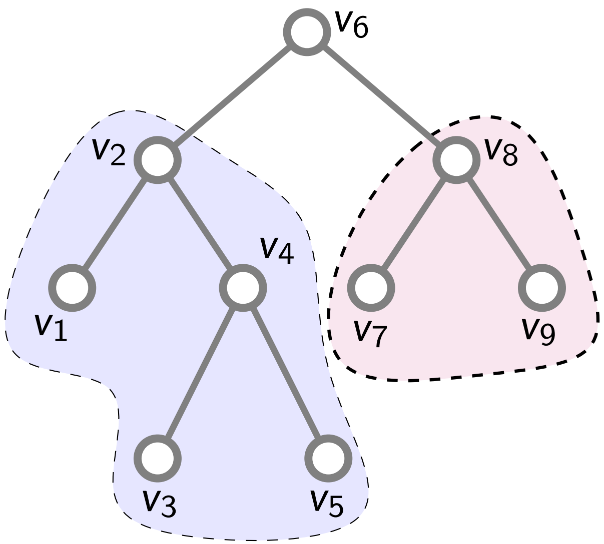
for $i \leftarrow 1$ to n
  for $t \leftarrow 1$ to $X[i] - 1$
    $S[i, t] = S[i-1, t]$
  for $t \leftarrow X[i]$ to T
    $S[i, t] = S[i-1, t] \vee S[i-1, t - X[i]]$
return $S[n, T]$

Time complexity: $O(nT)$. Space complexity: $O(nT)$

Optimal Binary Search Trees

Input : n keys $v_1, \dots, v_n$ and their frequencies $f[1], \dots, f[n]$ of searches.
Goal : A binary search tree of the keys with the minimum total cost of the searches.

$$\mathrm{Cost}(T, f[1 \dots n]) = \sum_{i=1}^{n} f[i] \cdot \mathrm{depth}(T, v_i)$$



If $f[i] > f[j]$, $\mathrm{depth}(v_i) \leq \mathrm{depth}(v_j)$ in any optimal binary search tree.

Suppose v_r is the root of the tree. Then

$$\mathrm{Cost}(T, f[1 \dots n]) = \sum_{i=1}^{n} f[i] + \boxed{\sum_{i=1}^{r-1} f[i] \cdot \mathrm{depth}(\mathrm{left}(T), v_i)} + \boxed{\sum_{i=r+1}^{n} f[i] \cdot \mathrm{depth}(\mathrm{right}(T), v_i)}$$

$$\mathrm{OPT}(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \leq r \leq k}\{\mathrm{OPT}(i, r-1) + \mathrm{OPT}(r+1, k)\} & \text{otherwise} \end{cases}$$

Optimal Binary Search Trees

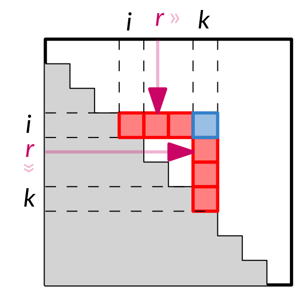
$$\text{OPT}(i,k) = \begin{cases} 0 & \text{if } i > k \\ \boxed{\sum_{j=i}^{k} f[j]} + \min_{i \le r \le k}\{\text{OPT}(i, r-1) + \text{OPT}(r+1, k)\} & \text{otherwise} \end{cases}$$

Let $F(i,k) = \sum_{j=i}^{k} f[j]$.

Then $F(i,k) = \begin{cases} f[i] & \text{if } i = k \\ F(i, k-1) + f[k] & \text{otherwise} \end{cases}$

for $i \leftarrow 1$ to n
  $F[i, i-1] = 0$
  for $k \leftarrow i$ to n
    $F[i,k] = F[i,k-1] + f[k]$

ComputeOPT(i, k)
Cost$[i,k] = \infty$
for $r \leftarrow i$ to k **do**
  if Cost$[i,k] >$ Cost$[i, r-1]$ + Cost$[r+1, k]$ **then**
    Cost$[i,k]$ = Cost$[i, r-1]$ + Cost$[r+1, k]$
Cost$[i,k]$ = Cost$[i,k]$ + $F[i,k]$



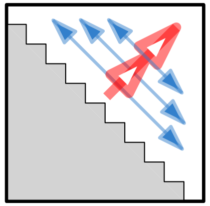
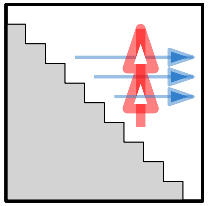
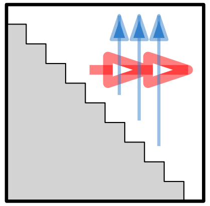
Optimal Binary Search Trees

$$\text{OPT}(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \le r \le k}\{\text{OPT}(i, r-1) + \text{OPT}(r+1, k)\} & \text{otherwise} \end{cases}$$

OptBST1($f[1 \dots n]$)
Compute $F[i,k]$ for all i, k
for $i \leftarrow 1$ to $n+1$ **do**
  $\text{Cost}[i, i-1] = 0$
for $d \leftarrow 0$ to $n-1$ **do**
  for $i \leftarrow 1$ to $n-d$ **do**
    ComputeOPT$(i, i+d)$
return Cost$[1, n]$

OptBST2($f[1 \dots n]$)
Compute $F[i,k]$ for all i, k
for $i \leftarrow n+1$ down to 1 **do**
  $\text{Cost}[i, i-1] = 0$
  for $j \leftarrow i$ to n **do**
    ComputeOPT(i, j)
return Cost$[1, n]$

OptBST3($f[1 \dots n]$)
Compute $F[i,k]$ for all i, k
for $j \leftarrow 0$ to $n+1$ **do**
  $\text{Cost}[j+1, j] = 0$
  for $i \leftarrow j$ down to 1 **do**
    ComputeOPT(i, j)
return Cost$[1, n]$

Time complexity: $O(n^3)$. Space complexity: $O(n^2)$.

Chain Matrix Multiplication

Suppose you want to multiply four matrices, $A \times B \times C \times D$.

$A : 50 \times 20 \qquad B : 20 \times 1 \qquad C : 1 \times 10 \qquad D : 10 \times 100.$

Matrix multiplication is not commutative but *associative*, so we can compute it in many different ways, depending on how we parenthesize it.

$A \times B$ with dimensions $A : m \times n$ and $B : n \times \ell$ costs $mn\ell$.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

Computing $A \times B \times C$ with minimum cost?

- $A : 2 \times 1, \quad B : 1 \times 2, \quad C : 2 \times 5.$
- $A : 1 \times 2, \quad B : 2 \times 3, \quad C : 3 \times 2.$

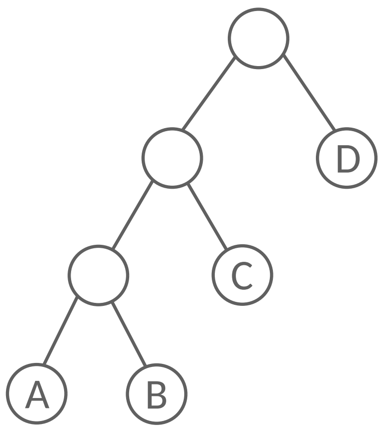
If we want to compute

$$A_1 \times A_2 \times \cdots \times A_n,$$

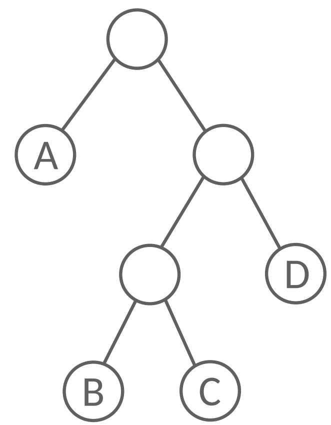
where A_i is a matrix with dimension $m_{i-1} \times m_i$,
how do we determine the optimal order of multiplications?

Chain Matrix Multiplication

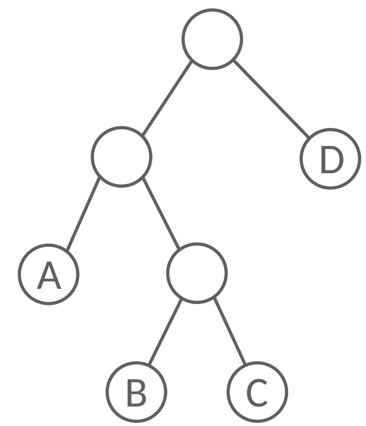
Parenthesization reminds us of the various full binary trees with n leaves, but there are exponentially many.



$((A \times B) \times C) \times D$



$A \times ((B \times C) \times D)$



$(A \times ((B \times C)) \times D$

For a tree to be optimal, its subtree must also be optimal!

Then what are the subproblems, with the properties - ordering and relation?

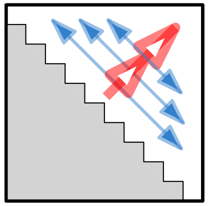
$$\text{Cost}(i, j) = \text{minimum cost of multiplying } A_i \times \cdots \times A_j.$$

$$\text{Cost}(i, j) = \min_{i \leqslant k < j} \{\text{Cost}(i, k) + \text{Cost}(k+1, j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

$$\text{Cost}(i,j) = \text{minimum cost of multiplying } A_i \times \cdots \times A_j.$$

$$\text{Cost}(i,j) = \min_{i \leqslant k < j} \{\text{Cost}(i,k) + \text{Cost}(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

for $\ell = 1, 2, \ldots, n$:
  for $i = 1, 2, \ldots, n - \ell$:
    $j = i + \ell$
    for $k = i, \ldots, j - 1$:
      cost $= C[i,k] + C[k+1,j] + m_{i-1}m_km_j$
      if cost $< C[i,j]$
        $C[i,j] =$ cost
        $\text{midx}(i,j) = k$



Time complexity: $O(n^3)$
Space complexity: $O(n^2)$

Input : A bag of capacity W and n items $(w_1, v_1), \ldots, (w_n, v_n)$.
Goal : An optimal selection of items.

For example, $W = 10$ and four items shown in the right table.

Unlimited quantities of each item available: $v_1 + 2v_4 = 47$
Only one of each item available: $v_1 + v_3 = 45$

Item	Weight	Value
1	6	\$29
2	3	\$14
3	4	\$16
4	2	\$9

Unlimited quantities of each item available:

Knapsack(w): max value achievable with capacity w.

Initialize $K[w] = 0$ for all $w = 0, \ldots, W$
for $w = 1$ to W
    $K[w] = \max_{1 \leqslant i \leqslant n}\{K[w - w_i] + v_i : w_i \leq w\}$
return $K[W]$

Time complexity: $O(Wn)$
Space complexity: $O(W + n)$

Knapsack

Input : A bag of capacity W and n items $(w_1, v_1), \ldots, (w_n, v_n)$.
Goal : An optimal selection of items.

For example, $W = 10$ and four items shown in the right table.

Unlimited quantities of each item available: $v_1 + 2v_4 = 47$

Only one of each item available: $v_1 + v_3 = 45$

Item	Weight	Value
1	6	\$29
2	3	\$14
3	4	\$16
4	2	\$9

Only one of each item available:

Knapsack(w, j): max value achievable with capacity w and items $1, \ldots, j$.

```
Initialize all K[0, j] = 0 and all K[w, 0] = 0
for j = 1 to n
    for w = 1 to W
        if w_j > w : K[w, j] = K[w, j − 1]
        else: K[w, j] = max{K[w, j − 1], K[w − w_j, j − 1] + v_j}
return K[W, n]
```

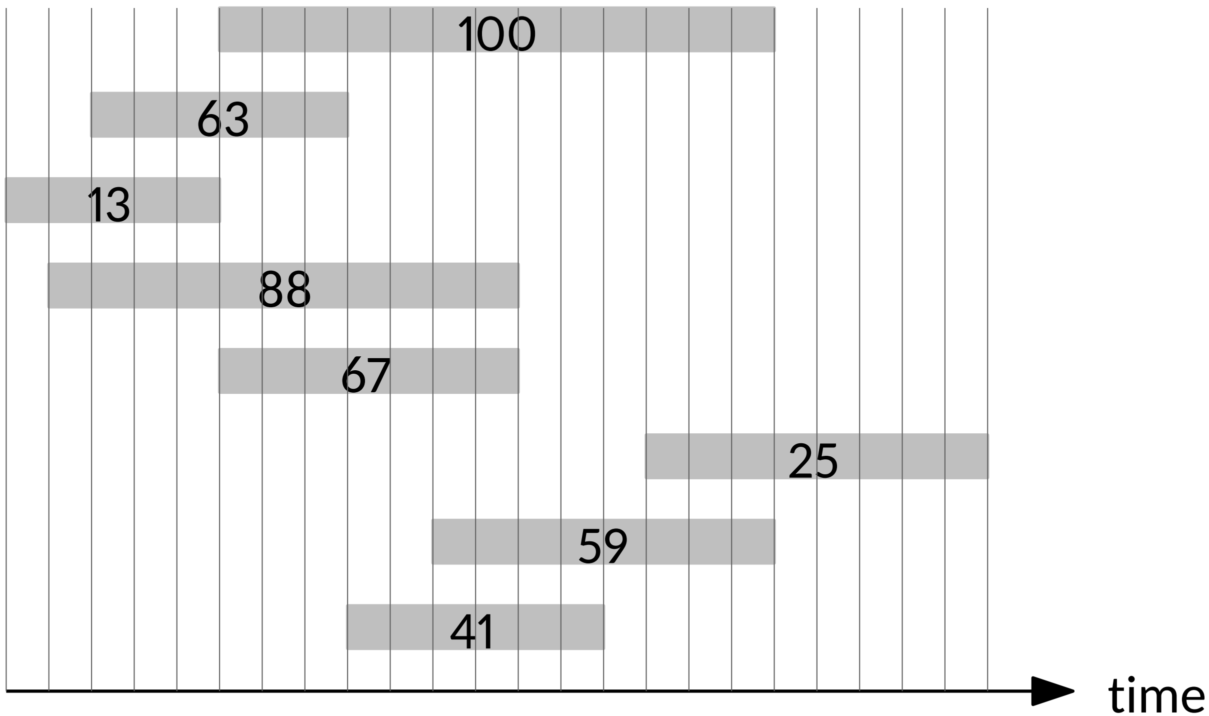
Time complexity: $O(Wn)$
Space complexity: $O(Wn)$

Weighted Interval Scheduling

Input : A set of n jobs, job j starts at s_j, finishes at f_j and has weight v_j.
Goal : A maximum weight subset of "mutually compatible" jobs.

Two jobs are compatible if they do not overlap.

Greedy algorithm works if all weights are the same. But may fail if arbitrary weights are allowed. Do you see why?



Interval Scheduling

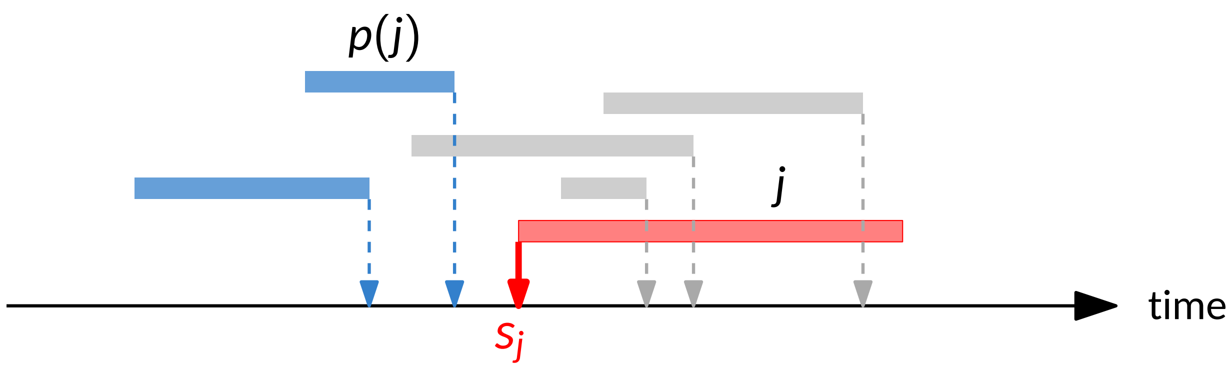
- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

Greedy template Consider jobs in some order. Take each job provided it is compatible with the ones already taken. Consider jobs

- (Earliest start time) in ascending order of start time s_j.
- (Earliest finish time) in ascending order of finish time f_j.
- (Shortest interval) in ascending order of interval length $f_j - s_j$.
- (Fewest conflicts) in ascending order of conflicts c_j.

Weighted Interval Scheduling

Sort and label jobs by their finish time: $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$.
Let $p(j)$ =last index $i < j$ s.t. job i is compatible with j.



The two jobs i and j ($i < j$) are compatible if and only if $f_i \leq s_j$.

Strategy : OPT(j) = Optimal solution to the subproblem of jobs 1, 2, ... , j.

- Case 1. OPT selects job j.
  We cannot use incompatible jobs.
  $\text{OPT}(j) = v_j + \text{OPT}(p(j))$.
- Case 2. OPT does not select job j.
  $\text{OPT}(j) = \text{OPT}(j-1)$.

$$\text{OPT}(j) = \max\{v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\}$$

Weighted Interval Scheduling

```
Sort the intervals by increasing order of finish time.
for j = 2 to n
    Compute p(j)
    tmp = v_j + OPT[p(j)]
    if tmp > OPT[j − 1] then
        OPT[j] = tmp
    else
        OPT[j] = OPT[j − 1]
return OPT[n]
```

$O(n \log n)$ time

$O(\log n)$ time
binary search with s_j on $f_1, \ldots, f_{j-1}$

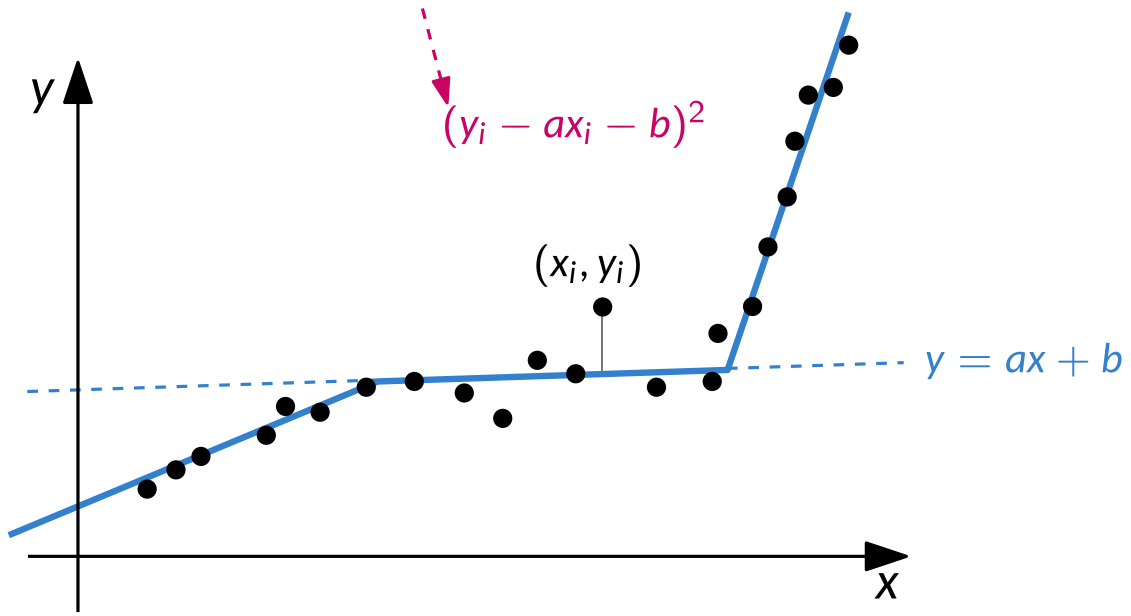
Time complexity: $O(n \log n)$
Space complexity: $O(n)$

Segmented Least Squares

Input : A set of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$ with $x_1 < \cdots < x_n$.
Goal : A polyline that minimizes

$$E + c \cdot L,$$

where E denotes the sum of sums of the squared errors in each segment, L denotes the number of segments, and c is a positive constant.

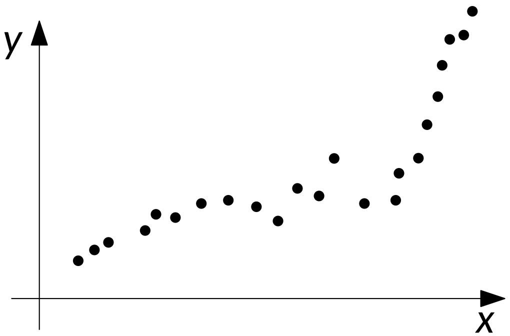


Input : A set of n points in the plane $(x_1, y_1), \dots, (x_n, y_n)$ with $x_1 < \cdots < x_n$.
Goal : A polyline that minimizes

$$E + c \cdot L,$$

where E denotes the sum of sums of the squared errors in each segment, L denotes the number of segments, and c is a positive constant.

There are exponentially many partitions of n points, but we want a polynomial number of subproblems...



Strategy : OPT(j) = minimum cost for points $p_1, p_2, \dots, p_j$.
Let $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \dots, p_j$ for a line.

$$\mathrm{OPT}(j) = \min_{1 \leqslant i \leqslant j}\{e(i,j) + c + \mathrm{OPT}(i-1)\}$$

Segmented Least Squares

Strategy : OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
Let $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$ for a line.

$$\mathrm{OPT}(j) = \min_{1 \leqslant i \leqslant j}\{e(i,j) + c + \mathrm{OPT}(i-1)\}$$

SegmentedLeastSquares(n)
Array $M[0 \ldots n]$
$M[0] = 0$
for all pairs (i,j) with $i \leq j$ **do**
  Compute $e(i,j)$
for $j \leftarrow 1$ to n **do**
  $M[j] = \min_{1 \leq i \leq j}\{e(i,j) + c + M[i-1]\}$
return $M[n]$

Once $e(i,j)$ values have been determined, the procedure takes $O(n^2)$ time.

$$e(i,j) = \sum_{k=i}^{j}(y_k - a_{ij}x_k - b_{ij})^2 \qquad p_i = (x_i, y_i)$$

$$a_{ij} = \frac{(j-i+1)\sum_{k=i}^{j} x_k y_k - (\sum_{k=i}^{j} x_k)(\sum_{k=i}^{j} y_k)}{(j-i+1)\sum_{k=i}^{j} x_k^2 - (\sum_{k=i}^{j} x_k)^2}$$

$$b_{ij} = \frac{\sum_{k=i}^{j} y_k - a_{ij}\sum_{k=i}^{j} x_k}{j-i+1}$$

The $e(i,j)$ value for each pair (i,j) can be computed in $O(j-i)$ time. Thus, $e(i,j)$ values for all pairs can be computed in $O(n^3)$ time.

Remark Can be improved to $O(n^2)$ time.

- For each i, precompute cumulative sums $\sum_{k=1}^{i} x_k, \sum_{k=1}^{i} y_k, \sum_{k=1}^{i} x_k^2, \sum_{k=1}^{i} x_k y_k$.
- Using the cumulative sums, we can compute $e(i,j)$ in $O(1)$ time.

Segmented Least Squares

Strategy : OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
Let $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$ for a line.

$$\text{OPT}(j) = \min_{1 \leqslant i \leqslant j}\{e(i,j) + c + \text{OPT}(i-1)\}$$

FindSegments(j)

if $j = 0$ **then**
  Output nothing
else
  Find an integer i minimizing $e(i,j) + c + M[i-1]$
  Output the segment for $p_i, \ldots, p_j$
  FindSegments($i-1$)