CS 350 Algorithms and Complexity

Winter 2019

Lecture 13: Dynamic Programming

Andrew P. Black

Department of Computer Science

Portland State University

Dynamic programming:

- Solves problems by breaking them into smaller sub-problems and solving those.
- Which algorithm design technique is this like?
  - Brute force
  - Decrease-and-conquer
  - Divide-and-conquer
  - None that we've seen so far

Question:

- Compare Dynamic Programming with Decrease-and-Conquer:

A. They are the same
B. They both solve a large problem by first solving a smaller problem
C. In decrease and conquer, we *don't* "memoize" the solutions to the smaller problem
D. In dynamic programming, we *do* "memoize" the smaller problems
E. B, C & D
F. B & C
G. B & D

Dynamic Programming

- Dynamic programming differs from decrease-and-conquer because in dynamic programming we remember the answers to the smaller sub-problems.
- Why?

A. To use more space

B. In the hope that we might re-use them

C. Because we *know* that the sub-problems overlap

Dynamic programming:

- Solves problems by breaking them into smaller sub-problems and solving those.
  - like: decrease & conquer
- Key idea: do not compute the solution to any sub-problem more than once;
  - instead: save computed solutions in a table so that they can be reused.
- Consequently: dynamic programming works well when the sub-problems overlap.
  - unlike: decrease & conquer

Why Dynamic Programming?

- *If* the subproblems are *not* independent, i.e. subproblems share sub-subproblems,
  - then a decrease and conquer algorithm repeatedly solves the common sub-subproblems.
  - thus: it does more work than necessary
- The “memo table” in DP ensures that each sub-problem is solved (at most) once.

- For dynamic programming to be applicable:
  - At most polynomial-number of subproblems
    - otherwise: still exponential
  - Solution to original problem is easy to compute from solutions to subproblems
  - Natural ordering on subproblems from “smallest” to “largest”
  - An easy-to-compute recurrence that allows solving a larger subproblem from a smaller subproblem

Optimization problems:

- Dynamic programming is typically (but not always) applied to *optimization problems*
  - In an optimization problem, the goal is to find a solution among many possible candidates that *minimizes* or *maximizes* some particular value.
- Such solutions are said to be <u>optimal</u>.

Example: Fibonacci Numbers

- The familiar recursive definition:

  fib 0 = 0

  fib 1 = 1

  fib (n+2) = fib (n+1) + fib n

- Grows very rapidly:

  0,1,1,2,3,5,8,13,21,34,55,89,144, …

  832040 (30th), …

  354224848179261915075 (100th), ...

Question

- What is the order of growth of the Fibonacci function?

A. $O(n)$

B. $O(n^2)$

C. $O(1.61803\ldots^n)$

D. $O(2^n)$

E. $O(e^n)$

F. $O(n!)$

- In fact, the i^{th} Fibonacci number is the integer closest to

$$\varphi^i/\sqrt{5}$$

where:

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots$$

(the “golden ratio”)

- Thus, the result of the Fibonacci function grows exponentially.

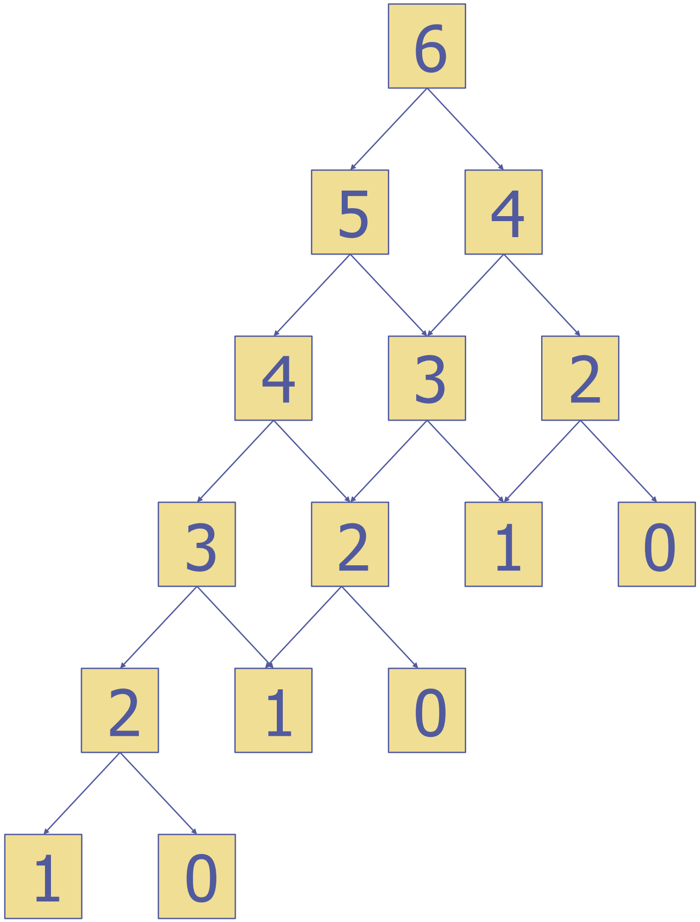
Complexity of brute-force *fib*:

let *nfib* be the number of calls needed to evaluate *fib n*, implemented according to the definition.

```
nfib 0     = 1
nfib 1     = 1
nfib(n+2)= 1 + nfib (n+1) + nfib n
```

- Grows even more rapidly than *fib*!
    - Hence *fib* is *at least* exponential ☹
- However: many calls to *fib* have the same argument …

Repeated calls, same argument:



Avoiding repeated calculations:

- We can use a table to avoid doing a calculation more than once:

```
table[0] ← 0;
table[1] ← 1;
table[2..max] ← -1;
int tableFib(int n) {
  if (table[n] = -1) {
      table[n] ← tableFib(n-1) + tableFib(n-2);
  }
  return table[n];
}
```

all other entries in the table are initially set to -1.

- Table size is fixed, but values can be shared over many calls.

Riding the wave:

- Alternatively, we can look at the way the entries in the table are filled:

0	1	1	2	3	5	8	13	21	34	55	89	...

This leads to code:

```
int a ← 0, b ← 1;
for i from 0 to n do {
    int c = a + b;
    a ← b;
    b ← c;
}
return a;
```

Complexity is O(n)!

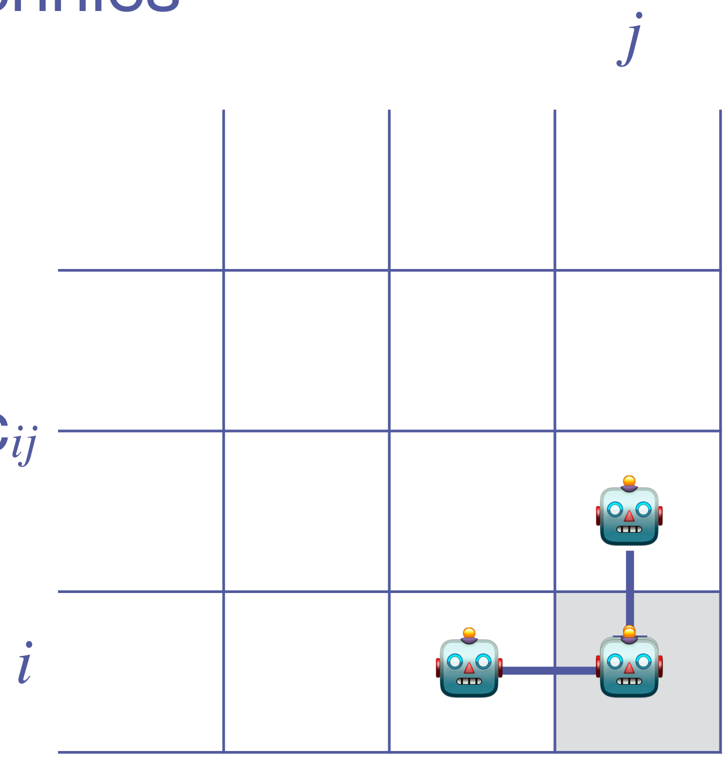
No limits on n now, but values cannot be reused.

Coin-collecting Problem

- Arrive at bottom-right with max number of pennies
- Robot can move right, or down
- Starts at top-left

  square (i, j) contains c_{ij}

- How can robot reach bottom-right?

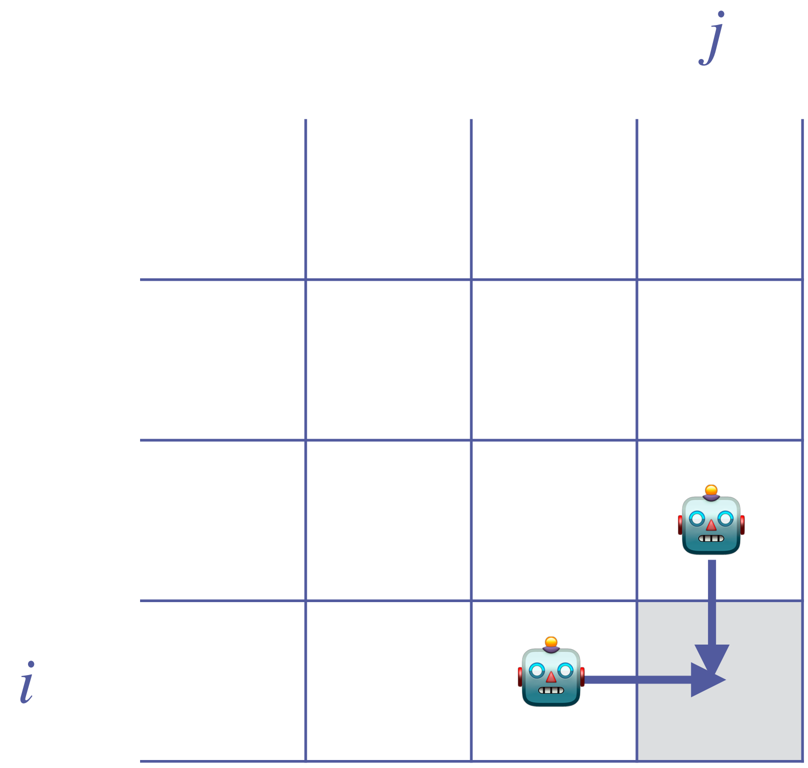


Coin-collecting Problem

- Either from above, or from left
- How many pennies can it bring?
- If from above:

  P(i-1 , j)
- If from left:

  P(i, j-1)



- Hence: P(i, j) = max(P(i-1 , j), P(i , j-1)) + c_{ij}

Example Problem

- You fill in the table

	1	2	3	4	5
1		●			
2	●				●●
3	●	●			
4		●		●	
5					

	0	0	0	0	0
0					
0					
0					
0					
0					

Quiz Question

- Can you fill in the table for the coin-collecting problem by rows, starting at the top-left?

  - A. Yes
  - B. No

Quiz Question

- Can you fill in the table for the coin-collecting problem by columns, starting at the left-top?

  - A. Yes
  - B. No

Quiz Question

- Can you fill in the table for the coin-collecting problem by rows, starting at the bottom-left?

  - A. Yes
  - B. No

Quiz Question

- Can you fill in the table for the coin-collecting problem by columns, starting at the top-right?
  - A. Yes
  - B. No

Discussion Question

- What constraints are there on filling in the rows and columns?
- Can we do this “top down” rather than “bottom up”?

Knapsack Problem by DP

- Given n items of

  integer weights: w_1 w_2 … w_n

  values: v_1 v_2 … v_n

  a knapsack of integer capacity W

  find most valuable subset of the items that fit into the knapsack

- How can we set this up as a recursion over smaller subproblems?

Knapsack Problem by DP

Consider problem instance defined by first i items and capacity j ($j \leq W$).

Let $V[i, j]$ be value of optimal solution of this problem instance. Then

$$V[i,j] = \begin{cases} \max\left(V[i\text{-}1, j],\ v_i + V[i\text{-}1, j\text{-}w_i]\right) & \text{if } j \geq w_i \\ V[i\text{-}1, j] & \text{if } j < w_i \end{cases}$$

Initial conditions: $V[0, j] = 0$ and $V[i, 0] = 0$

Knapsack Problem by DP (example)

Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

$$V[i,j] = \begin{cases} \max(V[i-1, j],\ v_i + V[i-1, j-w_i]) & \text{if } j \geq w_i \\ V[i-1, j] & \text{if } j < w_i \end{cases}$$

capacity, j

	i	0	1	2	3	4	5
	0						
$w_1 = 2, v_1 = 12$	1						
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

Can we do this “top down” ?

- Yes: use a memo function
  - Not: a “memory function”
  - D. Michie. “Memo” functions and machine learning. *Nature*, 218:19–22, 6 April 1968.
- Idea: record previously computed values “just in time”

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table V[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if V[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    V[i, j] ← value
return V[i, j]
```

Summary

- Dynamic programming is a good technique to use when:
  - Solutions defined in terms of solutions to smaller problems of the same type.
  - Many overlapping subproblems.

- Implementation can use *either:*
  - top-down, recursive definition with *memoization*
  - explicit bottom-up tabulation

Problem

a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

b. How many different optimal subsets does the instance of part (a) have?

c. In general, how can we use the table generated by the dynamic programming algorithm to tell whether there is more than one optimal subset for the knapsack problem's instance?

Problem:

- The sequence of values in a *row* of the dynamic programming table for an instance of the knapsack problem is always non-decreasing:
  - True or False?

Problem:

- The sequence of values in a *column* of the dynamic programming table for an instance of the knapsack problem is always non-decreasing:
  - True or False?

Problem

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

Apply the memo function method to the above instance of the knapsack problem. Which entries of the dynamic programming table are (i) *never* computed by the memo function, and (ii) retrieved without recomputation.

Solution

In the table below, the cells marked by a minus indicate the ones for which no entry is computed for the instance in question; the only nontrivial entry that is retrieved without recomputation is $(2, 1)$.

	i	capacity j 0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3,\ v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2,\ v_2 = 20$	2	0	0	20	-	-	45	45
$w_3 = 1,\ v_3 = 15$	3	0	15	20	-	-	-	60
$w_4 = 4,\ v_4 = 40$	4	0	15	-	-	-	-	60
$w_5 = 5,\ v_5 = 50$	5	0	-	-	-	-	-	65

Warshall's Algorithm

- Computes the transitive closure of a relation.
  - reachability in a graph is only an example of such a relation …

Warshall's Algorithm

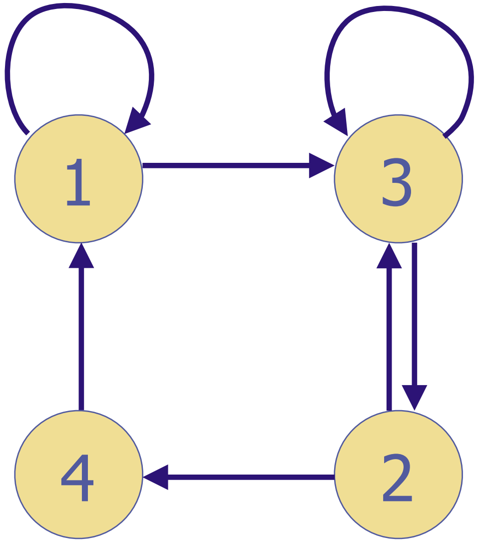
Warshall Algorithm 1

```
Warshall(M_R: n × n 0-1 matrix)
W := M_R (W = [w_ij])
for(k=1 to n) {
    for(i=1 to n) {
        for(j=1 to n) {
            w_ij = w_ij ∨ (w_ik ∧ w_kj)
            }
        }
    }
return W
```

Warshall Algorithm 2

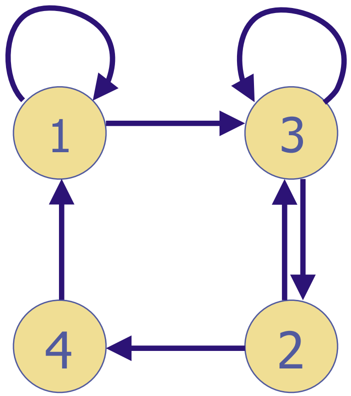
```
Warshall(M_R: n × n 0-1 matrix)
W := M_R (W = [w_ij])
for(k=1 to n){
    for(i=1 to n) {
        if(w_ik=1) {
            for(j=1 to n) {
                w_ij = w_ij ∨ w_kj)
                }
            }
        }
    }
return W
```

Example



$$M_R = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$W_0 = M_R =$ direct connections between nodes



$W_0 = M_R =$ direct connections between nodes

$W_1 = W_0 +$ connections thru node 1

$$M_R = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$w_{ij} \Leftarrow w_{ij} \vee (w_{i1} \wedge w_{1j})$$

$$w_{ij} \Leftarrow w_{ij} \vee (w_{i2} \wedge w_{2j})$$

add new 1 when $i = 4$ and $j = 3$

$$W_3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Warshall Algorithm 1

Warshall(M_R: $n \times n$ 0-1 matrix)
  $W := M_R$ ($W = [w_{ij}]$)
  for(k=1 to n) {
    for(i=1 to n) {
      for(j=1 to n) {
        $w_{ij} = w_{ij} \vee (w_{ik} \wedge w_{kj})$
      }
    }
  }
  return W

Floyd's Algorithm

- the all-pairs shortest-paths problem: generate the matrix that contains as element (i,j) the shortest path from vertex i to vertex j in a known graph

Floyd's Algorithm

- What's the recurrence?
- generate a series of distance matrices:

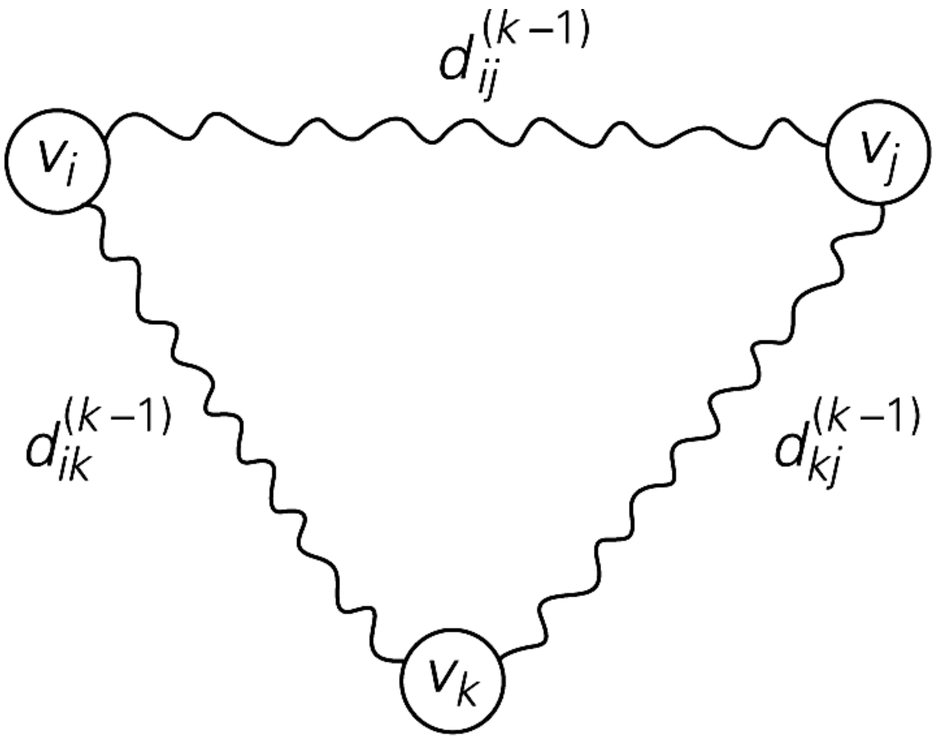
  $$D^{(0)}, D^{(1)}, ..., D^{(k)}, ..., D^{(n)}$$

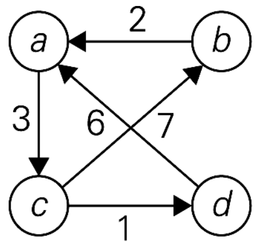
  where no path in $D^{(k)}$ uses an intermediate vertex with index greater than k
- $D^{(0)}$ is just the distance matrix of the graph

- Basic idea:
  - shortest path from i to j :

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)},\quad d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}, \qquad \text{for } k \geq 1,$$

$$d_{ij}^{(0)} = w_{ij}$$




a
b
c
d
2
3
6
7
1

Floyd's Algorithm Example

Solve the all-pairs shortest path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Solution

Applying Floyd's algorithm to the given weight matrix generates the following sequence of matrices:

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \mathbf{14} \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \mathbf{5} & \infty & \mathbf{4} & 0 \end{bmatrix}$$

Sequence Alignment

- In genetics, sequence alignment is the process of converting one gene-sequence into another at minimal cost
  - operations:
    - replace an element
    - remove an element
    - insert an element
  - What's the minimum edit distance between two sequences?

- A can be optimally edited into B by

1. insert first element of B, and optimally aligning A into tail of B, or
2. delete first element of A, and optimally aligning the tail of A and B, or
3. replacing the first element of A with the first element of B, and optimally aligning the tails of A and B

- Build matrix H, where H_{ij} is cost of aligning A[1..i] with B[1..j]

Sequence A = ACACACTA

Sequence B = AGCACACA

$$w(a,-) = w(-,b) = -1$$

$$w(\text{mismatch}) = -1$$

$$w(\text{match}) = +2$$

B

$$H(i,j) = \max \begin{Bmatrix} 0 \\ H(i-1,j-1) + \ w(a_i,b_j) \\ H(i-1,j) + \ w(a_i,-) \\ H(i,j-1) + \ w(-,b_j) \end{Bmatrix}$$

Deletion

Insertion

A

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{pmatrix}$$