Dynamic programming paradigm

- It is very similar to the recursive paradigm.

  Except the implementation is done in an iterative way. Otherwise, there may be exponentially many base cases in the recursion tree!

- This is best seen in examples:

Eg. 1. Longest common subsequence (LCS)

Defn: An array $C[\cdot]$ is a subsequence of an array $A[\cdot]$ if we can get C by removing some elements from A.

- Eg. A: a b a d e b c a

  C: a a e b a

$\{a_1, \dots, a_n\}$ $\{b_1, \dots, b_m\}$

LCS Input: Sequences $A[1\ldots n]$ & $B[1\ldots m]$.

Output: A sequence C s.t.

- C is a subsequence of A & B.
- C is the longest.

- Brute-force: The possibilities of C is $\min(2^n, 2^m)$.

Could you use recursive or The greedy paradigms?

(Exercise)

- Let's focus on the last element of C:

Observation: If $a_n = b_m$ then this element will be the last element in any LCS C.

Pf:

- Suppose $a_n = b_m$ & C' is a subsequence with the last element $\neq a_n$.
- We can consider $C' \cup \{a_n\}$. It is a subsequence of both A, B & is longer.

(Note: C' is infact a subsequence of $A \setminus \{a_n\}$ & $B \setminus \{b_m\}$.) $\square$

- Now, to attempt a recursive formulation let us define:

$LCS(i,j) :=$ an LCS of $A[1\ldots i]$ & $B[1\ldots j]$.

concatenation

▷ $a_n = b_m \Rightarrow LCS(n,m) = LCS(n-1,m-1) \circ a_n$.

<u>Observation</u>: If $a_n \neq b_m$ then either a_n or b_m is not the last element in LCS.

▷ So, in that case, we should pick the longer of $LCS(n-1,m)$ & $LCS(n,m-1)$.

▷ Base case: $LCS(i,0) = LCS(0,j) = \phi$.

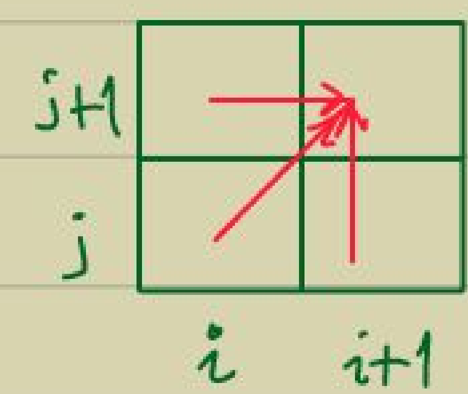
- <u>Qn</u>: What happens if you implement this in a recursive program?

- The time $T(n,m)$ may grow like $T(n-1,m) + T(n,m-1)$ which gives you $\min(2^n, 2^m)$.

- Is this exponential blowup avoidable?

- Yes: Iteratively compute $LCS(i,j)$ & then use it to solve the bigger cases $LCS(i+1,j)$, $LCS(i,j+1)$ or $LCS(i+1,j+1)$.

▷ See the recursive formulation as filling up an $n \times m$ matrix with (gradually) growing subsequences!



<u>Theorem</u>: LCS is computable in $O(nm)$ time.

<u>Pf</u>:

- As sketched above, the dynamic programming based algorithm is:

```
LCS(A[1...n], B[1...m]) {
    for(i=0 to n) LCS(i,0) ← φ;
    for(j=0 to m) LCS(0,j) ← φ;
```

.... (contd.)

```
for (i=1 to n)
    for (j=1 to m) {
        if (a_i = b_j)  LCS(i,j) ← LCS(i-1,j-1) ∘ a_i ;
        else {
            l_1 ← LCS(i-1, j) ;
            l_2 ← LCS(i, j-1) ;
            LCS(i,j) ← longer among l_1, l_2 ;
        }
    }//end for
}//end L(n,m)
```

$\square$

- Crucial steps in <u>dynamic programming</u>:

Recursive formulation

↓

Recursive algorithm

↓

Exponential time → Cause: Overlapping subproblems

↓

Polynomially many distinct subproblems

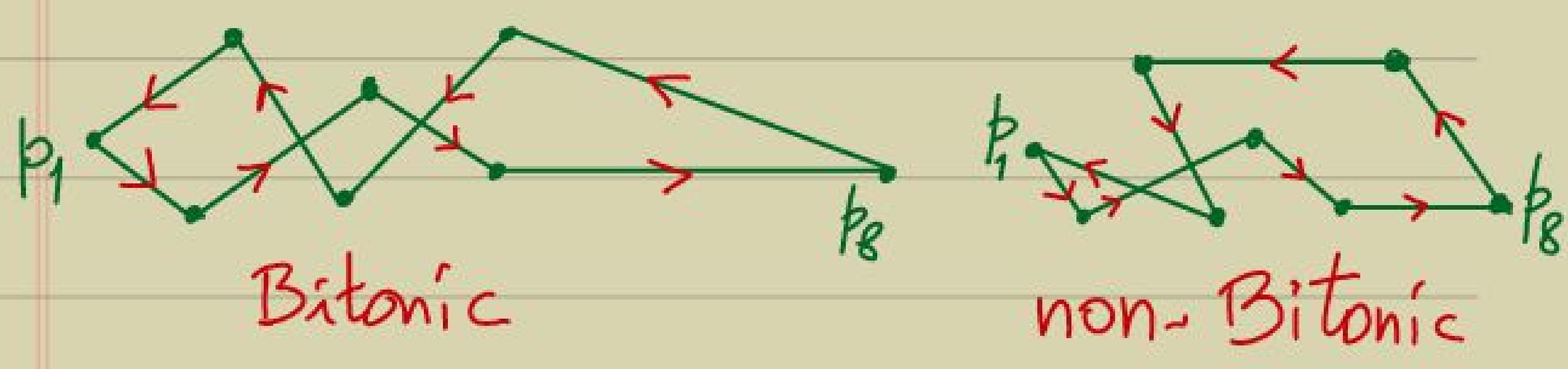
↓

Bottom-up iterative algorithm

Eg. 2. Optimal bitonic tour

- Input: Given n points $S \subset \mathbb{R}^2$ in the increasing order of x-coordinate. Let $S = \{p_1, \ldots, p_n\}$ & $\delta(p_i, p_j)$ be the distance.

- Output: A shortest bitonic tour, i.e. a tour where the x-coordinates monotonically increase first & later monotonically decrease.



▷ The tour has to cover each vertex exactly once.

- Idea 1: Shortest bitonic tour gives two disjoint paths from p_n to p_1.

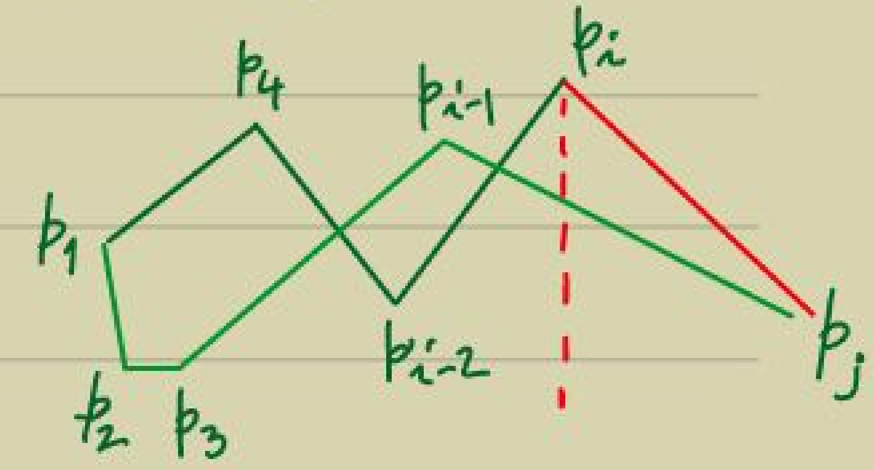
- Issue: This is not true for p_{n-1}!

- How do we get a recursive formulation? We should work with p_n & another vertex!

- Idea 2: Compute the least distance from p_i to p_1 & from p_j to p_1, along disjoint paths.

  Further, if $x(p_i) < x(p_j)$ then the two paths should cover $\{p_2, \ldots, p_{i-1}\}$.

  $\Rightarrow$ This will help in getting a bitonic tour on $\{p_1, p_2, \ldots, p_i\} \cup \{p_j\}$.



- Defn: For $i < j \in [n]$, define $T[i,j]$ to be the least distance travelled from p_i & p_j to p_1, using $\{p_2, \ldots, p_{i-1}\}$ exactly once & disjoint paths.

  $\triangleright$ (p_{n-1}, p_n) is an edge in any tour.

Aim: To compute $T[n-1, n]$.

▷ Least bitonic tour is $T[n-1,n] + \delta(p_{n-1}, p_n)$.

▷ Base case: $T[1,j] = \delta(p_1, p_j)$. $p_1 \leftarrow p_j$

- What is $T[i,j]$ for $1 < i < j \leq n$?

▷ p_{i-1} appears either in the path $p_i \rightsquigarrow p_1$ or the path $p_j \rightsquigarrow p_1$.

- Based on this we get the recurrence:

▷ $$T[i,j] = \min\big(T[i-1,j] + \delta(p_{i-1}, p_i),\; T[i-1,i] + \delta(p_{i-1}, p_j) \big).$$

- Again, a naive recursive implementation would take time 2^n.

- Instead, we should maintain an $[n-1] \times [2 \cdots n]$ matrix T with (i,j)-th entry $T[i,j]$.

- Matrix $\mathcal{T}$ can be filled bottom-up (iteratively) in time $O(n^2)$.

<u>Theorem</u> (Bentley 1990): Optimal bitonic tour is computable in $O(n^2)$ time.

Dynamic programming <u>properties</u>

- Problem has the <u>substructure</u> property. This is shared by greedy & recursive paradigms.

- Sometimes <u>generalizing</u> a problem helps.

- Coming up with the right recursive <u>formulation</u> may be nontrivial.