Dynamic Programming

Textbook reading

Chapter 6
1. Analyzing Algorithms, Big-O
   proof of correctness
   termination
   there is an answer

2. Graph Algorithms

3. Greedy Algorithms
   what is the best choice? do it!

4. Divide and Conquer
   divide - break into subproblems
   recurse
   conquer / combine - combine the solutions to solve the larger problems
Overview

**Design principle:**
- Make a table of all possible subproblems and combine them bottom-up

**Important tool:**
- Recurrence relations

**Problems:**
- Weighted interval scheduling
- RNA sequence secondary structure
- Sequence alignment
- Shortest paths
Problem 1: Weighted Interval Scheduling

**Given:** Set of activities competing for time intervals on a given resource.

**Goal:** Schedule non-conflicting activities so that the total time the resource is used is maximized.
Problem 2: RNA Sequence Secondary Structure (1)

DNA:

- Two RNA strands, that is, sequences of bases from the set \{A, C, G, T\}.
- Bases on the two strands are matched in pairs A-T and C-G.
**Problem 2: RNA Sequence Secondary Structure (2)**

**RNA secondary structure:**
- Single RNA strands tend to loop back and form base pairs between bases in the strand.

- Understanding this structure is important for understanding the behaviour of cells.
The pairing of an RNA strand $B = b_1 b_2 \ldots b_n$ approximately adheres to the following rules:

- **Proper pairing**: Only pairs $A-U$ and $C-G$ are allowed and every base is allowed to participate in at most one pair.
Problem 2: RNA Sequence Secondary Structure (3)

The pairing of an RNA strand $B = b_1 b_2 \ldots b_n$ approximately adheres to the following rules:

- **Proper pairing**: Only pairs $A-U$ and $C-G$ are allowed and every base is allowed to participate in at most one pair.

- **No sharp turns**: If $(b_i, b_j)$ is a pair, then $i < j - 4$. 

No sharp turns
The pairing of an RNA strand \( B = b_1 b_2 \ldots b_n \) approximately adheres to the following rules:

- **Proper pairing:** Only pairs \( A-U \) and \( C-G \) are allowed and every base is allowed to participate in at most one pair.

- **No sharp turns:** If \((b_i, b_j)\) is a pair, then \(i < j - 4\).

- **No crossings:** If \((b_i, b_j)\) and \((b_k, b_l)\) are two pairs with \(i < k\), we cannot have \(i < k < j < l\).
The pairing of an RNA strand \( B = b_1 b_2 \ldots b_n \) approximately adheres to the following rules:

- **Proper pairing:** Only pairs \( A-U \) and \( C-G \) are allowed and every base is allowed to participate in at most one pair.

- **No sharp turns:** If \((b_i, b_j)\) is a pair, then \(i < j - 4\).

- **No crossings:** If \((b_i, b_j)\) and \((b_k, b_l)\) are two pairs with \(i < k\), we cannot have \(i < k < j < l\).

- **Maximal number of pairs:** There are as many pairs as possible, subject to the above rules.
The pairing of an RNA strand $B = b_1b_2\ldots b_n$ approximately adheres to the following rules:

- **Proper pairing:** Only pairs $A-U$ and $C-G$ are allowed and every base is allowed to participate in at most one pair.

- **No sharp turns:** If $(b_i, b_j)$ is a pair, then $i < j - 4$.

- **No crossings:** If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs with $i < k$, we cannot have $i < k < j < l$.

- **Maximal number of pairs:** There are as many pairs as possible, subject to the above rules.

**Goal:** Predict the secondary structure of a given RNA sequence $B = b_1b_2\ldots b_n$. 
When you type “Dalhusy Computer Science” into Google, you get the question

Did you mean Dalhousie Computer Science?
Problem 3: Sequence Alignment (1)

When you type “Dalhusy Computer Science” into Google, you get the question

Did you mean Dalhousie Computer Science?

Can Google read your mind?
Problem 3: Sequence Alignment (1)

When you type “Dalhusy Computer Science” into Google, you get the question

Did you mean Dalhousie Computer Science?

Can Google read your mind? No.

- They use a clever algorithm to match your mistyped query against the information they do have in their database.
- “Dalhousie” is the closest match to “Dalhusy” in their database.
Problem 3: Sequence Alignment (1)

When you type “Dalhusy Computer Science” into Google, you get the question

Did you mean Dalhousie Computer Science?

Can Google read your mind? **No.**

- They use a clever algorithm to match your mistyped query against the information they do have in their database.
- “Dalhousie” is the closest match to “Dalhusy” in their database.

*But what’s a good similarity criterion?*
Problem 3: Sequence Alignment (2)

**Problem:** Extend strings $X$ and $Y$ to the same length by inserting gaps so that the following dissimilarity measure is minimized:

- **Gap penalty** $\delta > 0$.
- **Mismatch penalty** $p_{ab}$, for every pair $(a, b)$ of letters in the alphabet. (Assume that $p_{aa} = 0$, for all $a$.)
Problem: Extend strings $X$ and $Y$ to the same length by inserting gaps so that the following dissimilarity measure is minimized:

- **Gap penalty** $\delta > 0$.
- **Mismatch penalty** $p_{ab}$, for every pair $(a, b)$ of letters in the alphabet. (Assume that $p_{aa} = 0$, for all $a$.)

**Example:**

\[
\begin{align*}
\text{Dalh-u}s\text{-y} & \quad \text{Dalh-usy}\text{-e} & \quad \text{Dalh-u}s\text{-y} \\
\text{Dill hou}s\text{e} & \quad \text{Dalhousie} & \quad \text{Dalhousie} \\
\end{align*}
\]

Cost: $2\delta + p_{iy}$
Problem 3: Sequence Alignment (2)

Problem: Extend strings $X$ and $Y$ to the same length by inserting gaps so that the following dissimilarity measure is minimized:

- **Gap penalty** $\delta > 0$.
- **Mismatch penalty** $p_{ab}$, for every pair $(a, b)$ of letters in the alphabet. (Assume that $p_{aa} = 0$, for all $a$.)

Example:

```
Dalh-usy-
Dalhousie
```

Cost: $2\delta + p_{iy}$

Another (more important?) application:

- DNA sequence alignment to measure similarity between different DNA samples.
Dijkstra’s algorithm may fail in the presence of negative-weight edges:
Dijkstra’s algorithm may fail in the presence of negative-weight edges:

We need an algorithm that can deal with negative edge weights.
Problem 5: All-Pairs Shortest Paths

In some applications, it is necessary to compute the distances between all vertices in a graph.

Example:

```
In the example graph, the distances between all vertices are computed. The distances are as follows:

- a to b: 3
- a to c: 8
- a to d: 6
- a to e: 1
- b to a: 3
- b to c: 3
- b to d: 5
- b to e: 2
- c to a: 4
- c to b: 3
- c to d: 2
- c to e: 3
- d to a: 1
- d to b: 2
- d to c: 2
- d to e: 0
- e to a: 3
- e to b: 3
- e to c: 3
- e to d: 1
- e to e: 0

The time complexity for computing all pairs of shortest paths is \(O(n^3)\) with the Floyd-Warshall algorithm.
```
Problem 1: Weighted Interval Scheduling

**Given:** Set of activities competing for time intervals on a given resource.

**Goal:** Schedule non-conflicting activities so that the total time the resource is used is maximized.
**Problem 1: Weighted Interval Scheduling**

**Given:** Set of activities competing for time intervals on a given resource.

**Goal:** Schedule non-conflicting activities so that the total time the resource is used is maximized.
A Naive Solution to Interval Scheduling

**Naive algorithm:**

- Try all possible subsets.
- Check each subset for conflicts.
- Out of the non-conflicting ones, remember the one with maximal total length.
Naive algorithm:

■ Try all possible subsets.
■ Check each subset for conflicts.
■ Out of the non-conflicting ones, remember the one with maximal total length.

Cost:
A Naive Solution to Interval Scheduling

**Naive algorithm:**
- Try all possible subsets.
- Check each subset for conflicts.
- Out of the non-conflicting ones, remember the one with maximal total length.

**Cost:** $O(n \cdot 2^n)$
Can we be greedy?

1) pick the longest interval
Interval Scheduling: Problem Analysis

Approach:

- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?
Interval Scheduling: Problem Analysis

**Approach:**

- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?

**What are the choices we make?**
Interval Scheduling: Problem Analysis

**Approach:**
- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?

**What are the choices we make?**
- An interval can be in the solution or not.
Interval Scheduling: Problem Analysis

Approach:

- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?

What are the choices we make?

- An interval can be in the solution or not.

Toward a recurrence:
Interval Scheduling: Problem Analysis

Approach:
- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?

What are the choices we make?
- An interval can be in the solution or not.

Toward a recurrence:
- If the maximal-length subset of \( \{I_1, I_2, \ldots, I_n\} \) does not include \( I_n \), then it must be a maximal-length subset of \( \{I_1, I_2, \ldots, I_{n-1}\} \).
Interval Scheduling: Problem Analysis

**Approach:**
- As in a greedy algorithm, try to make a choice at a time.
- What are the choices we make?
- What can we say about the subproblem we obtain after making a certain choice?

**What are the choices we make?**
- An interval can be in the solution or not.

**Toward a recurrence:**
- If the maximal-length subset of \( \{I_1, I_2, \ldots, I_n\} \) does not include \( I_n \), then it must be a maximal-length subset of \( \{I_1, I_2, \ldots, I_{n-1}\} \).
- If the maximal-length subset of \( \{I_1, I_2, \ldots, I_n\} \) includes \( I_n \), then it must be \( \mathcal{O} \cup \{I_n\} \), where \( \mathcal{O} \) is a maximal-length subset of all intervals in \( \{I_1, I_2, \ldots, I_{n-1}\} \) that do not conflict with \( I_n \).
Number the intervals $I_1, I_2, \ldots, I_n$ by increasing finish time.
Cleaning Up the Modelling

Number the intervals \( I_1, I_2, \ldots, I_n \) by increasing finish time.

For \( 1 \leq j \leq n \), let

\[
p_j = \max(\{0\} \cup \{k \mid 1 \leq j < k \text{ and } I_k \text{ does not conflict with } I_j\}).
\]
Cleaning Up the Modelling

Number the intervals $I_1, I_2, \ldots, I_n$ by increasing finish time.

For $1 \leq j \leq n$, let

$$p_j = \max(\{0\} \cup \{k \mid 1 \leq j < k \text{ and } I_k \text{ does not conflict with } I_j\}).$$

If the maximal-length subset of $\{I_1, I_2, \ldots, I_n\}$ includes $I_n$, then it must be $\bigcup_{p_n} \cup \{I_n\}$, where $\bigcup_{p_n}$ is a maximal-length subset of all intervals in $\{I_1, I_2, \ldots, I_{p_n}\}$. 
A Recurrence for the Optimal Schedule

Let
- $\ell(j)$ be the length of the maximal-length schedule of intervals $I_1, I_2, \ldots, I_j$
- $|I_j|$ denote the length of interval $I_j$

Then

$$\ell(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(\ell(j - 1), |I_j| + \ell(p_j)) & \text{if } j > 0 
\end{cases}$$

What we are interested in is $\ell(n)$.
A Recursive Algorithm

**SCHEDULE**\((I, p, j)\)

1. if \(j = 0\)
2. then return \(0\)
3. else return \(\max(SCHEDULE(I, p, p[j]) + |I[j]|, SCHEDULE(I, p, j - 1))\)

\[ T(n) = 2T(n-1) + O(1) = \Theta(2^n) \]
A Recursive Algorithm

\[ \text{SCHEDULE}(I, p, j) \]
\begin{align*}
1 & \textbf{if } j = 0 \\
2 & \quad \textbf{then return } 0 \\
3 & \quad \textbf{else return } \max (\text{SCHEDULE}(I, p, p[j]) + |I[j]|, \\
4 & \quad \quad \text{SCHEDULE}(I, p, j - 1))
\end{align*}

Running time:
A Recursive Algorithm

\[ \text{SCHEDULE}(I, p, j) \]

1. if \( j = 0 \)
2. then return 0
3. else return \( \max(\text{SCHEDULE}(I, p, p[j]) + |I[j]|, \text{SCHEDULE}(I, p, j - 1)) \)

Running time: \( O(2^n) \)
Unfolding the Recursion

How many \( \ell(i) \) are there?
Unfolding the Recursion

How many $l(j)$ are there?
The recursive algorithm recomputes many values repeatedly.
The recursive algorithm recomputes many values repeatedly.

*There are only $n$ values to compute!*
Memoizing the Recursive Algorithm

**MEMOIZED-SCHEDULE**(*I*, *p*, *ℓ*, *j*)

1. if *j* = 0
2. then return 0
3. else if *ℓ*[*j*] < 0
4. then *ℓ*[*j*] ← \[\text{max}(\text{MEMOIZED-SCHEDULE}(*I*, *p*, *ℓ*, *p*[*j*]) + |*I*[*j]|, \\
        \text{MEMOIZED-SCHEDULE}(*I*, *p*, *ℓ*, *j* − 1))\]
5. return *ℓ*[*j*]
**Memoizing the Recursive Algorithm**

\[
\text{MEMOIZED-SCHEDULE}(I, p, \ell, j)
\]

1. \( \text{if } j = 0 \) 
2. \( \text{then return 0} \)
3. \( \text{else if } \ell[j] < 0 \)
4. \( \text{then } \ell[j] \leftarrow \max(\text{MEMOIZED-SCHEDULE}(I, p, \ell, p[j]) + |I[j]|, \text{MEMOIZED-SCHEDULE}(I, p, \ell, j - 1)) \)
5. \( \text{return } \ell[j] \)

**Running time:**
Memoizing the Recursive Algorithm

**MEMOIZED-SCHEDULE**(*I*, *p*, ℓ, *j*)

1. if *j* = 0
2. then return 0
3. else if ℓ[*j*] < 0
4. then ℓ[*j*] ← max(MEMOIZED-SCHEDULE(*I*, *p*, ℓ, *p[*j]*) + |*I[*j]*|,
   MEMOIZED-SCHEDULE(*I*, *p*, ℓ, *j* − 1))
5. return ℓ[*j*]

*Running time:* \(O(n)\)
Memoizing the Recursive Algorithm

\[ \text{MEMOIZED-SCHEDULE} (I, p, \ell, j) \]

\[
\begin{align*}
1 & \text{ if } j = 0 \\
2 & \quad \text{ then return } 0 \\
3 & \quad \text{ else if } \ell[j] < 0 \\
4 & \quad \quad \text{ then } \ell[j] \leftarrow \max (\text{MEMOIZED-SCHEDULE} (I, p, \ell, p[j]) + |I[j]|, \\
5 & \quad \quad \quad \text{MEMOIZED-SCHEDULE} (I, p, \ell, j - 1)) \\
6 & \quad \text{ return } \ell[j]
\end{align*}
\]

\textbf{Running time: } \mathcal{O}(n)

\textbf{Memoization: } Store already computed values in a table to avoid recomputing them.
Iterative Table Fill-In

**ITERATIVE-SCHEDULE**$(I, p)$

1. $\ell[0] \leftarrow 0$
2. for $j \leftarrow 1$ to $n$
   3. do $\ell[j] \leftarrow \max(\ell[p[j]] + |I[j]|, \ell[j - 1])$
4. return $\ell[n]$

- Total: $O(n)$
- $\ell[0] = ?$
- $\ell[n] = ?$

Only gives the $\ell$ values!
One way to recover the solutions:

Store extra information

\[
\begin{array}{c|c|c}
\text{choice} & \text{yes} & \text{no} \\
\hline
\text{weight} & \emptyset & \emptyset \\
\end{array}
\]

did we pick interval \( I_j \)
From the Value to the Solution

**Iterative-Schedule** $(I, p)$

1. $\ell[0] \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** $n$
3.   **do if** $\ell[p[j]] + |I[j]| > \ell[j - 1]$
4.     **then** $\ell[j] \leftarrow \ell[p[j]] + |I[j]|$
5.     $c[j] \leftarrow$ TRUE
6.   **else** $\ell[j] \leftarrow \ell[j - 1]$
7.     $c[j] \leftarrow$ FALSE
8. **return** $(c, p)$
**Find-Schedule** \((I, c, p)\)

1. \( j \leftarrow n \)
2. \( S \leftarrow \emptyset \)
3. While \( j > 0 \)
4.  
   a. If \( c[j] = \text{TRUE} \)
   b. \( S \leftarrow S \cup \{I[j]\} \)
   c. \( j \leftarrow p[j] \)
5.  
6.  
7.  
8. Else \( j \leftarrow j - 1 \)
**Find-Schedule** \((I, c, p)\)

1. \(j \leftarrow n\)
2. \(S \leftarrow \emptyset\)
3. While \(j > 0\) do if \(c[j] = \text{TRUE}\) then \(S \leftarrow S \cup \{I[j]\}\)
4. \(j \leftarrow p[j]\)
5. else \(j \leftarrow j - 1\)

**Lemma:** Given the intervals in sorted order and given the predecessor array \(p\), the weighted interval scheduling problem can be solved in \(O(n)\) time.
The Missing Details

What’s missing?
What’s missing?

- Sort the intervals by their ending times.
What’s missing?

- Sort the intervals by their ending times.
- Compute array $p$. 

\[0 \quad n-2\]
\[0 \quad n-1\]
\[\ldots\]
\[= O(n^2)\]
The Missing Details

What’s missing?

- Sort the intervals by their ending times.
- Compute array $p$.

Solution:
The Missing Details

What’s missing?

- Sort the intervals by their ending times.
- Compute array $p$.

Solution:

- Sorting is easily done in $O(n \lg n)$ time.
What’s missing?

- Sort the intervals by their ending times.
- Compute array $p$.

Solution:

- Sorting is easily done in $\mathcal{O}(n \log n)$ time.
- To compute $p[j]$, perform binary search with $I_j$’s starting time on sorted array.
The Missing Details

What’s missing?

- Sort the intervals by their ending times.
- Compute array $p$.

Solution:

- Sorting is easily done in $O(n \log n)$ time.
- To compute $p[j]$, perform binary search with $I_j$’s starting time on sorted array.

Theorem: The weighted interval scheduling problem can be solved in $O(n \log n)$ time.

$$c_1 n \log n + c_2 n \log n + c_3 n$$
The Dynamic Programming Paradigm

The paradigm:

- Compute the **value** of the optimal solution to **every** possible subproblem, bottom-up.
- Use auxiliary information set-up in the previous phase to compute the optimal solution top-down.

\[ T(n) \leq T(n-1) + T(n-1) + T(n-2) \]

\[ T(1) = 0 \quad T(2) = 1 \]

\[ T(n) \in \Theta(2^n) \]
The paradigm:

- Compute the value of the optimal solution to every possible subproblem, bottom-up.
- Use auxiliary information set-up in the previous phase to compute the optimal solution top-down.

For this to work, an optimal solution to a problem instance must be composed of optimal solutions to smaller problem instances.
The Dynamic Programming Paradigm

The paradigm:

- Compute the value of the optimal solution to every possible subproblem, bottom-up.
- Use auxiliary information set-up in the previous phase to compute the optimal solution top-down.

For this to work, an optimal solution to a problem instance must be composed of optimal solutions to smaller problem instances.

A speed-up over the naive algorithms is achieved if the problem exhibits overlapping subproblems.

The same subproblems occur over and over as the problem is recursively split into subproblems.
Step 1: Assume we have an optimal solution; what can we say about its structure? (Think top-down.)

Identify how the problem reduces to smaller subproblems.
Developing a Dynamic Programming Solution

**Step 1:** Assume we have an optimal solution; what can we say about its structure? (Think top-down.)

Identify how the problem reduces to smaller subproblems.

**Step 2:** Compute the value of an optimal solution bottom-up.

Since an optimal solution depends on optimal solutions of smaller subproblems, we need to solve those first.

Keep track of the choice that results in optimal solution for each of the subproblems.
Developing a Dynamic Programming Solution

**Step 1:** Assume we have an optimal solution; what can we say about its structure? (Think top-down.)

Identify how the problem reduces to smaller subproblems.

**Step 2:** Compute the value of an optimal solution bottom-up.

Since an optimal solution depends on optimal solutions of smaller subproblems, we need to solve those first.

Keep track of the choice that results in optimal solution for each of the subproblems.

**Step 3:** Construct the optimal solution top-down.

In order to obtain the optimal solution, we have to make the optimal choices identified in the previous step one by one, in a top-down manner.
RNA Sequence Secondary Structure

The pairing of an RNA strand $B = b_1 b_2 \ldots b_n$ approximately adheres to the following rules:

- **Proper pairing:** Only pairs $A-U$ and $C-G$ are allowed and every base is allowed to participate in at most one pair.

- **No sharp turns:** If $(b_i, b_j)$ is a pair, then $i < j - 4$.

- **No crossings:** If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs with $i < k$, we cannot have $i < k < j < l$.

- **Maximal number of pairs:** There are as many pairs as possible, subject to the above rules.

**Goal:** Predict the secondary structure of a given RNA sequence $B = b_1 b_2 \ldots b_n$. 

No sharp turns

No crossings
Let’s try to develop a recurrence for $p(j)$, the maximal number of base pairs in the string $b_1 b_2 \ldots b_j$ achievable while satisfying the constraints for a proper pairing.

What we are interested in, then, is $p(n)$. 
Let's try to develop a recurrence for $p(j)$, the maximal number of base pairs in the string $b_1 b_2 \ldots b_j$ achievable while satisfying the constraints for a proper pairing.

What we are interested in, then, is $p(n)$.

**Two basic observations:**

- If $j < 6$, there cannot be any base pairs because we are not allowed to make sharp turns; that is, $p(j) = 0$ in this case.
Let's try to develop a recurrence for \( p(j) \), the maximal number of base pairs in the string \( b_1 b_2 \ldots b_j \) achievable while satisfying the constraints for a proper pairing.

What we are interested in, then, is \( p(n) \).

**Two basic observations:**

- If \( j < 6 \), there cannot be any base pairs because we are not allowed to make sharp turns; that is, \( p(j) = 0 \) in this case.
- If \( j \geq 6 \), we have two choices:
RNA Secondary Structure: Problem Analysis

Let’s try to develop a recurrence for \( p(j) \), the maximal number of base pairs in the string \( b_1 b_2 \ldots b_j \) achievable while satisfying the constraints for a proper pairing.

What we are interested in, then, is \( p(n) \).

Two basic observations:

- If \( j < 6 \), there cannot be any base pairs because we are not allowed to make sharp turns; that is, \( p(j) = 0 \) in this case.
- If \( j \geq 6 \), we have two choices:
  - \( b_j \) forms a pair with some base \( b_t \), \( 1 \leq t < j - 4 \).
Let’s try to develop a recurrence for $p(j)$, the maximal number of base pairs in the string $b_1 b_2 \ldots b_j$ achievable while satisfying the constraints for a proper pairing.

What we are interested in, then, is $p(n)$.

**Two basic observations:**

- If $j < 6$, there cannot be any base pairs because we are not allowed to make sharp turns; that is, $p(j) = 0$ in this case.
- If $j \geq 6$, we have two choices:
  - $b_j$ forms a pair with some base $b_t$, $1 \leq t < j - 4$.
  - $b_j$ is not part of a base pair.
A recurrence:
A recurrence:

- If $b_j$ is not part of a base pair, then $p(j) = p(j - 1)$. 
A recurrence:

- If $b_j$ is not part of a base pair, then $p(j) = p(j - 1)$.
- If $(b_t, b_j)$ is a base pair, then $p(j) = 1 + p(t - 1) + ???$. 

\[
\begin{align*}
\text{\textbf{CSci 3110 • Dynamic Programming • 26/43}}
\end{align*}
\]
A recurrence:

- If $b_j$ is not part of a base pair, then $p(j) = p(j - 1)$.
- If $(b_t, b_j)$ is a base pair, then $p(j) = 1 + p(t - 1) + ???$.

Need a second parameter in the recurrence to be able to recurse on subsequence $b_{t+1}b_{t+2}\ldots b_{j-1}$.
A recurrence:

- If $b_j$ is not part of a base pair, then $p(j) = p(j - 1)$.
- If $(b_t, b_j)$ is a base pair, then $p(j) = 1 + p(t - 1) + ???$.

Need a second parameter in the recurrence to be able to recurse on subsequence $b_{t+1}b_{t+2} \ldots b_{j-1}$.

New goal:

- Develop recurrence for $p(i, j)$, the number of pairs in an optimal pairing of sequence $b_ib_{i+1} \ldots b_j$.
- We are then interested in $p(1, n)$. 
Recurrence:

\( j \leq i + 4:\)

\[ p(i, j) = 0 \text{ if } j \leq i + 4 \]
Recurrence:

- $j \leq i + 4$:
  
  $p(i, j) = 0$ if $j \leq i + 4$

- $j > i + 4$:
Recurrence:

- $j \leq i + 4$: 
  \[ p(i, j) = 0 \text{ if } j \leq i + 4 \]

- $j > i + 4$: 
  \[ p(i, j) = \max(p(i, j - 1), \quad \text{not pairing}_b) \]
Recurrence:

- $j \leq i + 4$:
  
  \[ p(i, j) = 0 \text{ if } j \leq i + 4 \]

- $j > i + 4$:
  
  \[ p(i, j) = \max(p(i, j - 1), \\
    1 + \max\{p(i, t - 1) + p(t + 1, j - 1) \mid \\
    i \leq t < j - 4 \text{ and bases } b_t \text{ and } b_j \text{ match}\} \]
The Algorithm

How to organize the computation?

need all subproblems $p(a, b)$ where $b < j$
How to organize the computation?

- Computation of $p(i, j)$ depends only on values $p(i', j')$ with $j' < j$. 

The Algorithm
How to organize the computation?

- Computation of $p(i, j)$ depends only on values $p(i', j')$ with $j' < j$.

**Algorithm**

**RNA-SECONDARY-STRUCTURE($B$)**

1. Allocate an $n \times n$ array $p$ and initialize all entries to 0.
2. Allocate an $n \times n$ array $c$ and initialize all entries to 0.
3. for $j \leftarrow 6$ to $n$ \( \mathcal{O}(n^2) \)
4. do for $i \leftarrow 1$ to $j - 5$ \( \mathcal{O}(n^2) \)
5. do $p[i, j] \leftarrow p[i, j - 1] + c[i, j]$
6. $c[i, j] \leftarrow 0$
7. for $t \leftarrow i$ to $j - 5$ \( \mathcal{O}(n^2) \)
8. do if bases $B[t]$ and $B[j]$ match and
   
   $1 + p[i, t - 1] + p[t + 1, j - 1] > p[i, j]$
9. then $p[i, j] \leftarrow 1 + p[i, t - 1] + p[t + 1, j - 1]$
10. $c[i, j] \leftarrow t$

Matrix of size $2 \cdot \mathcal{O}(n^2)$

Take linear time for each entry $= \mathcal{O}(n^3)$
How to organize the computation?

- Computation of $p(i, j)$ depends only on values $p(i', j')$ with $j' < j$.

**RNA-SECONDARY-STRUCTURE**(B)

1. Allocate an $n \times n$ array $p$ and initialize all entries to 0.
2. Allocate an $n \times n$ array $c$ and initialize all entries to 0.
3. for $j \leftarrow 6$ to $n$
4.   do for $i \leftarrow 1$ to $j - 5$
5.     do $p[i, j] \leftarrow p[i, j - 1]$
6.     $c[i, j] \leftarrow 0$
7.   for $t \leftarrow i$ to $j - 5$
8.     do if bases $B[t]$ and $B[j]$ match and
9.         $1 + p[i, t - 1] + p[t + 1, j - 1] > p[i, j]$
10.        then $p[i, j] \leftarrow 1 + p[i, t - 1] + p[t + 1, j - 1]$
11.        $c[i, j] \leftarrow t$

**Lemma:** Procedure RNA-SECONDARY-STRUCTURE takes $O(n^3)$ time.
From the Value to the Solution

\[
\text{extract-pairing}(c, i, j) \begin{cases} 
1 & \text{if } j \leq i + 4 \\
2 & \text{then return } \emptyset \\
3 & \text{else if } c[i, j] = 0 \\
4 & \text{then return } \text{extract-pairing}(c, i, j - 1) \\
5 & \text{else return } \text{extract-pairing}(c, i, c[i, j] - 1) \cup \\
& \text{extract-pairing}(c, c[i, j] + 1, j - 1) \cup \{(c[i, j], j)\} 
\end{cases}
\]

\[P(i, j) = \text{the best score for joining pairs between } i \text{ and } j;\]

\[
\text{answer} = P[1, n] \begin{cases} 
\text{if } c(i, j) = 0 \text{ then } \\
\text{if } c(i, j) > 0 \text{ then } 
\end{cases}
\]

CSci 3110 • Dynamic Programming • 29/43
if \( c(i_{jj}) = 0 \) then

\[ \text{go to } c(i_{jj}-1) \]

else \( c(i_{jj}) = x \)

add base pair \((x_{jj})\)
**From the Value to the Solution**

\[
\text{initially } \left( c_{ij} \right)_{n} \\
\]

**Algorithm: Extract-Pairing**

```plaintext
EXTRACT-PAIRING(c, i, j)
1  if j \leq i + 4
2       then return \emptyset \quad \text{base case}
3  else if c[i, j] = 0
4       then return EXTRACT-PAIRING(c, i, j - 1)
5  else return EXTRACT-PAIRING(c, c[i, j] - 1) \cup
6                  EXTRACT-PAIRING(c, c[i, j] + 1, j - 1) \cup \{(c[i, j], j)\}
```

**Theorem:** The RNA secondary structure can be computed (approximately) in \( O(n^3) \) time.
**Sequence Alignment**

**Problem:** Extend strings $X$ and $Y$ to the same length by inserting gaps so that the following dissimilarity measure is minimized:

- **Gap penalty** $\delta > 0$.
- **Mismatch penalty** $p_{ab}$, for every pair $(a, b)$ of letters in the alphabet. (Assume that $p_{aa} = 0$, for all $a$.)

**Example:**

```
Dalh-usy-
Dalhousie
```

**Cost:** $2\delta + p_{iy}$
Let $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ be the given sequences.

Three choices:
Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be the given sequences.

**Three choices:**

- $(x_m, y_n)$ are paired.
Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be the given sequences.

**Three choices:**

- $(x_m, y_n)$ are paired.
- $x_m$ is unmatched, that is, matched to a gap.
- $y_n$ is unmatched, that is, matched to a gap.
Let $D(i, j)$ be the dissimilarity of strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

Then we are interested in $D(m, n)$.
Let $D(i, j)$ be the dissimilarity of strings $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$.

Then we are interested in $D(m, n)$.

**Recurrence:**

- $i = 0$ or $j = 0$: 
Let $D(i, j)$ be the dissimilarity of strings $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$.

Then we are interested in $D(m, n)$.

**Recurrence:**

- $i = 0$ or $j = 0$:
  
  $D(i, j) = \delta \cdot \max(i, j)$
Let $D(i, j)$ be the dissimilarity of strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

Then we are interested in $D(m, n)$.

**Recurrence:**

- $i = 0$ or $j = 0$:
  
  $$D(i, j) = \delta \cdot \max(i, j)$$

- Otherwise:
Let $D(i, j)$ be the dissimilarity of strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

Then we are interested in $D(m, n)$.

**Recurrence:**

- $i = 0$ or $j = 0$:
  
  $D(i, j) = \delta \cdot \max(i, j)$

- Otherwise:
  
  
  $D(i, j) = \min(p_{x_i y_j} + D(i - 1, j - 1), \delta + D(i - 1, j), \delta + D(i, j - 1))$
The Algorithm

**SEQUENCE-ALIGNMENT**( \(X, Y\))

1. `for i ← 0 to m`
2. `do D[i, 0] ← i · δ`
3. `for j ← 0 to n`
4. `do D[0, j] ← j · δ`
5. `for i ← 1 to m`
6. `do for j ← 1 to n`
7. `do D[i, j] ← D[i − 1, j − 1] + p[X[i], Y[j]]`
8. `if D[i, j] > D[i − 1, j] + δ`  
   - `then D[i, j] ← D[i − 1, j] + δ`
9. `if D[i, j] > D[i, j − 1] + δ`  
   - `then D[i, j] ← D[i, j − 1] + δ`
The Algorithm

**Sequence-Alignment** \((X, Y)\)

1. for \(i \leftarrow 0\) to \(m\)
2. \(\textbf{do} \ D[i, 0] \leftarrow i \cdot \delta\)
3. for \(j \leftarrow 0\) to \(n\)
4. \(\textbf{do} \ D[0, j] \leftarrow j \cdot \delta\)
5. for \(i \leftarrow 1\) to \(m\)
6. \(\textbf{do for} \ j \leftarrow 1\) to \(n\)
7. \(\textbf{do} \ D[i, j] \leftarrow D[i - 1, j - 1] + p[X[i], Y[j]]\)
8. \(\textbf{if} \ D[i, j] > D[i - 1, j] + \delta\)
9. \(\textbf{then} \ D[i, j] \leftarrow D[i - 1, j] + \delta\)
10. \(\textbf{if} \ D[i, j] > D[i, j - 1] + \delta\)
11. \(\textbf{then} \ D[i, j] \leftarrow D[i, j - 1] + \delta\)

The actual solution can again be obtained by keeping track of the optimal choices and then recursively extracting the solution.
The Algorithm

**Sequence-Alignment** \((X, Y)\)

1. for \(i \leftarrow 0\) to \(m\)
2. \hspace{1em} do \(D[i, 0] \leftarrow i \cdot \delta\)
3. for \(j \leftarrow 0\) to \(n\)
4. \hspace{1em} do \(D[0, j] \leftarrow j \cdot \delta\)
5. for \(i \leftarrow 1\) to \(m\)
6. \hspace{1em} do for \(j \leftarrow 1\) to \(n\)
7. \hspace{2em} do \(D[i, j] \leftarrow D[i - 1, j - 1] + p[X[i], Y[j]]\)
8. \hspace{1em} if \(D[i, j] > D[i - 1, j] + \delta\)
9. \hspace{2em} then \(D[i, j] \leftarrow D[i - 1, j] + \delta\)
10. \hspace{1em} if \(D[i, j] > D[i, j - 1] + \delta\)
11. \hspace{2em} then \(D[i, j] \leftarrow D[i, j - 1] + \delta\)

The actual solution can again be obtained by keeping track of the optimal choices and then recursively extracting the solution.

**Theorem:** The sequence alignment problem can be solved in \(\mathcal{O}(mn)\) time.
\[ \alpha \]

\[ \text{all } P = 1 \]

\[ \text{gap } = 1 \]

\[ D(i,j) \]

\[ 0 + \text{Pad} = 1 \]

\[ 1 + \text{gap} = 2 \]

\[ 1 + \text{gap} = 2 \]

\[ D(0,1) + P_{aa} = 1 + 0 = 1 \]

\[ -a \quad p \quad h \quad q \]

\[ d \quad a \quad l \quad - \quad - \]
<table>
<thead>
<tr>
<th>dollhouse</th>
<th>drills</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 2, 4, 6, 8, 10, 12, 14, 16, 18</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2023-02-14</th>
<th>12:34:56</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 6, 8, 10, 12, 14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>6, 8, 10, 12, 14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>8, 10, 12, 14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>10, 12, 14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>12, 14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>14, 16, 18</td>
<td></td>
</tr>
<tr>
<td>16, 18</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

| gap=2 |
| mismatch=1 |
Dynamic Programming

1. Identify subproblems

   problem array is
   subproblem: array[i]
   array is

2. Create a recurrence
   - base case
   - identify how the larger problem is built from subproblems

   / limited number of values
③ figure out the optimal order

④ implement

⑤ prove correctness and running time

This only gives us the max (or min) numbers.

⑥ need to store choice information.

backtrack using this to compute the solution.
Dijkstra’s algorithm may fail in the presence of negative-weight edges:

We need an algorithm that can deal with negative edge weights.
Lemma: If $P = (u_0, u_1, \ldots, u_k)$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from $u_0$ to $u_{k-1}$. 
**Lemma:** If $P = (u_0, u_1, \ldots, u_k)$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from $u_0$ to $u_{k-1}$.

**Observation:** $P'$ has one edge less than $P$. 
Let \( \text{dist}_i(s, x) \) be the length of the shortest path with at most \( i \) edges from \( s \) to \( x \).

Then \( \text{dist}(s, x) = \text{dist}_{n-1}(s, x) \)
Let $\text{dist}_i(s, x)$ be the length of the shortest path with at most $i$ edges from $s$ to $x$.

Then $\text{dist}(s, x) = \text{dist}_{n-1}(s, x)$

\textbf{Recurrence:}
Let \( \text{dist}_i(s, x) \) be the length of the shortest path with at most \( i \) edges from \( s \) to \( x \).

Then \( \text{dist}(s, x) = \text{dist}_{n-1}(s, x) \)

**Recurrence:**
A Recurrence for Shortest Paths

- Let $\text{dist}_i(s, x)$ be the length of the shortest path with at most $i$ edges from $s$ to $x$.
- Then $\text{dist}(s, x) = \text{dist}_{n-1}(s, x)$

**Recurrence:**
Let \( \text{dist}_i(s, x) \) be the length of the shortest path with at most \( i \) edges from \( s \) to \( x \).

Then \( \text{dist}(s, x) = \text{dist}_{n-1}(s, x) \)

**Recurrence:**

- \( i = 0 \):
  - \( \text{dist}_0(s, s) = 0 \)
  - \( \text{dist}(s, x) = \infty \), for \( x \neq s \).

- \( i > 0 \):
  
  \[
  \text{dist}_i(s, x) = \min \left( \text{dist}_{i-1}(s, x), \min_{(y, x) \in E} \left( \text{dist}_{i-1}(s, y) + w(y, x) \right) \right)
  \]
The Bellman-Ford Algorithm

**Bellman-Ford**$(G, s)$

1. for every vertex $v$ of $G$
2. do dist[$v, 0$] ← $\infty$
3. dist[$s, 0$] ← 0
4. for $i \leftarrow 1$ to $n - 1$
5. do for every vertex $v$ of $G$
6. do dist[$v, i$] ← dist[$v, i - 1$]
7. for every in-edge $(u, v)$ of $G$
8. do if dist[$u, i - 1$] + $w[u, v]$ < dist[$v, i$]
9. then dist[$v, i$] ← dist[$u, i - 1$] + $w[u, v]$
The Bellman-Ford Algorithm

**Bellman-Ford**($G, s$)

1. for every vertex $v$ of $G$
2. do $\text{dist}[v, 0] \leftarrow \infty$
3. $\text{dist}[s, 0] \leftarrow 0$
4. for $i \leftarrow 1$ to $n - 1$
5. do for every vertex $v$ of $G$
6. do $\text{dist}[v, i] \leftarrow \text{dist}[v, i - 1]$
7. for every in-edge $(u, v)$ of $G$
8. do if $\text{dist}[u, i - 1] + w[u, v] < \text{dist}[v, i]$
9. then $\text{dist}[v, i] \leftarrow \text{dist}[u, i - 1] + w[u, v]$

**Lemma:** Algorithm Bellman-Ford is correct.
The Bellman-Ford Algorithm

```
Bellman-Ford(G, s)
1    for every vertex v of G
2        do dist[v, 0] ← ∞
3    dist[s, 0] ← 0
4    for i ← 1 to n − 1
5        do for every vertex v of G
6            do dist[v, i] ← dist[v, i − 1]
7        for every in-edge (u, v) of G
8            do if dist[u, i − 1] + w[u, v] < dist[v, i]
9                then dist[v, i] ← dist[u, i − 1] + w[u, v]
```

**Lemma:** Algorithm Bellman-Ford is correct.

**Lemma:** Algorithm Bellman-Ford takes $O(nm)$ time on a graph with $n$ vertices and $m$ edges.
Finding Shortest Paths

**Bellman-Ford**($G, s$)

1. for every vertex $v$ of $G$
2. do $\text{dist}[v, 0] \leftarrow \infty$
3. parent[$v$] $\leftarrow$ nil
4. $\text{dist}[s, 0] \leftarrow 0$
5. for $i \leftarrow 1$ to $n - 1$
6. do for every vertex $v$ of $G$
7. do $\text{dist}[v, i] \leftarrow \text{dist}[v, i - 1]$
8. for every in-edge $[u, v]$ of $G$
9. do if $\text{dist}[u, i - 1] + w[u, v] < \text{dist}[v, i]$
10. then $\text{dist}[v, i] \leftarrow \text{dist}[u, i - 1] + w[u, v]$
11. parent[$v$] $\leftarrow u$
All-Pairs Shortest Paths

**Goal:** Compute the distance $\text{dist}(v, w)$ from $v$ to $w$, for every pair $(v, w)$ of vertices in $G$.

**First idea:** Run single-source shortest paths from every vertex $v$.

**Complexity:**
- $O(n^2m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

**Improved algorithms for arbitrary edge weights:**
- Floyd-Warshall: $O(n^3)$
- Johnson: $O(n^2 \lg n + nm)$
A Recurrence for APSP

Number the vertices 1, 2, \ldots, n.

Let $\text{dist}_i(v, w)$ be the length of shortest path from $v$ to $w$ that uses only vertices 1, 2, \ldots, $i$ as intermediate vertices.
A Recurrence for APSP

Number the vertices 1, 2, \ldots, n.

Let \( \text{dist}_i(v, w) \) be the length of shortest path from \( v \) to \( w \) that uses only vertices 1, 2, \ldots, \( i \) as intermediate vertices.
Number the vertices $1, 2, \ldots, n$.

Let $\text{dist}_i(v, w)$ be the length of shortest path from $v$ to $w$ that uses only vertices $1, 2, \ldots, i$ as intermediate vertices.

$\text{dist}(v, w) = \text{dist}_n(v, w)$
Floyd and Warshall’s Algorithm

**FLOYD-WARSHALL($G$)**

1. for every vertex $v$ of $G$
2. do for every vertex $w$ of $G$
3. do if $v = w$
4. then dist[$v, w$] ← 0
5. else dist[$v, w$] ← $\infty$
6. for every edge $(v, w)$ of $G$
7. do dist[$v, w$] ← $w[v, w]$
8. for $i = 1$ to $n$
9. do for every vertex $v \neq i$ of $G$
10. do for every vertex $w \neq i$ of $G$
11. do if dist[$v, w$] > dist[$v, i$] + dist[$i, w$]
12. then dist[$v, w$] ← dist[$v, i$] + dist[$i, w$]
Floyd and Warshall’s Algorithm

**FLOYD-WARSHALL**(*G*)

1. for every vertex *v* of *G*
2.   do for every vertex *w* of *G*
3.     do if *v* = *w*
4.        then dist[*v*, *w*] ← 0
5.     else dist[*v*, *w*] ← ∞
6. for every edge (*v*, *w*) of *G*
7.   do dist[*v*, *w*] ← *w*[*v*, *w*]
8. for *i* = 1 to *n*
9.   do for every vertex *v* ≠ *i* of *G*
10.     do for every vertex *w* ≠ *i* of *G*
11.        do if dist[*v*, *w*] > dist[*v*, *i*] + dist[*i*, *w*]
12.           then dist[*v*, *w*] ← dist[*v*, *i*] + dist[*i*, *w*]

Theorem: The Floyd-Warshall algorithm solves the all-pairs shortest-path problem in \(O(n^3)\) time.
Dynamic programming is an efficient method for solving optimization problems.

*For problems that can be solved using this technique, the optimal solution to each non-trivial instance must contain optimal solutions to smaller instances.*
Summary (2)

The design of a dynamic programming algorithm proceeds in three phases:

1. Assume we have an optimal solution, analyze its structure to obtain a recurrence for the cost of an optimal solution.

2. Develop an iterative algorithm that uses the recurrence to compute the values of optimal solutions to all relevant subproblems bottom-up and records which are the optimal choices made in these optimal solutions.

3. Develop a recursive algorithm that uses the information computed by the previous algorithm to construct a solution top-down by making the recorded optimal choices.