CSE 6140/ CX 4140
Computational Science and Engineering ALGORITHMS

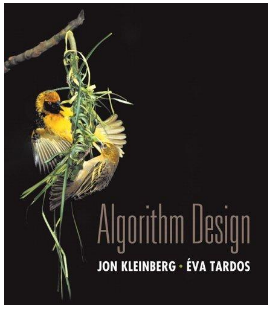
DP: RNA secondary structure

Instructor: Ziqi Zhang
PhD student
School of Computational Science and Engineering

RNA SECONDARY STRUCTURE [KT 6.5]

Adapted from Slides by
Kevin Wayne.

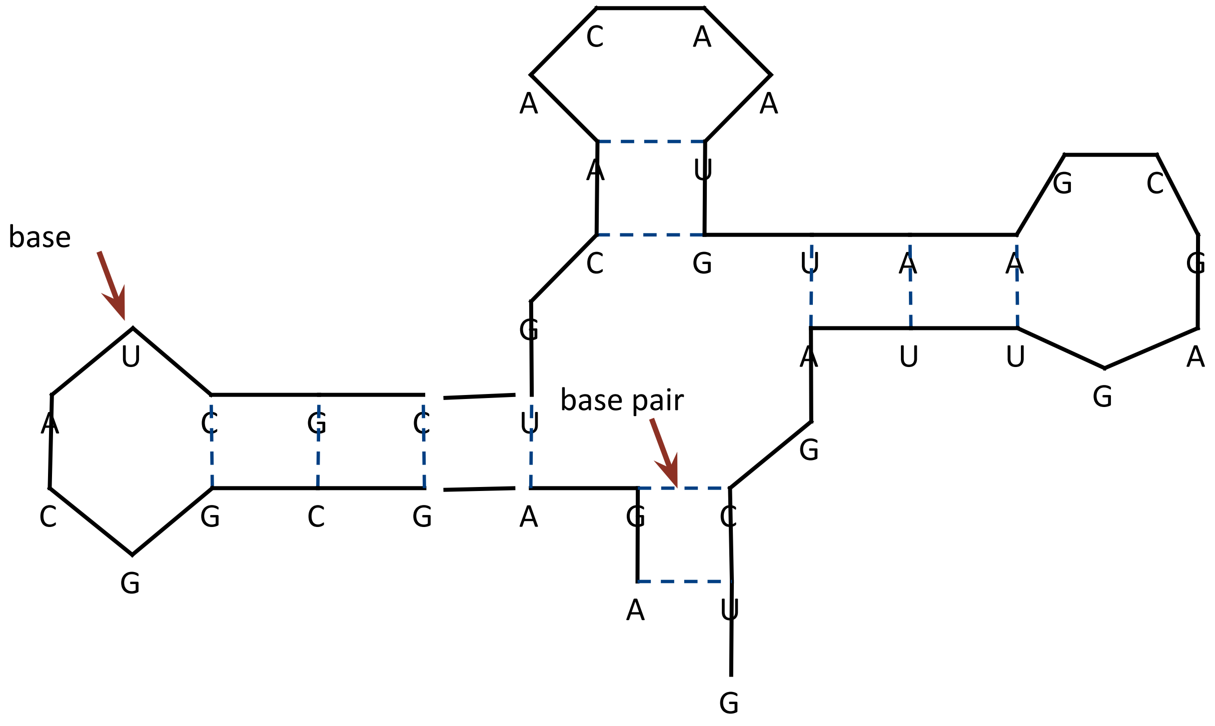
And Bistra Dilkina, Anne Benoit



RNA secondary structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet $\{ \text{A, C, G, U} \}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.



RNA secondary structure for GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

RNA secondary structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson–Crick] S is a matching and each pair in S is a Watson–Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

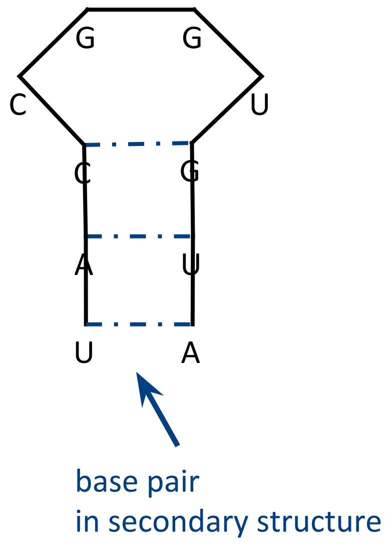
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy.

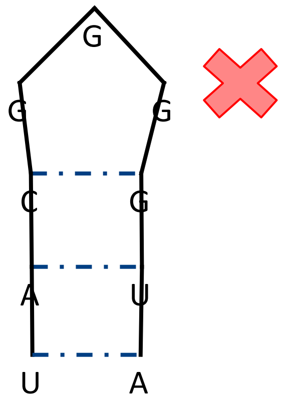
$\nwarrow$ approximate by number of base pairs

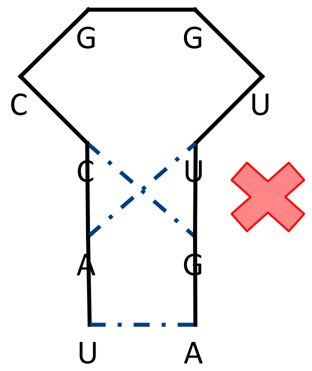
Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

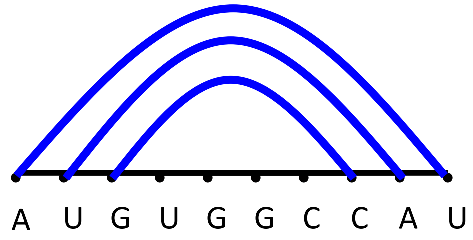
RNA secondary structure

- Examples.

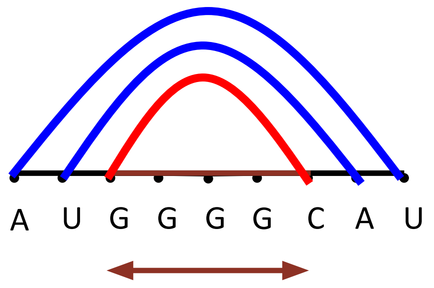






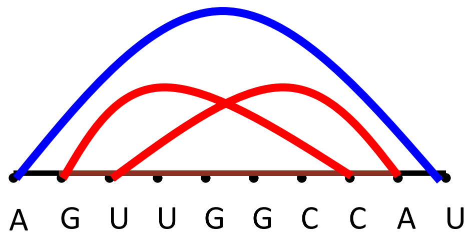


OK



sharp turn
(≤4 intervening bases)

If $(b_i, b_j) \in S$, then $i < j - 4$.



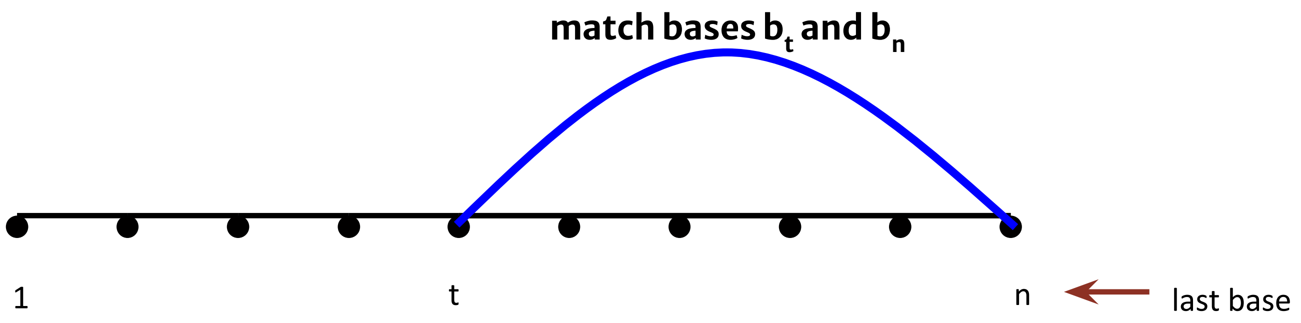
crossing

If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$

RNA secondary structure: subproblems

First attempt. $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.

Goal. $OPT(n)$.



Choice. Match bases b_t and b_n.

Difficulty. Results in two subproblems (but one of wrong form).

- Find secondary structure in $b_1 b_2 \ldots b_{t-1}$. ← OPT(t–1)
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{n-1}$. ← need more subproblems (first base no longer b_1)

Dynamic programming over intervals

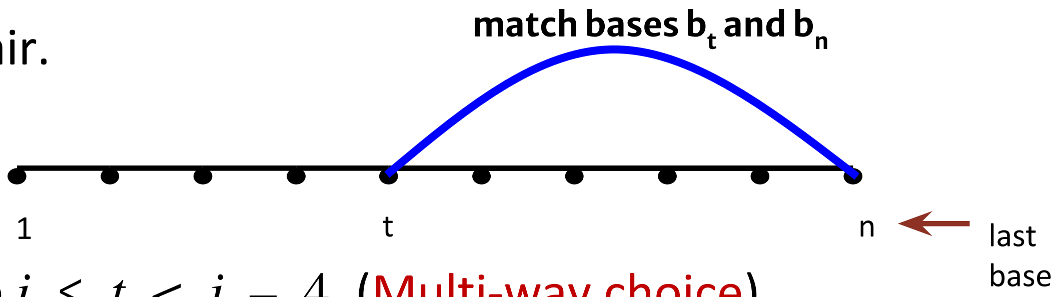
Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Case 1. If $i \geq j - 4$.

$OPT(i, j) = 0$ by no-sharp turns condition.

Case 2. Base b_j is not involved in a pair.

$OPT(i, j) = \mathrm{OPT}(i, j - 1)$.



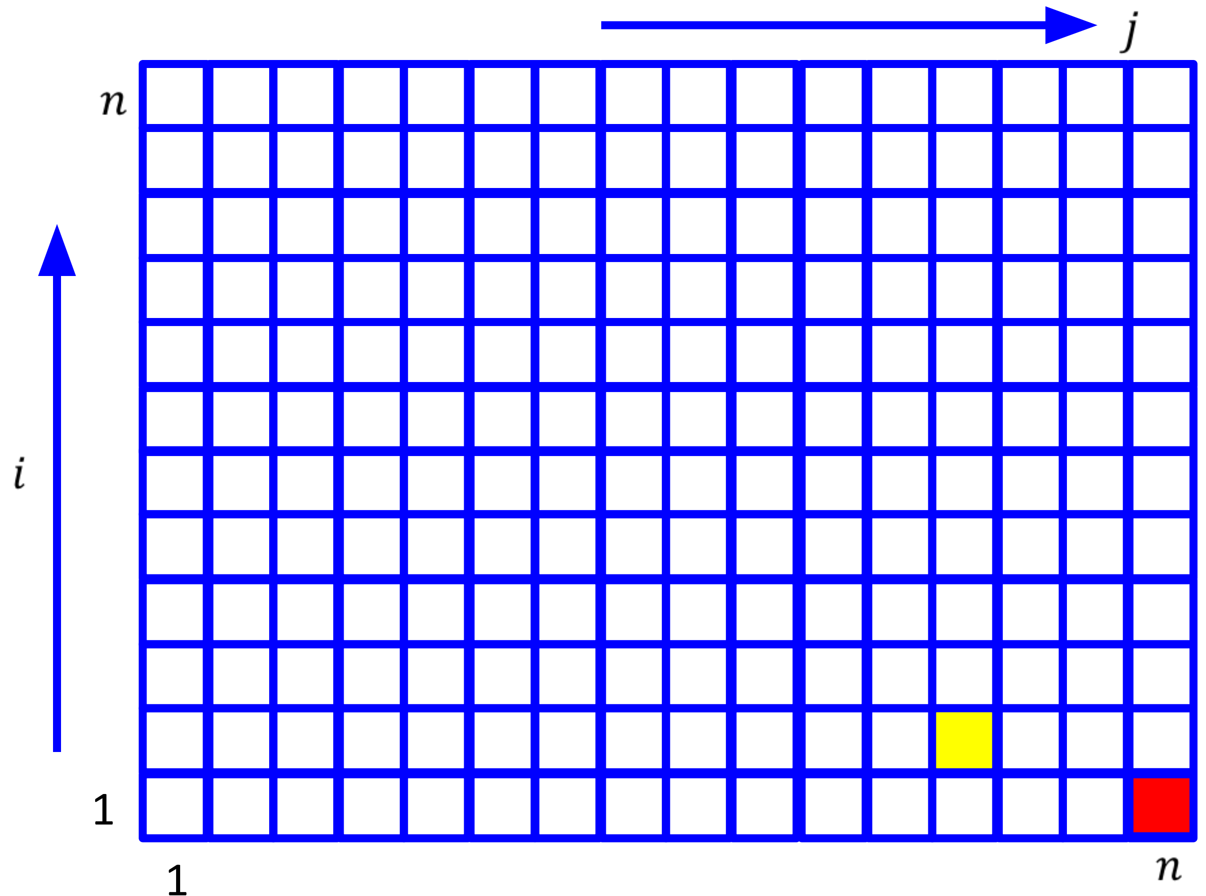
Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$. (Multi-way choice)

- Noncrossing constraint decouples resulting subproblems.
- $\mathrm{OPT}(i, j) = 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \}$.

take max over t such that $i \leq t < j - 4$ and
b_t and b_j are Watson–Crick complements (A-U, C-G)

Dynamic programming over intervals

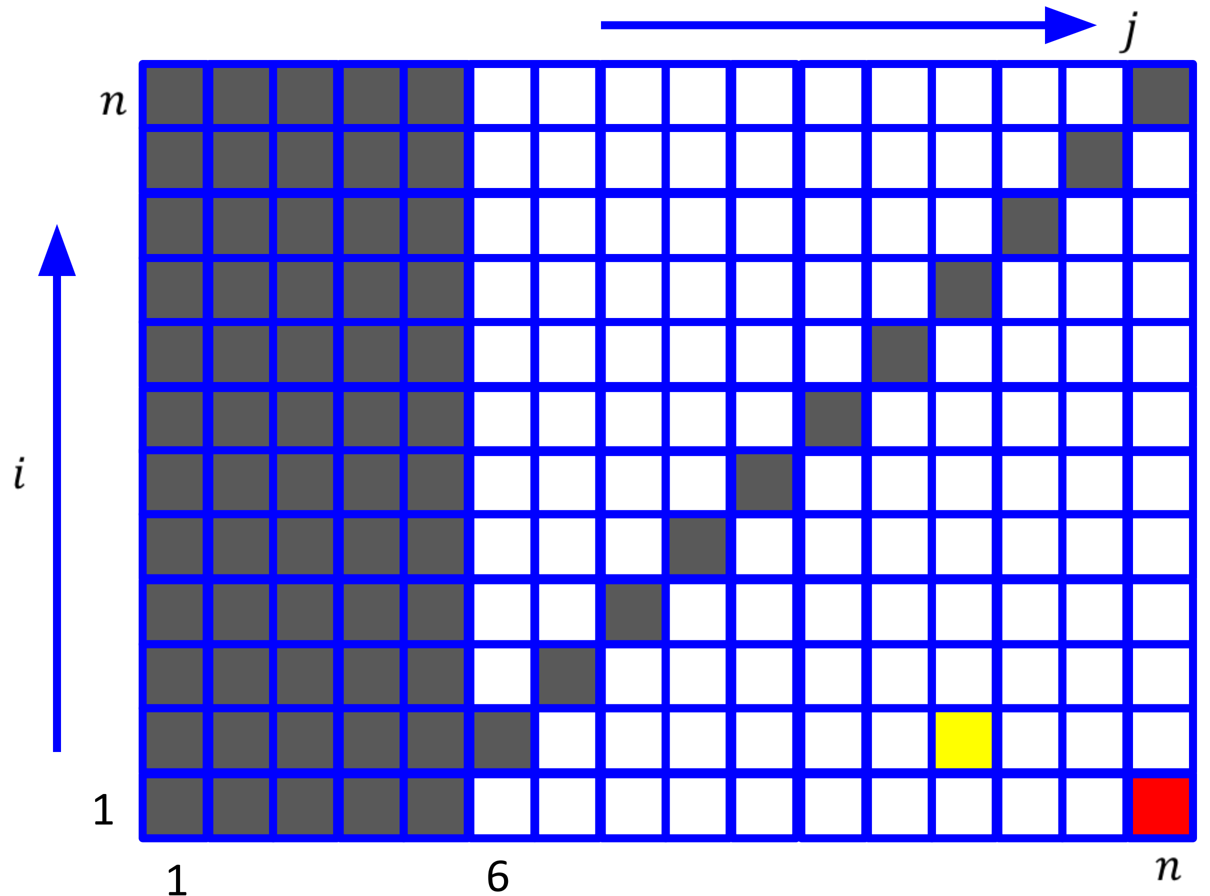
$$M(i,j) = \begin{cases} 0 & i >= j - 4 \\ \max\Big(M(i,j-1), 1 + \max_t \big(M(i,t-1) + M(t+1,j-1)\big)\Big) & \text{else} \end{cases}$$



Note: graphic is just for illustration purpose. It is not square but should be square

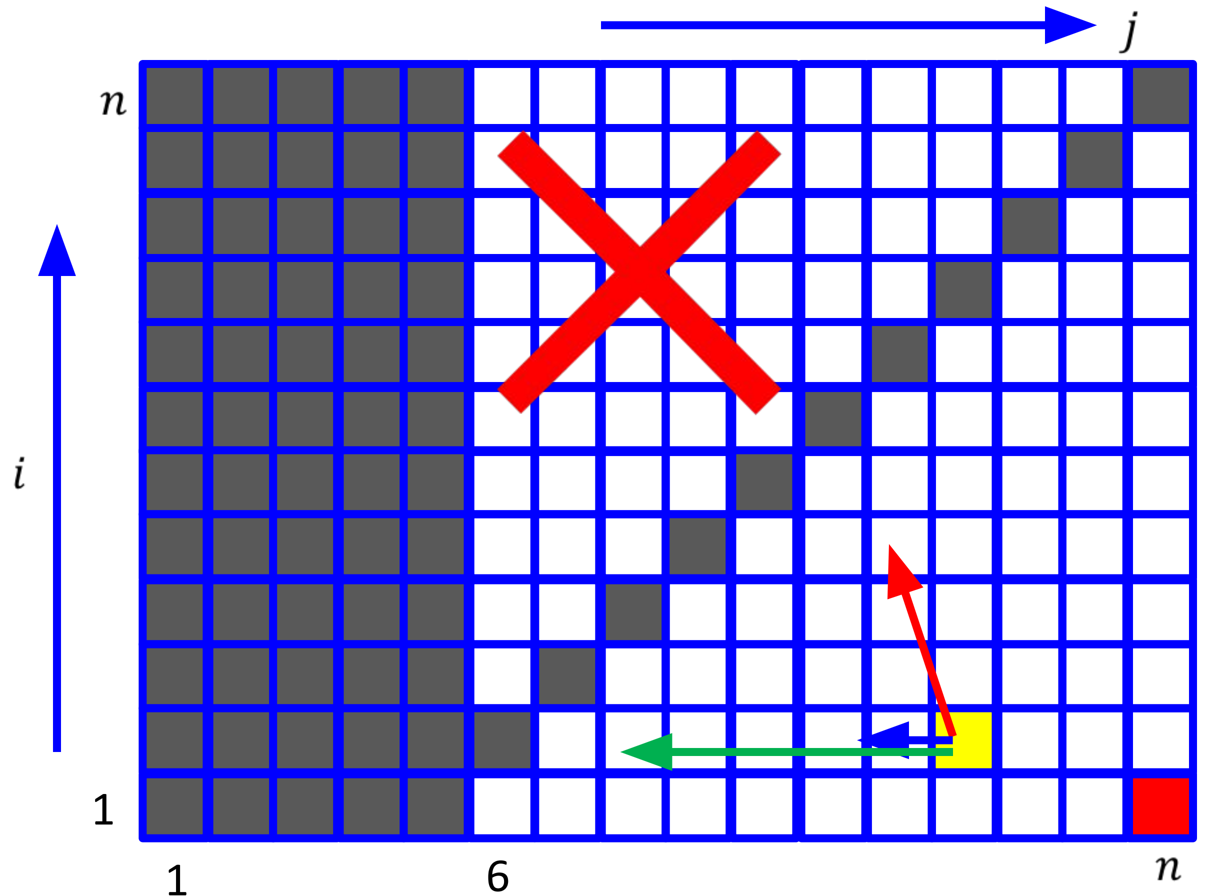
Dynamic programming over intervals

$$M(i,j) = \begin{cases} 0 & i >= j - 4 \\ \max\left(M(i,j-1), 1 + \max_t (M(i,t-1) + M(t+1,j-1)\right) & \text{else} \end{cases}$$



Dynamic programming over intervals

$$M(i,j) = \begin{cases} 0 & i \geq j - 4 \\ \max\left(M(i,j-1),\ 1 + \max_t\left(M(i,t-1) + M(t+1,j-1)\right)\right) & \text{else} \end{cases}$$



Bottom-up dynamic programming over intervals

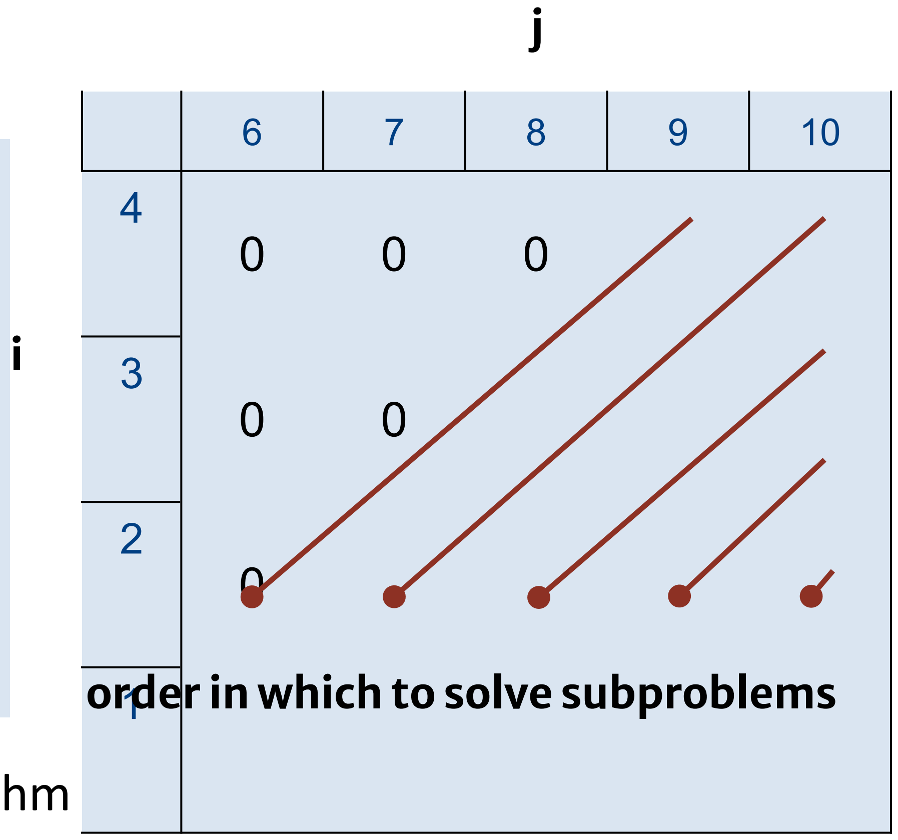
Q. In which order to solve the subproblems?

A. Do shortest intervals first.

```
RNA-SECONDARY-STRUCTURE $(n, b_1, \ldots, b_n)$

FOR $k = 5$ TO $n - 1$
    FOR $i = 1$ TO $n - k$
        $j \leftarrow i + k$.
        Compute $M[i, j]$ using formula.
RETURN $M[1, n]$.
```

all needed values are already computed



Theorem. The dynamic programming algorithm problem in $O(n^3)$ time and $O(n^2)$ space.

Dynamic Programming Summary

Recipe.

- Characterize structure of problem: identify **subproblems** whose optimal solutions can be used to build an optimal solution to original pb. Conversely, given an optimal solution to original pb, identify subparts of the solution that are optimal solutions for some subproblems. <Solve more!>
- Write the **recurrence** and **initial cases**, know where the solution of pb is.
- Look at precedence constraints (draw a figure) and write the algorithm (iterative, or recursive with memos).
- Study the pb complexity (straightforward with iterative algo; *don't forget the time to compute one subproblem*).
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: Weighted interval scheduling.
- Multi-way choice: Sequence alignment, RNA secondary structure.
- Dynamic programming over intervals: RNA secondary structure.
- Adding a new variable: Knapsack.

CSE 6140/ CX 4140

Computational Science and Engineering ALGORITHMS

NP Completeness 1

Instructor: Xiuwei Zhang

Assistant Professor

School of Computational Science and Engineering

Based on slides by Prof. Ümit V. Çatalyürek

Before NP-completeness...

Test 1: grades posted!

Test 2: 10/28

Test 3: 12/02

Dynamic programming chapter: review encouraged

NP-Completeness

- So far we have seen a lot of good news!
  - Such-and-such a problem can be solved quickly
    (i.e., in close to linear time, or at least a time that is some small polynomial function of the input size)
- NP-completeness is a form of bad news!
  - Evidence that many important problems cannot be solved quickly.
- NP-complete problems really come up all the time!

Why should we care?

- Knowing that they are hard lets you stop beating your head against a wall trying to solve them…
- Alternative ways:
  - **Restrict the problem:**
    find special restrictions/variants to the problem for which there is a polynomial time algorithm
  - **Use a heuristic:**
    come up with a method for solving a reasonable fraction of the common cases.
  - **Solve approximately:**
    come up with a method that finds solutions provably close to the optimal.
  - **Use an exponential time solution:**
    if you really have to solve the problem exactly and stop worrying about finding a better solution.

Optimization & Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no
- **Optimization problems**
  - Find a solution with the "best" value

- Optimization problems can be cast as decision problems that are easier to study
  - E.g.: Shortest path: G = unweighted directed graph
    - Find a path between u and v that uses the fewest edges
    - *Does a path exist from u to v consisting of at most k edges?*

The class P

- **Class P** consists of (decision) problems that are solvable in polynomial time
- Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$, for some constant k
- Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$

- n: size of the input data

Tractable/Intractable Problems

- Problems in P are also called **tractable**
- Problems **not** in P are **intractable**

- Are non-polynomial algorithms always worse than polynomial algorithms?

Tractable/Intractable Problems

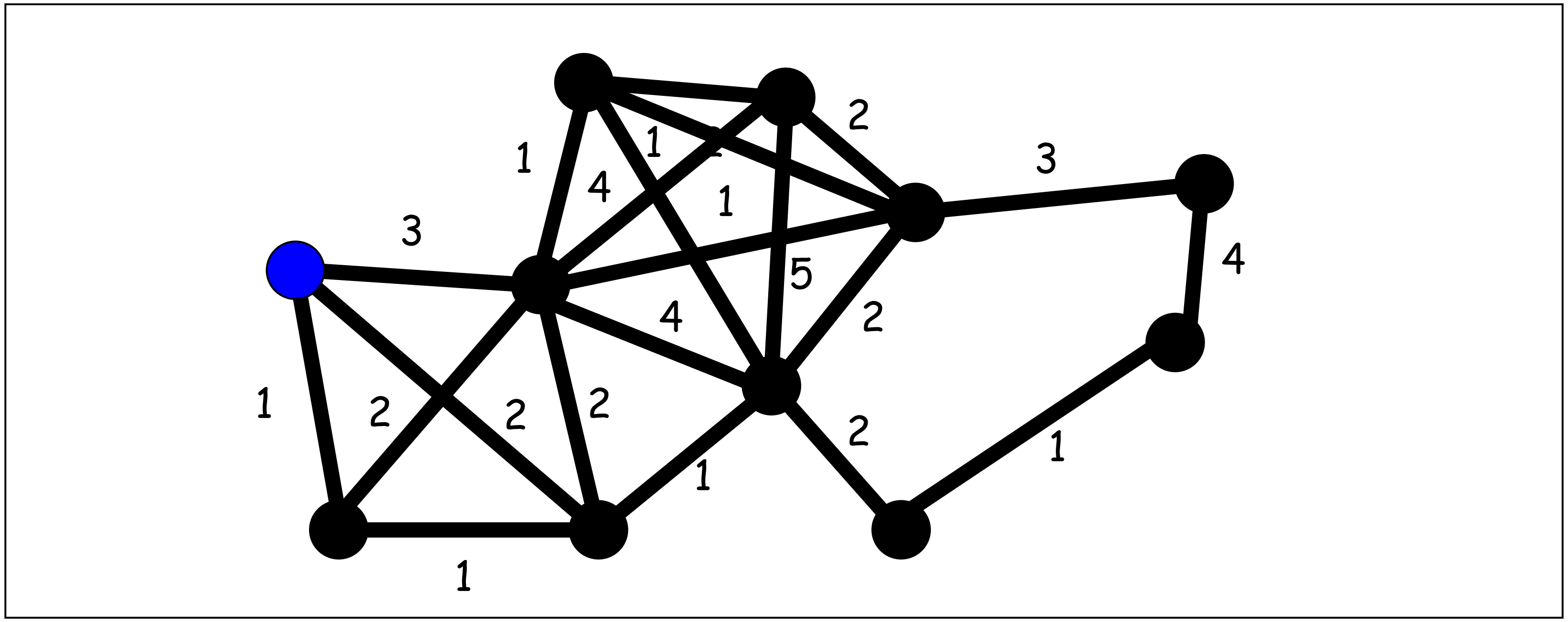
- Problems in P are also called **tractable**
- Problems **not** in P are **intractable**

- Are non-polynomial algorithms always worse than polynomial algorithms?

  $n^{1{,}000{,}000}$ is *technically* tractable, but really impossible

  $n^{\log \log \log n}$ is *technically* intractable, but easy

Example: traveling salesman problem (TSP)



- For each two cities, an integer cost is given to travel from one of the two cities to the other. The salesman wants to make a minimum cost circuit visiting each city exactly once and return to the original city.

- TSP: Given a complete graph G=(V,E), a cost function w:E->N, and an integer k, is there a cycle C going through each vertex once and only once, with $\sum_{e \in C} w(e) \leq k$? (decision version)

The Class NP

- NP is the class of problems for which a candidate solution can be verified in polynomial time

- NP does not stand for not-P!!
- NP=‘nondeterministic polynomial’

- P is a subset of NP

Nondeterministic and NP Algorithms

Nondeterministic algorithm = two stage procedure:

1) Nondeterministic ("guessing") stage:

   generate randomly an arbitrary candidate solution ("certificate")

2) Deterministic ("verification") stage:

   take the certificate and the instance to the problem and returns YES if the certificate represents a solution

NP algorithms (Nondeterministic polynomial)

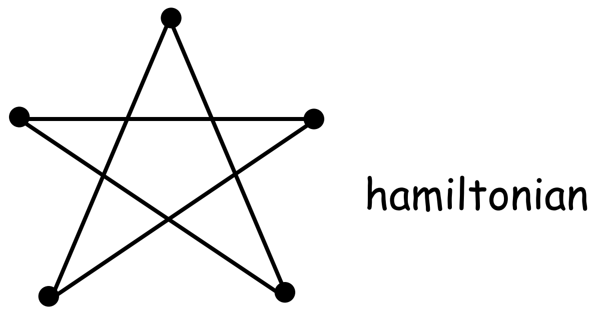
verification stage is polynomial

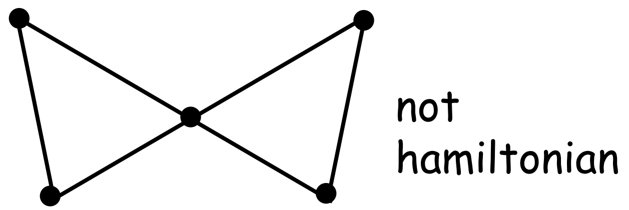
Verifying a Candidate Solution

- Difference between solving a problem and verifying a candidate solution:
- Solving a problem: is there a path in graph G from vertex u to vertex v with at most k edges?
- Verifying a candidate solution: is $v_0, v_1, \ldots, v_l$ a path in graph G from vertex u to vertex v with at most k edges?

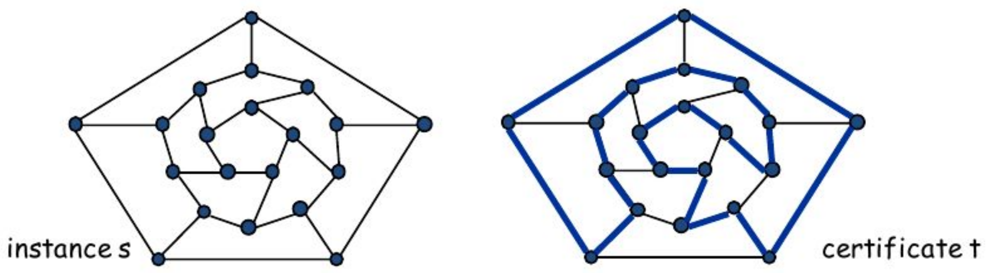
Verifying a Candidate Solution

- A Hamiltonian cycle in an undirected graph is a cycle that visits every vertex exactly once.
- Solving a problem: is there a Hamiltonian cycle in graph G?
- Verifying a candidate solution: is $v_0, v_1, \ldots, v_l$ a Hamiltonian cycle of graph G?





- **Certificate**: A list of n nodes.
- **Certifier**: Check that the list contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.
- **Conclusion**: HAM-CYCLE is in NP.



Verifying a Candidate Solution vs. Solving a Problem

- Intuitively it seems much harder (more time consuming) in some cases to solve a problem from scratch than to verify that a candidate solution actually solves the problem.

  - If there are many candidate solutions to check, then even if each individual one is quick to check, overall it can take a long time