# Dynamic programming

# General algorithmic techniques

- brute force (exhaustive search)
- recursion
- backtracking
- divide and conquer
- greedy algorithms (e.g. Dijkstra, …)
- dynamic programming (e.g. Floyd-Warshall, shortest paths in DAGs, ...)
- branch and bound
- randomization

# Dynamic Programming principles

- Characterize the structure of an optimal solution
- Express an optimal solution through optimal solutions of smaller problems (*dynamic programming relation*)
- Compute values of optimal solutions *bottom-up* (from smaller to larger)
- Construct an optimal solution from computed information

# *Example 1*: Fibonacci numbers

- Compute $n$-th Fibonacci number $F_n$

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

# *Example 1*: Fibonacci numbers

- Compute *n*-th Fibonacci number

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

- $O(n)$-time soluton: compute $F_n$ *iterative*ly from 0 to *n* (***dynamic programming***)

# *Example 1*: Fibonacci numbers

- Compute *n*-th Fibonacci number

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

- $O(n)$-time soluton: compute $F_n$ *iteratively* from 0 to *n* (***dynamic programming***)
- another $O(n)$-time soluton: ***memoization***
  - compute $F_n$ recursively (*top-down*)
  - store computed values
  - before computing $F_i$ check if it has been already computed
  - *Note*: *depth-first* exploration of the recursion (subproblem) tree

# *Example 2*: Coin changing problem

- *Problem*: given a coin system (i.e. denominations of coins $\{C_1, C_2, \ldots, C_k\}$ ) make change for $N$ "rubles" with the minimal number of coins *opt*($N$)
- Greegy strategy:
  - to change $N$, use the largest coin with value $C \leq N$
  - change $N - C$ recursively
  - *Ex*: 18 rubles = 10+5+2+1, 60 kopecks = 50+10
- Does greedy strategy always lead to an optimal solution?

# *Example 2*: Coin changing problem

- *Problem*: given a coin system (i.e. denominations of coins $\{C_1, C_2, \dots, C_k\}$ ) make change for $N$ "rubles" with the minimal number of coins $opt(N)$
- Greegy strategy:
  - to change $N$, use the largest coin with value $C \leq N$
  - change $N - C$ recursively
  - *Ex*: 18 rubles = 10+5+2+1, 60 kopecks = 50+10
- Does greedy strategy always lead to an optimal solution?

  NO!
  - if we had 9 roubles in addition, then 18=9+9
  - if we had {50,20,2,1} kopecks, then 60=20+20+20 would be better than 60=50+2+2+2+2+2 obtained by greedy

# Average min number of coins

- In the USA, there are coins of 1¢ 5¢ 10¢ 25¢. Replacing 10¢ by 18¢ would improve the average min number of coins in the change from 4.7 to 3.89 (which is optimal for 4 denominations)
- If we had to (1,5,10,25) add another denomination, the best to add would be 32¢ (improves average to 3.46)

**Mathematical Entertainments** Michael Kleber and Ravi Vakil, Editors

## What This Country Needs Is an 18¢ Piece*

Jeffrey Shallit

Most businesses in the United States currently make change using four different types of coins: 1¢ (cent),[1] 5¢ (nickel), 10¢ (dime), and 25¢ (quarter). For people who make change on a daily basis, it is desirable to make change in as efficient a manner as possible. One criterion for efficiency is to use the smallest number of coins. For example, to make change for 30¢, one could, at least in principle, give a customer 30 1-cent coins, but most would probably prefer receiving a quarter and a nickel.

Formally, we can define the *optimal representation problem* as follows:

For the current system, where $(e_1, e_2, e_3, e_4) = (1, 5, 10, 25)$, a simple computation determines that cost(100; 1, 5, 10, 25) = 4.7. In other words, on average a change-maker must return 4.7 coins with every transaction.

Can we do better? Indeed we can. There are exactly two sets of four denominations that minimize cost(100; $e_1, e_2, e_3, e_4$); namely, (1, 5, 18, 25) and (1, 5, 18, 29). Both have an average cost of 3.89. We would therefore gain about 17% efficiency in change-making by switching to either of these four-coin systems. The first system, (1, 5, 18, 25),

# Coin changing: solution 1

- Assume we always have coins of $1$ ruble
- *Idea*: try all decompositions of N into two amounts, solve each amount, take minimum

$$opt(N) = \begin{cases} 1, & \text{if there exist coins of } N \\ \min\limits_{i=1..N-1} \{opt(i) + opt(N-i)\}, & \text{otherwise} \end{cases}$$

- Can be seen as divide-and-conquer or brute-force
- Exponential time (tree of recursive calls)

# Coin changing: solution 2

- *Idea*: try all possible coins and solve the difference; take minimum

$$opt(N) = \begin{cases} 0, & \text{if } N = 0 \\ \min\limits_{C_i \le N}\{1 + opt(N - C_i)\}, & \text{otherwise} \end{cases}$$

- still exponential time (but better than the previous solution)

# Coin changing: solution 2

- *Idea*: try all possible coins and solve the difference; take minimum

$$opt(N) = \begin{cases} 0, & \text{if } N = 0 \\ \min_{C_i \le N}\{1 + opt(N - C_i)\}, & \text{otherwise} \end{cases}$$

- still exponential time (but better than the previous solution)
- *memoization* ⇒ time $O(N \cdot K)$ where $K$ is the number of disctinct denominations

# Coin changing: solution 3

- *Idea* (*dynamic programming*): solve the problem for amounts $1, 2, \ldots, N$. For each amount, use solutions for smaller amounts.

$$
\begin{aligned}
&opt(0) = 0; \\
&\text{for } j = 1 \text{ to } N \text{ do} \\
&\qquad opt(j) = 1 + min_{C_i \leq j} opt(j - C_i)
\end{aligned}
$$

- Replacing recursion by *iteration over subproblems*. Time $O(N \cdot K)$, where $K$ is the number of distinct denominations

# Example

$C = \{1,2,5\}, N = 10$ $\qquad$ $opt(j) = 1 + min_{C_i \leq j} opt(j - C_i)$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 2 |

# Example

$C = \{1,2,5\}, N = 10$ $\qquad$ $opt(j) = 1 + min_{C_i \leq j} opt(j - C_i)$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 2 |

How to recover an optimal decomposition?

# Example

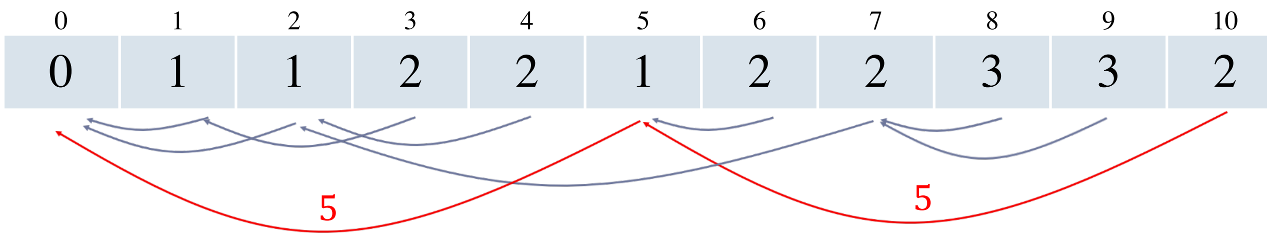
$C = \{1,2,5\}, N = 10 \qquad opt(j) = 1 + min_{C_i \leq j} opt(j - C_i)$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 3 | 3 | 2 |

How to recover an optimal decomposition?

# Example

$$C = \{1,2,5\}, N = 10 \qquad opt(j) = 1 + min_{C_i \leq j} opt(j - C_i)$$




How to recover an optimal decomposition?

# Coin changing: number of decompositions

- $\{1,10,25,50\}, N = 30$

- *Question*: how to count the number of all *distinct* decompositions?
- *Example*: $\{1,2,3\}, N = 5$

  $\{1,1,1,1,1\}, \{1,1,1,2\}, \{1,2,2\}, \{1,1,3\}, \{2,3\}$

# Coin changing: number of decompositions

- Assume denominations $\{C_1, C_2, \ldots, C_k\}$ in increasing order and $C_1 = 1$
- $NUM(n, j)$: number of decompositions of $n$ using coins $\{C_1, C_2, \ldots, C_j\}$
- main relation: $NUM(n, j) = NUM(n, j-1) + NUM(n - C_j, j)$

# Coin changing: number of decompositions

- Assume denominations $\{C_1, C_2, \dots, C_k\}$ in increasing order and $C_1 = 1$
- $NUM(n, j)$: number of decompositions of $n$ using coins $\{C_1, C_2, \dots, C_j\}$
- main relation: $NUM(n, j) = NUM(n, j-1) + NUM(n - C_j, j)$

$$NUM(n, j) = \begin{cases} 1, \text{if } j = 1 \text{ or } n = 0 \\ NUM(n, j-1), \text{elseif } n < C_j \\ NUM(n, j-1) + NUM\left(n - C_j, j\right) \text{ otherwise} \end{cases}$$

conditions are checked in order

final answer: $NUM(N, k)$

# Example

$C = \{1,2,3\}, N = 7$

$$NUM(n,j) = \begin{cases} 1, \text{if } j = 1 \text{ or } n = 0 \\ NUM(n, j-1), \text{elseif } n < C_j \\ NUM(n, j-1) + NUM\left(n - C_j, j\right) \text{ otherwise} \end{cases}$$

| | | N=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| $C_1 = 1$ | $j = 1$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $C_2 = 2$ | 2 | 1 | 1 | | | | | | |
| $C_3 = 3$ | 3 | 1 | 1 | | | | | | |

# Example

$C = \{1,2,3\}, N = 7$

$$NUM(n,j) = \begin{cases} 1, \text{if } j = 1 \text{ or } n = 0 \\ NUM(n, j-1), \text{elseif } n < C_j \\ NUM(n, j-1) + NUM(n - C_j, j) \text{ otherwise} \end{cases}$$

| | | N=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| $C_1 = 1$ | $j = 1$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $C_2 = 2$ | 2 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |
| $C_3 = 3$ | 3 | 1 | 1 | 2 | 3 | 4 | 5 | 7 | 8 |

{1,1,1,1,1,1,1}, {1,1,1,1,1,2}, {1,1,1,2,2}, {1,2,2,2},
{1,3,3}, {1,1,2,3}, {1,3,3}, {2,2,3},

# Similar approach works for *opt*

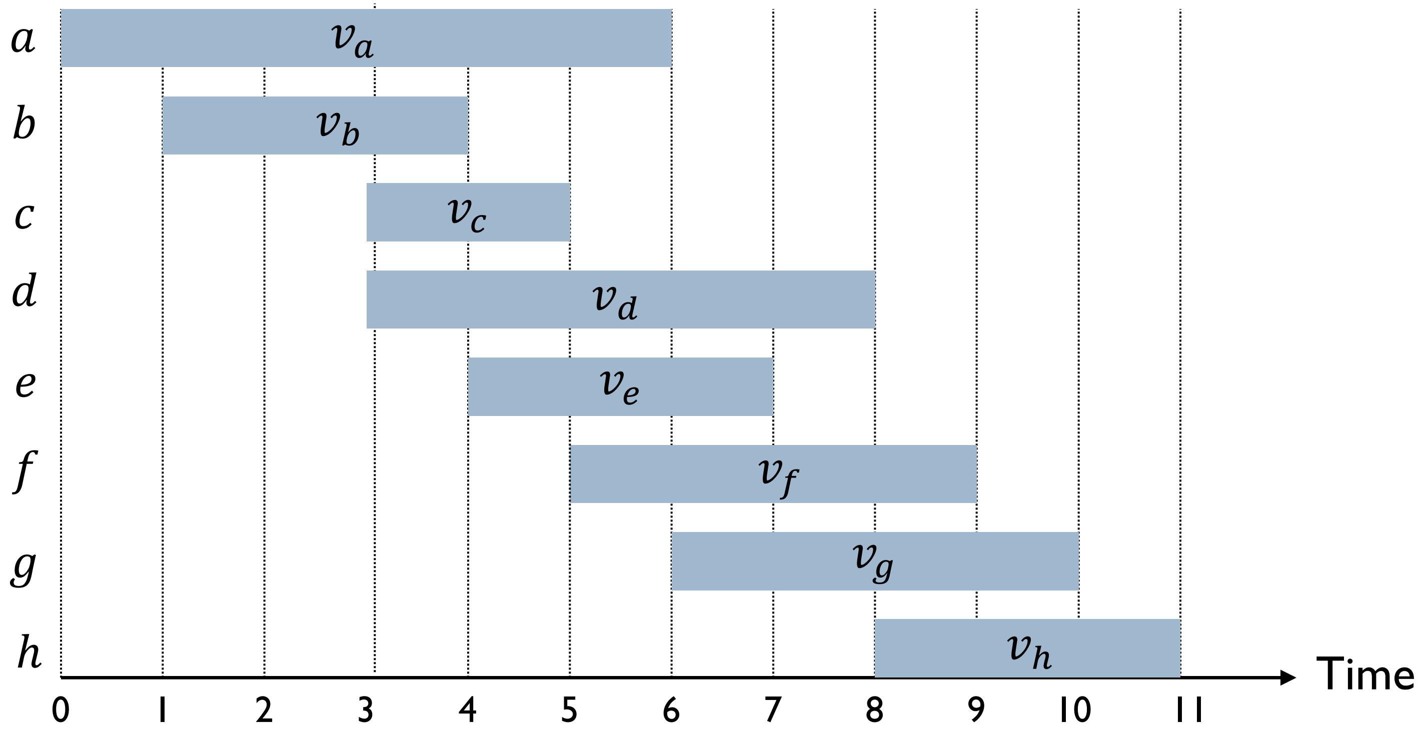
- Assume denominations $\{C_1, C_2, \dots, C_k\}$ in increasing order and $C_1 = 1$
- $opt(n, j)$: minimum number of coins needed to change $n$ using coins $\{C_1, C_2, \dots, C_j\}$

$$opt(n,j) = \begin{cases} n, \text{if } j = 1 \\ 1, \text{elseif } n = C_j \\ opt(n, j-1), \text{elseif } n < C_j \\ \max\{opt(n, j-1), opt\left(n - C_j, j\right)\} \text{ otherwise} \end{cases}$$

final answer: $opt(N, k)$

# (Weighed) interval scheduling problem

- Weighted interval scheduling problem.
  - Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
  - Two jobs compatible if they don't overlap
  - *Goal*: find maximum weight subset of mutually compatible jobs



# (Weighed) interval scheduling problem

- What about a greedy solution?
- Several possible greedy strategies:
  1. Choose the earliest starting next job
  2. Choose the shortest next job
  3. Choose the earliest finishing next job

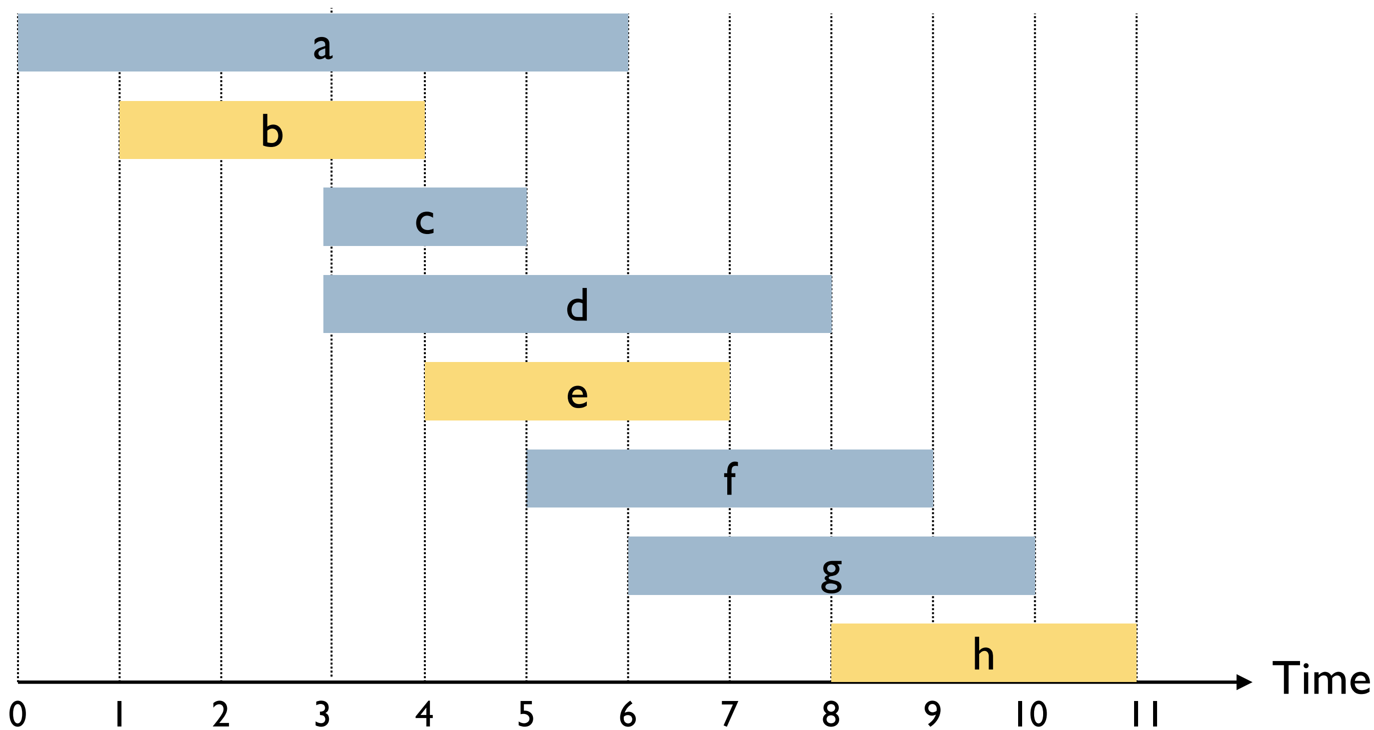
# (Weighed) interval scheduling problem

- What about a greedy solution?
- Several possible greedy strategies:
  1. Choose the earliest starting next job
  2. Choose the shortest next job
  3. Choose the earliest finishing next job

- Strategies 1 and 2 do not produce an optimal solution. Strategy 3 does but ***only if all intervals have equal weight***. Easy implementation:
  - Consider jobs in ascending order of finish time.
  - Go through the jobs and add a job to the current subset if it is compatible with previously chosen jobs.

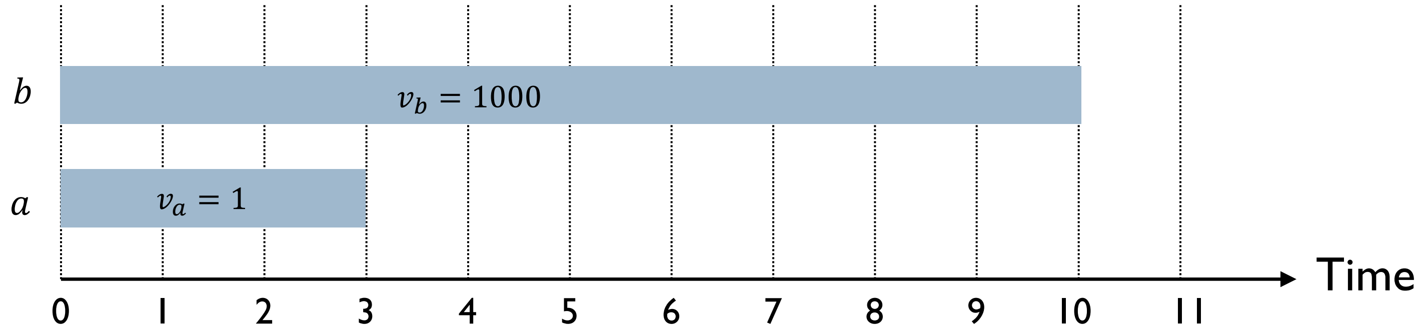
# (Weighed) interval scheduling problem

- *Remark*: Greedy algorithm works if all weights are $1$.
  - Consider jobs in ascending order of finish time.
  - Go through the jobs and add a job to the current subset if it is compatible with previously chosen jobs.



# (Weighed) interval scheduling problem

- *Observation*: Strategy 3 can fail spectacularly if arbitrary weights are allowed.

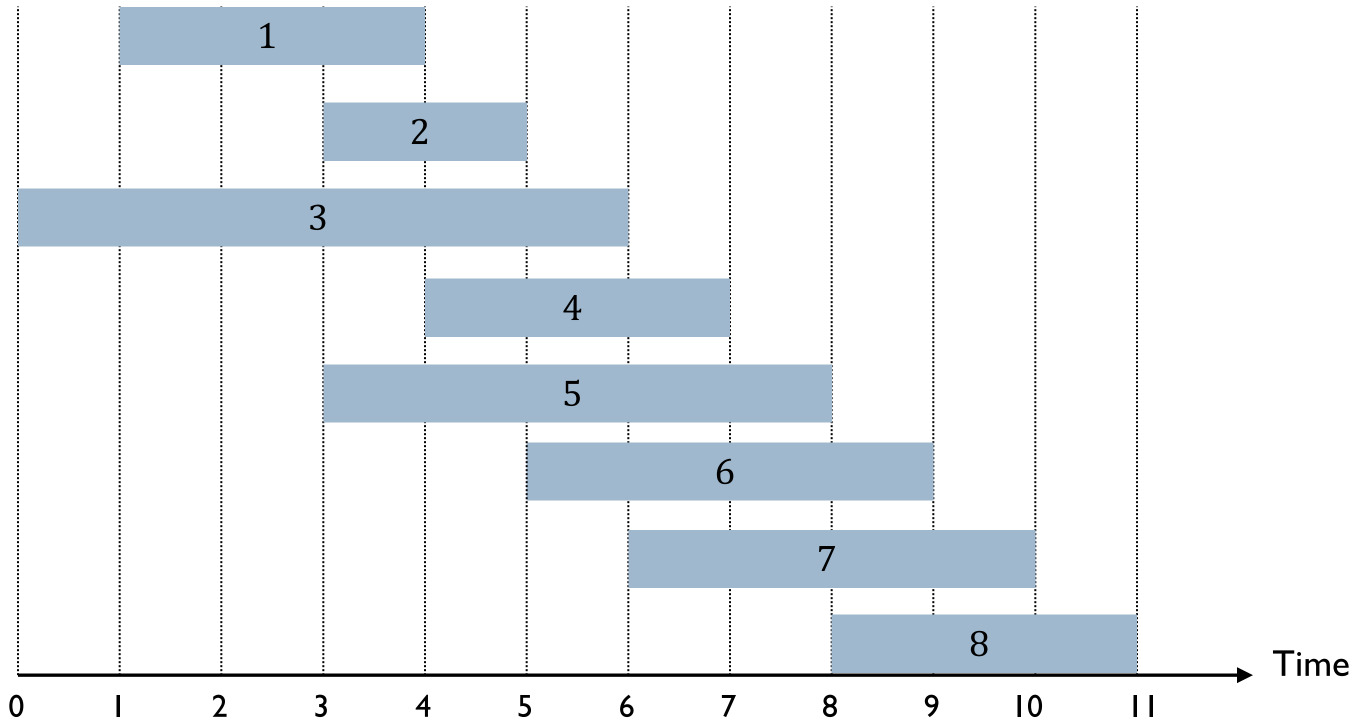


# (Weighed) interval scheduling problem

*Notation*: Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

*Def*: $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

*Ex*: $p(8) = 5, p(7) = 3, p(2) = 0.$



# (Weighed) interval scheduling problem

- *Notation*: *OPT*($j$) = value of optimal solution to the problem consisting of job requests $1, 2, ..., j$. Goal: compute *OPT*($n$)
  - *Case 1*: *OPT* selects job $j$.
    - can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
    - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$
  - *Case 2*: *OPT* does not select job $j$.
    - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, j - 1$

$$OPT(j) = \begin{cases} 0, \text{if } j = 0 \\ \max\{v_j + OPT\big(p(j)\big), OPT(j-1)\} \text{ otherwise} \end{cases}$$

# (Weighed) interval scheduling problem

- Bottom-up dynamic programming: compute OPT($i$) for $i$=1..$n$

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤
fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   OPT[0] = 0
   for j = 1 to n
      OPT[j] = max(vj + OPT[p(j)], OPT[j-1])
}
```

# (Weighed) interval scheduling problem

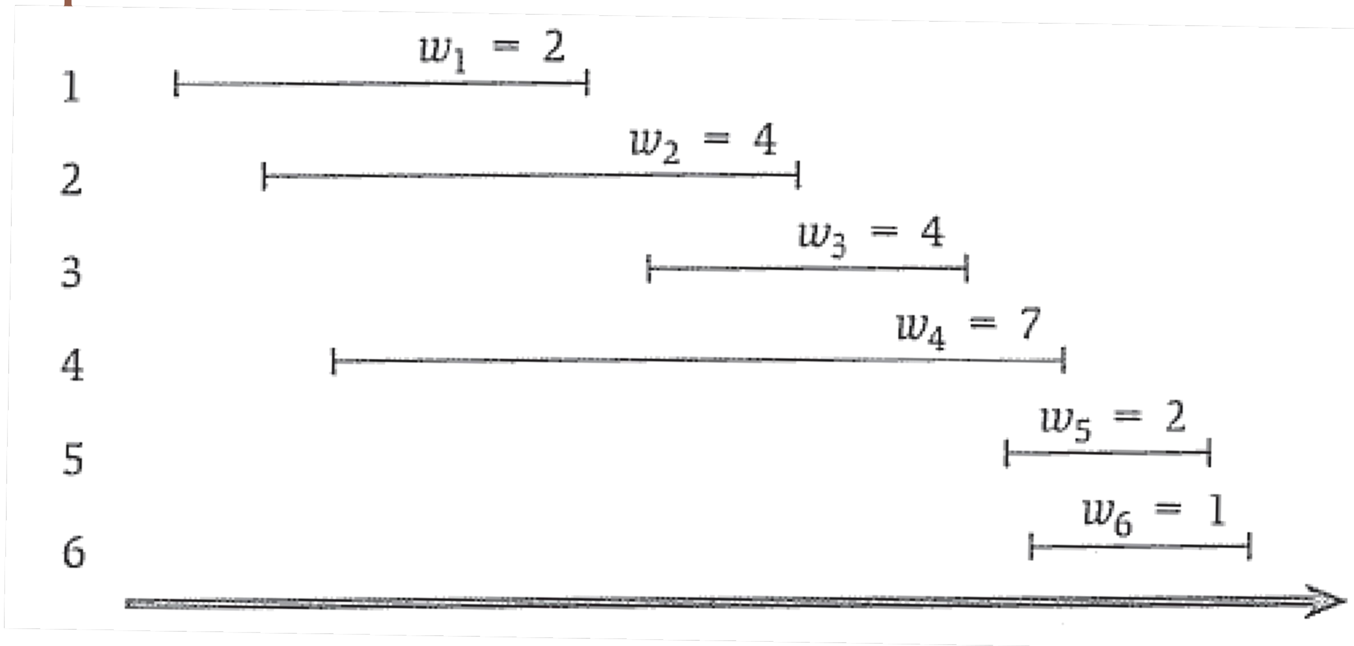
- *Memoization*: Store results of each sub-problem in an array; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   OPT[j] = empty
OPT[0] = 0

M-Compute-Opt(j) {
  if (OPT[j] is empty)
    OPT[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
}
```

# (Weighed) interval scheduling problem

- Time complexity:
  - Sort by finish time: $O(n \log n)$
  - Computing $p(\cdot)$: easy to compute in $O(n \log n)$ time (e.g. $n$ binary searches, one for each interval)
  - Dynamic programming: $O(n)$
  - Altogether: $O(n \log n)$

# Example

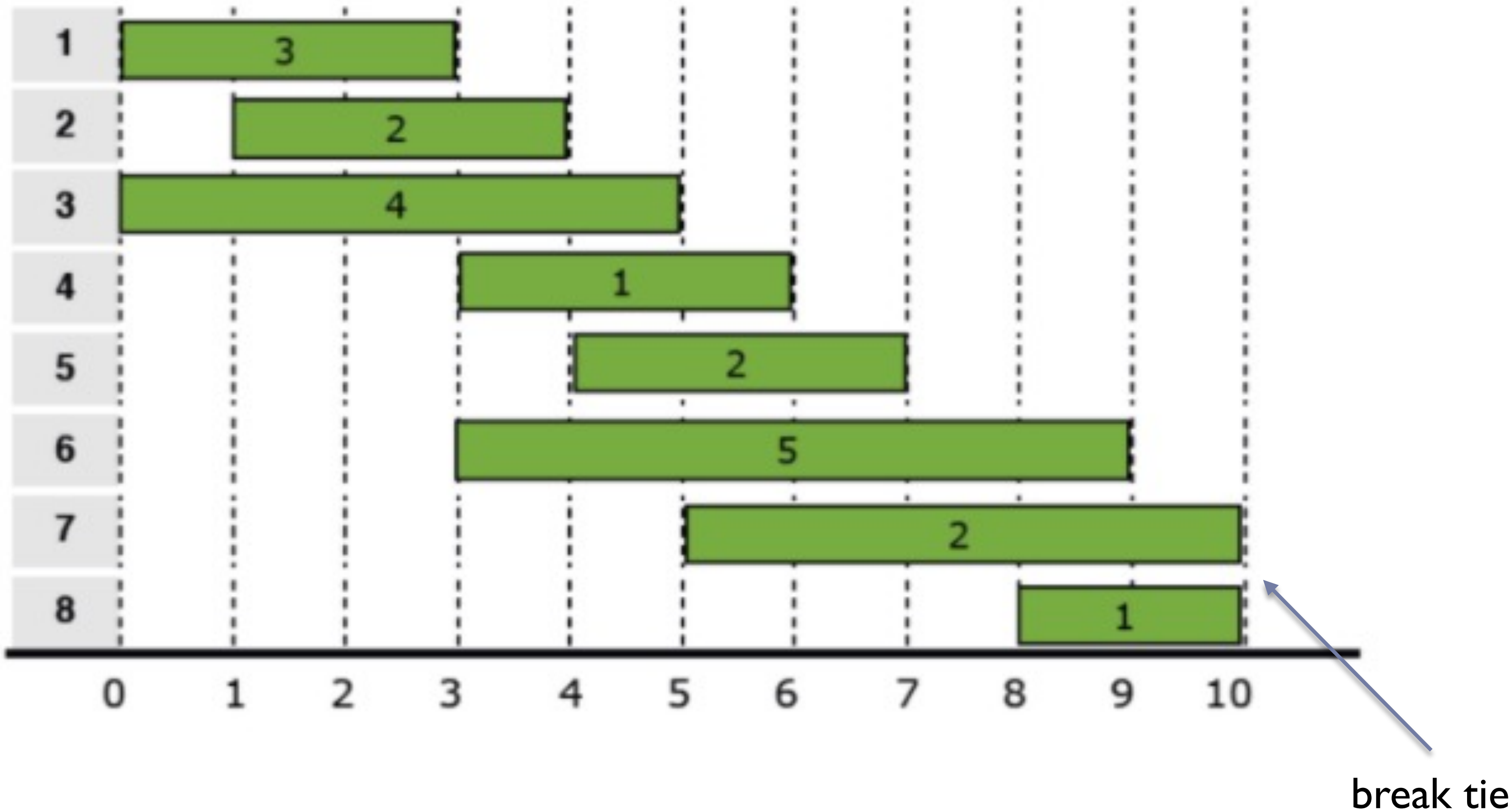



| $j$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $p(j)$ | 0 | 0 | 1 | 0 | 3 | 3 |

$$OPT(j) = \begin{cases} 0, \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} \text{ otherwise} \end{cases}$$

# Quiz 6.1

Solve the interval scheduling problem for the following case. Report the maximal sum of weights of scheduled jobs.



# Dynamic Programming scenario

- Analyse the structure of an optimal solution
  - usually to an optimization problem
- Express an optimal solution to an instance via optimal solutions to "smaller" instances
  - usually leads to a recursive relation whose direct application usually leads to exponential time
- Iterate through the instances from "smallest" to "largest" ("bottom-up") until obtaining the solution to the desired instance
- Construct the solution from computed information

# Some history

- Richard Bellman (1920-1984) pioneered the systematic study of Dynamic Programming in the 1950s
- Etymology
  - Dynamic programming = planning over time
  - Secretary of defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

  from [Bellman, R., *Eye of the Hurricane, An Autobiography*]

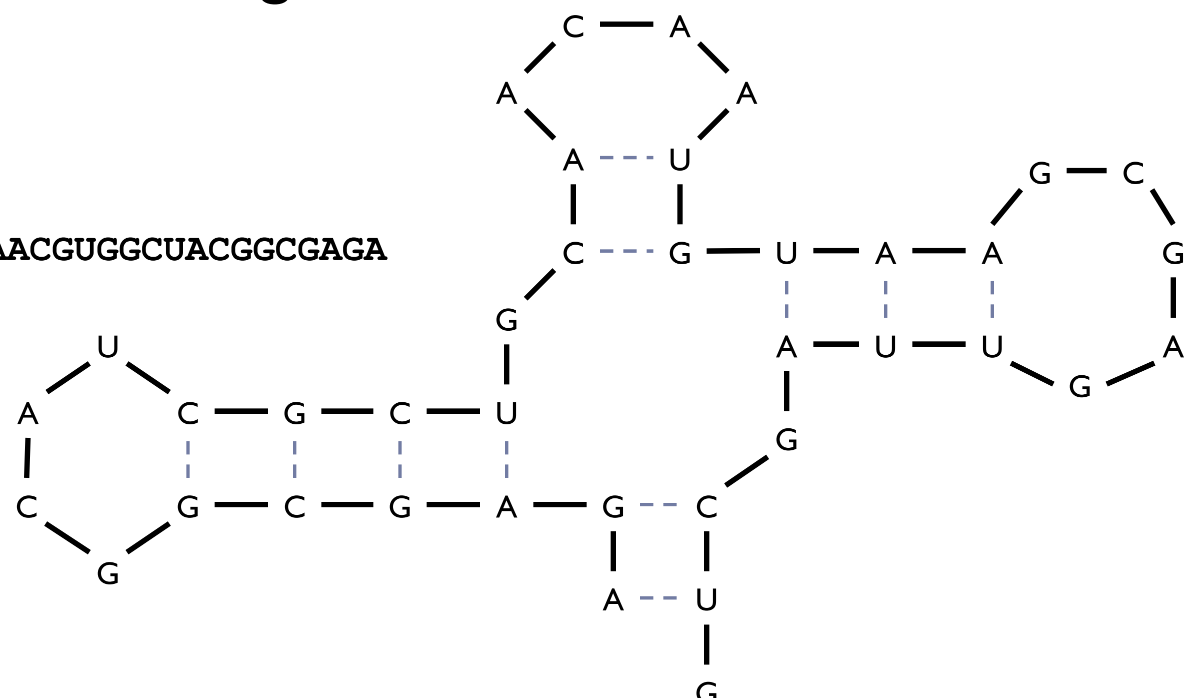
# RNA Secondary Structure

Dynamic Programming over intervals

# RNA Secondary Structure

- RNA: String $B = b_1 b_2 \ldots b_n$ over alphabet { 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.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



complementary base pairs: A-U, C-G

# 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 $j - i > 4$
  - [Non-crossing.] If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, then we cannot have $i < k < j < l$

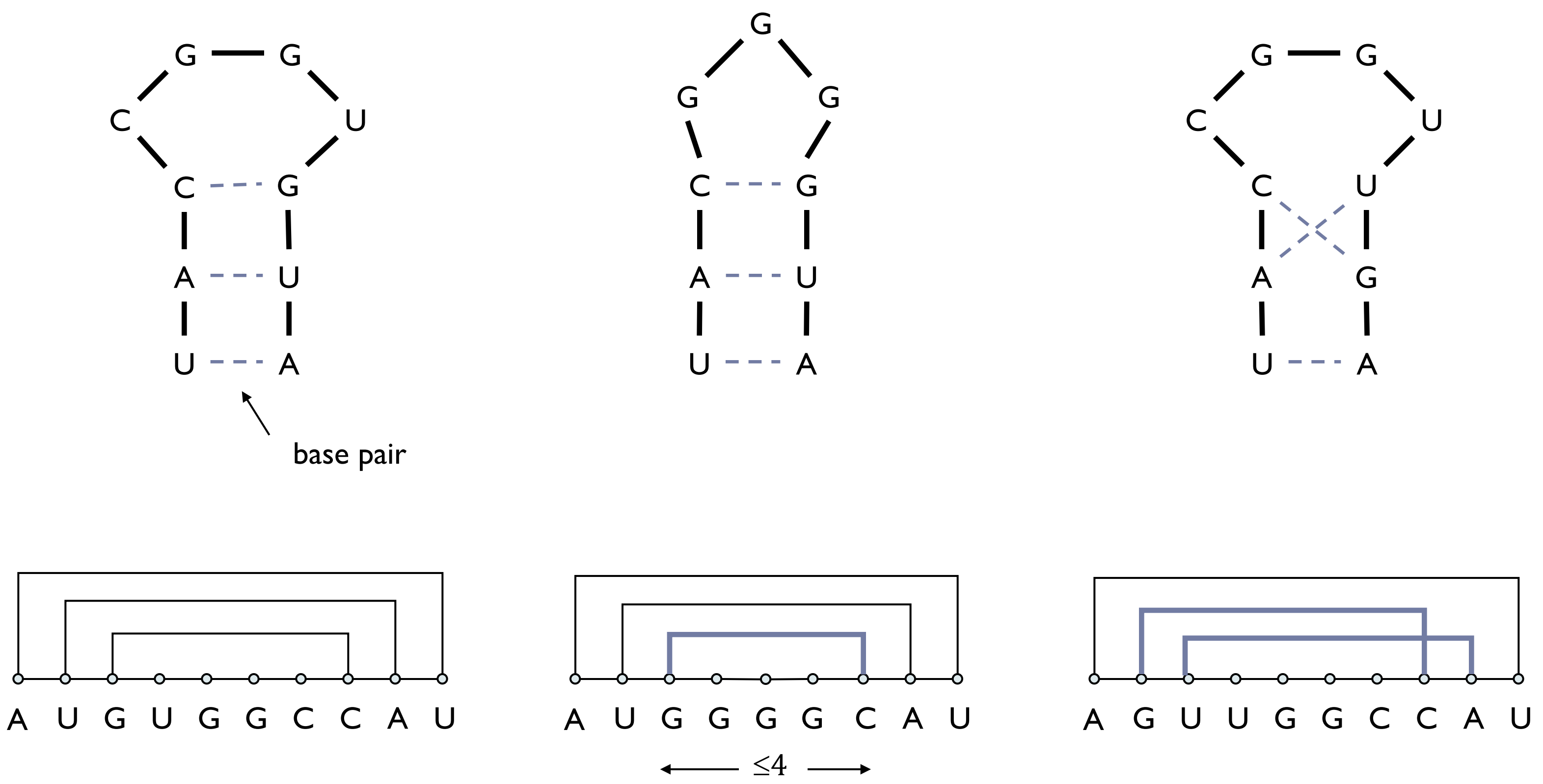
- *Free energy*: Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy

approximated by max number of unpaired bases

- *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

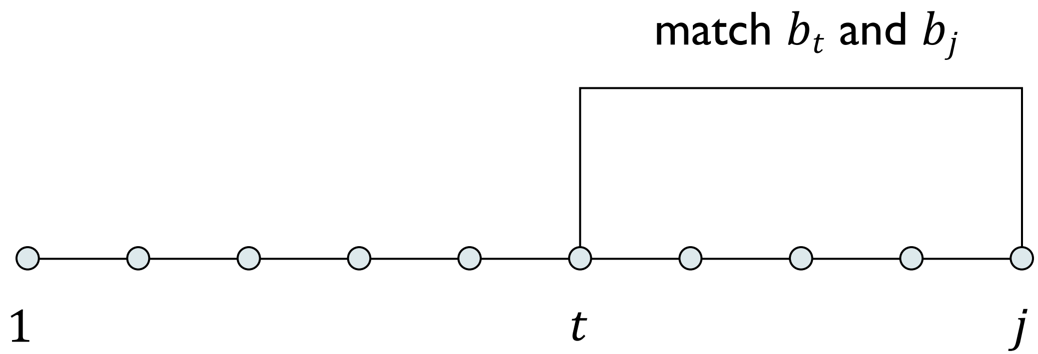
- *Examples*:




ok | sharp turn | crossing

# RNA Secondary Structure: Subproblems

- *First attempt*: $OPT(j)$ = maximum number of base pairs in a secondary structure of the prefix $b_1 b_2 \ldots b_j$




- *Difficulty*: Results in two sub-problems:
  - Finding secondary structure in: $b_1 b_2 \ldots b_{j-1}$ $\leftarrow$ $OPT(t-1)$
  - Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$ and $b_{t+1} b_{t+2} \ldots b_{j-1}$

need more sub-problems

# 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.
  - $OPT(i, j) \ = \ OPT(i, j - 1)$

- Case 3. Base $b_j$ pairs with $b_t$ for some $i \leq t \ < j - 4$
  - non-crossing constraint decouples resulting sub-problems
  - $OPT(i, j) = 1 + max_t\{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$

*max* over $t$ such that $i \leq t \ < j - 4$ and
$b_t$ and $b_j$ are Watson-Crick complements

# DP relation: summary

$$OPT(i,j) = \begin{cases} 0, \text{if } i \geq j-4 \\ \max\{1 + \max_t \{OPT(i,t-1) + OPT(t+1,j)\}, OPT(i,j-1)\} \end{cases}$$

max over $t$ such that $i \leq t < j - 4$ and
$b_t$ and $b_j$ are Watson-Crick complements

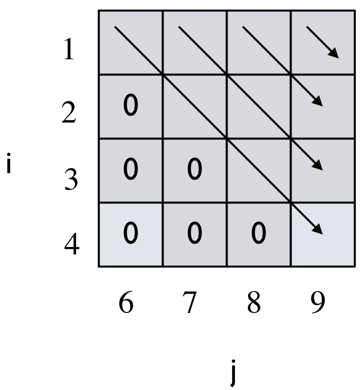
# Bottom Up Dynamic Programming Over Intervals

What order to solve the sub-problems?

*Possible order*: Shortest intervals first

```
RNA(b_1 … b_n) {
    for k = 5, 6, …, n-1
        for i = 1, 2, …, n-k
            j = i + k
            Compute OPT(i,j)

    return OPT
}
```

using recurrence




*Running time*: $O(n^3)$

# Example

$$\begin{array}{l} \quad\quad\;\; \scriptstyle 1\;2\;3\;4\;5\;6\;7\;8\;9 \\ B = \texttt{ACCGGUAGU} \end{array}$$

| | $j = 6$ | 7 | 8 | 9 |
|---|---|---|---|---|
| $i = 1$ | | | | |
| 2 | 0 | | | |
| 3 | 0 | 0 | | |
| 4 | 0 | 0 | 0 | |

# Example

$$B = \overset{1}{\texttt{A}}\overset{2}{\texttt{C}}\overset{3}{\texttt{C}}\overset{4}{\texttt{G}}\overset{5}{\texttt{G}}\overset{6}{\texttt{U}}\overset{7}{\texttt{A}}\overset{8}{\texttt{G}}\overset{9}{\texttt{U}}$$

| | $j = 6$ | 7 | 8 | 9 |
|---|---|---|---|---|
| $i = 1$ | 1 | | | |
| 2 | 0 | 0 | | |
| 3 | 0 | 0 | 1 | |
| 4 | 0 | 0 | 0 | 0 |

# Example

$$\begin{array}{cccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ B = & \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{U} & \mathtt{A} & \mathtt{G} & \mathtt{U} \end{array}$$

| | $j = 6$ | 7 | 8 | 9 |
|---|---|---|---|---|
| $i = 1$ | 1 | 1 | | |
| 2 | 0 | 0 | 1 | |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

# Example

$$B = \begin{array}{ccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 \\ \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{U} & \mathtt{A} & \mathtt{G} & \mathtt{U} \end{array}$$

| | $j = 6$ | 7 | 8 | 9 |
|---|---|---|---|---|
| $i = 1$ | 1 | 1 | 1 | |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

# Example

$$B = \begin{array}{ccccccccc} \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 \\ \mathtt{A} & \mathtt{C} & \mathtt{C} & \mathtt{G} & \mathtt{G} & \mathtt{U} & \mathtt{A} & \mathtt{G} & \mathtt{U} \end{array}$$

| | $j = 6$ | 7 | 8 | 9 |
|---|---|---|---|---|
| $i = 1$ | 1 | 1 | 1 | 2 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 |

# Quiz 6.2

- Which of the following orders of processing subproblems $(i, j)$ allows a correct computation of values $OPT(i, j)$?
  - Increasing lexicographic order by pairs $(-i, j)$ (sorting by decreasing left end)
  - Increasing lexicographic order of pairs $(j - i, i)$ (from shortest to longest)
  - Increasing lexicographic order of pairs $(j, i)$ (sorting by right end)
  - Increasing lexicographic order of pairs $(i, j)$ (sorting by left end)