# CSE 421

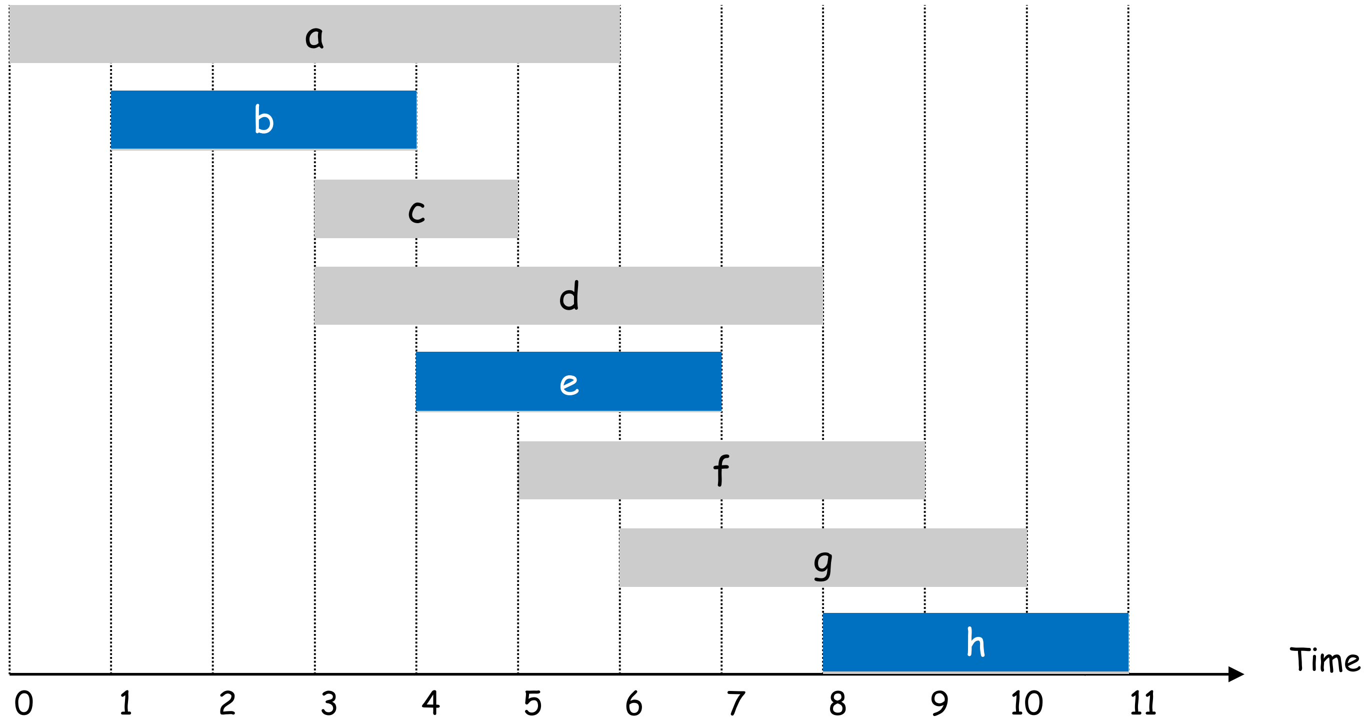
## Weighted Interval Scheduling, Knapsack, RNA Secondary Structure

Shayan Oveis Gharan

# Weighted Interval Scheduling

# Interval Scheduling

- Job j starts at $s(j)$ and finishes at $f(j)$ and has weight $w_j$
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



# Weighted Job Scheduling by Induction

Suppose $1, \dots, n$ are a

IH (strong ind): S
for $< n$ jobs.

This idea works for any
Optimization problem, e.g., vertex-
cover, independent set, etc.
For NP-hard problems there is no
ordering to reduce # subproblems

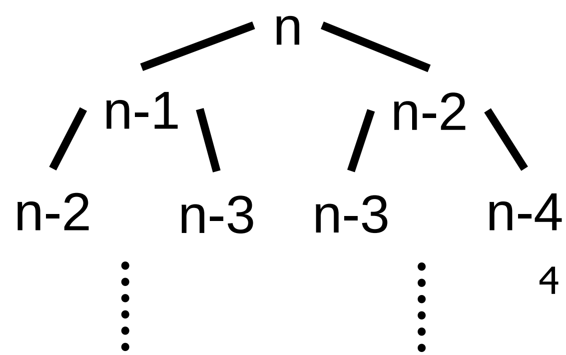
IS: Goal: For any n jobs we ca compu

Case 1: Job n is not in OPT.
-- Then, just return OPT of $1, \dots, n-1$.

Take best of the two

Case 2: Job n is in OPT.
-- Then, delete all jobs not compatible with n and recurse.

Q: Are we done?
A: No, How many subproblems are there?
Potentially $2^n$ all possible subsets of jobs.

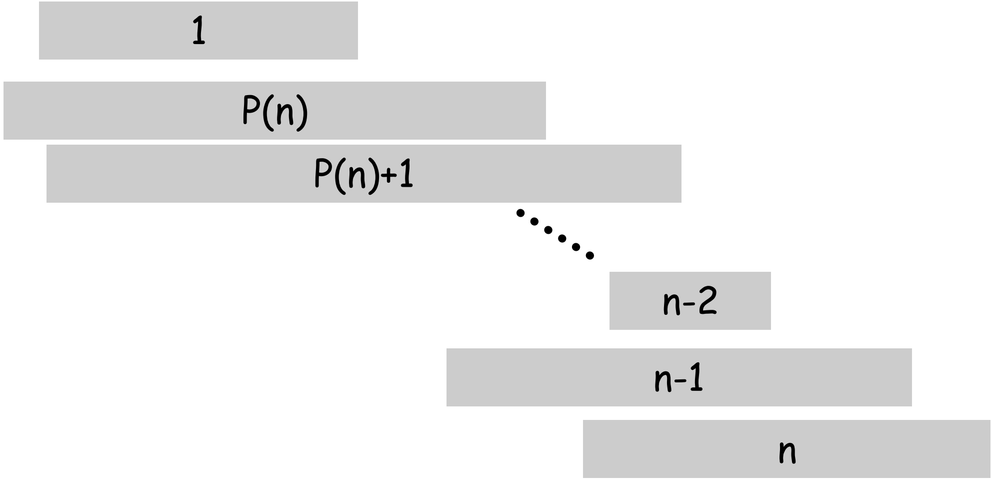


# Sorting to Reduce Subproblems

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$
IS: For jobs 1,…,n we want to compute OPT

Case 1: Suppose OPT has job n.

- So, all jobs i that are not compatible with n are not OPT
- Let p(n) = largest index i < n such that job i is compatible with n.
- Then, we just need to find OPT of $1, \ldots, p(n)$



# Sorting to reduce Subproblems

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$
IS: For jobs 1,…,n we want to compute OPT

Case 1: Suppose OPT has job n.

- So, all jobs i that are not compatible with n are not OPT
- Let p(n) = largest index i < n such that job i is compatible with n.
- Then, we just need to find OPT of $1, \ldots, p(n)$

Case 2: OPT does not select job n.

- Then, OPT is just the OPT of $1, \ldots, n-1$

Take best of the two

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form $1, \ldots, i$ for some $i$
So, at most $n$ possible subproblems.

# Weighted Job Scheduling by Induction

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

Def OPT(j) denote the OPT solution of $1, \ldots, j$

The most important part of a correct DP; It fixes IH

To solve OPT(j):

Case 1: OPT(j) has job j.

- So, all jobs i that are not compatible with j are not OPT(j)
- Let p(j) = largest index i < j such that job i is compatible with j.
- So $OPT(j) = OPT\big(p(j)\big) \cup \{j\}$.

Case 2: OPT(j) does not select job j.

- Then, $OPT(j) = OPT(j-1)$

$$\boxed{OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\Big(w_j + OPT\big(p(j)\big), OPT(j-1)\Big) & \text{o.w.} \end{cases}}$$

# Algorithm

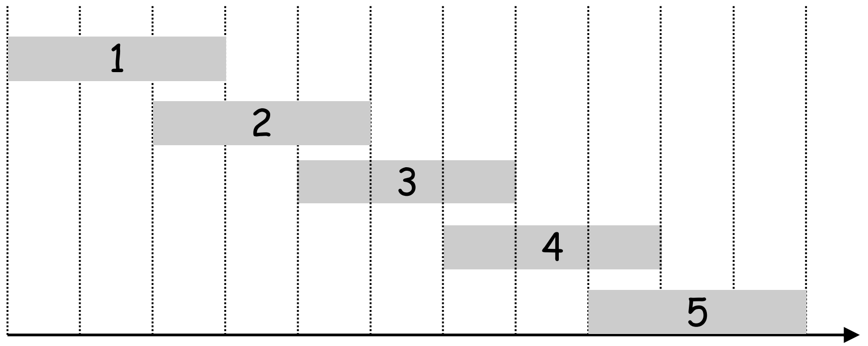
```
Input: n, s(1),...,s(n)  and f(1),...,f(n) and w_1,...,w_n.

Sort jobs by finish times so that f(1) ≤ f(2) ≤ ··· f(n).

Compute p(1),p(2),...,p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(w_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

# Recursive Algorithm Fails

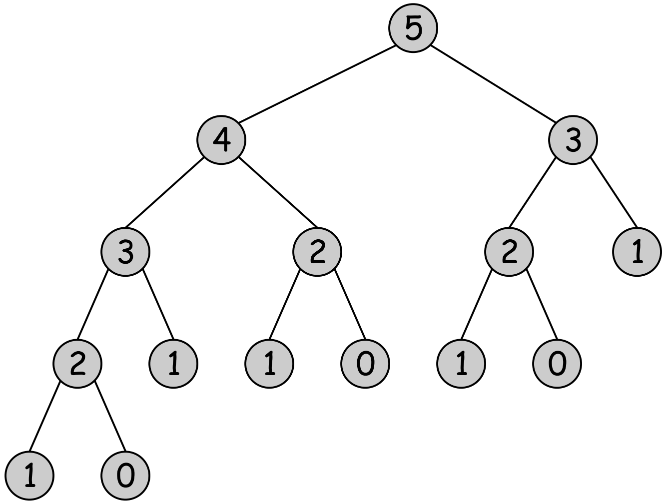
Even though we have only n subproblems, we do not store the solution to the subproblems

- So, we may re-solve the same problem many many times.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence




p(1) = 0, p(j) = j-2



# Algorithm with Memorization

Memoization. Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

```
Input: n, s(1),...,s(n)  and f(1),...,f(n) and w_1,...,w_n.

Sort jobs by finish times so that f(1) ≤ f(2) ≤ ⋯ f(n).

Compute p(1),p(2),...,p(n)

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

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

# Bottom up Dynamic Programming

You can also avoid recusion
- recursion may be easier conceptually when you use induction

**Input**: n, $s(1),\dots,s(n)$ and $f(1),\dots,f(n)$ and $w_1,\dots,w_n$.

**Sort** jobs by finish times so that $f(1) \le f(2) \le \cdots f(n)$.

**Compute** $p(1), p(2), \dots, p(n)$

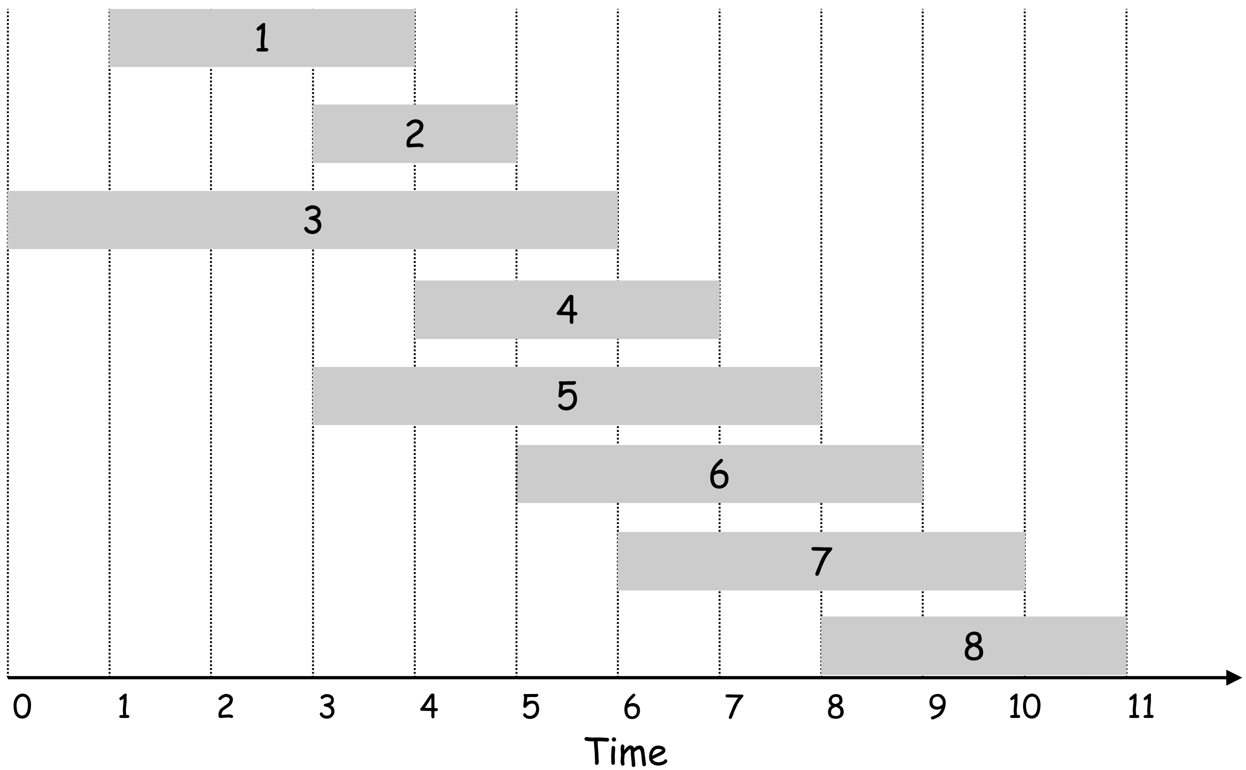
```
Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(w_j + M[p(j)], M[j-1])
}

Output M[n]
```

Claim: M[j] is value of OPT(j)
Timing: Easy.  Main loop is O(n); sorting is O(n log n)

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

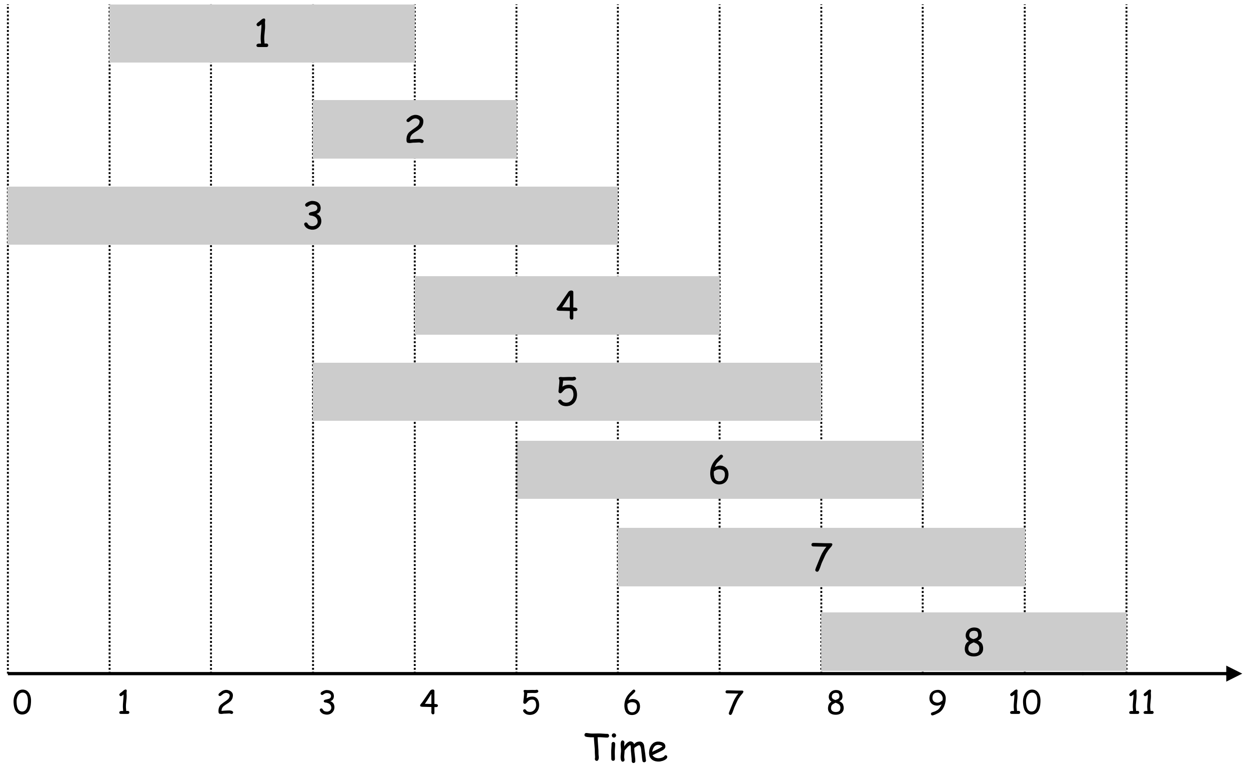



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | |
| 2 | 4 | 0 | |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

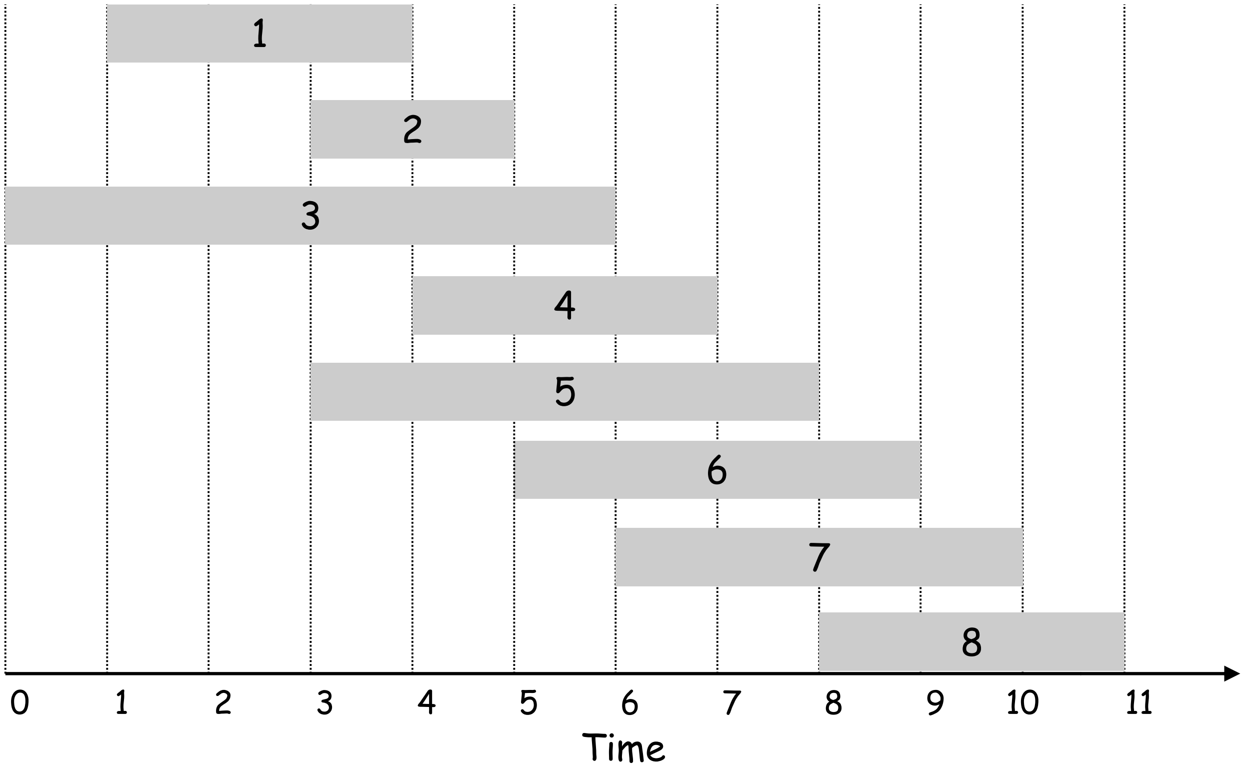



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

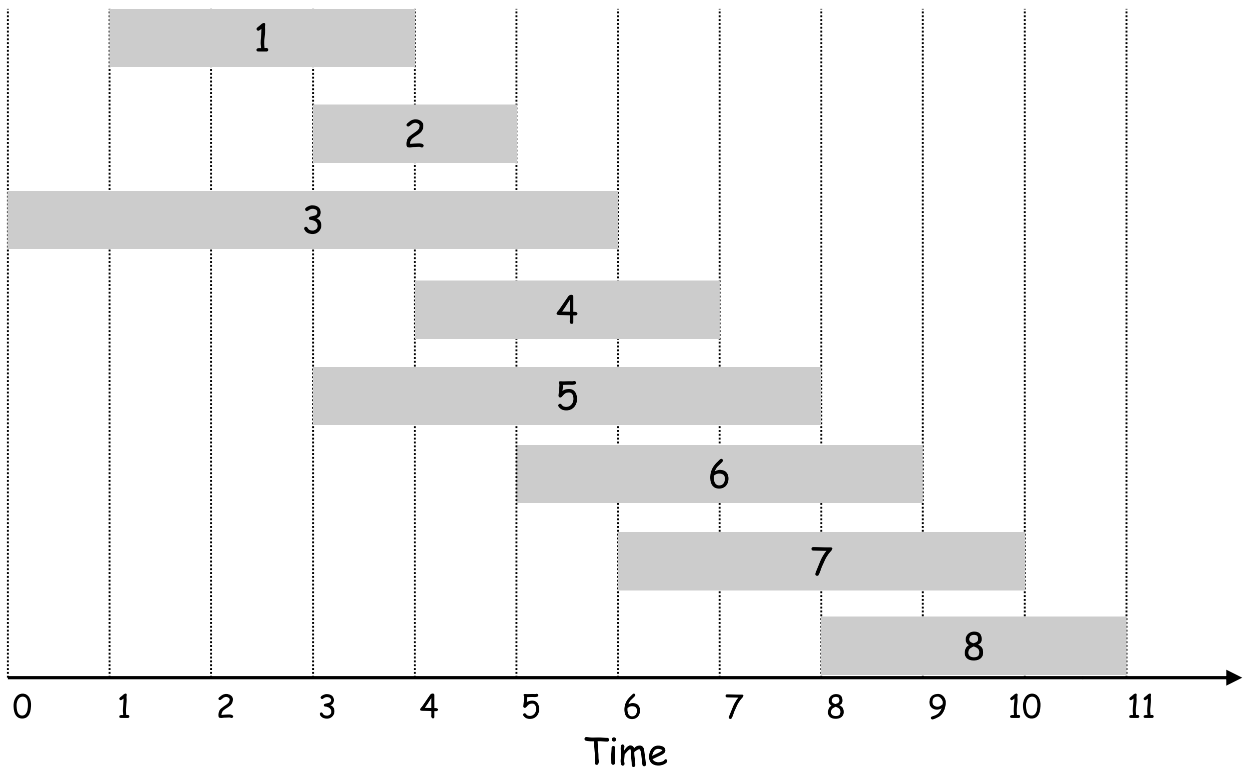



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | **4** |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

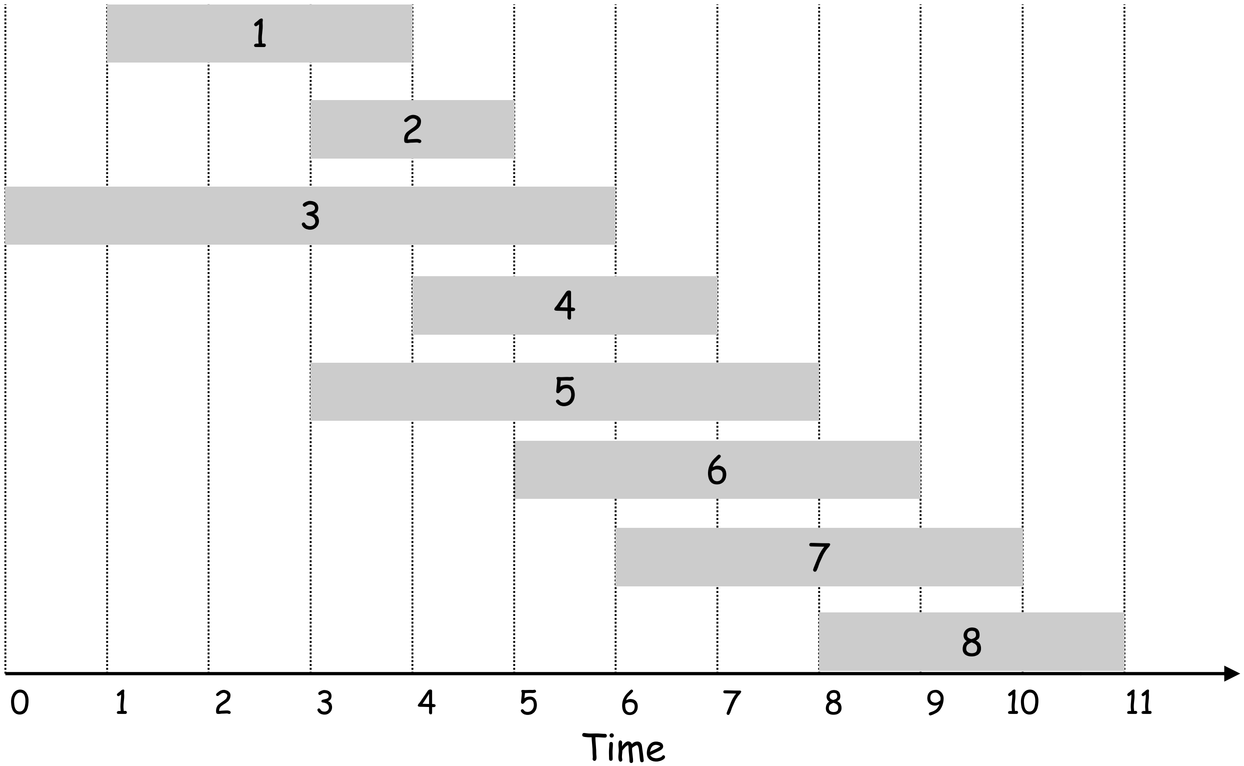



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | **4** |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

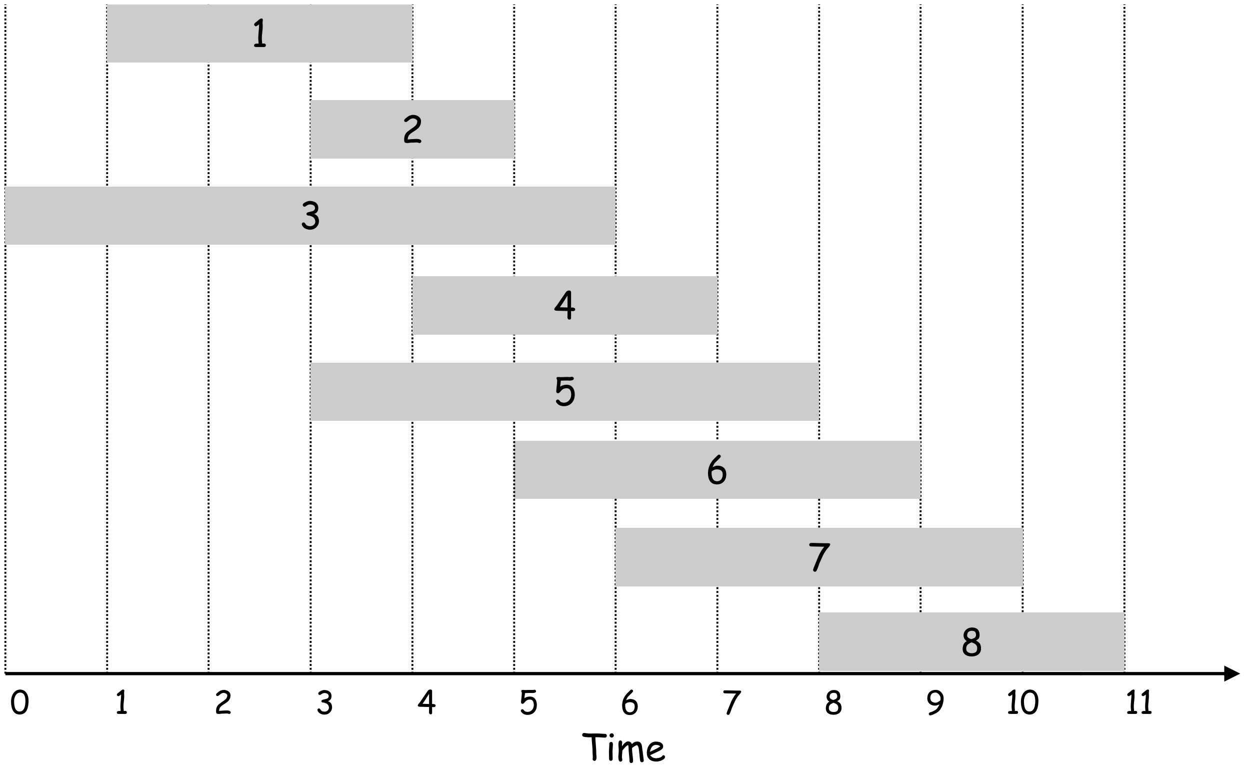



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

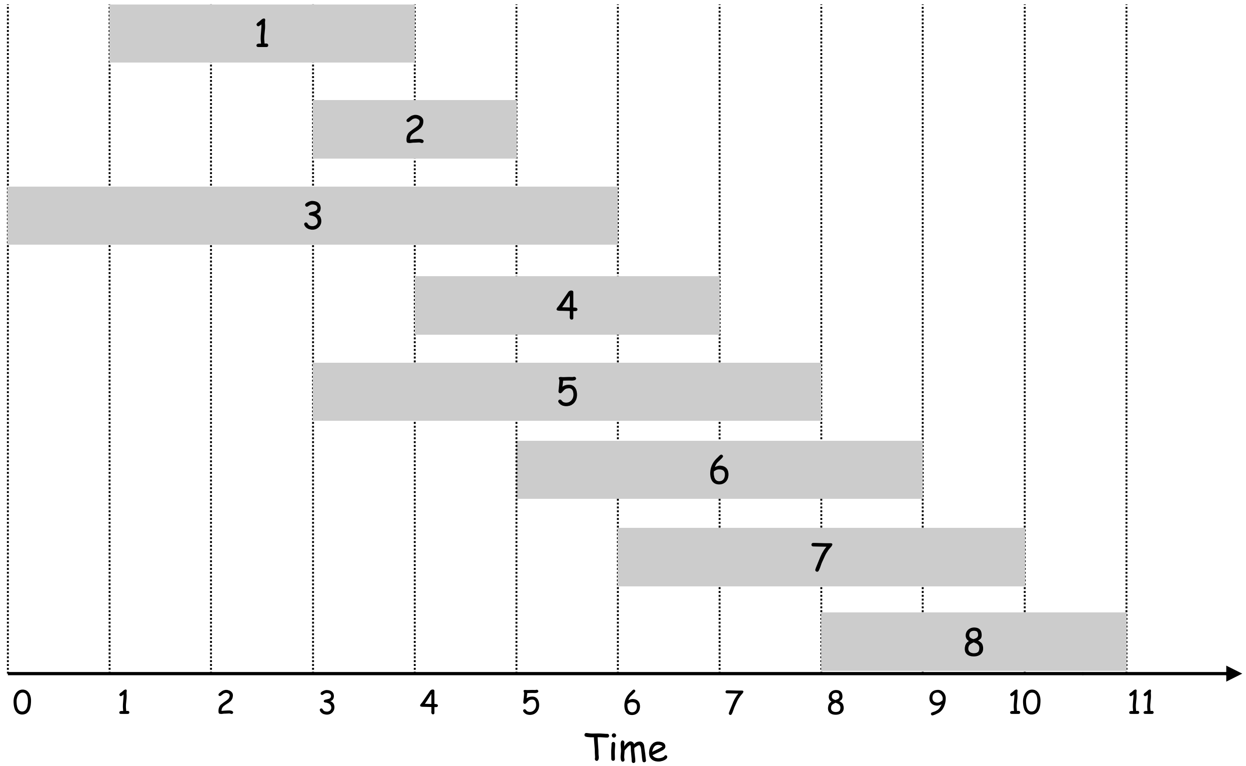



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | **6** |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

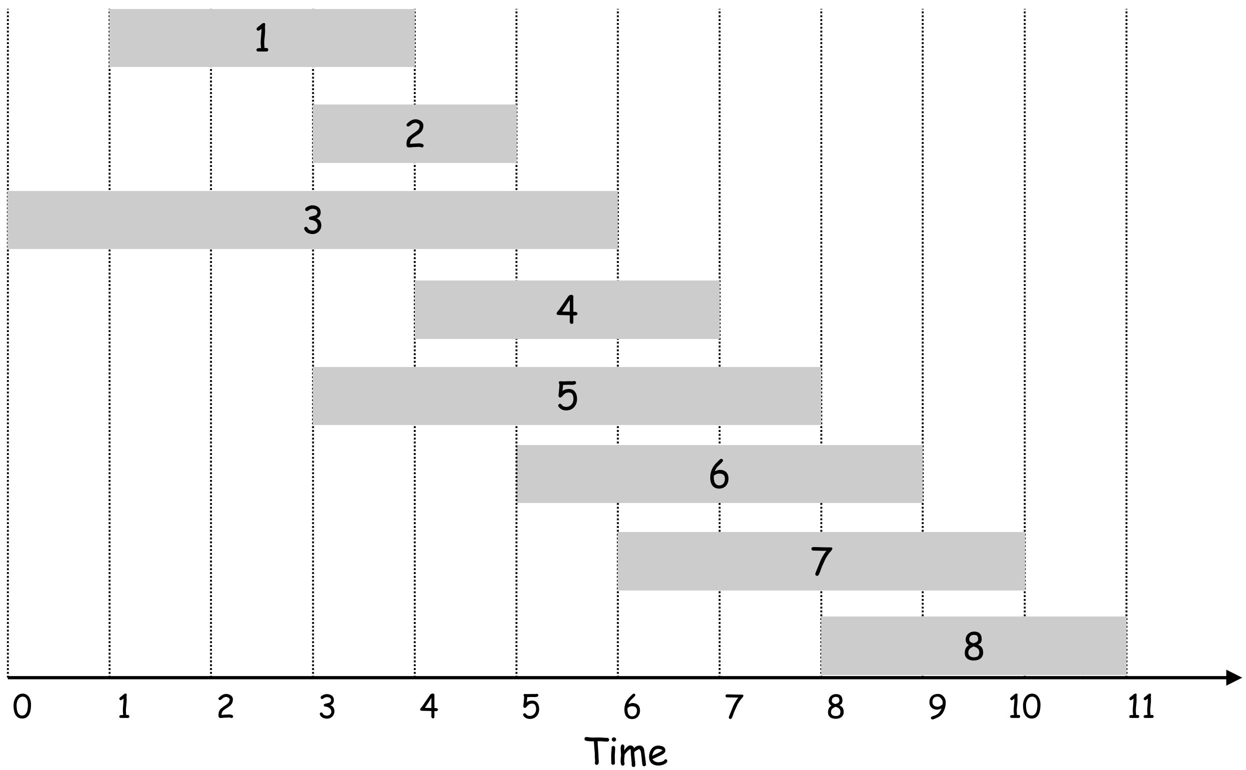



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

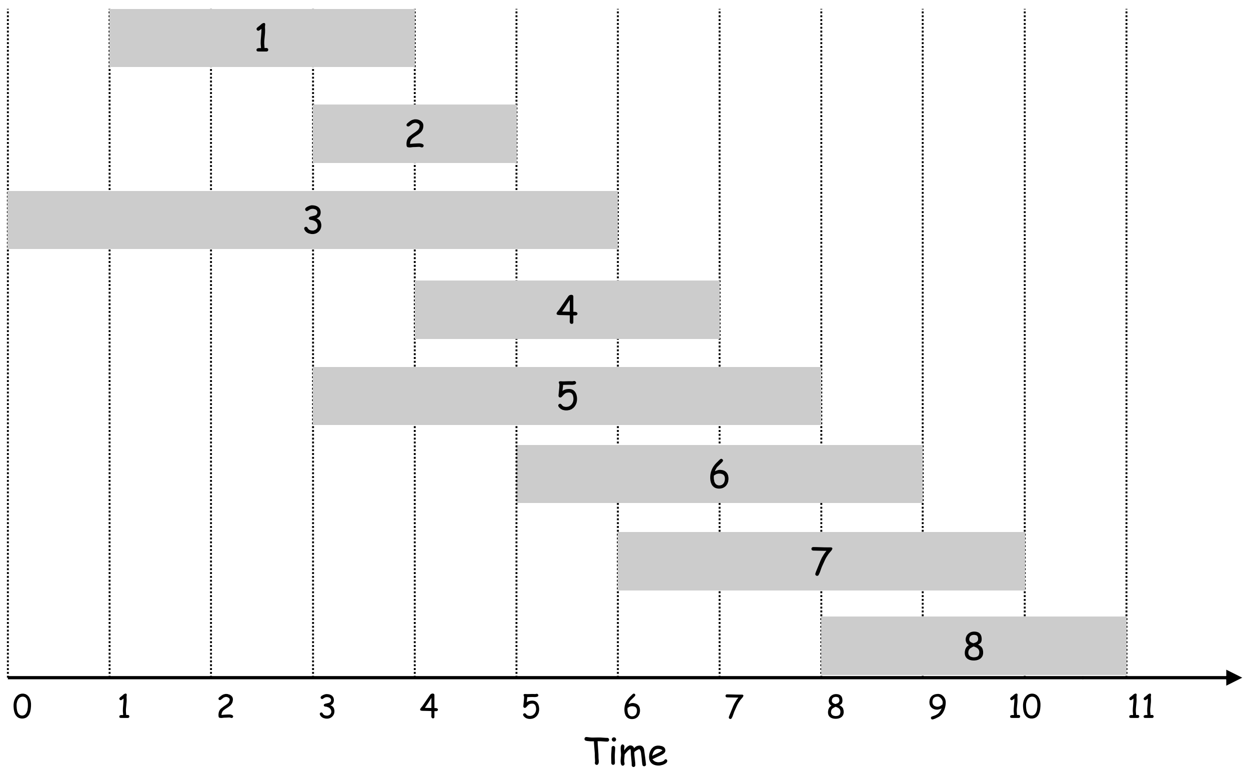



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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

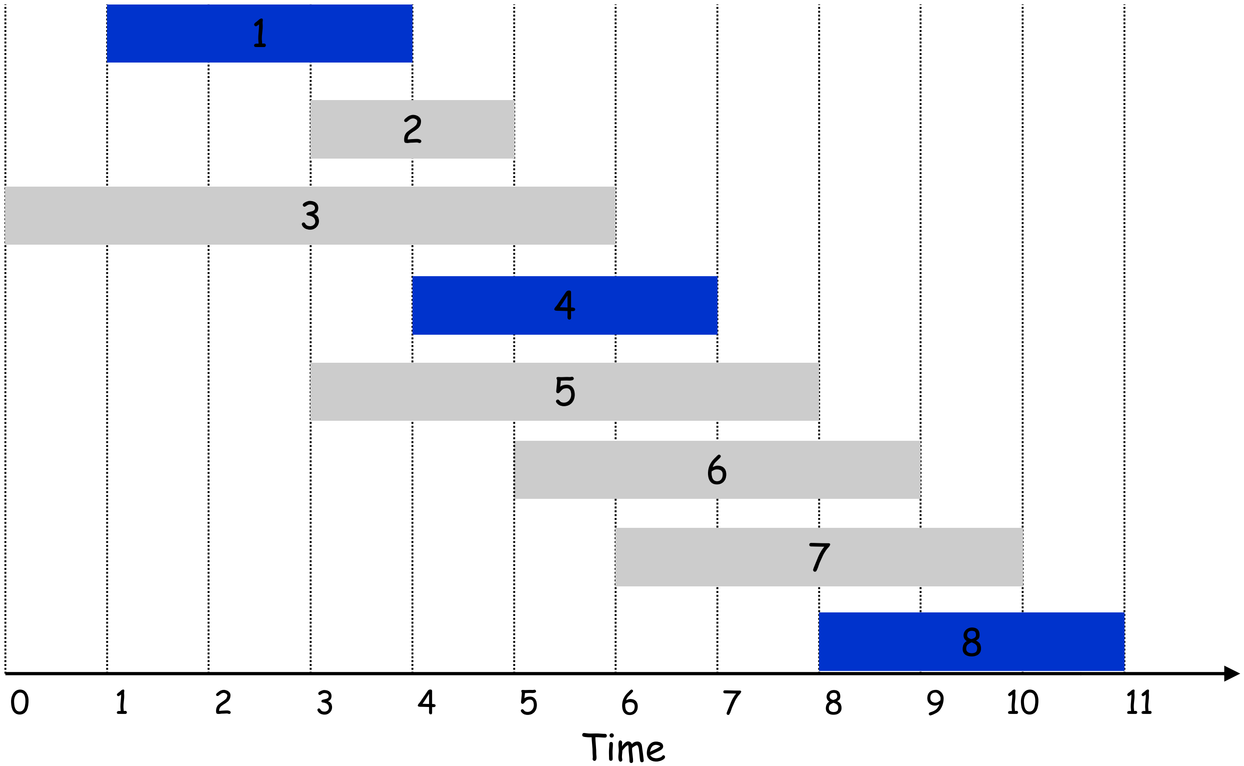



| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | **10** |

# Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

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




| j | $w_j$ | p(j) | OPT(j) |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | **10** |

# Knapsack Problem

# Knapsack Problem

Given n objects and a "knapsack.“

Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.

Knapsack has capacity of W kilograms.

Goal: fill knapsack so as to maximize total value.

Ex: OPT is { 3, 4 } with value 40.

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

# Dynamic Programming: First Attempt

Let OPT(i)=Max value of subsets of items $1, \ldots, i$ of weight $\leq\ W$.

Case 1: $OPT(i)$ does not select item i

- In this caes $OPT(i) = OPT(i-1)$

Case 2: $OPT(i)$ selects item $i$

- In this case, item $i$ does not immediately imply we have to reject other items
- The problem does not reduce to $OPT(i-1)$ because we now want to pack as much value into box of weight $\leq W - w_i$

Conclusion: We need more subproblems, we need to strengthen IH.

# Stronger DP (Strengthenning Hypothesis)

$0 \le w \le W$

Let $OPT(i, w)$ = Max value of subsets of items $1, \ldots, i$ of weight $\le w$

Case 1: $OPT(i, w)$ selects item $i$

- In this case, $OPT(i, w) = v_i + OPT(i-1, w-w_i)$

$w \ge w_i$

Take best of the two

Case 2: $OPT(i, w)$ does not select item $i$

- In this case, $OPT(i, w) = OPT(i-1, w)$.

Therefore,

$$OPT(i, w) = \begin{cases} 0 & \text{If } i = 0 \\ OPT(i-1, w) & \text{If } w_i > w \\ \max(OPT(i-1, w), v_i + OPT(i-1, w-w_i)) & \text{o.w.,} \end{cases}$$

# DP for Knapsack

```
Compute-OPT(i,w)
   if M[i,w] == empty
     if (i==0)
       M[i,w]=0
     else if (wi > w)
       M[i,w]=Comp-OPT(i-1,w)
     else
       M[i,w]= max {Comp-OPT(i-1,w), vi + Comp-OPT(i-1,w-wi)}
   return M[i, w]
```

recursive

```
for w = 0 to W
   M[0, w] = 0
for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Non-recursive

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| φ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | | | | | | | | | | | |
| { 1, 2 } | 0 | | | | | | | | | | | |
| { 1, 2, 3 } | 0 | | | | | | | | | | | |
| { 1, 2, 3, 4 } | 0 | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | 0 | | | | | | | | | | | |

W = 11

```
if (wi > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
```

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ϕ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | | | | | | | | | | | |
| { 1, 2, 3 } | 0 | | | | | | | | | | | |
| { 1, 2, 3, 4 } | 0 | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | 0 | | | | | | | | | | | |

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| φ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | | | | | | | | |
| { 1, 2, 3 } | 0 | 1 | | | | | | | | | | |
| { 1, 2, 3, 4 } | 0 | 1 | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | 0 | 1 | | | | | | | | | | |

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | | | | | |
| { 1, 2, 3, 4 } | 0 | 1 | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | 0 | 1 | | | | | | | | | | |

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | | |
| { 1, 2, 3, 4, 5 } | 0 | 1 | | | | | | | | | | |

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

# DP for Knapsack

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| { 1, 2, 3, 4, 5 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 34 | 40 |

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

# Knapsack Problem: Running Time

Running time: $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial."“
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:

There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum

in time Poly(n, log W).

# DP Ideas so far

- You may have to define an ordering to decrease #subproblems
- You may have to strengthen DP, equivalently the induction, i.e., you have may have to carry more information to find the Optimum.
- This means that sometimes we may have to use two dimensional or three dimensional induction

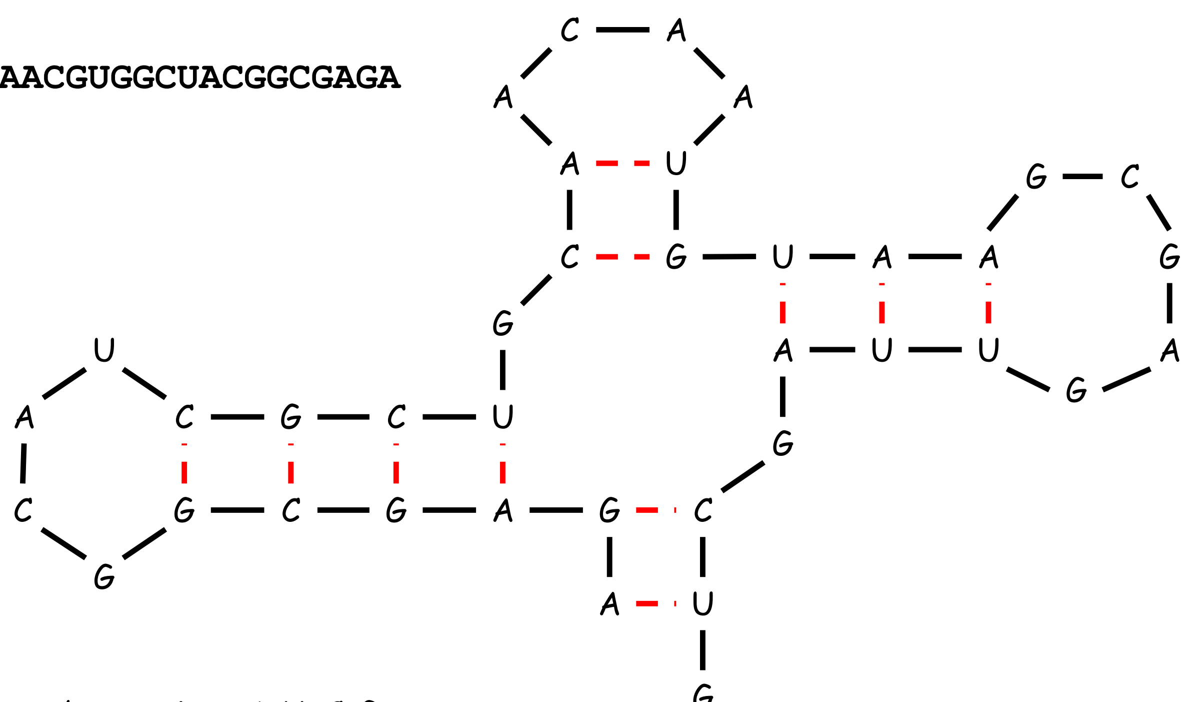
# RNA Secondary Structure

# RNA Secondary Structure

RNA: A 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 (Formal)

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 pair: 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 $i < j - 4$.

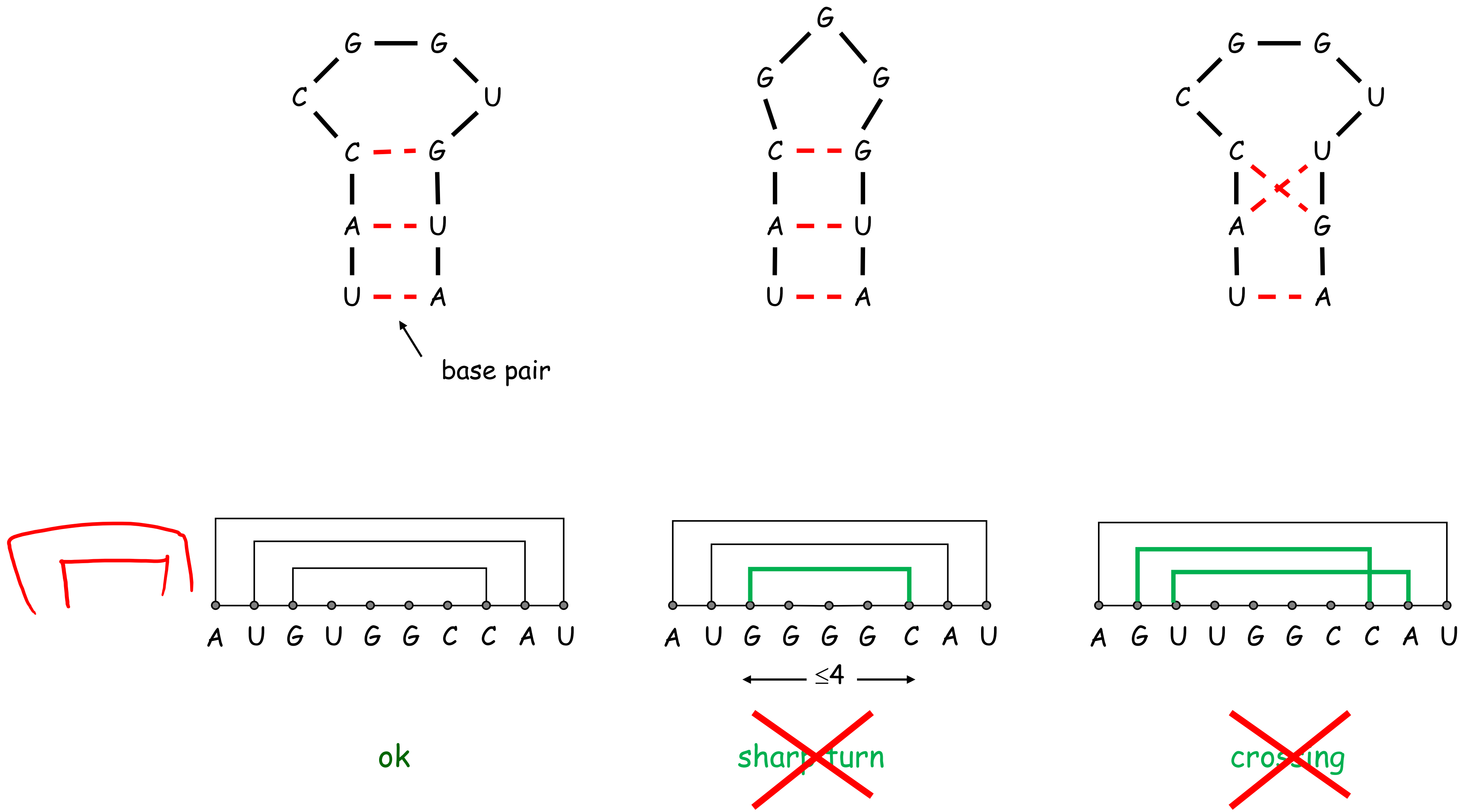
[Non-crossing.] If ($b_i$, $b_j$) and ($b_k$, $b_l$) are two pairs in S, then we cannot have $i < k < j < l$.

Free energy: Usual hypothesis is that an RNA molecule will maximize total free energy.

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.

# Secondary Structure (Examples)

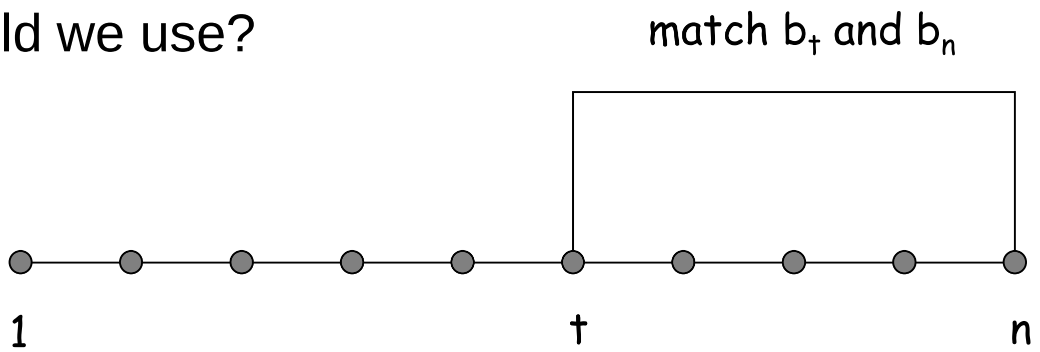


# DP: First Attempt

First attempt. Let $OPT(n)$ = maximum number of base pairs in a secondary structure of the substring $b_1b_2\ldots b_n$.

Suppose $b_n$ is matched with $b_t$ in $OPT(n)$.

What IH should we use?




Diffuclty: This naturally reduces to two subproblems

- Finding secondary structure in $b_1, \ldots, b_{t-1}$, i.e., OPT(t-1)
- Finding secondary structure in $b_{t+1}, \ldots, b_n$, ???

# DP: Second Attempt

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

The most important part of a correct DP; It fixes IH

Case 1: If $j - i \leq 4$.

- 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
- $OPT(i,j) = 1 + max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$