Weighted interval scheduling

Weighted Interval Scheduling (WIS) problem.

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

value (b + e + h) = \$15

value (d + h) = \$20

Recursive subproblems

consider the value of j_n

two cases:

- j_n *is part* of the optimal schedule O → consider the value of j_n
  - recurse on the last job compatible to j_n → $p(j_n)$
- j_n *is not part* of O → do not consider the value of j_n
  - recurse on job j_{n-1}

both are subproblems

We will explore these two options to find the full solution

The recursive step corresponds to solving a subproblem:

- a problem considering fewer jobs, either from $P(j_n)$ or j_{n-1}
- note that the subset of jobs is sequential — it contains *all* jobs before a certain index.

$j_1, j_2, \ldots, j_{n-1}$

or

$j_1, j_2, \ldots, P(j_n)$

WIS – notation for compatibility

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

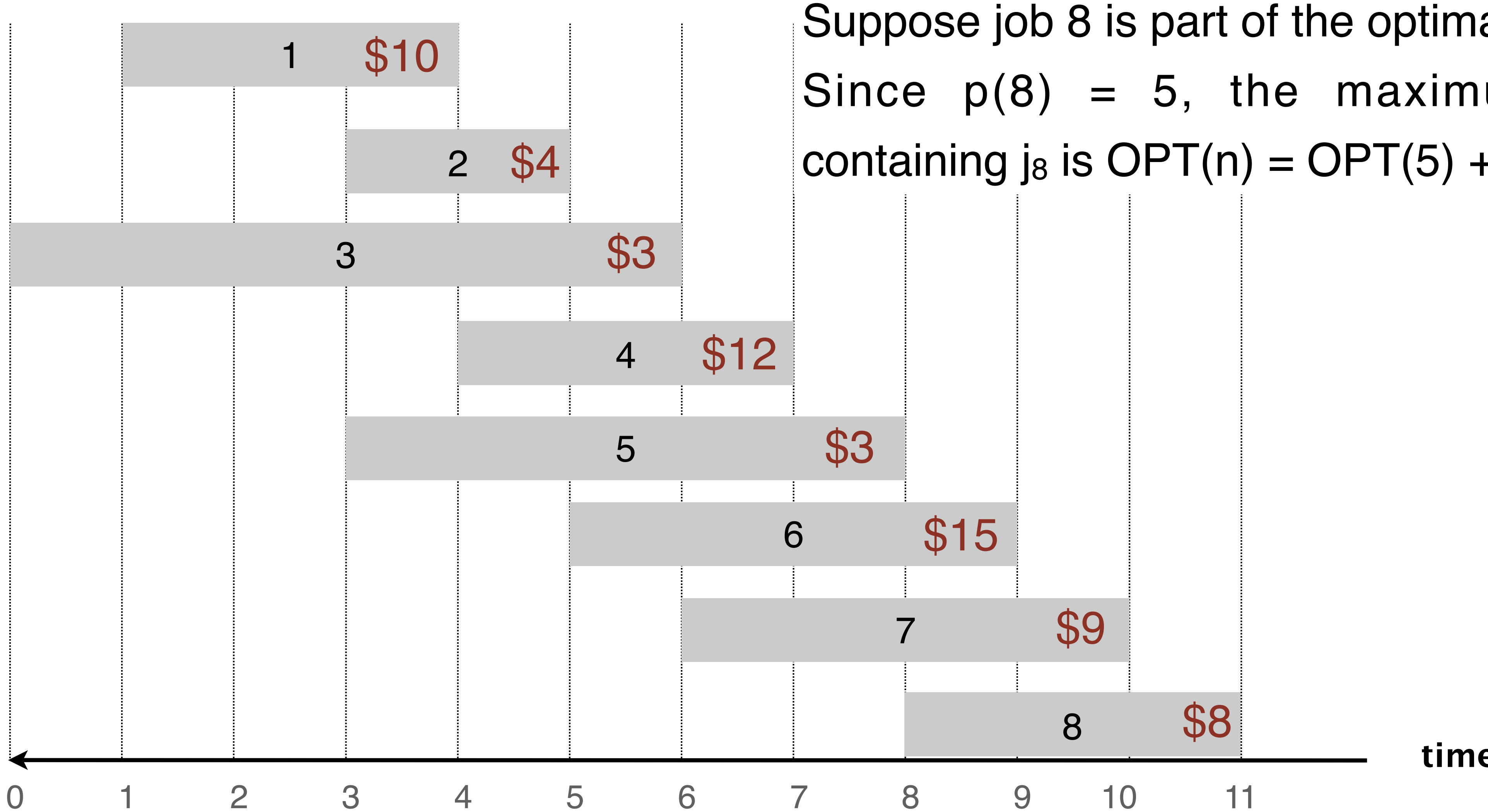
Def. $p(j)$ = largest index $i < j$ s.t. job i is compatible with j. (if none, then p(j) = 0)

Ex. $p(8) = 5, p(7) = 3, p(2) = 0$. it means tha j_2 is not compatible with anyone.

OPT(i) = maximum total value selection from jobs *1, 2, …, i*

Observation:

Suppose job 8 is part of the optimal solution. Since p(8) = 5, the maximum value containing j_8 is OPT(n) = OPT(5) + \$8



DP for WIS: recursive formula

Notation. $OPT(j)$ = opt solution, i.e. max total value selection from jobs $1, 2, ..., j$.

$OPT(n)$ = value of optimal solution to the original problem.

Case 1. $OPT(j)$ selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, ..., j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$. This is *OPT(p(j)).*

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

- Must include optimal solution to problem consisting of remaining jobs $1, 2, ..., j-1$. This is *OPT(j-1).*

Recursive formula : Choose the better from Case 1 and 2

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ \overbrace{v_j + OPT(p(j))}^{\text{collect job } j},\ \overbrace{OPT(j-1)}^{\text{do not collect job } j} \right\} & \text{otherwise} \end{cases}$$

WIS: exponential recursive algorithm

Algorithm 1: NaiveRecursiveWIS(n jobs: s_i, f_i, v_i)

---

sorted $\leftarrow$ sort jobs by increasing finish time $f_1 \prec \ldots \prec f_n$;
Compute $p(1), p(2), \ldots, p(n)$ `/* can be done in O(n) */`
return RecOpt(n)

---

memo = { }

if j in memo:
 return memo[j]

Algorithm 1: RecOpt(job index j)

---

if $j == 0$ **then**
  return 0
else
  $Opt(j) \leftarrow \max\{v_j + RecOpt(p(j)); RecOpt(j-1)\}$;
  memo[j] = Opt(j)
  return $Opt(j)$

Running time: $\Omega(2^{\frac{n}{2}})$

WIS – DP algorithm (recursive)

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

Memoization table M:

M[j] = OPT(j), array that contains the max value for jobs 0,1…,j

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

```
Sort jobs by finish time f_1 ≤ ... ≤ f_n;
Compute p(1), p(2), ... p(n);
M ← array(n+1)// Empty array of size n+1, indexed 0...n
M[0] ← 0// no jobs selected
return WISCompute(n);
```

Algorithm 2: WISCompute(j)

```
if M[j] is empty then
    M[j] ← max{v_j + WISCompute(p(j)) + WISCompute(j − 1)};
return M[j];
```

WIS – DP algorithm (bottom-up)

bottom-up algorithm to compute the optimal solution for WIS

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

Sort jobs by finish time $f_1 \leq \ldots \leq f_n$; → $\Theta(n \log n)$
Compute $p(1), p(2) \ldots p(n)$;
$M \leftarrow array(n+1)$`// empty array fo size n+1, indexed 0...n` } $\Theta(n)$
$\boxed{M[0] \leftarrow 0;}$
for $j = 1$ *to* n **do** — subproblems
  $M[j] \leftarrow \max\{\underbrace{v_j + M[p(j)]}; \underbrace{M[j-1]}\}$; } $\Theta(n)$
return $M[n]$; ← $\Theta(1)$

Two cases

Running time?

$\Theta(n \log n)$. The most expensive step is the pre-processing

WIS – DP algorithm – how to write a complete solution

$p(j) < j$

$j-1 < j$

1.: clearly define the subproblems, with proper indexing

OPT(j) = maximum value selection from job requests 1,..,j

Either OPT(p(j)) or OPT(j-1) for subproblems

2.: write the recursive formula:

p(j) = max {i: i <j and job i is compatible with j} = highest index of a job that doesn't overlap with j.

select j and recurse on compatible job

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

→ next subproblem, in case j is not selected

3.: bottom-up algorithm to compute the optimal solution

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

```
Sort jobs by finish time f_1 ≤ ... ≤ f_n;
Compute p(1), p(2) ... p(n);
M ← array(n+1)// empty array fo size n+1, indexed 0...n
M[0] ← 0;
for j = 1 to n do
    M[j] ← max{v_j + M[p(j)]; M[j − 1]};
return M[n];
```

4.: use backtracking to find set of jobs in optimal solution

WIS – DP algorithm (bottom-up/iterative)

Weighted Interval Scheduling: given n jobs, each with start time s_j, finish time f_j and value v_j find the compatible schedule with maximum total value.

OPT(j) = optimal solution for jobs (0),1,2,…,n

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

Memoization table M:

M[j] = OPT(j), array that contains the max value for jobs 0,1…,j

Instead of calling recursive functions, write your solution in the memoization table

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

Sort jobs by finish time $f_1 \leq \ldots \leq f_n$;
Compute $p(1), p(2) \ldots p(n)$;
$M \leftarrow array(n+1)$`// empty array fo size n+1, indexed 0...n`
$M[0] \leftarrow 0$;
for $j = 1$ *to* n **do**
  $M[j] \leftarrow \max\{v_j + M[p(j)]; M[j-1]\}$;
return $M[n]$;

WIS – DP algorithm (top-down/recursive)

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

Memoization table M:

M[j] = OPT(j), array that contains the max value for jobs 0,1…,j

Algorithm 1: WIS(n jobs: s_j, f_j, v_j)

Sort jobs by finish time $f_1 \leq \ldots \leq f_n$;
Compute $p(1), p(2), \ldots p(n)$;
$M \leftarrow array(n+1)$`// Empty array of size n+1, indexed 0...n`
$M[0] \leftarrow 0$`// no jobs selected`
return $WISCompute(n)$;

Algorithm 2: WISCompute(j)

if $M[j]$ *is empty* **then**
 $M[j] \leftarrow \max\{v_j + WISCompute(p(j)) + WISCompute(j-1)\}$;
return $M[j]$;

Go back to previous slide and notice the differences of each approach

Finding the set of optimal jobs – backtracking

A dynamic programming algorithm computes the optimal value.

How to find the solution itself?

We can reconstruct it from the table.

- backtrack based on the memoization table without explicitly storing values (by checking which case was chosen)

Use the cases from your original algorithm to retrieve the decisions you made in the past.

Algorithm 1: FindSolution(j)

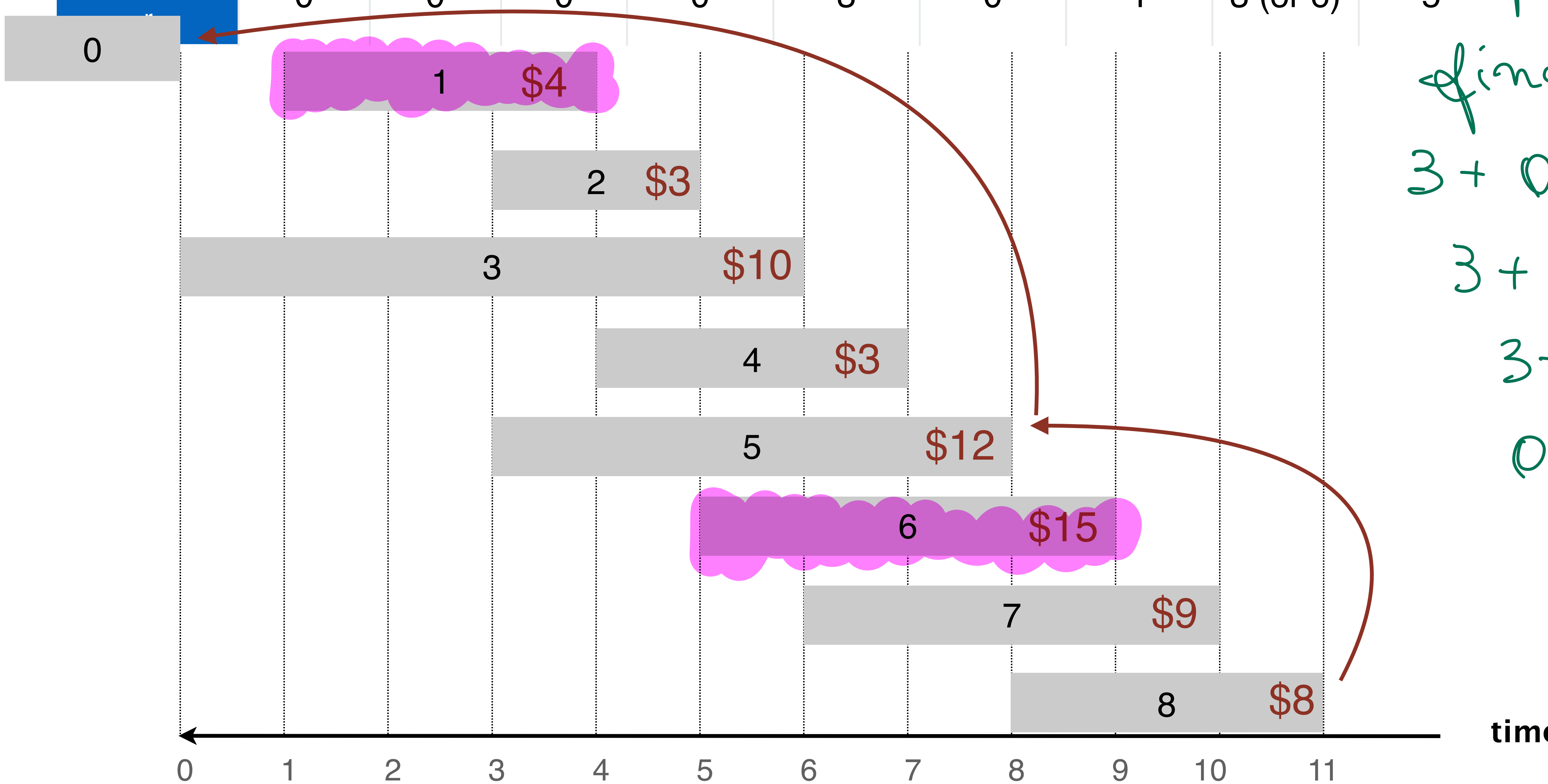
```
if j == 0 then
    return ∅;
else if v_j + M[p(j)] > M[j − 1] then
    return {j} ∪ FindSolution(p(j));
else
    return FindSolution(j − 1);
```

Finding the set of optimal jobs – backtracking

find(6)
15 + opt(j(6))
15 + opt(2)
15 + 4 = 19

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

j	0	1	2	3	4	5	6	7	8
OPT(j)	$0	$4	$~~3~~4	$10	$10	$12	$19	$19	$20
predecessor	0	0	0	0	3	0	1	3 (or 6)	5



find(j(6))
find(2)
3 + opt(j(2))
3 + opt(0)
3 + 0 = 3
opt(1)
4

Minimum Number of Operations - TopHat

Problem: Given an integer *n*, find the minimum number of operations to get from 0 to n, if you are only allowed to perform two specific operations: (1.) *add 1* (2.) *multiply by 2*.

example compute 12:

0 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12 //12 operations

(((0+1) x 2) x 2) + 1 +1 +1 +1 = 12 // 7 operations

(((0 +1) +1 + 1) x 2) x 2 = 12 // 5 operations

Question: Suppose OPT(j) is the minimum number of operations required to make the number j. What is the recursive formula for computing OPT(j)? we may assume the base case OPT(0) = 0.

A. $OPT(j) = \begin{cases} OPT(j-1) \text{ if } j \text{ is odd} \\ OPT(j/2) \text{ if } j \text{ is even} \end{cases}$

C. $OPT(j) = \begin{cases} OPT(j-1) + 1 \text{ if } j \text{ is odd} \\ 1 + \min\{OPT(j/2); OPT(j-1)\} \text{ if } j \text{ is even} \end{cases}$

B. $OPT(j) = \begin{cases} OPT(j-1) + 1 \text{ if } j \text{ is odd} \\ OPT(j/2) + 1 \text{ if } j \text{ is even} \end{cases}$

D. $OPT(j) = \begin{cases} OPT(j-1) \text{ if } j \text{ is odd} \\ \min\{OPT(j/2); OPT(j-1)\} \text{ if } j \text{ is even} \end{cases}$

Minimum Number of Operations

Problem: Given an integer *n*, find the minimum number of operations to get from 0 to n, if you are only allowed to perform two specific operations: (1.) *add 1* (2.) *multiply by 2*.

Have fun implementing the top-down (recursive) approach!

Algorithm 1: MinOperations(n)

```
M ← length-(n+1) array;
M[0] = 0;
for j = 1 to n do
    if j is odd then
        M[j] = M[j − 1] + 1;
    else if Mj − 1] + 1 < M[j/2] + 1 then
        M[j] = M[j − 1] + 1;
    else
        M[j] = M[j/2] + 1;
return M
```

Finding the sequence of operations– backtracking

A dynamic programming algorithm computes the optimal value.

How to find the solution itself?

We can reconstruct it from the table.

- backtrack based on the memoization table without explicitly storing values (by checking which case was chosen)

Algorithm 1: Findsolution(j, M)

if $j == 0$ **then**
    return $\emptyset$
if j *is odd* **then**
    return FindSolution($j - 1$,M).append('+1')
else if $M[j-1] + 1 < M[j/2] + 1$ **then**
    return FindSolution($j - 1$,M).append('+1')
else
    return FindSolution($j/2$,M).append('x2')

Remember the decisions you made in the past

Subset sum – dynamic programming

Subset Sum problem: given a set of n positive integer weights $w_1, w_2, \ldots, w_n$ and a weight limit W. Find the subset of weights S with maximum total weight that doesn't exceed W. That is, find

In this problem we have two input: set of weights + weight limit.
Every choice you make changes the remainder weight limit.

$$S: \max_{S \subseteq \{w_1 \ldots w_n\}} \sum_{w_i \in S} w_i \le W$$

DP: OPT(j) = the max weight solution among weights $w_1, w_2, \ldots, w_j$

OPT(j) = max{ ? }

Notice that you need more information to solve subproblems.

Complications:

- there are no compatibility issues as with overlapping jobs (good)
- once a weight is chosen the available weight limit is decreased. Can we express this with just a single variable in OPT?

Subset sum – dynamic programming – 2-dimensional DP

Subset Sum problem: given a set of n positive integer weights $w_1, w_2, \ldots, w_n$ and a weight limit W. Find the subset of weights S with maximum total weight that doesn't exceed W. That is, find

$$S: \max_{S \subseteq \{w_1 \ldots w_n\}} \sum_{w_i \in S} w_i \leq W$$

OPT(j, w) = the max weight solution among weights $w_1, w_2, \ldots, w_j$ with available weight limit w.

consequence: less weight left to solve subproblems

$$OPT(j,w) = \begin{cases} 0 & \text{if } j = 0 \text{ or } w = 0 \\ OPT(j-1,w) & \text{if } w_j > w \\ \max\{w_j + OPT(j-1, w - w_j); OPT(j-1,w)\} & \text{otherwise} \end{cases}$$

revenue for choosing item j

more space left for subproblems but no revenue.

Subset sum – 2D memoization table – TopHat

$$OPT(j,w) = \begin{cases} 0 & \text{if } j = 0 \quad \text{or} \quad w = 0 \\ OPT(j-1,w) & \text{if } w_j > w \\ \max\{w_j + OPT(j-1, w-w_j); OPT(j-1,w)\} & \text{otherwise} \end{cases}$$

Input: $w_1, w_2, \ldots, w_n$ and W (assume weights are ints)

Output: OPT(n,W)

Implementation:

Question: What is the size of the memoization table and what is the running time of the resulting DP algorithm?

A. n^2 & $O(n^3)$

B. W^2 & $O(W^2)$

C. nW & $O(n^2W)$

D. (n+1)(W+1) & O(nW)

Keep some space for base case!

Subset sum – 2D DP

$$OPT(j,w) = \begin{cases} 0 & \text{if } j = 0 \text{ or } w = 0 \\ OPT(j-1,w) & \text{if } w_j > w \\ \max\{w_j + OPT(j-1, w-w_j); OPT(j-1,w)\} & \text{otherwise} \end{cases}$$

Input: $w_1, w_2, \ldots, w_n$ and W (assume weights are ints)

Output: OPT(n,W)

Algorithm 1: SubsetSum($w_1, w_2, \ldots, w_n, W$)

```
M ← (n+1) × (W+1) table/* 2D array/ matrix */
/* set border cases */
M[0][*] = 0/* set row 0 to zeros */
M[*][0] = 0/* set column 0 to zeros */
for j = 1...n do
    for w = 1...W do
        /* apply recursive formula */
        M[j][w] = max{w_j + M[j-1][w-w_j]; M[j-1][w]};
return M[n][W]
```

$\Theta(nW)$

Subset sum – 2D DP – backtracking the solution

$$OPT(j,w) = \begin{cases} 0 & \text{if } j = 0 \\ OPT(j-1,w) & \text{if } w_j > w \\ \max\{w_j + OPT(j-1, w-w_j); OPT(j-1,w)\} & \text{otherwise} \end{cases}$$

Input: filled memoization table M

Output: set of weights in the optimal solution S

runtime?

Algorithm 1: SubsetSumSolution(M,$w = [w_1, \dots w_n]$,W)

```
S ← [ ] /* set of opt weights */
i ← n, j ← W;
while i > 0 AND j > 0 do
    if M[i][j] > M[i − 1][j] then
        /* the case where w_i is chosen */
        S.append(w[i]);
        i ← i − 1;
        j ← j − w_i;
    else
        /* w_i is not chosen */
        i ← i − 1;
return S
```

To keep or not to keep?
This is the question!

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max-profit on items $1, \ldots, i$ with weight limit w.

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i - 1 \}$ using weight limit w.

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit = $w - w_i$.
- $OPT(i, w\text{-}w_i)$ selects best of $\{ 1, 2, \ldots, i - 1 \}$ using this new weight limit.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ \underbrace{OPT(i-1, w)}_{\text{Don't put item in your bag}},\ \underbrace{v_i + OPT(i-1, w-w_i)}_{\text{Select the item and put it in your bag}} \} & \text{otherwise} \end{cases}$$

Knapsack problem example

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

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

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	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, 2, 3, 4, 5 }	0											

OPT(i, w) = max-profit subset of items 1, …, i with weight limit w.

Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\ W)$ time and $\Theta(n\ W)$ space.

weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\ W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$. ▪

Knapsack problem is $\boxed{\text{NP-complete}}$

We are going to see it again at the end of the semester.

Knapsack is in fact not polynomial in the *input size!*

Input:

2n integers: v_i and w_i

one additional integer W

How many bits to describe the input?

- W requires *log W* bits, w_i requires *O(log W)* bits
- overall *O(n log W)*

The algorithm would be polynomial in the input size, if the running time was a polynomial of *n* and *log W*

But the running time is $O(nW) = O(n\,2^{\log W})$

- Decision version of knapsack problem is **NP**-complete. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

DP algorithm – full solution

Here is how you would properly write out the solution to a DP problem:

Pay attention to the input values

1. precisely define the subproblem with proper indexing
   - OPT(i) = ……. or OPT(i,j) = …. is also possible! (or even more variables)
2. give the recursive formula to compute OPT( ) and argue about its correctness
   - make sure to define everything that needs to be, e.g. p(j) = …
   - don't forget about border cases (sometimes you may want to add a dummy index, e.g. j=0, OPT(0) = 0)

Recursive is more desirable.

3. write the DP algorithm.
   - bottom-up and recursive are equally good. The asymptotic running time is the same.
   - be clear about what values your memoization table holds,
     - e.g. M[i,j] = OPT(i,j), size of M is n x W
     - don't forget initialization steps for border cases
4. write an algorithm that prints the elements (e.g. jobs) in the optimal solution
   - sometimes called "back-tracking" the solution

Bounded Knapsack problem

As input we are given the weights w_i and values v_i of each of n items, further we are given a maximum capacity of W. Suppose there are *two* identical copies of each item available. Select a maximum value subset of the items within the capacity limit W, such that we can take at most *two* of each item.

Same input as before = OPT(i, w)

3 decisions: $\begin{cases} 0 \text{ items} \\ 1 \text{ item} \\ 2 \text{ items} \end{cases}$

Bounded Knapsack problem - backtracking

Backtracking the maximum choice over multiple items is tedious.

Instead: keep track of our decisions on the fly:

C = length (n+1)x(W+1) array

C[i][w] = how many copies of i we select for OPT(i,w)

Algorithm 1: BoundedKnapsack($i = 1 \dots n : (w_i, v_i), W$)

```
/* (w_i, v_i) weight and value of item i, W capacity */
M ← (n+1) × (W+1) array /* DP table */
C ← (n+1) × (W+1) array /* number of copies */
M[0][*] ← 0 and M[*][0] ← 0;
for i = 1 to n do
    for w = 1 to W do
        c_0 ← M[i−1][w] /* 0 of item i */
        c_1 = v_i + M[i−1][w − w_i] if w_i < w else c_1 ← −1 /* 1 of i */
        c_2 = 2v_i + M[i−1][w − 2w_i] if 2w_i < w else c_2 ← −1 /* 2 of i */
        M[i][w] ← max{c_0, c_1, c_2};
        C[i][w] ← argmax{c_0, c_1, c_2} /* index of max case */
return M, C
```

Bounded Knapsack problem - backtracking

Backtracking the maximum choice over multiple items is tedious.

Instead: keep track of our decisions on the fly:

C = length (n+1)x(W+1) array

C[i][w] = how many copies of i we select for OPT(i,w)

Algorithm 1: BKBacktrack(C, W)

$sol \leftarrow$ empty list;
$i \leftarrow n$ and $w \leftarrow W$;
while $i > 0$ *and* $w > 0$ **do**
  $sol.add(C[i][w] \times$ item $i)$`/* add0, 1 or 2 of item i */`
  $i \leftarrow i - 1$;
  $w \leftarrow w - C[i][w] \cdot w_i$;
return sol

Bounded Knapsack problem - TopHat

As input we are given the weights w_i and values v_i of each of n items, further we are given a maximum capacity of W. Suppose there are ~~*two*~~ *m* identical copies of each item available. Select a maximum value subset of the items within the capacity limit W, such that we can take at most ~~*two*~~ *m* of each item.

OPT(i,w) = maximum value within capacity w if we can consider items 1,…,i

What is the recursive formula for OPT(i,w) - excluding boundary cases?

A. $OPT(i,w) = \max_{j=0\ldots m} \{j \cdot v_i + OPT(i-1, w - j \cdot w_i)\}$ Considers all m+1 cases

B. $OPT(i,w) = \max\{OPT(i-1,w); m \cdot v_i + OPT(i-1, w - m \cdot w_i)\}$

takes m copies or nothing

C. $OPT(i,w) = \max_{j=0\ldots m} \{j \cdot v_i + OPT(i-1, \underbrace{w - w_i})\}$

does not account for the weight of all selected items

D. $OPT(i,w) = \max_{j=0\ldots m} \{j \cdot v_i + OPT(i-j, w - w_i)\}$

Coin change problem

In a far away country there are four different valued coins, the dream dollar amounts are \$1, \$4, \$7, \$13. In this country people always try to pay with the fewest number of coins possible. Design a DP algorithm to pay \$n with the fewest number of coins.

Similar to the problem presented in the previous slide.

bonus: the greedy algorithm - pay with the largest denomination while possible - doesn't work. However, you'll probably need to implement it to find the smallest \$n counter example

String matching example – plagiarism

Plagiarism is often not a verbatim copy.

source: https://www.turnitin.com/static/plagiarism-spectrum/ "Find-Replace" method

SOURCE TEXT

A Natural Setting: A History of Exploration and Settlement in Yosemite Valley

Since its first discovery by non-indigenous people in the mid-nineteenth century, Yosemite Valley has held a special, even religious, hold on the American conscience because its beauty makes it an incomparable valley and one of the grandest of all special temples of Nature. While Yosemite holds a special grip on the western mind, perceptions about the Valley have evolved over time due to changing politics, migration patterns and environmental concerns as man has become more attuned to his relationship and impact on nature.

STUDENT WORK

A Beautiful Setting in Yosemite

Since its first discovery by non-native people in the mid-19th century Yosemite Valley has held a special, even sacred, hold on the American psyche because its beauty makes it an incomparable valley and one of the grandest of all special temples of Nature. While Yosemite holds a special grip on the western mindset, perceptions about the Valley have evolved over time due to changing political movements, migration patterns and environmental issues as man has become more attuned to their relationship and impact on nature.

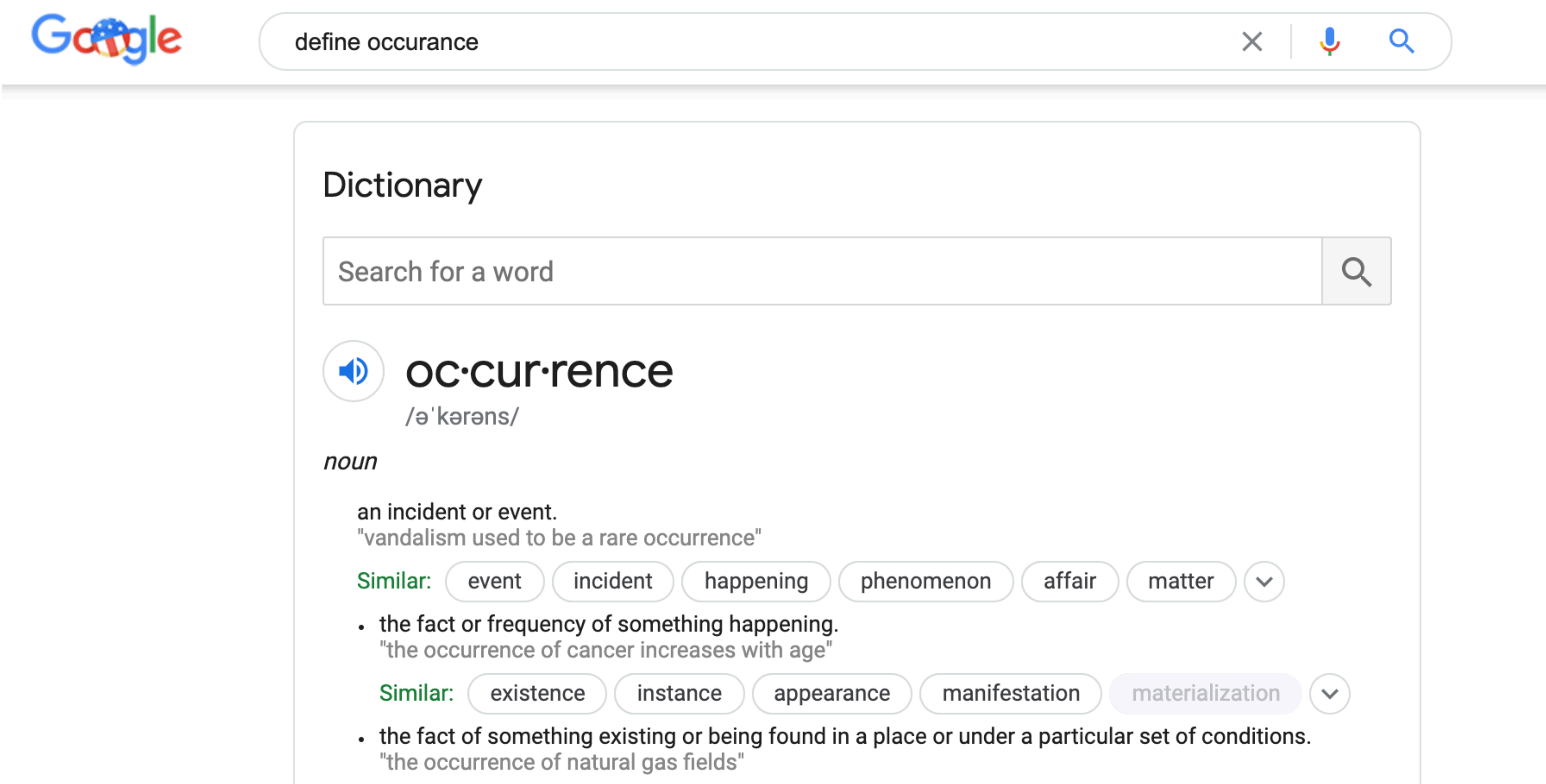
Some words, e.g. "the", "and", or given the topic of this text "Yosemite", naturally appear in both and are not plagiarized. We want to assign some kind of *similarity* score between the two texts.

String similarity

Type “Define ocurrance” in Google.

Google will correct your spelling. How does it know which word you wanted to find?

(source: google.com search result)



String similarity

Q. How similar are two strings? ocurrance and occurrence.

Which alignment is best depends on relative cost of gap and mismatch penalties

- The cost α of a mismatch and δ of a gap is part of the input

different characters = mismatch

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

gap

6 mismatches, 1 gap

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	–	u	r	r	–	a	n	c	e
o	c	c	u	r	r	e	–	n	c	e

0 mismatches, 3 gaps

$$\text{cost} = \alpha \cdot \text{ mismatch} + \delta \cdot \text{ gap}$$

Problem: given α and δ, what is the best alignment of the two strings.

String similarity – Levenshtein Distance

Edit distance:

- first introduced by Levenshtein (1966)
- number of single character edits (insertion, deletion or substitution) required to change one string into another.
  - sometimes called Levenshtein distance

Longest Common Subsequence: special case, only allows insertions and deletions, not substitutions

String similarity – Longest Common Subsequence (Substring)

Longest Common Subsequence: given two sequences $x = [x_1, x_2, \ldots, x_n]$ and $y = [y_1, y_2, \ldots, y_m]$ find a longest (not consecutive) subsequence common to them both.

- special case of the similarity problem with mismatch penalty = infinite, gap penalty = 1

application: Genome similarity

- used in computational biology
- algorithm is named after Needleman and Wunsch (1970s)

```
   cgtacgtacgtacgtacgtacgtatcgtacgt
   acgtacgtacgtacgtacgtacgtacgtacgt

  cgtacgtacgtacgtacgtacgta t cgtacgt
a cgtacgtacgtacgtacgtacgta   cgtacgt
```

Sequence alignment

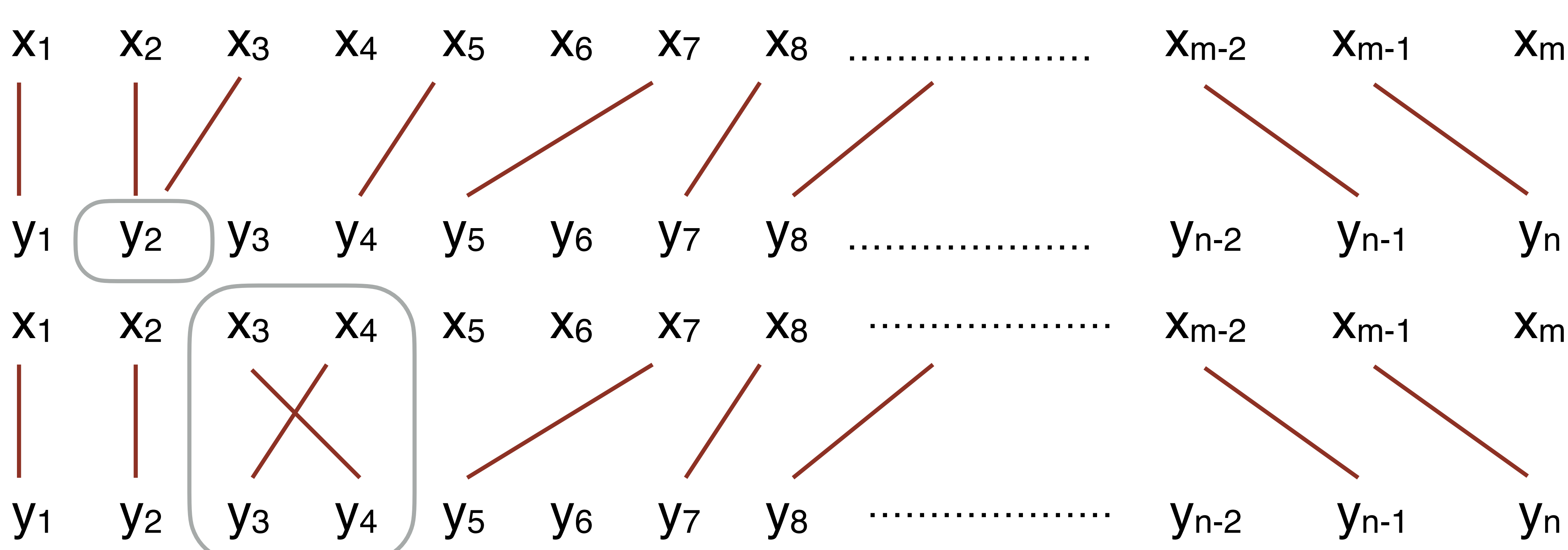
Problem. Given two strings $X = [x_1\ x_2 \ldots\ x_m]$ and $Y = [y_1\ y_2 \ldots y_n]$ and costs α, δ find the *minimum-cost alignment* Align(X,Y).

X and Y can have different lenght

Alignment. Given two strings X and Y, their alignment Align(X,Y) is a set of ordered pairs (a matching) (x_i, y_j), such that

- each character is matched at *most once*
- there are no two pairs (x_i, y_k) and (x_j, y_l), such that x_i comes before x_j but y_k after y_l or vice verse, i.e. there are *no crossing pairs*

invalid alignment:



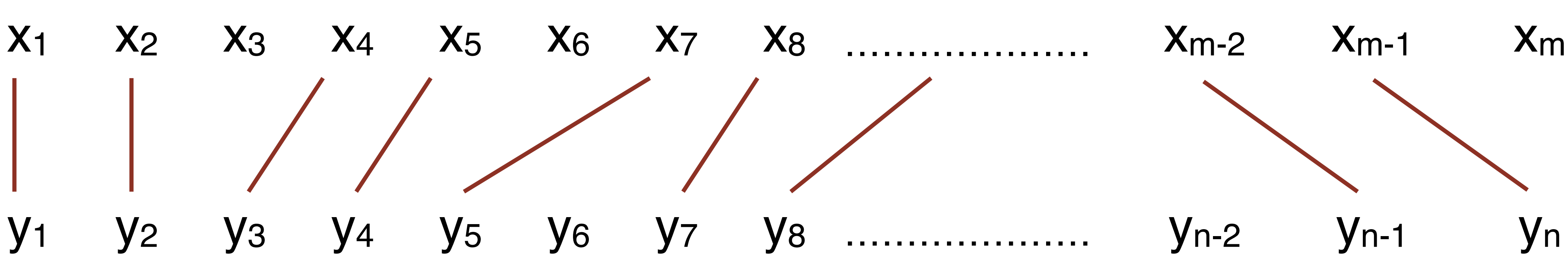
Sequence alignment

Problem. Given two strings $X = [x_1\ x_2 \dots\ x_m]$ and $Y = [y_1\ y_2 \dots y_n]$ and costs α, δ find the *minimum-cost alignment* Align(X,Y).

Alignment. Given two strings X and Y, their alignment Align(X,Y) is a set of ordered pairs (a matching) (x_i, y_j), such that

- each character is matched at *most once*
- there are no two pairs (x_i, y_k) and (x_j, y_l), such that x_i comes before x_j but y_k after y_l or vice verse, i.e. there are *no crossing pairs*

valid alignment:



Sequence alignment

Problem. Given two strings $X = [x_1\ x_2 \ldots\ x_m]$ and $Y = [y_1\ y_2 \ldots y_n]$ and costs α, δ find the *minimum-cost alignment* Align(X,Y).

Alignment. Given two strings X and Y, their alignment Align(X,Y) is a set of ordered pairs (a matching) (x_i, y_j), such that

- each character is matched at *most once*
- there are no two pairs (x_i, y_k) and (x_j, y_l), such that x_i comes before x_j but y_k after y_l or vice verse, i.e. there are *no crossing pairs*

Cost of an alignment:

α, δ are given as input.

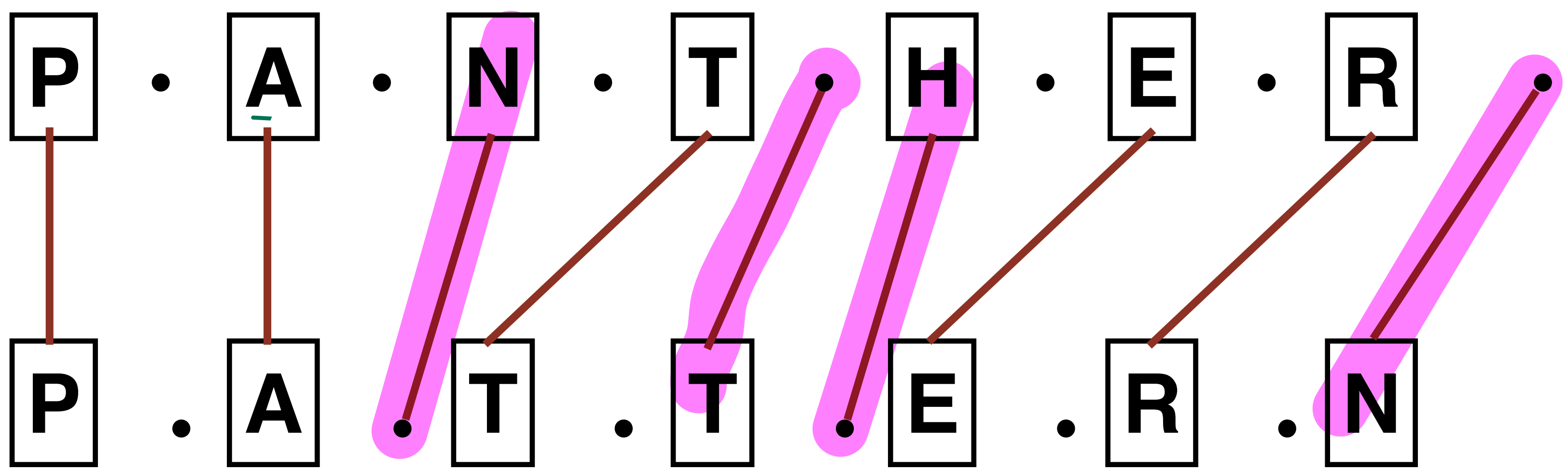
$$\text{cost}(\text{Align}(X,Y)) = \alpha \cdot \#(\text{mismatch}) + \delta \cdot \#(\text{gap})$$

# of unmatched characters in X +
# of unmatched chars in Y

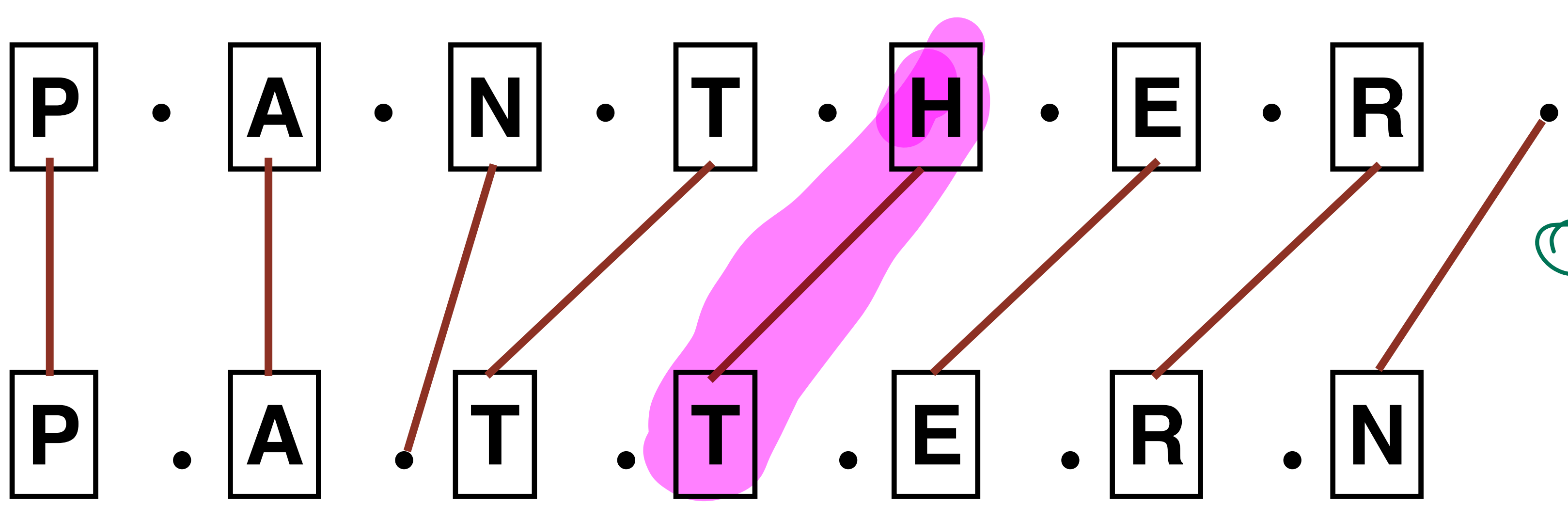
Question: cost of alignment?

mismatch: $\alpha = 3$

gap: $\delta = 2$



cost = $4 \cdot 2 + 0 \cdot 3 = 8$



cost = $2\ 2 + 1 \cdot 3 = 7$

Brute-force approach — TopHat

Algorithm: Try *all* possible valid alignments and return the min-cost

valid alignment:

- each character xi is matched to at most one character yj
- there are no crossing pairs

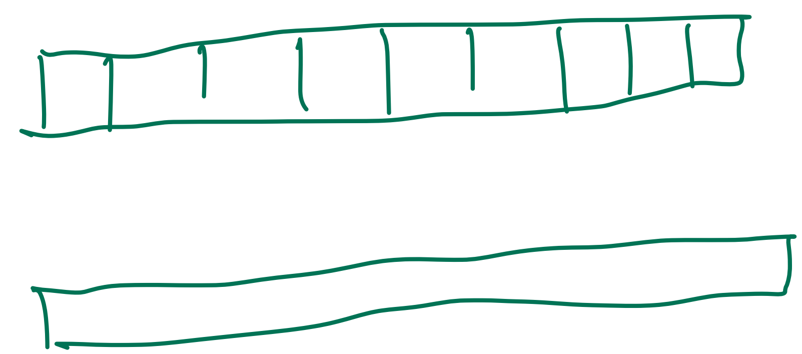
Question. Guess how many valid alignments there are if X is a string of m, and Y a string of n characters. (we may assume $m \leq n$)

A. $\sum_{k=0}^{m} \binom{m}{k}\binom{n}{k} = \binom{m+n}{m}$

B. $m!n! = O(m^m n^n)$

C. $m \cdot n$

D. m^n



$m = 2^m$

$n = 2^n$

$m \cdot m \cdot m$

Brute-force approach — number of valid alignments

Some facts:

- each character x_i in X is aligned to either a character y_j or a gap.
- if k characters in X are matched to characters in Y, then the number of matched characters in Y is also k.

compute number of valid assignments:

count how many ways there are to pick k among X and k among Y. Those are the characters matched to each other.

- note that the order in which these k are matched is fixed, and hence unambiguous

$$\sum_{k=0}^{m} \binom{m}{k} \binom{n}{k} = \binom{m+n}{m}$$

if m < n we get:

$$\geq \binom{2m}{m} = \Theta\left(\frac{4^m}{\sqrt{m}}\right)$$

choose the k characters in X that are assigned to characters in Y(as opposed to gaps)

choose the k characters in Y that are assigned to characters in X

tricks with binomial coefficients

Good to know:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

$$2^n = (1+1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} 1^k = \sum_{k=0}^{n} \binom{n}{k}$$

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\sum_{k=0}^{n} \binom{n}{k} \binom{m}{m-k} = \binom{n+m}{m}$$

Computation on previous slide:

$$\sum_{k=0}^{n} \binom{n}{k} \binom{m}{k} = \sum_{k=0}^{n} \binom{n}{k} \binom{m}{m-k} = \binom{n+m}{m} = \binom{n+m}{n}$$