7 Dynamic Programming

Motivation So far, we have seen that we can tackle some problems using

- greedy algorithms
- divide and conquer algorithms

in order to devise algorithms that work more efficiently than a simple brute-force approach which explicitly considers all possible cases separately.

However, for some problems, it is impossible to find a greedy algorithm that can be shown to derive the desired solution. Likewise, it is not always possible to find a useful divide and conquer strategy.

Key Idea The key idea of dynamic programming is to

1. divide an original problem into smaller sub-problems and to
2. construct the overall best solution by considering successively larger subproblems, thereby
3. efficiently discarding sub-solutions at the earliest possible opportunity, i.e. as soon as we can conclude that they are not relevant for deriving the globally best solution
4. In order to be able to apply ideas 1 to 3, the desired overall solution must be "best" w.r.t. some scoring function which has to have the property that the overall score for a complete solution can be written as a sum (or product) of scores for the corresponding partial solutions.

7.1 Example 1: Weighted Interval Scheduling

Reminder We saw in section 3 that we can use a greedy algorithm in order to find a set of non-overlapping intervals that is as large as possible, i.e. where the weight in each interval is set to 1 (see section 3.1).

Task Now Given a set S of n weighted intervals

$$\begin{aligned} S &= \{(s(1), f(1), v(1)), \ldots (s(n), f(n), v(n))\} \\ &= \{(s(1), f(1), v(1)) | i \in \{1, \ldots, n\}, s(i), f(i), v(i) \in \mathbb{R}+, s(i) < f(i)\} \end{aligned}$$

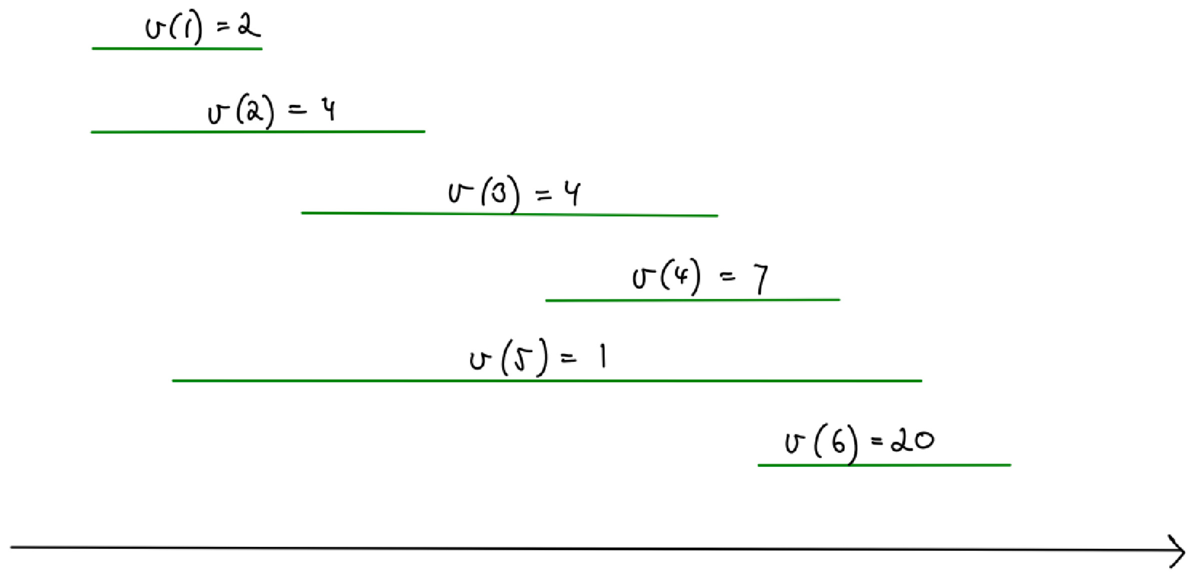
where $v(i)$ denotes the weight, $s(i)$ denotes the start point and $f(i)$ denotes the end point of the interval i,

Find an optimal subset T of S, i.e. $T \subseteq S$, such that

- T contains only non-overlapping intervals

- there is no other subset T' of S, $T' \neq T$, whose intervals are non-overlapping and where
$$\sum_{i in T'} v(i) > \sum_{i in T} v(i)$$
i,e. whose sum of interval weights is larger than the sum of interval weights in T.

Visually



Note: The weight $v(i)$ of interval i has nothing to do with its length, i.e. with $f(i) - s(i)$.

Assume In the following, assume WLOG that the intervals are ordered by their end points.

Define
- $S_j := \{(s(i), f(i), v(i)) | i \leq j\}, j \in \{1, \ldots, n\}$ the subset of intervals from S whose end points are $f(i) \leq f(j)$. $S_j \subseteq S$.
- $w(j) :=$ the sum of the weights for the best interval scheduling for set S_j, $w(0) = 0$
- $c(j) :=$ largest value of i, $i < j$, such that intervals i and j do not overlap; $c(j) = 0$ if there exists no such value of i
- $\mathbb{N}_i := \{1, 2, \ldots, i\}$

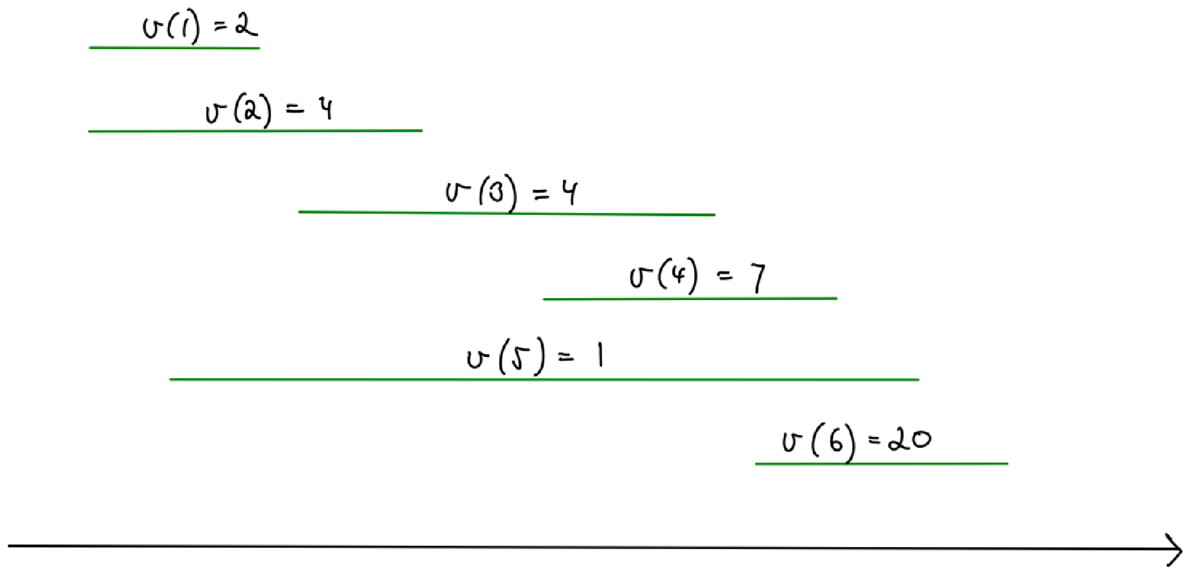
Group Work What is $c(4)$ in the previous example?

Answer 2, because intervals 2 and 4 are non-overlapping, but 3 and 4 are.

Algorithm WeightedIntervalScheduling(S)

```
• set $w(0) = 0$
• for $(i = 1$ to $n)\{$
    – $w(i) = \max\{w(i-1), v(i) + w(c(i))\}$
    }
• $T = \varnothing$
• $best\_i = n$
• while ($best\_i > 0$) {
    – if ($w(best\_i) > w(best\_i - 1)$){
        ∗ add $best\_i$ to set $T$
        ∗ $best\_i = c(best\_i)$
      }
    – else {
        ∗ $best\_i = best_i - 1$
      }
  }
• return set $T$
```

Example Execute algorithm on the following set of intervals (group work)



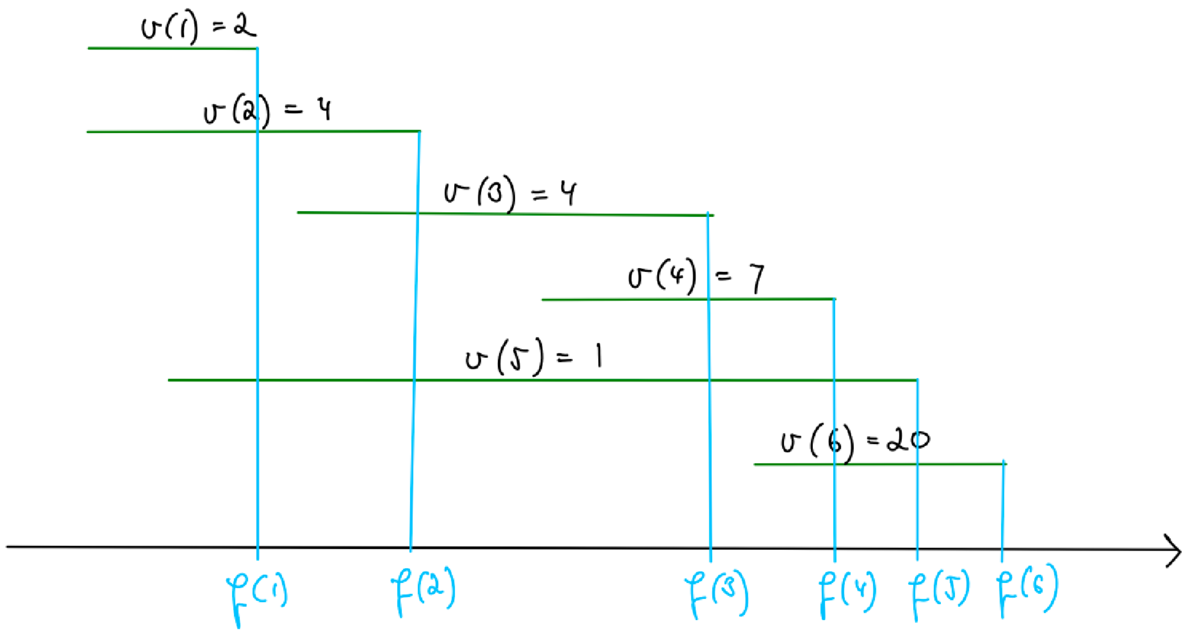
Answer:

$$\begin{aligned}
w(0) &= 0 \\
w(1) &= \max\{w(0), v(1) + w(0)\} = v(1) = 2 \\
w(2) &= \max\{w(1), v(2) + w(c(2))\} \\
&= \max\{2, 4 + 0\} \\
&= 4 \\
w(3) &= \max\{w(2), v(3) + w(c(3))\} \\
&= \max\{4, 4 + 2\} \\
&= 6 \\
w(4) &= \max\{w(3), v(4) + w(c(4))\} \\
&= \max\{6, 7 + 4\} \\
&= 11 \\
w(5) &= \max\{w(4), v(5) + w(c(5))\} \\
&= \max\{11, 1 + 0\} \\
&= 11 \\
w(6) &= \max\{w(5), v(6) + w(c(6))\} \\
&= \max\{11, 20 + 6\} \\
&= 26
\end{aligned}$$

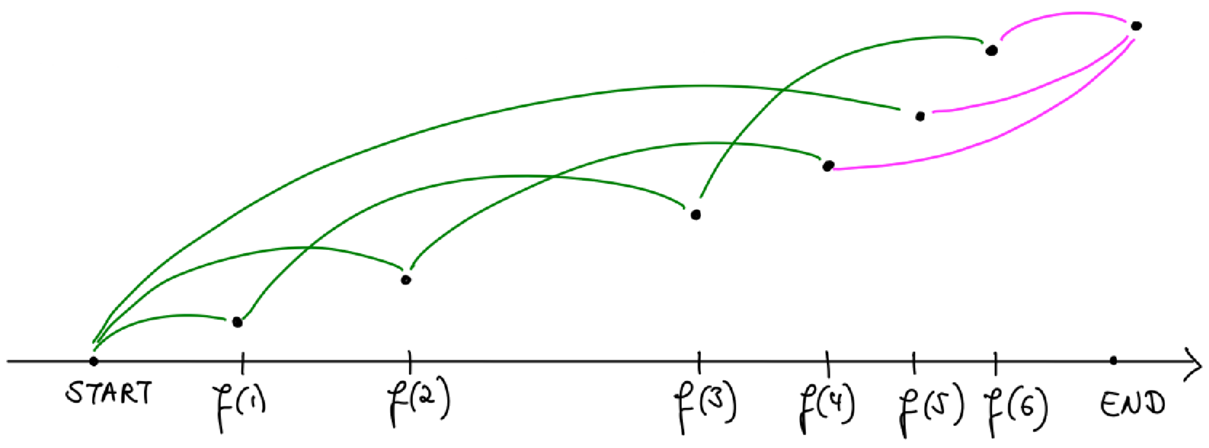
Answer (cont'd) : (traceback part of the algorithm)

- $T = \varnothing$
- $best_i = 6$
- $w(6) > w(5) \Rightarrow T = \{6\}, best_i = c(6) = 3$
- $w(3) > w(2) \Rightarrow T = \{3, 6\}, best_i = c(3) = 1$
- $w(1) > w(0) \Rightarrow T = \{1, 3, 6\}, best_i = c(1) = 0 \Rightarrow T = \{1, 3, 6\}$

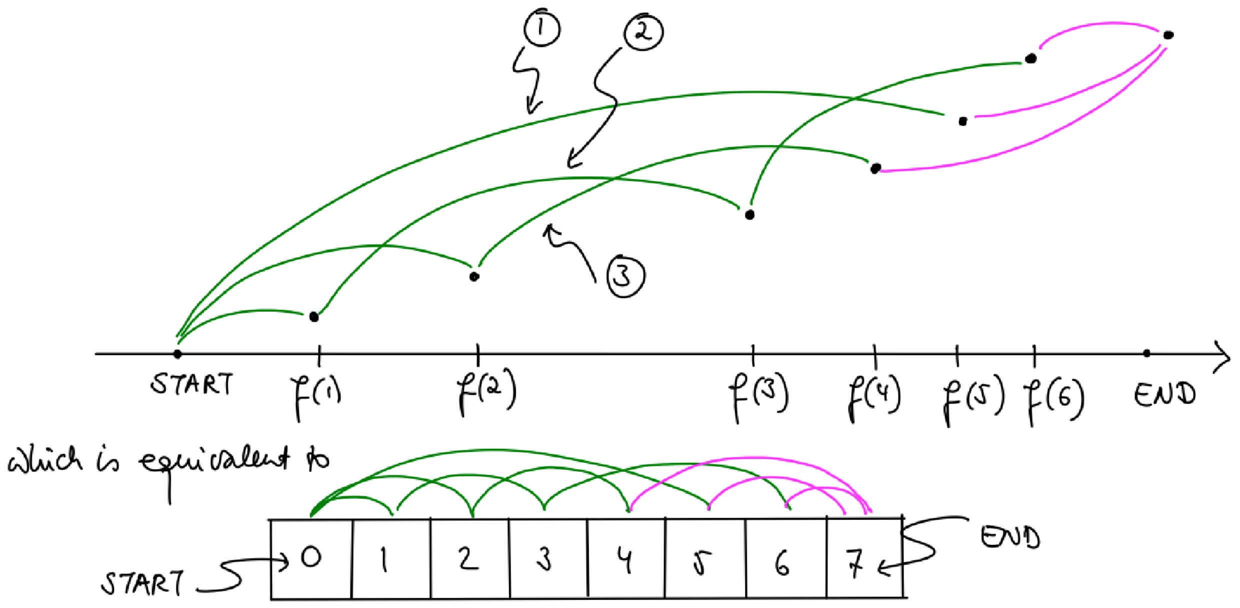
Question How can we best visualize how the algorithm operates?



1. order intervals by end-points $f(i)$
2. draw a box at $f(i)$ for every interval and at 0 (START)
3. draw a *line* connecting interval i to $c(i)$



4. add an END point and connect all intervals $i \in \{1, \ldots, n\}$ to that point if they do not have an interval $j > i, j \in \{1, 2, \ldots, n\}$ that is compatible with it.



5. finding the best solution $w(i)$ then corresponds (visually) to identifying the best path in the agove figure connecting the START and END points where "best" means having the highest sum of interval weights.

Definition weight of a path := sum of weights of intervals i on that path. In this example, we have three possible paths:

$$\begin{aligned} T &= \{5\} && \text{weight} = v(5) = 1 \\ T &= \{1,3,6\} && \text{weight} = v(1) + v(3) + v(6) = 26 \\ T &= \{2,4\} && \text{weight} = 11 \end{aligned}$$

[Note: Different paths need not be disjoint as we will see in other examples.]

Group Work 1. Why does it suffice to study the above paths in order to derive the *optimal* solution? For example, what about connecting 1 and 6 or connecting 2 and 3?

2. How cumbersome is it to calculate the values c(i) for all intervals $i \in \{1, \ldots n\}$? Do the $c(i)$ values change during the algorithm? How do we calculate them most efficiently?

3. What are the START and END points for?

Answers 1. Any additional lines would add either sub-optimal or invalid paths. We can also not remove any of the existing lines shown above as these correspond to valid paths.

2. Based on their definition, the $c(i)$ values do *not* change during the algorithm. We can obtain $c(i)$ for a given interval i by considering intervals $(i-1), (i-2)$ etc. and stopping as soon as the interval is compatible with interval i.
3. A set T of compatible intervals can correspond to any subset of input set S. By introducing a START and an END interval which are each assigned no weight, we can write the algorithm more elegantly.

Rehash The *lines* connecting intervals $i \in \{1, \ldots, n\}$ the END interval can be formerly viewed as "reverse $c(i)$" values, i.e. values $\tilde{c}(i)$ defined as follows:

$\tilde{c}(j)$:= smallest values of $i, j < i$ such that intervals i and j do not overlap; $c(j) =$ END $= n + 1$ if no such value of $i \in \{1, \ldots, n\}$ exists.

[Any green line between i and $c(i) = j$, $i, j \in \{1, \ldots, n\}$, can thus be viewed as pink line between $j = \tilde{c}(j)$ and j.]

Devising an equivalent new algorithm that does not require $c(j)$ values

Definitions

- S and S_j as before, n is the number of intervals, i.e. $n = |S|$
- $w(j)$:= the sum of weights for the best interval scheduling for set S_j *that ends and includes interval* j (different with respect to previous definition)
- $$t(j,i) := \begin{cases} 1 & \text{if interval } i \text{ and } j \text{ are compatible (i.e. do not overlap)} \\ 0 & \textit{else} \end{cases}$$
- $\mathbb{N}_i^0 := \{0, 1, 2, \ldots, i\} = \mathbb{N}_i \cup \{0\}$
- START and END are to fake intervals whose indices are 0 and $n+1$ respectively. They are assigned a weight of 0 and are compatible with all other intervals.

Algorithm WeightedIntervalScheduling(S) (Version 2)

- initialization: $w(0) = 0$ ((START Interval))
- recursion:

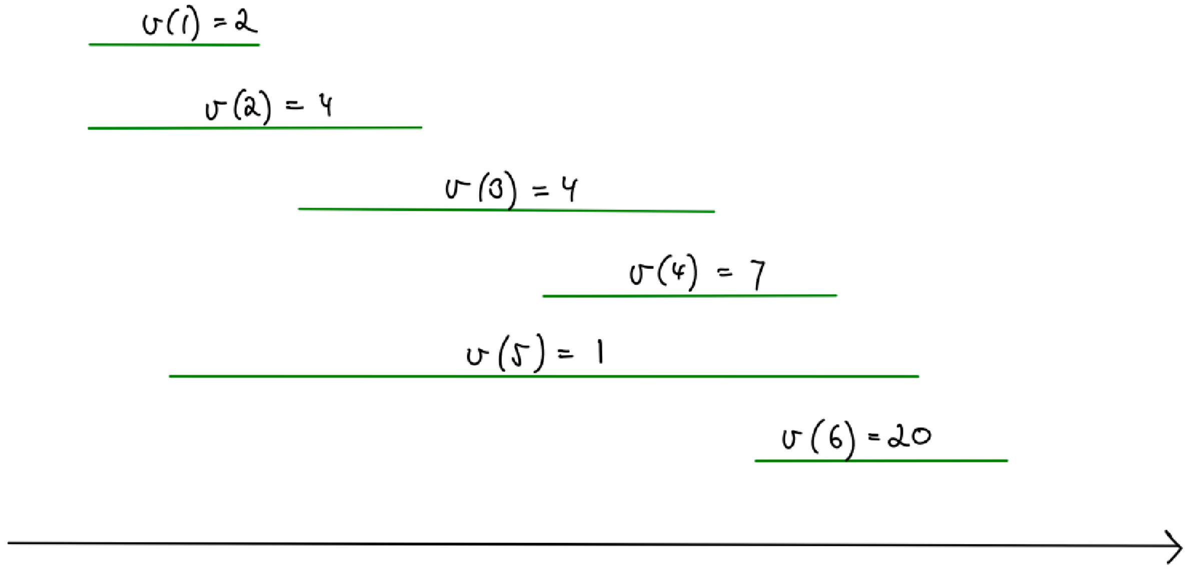
```
for ($i = 1, \ldots, n$){
    $w(i) = \max_{j \in \mathbb{N}_{i-1}^0} \{w(j) \cdot t(j,i)\} + v(i)$
}
```

- termination: $w(n+1) = \max_{j \in \mathbb{N}_n^0} \{w(j) t(j, n+1)\}$
  [We do not add $v(n+1)$ here as $v(n+1) = v(0) = 0$]

- traceback:

```
$T = \varnothing$
$best_i = \text{argmax}_{j \in \mathbb{N}_n^0}\{w(j)t(j, n+1)\}$        [ i.e. $best_i = j \in \mathbb{N}_n$ that maximizes $w(j)^n t(j, n+1)$]
while($best_i > 0$){
    add $best_i$ to set $T$
    $best_i = \text{argmax}_{j \in N^0_{best_i - 1}}\{w(j)t(j, best_i)\}$
}
return $T$
```

Example Execute algorithm on the following set of intervals (group work):



Answer (a) initialization: $w(0) = 0$

(b) recursion:

$$
\begin{aligned}
(3) \qquad w(1) &= \max_{j \in \mathbb{N}_0^0 = \{0\}} \{w(j)t(j,1)\} + v(1) = 0 + 2 = 2 \\
&(t(0,1) = 1) \\
w(2) &= \max_{j \in \mathbb{N}_1^0 = \{0,1\}} \{w(j)t(j,2)\} + v(2) = w(0) + v(2) = 0 + 4 = 4 \\
&(t(0,2) = 1) \\
&(t(1,2) = 0) \\
(2) \qquad w(3) &= \max_{j \in \mathbb{N}_2^0 = \{0,1,2\}} \{w(j)t(j,3)\} + v(3) \\
&= \max\{w(0), w(1)\} + v(3) = w(1) + v(3) = 2 + 4 = 6 \\
&(t(0,3) = 1) \\
&(t(1,3) = 1) \\
&(t(2,3) = 0) \\
w(4) &= \max_{j \in \mathbb{N}_3^0 = \{0,1,2,3\}} \{w(j)t(j,4)\} + v(4) \\
&= \max\{w(0), w(1), w(2)\} + v(4) = w(2) + v(4) = 4 + 7 = 11 \\
&(t(0,4) = 1) \\
&(t(1,4) = 1) \\
&(t(2,4) = 1) \\
&(t(3,4) = 0) \\
w(5) &= \max_{j \in \mathbb{N}_4^0 = \{0,4\}} \{w(j)t(j,5)\} + v(5) = w(0) + v(5) = 1
\end{aligned}
$$

Note: This is *different* from the result obtained from algorithm 1. The difference is due to the different definitions in $w(j)$ in two algorithms. In *this* algorithm, $w(5) = 1$ is the weight of the best path up to and *including* interval 5, whereas in the previous algorithm, $w(5) = 11$ is the weight of the best path up to (but not necessarily including) interval 5 (it includes intervals 2 and 4, but not 5!).

$$
\begin{aligned}
(1) \qquad w(6) &= \max_{j \in \mathbb{N}_5^0 = \{0,1,2,3,4,5\}} \{w(j)t(j,6)\} + v(6) \\
&= \max\{w(0), w(1), w(2), w(3)\} + v(6) = 6 + 20 = 26 \\
&(t(0,6) = 1) \\
&(t(1,6) = 1) \\
&(t(2,6) = 1) \\
&(t(3,6) = 1) \\
&(t(4,6) = 0) \\
&(t(5,6) = 0)
\end{aligned}
$$

(c) termination:

$$(*) \qquad w(7) = \max_{j \in \mathbb{N}_6^0 = \{0,1,2,3,4,5,6\}} \{w(j)t(j,7)\}$$
$$= \max\{w(1), w(2), w(3), w(4), w(5), w(6)\} = w(6) = 26$$
$$(t(0,7) = 0) \text{ (!!!!)}$$
$$(t(1,7) = \ldots = t(6,7) = 1)$$

(d) traceback:

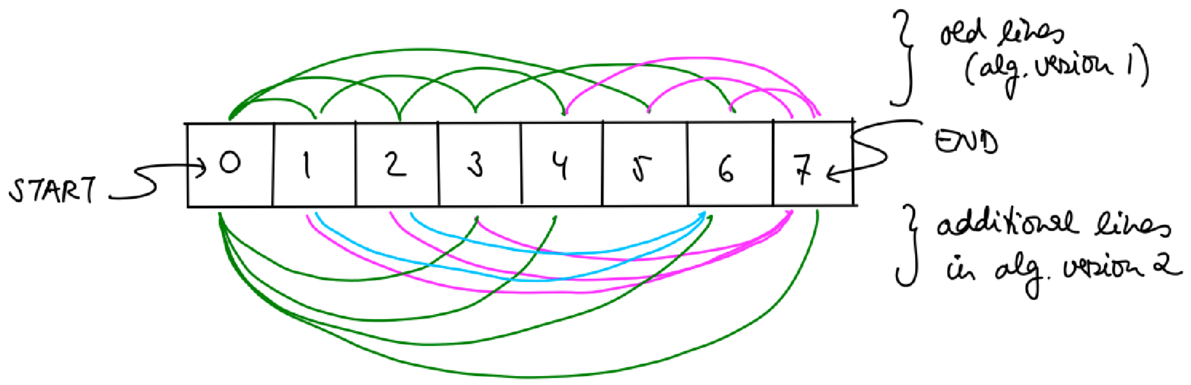
- $T = \varnothing$
- $best_i = \text{argmax}_{j \in \mathbb{N}_6^0}\{w(j)t(j,7)\} = 6$ (see $(*)$ above)
- enter while-loop because $best_i = 6 > 0$:
  - $T = \{6\}$
  - $best_i = \text{argmax}_{j \in \mathbb{N}0_5}\{w(j)t(j,6)\} = 3$ (see (1) above)
  - $T = \{3,6\}$
  - $best_i = \text{argmax}_{j \in \mathbb{N}0_2}\{w(j)t(j,3)\} = 1$ (see (2) above)
  - $T = \{1,3,6\}$
  - $best_i = \text{argmax}_{j \in \mathbb{N}0_0}\{w(j)t(j,1)\} = 0$ (see (3) above)
- exit while loop because $best_i = 0$
- return $T = \{1,3,6\}$ as answer

Runtime Analysis of the Above Algorithm

As is apparent from the above pseudo-code, the algorithm runs in $O(n^2)$ time.

Group Work Which visualization does this algorithm have? And how does this compare to the visualization of the first algorithm?

Answer Draw the same figure as before (including START and END points) but now have a connecting line between *any pair* of intervals $(i,j), i,j \in \{1,\ldots,n\}$ that are compatible. Any inverse $i \in \{1,2,\ldots,n\}$ is compatible with START and END, i.e. we have corresponding lines.


old lines (alg. version 1)
END
START
0
1
2
3
4
5
6
7
additional lines in alg. version 2

Group Work What are the differences between the two algorithms?

Answer (1) The main difference is that the

- first algorithm considers paths via interval i even if the weight of the interval i is *not* added to the total weight of the path (and the traceback therefore does *not* include the interval i to return set T. See for loop: $w(i) = \max\{w(i-1), v(i) + w(c(i))\}$
- whereas the second version of the algorithm considers only paths involving intervals i whose weight is added to the overall weight of that path. See for loop: $w(i) = \max_{j \in \mathbb{N}^0_{i-1}}\{w(j)t(j,i)\} + v(i)$

(2) The first algorithm requires $c(i)$ values, whereas the second one refers to $t(j,i)$ values. The second algorithm therefore considers more paths, including all those that the first algorithm considers.

The additional paths that the second algorithm considers can all be shown to be suboptimal ones, but all of them are valid, i.e. do not contain any pair of mutually exclusive intervals.

Group Work Which key feature makes the algorithms (in particular alg. version 2) so efficient?

Answer The brute-force approach to finding the highest-scoring path would be to list all possible paths and to then calculate their overall score (i.e. weight) and rank the remaining paths accordingly.

Algorithm 2, however, discards potential sub-paths as it loops from $i = 1$ to $i = n$ in the recursion because it discards all but a simple previous subpath due to the max operation.

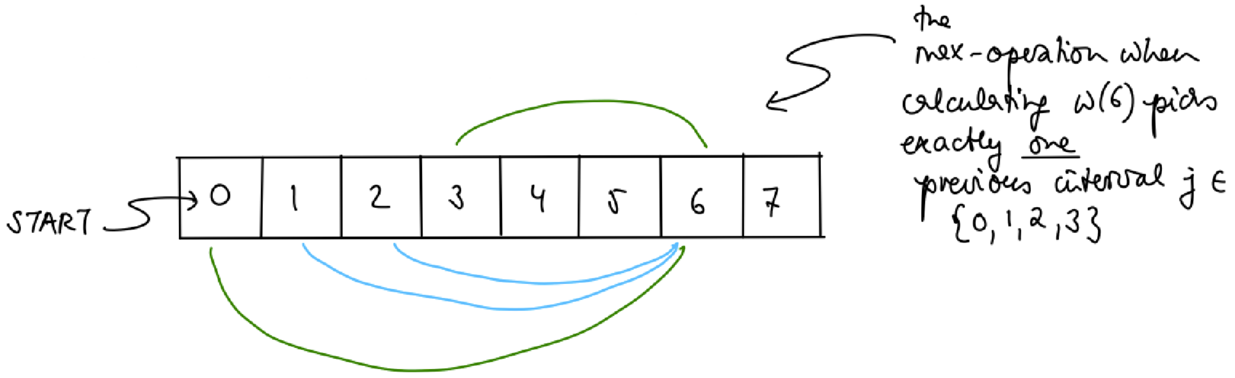
Answer (cont'd) :

Reminder: in the recursion of the algorithm we have:

```
• for ( i = 1 ... n ) {
      $w(i) = \max_{j \in \mathbb{N}^0_{i-1}} \{w(j) \cdot t(j,i)\} + v(i)$
  }
```

Suppose we are at $i = 6$:

- the only values of $j \in \mathbb{N}^0_{i-1}$ with $t(j,i) = 1$ are $j \in \{0,1,2,3\}$.


START
0
1
2
3
4
5
6
7
the max-operation when calculating w(6) picks exactly one previous interval j ∈ {0,1,2,3}

- when we thus calculate

$$\begin{aligned} w(6) &= \max_{j \in \{0,1,2,3\}} \{w(j) \cdot t(j,6)\} + v(6) \\ &= \max j \in \{0,1,2,3\}\{w(j)\} + v(6) \end{aligned}$$

  i.e. the max-operation picks the highest scoring subpath that links one of the earlier intervals ($j \in \{0,1,2,3\}$) to the current interval $i = 6$ without causing a clash (i.e. consider only j-values with $t(j, 6 \neq 0)$.

  Note that this decision among the j-values $j \in \{0,1,2,3\}$ is *not* influenced by $v(6)$ itself, but *only* depends on the best sub-path that ends at j and whose weight is specified by $w(j)$.

Conclusion: The algorithm discards sub-optimal paths as it goes along.

Big Question: Does it possibly discard any subpath that would belong to the overall winning path? If this was the case, the algorithm would not be guaranteed to derive the optimal solution.

Theorem: The algorithm (version 2) retrieves the optimal solution.

Proof: Strategy:

We first show that $w(i)$ correspond to the weight of the best path up to and including interval i.

This immediately implies that $w(n+1)$ is the weight of the best overall path.

We then show that the traceback procedure recovers the underlying path that corresponds to $w(n+1)$.

(1) Proof by induction:

$i = 0$: $w(0) = 0$ is the weight of the best path starting at interval 0 and ending at interval 0

Suppose that $w(i)$ corresponds to the weight of the best path that starts at interval 0 and finishes at interval i.

When we now calculate $w(i+1)$ via:

$$w(i+1) = \max_{j \in \mathbb{N}_i^0} \{w(j)t(j, i+1)\} + v(i+1)$$

we know that the max-operation will pick the interval $j \in \mathbb{N}_i^0 = \{0, 1, \ldots, i\}$ that is

- compatible with interval $(i+1)$, i.e. $t(j, i+1) = 1$, and
- has the highest value $w(j)$

We already know that the values $w(0)$ to $w(i)$ correspond to the weight of the best state path up to that interval.

By adding the weight of the current interval $(i+1)$, i.e. $v(i+1)$ to the return value of the max-operation. We therefore obtain the weight of the best path from interval 0 to interval $(i+1)$. *Note:* For $i = n$, we do not need to add weight $v(n+1)$ as this is 0.

(2) We know that $w(n+1)$ corresponds to the weight of the best path. In order to derive the corresponding path, we start at the END interval $(n+1)$ and proceed via intervals that were identified via previously executed max-operations until we reach the START interval 0.

$w(n+1)$ derives from the interval $j \in \mathbb{N}_n$ that maximizes $w(j)t(j, n+1)$ (see termination step of algorithm 2).

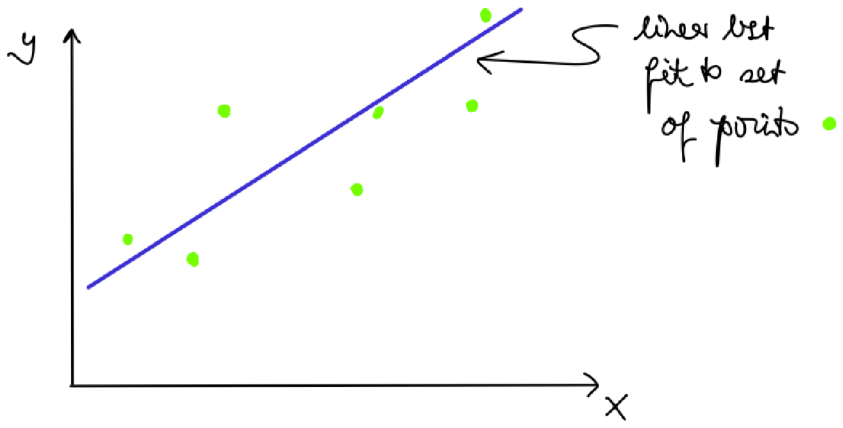
This is equal to the first value of $Best_i$ in the traceback part of the algorithm which is an interval on the optimal path which we therefore add to the return set T

When at interval $Best_i$, we know that the previous interval j is the interval $j \in \mathbb{N}_{best_{i-1}}^0$ that maximizes the expression $w(j)t(j, Best_i)$ (see termination step of algorithm 2)

We therefore set the new value of $Best_i$ to $\arg\max_{j \in \mathbb{N}_{Best_{i-1}}^0} \{w(j)t(j, Best_i)\}$, add interval $Best_i$ to T and continue until we reach the START, i.e. $Best_i = 0$. $\square$

7.2 Example 2: Segmented least squares problem

Problem Setting: Given a set S of n points in two dimensions, determine a linear fit to these points that minimizes the overall distance to all n points.



Definitions

- S:= $\{p_1, p_2, \ldots, p_n\}$, where points $p_i = \binom{x_i}{y_i]}$ and the points are ordered according to their x-coordinates, i.e. $x_1 < x_2 < \ldots < x_n$ Note n:= $|S|$.
- a line in two dimensions is defined via a function

$$l(x) := a \cdot x + b = y$$

i.e. depends on two parameters $a, b \in \mathbb{R}$. x and y are elements from $\mathbb{R}^2$, i.e. small vectors.

a is the slope of the line and b the intersection of the line as the y-axis.

- the distance between a line l (as defined above) and a set of points S, $n = |S|$ is defined as:

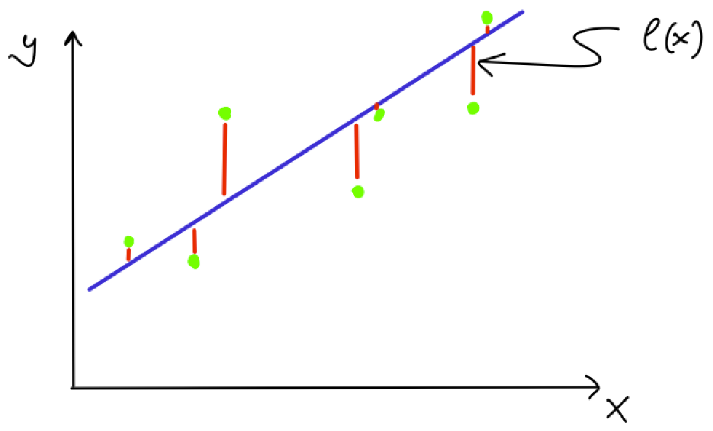
$$e(l,s) := \sum_{i=1}^{n} (y_i - l(x_i))^2 = \sum_{i=1}^{n} (y_i - (ax_i + 6))^2$$
$$= \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

i.e. the sum of the squared distances between the y-coordinates of points and the respective y-coordinates of the line at the same x_i-coordinate.

y_i = y-coordinate of point p_i

$l(y_i)$ = y-coordinate of line at x_i

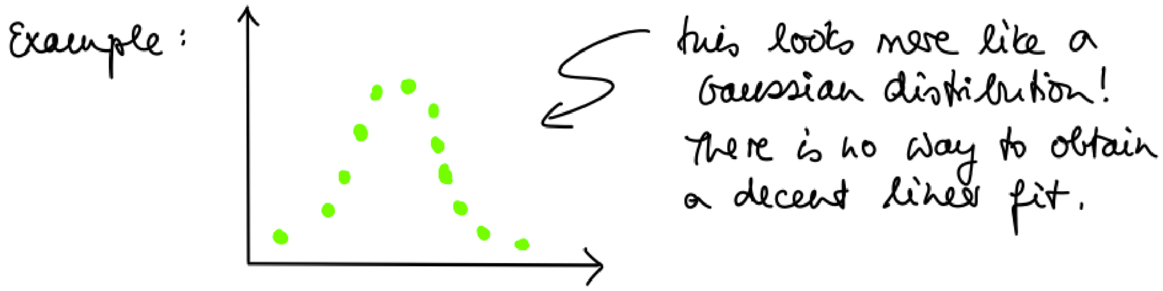
We can visualize $e(l,s)$ as follows:



$$e(l,S) = \sum_{i=1}^{n} (|)_i^2$$

Goal Given the n points of set S_i we we want to find the line, i.e. values a and b_i that minimizes $e(l,s)$

Comment It does not always make sense to want to fit a line to a given set of points.


Example:
this looks more like a Gaussian distribution! There is no way to obtain a decent liner fit.

Problem In order to derive the optimal combination of a and b that correspond to the line of best fit, we need to express a and b as a function of points in S.

Group Work How do we go about this?

Answer Calculus. Interpret $e(l,S)$ as function $g(a,b)$ of the two free parameters as S is kept fixed.

Set partial derivatives $\frac{\delta g}{\delta a} = 0$ and $\frac{\delta g}{\delta b} = 0$ and we use these two equations to obtain the solutions for a and b.

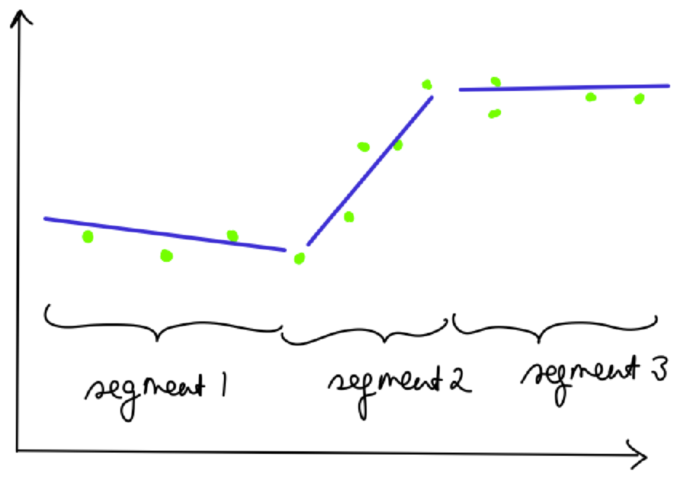
(After a lengthy calculation, one obtains...)

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$b = \frac{\sum_{i=1}^{n} y_i - a \cdot \sum_{i=1}^{n} x_i}{n}$$

Those a and b values define a line $l(x) = ax + b$ that provides the best linear fit to points $S = \{p_1, p_2, \ldots, p_n\}$ with $p_i = \binom{x_i}{y_i}$, i.e. the lines that minimizes $e(l, S)$.

Motivation Often, the n points in set $p = \{p_1, p_2, \ldots, p_n\}$ are such that a simple line produces an insufficient fit.



In the above example, a piecewise-linear fit would reduce the total error and yield an overall better fit than a single line.

Definitions

- a partition P into N segments is a decomposition of set $P = \{p_1, p_2, \ldots, p_n\}$ into N subsets S_i, $i \in \mathbb{N}_N$ such that $\cup_{i=1}^{N} S_i = p$, $S_i \neq \emptyset$, $S_i \cap S_j = \emptyset$.
  for $i \neq j$ and $S_i = \{P_{s(i)}, P_{s(i)+1}, \ldots, P_{e(i)-1}, P_{e(i)}\}$, where $s(i)$ is the start index and $e(i)$ is the end-index of segment S_i and where $e(i-1)+1 = s(i)$ and $e(i)+1 = s(i+1)$.
  Example:

$$\begin{aligned} p &= \{p_1, \ldots, p_{15}\} \\ S_1 &= \{p_1, p_2, p_3\} \\ S_2 &= \{p_4, p_5\} \\ S_3 &= \{p_6, \ldots, p_{15}\} \end{aligned}$$

- $e_{i,j}$ is defined as the minimal error $e(l, S)$ for the optimal line $l(x)$ through points $\{P_i, \ldots, P_j\}$, $i \leq j$ (i.e. the line $l(x) = a \cdot x + b$ with a and b determined by formula $(*)$ before)

- $P(S_1, \ldots, S_N)$:= total penalty for a partition of P into segment S_i, $i \in \mathbb{N}_N$

$$= \sum_{i=1}^{N} \left(e_{s(i),e(i)} + c\right)$$

  where $C \in \mathbb{R}_+$ denotes a constant penalty for *each* segment (segment penalty).
- $M(i)$:= minimum total penalty for all points $P_1, P_2, \ldots, P_i$

Goal Given a set of n points $P = \{p_1, p_2, \ldots, p_n\}$ with $x_1 < x_2 < \ldots < x_n$, find the optimal partition into segments that minimize the overall penalty P defined above.

Group Work What happens to the optimal partition of P

1. as we increase C?
2. as we decrease C? (What about $C = 0$)?

Answers

1. If C increases, having more segments becomes more costly. The optimal partition of P thus tends to contain fewer segments as we increase C.
2. if C decreases, the optimal partition tends to contain more segments.
   For $C = 0$, we can obtain the optimal solution by fitting a simple line through each of the n points in P, i.e. by having n segments that each contain only a a single point of P. Each line would provide a perfect fit to its corresponding single point (set $b = y_i$ and $a = 0$ in those cases, i.e. $l(x) = y_i$).

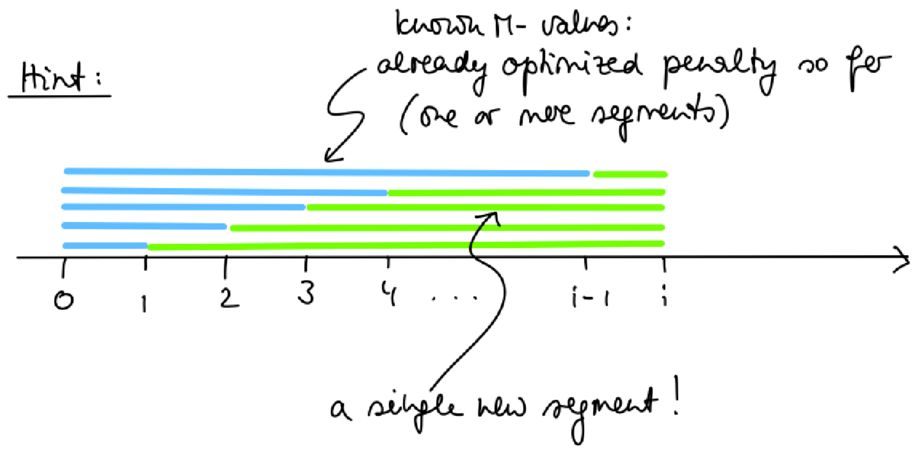
Question Given a fixed segment penalty $C \in \mathbb{R}_+$, derive the corresponding optimal partition of P_i i.e. the corresponding segments $S_1, S_2, \ldots, S_N$ that minimize

$P(S_1, S_2, \ldots, S_N)$, i.e. the overall penalty.

Key Observation Suppose we know the values of $M(1)$ to $M(i-1)$, where (reminder)

$M(i)$ corresponds to the minimum penalty up to points p_i

Group Work Can we use those values $M(1)$ to $M(i-1)$ to derive $M(i)$?



Answer

$$M(i) = \min_{j \in \mathbb{N}_i}\{ \underbrace{M(j-1)}_{\substack{\text{segment}(s)\\ \text{covering}\{p_1,\dots,p_{j-1}\}}} + \underbrace{e(j,i)+C\}}_{\substack{\textit{one}\text{ segment for}\\ \{p_j,\dots,p_i\}}} \quad \text{(key idea)}$$

Observations The previous equation already suggests the recursion of the corresponding algorithm. Note that $j = 1$ in the min-bracket yields the term $M(0) = 0$ and a term $e(0,i) + C$ which corresponds to a single segment comprising all points from $p_1, \dots p_i$ (visually, this would correspond to a single green line in the above figure (not shown)).

Definition Segmented-Least-Squares(P, N) algorithm

```
• allocate 2-dimensional $n \times n$ matrix $E$
• for ($i = 0 \dots n$) {
      for ($j = 0 \dots i$) {
          calculate $e_{j,i}$
      }
  }

• allocate l-dimensional array $M$ (length $n$)
• $M[0] = 0$
• for ($i = 0 \dots n$) {
      $M[i] = \min_{j \in \mathbb{N}_i}\{M[j-1] + e_{j,i} + C\}$
  }

• $T = \emptyset$      (set of all $s(i)$ coordinates of optimal partition)
      $best_i = \arg\min_{j \in \mathbb{N}_n}\{M[j-1] + e_{j,n} + C\}$      (segment from $\{P_{best_i}, P_n\}$)
• while ($best_i > 0$) {
      merge $best_i$ and $T$

  $best_i = \arg\min_{j \in \mathbb{N}_{best_{i-1}}}\{M[j-1] + e_{j,best_{i-1}} + C\}$      (segment from $\{P_{best_i}, P_{best_{i-1}}\}$)
```

- return set T and $|T| = N$ (the number of segments in optimal partition)

Note: The first section of the algorithm is to calculate all errors for all possible linear fits. The second one derives the optimal total penalty (which is the recursion part), and the last part is the traceback to derive the optimal partition.

Group Work Which value of M contains the minimum penalty?

Answer $M[n]$ because it corresponds (per definition) to the minimum overall penalty from p_1 up to p_n, i.e. all of set P.

Proof of algorithm's correctness as in 7.1 by induction

Time Requirements of Algorithm :

- calculation of E-matrix: $O(n^3)$,
  because n^2 values $e_{j,i}$ are calculated which each require $O(n)$ time.
- recursion: $O(n^2)$
- traceback: $O(n^2)$
- $\Rightarrow$ Overall: $O(n^3)$

This algorithm is therefore significantly more efficient than a brute-force approach which would first list all possible partitions of P and then rank them according to their overall penalty.

7.3 Sequence Alignment

Motivation We would like to find words that are similar to a word under consideration.

Example STOP should identify TOPS via the sequence alignment

```
STOP-
-TOPS
```

where "–" is a gap character.

Motivation We need to find a way to quantitatively express how similar two words are and we need a way to globally align them as shown in the above example.

Key Idea • judge any possible global alignment between two given words based on

1. how many gaps, and
2. how many mis-matches it contains.

Definitions • a mis-match consists of two characters p and q, $p \neq q$, that are aligned

- sequence $X = (x_1, x_2, \dots, x_{L_x})$ of length L_x
- sequence $Y = (y_1, y_2, \dots, y_{L_y})$ of length L_y
- a (global) alignment A between two sequences X and Y is a pair of strings $\tilde{X} = (\tilde{x}_1, \dots, \tilde{x}_{L_A})$ and $\tilde{Y} = (\tilde{y}_1, \dots, \tilde{y}_{L_A})$ of alignment length L_A, where $\tilde{x}_i \in A_x \cup \{-\}$ and $\tilde{y}_i \in A_y \cup \{-\}$, (where A_x and A_y denote the alphabets from which the characters x and y, respectively, derive from) such that
  - when we remove the gaps from $\tilde{x}$, we get x
  - when we remove the gaps from $\tilde{y}$, we get y
  - two gaps are never aligned (i.e. $\tilde{x}_i = \tilde{y}_j =$ "–" for no $i \in \{1, 2, \dots, L_A\}$)
- a_i = alignment column number i

$$\tilde{X} = S\ T\ O\ P\ -$$
$$\tilde{y} = \underbrace{-}_{a_1}\ \underbrace{T}_{a_2}\ \underbrace{O}_{a_3}\ \underbrace{P}_{a_4}\ \underbrace{S}_{a_5}$$

here 5 alignment columns
and the alignment length is 5

Group Work Determine the alignment M for the following example

```
STOP-
-TOPS
```

Answer :

$\tilde{X}$ = (S,T,O,P,-)

$\tilde{Y}$ = (-,T,O,P,S) and $L_A = S$ and $A = (\tilde{X}, \tilde{Y})$

Definitions • $\delta \in \mathbb{R}_+$ is the gap penalty, i.e. the penalty assigned to each gap (the above example has two gaps)

- α_{pq} is the mis-match cost for aligning p and q (typically, we choose $\alpha_{pq} = 0$ for $p = q$ and $\alpha_{pq} > 0$ for $p \neq q$)
- the cost of alignment A between sequences X and Y $c(M, X, Y)$ is the sum of the corresponding gap and mis-match costs for each column i in the alignment of length L_A, i.e.

$$c(A, X, Y) = \sum_{i=1}^{L_A} c_i \qquad \text{where } c_i \text{ is the cost of alignment column } c_i$$

Group Work Given A, X, and Y, how do you calculate the corresponding cost $c(A, X, Y)$ for the above example?

Answer

$$c(A,X,Y) = \underbrace{\delta}_{c_1} + \underbrace{\alpha_{T,T}}_{c_2} + \underbrace{\alpha_{0,0}}_{c_3} + \underbrace{\alpha_{P,P}}_{c_4} + \underbrace{\delta}_{c_5}$$

Goal Given two sequences X and Y as defined above, determine the alignment with the smallest total cost, i.e. find A such that $c(A,X,Y)$ is minimized, i.e. want to find

$$A^* = \arg\min_A \{c(A,X,Y)\} = \qquad \text{optimal alignment of } X \text{ and } Y$$

Remark The goal makes sense given the definitions above because the cost of the alignment will decrease as the quality of the alignment increases.

Question What is the number of possible alignments for two given sequences x (of length L_x) and y (of length L_y)?

Group Work What is the shortest and longest length of an alignment?

Answer

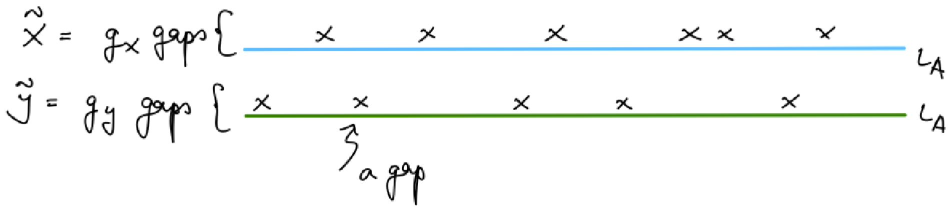
- The shortest alignment length is $\max\{L_x, L_y\}$ which corresponds to no gaps in the longer sequence and $\max\{L_x, L_y\} - \min\{L_x, L_y\}$ gaps in the shorter sequence.
- The longest alignment is $L_x + L_y$ long and corresponds to the case where all characters in x and all characters in y are aligned to gaps.

Definitions

- $N(L_x, L_y)$ = number of possible alignments between x of length L_x and sequence y of length L_y
- $L_{max} = L_x + L_y$, maximal alignment length
- $L_{min} = \max\{L_x, L_y\}$, minimal alignment length
- L_A = length of the alignment
- $g_x = L_A - L_x$ = number of gaps inserted in x
- $g_y = L_A - L_y$ = number of gaps inserted in y

Calculating $N(L_x, L_y)$

1. What is the number of possible alignments for a fixed alignment length L_A? We already know that $L_A \in \{L_{max}, \ldots, L_x + L_y\}$.
   Given $L_A \Rightarrow$ know the number of gaps in $\tilde{x}$ and $\tilde{y}$, i.e. know g_x and g_y.
   (a) There are $\binom{L_A}{g_x} = \binom{L_A}{L_A - L_y}$ different ways of distributing g_x (indistinguishable) gaps in L_A positions in $\tilde{x}$.
   (b) For any of the possibilities of (a), there are $\binom{L_x}{g_y} = \binom{L_x}{L_A - L_y}$ different ways of inserting g_y (indistinguishable) gaps into any of the L_X sequence positions in sequence $\tilde{y}$ that do *not* correspond to a gap in $\tilde{x}$ (we mustn't align a gap in $\tilde{x}$ to a gap in $\tilde{y}$).



Can combine any possible case (a) for sequence $\tilde{x}$ in the alignment with any possible case (b) for sequence $\tilde{y}$ in the alignment:

(a),(b) $\Rightarrow \binom{L_A}{L_A - L_x}\binom{L_x}{L_A - L_y}$ is the number of possible alignments of length L_A

2.

$$N(L_x, L_y) = \sum_{L_A = L_{min}}^{L_{max}} \binom{L_A}{L_A - L_x}\binom{L_x}{L_A - L_y}$$

where L_{min} is the shortest and L_{max} the longest possible alignment length.

Example

$$N(L, L) \simeq \frac{(1+\sqrt{2})^{(2L+1)}}{\sqrt{L}}$$

e.g. $L = 40 \Rightarrow N(40, 40) \simeq 3.8 \cdot 10^{29}$ possible alignments!

Conclusion Even for two rather short sequences, the number of possible alignments grows too fast to explicitly list and then rank them in a brute-force approach!

Observation 1 In any possible alignment A of two sequences x and y, any possible pair of sequence positions (m, n) is

- aligned (i.e. $\tilde{x}_i = x_m = y_n = \tilde{y}_i$ for some $i \in \{1, 2, \ldots, L_A\}$) or
- not aligned to each other in A

Remark Statement 6.14 in the textbook and the corresponding proof are wrong.

Definitions • $M(i, j)$ = minimum cost of an alignment between subsequences $x_i = (x_1, x_2, \ldots, x_i)$ and $y_i = (y_1, y_2, \ldots, y_j)$, $i \in \mathbb{N}_{L_x}$, $j \in \mathbb{N}_{L_y}$

Observation 2 Suppose we already know values $M(i, j-1)$, $M(i-1, j-1)$ and $M(i-1, j)$ then we can derive $M(i, j)$ as follows:

$$\begin{aligned} M(i,j) = \min\{ & M(i-1, j-1) + \alpha_{x_i y_j}, && \text{(align } x_i \text{ and } y_j) \\ & M(i-1, j) + \quad \delta, && \text{(align } x_i \text{ to a gap)} \\ & \underbrace{M(i, j-1)}_{\text{already known values}} + \underbrace{\delta}_{\text{NEW!}} \} && \text{(align } y_j \text{ to a gap)} \end{aligned}$$

Reminder $M(i, j)$ is the cost of the optimal alignment of x_i and y_j. δ links already known answer to $M(i, j)$ which is where we are now.

Group Work In the above equation, why don't we consider the cases

1. $M(i-1, j) + \alpha_{x_i y_j}$
2. $M(i-1, j-1) + \delta$
3. $M(i, j-1) + \alpha_{a_i y_j}$

when calculating the value of $M(i, j)$?

Answer
1. $M(i-1, j)$ corresponds to an alignment of x_{i-1} and y_j, i.e. y_j is already part of that alignment and *cannot* be aligned to x_i (by $\alpha_{x_i y_j}$) as well.
2. $M(i-1, j-1)$ corresponds to an alignment of x_{i-1} and y_{j-1} and by aligning either x_i or y_j to a gap (by δ) does not amount to an alignment of x_i and y_j (which is what $M(i, j)$ corresponds to).
3. $M(i, j-1)$ corresponds to an alignment of sequences x_i and y_{j-1}. By aligning x_i to y_j (see $\alpha_{x_i y_j}$) we would be aligning x_i again, which is not possible.

Conclusion Observation 2 suggests the recursion of a dynamic programming algorithm.

Definition Optimal Alignment (x, y)

```
• allocate 2-dimensional $(L_x + 1) \times (L_y + 1)$ matrix $M$
• $M[0, 0] = 0$      (initialization step)
• for $(i = 1 \ldots L_x)\{M[i, 0] = i \cdot \delta\}$      (1)
• for $(i = 1 \ldots L_y)\{M[0, i] = i \cdot \delta\}$      (2)
• for $(j = 1 \ldots L_y)\{$
      for $(i = 1 \ldots L_x)\{$
          $M(i,j) = \min \begin{cases} M(i-1, j-1) + \alpha_{x_i y_j} \\ M(i-1, j) + \delta \\ M(i, j-1) + \delta \end{cases}$
      $\}$
  $\}$
• return $M[L_x, L_y]$ (cost of optimal alignment between $x$ and $y$)
```

Note As for the previous examples, we could again add a traceback procedure to the alignment to retrieve the corresponding optimal alignment A^* itself (in addition to the cost of the optimal alignment $= M[L_x, L_y]$).

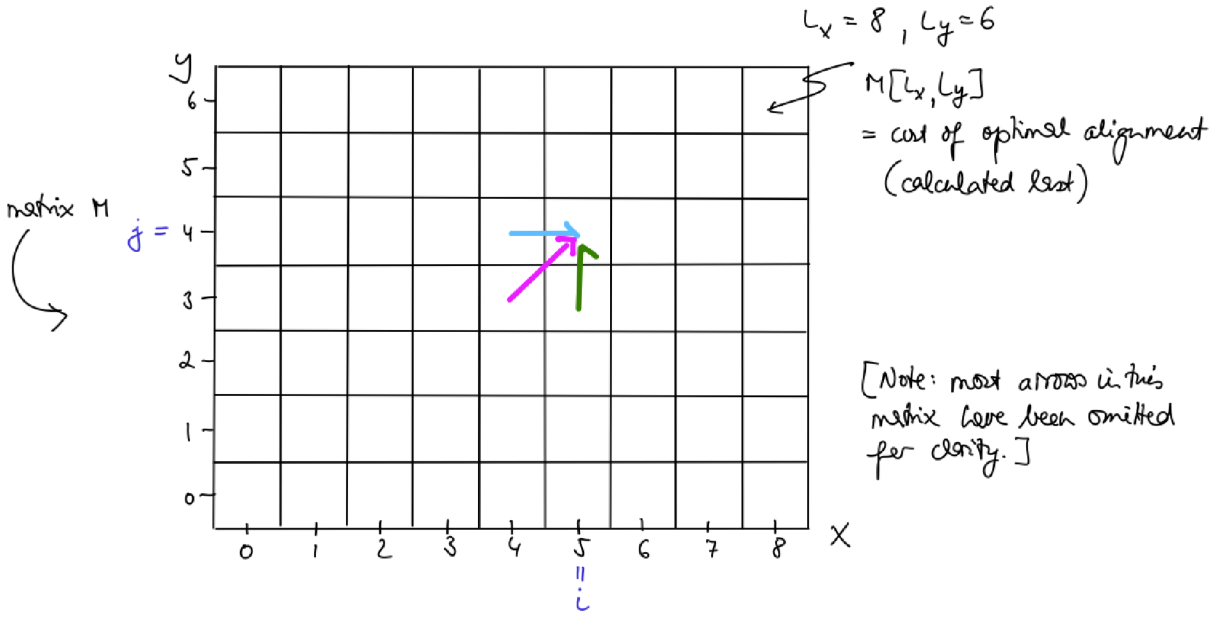
Group Work What are lines (1) and (2) above for?

Answer They correspond to alignments that start with i gaps in either $\tilde{x}$ or $\tilde{y}$.

$M[i,0]$ corresponds to an alignment of x_i to i gaps in $\tilde{y}$

$M[0,i]$ corresponds to an alignment of y_i to i gaps in $\tilde{x}$

Comment We can visualize the algorithm as follows:



$L_x = 8$, $L_y = 6$
$M[L_x, L_y]$
= cost of optimal alignment (calculated last)
matrix M
$j = 4$
$i = 5$
y
x
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6
[Note: most arrows in this matrix have been omitted for clarity.]

reminder: recursion

$$M(i,j) = \min \begin{cases} M(i-1,j-1) + \alpha_{x_i y_j} & (1) \\ M(i-1,j) + \delta & (2) \\ M(i,j-1) + \delta & (3) \end{cases}$$

Group Work Using

$$\alpha_{pq} = \begin{cases} \alpha & p \neq q \\ 0 & p = q \end{cases}$$

$\alpha \in \mathbb{R}^+$ and $\delta = 2\alpha$ as mismatch and gap penalties, respectively, calculate the optimal alignment between

X = HELLO	$L\_x = 5$
y = ELLA	$L\_y = 4$

Answer In that case, M is a 6×5 matrix.

$$
\begin{aligned}
&M[0,0] = 0 && \\
&M[i,0] = i \cdot \delta = i2\alpha && \text{for } i \in \{1,\dots,L_x\} \\
&M[0,i] = i \cdot \delta = i2\alpha && \text{for } i \in \{1,\dots,L_y\}
\end{aligned}
$$

the recursion amounts to

```
for($j = 1 \dots 4$) {
    for($i = 1 \dots 5$) {
        $M(i,j) = \min \begin{cases} M(i-1,j-1) + \alpha \cdot \delta_{x_i y_j} \\ M(i-1,j) + 2\alpha \\ M(i,j-1) + 2\alpha \end{cases}$
    }
}
```

$$
\begin{aligned}
j=1: \quad & M[1,1] = \min\{\underbrace{M[0,0]}_{0} + \underbrace{\alpha}_{H\neq E}, \underbrace{M[0,1]}_{2\alpha} + 2\alpha, \underbrace{M[1,0]}_{2\alpha} + 2\alpha\} = \alpha \\
& M[2,1] = \min\{\underbrace{M[1,0]}_{2\alpha} + \underbrace{\alpha}_{E=E}, \underbrace{M[1,1]}_{\alpha} + 2\alpha, \underbrace{M[2,0]}_{4\alpha} + 2\alpha\} = 2\alpha \\
& M[3,1] = \min\{\underbrace{M[2,0]}_{4\alpha} + \underbrace{\alpha}_{L\neq E}, \underbrace{M[2,1]}_{2\alpha} + 2\alpha, \underbrace{M[3,0]}_{6\alpha} + 2\alpha\} = 4\alpha \\
& M[4,1] = \min\{\underbrace{M[3,0]}_{6\alpha} + \underbrace{\alpha}_{L\neq E}, \underbrace{M[3,1]}_{4\alpha} + 2\alpha, \underbrace{M[4,0]}_{8\alpha} + 2\alpha\} = 6\alpha \\
& M[5,1] = \min\{\underbrace{M[4,0]}_{8\alpha} + \underbrace{\alpha}_{L\neq O}, \underbrace{M[4,1]}_{6\alpha} + 2\alpha, \underbrace{M[5,0]}_{10\alpha} + 2\alpha\} = 8\alpha \\
j=2: \quad & M[1,2] = \min\{\underbrace{M[0,1]}_{2\alpha} + \underbrace{\alpha}_{H\neq L}, \underbrace{M[0,2]}_{4\alpha} + 2\alpha, \underbrace{M[1,1]}_{\alpha} + 2\alpha\} = 3\alpha \\
& M[2,2] = \min\{\underbrace{M[1,1]}_{\alpha} + \underbrace{\alpha}_{E\neq L}, \underbrace{M[1,2]}_{3\alpha} + 2\alpha, \underbrace{M[2,1]}_{2\alpha} + 2\alpha\} = 2\alpha \\
& M[3,2] = \min\{\underbrace{M[2,1]}_{2\alpha} + \underbrace{0}_{L=L}, \underbrace{M[2,2]}_{2\alpha} + 2\alpha, \underbrace{M[3,1]}_{4\alpha} + 2\alpha\} = 2\alpha \\
& M[4,2] = \min\{\underbrace{M[3,1]}_{4\alpha} + \underbrace{0}_{L=L}, \underbrace{M[3,2]}_{2\alpha} + 2\alpha, \underbrace{M[4,1]}_{6\alpha} + 2\alpha\} = 4\alpha \\
& M[5,2] = \min\{\underbrace{M[4,1]}_{6\alpha} + \underbrace{\alpha}_{O\neq L}, \underbrace{M[4,2]}_{4\alpha} + 2\alpha, \underbrace{M[5,1]}_{8\alpha} + 2\alpha\} = 6\alpha \\
j=3: \quad & M[1,3] = \min\{\underbrace{M[0,2]}_{4\alpha} + \underbrace{\alpha}_{H\neq L}, \underbrace{M[0,3]}_{6\alpha} + 2\alpha, \underbrace{M[1,2]}_{3\alpha} + 2\alpha\} = 5\alpha \\
& M[2,3] = \min\{\underbrace{M[1,2]}_{3\alpha} + \underbrace{\alpha}_{E\neq L}, \underbrace{M[1,3]}_{5\alpha} + 2\alpha, \underbrace{M[2,2]}_{2\alpha} + 2\alpha\} = 4\alpha \\
& M[3,3] = \min\{\underbrace{M[2,2]}_{2\alpha} + \underbrace{0}_{L=L}, \underbrace{M[2,3]}_{4\alpha} + 2\alpha, \underbrace{M[3,2]}_{2\alpha} + 2\alpha\} = 2\alpha \\
& M[4,3] = \min\{\underbrace{M[3,2]}_{2\alpha} + \underbrace{0}_{L=L}, \underbrace{M[3,3]}_{2\alpha} + 2\alpha, \underbrace{M[4,2]}_{4\alpha} + 2\alpha\} = 2\alpha \\
& M[5,3] = \min\{\underbrace{M[4,2]}_{4\alpha} + \underbrace{\alpha}_{O\neq L}, \underbrace{M[4,3]}_{2\alpha} + 2\alpha, \underbrace{M[5,2]}_{6\alpha} + 2\alpha\} = 4\alpha \\
j=4: \quad & M[1,4] = \min\{\underbrace{M[0,3]}_{6\alpha} + \underbrace{\alpha}_{H\neq A}, \underbrace{M[0,4]}_{8\alpha} + 2\alpha, \underbrace{M[1,3]}_{5\alpha} + 2\alpha\} = 7\alpha \\
& M[2,4] = \min\{\underbrace{M[1,3]}_{5\alpha} + \underbrace{\alpha}_{E\neq A}, \underbrace{M[1,4]}_{7\alpha} + 2\alpha, \underbrace{M[2,3]}_{4\alpha} + 2\alpha\} = 6\alpha \\
& M[3,4] = \min\{\underbrace{M[2,3]}_{4\alpha} + \underbrace{\alpha}_{L\neq A}, \underbrace{M[2,4]}_{6\alpha} + 2\alpha, \underbrace{M[3,3]}_{2\alpha} + 2\alpha\} = 4\alpha \\
& M[4,4] = \min\{\underbrace{M[3,3]}_{2\alpha} + \underbrace{\alpha}_{L\neq A}, \underbrace{M[3,4]}_{4\alpha} + 2\alpha, \underbrace{M[4,3]}_{2\alpha} + 2\alpha\} = 3\alpha \\
& M[5,4] = \min\{\underbrace{M[4,3]}_{2\alpha} + \underbrace{\alpha}_{O\neq A}, \underbrace{M[4,4]}_{3\alpha} + 2\alpha, \underbrace{M[5,3]}_{4\alpha} + 2\alpha\} = 3\alpha \\
& = \text{ minimal cost for aligning } X \text{ and } Y
\end{aligned}
$$

Traceback to retrieve corresponding optimal alignment:

$$\begin{aligned}
M[5,4] &\rightarrow M[4,3] \text{ and } (x_5, y_4) \text{ aligned}\\
&\rightarrow M[3,2] \text{ and } (x_4, y_3) \text{ aligned}\\
&\rightarrow M[2,1] \text{ and } (x_3, y_2) \text{ aligned}\\
&\rightarrow M[1,0] \text{ and } (x_2, y_1) \text{ aligned}\\
&\rightarrow M[0,0] \text{ and } (x_1, -) \text{ aligned}
\end{aligned}$$

$\Rightarrow$ optimal alignment:

```
HELLO
-ELLA
```

Group Work What happens to the above optimal alignment if we set

$$\alpha_{pq} = \begin{cases} \alpha & p \neq q \\ 0 & p = q \end{cases}$$

and $\delta = \frac{\alpha}{2}$?

Answer A gap is now half as expensive as misaligning two dissimilar characters, i.e. we would expect the optimal alignment to have more gaps than the previously determined optimal alignment above.

Conclusion The optimal alignment depends on α_{pq} and δ.

Time requirements of the algorithm $O(L_x L_y)$

Proof Step (1) and (2) are linear in Lx and Ly.

The recursion is $O(LxLy)$ as the min-operation is $O(1)$, i.e. can be completed in constant time.

The traceback algorithm requires $O(L_x + L_y)$.

(This is also clear from considering the dimension of the M matrix (i.e. the size of the search space and the amount of time required to calculate each matrix element.)