EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE VIII: DYNAMIC PROGRAMMING, PART 2

DISCLAIMER

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos and ***Analysis of Algorithms*** 2nd edition, by Jeffery McConnell.

OUTLINE

- Three more examples:
  - interval scheduling revisited
  - segmented linear regression
  - shortest path revisited (Bellman-Ford algorithm)

INTERVAL SCHEDULING REVISITED

- Recall the interval scheduling problem (that we have discussed while learning greedy algorithms): Given a single-threaded CPU server, and a set of jobs J that need to be scheduled. Each job $j_i \in J$ has an associated start time s_i and finishing time f_i. Find the maximal subset $J' \subseteq J$ such that for any two jobs $j_i, j_j \in J'$ either $f_i \leq s_j$ or $f_j \leq s_i$.

- In layman terms: Two jobs cannot be executed at the same time since we only have a single-threaded server. Schedule as many jobs as possible.

INTERVAL SCHEDULING REVISITED

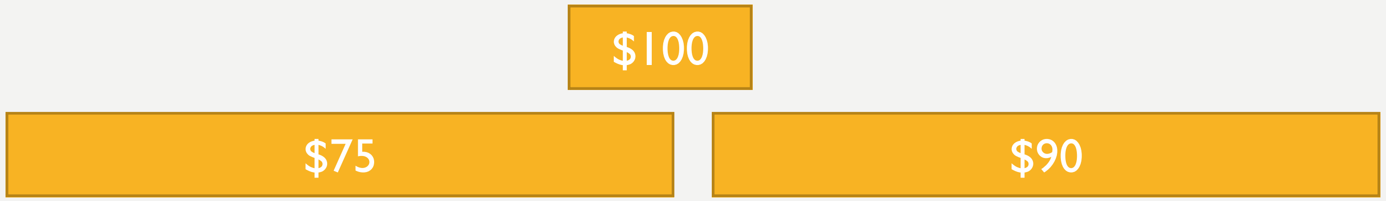
- In a more practical case, the users spend money to bid the server time.
  - in this case, each job j_i is further associated with a weight w_i (you can interpret it as the money the corresponding user is willing to pay to run it), in addition to the start time s_i and finishing time f_i
  - our objective is then changed into find the conflict-free subset $J' \subseteq J$ such that $\sum_{j_i \in J'} w_i$ is maximized (that is, earn as much money as possible)
  - this is also called **<u>weighted interval scheduling</u>**

INTERVAL SCHEDULING REVISITED

- First, let’s see if we can still solve the problem using the original greedy algorithm (which puts the earliest-finishing job first):
  - the original algorithm finds the maximum number of jobs we can execute
  - imagine that there is a job that requests the server for all time, but it pays more than all other jobs combined
  - obviously we should pick this job under the weighted interval scheduling formulation, but the original greedy algorithm will not pick it

INTERVAL SCHEDULING REVISITED

- Second, let's try a different greedy algorithm (which puts the most "valuable" job first, or most "valuable per requested time" job first)
  - it does not seem to work for the following example



INTERVAL SCHEDULING REVISITED

- Hence, we would need a different algorithm to solve the problem. (And you know it should be dynamic programming since it is the topic of this module.)

- Let's try to define a proper subproblem.
  - there are basically two input restrictions that limit us from obtaining infinity profit: a limited amount of time and a limited set of jobs
  - and we should be defining the subproblem via reducing these two factors:
    - an instance with a shorter time
    - an instance with a subset of jobs
    - an instance with both a shorter time and a subset of jobs

INTERVAL SCHEDULING REVISITED

- Let's first investigate the subproblem defined by subsets of jobs
  - this strategy is unlikely to work
  - because the solution we found for a subset of jobs could become invalidate if some conflicting new jobs are added in
  - this also explains why the subproblem defined on both a shorter time and a subset of jobs does not work

INTERVAL SCHEDULING REVISITED

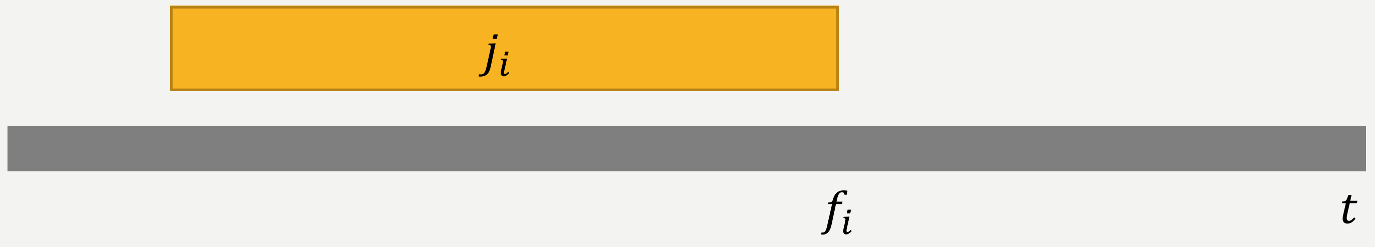
- We only choice remains is to use a shorter time as the subproblem
  - we still need to investigate whether it has an optimal substructure
  - let's denote $V(t')$ as the maximum profit we can obtain up to time t'.
  - the solution to the subproblem is obviously invariant, since for a given set of jobs and a given amount of time, the maximum profit one can make is fixed

INTERVAL SCHEDULING REVISITED

- Can we construct the solution for a larger problem using the solutions for the subproblems?
  - well, if we know the solution of $V(t')$, it can clearly be used to construct the solution for $V(t)$ where $t > t'$
  - note that the time cutoff (t or t') implicitly defines a subset of jobs that ends before it, because we will surely not be able to account for jobs that ends after the time cutoff
  - so, the jobs defined by t' is a subset of the jobs defined by t (or $J^{t\prime} \subseteq J^t$)
  - also, note that what defines $V(t)$ is no more than a combination of J^t, and $V(t')$ offers the best combination for $J^{t\prime}$; and since $J^{t\prime} \subseteq J^t$, $V(t')$ considers combinations on a subset of the elements, and thus provides information to construct the solution of the larger problem
  - this only a conceptual argument, and we will see some real examples later

INTERVAL SCHEDULING REVISITED

- A final restriction we should be placing on the subproblem is that we expect the time cutoff to be the finishing time of some jobs.
  - this is because no job can be scheduled between the finishing time of the last job and the time cutoff to increase profit
  - we may just shrink the time cutoff to the finishing time
  - and it discretize time and lead to only a linear number of subproblems (w.r.t the jobs)



no job can be scheduled

INTERVAL SCHEDULING REVISITED

- One key observation:
  - if $t > t'$, then $V(t) \geq V(t')$
  - this is because for a given set of jobs, the more time we have, the more (or at least not less) profit we can make

INTERVAL SCHEDULING REVISITED

- Next, we shall see how can we construct solutions for larger problems using solutions for smaller subproblems:
  - we will be viewing this problem in a "backward" way, which is consistent with the writing of recursive functions
- For a given problem $V(t)$, we know from the previous lemma that $t = f_i$ for some job j_i. To define the set of subproblems, we should pre-sort the jobs based on their finishing time. The time needed for this step is obviously $O(n \log n)$ for n jobs.
  - we will break the ties by sorting the job with an earlier starting time ahead
  - and if two jobs have identical start and end time, we will pick the one with a higher weight

INTERVAL SCHEDULING REVISITED

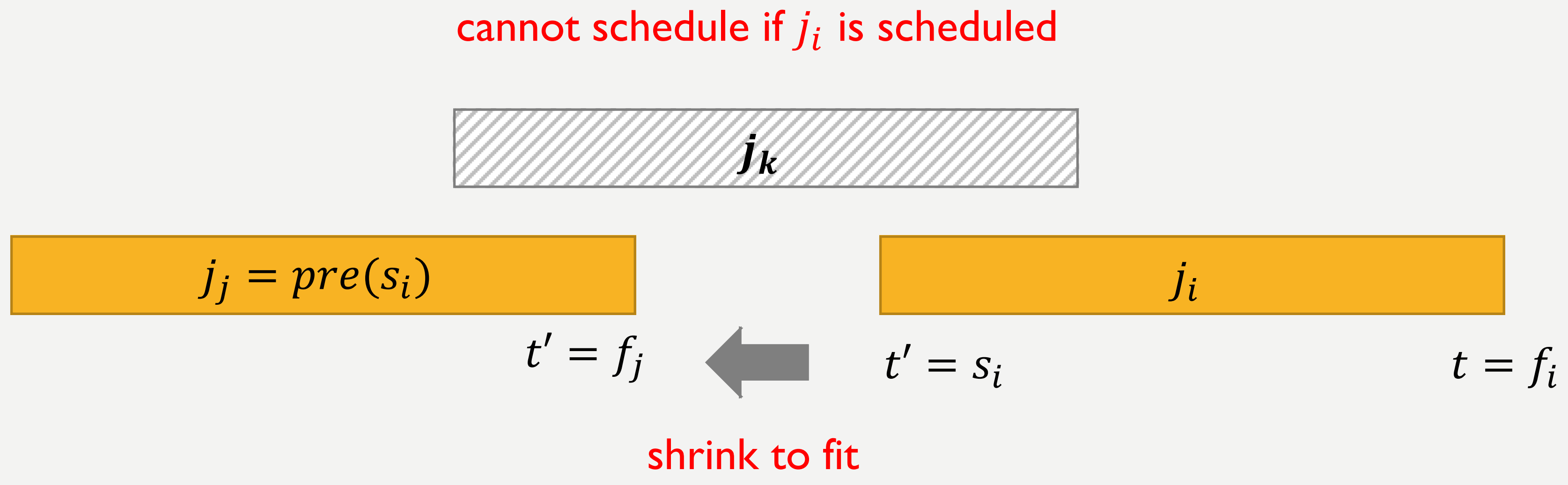
- Now, consider a larger problem $V(t)$, and its associated job j_i. There will be two cases we need to discuss:
  - either that j_i is scheduled
  - or that j_i is not scheduled

INTERVAL SCHEDULING REVISITED

- Case 1: If j_i is scheduled, we are sure that no more job can be scheduled after j_i have started. So, the subproblem we should be using is how much profit can we get using the time before j_i is started. In other words, the subproblem $V(s_i)$.

- While s_i does not necessarily correspond to the finishing time for another job j_j, we can shrink (per the previous proven lemma) the time s_i to the latest time point t' such that $t' = j_j$.

INTERVAL SCHEDULING REVISITED



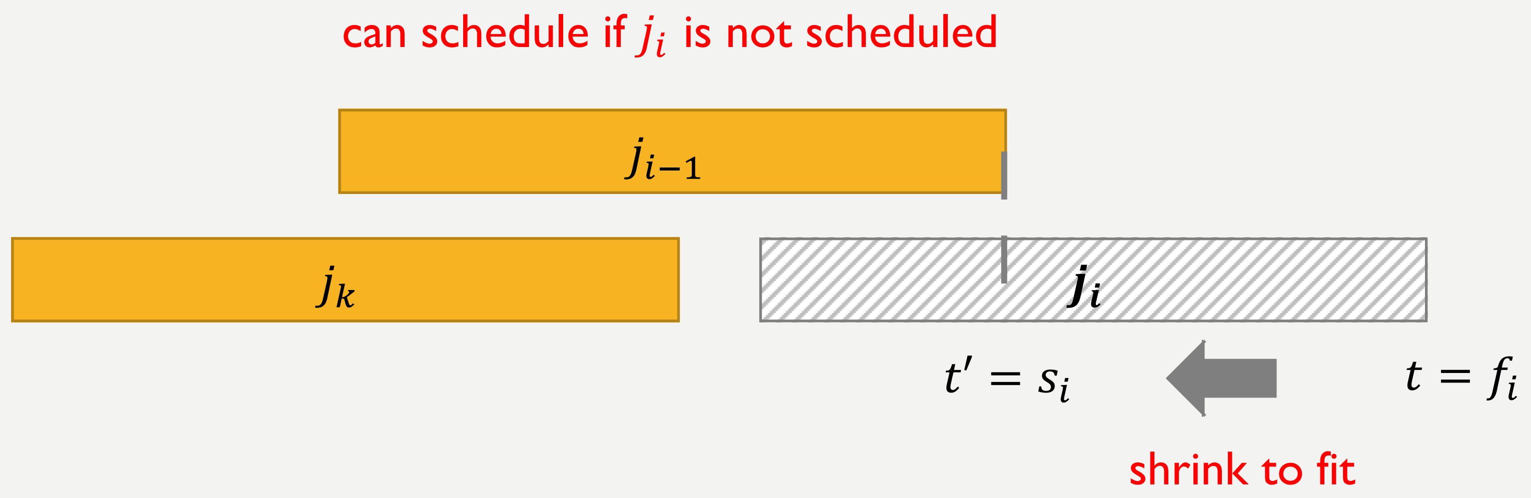
INTERVAL SCHEDULING REVISITED

- If we define the last job whose ending time is before a given time point t as $pre(t)$, then we can simply write down the recursive function for this scenario (when j_i is scheduled) as:

$$V(f_i) = V\left(f_{pre(s_i)}\right) + w_i$$

INTERVAL SCHEDULING REVISITED

- Case 2: If j_i is not scheduled, then it will not invalidate any jobs due to time conflict, and we can simply jump to its immediately previous job.



INTERVAL SCHEDULING REVISITED

- Then, it is easy to write the recursive function for the second scenario, where j_i is not scheduled:

$$V(f_i) = V(f_{i-1})$$

- Taken together, we have the complete recursive functions as:

$$V(f_i) = \max \begin{cases} V\left(f_{pre(s_i)}\right) + w_i \\ V(f_{i-1}) \end{cases}$$

INTERVAL SCHEDULING REVISITED

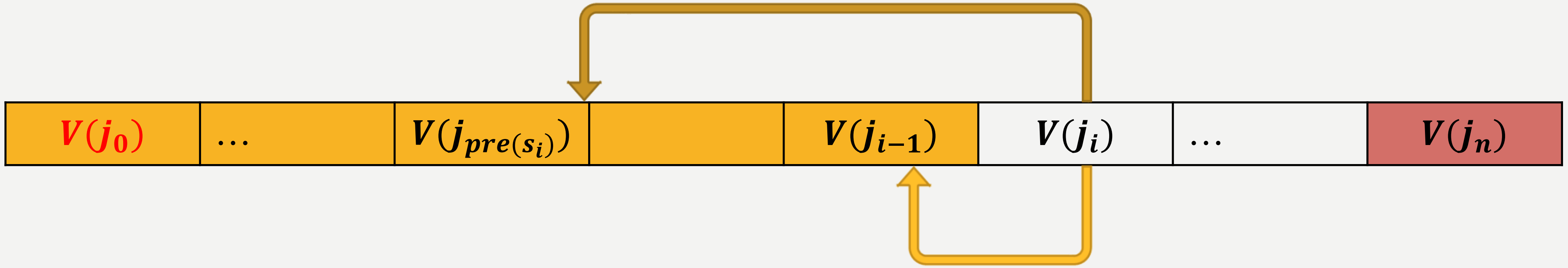
- A note on the implementation of the function $pre(.)$:
  - we can obtain a list of sorted finishing time (note that we need to sort the jobs based on their finishing time anyway)
  - then, for a given time t, we will just perform binary search of t against the list to find the latest job whose finishing time is before t (needs $O(n \log n)$ time)

INTERVAL SCHEDULING REVISITED

- Now that we have the recursive function, we should be able to set up the dynamic programming table and the detailed algorithm.

- It is easy to see that the dynamic programming table we need is one-dimensional, since we only have one variable (i.e., time) when defining subproblems
  - and to solve problems with a later time, we would refer to the results of problems with earlier time
  - so we should be filling up the table simply from left to right

INTERVAL SCHEDULING REVISITED



Note: j_0 is a dummy job such that $j_0 = pre(s_i) = pre(f_i)$ for all jobs j_i where no job ends before it and $w_0 = 0$. In this case, all components of the recursive functions are well-defined. This is also the only initialization we need to do.

INTERVAL SCHEDULING REVISITED

- Traceback: we simply need to record "Y" or "N" for each cell. "Y" indicates that the maximum profit comes from the first scenario where the corresponding job is taken, and "N" indicates the other scenario where it is not taken. (of course, you also need to remember the pointer to the proper subproblem.)

- We will start from the very last cell, since it define the optimal results for the entire problem. During the traceback, we will schedule tasks labeled with "Y".
  - the process clearly takes $O(n)$ time

INTERVAL SCHEDULING REVISITED

- A toy-example:

$s_2 = 2, f_2 = 15, w_2 = \$70$

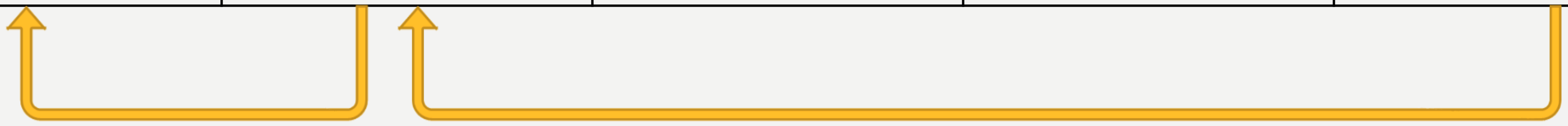
$s_1 = 0, f = 10, w_1 = \$50$

$s_4 = 12, s_1 = 25, w_4 = \$45$

$s_3 = 8, f_3 = 18, w_3 = \$85$

red color indicates the better solution
the first number indicates case 1, the second number indicates case 2

\$0	(\$0+\$50, \$0) (Y)	(\$0+\$70, \$50) (Y)	(\$0+\$85,\$70) (Y)	(\$50+\$45,\$85) (Y)



the arrows indicate traceback path; the maximum profit is \$95 by taking jobs 1 and 4.

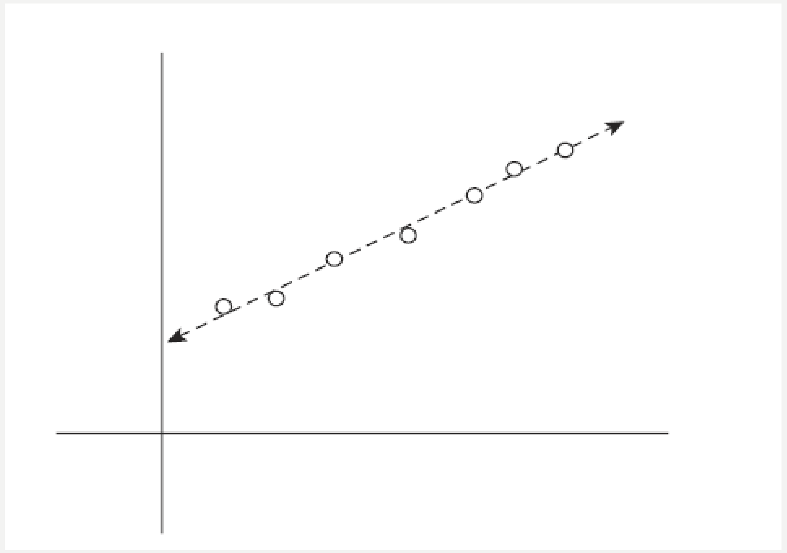
INTERVAL SCHEDULING REVISITED

- Time complexity analysis:
  - we have a one-dimensional table to fill, which has a size of $O(n)$
  - to compute each cell, we need $O(\log n)$ time (using binary search to compute the function $pre(.)$)
  - taken together, the overall time complexity is $O(n \log n)$
  - this is also the same as the time complexity required for pre-sorting the jobs (which serves as a lower bound for any algorithm)

- Space complexity is clearly $O(n)$.

SEGMENTED LINEAR REGRESSION

- Recall (or consider it as a short introduction) the single-variable linear regression problem: given a set P with n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ ordered by their increasing x-coordinates, find two parameters a (**slope**) and b (**intersection**), such that $Res(P, a, b) = \sum_{i=1}^{n}(y_i - ax_i - b)^2$ is minimized (i.e., minimizing the **residual**).



SEGMENTED LINEAR REGRESSION

- To find the slope and the residual, we can take the derivative of $Res(P, a, b) = \sum_{i=1}^{n}(y_i - ax_i - b)^2$ w.r.t a and b, respectively, and set the derivatives to 0. (You can also take the second derivative to verify that they correspond to the minimum-value case.) And we get:

$$a = \frac{n\sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n\sum_i x_i^2 - (\sum_i x_i)^2}$$

$$b = \frac{\sum_i y_i - a\sum_i x_i}{n}$$

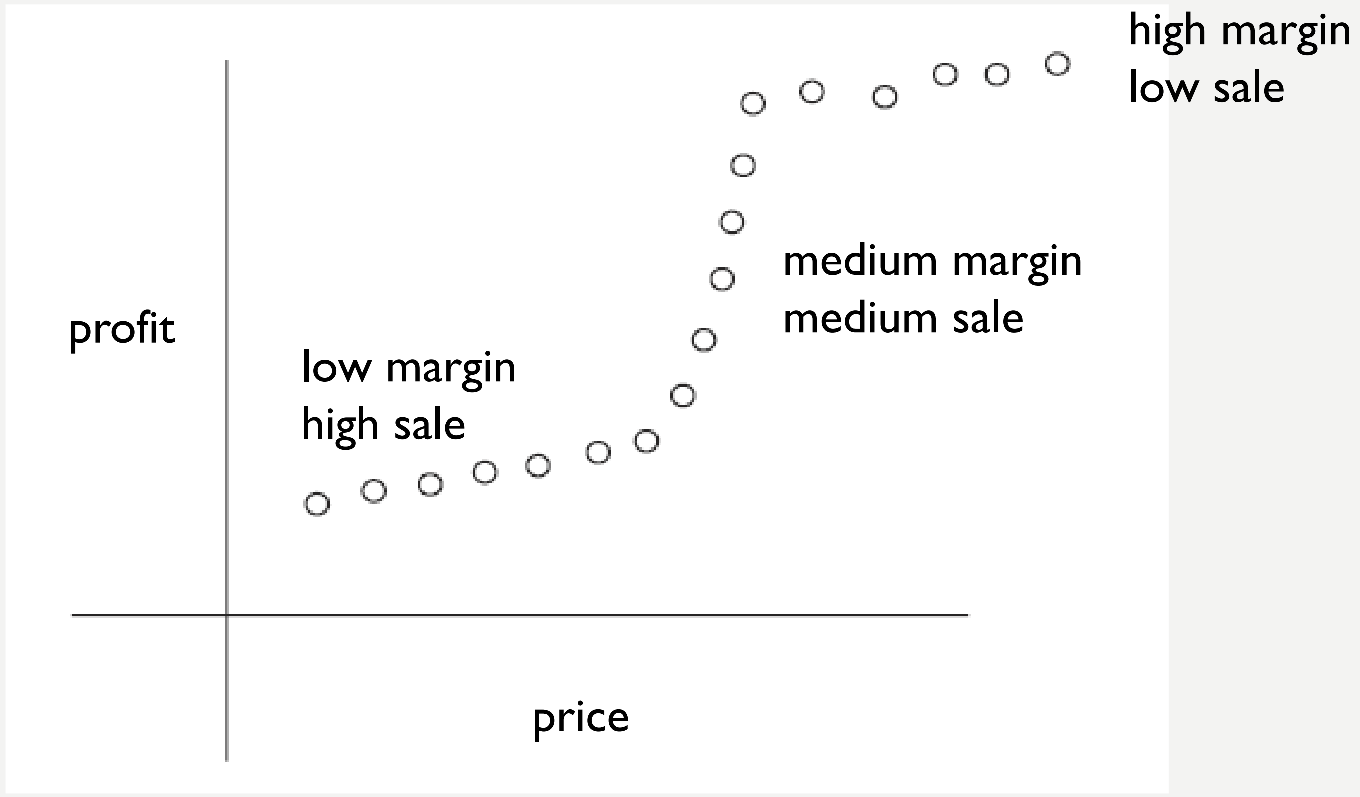
SEGMENTED LINEAR REGRESSION

- While you do not need to memorize these formulae (of course it is better that you do), you would like to observe two important facts that would help the study of the segmented linear regression problem:
  - it takes $O(n)$ time to compute a and b
  - the parameters a and b are functions of the set of input points

- Once you have estimated a and b, you can predict the corresponding y (called **dependent variable**) value of a previously unseen **independent variable** $x_u \notin P$ (a slight misuse of the notation here).

SEGMENTED LINEAR REGRESSION

- However, in real cases, it is possible that our data points are governed by more than one linear relation.



SEGMENTED LINEAR REGRESSION

- Note that although we can see how many linear relations the data has with our naked eyes, but it is more challenging for computer to see it. Hence, we must formulate a proper computational problem.

- If we were allowed to fit the data with $n-1$ lines, then we shall be able to obtain 0 residual
  - but it does not make any real sense (and in machine learning it is called **overfitting**)
  - one way to resolve it is to associate some penalty with each added line, such that unless adding a line can explain a significant amount of data that overturns the penalty, we are better off with fewer lines (in machine learning it is called **regularization**)

SEGMENTED LINEAR REGRESSION

- Now, we can formulate the segmented linear regression problem: given a set P with n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ ordered by their increasing x-coordinates, partition the set into k subsets $P_1, P_2, \ldots, P_k$, where for all points within each subset have continuous rankings, and find the corresponding k sets of parameters $(a_1, b_1), (a_1, b_1), \ldots, (a_k, b_k)$ to minimize

$$R = \sum_k Res(P_k, a_k, b_k) - (k-1)C$$

Here, C is the penalty associated with each added fitting line. Since a_k, b_k are fixed given P_k, we will simplify $Res(P_k, a_k, b_k)$ as $Res(P_k)$.

- Note that we are required to find k and the specific partition strategy simultaneously.

SEGMENTED LINEAR REGRESSION

- Since the problem has only one type of input (i.e., the points), and their order is determined (sorted by their x-coordinates), we only have one way to define subproblems. That is, we define subproblem as the optimal segmented fitting on a subset of points.

- Formally, define $R(i)$ as the minimum residual (with regularization penalty) obtained by fitting the first i data points. And we shall figure out a way to construct the solution of $R(i)$ using $R(i')$ for some $i' < i$.

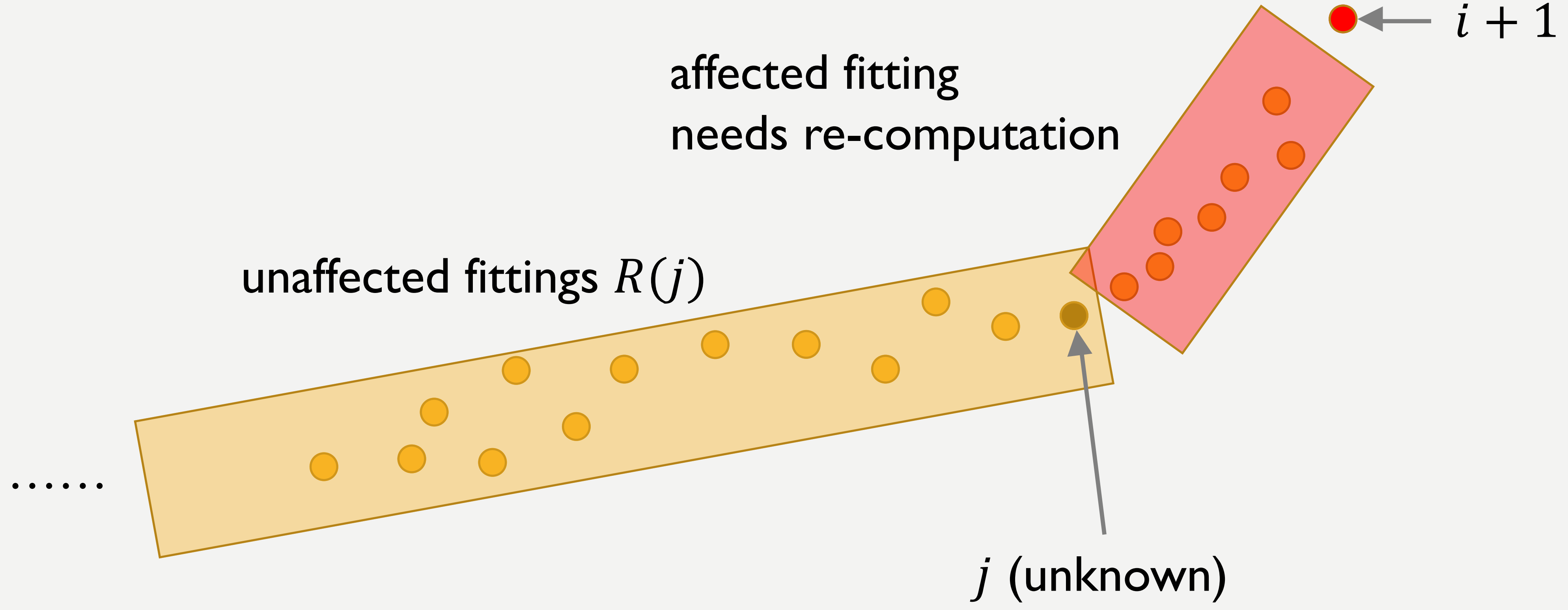
SEGMENTED LINEAR REGRESSION

- First, we need to check whether the subproblem definition satisfies optimal substructure
  - given the index i, the set of points that are considered for the regression is fixed
  - so, the minimum residual will also be fixed
  - hence the subproblem definition satisfies optimal substructure

SEGMENTED LINEAR REGRESSION

- However, when we add a new point, say $i + 1$, the input data points are changed and we will need to re-perform fitting on part of the data
  - we will re-perform fitting only on the segment that is affected by the addition of data point $i + 1$
  - since the points in each partition have continuous ranking, we know that in the worst-case scenario, the point $i + 1$ will only affect the last segment
  - it is also possible that $i + 1$ does not affect any segment

SEGMENTED LINEAR REGRESSION



SEGMENTED LINEAR REGRESSION

- Case 1: the inclusion of $i + 1$ affects the fitting of the last segment
  - we shall re-compute the fitting between the first point of the last segment, j, to the newly added data point $i + 1$
  - however, we do not know which point could be the the first point of the last segment, so we have to try all possible points $1 \dots i$
  - and we end up with the recursive function (note that we have shifted the index by 1 to simplify the recursion)

$$R(i) = \max_{1 \le j \le i-1} (R(j) + Res(j \dots i))$$

  - note that we did not include a new segment so we do not need to add the penalty

SEGMENTED LINEAR REGRESSION

- Case 2: if $i+1$ is itself the first point of a segment (such that it does not affect the fitting of its previous points)
  - then we do not need to perform re-fitting
  - but we would need to add the penalty
  - and we will have the recursive function for this case (also with index shifted by 1)

$$R(i) = R(i-1) + C$$

SEGMENTED LINEAR REGRESSION

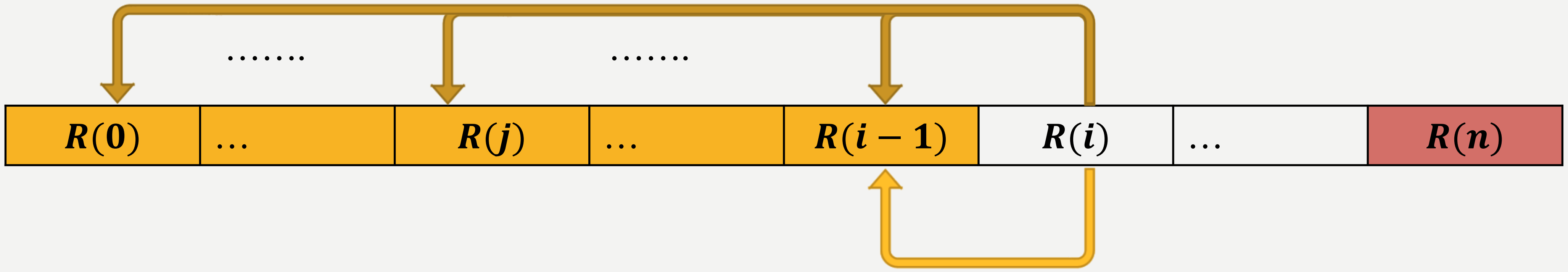
- Taken together, we have the final recursive function:

$$R(i) = \max \begin{cases} \max\limits_{1 \le j \le i-1} (R(j) + Res(j \ldots i)) \\ R(i) = R(i-1) + C \end{cases}$$

SEGMENTED LINEAR REGRESSION

- Details of the implementation of the algorithm:
  - we first note that the recursive function has only one index, i.e., the index of the data point, so using a one-dimensional dynamic programming table is sufficient
  - because computing problems defined with larger indexes requires results for subproblems with smaller indexes, we should be filling up the array from left to right
  - for the boundary case, we define $R(1) = 0$
    - because we can always fit a single point perfectly (i.e., with a residual of 0)
    - no penalty is needed with only one line

SEGMENTED LINEAR REGRESSION



$$R(i) = \max\begin{cases} \max\limits_{1\le j\le i-1}(R(j) + Res(j \dots i)) \\ R(i) = R(i-1) + C \end{cases}$$

SEGMENTED LINEAR REGRESSION

- For the traceback:
  - we need to store two types of information
    - the pointer to the subproblem that leads to the optimal solution
    - whether the current point corresponds to the start of a segment
  - note that both cases could use $R(i-1)$ to generate the optimal result, but they have different meanings
    - in the first case it means that the two points $i-1$ and i are in the last segment
    - in the second case it means that only the point i is in the last segment
  - the traceback takes $O(n)$ time

SEGMENTED LINEAR REGRESSION

- Time complexity analysis:
  - we need to fill up a one-dimensional table with $O(n)$ cells
  - to fill up each cell, we need to consider $O(n)$ subproblems (for case 1)
  - for each subproblem, we further need $O(n)$ time to re-compute the fitting
  - the overall time complexity is then $O(n^3)$

- Space complexity: $O(n)$

SHORTEST PATH REVISITED

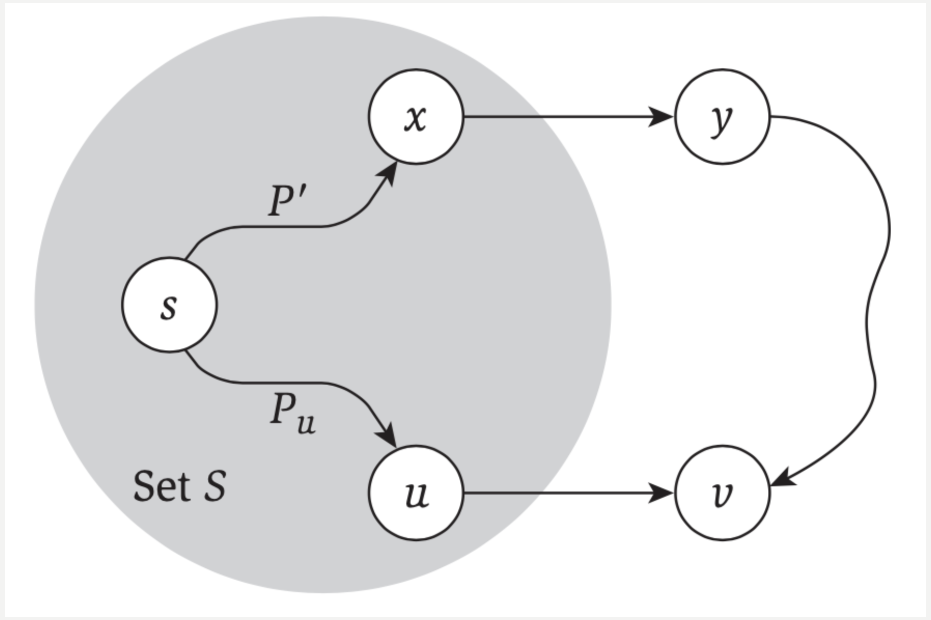
- Recall the shortest path problem: Given a graph $G = \{V, E\}$, and the cost/length for each edge (denote the length of edge e as l_e and $l_e \geq 0$ for all $e \in E$), find the shortest path from vertex u or v whose total length is minimized among all paths from u to v.

- We have presented the Dijkstra's algorithm, which was designed under the greedy schema, to solve the problem:
  - recall that we continuously select the vertex v the extend the current path, where
    $$\underset{v,e=(u,v):u\in S}{\operatorname{argmin}} \ (d(u) + l_e)$$

SHORTEST PATH REVISITED

- However, a key assumption to use Dijkstra's algorithm is that all edges are non-negative.
- If there exists some negative edges, the Dijkstra's algorithm will not work.

SHORTEST PATH REVISITED



- If we allow negative edges or paths (note that we still do not allow negative cycle), we can simply make up a **counter-example** to disprove the claim
  - let d_y be 10 and d_v be 7
  - let the path $P_{y,v}$ have a length of -5
  - the shortest path between s and v actually goes through y (with a total length of 5), but not going through u; as a result, d_v should be 5 instead of 7

SHORTEST PATH REVISITED

- Hence, it is necessary to design an algorithm that would work even with negative edges present.

- We present the Bellman-Ford algorithm, which was designed under the dynamic programming principal
  - in fact, Richard Bellman is considered as the key inventor of dynamic programming

SHORTEST PATH REVISITED

- The limitation of the Dijkstra’s greedy algorithm:
  - there is only one way to extend the path; that is, we can only extend the path towards v
  - other extensions are not recorded, even though they may be connecting future negative edges that actually bring down the overall path length
- To address the limitation of the greedy algorithm
  - we need to record multiple path extensions
  - something dynamic programming is good at (memorization)

SHORTEST PATH REVISITED

- First, we need to define the subproblems:
  - the input to the problem is the graph, and the source s and the target t (in the all-pairs shortest path setting, we can try all possible s)
  - the Dijkstra's algorithm partitions the graph into two subsets, the set of visited nodes S and the set of unvisited nodes $V - S$
  - an intuitive way to define subproblems is to adopt the the same way as the Dijkstra's algorithm
    - we will define the set of visited nodes as the subproblem
    - conceptually, define $D(S)$ and the shortest paths identified for all vertices that have been visited

SHORTEST PATH REVISITED

- Defining the subproblem (cont.)
  - however, we immediately find that there are two issues related with such a definition
  - first, it is difficult to compute a set of paths (from the source s to each node in S)
    - we can define $D(u)$ for each $u \in S$
  - second, we don't know how the paths are extended (this is because we have to keep track of all path extensions in dynamic programming, but in the greedy algorithm we only have a single way for path extension)
    - we need an additional index to characterize "visited" vs. "unvisited"
    - we will use the number of edges involved between s and u (less numbers of edges involved are the subproblems for larger numbers of edges involved)

SHORTEST PATH REVISITED

- Define $D(i,u)$ as the shortest path between the source s and the node u where less than or equal to i edges involved.

- Observation:
  - $D(i,u) \leq D(i',u)$ for $i > i'$
  - that is, when more edges are allowed to involve, we can at least maintain the existing optimal solution and not make things worse

SHORTEST PATH REVISITED

- Does the subproblem definition satisfy the optimal substructure?
  - the problem defined in $D(i, u)$, can be solved by referring to all $D(i-1, v)$ (for a set of vs that are immediately before u)
  - we may also explore the possibility that the additional edge was not take, in which case the result is directly available as $D(i-1, u)$
  - hence, the subproblem satisfies the optimal substructure

SHORTEST PATH REVISITED

- With the above analysis, we can easily write down the recursive functions:

$$D(i,u) = \min \begin{cases} \min\limits_{v;(v,u)\in E} \{D(i-1,v) + l_{(v,u)}\} \\ D(i-1,u) \end{cases}$$

SHORTEST PATH REVISITED

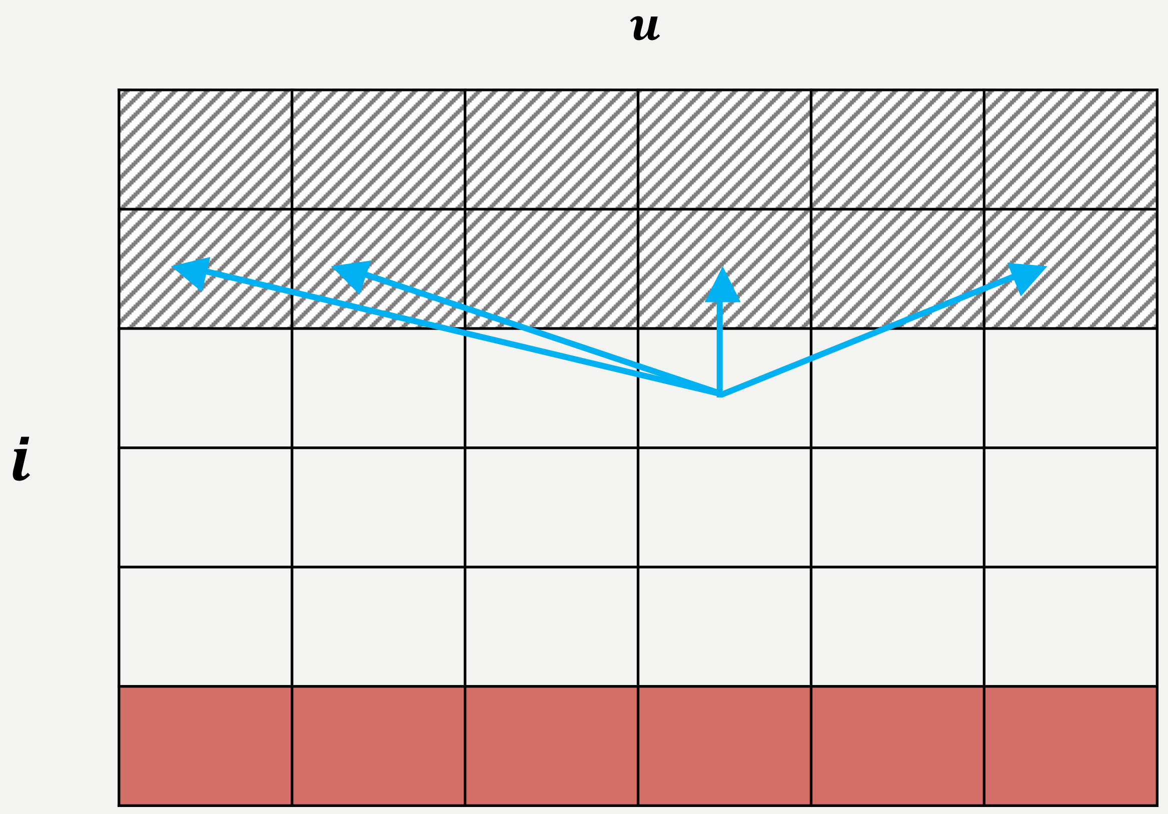
- With the recursive function, we will design the detailed algorithm:
  - we have two indices i and u, so we will be working on a two-dimensional dynamic programming table
  - while i has a clear order, u does not; that is, there is no clear order that allows us to prioritize any node before the others. So, we do not care the order of the nodes
  - in this case, one dimension of the table corresponds to i with an ordering from 1 to $|V|$ (the maximum number of edges a path can involve, since we assumed no negative cycle), and the other dimension corresponds to the set of nodes (without any specific order)

SHORTEST PATH REVISITED

- Since the set of nodes are unsorted, it is possible that we will need to refer to any column index when performing the algorithm, depending on the graph topology.

- It is therefore obvious that we should be filling up the table row-by-row.

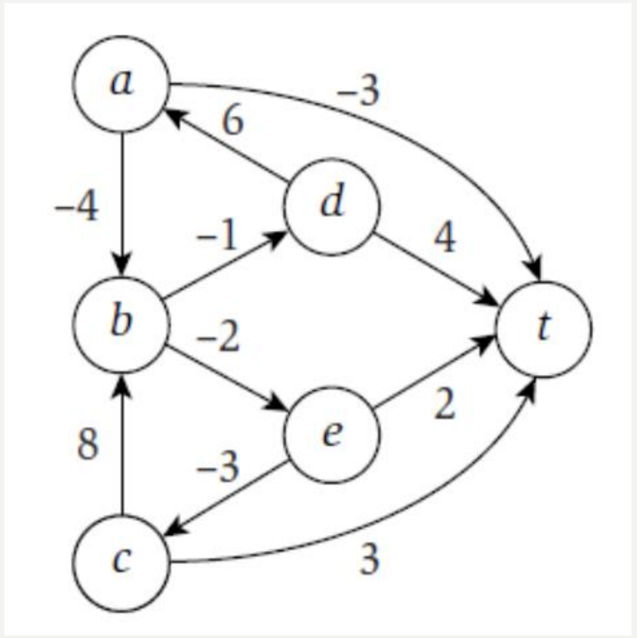
- The results locate in the last row.



SHORTEST PATH REVISITED

- Initialization:
  - given the table filling order, it is necessary that we initialize the values in the first row, which corresponds the values $D(1, u)$ for all u
  - it is essentially asking, what is the shortest path from s to u if you are allowed to use only one edge?
  - intuitively, if an edge exists between s and u, the path length will simply be the edge weight; otherwise, it will be infinity
  - note that we will add an edge that goes from s to itself with a cost of 0

SHORTEST PATH REVISITED



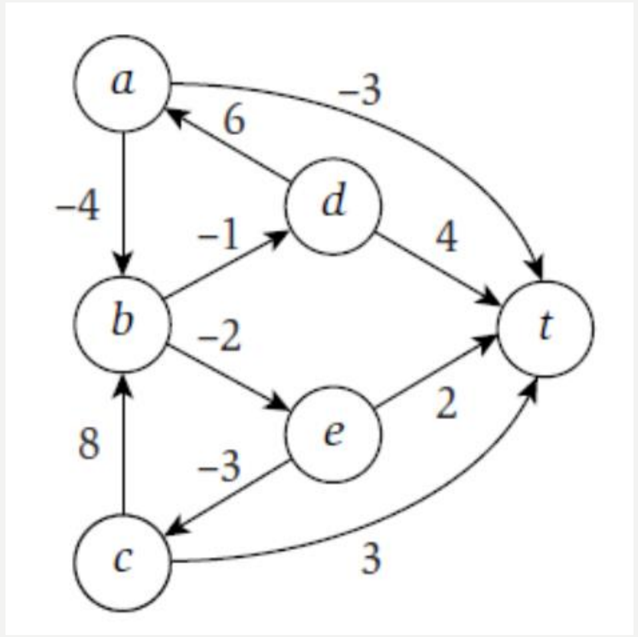
assuming the source to be a

	a	b	c	d	e	t
1	0	-4	∞	∞	∞	-3
2	0	-4	∞	-5	-6	-3
3	0	-4	-9	-5	-6	-4
4	0	-4	-9	-5	-6	-6
5	0	-4	-9	-5	-6	-6

Terminate after convergence

SHORTEST PATH REVISITED

- For the traceback, we simply record the previous node of the path that generates the optimal result.



	a	b	c	d	e	t
1	0	-4	∞	∞	∞	-3
2	0	-4	∞	-5	-6	-3
3	0	-4	-9	-5	-6	-4
4	0	-4	-9	-5	-6	-6
5	0	-4	-9	-5	-6	-6

Terminate after convergence

SHORTEST PATH REVISITED

- Time complexity analysis:
  - we have a two-dimensional table to fill, which has a size of $O(|V|^2)$
  - to compute each entry, we will need to try all of its adjacent nodes, which is $O(d)$, where d is the average degree of the graph
  - taken together, the overall time complexity is $O(|V|^2 d) = O(|V| * |V| * d) = O(|V||E|)$

- Space complexity analysis:
  - $O(|V|^2)$ because of the table

SUMMARY

- Two more examples:
  - interval scheduling revisited
  - segmented linear regression
  - shortest path revisited