Weighted Activity Selection

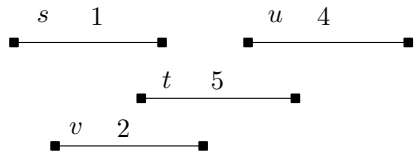
Problem

This problem is a generalization of the activity selection problem that we solvd with a greedy algorithm. Given a set of activities $A = \{[l_1, r_1], [l_2, r_2], \ldots, [l_n, r_n]\}$ and a positive weight function $w : A \to \mathbb{R}^+$, find a subset $S \subseteq A$ of the activities such that

- $s \cap t = \emptyset$, for $s, t \in S$, and
- $\sum_{s \in S} w(s)$ is maximized.

We will solve this problem iteratively, refining our algorithm at each stage to get a more efficient solution.

Example:



Greedy algorithm for unweighted activity selection chooses $\{s, u\}$. Chosing the greatest weight first selects $\{t\}$. The optimal solution is $\{u, v\}$.

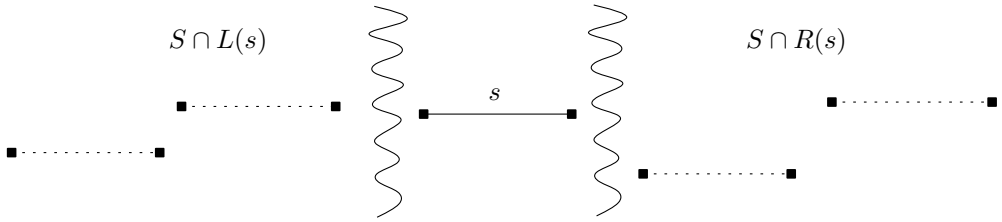
Optimal Substructure:

The key to creating dynamic programming algorithms is observing some optimal substructure: How part of an optimal solution (substructure) is an *optimal* solution to a subproblem. We did this with the unweighted activity selection problem. Recall our proof of correctness of our greedy activity selection: We showed that the choices made after the first selection were an optimal solution to a subproblem — the set of activities that did not conflict with with the first choice.

Our identification of the optimal substructure will be difference because we do not have a greedy strategy to choose a first activity. We notice that if the set of activities A is non-empty, an optimal solution $S \subseteq A$ must contain some activity because the weight function w is positive. So consider some $s \in S$ (we do not care which one). Then all of the activities in $S \setminus \{s\}$ are either left of s or right of s because activities of S do not overlap. Let

$$\begin{aligned} L(s) &= \{a \in A : a \cap s = \emptyset \text{ and } a \text{ is left of } s\} \\ R(s) &= \{a \in A : a \cap s = \emptyset \text{ and } a \text{ is right of } s\} \end{aligned}$$

Then $S \cap L(s)$, $S \cap R(s)$, and $\{s\}$ partition S.



This problem is similar to the unweighted variant. So we guess that $S \cap L(s)$ is an optimal solution the weighted activity selection problem restricted to $L(s)$. Is this true? Yes! To see why, consider an optimal selecton T of $L(s)$. Recall, that this means that

- $t \cap u = \emptyset$, for $t, u \in T$, and
- $\sum_{t \in T} w(t)$ is maximum — in particular, $\sum_{t \in T} w(t) \geq \sum_{t \in S \cap L(s)} w(t)$.

To prove that $S \cap L(s)$ is an optimal solution to the subprobem, we substitute T for $S \cap L(s)$ in S. Then the optimality of S will force the above inequality to be an equality. First, we must show that this substitution is valid — that is, $t \cap u = \emptyset$, for $t, u \in (S \setminus L(s)) \cup T$. By construction, any two activities from $S \setminus L(s)$ do not overlap and any two activities from T do not overlap. So the last case to consider is when $u \in S \setminus L(s)$ and $t \in T$. When this happens, t is left of u because t is left of s and s is the leftmost element of $S \setminus L(s)$.

Let us look at the weight of the selections of A.

$$\begin{aligned}
\sum_{t \in S} w(t) &= \sum_{t \in S \setminus L(s)} w(t) + \sum_{t \in S \cap L(s)} w(t) \\
&\leq \sum_{t \in S \setminus L(s)} w(t) + \sum_{t \in T} w(t) \text{ by optimality of } T \\
&= \sum_{t \in (S \setminus L(s)) \cup T} w(t)
\end{aligned}$$

But, $\sum_{t \in S} w(t) \geq \sum_{t \in (S \setminus L(s)) \cup T} w(t)$ by optimality of S. Therefore, $\sum_{t \in S} w(t) = \sum_{t \in (S \setminus L(s)) \cup T} w(t)$ and $\sum_{t \in T} w(t) = \sum_{t \in S \cap L(s)} w(t)$. Thus, $S \cap L(s)$ is an optimal selection for $L(s)$.

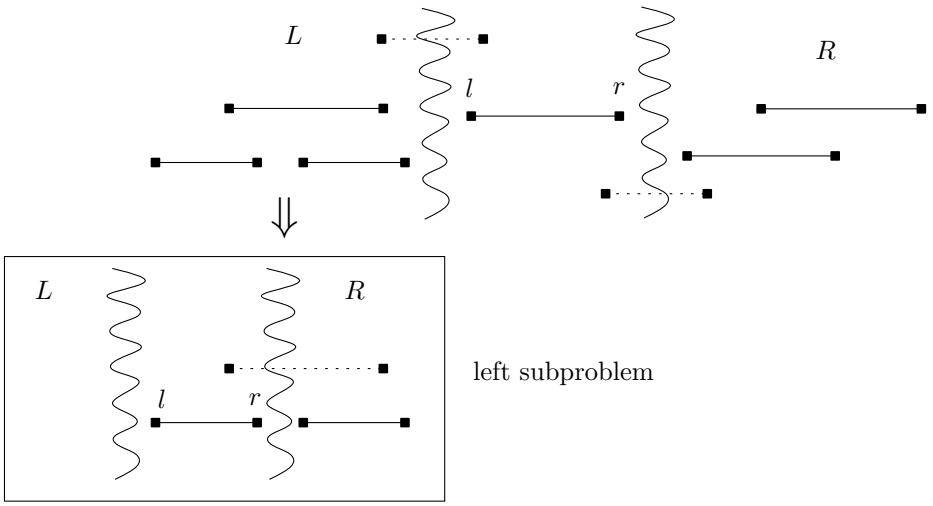
Notice that we have just proven that we can substitute any optimal solution T to $L(s)$ for $S \cap L(s)$ in S because $\sum_{t \in T} w(t) = \sum_{t \in S \cap L(s)} w(t)$! This suggests a divide-and-conquer algorithm for solving this problem:

- Choose an s;
- recursively solve for $L(s)$ and $R(s)$;
- combine them with s to see if we get an optimal solution.

We do not know which s to pick, so we try them all!

```
Algorithm DCMaxSelect(A)
    S ← ∅
    max ← ∞

    for each [l, r] ∈ A do
        L ← {[p, q] ∈ A : q < l}
        R ← {[p, q] ∈ A : r < p}

        T ←DCMaxSelect(L)
        U ←DCMaxSelect(R)

        δ = w([l, r]) + Σ_{s∈T} w(s) + Σ_{s∈U} w(s)
        if max < δ then
            S ← T ∪ U ∪ {[l, r]}
            max ← δ

        return S
```

We use caching to turn this into a dynamic programming solution. The input to `DCMaxSelect` is a subset of A, which has n elements. Recall that there 2^n possible subsets. So just initializing a table large enough to be indexed by all possible inputs would take exponential time. Fortunately, not all 2^n subsets of A occur in recursive calls to `DCMaxSelect`.



left subproblem

We can prove inductively (on the depth of recursion) that the input to every recursive call is a subset of the form

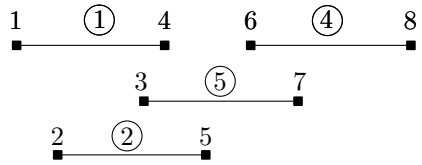
$$\{[l, r] \in A : \alpha < l \text{ and } r < \omega\}$$

for some α and ω. In fact, α will be $-\infty$ or l_i and ω will be ∞ or r_j, for some i and j. So there are only $O\left(n^2\right)$ such inputs. But we cannot index a table (cache) with continuous α and ω. However, α and ω only take on a finite set of values. Let

$$\begin{aligned} X &= \{l_1, l_2, \ldots, l_n\} \cup \{r_1, r_2, \ldots, r_n\} \cup \{-\infty, \infty\} \\ &= \{x_1, x_2 \ldots, x_k\}, \text{ where } k \leq 2n + 2 \text{ and } x_i < x_j, \text{ for } i < j \end{aligned}$$

Then we represent α and ω with an index from $1 \ldots k$. Let max$[i, j]$ =the maximum weight activity selection of all of the activities of A contained in the interval (x_i, x_j), where $i < j$.

Example:



x_i \ x_j

	$-\infty$	1	2	3	4	5	6	7	8	∞
$-\infty$	-	0	0	0	0	1	2			
1	-	-	0	0	0	0	2	2		
2	-	-	-	0	0	0	0	0	5	
3	-	-	-	-	0	0	0	0	0	4
4	-	-	-	-	-	0	0	0	0	4
5	-	-	-	-	-	-	0	0	0	4
6	-	-	-	-	-	-	-	0	0	0
7	-	-	-	-	-	-	-	-	0	0
8	-	-	-	-	-	-	-	-	-	0
∞	-	-	-	-	-	-	-	-	-	-

Before calling the following subroutine, we create max from the input and initialize every element of max to $-\infty$.

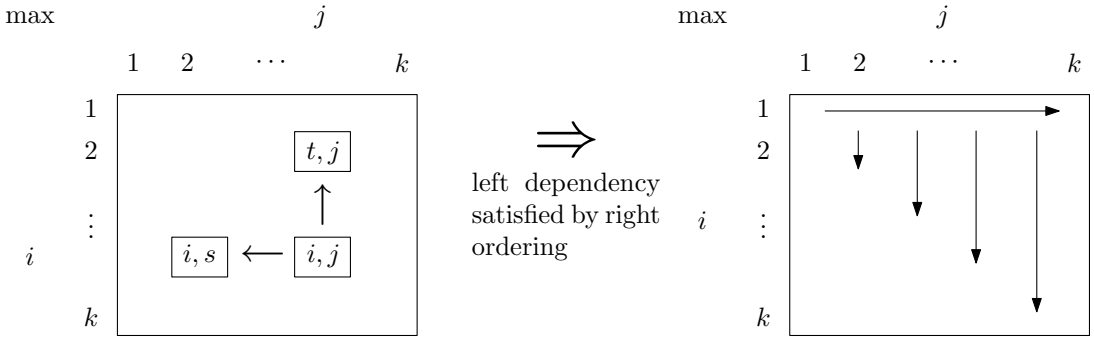
```
Algorithm DPMaxActivitySelect(x_i, x_j)
   if x_i ≥ x_j then
      return 0
   if max[i, j] ≠ −∞ then
      return max[i, j]

   for each [x_s, x_t] ∈ A do
      if x_i < x_s and x_t < x_j then
         max[i, j] ← { max[i, j],
                       DPMaxActivitySelect(x_i, x_s) + w([x_s, x_t]) +
                       DPMaxActivitySelect(x_t, x_j) }
```

Runtime complexity: In the worst case, we fill every cell in the table ($O(n^2)$ cells), where each cell could make $O(n)$ recursive calls (i.e. we must try every interval in the region corresponding to the cell). So the total work done is $O(n^3)$.

Filling the table recursively with caching is a good option because many of the table cells are unused: Certain cells are unused because α is in $\{-\infty, l_1, l_2, \ldots, l_n\}$ and ω is in $\{r_1, r_2, \ldots r_n, \infty\}$.

Iterative Solution:

For the subproblems that result from calculating $\max[i, j]$, we need the values of $\max[i, s]$ and $\max[t, j]$ where $i < s < t < j$ (i.e. $[x_s, x_t]$ is an activity in the range (x_i, x_j)). To satisfy this ordering dependency, we could fill in the table max one column at a time starting from the leftmost column and proceding to the rightmost. Then within each column, we would fill it from top to bottom.



Eliminating a Dimension:

The greedy solution to the unweighted activity selection problem iteratively added activities to the end of the schedule, but our latest dynamic programming solution to the weighted variant inserts activities arbitrarily. By changing our dynamic programming solution to be more like our greedy algorithm, we get a better solution.

Let S be an optimal solution to the weighted variant. Then it contains a *rightmost* activity s. As before, let $L(s)$ be all the activities in A that are strictly to the left of s. Then $S \cap L(s)$ is an optimal solution to the problem restricted to $L(s)$ by the same reasoning as before. Similarly if T is an optimal solution to $L(s)$, then $T \cup \{s\}$ is an optimal solution to A. This leads directly to a divide-and-conquer algorithm similar to our previous version, except that it makes one recursive call inside its main loop instead of two.

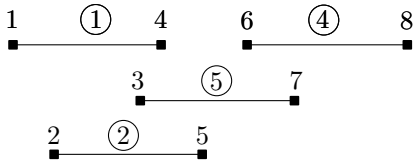
A major improvement of this approach is that all of the subproblems are of the form$\{[l, r] \in A : r < \omega\}$, where $\omega = r_i$ or ∞. So there are only $O(n)$ of these subsets and we can reduce the dimension of our cache by one. Let

$$\begin{aligned} Y &= \{l : [l, r] \in A\} \cup \{\infty\} \\ &= \{y_1, y_2, \ldots, y_k\}, \text{ where } k \leq n + 1 \text{ and } y_i < y_j, \text{ for } i < j \end{aligned}$$

Let $\max[i]$ =maximum weight solution to the subproblem $\{[l,r] \in A : r < y_i\}$. Then by maximizing over all choices of last activity

$$\max[i] = \begin{cases} 0 & \text{if } i = 1 \\ \max\{\max[j] + w([y_j, r]) : [y_j, r] \in A \text{ and } r < y_i\} & \text{otherwise} \end{cases}$$

Example:

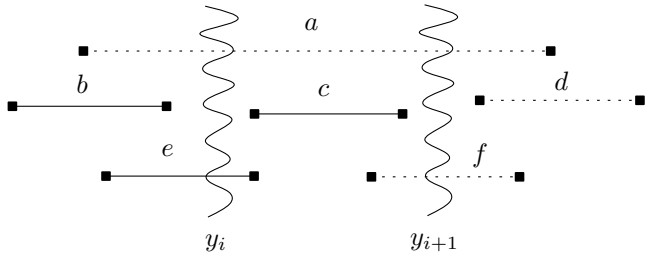


i	1	2	3	4	5
y_i	1	2	3	6	∞
max	0	0	0		

This gives us a much better iterative algorithm ($O\left(n^2\right)$). Digging deeper by expanding max recursively, we see that successive values of max have quite a bit of computation redundancy. In the example above

$$\begin{aligned} \max[4] &= \max\left\{ \begin{array}{l} \max[1] + w([y_1, 4]), \\ \max[2] + w([y_2, 5]) \end{array} \right\} \\ \max[5] &= \max\left\{ \begin{array}{l} \max[1] + w([y_1, 4]), \\ \max[2] + w([y_2, 5]), \\ \max[3] + w([y_3, 7]), \\ \max[4] + w([y_4, 8]) \end{array} \right\} = \max\left\{ \begin{array}{l} \max[4], \\ \max[3] + w([y_3, 7]), \\ \max[4] + w([y_4, 8]) \end{array} \right\} \end{aligned}$$

Just what is the precise overlap between $\max[i]$ and $\max[i+1]$?



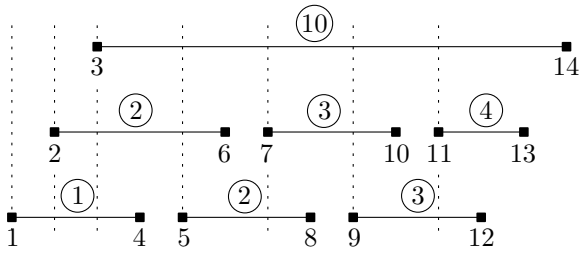
When evaluating the maximization, $\max[i+1]$ considers all the ending activities that $\max[i]$ does, plus activities like c and e, whose right endpoint lies in the range $[y_i, y_{i+1})$.

So

$$\max[i] = \begin{cases} 0 & \text{if } i = 1 \\ \max\left\{ \begin{array}{l} \max[i-1], \\ \max\{\max[j] + w([y_i, r]) : [y_j, r] \in A \text{ and } r \in [y_{i-1}, y_i]\} \end{array} \right\} & \text{otherwise} \end{cases}$$

In particular, each activity contributes to the calculation of exactly one $\max[i]$.

Example:



i	1	2	3	4	5	6	7	8
y_i	1	2	3	5	7	9	11	∞
$\max[i]$	0	0	0	1	2	3		

Removing this computational overhead leads to a very fast dynamic programming solution.

```
Algorithm FastDPMaxActivitySelect(A)

1:   sort A by right endpoint
2:   create Y ← {y_1, y_2, ..., y_k} from A
3:   for i ← 1 to k do
4:       max[i] ← 0

5:   for i ← 1 to n do
6:     binary search to find t such that r_i ∈ [y_t, y_{t+1})
7:     binary search to find s such that l_i = y_s
8:     max[t] ← max { max[t − 1],
                    { max[s] + w([l_i, r_i])
```

What is the runtime of this algorithm? Lines 1 and 2 can be implemented in $\Theta(n \log n)$ with comparison sorts. The loop on line 3 is $O(n)$. The loop at line 5 executes n times. Within the loop, lines 6 and 7 take $O(\log n)$ time, and line 8 takes $\Theta(1)$ time. So the cost of the loop is $O(n \log n)$. Therefore, the total runtime is $\Theta(n \log n)$.

Note that the binary search on line 6 can be eliminated with some book-keeping. What about the binary search on line 7?