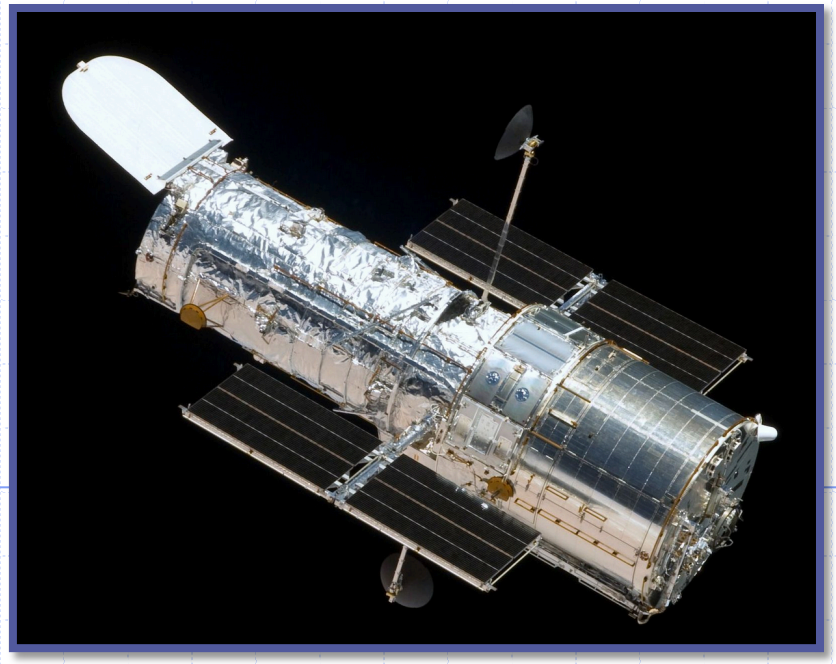
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Dynamic Programming: Telescope Scheduling



Hubble Space Telescope. Public domain image, NASA, 2009.

Motivation

- Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them.
- This high demand for observation times is especially true, for instance, for a space telescope, which could receive thousands of observation requests per month.

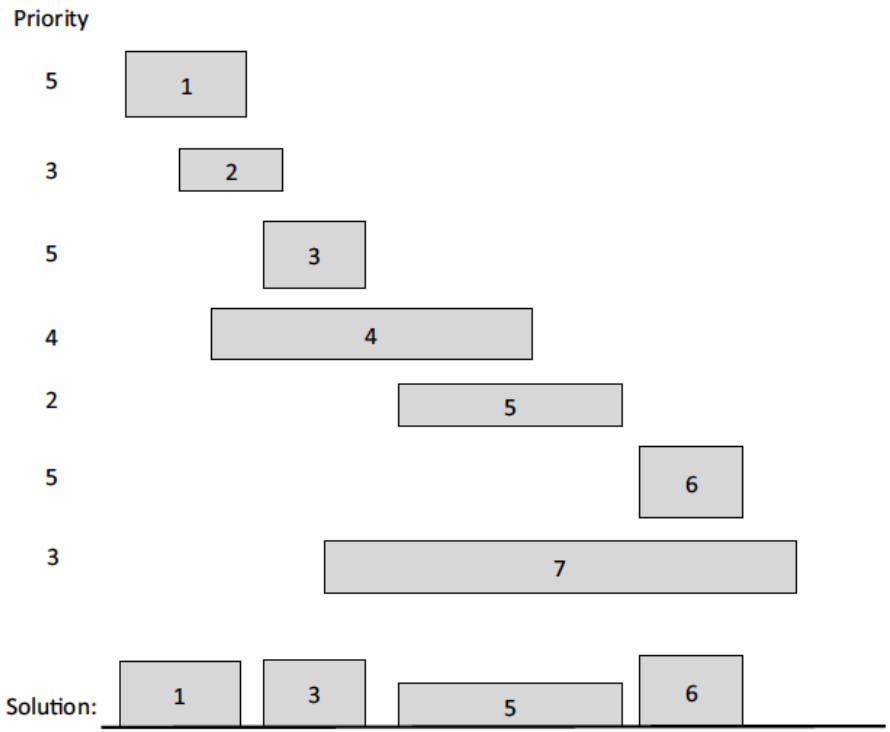
Telescope Scheduling Problem

- The input to the telescope scheduling problem is a list, **L**, of observation requests, where each request, **i**, consists of the following elements:
  - a **requested start time**, $\mathbf{s_i}$, which is the moment when a requested observation should begin
  - a **finish time**, $\mathbf{f_i}$, which is the moment when the observation should finish (assuming it begins at its start time)
  - a positive numerical **benefit**, $\mathbf{b_i}$, which is an indicator of the scientific gain to be had by performing this observation.
- The start and finish times for an observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee.

Telescope Scheduling Problem

- To get the benefit, $\mathbf{b_i}$, for an observation request, **i**, that observation must be performed by the telescope for the entire time period from the start time, $\mathbf{s_i}$, to the finish time, $\mathbf{f_i}$.
- Thus, two requests, **i** and **j**, **conflict** if the time interval $\mathbf{[s_i, f_i]}$, intersects the time interval, $\mathbf{[s_j, f_j]}$.
- Given the list, **L**, of observation requests, the optimization problem is to schedule observation requests in a nonconflicting way so as to maximize the total benefit of the observations that are included in the schedule.

Example



The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request's benefit (Priority) on the left. The optimal solution has total benefit 17=5+5+2+5.

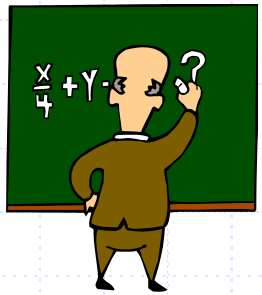
False Start 1: Brute Force

- There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of L and choose the one that has the highest total benefit without causing any scheduling conflicts.
- Implementing this brute-force algorithm would take $O(n2^n)$ time, where n is the number of observation requests.
- We can do much better than this, however, by using the dynamic programming technique.

False Start 2: Greedy Method

- A natural **greedy** strategy would be to consider the observation requests ordered by nonincreasing benefits, and include each request that doesn't conflict with any chosen before it.
  - This strategy doesn't lead to an optimal solution, however.
- For instance, suppose we had a list containing just 3 requests—one with benefit 100 that conflicts with two nonconflicting requests with benefit 75 each.
  - The greedy method would choose the observation with benefit 100, whereas we can achieve a total benefit of 150 by taking the two requests with benefit 75 each.
  - So a greedy strategy based on repeatedly choosing a nonconflicting request with maximum benefit won't work.

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Defining Simple Subproblems

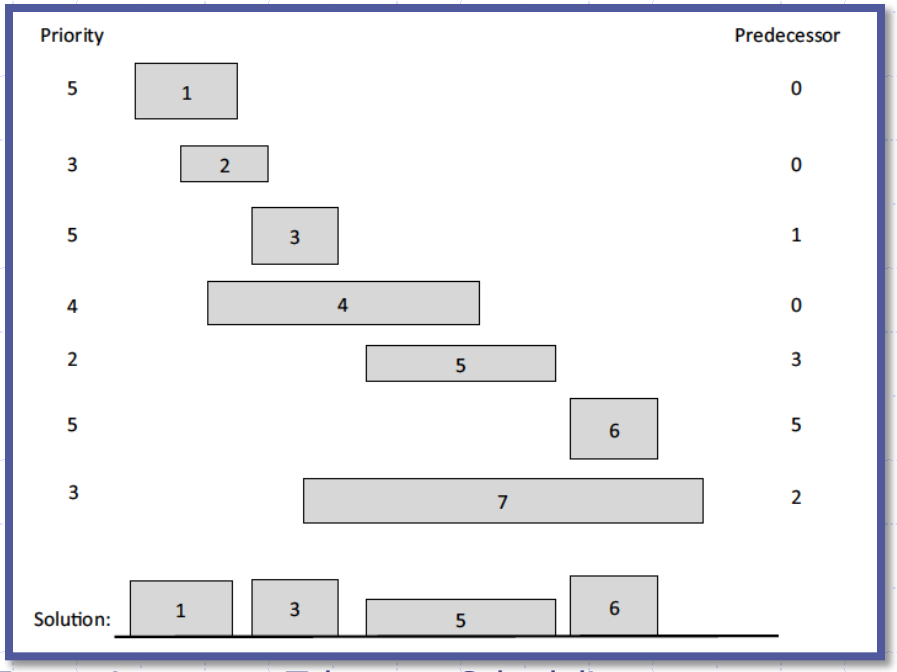
- A natural way to define subproblems is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits.
  - We already saw that ordering by benefits is a false start.
  - Start times and finish times are essentially symmetric, so let us order observations by finish times.

B_i = the maximum benefit that can be achieved with the first i requests in L.

So, as a boundary condition, we get that $B_0 = 0$.

Predecessors

- For any request **i**, the set of other requests that conflict with **i** form a contiguous interval of requests in **L**.
- Define the **predecessor**, pred(**i**), for each request, **i**, then, to be the largest index, **j < i**, such that requests **i** and **j** don't conflict. If there is no such index, then define the predecessor of **i** to be 0.



Subproblem Optimality

- A schedule that achieves the optimal value, $\mathbf{B_i}$, either includes observation **i** or not.

- If the optimal schedule achieving the benefit B_i includes observation i, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most $\text{pred}(i)$.
- On the other hand, if the optimal schedule achieving the benefit B_i does not include observation i, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves B_{i-1}.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1},\ B_{\text{pred}(i)} + b_i\}.$$

Subproblem Overlap

- The above definition has subproblem overlap.
- Thus, it is most efficient for us to use memoization when computing $\mathbf{B_i}$ values, by storing them in an array, **B**, which is indexed from 0 to **n**.
- Given the ordering of requests by finish times and an array, **P**, so that **P**[**i**] = pred(**i**), then we can fill in the array, **B**, using the following simple algorithm:

$B[0] \leftarrow 0$
for $i = 1$ to n **do**
    $B[i] \leftarrow \max\{B[i-1],\ B[P[i]] + b_i\}$

After this algorithm completes, the benefit of the optimal solution will be $B[n]$

Analysis of the Algorithm

- It is easy to see that the running time of this algorithm is **O(n)**, assuming the list **L** is ordered by finish times and we are given the predecessor for each request **i**.
- Of course, we can easily sort **L** by finish times if it is not given to us already sorted according to this ordering.
- To compute the predecessor of each request, note that it is sufficient that we also have the requests in **L** sorted by start times.
  - In particular, given a listing of **L** ordered by finish times and another listing, **L′**, ordered by start times, then a merging of these two lists, as in the merge-sort algorithm (Section 8.1), gives us what we want.
  - The predecessor of request **i** is literally the index of the predecessor in **L** of the value, $\mathbf{s_i}$, in **L′**.