24 Feb 2021

Finishing Weighted Interval Scheduling (§6.1)
Knapsack Problem (§6.4)

Recap of weighted interval scheduling

Compute-Opt(n):

if M[n] is non-null, return M[n].   "dynamic programming table"

else   // This is the first time we've been asked to solve Compute-Opt(n).

  if $n=0$

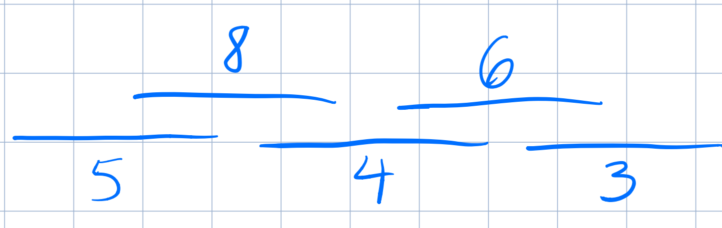
    set $M[n]=0$

  else

    compute $p(n) = \max\{ i \mid f_i < s_n \}$

    set $M[n] = \max \begin{cases} \text{Compute-Opt}(n-1), \\ v_n + \text{Compute-Opt}(p(n)) \end{cases}$

return M[n].



Pending issues: [1] Running time analysis
[2] How to output the contents of the opt set?
(3] What if....?

Compute-Opt(n):

if $M[n]$ is non-null, return $M[n]$. ← $O(1)$

else // This is the first time we've been asked to solve Compute-Opt(n).

  if $n=0$
    set $M[n]=0$ $\Big\}$ $O(1)$

  else
    $O(1)$
    ~~Compute~~ look up $p(n) = \max\{i \mid f_i < s_n\}$ ← Precomputed in $O(n \log n)$ before we start calling Compute-Opt.

    set $M[n] = \underline{\max} \begin{cases} \underline{\text{Compute-Opt}(n-1)}, \\ V_n + \underline{\text{Compute-Opt}(p(n))} \end{cases}$

  return $M[n]$.

$O(1)$

Everything is $O(1)$ except these.

PreCompute Predecessors:

  // Recall $[s_1, f_1], \ldots, [s_n, f_n]$ are sorted by increasing finish time $f_1 \le f_2 \le \ldots \le f_n$.

for $i = 1, 2, \ldots, n$:
  use binary search to find
  $p(n) = \max\{i \mid f_i < s_n\}$.

store $p(\cdot)$ table for later use.

Everything in Compute-Opt is $O(1)$ except the two recursive calls to Compute-Opt. Those are also $O(1)$ if they retrieve a value stored in the dynamic prog table.

The only unaccounted-for running time is from the <u>first</u> call to Compute-Opt(k) for each $k = 0, 1, \ldots, n$.

That takes $O(1)$ time excluding recursive first-calls to Compute-Opt(j) $(j < k)$.

Total running time:

$O(n \log n)$ to PreComputePredecessors.

$O(1)$ on each of

Compute-Opt(0)
Compute-Opt(1)
⋮
Compute-Opt(n)

$\Big\}$ $O(n)$ in total.

Overall this sums to $O(n \log n)$.

[2] How to modify to output the set of intervals?

Make the return value an ordered pair (v, S) where v is the value and S is the set.

Compute-Opt(n):
  if M[n] is non-null return M[n]
  else if $n = 0$:
    return $(0, \emptyset)$
  else:
    let (a_0, S_0) = Compute-Opt(n-1)
    Let (a_1, S_1) = Compute-Opt(p(n))
    if $v_n + a_1 > a_0$:
      $(v, S) = (v_n + a_1, \{n\} \cup S_1)$
    else:
      $(v, S) = (a_0, S_0)$
    $M[n] = (v, S)$
    return (v, S)

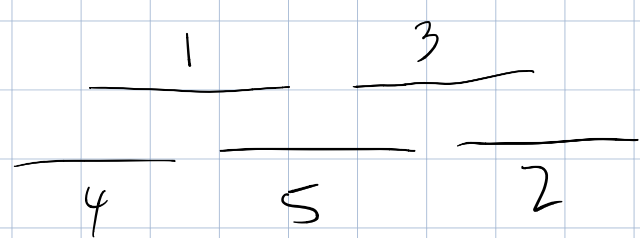
[3] What if we sorted intervals in a different order?

For example what if $v_1 \le v_2 \le \dots \le v_n$?

It still holds that the maximum weight conflict-free set either contains the n-th interval or it doesn't.

$$\mathrm{Opt}(\{1,\dots,n\}) = \max\left\{ \begin{array}{l} \mathrm{Opt}(\{1,\dots,n-1\}), \\ v_n + \mathrm{Opt}(S_n) \end{array} \right\}$$

S_n denotes {intervals that don't conflict with the nth one}.



$S_5 = \{2, 4\}$ for example.

The Knapsack Problem:

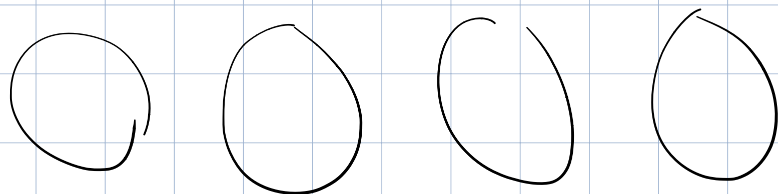
n items with weights $w_1, w_2, \dots, w_n$

Assume integers in the range $\{1, 2, \dots, W\}$.

values $v_1, v_2, \dots, v_n$

Values could be arbitrary (e.g. floating point)

You have a "knapsack" with capacity W.
Select a subset with combined weight $\leq W$.
Maximize combined value.



$w_1 = 4$ $\quad$ $w_2 = 6$ $\quad$ $w_3 = 1$ $\quad$ $w_4 = 4$

$v_1 = 50$ $\quad$ $v_2 = 60$ $\quad$ $v_3 = 10$ $\quad$ $v_4 = 50$

$$\boxed{W = 7}$$

optimal set is items 2 and 3
Combined weight 7.
value 70.

Plan of attack: optimal set either contains item n or it doesn't.

→ If you omit n, you should choose the optimal subset of items $1, \ldots, n-1$.

→ If you choose item n, you also choose the opt subset $1, \ldots, n-1$ but with a modified capacity!

Capacity $W - w_n$ rather than W.

How to design a recursive algorithm to find the optimal set that fits the knapsack, when the knapsack's capacity may change in the recursive calls?

Pass in the knapsack capacity as a parameter to the recursive function.

[int] × [int]

\# of items capacity constraint

return ordered pair (value of opt set, contents of opt set)

[float] × [set]

Compute-Opt(n, W): // find optimal knapsack solution using items $1, \ldots, n$ with total weight $\leq W$

```
if M[n,W] is non-null:   ← uninitialized table entry.
    return M[n,W]
else:
    if n=0:
        return (0, ∅)      ← empty set
    else if w_n > W:       // item n doesn't fit
        return Compute-Opt(n-1, W)
else:
```

$(a_0, S_0) = \text{Compute-Opt}(n-1, W)$

$(a_1, S_1) = \text{Compute-Opt}(n-1, W - w_n)$

if $v_n + a_1 > a_0$:

$\quad (v, S) = (v_n + a_1, \{n\} \cup S_1)$

else:

$\quad (v, S) = (a_0, S_0)$

$M[n, W] = (v, S)$

return (v, S)

Running time $O(nW)$.

"pseudopolynomial": could be exponentially bigger than # bits in the problem input if W is written in binary.