Dynamic Programming(Weighted Interval Scheduling)

Arash Rafiey

17 November, 2016

Dynamic Programming

1. Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).

Dynamic Programming

1. Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).
2. We use the basic idea of divide and conquer. Dividing the problem into a number of subproblems.

Dynamic Programming

1. Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).
2. We use the basic idea of divide and conquer. Dividing the problem into a number of subproblems.
3. There are polynomial number of subproblems (If the input is of size n then there are at most n, n^2,.., n^5 subproblems and not $2^{\frac{n}{4}}$ sub-problems).

Dynamic Programming

1. Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).
2. We use the basic idea of divide and conquer. Dividing the problem into a number of subproblems.
3. There are polynomial number of subproblems (If the input is of size n then there are at most n, n^2,.., n^5 subproblems and not $2^{\frac{n}{4}}$ sub-problems).
4. The solution to the original problem can be easily computed from the solution to the subproblems (e.g. sum the solutions to the subproblems and get the solution to the original)

Dynamic Programming

1. Dynamic programming algorithms are used for optimization (for example, finding the shortest path between two points, or the fastest way to multiply many matrices).
2. We use the basic idea of divide and conquer. Dividing the problem into a number of subproblems.
3. There are polynomial number of subproblems (If the input is of size n then there are at most n, n^2,.., n^5 subproblems and not $2^{\frac{n}{4}}$ sub-problems).
4. The solution to the original problem can be easily computed from the solution to the subproblems (e.g. sum the solutions to the subproblems and get the solution to the original)
5. The idea is to solve each subproblem only once. Once the solution to a given subproblem has been computed, it is stored or memorized: (the next time the same solution is needed, it is simply looked up). This way we reduce the number of computations.

Problem Statement:

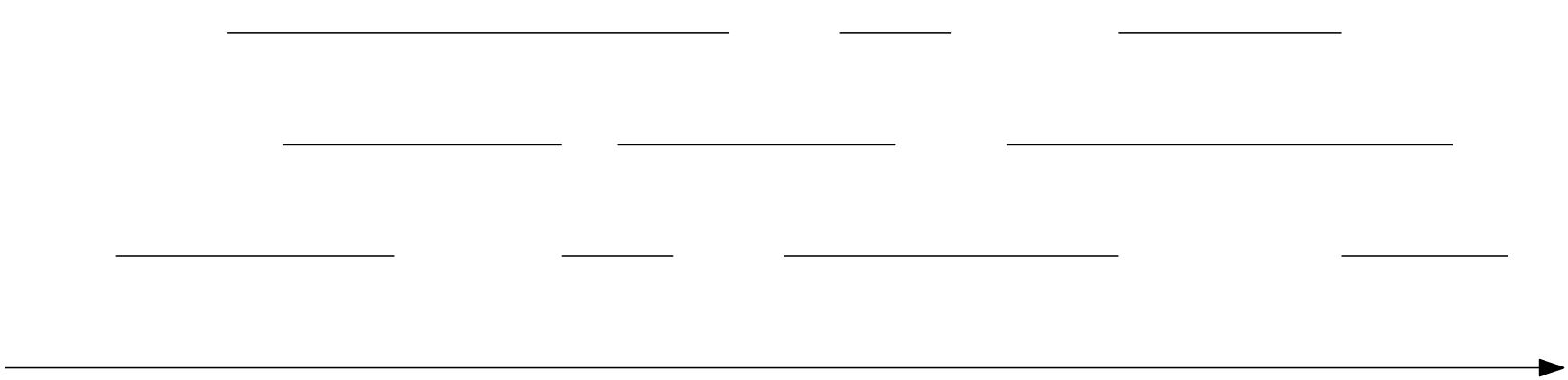
1. We have a resource and many people request to use the resource for periods of time (an interval of time)
2. Each interval (request) i has a start time s_i and finish time f_i
3. Each interval (request) i, has a value v_i

Conditions:

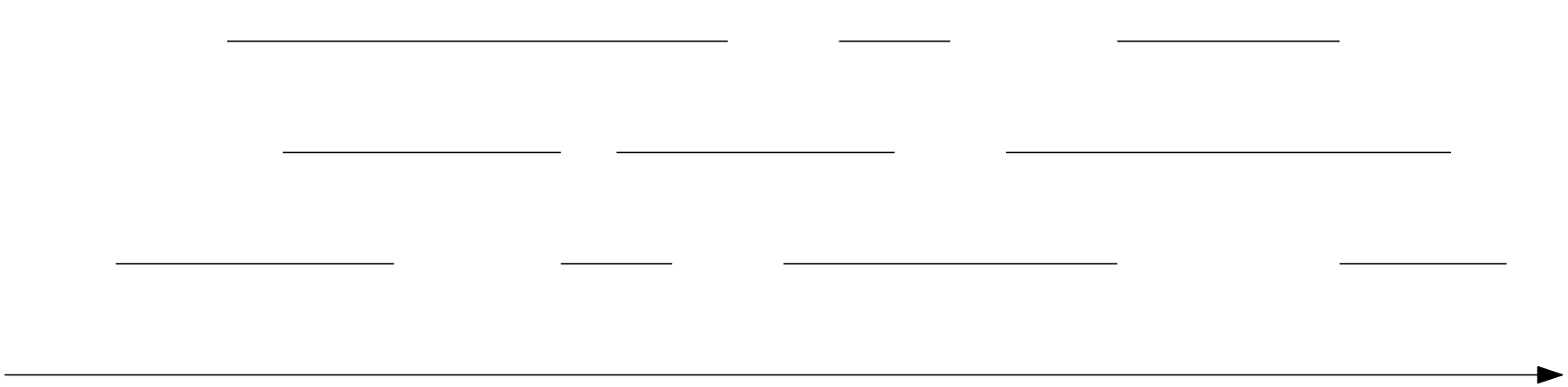
- the resource can be used by at most one person at a time.
- we can accept only *compatible* intervals (requests) (overlap-free).

Goal: a set of *compatible* intervals (requests) with a maximum total value .

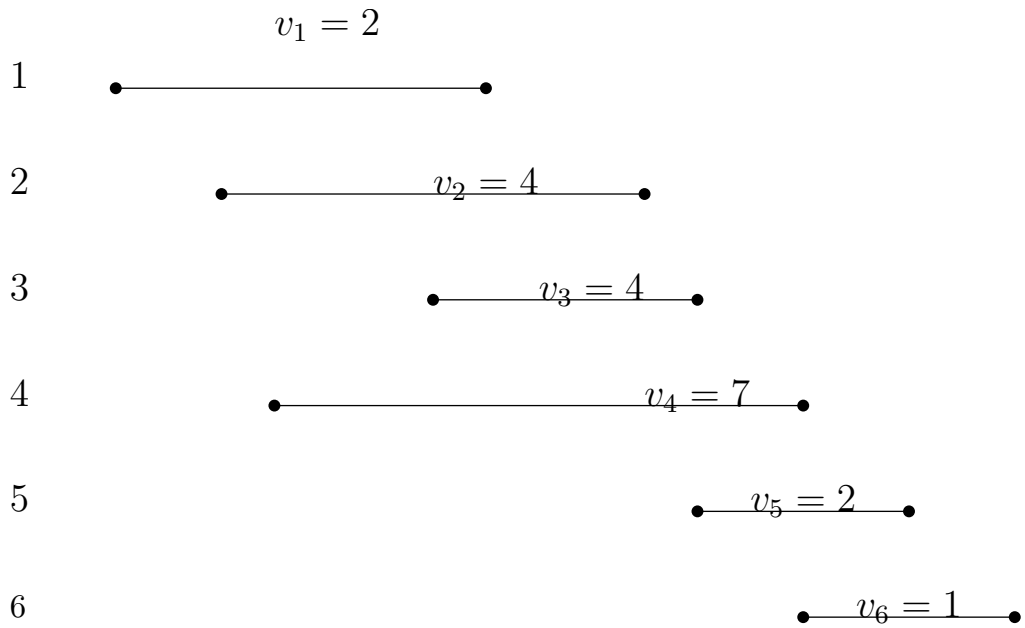
What if all the values are the same ?

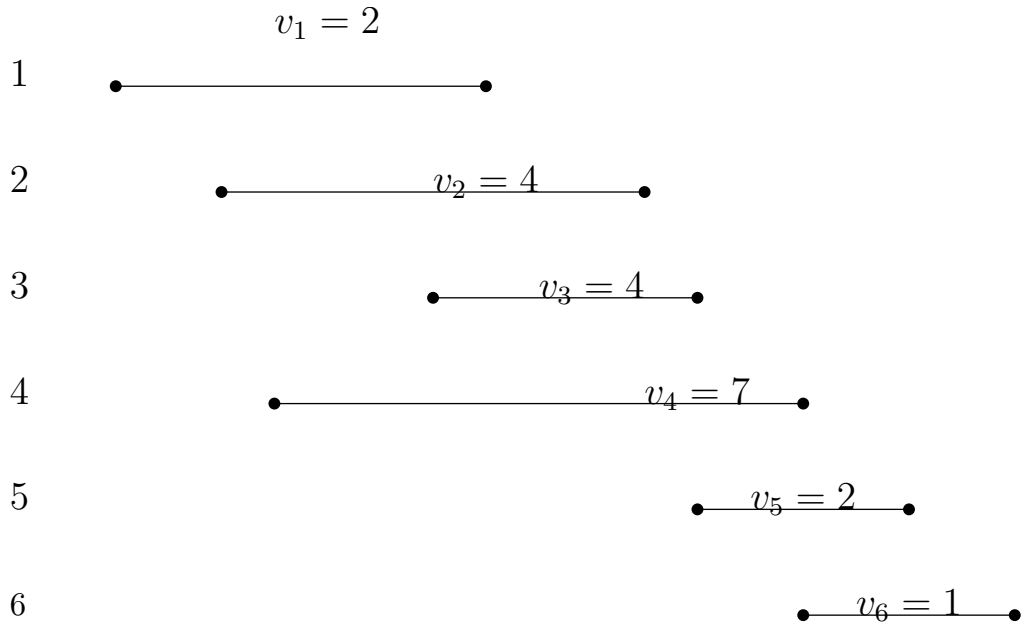


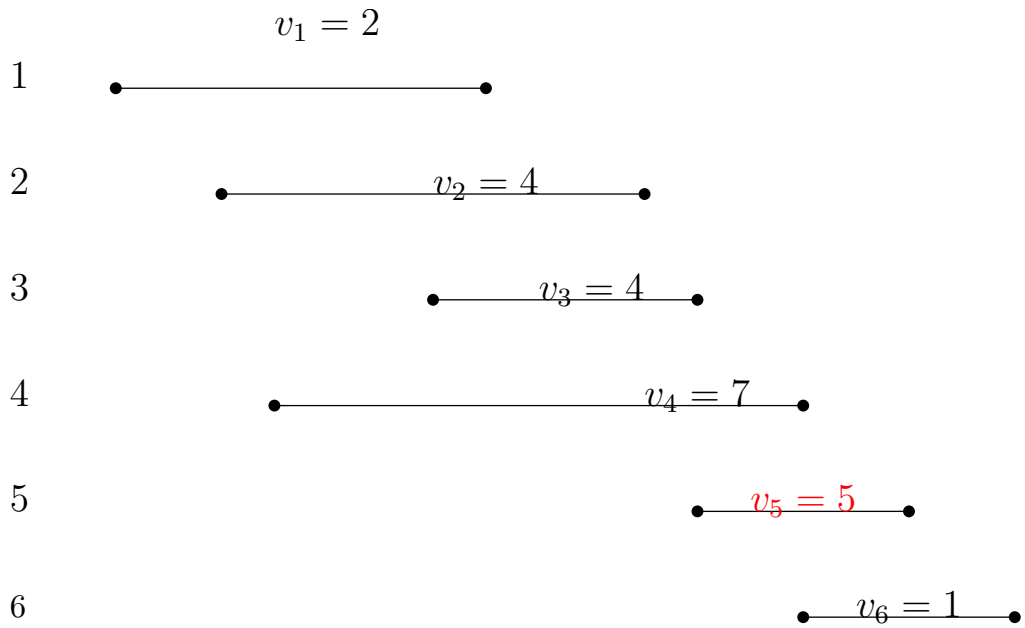
What if all the values are the same ?



We start with the one which has the smallest finish time and add it into our solution and remove the ones that have intersection with it and repeat!


$v_1 = 2$
1
2
$v_2 = 4$
3
$v_3 = 4$
4
$v_4 = 7$
5
$v_5 = 2$
6
$v_6 = 1$


$v_1 = 2$
1
2
$v_2 = 4$
3
$v_3 = 4$
4
$v_4 = 7$
5
$v_5 = 2$
6
$v_6 = 1$


$v_1 = 2$
1
2
$v_2 = 4$
3
$v_3 = 4$
4
$v_4 = 7$
5
$v_5 = 5$
6
$v_6 = 1$

We order the intervals (requests) based on their finishing time (non-increasing order of finishing time) :

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

We order the intervals (requests) based on their finishing time (non-increasing order of finishing time) :

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

For every interval j let $p(j)$ be the largest index $i < j$ such that intervals i, j do not overlap (have more than one points in common)

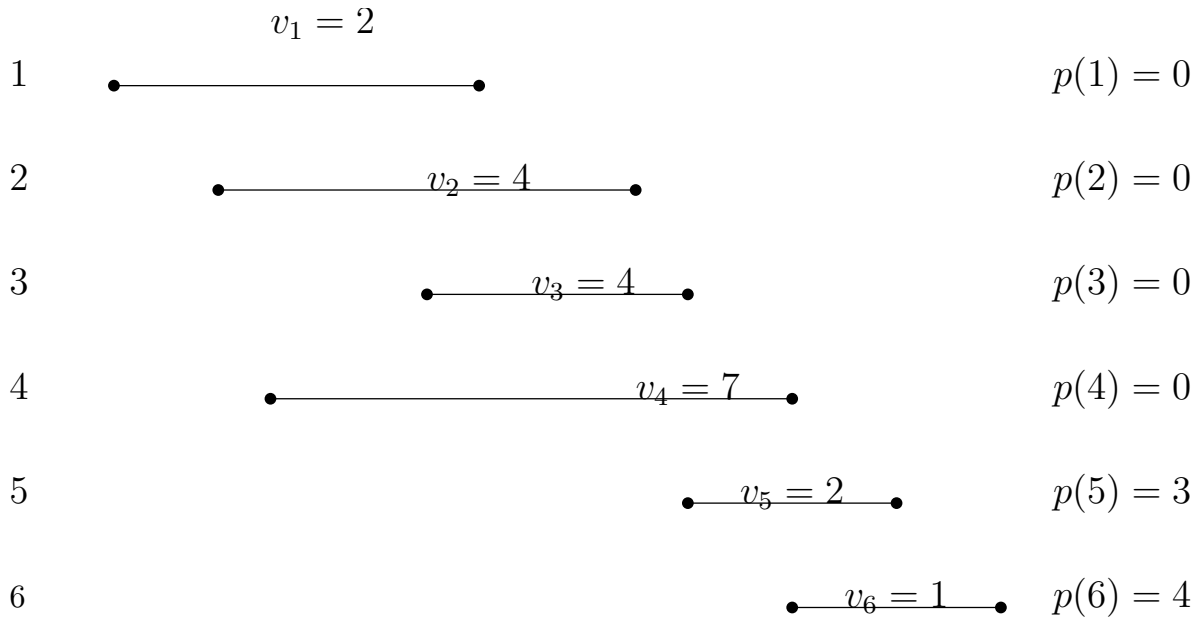
$p(j) = 0$ if there is no interval before j and disjoint from j.

We order the intervals (requests) based on their finishing time (non-increasing order of finishing time) :

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

For every interval j let $p(j)$ be the largest index $i < j$ such that intervals i, j do not overlap (have more than one points in common)

$p(j) = 0$ if there is no interval before j and disjoint from j.



Let $OPT(j)$ be the value of the optimal solution considering only intervals from 1 to j (according to their order).

Let $OPT(j)$ be the value of the optimal solution considering only intervals from 1 to j (according to their order).

We can write a recursive formula to compute $OPT(j)$,

Either interval j is in the optimal solution or j is not in the solution.

Let $OPT(j)$ be the value of the optimal solution considering only intervals from 1 to j (according to their order).

We can write a recursive formula to compute $OPT(j)$,

Either interval j is in the optimal solution or j is not in the solution. Therefore :

$$OPT(j) = \max\{OPT(j-1), v_j + OPT(p(j))\}$$

Let $OPT(j)$ be the value of the optimal solution considering only intervals from 1 to j (according to their order).

We can write a recursive formula to compute $OPT(j)$,

Either interval j is in the optimal solution or j is not in the solution. Therefore :

$$OPT(j) = \max\{OPT(j-1), v_j + OPT(p(j))\}$$

Try not to implement using recursive call because the running time would be exponential!

Recursive function is easy to implement but time consuming!

Iterative-Compute-Opt()

1. $M[0] := 0$;
2. **for** $j = 1$ to n
3. $\quad M[j] = \max\{v_j + M[p(j)], M[j-1]\}$

Iterative-Compute-Opt()

1. $M[0] := 0$;
2. **for** $j = 1$ to n
3. $\quad M[j] = \max\{v_j + M[p(j)], M[j-1]\}$

Find-Solution(j)

1. **if** $j = 0$ then return
2. **else**
3. $\quad$ **if** $(v_j + M[p(j)] \geq M[j-1])$
4. $\quad\quad$ print j and print " "
5. $\quad\quad$ Find-Solution(p(j))
6. $\quad$ **else**
7. $\quad\quad$ Find-Solution(j-1)

Subset Sums and Knapsack Problem

Subset Sums (Problem Statement):

1. We are given n items $\{1, 2, \ldots, n\}$ and each item i has weight w_i
2. We are also given a bound W

Goal: select a subset S of the items so that :

1. $\sum_{i \in S} w_i \leq W$
2. $\sum_{i \in S} w_i$ *is maximized*

A greedy approach won't work if it is based on picking the biggest value first. Suppose we have a set $\{W/2+1, W/2, W/2\}$ of items. If we choose $W/2+1$ then we can not choose anything else. However the optimal is $W/2+W/2$.

A greedy approach won't work if it is based on picking the biggest value first. Suppose we have a set $\{W/2+1, W/2, W/2\}$ of items. If we choose $W/2+1$ then we can not choose anything else. However the optimal is $W/2+W/2$.

A greedy approach won't work if it is based on picking the smallest value first. Suppose we have a set $\{1, W/2, W/2\}$ of items. If we choose 1 then we can only choose $W/2$ and nothing more. However the optimal is $W/2+W/2$.

A greedy approach won't work if it is based on picking the biggest value first. Suppose we have a set $\{W/2+1, W/2, W/2\}$ of items. If we choose $W/2+1$ then we can not choose anything else. However the optimal is $W/2 + W/2$.

A greedy approach won't work if it is based on picking the smallest value first. Suppose we have a set $\{1, W/2, W/2\}$ of items. If we choose 1 then we can only choose $W/2$ and nothing more. However the optimal is $W/2 + W/2$.

Exhaustive search! Produce all the subset and check which one satisfies the constraint ($\leq W$) and has maximum size. Running time $\mathcal{O}(2^n)$.

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subseteq \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subseteq \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

If $w_i > w$ then $OPT[i, w] = OPT[i-1, w]$ otherwise

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subseteq \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

If $w_i > w$ then $OPT[i, w] = OPT[i-1, w]$ otherwise

$$OPT[i, w] = \max\{OPT[i-1, w], w_i + OPT[i-1, w-w_i]\}$$

Subset-Sum(n,W)

1. Array $M[0, .., n, 0, .., W]$
2. **for** ($i = 1$ to W) $M[0, i] = 0$
3. **for** ($i = 1$ to n)
4. **for** ($w = 1$ to W)
5. **if** $(w < w_i)$ then $M[i, w] = M[i - 1, w]$
6. **else**
7. **if** $(w_i + M[i - 1, w - w_i] > M[i - 1, w])$
8. $M[i, w] = w_i + M[i - 1, w - w_i]$
9. **else** $M[i, w] = M[i - 1, w]$

Subset-Sum(n,W)

1. Array $M[0, .., n, 0, .., W]$
2. **for** ($i = 1$ to W) $M[0, i] = 0$
3. **for** ($i = 1$ to n)
4. **for** ($w = 1$ to W)
5. **if** $(w < w_i)$ then $M[i, w] = M[i - 1, w]$
6. **else**
7. **if** $(w_i + M[i - 1, w - w_i] > M[i - 1, w])$
8. $M[i, w] = w_i + M[i - 1, w - w_i]$
9. **else** $M[i, w] = M[i - 1, w]$

Time complexity $\mathcal{O}(nW)$

Exercise

Suppose $W = 6$ and $n = 3$ and the items of sizes $w_1 = w_2 = 2$ and $w_3 = 3$.
Fill out matrix M.

Problem : We are given n jobs $\{J_1, J_2, \ldots, J_n\}$ where each J_i has a processing time $p_i > 0$ (an integer).

We have two identical machines M_1, M_2 and we want to execute all the jobs.

Schedule the jobs so that the finish time is minimized.

Knapsack Problem

Knapsack Problem :

1. We are given n items $\{1, 2, \ldots, n\}$ and each item i has weight w_i
2. Each item has value v_i
3. We are also given a bound W

Goal: select a subset S of the items so that :

1. $\sum_{i \in S} w_i \leq W$
2. $\sum_{i \in S} v_i$ *is maximized*

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subset \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subset \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

If $w_i > w$ then $OPT[i, w] = OPT[i-1, w]$ otherwise

Let $OPT[i, w]$ be the optimal maximum value of a set $S \subset \{1, 2, \ldots, i\}$ where the total value of the items in S is at most w.

If $w_i > w$ then $OPT[i, w] = OPT[i-1, w]$ otherwise

$$OPT[i, w] = \max\{OPT[i-1, w], v_i + OPT[i-1, w - w_i]\}$$