# CS 341: ALGORITHMS (S18) — LECTURE 6
# INTERVAL SCHEDULING

ERIC BLAIS

In the last two lectures, we explored how the divide and conquer technique is useful for designing algorithms for many different problems. Today, we start examining another technique: *greedy algorithms.*

## 1. Greedy algorithms

A *greedy* algorithm is one in which we:

(1) Break down a problem into a sequence of decisions that need to be made, then
(2) Make the decisions one at a time, each time choosing the option that is optimal at the moment (and not worrying about later decisions).

This is one of the simplest algorithm design techniques around,[1] and as we will see it yields efficient algorithm for many different problems. But the challenge with this technique in many instances is proving the algorithm's correctness—or, often, *determining* whether the algorithm is correct or not in the first place.

You have already seen one classic greedy algorithm in CS 240: Huffman codes are optimal prefix codes obtained by running the greedy algorithm to join trees with different frequencies together. This algorithm is both correct and efficient.

We also use a greedy algorithm in real life when we make change.

**Problem 1** (Making change)**.** *Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $v \geq 1$, determine the minimum number of coins whose denominations sum to $v$. (I.e., a valid solution $a_1, \ldots, a_n \in \mathbb{N}$ is one that satisfies $\sum_{i=1}^n a_i d_i = v$ and minimizes $\sum_{i=1}^n a_i$.)*

For example, we have coins worth 200, 100, 25, 10, 5, and 1 cents in current circulation in Canada. How would you make change for 1.43$ using those coins? You would start by taking a coin with maximal denomination that is at most 143 (here the loonie worth 100 cents), subtract that amount from the total (leaving 43 cents in this case), and repeat.

**Algorithm 1:** GreedyChange($d_1 > d_2 > \cdots > d_n ; v$)

```
for $i = 1, \ldots, n$ do
    $a_i \leftarrow \lfloor v \,/\, d_i \rfloor$;
    $v \leftarrow v \bmod d_i$;
return $(a_1, \ldots, a_n)$;
```

[1] As an aside: There is an even simpler technique that we will not cover in this course but for which I am especially fond: the *random choice* technique, where you don't even try to make the best decision but instead choose one at random instead.

Does this algorithm always return a valid solution to the Making Change problem? It does for every value $v$ when the coins have denominations 200, 100, 25, 10, 5, and 1 but the proof of correctness in this case is not trivial. In general, however, there are denominations and choices of $v$ where the algorithm does not return the minimal number of coins. Take for example the case where the denominations are 8, 7, and 1 and the value to return is 14. The greedy algorithm will return the solution $(1, 0, 6)$, for a total of 7 coins, when the solution $(0, 2, 0)$ requires only two coins.

Today, we explore another fundamental problem that can be solved using greedy algorithms.

## 2. Interval scheduling problem

Many real-world scheduling problems can be formulated in terms of the abstract *interval scheduling problem.*

**Definition 6.1** (Interval scheduling problem)**.** An instance of the *interval scheduling problem* is a set of $n$ pairs of start and finish times $(s_1, f_1), \dots, (s_n, f_n)$ where each pair $(s_i, f_i)$ satisfies $s_i < f_i$. A valid solution to an instance is a a maximum subset $I \subseteq \{1, 2, \dots, n\}$ such that no two intervals in $I$ overlap. I.e., for every $i, j \in I$, $s_i > f_j$ or $s_j > f_i$.

It is quite natural to try to use the greedy method to solve this problem: to do so, we just need to decide how the algorithm chooses the "best" interval to add to $I$ given the intervals that have already been added to this set. We have many possibilities on how to define the notion of "best" interval:

**Earliest starting time:** Pick the interval with the earliest starting time that does not overlap any of the intervals we already added to $I$.

**Earliest finish time:** Pick the interval with the earliest finish time that does not overlap any of the intervals we already added to $I$.

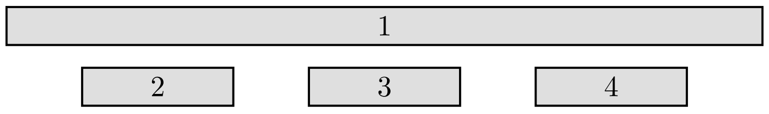
**Shortest interval:** Pick the interval with the shortest length $f_i - s_i$ among those that don't overlap any of the intervals we already added to $I$.

**Minimum conflicts:** Let $J$ be the set of intervals that don't overlap with any interval in $I$. Pick the interval in $J$ that overlaps with the fewest other number of other intervals in $J$.

All four options sound perfectly reasonable, but only one yields an algorithm that always outputs a valid solution to the interval scheduling problem.

**Proposition 6.2.** *The greedy algorithm with earliest starting time does not always find a valid solution to the interval scheduling problem.*

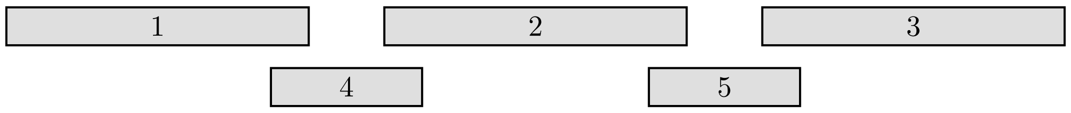
*Proof.* Consider the set of intervals



The valid solution is $I = \{2, 3, 4\}$ but the greedy algorithm outputs $I = \{1\}$ instead. □

**Proposition 6.3.** *The greedy algorithm with shortest interval does not always find a valid solution to the interval scheduling problem.*

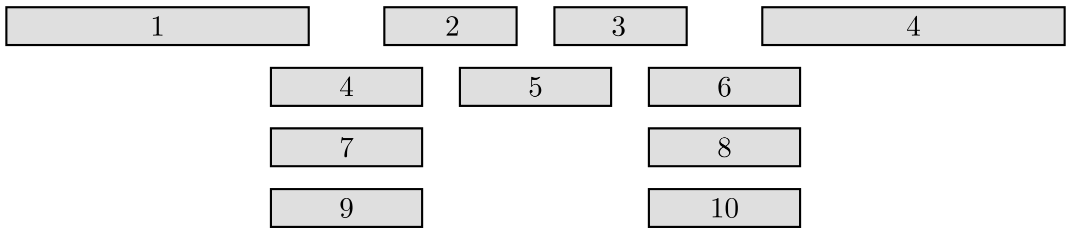
*Proof.* Consider the set of intervals



The valid solution is $\{1, 2, 3\}$ but the greedy algorithm outputs $I = \{4, 5\}$ instead. □

**Proposition 6.4.** *The greedy algorithm with minimal conflicts does not always find a valid solution to the interval scheduling problem.*

*Proof.* Consider the set of intervals



The valid solution is $\{1, 2, 3, 4\}$ but the greedy algorithm outputs $I = \{1, 4, 5\}$ instead. □

We are now left with a single candidate algorithm: earliest finish time. We can implement this greedy algorithm in a simple way: sort the intervals by finish time, and when we go through the list we add an interval $(s_i, f_i)$ iff its start time is larger than the last (and therefore most recently added) finish time of any interval in $I$.

```
Algorithm 2: GreedyScheduler($(s_1, f_1)$,...,$(s_n, f_n)$)
A ← indices {1,...,n} sorted by $f_i$;
I ← A[1];
$f^* \leftarrow f_{A[1]}$;
for i = 2,...,n do
    if $s_{A[i]} > f^*$ then
        $I \leftarrow I \cup A[i]$;
        $f^* \leftarrow f_{A[i]}$;
return I;
```

The time complexity of the GreedyIntervalScheduler is $\Theta(n \log n)$. It remains to show that it always returns a valid solution to the interval scheduling problem. We do so via an *exchange argument*: we show that for any maximum set $I^* \subseteq [n]$ of non-interlapping intervals, we can exchange some of the intervals in $I^*$ in a way that we end up with the output of the GreedyIntervalScheduler.

As a first step, let us show how we can always exchange the *first* interval in a maximum set $I^*$ of non-overlapping intervals with the first interval chosen by GreedyIntervalScheduler and still obtain a valid solution to the interval scheduling problem.

**Proposition 6.5.** *Let $I^* = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, n\}$ be the indices of a maximum set of non-overlapping intervals in some instance of the Interval Scheduling problem sorted by finish times so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$. Let $j \in [n]$ be the index of the first interval selected by the GreedyIntervalScheduler. Then $I^\dagger = \{j, i_2, i_3, \ldots, i_k\}$ is also a maximum set of non-overlapping intervals.*

*Proof.* The set $I^\dagger$ must have the same cardinality as $I^*$ since $f_j \leq f_{i_1} < f_{i_\ell}$ for each $\ell \in \{2, 3, \ldots, k\}$ and so $j \notin \{i_2, \ldots, i_k\}$. We need to show that $I^\dagger$ is a set of non-overlapping intervals. Since $I^*$ is a set of non-overlapping intervals, the only thing we need to show is that the interval $j$ does not overlap with any of the intervals $i_2, \ldots, i_k$.

The fact that $I^*$ contains non-overlapping intervals and that $f_{i_1}$ is the smallest finish time implies that we must have $f_{i_1} < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$. And the definition of the GreedyIntervalScheduler guarantees that $f_j \leq f_{i_1}$, so we also have $f_j < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$ and the interval $j$ does not overlap with any of the intervals $i_2, \ldots, i_k$, as we wanted to show. □

We can now use a proof by induction to establish the correctness of the GreedyScheduler algorithm.

**Theorem 6.6.** *The* GreedyScheduler *solves the interval scheduler problem.*

*Proof.* Let $I^* = \{i_1, i_2, \ldots, i_k\}$ be a maximum set of non-overlapping intervals, and let $I^\dagger = \{j_1, j_2, \ldots, j_m\}$ be the set of non-overlapping intervals returned by the algorithm. We again sort the indices so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$ and $f_{j_1} < \cdots < f_{j_m}$. Clearly, $m \leq k$; we want to show that in fact $m = k$.

Let us now use a proof by induction on $c$ to show that for every $c$ in the range $1 \leq c \leq m$, the set $I^{(c)} = \{j_1, j_2, \ldots, j_c, i_{c+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. The base case was established in Proposition 6.5.

For the induction step with $c \geq 2$, the induction hypothesis is that $I^{(c-1)}$ is a maximum set of non-overlapping intervals; we want to show the same is true of $I^{(c)}$. Note that $I^{(c)} = (I^{(c-1)} \setminus \{i_c\}) \cup \{j_c\}$. We must have $f_{j_c} \leq f_{i_c} < f_{i_{c+1}} < \cdots < f_{i_k}$ so $j_c \notin \{j_1, \ldots, j_{c-1}, i_{c+1}, \ldots, i_k\}$ and $|I^{(c)}| = |I^{(c-1)}|$. Furthermore, since $I^\dagger$ and $I^{(c-1)}$ are sets of non-overlapping intervals and since $f_{j_c} \leq f_{i_c} < s_{i_{c+1}} < \cdots < s_{i_k}$, the set $I^{(c)}$ is a maximum set of non-overlapping intervals.

The proof by induction we just completed shows that $I^{(m)} = \{j_1, j_2, \ldots, j_m, i_{m+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. But the greedy algorithm returns $I^\dagger = \{j_1, \ldots, j_m\}$ only if all other intervals with finish times greater than $j_m$ intersect with some interval already in $I^\dagger$; this means that we must have $m = k$ and the algorithm returns a maximum set of non-overlapping intervals. □

# CS 341: ALGORITHMS (S18) — LECTURE 7
# MORE GREEDY ALGORITHMS

ERIC BLAIS

We saw in the last lecture how the greedy algorithm method can be used to solve the interval scheduling problem. In this lecture, we will examine a few other problems that can be solved by greedy algorithms and see how a common exchange method can be used to establish the correctness of these algorithms.

## 1. Minimizing lateness

There are a number of other scheduling problems for which the greedy algorithm is effective. Here's one that may be relevant with respect to juggling coursework for multiple classes at the same time:

**Definition 7.1** (Minimal lateness problem)**.** An instance of the *minimal lateness problem* is a set of $n$ tasks with processing times $p_1, \ldots, p_n$ and deadlines $d_1, \ldots, d_n$. A valid solution to the instance is an ordering of the tasks that minimizes the maximum lateness of any task when they are performed one at a time in that order. Formally, given a permutation $\pi$ of the set $\{1, 2, \ldots, n\}$, the *lateness* of the $k$th task in this order is $L_k^{(\pi)} := \sum_{i:\pi(i)\leq\pi(k)} p_i - d_k$ and $\pi$ is a valid solution if $\max_{k\leq n} L_k^{(\pi)}$ is minimal among all permutations.

For example, an input to the problem may be tasks with the following processing times and deadlines:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $p_i$ | 4 | 2 | 4 | 3 | 1 | 4 | 6 |
| $d_i$ | 5 | 9 | 1 | 3 | 3 | 4 | 10 |

If we process the tasks in the order above, we obtain lateness values

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $L_i$ | -1 | -3 | 9 | 10 | 11 | 14 | 14 |

so that the maximum lateness of this ordering is 14. For this instance, every ordering of the tasks has maximum lateness at least 14, so the order $1, 2, \ldots, n$ is a valid solution.

Let's explore how we can solve this problem using greedy algorithms. Here the natural way to break down the problem into individual decisions is to pick which task to do first, then which one to do second, etc. There are a number of different criteria we could use to decide which task to schedule next.

**Shortest processing time first:** Sort the tasks in order of increasing processing times.

**Earliest deadline:** Sort the tasks in order of increasing deadlines.

**Smallest slack:** At each step, pick the task with the smallest remaining time until its deadline: if the first $j$ tasks already selected have total processing time $t_j$, pick the task that minimizes $d_j - t_j$ next.

Of those three options, only the earliest deadline criterion yields a greedy algorithm that solves the Minimal Lateness problem.[1]

**Algorithm 1:** GreedyLateness($p_1, \dots, p_n, d_1, \dots, d_n$)

```
$\pi \leftarrow$ a permutation of $\{1, \dots, n\}$ for which $d_{\pi(1)} \le d_{\pi(2)} \le \dots \le d_{\pi(n)}$;
return $\pi$;
```

The permutation $\pi$ can be computed in time $\Theta(n \log n)$, and this is also the time complexity of the algorithm. The more challenging aspect of the analysis of this algorithm is its proof of correctness. We again use an exchange argument. We also use the fact that we can sort an out-of-order sequence by swapping pairs of consecutive elements.

**Fact 1** (Bubble sort)**.** *For $\pi$ any ordering of $1, 2, \dots, n$, if we repeatedly*

(1) *Find a pair of elements $i < j$ where $j$ is right before $i$ in $\pi$;*
(2) *Update $\pi$ by swapping elements $i$ and $j$;*

*then after at most $\Theta(n^2)$ swaps, we end up with the sorted ordering $1\,2\,3 \cdots n$.*

For example, starting with the ordering 1 5 3 4 2, the sorting algorithm described above yields the following sequence of orderings:

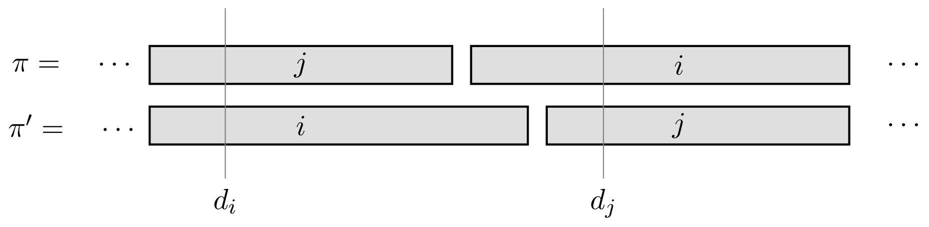
$$1\,\mathbf{5}\,\mathbf{3}\,4\,2$$
$$1\,3\,\mathbf{5}\,\mathbf{4}\,2$$
$$1\,3\,4\,\mathbf{5}\,\mathbf{2}$$
$$1\,3\,\mathbf{4}\,\mathbf{2}\,5$$
$$1\,\mathbf{3}\,\mathbf{2}\,4\,5$$
$$1\,2\,3\,4\,5$$

We are now ready to prove the correctness of the GreedyLateness algorithm.

**Theorem 7.2.** *The* GreedyLateness *algorithm solves the Maximal Lateness problem.*

*Proof.* For simplicity, let us reorder the tasks by increasing deadline so that the greedy algorithm performs them in order (i.e., 1, then 2, then 3, etc.) We want to prove that this order is a valid solution.

Consider any other ordering $\pi$. Then in that order there must be two consecutive tasks $i, j$ with $d_i \le d_j$ but $j$ performed right before $i$. Swap those two tasks to obtain a new ordering $\pi'$.



[1]Exercise: Prove that statement!

Then every task except $i$ and $j$ have the same lateness in $\pi$ and in $\pi'$. The lateness of $i$ satisfies

$$L_i^{(\pi')} \le L_i^{(\pi)}$$

since we do task $i$ earlier in $\pi'$. And the lateness of $j$ satisfies

$$L_j^{(\pi')} \le L_i^{(\pi)}$$

because $d_i \le d_j$. Therefore, the maximum lateness of $\pi'$ is at most that of $\pi$. This means that we can continue swapping in this way until we obtain the ordering $1\,2\,\cdots\,n$ of the greedy algorithm, and at every step along the way we never increase the maximum lateness so that the algorithm's solution is valid. □

## 2. Fractional knapsack

Now a completely different example.

**Definition 7.3** (Fractional knapsack)**.** An instance of the *fractional knapsack problem* is a set of $n$ items that have positive weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, as well as a maximum weight capacity $W$ of the knapsack. A valid solution to the problem is a set of amounts $x_1, \dots, x_n$ of each item that you put in your backpack that maximizes the total value $\sum_{i=1}^n \frac{x_i}{w_i} v_i$ among all possible sets of amounts where for each item $i$ we have $0 \le x_i \le w_i$ and the total weight taken is $\sum_{i=1}^n x_i \le W$.

For example, an instance may have three items with weights and values

| | 1 | 2 | 3 |
|---|---|---|---|
| $w_i$ | 4 | 3 | 3 |
| $v_i$ | 12 | 7 | 6 |

and a total knapsack capacity weight $W = 6$.

There is a natural greedy algorithm for this problem: sort the items by decreasing relative value $v_i/w_i$, then consider each element in turn and put as much of it in the knapsack as can fit.

```
Algorithm 2: GreedyKnapsack($w_1, \dots, w_n, v_1, \dots, v_n, W$)
Order the items by decreasing value of $v_i/w_i$;
for $i = 1, \dots, n$ do
    $x_i \leftarrow \min\{W, w_i\}$;
    $W \leftarrow W - x_i$;
return $x_1, \dots, x_n$;
```

**Theorem 7.4.** *The* GreedyKnapsack *algorithm solves the fractional knapsack problem.*

*Proof.* Let $y_1, \dots, y_n$ be a valid solution to the fractional knapsack problem, using the ordering defined by the algorithm where $v_i/w_i$ is decreasing. And let $x_1, \dots, x_n$ be the solution returned by the GreedyKnapsack algorithm. We prove that $x_1, \dots, x_n$ is a valid solution by induction on the number of indices $i \le n$ for which $x_i \ne y_i$.

In the base case, when $x_i = y_i$ for each $i = 1, 2, \dots, n$, then $x_1, \dots, x_n$ is the same solution as $y_1, \dots, y_n$ so it is a valid solution.

For the induction step, let $m = |\{i \le n : x_i \ne y_i\}| \ge 1$ be the number of differences in the solution. By the induction hypothesis, if there is a valid solution $y'_1, \dots, y'_n$ with

$|\{i \le n : x_i \ne y'_i\}| < m$, then $x_1, \ldots, x_n$ is also a valid solution. Our goal is to now use an exchange argument to show that such a solution $y'_1, \ldots, y'_n$ exists.

Let $k \le n$ be the smallest index where $x_k \ne y_k$. Then it must be that $x_k > y_k$ since GreedyKnapsack maximizes the value of $x_k$. And since $\sum x_i = \sum y_i$, there must be an index $\ell > k$ for which $y_\ell > x_\ell$. Let's exchange a $\delta$ amount of weight of element $\ell$ for element $k$ to obtain the solution $y'$ where

$$y'_k = y_k + \delta \qquad \text{and} \qquad y'_\ell = y_\ell - \delta.$$

Choose $\delta = \min\{x_k - y_k, y_\ell - x_\ell\}$ so that $y'_k = x_k$ or $y'_\ell = x_\ell$ and, therefore, $|\{i \le n : x_i \ne y'_i\}| < m$. To complete the proof, we must show that $y'_1, \ldots, y'_n$ is a valid solution. The difference in the total value of the solutions $y'$ and $y$ satisfies

$$\delta(v_k/w_k) - \delta(v_\ell/w_\ell) = \delta(v_k/w_k - v_\ell/w_\ell).$$

The ordering of the elements guarantees that $v_k/w_k \ge v_\ell/w_\ell$ so that the total value of $y'$ is at least as large as that of $y$ and, therefore, $y'$ is a valid solution. □

What if we now consider the variant of the problem where we are only allowed to choose $x_i \in \{0, w_i\}$? (I.e., where the items are indivisible.) Does the greedy algorithm above still work? You should convince yourself that it is no longer correct—and you should be able to see how the exchange argument we used above fails in this situation! We will revisit this version of the knapsack problem in the next section, when we see how the *dynamic programming* algorithm design technique can be used to solve it efficiently when $W$ is not too large. (And we will see the problem again in the NP-completeness section, when we will see why we shouldn't expect to find an efficient algorithm for this problem in general.)

## 3. Bonus: Another proof of correctness for maximum lateness

The argument we developed earlier to establish the correctness of the GreedyLateness algorithm is not the only possible argument we could use. There is another exchange argument where we use a proof by induction on the number of *inversions* in an ordering of the tasks.

**Definition 7.5** (Inversions). The *number of inversions* between two permutations $\pi$ and $\sigma$ of $\{1, 2, \ldots, n\}$ is

$$\mathrm{inv}(\pi, \sigma) = |\{1 \le i \ne j \le n : \pi(i) < \pi(j) \wedge \sigma(i) > \sigma(j)\}|,$$

the total number of pairs of elements in the sequence that have different relative order in $\pi$ and $\sigma$.

We use two basic facts about the number of inversions.

**Fact 2.** *Two permutations $\pi$ and $\sigma$ satisfy $\mathrm{inv}(\pi, \sigma) = 0$ if and only if $\pi = \sigma$.*

**Fact 3.** *When two permutations $\pi$ and $\sigma$ on $\{1, \ldots, n\}$ satisfy $\mathrm{inv}(\pi, \sigma) \ge 1$, then there must exist an index $k < n$ such that $\pi(k) > \pi(k+1)$ and $\sigma(k) < \sigma(k+1)$.*[2]

It is also convenient for this proof to change the definitions slightly. For a given instance of the problem, redefine the *lateness* of the $k$th task in this order to be $L_k^{(\pi)} := \sum_{i \le k} p_{\pi(i)} - d_{\pi(k)}$. (I.e., the difference is that with this notation $L_k^{(\pi)}$ is the lateness of the $k$th task performed in the order defined by $k$, not the original task $k$.)

[2]Exercise: And prove this statement as well!

**Theorem 7.6.** *The* GreedyLateness *algorithm solves the Maximal Lateness problem.*

*Proof.* Let $\sigma$ be an ordering of $\{1, 2, \ldots, n\}$ that is a valid solution to the maximal lateness problem, and let $\pi$ be the ordering returned by the GreedyLateness algorithm. We prove that $\pi$ is a valid solution by induction on the number of *inversions* between $\pi$ and $\sigma$.

In the base case, when $\text{inv}(\pi, \sigma) = 0$ then $\pi = \sigma$ so $\pi$ is a valid solution.

For the induction step, let us now consider a valid solution $\sigma$ with $\text{inv}(\pi, \sigma) = m \geq 1$ inversions and the induction hypothesis is that if we have a valid solution $\sigma'$ with $\text{inv}(\pi, \sigma') \leq m - 1$ inversions, then $\pi$ is also a valid solution. Let $k < n$ be an index for which $\pi(k) > \pi(k+1)$ and $\sigma(k) < \sigma(k+1)$. The fact that $\pi(k) > \pi(k+1)$ implies that we must have $d_{\sigma(k)} \geq d_{\sigma(k+1)}$. Let $\sigma'$ be the permutation obtained by exchanging the values $\sigma'(k+1) = \sigma(k)$ and $\sigma'(k) = \sigma(k+1)$ and leaving the rest of the values $\sigma'(i) = \sigma(i)$ as in $\sigma$ when $i \notin \{k, k+1\}$.

To complete the proof, we want to show that the maximal lateness of $\sigma'$ is no larger than the maximum lateness of $\sigma$. First, we observe that for every $i < k$ and for every $i > k+1$, we have $L_i^{(\sigma')} = L_i^{(\sigma)}$. So the only latenesses that are different in $\sigma$ and $\sigma'$ are for $i = k$ or $i = k+1$. Define $C = \sum_{i<k} p_{\sigma(i)}$. Then $C = \sum_{i<k} p_{\sigma'(i)}$ as well, so

$$\begin{aligned} L_k^{(\sigma')} &= C + p_{\sigma'(k)} - d_{\sigma'(k)} \\ &= C + p_{\sigma(k+1)} - d_{\sigma(k+1)} \end{aligned}$$

and

$$\begin{aligned} L_{k+1}^{(\sigma')} &= C + p_{\sigma(k)} + p_{\sigma(k+1)} - d_{\sigma(k+1)} \\ &= C + p_{\sigma(k)} + p_{\sigma(k+1)} - d_{\sigma(k)}. \end{aligned}$$

Meanwhile,

$$L_{k+1}^{(\sigma)} = C + p_{\sigma(k)} + p_{\sigma(k+1)} - d_{\sigma(k+1)}.$$

Since $p_{\sigma(k+1)} \geq 0$, then $L_k^{(\sigma')} \leq L_{k+1}^{(\sigma)}$. And since $d_{\sigma(k)} > d_{\sigma(k+1)}$, then $L_{k+1}^{(\sigma')} \leq L_{k+1}^{(\sigma)}$ also. Therefore,

$$\max\left\{L_k^{(\sigma')}, L_{k+1}^{(\sigma')}\right\} \leq L_{k+1}^{(\sigma)} \leq \max\left\{L_k^{(\sigma)}, L_{k+1}^{(\sigma)}\right\}$$

and $\sigma'$ must be a valid solution as well. □

# CS 341: ALGORITHMS (S18) — LECTURE 8
# DYNAMIC PROGRAMMING I

ERIC BLAIS

We now turn our attention to another general technique for designing algorithms: *dynamic programming*. The main idea for this technique can be summarized as follows: We can solve a big problem by

(1) Breaking it up into smaller sub-problems;
(2) Solving the sub-problems from smallest to largest; and
(3) Storing solutions along the way to avoid repeating our work.

We have already seen one example of this algorithm design technique in the first lecture, when we computed Fibonacci numbers. Over the next few lectures, we will explore a range of other problems where dynamic programming is also useful.

## 1. Weighted Interval Scheduling

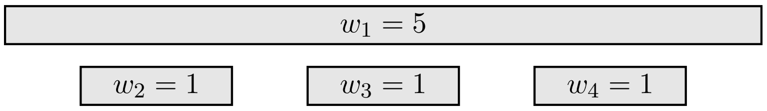
Let's revisit the interval scheduling problem, with a slight twist.

**Definition 8.1** (Weighted interval scheduling)**.** In the *weighted interval scheduling problem*, an instance is a set of $n$ pairs of intervals that have start and finish times $(s_1, f_1), \dots, (s_n, f_n)$ and positive *weights* $w_1, \dots, w_n$. A valid solution is a subset $I \subseteq \{1, 2, \dots, n\}$ of non-overlapping intervals that maximizes $\sum_{i \in I} w_i$.

Without weights, we saw that the greedy algorithm that sorts the intervals by finish time and considers them in that order solves the problem. That's no longer the case when we have weights.

**Proposition 8.2.** *The* GreedyIntervalScheduler *algorithm does not solve the weighted interval scheduling problem.*

*Proof.* There are many instances where the algorithm does not return a valid solution, such as this one.



The valid solution is $I = \{1\}$ but the greedy algorithm outputs $I = \{2, 3, 4\}$ instead. □

A simple variation on the same counter-example shows that a modified greedy algorithm that considers the intervals by order of weights does not solve the problem either. Instead, we need to take a different approach.

As it turns out, the key to solving the weighted interval scheduler problem is to again sort the intervals by finish times and consider them in order. But instead of making greedy decisions, for a fixed instance of the problem let us define and for every $k \leq n$, we define

- $\mathrm{OPT}(k) \subseteq \{1, 2, \ldots, k\}$ is a set of non-overlapping intervals with maximum total weight; and
- $w_{\mathrm{OPT}}(k) = \sum_{j \in \mathrm{OPT}(k)} w_j$ is its weight.

This immediately suggests how to solve the weighted interval scheduler problem using dynamic programming method: we compute $\mathrm{OPT}(1), \mathrm{OPT}(2), \ldots, \mathrm{OPT}(n)$ in that order, and output $\mathrm{OPT}(n)$.

To complete the algorithm, we only need to answer one more question: after we have computed $\mathrm{OPT}(1), \ldots, \mathrm{OPT}(k-1)$ and the weights of all those sets, how can we use those values to compute $\mathrm{OPT}(k)$? If we examine the problem carefully, we realize there are exactly two possibilities to consider:

**Case 1: $k \notin \mathrm{OPT}(k)$:** The first possibility is that the $k$th interval is not in the set $\mathrm{OPT}(k)$. If that's the case, then $\mathrm{OPT}(k) = \mathrm{OPT}(k-1)$ and $w_{\mathrm{OPT}}(k) = w_{\mathrm{OPT}}(k-1)$.

**Case 2: $k \in \mathrm{OPT}(k)$:** Otherwise, if $k \in \mathrm{OPT}(k)$ and we let $j < k$ be the largest value such that $f_j < s_k$, then $\mathrm{OPT}(k)$ can only be a set of non-overlapping intervals if $\mathrm{OPT}(k) \subseteq \{1, 2, \ldots, j, k\}$ so $\mathrm{OPT}(k) = \mathrm{OPT}(j) \cup \{k\}$ and $w_{\mathrm{OPT}}(k) = w_{\mathrm{OPT}}(j) + w_k$.

We can compute the value of both candidates to determine which is correct. The resulting dynamic programming algorithm is as follows.

```
Algorithm 1: WeightedISP($(s_1, f_1), \ldots, (s_n, f_n), w_1, \ldots, w_n$)
(Sort intervals so that $f_1 \le f_2 \le \cdots \le f_n$);
$I[0] \leftarrow \emptyset$;
$W[0] \leftarrow 0$;
$f_0 \leftarrow 0$;
for $k = 1, \ldots, n$ do
    $j \leftarrow \max\{0 \le i < k : f_i < s_k\}$;
    if $W[k-1] > W[j] + w_k$ then
        $I[k] \leftarrow I[k-1]$;
        $W[k] \leftarrow W[k-1]$;
    else
        $I[k] \leftarrow I[j] \cup \{k\}$;
        $W[k] \leftarrow W[j] + w_k$;
return $I(n)$;
```

What is the running time of this algorithm? For the moment, let's assume that we can read, write, and manipulate the values $I[i]$ and $W[i]$ in constant time. Under this assumption, the bottleneck in the above implementation is the search for the value of $j$ in every iteration of the for loop. With the naïve linear search for this value (and with the $\Theta(n \log n)$ time complexity of the sorting operation in the first step), the time complexity of the algorithm is

$$T(n) = \Theta(n \log n) + \sum_{k=1}^{n} \Theta(k) = \Theta(n^2).$$

But there's a better implementation: since the intervals are already sorted by finish times, we can also use binary search to find the value of $j$ insteads. With this implementation, we have the improved running time

$$T(n) = \Theta(n \log n) + \sum_{k=1}^{n} \Theta(\log k) = \Theta(n \log n).$$

But now, as a final step, let us revisit our assumption that we can read, write, and manipulate the values $I[i]$ and $W[i]$ in constant time. Is this assumption reasonable? You should be able to convince yourself that it is when we are considering the weights $W[i]$—but it is *not* reasonable for the sets $I[i]$; in general, storing these sets takes $\Theta(n)$ space and operations on those sets have time complexity $\Omega(n)$. So it would be more accurate to say that the algorithm has time complexity $\Theta(n^2)$. To get a "true" time complexity of $\Theta(n \log n)$, we should figure out if it's possible to modify the algorithm so that it computes the values $W[1], W[2], \ldots, W[n]$ *without* computing the sets $I[1], I[2], \ldots, I[n]$. It is indeed possible to do so; you should try to see how that can be done!

# CS 341: ALGORITHMS (S18) — LECTURE 9
# DYNAMIC PROGRAMMING II: COMPARING STRINGS

ERIC BLAIS

In this lecture, we continue our exploration of the dynamic programming technique by examining problems related to comparisons of strings.

## 1. Longest common subsequence

A *common subsequence* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is a string $z_1, \ldots, z_k$ for which there are indices $1 \le i_1 < i_2 < \cdots < i_k \le m$ and $1 \le j_1 < j_2 < \cdots < j_k \le n$ where for each $\ell \le k$, $z_\ell = x_{i_\ell} = y_{j_\ell}$. For example, in the pair of strings

$$x = \texttt{POLYNOMIAL}$$
$$y = \texttt{EXPONENTIAL},$$

the string $z = \texttt{PONIAL}$ is a subsequence of $x$ and $y$. (As are the strings `POL`, `PNA`, etc.)

**Definition 9.1** (Longest common subsequence)**.** An instance of the *longest common subsequence (LCS)* problem is a pair of strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. The valid solution to an instance is the length of the longest common subsequence of $x$ and $y$.

The natural way to break down the LCS problem into smaller subproblems is to consider the longest common subsequence of prefixes of $x$ and $y$. For $0 \le i \le m$ and $0 \le j \le n$, define

$$M(i,j) = \text{length of LCS of } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When $i$ or $j$ is 0 (which corresponds to $x$ or $y$ being an empty string), then

$$M(0,j) = 0 \qquad \text{and} \qquad M(i,0) = 0.$$

Given that we have computed $M(i-1,j)$, $M(i,j-1)$, and $M(i-1,j-1)$, can we now compute $M(i,j)$? Indeed we can!

$$M(i,j) = \max \begin{cases} M(i-1,j-1)+1 & \text{if } x_i = y_j \\ M(i-1,j) \\ M(i,j-1) \end{cases}$$

This gives the following algorithm.

**Algorithm 1:** LCS($x_1, \ldots, x_m, y_1, \ldots, y_n$)

```
for $i = 1, \ldots, m$ do $M[i,0] = 0$;
for $j = 1, \ldots, n$ do $M[0,j] = 0$;

for $i = 1, \ldots, m$ do
    for $j = 1, \ldots, n$ do
        $M[i,j] = \max\{M[i-1,j], M[i,j-1]\}$;
        if $x_i = y_j$ then $M[i,j] = \max\{M[i,j], M[i-1,j-1]+1\}$;
return $M[m,n]$;
```

We can picture the algorithm as filling out the table of values $M[i,j]$, row by row. With the instance $x =$ `ALGORITHM`, $y =$ `ANALYSIS`, the table looks as follows.

| | ∅ | A | N | A | L | Y | S | I | S |
|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| L | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| G | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| O | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| R | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| I | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| T | 0 | 1 | 1 | . | . | . | . | . | . |
| H | 0 | . | . | . | . | . | . | . | . |
| M | 0 | . | . | . | . | . | . | . | . |

The time complexity of the algorithm is $\Theta(mn)$.

## 2. Edit distance

The length of the longest common subsequence can be interpreted as a measure of how similar two strings are. A more sophisticated measure of the similarity between different strings is known as *edit distance.*

**Definition 9.2** (Edit distance)**.** The *edit distance* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is the minimum number of edit operations required to transform $x$ into $y$, where the 3 possible edit operations are:

**Adding:** a letter to $x$,
**Deleting:** a letter from $x$, and
**Replacing:** a letter in $x$ with another one.

For example, the edit distance between the strings `POLYNOMIAL` and `EXPONENTIAL` is 6, as this is the minimum number of edit operations required to go from one string to the other:

```
--POLYNOMIAL
EXPONEN-TIAL
```

In this example, the full list of operations that transformed `POLYNOMIAL` into `EXPONENTIAL` is as follows.

| | |
|---|---|
| $\texttt{POLYNOMIAL}$ | |
| $\texttt{\underline{E}POLYNOMIAL}$ | (Add E) |
| $\texttt{E\underline{X}POLYNOMIAL}$ | (Add X) |
| $\texttt{EXPO\underline{N}YNOMIAL}$ | (Replace L with N) |
| $\texttt{EXPON\underline{E}NOMIAL}$ | (Replace Y with E) |
| $\texttt{EXPONEN-MIAL}$ | (Delete O) |
| $\texttt{EXPONEN\underline{T}IAL}$ | (Replace M with T) |

A basic computational problem is to find the edit distance between two strings.

**Definition 9.3** (Edit distance problem)**.** An instance of the *edit distance problem* is two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$; the valid solution to an instance is the edit distance between $x$ and $y$.

We can again use the dynamic programming method to solve this problem by considering the subproblems obtained by computing the edit distance between prefixes of $x$ and $y$. For $0 \le i \le m$ and $0 \le j \le n$, define

$$M(i,j) = \text{edit distance between } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When $i = 0$ or $j = 0$, the edit distance is easy to compute: it is exactly the length of the other string. So

$$M(i,0) = i \qquad \text{and} \qquad M(0,j) = j.$$

What about for the other entries? If $x_i = y_j$, then we can match those characters together and we get $M(i,j) = M(i-1,j-1)$. If not, we have three choices:

(1) Replace $x_i$ with $y_j$. In this case, we get $M(i,j) = M(i-1,j-1)+1$.
(2) Delete $x_i$. With this choice, $M(i,j) = M(i-1,j)+1$.
(3) Add the character $y_j$ to the string $x$ right before $x_i$. This choice gives us $M(i,j) = M(i,j-1)+1$.

To compute $M(i,j)$, we want to choose the option among the ones above that has minimum value. So this means that we get

$$M(i,j) = \max \begin{cases} M(i-1,j-1) & \text{if } x_i = y_j \\ M(i-1,j-1)+1 & \text{if } x_i \neq y_j \\ M(i-1,j)+1 \\ M(i,j-1)+1. \end{cases}$$

As long as we compute the values of $M$ in an order where $M(i-1,j-1)$, $M(i-1,j)$, and $M(i,j-1)$ have all been computed before we compute $M(i,j)$, computing this value takes $\Theta(1)$ time. We can again proceed row by row, obtaining an algorithm that looks very similar to the one we used to compute the length of the longest common subsequence.

**Algorithm 2:** EditDistance($x_1, \ldots, x_m, y_1, \ldots, y_n$)

```
for $i = 1, \ldots, m$ do $M[i, 0] = i$;
for $j = 1, \ldots, n$ do $M[0, j] = j$;

for $i = 1, \ldots, m$ do
    for $j = 1, \ldots, n$ do
        if $x_i = y_j$ then
            $r = M[i - 1, j - 1]$;
        else
            $r = M[i - 1, j - 1] + 1$;
        $M[i, j] = \max\{M[i - 1, j] + 1, M[i, j - 1] + 1, r\}$;
return $M[m, n]$;
```

The time complexity of this algorithm is again $\Theta(mn)$.

## 3. Finding the LCS

The LCS algorithm introduced in the first section determines the length of the longest common subsequence between the two strings given as input, but it does not identify the subsequence itself. What if we want to identify it? There are a number of different ways we can modify the algorithm to do so. Or, if we have already computed the matrix $M$ of LCS lengths for all prefixes, we can identify the LCS itself by working backwards from $M(m, n)$.

**Algorithm 3:** PrintLCS($x$, $y$, $M$, $i$, $j$)

```
if $i > 1$ and $M[i, j] = M[i - 1, j]$ then
    PrintLCS($x$, $y$, $M$, $i - 1$, $j$);
else if $j > 1$ and $M[i, j] = M[i, j - 1]$ then
    PrintLCS($x$, $y$, $M$, $i$, $j - 1$);
else
    /* We must have matched $x_i = y_j$                    */
    PrintLCS($x$, $y$, $M$, $i - 1$, $j - 1$);
    Print $x_i$;
```

Calling PrintLCS($x$, $y$, $M$, $m$, $n$) will print the longest common subsequence of $x$ and $y$. Interestingly, by calling it with any other $i \leq m$ and $j \leq n$ we can also print the longest common subsequence of any prefixes of $x$ and $y$ just as efficiently.

A similar idea can be used to output a minimum set of edit operations that transform the string $x$ to the string $y$ after the EditDistance algorithm has been run. The details are left as an exercise.

# CS 341: ALGORITHMS (S18) — LECTURE 10
# DYNAMIC PROGRAMMING III: COINS AND KNAPSACKS

ERIC BLAIS

In this lecture, we complete our exploration of the dynamic programming technique by revisiting problems related to coins and knapsacks.

## 1. Making change

Recall the Making Change problem that we saw at the beginning of Lecture 6:

**Problem 1** (Making change)**.** *Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $V \geq 1$, determine the minimum number of coins whose denominations sum to $V$.*

We saw that there is a simple greedy algorithm that correctly solves the problem for some, but not all, coin denominations. We can use the dynamic programming method to solve the problem correctly for all coin denominations.

### 1.1. Dynamic programming algorithm.

The subproblems for the problem correspond to making change for values $v = \{0, 1, 2, \ldots, V\}$. Define $C(v)$ to be the minimum number of coins required to make change for value $v$. Then $C(0) = 0$ and for every $v \geq 1$, we can try returning a coin of every possible denomination so that

$$C(v) = \min_{i=1,\ldots,n : d_i \leq v} \big\{1 + C(v - d_i)\big\}.$$

The resulting algorithm is as simple as the greedy algorithm, but not as efficient.

**Algorithm 1:** CoinChange$(d_1, \ldots, d_n, V)$

```
$C[0] \leftarrow 0$;
for $v = 1, \ldots, V$ do
    $C[v] \leftarrow \min_{i \leq n \,:\, d_i \leq v} \big\{1 + C[v - d_i]\big\}$;
return $C[V]$;
```

**Theorem 10.1.** *The* CoinChange *algorithm solves the making change problem.*

*Proof.* Let $\mathrm{OPT}(v)$ denote the minimum number of coins required to return value $v$ in coins with some fixed denominations $d_1, \ldots, d_n$. To establish the correctness of the algorithm, we prove that $C[v] = \mathrm{OPT}(v)$ by induction on $v = 0, 1, 2, \ldots, V$. In the base case, $\mathrm{OPT}(0) = C[0] = 0$ since zero coins are required to make change of value 0.

For the induction step, assume that $C[0] = \mathrm{OPT}(0)$, $C[1] = \mathrm{OPT}(1)$, $\ldots$, $C[v-1] = \mathrm{OPT}(v-1)$. Consider now a minimum set of coins with total value $v \geq 1$. This set includes a coin of denomination $d_k$, $1 \leq d_k \leq v$, for some $k \in \{1, 2, \ldots, n\}$. This means that $\mathrm{OPT}(v) = 1 + \mathrm{OPT}(v - d_k)$ and that for each $i \neq k$, $\mathrm{OPT}(v) \leq 1 + \mathrm{OPT}(v - d_i)$. By the induction hypothesis, we then have

$$C[v] = \min_{i : d_i \leq v} \{1 + C[v - d_i]\} = \min_{i : d_i \leq v} \{1 + \mathrm{OPT}(v - d_i)\} = 1 + \mathrm{OPT}(v - d_k) = \mathrm{OPT}(v). \quad \square$$

**Theorem 10.2.** *The time complexity of the* CoinChange *algorithm is $\Theta(nV)$.*

*Proof.* The for loop is run $V$ times, and for each loop iteration we compare up to $n$ values, so the time complexity of the algorithm is $O(nV)$. Furthermore, when $V \geq 2d_1$, then for $V/2$ iterations of the loops, all $n$ denominations satisfy the condition $d_i \leq v$ so that on those instances the algorithm has time complexity at least $\frac{V}{2} \cdot n = \Omega(nV)$. □

1.2. **Decision problem variant.** If we don't have a coin with denomination 1, then we can't make change for all possible values. In this case, it makes sense to ask whether we can make change that adds up to some value $V$ or not.

**Problem 2** (Making change—decision variant)**.** *Given coins with denominations $d_1 > d_2 > \cdots > d_n > 1$ and a value $V \geq 1$, determine whether it is possible to choose some coins whose denominations sum to $V$.*

Dynamic programming can be used to solve this problem. In fact, the algorithm we designed for the original version of the making change problem can easily be modified to solve this problem as well.

**Algorithm 2:** CoinChangeDec$(d_1, \ldots, d_n, V)$

```
$C[0] \leftarrow$ True;
for $v = 1, \ldots, V$ do
    $C[v] \leftarrow$ False;
    for $i = 1, \ldots, n$ do
        if $d_i \leq v$ then $C[v] = C[v] \vee C[v - d_i]$;
return $C[V]$;
```

**Theorem 10.3.** *The* CoinChangeDec *algorithm solves the decision variant of the making change problem.*

*Proof.* Let CanChange$(v)$ be `True` if it is possible to choose coins of some fixed denominations $d_1, \ldots, d_n$ with total value $v$; `False` otherwise. To establish the correctness of the algorithm, we prove that $C[v] = \text{CanChange}(v)$ by induction on $v = 0, 1, 2, \ldots, V$. In the base case, $\text{CanChange}(0) = C[0] =$ `True` since an empty set of coins has total value 0.

For the induction step, the induction hypothesis states that $C[v'] = \text{CanChange}(v')$ for all $v' < v$. We consider two cases.

(1) If CanChange$(v)$ is `True`, then there exists a non-empty set $S$ of coins with total value $v$, and we can let $d_k$ be the denomination of one of the coins in this set. Then CanChange$(v - d_k)$ is `True` (by removing a coin of denomination $d_k$ from $S$) and by the induction hypothesis $C[v - d_k] =$ `True` as well, so

$$C[v] = \bigvee_{i \leq n \,:\, d_i \leq v} C[v - d_i] = \texttt{True} = \text{CanChange}(v).$$

(2) If CanChange$(v)$ is `False`, then for any denomination $d_i$ we must have that CanChange$(v - d_i)$ is also `False`, otherwise we would take the set $S$ of coins that make change for $v - d_i$ and add a coin of denomination $d_i$ to make change for $v$. So by the induction hypothesis,

$$C[v] = \bigvee_{i \leq n \,:\, d_i \leq v} C[v - d_i] \bigvee_{i \leq n \,:\, d_i \leq v} \text{CanChange}(v - d_i) = \texttt{False} = \text{CanChange}(v). \quad \square$$

## 2. Knapsack

We saw a variant of the knapsack problem where we were allowed to divide items and only include a fraction of them in our knapsack. In the standard version of the problem, we no longer have that power: we either include or exclude an item in the knapsack.

**Definition 10.4** (Knapsack)**.** An instance of the *knapsack problem* is a set of $n$ items that have positive integer weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, as well as a maximum weight capacity $W$ of the knapsack. A valid solution to the problem is a subset $S \subseteq \{1, 2, \dots, n\}$ of the items that you put in your backpack that satisfies $\sum_{i \in S} w_i \le W$ and maximizes the total value $V = \sum_{i \in S} v_i$ among all sets that satisfy the weight condition.

To distinguish this problem explicitly from the fractional knapsack problem, it is also sometimes called the 0-1 *knapsack* problem.

We can solve the knapsack problem using the dynamic programming technique. Let's consider the natural way to do this, following the approach that we used in previous lectures. A natural way to break down the problem into smaller subproblems is to consider only the items $1, \dots, k$ for each $k \in \{1, 2, \dots, n\}$ (along with the trivial subproblem when $k = 0$).

Then, as in the other problems we consider, we have a simple observation that can let us solve the subproblem with the first $k$ items when we already solved it with the first $k-1$ items: either $k$ is in the optimal subset $S_k \subseteq \{1, 2, \dots, k\}$ of items we put in the knapsack, or it is not. If it is not, then the optimal value $V_k = V_{k-1}$. But if it is, we realize that there is a twist that we need to consider: we need to find the maximum subset $S'_{k-1} \subseteq \{1, 2, \dots, k-1\}$ that fit in a knapsack with capacity $W - w_k$ (not $W$!) if we are to put these items into the knapsack along with item $k$.

Therefore, we need to consider subproblems where we consider the first $k$ elements *and* where we fix the capacity of a knapsack to be $w$, for $k \in \{0, 1, 2, \dots, n\}$ and for $w = \{0, 1, 2, \dots, W\}$. We do so by defining

$$M(k, w) = \max_{S \subseteq \{1,2,\dots,k\} : \sum_{i \in S} w_i \le w} \sum_{i \in S} v_i.$$

Then for every $w \le W$ and $k \le n$,

$$M(0, w) = 0 \qquad \text{and} \qquad M(k, 0) = 0.$$

For $k \ge 1$, we then have two possibilities: either $w_k > w$, in which case the item $k$ does not fit into the knapsack and $M(k, w) = M(k-1, w)$, or $w_k \le w$ in which case $M(k, w)$ is the maximum of the optimal value $M(k-1, w)$ obtained by leaving out item $k$ and the optimal value $v_k + M(k-1, w - w_k)$ obtained by including item $k$ in the knapsack. So for each $k = \{1, 2, \dots, n\}$ we have

$$M(k, w) = \begin{cases} M(k-1, w) & \text{if } w_k > w \\ \max\{M(k-1, w), v_k + M(k-1, w - w_k)\} & \text{if } w_k \le w. \end{cases}$$

The resulting algorithm is as follows.

**Theorem 10.5.** *The* Knapsack *algorithm solves the knapsack problem.*

*Proof.* Let $\mathrm{OPT}(k, w)$ denote the maximum value of a subset of items $1, \dots, k$ that fit into a knapsack with capacity $w$. We show $M[k, w] = \mathrm{OPT}(k, w)$ for all $k = 0, 1, \dots, n$ and $w = 0, 1, \dots, W$ by induction on $k$ and $w$. In the base cases, when $k = 0$ or $w = 0$, then $M[k, w] = 0 = \mathrm{OPT}(k, w)$.

**Algorithm 3:** KNAPSACK($w_1, \ldots, w_n, v_1, \ldots, v_n, W$)

```
for $w = 0, 1, 2, \ldots, W$ do $M[0, w] \leftarrow 0$;
for $k = 0, 1, 2, \ldots, n$ do $M[k, 0] \leftarrow 0$;

for $k = 1, \ldots, n$ do
    for $w = 1, \ldots, W$ do
        if $w_k \leq w$ then
            $M[k, w] \leftarrow \max\{M[k-1, w], v_k + M[k-1, w - w_k]\}$;
        else
            $M[k, w] \leftarrow M[k-1, w]$;
return $M[n, W]$;
```

For the induction step, the induction hypothesis lets us assume that $M[k-1, w'] = \mathrm{OPT}(k-1, w')$ for every $w' \leq w$. Consider now the value $\mathrm{OPT}(k, w)$and a corresponding set $S \subseteq \{1, 2, \ldots, k\}$ of items with total weight at most $w$ and value $\mathrm{OPT}(k, w)$. There are two cases to consider.

(1) If $w_k > w$, then it must be that $k \notin S$ so that $\mathrm{OPT}(k, w) = \mathrm{OPT}(k-1, w)$ and by the induction hypothesis
$$M[k, w] = M[k-1, w] = \mathrm{OPT}(k-1, w) = \mathrm{OPT}(k, w).$$
(2) If $w_k \leq w$, then $\mathrm{OPT}(k, w) = \mathrm{OPT}(k-1, w)$ (if $k \notin S$) or $\mathrm{OPT}(k, w) = v_k + \mathrm{OPT}(k-1, w - w_k)$ (if $k \in S$), whichever is larger. So by the induction hypothesis
$$\begin{aligned} M[k, w] &= \max\{M[k-1, w], v_k + M[k-1, w - w_k]\} \\ &= \max\{\mathrm{OPT}(k-1, w), v_k + \mathrm{OPT}(k-1, w - w_k)\} = \mathrm{OPT}(k, w). \end{aligned}$$ □

**Theorem 10.6.** *The time complexity of the* KNAPSACK *algorithm is* $\Theta(nW)$.

*Proof.* The code inside the two nested for loops is run a total of $nW$ times and has complexity $\Theta(1)$. The two initialization for loops have time complexity $\Theta(W)$ and $\Theta(n)$, respectively. So the total time complexity of the algorithm is $\Theta(nW + n + W) = \Theta(nW)$. □

If we want to find the optimal set of items to put in the knapsack, we can again use the backtracking approach to find which items to put in the knapsack based on the $M[k, w]$ values.

## 3. Pseudopolynomial-time algorithms

We will spend more time exploring polynomial-time algorithms later in the course, but it's worth pausing here to ask whether the algorithms we introduced in this lecture are polynomial-time or not. To answer this question, we need to start with a formal definition.

**Definition 10.7** (Polynomial-time algorithm)**.** A *polynomial-time algorithm* is an algorithm $A$ with time complexity that is polynomial in the length of its input.

The key part of the definition is the phrase *the length of its input*: we usually consider the time complexity of an algorithm in terms of $n$, with $n$ representing different parameters for different problems, but it's not sufficient to show that an algorithm has time complexity polynomial in $n$ to show that it is a polynomial-time algorithm—unless we also show that the input size is polynomial in $n$.

Is that the case in the Making Change and the Knapsack problems? NO! That's because the values $V$ and $W$ require only $\log V$ and $\log W$ bits to specify in the inputs to the problem, respectively. So while the time complexity of both algorithms is polynomial in $n$, it is also *exponential* in the length of the other parameter ($V$ or $W$). These algorithms are two examples of *pseudopolynomial-time algorithms.*

Can we obtain a polynomial-time algorithm for, say, the Knapsack problem? Such an algorithm would have time complexity $O(n^c(\log W)^{c'})$ for some constants $c, c'$. As we will see in the NP-completeness section of the course, obtaining such an algorithm would be a huge breakthrough, as it would solve the infamous P vs. NP problem and show that *every* problem in NP also can be solved by a polynomial-time algorithm.

## 4. Bonus: Memoization

All the dynamic programming algorithms we have introduced in this course follow the same pattern of solving the subproblems from smallest to largest, in that order. This is sometimes called "bottom-up dynamic programming", and there is another way to implement essentially all the dyanmic programming algorithms we have seen in a "top-down" fashion using recursve algorithms. What distinguishes the dynamic programming recursive algorithms from the usual ones is that the algorithm stores a global array of values that it computes along the way, so that recursive calls are made only when the value has not already been computed earlier. This top-down dynamic programming technique is known as *memoization.*

For example, the coin change algorithm we described above can be implemented using the memoization technique as follows.

**Algorithm 4:** CoinChangeRec($v$)

```
/* $C[0..V]$ is a global array of precomputed values                */
/* Initially, $C[i] = \perp$ for all $i \in 1, 2, \ldots, V$ and $C[0] = 0$.  */
/* $d_1, \ldots, d_n$ are also stored globally                      */
if $C[v] = \perp$ then
    $C[v] \leftarrow \min_{i \le n \,:\, d_i \le v} \{1 + \text{CoinChangeRec}(v - d_i)\}$;
return $C[v]$;
```

In most settings, for the types of examples we have covered in this course, the bottom-up dynamic programming technique is preferable: it is usually simpler to analyze (both for correctness and for time complexity) and the implementations of those algorithms usually outperform their memoization analogues in practice because of lack of overhead with recursion and because of low-level data-access pattern optimizations. But there are settings when the top-down approach can be better—for example if you only need to precompute a small fraction of the subproblems instead of all of them (and it's not clear from the instance which subproblems need to be solved).