INT202
Complexity of Algorithms
Fundamental Techniques

XJTLU/SAT/INT

SEM2 AY2020-2021

Optimization Problems

➢ Optimization problems can have many possible solutions, and each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

➢ Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.

➢ A greedy algorithm is a process that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Greedy Method

➢ A word of warning! The greedy approach does *not always* lead to an optimal solution.

➢ Problems that have a greedy solution are said to possess the greedy-choice property.

➢ The greedy method is also used for some hard (i.e. difficult to solve) problems in order to generate *approximate solutions*.

Greedy Method (The Knapsack Problem)

Let S be a set of n items, where each item i has a positive weight w_i and a positive benefit b_i.

Goal: Find the maximum benefit subset that does not exceed a given total weight (capacity) W.

The Greedy method does not give you the optimal solution of this problem!

Try this out: (3, 7); (4, 9); (5, 9); (2, 2) for a maximum weight of 6.

(2, 2); (3, 7); (4, 9); (5, 9);

Selection:　1　　　1　　　0　　　0

Fractional Knapsack Problem (FKP)

Goal: Find the *maximum benefit* subset that does not exceed total weight W.

In an FKP we are allowed to take *arbitrary fractions*, x_i, of each item, i.e. the solution is a set of values x_i such that

$$0 \leq x_i \leq w_i \text{ for all } i, \text{ and}$$

$$\sum_{i \in S} x_i \leq W.$$

The total benefit of the items taken is determined by the sum

$$\sum_{i \in S} b_i(x_i / w_i).$$

Fractional Knapsack Problem (cont.)

In the solution to an FKP, we use a *heap-based* priority queue to store items of *S*, where the *key* of each item is its *value index* b_i/w_i

In this case, the heap maintains the *maximum* value at the root (max-heap).

Object:	1	2	3	4
Benefit:	7	9	9	2
Weight:	3	4	5	2
Value index:	2.33	2.25	1.8	1

Fractional Knapsack Problem (cont.)

FKP satisfies the *greedy-choice property*, hence:

➢ Obeying a greedy strategy, each time one can begin by taking as much as possible of the item with the greatest value index.

➢ With a heap, each greedy choice (which removes an item with the greatest value index) requires $O(\log n)$ time.

➢ If the supply of that item is exhausted and but can still carry more, one take as much as possible of the item with the next greatest value index, and so forth, until he reaches the weight limit W. Thus, by sorting the items by value index, this greedy algorithm runs in $O(n\log n)$ time.

FRACTIONAL KNAPSACK(S, W)

// Input: Set S of items, item i has weight w_i and benefit b_i,
maximum total weight W.
// Output: Amount x_i of each item to maximize the total benefit.

```
1   for i ← 1 to |S| do
2              x_i ← 0
3              v_i ← b_i/w_i
4              Insert (v_i, i) into a heap H (max value index at root).
5   w ← 0
6   while w < W do
7              Remove the max value from H.
8              a ← min{w_i, W − w}
9              x_i ← a
10             w ← w + a
```

Interval Scheduling

We now consider a problem sometimes referred to as the *Interval Scheduling Problem*.

Here we have a collection of tasks (or intervals of requested time to use a room, etc.), and a single machine that can process these tasks (or a single room in which meetings can be held, etc.).

Goal: We want to select a subset of the tasks in order to maximize the *number* of tasks that we can schedule on the machine.

Interval Scheduling (cont.)

More formally (and keeping with the task/machine terminology), suppose we are given a set T of n tasks, where each task i has a start time s_i and a finish (or completion) time f_i

Two tasks *i* and *j* are *non-conflicting* (or *compatible*) if

$$f_i \leq s_j \ \text{ or } \ f_j \leq s_i.$$

So two tasks can be executed on the machine only if they are *non-conflicting*.

Goal: Select a subset of non-conflicting tasks having *maximum size (number of tasks)*.

Interval Scheduling (cont.)

Can we find a greedy algorithm for finding this solution?

There are several methods we might happen to try in our search for a method that works.

Suppose, for example, we always pick the first available task with the *earliest start time*.
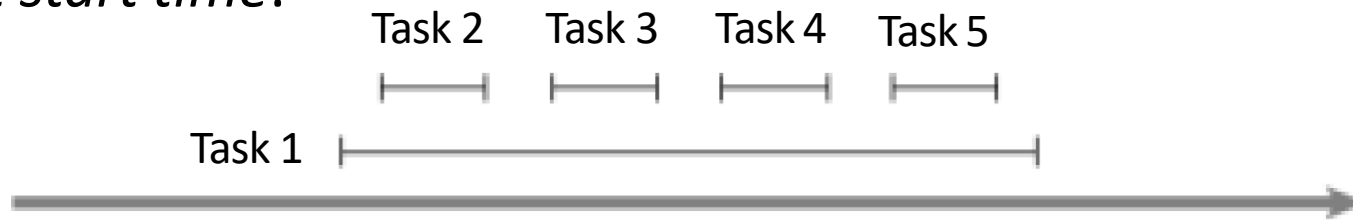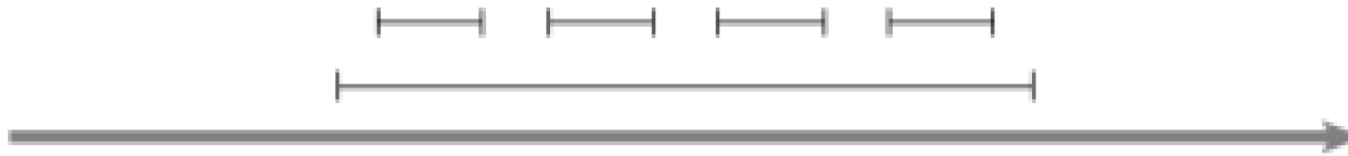

In this case our greedy method would accept a single request, while the optimal solution could accept many

counter-example: S={1,2,3,4,5}; $S\_1=0$, $f\_1=20$; $S\_2=1$, $f\_2=4$; $S\_3=6$, $f\_3=10$; $S\_4=12$, $f\_4=15$; $S\_5=16$, $f\_5=19$.

Interval Scheduling (cont.)

Can we find a greedy algorithm for finding this solution?

There are several methods we might happen to try in our search for a method that works.


In this case our greedy method would accept a single request, while the optimal solution could accept many

Suppose, for example, we always pick the first available task with the *earliest start time*.

This idea doesn't work, so we might try to *accept "short" intervals first*, i.e. take ones for which $f_i - s_i$ is smallest first.

Interval Scheduling (cont.)

Suppose, for example, we always pick the *"short" intervals first*.

Task 1

Task 3

Task 2
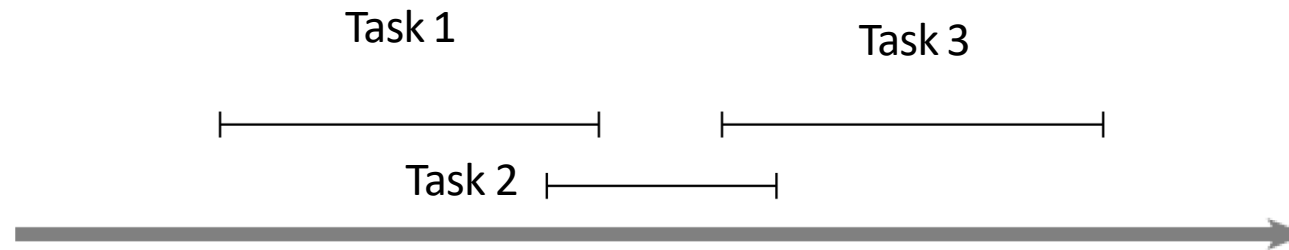
The short interval in the middle would prevent us from accepting the other two
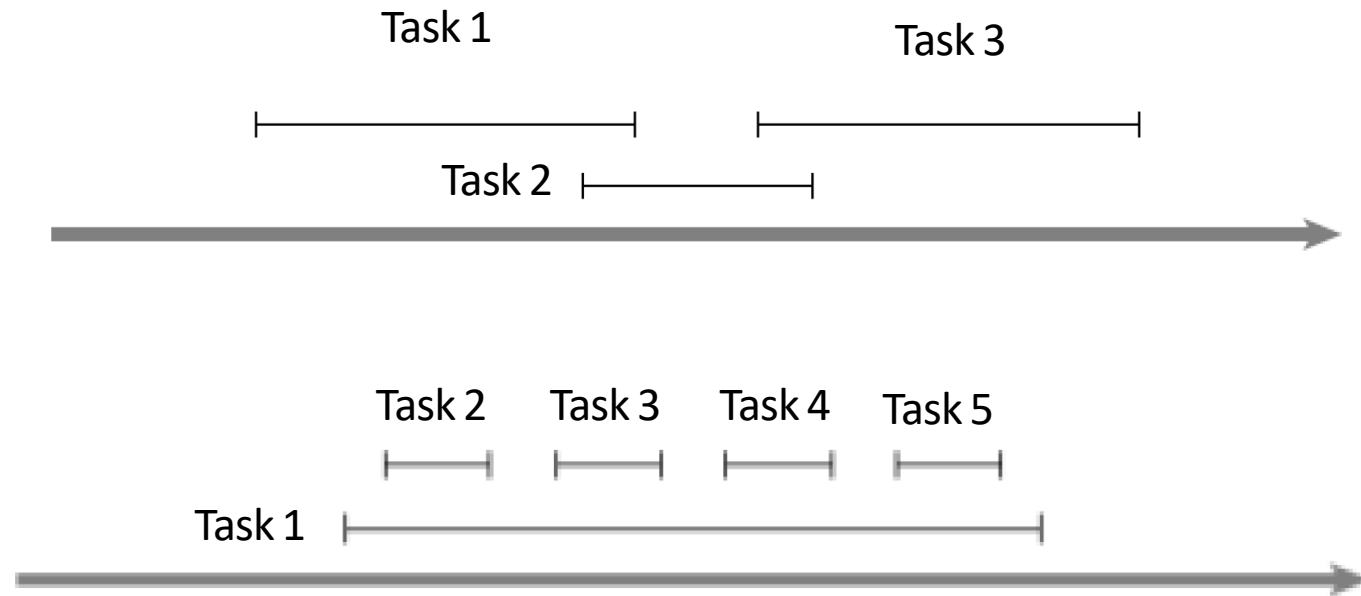
Accept "short" intervals first, i.e. take ones for which $f_i - s_i$ is smallest first.

This idea doesn't work!

How about accepting the task that finishes first? In other words, we want to "free up" the machine as soon as possible to start another task.

Interval Scheduling (cont.)

Suppose, for example, we always accepting the task that finishes first.

Task 1

Task 3

Task 2

Task 2   Task 3   Task 4   Task 5

Task 1

How about accepting the task that finishes first? In other words, we want to "free up" the machine as soon as possible to start another task.

Interval Scheduling (cont.)

intervals numbered in order

Selecting interval 1

Selecting interval 3

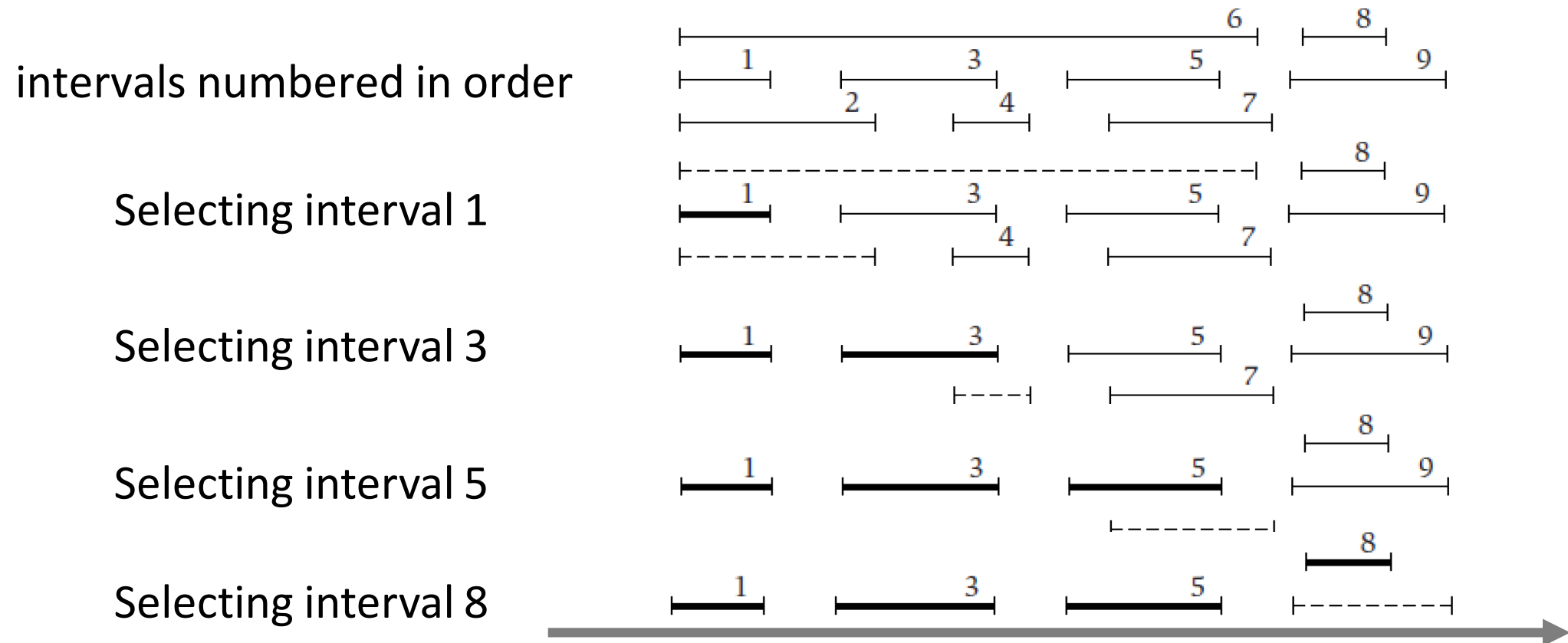Selecting interval 5

Selecting interval 8

Figure Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

Interval Scheduling (cont.)

➢ Order the tasks in terms of their completion times. Then select the task that finishes first, remove all tasks that conflict with this one, and repeat until done.

➢ Furthermore, we can make this algorithm run in time O(nlogn) (the main time will lie in sorting the tasks by their completion times).
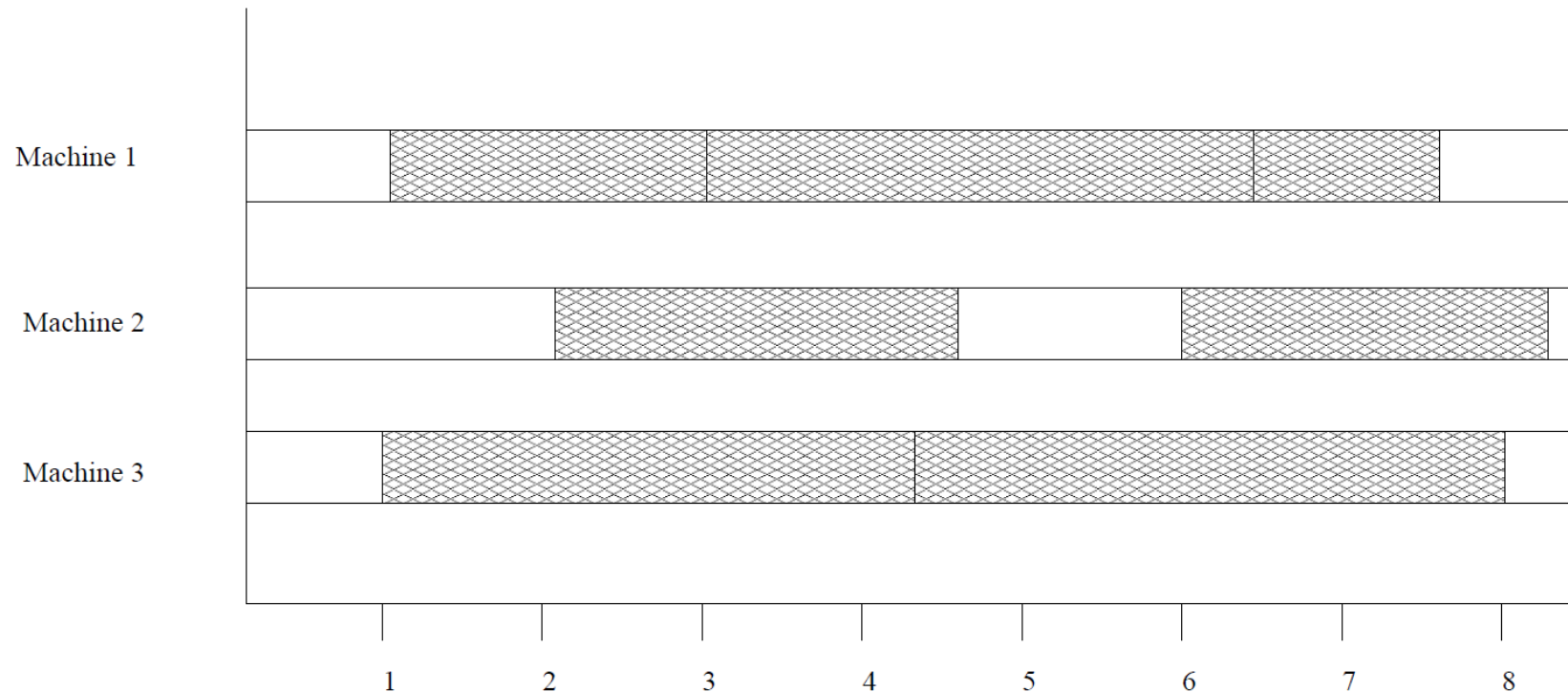
INTERVALSCHEDULE(T)

▷ Input: A set, T, of tasks with start times and end times.
▷ Output: A maximum-size subset, A, of non-conflicting tasks.
1   $A \leftarrow \emptyset$
2   **while** $T \neq \emptyset$
3   **do**
4   Remove the task t with earliest completion time.
5   Add t to A
6   Delete all tasks from T that conflict with t
7   Return the set A as the set of scheduled tasks

Task Scheduling

➢ Now we consider a different, yet related problem. Suppose we still have a set T of n tasks with start and completion times as before.

➢ Now we want to schedule all of the tasks using as few machines as possible (in a non-conflicting way).

➢ In this case, a greedy algorithm will still work to solve this problem.

- This time we consider the tasks ordered by their start times.

- Then for each task i, if we have the machine that can handle task i, it is scheduled on that machine.

- Otherwise, we allocate a new machine, schedule task i on it, and repeat the greedy selection process until we have considered all tasks in T.

TASKSCHEDULE(T)

    ▷ Input: A set, T, of tasks with start times and end times.
    ▷ Output: A non-conflicting schedule of the T tasks.
1 $m \leftarrow 0$
2 **while** $T \neq \emptyset$
3   **do**
4   REMOVEMIN(T)
5   **if** $\exists$ machine j with no conflicts
6   **then** SCHEDULE(i, j)
7   **else**
8            $m \leftarrow m + 1$
9   SCHEDULE(i, m)

Dynamic Programming

➢ The *Dynamic Programming* technique is similar to divide-and-conquer.

➢ The main difference is in replacing (possibly) repetitive recursive calls by a reference to already computed values stored in a *special table*.

➢ Dynamic programming is primarily used for optimization problems. It is often applied where a brute force search for optimal value is *infeasible*.

➢ However, dynamic programming is efficient only if the problem has a certain amount of structure that can be exploited.

Dynamic Programming (cont.)

Hallmarks:

➢ Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems

➢ Overlapping subproblems: few subproblems in total, many recurring instances of each (a recursive algorithm revisits the same problem repeatedly)

Basic idea:

➢ Solve bottom-up, building a table of solved subproblems that are used to solve larger ones

Variations:

➢ "Table" could be 3-dimensional, a tree, etc.

The $\{0-1\}$ Knapsack Problem

The $\{0, 1\}$ *Knapsack Problem* is the knapsack problem where taking fractions of items is *forbidden*.

Recall that there is a set, *S*, of *n* items where item *i* has benefit b_i and an integer weight w_i.

We have the following objective: find a subset $T \subseteq S$ that

$$\text{maximizes} \sum_{i \in T} b_i$$

$$\text{subject to} \sum_{i \in T} w_i \leq W$$

The $\{0-1\}$ Knapsack Problem

Exponential solution: We can "easily" solve the $\{0, 1\}$ knapsack problem in $O(2^n)$ time by *testing all possible* subsets of the *n* items.

Unfortunately, *exponential complexity* is unacceptable for large *n* and we have to focus on nice characterizations for sub-problems in order to use dynamic programming.

The $\{0-1\}$ Knapsack Problem

Let $S_k = \{i \mid i = 1, 2, \ldots, k\}$ denotes a set containing the first k items, and define $S_0 = \emptyset$.

Let $B[k, w]$ be the *maximum total benefit* obtained using a subset of S_k, and having total weight *at most w*.

Then we have $B[0, w] = 0$, for each $w \leq W$, and

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w < w_k \\ \max\{B[k-1, w], b_k + B[k-1, w-w_k]\} & \text{otherwise} \end{cases}$$

The $\{0 - 1\}$ Knapsack Problem

Given an integer W and a set S of n items, each of which has a positive benefit and a positive integer weight, we can find the highest benefit subset of S with total weight at most W in *O(nW)* time.

Solve the following instance of the 0-1 Knapsack Problem with four items where the maximum allowed weight is *W* = 10.

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

The {0 − 1} Knapsack Problem

Using the given algorithm, we can build up the following table:

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w < w_k \\ \max \{B[k - 1, w], b_k + B[k - 1, w - w_k]\} & \text{otherwise} \end{cases}$$

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	25	25	25	25
2	0	0	15	15	15	15	15	25	25	40	40
3	0	0	15	20	20	35	35	35	35	40	45
4	0	0	15	20	20	35	36	36	51	56	56

The solution is to pick up items 3 and 4 for a maximum benefit of 56.