Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

Knapsack problem

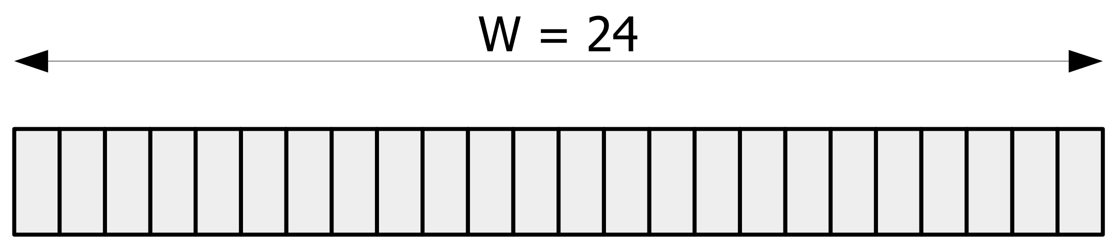
prerequisites:

<none>

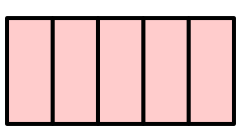
Problem statement

Let's start this time from problem definition.

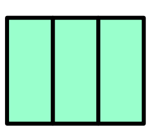
We have a knapsack of capacity **W**:



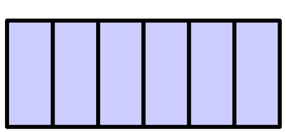
And we have several items, each having its weight **w[i]**, and bonus **b[i]**:



$\mathbf{w_1 = 5, b_1 = 11}$



$\mathbf{w_2 = 3, b_2 = 4}$



$\mathbf{w_3 = 6, b_3 = 12}$

Problem statement

Our task is to place some items in the knapsack in such a way that:

$$\sum w_i \leq W$$

$$\sum b_i = B \rightarrow max$$

This two constraints have quite natural meaning:

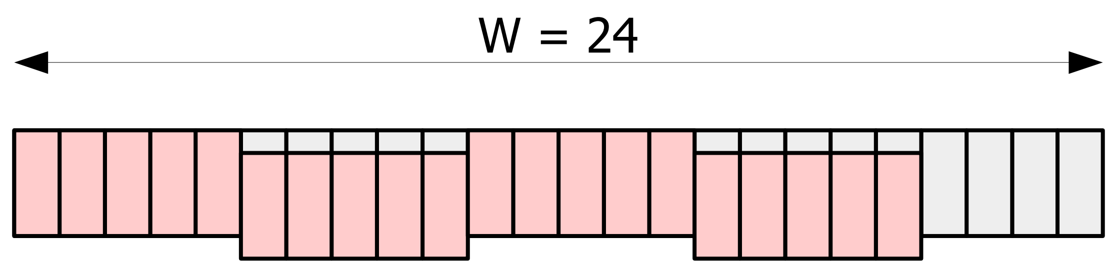
- First means that all the picked items should fit in the knapsack,
- Second means that we want to take with ourselves as larger bonus as possible.

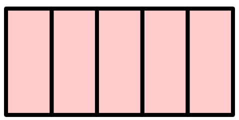
Problem statement

Let's try to figure out solution for the presented example.

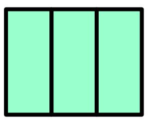
Try 1:

$B = 4*b_1 = 44$

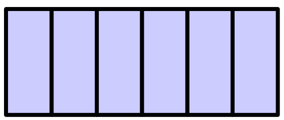




$w_1 = 5, b_1 = 11$



$w_2 = 3, b_2 = 4$



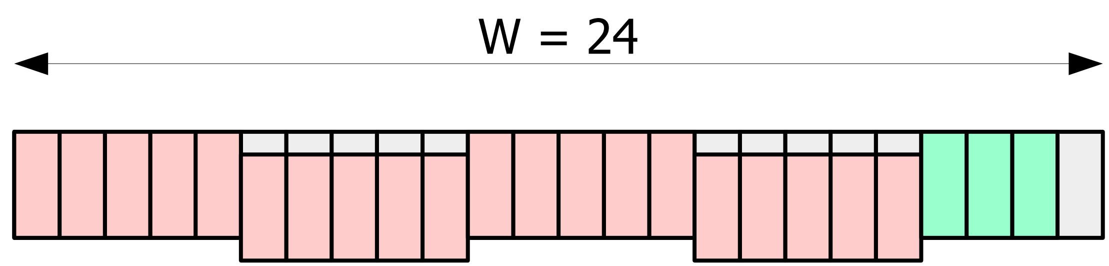
$w_3 = 6, b_3 = 12$

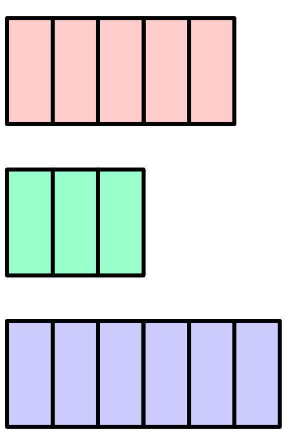
... not optimal.

Problem statement

Try 2:

$B = 4 * b_1 + b_2 =$
$= 44 + 4 =$
$= 48$





$w_1 = 5, b_1 = 11$

$w_2 = 3, b_2 = 4$

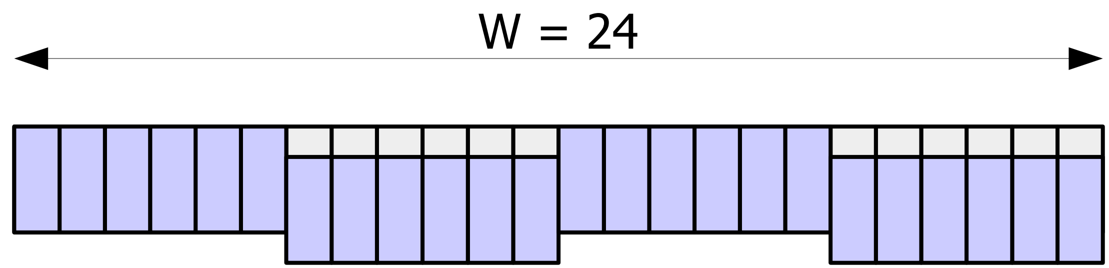
$w_3 = 6, b_3 = 12$

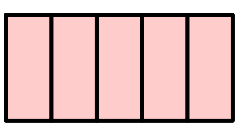
... not optimal.

Problem statement

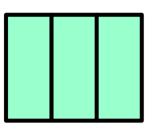
Try 3:

$B = 4 * b_3 = 48$

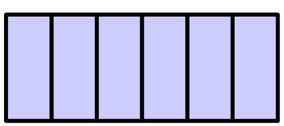




$w_1 = 5, b_1 = 11$



$w_2 = 3, b_2 = 4$



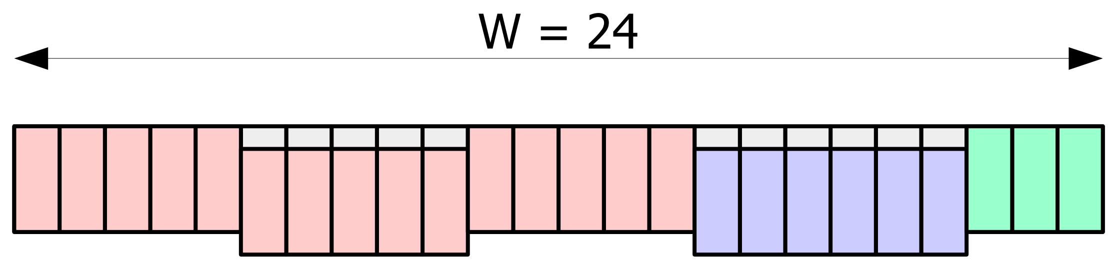
$w_3 = 6, b_3 = 12$

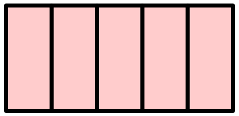
... not optimal.

Problem statement

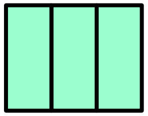
Try 4:

B =
$$= 3 * b_1 + b_3 + b_2$$
$$= 33 + 12 + 4$$
$$= 49$$

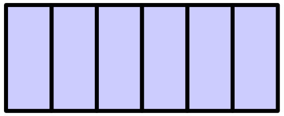




$w_1 = 5, b_1 = 11$



$w_2 = 3, b_2 = 4$



$w_3 = 6, b_3 = 12$

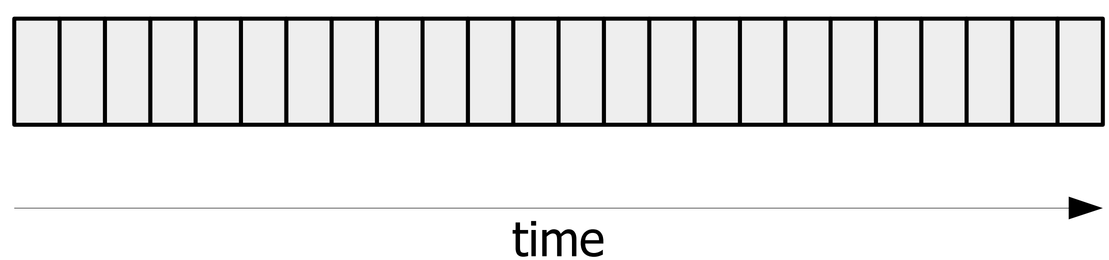
... optimal.

Usage

There are many practical applications of the knapsack problem:

Usage 1 – time scheduling:

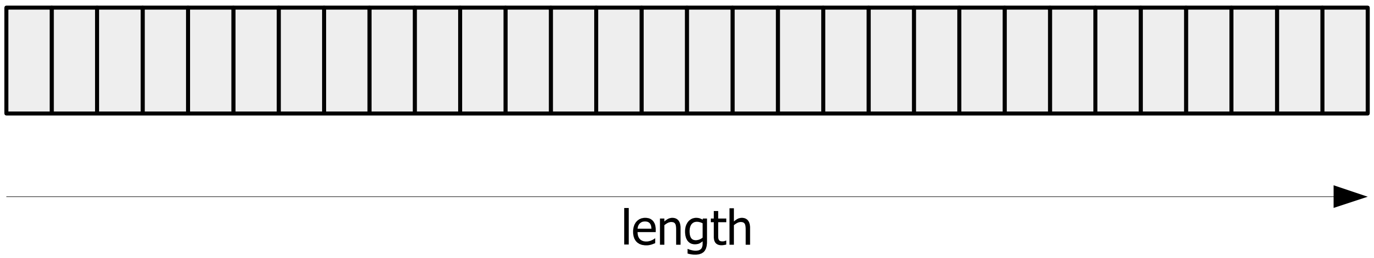
- We have some time allocated, and tasks which can be done during it,
- Each task has its duration, and benefit that we will receive, if completed.
- We need to do those tasks, which will give us maximal benefit.



Usage

Usage 2 – packaging:

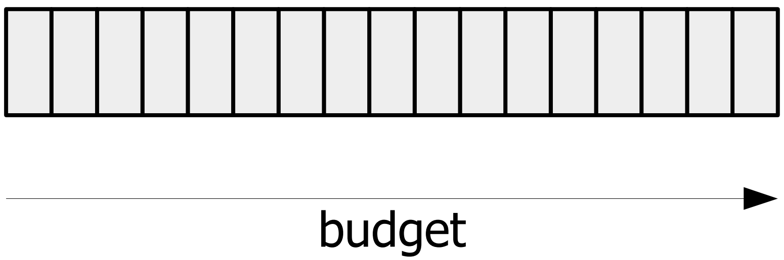
- When we have a carrier with certain capacity, and objects which should be placed there,
- Each object has its size and its price,
- We want to pick those objects, which will maximize the price.



Usage

Usage 3 – budget planning:

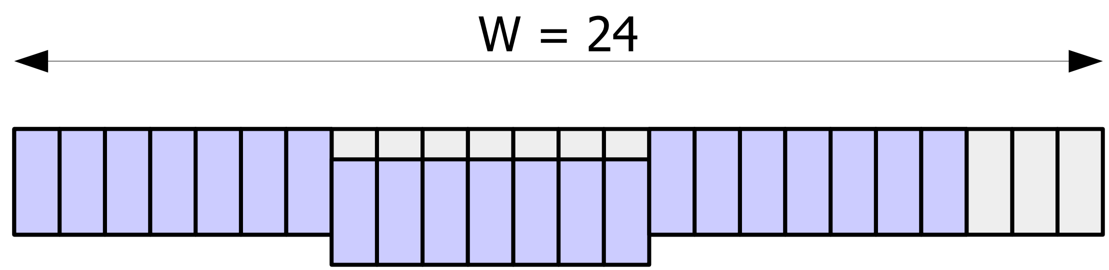
- When we have an allocated budget, and possible spendings,
- Each spending has its price, and probability to succeed,
- We want to spend our budget on such spendings, which together will maximize our success probability.



The difficulty of KP

Why it is <u>not easy</u> to find the solution?

Approach 1) - placing the item with <u>maximal bonus first</u>.

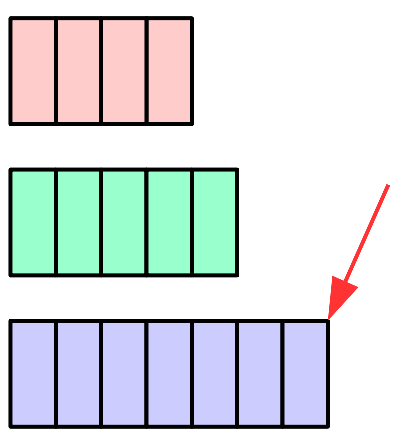


B = 3*10 = 30

$(w_1, b_1) = (4, 6)$

$(w_2, b_2) = (5, 9)$

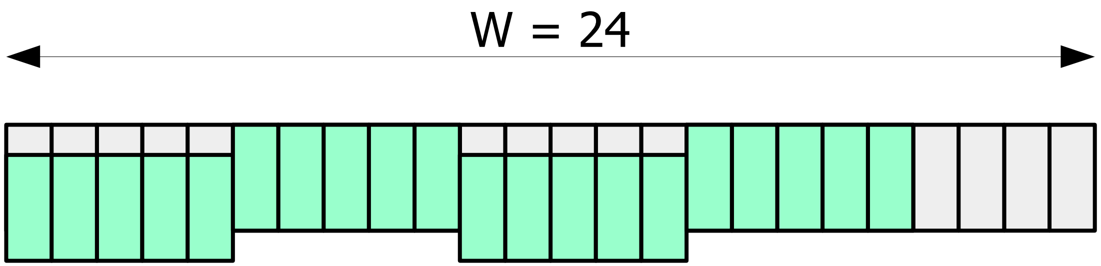
$(w_3, b_3) = (7, 10)$



... but this is <u>not optimal</u>, as **(w_3,b_3)** <u>takes a lot of space</u>.

The difficulty of KP

... taking some of $(\mathbf{w_2},\mathbf{b_2})$ will be better:

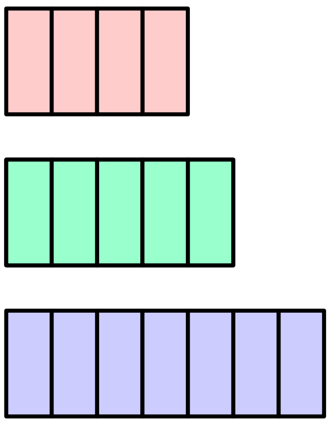


B = 4*9 = 36

$(\mathbf{w_1}, \mathbf{b_1}) = (\mathbf{4}, \mathbf{6})$

$(\mathbf{w_2}, \mathbf{b_2}) = (\mathbf{5}, \mathbf{9})$

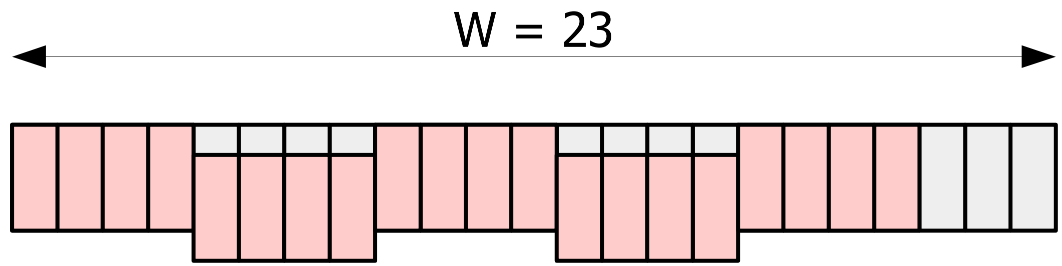
$(\mathbf{w_3}, \mathbf{b_3}) = (\mathbf{7}, \mathbf{10})$



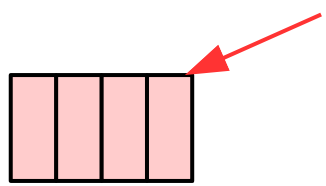
The difficulty of KP

Approach 2) - placing the item with minimal weight first.

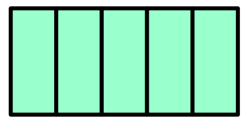
B = 5*6 = 30



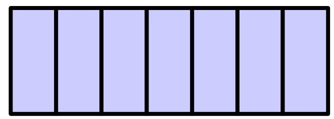
$(w_1, b_1) = (4, 6)$



$(w_2, b_2) = (5, 9)$



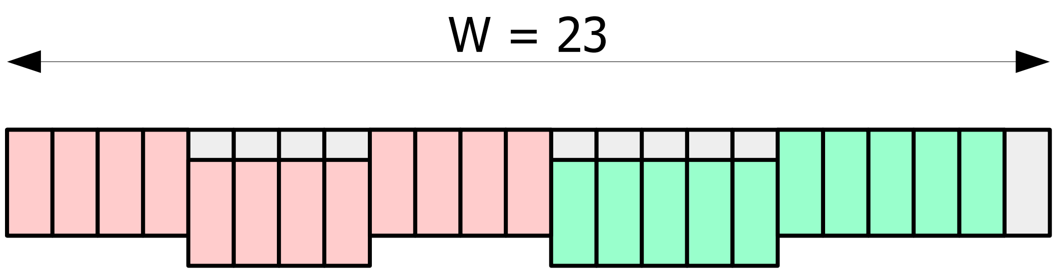
$(w_3, b_3) = (7, 10)$



... but this is not optimal, as **(w_1,b_1)** uses space inefficiently.

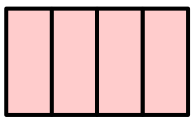
The difficulty of KP

… taking some of (w_2,b_2) instead will be better:

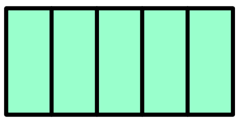


B = 3*6 + 2*9 =
= 18 + 18 = 36

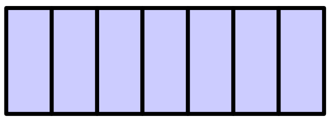
$(w_1, b_1) = (4, 6)$



$(w_2, b_2) = (5, 9)$



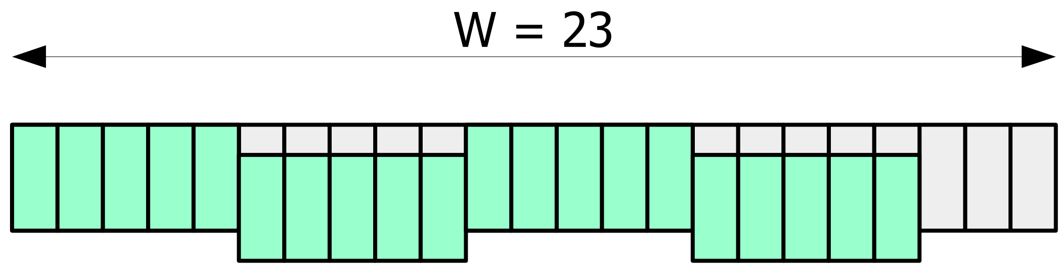
$(w_3, b_3) = (7, 10)$

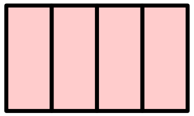


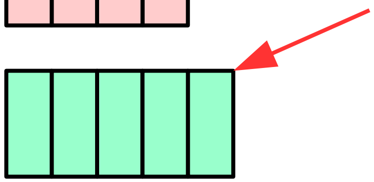
The difficulty of KP

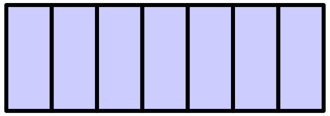
Approach 3) - placing the item with <u>maximal bonus/weight first</u>.

B = 4*9 = 36



(4, 6)  **6/4 = 1.5**

(5, 9)  **9/5 = 1.8**

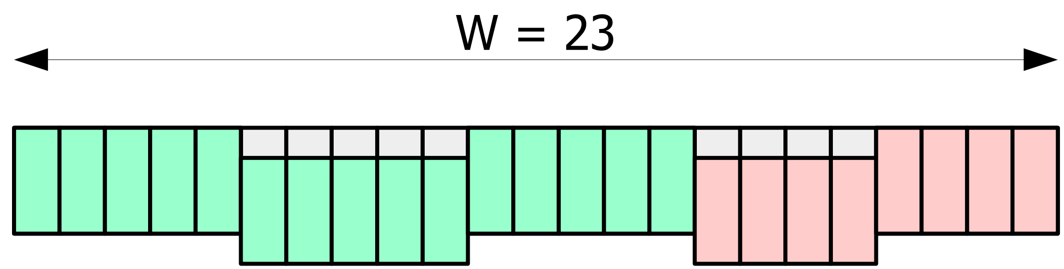
(7, 10)  **10/7 ~ 1.43**

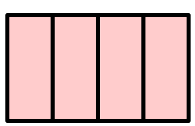
... but this is <u>not optimal</u>, as it will be better <u>to not fill</u> **(w_2,b_2)** till the end.

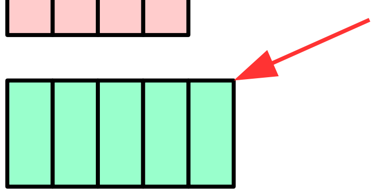
The difficulty of KP

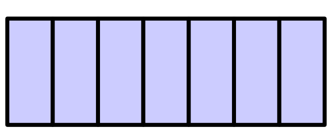
... rewinding one **(w_2,b_2)** will be better:

B = 3*9 + 2*6 =
= 27 + 12 = 39



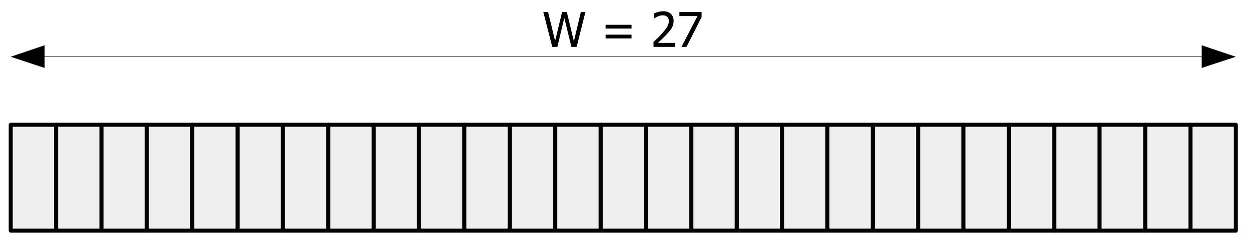
(4, 6)  **6/4 = 1.5**

(5, 9)  **9/5 = 1.8**

(7, 10)  **10/7 ~ 1.43**

Exercise

Solve KP for the following input set:

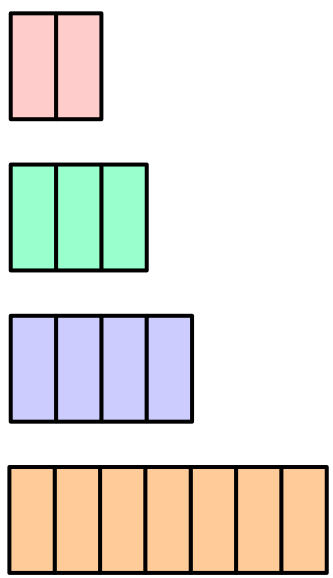


$(w_1, b_1) = (2, 1)$

$(w_2, b_2) = (3, 4)$

$(w_3, b_3) = (4, 7)$

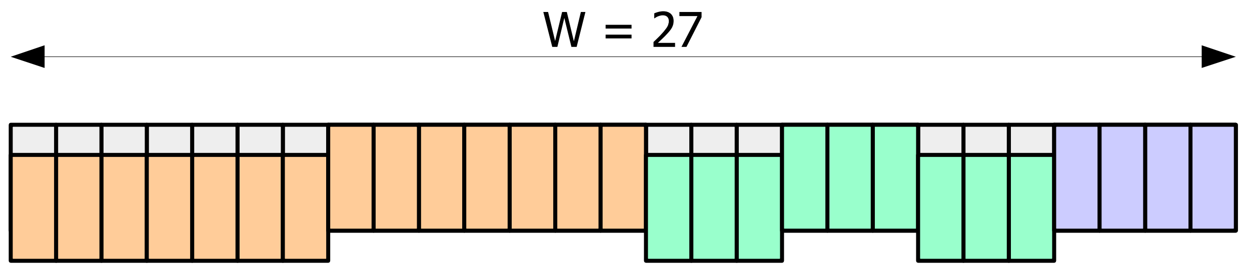
$(w_4, b_4) = (7, 14)$



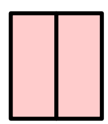
Exercise

(solution)

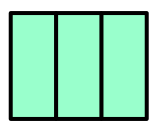
Solve KP for the following input set:



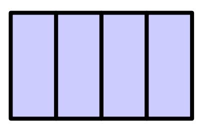
$(w_1, b_1) = (2, 2)$



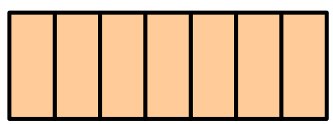
$(w_2, b_2) = (3, 4)$



$(w_3, b_3) = (4, 6)$



$(w_4, b_4) = (7, 14)$



$$
\begin{aligned}
\mathbf{B} &= 2*14 + 3*4 + 6 = \\
&= 28 + 12 + 6 = \\
&= 46
\end{aligned}
$$

Variants of KP

We have already formalized the Knapsack problem:

$$\sum x_i w_i \leq W$$

$$\sum x_i b_i = B \rightarrow max$$

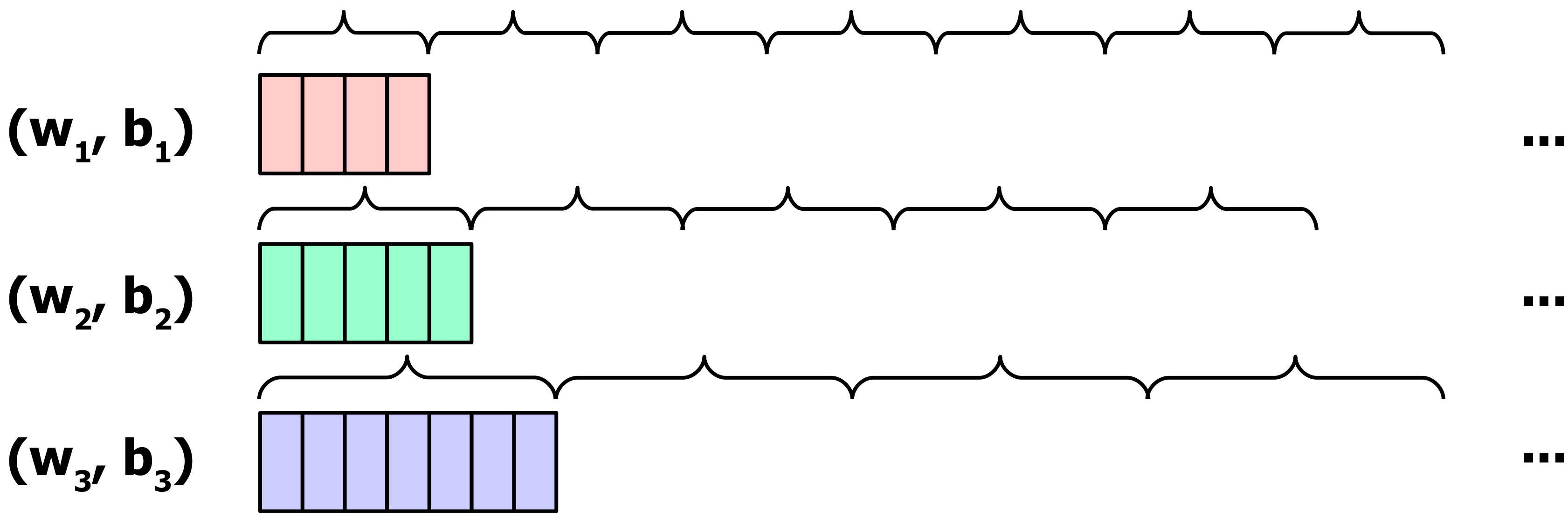
In statement that we have seen so far $\mathbf{x_i}$ are non-negative integers.

This variant of is called Unbounded Knapsack Problem (UKP).

$$\forall i, \quad x_i \in \{0, 1, 2, 3, \dots\}$$

Variants of KP

The impression is like:

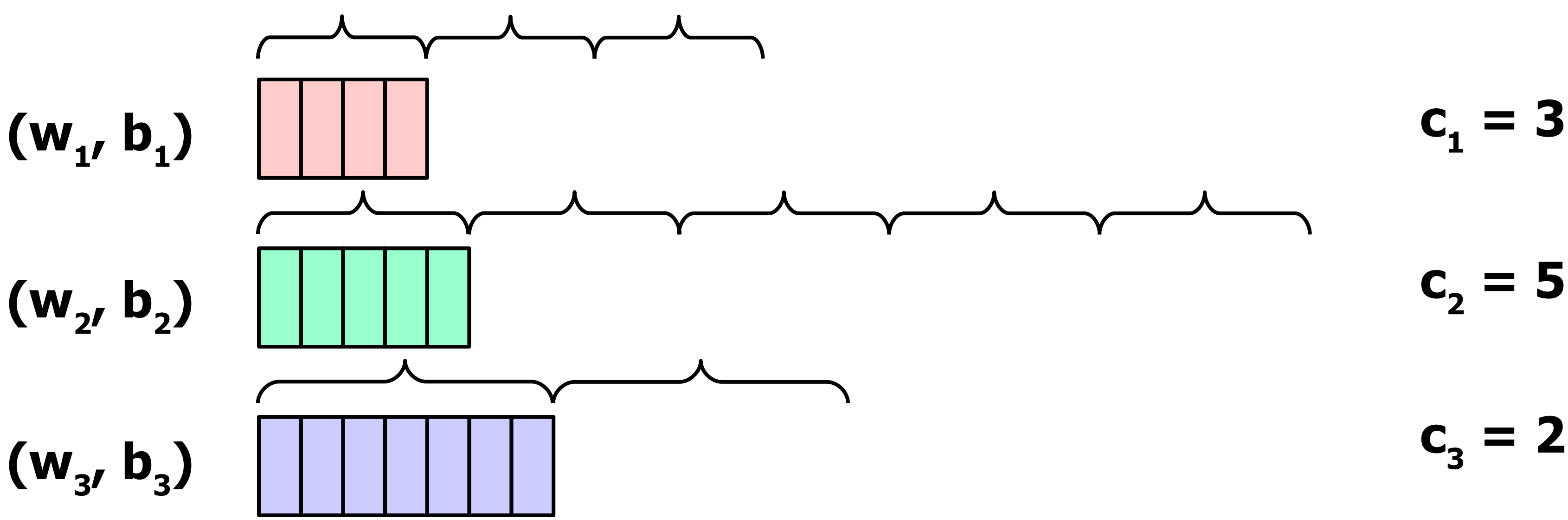


Variants of KP

Another variant is called Bounded Knapsack Problem (BKP), where:

$$\forall i\,,\ \ x_i \in \{0, 1, 2, 3, \ldots, c_i\}$$

The impression is like:



Variants of KP

The other variant is called <u>0-1 Knapsack Problem</u> (0-1 KP), where:

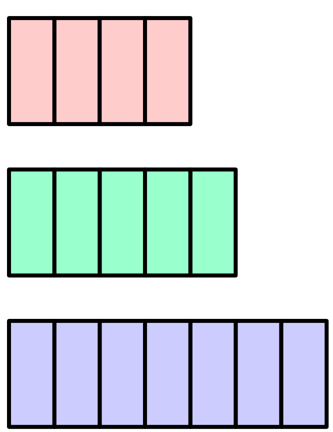
$$\forall i, \ \ x_i \in \{0, 1\}$$

... so we just either take or not take that item.

(w_1, b_1)

(w_2, b_2)

(w_3, b_3)



Variants of KP

Question: Which variant of Knapsack problem is easier?

Variants of KP

Question: Which variant of Knapsack problem is easier?

Answer: No one.

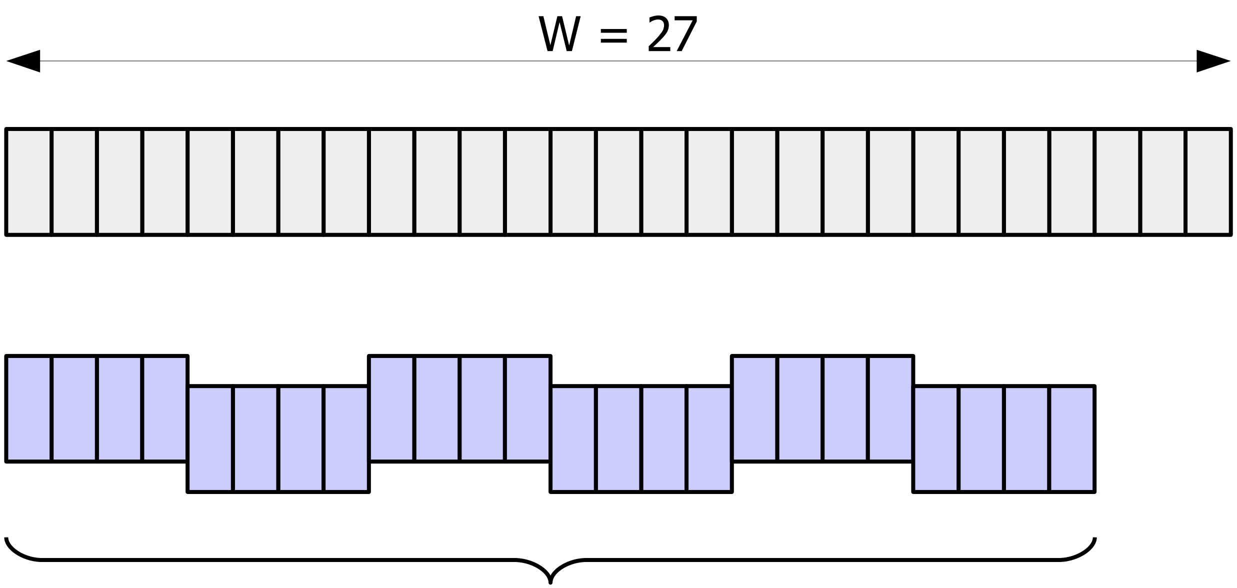
Variants of KP

Question: Can one of those variants be converted to some other?

Variants of KP

Question: Can one of those variants be converted to some other?

Answer: UKP can be brought to BKP, as any item '**i**' can be used at most $\lfloor \mathbf{W} / \mathbf{w_i} \rfloor$ times.

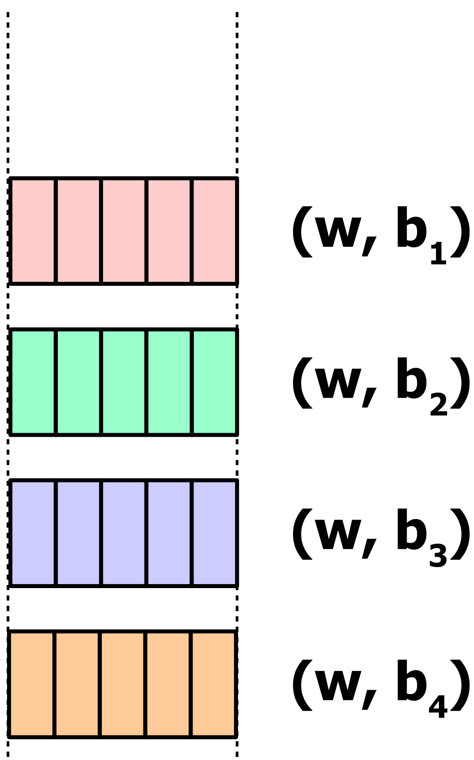


$$\mathbf{c_i} = \lfloor \mathbf{W} / \mathbf{w_i} \rfloor$$

Trivial cases

Before moving to the general solution, let's consider several trivial cases.

Case 1) - all weights are equal : $\mathbf{w_i = const}$



- This means we will always place $\lfloor \mathbf{W / w} \rfloor$ items,
- For UKP we will just take the one with $\mathbf{b_i} \rightarrow \mathbf{max}$.
- For "0-1 KP" we will place the items in <u>decreasing order</u> of $\mathbf{b_i}$.

Trivial cases

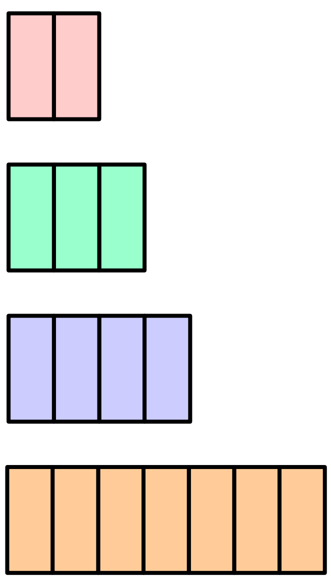
Case 2) - all bonuses are equal : $\mathbf{b_i = const}$

(w_1, b)

(w_2, b)

(w_3, b)

(w_4, b)



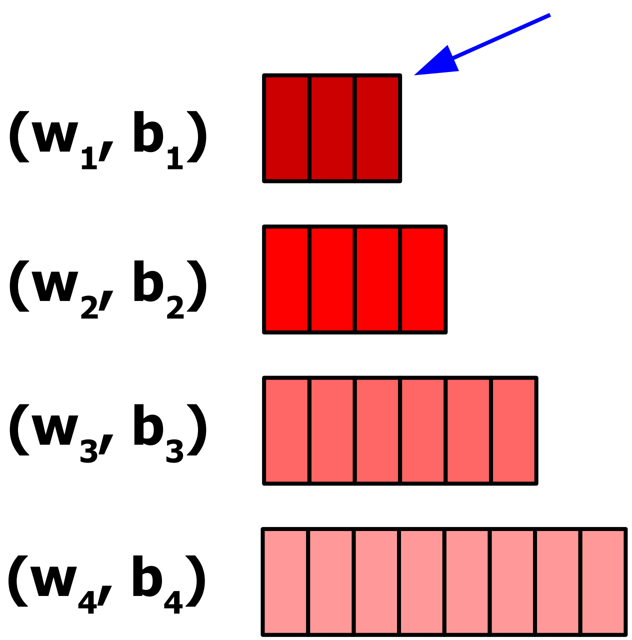
- This means that we want to place <u>as many items as possible</u>,
- For UKP we will just take the one with $\mathbf{w_i} \rightarrow \mathbf{min}$.
- For "0-1 KP" we will place the items in <u>increasing order</u> of $\mathbf{w_i}$.

Trivial cases

Case 3) – bonuses decrease, together with increase of weights:

$$w_1 \leq w_2 \leq w_3 \leq \ldots \leq w_N$$

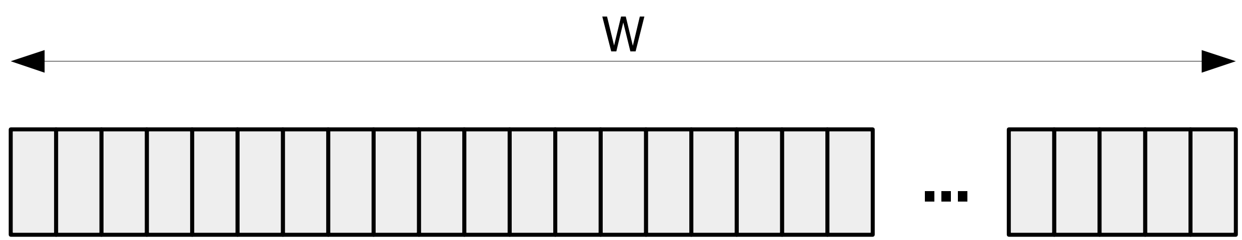
$$b_1 \geq b_2 \geq b_3 \geq \ldots \geq b_N$$



- This is also an easy case, as our preference here is clear,
- For UKP we will use <u>the first item only</u>,
- For 0-1 KP we will <u>place items from left to right</u>.

Trivial cases

Case 4) – 0-1 knapsack, so large that it will fit all items inside:

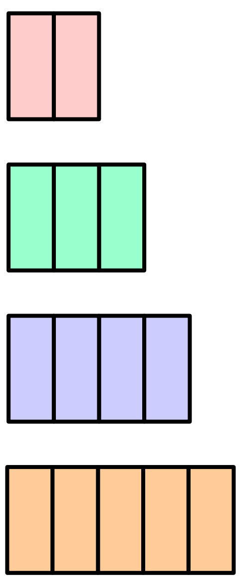


(w_1, b_1)

(w_2, b_2)

(w_3, b_3)

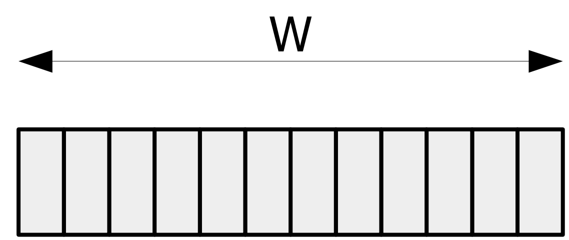
(w_4, b_4)



- We will just place all the items.

Trivial cases

Case 5) – knapsack so small, that it will fit only one item:

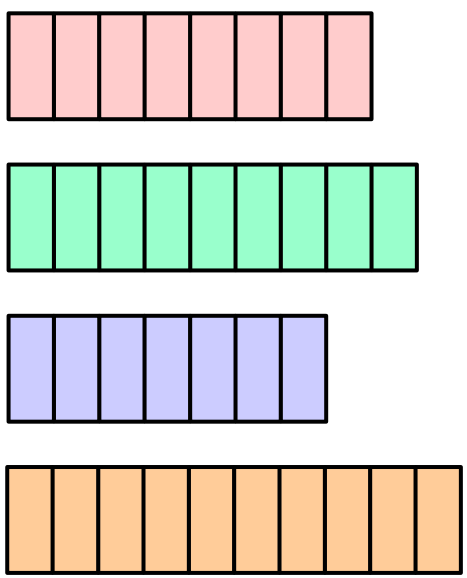


(w_1, b_1)

(w_2, b_2)

(w_3, b_3)

(w_4, b_4)



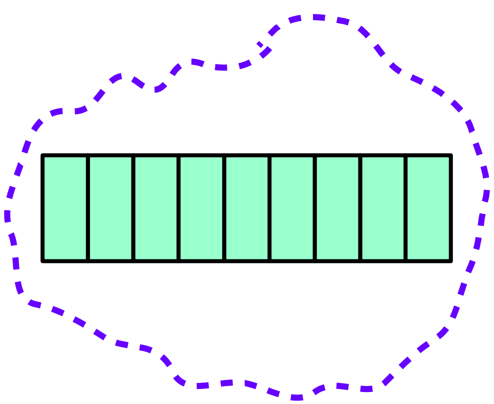
- We will place the one with **b_i -> max**.

Trivial cases

Before finishing the trivial cases part, let me point that complexity of KP comes from the fact that $\mathbf{x}_i$ must be integers.

... which means we can't cut items apart.

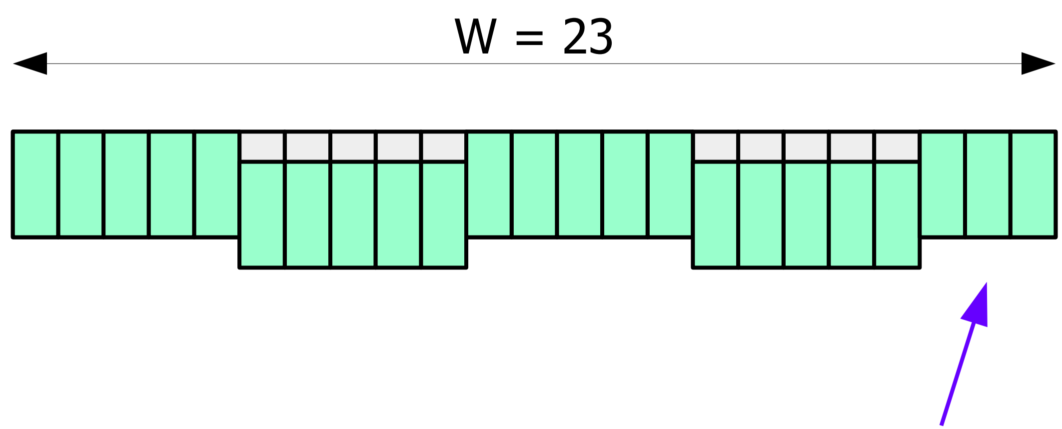
(w_2, b_2)



Otherwise, if we would be allowed to cut itmes apart (i.e. if $\mathbf{x}_i$ could be real numbers), then...

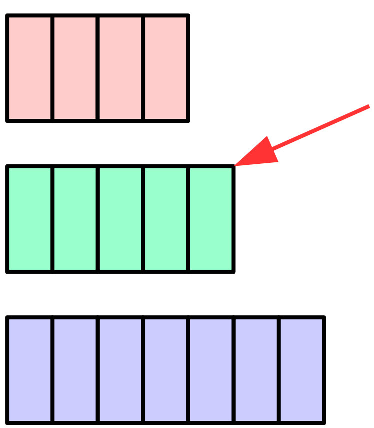
Trivial cases

For UKP we will just pick the item with $\mathbf{b_i/w_i}$ **-> max**, and fill the knapsack with it till the end.



$\mathbf{x_2 = 4.6}$

$\mathbf{B = 4.6*9 = 41.4}$



(4, 6) — **6/4 = 1.5**

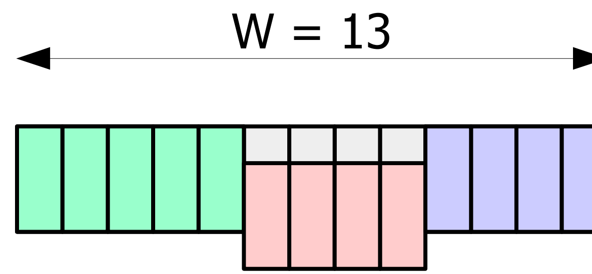
(5, 9) — **9/5 = 1.8**

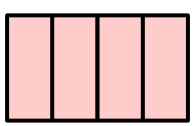
(7, 10) — **10/7 ~ 1.43**

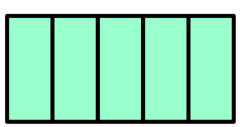
Trivial cases

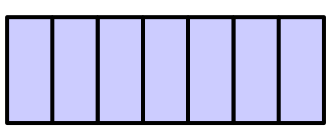
For 0-1 KP we will place all the items in decreasing order of $\mathbf{b_i/w_i}$ ratio.

B = 1*9 + 1*6 + (4/7)*10 =
= 9 + 6 + 5.714... =
= 20.714...



(4, 6)  **6/4 = 1.5**

(5, 9)  **9/5 = 1.8**

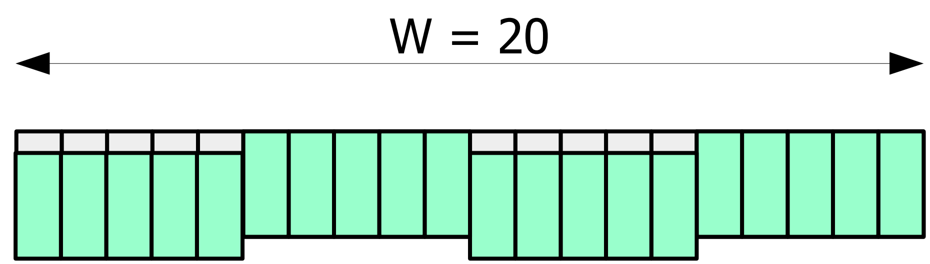
(7, 10)  **10/7 ~ 1.43**

In all cases we are sure that every unit of **W** carries as much bonus as it is possible.

Trivial cases

One more aspect is that complexity of KP comes when we would like to cut some items... which is actually not allowed.

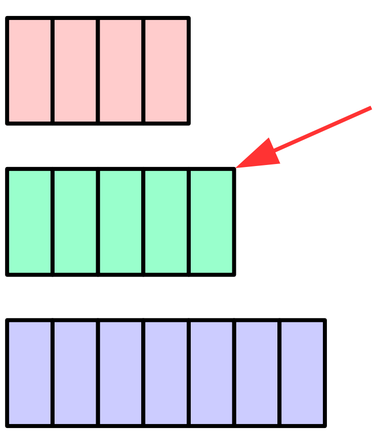
B = 4*9 = 36



(4, 6)

(5, 9)

(7, 10)



6/4 = 1.5

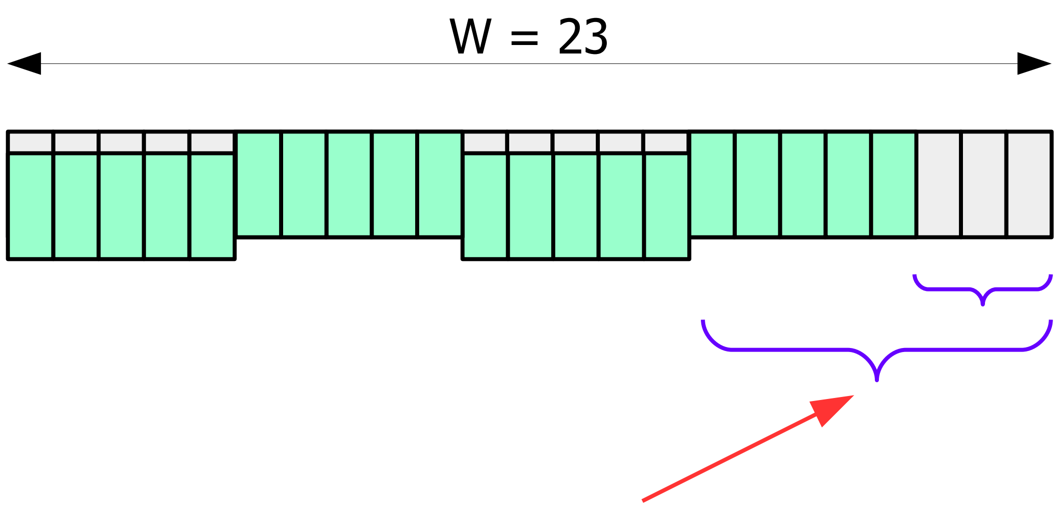
9/5 = 1.8

10/7 ~ 1.43

In UKP, if no need to cut ever arises, we can just fill the **W** with the best item.

Knapsack problem

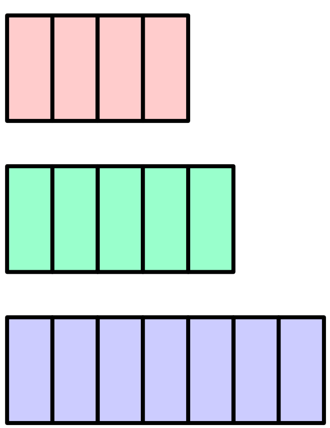
So the problem arises when we don't know <u>how to pack things in the remaining, smaller area...</u>



(4, 6)

(5, 9)

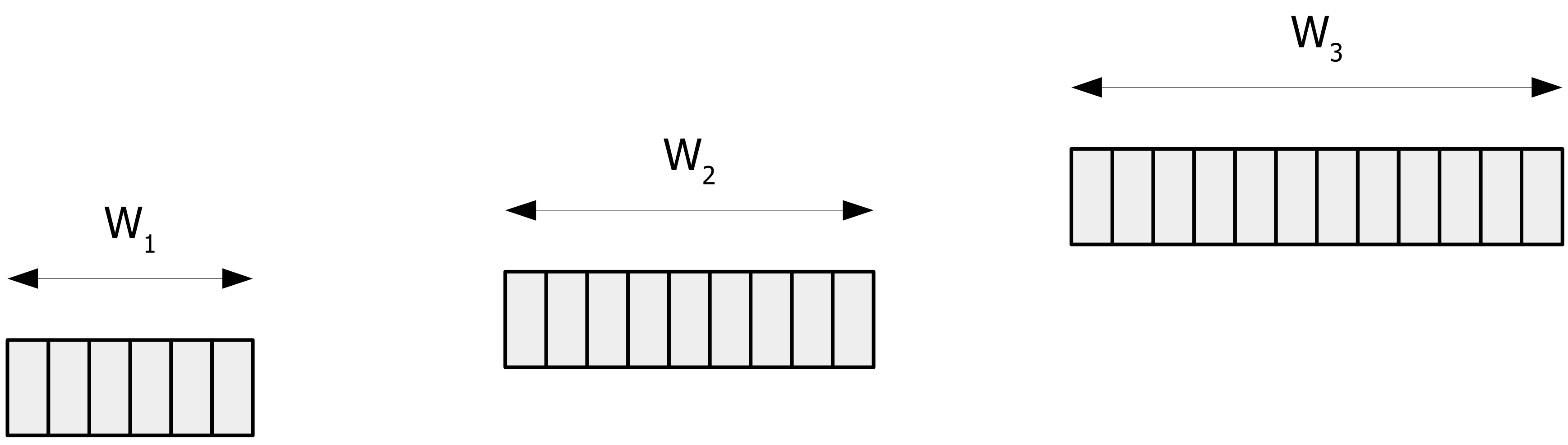
(7, 10)



Knapsack problem

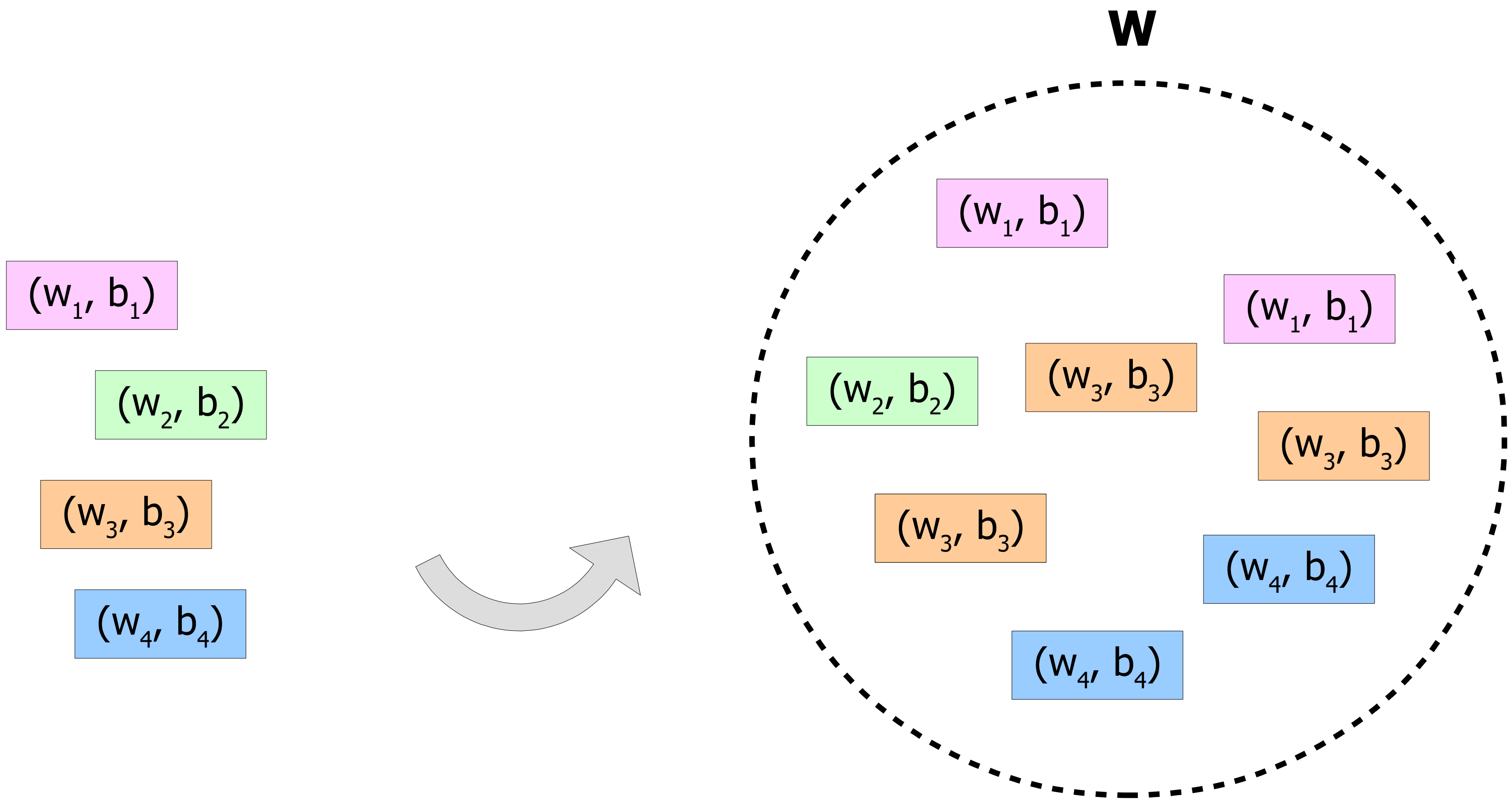
Which brings us to the idea that...

... instead of trying to fill knapsack **W**, perhaps it will be better to find at first solutions for smaller knapsacks?



Solution of UKP

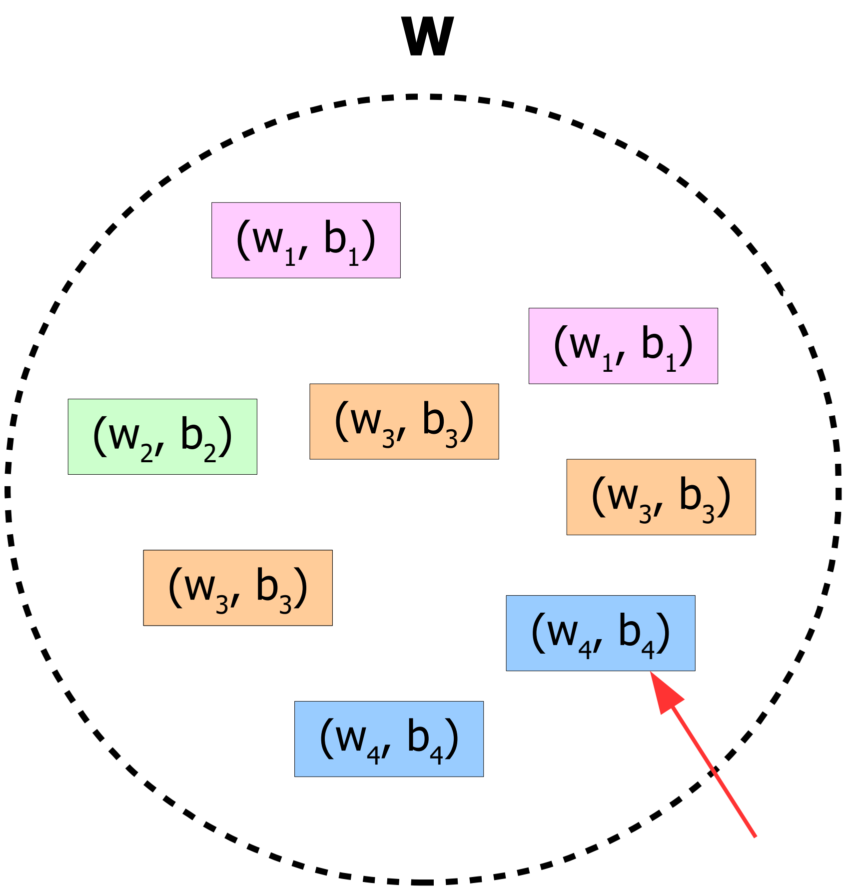
Suppose we have found the optimal solution for UKP.



Solution of UKP

Suppose we have found the optimal solution for UKP.

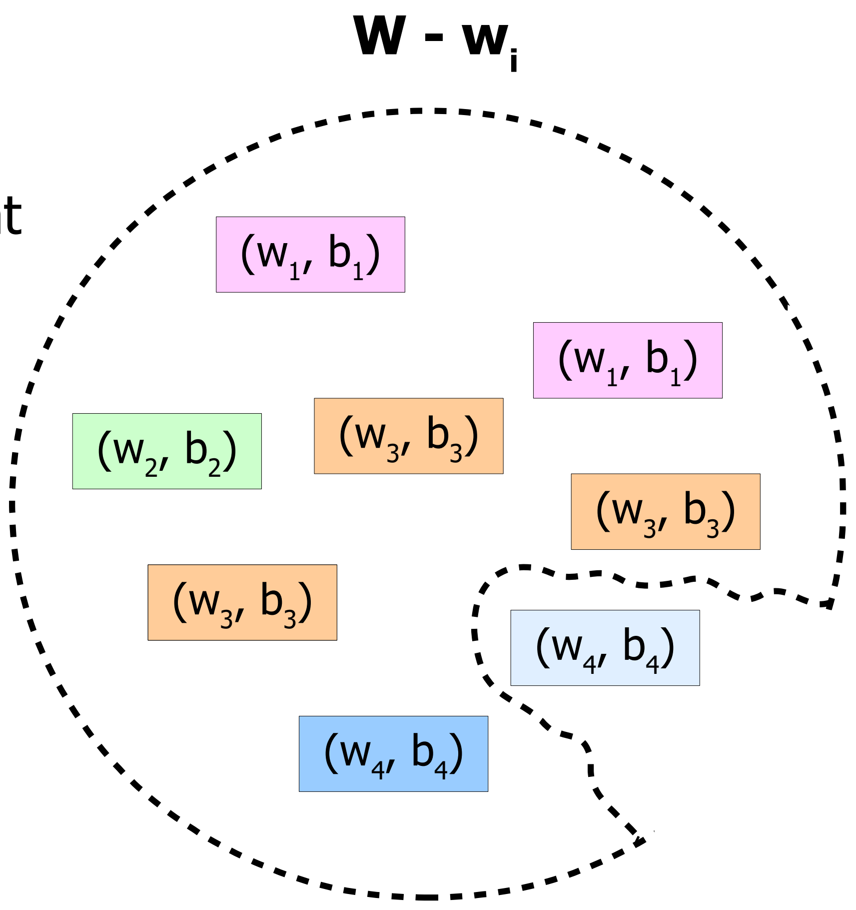
- One of those items $(\mathbf{w_i},\mathbf{b_i})$ <u>was placed the last</u>.



Solution of UKP

Suppose we have found the optimal solution for UKP.

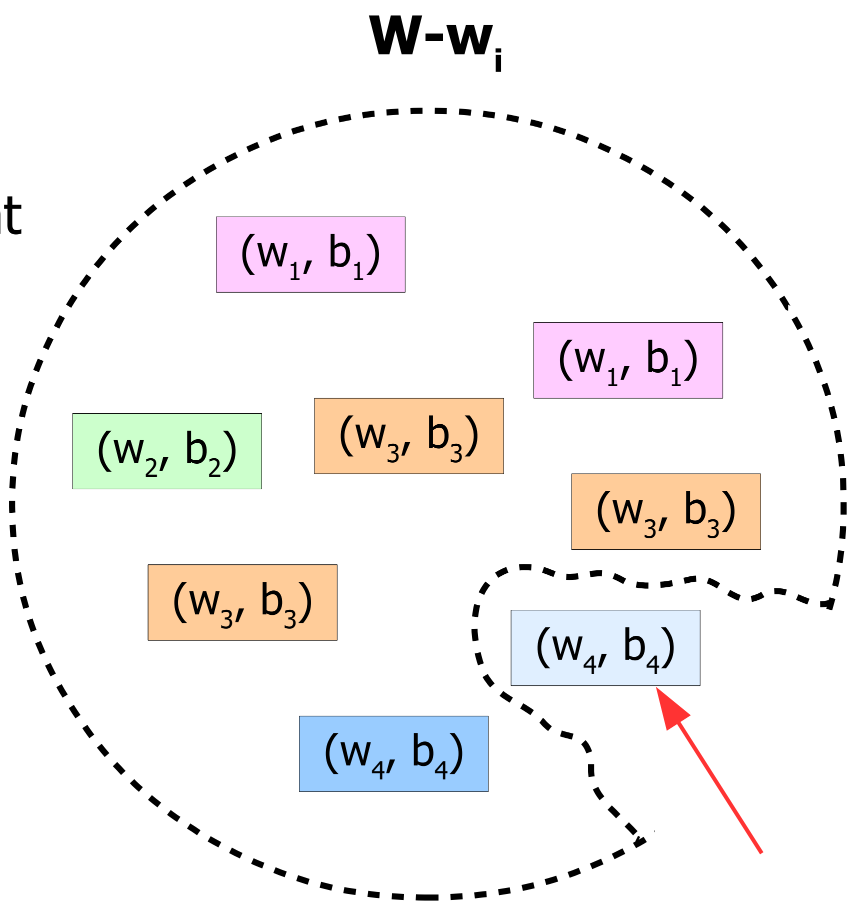
- One of those items $(\mathbf{w_i},\mathbf{b_i})$ was placed the last.
- Which means that in the previous moment, we had optimal placement for knapsack of size **"W – $\mathbf{w_i}$"**.



Solution of UKP

Suppose we have found the optimal solution for UKP.

- One of those items $\mathbf{(w_i,b_i)}$ <u>was placed the last</u>.
- Which means that in the previous moment, we had optimal placement for knapsack of size **"W–$\mathbf{w_i}$"**.

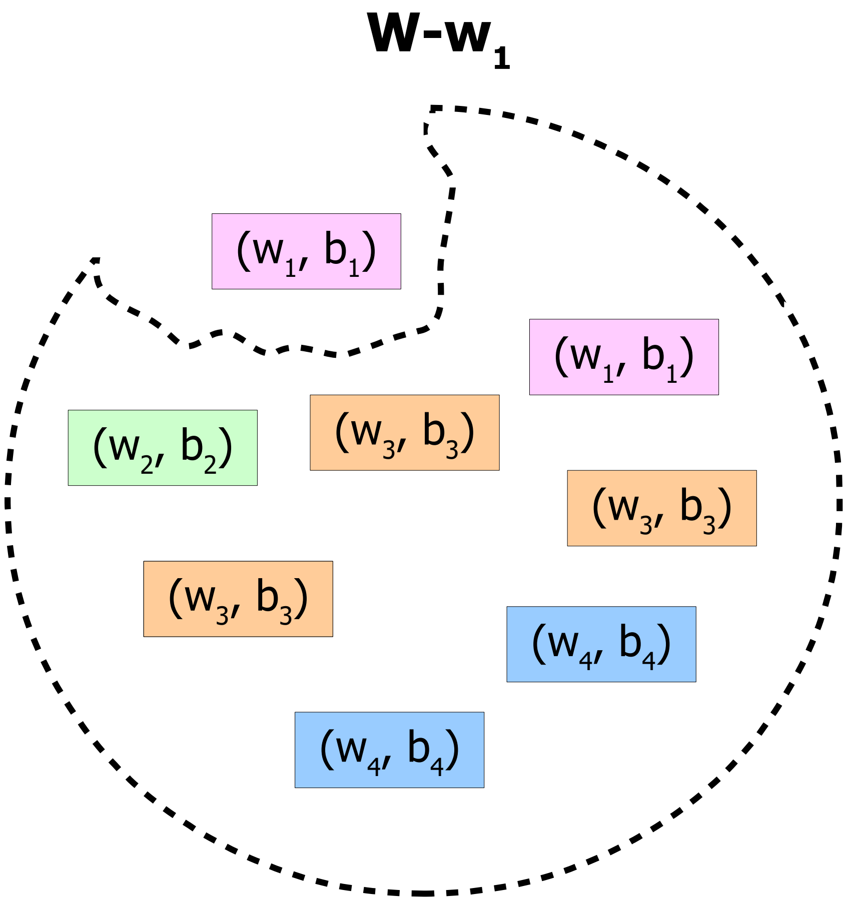


So <u>if we knew</u> which item will be placed the last, we can reduce our problem to knapsack of size "**W-$\mathbf{w}$**$_i$".

Solution of UKP

But the last placed item is definitely one of the **N** existing items:

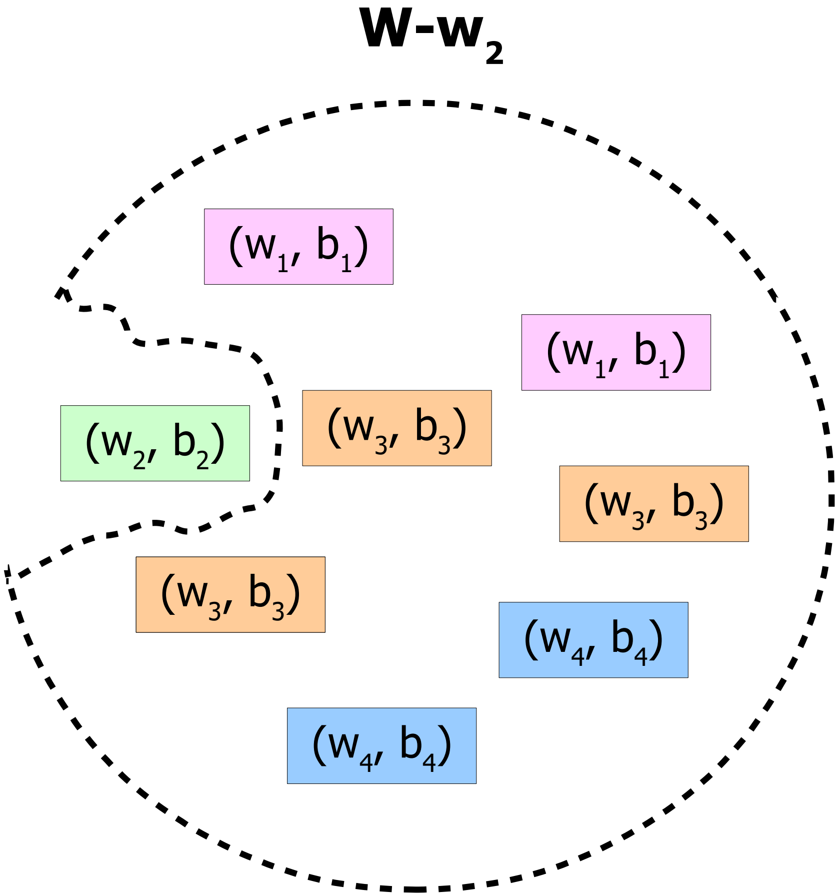
- So we can try **N** variants.
- ...
- for $\mathbf{(w_1, b_1)}$,



Solution of UKP

But the last placed item is definitely <u>one of the **N** existing items</u>:

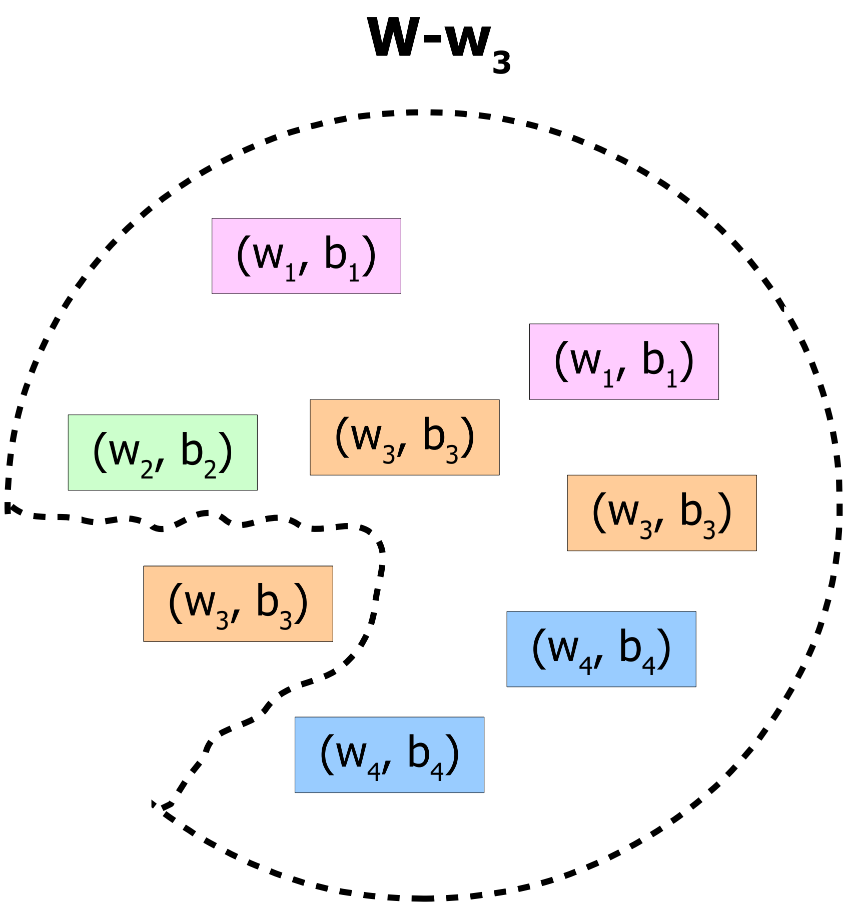
- So we can try **N** variants.
- ...
- for **(w_2, b_2)**,



Solution of UKP

But the last placed item is definitely one of the **N** existing items.

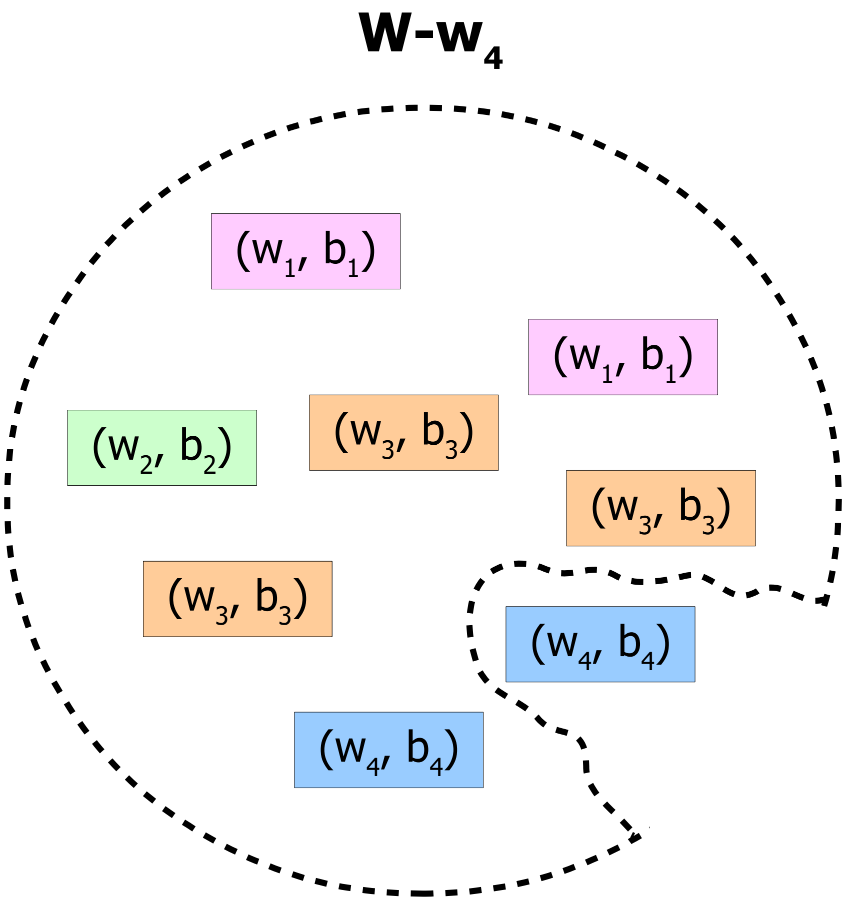
- So we can try **N** variants.
- ...
- for $\mathbf{(w_3, b_3)}$,



Solution of UKP

But the last placed item is definitely <u>one of the **N** existing items</u>:

- So we can try **N** variants.
- ...
- and finally for **(w_4, b_4)**,



Solution of UKP

This brings us to the following recursive formula:

```
B[W] = max(
    B[ W−w₁ ] + b₁,
    B[ W−w₂ ] + b₂,
    ...
    B[ W−w_N ] + b_N ),
```

where **B[i]** is the optimal bonus for knapsack of size '**i**'.

Exit-case for such recursive formula will be:

```
B[0] = 0,

B[−i] : (not allowed).
```

Solution of UKP

We can already write the pseudo-code:

```
N, W : Integer,
w[0..N), b[0..N) : Array of Integers,
B[0..W] : Array of Integers,

procedure UKP_DP()
    B[0] := 0
    for i := 1 to W
        for k := 0 to N-1
            if w[k] <= i
                B[i] := max( B[i], B[i-w[k]] + b[k] )
```

Solution of UKP

So result of this algorithm is array "**B[]**",

	0	1	2	3	4	5	6	7	8	9		W-2	W-1	W
B[] :											**...**			

Time complexity is **O(WN)**, as filling each cell of "**B[]**" requires **O(N)** time.

Solution of UKP

Question: Which approach will work faster here, DP or memoization?

Solution of UKP

... let's see how memoization will be written:

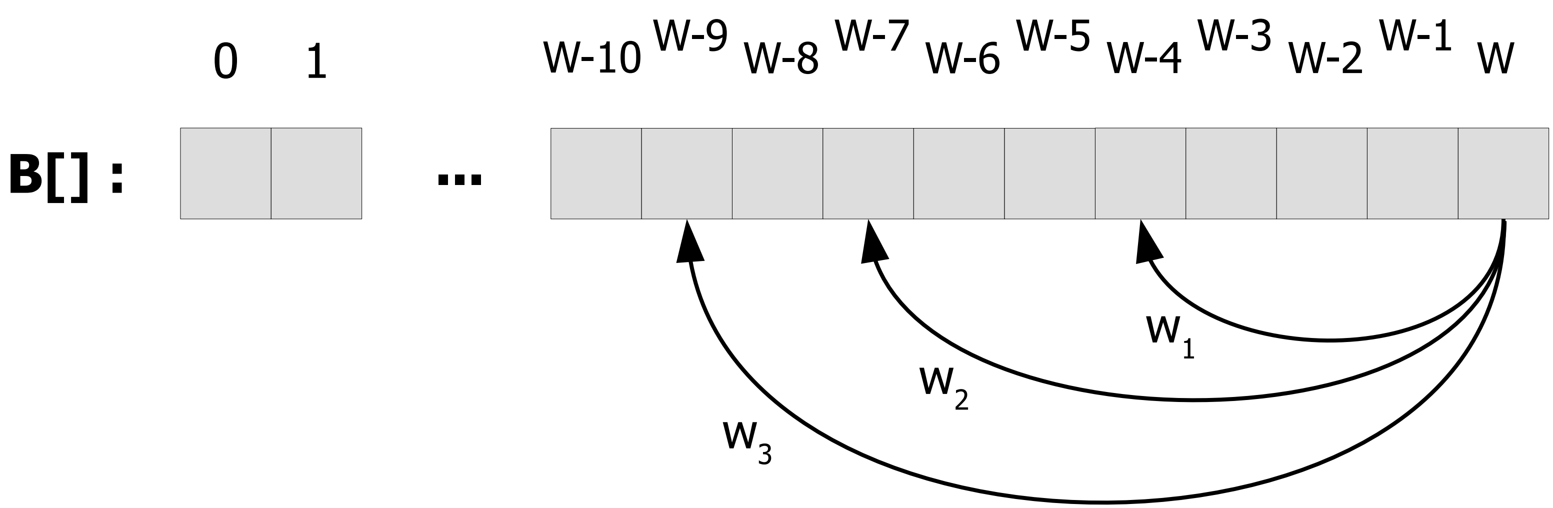
```
N, W : Integer,
w[0..N), b[0..N) : Array of Integers,
B[0..W] = {-1} : Array of Integers,

function UKP_memoization( i: Integer ) : Integer
    if B[i] != -1
        return B[i]        // Already was calculated
    if i == 0
        return B[0] := 0   // The answer for B[0]
    for k := 0 to N-1      // General case
        if w[k] <= i
            B[i] := max( B[i],
                    UKP_memoization( i-w[k] ) + b[k] )
    return B[i]
```

Solution of UKP

Question: Which approach will work faster here, DP or memoization?

Answer: Memoization addresses & calculates <u>some of the previous cells</u>,

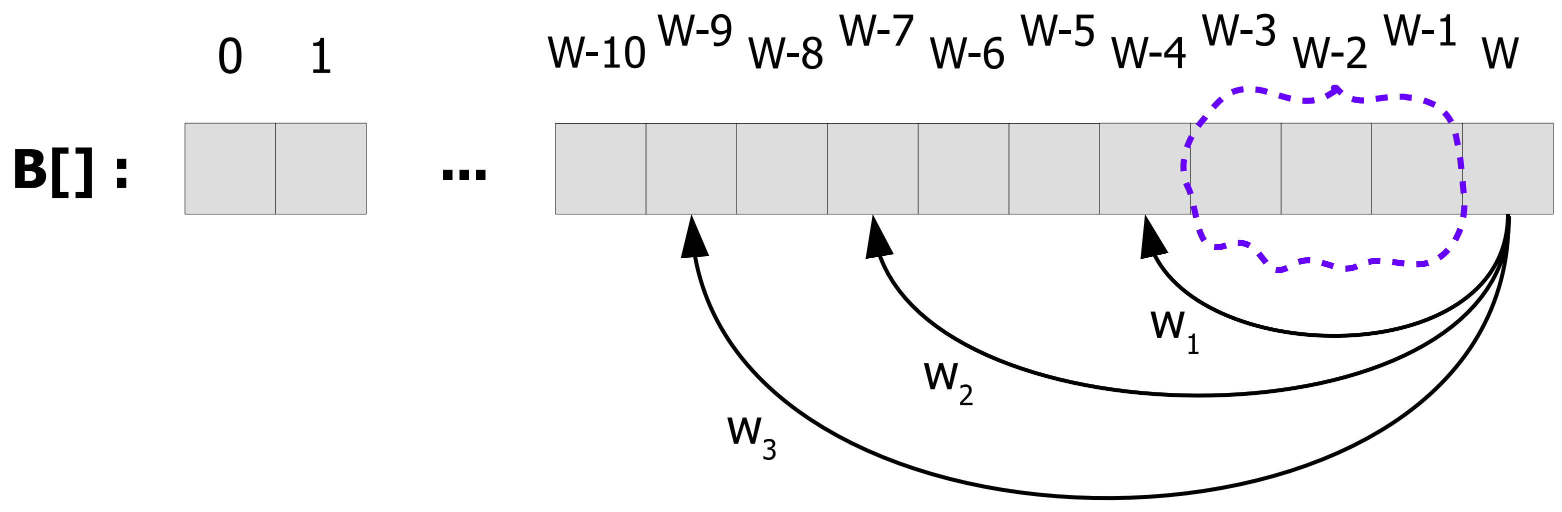


Solution of UKP

Question: Which approach will work faster here, DP or memoization?

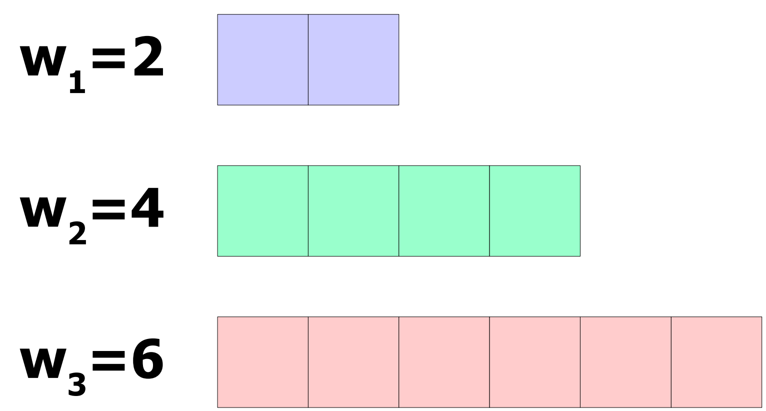
Answer: Memoization addresses & calculates some of the previous cells,

... which means that some other cells will remain not calculated.



Solution of UKP

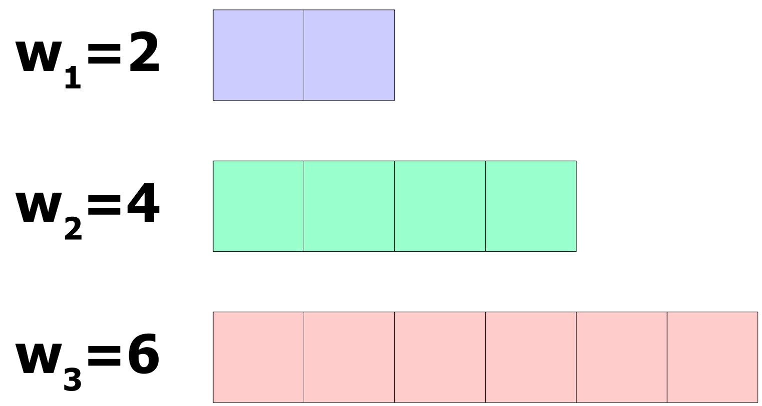
Question: If all weights $\mathbf{w_1, w_2, \ldots, w_N}$ are even, can we somehow optimize DP approach?

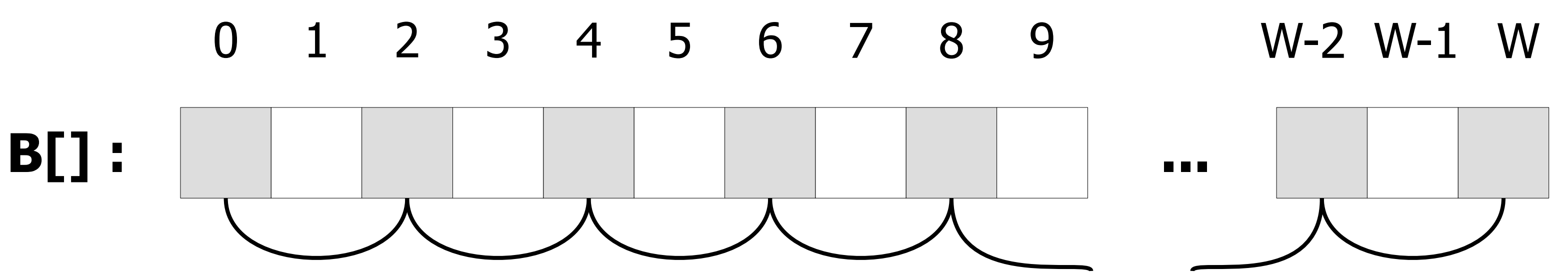


Solution of UKP

Question: If all weights $\mathbf{w_1, w_2, \ldots, w_N}$ are even, can we somehow optimize DP approach?

Answer: Yes, we can calculate only even indexes of **B[]**, as the odd ones will definitely remain **0**.





Solution of UKP

Question: If all weights $\mathbf{w_1}$, $\mathbf{w_2}$, **...**, $\mathbf{w_N}$ are even, can we somehow optimize DP approach?

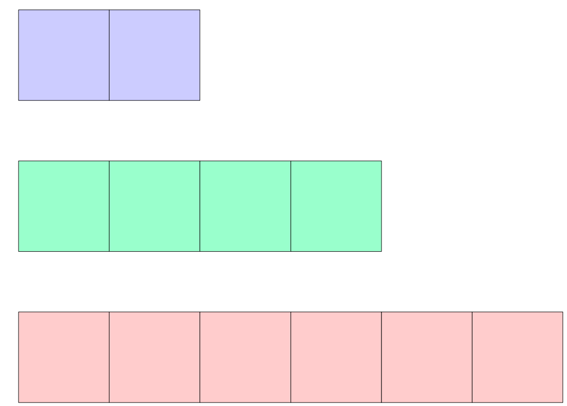
Answer: Yes, we can calculate only even indexes of **B[]**, as the odd ones will definitely remain **0**.

... note, if we do memoization, that <u>will be optimized automatically</u>.

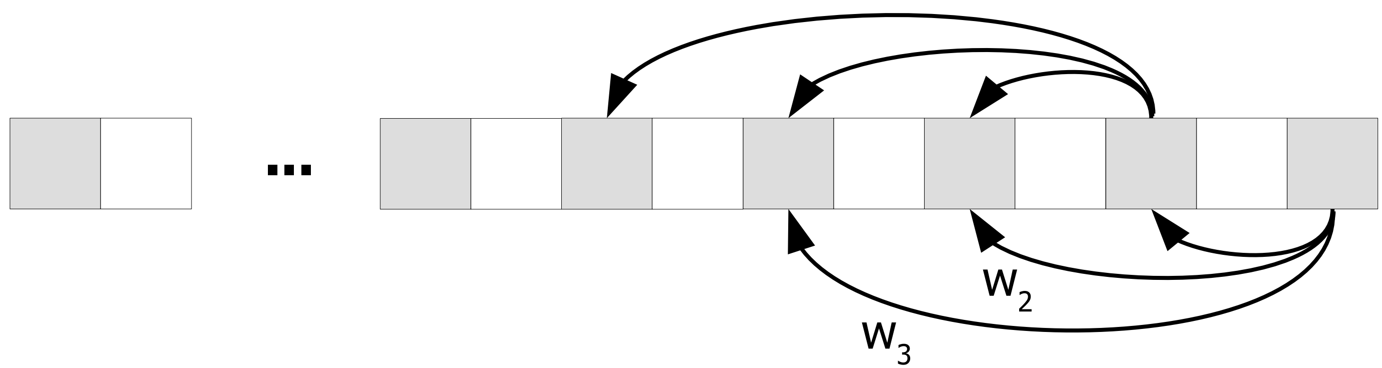
$\mathbf{w_1=2}$

$\mathbf{w_2=4}$

$\mathbf{w_3=6}$



B[] :



Obtaining items for UKP

This method constructs the array **B[]**, where **B[i]** shows optimal bonus for kanpsack of weight '**i**'.

	0	1	2	3		W-10	W-9	W-8	W-7	W-6	W-5	W-4	W-3	W-2	W-1	W
B[] :	0	0	3	5	**...**	24	24	26	26	27	29	29	31	36	36	37

Obtaining items for UKP

This method constructs the array **B[]**, where **B[i]** shows optimal bonus for kanpsack of weight '**i**'.

... but can we identify <u>exact set of items</u>, which gives us bonus **B[W]**?

	0	1	2	3		W-10	W-9	W-8	W-7	W-6	W-5	W-4	W-3	W-2	W-1	W
B[] :	0	0	3	5	**...**	24	24	26	26	27	29	29	31	36	36	37

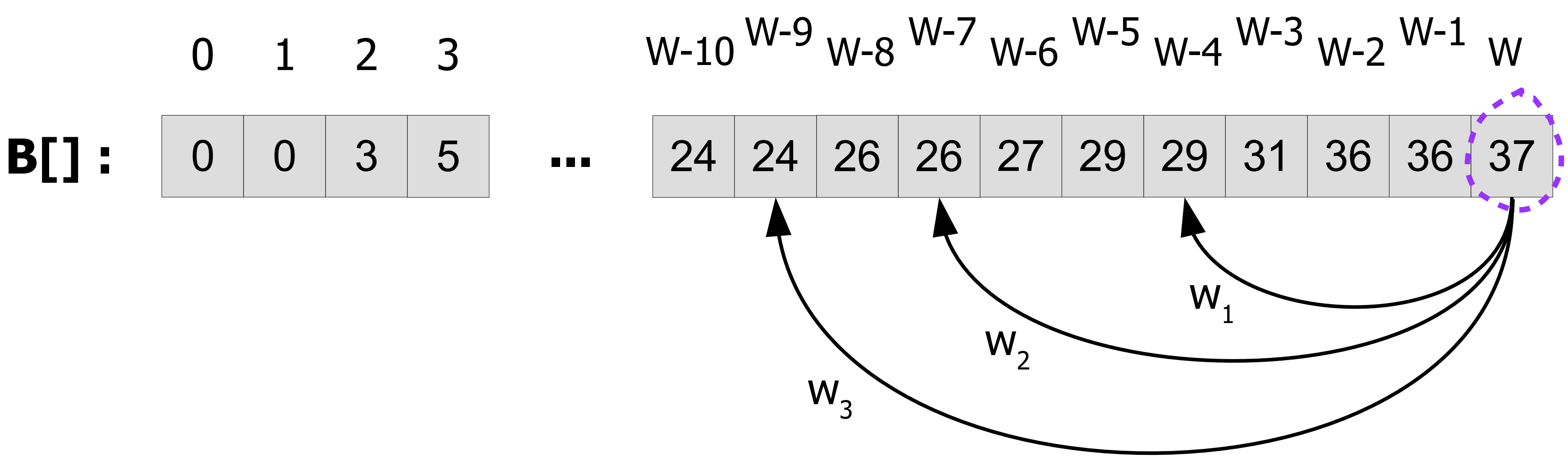
Obtaining items for UKP

This method constructs the array **B[]**, where **B[i]** shows optimal bonus for kanpsack of weight '**i**'.

... but can we identify <u>exact set of items</u>, which gives us bonus **B[W]**?

In order to do that, we must <u>move back – from right to left</u>.

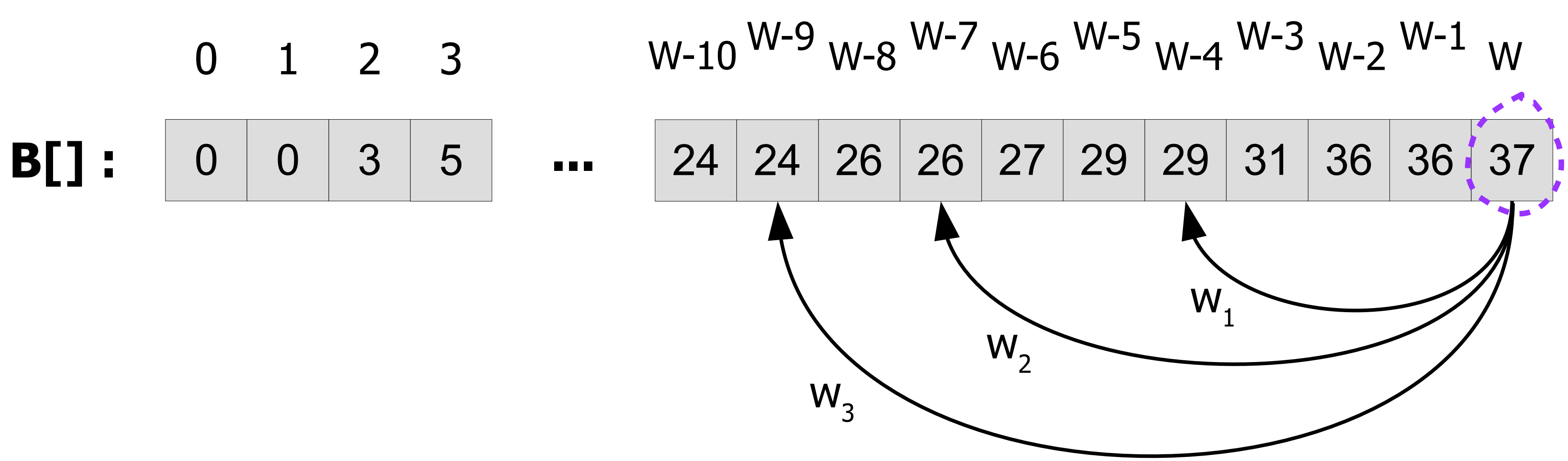
The item **B[W]** was calculated by one of items **B[W-w_1], B[W-w_2], ..., B[W-w_N]**.



Obtaining items for UKP

Let's recall the formula for **B[i]**:

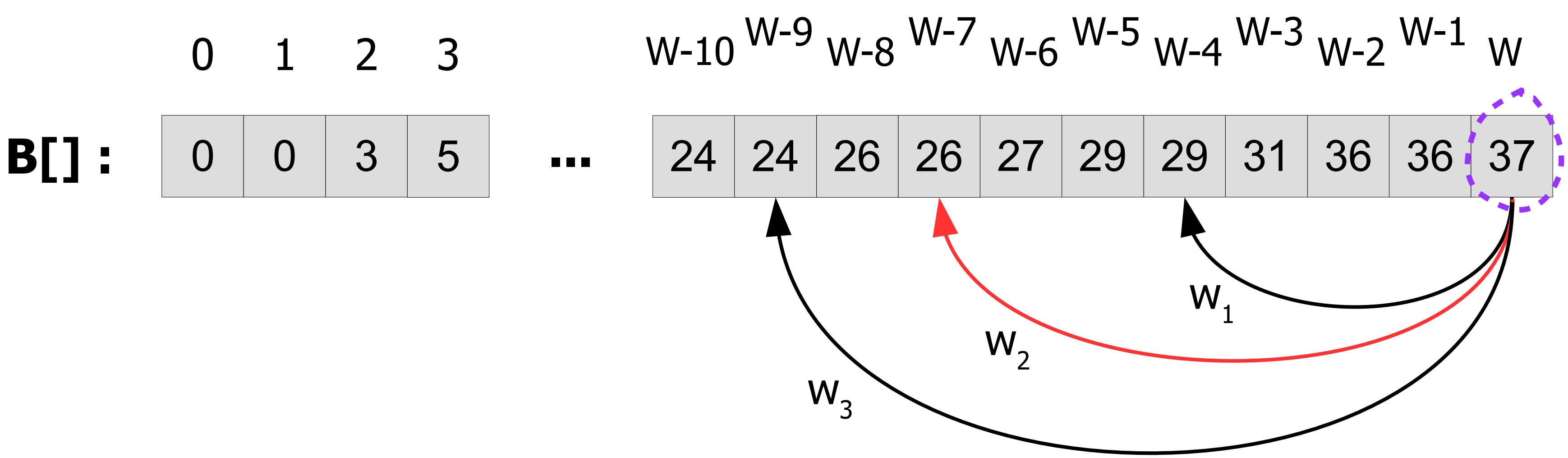
$$
\begin{aligned}
&B[W] = \max(\\
&\quad B[\,W-w_1\,] + b_1, \\
&\quad B[\,W-w_2\,] + b_2, \\
&\quad \ldots \\
&\quad B[\,W-w_N\,] + b_N\,),
\end{aligned}
$$



Obtaining items for UKP

So we can just check the **N** options, and see which one gives:

$$B[W] = B[\ W-w_i\ ] + b_i.$$

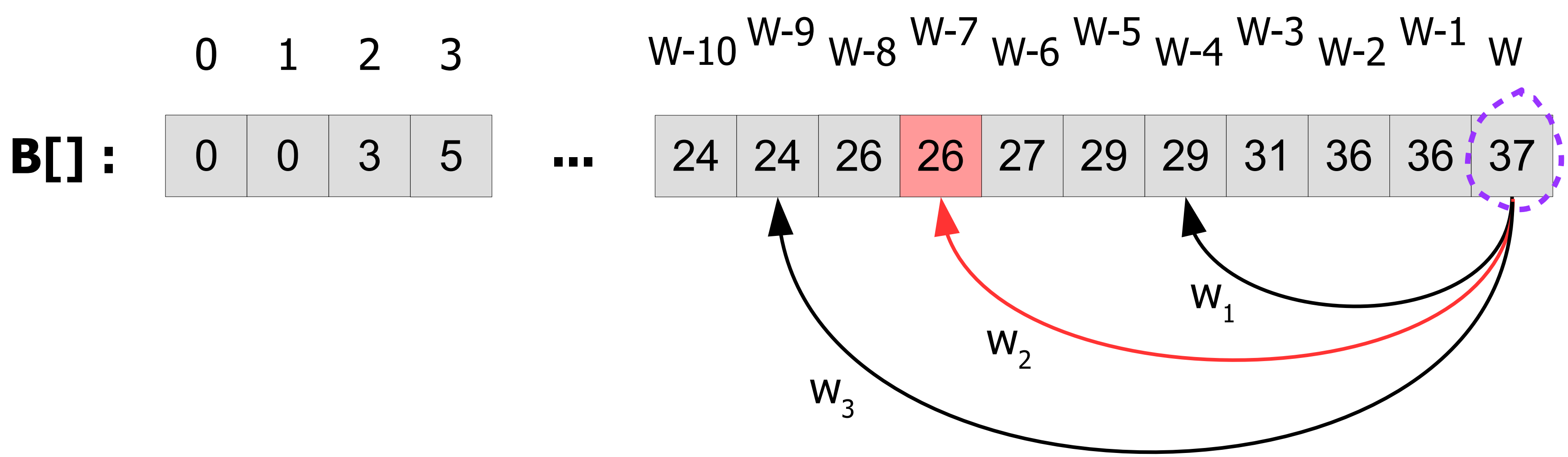


Obtaining items for UKP

So we can just check the **N** options, and see which one gives:

$$B[W] = B[\ W - w_i\ ] + b_i.$$

Once found, we know that the last placed item was **(w_i, b_i)**,



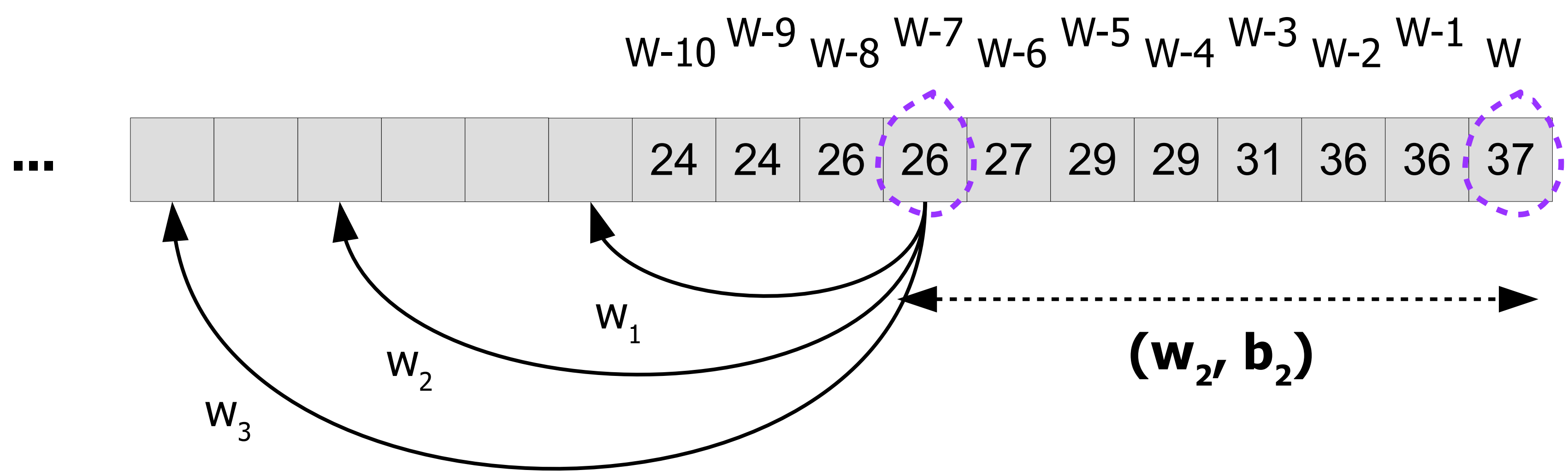
Obtaining items for UKP

So we can just check the **N** options, and see which one gives:

```
B[W] = B[ W-w_i ] + b_i.
```

Once found, we know that the last placed item was **(w_i, b_i)**,

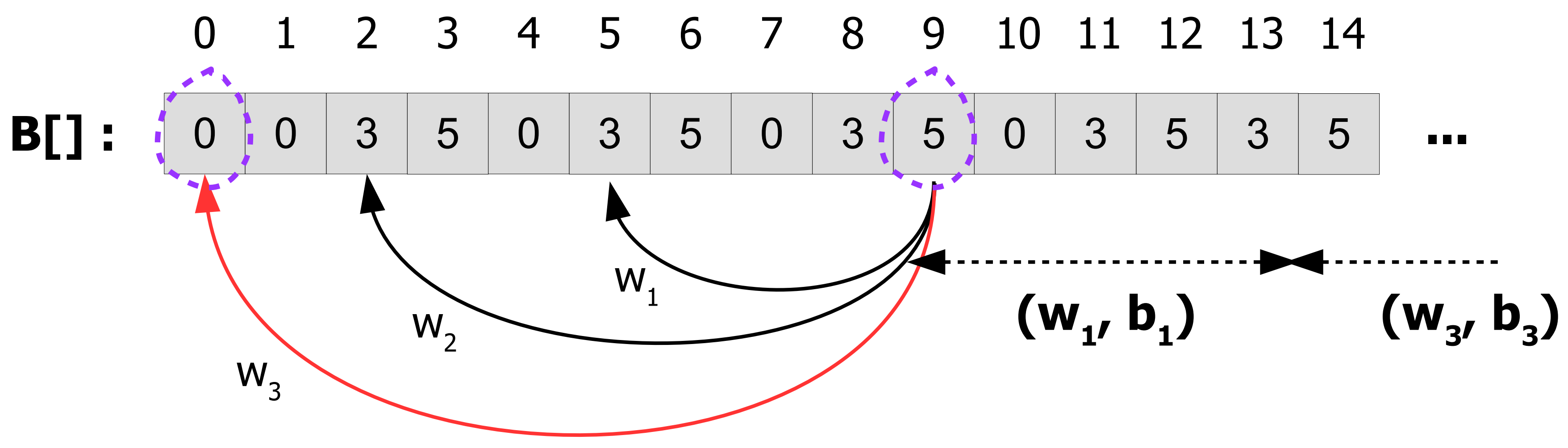
... and we can continue obtaining other items from **"W-w_i"**.



Obtaining items for UKP

This process will <u>finish when</u> we reach **B[0]**.

... at that point of time, all items which compose **B[W]** are found.



Obtaining items for UKP

... the pseudo-code of path restoration becomes:

```
N, W : Integer,
w[0..N), b[0..N) : Array of Integers,
B[0..W] = {-1} : Array of Integers,

procedure UKP_restore_path( x: Integer )
    if x == 0  // Check if all items are reported
        return
    for k := 0 to N-1  // Try item (w[k],b[k])
        if B[i] == B[i-w[k]] + b[k]
            report (w[k],b[k])
            UKP_restore_path( i-w[k] )
            break
```

Obtaining items for UKP

Question: Can we restore the paths faster?

Obtaining items for UKP

Question: Can we restore the paths faster?

Answer: Yes.

- Every value **B[i]** was calculated at some point of time,

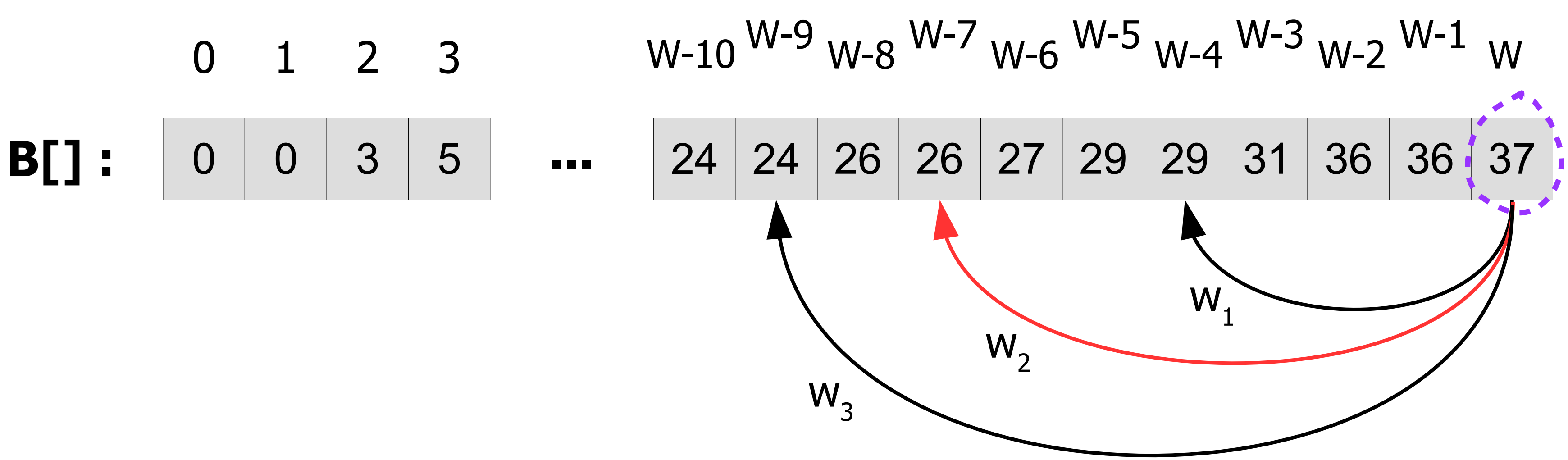
	0	1	2	3		W-10	W-9	W-8	W-7	W-6	W-5	W-4	W-3	W-2	W-1	W
B[] :	0	0	3	5	**…**	24	24	26	26	27	29	29	31	36	36	37

Obtaining items for UKP

Question: Can we restore the paths faster?

Answer: Yes.

- Every value **B[i]** was calculated at some point of time,
- It's value was **max()** from several variants **i** $\in$ **[1, N]**.



Obtaining items for UKP

Question: Can we restore the paths faster?

Answer: Yes.

- Every value **B[i]** was calculated at some point of time,
- It's value was **max()** from several variants **i** $\in$ **[1, N]**.
- So at the moment of calculation we can remember that index too.

	0	1	2	3		W-10	W-9	W-8	W-7	W-6	W-5	W-4	W-3	W-2	W-1	W
B[] :	0	0	3	5	...	24	24	26	26	27	29	29	31	36	36	37
last[] :	-	-	0	0		1	0	2	2	1	1	1	0	1	2	2

Obtaining items for UKP

So '**last[x]**' gives us index of the item, which will be placed the last to obtain weight '**x**'.

The pseudocode becomes shorter:

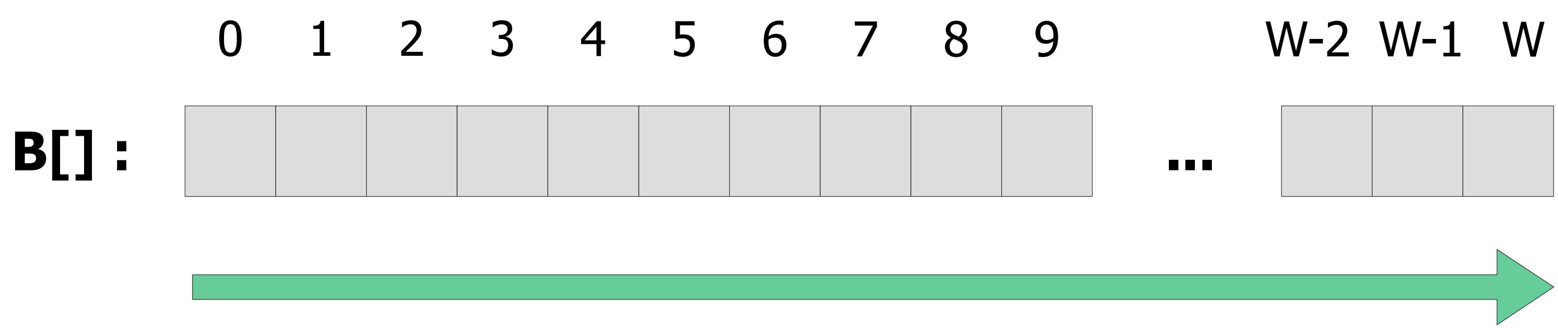
```
N, W : Integer,
w[0..N), b[0..N) : Array of Integers,
B[0..W] = {-1} : Array of Integers,
last[0..W] : Array of Integers,

procedure UKP_restore_path( x: Integer )
    if x == 0 or last[x] == -1  // Check for completion
        return
    report ( w[last[x]], b[last[x]] )  // Report last item
    UKP_restore_path( x - w[last[x]] )  // Continue
```

Solution of 0-1 KP

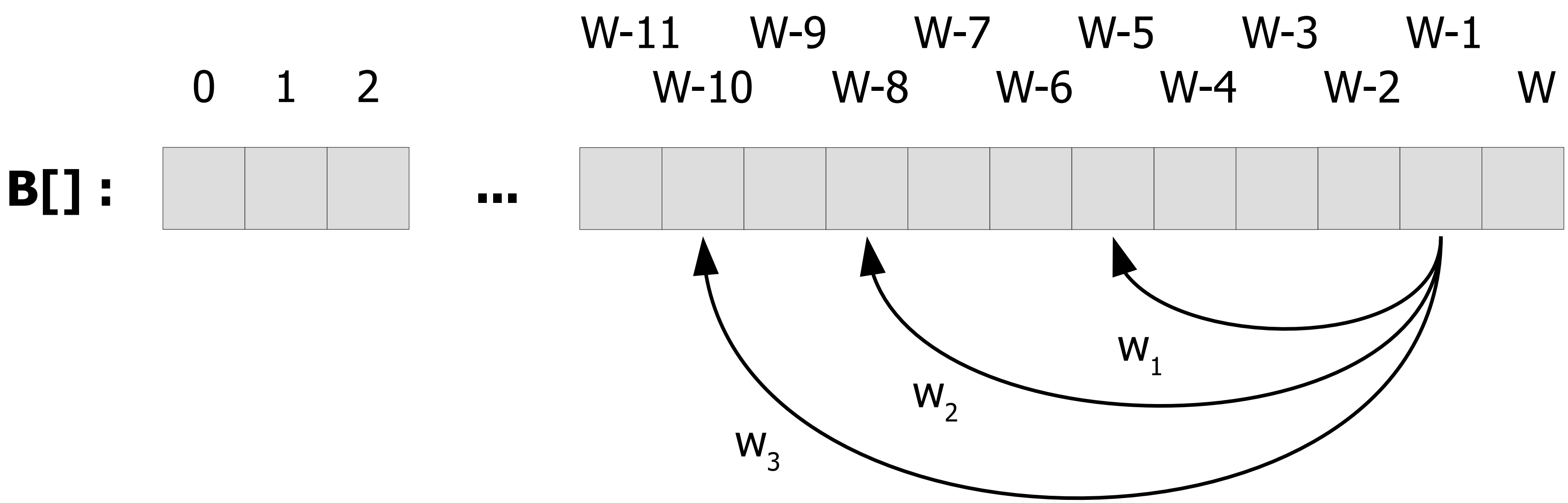
If solving problem of 0-1 KP, can we similarly fill the array "**B[]**", from left to right?

... reminder, now we can user <u>every item only once</u>.



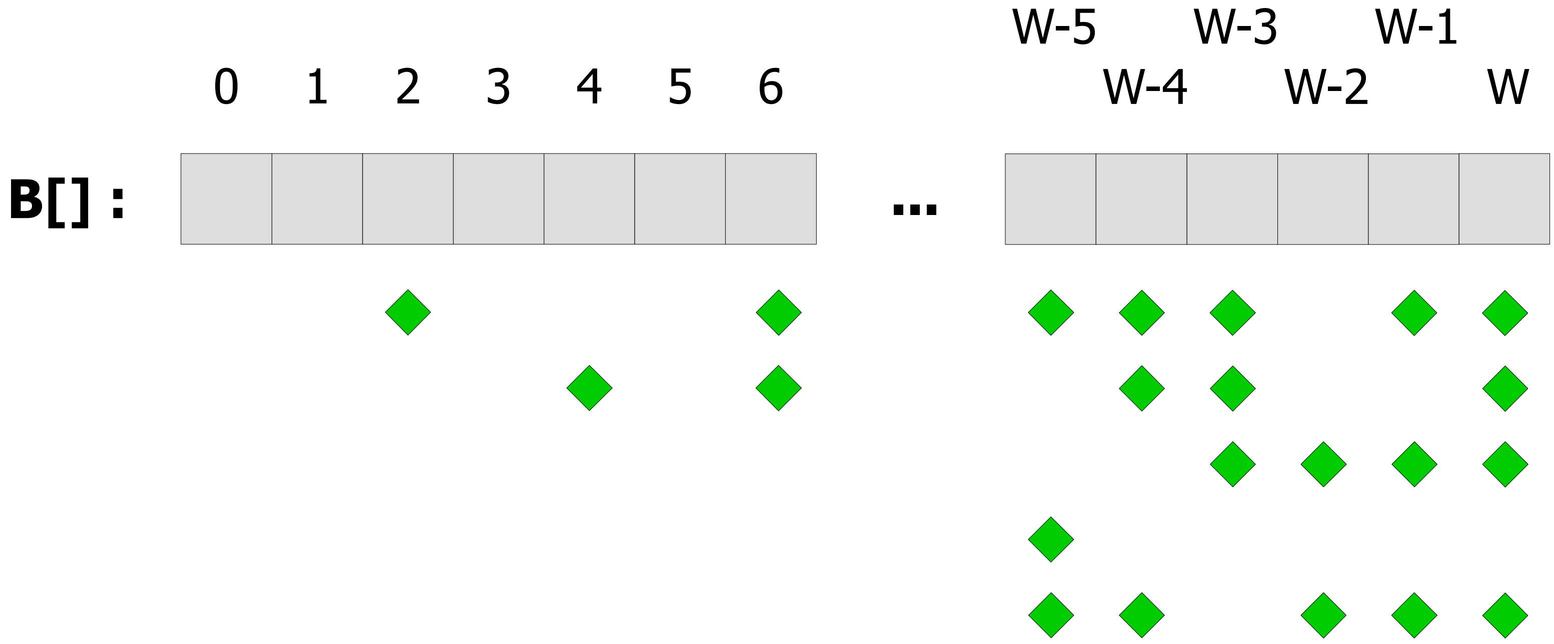
Solution of 0-1 KP

No we can't because when calculating some **B[x]**, we must know if which items were already used:



Solution of 0-1 KP

Then, maybe for every **B[x]** we should <u>also store the exact set of items</u> used there?

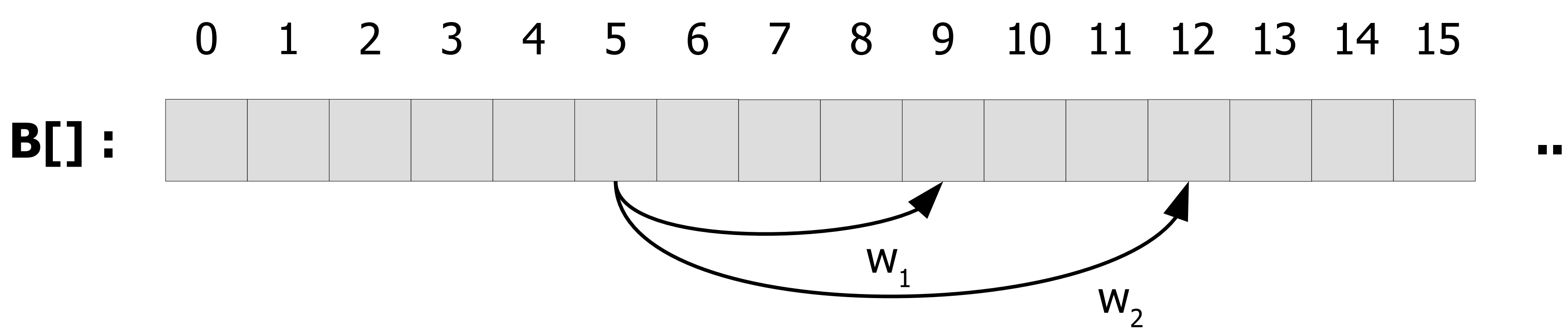


Solution of 0-1 KP

Then, maybe for every **B[x]** we should also store the exact set of items used there?

No, because some sets of items can be more preferable than others.

- For example, **B[5]** can be used for both **B[9]** and **B[12]**,
- For being used in **B[9]**, its set should not contain **(w_1,b_1)**,
- For being used in **B[12]**, its set should not contain **(w_2,b_2)**,
- So we need to keep all the sets, which give maximal **B[x]** then...



Solution of 0-1 KP

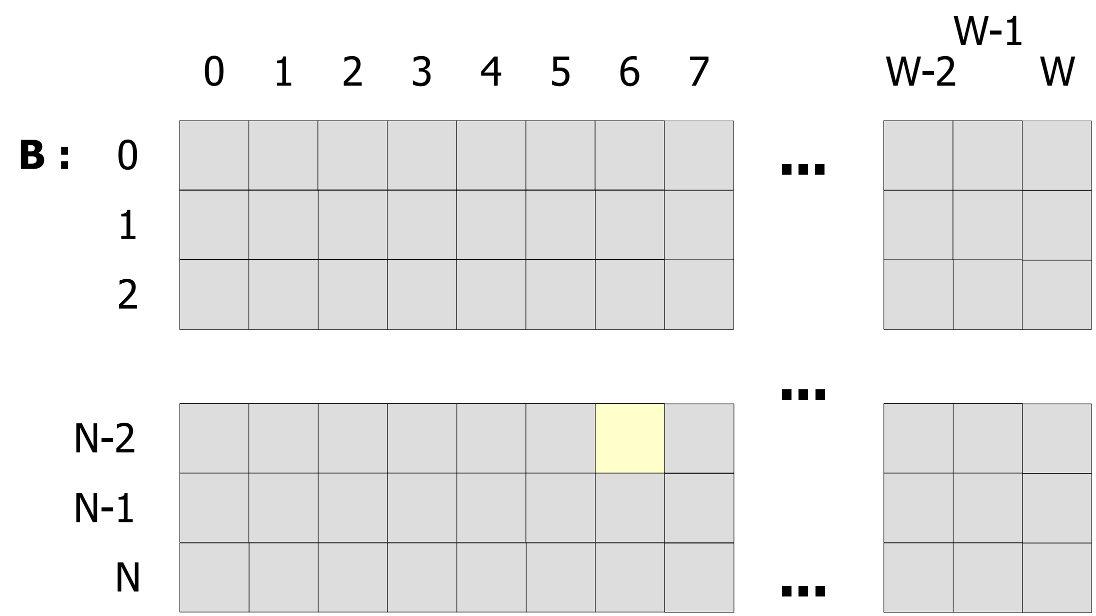
Obviously, this leads to a waste of computational time and memory.

... so probably we can't behave here the same way, as we did for UKP.

Solution of 0-1 KP

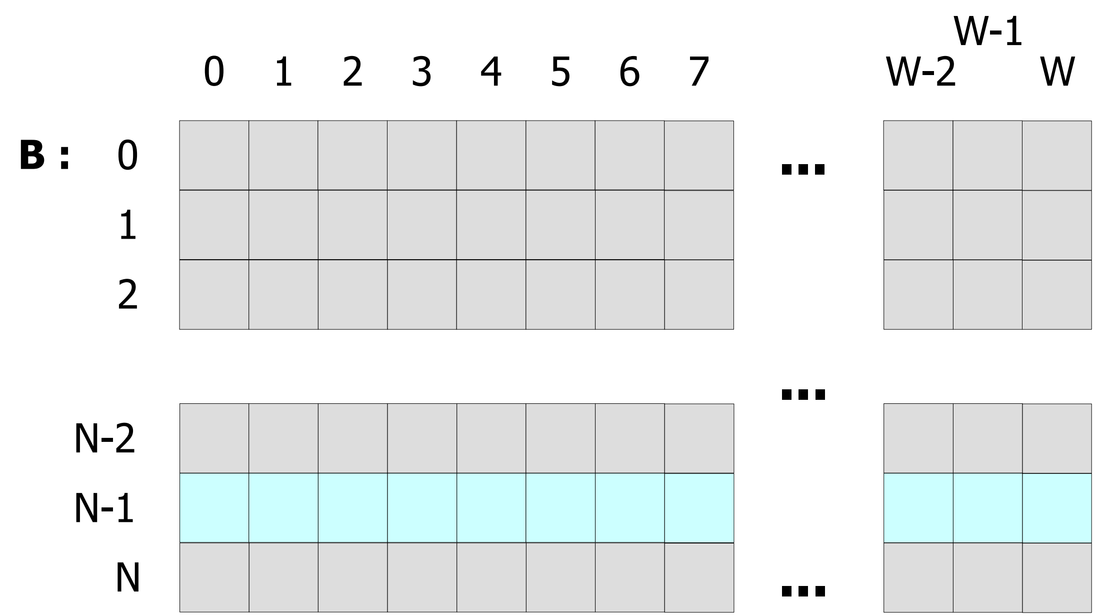
Let's <u>reduce the problem</u> now not only by weight '**W**', but also by number of used items '**k**'.

- so here instead of an array we will have a matrix "**B[0..N][0..W]**",
- where "**B[k][x]**" will show the maximal bonus that we can place in '**x**' weight, using only first '**k**' items.



Solution of 0-1 KP

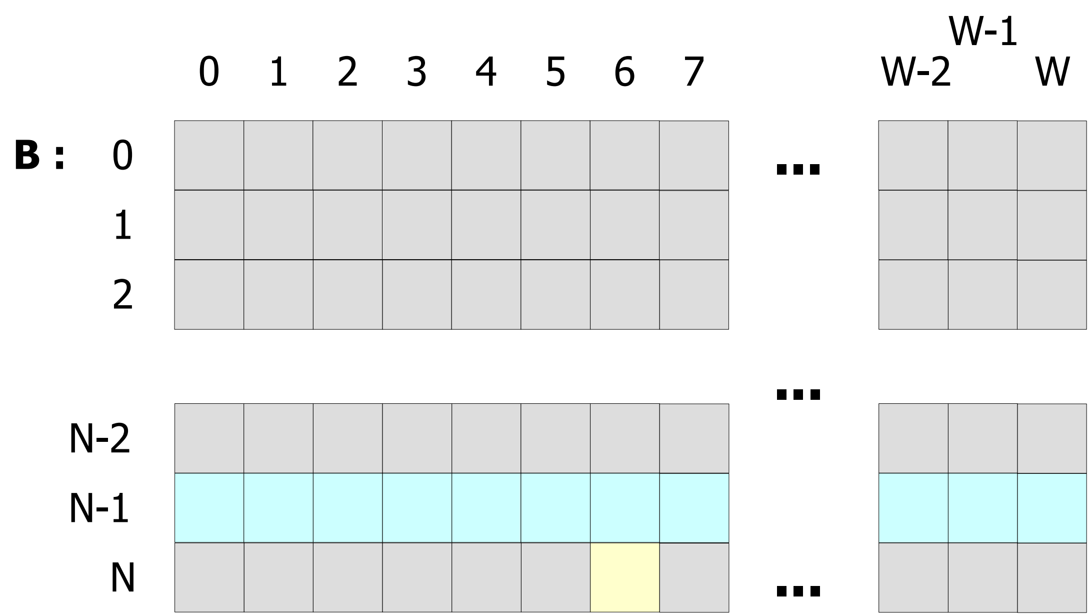
Assume we have calculated the <u>one-before-last row</u>, i.e. we know maximal bonuses for **[0..W]** knapsacks, when using first **N-1** items.



Solution of 0-1 KP

Assume we have calculated the <u>one-before-last row</u>, i.e. we know maximal bonuses for **[0..W]** knapsacks, when using first **N-1** items.

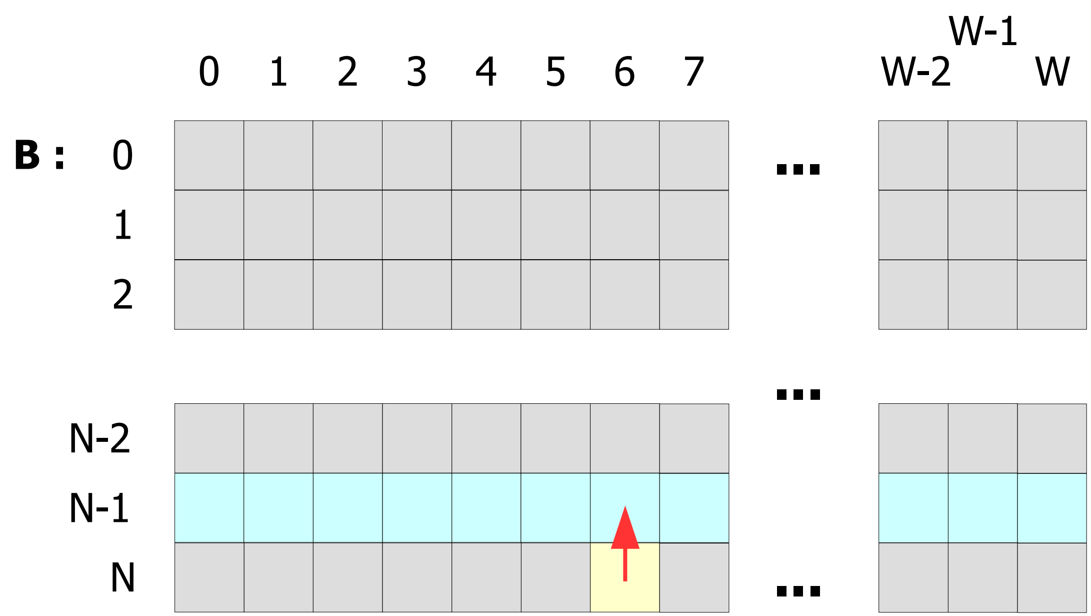
Then the **N**'th item arrives. <u>How it can affect</u> current solutions? What will cells of the last row be equal to?



Solution of 0-1 KP

If we are allowed to use all the **N** items, current solution will either use the **N**'th item, or it will not.

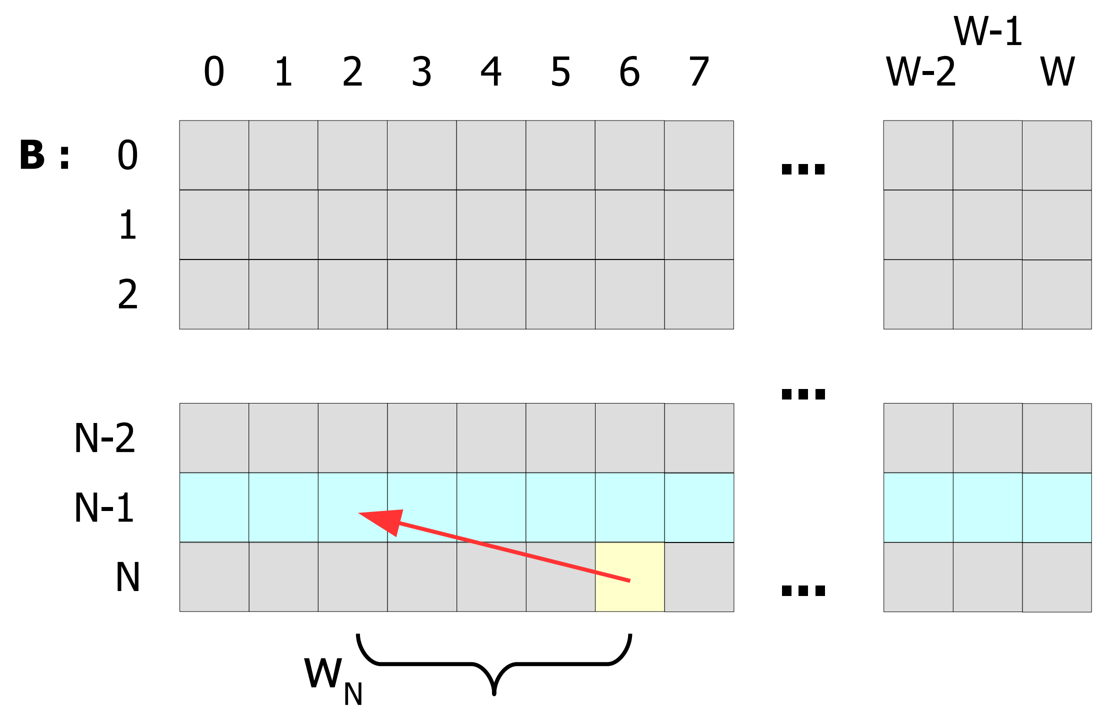
- If it doesn't use the **N**'th item, then **B[N][x] = B[N-1][x].**



Solution of 0-1 KP

If we are allowed to use all the **N** items, current solution will either use the **N**'th item, or it will not.

- If it doesn't use the **N**'th item, then **B[N][x] = B[N-1][x].**
- Otherwise, we reduce our capacity to "$x\text{-}w_N$", and the answer becomes: $\mathbf{B[N][x] = B[N-1][x-w_N] + b_N}$.

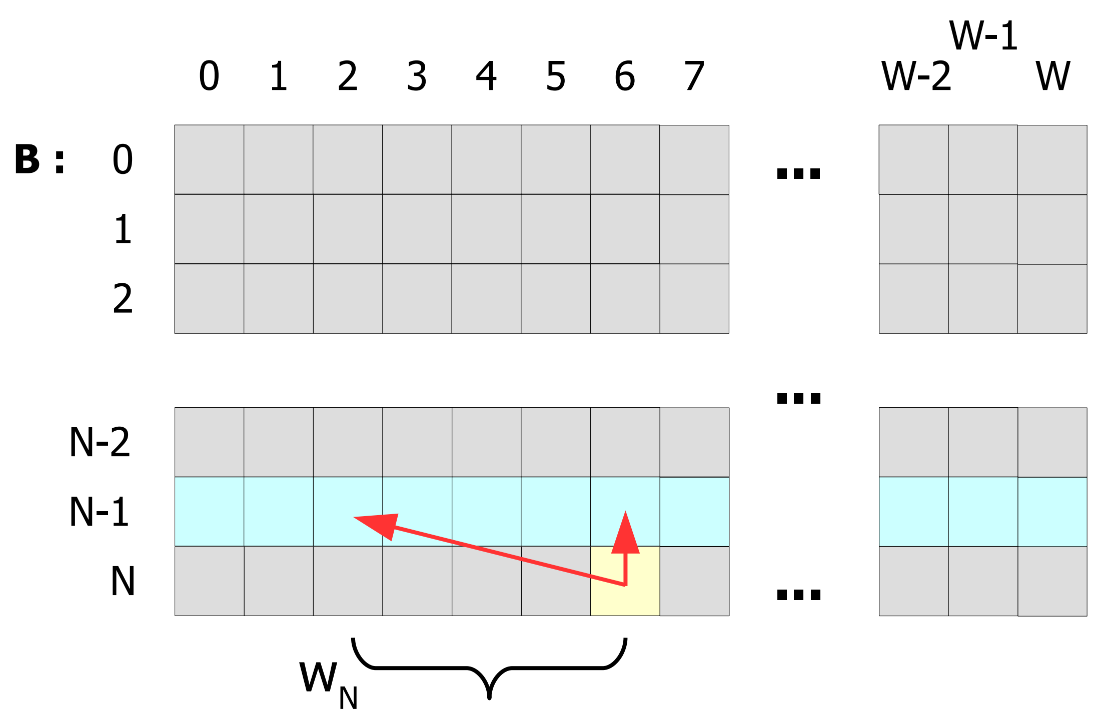


Solution of 0-1 KP

If we are allowed to use all the **N** items, current solution will either use the **N**'th item, or it will not.

- If it doesn't use the **N**'th item, then **B[N][x] = B[N-1][x].**
- Otherwise, we reduce our capacity to "x-w_N", and the answer becomes: **B[N][x] = B[N-1][x-w_N] + b_N**.

So it just remains to <u>choose between this **2** options</u>.

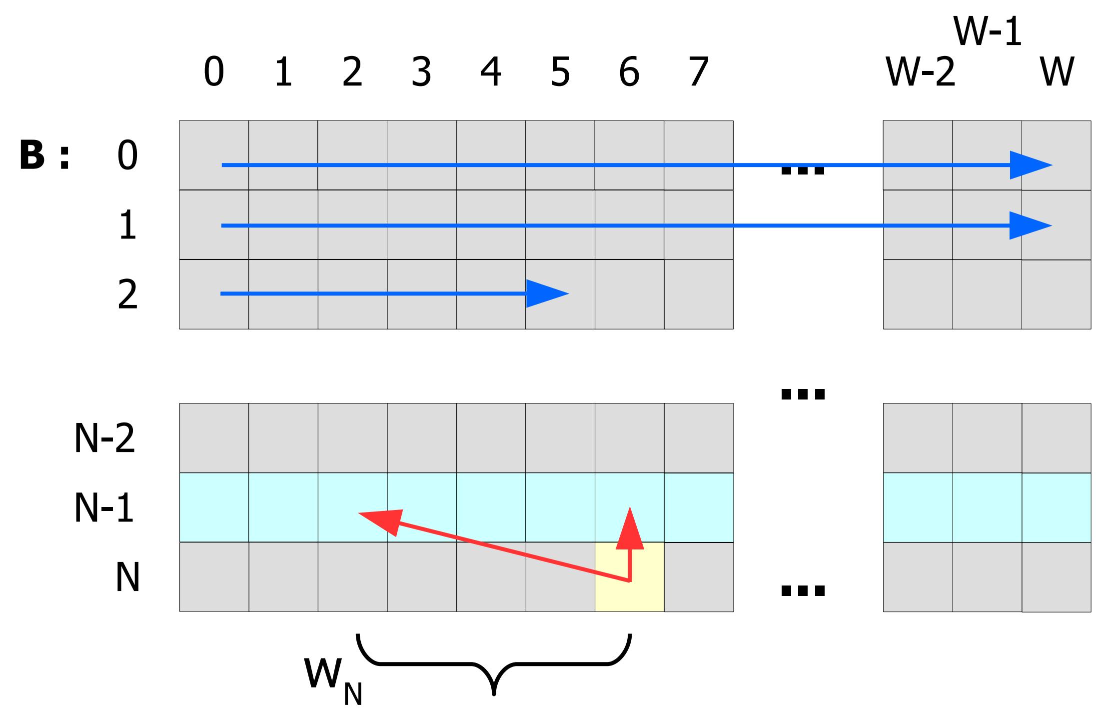


Solution of 0-1 KP

The formula for 0-1 KP becomes:

```
B[k][x] = max(
    B[k-1][x],
    B[k-1][x-wk] + bk )
```

And we can iterate over the matrix, filling every cell in **O(1)** time.



Solution of 0-1 KP

The pseudocode becomes:

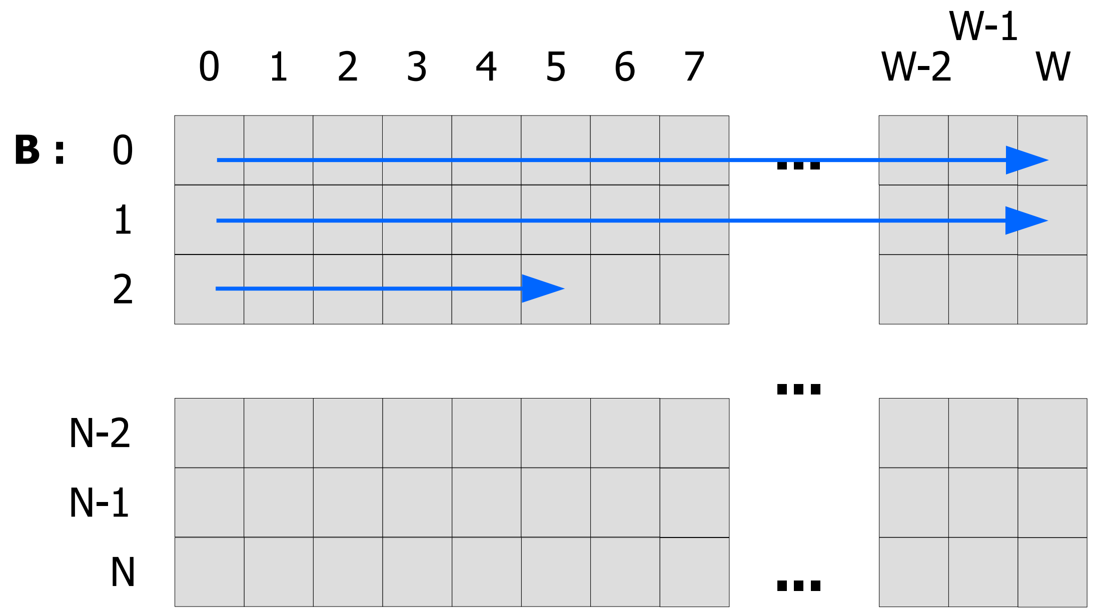
```
N, W : Integer,
w[1..N], b[1..N] : Array of Integers,
B[0..N][0..W] = {0} : Matrix of Integers,

procedure calculate_0_1_KP()
    // First row is already zeroes.
    // First column is also already zeroes.
    for k:=1 to N
        for x:=1 to W
            B[k][x] := B[k-1][x]  // If we don't use k-th
            if x >= w[k]      // If we can use k-th item
                B[k][x] := max(
                        B[k][x],
                        B[k-1][x-w[k]] + b[k] )
```

Solution of 0-1 KP

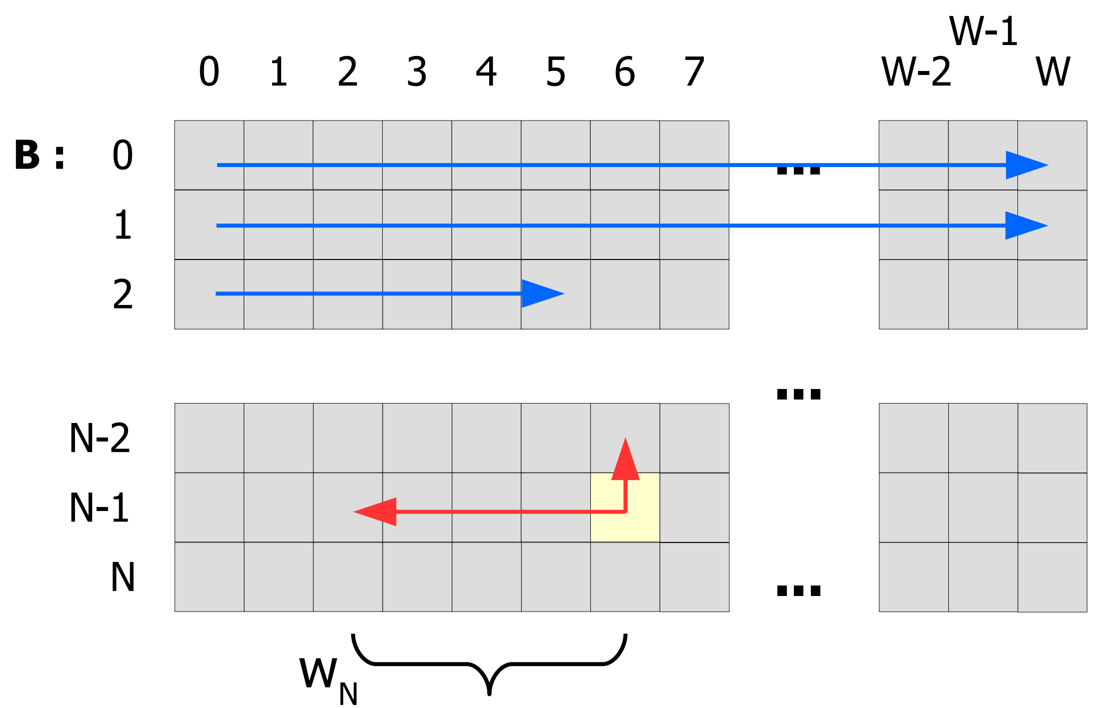
Time and memory complexity of the DP algorithm becomes **O(N*W)**,

... as we fill every cell in **O(1)** time.



Solution of 0-1 KP

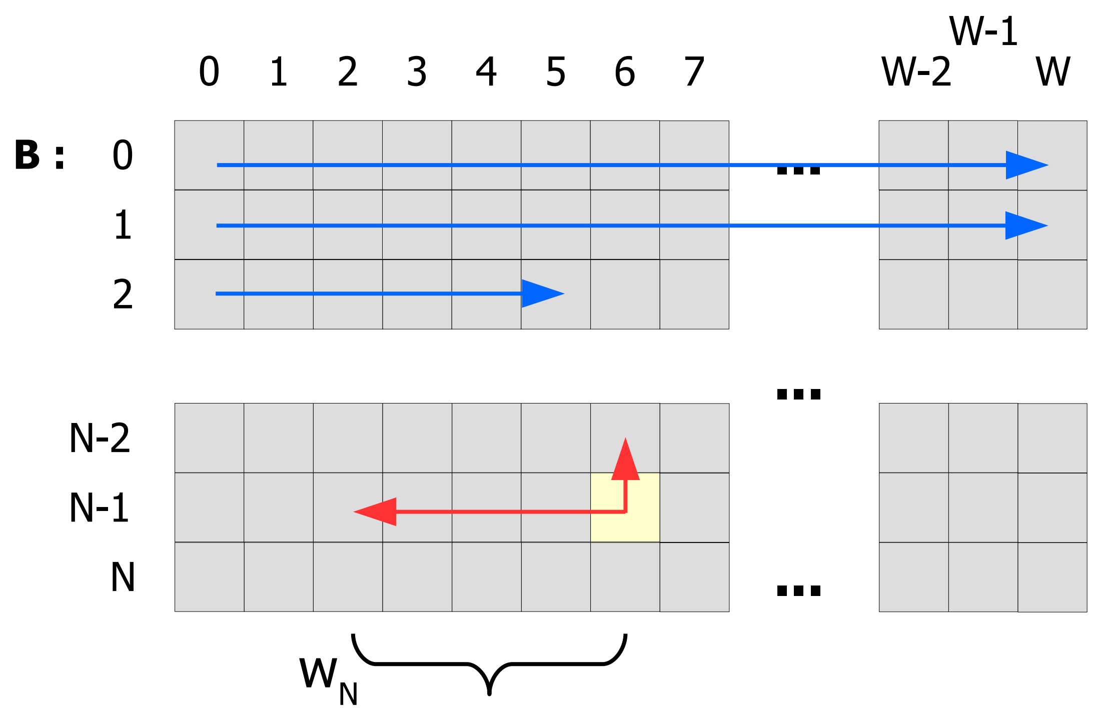
Question: What will happen if we will pick second option not from previous row, but from the current one?



Solution of 0-1 KP

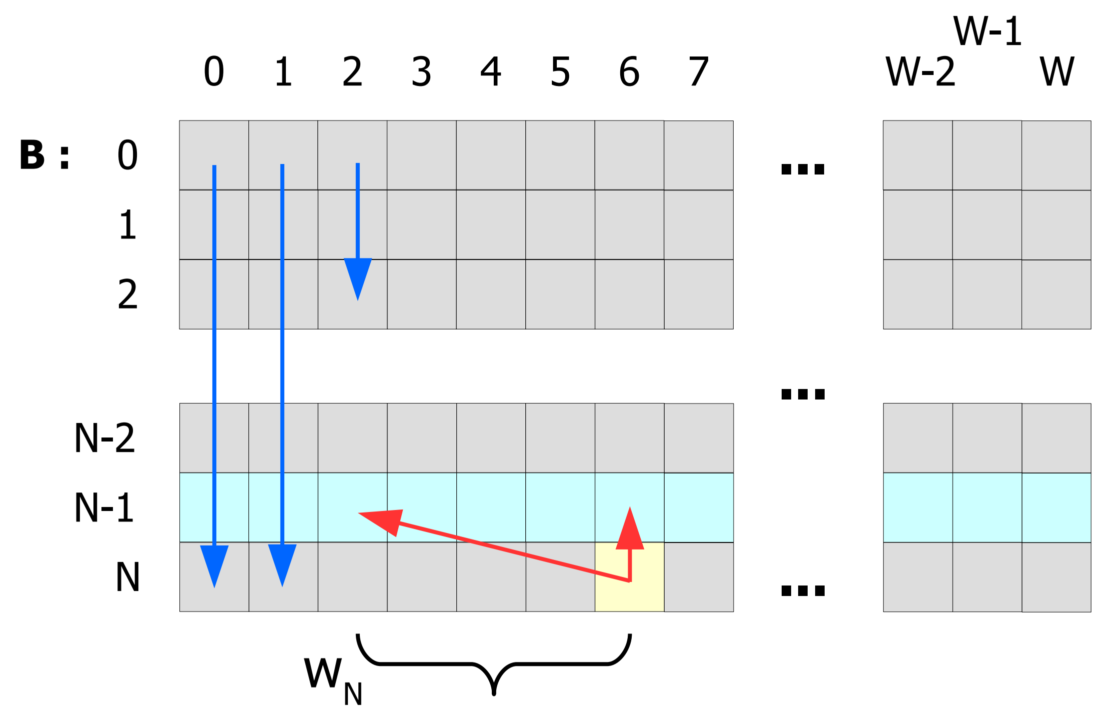
Question: What will happen if we will pick second option not from previous row, but from the current one?

Answer: We will receive solution of UKP, as the same **k**'th item can be used several times then.



Solution of 0-1 KP

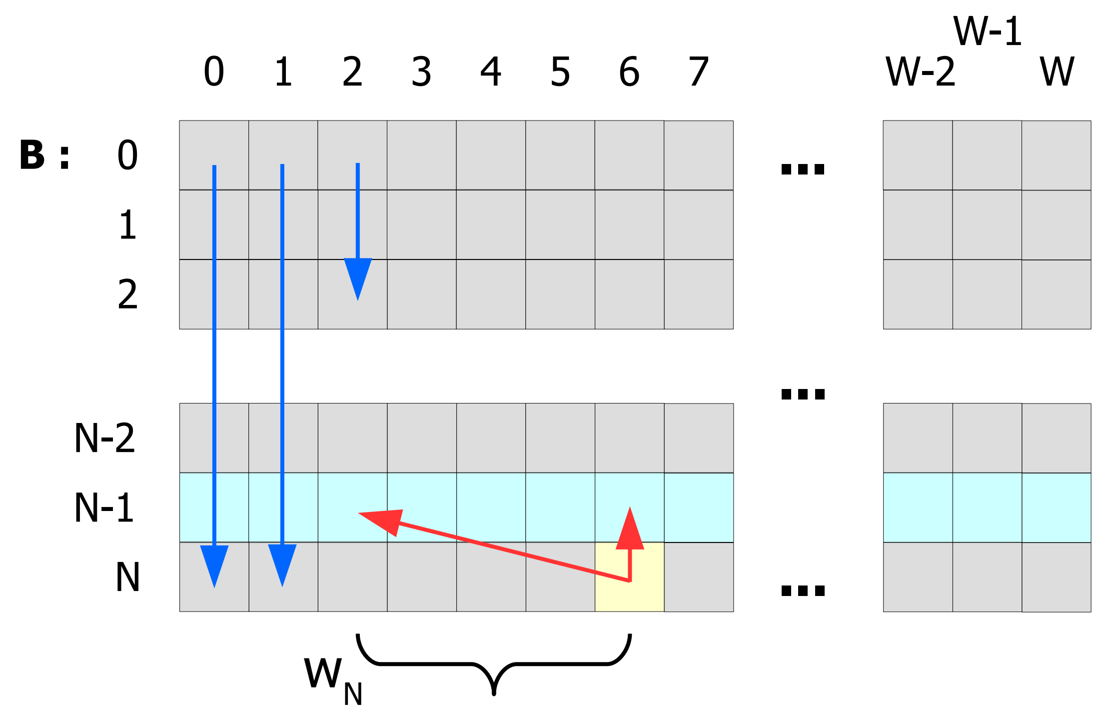
Question: Can we iterate over the cells in the other direction?



Solution of 0-1 KP

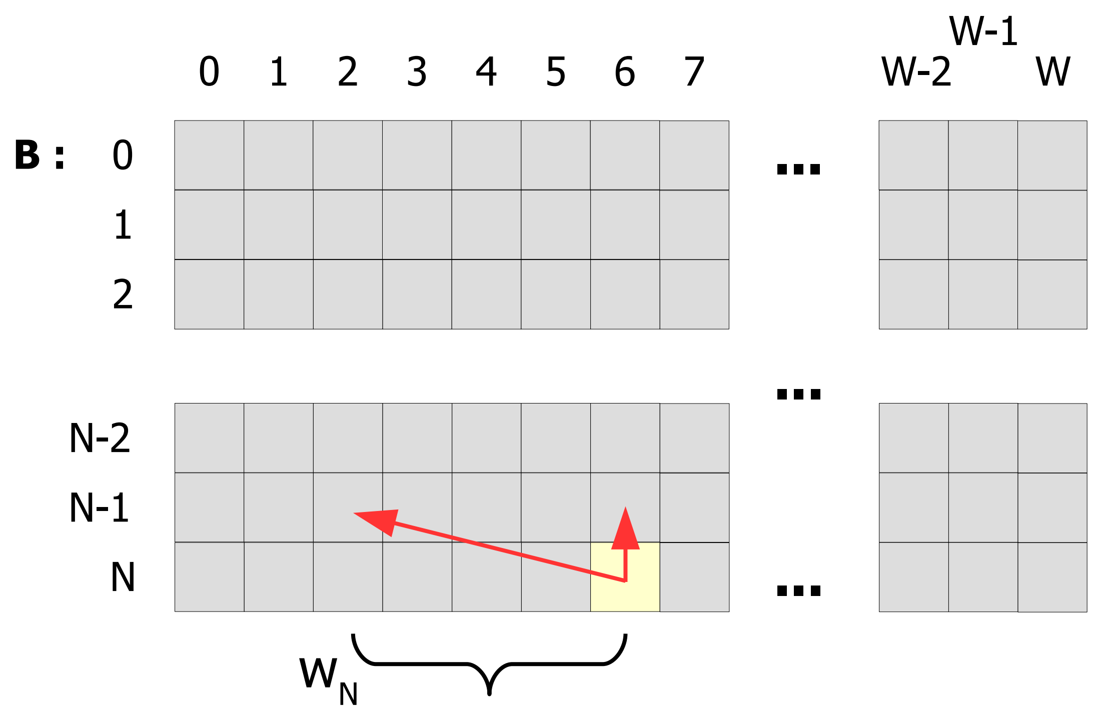
Question: Can we iterate over the cells in the other direction?

Answer: Yes, as the dependencies are not violated.



Solution of 0-1 KP

Question: How is it more preferable to use this problem, by DP or by memoization?

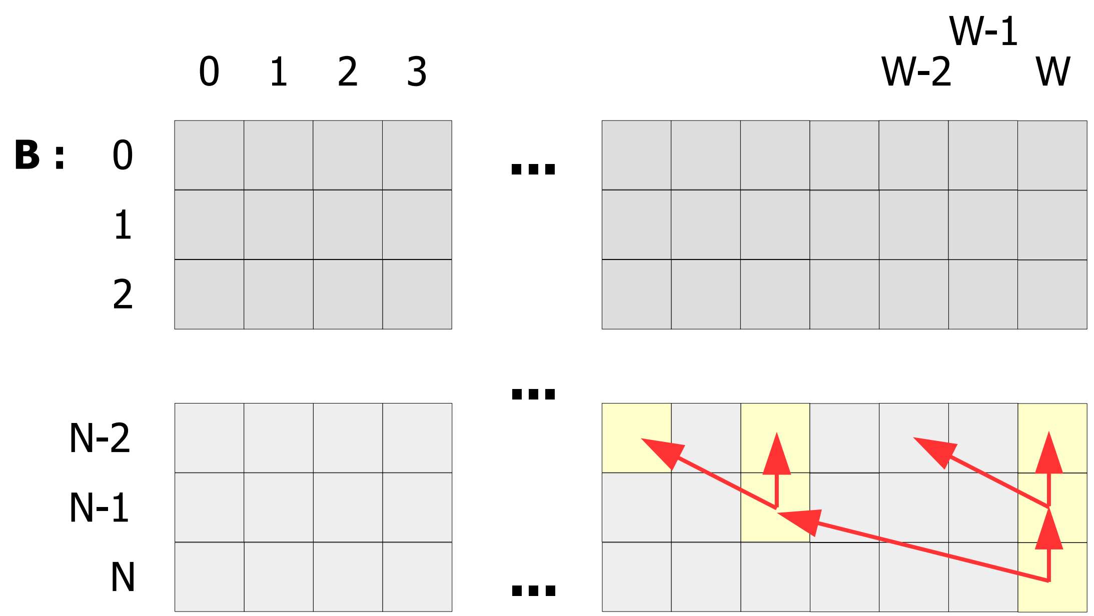


Solution of 0-1 KP

Question: How is it more preferable to use this problem, by DP or by memoization?

Answer: Memoization can be preferable, as it might significantly decrease number of calculated cells,

... because we are interested only in **B[N][W]**.



Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

Thank you!