

# The Online Knapsack Problem

Term Paper - Theoretical Computer Science

Marco Wettstein, ZHAW

2015-06-04

## Contents

<b>1 Preamble</b>	<b>3</b>
1.1 About this paper . . . . .	3
<b>2 The knapsack problem</b>	<b>4</b>
2.1 The simple-knapsack problem . . . . .	5
2.2 The Online-Knapsack-Problem . . . . .	5
<b>3 Online-Algorithm with advice</b>	<b>8</b>
3.1 Optimal online algorithm with advice . . . . .	8
3.2 1 Advice bit . . . . .	9
<b>4 Randomized online algorithms</b>	<b>10</b>
4.1 RONE - AONE with random advice bit . . . . .	11
4.2 2-competiveness with 1 advice bit . . . . .	13
4.3 The limit . . . . .	14
<b>5 Whats next</b>	<b>14</b>
<b>6 Setup</b>	<b>15</b>
6.1 Create items . . . . .	15
6.2 Experiment templates . . . . .	15
6.3 Initialize and reset experiment . . . . .	16
6.4 Running the experiment . . . . .	17
6.5 Chart . . . . .	17
6.6 GUI for the Ticker . . . . .	18
<b>A Additional source files</b>	<b>19</b>
A.1 client/app.jade (template) . . . . .	19
A.2 client/app.styl (stylesheet) . . . . .	22
<b>B Code &amp; Live-Version</b>	<b>23</b>
<b>References</b>	<b>23</b>

# 1 Preamble

Online problems and algorithms are problems where the inputs for the algorithm are not known at the beginning, but appear one by one. This could be a job-scheduler in an operating system where you have to decide whether to do a job immediately after it appears or wait for other, maybe shorter jobs. It can be a memory-management that handles paging (also operation systems).

It could also be the decision whether to buy new ski-gear at the beginning of the season or rent it every day, when you do not know, how the weather will be like during the season. Every day, it could snow, rain or be a perfect powder-day, but should you buy skis on one sunny day, when you do not know if there will be another nice day to go to the mountains?

It would be nice to have some information about the future, something like an omniscient oracle, that gives us a glimpse of whats coming next.

We introduce such an oracle for online problems and try to find out, how much information do we need from this oracle to get an optimal solution.

In this interactive paper, we deal with the so called online simple knapsack problem, where we have knapsack that we want to fill with a maximum amount of value but respect the maximum capacity of it.

## 1.1 About this paper

This paper is written as *literate CoffeeScript*-source-code<sup>1</sup> of a set of experiments with these online problem running on *Meteor*<sup>2</sup>.

It is compiled using *pandoc*<sup>3</sup>, which enables you to compile Markdown and many other formats to latex+pdf.

A live version of the experiments is available at: <http://online-knapsack.macrozone.ch>, the source-code is available on github: [https://github.com/macrozone/seminar\\_np](https://github.com/macrozone/seminar_np).

---

<sup>1</sup>See <http://coffeescript.org/#literate>

<sup>2</sup><https://www.meteor.com/>

<sup>3</sup><http://pandoc.org/>

## 2 The knapsack problem

Consider a knapsack with a certain capacity of weight (or volume) and a set of items, each with a value and a weight.

Which subset of these items would you put into the knapsack to get the maximum possible total value respecting the capacity of the knapsack?

This question is the so called knapsack problem.

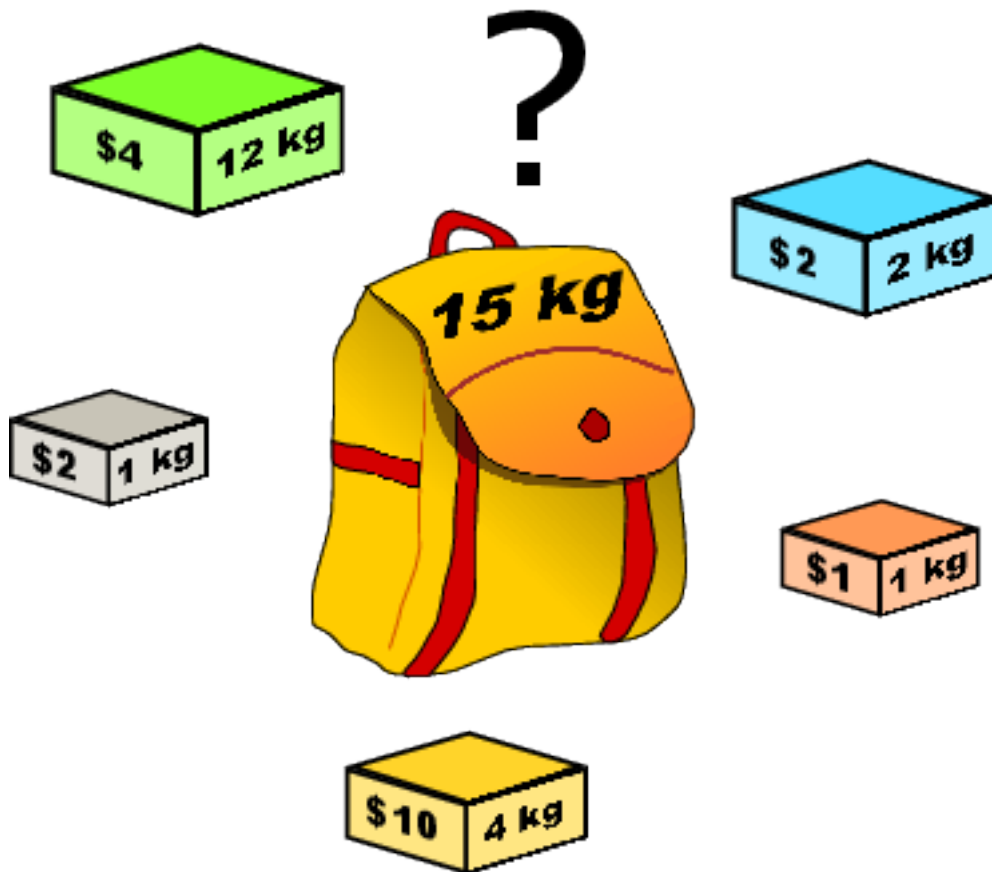


Figure 1: The knapsack problem (Source: wikipedia)

## 2.1 The simple-knapsack problem

In this paper, we only consider the so called *simple-knapsack problem* where the value of one item is the same as its weight and where the knapsack has always a capacity of 1.

We call the total value of all items in the knapsack as the *gain*.

Let's define such a knapsack:

```

1  Knapsack = class
2      constructor: ->
3          @size = 1
4          @dep = new Tracker.Dependency
5          @reset()
6
7      fits: (item) ->
8          @gain() + item.value <= @size
9
10     addItem: (item) ->
11         if @fits item
12             @items.push item
13             @dep.changed()
14
15     gain: ->
16         roundValue _.reduce @getItems(), ((total, item) -> total+item.
17             value), 0
18
19     getItems: ->
20         @dep.depend()
21         @items
22
23     reset: ->
24         @items = []
25         @dep.changed()

```

## 2.2 The Online-Knapsack-Problem

In the former *offline*-knapsack problem, we know all items that we want to put in the knapsack. In the *online*-version of this problem, we do not know every item, but get the items one by one. We therefore have to decide after every item, whether we put the item in the knapsack or not.

We create a base-algorithm for that:

```

1  Algorithm = class
2      constructor: ->
3          @act = new ReactiveVar
4          @_knapsack = new Knapsack
5      knapsack: -> @_knapsack
6
7      handle: (item) ->
8          if @decide item
9              @_knapsack.addItem item
10             return yes
11          else
12              return no
13      decide: (item) ->
14          # implement me and return yes or no
15
16      reset: ->
17          @_knapsack.reset()

```

```

18     @act.set null
19     doAct: (like) -> @act.set like
20     acts: (like) -> @act.get() is like

```

What maximum gain would we can achieve and how would an online-algorithm perform in comparison with an optimal offline-algorithm, which would know every item?

Let's try out.

```

1 experiments = []

```

Lets start with the greedy approach. Here, we just take every item we get, if it fits:

```

1 decideGreedy = (item) -> if @knapsack().fits item then yes else no

```

and we define an algorithm with it:

```

1 Greedy = class extends Algorithm
2     decide: decideGreedy

```

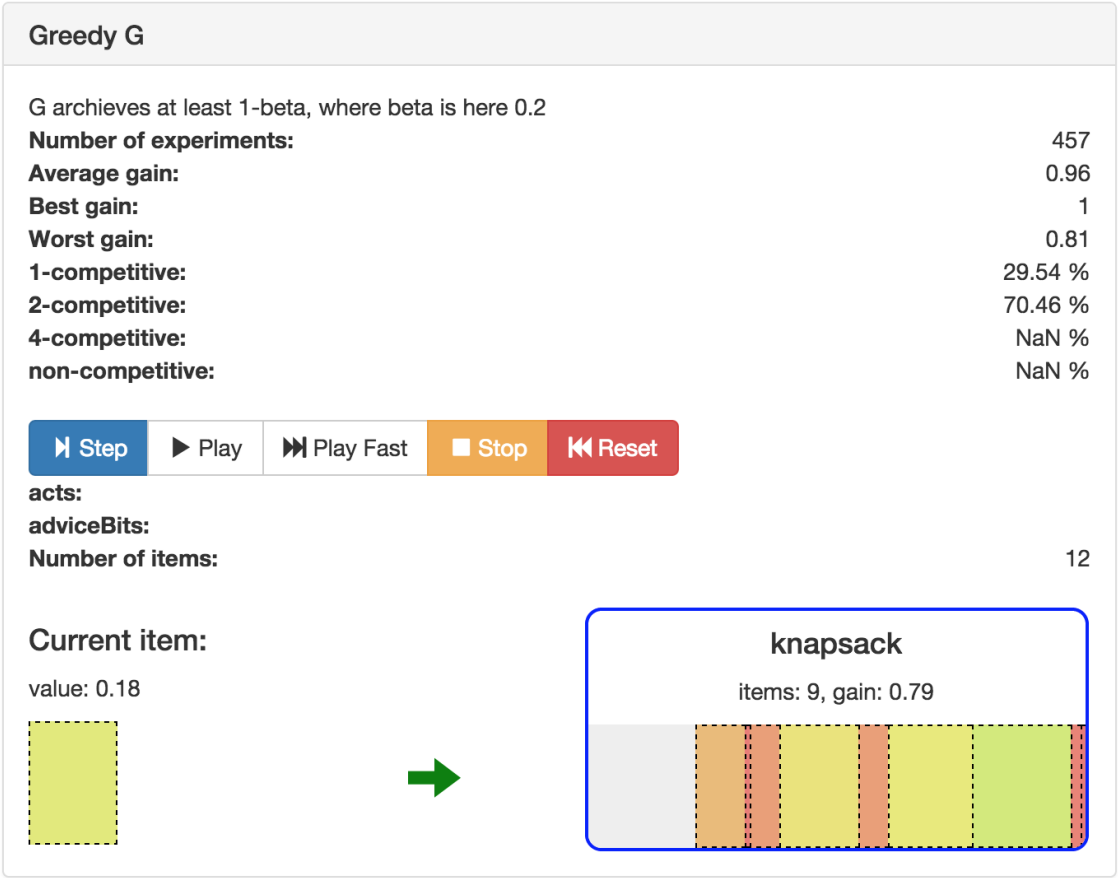
The gain of this algorithm is at least  $1 - \beta$ , where  $\beta$  is the size of the item with the highest value (weight). The proof is simple: if we get this item with value  $\beta$ , the gain is certainly higher than  $\beta$ . If this item does not fit anymore in the knapsack, we will have at least  $1 - \beta$  gain.

Lets do some experiments with it to verify this:

```

1 experiments.push
2     name: -> "Greedy G"
3     description: -> "G achieves at least 1-beta, where beta is here
4         #{@beta}"
5     beta: 0.5
6     Algorithm: Greedy
7
8 experiments.push
9     name: -> "Greedy G"
10    description: -> "G achieves at least 1-beta, where beta is here
11        #{@beta}"
12    beta: 0.2
13    Algorithm: Greedy
14
15 experiments.push
16     name: -> "Greedy G"
17     description: -> "G achieves at least 1-beta, where beta is here
18         #{@beta}"
19     beta: 0.8
20     Algorithm: Greedy

```



### 3 Online-Algorithm with advice

Imaging you had an oracle, that would know all items that will come. How many bits of information from this oracle would you need to get an optimal solution? And for a given amount of these advice bits, how good would your algorithm perform?

We define such an algorithm as *online algorithm with advice*.

Let  $I$  be an input of such an online algorithm  $A$  and  $\Phi$  an (infinite) sequence of bits (1 or 0), called \*advice bits. The online-algorithm can read a finit prefix of this sequence.

The gain of this Algorithm is  $gain(A^\Phi(I))$ .

If we have  $n$  items in a solution and have read  $s(n)$  advice-bits while computing this solution in the algorithm we call  $s(n)$  the *advice-complexity*.

If we compare the *gain* of this algorithm with the gain of an optimal offline algorithm  $OPT$ , we can define its *competitiveness*:

$$gain(A^\Phi(I)) \geq \frac{1}{c} * gain(OPT(I)) - \alpha$$

where  $\alpha$  is a constant and we call this algorithm *c-competitive*. If  $\alpha = 0$ ,  $A$  is *strictly c-competitive*.

The param  $\alpha$  is needed when the length of the input may vary and the algorithm could be bad for short, but perform well for long inputs (See also Komm, 4). For our knapsack problem, we can set  $\alpha = 0$  and only consider *strict-competitiveness*, because the capacity of the knapsack is bound to 1 (Böckenhauer et al., 64).

Let's implement a base class for such an algorithm:

```
1 AlgorithmWithAdvice = class extends Algorithm
2   constructor: ->
3     @adviceBits = new ReactiveVar
4     super
5   askOracle: (items) ->
6     if @oracle?
7       @adviceBits.set @oracle items
8   oracle: (items) ->
9     # implement me and return an array of advice-bits
10  readAdviceBit: (index) ->
11    @adviceBits.get()[index]
12  reset: ->
13    super
14    @adviceBits.set null
```

#### 3.1 Optimal online algorithm with advice

Let's go back to the first question with the first question: how many advice bits do we have to read to get an optimal solution?

Consider an algorithm with an oracle, that would give us a bit for every item coming with

- value 1 if the item is part of the solution
- value 0 if the item does not belong to the solution

Obviously, we need  $n$  bits of advice for that, or  $n-1$ , because for the last item, we can assume that it is part of the optimal solution.

We now define an algorithm for that.

Note: The items are prepared in a way, that some are already marked as solution. That makes it easier to define the oracle here:



```

1 TotalInformation = class extends AlgorithmWithAdvice
2   oracle: (items) ->
3     bits = []
4     for item in items
5       bits[item.index] = if item.isPartOfSolution then 1 else 0
6   # we do not need the last bit
7   bits.pop()
8   return bits

```

The decision is now easy. If we have a bit (yes / no), we use it:

```

1   decide: (item) ->
2     adviceBit = @readAdviceBit item.index
3     if adviceBit? then adviceBit else yes

```

Lets do an experiment with it:

```

1 experimentsWithAdvice = []
2 experimentsWithAdvice.push
3   name: -> "Total Information"
4   beta: 0.4
5   Algorithm: TotalInformation

```

As (Böckenhauer et al., 64) states, any algorithm for the online simple knapsack problem needs at least  $n-1$  bits to be optimal.

## 3.2 1 Advice bit

What's the best gain if we had only 1 advice bit?

Let's do an experiment where we have an oracle that gives us one bit:

```

1 AONE = class extends AlgorithmWithAdvice
2   oracle: (allItems) -> [ _.some allItems, (item) -> item.value >
3     0.5 ] # array with one bit

```

The bit tells us:

- 1: There exists an item with a size  $> 0.5$
- 0: There is no such item

If the bit is 0, the algorithm acts greedy (like before). If the bit is 1, the algorithm waits until the item with size  $> 0.5$  appears and will start acting greedily:

```

1   decide: (item)->
2     adviceBit = @readAdviceBit item.index
3     if adviceBit? # existence
4       if adviceBit is false then @doAct "greedy" else @doAct "
5         wait"
6       if @acts "greedy" then decideGreedy.call @, item else @wait
7         item
8
9   wait: (item) ->
10     if item?.value > 0.5
11       @doAct "greedy"
12       decideGreedy.call @, item
13     else
14       no

```

This algorithm is 2-competitive:

- If there is no item with weight  $> 1/2$ , the gain is at least  $1/2$  as we have already seen in the greedy approach.
- On the other hand if such an item exists, the algorithm will wait for it and put it in, so it will get a gain of at least  $1/2$

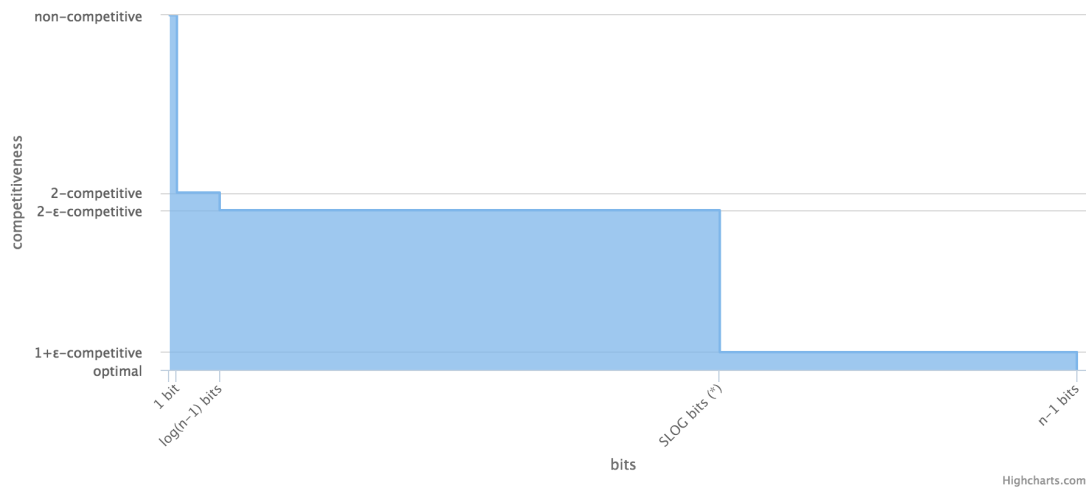
We do an experiment with a max size of one item of 0.55 to verify this:

```
1 experimentsWithAdvice.push
2   name: "AONE - with one advice bit"
3   description: "AONE is 2-competitive"
4   beta: 0.55
5   Algorithm: AONE
```

This one single bit gives us an competitive-ratio of 2, but what happens if we increase the amount of bits? Can we achieve a better ratio?

Unfortunately, more advice bits does not give us a better competitive-ratio, at least for a sub-logarithmic amount  $s(n)$  of advice bits. Figure 3 shows the number of bits compared with the achieved competitive-ratio.

There is a second jump at  $SLOG$ -bits, where competitiveness is  $1 + \varepsilon$ . The proof for these intervals is found in the source (Böckenhauer et al., 65).



$$(*) \text{ SLOG} = \left\lceil \frac{2\varepsilon + 2}{\varepsilon} \right\rceil \cdot \lceil \log n \rceil + 2 \cdot \left\lceil \log \frac{2\varepsilon + 2}{\varepsilon} \right\rceil + 2 \cdot \lceil \log \lceil \log n \rceil \rceil + 1$$

Figure 3: Number of bits VS competitiveness. There are several jumps in the competitive-ratio

## 4 Randomized online algorithms

Obviously in real online problems, we do not have an omniscient oracle. But we can use the idea of the oracle and just guess the advice bits *randomly*.

We can then estimate the competitiveness of this *randomized online-algorithm*.

### 4.1 RONE - AONE with random advice bit

Let's start with AONE from the previous experiment, but guess the adviceBit randomly:

```
1 RONE = class extends AONE
2   oracle: ->
3     [Math.random() < 0.5]
```

If we guess wrong, we might get a lower gain than 0.5 or even 0, if the adviceBit is 1 and we have no item with size > 0.5.

So while we have a 2-competitiveness in AONE, we have here a 4-competitiveness in expectation (in 50% of the cases, we are wrong).

```
1 randomExperiments = []
2 randomExperiments.push
3   name: "RONE - one random bit"
4   description: "Is 4-competitive in expectation"
5   beta: 0.55
6   Algorithm: RONE
```

The experiment does not show this directly, because the items are prepared in a way, that not all possible cases are evenly distributed. We expected that in 50% of the cases, the algorithm would guess wrongly and we would gain nothing, but in the experiment this probability is lower (See 4).

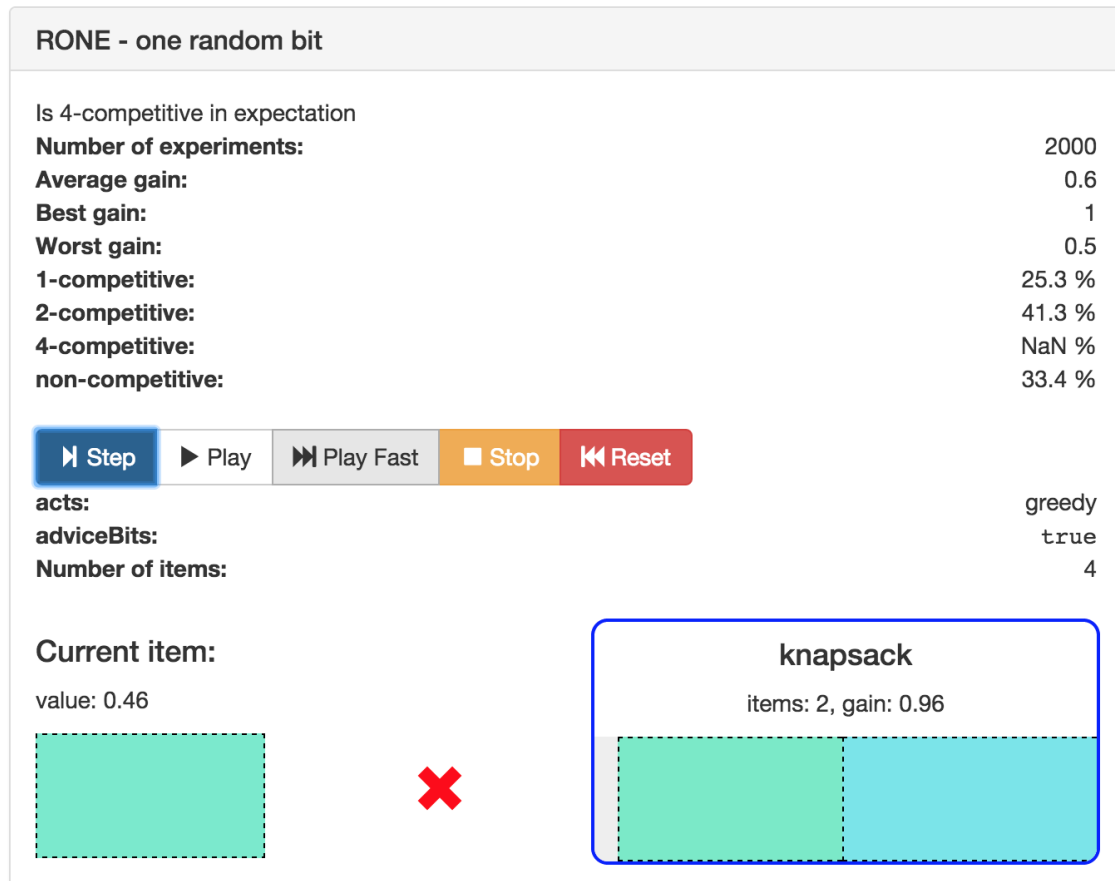


Figure 4: In the experiment with RONE, the probability of gaining nothing is lower than 50%. It's not obvious, that this algorithm is 4-competitive in expectation. In fact, it never performs in the range of a 4-competitive-algorithm (from 0.25-0.5 gain), because it either performs better than 0.5 (2-competitive) or gains nothing. But in expectation (considering the random-variable) the competitive-ratio would be 4.

## 4.2 2-competitiveness with 1 advice bit

The competitive-ratio of 4 is somewhat obvious. But surprisingly, we can also achieve a ratio of 2 with only 1 advice bit.

Consider an algorithm that chooses randomly between two algorithms A1 and A2. A1 is the greedy approach we already know:

```
1 A1 = Greedy
```

A2 internally simulates A1 at the beginning:

```
1 A2 = class extends Algorithm
2   reset: ->
3     super
4     @a1 = new A1
5     @doAct "simulateA1"
```

To decide whether it will use the item or not, it first offers it to the simulated A1-Algorithm. As soon as A1 won't take the item anymore (A1's knapsack is full), A2 starts to act greedily:

```
1   decide: (item) ->
2     if @acts "simulateA1"
3       if @a1.handle item
4         return no
5       else
6         @doAct "greedy"
7         return @decide item
8     else if @acts "greedy"
9       return decideGreedy.call @, item
```

We now compose an algorithm "RONE2", that chooses randomly between A1 and A2:

```
1 RONE2 = class extends AlgorithmWithAdvice
2   constructor: ->
3     @a1 = new A1
4     @a2 = new A2
5     super
6   oracle: -> [Math.random() < 0.5]
7   reset: ->
8     super
9     @a1.reset()
10    @a2.reset()
11   knapsack: -> @algorithm().knapsack()
12   # handle decides and put the item in the knapsack
13   handle: (item) ->
14     adviceBit = @readAdviceBit item.index
15     if adviceBit? # existence of the first bit
16       if adviceBit then @doAct "A1" else @doAct "A2"
17     @algorithm().handle item
18   algorithm: ->
19     if @acts "A1" then @a1 else @a2
```

We do now an experiment with it:

```
1 randomExperiments.push
2   name: "RONE2 - one random bit"
3   description: "Is 2-competitive in expectation"
4   beta: 0.55
5   Algorithm: RONE2
```

To show that this algorithm is 2-competitive in expectation, we consider two cases:

- If the sum of all items is less than the knapsack's capacity, A1 is optimal, while A2 gains 0. Because we chose randomly between the two algorithm, we have a 50% chance to get an optimal gain (or to get 0).
- If the sum is greater, the total gain of A1 and A2 is at least 1. Because we chose randomly between the two, we get a 0.5 gain in expectation.

Considering both cases, we get a gain of 0.5 in expectation, so the algorithm is 2-competitive.

### 4.3 The limit

While we can achieve different levels of competitiveness by increasing the number of bits in online algorithms with advice, this is not the case in randomized online algorithms.

As (Böckenhauer et al.) states, there is no algorithm that performs better than 2-competitive in expectation. So 2-competitiveness with 1 bit is the best we can achieve.

## 5 Whats next

**Resource augmentation:** If we allow the online algorithm to pack a little bit more ( $\delta$ ) in the knapsack than allowed, we can achieve up to  $(2 - \delta)$ -competitiveness.

**The weighted case:** In this paper, we only considered items, where the value is equal to the weight of the item. If we introduce a different weight of each item, we will see, that online algorithms for this weighted knapsack problem is only competitive for at least a logarithmic amount of advice bits.

**Randomized online algorithms for the weighted knapsack:** If we create a randomized online algorithm for the weighted case, we see that these algorithms are not competitive at all, with and without resource-augmentation.

Further details and proof for these extensions can be found in the source (Böckenhauer et al.) .

## 6 Setup

The following code sets the experiments up and defines templates, etc.

First, define some constants and helpers:

```
1 Constants =  
2     SCALE: 300  
3  
4 roundValue = (value) -> Math.round(value*100)/100
```

### 6.1 Create items

The creation of items is done here. The items are prepared in a way, so that we know which elements are part of the solution (for experiment “Total information”). This makes it easier to setup a oracle later. Of course, we should not use this information directly when deciding an item.

```
1 createItems = ({beta, maxSize}) ->  
2     items = []  
3     beta ?= 0.5  
4     maxSize ?= 1  
5     totalSize = 0  
6  
7     loop  
8         randomValue = -> roundValue Math.random()*beta  
9         value = randomValue()  
10        if totalSize+value < maxSize  
11            totalSize += value  
12            items.push {value, isPartOfSolution: yes}  
13        else  
14            # add one that fits exactly  
15            items.push  
16                value: roundValue maxSize - totalSize  
17                isPartOfSolution: yes  
18            # add the one that does not fit  
19            items.push {value}  
20            break  
21  
22        items = _.shuffle items  
23        for item, index in items  
24            item.index = index  
25        # we later pop the elements out (from the end) because it is  
26        # faster. So we reverse here:  
27        return items.reverse()
```

### 6.2 Experiment templates

Add the experiments to the template:

```
1 Template.experiments.helpers  
2     experiments: -> experiments  
3     experimentsWithAdvice: -> experimentsWithAdvice  
4     randomExperiments: -> randomExperiments
```

Initialize it. We use some ReactiveVars to store the state of the experiment on the template instance.

```

1 Template.Experiment.onCreated ->
2   @items = []
3   @currentItem = new ReactiveVar
4   @numberOfItems = new ReactiveVar
5   @algorithm = new @data.Algorithm

```

Lets define a history, where we can read some stats about the experiments from:

```

1   @gainHistory =
2     history: []
3     dep: new Tracker.Dependency
4     add: (gainValue) ->
5       if gainValue > 0
6         @worstGain = Math.min @worstGain ? gainValue,
          gainValue
7         @bestGain = Math.max @bestGain ? gainValue, gainValue
8         @history.push gainValue
9         @dep.changed()
10    size: ->
11      @dep.depend()
12      @history.length
13    worst: ->
14      @dep.depend()
15      @worstGain
16    best: ->
17      @dep.depend()
18      @bestGain
19    competitiveCount: ->
20      @dep.depend()
21      _.countBy @history, (value) ->
22        if value is 1
23          "1-competitive"
24        else if 0.5 <= value < 1
25          "2-competitive"
26        else if 0.25 <= value < 0.5
27          "4-competitive"
28        else
29          "non-competitive"
30
31    competitivePercentage: (cGroup)->
32      @dep.depend()
33      if @history.length > 0
34        roundValue 100 * @competitiveCount()[cGroup] /
          @history.length
35    avg: ->
36      @dep.depend()
37      if @history.length > 0
38        roundValue (_.reduce @history, (total, value) -> total
          +value)/@history.length
39    reset: ->
40      @history = []
41      @bestGain = null
42      @worstGain = null
43      @dep.changed()

```

### 6.3 Initialize and reset experiment

```

1   resetExperiment = =>

```



```

2      @items = createItems beta: @data.beta
3      @algorithm.reset?()
4      @algorithm.askOracle? @items
5      @numberOfItems.set @items.length
6      @currentItem.set @items.pop()
7
8      do reset = =>
9          @gainHistory.reset()
10         resetExperiment()

```

## 6.4 Running the experiment

The Ticker is a package, that can run a callback in a loop. We can run it step-by-step or fast.

```

1      @ticker = new Ticker
2      reset: => reset()
3      turn: =>
4          item = @currentItem.get()
5          if item?
6              @algorithm.handle item
7              @currentItem.set @items.pop()
8          else
9              # no more items
10             @gainHistory.add @algorithm.knapsack().gain()
11             resetExperiment()

```

Expose the state to the template:

```

1  Template.Experiment.helpers
2      adviceBits: -> Template.instance().algorithm.adviceBits?.get()
3      act: -> Template.instance().algorithm.act.get()
4      knapsack: -> Template.instance().algorithm.knapsack()
5      ticker: -> Template.instance().ticker
6      currentItem: -> Template.instance().currentItem?.get()
7      gainHistory: -> Template.instance().gainHistory
8      numberOfItems: -> Template.instance().numberOfItems.get()
9      willMatch: ->
10         ctx = Template.instance()
11         ctx.currentItem?.get()?.value + ctx.algorithm.knapsack().gain
12         () <= ctx.algorithm.knapsack().size
13
14  Template.Knapsack.helpers
15      totalWidth: ->
16         @size * Constants.SCALE + 2
17      items: ->
18         @getItems()
19
20  Template.KnapsackItem.helpers
21      width: ->
22         @value * Constants.SCALE
23      color: ->
24         hue = @value*360
25         "hsl(#{hue}, 73%, 69%)"

```

## 6.5 Chart

The chart from 3 is created by this code:

```

1 Template.competitivenessChart.helpers
2   chartObject: ->
3     legend: enabled: false
4     title: text: ""
5     yAxis:
6       title: text: "competitiveness"
7       tickPositioner: -> [1,1.1,1.9,2,3]
8       labels:
9         formatter: ->
10          switch @value
11            when 1 then "optimal"
12            when 1.1 then "1+eps-competitive"
13            when 1.9 then "2-eps-competitive"
14            when 2 then "2-competitive"
15            when 3 then "non-competitive"
16
17     xAxis:
18       title: text: "bits"
19       tickPositioner: -> [0,1,7,77,127]
20       labels:
21         rotation: -45
22         formatter: ->
23          switch @value
24            #when 0 then "0 bits"
25            when 1 then "1 bit"
26            when 7 then "log(n-1) bits"
27            when 77 then "SLOG bits (*)"
28            when 127 then "n-1 bits"
29
30     series: [
31       type: "area"
32       step: "left"
33       data: [
34         (x: 0, y: 3, name: "non-competitive")
35         (x: 1, y: 2, name: "2-competitive")
36         (x: 7, y: 1.9, name: "2-eps-competitive")
37         (x: 77, y: 1.1, name: "1+eps-competitive")
38         (x: 127, y: 1, name: "optimal")
39       ]
40     ]

```

## 6.6 GUI for the Ticker

```

1 Template.TickerGui.helpers
2   counter: -> @ticker.getCounter()
3
4 Template.TickerGui.events
5   'click .btn-step': -> @ticker.step()
6   'click .btn-play': ->
7     @ticker.setTimeout 100
8     @ticker.play()
9   'click .btn-play-fast': ->
10     @ticker.setTimeout 0
11     @ticker.play()
12   'click .btn-stop': -> @ticker.stop()
13   'click .btn-reset': -> @ticker.reset()

```

## A Additional source files

### A.1 client/app.jade (template)

```
1
2 head
3   title Advice Complexity of the Online-Knapsack-Problem
4 body
5   .container
6     .page-header
7       h1 Advice Complexity of the Online-Knapsack-Problem
8       img(src="knapsack.svg" style="display:block; margin: 0
9         auto")
10      +experiments
11
12
13 template(name="experiments")
14
15   each experiments
16     +Experiment
17   h2 Online algorithms with advice
18   each experimentsWithAdvice
19     +Experiment
20   +competitivenessChart
21   h2 Randomized online algorithm
22   each randomExperiments
23     +Experiment
24
25
26
27 template(name="Experiment")
28   .panel.panel-default
29     .panel-heading
30       h2.panel-title {{name}}
31     .panel-body
32       {{description}}
33     .row
34       .col-xs-12.col-sm-5
35         table.table
36           tr
37             th Number of experiments:
38             td {{gainHistory.size}}
39           tr
40             th Average gain:
41             td {{gainHistory.avg}}
42           tr
43             th Best gain:
44             td {{gainHistory.best}}
45           tr
46             th Worst gain:
47             td {{gainHistory.worst}}
48           tr
49             th 1-competitive:
50             td {{gainHistory.competitivePercentage "1-
51               competitive"}} %
52           tr
53             th 2-competitive:
54             td {{gainHistory.competitivePercentage "2-
55               competitive"}} %
```

```

54         tr
55             th 4-competitive:
56             td {{gainHistory.competitivePercentage "4-
                    competitive"}} %
57         tr
58             th non-competitive:
59             td {{gainHistory.competitivePercentage "
                    non-competitive"}} %
60     .row
61     .col-xs-12
62     +TickerGui ticker=ticker
63     .row
64     .col-xs-12.col-sm-5
65     table.table
66     tr
67         th acts:
68         td {{act}}
69     tr
70         th adviceBits:
71         td.adviceBits=adviceBits
72     tr
73         th Number of items:
74         td {{numberOfItems}}
75
76     .row
77     .currentItemContainer.col-xs-6
78     h4 Current item:
79
80     if currentItem
81         p value: {{currentItem.value}}
82
83         +KnapsackItem(currentItem)
84         if willMatch
85             .matches.yes
86             .glyphicon.glyphicon-arrow-right
87         else
88             .matches.no
89             .glyphicon.glyphicon-remove
90
91     .knapsackContainer.col-xs-6
92     +Knapsack knapsack
93
94 template(name="Knapsack")
95     .knapsack(style="width: {{totalWidth}}px")
96     h4 knapsack
97     p items: {{items.length}}, gain: {{gain}}
98     .items
99
100
101     each items
102     +KnapsackItem
103
104
105 template(name="KnapsackItem")
106     .knapsack-item(style="width: {{width}}px; background-color: {{
        color}}")
107
108 template(name="competitivenessChart")
109
110
111     .panel.panel-default

```

```
112     .panel-heading
113         h2.panel-title Bits VS competitiveness
114     .panel-body
115         +highchartsHelper chartId="competitivenessChart"
116             chartWidth="100%" chartHeight="400px" chartObject=
117                 chartObject
118         p (*) SLOG =
119             img(src="slog.png" height="40px")
120
121 template(name="TickerGui")
122     .ticker
123         .btn-group
124             button.btn.btn-step.btn-primary
125                 .glyphicon.glyphicon-step-forward
126                 | Step
127             button.btn.btn-play.btn-default
128                 .glyphicon.glyphicon-play
129                 | Play
130             button.btn.btn-play-fast.btn-default
131                 .glyphicon.glyphicon-fast-forward
132                 | Play Fast
133             button.btn.btn-stop.btn-warning
134                 .glyphicon.glyphicon-stop
135                 | Stop
136             button.btn.btn-reset.btn-danger
137                 .glyphicon.glyphicon-fast-backward
138                 | Reset
```

**A.2 client/app.styl (stylesheet)**

```
1
2 body
3     background-image: url("bg.png")
4
5 .knapsack
6     border: 2px solid blue
7     box-sizing: content-box
8     border-radius: 10px
9     overflow: hidden
10    text-align: center
11    .items
12        background-color: #eee
13        height: 75px
14        .knapsack-item
15            float: right;
16            border-right: none
17 .knapsack-item
18     border: 1px dashed black
19     height: 75px
20     box-sizing: border-box
21
22 .currentItemContainer
23     position: relative
24     .matches
25         position: absolute
26         right: 60px
27         bottom: 16px
28         border-radius: 10px
29         font-size: 32px
30         &.yes
31             color: green
32         &.no
33             color: red
34
35 .adviceBits
36     font-family: monospace
37
38 .table
39     td
40         text-align: right
41     th
42         text-align: left
43
44 .panel
45     margin-bottom: 200px
```

## B Code & Live-Version

- Code: [github.com/macrozone/seminar\\_np](https://github.com/macrozone/seminar_np)
- Live-Demo of the experiments: [online-knapsack.macrozone.ch](https://online-knapsack.macrozone.ch)

## References

Böckenhauer, Hans-Joachim, Dennis Komm, Richard Kralovic, and Peter Rossmanith. “The Online Knapsack Problem: Advice and Randomization.” <http://e-collection.library.ethz.ch/eserv/eth:5665/eth-5665-01.pdf>.

Komm, Dennis. “Eine Einführung in Online-Algorithmen.” <http://www.ita.inf.ethz.ch/~dkomm/online-algorithmen.pdf>.