Knapsack problem/Continuous

From Rosetta Code
< Knapsack problem

See also: Knapsack problem and Wikipedia.

A robber burgles a butcher's shop, where he can select from some items. He knows the weights and prices of each items. Because he has a knapsack with 15 kg maximal capacity, he wants to select the items such that he would have his profit maximized. He may cut the items; the item has a reduced price after cutting that is proportional to the original price by the ratio of masses. That means: half of an item has half the price of the original.

This is the item list in the butcher's:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight (kg)</th>
<th>Price (Value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>beef</td>
<td>3.8</td>
<td>36</td>
</tr>
<tr>
<td>pork</td>
<td>5.4</td>
<td>43</td>
</tr>
<tr>
<td>ham</td>
<td>3.6</td>
<td>90</td>
</tr>
<tr>
<td>greaves</td>
<td>2.4</td>
<td>45</td>
</tr>
<tr>
<td>flitch</td>
<td>4.0</td>
<td>30</td>
</tr>
<tr>
<td>brawn</td>
<td>2.5</td>
<td>56</td>
</tr>
<tr>
<td>welt</td>
<td>3.7</td>
<td>67</td>
</tr>
<tr>
<td>salami</td>
<td>3.0</td>
<td>95</td>
</tr>
<tr>
<td>brawn</td>
<td>2.5</td>
<td>56</td>
</tr>
<tr>
<td>sausage</td>
<td>5.9</td>
<td>98</td>
</tr>
<tr>
<td>Knapsack</td>
<td>&lt;=15 kg</td>
<td>?</td>
</tr>
</tbody>
</table>

Which items does the robber carry in his knapsack so that their total weight does not exceed 15 kg, and their total value is maximised?

Contents

- 1 Ada
- 2 BBC BASIC
- 3 C
Ada

with Ada.Text_IO;
with Ada.Strings.Unbounded;

procedure Knapsack_Continuous is
  package US renames Ada.Strings.Unbounded;

  type Item is record
    Name   : US.Unbounded_String;
    Weight : Float;
    Value  : Positive;
    Taken  : Float;
  end record;

  function "<" (Left, Right : Item) return Boolean is
    begin
      return Float (Left.Value) / Left.Weight <
              Float (Right.Value) / Right.Weight;
    end "<";

  type Item_Array is array (Positive range <>) of Item;

function Total_Weight (Items : Item_Array) return Float is
  Sum : Float := 0.0;
begin
  for I in Items'Range loop
    Sum := Sum + Items (I).Weight * Items (I).Taken;
  end loop;
  return Sum;
end Total_Weight;

function Total_Value (Items : Item_Array) return Float is
  Sum : Float := 0.0;
begin
  for I in Items'Range loop
    Sum := Sum + Float (Items (I).Value) * Items (I).Taken;
  end loop;
  return Sum;
end Total_Value;

procedure Solve_Knapsack_Continuous
  (Items : in out Item_Array;
   Weight_Limit : Float)
is
begin
  -- order items by value per weight unit
  Sorting : declare
    An_Item : Item;
    J : Natural;
  begin
    for I in Items'First + 1 .. Items'Last loop
      An_Item := Items (I);
      J := I - 1;
      while J in Items'Range and then Items (J) < An_Item loop
        Items (J + 1) := Items (J);
        J := J - 1;
      end loop;
      Items (J + 1) := An_Item;
    end loop;
  end Sorting;
  declare
    Rest : Float := Weight_Limit;
  begin
    for I in Items'Range loop
      if Items (I).Weight <= Rest then
        Items (I).Taken := Items (I).Weight;
      else
        Items (I).Taken := Rest;
      end if;
      Rest := Rest - Items (I).Taken;
      exit when Rest <= 0.0;
    end loop;
  end Solve_Knapsack_Continuous;

All_Items : Item_Array :=
  ((US.To_Unbounded_String ("beef"), 3.8, 36, 0.0),
   (US.To_Unbounded_String ("pork"), 5.4, 43, 0.0),
   (US.To_Unbounded_String ("ham"), 3.6, 90, 0.0),
   (US.To_Unbounded_String ("greaves"), 2.4, 45, 0.0),
   (US.To_Unbounded_String ("flitch"), 4.0, 30, 0.0),
   (US.To_Unbounded_String ("brawn"), 2.5, 56, 0.0),
   (US.To_Unbounded_String ("welt"), 3.7, 67, 0.0),
   (US.To_Unbounded_String ("salami"), 3.0, 95, 0.0),
   (US.To_Unbounded_String ("sausage"), 5.9, 98, 0.0));
Solve_Knapsack_Continuous (All_Items, 15.0)
Ada.Text_IO.Put_Line
("Total Weight: " & Float'Image (Total_Weight (All_Items)));
Ada.Text_IO.Put_Line
("Total Value: " & Float'Image (Total_Value (All_Items)));
Ada.Text_IO.Put_Line ("Items:");
for I in All_Items'Range loop
  if All_Items (I).Taken > 0.0 then
    Ada.Text_IO.Put_Line
      ("   " & Float'Image (All_Items (I).Taken) & 
       " of " & 
       US.To_String (All_Items (I).Name));
  end if;
end loop;
end Knapsack_Continuous;

BBC BASIC

Works with: BBC BASIC for Windows

INSTALL @lib$+"SORTSALIB"
Sort% = FN_sortSAinit(1, 0) : REM Descending

nItems% = 9
maxWeight = 15.0

DIM items{(nItems%-1) name$, weight, price, worth}
FOR item% = 0 TO nItems%-1
  READ items{(item%)}.name$, items{(item%)}.weight, items{(item%)}.price
  items{(item%)}.worth = items{(item%)}.price / items{(item%)}.weight
NEXT

DATA "beef", 3.8, 36, "pork", 5.4, 43, "ham", 3.6, 90
DATA "greaves", 2.4, 45, "flitch", 4.0, 30, "brawn", 2.5, 56
DATA "welt", 3.7, 67, "salami", 3.0, 95, "sausage", 5.9, 98

C% = nItems% : D% = 0
CALL Sort%, items{()}, items{(0)}.worth

TotalWeight = 0
TotalPrice = 0
FOR i% = 0 TO nItems%-1
  IF TotalWeight + items{(i%)}.weight < maxWeight THEN
    TotalWeight += items{(i%)}.weight
    TotalPrice += items{(i%)}.price
    PRINT "Take all the " items{(i%)}.name$
  ELSE
    weight = maxWeight - TotalWeight
    price = weight * items{(i%)}.worth
    TotalWeight += weight
    TotalPrice += price
    PRINT "Take "; weight " kg of " items{(i%)}.name$
  ENDIF
NEXT

PRINT "Total weight = "; TotalWeight " kg"
PRINT "Total price  = "; TotalPrice
Output:

Take all the salami
Take all the ham
Take all the brawn
Take all the greaves
Take 3.5 kg of welt

Total weight = 15 kg
Total price  = 349.378379

C

#include <stdio.h>
#include <stdlib.h>

struct item { double w, v; const char *name; } items[] = {
    { 3.8, 36, "beef" },
    { 5.4, 43, "pork" },
    { 3.6, 90, "ham" },
    { 2.4, 45, "greaves" },
    { 4.0, 30, "flitch" },
    { 2.5, 56, "brawn" },
    { 3.7, 67, "welt" },
    { 3.0, 95, "salami" },
    { 5.9, 98, "sausage" },
};

int item_cmp(const void *aa, const void *bb)
{
    const struct item *a = aa, *b = bb;
    double ua = a->v / a->w, ub = b->v / b->w;
    return ua < ub ? -1 : ua > ub;
}

int main()
{
    struct item *it;
    double space = 15;

    qsort(items, 9, sizeof(struct item), item_cmp);
    for (it = items + 9; it--; space -= it->w)
        if (space >= it->w)
            printf("take all %s\n", it->name);
        else
            printf("take %g kg of %g kg of %s\n",
                    space, it->w, it->name);

    return 0;
}

output

take all salami
take all ham
take all brawn
take all greaves
take 3.5 kg of 3.7 kg of welt
C++

```cpp
#include<iostream>
#include<algorithm>
#include<string>
using namespace std;

double result;
double capacity = 15;
int NumberOfItems;
int number;

struct items {
    char name[32];
    double weight;
    double price;
    double m;
} item[256];

bool cmp(items a,items b) {
    return a.price/a.weight > b.price/b.weight; // the compare function for the sorting algorithm
}

int main()
{
    NumberOfItems=9;
    strcpy(item[1].name,"beef");
    item[1].weight=3.8;
    item[1].price=36;

    strcpy(item[2].name,"pork");
    item[2].weight=5.4;
    item[2].price=43;

    strcpy(item[3].name,"ham");
    item[3].weight=3.6;
    item[3].price=90;

    strcpy(item[4].name,"greaves");
    item[4].weight=2.4;
    item[4].price=45;

    strcpy(item[5].name,"flitch");
    item[5].weight=4.0;
    item[5].price=30;

    strcpy(item[6].name,"brawn");
    item[6].weight=2.5;
    item[6].price=56;

    strcpy(item[7].name,"welt");
    item[7].weight=3.7;
    item[7].price=67;

    strcpy(item[8].name,"salami");
    item[8].weight=3.0;
    item[8].price=95;
}
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
```c++
strcpy(item[9].name,"sausage");
item[9].weight=5.9;
item[9].price=98;

sort(item+1,item+NumberOfItems+1,cmp); // We'll sort using Introsort from STL

number = 1;
while(capacity>0&&number<=NumberOfItems)
{
  if(item[number].weight<=capacity)
  {
    result+=item[number].price;
    capacity-=item[number].weight;
    item[number].m=1;
  }
  else
  {
    result+=(item[number].price)*(capacity/item[number].weight);
    item[number].m=(capacity/item[number].weight);
    capacity=0;
  }
  ++number;
}

D
```

```c++
import std.stdio, std.algorithm, std.string, std.conv;

struct Item {
  string name;
  real amount, value;

  @property real valuePerKG() const pure nothrow {
    return value / amount;
  }

  string toString() const /*pure nothrow*/ {
    return format("%10s %7.2f %7.2f %7.2f", 
        name, amount, value, valuePerKG);
  }
}

real sum(string field)(in Item[] items) pure nothrow {
  return reduce!("a + b." - field)(0.0L, items);
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
```cpp
void main() {
    Item[] raw = {{"beef", 3.8, 36.0},
                  {{"pork", 5.4, 43.0},
                  {{"ham", 3.6, 90.0},
                  {{"greaves", 2.4, 45.0},
                  {{"flitch", 4.0, 30.0},
                  {{"brawn", 2.5, 56.0},
                  {{"welt", 3.7, 67.0},
                  {{"salami", 3.0, 95.0},
                  {{"sausage", 5.9, 98.0}};

    // Reverse sorted by Value per amount.
    const items = raw.sort!q{a.valuePerKG > b.valuePerKG}.release;

    immutable(Item[]) chosen;
    real space = 15.0;
    foreach (item; items)
        if (item.amount < space) {
            chosen ~= item;
            space -= item.amount;
        } else {
            chosen ~= Item(item.name, space, item.valuePerKG * space);
            break;
        }

    writeln(" ITEM AMOUNT VALUE $/unit");
    writeln("%10s %7s %7s %7s", "ITEM", "AMOUNT", "VALUE", "$/unit");
    writeln("%(%s

")% chosen);
    Item("TOTAL", chosen.sum!"amount", chosen.sum!"value").writeln;
}
```

**Output:**

<table>
<thead>
<tr>
<th>ITEM</th>
<th>AMOUNT</th>
<th>VALUE</th>
<th>$/unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>salami</td>
<td>3.00</td>
<td>95.00</td>
<td>31.67</td>
</tr>
<tr>
<td>ham</td>
<td>3.60</td>
<td>90.00</td>
<td>25.00</td>
</tr>
<tr>
<td>brawn</td>
<td>2.50</td>
<td>56.00</td>
<td>22.40</td>
</tr>
<tr>
<td>greaves</td>
<td>2.40</td>
<td>45.00</td>
<td>18.75</td>
</tr>
<tr>
<td>welt</td>
<td>3.50</td>
<td>63.38</td>
<td>18.11</td>
</tr>
<tr>
<td>TOTAL</td>
<td>15.00</td>
<td>349.38</td>
<td>23.29</td>
</tr>
</tbody>
</table>

**Alternative Version**

```cpp
void main() {
    import std.stdio, std.algorithm;

    static struct T { string item; double weight, price; }

    auto items = [T("beef", 3.8, 36.0),
                  T("pork", 5.4, 43.0),
                  T("ham", 3.6, 90.0),
                  T("greaves", 2.4, 45.0),
                  T("flitch", 4.0, 30.0),
                  T("brawn", 2.5, 56.0),
                  T("welt", 3.7, 67.0),
                  T("salami", 3.0, 95.0),
                  T("sausage", 5.9, 98.0)];
    .schwartzSort!q{ -a.price / a.weight };```
auto left = 15.0;

foreach (it; items)
    if (it.weight <= left) {
        writeln("Take all the ", it.item);
        if (it.weight == left)
            return;
        left -= it.weight;
    } else
        return writelnf("Take %.1fkg %s", left, it.item);
}

Output:
Take all the salami
Take all the ham
Take all the brawn
Take all the greaves
Take 3.5kg welt

Forth
Translation of: D
Works with: 4tH version 3.62.0

include lib/selcsort.4th                \ use a tiny sorting algorithm
150 value left                         \ capacity in 1/10th kilo
create items                           \ list of items
    ", beef" 38 , 3600 ,           \ description, weight, price (cents)
    ", pork" 54 , 4300 ,           \ weight in 1/10 kilo
    ", ham" 36 , 9000 ,            \ 
    ", greaves" 24 , 4500 ,     \ 
    ", flitch" 40 , 3000 ,       \ 
    ", brawn" 25 , 5600 ,        \ 
    ", welt" 37 , 6700 ,         \ 
    ", salami" 30 , 9500 ,       \ 
    ", sausage" 59 , 9800 ,      \ 
here items - 3 / constant #items       \ total number of items
: redo items swap 3 cells * + ;        \ calculate address of record
#items array (items)                   \ array for sorting
    ( a -- n)
: price/weight dup 2 cells + @c swap cell+ @c / ;
: weight@ @ cell+ @c ;                 \ ( a -- n)
: .item @ @c count type cr ;           \ ( a --)
    \ how to sort: on price/weight
: noname >r price/weight r> price/weight > ; is precedes
: knapsack (items) dup #items dup 0 ?do i items (items) i th ! loop sort
begin                                    \ use the sorted array
    dup weight@ left <=                 \ still room in the knapsack?
while
    ". Take all of the " dup .item       \ take all of the item
    left over weight@ - to left cell+ \ adjust knapsack, increment item
repeat
    left 100 * dup                       \ so how much is left?
    \ if room, take as much as possible

Fortran

Works with: Fortran version 90 and later

```fortran
program KNAPSACK_CONTINUOUS
  implicit none

  real, parameter :: maxweight = 15.0
  real :: total_weight = 0, total_value = 0, frac
  integer :: i, j

  type Item
    character(7) :: name
    real :: weight
    real :: value
  end type Item

type(Item) :: items(9), temp

  items(1) = Item("beef", 3.8, 36.0)
  items(2) = Item("pork", 5.4, 43.0)
  items(3) = Item("ham", 3.6, 90.0)
  items(4) = Item("greaves", 2.4, 45.0)
  items(5) = Item("flitch", 4.0, 30.0)
  items(6) = Item("brawn", 2.5, 56.0)
  items(7) = Item("welt", 3.7, 67.0)
  items(8) = Item("salami", 3.0, 95.0)
  items(9) = Item("sausage", 5.9, 98.0)

  ! sort items in desending order of their value per unit weight
  do i = 2, size(items)
    j = i - 1
    temp = items(i)
    do while (j>1 .and. items(j)%value / items(j)%weight < temp%value / temp%weight)
      items(j+1) = items(j)
      j = j - 1
    end do
    items(j+1) = temp
  end do

  i = 0
  write(*, "(a4, a13, a6)") "Item", "Weight", "Value"
  do while(i < size(items) .and. total_weight < maxweight)
    i = i + 1
    if(total_weight+items(i)%weight < maxweight) then
      total_weight = total_weight + items(i)%weight
      total_value = total_value + items(i)%value
      write(*, "(a7, 2f8.2)") items(i)
    else
      frac = (maxweight-total_weight) / items(i)%weight
      total_weight = total_weight + items(i)%weight * frac
      total_value = total_value + items(i)%value * frac
      write(*, "(a7, 2f8.2)") items(i)%name, items(i)%weight * frac, items(i)%value * frac
    end if
  end do

  write(*, "(f15.2, f8.2)") total_weight, total_value
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
Go

package main

import {
    "fmt"
    "sort"
}

type item struct {
    item string
    weight float64
    price float64
}

type items []item

var all = []item{
    {"beef", 3.8, 36},
    {"pork", 5.4, 43},
    {"ham", 3.6, 90},
    {"greaves", 2.4, 45},
    {"flitch", 4.0, 30},
    {"brawn", 2.5, 56},
    {"welt", 3.7, 67},
    {"salami", 3.0, 95},
    {"sausage", 5.9, 98},
}

// satisfy sort interface
func (z items) Len() int    { return len(z) }
func (z items) Swap(i, j int) { z[i], z[j] = z[j], z[i] }
func (z items) Less(i, j int) bool {
    return z[i].price/z[i].weight > z[j].price/z[j].weight
}

func main() {
    left := 15.
    sort.Sort(all)
    for _, i := range all {
        if i.weight <= left {
            fmt.Println("take all the", i.item)
            if i.weight == left {
                return
            }
            left -= i.weight
        } else {
            fmt.Printf("take %.1fkg %s
", left, i.item)
            return
        }
    }
}

Output:

take all the salami
take all the ham
take all the brawn
take all the greaves
take 3.5kg welt

**Groovy**

Solution: obvious greedy algorithm

```groovy
import static java.math.RoundingMode.*

def knapsackCont = { list, maxWeight = 15.0 ->
    list.sort{ it.weight / it.value }
    def remainder = maxWeight
    def sack = []
    for (item in list) {
        if (item.weight < remainder) {
            sack << [name: item.name, weight: item.weight,
                     value: (item.value as BigDecimal).setScale(2, HALF_UP)]
        } else {
            sack << [name: item.name, weight: remainder,
                     value: (item.value * remainder / item.weight).setScale(2, HALF_UP)]
            break
        }
        remainder -= item.weight
    }
    sack
}

test:

def possibleItems = [
    [name: 'beef', weight: 3.8, value: 36],
    [name: 'pork', weight: 5.4, value: 43],
    [name: 'ham', weight: 3.6, value: 90],
    [name: 'greaves', weight: 2.4, value: 45],
    [name: 'flitch', weight: 4.0, value: 30],
    [name: 'brawn', weight: 2.5, value: 56],
    [name: 'welt', weight: 3.7, value: 67],
    [name: 'salami', weight: 3.0, value: 95],
    [name: 'sausage', weight: 5.9, value: 98],
]

def contents = knapsackCont(possibleItems)
println "Total Value: ${contents*.value.sum()"
contents.each { 
    printf("    name: %-7s  weight: $it.weight  value: $it.value\n", it.name)
}

Output:

Total Value: 349.38
    name: salami  weight: 3.0  value: 95.00
    name: ham    weight: 3.6  value: 90.00
    name: brawn  weight: 2.5  value: 56.00
    name: greaves weight: 2.4  value: 45.00

http://rosettacode.org/wiki/Knapsack_problem/Continuous
Haskell

We use a greedy algorithm.

```haskell
import Control.Monad
import Data.List (sortBy)
import Data.Ord (comparing)
import Data.Ratio (numerator, denominator)
import Text.Printf

maxWgt = 15

data Bounty = Bounty
  { itemName :: String,
    itemVal, itemWgt :: Rational }

items = [Bounty "beef" 36 3.8,
         Bounty "pork" 43 5.4,
         Bounty "ham" 90 3.6,
         Bounty "greaves" 45 2.4,
         Bounty "flitch" 30 4.0,
         Bounty "brawn" 56 2.5,
         Bounty "welt" 67 3.7,
         Bounty "salami" 95 3.0,
         Bounty "sausage" 98 5.9]

solution :: [(Rational, Bounty)]
Solution = g maxWgt $ sortBy (flip $ comparing f) items
  where g room (b@(Bounty _ _ w) : bs) = if w < room
            then (w, b) : g (room - w) bs
            else [(room, b)]
            f (Bounty _ v w) = v / w

main = do
  forM_ solution $ \(w, b) ->
    printf "%s kg of %s\n" (mixedNum w) (itemName b)
    printf "Total value: %s\n" $ mixedNum $ sum $ map f solution
    where f v wtot = v * (w / wtot)
    mixedNum q = if b == 0
      then show a
      else printf "%d %d/%d" a (numerator b) (denominator b)
      where a = floor q
            b = q - toEnum a

Or similar to above (but more succinct):

```
Icon and Unicon

This implements the greedy algorithm. This also uses a Unicon extension to reverse which reverses a list. In Icon, an IPL procedure is available to do the same.

```
link printf

procedure main()
    room := 15
    every (x := ![choices := get_items()]).uprice := x.price / x.weight
    choices := reverse(sortf(choices, $))

    every (value := 0, x := ![choices]) do {
        if x.weight <= room then {
            printf("Take all of the %s (%r kg) worth $%r\n", x.name, x.weight, x.price)
            value +:= x.price
            room -:= x.weight
        }
        else {
            fvalue := x.uprice * room
            printf("Take (%r kg) of the %s worth $%r\n", room, x.name, fvalue)
            value +:= fvalue
            break
        }
    }
    printf("Total value of a full knapsack is $%r\n", value)
end

record item(name, weight, price, uprice)

procedure get_items()
    return [ item("beef", 3.8, 36),
             item("pork", 5.4, 43),
             item("ham", 3.6, 90),
             item("greaves", 2.4, 45),
             item("flitch", 4.0, 30),
             item("brawn", 2.5, 56),
             item("welt", 3.7, 67),
             item("salami", 3.0, 95),
             item("sausage", 5.9, 98) ]
end
```

Library: Icon Programming Library
printf.icn provides printf (http://www.cs.arizona.edu/icon/library/src/procs/printf.icn)

Output:

Take all of the salami (3.000000 kg) worth $95.000000
Take all of the ham (3.600000 kg) worth $90.000000
Take all of the brawn (2.500000 kg) worth $56.000000
Take all of the greaves (2.400000 kg) worth $45.000000
Take (3.500000 kg) of the welt worth $63.378378
Total value of a full knapsack is $349.378378

J

We take as much as we can of the most valuable items first, and continue until we run out of space. Only one item needs to be cut.

```j
'names numbers'=: |: ; _2] 0 : 0
beef  3.8 36
pork  5.4 43
ham  3.6 90
greaves 2.4 45
'flitch' 4.0 30
brawn 2.5 56
welt  3.7 67
'salami' 3.0 95
'sausage' 5.9 98
')
'weights prices'=: |: 'names
order=: \:prices%weights
take=: 15&<.&.+/\ order(\weights
result=: (*take)#(order(\names)' '),'.',:)\take
This gives the result:

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>salami</td>
<td>3</td>
</tr>
<tr>
<td>ham</td>
<td>3.6</td>
</tr>
<tr>
<td>brawn</td>
<td>2.5</td>
</tr>
<tr>
<td>greaves</td>
<td>2.4</td>
</tr>
<tr>
<td>welt</td>
<td>3.5</td>
</tr>
</tbody>
</table>

For a total value of:

```
+/prices * (take/\order) % weights
349.378
```

Java

http://rosettacode.org/wiki/Knapsack_problem/Continuous
Greedy solution.

```java
package hu.pj.alg.test;
import hu.pj.alg.ContinuousKnapsack;
import hu.pj.obj.Item;
import java.util.*;
import java.text.*;

public class ContinousKnapsackForRobber {
    final private double tolerance = 0.0005;

    public ContinousKnapsackForRobber()
    {
        ContinuousKnapsack cok = new ContinuousKnapsack(15); // 15 kg
        // making the list of items that you want to bring
        cok.add("beef", 3.8, 36); // marhahús
        cok.add("pork", 5.4, 43); // disznóhús
        cok.add("ham", 3.6, 90); // sonka
        cok.add("greaves", 2.4, 45); // tepertő
        cok.add("flitch", 4.0, 30); // oldalas
        cok.add("brawn", 2.5, 56); // disznősajt
        cok.add("welt", 3.7, 67); // hurka
        cok.add("salami", 3.0, 95); // szalámi
        cok.add("sausage", 5.9, 98); // kolbász

        // calculate the solution:
        List<Item> itemList = cok.calcSolution();
        // write out the solution in the standard output
        if (cok.isCalculated()) {
            NumberFormat nf = NumberFormat.getInstance();
            System.out.println("Maximal weight           = " + nf.format(cok.getMaxWeight()) + " kg");
            System.out.println("Total weight of solution = " + nf.format(cok.getSolutionWeight()) + " kg");
            System.out.println("Total value (profit)     = " + nf.format(cok.getProfit()));
            System.out.println("You can carry the following materials " + " in the knapsack:"
            for (Item item : itemList) {
                if (item.getInKnapsack() > tolerance) {
                    System.out.format("%1$-10s %2$-15s %3$-15s
                        " + " kg ",
                    nf.format(item.getInKnapsack()) + " kg ",
                    item.getName(),
                    "(value = " + nf.format(item.getInKnapsack() * (item.getValue() / item.getWeight())) + ")" );
                }
            }
        }
    }
}
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
"The problem is not solved. " +
"Maybe you gave wrong data."
;
}

global public static void main(String[] args) {
    new ContinuousKnapsackForRobber();
}
} // class

package hu.pj.alg;
import hu.pj.obj.Item;
import java.util.*;

class ContinuousKnapsack {
    protected List<Item> itemList = new ArrayList<Item>();
    protected double maxWeight = 0;
    protected double solutionWeight = 0;
    protected double profit = 0;
    protected boolean calculated = false;

class ContinuousKnapsack{
    public ContinuousKnapsack() {}
    public ContinuousKnapsack(double _maxWeight) {
        setMaxWeight(_maxWeight);
    }
}

public List<Item> calcSolution() {
    int n = itemList.size();

    setInitialStateForCalculation();
    if (n > 0 && maxWeight > 0) {
        Collections.sort(itemList);
        for (int i = 0; (maxWeight - solutionWeight) > 0.0 && i < n; i++) {
            Item item = itemList.get(i);
            if (item.getWeight() >= (maxWeight - solutionWeight)) {
                item.setInKnapsack(maxWeight - solutionWeight);
                solutionWeight = maxWeight;
                profit += item.getInKnapsack() / item.getWeight() * item.getValue();
                break;
            } else {
                item.setInKnapsack(item.getWeight());
                solutionWeight += item.getInKnapsack();
                profit += item.getValue();
            }
        }
        calculated = true;
    }

    return itemList;
}

// add an item to the item list
public void add(String name, double weight, double value) {
    if (name.equals("")
        name = "" + (itemList.size() + 1);
    itemList.add(new Item(name, weight, value));
    setInitialStateForCalculation();
}
```java
public double getMaxWeight() {
    return maxWeight;
}
public double getProfit() {
    return profit;
}
public double getSolutionWeight() {
    return solutionWeight;
}
public boolean isCalculated() {
    return calculated;
}

public void setMaxWeight(double _maxWeight) {
    maxWeight = Math.max(_maxWeight, 0);
}

// set the member with name "inKnapsack" by all items:
private void setInKnapsackByAll(double inKnapsack) {
    for (Item item : itemList)
        item.setInKnapsack(inKnapsack);
}

// set the data members of class in the state of starting the calculation:
protected void setInitialStateForCalculation() {
    setInKnapsackByAll(-0.0001);
    calculated = false;
    profit = 0.0;
    solutionWeight = 0.0;
}

} // class

package hu.pj.obj;

public class Item implements Comparable {

    protected String name = "";
    protected double weight = 0;
    protected double value = 0;
    protected double inKnapsack = 0; // the weight of item in solution

    public Item() {}

    public Item(Item item) {
        setName(item.getName);
        setWeight(item.weight);
        setValue(item.value);
    }

    public Item(double _weight, double _value) {
        setWeight(_weight);
        setValue(_value);
    }

    public Item(String _name, double _weight, double _value) {
        setName(_name);
        setWeight(_weight);
        setValue(_value);
    }

    public void setName(String _name) {name = _name;}
    public void setWeight(double _weight) {weight = Math.max(_weight, 0);}
    public void setValue(double _value) {value = Math.max(_value, 0);}

    public void setInKnapsack(double _inKnapsack) {
        inKnapsack = Math.max(_inKnapsack, 0);
    }
```
```java
public void checkMembers() {
    setWeight(weight);
    setValue(value);
    setInKnapsack(inKnapsack);
}

public String getName() {
    return name;
}

public double getWeight() {
    return weight;
}

public double getValue() {
    return value;
}

public double getInKnapsack() {
    return inKnapsack;
}

// implementing of Comparable interface:
public int compareTo(Object item) {
    int result = 0;
    Item i2 = (Item)item;
    double rate1 = value / weight;
    double rate2 = i2.value / i2.weight;
    if (rate1 > rate2) result = -1;  // if greater, put it previously
    else if (rate1 < rate2) result = 1;
    return result;
}

} // class
```

output:

Maximal weight = 15 kg
Total weight of solution = 15 kg
Total value (profit) = 349,378

You can carry the following materials in the knapsack:

3 kg salami (value = 95)
3.6 kg ham (value = 90)
2.5 kg brawn (value = 56)
2.4 kg greaves (value = 45)
3.5 kg welt (value = 63,378)

**Mathematica**

The problem is solved by sorting the original table by price to weight ratio, finding the accumulated weight, and the index of the item which exceeds the carrying capacity (overN) The output is then all items prior to this one, along with that item corrected for it's excess weighter (overW)

```mathematica
Knapsack[shop_, capacity_] := Block[{sortedTable, overN, overW, output},
  sortedTable = SortBy[{#1, #2, #3, #3/#2} & @@@ shop, -#4 &];
  overN = Position[Accumulate[sortedTable[[1 ;;, 2]]], a_ /; a > capacity, 1, 1][[1, 1]];
  overW = Accumulate[sortedTable[[1 ;;, 2]]][[overN]] - capacity;
  output = Reverse[sortedTable[[Ordering[sortedTable[[1 ;;, 4]], -overN]]]]; 
  output[[-1, 2]] = output[[-1, 2]] - overW;
  output[[-1, 3]] = output[[-1, 2]] output[[-1, 4]];
  Append[output[[1 ;;, 1 ;; 3]], {"Total", Sequence @@ Total[output[[1 ;;, 2 ;; 3]]]}]]
```

A test using the specified data:

http://rosettacode.org/wiki/Knapsack_problem/Continuous
weightPriceTable = 
{"beef", 3.8, 36}, 
{"pork", 5.4, 43}, 
{"ham", 3.6, 90}, 
{"greaves", 2.4, 45}, 
{"flitch", 4., 30},
{"brawn", 2.5, 56}, 
{"welt", 3.7, 67}, 
{"salami", 3., 95}, 
{"sausage", 5.9, 98};
carryCapacity = 15;
Knapsack[weightPriceTable, carryCapacity] // Grid

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>salami</td>
<td>3.</td>
<td>95</td>
</tr>
<tr>
<td>ham</td>
<td>3.6</td>
<td>90</td>
</tr>
<tr>
<td>brawn</td>
<td>2.5</td>
<td>56</td>
</tr>
<tr>
<td>greaves</td>
<td>2.4</td>
<td>45</td>
</tr>
<tr>
<td>welt</td>
<td>3.5</td>
<td>63.3784</td>
</tr>
<tr>
<td>Total</td>
<td>15.</td>
<td>349.378</td>
</tr>
</tbody>
</table>

Mathprog

/*Knapsack
 This model finds the optimal packing of a knapsack
 Nigel_Galloway
 January 10th., 2012
*/
set Items;
param weight{t in Items};
param value{t in Items};
var take{t in Items}, >=0, <=weight[t];
knap_weight : sum{t in Items} take[t] <= 15;
maximize knap_value: sum{t in Items} take[t] * (value[t]/weight[t]);
data;
param : Items          : weight   value :=
beef            3.8     36
pork            5.4   43
ham     3.6   90
greaves           2.4     45
flitch            4.0     30
brawn            2.5     56
welt            3.7     67
salami            3.0     95
sausage           5.9     98
;
end;
The solution is here at Knapsack problem/Continuous/Mathprog.

OCaml

let items =
[ "beef", 3.8, 36; 
 "pork", 5.4, 43; 
 "ham", 3.6, 90; 
]
"greaves",  2.4,  45;
"flitch",  4.0,  30;
"brawn",  2.5,  56;
"welt",   3.7,  67;
"salami",  3.0,  95;
"sausage",  5.9,  98; ]

let () =
  let items = List.map (fun (name, w, p) -> (name, w, p, float p /. w)) items in
  let items = List.sort (fun (_r,_,_,v1) (_r,_,_,v2) -> compare v2 v1) items in

let rec loop acc weight =
  function
  |
  (name, w,_,_) as item ::
    tl ->
    if w +. weight > 15.0
    then (weight, acc, item)
    else loop (item::acc) (w +. weight) tl
  |
  [] -> assert false in

let weight, res,
  (last,w,p,v) =
  loop [] 0.0 items in

print_endline "    Items  Weight Price"

let price =
  List.fold_left (fun price (name,w,p,_) ->
    Printf.printf " %7s: %6.2f %3d
" name w p;
  (p + price)) 0 res in

let rem_weight = 15.0 -. weight in
let last_price = v *. rem_weight in

Printf.printf " %7s: %6.2f %6.2f
" last rem_weight last_price in

Printf.printf " Total Price: %.3f
" (float price +. last_price);

perl

my @items = sort { $b->[2]/$b->[1] <=> $a->[2]/$a->[1] } qw

[ 'beef    3.8 36'
, 'pork    5.4 43'
, 'ham     3.6 90'
, 'greaves 2.4 45'
, 'flitch  4.0 30'
, 'brawn   2.5 56'
, 'welt    3.7 67'
, 'salami  3.0 95'
, 'sausage 5.9 98'
]

my ($limit, $value) = (15, 0);

print "item   fraction weight value
"
for (@items) {
  print "$_->[0]t"
  $value += $_->[2] * $ratio;
  $limit -= $_->[1];
  if ($ratio == 1) {
    print " all\t$_->[1]\t$_->[2]\n"
  } else {
    printf "%5.3f  %s  %8.3f\n", $ratio, $_->[1] * $ratio, $_->[2] * $ratio;
    last;
  }
}
print "-" x 40, "\ntotal value: $value\n";

Output:

<table>
<thead>
<tr>
<th>item</th>
<th>fraction</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>salami</td>
<td>all</td>
<td>3.0</td>
<td>95</td>
</tr>
<tr>
<td>ham</td>
<td>all</td>
<td>3.6</td>
<td>90</td>
</tr>
<tr>
<td>brawn</td>
<td>all</td>
<td>2.5</td>
<td>56</td>
</tr>
<tr>
<td>greaves</td>
<td>all</td>
<td>2.4</td>
<td>45</td>
</tr>
<tr>
<td>welt</td>
<td>0.946</td>
<td>3.5</td>
<td>63.378</td>
</tr>
</tbody>
</table>
----------------------------------------
total value: 349.378378378378

Perl 6

This Solutions sorts the item by WEIGHT/VALUE

class KnapsackItem {
  has $.name;
  has $.weight is rw;
  has $.price is rw;
  has $.ppw;

  method new (Str $n, $w, $p) {
    KnapsackItem.bless(*, :name($n), :weight($w), :price($p), :ppw($w/$p))
  }

  method cut-maybe ($max-weight) {
    return False if $max-weight > $.weight;
    $.price = $max-weight / $.ppw;
    $.weight = $.ppw * $.price;
    return True;
  }

  method Str () { printf "%8s %1.2f %3.2f",
      $.name,
      $.weight,
      $.price }
}

my $max-w = 15;
say "Item   Portion  Value";
.say for gather
  for <$beef 3.8 36
       <pork 5.4 43
       <ham 3.6 90
       <greaves 2.4 45
       <flitch 4.0 30
       <brawn 2.5 56
       <welt 3.7 67
       <salami 3.0 95
       <sausage 5.9 98 >
    => map { KnapsackItem.new($`a, $`b, $`c) }
    => sort *.ppw
    {
      my $last-one = .cut-maybe($max-w);
      take $_;
```perl
$max -w -= .weight;
  last if $last-one;
}

Output:

$perl6 knapsack_continous.p6
Item    Portion Value
salami 3.00   95.00
ham 3.60   90.00
brawn 2.50   56.00
greaves 2.40   45.00
welt 3.50   63.38

PicoLisp

(scl 2)
(de *Items
  ("beef" 3.8 36.0)
  ("pork" 5.4 43.0)
  ("ham" 3.6 90.0)
  ("greaves" 2.4 45.0)
  ("flitch" 4.0 30.0)
  ("brawn" 2.5 56.0)
  ("welt" 3.7 67.0)
  ("salami" 3.0 95.0)
  ("sausage" 5.9 98.0) )

(let K
  (make
    (let Weight 0
      (for I (by '((L) (*/ (caddr L) -1.0 (cadr L))) sort *Items)
        (T (= Weight 15.0))
        (inc 'Weight (cadr I))
        (T (> Weight 15.0)
          (let W (- (cadr I) Weight -15.0)
            (link (list (car I) W (*/ W (caddr I) (cadr I)))) )
          (link I) ) ) )
    (for I K
      (tab (3 -9 8 8)
        NIL
        (car I)
        (format (cadr I) *Scl)
        (format (caddr I) *Scl) ) )
    (tab (12 8 8)
      NIL
      (format (sum cadr K) *Scl)
      (format (sum caddr K) *Scl) ) )

Output:

salami       3.00   95.00
ham          3.60   90.00
brawn        2.50   56.00
greaves      2.40   45.00
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
Prolog

Works with SWI-Prolog and library(simplex) written by Markus Triska

```prolog
:- use_module(library(simplex)).  % tuples (name, weights, value).
knapsack :-
    L = [(beef, 3.8, 36),
         (pork, 5.4, 43),
         (ham, 3.6, 90),
         (greaves, 2.4, 45),
         (flitch, 4.0, 30),
         (brawn, 2.5, 56),
         (welt, 3.7, 67),
         (salami, 3.0, 95),
         (sausage, 5.9, 98)],
    gen_state(S0),
    length(L, N),
    numlist(1, N, LN),
    (  ( create_constraint_N(LN, L, S0, S1, [], LW, [], LV),
          constraint(LN =< 15.0, S1, S2),
          maximize(LV, S2, S3)
    ),
    compute_lenword(L, 0, Len),
    sformat(A1, '---w---t---w', [Len]),
    sformat(A2, '---t---2f---w', [10]),
    sformat(A3, '---t---2f---w', [10]),
    print_results(S3, A1,A2,A3, L, LN, 0, 0).

create_constraint_N([], [], S, S, LW, LW, LV).

create_constraint_N([HN|TN], [_, W, V | TL], S1, SF, LW, LWF, LV, LVF) :-
    constraint([x(HN)] >= 0, S1, S2),
    constraint([x(HN)] =< W, S2, S3),
    X is V/W,
    create_constraint_N(TN, TL, S3, SF, [x(HN) | LW], LWF, [X * x(HN) | LV], LVF).

%-----------------------------------------------------------------
% compute_lenword([], N, N).
compute_lenword([Name, _, _]|T], N, NF):-
    atom_length(Name, L),
    ( L > N -> N1 = L; N1 = N),
    compute_lenword(T, N1, NF).
%-----------------------------------------------------------------
% print_results(_, A1, A2, A3, [], [], WM, VM) :-
%    sformat(W1, A1, [' ']),
%    sformat(W2, A2, [WM]),
%    sformat(W3, A3, [VM]),
%    format('---w---w---n', [W1,W2,W3]).

print_results(S, A1, A2, A3, [(Name, W, V)|T], [N|TN], W1, V1) :-
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
variable_value(S, x(N), X),
( X = 0 -> W1 = W2, V1 = V2
);
  sformat(S1, A1, [Name]),
  sformat(S2, A2, [X]),
  Vtemp is X * V/W,
  sformat(S3, A3, [Vtemp]),
  format('-w-w-w-n', [S1,S2,S3]),
  W2 is W1 + X,
  V2 is V1 + Vtemp ),
print_results(S, A1, A2, A3, T, TN, W2, V2).

Output:

?- knapsack.
ham          3.60     90.00
greaves      2.40     45.00
brawn        2.50     56.00
welt         3.50     63.38
salami       3.00     95.00
 15.00    349.38
true .

PureBasic

Using the greedy algorithm.

$structure item
  name.s
  weight.f ;units are kilograms (kg)
  Value.f
  vDensity.f ;the density of the value, i.e. value/weight, and yes I made up the term ;)
$endstructure

#maxWeight = 15
$global itemCount = 0 ;this will be increased as needed to match actual count
$global Dim items.item(itemCount)

$procedure addItem(name.s, weight.f, Value.f)
  $if itemCount >= ArraySize(items())
    $redim items.item(itemCount + 10)
  $endif
  $with items(itemCount)
    \name = name
    \weight = weight
    \Value = Value
    $if $not \weight
      \vDensity = \Value
    $else
      \vDensity = \Value / \weight
    $endif
  $endwith
  itemCount + 1
$endprocedure

;build item list
addItem("beef", 3.8, 36)
addItem("pork", 5.4, 43)
addItem("ham", 3.6, 90)
addItem("greaves", 2.4, 45)
addItem("flitch", 4.0, 30)
addItem("brawn", 2.5, 56)
addItem("welt", 3.7, 67)
addItem("salami", 3.0, 95)
addItem("sausage", 5.9, 98)

SortStructuredArray(items(), #PB_Sort_descending, OffsetOf(item\vDensity), #PB_Sort_Float, 0, itemCount - 1)

Define TotalWeight.f, TotalValue.f, i
NewList knapsack.item()

For i = 0 To itemCount
  If TotalWeight + items(i)\weight < #maxWeight
    AddElement(knapsack())
    knapsack() = items(i)
    TotalWeight + items(i)\weight
    TotalValue + items(i)\Value
  Else
    AddElement(knapsack())
    knapsack() = items(i)
    knapsack()\weight = #maxWeight - TotalWeight
    knapsack()\Value = knapsack()\weight * knapsack()\vDensity
    TotalWeight = #maxWeight
    TotalValue + knapsack()\Value
    Break
  EndIf
Next

If OpenConsole()
  PrintN(LSet("Maximal weight", 26, " ") + "= " + Str(#maxWeight) + " kg")
  PrintN(LSet("Total weight of solution", 26, " ") + "= " + Str(#maxWeight) + " kg")
  PrintN(LSet("Total value", 26, " ") + "= " + StrF(TotalValue, 3) + " " + #CRLF$)
  PrintN("You can carry the following materials in the knapsack: ")
  ForEach knapsack()
    PrintN(RSet(StrF(knapsack()\weight, 1), 5, " ") + " kg " + LSet(knapsack()\name, 10, " ") + " (Value = ")
  Next
  Print(#CRLF$ + #CRLF$ + "Press ENTER to exit")
  Input()
  CloseConsole()
EndIf

Sample output:

Maximal weight = 15 kg
Total weight of solution = 15 kg
Total value = 349.378

You can carry the following materials in the knapsack:
3.0 kg salami (Value = 95.000)
3.6 kg ham (Value = 90.000)
2.5 kg brawn (Value = 56.000)
2.4 kg greaves (Value = 45.000)
3.5 kg welt (Value = 63.378)
I think this greedy algorithm of taking the largest amounts of items ordered by their value per unit weight is maximal:

```python
#        NAME, WEIGHT, VALUE (for this weight)
items = ["beef", 3.8, 36.0],
        "pork", 5.4, 43.0],
        "ham", 3.6, 90.0],
        "greaves", 2.4, 45.0],
        "flitch", 4.0, 30.0],
        "brawn", 2.5, 56.0],
        "welt", 3.7, 67.0],
        "salami", 3.0, 95.0],
        "sausage", 5.9, 98.0]]

MAXWT = 15.0

sorted_items = sorted(((value/amount, amount, name) for name, amount, value in items), reverse = True)

wt = val = 0

for unit_value, amount, name in sorted_items:
    portion = min(MAXWT - wt, amount)
    wt += portion
    addval = portion * unit_value
    val += addval
    bagged += [(name, portion, addval)]
if wt >= MAXWT:
    break

print("    ITEM   PORTION VALUE")
print("\n".join("%10s %6.2f %6.2f" % item for item in bagged))
print("\nTOTAL WEIGHT: %5.2f\nTOTAL VALUE: %5.2f" % (wt, val))
```

**Sample Output**

```
ITEM   PORTION VALUE
salami 3.00 95.00
ham 3.60 90.00
brawn 2.50 56.00
greaves 2.40 45.00
welt 3.50 63.38

TOTAL WEIGHT: 15.00
TOTAL VALUE: 349.38
```

**Racket**

```
#lang racket
(define shop-inventory
  '((beef 3.8 36)
    (pork 5.4 43)
    (ham 3.6 90)
    (greaves 2.4 45)
    (flitch 4.0 30)
    (brawn 2.5 56)
    (welt 3.7 67))
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
(salami 3.0 95)
(sausage 5.9 98))

(define (continuous-knapsack shop sack sack-capacity sack-total-value)
  ;; solved by loading up on the highest value item...
  (define (value/kg item) (/ (third item) (second item)))
  (if (zero? sack-capacity)
      (values (reverse sack) sack-total-value)
      (let* ((best-value-item (argmax value/kg shop))
              (bvi-full-weight (second best-value-item))
              (amount-can-take (min sack-capacity bvi-full-weight))
              (bvi-full-value (third best-value-item))
              (bvi-taken-value (* bvi-full-value (/ amount-can-take bvi-full-weight))))
        (continuous-knapsack (remove best-value-item shop)
                              (cons (list (first best-value-item)
                                         (if (= amount-can-take bvi-full-weight)
                                             'all-of amount-can-take)
                                         bvi-taken-value)
                                   sack)
                                   (- sack-capacity amount-can-take)
                                   (+ sack-total-value bvi-taken-value)))))

(define (report-knapsack sack total-value)
  (for-each (lambda (item)
              (if (eq? 'all-of (second item))
                  (printf "Take all of the ~a (for ~a)~%n" (first item) (third item))
                (printf "Take ~a of the ~a (for ~a)~%n" (real->decimal-string (second item))
                        (first item) (real->decimal-string (third item))))
              sack)
  (printf "For a grand total of: ~a" (real->decimal-string total-value)))

(call-with-values (lambda () (continuous-knapsack shop-inventory null 15 0))
                     report-knapsack)

Output:

Take all of the salami (for 95.0)
Take all of the ham (for 90.0)
Take all of the brawn (for 56.0)
Take all of the greaves (for 45.0)
Take 3.50 of the welt (for 63.38)
For a grand total of: 349.38

REXX

Any resemblance to the Fortran code is 120% coincidental.
Knapsack problem/Continuous - Rosetta Code

```plaintext
nL=length('total weight');  WL=length('weight');  vL=length(' value ')
totW=0; totV=0
    do j=1 while @$j\lt=-' ;  parse var @$j n w v .
        nL=max(nL,length(n)) ;  n.j=n
totW=totW+w ;  w.j=w
totV=totV+v ;  v.j=v
    end  /*j*/
items=j-1
    /*items is the number of items. */
    nL=nL+nL%4
    /*nL: max length name + 25%. */
    WL=max(WL,length(format(totw,,2)))
    /*WL: max formatted weight width*/
    vL=max(vL,length(format(totv,,2)))
    /*vL: max formatted value width*/
totW=0; totV=0
call show 'before sorting'
    do j=2 to items
        /*sort by desending value/unit wt*/
        k=j-1;
        _n=n.j;  _w=w.j;  _v=v.j
        do k=k by -1 to 1 while _v/k/_w < _v/w
            kpl=k+1;  n.kpl=n.k;  w.kpl=w.k;  v.kpl=v.k
            /*k*/
            kpl=k+1;  n.kpl=_n;  w.kpl=_w;  v.kpl=_v
            end  /*j*/
    call show 'after sorting'
call hdr "burgler's knapsack contents"
    maxW=15
        /*burgler's knapsack max weight. */
        do j=1 for items while totW < maxW
            if totW+w.j<maxW then do
                totW=totW + w.j
                totV=totV + v.j
                call syf n.j, w.j, v.j
            end
            else do
                f=(maxW-totW) / w.j
                totW=totW + w.j*f
                totV=totV + v.j*f
                call syf n.j, w.j*f, v.j*f
            end
        end  /*j*/
    call sep
call sy left('total weight',nL,')', format(totW,,2)
call sy left('total value',nL,')', format(totV,,2)
exit  /*stick a fork in it, we're done.*/

one-liner subroutines
hd:
    indent=left('',9);  call verse arg(1);  call title;  call sep;  return
sep:
    call sy copies(='',nL), copies('',WL), copies('',vL);  return
show:
    call hdr arg(1);  do j=1 for items;  call syf n.j,w.j,v.j;end;  say;  return
sy:
    say indent left(arg(1),nL) right(arg(2),WL) right(arg(3),vL);  return
syf:
    call sy arg(1), format(arg(2),',2), format(arg(3),',2);  return
title:
    call sy center('item',nL),center('weight',WL),center('value',vL);  return
verse:
    say center(arg(1),50,');  say;
```

Output:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'sausage'</td>
<td>5.9</td>
<td>98</td>
</tr>
<tr>
<td>'salami'</td>
<td>3</td>
<td>95</td>
</tr>
<tr>
<td>'ham'</td>
<td>3.6</td>
<td>90</td>
</tr>
<tr>
<td>'welt'</td>
<td>3.7</td>
<td>67</td>
</tr>
</tbody>
</table>

The program calculates the total weight and value, sorts the items by descending value/unit weight, and then iterates through the items to find the maximum weight capacity.
Run BASIC

```basic
dim name$(9)
dim wgt(9)
dim price(9)
dim tak$(100)

name$(1) = "beef" : wgt(1) = 3.8 : price(1) = 36
name$(2) = "pork" : wgt(2) = 5.4 : price(2) = 43
name$(3) = "ham" : wgt(3) = 3.6 : price(3) = 90
name$(4) = "greaves" : wgt(4) = 2.4 : price(4) = 45
name$(5) = "flitch" : wgt(5) = 4.0 : price(5) = 30
name$(6) = "brawn" : wgt(6) = 2.5 : price(6) = 56
name$(7) = "welt" : wgt(7) = 3.7 : price(7) = 67
name$(8) = "salami" : wgt(8) = 3.0 : price(8) = 95
name$(9) = "sausage" : wgt(9) = 5.9 : price(9) = 98

for beef = 0 to 15 step 3.8
    for pork = 0 to 15 step 5.4
        for ham = 0 to 15 step 3.6
            for greaves = 0 to 15 step 2.4
                for flitch = 0 to 15 step 4.0
                    for brawn = 0 to 15 step 2.5
                        for welt = 0 to 15 step 3.7
                            for salami = 0 to 15 step 3.0
                                for sausage = 0 to 15 step 5.9
                                    if beef + pork + ham + greaves + flitch + brawn + welt + salami + sausage <= 15 then
                                        totPrice = beef / 3.8 * 36 + _
pork / 5.4 * 43 + _
ham / 3.6 * 90 + _
greaves / 2.4 * 45 + _
flitch / 4.0 * 30 + _
brawn / 2.5 * 56 + _
welt / 3.7 * 67 + _
salami / 3.0 * 95 + _
sausage / 5.9 * 98
                                    if totPrice >= maxPrice then
                                        maxPrice = totPrice
                                        theMax = max(totPrice,maxPrice)
                                        t = t + 1
                                        tak$(t) = str$(maxPrice);",";beef",";pork",";ham",";greaves",";flitch",";brawn",";welt"
```

http://rosettacode.org/wiki/Knapsack_problem/Continuous
end if
end if

print "Best 2 Options":print
for i = t-1 to t
    totTake = val(word$(tak$(i),1","))
    if totTake > 0 then
        totWgt  = 0
        for j   = 2 to 10
            wgt    = val(word$(tak$(i),j","))
            totWgt = totWgt + wgt
            value  = wgt / wgt(j - 1) * price(j - 1)
            if wgt <> 0 then print name$(j-1);chr$(9);"Value: ";using("###.#",value);chr$(9);"Weight: ";using("##.#",wgt)
        next j
        print "-------- Total ";using("###.#",totTake);chr$(9);"Weight: ";totWgt
    end if
next i

--- Output:

Best 2 Options
salami Value: 285.0 Weight: 9.0
sausage Value: 98.0 Weight: 5.9
-------- Total 383.0 Weight: 14.9
salami Value: 475.0 Weight: 15.0
-------- Total 475.0 Weight: 15.0

---

Tcl

package require Tcl 8.5

# Uses the trivial greedy algorithm
proc continuousKnapsack {items massLimit} {
    # Add in the unit prices
    set idx -1
    foreach item $items {
        lassign $item name mass value
        lappend item [expr {$value / $mass}]
        lset items [incr idx] $item
    }

    # Sort by unit prices
    set items [lsort -decreasing -real -index 3 $items]

    # Add items, using most valuable-per-unit first
    set result {}
    set total 0.0
    set totalValue 0
    foreach item $items {
        lassign $item name mass value unit
        if {($total + $mass < $massLimit)} {
            lappend result [list $name $mass $value]
            set total [expr {$total + $mass}]
            set totalValue [expr {$totalValue + $value}]
        } else {
            set mass [expr {$massLimit - $total}]
        }
    }
```tcl
set value [expr {$unit * $mass}]
lappend result [list $name $mass $value]
set totalValue [expr {$totalValue + $value}]
break
}

# We return the total value too, purely for convenience
return [list $result $totalValue]
```

Driver for this particular problem:

```tcl
set items {
    {beef 3.8 36}
    {pork 5.4 43}
    {ham 3.6 90}
    {greaves 2.4 45}
    {flitch 4.0 30}
    {brawn 2.5 56}
    {welt 3.7 67}
    {salami 3.0 95}
    {sausage 5.9 98}
}
lassign [continuousKnapsack $items 15.0] contents totalValue
puts [format "total value of knapsack: %.2f" $totalValue]
puts "contents:"oreach item $contents {
    lassign $item name mass value
    puts [format "\t%.1fkg of %s, value %.2f" $mass $name $value]
}
```

Output:

```
total value of knapsack: 349.38
contents:
  3.0kg of salami, value 95.00
  3.6kg of ham, value 90.00
  2.5kg of brawn, value 56.00
  2.4kg of greaves, value 45.00
  3.5kg of welt, value 63.38
```

Ursala

We might as well leave this one to the experts by setting it up as a linear programming problem and handing it off to an external library (which will be either lpsolve (http://sourceforge.net/projects/lpsolve) or glpk (http://www.gnu.org/software/glpk/glpk.html) depending on the run-time system (http://www.basis.netii.net/avram) configuration).

```ursa
#import flo
#import lin
items = # name: (weight,price)
```
\[ \text{'beef '}: (3.8,36.0), \\
\text{'pork '}: (5.4,43.0), \\
\text{'ham '}: (3.6,90.0), \\
\text{'greaves': (2.4,45.0), \\
\text{'flitch ': (4.0,30.0), \\
\text{'brawn '}: (2.5,56.0), \\
\text{'welt '}: (3.7,67.0), \\
\text{'salami ': (3.0,95.0), \\
\text{'sausage': (5.9,98.0)>} \]

\[
\text{system} = \#\text{ a function to transform the item list to the data structure needed by the solver}
\]

\[
\text{linear_system}\[
\text{lower_bounds: *nS ~&\0.,} \quad \# \text{ all zeros because we can't steal less than zero}
\text{upper_bounds: ~&nmlPXS,} \quad \# \text{ can't steal more than what's in the shop}
\text{costs: * ^|/~& negative+ vid,} \quad \# \text{ prices divided by weights, negated so as to maximize}
\text{equations: ~&iNC\15.+ 1.-*@nS[] \# 1 equation constraining the total weight to 15}
\]

\#\text{cast %em}

\text{main} = \text{solution system items}

\text{output:}

\[
\text{'brawn ': 2.500000e+00,} \\
\text{'greaves': 2.400000e+00,} \\
\text{'ham ': 3.600000e+00,} \\
\text{'salami ': 3.000000e+00,} \\
\text{'welt ': 3.500000e+00>}
\]

**XPL0**

```
int Name, Price, I, BestItem;
real Weight, Best, ItemWt, TotalWt;
def Items = 9;
real PricePerWt(Items);
int Taken(Items);
include c:\cxpl\codes;

\{Name:= ["beef","pork","ham","greaves","flitch","brawn","welt","salami","sausage"];
Weight:= [ 3.8, 5.4, 3.6, 2.4, 4.0, 2.5, 3.7, 3.0, 5.9];
Price:= [ 36, 43, 90, 45, 30, 56, 67, 95, 98];

for I:= 0 to Items-1 do
[PricePerWt(I):= float(Price(I)) / Weight(I);
 Taken(I):= false;]
Format(2,1);
TotalWt:= 0.0;
repeat  Best:= 0.0;
for I:= 0 to Items-1 do
  if not Taken(I) and PricePerWt(I) > Best then
    [Best:= PricePerWt(I);  BestItem:= I];
Taken(BestItem):= true; \take item
ItemWt:= Weight(BestItem); \get its weight
TotalWt:= TotalWt + ItemWt; \add to total weight
```
if TotalWt > 15.0 then  
  ItemWt:= ItemWt - (TotalWt-15.0);  \ if total is too much, reduce  
  RlOut(0, ItemWt);  Text(0, " kg of ");  \ show weight and item  
  Text(0, Name(BestItem));  CrLf(0);  \ all we can steal  
until   TotalWt >= 15.0;  
}

Output:

3.0 kg of salami
3.6 kg of ham
2.5 kg of brawn
2.4 kg of greaves
3.5 kg of welt


Categories: Programming Tasks | Classic CS problems and programs | Ada | BBC BASIC | C | C++ | D
  | Forth | Fortran | Go | Groovy | Haskell | Icon | Unicon | Icon Programming Library | J | Java | Mathematica
  | Mathprog | OCaml | Perl | Perl 6 | PicoLisp | Prolog | PureBasic | Python | Racket | REXX | Run BASIC
  | Tcl | Ursala | XPL0 | Puzzles | GUISS/Omit

- This page was last modified on 5 June 2013, at 20:47.
- Content is available under GNU Free Documentation License 1.2.