Algorithms and Data Structures 2

19 – Advanced design techniques

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Recap on algorithm design techniques**
- **0-1 knapsack problem**
- **Dynamic programming**
- **Memoisation**
- **Fractional knapsack problem**
- **Greedy algorithms**

Recap

- **We have already studied three algorithm design techniques**

1. **Incremental approach**
   - INSERTION-SORT, SELECTION-SORT, HEAPSORT
2. **Divide an conquer**
   - MERGE-SORT, QUICKSORT
3. **Randomisation**
   - Randomised QUICKSORT, universal hashing

Knapsack problem

- **Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack**
  - Classical problem in combinatorial optimisation

- **There are two versions of this problem**

1. **0-1 knapsack problem**
   - Items are indivisible; you either take an item or not

2. **Fractional knapsack problem**
   - Items are divisible: you can take any fraction of an item

0-1 knapsack problem

- **Given two n-tuples of positive integers v[1,..,n] and w[1,..,n] and W > 0 determine subset S ⊆ {1,..,n} that maximises subject to**
  - v_i is the value of i-th item
  - w_i is the weight of i-th item

- **Example**

Knapsack with
capacity W = 16

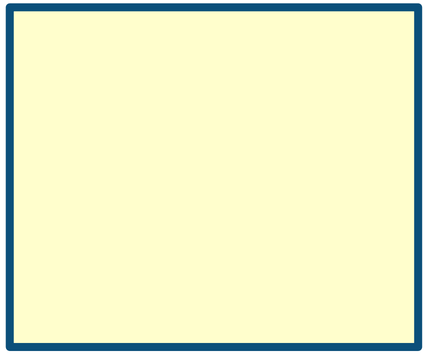
Items	value	weight
1	5	12
2	3	5
3	4	3
4	2	1

0-1 knapsack problem (cont.)

- **Given two n-tuples of positive integers v[1,..,n] and w[1,..,n] and W > 0 determine subset S ⊆ {1,..,n} that maximises subject to**
  - v_i is the value of i-th item
  - w_i is the weight of i-th item

- **Example**

Knapsack with
capacity W = 16



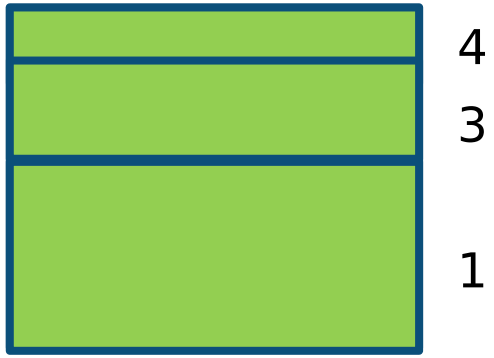
Items	value	weight
1	5	12
2	3	5
3	4	3
4	2	1

0-1 knapsack problem (cont.)

- **Given two n-tuples of positive integers v[1,..,n] and w[1,..,n] and W > 0 determine subset S ⊆ {1,..,n} that maximises subject to**
  - v_i is the value of i-th item
  - w_i is the weight of i-th item

- **Example**

Knapsack with
capacity W = 16



Maximum value 11
S = {1,3,4}

Items

value	weight
5	12
3	5
4	3
2	1

0-1 knapsack problem (cont.)

- **This is an optimisation problem**
- **Since there are n items, there are 2^n possible combinations of items**
- **Brute force algorithm**
  - Go through all combinations and find the one with maximum value and with total weight less or equal to W
  - Running time is $O(2^n)$
- **Can we do better?**

Divide and conquer algorithm

- **Consider all subsets of items and calculate the total weight and value of each subset**
  - Consider only subsets whose total weight is smaller than W

- **From all such subsets, select the subset with maximum total value**
  - For each item, select the maximum of two cases

1. **An item is included in the optimal subset**
   - Value of n-th item plus maximum value obtained by n-1 items and W minus weight of n-th item

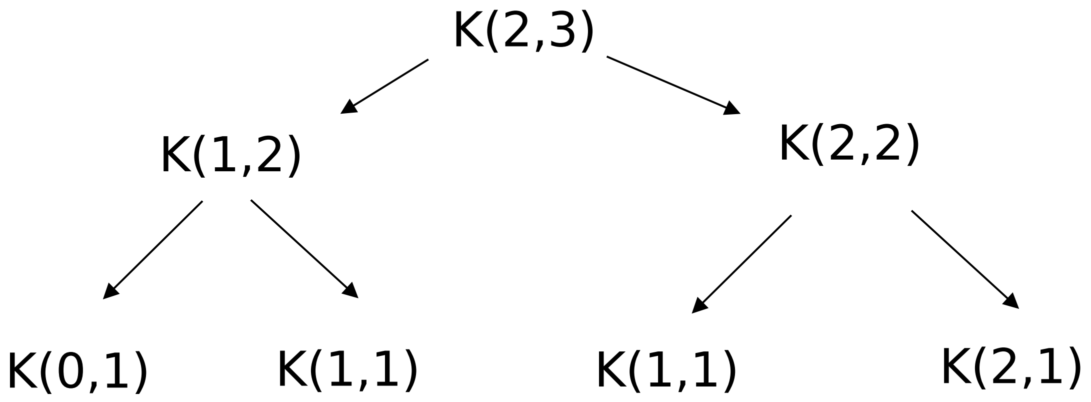
2. **An item is not included in the optimal set**
   - Maximum value obtained by n-1 items and W weight

Pseudocode

```
KNAPSACK-REC(W,w,v,n)
  if n < 1 or W = 0
    return 0
  // weight of n-th item is more than W
  if w[n] > W
    return KNAPSACK-REC(W,w,v,n-1)
  else
    // n-th item included
    a := v[n] + KNAPSACK-REC(W-w[n],w,v,n-1)
    // n-th item not included
    b := KNAPSACK-REC(W,w,v,n-1)
    return MAX(a,b)
```

Recursion tree

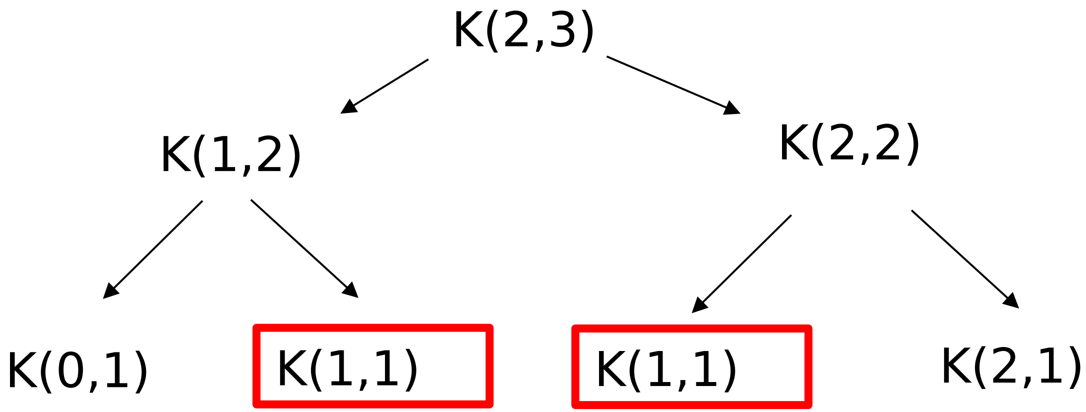
- **W = 2, n = 3, v = [5, 8 ,7] , w = [1,1,1]**
- **K(W,n) = KNAPSACK-REC(W,w,v,n)**




```
KNAPSACK-REC(W,w,v,n)
  if n < 1 or W = 0
    return 0
  // weight of n-th item is more than W
  if w[n] > W
    return KNAPSACK-REC(W,w,v,n-1)
  else
    // n-th item included
    a := v[n] + KNAPSACK-REC(W-w[n],w,v,n-1)
    // n-th item not included
    b := KNAPSACK-REC(W,w,v,n-1)
    return MAX(a,b)
```

Recursion tree

- **W = 2, n = 3, v = [5, 8 ,7] , w = [1,1,1]**
- **K(W,n) = KNAPSACK-REC(W,w,v,n)**



Subproblems are not independent

```
KNAPSACK-REC(W,w,v,n)
  if n < 1 or W = 0
    return 0
  // weight of n-th item is more than W
  if w[n] > W
    return KNAPSACK-REC(W,w,v,n-1)
  else
    // n-th item included
    a := v[n] + KNAPSACK-REC(W-w[n],w,v,n-1)
    // n-th item not included
    b := KNAPSACK-REC(W,w,v,n-1)
    return MAX(a,b)
```

Overlapping subproblems

- **When subproblems overlap, then a divide and conquer algorithm repeatedly solves the common sub-subproblems**
  - More work than necessary!

- **Time complexity of this naive recursive solution is exponential ($O(2^n)$)**

- **We will study how to speed this up later in this lecture**

Dynamic programming

- **R. Bellman began the systematic study of dynamic programming in 1955**
  - The word “programming” refers to using a tabular solution method

- **Applies to optimisation problems in which we make a set of choices in order to arrive at an optimal solution**

- **Effective when a given subproblem may arise from more than one partial set of choices (overlapping subproblems)**
  - Store the solution to each subproblem in a table so they can be reused repeatedly later
  - We trade space for time

Dynamic programming (cont.)

- **A dynamic programming (DP) algorithm consists of a sequence of four steps**

1. **Structure**
   - Characterise the structure of an optimal solution
2. **Principle of optimality**
   - Recursively define the value of an optimal solution
3. **Bottom-up computation**
   - Compute the value of an optimal solution using a table
4. **Construction of an optimal solution**
   - Use computed information to construct an optimal solution

Dynamic programming (cont.)

- **A dynamic programming (DP) algorithm consists of a sequence of four steps**

1. **Structure**
   - Characterise the structure of an optimal solution
2. **Principle of optimality**
   - Recursively define the value of an optimal solution
3. **Bottom-up computation**
   - Compute the value of an optimal solution using a table
4. **Construction of an optimal solution**
   - Use computed information to construct an optimal solution

Steps 1-3 form the basis of a dynamic programming solution to a problem

Step 4 can be omitted if only the value of an optimal solution is required

DP algorithm for 0-1 knapsack

1. **Structure**

   – Construct a table V[0,..,n][0,..,W]

   – For $1 \leq i \leq n$ and $0 \leq j \leq W$, entry V[i][j] stores the maximum total value of any subset of items {1,..,i} of combined weight at most j

   – Entry V[n][W] contains the maximum weight of the items that can fit into a knapsack of capacity W

DP algorithm for 0-1 knapsack

2. Principle of optimality

- Similar to recursive definition of the problem used in the divide and conquer algorithm

- Base: $V[0][j] = 0$ with $0 \leq j \leq W$ (set with no items)
  $V[i][j] = -\infty$ with $j < 0$ (illegal)

- Recursive step: $V[i][j] = \max(V[i-1][j], v[i] + V[i-1][j-w[i]])$ where $1 \leq i \leq n$ and $0 \leq j \leq W$

DP algorithm for 0-1 knapsack

3. **Bottom-up computation**

   - Use iteration instead of recursion to compute table V row by row

   - Use the characterisation defined in step 2

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								

weigh
t

items

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1								
2								
3								
4								

$i = 0$

Base: $V[0][j] = 0$ for $0 \leq j \leq 7$

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2								
3								
4								

i = 1 v[1] = 5 w[1] = 2

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[1][2] = max(V[0][2], v[1] + V[0][2-w[1]])

= max(0, 5 + V[0][2-2])

= max(0, 5 + V[0][0])

= max(0, 5 + 0) = 5

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3								
4								

i = 2   v[2] = 2   w[2] = 3

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[2][5] = max(V[1][5], v[2] + V[1][5-w[2]])

= max(5, 2 + V[1][5-3])

= max(5, 2 + V[1][2])

= max(5, 2 + 5) = 7

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4								

i = 3 v[3] = 1 w[3] = 4

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4						

i = 4 v[4] = 4 w[4] = 1

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[4][1] = max(V[3][1], v[4] + V[3][1-w[4]])
= max(0, 4 + V[3][1-1])
= max(0, 4 + V[3][0])
= max(0, 4 + 0) = 4

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5					

i = 4 v[4] = 4 w[4] = 1

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[4][2] = max(V[3][2], v[4] + V[3][2-w[4]])
= max(5, 4 + V[3][2-1])
= max(5, 4 + V[3][1])
= max(5, 4 + 0) = 5

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9				

i = 4 v[4] = 4 w[4] = 1

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[4][3] = max(V[3][3], v[4] + V[3][3-w[4]])

= max(5, 4 + V[3][3-1])

= max(5, 4 + V[3][2])

= max(5, 4 + 5) = 9

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	

i = 4 v[4] = 4 w[4] = 1

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[4][6] = max(V[3][6], v[4] + V[3][6-w[4]])

= max(7, 4 + V[3][6-1])

= max(7, 4 + V[3][5])

= max(7, 4 + 7) = 11

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

i = 4 v[4] = 4 w[4] = 1

Recursive step:

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j-w[i]])

V[4][7] = max(V[3][7], v[4] + V[3][7-w[4]])

= max(7, 4 + V[3][7-1])

= max(7, 4 + V[3][6])

= max(7, 4 + 7) = 11

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

A knapsack with capacity W = 7 and four items as in v and w can contain a combination of items with total value at most V[4][7] = 11

Which items are in the optimal solution?

KNAPSACK algorithm

- **DP algorithm for 0-1 knapsack**
- **Running time is O(nW)**

```
KNAPSACK(W,w,v,n)
  for j = 0 to W
    V[0][j] := 0
  for i = 1 to n
    for j = 0 to W
      if w[i] ≤ j
        V[i][j] := MAX(V[i-1][j], v[i] + V[i-1][j-w[i]])
      else
        V[i][j] := V[i-1][j]
  return V[n][W]
```

DP algorithm for 0-1Knapsack

4. Construction of an optimal solution

– Use table V to construct an optimal solution

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

V[4][7] != V[3][7]

Mark 4

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

Repeat the loop

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

V[3][6] = V[2][6]

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

Repeat the loop

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

V[2][6] != V[1][6]

Mark 2

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

Repeat the loop

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

V[1][3] != V[0][3]

Mark 1

Example

- **Find a combination of items that maximise the value of a knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

```
i := n
k := W
while i > 0 and k > 0
  if V[i][k] != V[i-1][k]
    mark i as in the knapsack
    i := i – 1
    k := k – w[i]
  else
    i := i - 1
```

Termination

Optimal combination is {1,2,4}

Top-down with memoisation

- **Alternative way to implement a dynamic programming approach**

- **Recursive algorithms with overlapping subproblems are made more efficient by saving the result of each subproblem (usually in an array or hash table)**
  - The algorithm first checks whether it has previously solved this subproblem
  - If so, it returns the saved value, saving further computation at this level
  - If not, the procedure computes the value in the usual manner

- **We say that the recursive procedure has been memoised**
  - From “memo”
  - The algorithm remembers what results it has already computed

Example

- T[0,..,W][0,…n] is initialised to -1

```
KNAPSACK-MEMO(W,w,v,n,T)
  if n < 1 or W = 0
    return 0
  if T[W][n] != -1
    return T[W][n]
  if w[n] > W
    T[W][n] := KNAPSACK-MEMO(W,w,v,n-1)
  else
    a := v[n] + KNAPSACK-MEMO(W-w[n],w,v,n-1)
    b := KNAPSACK-MEMO(W,w,v,n-1)
    T[W][n] := MAX(a,b)
    return T[W][n]
```

Fractional knapsack

- **Same as 0-1 knapsack but fractional amounts of each object can be included in the knapsack**

- **Formally, maximise subject to where $0 \leq f_i \leq 1$**

- **How to find a solution**
  - Sort items in non decreasing order of their value/weight ratio
  - When an object is considered choose a maximal amount such that the knapsack capacity is not violated

- **Greedy approach**

Greedy algorithms

- **Applies to optimisation problems in which we make a set of choices in order to arrive at an optimal solution**
  - Each choice in made in a locally optimal manner

- **A greedy approach provides an optimal solution for many problems much more quickly than a dynamic programming approach**

- **We cannot always easily tell whether a greedy approach will be effective**
  - Only when the greedy choice property holds: an optimal solution can be obtained by making the greedy choice at each step

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity $W = 7$**
  - $v = [5,2,1,4]$ and $w = [2,3,4,1]$

  - Compute the value/weight ratios $r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]$
  - Sort v and w according to r: $v = [4, 5, 2, 1]$ and $w = [1, 2, 3, 4]$

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]
  - Compute the value/weight ratios r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]
  - Sort v and w according to r: v = [4, 5, 2, 1] and w = [1, 2, 3, 4]
  - Add maximal amount of 4 to knapsack (available space 7 – 1 = 6)
  - Solution = 4 * 1

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]
  - Compute the value/weight ratios r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]
  - Sort v and w according to r: v = [4, 5, 2, 1] and w = [1, 2, 3, 4]
  - Add maximal amount of 4 to knapsack (available space 7 – 1 = 6)
  - Add maximal amount of 5 to knapsack (available space 6 – 2 = 4)
  - Solution = 4 * 1 + 5 * 1

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]
  - Compute the value/weight ratios r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]
  - Sort v and w according to r: v = [4, 5, 2, 1] and w = [1, 2, 3, 4]
  - Add maximal amount of 4 to knapsack (available space 7 – 1 = 6)
  - Add maximal amount of 5 to knapsack (available space 6 – 2 = 4)
  - Add maximal amount of 2 to knapsack (available space 4 – 3 = 1)
  - Solution = 4 * 1 + 5 * 1 + 2 * 1

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

  - Compute the value/weight ratios r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]
  - Sort v and w according to r: v = [4, 5, 2, 1] and w = [1, 2, 3, 4]
  - Add maximal amount of 4 to knapsack (available space 7 – 1 = 6)
  - Add maximal amount of 5 to knapsack (available space 6 – 2 = 4)
  - Add maximal amount of 2 to knapsack (available space 4 – 3 = 1)
  - Add maximal amount of 1 to knapsack (available space 1 – (4 * ¼) = 0)

  - Solution = 4 * 1 + 5 * 1 + 2 * 1 + 1 * ¼

Example

- **Find a combination of items that maximise the value of a fractional knapsack with maximum capacity W = 7**
  - v = [5,2,1,4] and w = [2,3,4,1]

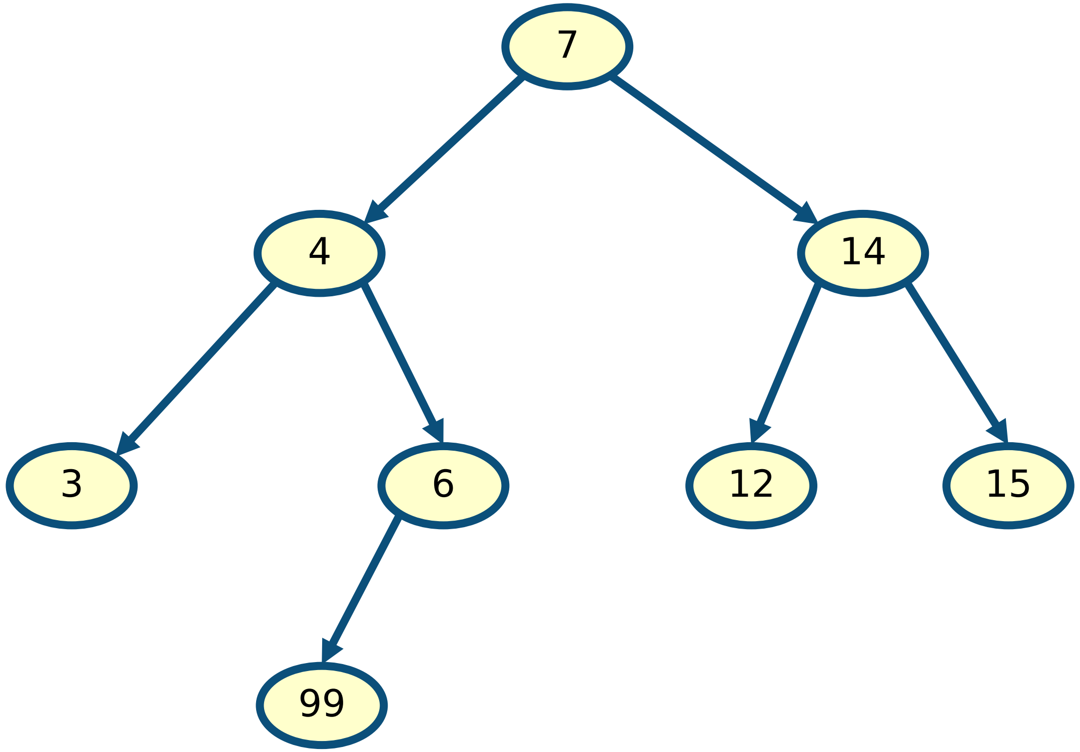
  - Compute the value/weight ratios r = [5/2, 2/3, 1/4, 4/1] = [2.5, 0.67, 0.25, 4]
  - Sort v and w according to r: v = [4, 5, 2, 1] and w = [1, 2, 3, 4]
  - Add maximal amount of 4 to knapsack (available space 7 – 1 = 6)
  - Add maximal amount of 5 to knapsack (available space 6 – 2 = 4)
  - Add maximal amount of 2 to knapsack (available space 4 – 3 = 1)
  - Add maximal amount of 1 to knapsack (available space 1 – (4 * ¼) = 0)

  - Solution = 4 * 1 + 5 * 1 + 2 * 1 + 1 * ¼ = 11.25

Another example

- **Find the path from the root to a leaf with the maximum sum in the tree on the right**

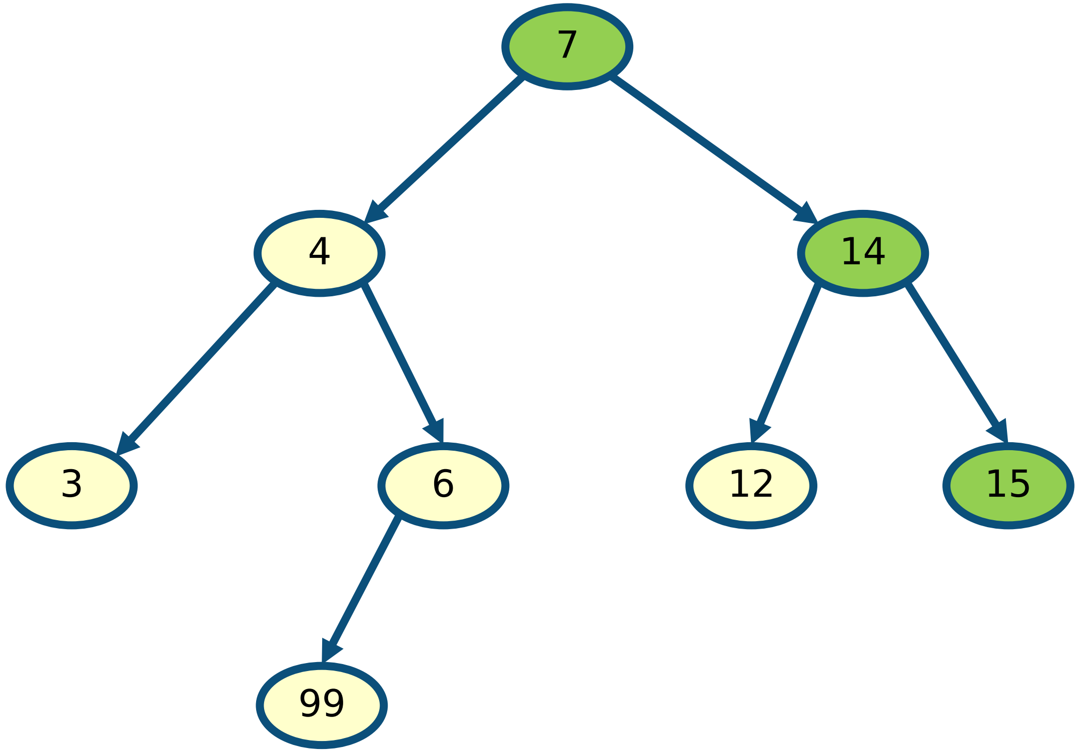
- **Pick the maximum at each step**



Another example

- **Find the path from the root to a leaf with the maximum sum in the tree on the right**

- **Pick the maximum at each step**
  - 7 + 14 + 15 = 36



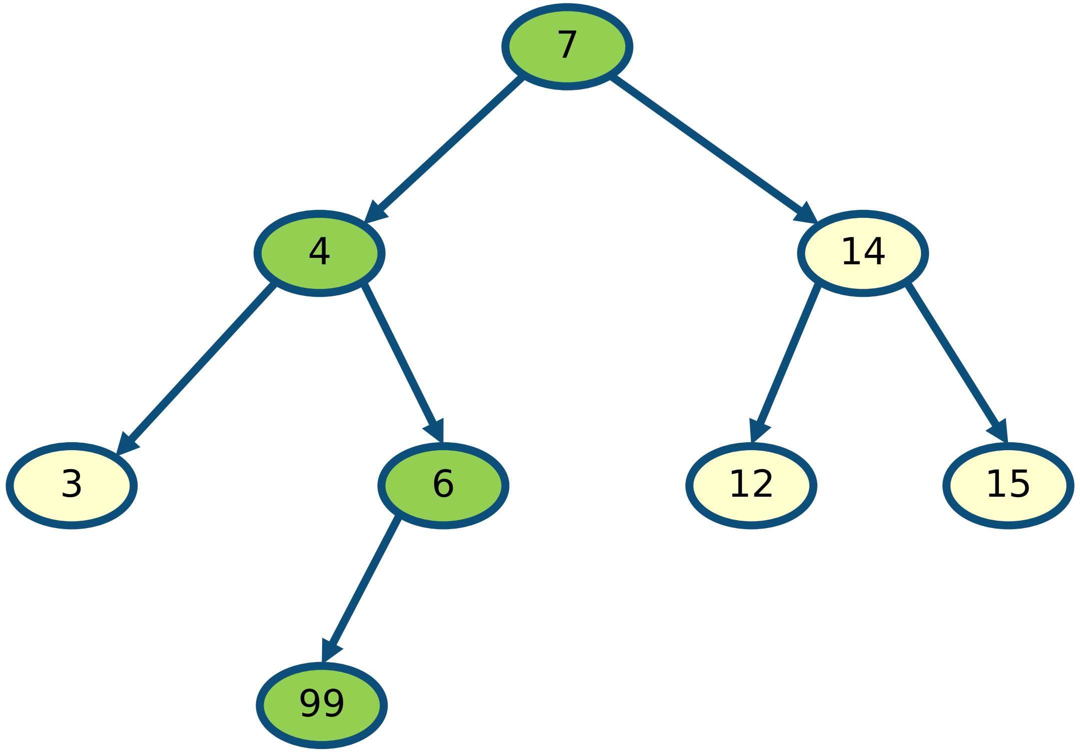
Another example

- **Find the path from the root to a leaf with the maximum sum in the tree on the right**

- **Pick the maximum at each step**
  - 7 + 14 + 15 = 36

- **The correct solution is**
  - 7 + 4 + 6 + 99 = 116

- **The greedy choice property does not hold in this case**



Summary

- Recap on algorithm design techniques
- 0-1 knapsack problem
- Dynamic programming
- Memoisation
- Fractional knapsack problem
- Greedy algorithms