Knapsack Problem

Backtracking

Subset Sum Problem

Given a set W of n elements and a sum S

We have to find all possible subsets T of W, whose elements add up to at most S, such that there is no other subset of W whose elements add up to more than that of T.

e.g. W = { 8 6 7 5 3 10 9 } and S = 15

We get four subsets

{ 8 7 }

{ 6 9 }

{ 7 5 3 }

{ 5 10 }

Application: say we have a CPU with S free cycles and a set W of n jobs, we want to choose the subset of jobs (each job i taking W[i] time) that minimizes the number of idle cycles.

Subset Sum Problem

The subset sum problem can be stated as follows:

Given a set W of n elements and a sum S

We have to find a subset T of W, whose elements add up to <u>at most</u> S, such that there is no other subset of W whose elements add up to more than that of T.

In other words it is an optimization problem denoted as:

$$SubsetSum(S,k) = \begin{cases} SubsetSum(S, k-1), if\ W_k > S \\ \max\begin{cases} SubsetSum(S, k-1) \\ W_k + SubsetSum(S - W_k, k-1)' \end{cases} & otherwise \end{cases}$$

Subset Sum Problem algorithm using Backtracking

```
Algorithm subsetSum(s, k, x)
// finds the maximal subset of the given set w[1 … n] of n elements, whose elements add up to at most s
// x[1 … n] represents the solution:
//        x[k] = 0 implies that the kth element of w is NOT taken, and x[k] = 1 implies that the kth element of w is taken
// initial call is: subsetSum(s, n, x = {})
    if (k == 0) return 0
    let y[1 … k], z[1 … k] be two arrays
    if (w[k] ≤ s) sumWithk = w[k] + subsetSum(s - w[k], k - 1, y)
    else sumWithk = 0
    sumWithoutk = subsetSum(s, k - 1, z)
    if (sumWithk > sumWithoutk)
        for (i = 1 to k - 1) x[i] = y[i]
        x[k] = 1
        return sumWithk
    else
        for (i = 1 to k - 1) x[i] = z[i]
        x[k] = 0
        return sumWithoutk
```

The 0/1 knapsack problem can be stated as follows:

Given a set of n elements, each element k having weight W_k and value V_k , and a knapsack capacity c

We have to find a subset T of the given set, whose weights add up to at most c, such that there is no other subset whose values add up to more than that of T.

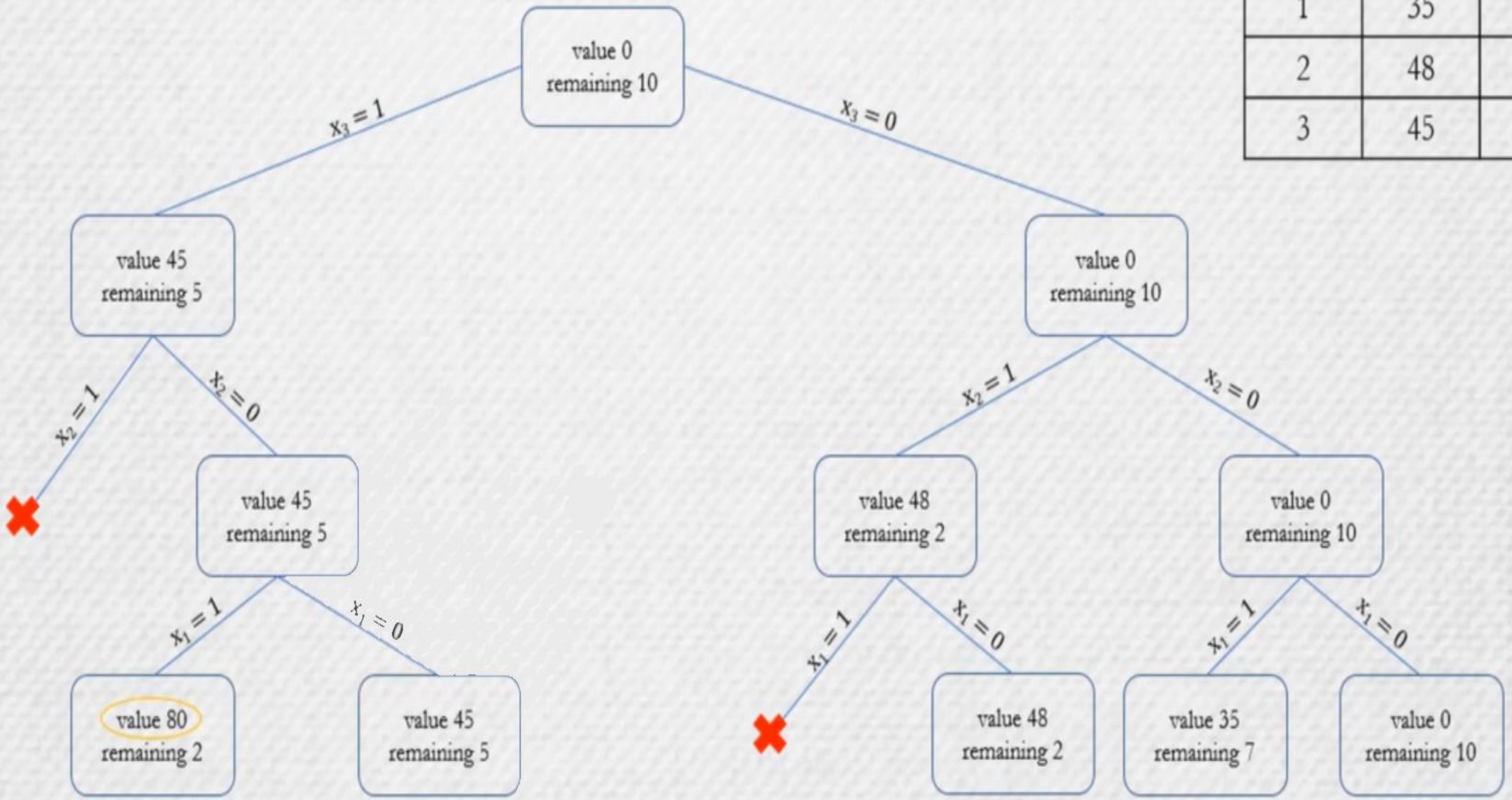
In other words it is an optimization problem denoted as:

$$knapsack(c,k) = \begin{cases} knapsack(c, k-1), if\ W_k > c \\ \max\begin{cases} knapsack(c, k-1) \\ V_k + knapsack(c - W_k, k-1)' \end{cases} \quad otherwise \end{cases}$$

Difference with Subset Sum: want to maximize value instead of weight.

```
Algorithm knapsack(c, k, x)
// finds a subset with maximal value, of the given set of n elements having weights w[1 … n] and values v[1 … n]
// whose elements' weights add up to at most c (the knapsack capacity)
// x[k] = 0 implies that the kth element of w is NOT taken, and x[k] = 1 implies that the kth element of w is taken
// initial call is: knapsack(c, n, x = {})
    if (k == 0) return 0
    let y[1 … k], z[1 … k] be two arrays
    if (w[k] ≤ c) valWithk = v[k] + knapsack(c - w[k], k - 1, y)
    else valWithk = 0
    valWithoutk = knapsack(c, k - 1, z)
    if (valWithk > valWithoutk)
        for (i = 1 to k - 1) x[i] = y[i]
        x[k] = 1
        return valWithk
    else
        for (i = 1 to k - 1) x[i] = z[i]
        x[k] = 0
        return valWithoutk
```

Capacity = 10		
item	value	weight
1	35	3
2	48	8
3	45	5



Number of nodes explored = 11