COMP 6651

Algorithm Design Techniques

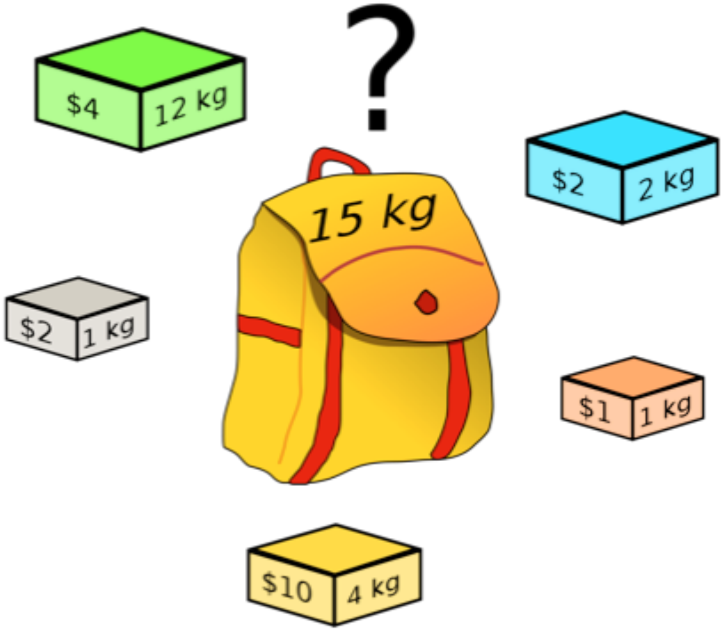
Week 5

Dynamic Programming.

(some material is taken from web or other various sources with permission)

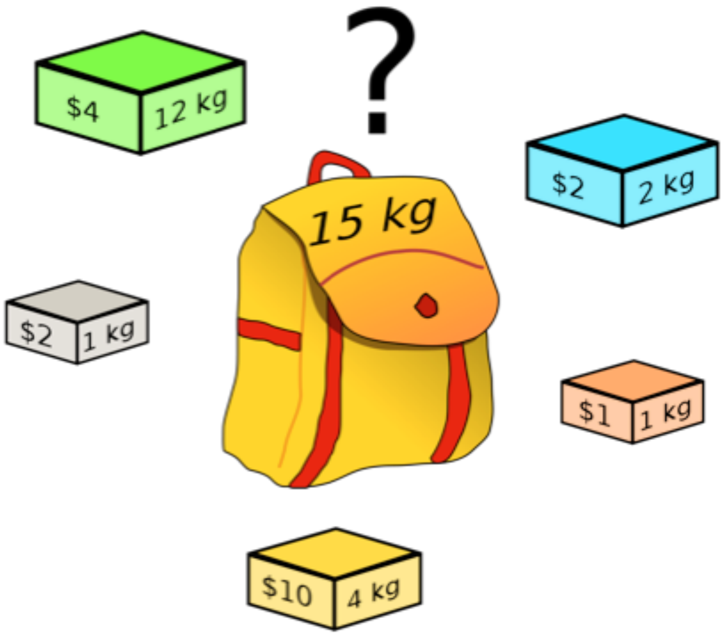
Knapsack Problem 1

- I have a collection of n objects
- Each object has a weight and a value
- Pack objects such that we maximize the value
- Given a maximum weight
- Assume I have infinitely many objects of each type



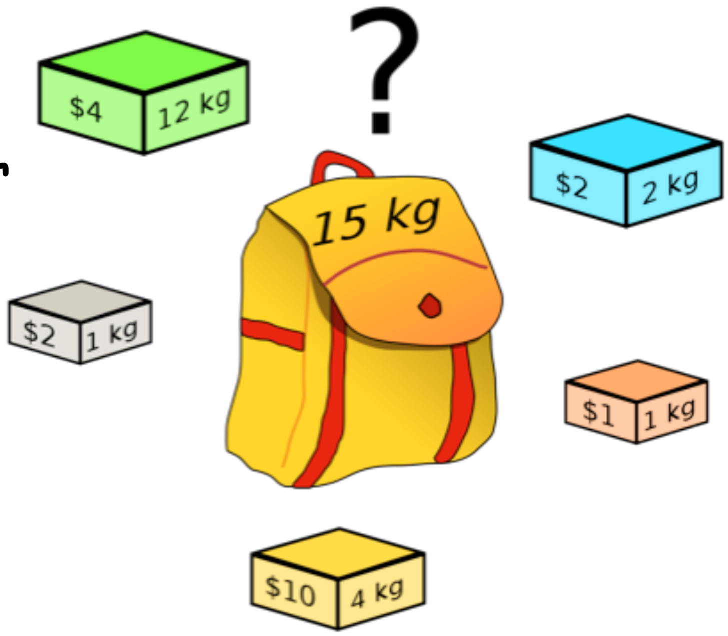
Knapsack Problem 1

- I have a collection of n objects
- Each object has a weight and a value
- Pack objects such that we maximize the value
- Given a maximum weight
- Assume I have infinitely many objects of each type

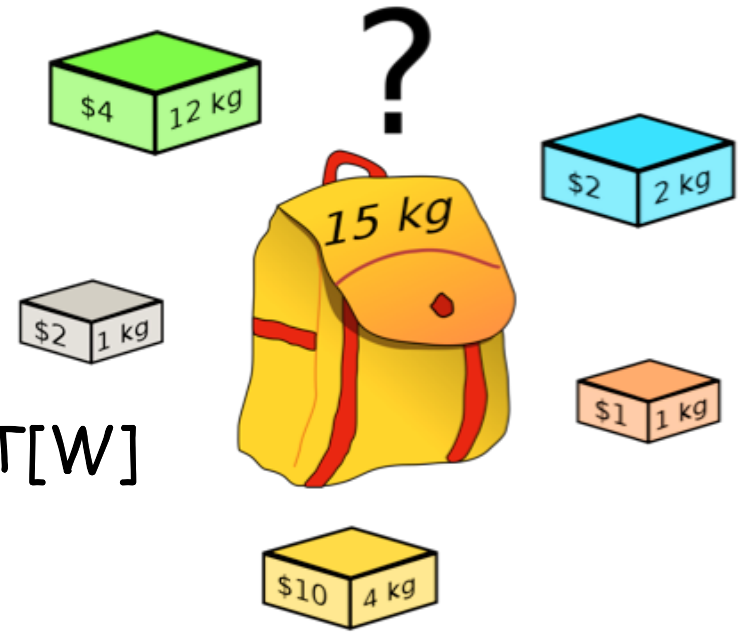


Knapsack Problem 1

- Greedy strategy:
- Pick the elements most valuable per unit of weights
- Example:
  - Values = {11/3, 2}
  - Weights = {5, 3}
  - W = 6
  - Ratios = {11/15, 10/15}
  - Greedy solution: Select first element
  - Optimal solution: select the second and third



Knapsack Problem 1



- Dynamic Programming strategy:
- Values v[i], i=1..n
- Weights w[i], i=1..n
- Maximum weight W
- Want maximum value of elements: T[W]
- What can we say about T:
- T[0] = 0
- T[negative number]=- inf (convention for invalid case)
- Greedy not optimal, so we need to look back
- In a smart way
- $T[W] = max_i(v[i] + T[W - w[i]])$

Knapsack Problem 1

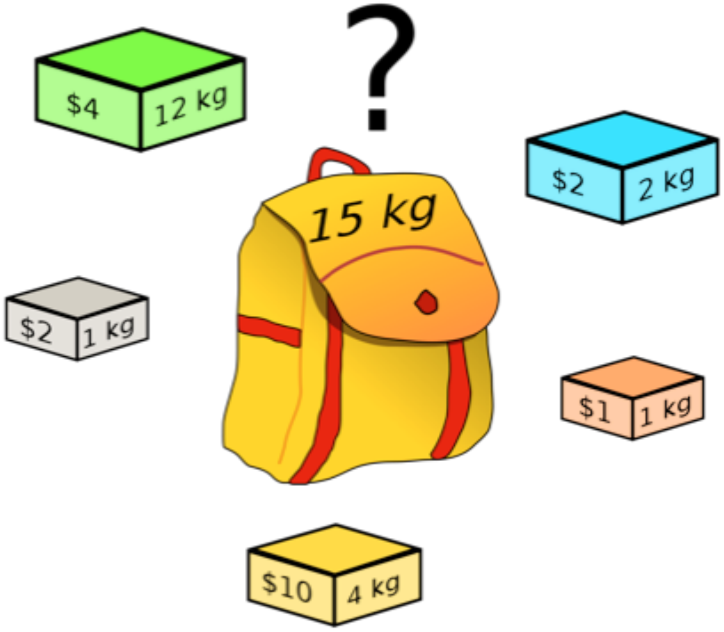
```
int knapsack1(int W, int values[], int weights[], int n){
    define T[0..W];
    T[0] = 0;

    for(int i=1;i<=W;++i){

        T[i] = 0;
        for(int j=0;j<n;++j){
            if(i-weights[j]>=0){
                int tmp = values[j] + T[i-weights[j]]);
                if(T[i]<tmp)
                    T[i] = tmp;
            }
        }
    }

    return T[W];
}
```

Knapsack Problem 1

```
int knapsack1(int W, int values[], int weights[], int n, int solution[]){
    define T[0..W];
    define J[0..W];
    T[0] = 0;
    J[0] = -1;
    for(int i=1;i<=W;++i){
        T[i] = 0;
        J[i] = -1;
        for(int j=0;j<n;++j){
            if(i-weights[j]>=0){
                int tmp = values[j] + T[i-weights[j]]);
                if(T[1]<tmp){
                    T[i] = tmp;
                    J[i] = j;
                }
            }
        }
    }
    while(W>=0 && J[W]>=0){
        solution.push_back(J[W]);
        W-=T[J[W]];
    }
    return T[W];
}
```

Knapsack Problem 2

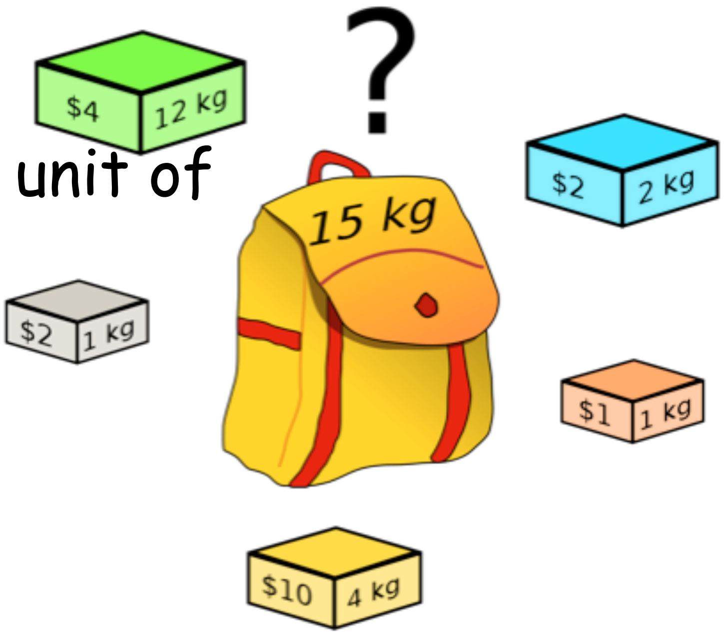
- I have a collection of n objects
- Each object has a weight and a value
- Pack objects such that we maximize the value
- Given a maximum weight
- I have a fixed number of each object



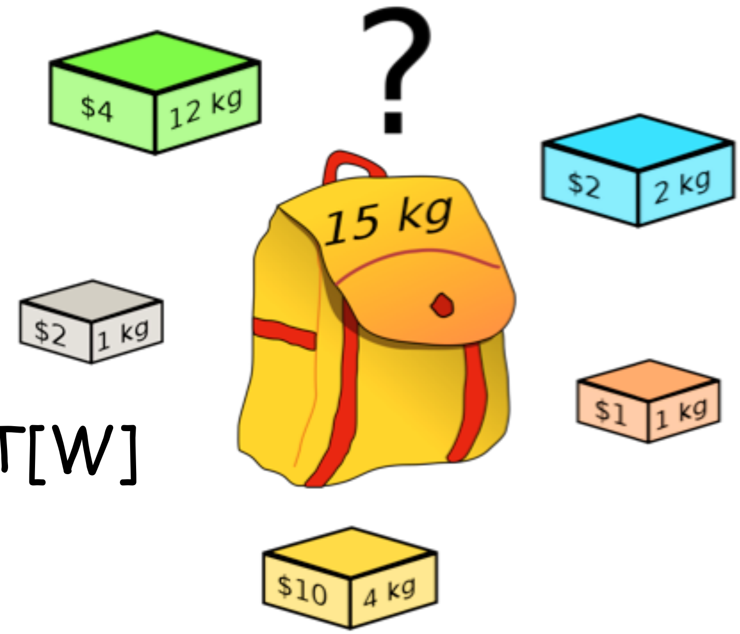
UNIVERSITÉ Concordia UNIVERSITY

Knapsack Problem 2

- Greedy strategy:
- Pick the elements most valuable per unit of weights
- Example:
  - Values = {11/3, 2, 2}
  - Weights = {5, 3, 3}
  - W = 6
  - Ratios = {11/15, 10/15, 10/15
  - Greedy solution: Select first element
  - Optimal solution: select the second and third



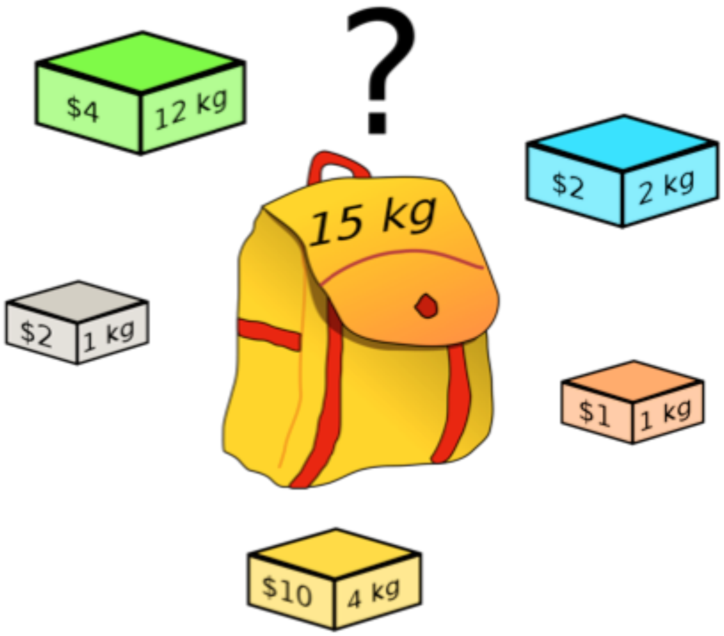
Knapsack Problem 2



- Dynamic Programming strategy:
- Values v[i], i=1..n
- Weights w[i], i=1..n
- Maximum weight W
- Want maximum value of elements: T[W]
- What can we say about T:
- T[0] = 0
- T[negative number]=- inf (convention for invalid case)
- Greedy not optimal, so we need to look back
- In a smart way
- $T[w,i] = max(T[w,i-1], v[i] + T[W-w[i], i-1])$

Knapsack Problem 2

- Dynamic Programming strategy:
- $T[W, i] = max_{j \le i}(v[j] + T[W - w[j], i - 1])$
- Boundary conditions:
  - $T[0, i] = 0 \; \forall 0 \le i < n$
  - $T[w, 0] = 0 \; \forall 0 \le w \le W$



Knapsack Problem 2

```
Int knapsack2(intW,intvalues[],intweights[],intn){
   define T[0..W,1..n];
   values[1..n]
   weight[1..n]

   // initialization
   for(inti=0;i<=W;++i)
       T[i,0] =0;
   for(inti=0;i<n;++i)
       T[0, i] =0;

   for(inti=1;i<=W;++i){
       for(intj=1;j<n;++j){

           int left = T[i, j-1];
           int right =0;
           if(i - w[h]>=0)
               right = v[j] + T[i - w[h], j-1];

           T[i,j] = max(left, right);
   }

   return T[W, n];
}
```

Knapsack Problem 2

```
Int knapsack2(intW,intvalues[],intweights[],intn){
   define T[0..W,1..n];
   Define R[0..W, 1..n];// stores the recursion link
   values[1..n] weight[1..n]
   // initialization
   for(inti=0;i<=W;++i){
       T[i,0] =0;
          R[I,0] = 0;
   }
   for(inti=0;i<n;++i){
          R[0, i] = 0;
       T[0, i] =0;
    }
   for(inti=1;i<=W;++i){
       for(intj=1;j<n;++j){

           int left = T[i, j-1];
           int right =0;
           if(i – w[h]>=0)
               right = v[j] + T[i – w[h], j-1];

            if(left>right){
                 R[i,j]=1;
               T[i,j] = left;
          } else {
                    R[i,j]=2;
               T[I,j] = right;
           }

   }

   return T[W, n];
}
```

Knapsack Problem 2

```
objects [];
    total =0;
    intW,int n;
    while(R[W, n]!=0){
                if(R[W, n]==1){
                    }else{
                    objects.push_back(n);
                    total+=value[n];
                    W-=weight[n];
                }

              n -=1;

    }
```

Longest Common Subsequence

- DNA matching

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

GTCGTCGGAAGCCGGCCGAA

Longest Common Subsequence

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

GTCGTCGGAAGCCGGCCGAA

- Optimal subsequence
- Recurrences like before
- Need to make some keen observations
- Previous examples:
  - Knapsack 1 → what is the latest element added
  - Knapsack 2 → Do I use the element n or not
  - ?

Longest Common Subsequence

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

GTCGTCGGAAGCCGGCCGAA

- Optimal subsequence
- Recurrences like before
- Are the last 2 characters the same of S1 and S2?
- Why I ask this question?
- Because if they are:
  - The last element belongs to the optimal subsequence so we can focus on the smallest subsequence (we found our recurrence)
- What if they are not?

Longest Common Subsequence

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

GTCGTCGGAAGCCGGCCGAA

- Are the last 2 characters the same of S1 and S2?
- Why I ask this question?
- Because if they are:
  - The last element belongs to the optimal subsequence so we can focus on the smallest subsequence (we found our recurrence)
- What if they are not?
  - It means that at least oen of them is not part of the common subsequence

Longest Common Subsequence

$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

GTCGTCGGAAGCCGGCCGAA

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

- Who are i, j
- Look for?
- C[n, n]

Longest Common Subsequence

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

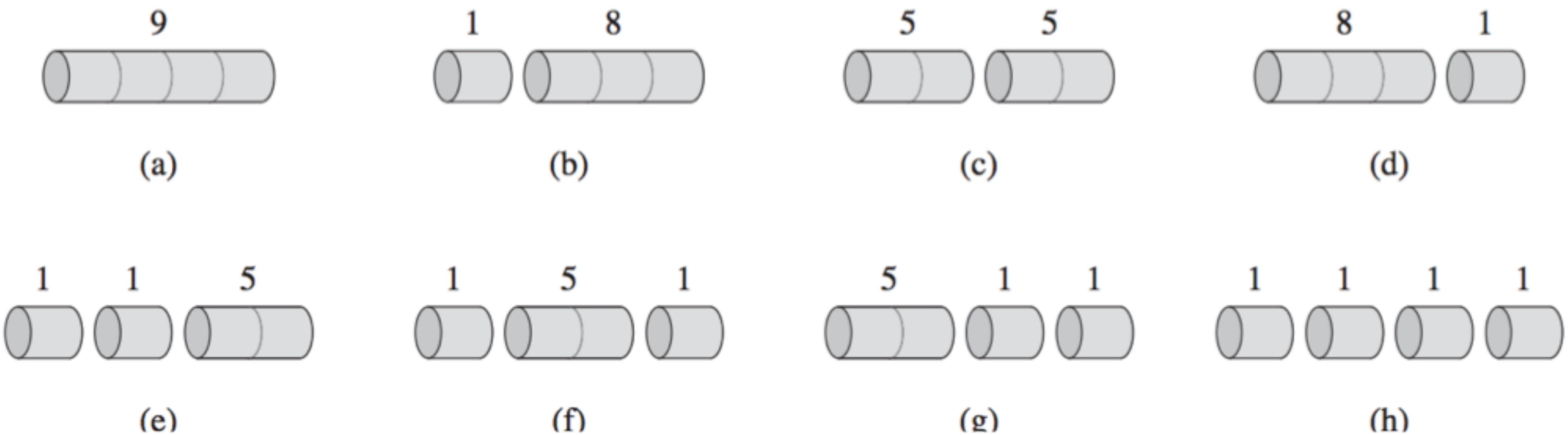
Longest Common Subsequence

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Rod cutting



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

Rod cutting

BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Putting things together

- Algorithm Analysis
- Recursion
- Divide and Conquer
- Greedy
- Dynamic Programming
- Theory and Practice

Putting things together

- Algorithm Analysis
  - Theoretical framework to assess algorithm performance
  - Compare algorithms
  - Assess their practicality

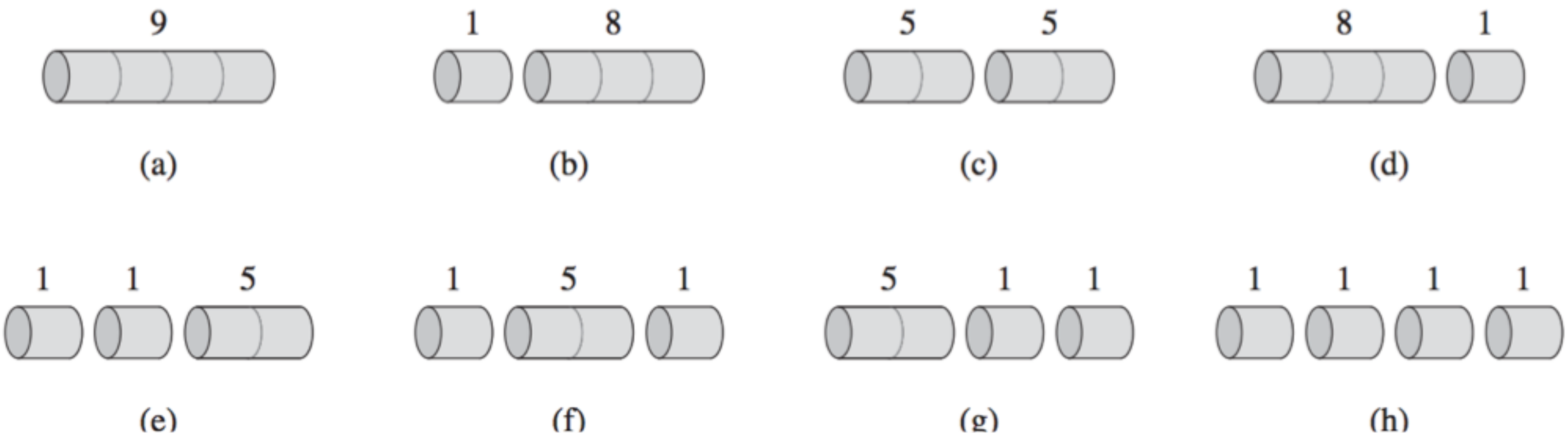
Putting things together

- Recursion
  - Algorithmic concept of solving a problem by repeating calls to the same code
  - Intuitive for many problems
  - Difficult to analyse
  - Master Theorem!!!!!

Putting things together

- Divide and Conquer
  - Algorithmic strategy of breaking down the problem into sub-problems
  - Intuitive
  - Can use recursion
  - What can go wrong?

Rod cutting



(a) (b) (c) (d) (e) (f) (g) (h)

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

Rod cutting

```
CUT-ROD(p, n)
    if n == 0
        return 0
    q = −∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n − i))
    return q
```

Recursive divide and conquer approach

Rod cutting

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$$T(n) = 2^n$$

Recursive divide and conquer approach

Putting things together

- Divide and Conquer
  - Algorithmic strategy of breaking down the problem into sub-problems
  - Intuitive
  - Can use recursion
  - What can go wrong?
  - Unnecessary repeats
  - How to fix?
  - Dynamic Programming
  - Memoization

Putting things together

- Dynamic Programming
  - Very efficient
  - Pseudo polynomial
  - Example: knapsack O(nW), what if W = 2^n?
  - Even if it is polynomial in n, it can have a large degree
  - What to do?
  - Try Greedy….

Putting things together

- Greedy
  - Very efficient when it works
  - Rarely works
  - Some cool famous problems
  - Mostly sub-optimal approximations