Algorithms and Datastructures

Lecture 9

Manfred Jaeger

AALBORG UNIVERSITET

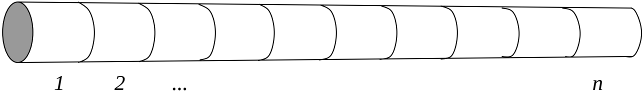
Dynamic Programming

Not a style of computer programming – a method for solving optimization problems developed in the 1950's:

> *My first task was to find a name for multistage decision processes [...] I felt I had to do something to shield [...] the Air Force from the fact that I was really doing mathematics [...] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.*

(Richard Bellman, Autobiography, 1984)

Cut a steel rod of length n into integer-sized pieces to maximize total revenue:



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

Problem Formalization

Solution space: *Partitions of n*: numbers $i_1, i_2, \dots, i_k$, so that $n = i_1 + i_2 + \cdots + i_k$

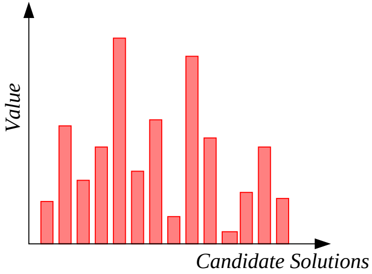
Value function: Partition $i_1, i_2, \dots, i_k$ has value $p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

Objective: Find optimal value and optimal partition:

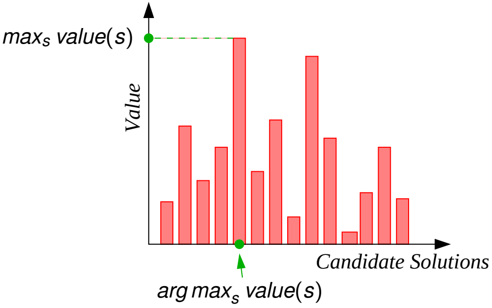
$$\begin{aligned} r_n &= max(p_{i_1} + p_{i_2} + \cdots + p_{i_k}) \\ s_n &= arg\,max(p_{i_1} + p_{i_2} + \cdots + p_{i_k}) \end{aligned}$$

where the maximimum is taken over all possible solutions $i_1, i_2, \dots, i_k$.

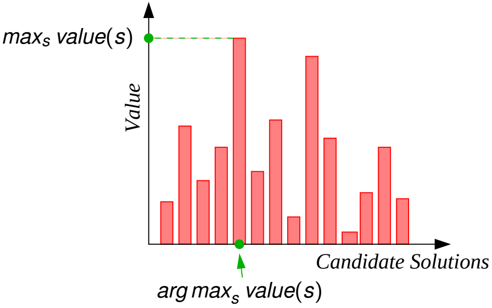
General optimization scenario:



General optimization scenario:



General optimization scenario:



- Instead of *maximizing* a *value* function, one may *minimize* a *cost* function
- Another example: Maximal subarray problem
  - Solution space: set of all subarrays of input array
  - Value function: sum of values in subarray

Brute-force approach for solving rod-cutting problem with $n = 4$: compute values of all possible solutions:

Solution	Value
1,1,1,1	4 = 1+1+1+1
1,1,2	7 = 1+ 1+ 5
1,3	9 = 1 + 8
2,2	10 = 5 + 5
4	9

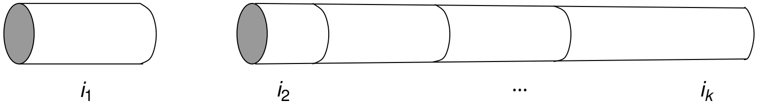
$\rightarrow$ best value: 10, best solution: 2,2.

☞ the number of candidate solutions is exponential in n

☞ brute force approach can only work for very small values of n

Optimal Substructure

Consider optimal solution $i_1, i_2, \ldots, i_k$ for problem size n:



Then: $i_2, \ldots, i_k$ is optimal solution for problem size $n - i_1$, and the optimal value for size n is $p_{i_1} + r_{n-i_1}$

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

☞ The optimal solution for a given problem instance incorporates optimal solution(s) of simpler/smaller subproblems.

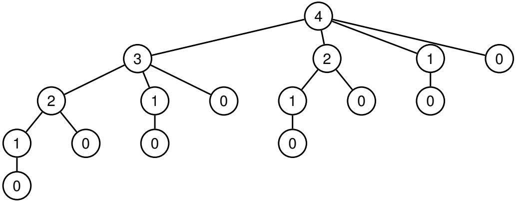
input : Positive integer n, Array $p[1 \mathrel{..} n]$ of positive integers
output: Value of optimal rod-cutting solution for problem size n and price table p

```
CutRod(p, n)
if n==0 then
    return 0
q = −∞
for i=1 to n do
    q = max(q, p[i] + CutRod(p, n − i))
return q
```

input : Positive integer n, Array $p[1 \,..\, n]$ of positive integers
output: Value of optimal rod-cutting solution for problem size n and price table p

```
CutRod(p, n)
if n==0 then
    return 0
q = −∞
for i=1 to n do
    q = max(q, p[i] + CutRod(p, n − i))
return q
```

Concrete recursion tree for $\textsc{CutRod}(p, 4)$:



Number of nodes in tree for $\textsc{CutRod}(p, n)$:

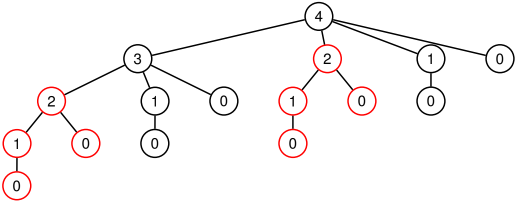
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

Solution:

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

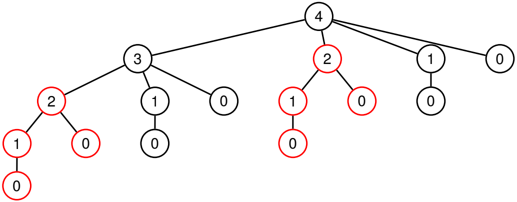
How fast (worst-case) can we solve the rod cutting problem using dynamic programming?

- A 2^n
- B n^3
- C n^2
- D $n \lg n$
- E n



Computations performed in subtrees with the same root are identical!

Idea: Instead of repeating identical computations, store the results and look them up!



Computations performed in subtrees with the same root are identical!

Idea: Instead of repeating identical computations, store the results and look them up!

Top-down Computation with Memoization

```
MemoizedCutRod(p, n)
let r[1 .. n] be a new array
for i=0 to n do
    r[i] = −∞
return MemoizedCutRodAux(p, n, r)
```

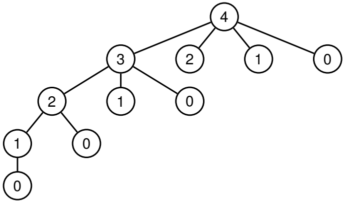
```
MemoizedCutRodAux(p, n, r)
if r[n] ≥ 0 then
    return r[n]
if n == 0 then
    q = 0
else
    q = −∞
    for i=1 to n do
        q = max(q,
            p[i] + MemoizedCutRodAux(p,n-i,r))
    r[n] = q
return q
```

Bottom-Up Version

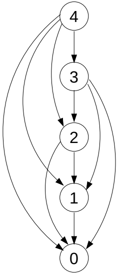
```
BottomUpCutRod(p, n)
let r[0 .. n] be a new array
r[0] = 0
for j=1 to n do
    q = −∞
    for i=1 to j do
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Complexity: $\Theta(n^2)$

Recursion tree for memoized cut rod:



Subproblem Graph



The **Subproblem Graph** for SOMEALGORITHM(x)

- contains one node for each distinct recursive function call occurring in the concrete recursion tree of SOMEALGORITHM(x)
- an edge between node z and node y if evaluating SOMEALGORITHM(z) leads to a call SOMEALGORITHM(y)

☞ the edges in the subproblem graph correspond to the edges in the recursion tree with memoization

☞ the complexity of top-down computation with memoization is $\Omega(e)$, where e is the number of edges in the subproblem graph.

MEMOIZEDCUTROD and BOTTOMUPCUTROD only compute the *value* of the optimal solution, not the solution itself!

```
EXTENDEDBOTTOMUPCUTROD(p, n)
let r[1 .. n] and s[1 .. n] be new arrays
r[0] = 0
for j=1 to n do
    q = −∞
    for i=1 to j do
        if q < p[i] + r[j − i] then
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r[n]
```

Result for $n = 10$:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Optimal Binary Search Tree

Before (Red-Black Trees etc.): goal is to construct BST with $\lg n$ *worst-case* complexity for all operations *insert*,*delete*, *search*.

Now: goal is to construct BST with best *average-case* complexity for repeated *search* operations.

Example

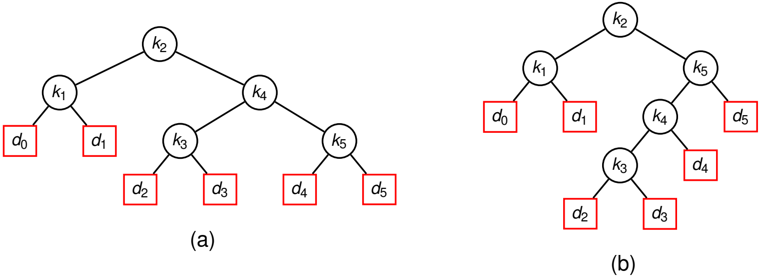
- Want to build search tree for keys $k_1 < k_2 < k_3 < k_4 < k_5$
- Have the following probabilities:
  - p_i: probability that search is for key k_i ($i = 1, \ldots, 5$)
  - q_i: probability that search is for key in between k_i and k_{i+1} ($i = 1, \ldots, 4$), a key less than k_1 ($i = 0$), or greater than k_5 ($i = 5$)

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

$$\sum_{i=1}^{5} p_i + \sum_{i=0}^{5} q_i = 1$$

☞ Want to minimize average search time for repeated searching according to these probabilities

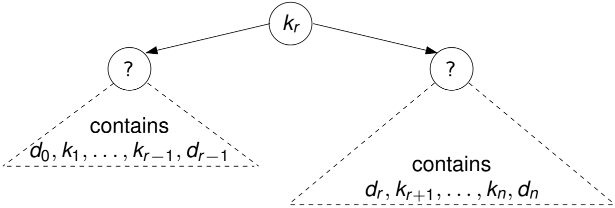
Two candidate trees. Unsuccessful searches represented by leaves with dummy keys $d_0, \ldots, d_5$:



Computing average search time (search cost for (dummy) key: 1+ depth of node):

Node	Probability	(a) Depth	(a) Contribution	(b) Depth	(b) Contribution
k_1	0.15	1	0.3	1	0.3
k_2	0.1	0	0.1	0	0.1
k_3	0.05	2	0.15	3	0.2
k_4	0.1	1	0.2	2	0.3
k_5	0.2	2	0.6	1	0.4
d_0	0.05	2	0.15	2	0.15
d_1	0.1	2	0.3	2	0.3
d_2	0.05	3	0.2	4	0.25
d_3	0.05	3	0.2	4	0.25
d_4	0.05	3	0.2	3	0.2
d_5	0.1	3	0.4	2	0.3
Total			2.8		2.75

BST for (dummy) keys $d_0, k_1, \ldots, d_{r-1}, k_r, \ldots, k_n, d_n$ with k_r the root:



☞ every subtree of a BST contains a contiguous range of (dummy) keys $d_{i-1}, k_i, d_i, \ldots, k_j, d_j$.

Defining $e[i, j]$

- For tree T containing $d_{i-1}, k_i, d_i, \ldots, k_j, d_j$, the expected search cost is

$$e_T = \sum_{h=i}^{j} p_i \cdot (depth_T(k_i) + 1) + \sum_{h=i-1}^{j} q_i \cdot (depth_T(d_i) + 1)$$

- $e[i, j]$: minimum value of e_T for all possible trees T containing $d_{i-1}, k_i, d_i, \ldots, k_j, d_j$.

Recursion for $e[i,j]$

Assuming the optimal tree for $d_{i-1}, k_i, d_i, \ldots, k_j, d_j$ has root k_r ($i \leq r \leq j$). Then:

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$$

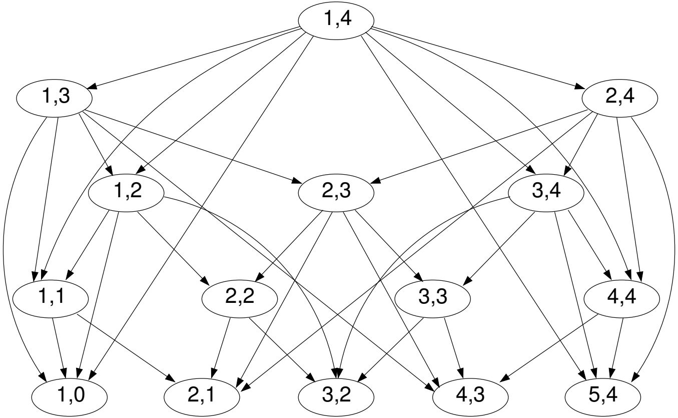
where

$$w(i,j) = \sum_{h=i}^{j} p_i + \sum_{h=i-1}^{j} q_i$$

Optimizing over possible choices of root:

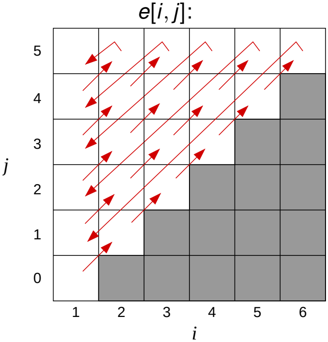
$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ min_{i \leq r \leq j}\, e[i,r-1] + e[r+1,j] + w(i,j) & \text{if } i \leq j \end{cases}$$

Subproblem graph for the computation of $e[1,4]$:



Number of edges: $\Theta(n^3)$

Reminder : $e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ min_{i \le r \le j}\, e[i,r-1] + e[r+1,j] + w(i,j) & \text{if } i \le j \end{cases}$



Grey area: not needed

Arrows indicate sequence in which array entries are computed

- Iteratively fill in matrix $e[1 \mathinner{..} n+1, 0 \mathinner{..} n]$, so that when $e[i,j]$ is computed all required $e[i,r-1]$ and $e[r+1,j]$ are already computed.
- In parallel, compute additional matrices:
  - $w[1 \mathinner{..} n+1, 0 \mathinner{..} n]$ for the $w(i,j)$ values
  - $root[1 \mathinner{..} n, 1 \mathinner{..} n]$ for the index of the optimal root
- Complexity: $\Theta(n^3)$.

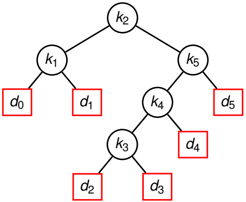
Computed values in $e[i, j]$ and $root[i, j]$ matrices:

$e[i, j]$

j \ i	1	2	3	4	5	6
5	2.75	2.00	1.30	0.90	0.50	0.10
4	1.75	1.20	0.60	0.30	0.05	
3	1.25	0.70	0.25	0.05		
2	0.90	0.40	0.05			
1	0.45	0.10				
0	0.05					

$root[i, j]$

j \ i	1	2	3	4	5	6
5	2	4	5	5	5	
4	2	2	4	4		
3	2	2	3			
2	1	2				
1	1					
0						



Green cells in $root[i, j]$: entries needed to retrieve structure of optimal BST.

Summary

Optimization Problems

The computational problem can be defined as optimization of a value function over a solution space.

Optimal Substructure

The optimal solution contains optimal solutions of (smaller) subproblems.

Often: optimal solution is determined by:

- one initial (optimal) choice
  - Rod cutting: choice of first length to cut off
  - Optimal BST: choice of key at the root
- optimal solutions of smaller subproblems
  - Rod cutting: optimal cutting of remaining length of rod
  - Optimal BST: optimal BST for keys left and right of root

Overlapping Subproblems

In *recursive solution* approach identical subproblems occur at several nodes of the concrete recursion tree.

- Rod cutting: optimal cutting of rod of length i
- Optimal BST: optimal BST for keys in range $k_i, \ldots, k_j$

Typical: subproblems are defined by 1,2,3, ... (a small number) integer arguments.

Subproblem graph: graph obtained from concrete recursion tree by merging nodes representing identical subproblems.

Top-Down Solution

Use recursive computation, save results of subproblems already solved (Memoization, Caching)

Bottom-Up Computation

Typical scenario: subproblems are defined by k integer arguments (e.g. $k = 2$). Then: iteratively fill up a k-dimensional array containing the solutions for all subproblems needed to solve the original problem.

Complexity

A lower bound is given by the number of edges in the subproblem graph:

- In top-down approach, an edge represents a recursive function call
- In bottom-up approach, an edge represents an access to an already computed subproblem solution

Typical: this lower bound is also an upper bound

Time-Space Tradeoff

Dynamic programming . . .

- saves time compared to recursive solutions
- needs extra space to store solutions of subproblems