Dynamic Programming

Lecture #4 of Algorithms & Data Structures (Module 7)

Rom Langerak

E-mail: r.langerak@utwente.nl

February 14, 2019

© JPK&GfH

Plan for today

Maximum subsequence problems

- separated subsequences
- (inefficient) recursive solutions
- dynamic programming: improvement by bookkeeping
- dynamic programming; further improvement by topological sort

Examples of a different kind, where the same approach works

- optimal matrix multiplication
- optimal binary search trees

What is dynamic programming?

[Bellman 1957]

Main original motivation:

- replace an exponential-time computation by a polynomial-time computation

Splitting problems into sub-problems may be very expensive

- if not controlled correctly, many subproblems will be solved repeatedly

Dynamic programming:

- basic idea: store results for subproblems rather than recomputing them
- applicable to problems where a recursive algorithm solves many subproblems repeatedly

© JPK&GfH

Maximal Subsequence Sum problems

The solution of a subsequence sum problem is a subsequence

$0 \leqslant i_0 < \cdots < i_{k-1} < n$ of indices in a given sequence of numbers $\langle a_0, \cdots, a_{n-1} \rangle$, together with the sum $s = \sum_{j=0}^{k-1} a_{i_j}$ of that subsequence.

The solution must satisfy constraints $R(\langle i_0, \cdots, i_{k-1} \rangle, s)$.

- E.g. the sum s must satisfy $s < M$, or the subsequence must be a contiguous segment, so $i_{j+1} = i_j + 1$, or ...

And also: the solution must be the maximal one.

- For every other subsequence of indices $0 \leqslant i'_0 < \cdots < i'_{m-1} < n$ with corresponding sum $s' = \sum_{j=0}^{m-1} a_{i'_j}$ satisfying $R(\langle i'_0, \cdots, i'_{m-1} \rangle, s')$ we find $s' \leqslant s$.

Maximal separated subsequence

We shall analyze msss, the maximum separated subsequence sum problem with the following constraint:

- two consecutive elements of the subsequence are always separated by at least one element of the sequence which is *not* in the subsequence.
- or to put it formally: $i_j + 1 < i_{j+1}$

Example: a sequence of 12 elements with three separated subsequences and their sum:

index	0	1	2	3	4	5	6	7	8	9	10	11	
A	3	6	2	3	7	5	9	7	4	7	1	8	
even	3		2		7		9		4		1		sum 26
odd		6		3		5		7		7		8	sum 36
best		6			7		9			7		8	sum 37

The algorithm for msss

Suppose we want to find the msss starting from index i.

Then we can choose to either have $A[i]$ in the subsequence, or not:

```
def msssRec(A,i):
    if i>=len(A):
        return 0
    else:
        return max(msssRec(A,i+1),A[i]+msssRec(A,i+2))
```

The subproblems for msss

To find the solution for *msss* one must solve subproblems.

To solve *msss* for $\langle a_i \rangle_{i=0}^{n-1}$, we need the solutions of *msss* for $\langle a_i \rangle_{i=1}^{n-1}$, and $\langle a_i \rangle_{i=2}^{n-1}$.

To solve *msss* for $\langle a_i \rangle_{i=0}^{n-1}$, we need the solutions of *msss* for $\langle a_i \rangle_{i=n-k}^{n-1}$, for all k with $0 \leqslant k \leqslant n$.

We use *msss*(k) to refer to the subproblem of solving *msss* for the last k elements of $\langle a_i \rangle_{i=0}^{n-1}$. *msss*$(n)$ stands for solving the problem for the entire sequence.

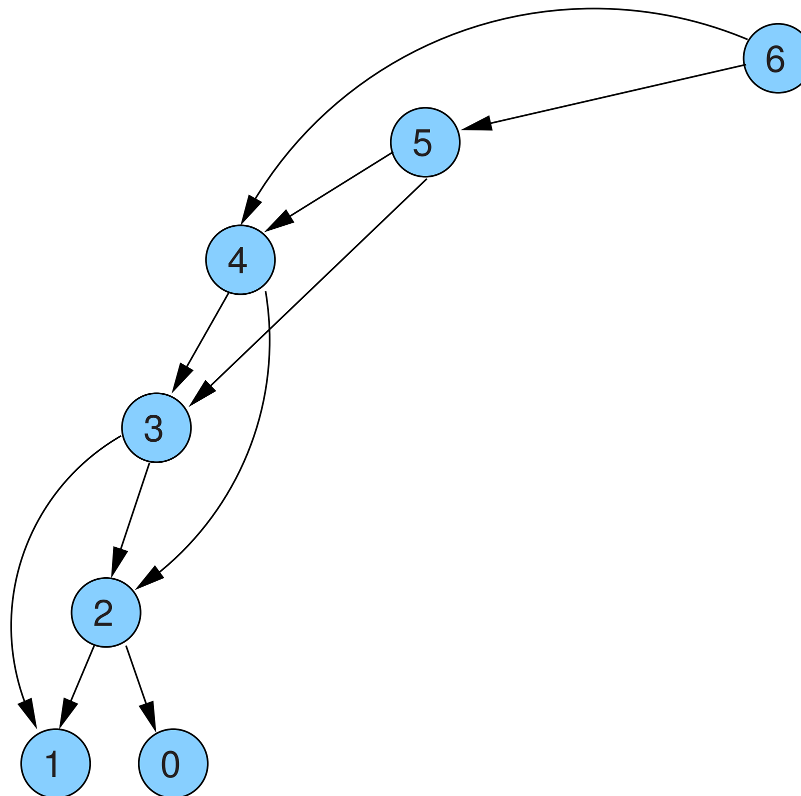
Recurrence equations

We can define $\mathit{msss}(k)$ by recurrence equations.

$$\begin{aligned} \mathit{msss}(k) &= 0, \text{ if } k \leqslant 0 \\ \mathit{msss}(k) &= \mathit{max}(a_{n-k}+\mathit{msss}(k-2), \mathit{msss}(k-1)) \text{ if } k \geqslant 1 \end{aligned}$$

Subproblem dependency graph

We can model the dependencies between subproblems by a dependency graph for instances of *msss*(k)



Time complexity

$$
\begin{aligned}
T_{msss}(k) &= 0, \text{ if } k \leqslant 0 \\
T_{msss}(k) &= 4 + T_{msss}(k-1) + T_{msss}(k-2) \text{ if } k \geqslant 1
\end{aligned}
$$

4: a max and three + operations

This behaviour is exponential, one can show by induction (cf Fibonacci)

$T_{msss}(n) \in \Omega((\sqrt{2})^n)$

The problem is: we are revisiting subproblems and repeat the construction of their solution. We should remember the solutions of subproblems we have computed.

Bookkeeping

```
def msssDP(A,k):
    n=len(A)
    color=[blue]*(n+1)
    solns=[0]*(n+1)

    if k<=0:
        return 0
    else:
        if color[k]!=green:
            solns[k]=max(msssDP(A,k-1),A[n-k]+msssDP(A,k-2))
            color[k]=green
        return solns[k]
```

Dynamic Programming

Dynamic programming in a nutshell

- Start with a recursive solution (recurrence equations) which shows the relevant subproblems and the subproblem dependency graph
- Introduce an object (array) to store the subproblem solution for every vertex in de subproblem dependency graph
- Introduce an an object (array of colors) to signal the status (solved or not) of a subproblem
- Copy the recursive solution, but
  - instead of computing solutions to subproblems, first inspect their status
  - if the status shows that the subproblem has been solved, retrieve the solution
  - otherwise compute the solution (recursively) and store it

Analysis of time complexity for DP

DP requires

- a constant number of computational steps in each vertex of the subproblem graph
- an inspection of a computation result for each edge of the subproblem graph

The complexity of a DP solution is like the complexity of a depth first search:

$$\Theta(|V| + |E|)$$

The DP solution of *msss* is linear, the subproblem graph for a sequence of n elements has approximately n vertices and $2n$ edges.

© JPK&GfH

Further improvement

```
def msssIter(A):
    n=len(A)
    solns=[0]*(n+1)
    solns[1]=A[n-1]

    for k in range(2,n+1):
        solns[k]=max(solns[k-1],A[n-k]+solns[k-2])
    return solns[n]
```

© JPK&GfH

Motivation behind *msssIter*

To make bookkeeping and traversal of the subproblem graph easier, one can apply a topological sort to the subproblem precedence graph. This is the transpose of the subproblem dependency graph.

In the *msss* example, ordering the subproblems by their size (the k of *msss*(k)) is an adequate topological ordering.

Solving subproblems in the order of the topological sort makes coloring redundant. If we start to solve a problem we are sure that all subproblems it depends on (which precede it) have been solved.

mssslter Example

index	0	1	2	3	4	5	6	7	8	9	10	11
A	3	6	2	3	7	5	9	7	4	7	1	8
soln reversed	37	37	33	31	31	27	24	22	15	15	8	8
size k	12	11	10	9	8	7	6	5	4	3	2	1

© JPK&GfH

Further improvement

```
subseq=[False]*n

def mssss(A):
    n=len(A)
    solkmin1=0
    solk=0
    for k in range(1,n+1):
        solkmin2=solkmin1
        solkmin1=solk
        solk=A[n-k]+solkmin2
        if solk>solkmin1:
            subseq[n-k]=True
        else:
            solk=solkmin1
    return solk
```

msss Example

Now there is sufficient information to extract the subsequence:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	3	6	2	3	7	5	9	7	4	7	1	8
solns												
reversed	37	37	33	31	31	27	24	22	15	15	8	8
subseq	0	1	1	0	1	1	1	1	0	1	0	1
		6			7		9			7		8

(the first 1 after the next item indicates the next element in the subsequence)

Matrix multiplication

Recall matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}{\cdot}b_{11} + a_{12}{\cdot}b_{21} & a_{11}{\cdot}b_{12} + a_{12}{\cdot}b_{22} & a_{11}{\cdot}b_{13} + a_{12}{\cdot}b_{23} \\ a_{21}{\cdot}b_{11} + a_{22}{\cdot}b_{21} & a_{21}{\cdot}b_{12} + a_{22}{\cdot}b_{22} & a_{21}{\cdot}b_{13} + a_{22}{\cdot}b_{23} \end{bmatrix}$$

Multiplying a $p \times q$-matrix and a $q \times r$-matrix requires $p{\cdot}q{\cdot}r$ element-wise multiplications

Matrix multiplication is associative: $(A{\times}B){\times}C = A{\times}(B{\times}C)$

© JPK&GfH

The matrix multiplication order problem

What is the *best order* to compute $A_1 \times A_2 \times \ldots \times A_n$ for $n > 2$?

- where matrix A_i has cardinality $d_{i-1} \times d_i$ for $i > 0$

Consider matrices $A_1 : 30 \times 1$, $A_2 : 1 \times 40$, $A_3 : 40 \times 10$, $A_4 : 10 \times 25$

Multiplication order	1st product		2nd product		3rd product		# Mults
$((A_1 A_2) A_3) A_4$	$30 \cdot 1 \cdot 40$	+	$30 \cdot 40 \cdot 10$	+	$30 \cdot 10 \cdot 25$	=	20,700
$A_1 (A_2 (A_3 A_4))$	$40 \cdot 10 \cdot 25$	+	$1 \cdot 40 \cdot 25$	+	$30 \cdot 1 \cdot 25$	=	11,750
$(A_1 A_2)(A_3 A_4)$	$30 \cdot 1 \cdot 40$	+	$40 \cdot 10 \cdot 25$	+	$30 \cdot 40 \cdot 25$	=	41,200
$(A_1 (A_2 A_3)) A_4$	$1 \cdot 40 \cdot 10$	+	$30 \cdot 1 \cdot 10$	+	$30 \cdot 10 \cdot 25$	=	8,200
$A_1 ((A_2 A_3) A_4)$	$1 \cdot 40 \cdot 10$	+	$1 \cdot 10 \cdot 25$	+	$30 \cdot 1 \cdot 25$	=	1,400

What is the minimal number of multiplications? What is the order?

Greedy strategy: each time choose cheapest multiplication. Does this work?

Try matrices $5 \times 3, 3 \times 10, 10 \times 1, 1 \times 2$

Problem decomposition

Choose the *last* multiplication to be at position i

The two remaining subproblems are:

- $A_1 \times \ldots \times A_i = B$ (of dimension $d_0 \times d_i$)
- $A_{i+1} \times \ldots \times A_n = C$ (of dimension $d_i \times d_n$)
- last step is to multiply B and C (taking $d_0 \times d_i \times d_n$ multiplications)

Identify original problem $(d_0, d_1) \times \ldots \times (d_{n-1}, d_n)$ as $(0, n)$

- after choice of i, the remaining subproblems are $(0, i)$ and (i, n)
- for subproblem (j, m) the choice of k results in subproblems (j, k) and (k, m)

This creates $\Theta(n^2)$ vertices in the subproblem graph

To determine the *minimal cost* we need to check *all* possibilities for i

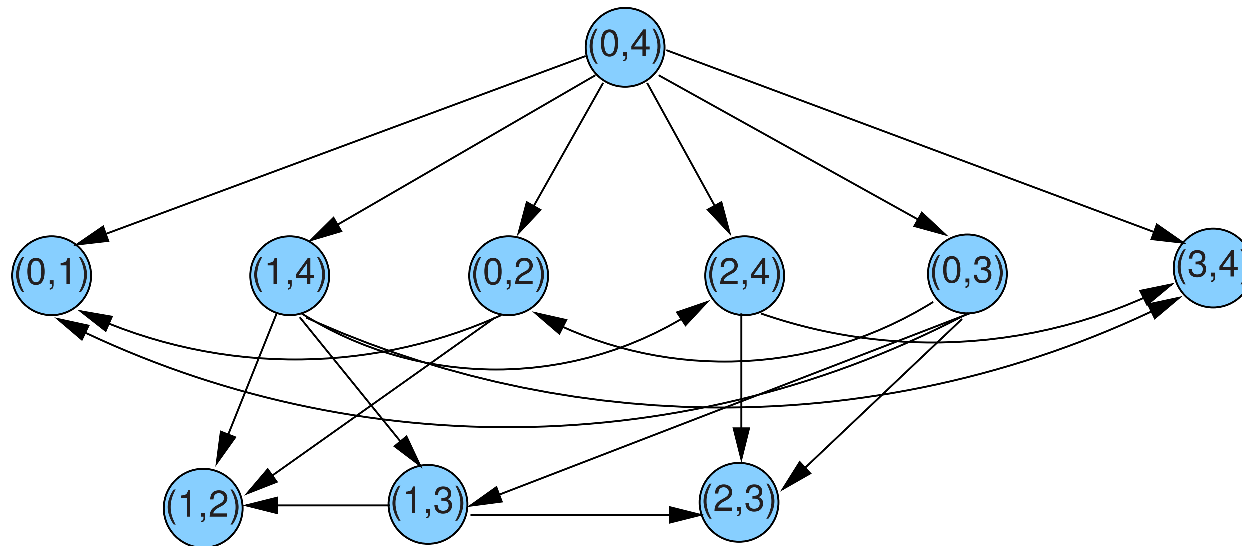
Recursive solution

```
def matMult2(d,l,h):                # solve subproblem (l,h)
                                    # d: array with dimensions
    if (h-l)==1:
        bestCost=0                  # one matrix, so no costs
    else:
        bestCost = infinity         # or a very large number
        for i in range(l+1,h):
            a=matMult2(d,l,i)       # solve subproblem (l,i)
            b=matMult2(d,i,h)       # solve subproblem (i,h)
            c=d[l]*d[i]*d[h]        # cost of multiplication
            bestCost=min(bestCost,a+b+c)
    return bestCost
```

It can be shown that $T_{matMult2}(n) \in \Omega(2^n)$

Subproblem graph – example

The subproblem graph of $A_1 \times \ldots \times A_4$, i.e., subproblem $(0, 4)$:



© JPK&GfH

Subproblem graph – analysis

Vertices are identified by (i, j) with $0 \leqslant i < j \leqslant n$

- this yields about $n^2/2$ vertices in total

For vertex (i, j) the following subproblems have to be solved:

- (i, k) and (k, j) for all k with $i < k < j$
- thus, the number of outgoing edges per vertex is less than $2n$

Subproblem graph has $n^2/2$ vertices and at most n^3 edges

- worst-case time-complexity of a DFS on the subproblem graph is $O(n^3)$

It is feasible to convert recursive problem into DP version

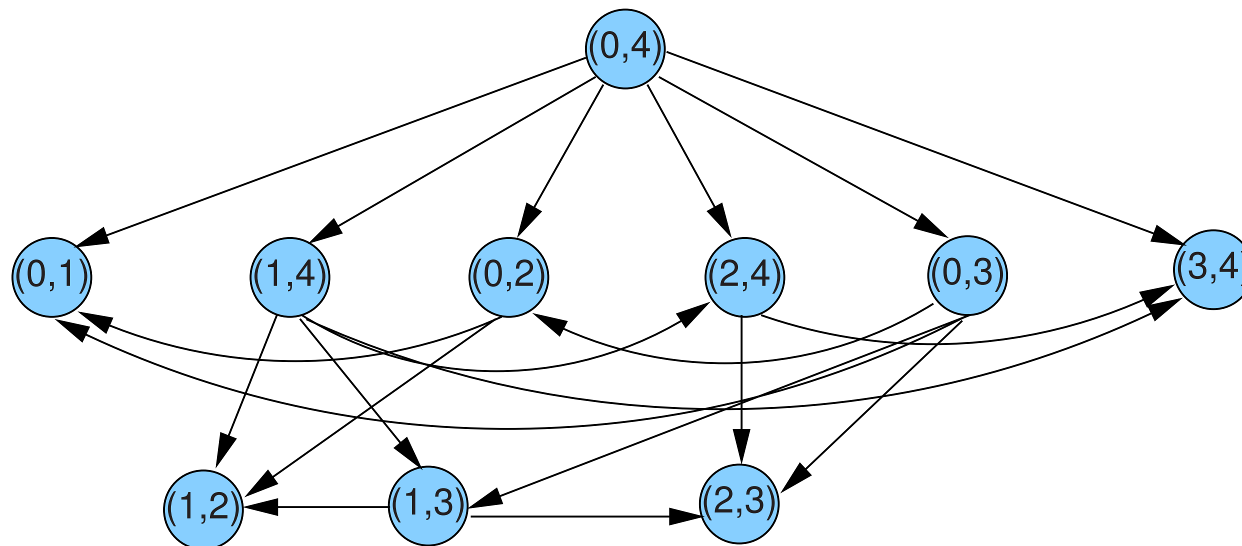
- as this boils down to a DFS on the subproblem graph
- this reduces the time-complexity from $\Omega(2^n)$ to $O(n^3)$

Dynamic programming version

```
color=[[blue for i in range(n+1)] for j in range(n+1)]
cost=[[0 for i in range(n+1)] for j in range(n+1)]
def matMult2(d,l,h):        # solve subproblem (l,h)
    if (h-l)==1:
        cost[l,h]=0                    # one matrix, so no costs
    else:
        if color[l,h] != green:
            bestCost = infinity        # or a very large number
            for i in range(l+1,h):
                a=matMult2(d,l,i)      # solve subproblem (l,i)
                b=matMult2(d,i,h)      # solve subproblem (i,h)
                c=d[l]*d[i]*d[h]       # cost of multiplication
                bestCost=min(bestCost,a+b+c)
            cost[l,h]=bestCost
            color[l,h]=green
    return cost[l,h]
```

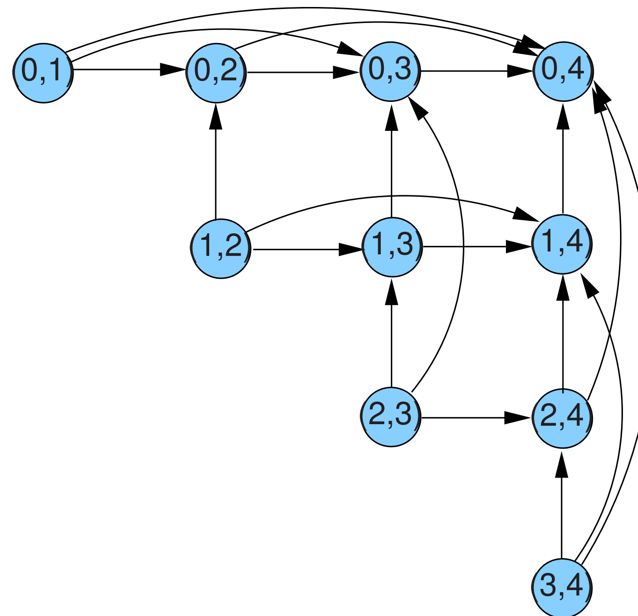
Recall the subproblem graph

The subproblem graph of $A_1 \times \ldots \times A_4$, i.e., subproblem $(0,4)$:



Reverse topological order

The dependency graph of $A_1 \times \ldots \times A_4$, i.e., subproblem $(0,4)$:



$$cost[l,h] = \begin{cases} 0 & \text{if } l{+}1 = h \\ min_{l<i<h}(cost[l,i] + cost[i,h] + d_l \cdot d_i \cdot d_h) & \text{if } l{+}1 < h \end{cases}$$

© JPK&GfH

Simplified DP matrix-multiplication order

```
def matMult2(d,n):
    cost=[[0 for i in range(n+1)] for j in range(n+1)]
    for l in range(n-1,-1,-1):
        for h in range(l+1,n+1):
            if (h-l)==1:
                cost[l,h]=0              # one matrix, so no costs
            else:
                bestCost = infinity      # or a very large number
                for i in range(l+1,h):
                    a=cost[l,i]          # subproblem (l,i)
                    b=cost[i,h]          # subproblem (i,h)
                    c=d[l]*d[i]*d[h]     # cost of multiplication
                    bestCost=min(bestCost,a+b+c)
                cost[l,h]=bestCost
    return cost[0,n]
```

© JPK&GfH

Example computation

$$\begin{bmatrix} - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix} \qquad \begin{bmatrix} - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & 0 & 10,000 \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix}$$

$$\begin{bmatrix} - & - & - & - & - \\ - & - & 0 & 400 & 650 \\ - & - & - & 0 & 10,000 \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix} \qquad \begin{bmatrix} - & 0 & 1200 & 700 & 1400 \\ - & - & 0 & 400 & 650 \\ - & - & - & 0 & 10,000 \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix}$$

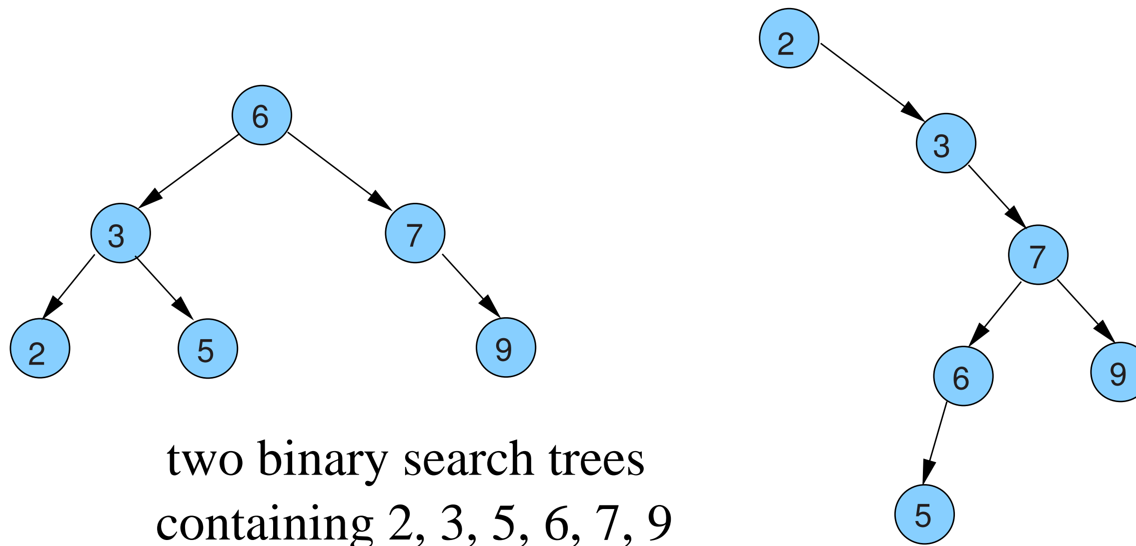
$$\mathit{cost}[l,h] = \begin{cases} 0 & \text{if } l+1 = h \\ \mathit{min}_{l<i<h}(\mathit{cost}[l,i] + \mathit{cost}[i,h] + d_l \cdot d_i \cdot d_h) & \text{if } l+1 < h \end{cases}$$

Binary search trees – revisited

In a binary search tree (BST) keys are stored such that:

- the key at a node is *at least* all keys in its *left* subtree
- the key at a node is *at most* all keys in its *right* subtree

An inorder traversal of a binary search tree yields a *sorted* list of keys



two binary search trees
containing 2, 3, 5, 6, 7, 9

Optimal binary search trees

Keys $K_1, \ldots, K_n$ with probability to be sought $p_1, \ldots, p_n$

Average #nodes examined in BST T with keys $K_1, \ldots, K_n$ is:

$$A(T) = \sum_{i=1}^{n} p_i{\cdot}c_i$$

- where c_i is the number of comparisons made to locate key K_i in BST T
- so c_i is $1+$ the level K_i in T
- and $p_i{\cdot}c_i$ is the weighted retrieve cost for key K_i

How to organise T such that $A(T)$ is minimal?

- how to organise the key values in a BST to minimise the average number of key comparisons?

Optimal BSTs, greedy approach

Let $T_{(l,h)}$ stand for a BST tree for keys $K_l, \ldots, K_h$.

Let $T^r_{(l,h)}$ denote a BST tree for keys $K_l, \ldots, K_h$ with K_r as root key ($l \leqslant r \leqslant h$)

What if we (greedily) choose as root key the K_r for which p_r is maximal?

This will *not* work.

- there is a trade-off between balance of the whole tree and short paths to keys with higher probability
- think of 3 keys $K_1 < K_2 < K_3$ with probabilities $p_1 = 0.20$, $p_2 = 0.25$ and $p_3 = 0.30$ (and a probability of 0.25 that we search for something that is not in the BST)
- $A(T^3_{(1,3)}) \geqslant 1.4$, and $A(T^2_{(1,3)}) = 1.25$. Being greedy is the wrong approach

We must consider *all* possible roots

Optimal BSTs

$$A(T^r_{(l,h)}) =$$
$$p_r + \sum_{i=l}^{r-1} p_i(1 + (depth\ K_i\ in\ T_{(l,r-1)})) + \sum_{j=r+1}^{h} p_j(1 + (depth K_j\ in\ T_{(r+1,h)}))$$
$$= p_r + p(l, r-1) + A(T_{(l,r-1)}) + p(r+1, h) + A(T_{(r+1,h)})$$

where $p(l, h) = \sum_{i=l}^{h} p(i)$ is the probability to search some key in $K_l, \ldots, K_h$

This can be simplified to:

$$A(T^r_{(l,h)}) = p(l, h) + A(T_{(l,r-1)}) + A(T_{(r+1,h)})$$

The problem is now to compute *min*$\{ A(T^r_{(l,h)}) \mid l \leqslant r \leqslant h \}$

Naively, a recursive algorithm yields an exponential time-complexity

DP solution to optimal BSTs

```
def optimalBST(f,n):
    cost=[[0 for i in range(n+2)] for j in range(n+2)]
    for l in range(n+1,0,-1):
        for h in range(l-1,n+1):
            if h<l:
                cost[l,h]=0
            else:
                bestCost = infinity       # or a very large number
                for r in range(l,h+1):
                    c=p(l,h)+cost[l,r-1]+cost[r+1,h]
                    bestCost=min(bestCost,c)
                cost[l,h]=bestCost
    return cost[1,n]
```

Dynamic programming recipe

1. Tackle the problem “top-down”; this yields a recurrence equation
2. Define an appropriate dictionary; transform the recurrence equation into a DP algorithm
3. Complexity of DP-algorithm is complexity DFS on subproblem graph
4. Choose an appropriate data structure for implementing the dictionary
5. If possible, analyse the subproblem graph and find a reverse topological order; simplify your DP-algorithm accordingly
6. Decide how to get the solution to the problem from the data in the dictionary

© JPK&GfH