Week 6, Lecture 2

Algorithm Analysis and Design

Pratyay Suvarnapathaki, 2020111016

The Chain Matrix Multiplication Problem

Given a set of matrices $A_1, \cdots, A_n$ of dimensions $d_0 \times d_1, \cdots, dn-1 \times d_n$, their product is to be found in the most efficient way possible.
Since, matrix multiplication in general is associative but not commutative, the problem can be rephrased to be:
'to find the optimum multiplicative paranthesization of the given matrices'.

An example

Firstly, we know that the multiplication of matrices of orders $m \times n$ and $n \times p$ costs mnp operations in total.
Taking the matrices:
$A_{50\times20}, B_{20\times1}, C_{1\times10}$, and $D_{10\times100}$, we have:

Method	Paranthesization	Cost
In order	$((AB)C)D$	51500
Greedy	$(A(BC))D$	60200
Optimal	$(AB)(CD)$	7000

As we can see, the paranthesization has a huge impact on the multiplication cost, and the greedy approach isn't even close to the optimal solution in terms of cost.

The Dynamic Programming Solution

From the above table, we can intuitively observe that a natural visualization of the problem would be in the form of a **binary tree** (which is indeed a DAG *if we assign an arbitrary direction to the edges*), wherein:

- The leaves are the different matrices

- Non-leaf nodes are intermediate products of matrix multiplication, and
- The root is the final product $ABCD$

Defining the Subproblems

Also, here the **optimum substructure property** is followed, since for a tree to represent the optimal paranthesization, *its subtrees need to represent their respective optimal sub-paranthesization.*
The subproblem corresponding to each subtree is of the form 'find the minimum cost for the multiplication $A_i \times \cdots \times A_j : 1 \leq i \leq j \leq n$'.
Let us term this subproblem as $C(i, j)$.
The complete problem to solve is hence, $C(1, n)$.

Base Cases and General Cases

Naturally, the base cases are the subproblems of the form $C(i, i) = 0$, as there is no multiplication occurring here.
Now, considering a general subtree representing $A_i \times \cdots \times A_j : ij$, we observe that the two main branches (with the root as the parent) represent $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$, such that $i \leq kj$
Hence, we obtain the following relation:
$C(i, j) = min_{i \leq kj}\{C(i, k) + C(k + 1, j) + m_{i-1} \times m_k \times m_j\}$

Algorithm

Using a two dimensional memoization table, we obtain the following algorithm:

```
# m has been assumed to be a list of matrix dimensions as specified in the first
paragraph of this document.
for i in range(1,n+1):
C[i][i] = 0
for s in range(1,n):
for i in range(1,n-s+1):
j = i + s
C[i][j] = min(C[i][k] + C[k+1][j] + m[i-1] * m[k] * m[j] : i  k < j)
return C[1][n]
```

The complexity of the above algorithm is naturally $O(n^3)$ as each entry in C takes $O(n)$ time to compute.

The Knapsack Problem

The problem statement is quite simple. Given the weight capacity W of a burglar's knapsack, and stealable items $I_1, \cdots, I_n$, each with its own weight

and value (w_i, v_i),
The **maximum net value** that can be stolen away by the burglar is to be found, keeping in mind the weight capacity of the knapsack, of course.

The Dynamic Programming Solution

Considering repetitions are allowed

If repetitions are allowed, i.e. an unlimited stock of each I_i is available for the burglar to steal from, then we can define the subproblems in the following way.
Define the subproblem $K(w)$ = maximum possible value for knapsack capacity $w(w \leq W)$. The final answer is hence $K(W)$.
Now, to obtain the relation, in the spirit of dynamic programming, we try to find smaller subproblems.
Say, the optimal solution of $K(w)$ contains item I_i. Now, removing I_i from the sacks, *what remains is an optimal solution to* $K(w - w_i)$.
Hence, $K(w) = K(w - w_i) + v_i$. Thus, we obtain our relation:
$K(w) = max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$

Algorithm:

```
# Consider arrays w and v containing the weights and values of the different items
K[0]=0 #base case
for weight in range(1,W+1):
K[weight] = max(K (weight  w[i]) + v[i] : w[i] <= weight)
return K[W]
```

Considering repetitions are not allowed

If repetitions are allowe, i.e. only one of each I_i is available for the burglar to steal from, then we can define the subproblems in the following way.
Define the subproblem $K(w, j)$ = maximum possible value for knapsack capacity $w(w \leq W)$ and using items $1, \cdots, j(0 \leq j \leq n)$. The final answer is hence $K(W, n)$.
Now, to obtain the relation, in the spirit of dynamic programming, we try to find smaller subproblems.
Now quite simply, either the item under consideration, i.e. I_j, is in the optimal solution or isn't. Hence, the relation is straightforward:
$K(w) = max\{K(ww_j, j1) + v_j, K(w, j - 1)\}$

Algorithm:
We now use a two dimensional memoization table.

```
# base cases
K[0][*] = 0
```

```
K[*][0] = 0

for j in range(1,n+1):
for weight in range(1,W+1):
if w[j] > weight:
K[weight][j] = K [weight][j1]
else:
K[w][j]= max(K[weight][j1], K[weightw[j][j1] + v[j])
return K[W][n]
```

The time complexity in both cases is $O(nW)$ (in the second case too, filling each table entry takes constant time).
