CS 218 Design and Analysis of Algorithms

Nutan Limaye

Indian Institute of Technology, Bombay
nutan@cse.iitb.ac.in

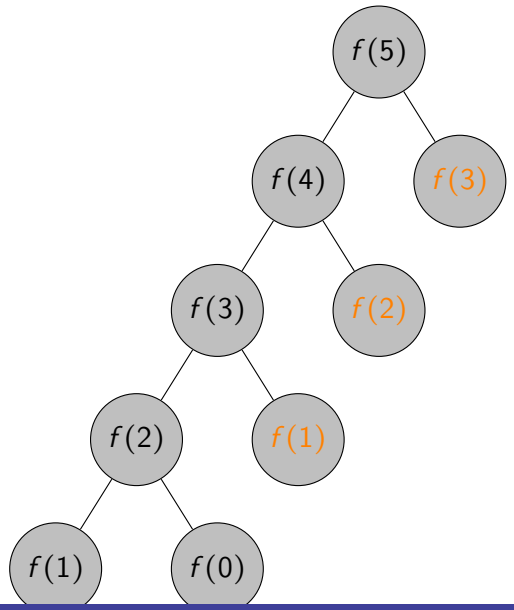
Module 1: Basics of algorithms

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

```
1:  if Table contains f(k) then
2:      Return the table value.
3:  end if
4:  if k = 0 or k = 1 then
5:      val ← k
6:  else
7:      val ← Fib(k − 1) + Fib(k − 2)
8:      Store val as the kth entry in the table
9:      return val
10: end if
```

k	$f(k)$
0	0
1	1
2	1
3	2
4	3
5	5



Running time analysis

```
1: if Table contains f(k) then
2:     Return the table value.
3: end if
4: if k = 0 or k = 1 then
5:     val ← k
6: else
7:     val ← Fib(k − 1) + Fib(k − 2)
8:     Store val as the kth entry in the
       table
9:     return val
10: end if
```

The table gets filled from smaller values to larger values.

The recursion for each index i is evaluated at most once.

Look up into the table takes $O(1)$ time.

Overall running time $O(k)$.

Weighted Interval Scheduling

Problem Description

Input: For each $i \in [n]$, start and finish times $(s(i), f(i))$ and a weight for each interval $w(i) \in \mathbb{N}$

Output: Maximum weight non-conflicting jobs

Seen this problem in Problem Sheet 1.

Does not have good greedy heuristic.

What about a recursion-like algorithm for it?

Can we design that?

Weighted Interval Scheduling

Problem Description

Input: For each $i \in [n]$, start and finish times $(s(i), f(i))$ and a weight for each interval $w(i) \in \mathbb{N}$

Output: Maximum weight non-conflicting jobs

A possible recursive strategy

Let us sort the jobs based on their finish time. $f(1) \leq f(2) \ldots \leq f(n)$.

Let $p(i)$ be j if j is the last job (in the above ordering) that is non-conflicting with i.

If the nth job belongs to OPT

for all $i > p(n)$, ith job does not belong to OPT.
Recurse on $\{1, \ldots, p(n)\}$ jobs.

If the nth job does not belong to OPT

recurse on $\{1, \ldots, n-1\}$ jobs.

Weighted Interval Scheduling

Problem Description

Input: For each $i \in [n]$, start and finish times $(s(i), f(i))$ and a weight for each interval $w(i) \in \mathbb{N}$

Output: Maximum weight non-conflicting jobs

Assume that we have sorted the jobs in non-decreasing order of their finish times.

We have also computed $p(i)$ for each $i \in [n]$.

`WtIntSc(`i`)`

if $i = 0$ **then**
  return i
else if Table(i) is non-empty **then**
  return Table(i)
else
  Table(i) $\leftarrow \max\{w(i) +$ `WtIntSc(p(i)), WtIntSc(i-1)`$\}$
end if

Correctness of the recursive algorithm

Lemma

For every $i \in [n]$, `WtIntSc(i)` computes the the optimal solution for the sub-problem $\{1, \ldots i\}$, where the jobs are ordered according to their finish times.

Let $\texttt{Opt}(i)$ denote the value of the optimal solution for the sub-problem $\{0, \ldots, i\}$.

Trivially $\texttt{Opt}(0) = 0$. Assume that the induction hypothesis holds for $\forall j < i$.

Proof

Due to this, we know that $\texttt{WtIntSc}(p(i)) = \texttt{Opt}(p(i))$ and $\texttt{WtIntSc}(i-1) = \texttt{Opt}(i-1)$.

From our previous reasoning, the correct answer for the subproblem $\{1, \ldots, i\}$ is max of $w(i) + \texttt{Opt}(p(i))$ and $\texttt{Opt}(i-1)$.

Therefore the algorithm computes it correctly.

Running time analysis

Running time analysis of the dynamic programming algorithm.

The sorting of the jobs according to their finish time takes time $O(n \log n)$.

Assuming writing the values into the table (array) and reading from it takes time $O(1)$the time taken by the algorithm is
$O(\text{the number of calls made to } \texttt{WtIntSc}) = O(n)$.

So the overall time taken is $O(n \log n)$.

Steps for Dynamic Programming

Dynamic programming approach has a template.

Step 1 Figure out the types of sub-problems.

Step 2 Define a recursive procedure.

Step 3 Decide on the memoization strategy.

Step 4 Check that the sub-problem dependencies are acyclic.

Step 5 Analyse the time complexity using the recursion.

Fibonacci and Weighted Interval Scheduling

Fibonacci computation

Step 1 Figure out the types of sub-problems.

Step 2 Define a recursive procedure.

Step 3 Decide on the memoization strategy.

Step 4 Check that the sub-problem dependencies are acyclic.

Step 5 Analyse the time complexity using the recursion.

`Fib(k)`

We got the sub-problems from the recursive definition.

Memoization was done to avoid repeated computation of sub-problems.

If $i < j$ then `Fib`(i) never called `Fib`(j).

Time = Time/sub-problem $\times$ # sub-problem

$O(n)$.

Fibonacci and Weighted Interval Scheduling

Fibonacci computation

Step 1 Figure out the types of sub-problems.	`WtIntSc(n)` Some clever reasoning involved in figuring out the sub-problems.
Step 2 Define a recursive procedure.	From the sub-problems the recurrence was easy.
Step 3 Decide on the memoization strategy.	Maintain the table for the opt values of the sub-problems.
Step 4 Check that the sub-problem dependencies are acyclic.	If $i < j$ then the subroutine for the sub-problem $\{1, \ldots, i\}$ does not make calls to the sub-problem $\{1, \ldots, j\}$.
Step 5 Analyse the time complexity using the recursion.	Overall time analyses as in the case of Fibonacci numbers.

Problems on strings/sequences

Many interesting problems on strings/sequences.

- Parenthesization Figure out the paranthesization of an expression that minimises the overall cost of evaluating the expression.
- Longest increasing subsequence Given a string of positive numbers, find the longest subsequence that is increasing.
- Segmentation Given a long string of letters, find a way (if one exists) of segmenting the string into chunks such that the segmented string is a statement in English.
- Longest common subsequence Given two string of letters, find the longest subsequence that is common to both.
- Edit distance Given two strings x, y, find the smallest number of updates need to convert x into y.

Problems on strings/sequences

Many interest problems on strings/sequences and applying the DP approach

Step 1 Figure out the types of sub-problems.
- ★ Usually suff$[1, i]$, pre$[i, n]$ or substring$[i, j]$.

Step 2 Define a recursive procedure.

Usually the clever part of developing a DP.

Depends heavily on the problem.

Step 3 Decide on the memoization strategy.

Store the values of the sub-problems.

Step 4 Check that the sub-problem dependencies are acyclic.

If Step 2 is defined correctly, this is again not hard to argue.

Step 5 Analyse the time complexity using the recursion.

Use the appropriate methods to analyse recursion.

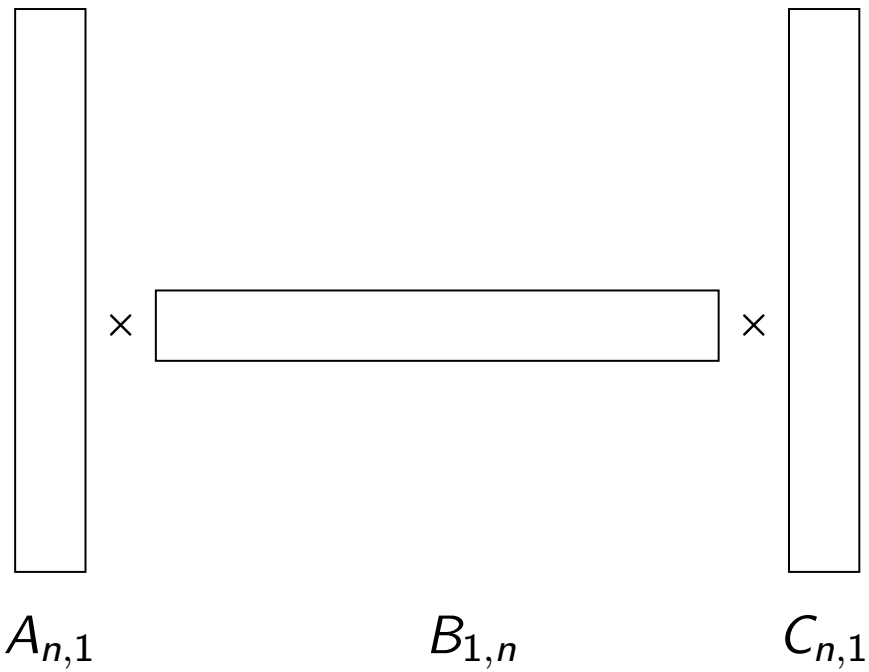
Parenthesization problem

Problem description

Input: Given a string of matrices

Output: Find the minimum cost parenthesization to multiply the matrices

Example



$(A \times B) \times C$. Costs $O(n^2)$ operations.

$A \times (B \times C)$. Costs $O(n)$ operations.

Find the best parenthesization.

Parenthesization problem

Problem description

Input: Given a string of matrices $A_1, A_2, \ldots, A_n$
(along with their dimensions) $c_i \times r_i$ for each $i \in [n]$.

Output: Find the minimum cost parenthesization to multiply them.

Step 1 Figure out the types of sub-problems.

Suppose we guess that the last operation of the best parenthesization.
$(A_1 \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \ldots \times A_n)$
This seems like prefixes and suffixes.
The next time, it may split each such sub-problem further.
$(A_1 \times \ldots \times A_{k'}) \times (A_{k'+1} \times A_{k'+2} \ldots \times A_k)$
$\times (A_{k+1} \times \ldots \times A_{k''}) \times (A_{k''+1} \times A_{k''+2} \ldots \times A_n)$
These are essentially sub-strings.
Therefore, Sub-problems are intervals.
We will denote $A_i \times A_{i+1} \times \ldots \times A_j$ by $A_{[i,j]}$

Parenthesization problem

Problem description

Input: Given a string of matrices $A_1, A_2, \ldots, A_n$
(along with their dimensions) $c_i \times r_i$ for each $i \in [n]$.

Output: Find the minimum cost parenthesization to multiply them.

Step 2 Come up with a recursive procedure

$\texttt{Para}(i,j)$ recursive procedure to compute the minimum cost parenthesization of the sub-string of matrices defined by the interval $[i,j]$.

We will assume that $i < j$. That is $A_i \times A_{i+1} \times \ldots \times A_j$.

$$\texttt{Para}(i,j) = \min_{i \le k < j} \left\{ \texttt{Para}(i,k) + \texttt{Para}(k+1,j) + \textbf{Cost}(i,k,j) \right\},$$

where $\textbf{Cost}(i,k,j)$ is the cost of multiplying $A_{[i,k]}$ and $A_{[k+1,j]}$.

Note the cost can be computed for

Parenthesization problem

Problem description

Input: Given a string of matrices $A_1, A_2, \ldots, A_n$
(along with their dimensions) $c_i \times r_i$ for each $i \in [n]$.

Output: Find the minimum cost parenthesization to multiply them.

Step 3 Decide memoization strategy

For any $i, k, j \in [n]$ such that $i \leq k < j$, **Cost**(i, j, k) can be computed in time $O(1)$.
So this can be computed on the fly.

We store the recursively computed values of $\texttt{Para}(i, j)$.

Starting from smaller intervals and building larger intervals.

Parenthesization problem

Problem description

Input: Given a string of matrices $A_1, A_2, \ldots, A_n$
(along with their dimensions) $c_i \times r_i$ for each $i \in [n]$.

Output: Find the minimum cost parenthesization to multiply them.

Step 4 Sub-problem dependencies are acyclic

The `Para` routine uses the values stored for the smaller intervals to compute the values of the larger intervals.

Therefore easy to see that the dependencies are acyclic.

Parenthesization problem

Problem description

Input: Given a string of matrices $A_1, A_2, \ldots, A_n$
(along with their dimensions) $c_i \times r_i$ for each $i \in [n]$.

Output: Find the minimum cost parenthesization to multiply them.
the matrices

Step 5 Analysing the time complexity of the problem

The total number of sub-problems is $O(n^2)$ one for each interval $[i, j]$.

The time per sub-problem is the time needed to compute the min over a set of values.
For an interval $[i, j]$ there are $O(j - i)$ values.
In the worst case, time per sub-problem is $O(n)$.

Total time $= O(n) \times O(n^2) = O(n^3)$.

Shortest path

Problem Description

Input: Given a directed graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$ and designated vertices $s, t \in V$.

Output: the length of the shortest path from s to t.

Points to note.

Similar task at hand as was when designing Dijkstra's algorithm.

Weights can be negative here.

What if the graph has cycles?

Shortest path

Problem Description

Input: Given a directed acyclic graph $G = (V, E)$, a weight function $w : E \to \mathbb{Z}$ and designated vertices $s, t \in V$.

Output: the length of the shortest path from s to t.

Can Dijkstra's algorithm work here?

It does not work. (See Fig. 6.21 (a) in Klienberg and Tardos.)

Shortest path

Problem Description

Input: Given a directed acyclic graph $G = (V, E)$, a weight function $w : E \to \mathbb{Z}$ and designated vertices $s, t \in V$.

Output: the length of the shortest path from s to t.

If we boost the weights to make them positive, will Dijkstra work?

It does not work. (See Fig. 6.21 (b) in Klienberg and Tardos.)