Algorithms M2 IF
Dynamic Programming

Michael Lampis

Fall 2019

Dynamic Programming vs Divide and Conquer

Dynamic Programming

- DP is a general algorithmic technique for solving optimization problems.
- Key idea: finding the optimal solution to the input instance can be reduced to finding the optimal solution to some smaller instance(s).
- This can then be done with the same algorithm, until we arrive at trivial instances of constant size.

Dynamic Programming vs Divide and Conquer

Dynamic Programming

- DP is a general algorithmic technique for solving optimization problems.
- Key idea: finding the optimal solution to the input instance can be reduced to finding the optimal solution to some smaller instance(s).
- This can then be done with the same algorithm, until we arrive at trivial instances of constant size.

So what is the difference with Divide&Conquer?

An example: Fibonacci

Recall the Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

- $F(n) = F(n-1) + F(n-2)$

Recursive implementation:

```
int fibo(int n){
        if(n<=2) return 1;
        return fibo(n-1)+fibo(n-2); }
```

Implementation with loop:

```
int fibo(int n){
        int a=1, b=1, c;
        while(n--){
                c=a+b;
                b=a;
                a=c;
        }
        return a; }
```

Fibonacci continued

Let's compare the complexities of the two algorithms:

- Second algorithm runs in $O(n)$. (easy to see)
- First algorithm has complexity $T(n) \leq T(n-1) + T(n-2)$

Fibonacci continued

Let's compare the complexities of the two algorithms:

- Second algorithm runs in $O(n)$. (easy to see)
- First algorithm has complexity $T(n) \leq T(n-1) + T(n-2)$
- Let's be generous: say $T(n) \leq 2T(n-2)$
  - $\Rightarrow T(n) = \Omega(1.4^n)$
  - (Correct ratio is $\approx 1.618^n \approx F(n)$)
- Linear vs Exponential!
- What went wrong?

Fibonacci continued

Let's compare the complexities of the two algorithms:

- Second algorithm runs in $O(n)$. (easy to see)
- First algorithm has complexity $T(n) \leq T(n-1) + T(n-2)$
- Let's be generous: say $T(n) \leq 2T(n-2)$
  - $\Rightarrow T(n) = \Omega(1.4^n)$
  - (Correct ratio is $\approx 1.618^n \approx F(n)$)
- Linear vs Exponential!
- What went wrong?
- The recursive algorithm solves the same sub-instances many times.
- **Key idea** of Dynamic Programming (difference with D&C)

  Build solution bottom-up, store solutions to smaller sub-problems so that they don't need to be recomputed.

Longest Increasing Subsequence

Longest Increasing Subsequence

- Input: an array A of n integers.
- Output: a subsequence (not necessarily consecutive) of A that is increasing and has maximum length.

Longest Increasing Subsequence

- Input: an array A of n integers.
- Output: a subsequence (not necessarily consecutive) of A that is increasing and has maximum length.

Example:

$$A = [2, 5, 3, 9, 1, 4, 7, 6]$$

- $2, 5, 9$ is a valid solution
- $2, 3, 9, 7$ is not (not increasing)
- $1, 2, 3$ is not (not a subsequence)

Longest Increasing Subsequence

- Input: an array A of n integers.
- Output: a subsequence (not necessarily consecutive) of A that is increasing and has maximum length.

Example:

$$A = [2, 5, 3, 9, 1, 4, 7, 6]$$

- $2, 5, 9$ is a valid solution
- $2, 3, 9, 7$ is not (not increasing)
- $1, 2, 3$ is not (not a subsequence)
- $2, 3, 4, 6$ is an optimal solution

Longest Increasing Subsequence

- Input: an array A of n integers.
- Output: a subsequence (not necessarily consecutive) of A that is increasing and has maximum length.

Example:

$$A = [2, 5, 3, 9, 1, 4, 7, 6]$$

- $2, 5, 9$ is a valid solution
- $2, 3, 9, 7$ is not (not increasing)
- $1, 2, 3$ is not (not a subsequence)
- $2, 3, 4, 6$ is an optimal solution

Objective: a polynomial-time (in n) algorithm that computes the length of the LIS.

Note: computing the length of the optimal solution is probably good enough. . .

Longest Increasing Subsequence

- Define $L(i)$: length of LIS of $A[1 \ldots i]$ which contains $A[i]$.
  - $L(0) = 0, L(1) = 1$ (base case)
  - $L(n) = \mathrm{OPT}$ (what we want to know)

Longest Increasing Subsequence

- Define $L(i)$: length of LIS of $A[1 \dots i]$ which contains $A[i]$.
    - $L(0) = 0, L(1) = 1$ (base case)
    - $L(n) = \mathrm{OPT}$ (what we want to know)
    - $L(i) = 1 + L(j)$, where j is the position of the second from the end element of the LIS.
        - $j < i$
        - $A[j] < A[i]$

Longest Increasing Subsequence

- Define $L(i)$: length of LIS of $A[1 \dots i]$ which contains $A[i]$.
  - $L(0) = 0, L(1) = 1$ (base case)
  - $L(n) = \mathrm{OPT}$ (what we want to know)
  - $L(i) = 1 + L(j)$, where j is the position of the second from the end element of the LIS.
    - $j < i$
    - $A[j] < A[i]$
  - Therefore $L(i) = \max_{j<i \wedge A[j]<A[i]} L(j) + 1$

Longest Increasing Subsequence

- Define $L(i)$: length of LIS of $A[1\ldots i]$ which contains $A[i]$.
  - $L(0) = 0, L(1) = 1$ (base case)
  - $L(n) = \mathrm{OPT}$ (what we want to know)
  - $L(i) = 1 + L(j)$, where j is the position of the second from the end element of the LIS.
    - $j < i$
    - $A[j] < A[i]$
  - Therefore $L(i) = \max_{j<i \wedge A[j]<A[i]} L(j) + 1$

$$
\begin{array}{rcl}
A & = & [2,5,3,9,1,4,7,6] \\
L(i) & = & [1,2,2,3,1,3,4,4]
\end{array}
$$

Correctness and DP implementation

- Similar to Divide&Conquer:
  - Finding recursive formula for L leads to an algorithm
  - Also to a correctness proof by induction:
    - Suppose that $L(j)$ is correctly computed
    - $\rightarrow$ then $L(i)$ is correctly computed because we consider all feasible j's (subsequence must increase) and we pick the best (exchange argument).

Correctness and DP implementation

- Similar to Divide&Conquer:
  - Finding recursive formula for L leads to an algorithm
  - Also to a correctness proof by induction:
    - Suppose that $L(j)$ is correctly computed
    - $\to$ then $L(i)$ is correctly computed because we consider all feasible j's (subsequence must increase) and we pick the best (exchange argument).
- DP: we **do not** implement this with recursion!
  - Would take exponential time for $L(n)$!!
- We construct a table $L(i)$ bottom-up (starting from smaller values)
- Running time $O(n^2)$
  - $O(n)$ to find max, repeated n times

Correctness and DP implementation

- Similar to Divide&Conquer:
  - Finding recursive formula for L leads to an algorithm
  - Also to a correctness proof by induction:
    - Suppose that $L(j)$ is correctly computed
    - $\rightarrow$ then $L(i)$ is correctly computed because we consider all feasible j's (subsequence must increase) and we pick the best (exchange argument).
- DP: we **do not** implement this with recursion!
  - Would take exponential time for $L(n)$!!
- We construct a table $L(i)$ bottom-up (starting from smaller values)
- Running time $O(n^2)$
  - $O(n)$ to find max, repeated n times
- From DP table we can also deduce the actual LIS.
  - Can use secondary table $L'(i)$ which stores that indices j used to maximize $L(i)$

Subset Sum

Story:

- Your friend gave you a 100$ gift card for Christmas. You can use it in an online store.
- The card cannot be used in combination with other payment methods.
- The items in the store have the following values:

$$[14, 17, 19, 23, 28, 31, 45, 47]$$

- You want to select a set of items that
  - Has maximum total value.
  - Has total cost at most $100\$$.

Knapsack

Story:

- Your friend gave you a 100$ gift card for Christmas. You can use it in an online store.
- The card cannot be used in combination with other payment methods.
- The items in the store have the following values:

$$[14, 17, 19, 23, 28, 31, 45, 47]$$

- You want to select a set of items that
  - Has maximum total value.
  - Has total cost at most $100\$$.

  Example:

  - $45 + 47 = 92$ (Greedy algorithm, buy most expensive feasible item)
  - $19 + 31 + 47 = 97$
  - $23 + 28 + 47 = 98$

Knapsack

- Input: array of values A, budget B.
- Output: subset of values with sum $\leq B$ such that sum is maximized.

Knapsack

- Input: array of values A, budget B.
- Output: subset of values with sum $\leq B$ such that sum is maximized.

- Break down the problem into sub-problems.
- Let $P(i, W)$ be the maximum value I can achieve if items $A[1, \ldots, i]$ are available and my budget is W.
  - I want to know $P(n, B)$
  - $P(i, 0)$ is easy, $P(0, W)$ is easy.

Knapsack

- Input: array of values A, budget B.
- Output: subset of values with sum $\leq B$ such that sum is maximized.

- Break down the problem into sub-problems.
- Let $P(i, W)$ be the maximum value I can achieve if items $A[1, \ldots, i]$ are available and my budget is W.

  - I want to know $P(n, B)$
  - $P(i, 0)$ is easy, $P(0, W)$ is easy.

$$P(n, W) = \max\{P(n-1, W), (P(n-1, W - A[n]) + A[n])\}$$

Knapsack

- Input: array of values A, budget B.
- Output: subset of values with sum $\leq B$ such that sum is maximized.

- Break down the problem into sub-problems.
- Let $P(i, W)$ be the maximum value I can achieve if items $A[1, \ldots, i]$ are available and my budget is W.
  - I want to know $P(n, B)$
  - $P(i, 0)$ is easy, $P(0, W)$ is easy.

$$P(n, W) = \max\{P(n-1, W), (P(n-1, W - A[n]) + A[n])\}$$

Explanation:

- I can either
  - Ignore last element
  - Or take it, gain $A[n]$ in profit, but decrease budget accordingly.
  - (Note: clearly, if $A[n] > W$ only first choice is feasible)

Knapsack DP

- Implementation: construct an $n \times B$ matrix to represent $P(i, W)$.
- Use formula of previous slide to fill each row after the previous row has been filled.
- Complexity: $O(nB)$. Polynomial?
  - Not quite! Since B is written in binary, it could be a huge number! We call this type of complexity **pseudo-polynomial**: polynomial if all values are small.

Knapsack DP

- Implementation: construct an $n \times B$ matrix to represent $P(i, W)$.
- Use formula of previous slide to fill each row after the previous row has been filled.
- Complexity: $O(nB)$. Polynomial?
  - Not quite! Since B is written in binary, it could be a huge number! We call this type of complexity **pseudo-polynomial**: polynomial if all values are small.

Example:

$$A = [3, 4, 5, 6], B = 12$$

Item	Budget – Profit											
(3)	0	0	0	3	3	3	3	3	3	3	3	3
(4)	0	0	0	3	4	4	4	7	7	7	7	7
(5)	0	0	0	3	4	5	5	7	8	9	9	12
(6)	0	0	0	3	4	5	6	7	8	9	11	12

Matrix Chain Multiplication

Matrix Multiplication (again)

- Input: We are given n matrices $A_1, A_2, \ldots, A_n$ with dimensions $r_0 \times r_1, r_1 \times r_2, \ldots, r_{n-1} \times r_n$
- Output: Optimal way to compute $A_1 \times A_2 \times \ldots \times A_n$.

Matrix Multiplication (again)

- Input: We are given n matrices $A_1, A_2, \ldots, A_n$ with dimensions $r_0 \times r_1, r_1 \times r_2, \ldots, r_{n-1} \times r_n$
- Output: Optimal way to compute $A_1 \times A_2 \times \ldots \times A_n$.
- Important: this is a meta-problem. We want to **plan** how to perform the multiplication.
- Assumption: multiplying an $a \times b$ matrix with a $b \times c$ matrix takes time $O(abc)$.
- Reminder: Multiplication is associative $ABC = (AB)C = A(BC)$.

Matrix Multiplication (again)

- Input: We are given n matrices $A_1, A_2, \ldots, A_n$ with dimensions $r_0 \times r_1, r_1 \times r_2, \ldots, r_{n-1} \times r_n$
- Output: Optimal way to compute $A_1 \times A_2 \times \ldots \times A_n$.
- Important: this is a meta-problem. We want to **plan** how to perform the multiplication.
- Assumption: multiplying an $a \times b$ matrix with a $b \times c$ matrix takes time $O(abc)$.
- Reminder: Multiplication is associative $ABC = (AB)C = A(BC)$.

Example:

$$\begin{aligned} A_1 &: 2 \times 100 \\ A_2 &: 100 \times 2 \\ A_3 &: 2 \times 2 \end{aligned}$$

Best order?

Matrix Multiplication

Another example (from [DPV]):

$$\begin{aligned} A_1 &: 50 \times 20 \\ A_2 &: 20 \times 1 \\ A_3 &: 1 \times 10 \\ A_4 &: 10 \times 100 \end{aligned}$$

Possible solutions:

Order	Cost Analysis	Cost
$A_1 \times ((A_2 \times A_3) \times A_4)$	$20 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A_1 \times (A_2 \times A_3)) \times A_4$	$20 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A_1 \times A_2) \times (A_3 \times A_4)$	$50 \cdot 20 + 10 \cdot 100 + 50 \cdot 100$	$7,000$

Note: greedy algorithm (make easy multiplication first), is **not** optimal.

Dynamic Programming solution

Main idea: define $C[i,j]$ for $1 \leq i < j \leq n$ as the minimum cost of multiplying matrices $A_i, \ldots, A_j$.

- Base case: $C[i,i] = 0, C[i,i+1] = r_{i-1}r_ir_{i+1}$.
- Want to know: $C[1,n]$.
- What is a “smaller” subproblem?

Dynamic Programming solution

Main idea: define $C[i, j]$ for $1 \leq i < j \leq n$ as the minimum cost of multiplying matrices $A_i, \ldots, A_j$.

- Base case: $C[i, i] = 0, C[i, i+1] = r_{i-1} r_i r_{i+1}$.
- Want to know: $C[1, n]$.

- What is a “smaller” subproblem?
- We will calculate $C[i, j]$ in order of increasing $(j - i)$.

$$C[i, j] = \min_{k : i < k < j} C[i, k] + C[k+1, j] + r_{i-1} r_k r_j$$

Dynamic Programming solution

Main idea: define $C[i,j]$ for $1 \leq i < j \leq n$ as the minimum cost of multiplying matrices $A_i, \ldots, A_j$.

- Base case: $C[i,i] = 0, C[i,i+1] = r_{i-1}r_ir_{i+1}$.
- Want to know: $C[1,n]$.

- What is a "smaller" subproblem?
- We will calculate $C[i,j]$ in order of increasing $(j-i)$.

$$C[i,j] = \min_{k:i<k<j} C[i,k] + C[k+1,j] + r_{i-1}r_kr_j$$

- Explanation: there will be several multiplications that will be done for the matrices $A_i, \ldots, A_j$. The last multiplication will involve the product of matrices $A_i, \ldots, A_k$, with the product of matrices $A_{k+1}, \ldots, A_j$.
- If we are given k the best way to do this is to
  - Optimally do $A_i \ldots A_k$
  - Optimally do $A_{k+1} \ldots A_j$
  - Do the last multiplication (fixed cost)
- We pick the best k

Complexity

- We need to fill up the $C[i, j]$ table
- Table has $O(n^2)$ elements.
- For each element we spend $O(n)$ time.
- $\Rightarrow$ algorithm to find optimal planning takes $O(n^3)$.

Complexity

- We need to fill up the $C[i, j]$ table
- Table has $O(n^2)$ elements.
- For each element we spend $O(n)$ time.
- $\Rightarrow$ algorithm to find optimal planning takes $O(n^3)$.
- As before, algorithm can be modified to output the optimal planning instead of just its cost.

Summary

Important lessons to remember.

- Induction/Recursion are powerful techniques
  - Solve problem by solving sub-problems.
- Divide&Conquer:
  - Implement with recursion
  - Sub-problems usually much smaller
  - Analyze running time with Master Theorem/recurrence relations
- Dynamic Programming:
  - More efficient/powerful by making more clever us of memory.
  - Avoid recomputing the same subproblems.
  - Running time usually close to memory usage.