Introduction to Algorithms: Lecture 5

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Homework & Experiments

1. All office hours in April are in 3A103
2. Experiment 2 is due on Thursday
3. HW 3 will be due on next Wednesday

Outline

Overview

Next, discuss two powerful techniques in algorithm-design:

1. Dynamic programming
2. Greedy method

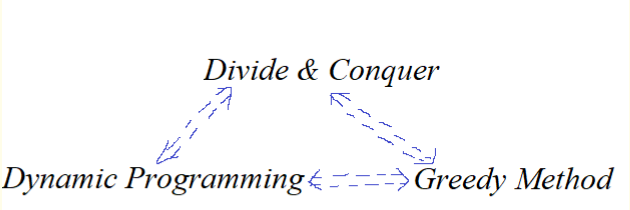
Similar to divide & conquer, they optimize the running time.

Overview

Next, discuss two powerful techniques in algorithm-design:

1. Dynamic programming
2. Greedy method

Similar to divide & conquer, they optimize the running time.



While these three methods have similar ideas, their implementations, analyses, and proofs are quite different.

Introduction

Dynamic Programming (DP): Divide & Conquer ++

Introduction

Dynamic Programming (DP): Divide & Conquer ++

Recall 3 steps in divide & conquer

1. Divide problem into subproblems
2. Solve subproblems recursively
3. Combine solutions of subproblems

Introduction

Dynamic Programming (DP): Divide & Conquer ++

Recall 3 steps in divide & conquer

1. Divide problem into subproblems
2. Solve subproblems recursively
3. Combine solutions of subproblems

Usually, we apply dynamic programming for optimization problems.

1. Similar to divide & conquer: Reduce to subproblems.
2. Differences: (1) division step in DP is more complicated; (2) DP handles subproblems by Memoization

Outline

Background

Machine learning and data science always need to compute a sequence of matrix multiplications.

Example: linear regression

Given $A \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, $\min_\beta \|A\beta - y\|_2$ has $\beta^* = (A^\top A)^{-1} A^\top y$.

Background

Machine learning and data science always need to compute a sequence of matrix multiplications.

Example: linear regression

Given $A \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, $\min_\beta \|A\beta - y\|_2$ has $\beta^* = (A^\top A)^{-1} A^\top y$.

After computing $(A^\top A)^{-1}$,

$(A^\top A)^{-1} \cdot (A^\top \cdot y)$ is faster than $\left((A^\top A)^{-1} \cdot A^\top \right) y$.

Background

Machine learning and data science always need to compute a sequence of matrix multiplications.

Example: linear regression

Given $A \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, $\min_\beta \|A\beta - y\|_2$ has $\beta^* = (A^\top A)^{-1} A^\top y$.

After computing $(A^\top A)^{-1}$,
$(A^\top A)^{-1} \cdot (A^\top \cdot y)$ is faster than $\left((A^\top A)^{-1} \cdot A^\top\right) y$.

Problem

Given dimensions $p_0, \ldots, p_n$ and matrices $A_1, \ldots, A_n$ with $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$, fully parenthesize product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications.

Intuition

Problem

Given $p_0, \ldots, p_n \in \mathbb{Z}^+$ and matrices $A_1, \ldots, A_n$ with $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$, fully parenthesize product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications.

There are $2^{\Omega(n)}$ methods to add parenthesizes on $A_1 A_2 \cdots A_n$:

$$(A_1 A_2) \cdot (A_3 A_4), ((A_1 A_2) A_3) \cdot A_4, (A_1 (A_2 A_3)) \cdot A_4, A_1 \cdot ((A_2 A_3) A_4), \ldots$$

Intuition

Problem

Given $p_0, \ldots, p_n \in \mathbb{Z}^+$ and matrices $A_1, \ldots, A_n$ with $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$, fully parenthesize product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications.

There are $2^{\Omega(n)}$ methods to add parenthesizes on $A_1 A_2 \cdots A_n$:

$$(A_1 A_2) \cdot (A_3 A_4), ((A_1 A_2) A_3) \cdot A_4, (A_1 (A_2 A_3)) \cdot A_4, A_1 \cdot ((A_2 A_3) A_4), \ldots$$

Intuition

Problem

Given $p_0, \dots, p_n \in \mathbb{Z}^+$ and matrices $A_1, \dots, A_n$ with $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$, fully parenthesize product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications.

There are $2^{\Omega(n)}$ methods to add parenthesizes on $A_1 A_2 \cdots A_n$:

$$(A_1 A_2) \cdot (A_3 A_4), ((A_1 A_2) A_3) \cdot A_4, (A_1 (A_2 A_3)) \cdot A_4, A_1 \cdot ((A_2 A_3) A_4), \dots$$

Basic idea

1. The last product $\cdot$ splits them into two independent subproblems.
2. Global optimal solution $\Leftrightarrow$ optimal on the two subproblems.

Intuition

Problem

Given $p_0, \ldots, p_n \in \mathbb{Z}^+$ and matrices $A_1, \ldots, A_n$ with $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$, fully parenthesize product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications.

There are $2^{\Omega(n)}$ methods to add parenthesizes on $A_1 A_2 \cdots A_n$:

$$(A_1 A_2) \cdot (A_3 A_4), ((A_1 A_2) A_3) \cdot A_4, (A_1 (A_2 A_3)) \cdot A_4, A_1 \cdot ((A_2 A_3) A_4), \ldots$$

Basic idea

1. The last product $\cdot$ splits them into two independent subproblems.
2. Global optimal solution $\Leftrightarrow$ optimal on the two subproblems.

Question: How to split them? — Let us try all choices of splits.

Memoization

---

function $\text{ChainOrder}(s, t)$
    if $s = t$ **then**
        Return 0
    else if $s + 1 = t$ **then**
        Return $p_{s-1} \times p_s \times p_t$
    else
        $ans = +\infty$
        for $i \in [s, t)$ **do** //Enumerate the split
            $ans = \min\left\{ans, \text{ChainOrder}(s, i) + \text{ChainOrder}(i + 1, t) + p_{s-1} \times p_i \times p_t\right\}$
        Return ans

Memoization

---

function $\text{ChainOrder}(s, t)$
  if $s = t$ **then**
    Return 0
  else if $s + 1 = t$ **then**
    Return $p_{s-1} \times p_s \times p_t$
  else
    $ans = +\infty$
    for $i \in [s, t)$ **do** //Enumerate the split
      $ans = \min\{ans, \text{ChainOrder}(s, i) + \text{ChainOrder}(i + 1, t) + p_{s-1} \times p_i \times p_t\}$
    Return ans

While it does find the answer, $2^{\Omega(n)}$ calls of ChainOrder. However, there are only $\binom{n+1}{2}$ pairs (s, t).

Memoization

Solution

Once calculated $\textsc{ChainOrder}(s,t)$, Let $f[s,t]$ record the answer $\textsc{ChainOrder}(s,t)$ for future calls.

---

procedure $\textsc{ChainM}(s,t)$
// preprocess $s = t$ and $s + 1 = t$
    if $f[s,t] = +\infty$ **then**
        for $i \in [s,t)$ **do**
            $f[s,t] \;=\; \min\left\{f[s,t], \textsc{ChainM}(s,i) + \textsc{ChainM}(i+1,t) + \right.$
$\left. + p_{s-1} \times p_i \times p_t\right\}$
    Return $f[s,t]$

Memoization

Solution

Once calculated $\textsc{ChainOrder}(s,t)$, Let $f[s,t]$ record the answer $\textsc{ChainOrder}(s,t)$ for future calls.

---

procedure $\textsc{ChainM}(s,t)$
// preprocess $s=t$ and $s+1=t$
  if $f[s,t]=+\infty$ **then**
    for $i\in[s,t)$ **do**
      $f[s,t] = \min\left\{f[s,t], \textsc{ChainM}(s,i)+\textsc{ChainM}(i+1,t)+ +p_{s-1}\times p_i\times p_t\right\}$
  Return $f[s,t]$

Since we start by calling $\textsc{ChainM}(1,n)$, this is called top-down with memoization.

Bottom-Up Method

(s, i) and $(i+1, t)$ in loop are strictly smaller than (s, t) — we can compute $f[s, t]$ from smaller intervals to large ones, called bottom-up method.

procedure MAIN
  Pre-process $f[s, t]$ where $s = t$ or $s + 1 = t$
  for $\ell = 2, \ldots, n-1$ **do** // $\ell = t - s$
    for $s = 1, \ldots, n - \ell$ **do**
      $t = s + \ell$, $f[s, t] = +\infty$
      for $i \in [s, t)$ **do**
        $f[s, t] = \min\left\{f[s, t], f[s, i] + f[i+1, t] + p_{s-1} \times p_i \times p_t\right\}$
  Return $f[1, n]$

Bottom-Up Method

(s, i) and $(i+1, t)$ in loop are strictly smaller than (s, t) — we can compute $f[s, t]$ from smaller intervals to large ones, called bottom-up method.

---

procedure MAIN
    Pre-process $f[s, t]$ where $s = t$ or $s + 1 = t$
    for $\ell = 2, \dots, n-1$ **do** // $\ell = t - s$
        for $s = 1, \dots, n - \ell$ **do**
            $t = s + \ell$, $f[s, t] = +\infty$
            for $i \in [s, t)$ **do**
                $f[s, t] = \min\left\{f[s, t], f[s, i] + f[i+1, t] + p_{s-1} \times p_i \times p_t\right\}$
    Return $f[1, n]$

We enumerate (ℓ, s) instead of (s, t) to guarantee $f[s, i]$ and $f[i+1, t]$ have been calculated.

Discussion

Analysis

1. Correctness follows by an induction proof.
2. Running time $O(n^3)$ — 3 loops.

Question: How to find the construction?

Discussion

Analysis

1. Correctness follows by an induction proof.
2. Running time $O(n^3)$ — 3 loops.

Question: How to find the construction?

---

procedure $\textsc{Parens}(s, t)$
  if $s = t$ **then**
    Print(A_i)
  else
    Print("(")
    Parens($s, d[s,t]$)
    Parens($d[s,t]+1, t$)
    Print(")")

---

1. Record $\arg\min_i$ as $d[s,t]$ in $\textsc{Main}$

Discussion

Analysis

1. Correctness follows by an induction proof.
2. Running time $O(n^3)$ — 3 loops.

Question: How to find the construction?

---

procedure $\textsc{Parens}(s, t)$
    if $s = t$ **then**
        Print(A_i)
    else
        Print("(")
        Parens($s, d[s, t]$)
        Parens($d[s, t] + 1, t$)
        Print(")")

---

1. Record $\arg\min_i$ as $d[s, t]$ in $\textsc{Main}$
2. Print the construction from top-bottom.

Summary

While dynamic programming utilizes answers from subproblems, two differences between divide & conquer:

1. DP enumerates all splits.
2. DP memorize answers of all subproblems

Usually DP analyzes the problem from top to bottom but implement it from bottom to top (faster).

Summary

While dynamic programming utilizes answers from subproblems, two differences between divide & conquer:

1. DP enumerates all splits.
2. DP memorize answers of all subproblems

Usually DP analyzes the problem from top to bottom but implement it from bottom to top (faster).

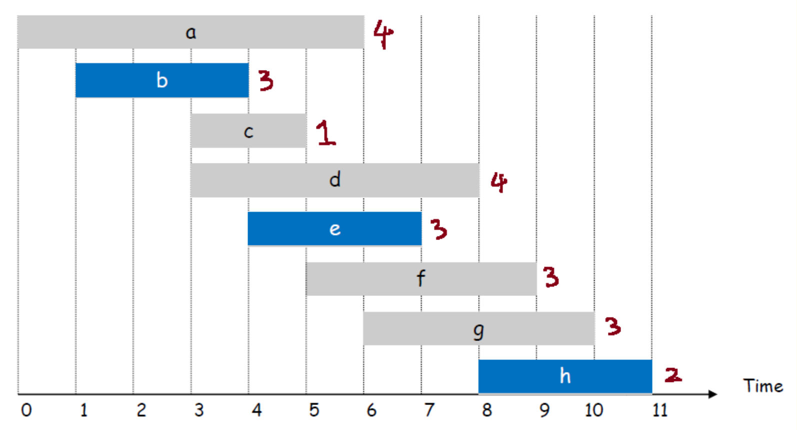
More examples

(1) Different problems consider various splits.
(2) Improve the running time of DP.

Outline

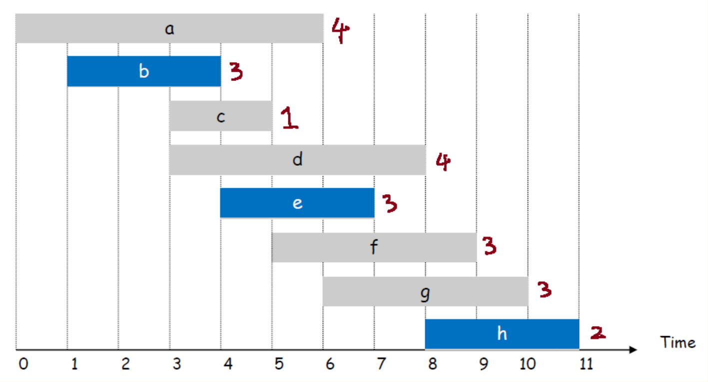
Interval Scheduling

1. n jobs in $[0, T]$
2. Job j starts from $h(j)$ and ends at $e(j)$ with weight w_j
3. Two jobs are compatible if they don't overlap.
4. Goal: find max-weight subset with compatible jobs.



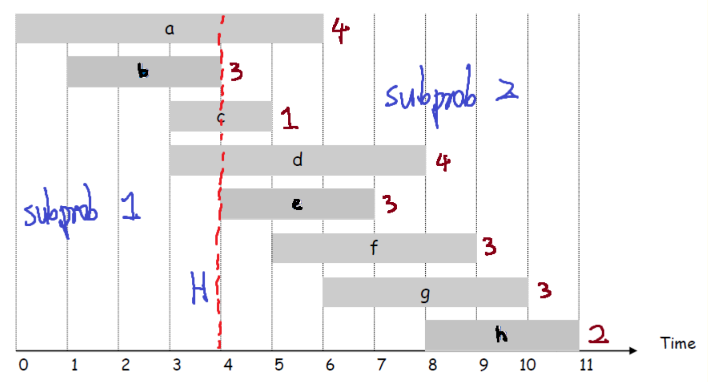
Basic Idea

Reduce it to subproblems: max-weight subsets in $[0, H]$ and $[H, T]$



Basic Idea

Reduce it to subproblems: max-weight subsets in $[0, H]$ and $[H, T]$

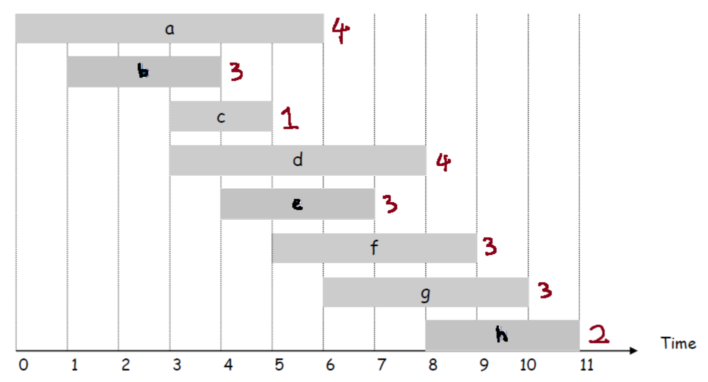


Key point

The optimal solution is optimal again on the two subproblems!

Basic Idea

Reduce it to subproblems: max-weight subsets in $[0, H]$ and $[H, T]$



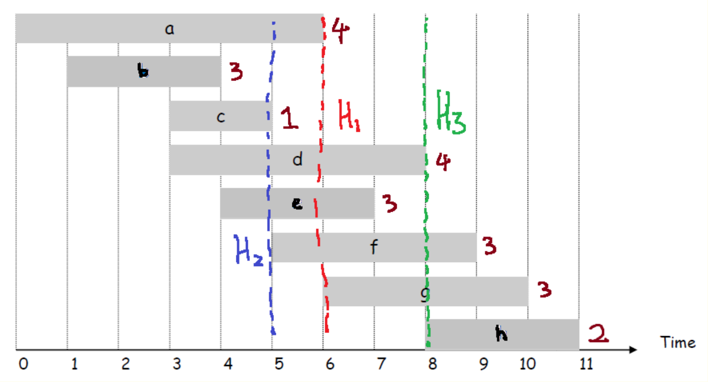
Key point

The optimal solution is optimal again on the two subproblems!

How to choose H to divide them?

Basic Idea

Reduce it to subproblems: max-weight subsets in $[0, H]$ and $[H, T]$



Key point

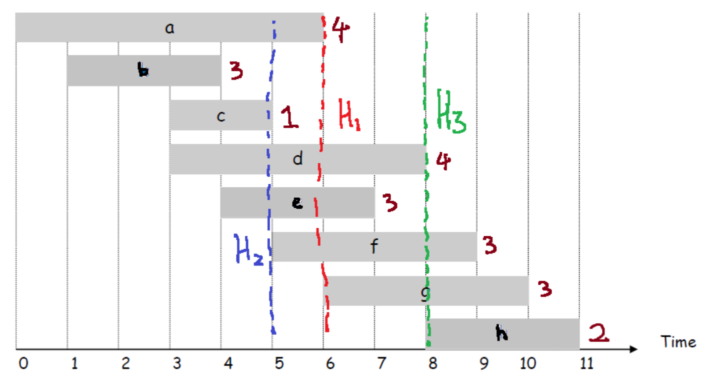
The optimal solution is optimal again on the two subproblems!

How to choose H to divide them?

We do not know the optimal solutions yet — enumerate H

Subproblem

Let $OPT(s,t)$ denote the max-weight of compatible subsets from time s to t.



Enumerate the split H of optimal solutions:

$$OPT(s,t) = \max_{H \in (s,t)} \left\{ OPT(s,H) + OPT(H,t) \right\}.$$

Bottom-Up method

---

procedure MAIN
  for $\ell = 1, \ldots, T$ **do**
    for $s = 0, \ldots, T - s$ **do**
      $t = s + \ell, opt[s, t] = 0$
      for $j = 1, \ldots, n$ **do**   // solution with one interval in $[s, t]$
        if $\big[h[j], e[j]\big] \subseteq [s, t]$ **then**
          $opt[s, t] = \max\{opt[s, t], w[j]\}$
      for $H = s + 1, \ldots, t - 1$ **do** // solutions with $\geqslant 2$ intervals
        $opt[s, t] = \max\{opt[s, t], opt[s, H] + opt[H, t]\}$
  Return $opt[0, T]$

While it works, it is really slow — $O(T^3)$.

Question

How to improve it?

Improve the running time

1. In the compatible subsets, the order doesn't matter — enumerate the last interval j instead of H.

Improve the running time

1. In the compatible subsets, the order doesn't matter — enumerate the last interval j instead of H.
2. 2 cases depend on whether last interval j ends at time t or not:

$$f[0,t] = \max\left\{ f[0,t-1], w(j) + f[0,h(j)] \text{ for all } j : e(j) = t \right\}$$

Improve the running time

1. In the compatible subsets, the order doesn't matter — enumerate the last interval j instead of H.
2. 2 cases depend on whether last interval j ends at time t or not:

$$f[0,t] = \max\left\{ f[0,t-1], w(j) + f[0,h(j)] \text{ for all } j : e(j) = t \right\}$$

3. Each j only appears once when $t = e(j)$ — just consider end points $T = e(1), \ldots, e(n)$

Improve the running time

1. In the compatible subsets, the order doesn't matter — enumerate the last interval j instead of H.
2. 2 cases depend on whether last interval j ends at time t or not:

$$f[0,t] = \max\left\{ f[0,t-1], w(j) + f[0,h(j)] \text{ for all } j : e(j) = t \right\}$$

3. Each j only appears once when $t = e(j)$ — just consider end points $T = e(1), \ldots, e(n)$

---

procedure MAIN(n, h, e)

Sort n intervals s.t. $e(1) \leqslant e(2) \leqslant \ldots \leqslant e(n)$

Compute the last compatible interval $p(1), \ldots, p(n)$ for each j, i.e., $p(j) = \max_{i:e(i)\leqslant h(j)}\{i\}$

$g[0] = 0$

for $j = 1, \ldots, n$ **do**

$$g(j) = \max\left\{ g[j-1], w(j) + g[p(j)] \right\}$$

Discussion

1. Correctness follows from the above discussion.
2. Running time is $O(n \log n)$ — but how to compute p? Use a BST or

Discussion

1. Correctness follows from the above discussion.
2. Running time is $O(n \log n)$ — but how to compute p? Use a BST or
   1. Given $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$, compute the order of h s.t. $h\big(rank(1)\big) \leqslant h\big(rank(2)\big) \leqslant \cdots \leqslant h\big(rank(n)\big)$

Discussion

1. Correctness follows from the above discussion.
2. Running time is $O(n \log n)$ — but how to compute p? Use a BST or
   1. Given $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$, compute the order of h s.t. $h\big(rank(1)\big) \leqslant h\big(rank(2)\big) \leqslant \cdots \leqslant h\big(rank(n)\big)$
   2. Compute $p\big(rank(1)\big), \ldots, p\big(rank(n)\big)$ by moving an index on $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$

Discussion

1. Correctness follows from the above discussion.
2. Running time is $O(n \log n)$ — but how to compute p? Use a BST or
   1. Given $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$, compute the order of h s.t. $h\big(rank(1)\big) \leqslant h\big(rank(2)\big) \leqslant \cdots \leqslant h\big(rank(n)\big)$
   2. Compute $p\big(rank(1)\big), \ldots, p\big(rank(n)\big)$ by moving an index on $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$

Improve the running time

1. $O(T^3)$ time DP at first
2. Realize $f[s, t]$ could be reduce to $g(j)$ where $j = 1, \ldots, n$
3. Furthermore, consider job j with $e(j) = t$ instead of enumerating H

Discussion

1. Correctness follows from the above discussion.
2. Running time is $O(n \log n)$ — but how to compute p? Use a BST or
    1. Given $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$, compute the order of h s.t. $h\big(rank(1)\big) \leqslant h\big(rank(2)\big) \leqslant \cdots \leqslant h\big(rank(n)\big)$
    2. Compute $p\big(rank(1)\big), \ldots, p\big(rank(n)\big)$ by moving an index on $e(1) \leqslant e(2) \leqslant \cdots \leqslant e(n)$

Improve the running time

1. $O(T^3)$ time DP at first
2. Realize $f[s, t]$ could be reduce to $g(j)$ where $j = 1, \ldots, n$
3. Furthermore, consider job j with $e(j) = t$ instead of enumerating H

We can find solution again by memorizing all decisions.

Summary

A rough blue-print to apply dynamic programming:

1. The solution has lots of decisions (e.g., every chosen job in scheduling and the order of matrix multiplications)

Summary

A rough blue-print to apply dynamic programming:

1. The solution has lots of decisions (e.g., every chosen job in scheduling and the order of matrix multiplications)
2. Consider the optimal one and enumerate one choice to split the problem into subproblems

Summary

A rough blue-print to apply dynamic programming:

1. The solution has lots of decisions (e.g., every chosen job in scheduling and the order of matrix multiplications)
2. Consider the optimal one and enumerate one choice to split the problem into subproblems
3. Prove that
   $\Rightarrow$ Optimal solution is optimal on subproblems.
   $\Leftarrow$ Subproblems are independent s.t. any optimal solutions of them could be combined to a global optimal solution

Summary

A rough blue-print to apply dynamic programming:

1. The solution has lots of decisions (e.g., every chosen job in scheduling and the order of matrix multiplications)
2. Consider the optimal one and enumerate one choice to split the problem into subproblems
3. Prove that
   $\Rightarrow$ Optimal solution is optimal on subproblems.
   $\Leftarrow$ Subproblems are independent s.t. any optimal solutions of them could be combined to a global optimal solution

Different from divide & conquer: The last two proofs could be tricky.

Summary (II)

To design DP,

1. Figure out the space of subproblems, i.e., define $OPT[s, t]$
2. Follow a bottom-up order
3. Enumerate the choice and combine solutions of subproblems

Summary (II)

To design DP,

1. Figure out the space of subproblems, i.e., define $OPT[s, t]$
2. Follow a bottom-up order
3. Enumerate the choice and combine solutions of subproblems

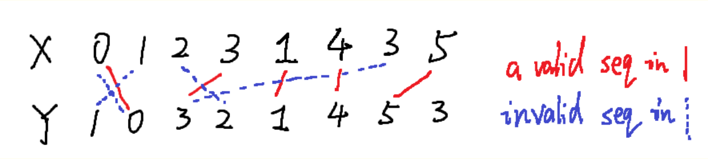
Roughly, running time is $O\Big(\text{size(space)} \times \#\text{choices}\Big)$

— Optimizing the running time is highly non-trivial!

Outline

Problem Intro

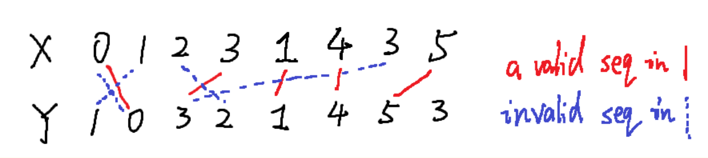
Given two sequences $X = (x_1, \ldots, x_m)$ and $Y = (Y_1, \ldots, Y_n)$, find the longest common sequence.



Formally, $Z = (z_1, \ldots, z_k)$ is a common sequence of X if $\exists$ strictly increasing seq $i_1 < \ldots < i_k$ of indices of X s.t. $z_j = X_{i_j}$ for $j \in [k]$.

Problem Intro

Given two sequences $X = (x_1, \ldots, x_m)$ and $Y = (Y_1, \ldots, Y_n)$, find the longest common sequence.



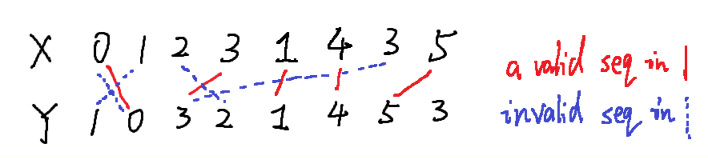
Formally, $Z = (z_1, \ldots, z_k)$ is a common sequence of X if $\exists$ strictly increasing seq $i_1 < \ldots < i_k$ of indices of X s.t. $z_j = X_{i_j}$ for $j \in [k]$.

Goal

Find the longest sequence Z that is common in both X and Y.

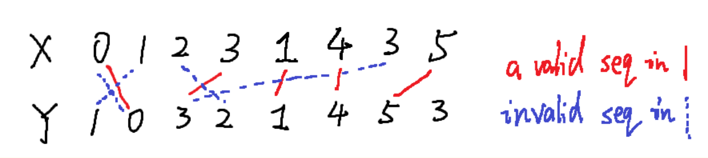
Basic Idea

Each red line denotes one choice.



Basic Idea

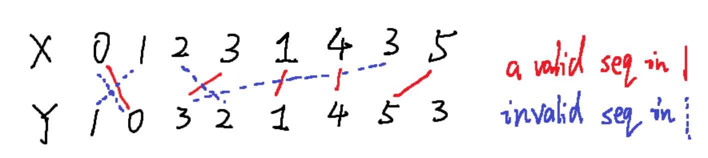
Each red line denotes one choice.



1. Similar to the interval scheduling problem, consider the last line in the optimal solution say $x_i \leftrightarrow y_j$

Basic Idea

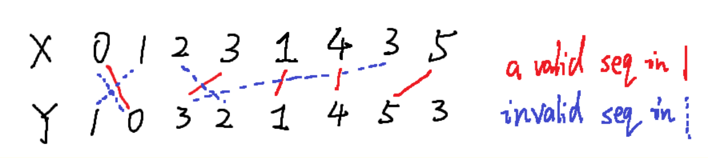
Each red line denotes one choice.



1. Similar to the interval scheduling problem, consider the last line in the optimal solution say $x_i \leftrightarrow y_j$
2. The optimal solution must be optimal on subproblems $(x_1, \ldots, x_{i-1})$ and $(y_1, \ldots, y_{j-1})$

Basic Idea

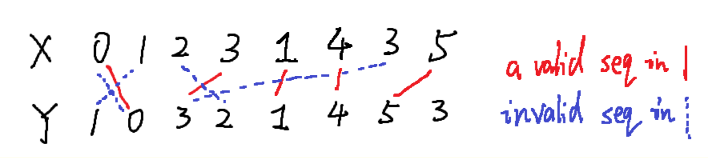
Each red line denotes one choice.



1. Similar to the interval scheduling problem, consider the last line in the optimal solution say $x_i \leftrightarrow y_j$
2. The optimal solution must be optimal on subproblems $(x_1, \ldots, x_{i-1})$ and $(y_1, \ldots, y_{j-1})$
3. Moreover any optimal solution on the subproblem is good.

Basic Idea

Each red line denotes one choice.



1. Similar to the interval scheduling problem, consider the last line in the optimal solution say $x_i \leftrightarrow y_j$
2. The optimal solution must be optimal on subproblems $(x_1, \ldots, x_{i-1})$ and $(y_1, \ldots, y_{j-1})$
3. Moreover any optimal solution on the subproblem is good.

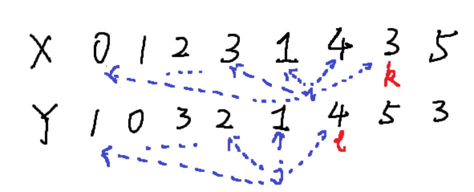
So the space of subproblems is $[1, \ldots, i] \times [1 \ldots, j]$ for all $i \in [n]$ and $j \in [m]$.

Formal Description

Let $c[k, \ell]$ be the length of the longest common sequence between $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_\ell)$.

Formal Description

Let $c[k, \ell]$ be the length of the longest common sequence between $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_\ell)$.

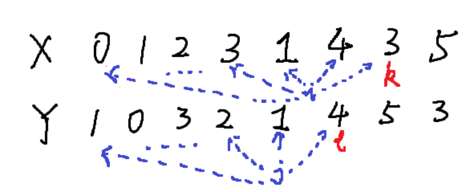


1. To compute it, try enumerating the position of the last line $x_i \leftrightarrow y_j$:

$$c[k, \ell] = \max_{i \leqslant k, j \leqslant \ell : x_i = y_j} \{c[i-1, j-1] + 1\}$$

Formal Description

Let $c[k, \ell]$ be the length of the longest common sequence between $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_\ell)$.



1. To compute it, try enumerating the position of the last line $x_i \leftrightarrow y_j$:

$$c[k, \ell] = \max_{i \leqslant k, j \leqslant \ell : x_i = y_j} \{c[i-1, j-1] + 1\}$$

2. While it is correct, running time is relatively slow $O(m^2 n^2)$.
3. Can we do better?

Improvement

Key OBS: Only need to check $x_k = y_j$ or not.

Theorem (15.1 in CLRS)

Let $Z = (z_1, \ldots, z_k)$ be any LCS of $(x_1, \ldots, x_m)$ and $(y_1, \ldots, y_n)$.

1. $x_m = y_n$: $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. $x_m \neq y_n$: If $z_k \neq x_m$, Z must be an LCS of X_{m-1} and Y_n.
3. $x_m \neq y_n$: If $z_k \neq y_n$, Z must be an LCS of X_m and Y_{n-1}.

Faster DP

While we do not know Z to apply the THM, it still tells there are only 3 cases:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\left\{ c[i-1, j], c[i, j-1] \right\} & o.w. \end{cases}$$

The running time becomes $O(nm)$ ☺

Faster DP

While we do not know Z to apply the THM, it still tells there are only 3 cases:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\left\{c[i-1, j], c[i, j-1]\right\} & o.w. \end{cases}$$

The running time becomes $O(nm)$ ☺

Constructing LCS

Remember where does $c[i,j]$ come from (called the decision) and trace back to output the sequence

Summary

Usually it is highly non-trivial to improve the running time of DP.

1. Running time = Space of subproblems $\times$ enumeration
2. Many ways to improve the running time

Summary

Usually it is highly non-trivial to improve the running time of DP.

1. Running time = Space of subproblems $\times$ enumeration
2. Many ways to improve the running time
3. Reduce the space
4. Use data structure or powerful lemmas to reduce the enumeration

Summary

Usually it is highly non-trivial to improve the running time of DP.

1. Running time = Space of subproblems $\times$ enumeration
2. Many ways to improve the running time
3. Reduce the space
4. Use data structure or powerful lemmas to reduce the enumeration

Next

All previous DP algorithms are on lines/sequences. There are DP on trees (graphs) and matrices.

Outline

Problem Intro

Build a binary search tree to minimize the search time.

1. There are n keys $k_1, \ldots, k_n$ and $n+1$ dummy keys $d_0, \ldots, d_n$ s.t. $d_0 < k_1 < d_1 < k_2 < \cdots < k_n < d_n$.

Problem Intro

Build a binary search tree to minimize the search time.

1. There are n keys $k_1, \ldots, k_n$ and $n+1$ dummy keys $d_0, \ldots, d_n$ s.t. $d_0 < k_1 < d_1 < k_2 < \cdots < k_n < d_n$.
2. Each k_i appears with prob. p_i; each d_i appears with prob. q_i
3. $\sum_i p_i + \sum_i q_i = 1$

Problem Intro

Build a binary search tree to minimize the search time.

1. There are n keys $k_1, \ldots, k_n$ and $n+1$ dummy keys $d_0, \ldots, d_n$ s.t. $d_0 < k_1 < d_1 < k_2 < \cdots < k_n < d_n$.
2. Each k_i appears with prob. p_i; each d_i appears with prob. q_i
3. $\sum_i p_i + \sum_i q_i = 1$
4. Design a BST T s.t. (1) Internal nodes are keys; (2) Leaves are dummy keys; (3) Minimize the expected cost of a search in T:

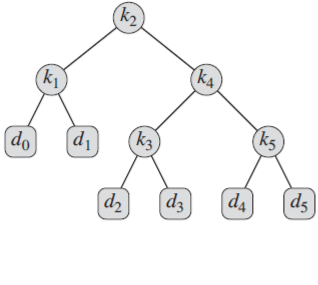
$$\mathbb{E}[\text{search cost in } T] := \mathop{\mathbb{E}}_{key \sim \{d_0, \ldots, d_n, k_1, \ldots, k_n\}} [\text{depth}(key) \text{ in } T].$$

Example

Figure 15.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

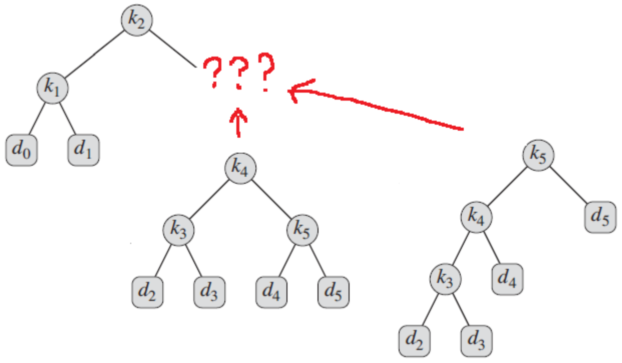
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two binary search trees with expected cost 2.8 and 2.75 separately:



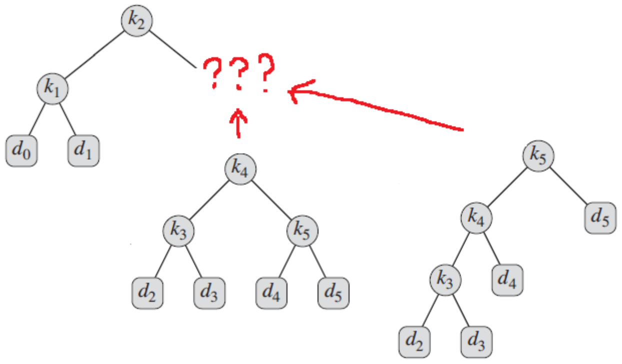
(a) (b)

Step 1: Reduce to subproblems



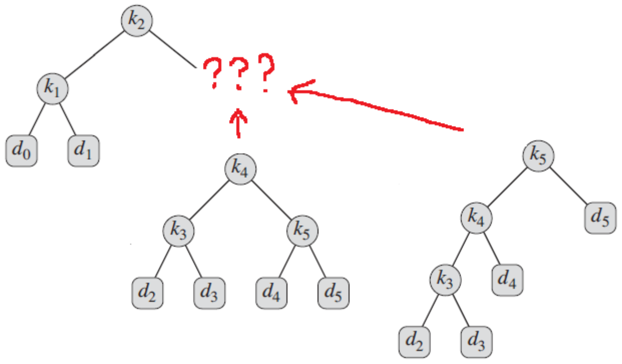
1. Say the optimal BST has root k_2 — consider its left/right subtrees.

Step 1: Reduce to subproblems



1. Say the optimal BST has root k_2 — consider its left/right subtrees.
2. Various possible subtrees with roots k_3, k_4, k_5 separately.

Step 1: Reduce to subproblems



1. Say the optimal BST has root k_2 — consider its left/right subtrees.
2. Various possible subtrees with roots k_3, k_4, k_5 separately.
3. In the optimal BST, this subtree must be optimal among k_3, k_4, k_5 and $d_2, \ldots, d_5$.
4. Vice versa — any optimal subtree could be plugged into the optimal BST to give one optimal construction

Step: Recursive Formula

The subtree could contain any section of $k_1, \dots, k_n$.

1. Consider $e[i, j]$ as the min-expected-cost BST with keys $k_i, \dots, k_j$ and $d_{i-1}, \dots, d_j$.

Step: Recursive Formula

The subtree could contain any section of $k_1, \ldots, k_n$.

1. Consider $e[i, j]$ as the min-expected-cost BST with keys $k_i, \ldots, k_j$ and $d_{i-1}, \ldots, d_j$.
2. If $j = i - 1$, BST only has d_{i-1}: $e[i, j] = q_{i-1}$

Step: Recursive Formula

The subtree could contain any section of $k_1, \ldots, k_n$.

1. Consider $e[i,j]$ as the min-expected-cost BST with keys $k_i, \ldots, k_j$ and $d_{i-1}, \ldots, d_j$.
2. If $j = i - 1$, BST only has d_{i-1}: $e[i,j] = q_{i-1}$
3. Otherwise, enumerate the root r:

$$e[i,j] = \min_{r \in [i,j]} \{e[i, r-1] + e[r+1, j] + \text{ one more depth on them}\}$$

Note the last term $:= \underbrace{p_i + \cdots + p_j + q_{i-1} + \cdots + q_j}_{\text{define the sum as } w[i,j]}$.

OPTIMAL-BST(p, q, n)

```
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Running Time

1. $O(n^3)$ because of the 3 loops

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Running Time

1. $O(n^3)$ because of the 3 loops
2. Improve it to $O(n^2)$: $root[i, j-1] \leqslant root[i, j] \leqslant root[i+1, j]$ by Knuth

Summary

1. DP extends the methods of divide & conquer to optimization problems.
2. Key structure: Optimal solutions of subproblems $\Leftrightarrow$ optimal solution

Summary

1. DP extends the methods of divide & conquer to optimization problems.
2. Key structure: Optimal solutions of subproblems $\Leftrightarrow$ optimal solution
3. Running time: Space of subproblems $\times$ enumerate

Summary

1. DP extends the methods of divide & conquer to optimization problems.
2. Key structure: Optimal solutions of subproblems $\Leftrightarrow$ optimal solution
3. Running time: Space of subproblems $\times$ enumerate
4. Optimizing running time could be challenging.

Questions?