



CS202: Design and Analysis of Algorithms

Week 5

DAI, Bing Tian and SUN, Jun



Dynamic Programming

Dynamic Programming

Dynamic programming is used to solve optimization problems with the following characteristics:

Optimal substructure (a.k.a. Principle of Optimality)

- If problem S is composed from S_1, S_2, \dots, S_n , the optimality of S implies the optimalities of S_1, S_2, \dots, S_n
- Example: If $A_{1..6} = A_{1..3} A_{4..6}$, the solutions for both $A_{1..3}$ and $A_{4..6}$ must be optimal too

Overlapping subproblems

- There exist some places where we solve the same subproblem more than once
- Example: In MCM, $A_{2..3}$ is common to the sub-problems $A_{1..3}$ and $A_{2..4}$
- Effort wasted in solving common sub-problems repeatedly

Coin Change with Limited Supply

It violates the principle of optimality with limited supplies of coins

$$\text{MinCoin}(n, k) = \begin{cases} 0 & n = 0 \\ \min(\text{MinCoin}(n, k-1), \text{MinCoin}(n - \text{denom}[k], k) + 1) & n > 0 \end{cases}$$

To make a change of n cents with up to k kinds of denominations, it is the minimum of

1. Using the first $k-1$ kinds of denominations, or
2. Using k kinds of denominations for $n - \text{denom}[k]$, and add 1 coin of $\text{denom}[k]$ cents

To limit the usage of coins with denomination $\text{denom}[k]$

- An additional check needed on the number of $\text{denom}[k]$ cents coins already used for $n - \text{denom}[k]$ cents

But to your surprise, this method is wrong!

A counter example: with 2 pieces of 100-cents coins, the optimal solution for 300 cents does not imply the optimality for 200 cents

Knapsack Example

$$\text{MaxValue}(i, W) = \begin{cases} 0 & i = 0 \\ \max\{\text{MaxValue}(i-1, W-w[i]) + v[i], \text{MaxValue}(i-1, W)\} & i > 0 \end{cases}$$

0/1 Knapsack: To maximize the value of a knapsack with capacity W , either

- Do not take item i , items are selected from $1, 2, \dots, i-1$ with capacity W , or
- Take item i , the remaining items are selected from $1, 2, \dots, i-1$ with capacity $W-w[i]$

Bounded Knapsack: consider taking $0, 1, 2, \dots, \text{qty}[i]$ pieces of item i , which one maximizes the total value?

$$\text{MaxValue}(i, W) = \begin{cases} 0 & i = 0 \\ \max_{k=0}^{\text{qty}[i]} \{\text{MaxValue}(i-1, W-k \cdot w[i]) + k \cdot v[i]\} & i > 0 \end{cases}$$

Longest Common Subsequence

Problem

Given two sequences $[a_1, a_2, \dots, a_m]$ and $[b_1, b_2, \dots, b_n]$, where each element can be a character or a word.

Find the longest sequence $[c_1, c_2, \dots, c_k]$ (i.e., maximize k) such that $[c_1, c_2, \dots, c_k]$ is a sub-sequence to both given sequences.

Example

After stemming, two sentences are converted to lists of words:

- ['in', 'park', 'several', 'kid', 'play', 'soccer']
- ['park', 'his', 'car', 'before', 'watch', 'play']

The longest common subsequence is

['park', 'play']

In-Class Exercise 1

Problem

Given two sequences $[a_1, a_2, \dots, a_m]$ and $[b_1, b_2, \dots, b_n]$, where each element can be a character or a word.

Find the longest sequence $[c_1, c_2, \dots, c_k]$ (i.e., maximize k) such that $[c_1, c_2, \dots, c_k]$ is a sub-sequence to both given sequences.

Questions

- What are the sub-problems?
- Do sub-problems satisfy optimality?
- Are sub-problems overlapping?

Divide into Sub-Problems

List A	in	park	several	kid	play	soccer
			↕			
List B	park	his	car	before	watch	play

A sub-problem takes smaller inputs:

- What is the LCS up to the position of 'several' and the position of 'car'?

Case 1: the two words are not the same

- Where can the LCS come from?

Case 2: the two words are the same

- How do you construct the LCS?

Edit Distance

The edit distance between string s_1 and string s_2 is the minimum number of basic operations that convert s_1 to s_2

The basic operations are

- insert
- delete
- replace

Examples:


- `edit_dist('diary', 'binary') = 2`
- `edit_dist('hexagon', 'heptagon') = 2`
- `edit_dist('sleeplessness', 'selflessness') = 3`

Applications:

- DNA sequence comparison
- Approximate string matching
- Auto-correct spell check

Tabulate the Sub-Problems

	b	i	n	a	r	y	\0
d							
i							
a							
r							
y							
\0							



How is `edit_dist('iary', 'nary')` related to `edit_dist('iary', 'ary')`, `edit_dist('ary', 'nary')` and `edit_dist('ary', 'ary')`?

In-Class Exercise 2

Solve the edit distance problem using dynamic programming.



Greedy Algorithms



Greedy Algorithms

Reference:

- CLRS chapters 16.1-16.3 (skim 16.4 and 16.5)

Objectives:

- To learn the greedy algorithmic paradigm
- To apply greedy methods to solve several optimization problems
- To analyse the correctness of greedy algorithms

Greedy Algorithms

Key idea: makes the choice that looks best at the moment

- The hope: a locally optimal choice will lead to a globally optimal solution

Everyday examples:

- Driving
- Shopping

- Dynamic programming can be overkill
- Greedy algorithms tend to be easier to code
- Sometimes much harder to prove

Applications of Greedy Algorithms

Scheduling

- Activity Selection (Chap 16.1)
- Scheduling on a single processor (Chap. 16.5)

Graph Algorithms

- Minimum Spanning Trees (Chap 23)
- Dijkstra's (shortest path) Algorithm (Chap 24)

Other Combinatorial Optimization Problems

- Fractional Knapsack (Chap 16.2)
- Traveling Salesman approximation (Chap 35.2)
- Set-covering approximation (Chap 35.3)

Activity Selection Problem

Input: a list S of n activities = $\{a_1, a_2, \dots, a_n\}$

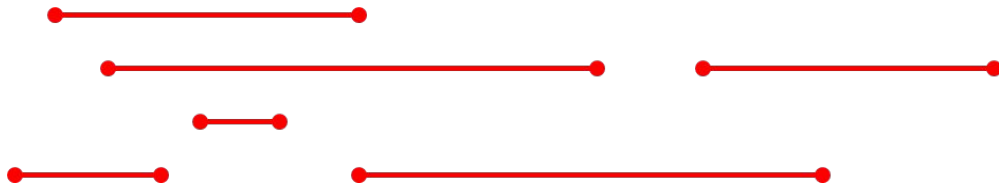
- s_i = start time of activity i
- f_i = finish time of activity i
- S is sorted by finish time, i.e. $f_1 \leq f_2 \leq \dots \leq f_n$

Output: a subset A of compatible activities of maximum size

Activities are compatible if $[s_i, f_i) \cap [s_j, f_j)$ is null.

Example:

How many possible solutions are there?



Real-world Applications of Activity Selection

Tourism

- Theme park itinerary planning
 - A passport lets you get onto any ride and show
 - Lots of rides and shows, each start and end at different times
 - Your goal is to maximize your “happiness”
- Multi-day tour itinerary planning

Berth allocation

- Allocation of berths to vessels

Event scheduling

- Allocation of rooms to events (e.g. examination)

Workforce scheduling

- Allocation of employees to tasks

Activity Selection Algorithms

Dynamic Programming:

S_{ij} : The set of activities

- start after activity a_i finishes, and
- finish before activity a_j starts

i.e.: S_{ij} is the set of activities compatible with a_i and a_j

A_{ij} : The optimal solution for S_{ij} , the recurrence relationship is defined by

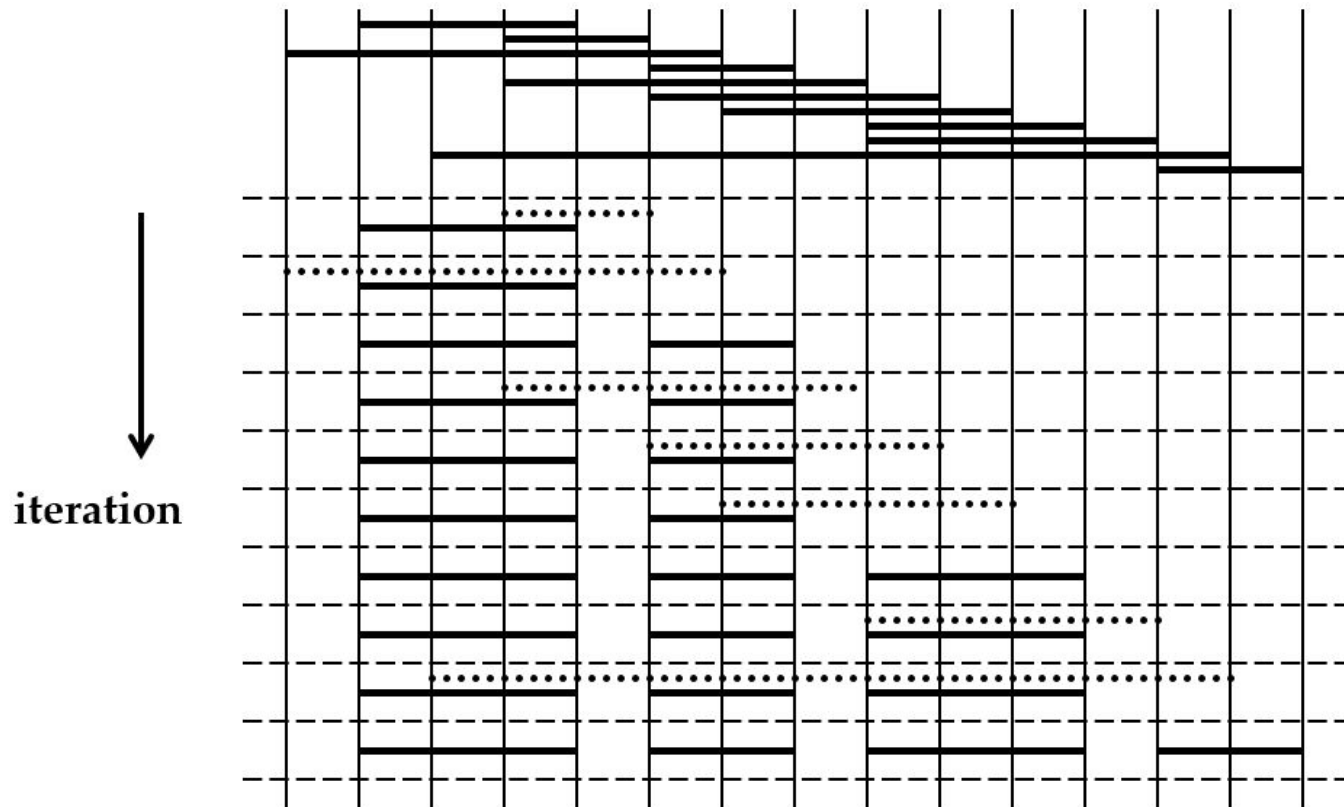
$$A_{ij} = \max_{k=i+1}^{j-1} (A_{ik} \cup \{a_k\} \cup A_{kj})$$

What's the time complexity?

Greedy Algorithm:

```
acty_size = len(activity)
selected = [0]
i = 0
for j in range(1, acty_size):
    if activity[j][0] >= activity[i][1]:
        selected.append(j)
        i = j
print('the selected activities by
greedy algorithm:', selected)
```

Example Run



When does Greedy work?

Two key ingredients:

1. Problem exhibits **optimal substructure**:

An optimal solution to the entire problem contains optimal solutions to subproblems

2. Algorithm exhibits **greedy choice property**:

Locally optimal solution \Rightarrow globally optimal solution, i.e., greedy choice is always safe.

Optimal substructure plus greedy choice establish the **correctness and optimality** of the greedy algorithm

Greedy choice property:

- Make the best choice at the moment
- Greedy choice: From a locally optimal solution S_j to sub-problem Q_j , a globally optimal solution S_i to problem Q_i (Q_j is a sub-problem of Q_i) can be constructed.
- Proofs often use exchange argument
 - If there were a “better” solution for Q_i that does not contain the greedy choice,
 - We could exchange the greedy choice into the “better” solution to get an equally optimal solution, leads to a contradiction that the “better” solution is better.

Proof

Let A_{ij} be an optimal solution for sub-problem S_{ij} and the solution A_{ij} includes activity a_k

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \quad \text{implies}$$

$$\implies \begin{cases} A_{ik} \text{ is an optimal solution for } S_{ik} \\ A_{kj} \text{ is an optimal solution for } S_{kj} \end{cases}$$

This proved the optimal substructure property.

Greedy choice property:

Let a_f be the first activity that is compatible with the current local solution A . Suppose there were a better solution B contains A but not a_f , we can always replace the first activity in $B \setminus A$ by a_f . After the exchange, the number of activities remains the same, thus the optimal solution, which we constructed from A with the presence of a_f , becomes "better".



In-class Exercise 3

A single server (a processor, an equipment, a cashier in a bank, etc) has n customers to serve

The service time required by each customer (or task) is given:

- customer i will take time t_i ; $1 \leq i \leq n$

Let C_i denote the completion time of customer i

We want to minimize the average completion time of all tasks, i.e., minimize the objective function

$$T = \frac{1}{n} \sum_{i=1}^n C_i$$

Example: Suppose we have 3 customers with

$$t_1 = 5, \quad t_2 = 10, \quad t_3 = 3$$

<i>Order</i>	<i>T</i>
1 2 3:	$5 + (5+10) + (5+10+3) = 38$
1 3 2:	$5 + (5+3) + (5+3+10) = 31$
2 1 3:	$10 + (10+5) + (10+5+3) = 43$
2 3 1:	$10 + (10+3) + (10+3+5) = 41$
3 1 2:	$3 + (3+5) + (3+5+10) = 29 \quad \leftarrow \text{optimal}$
3 2 1:	$3 + (3+10) + (3+10+5) = 34$

In-class Exercise 3: Proof

Can you use exchange argument to prove why it is always better to process customers with shorter service time?

Fractional Knapsack Problem

Knapsack Problem:

- There are n items, where the i^{th} item worth v_i dollars and weighs w_i kg
- knapsack carries at most W kg
- Question: what items should you take to maximize the profit?
- Assume n , v_i , w_i , and W are all integers

0-1 Knapsack problem:

- “0-1” since each item must be taken or left in entirety

A variant - Fractional Knapsack problem:

- can take fractions of items
- can be solved by a greedy algorithm

In-Class Exercise 4

The 0-1 problem cannot be solved with a greedy approach

- It can, however, be solved with dynamic programming, which has been gone through earlier this class

The fractional problem can be solved greedily

- Can you try to implement the greedy algorithm to solve the fractional problem?

Data Compression Problem

In the past,

- data storage space is small,
- communication speed is low,
- thus data compression is needed.

Nowadays,

- data storage space is very large,
- data transfer rate is very high,
- yet data compression is still needed as more and more files of bigger size (images, videos, etc) are stored or transferred.

Data Compression Problem

A data file of 100,000 characters contains only the characters a-f, with the frequencies as follows:

	a	b	c	d	e	f
Freq (x1000)	45	13	12	16	9	5
Fixed Length	000	001	010	011	100	101

Code: Each character is represented by a unique binary string, and no ambiguity in decoding

Fixed-length code

- need $3 \times 100,000 = 300,000$ bits to encode the file

	a	b	c	d	e	f
Freq (x1000)	45	13	12	16	9	5
Fixed Length	000	001	010	011	100	101
Variable Length	0	101	100	111	1101	1100

Variable-length code: short codes to frequent characters and longer codes to infrequent characters

- need $[1 \times 45 + 3 \times (13 + 12 + 16) + 4 \times (9 + 5)] \times 1000 = 224,000$ bits \Rightarrow A saving of 25%

Code Tree

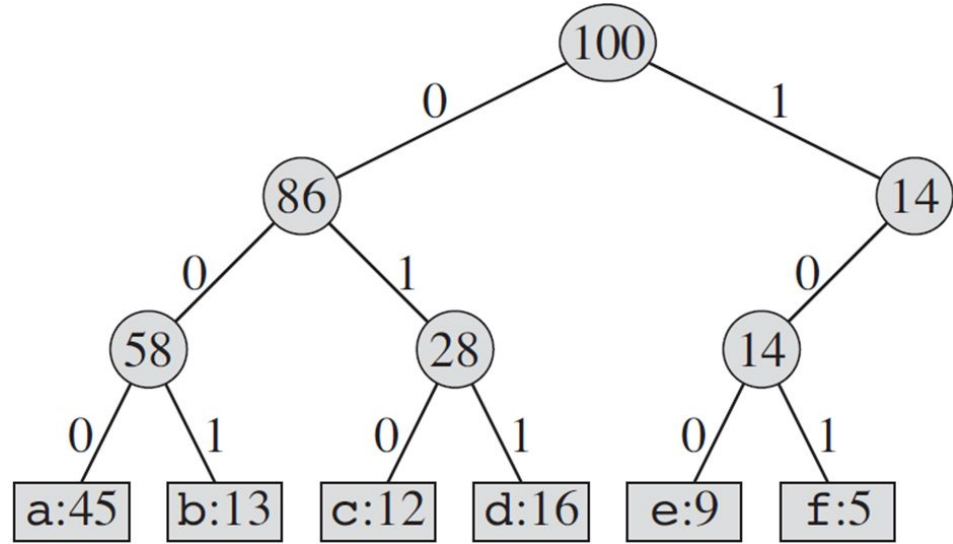
Codes can be represented as binary trees.

leaf nodes: characters

internal node: sum of frequencies of leaves in subtree

path from the root to each leaf node: code of the character

- Fixed length coding: all leaves are on the same level
- Variable length coding: leaves could appear on different levels



Question: why can't one character map to an internal node?

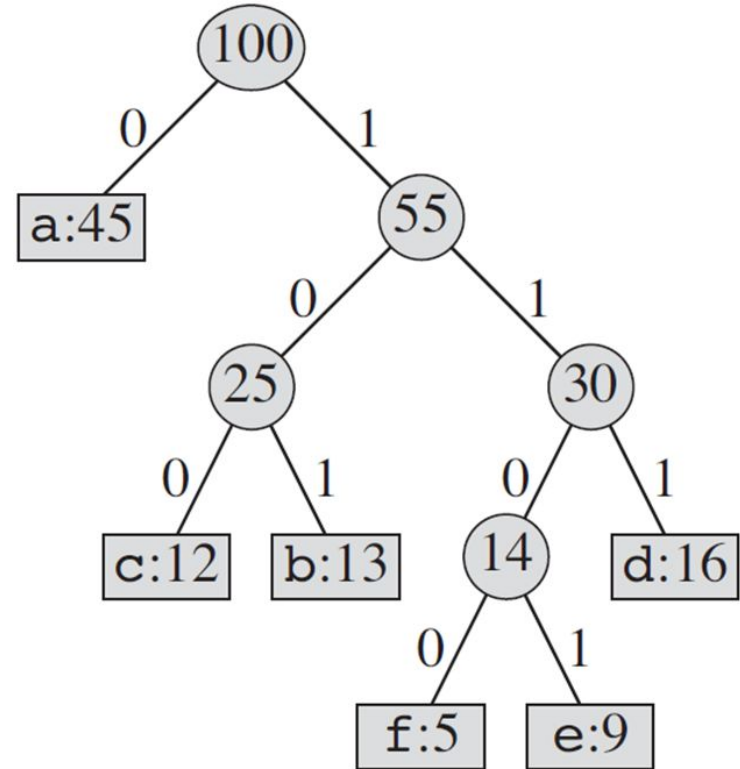
Prefix Code

Prefix codes: no code is allowed to be a prefix of some other code

Using prefix codes will avoid **ambiguity** in decoding.

Example:

- “001011101” is uniquely decoded to aabe
- If z is represented as 01, then the above string could be decoded as azze
- Why? the code for a being a prefix of the code for z



Optimal Coding

The fewer bits a coding requires, the more optimal the coding is.

Problem:

Given C and f , the set of characters and the frequencies of all characters, how to find a code tree T that minimizes $B(T)$?

Solution:

Huffman codes: An optimal prefix code constructed by a greedy algorithm

Number of bits required to encode a file using code tree T :

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

C : the set of characters

$f(c)$: the frequency of character c

$d_T(c)$: the depth of c in code tree T , i.e., the length of the code

Huffman Codes

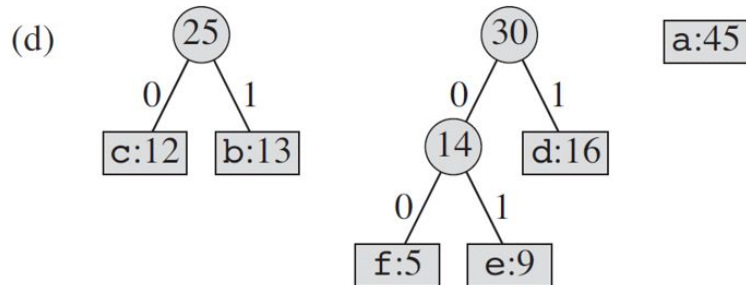
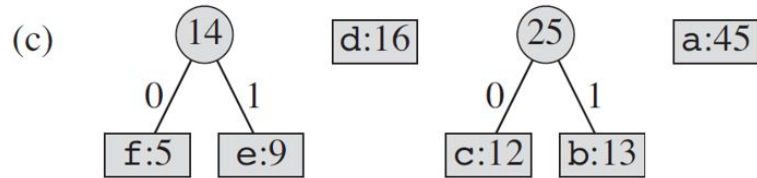
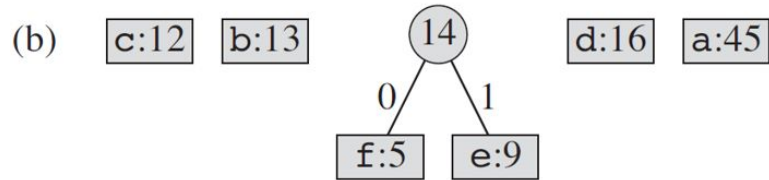
Build the tree in a bottom-up fashion:

1. Start with all leaves maintained by a min-priority queue (a heap), in the ascending of frequencies
2. Pop two nodes A, B (initially all leaf nodes, but eventually including internal nodes) from the heap
3. Merge A, B into a new node with A, B being the left child and right child
4. Repeat step 2 & 3 for $n-1$ times where n is the number of characters

```
n = len(heap)
heapq.heapify(heap)
for _ in range(n-1):
    lval, lnode = heapq.heappop(heap)
    rval, rnode = heapq.heappop(heap)
    nnode = Node(lnode.value + rnode.value,
                  lval + rval, lnode, rnode)
    lnode.parent, rnode.parent = nnode, nnode
    heapq.heappush(heap, (lval + rval, nnode))
_, root = heapq.heappop(heap)
root.traverse('')
```

Example of Huffman Codes

(a) f:5 e:9 c:12 b:13 d:16 a:45



(a) **f** and **e** have the minimum frequencies, they are then merged into a new node "14", where "14" represents the total frequency of the two characters

(b) **c** and **b** are then merged into a new node "25"

(c) An internal node "14" (merged from **f** and **e**) are now merged with **d** to construct the new node "30", which is the root of a binary tree of height 2.

(d) This process repeats until there is only one node left in the heap, that node is the root of the final binary tree constructed by Huffman codes.

Analysis

Efficiency:

Using heap, the time complexity is **$O(n \log n)$**

Correctness:

1. Optimal substructure
2. Greedy choice property

What is the greedy choice property here?

- Cost of tree is cost of all mergers
- Greedy choice is merger of least cost, i.e., lowest frequency character at the time

Optimal Substructure

What is a substructure here?

- $C = \{\mathbf{a}:45, \mathbf{b}:13, \mathbf{c}:12, \mathbf{d}:16, \mathbf{e}:9, \mathbf{f}:5\}$
- $C' = \{\mathbf{a}:45, \mathbf{b}:13, \mathbf{c}:12, \mathbf{d}:16, \mathbf{g}:14\}$

Let T be the optimal prefix code for C , and T' be the optimal prefix code for C' , which is constructed by 1) removing the two characters with lowest frequencies and share the same parent, and 2) adding a new character with frequency being the sum of the two removed.

In this case, $C' = C \setminus \{\mathbf{e}, \mathbf{f}\} \cup \{\mathbf{g}\}$

Want to prove:

If T is optimal for C , T' is optimal for C' .

Prove by Contradiction:

$$B(T) = B(T') + f(\mathbf{e}) + f(\mathbf{f}) \text{ since } f(\mathbf{e})D_T(\mathbf{e}) + f(\mathbf{f})D_T(\mathbf{f}) = f(\mathbf{e})[D_{T'}(\mathbf{g})+1] + f(\mathbf{f})[D_{T'}(\mathbf{g})+1] = f(\mathbf{g})D_{T'}(\mathbf{g}) + f(\mathbf{e}) + f(\mathbf{f})$$

Suppose T' is not optimal, then there exists a T'' such that $B(T'') < B(T')$. Since \mathbf{g} is a leaf in T'' , by adding \mathbf{e} and \mathbf{f} as children of \mathbf{g} in T'' , the prefix tree for C is $B(T'') + f(\mathbf{e}) + f(\mathbf{f}) < B(T') + f(\mathbf{e}) + f(\mathbf{f}) = B(T) \Rightarrow$ Contradicts $B(T)$ is optimal for C .

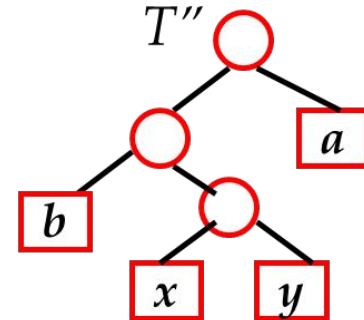
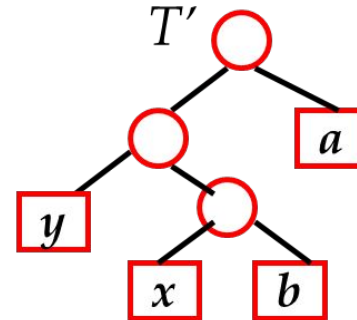
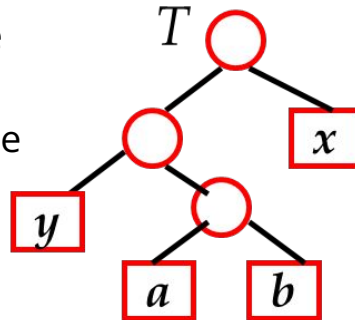
Therefore, optimal solution for the character set C implies optimal solutions for C' , which is a sub-structure of C .

Greedy Choice Property

Let \mathbf{x} and \mathbf{y} be two lowest frequency characters, greedy choice property says there exists an optimal prefix code for C , where \mathbf{x} and \mathbf{y} appear as sibling leaves of highest depth in the code tree (i.e. they have the same code length and differ in only the last bit)

Prove by Contradiction:

Assume there is no optimal code tree with \mathbf{x} and \mathbf{y} at the deepest level, instead, in this optimal code tree T , \mathbf{a} and \mathbf{b} are siblings at the deepest level, \mathbf{x} and \mathbf{y} are in some arbitrary positions in T



Assume WLOG that $f(\mathbf{x}) \leq f(\mathbf{y})$ and $f(\mathbf{a}) \leq f(\mathbf{b})$

Exchange \mathbf{x} with \mathbf{a} (and \mathbf{y} with \mathbf{b})

Then $B(T'') \leq B(T') \leq B(T)$, so T'' must be an optimal code tree with \mathbf{x} and \mathbf{y} at the deepest level \Rightarrow Contradicts that there is no optimal code tree with \mathbf{x} and \mathbf{y} at the deepest level.

Exercise 5

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

Summary

	Dynamic Programming	Greedy Algorithm
Optimal substructure	Yes	Yes
Overlapping subproblems	Yes, otherwise no advantage	Yes, but directly skips many
Greedy choice property	Not needed	Yes
Applicability	More applicable	Less applicable due to the limitation of greedy choice