

Greedy Algorithms

CHANG Yi-Jun Steven HALIM PRASHANT Nalini Vasudevan

School of Computing
National University of Singapore

CS3230 Lec07; Tue, 30 Sep 2025

Overview

Introduction

Fractional Knapsack

Huffman Code

LeetCode Demo

Wrapping-Up

Dynamic Programming (DP) Paradigm, Revisited

- ▶ Expressing the solution recursively
- ▶ There are only a small (e.g., poly) number of subproblems
- ▶ But there is a (huge) overlap among the subproblems
- ▶ So the recursive algorithm takes exponential time (solving same subproblem multiple times)
- ▶ So we compute the top-down recursion with memoization or compute it iteratively in a bottom-up fashion
- ▶ This avoids wastage of computation and leads to an efficient implementation

Today, Greedy Algorithms

- ▶ This is also a very general problem-solving technique, like complete search (brute force), Divide and Conquer (D&C), and Dynamic Programming (DP)
- ▶ The technique is to recast the problem so that only one subproblem needs to be solved at each step
- ▶ Greedy algorithm beats complete search, D&C, and DP, **when it works***

*Many times it does **not** work...

Knowing when greedy is applicable for a given computational problem is the key skill

Fractional Knapsack (1)

Input: $(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n), W$
where (w_i, v_i) is the weight and value of item i
and W is the capacity of the bag (knapsack)

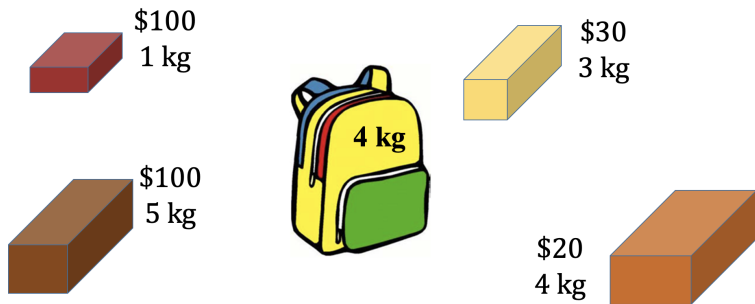
Output: Weights x_1, x_2, \dots, x_n that maximize $\sum_i v_i \cdot \frac{x_i}{w_i}$, subject to:

$$\sum_i x_i \leq W$$

$$0 \leq x_j \leq w_j, \forall j \in [n]$$

Compared with the integral, 0/1 knapsack version from DP lecture, in fractional knapsack, we can take a fraction of each item

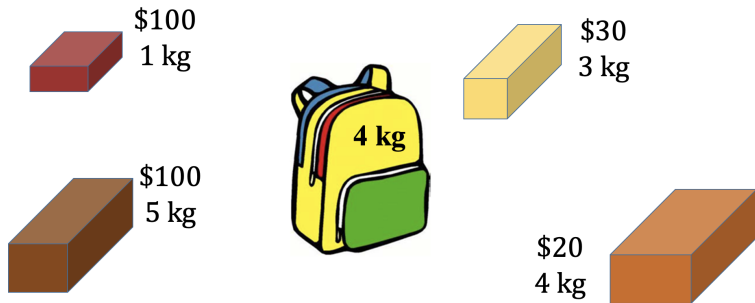
Fractional Knapsack (2)



0/1 optimal sol: 1 kg of \$100 (top-left) + 3 kg of \$30 (top-right)
= 100\$ + 30\$ = 130\$

Can we do better if we can take fraction of each item?

Fractional Knapsack (3)



Fractional optimal sol: 1 kg of \$100 (top-left) + $\frac{3}{5}$ kg of \$100 (bottom-left) = 100\$ + 60\$ = 160\$

Optimal Substructure

If we remove y **kg of one item j** from the optimal knapsack, then the remaining load must be the optimal knapsack weighing **at most $W - y$ kg** that one can take from the $n - 1$ **other items and $w_j - y$ kg of item j**

Optimal Substructure: Proof (cut-and-paste argument)

Let X be the value of the optimal knapsack

Suppose that the remaining load after removing y kg of item j **was not** the optimal knapsack weighing at most $W - y$ kg that one can take from the $n - 1$ other items and $w_j - y$ kg of item j

This means that there is other knapsack of value $> X - v_j \cdot \frac{y}{w_j}$ with weight $\leq W - y$ kg among the $n - 1$ other items and $w_j - y$ kg of item j

If we combine this other knapsack with y kg of item j , we will have a 'more optimal knapsack' of value $> X$ with weight $\leq W$...

This is absurd, we said X is the optimal knapsack value...

Contradiction!

So, the optimal substructure must be optimal

Dynamic Programming?

In the integral (0/1) knapsack problem, we used the optimal substructure to formulate DP for deciding whether to add item j (in 0/1 fashion (only two choices), not fractionally...)

Then we use $O(n \cdot W)$ bottom-up (or top-down with memoization)

But for fractional knapsack problem, we can do better...

Despite item j can be possibly be taken fractionally

Question 1 at VisuAlgo Online Quiz

For the given fractional knapsack, what strategy will you use?

- A. First, take the item with the **maximum value**, then the item with the **second maximum value**, and so on, until we exceed W (the last chosen item could be fractional)
- B. First, take the item with the **minimum weight**, then the item with the **second minimum weight**, and so on, until we exceed W (the last chosen item could be fractional)
- C. First, take the item with the **maximum value/weight**, then the item with the **second maximum value/weight**, and so on, until we exceed W (the last chosen item could be fractional)

Greedy-Choice Property

Let j^* be the item with the **maximum value/weight**: $\frac{v_j}{w_j}$

Then, there exists an optimal knapsack containing $\min(w_{j^*}, W)$ kg of item j^*

Why? Let's prove this with an 'exchange argument':

Suppose an optimal knapsack contains y_{j^*} of item j^*

From the rest of the knapsack, pick x_1 of item 1, x_2 of item 2, \dots , x_n of item n such that $x_1 + x_2 + \dots + x_n = \min(w_{j^*}, W) - y_{j^*}$

We can replace **these weights** by $\min(w_{j^*}, W) - y_{j^*}$ kg of item j^*

Total weight does not change (obviously) and total value does not decrease⁺ as $\frac{v_j}{w_j}$ of j^* is the maximum, so, knapsack stays optimal, and it is 'safe' to use this greedy-choice

Strategy for Greedy Algorithm

Use greedy-choice: put $\min(w_{j^*}, W)$ kg of item j^* in knapsack

If knapsack now weighs W kg, we are done

Otherwise, use optimal substructure to solve subproblem where all of item j^* is removed (as we use the entirety of item j^*) and the knapsack weight limit is now set to $W - w_{j^*}$

Pseudo-Code

```
Iter-Frac-Knapsack(v, w, W): # Overall,  $O(n \log n)$ 
....value_per_kg <- [1, 2, ..., n]
....sort value_per_kg in  $O(n \log n)$  time,
.....with a custom comparison function, such that
.....i <= j iff  $v[i]/w[i] \leq v[j]/w[j]$  or
..... $v[i]*w[j] \leq v[j]*w[i]$  # cool strat
....for i = n down to 1: #  $O(n)$ , largest to smallest
.....if W == 0: break
.....j_star <- value_per_kg[i]
.....k <- min(w[j_star], W)
.....print "Take {k} kg of item {j_star}"
.....W <- W-k
```

Paradigm for Greedy Algorithm

1. Cast the problem where we have to make a greedy choice and are left with **just one** subproblem to solve
2. Prove (usually using exchange argument) that there is always an optimal solution to the original problem **that makes the greedy choice**, so the greedy choice is safe
3. Use optimal substructure (usually using proof by contradiction; cut-and-paste argument) to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem

PS: You probably have seen other greedy algorithms before, e.g., Dijkstra's algorithm (for weighted SSSP) or Prim's/Kruskal's algorithm (for MST), etc

Question 2 at VisuAlgo Online Quiz

Who is the master of algorithms pictured below?



- A). Joseph Kruskal
- B). David A. Huffman
- C). Edsger W. Dijkstra
- D). Robert C. Prim

Huffman Code

With applications in data compression...

Binary Coding

Given an alphabet set $A : \{a_1, a_2, \dots, a_n\}$

1. Question (Q): How many bits are needed to encode a text file F containing a sequence of m alphabets drawn from set A if we use fixed-length coding?

Answer (A): $m \cdot \lceil \log_2 n \rceil$ bits

Fixed-Length Coding (1)

Given an alphabet set $A : \{a_1, a_2, \dots, a_n\}$

2. Q: What is a binary coding of A ?

A: A function gamma $\gamma : A \rightarrow$ binary strings

3. Q: What is a **fixed-length** coding of A ?

A: Each alphabet is encoded with a unique binary string of fixed-length $\lceil \log_2 n \rceil$

4. Q: How to **decode** a fixed-length binary coding?

A: Easy, suppose each alphabet has fixed-length of 4 bits, we will interpret

0100101000001011... as

0100|1010|0000|1011|...

Fixed-Length Coding (2)

Given an alphabet set $A : \{a_1, a_2, \dots, a_n\}$

5. Q: Can we use **fewer** bits to store alphabet set A ?

A: No

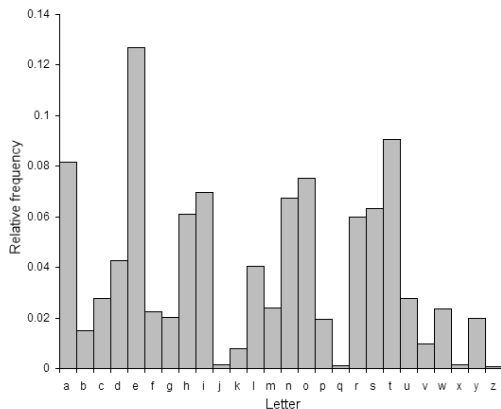
6. Q: Can we use fewer bits to store a **file**?

A: Yes

Letter Frequency Variation (1)

There is a **huge** variation in the frequency of alphabets in an (English) text, source:

<https://mathcenter.oxford.emory.edu/site/math125/englishLetterFreqs/>



Letter Frequency Variation (2)

1. Q: How to exploit variation in the frequencies of alphabets?

A: the 'greedy sense' or 'intuition' are:

More frequent alphabets → coding with **shorter bit string**

Less frequent alphabets → coding with *longer bit string*

Variable-Length Encoding (1)

Average Bit Length (ABL) per symbol using γ :

Alphabet	freq f	encoding γ
a	0.45	0
b	0.18	11
c	0.15	110
d	0.12	101
e	0.10	100

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$

$$= 0.45 \cdot 1 + 0.18 \cdot 2 + (0.15 + 0.12 + 0.10) \cdot 3 = 1.92$$

(smaller than $\lceil \log_2 5 \rceil = 3$ bits)

But there is a serious problem with the γ encoding...

Can you see the issue?

Variable-Length Encoding (2)

2. Q: How will you decode 0110101?

A: Depends,
“abad” 0|11|0|101 or
“acd” 0|110|101 :(

3. Q: What is the source of this ambiguity?

A: $\gamma(b)$ is a prefix of $\gamma(c)$...
Can you fix it?

Alphabet	freq f	encoding γ
a	0.45	0
b	0.18	11
c	0.15	110
d	0.12	101
e	0.10	100

Variable-Length Encoding (3)

Now

Alphabet	freq f	encoding γ	
a	0.45	0	$ABL(\gamma) = \sum_{x \in A} f(x) \cdot \gamma(x) $ $= 0.45 \cdot 1 + (0.18 + 0.15 + 0.12 + 0.10) \cdot 3 = 2.10$ <p>This is a slightly larger <i>ABL</i> (2.10 vs 1.92, but there is no more ambiguity, and is still less than 3 bits)</p>
b	0.18	111	
c	0.15	110	
d	0.12	101	
e	0.10	100	

Prefix Coding

Definition: A coding $\gamma(A)$ is called **prefix coding** if there does not exist $x, y \in A$ such that $\gamma(x)$ is a **prefix** of $\gamma(y)$

Algorithmic Problem: Given a set A of n alphabets and their (non-uniform) frequencies, compute coding γ such that γ is a prefix coding **and $ABL(\gamma)$ is the minimum**

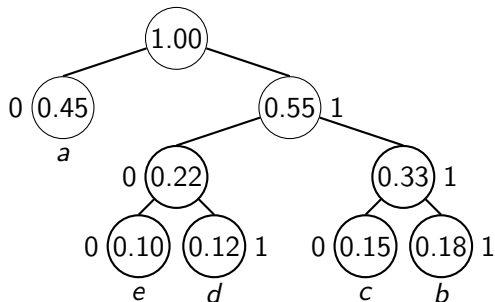
The Challenge

Among all possible prefix-free binary coding of A
(there are a lot of possibilities),
how to find the optimal prefix coding, efficiently?

The Novel Idea of Huffman

Binary Coding \Leftrightarrow Binary Tree

A Labeled Binary Tree



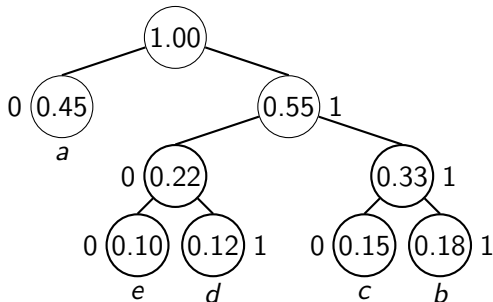
Leaf vertices are the alphabets

Label of path from root to leaf is the binary code of the alphabet

Examples from that picture: '0' is 'a', '100' is 'e', ..., '111' is 'b'

How to Build the Labeled Binary Tree?

Alphabet	freq f	encoding γ
a	0.45	0
b	0.18	111
c	0.15	110
d	0.12	101
e	0.10	100



Prefix Code and Labeled Binary Tree

Theorem:

For each prefix code of a set A of n alphabets, there exists a binary tree T on n leaves such that:

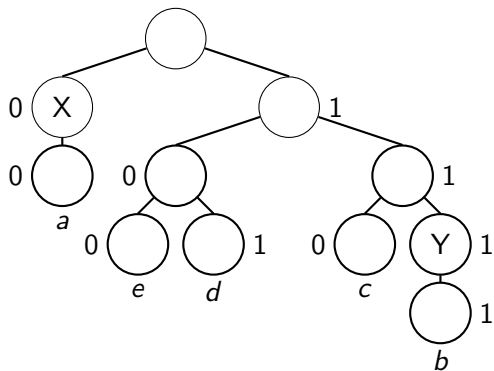
- ▶ There is a bijjective (one to one) mapping between the **alphabets** and the **leaves**
- ▶ The **label of a path from root to a leaf** vertex corresponds to the **prefix code** of the corresponding alphabet (at leaf)

Now we can express **ABL** of γ in terms of its binary tree T :

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)| = \sum_{x \in A} f(x) \cdot \text{depth}_T(x)$$

Observation 1

Q: Is the following prefix coding optimal?



A: No, vertex X and Y (just one child) are not necessary

The binary tree corresponding to optimal prefix coding must be a **full binary tree** (internal vertex has degree **2**)

The Influence of Frequencies (1)

Now we need to see the influence of (non-uniform) frequencies on an optimal binary tree

Let a_1, a_2, \dots, a_n be the alphabets of A in **non-decreasing** order of their **frequencies**, i.e., a_1 is the least frequent alphabet

Intuitively, **more frequent** alphabets should be **closer to the root** and *less frequent* alphabets should be *farther from the root* – this is the idea that came to Huffman in 1952

The Influence of Frequencies (2)

But how to organize them in order to achieve optimal prefix code?

We shall now make some simple observations about the structure of the binary tree corresponding to the optimal prefix codes

These observations will be about some local structure in the tree

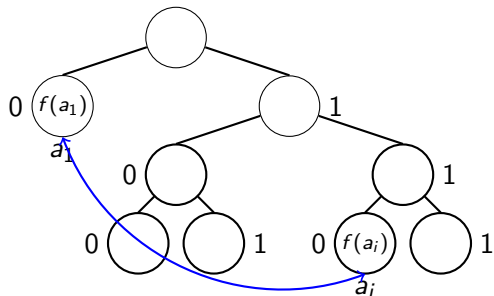
Nevertheless, these observations will play a crucial role in the design of a binary tree with optimal prefix code for given A

HEAVY SLIDES SOON:

Please pay full attention on the next few slides!!

Observation 2

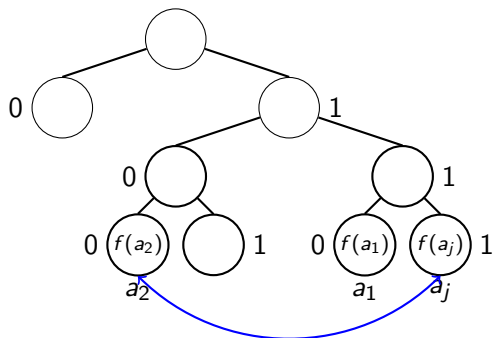
Q: Can a_1 (the least frequent alphabet) be present at a **higher level** than the last (deepest) level?



A: No, swapping a_1 with a_i cannot increase **ABL**, and a vertex at deepest level must be a leaf vertex

Observation 3

Q: What about a_2 (the second least frequent alphabet)?



A: The sibling of a_1 can be a_2

Observation 3 - More Details

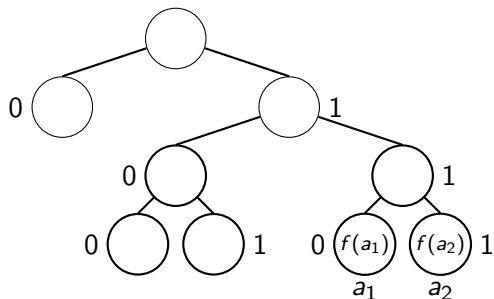
Lemma: There exists an optimal prefix coding in which a_1 and a_2 **appear as siblings** in the corresponding labeled binary tree

Important note: It is **inaccurate** to claim that “in every optimal prefix coding, a_1 and a_2 always appear as siblings in the labeled binary string”

But the algorithmic implication of the Lemma above is important as it allows us to focus on constructing a binary tree of the optimal coding in which a_1 and a_2 appear as siblings – Huffman code

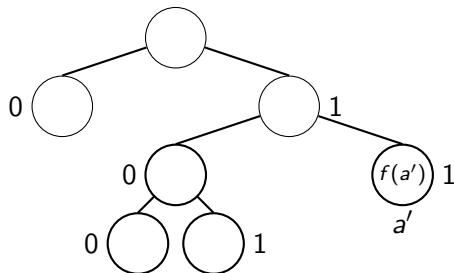
Observation 4 (1)

Observe a_1 and a_2



Observation 4 (2)

Observe a_1 and a_2 are merged into a' and $f(a') = f(a_1) + f(a_2)$



Key Idea of Huffman Algorithm (1)

Let $A = \{a_1, a_2, a_3, \dots, a_n\}$ be n alphabets in non-decreasing order of frequencies

After merging a_1 and a_2 , we (may⁺) have:

$A' = \{a_3, \dots, a', \dots, a_n\}$ be $n - 1$ alphabets in non-decreasing order of frequencies with $f(a') = f(a_1) + f(a_2)$

⁺We do not know the actual position of a' in the sorted A'

Is this greedy-choice correct?

Does an optimal prefix code of A' implies optimal prefix code of A ?

Key Idea of Huffman Algorithm (2)

Let's define:

- ▶ $OPT_{ABL}(A)$: The minimum **ABL** value over all prefix code/labeled binary tree for alphabet A
- ▶ $OPT(A)$: An optimal prefix code/labeled binary tree for alphabet A with **ABL** value $OPT_{ABL}(A)$

We claim that $OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$

If this claim is true, then the following algorithm is optimal

Pseudo-Code

```
OPT(A) # Overall,  $O(n \log n)$ , Huffman Algorithm
....if  $|A| = 2$ , return () # PS:  $|A|$  is always  $\geq 2$ 
..... / \
..... (0) a1 a2 (1)
....else
.....a1 and a2 are the two least frequency alphabets
.....remove a1 and a2 from A # 2x extract min heap
.....create a new alphabet a'
..... $f(a') \leftarrow f(a1)+f(a2)$ 
.....insert a' into A # insert into min heap
.....T  $\leftarrow$  OPT(A) # recurse
.....replace vertex a' in T by.... ()
..... / \
..... (0) a1 a2 (1)
.....return T
```

How to Prove?

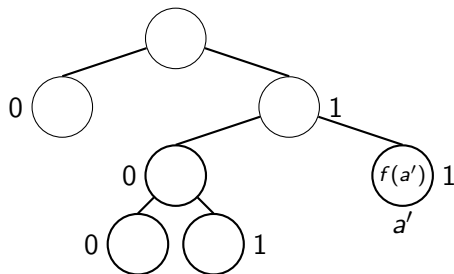
Prove $OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$!

Two parts:

1. Derive a prefix coding for A from $OPT(A')$!
2. Derive a prefix coding for A' from $OPT(A)$!

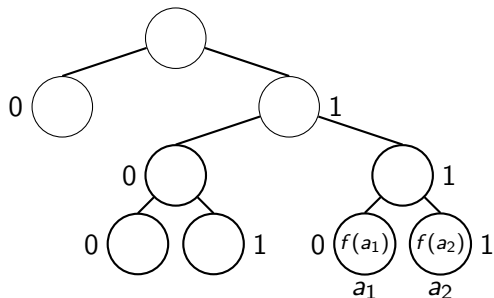
A prefix coding for A from $OPT(A')$ (1)

Let the following T' be the optimal binary tree corresponding to $OPT_{ABL}(A')$



A prefix coding for A from $OPT(A')$ (2)

Let the following T' be the optimal binary tree corresponding to $OPT_{ABL}(A')$



This gives a prefix coding for A with **ABL** = ...

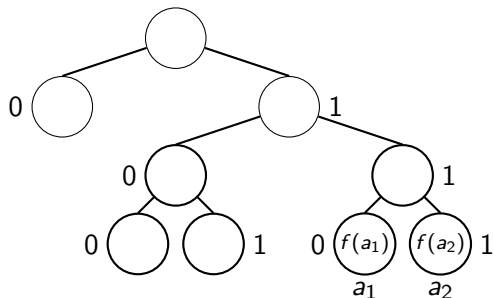
Question 3 at VisuAlgo Online Quiz

Express **ABL** in terms of $OPT_{ABL}(A')$, $f(a_1)$, and $f(a_2)$

- A. **ABL** = $OPT_{ABL}(A') + f(a_1) + f(a_2)$
- B. **ABL** = $OPT_{ABL}(A')$
- C. **ABL** = $OPT_{ABL}(A') + \max(f(a_1) + f(a_2))$
- D. **ABL** = $OPT_{ABL}(A') - f(a_1) - f(a_2)$

A prefix coding for A from $OPT(A')$ (3)

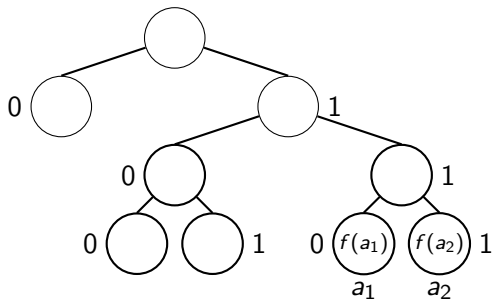
Let the following T' be the optimal binary tree corresponding to $OPT_{ABL}(A')$



This gives a prefix coding for A with $\mathbf{ABL} =$
 $OPT_{ABL}(A') + f(a_1) + f(a_2)$
 $\Rightarrow OPT_{ABL}(A) \leq OPT_{ABL}(A') + f(a_1) + f(a_2)$
(we have constructively shown \leq)

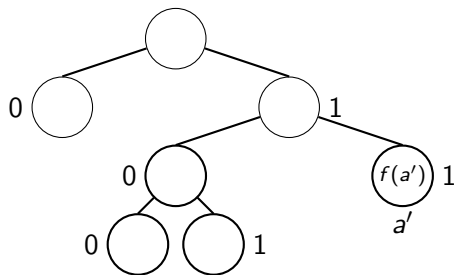
A prefix coding for A' from $OPT(A)$ (1)

Now the other direction: Let the following T be the optimal binary tree corresponding to $OPT_{ABL}(A)$



A prefix coding for A' from $OPT(A)$ (2)

Now the other direction: Let the following T be the optimal binary tree corresponding to $OPT_{ABL}(A)$



This gives a prefix coding for A' with **ABL** = $OPT_{ABL}(A) - f(a_1) - f(a_2)$ (see the previous sketch again)
 $\Rightarrow OPT_{ABL}(A') \leq OPT_{ABL}(A) - f(a_1) - f(a_2)$
(we have constructively shown \leq for this one too)

How to Prove?

Prove $OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$!

We have shown:

1. How to derive a prefix coding for A from $OPT(A')$
 $OPT_{ABL}(A) \leq OPT_{ABL}(A') + f(a_1) + f(a_2)$
2. How to derive a prefix coding for A' from $OPT(A)$
 $OPT_{ABL}(A') \leq OPT_{ABL}(A) - f(a_1) - f(a_2)$ or rearrange to
 $OPT_{ABL}(A) \geq OPT_{ABL}(A') + f(a_1) + f(a_2)$

Combined, we have:

$$OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$$

Huffman algorithm works!

Summary on Proof Techniques (for Greedy)

- ▶ For Greedy Choice
Exchange argument
- ▶ For Optimal Substructure
Proof by contradiction; cut-and-paste argument, or
Constructive proof (as shown in the Huffman Code)

Practice Problems (DP and Greedy)

To succeed in mastering these two topics is to solve as many problems as you can

Try solving exercises from textbooks (e.g., CLRS, CP4 (ADS))

Look for more practice problems over the Internet (leetcode (one demo coming up), Kattis, old: UVa)

Let's solve /assign-cookies

Midterm Test Admin Update

Key information:

- ▶ Venue: MPSH5
- ▶ Date: Sat, 04 Oct 25 (end of Wk7)
- ▶ Time: 2-4pm (120 minutes), start entering MPSH5 by 1.30pm
- ▶ Seating plan is as shown in this link
- ▶ Open Book, Pen and Paper
- ▶ No electronic device except a **non-programmable** calculator
- ▶ 20 MCQs (tricky, as usual) and 3 essays (10 sub-boxes)
- ▶ Marks: $20 \cdot 1.5 + 3 \cdot 10 = 60$, scaled to 30%
- ▶ An essay sub-box, if left blank/**crossed**, gets free 0.5 marks
- ▶ Special needs (a few pax) will have separate venue/invigilator

Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere.

List of credits: Erik D. Demaine, Charles E. Leiserson, Surender Baswana, Leong Hon Wai, Lee Wee Sun, Ken Sung, Arnab Battacharya, Diptarka Chakraborty, Steven Halim, Sanjay Jain, Chang Yi-Jun, Warut Suksompong, Eric Han Liang Wee.