算法设计与分析

Lecture 7: Greedy Algorithms

卢杨

厦门大学信息学院计算机科学系

luyang@xmu.edu.cn

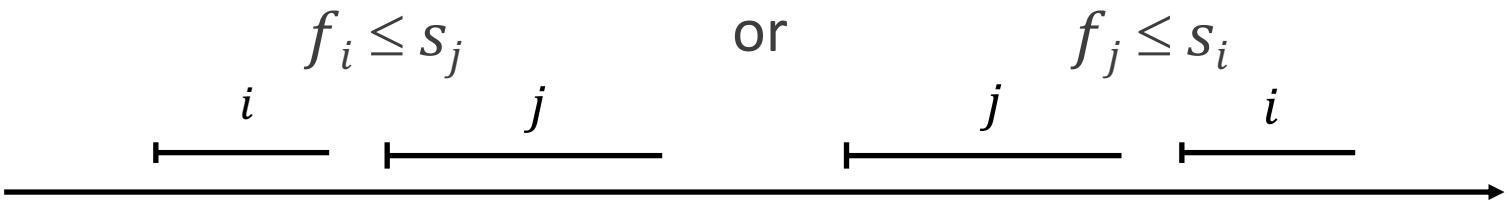
Greedy Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- A greedy algorithm (贪心算法) always makes the choice that looks best at the moment.
- Greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
  - Don't think greedy approach is evil due to its name "greedy" with negative meaning. It often lead to very efficient and simple solution.
- Everyday examples:
  - Playing cards
  - Invest on stocks
  - Choose a university

ACTIVITY SELECTION PROBLEM

Activity Selection Problem

- Schedule n activities $S = \{a_1, \ldots, a_n\}$ that require use of a common resource.
  - We can only do one activity at the same time.
- a_i needs resource during period $[s_i , f_i)$.
  - s_i = start time and f_i = finish time of activity a_i.
  - $0 \leq s_i < f_i < \infty$
- Activities a_i and a_j are compatible (兼容的) if the intervals $[s_i , f_i)$ and $[s_j, f_j)$ do not overlap:



Activity Selection Problem

- Select the largest possible set of mutually compatible (相互兼容) activities.
  - We treat each activity gives same profit.
- For example,

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

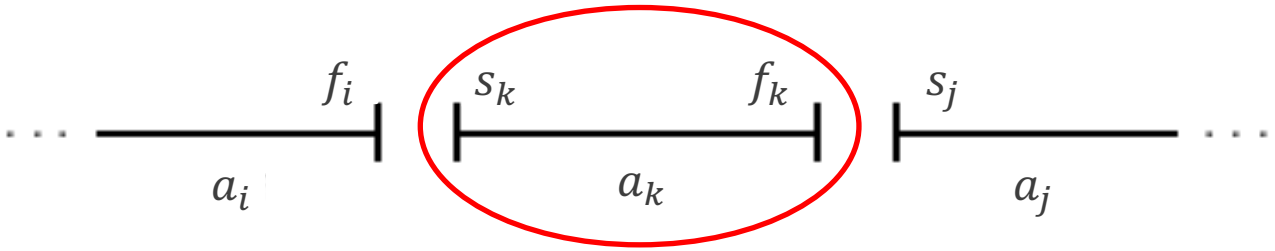
  - Activities are sorted in increasing order of finish times.
  - A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$.
  - Maximal set of mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$.

Representing the Problem

- Define the subproblems:

$$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$$

as activities whose periods are after a_i finishes and before a_j starts.



- Activities that are compatible with the ones in S_{ij}:
  - All activities that finish before f_i.
  - All activities that start no earlier than s_j.

Representing the Problem

- Based on the definition of S_{ij}:

$$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$$

  S_{1n} does not cover the original problem because a_1 and a_n are excluded.

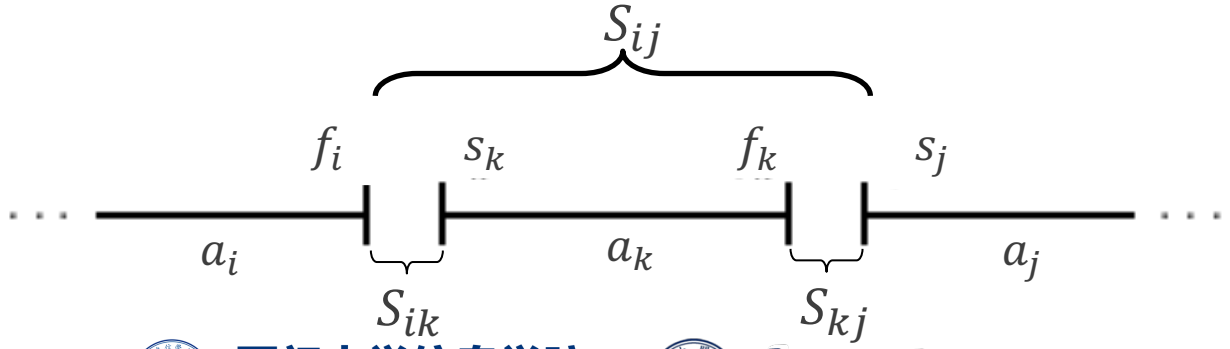
- We can add two fictitious (虚拟的) activities:
  - $a_0 = [-\infty, 0)$;
  - $a_{n+1} = [\infty, \infty + 1)$.
- Thus, $S_{0(n+1)} = S$ covers the entire space of activities $a_1, \ldots, a_n$.
- Assume that activities are sorted in increasing order of finish times

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1},$$

  We only need to consider sets S_{ij} with $0 \leq i < j \leq n + 1$, because $S_{ij} = \emptyset$ if $i > j$.

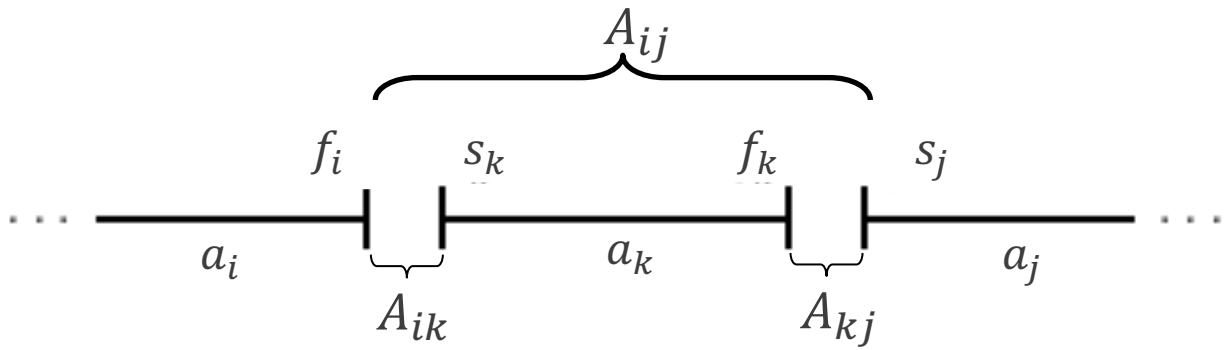
Dynamic Programming Solution

- Recall the optimal substructure of matrix-chain multiplication problem and optimal BST problem, the optimal subproblem has contains a position k.
- In this problem, assume that a solution to the above a subproblem includes activity a_k.
  - Solution to S_{ij} = (Solution to S_{ik}) ∪ $\{a_k\}$ ∪ (Solution to S_{kj}).
  - $|$Solution to $S_{ij}|$ = $|$Solution to $S_{ik}|$ + 1 + $|$Solution to $S_{kj}|$.



Optimal Substructure

- Assume that A_{ij} is an optimal solution to S_{ij} and includes activity a_k. Sets A_{ik} and A_{kj} must also be optimal solutions.
  - Prove by contradiction.



Recursive Equation

- Let $c[i,j] = |A_{ij}|$as the size of the maximum subset of mutually compatible activities in S_{ij}.
- If $S_{ij} = \emptyset$, then $c[i,j] = 0$.
- If $S_{ij} \neq \emptyset$ and if we consider that a_k is used in an optimal solution, we have:

$$c[i,j] = c[i,k] + c[k,j] + 1$$

Recursive Equation

- The recursion equation is:

$$c[i,j] = \begin{cases} 0 & S_{ij} = \emptyset \\ \max\limits_{i<k<j, a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & S_{ij} \neq \emptyset \end{cases}$$

- There are $j - i - 1$ possible values for k.
  - $k = i + 1, \ldots, j - 1$.
  - We check all the values and take the best one.
  - Nothing special, very similar to the matrix-chain multiplication problem and the optimal BST problem.

Simpler Idea

- Is dynamic programming the most efficient to solve this problem?
- How about just simply select the activity with earliest finish time?

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- It seems work. But how to prove the correctness?

Simpler Idea

Theorem

Let $S_{ij} \neq \emptyset$ and a_m be the activity in S_{ij} with the earliest finish time:

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

Then:

(1) Activity a_m must be in some optimal subset A_{ij}.

- i.e. there exist some optimal solutions that contains a_m.

(2) $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} the only nonempty subproblem.

Simpler Idea

Proof of (1) Activity a_m must be in some optimal subset A_{ij}:

- Assume activities in A_{ij} are in increasing order of finish time, and let a_k be the first activity in A_{ij}: $A_{ij} = \{a_k, \ldots\}$.
- If $a_k = a_m$, done!
- Otherwise, replace a_k with a_m (resulting in a set A_{ij}')
  - Because f_m is the earliest finish time, $f_m \leq f_k$. The activities in A_{ij}' will continue to be compatible.
  - A_{ij}' will have the same size with A_{ij}'.

Simpler Idea

Proof of (2) $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} the only nonempty subproblem:

- Assume $S_{im} \neq \emptyset$, i.e. there exists an activity $a_k \in S_{im}$:

$$f_i \leq s_k < f_k \leq s_m < f_m$$

- $f_k < f_m$ contradicts with the fact that a_m has the earliest finish time.
- Therefore, there is no $a_k \in S_{im}$, which implies $S_{im} = \emptyset$.

Why is the Theorem Useful?

- Given the theorem, what can we do and how can we improve?

	Dynamic programming	Using the theorem
Number of subproblems in the optimal solution	2 subproblems: S_{ik}, S_{kj}	1 subproblem: S_{mj} $S_{im} = \emptyset$
Number of choices to consider	$j - i - 1$ choices	1 choice: the activity with the earliest finish time in S_{ij}

- Making the greedy choice (the activity with the earliest finish time in S_{ij})
  - Reduce the number of subproblems and choices.
  - Solve each subproblem in a top-down fashion.

Greedy Approach

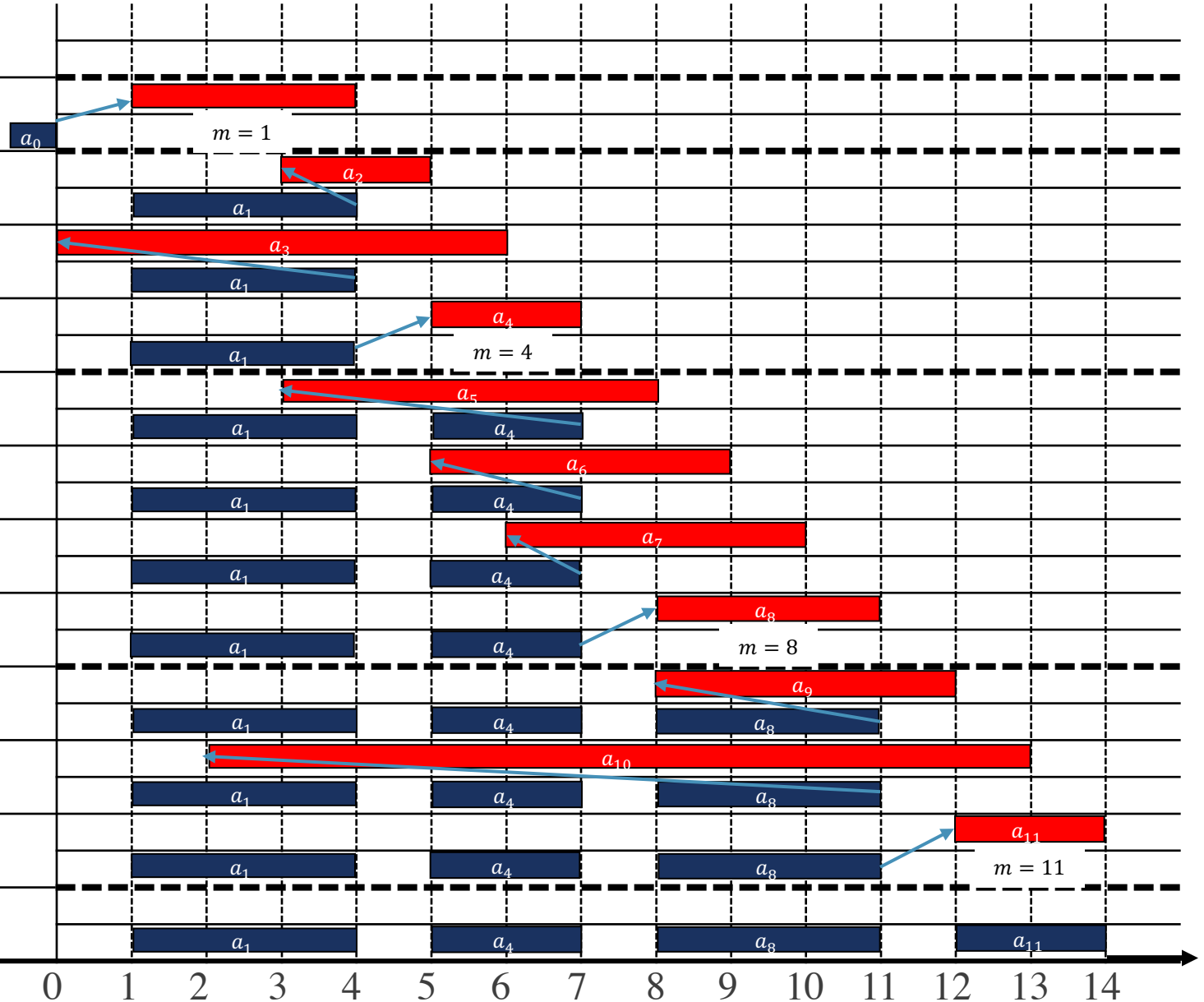
- To select a maximum size subset of mutually compatible activities from set S_{ij}:
  - Choose am $a_m \in S_{ij}$ with earliest finish time (greedy choice).
  - Add a_m to the set of activities used in the optimal solution.
  - Solve the same problem for the set S_{mj}.
- From the theorem, it is proved that by choosing a_m we are guaranteed to have used an activity included in an optimal solution
  - We do not need to solve the subproblem S_{mj} before making the choice!
- This problem has the greedy choice property (贪心选择性质).

A Recursive Greedy Algorithm

- Activities are ordered in increasing order of finish time.
- Running time: $O(n)$
  - Each activity is examined only once.
- Initial call: RecursiveTaskSelect$(s, f, 0, n+1)$

```
RecursiveTaskSelect(s, f, i, j)
1 m ← i + 1
2 while m < j and s_m < f_i do
3       m ← m + 1
4 if m < j then return {a_m} ∪
        RecursiveTaskSelect(s, f, m, j)
5 else return ∅
```

k	s_k	f_k
0	-	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	-



An Iterative Greedy Algorithm

- It totally not necessary to use recursion.
- Activities are ordered in increasing order of finish time.
- Running time: $O(n)$
  - Each activity is examined only once.

```
GreedyTaskSelect(s, f)
1  A ← {a_1}
2  i ← 1
3  for m ← 2 to n do
4      if s_m ≥ f_i then
5          A ← A ∪ {a_m}
6              i ← m
7  return A
```

Designing Greedy Algorithms

1. Cast the optimization problem as: we make a choice and are left with only one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem when making the greedy choice.
   - Making the greedy choice is always safe.
3. Demonstrate that after making the greedy choice: an optimal solution = the greedy choice + an optimal solution to the resulting subproblem.

Correctness of Greedy Algorithms

- Greedy choice property
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal substructure property
  - We know that we have arrived at a subproblem by making a greedy choice.
  - optimal solution for the original problem = optimal solution to subproblem + greedy choice.

Correctness of Greedy Algorithm for Activity Selection

- Greedy choice property
  - There exists an optimal solution that includes the greedy choice: The activity a_m with the earliest finish time in S_{ij}.
- Optimal substructure property
  - An optimal solution to subproblem S_{ij} = selecting activity a_m + optimal solution to subproblem S_{mj}.

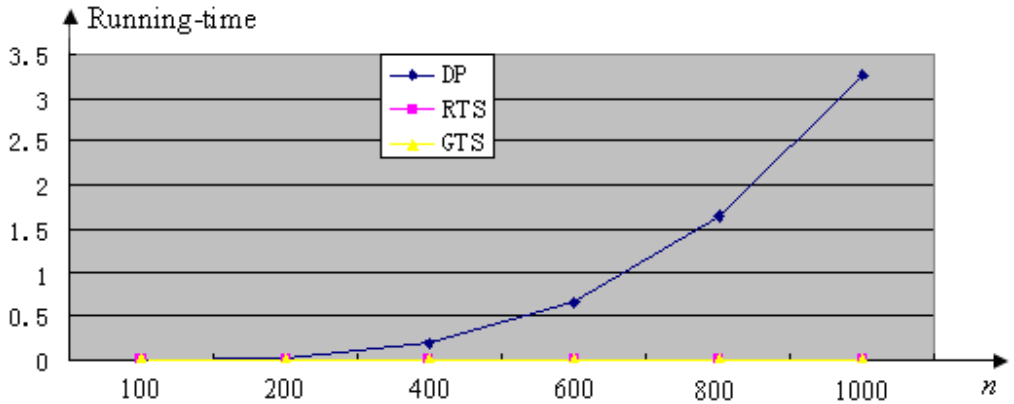
Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - Make a choice at each step.
  - The choice depends on solutions to subproblems.
  - Bottom up solution, from smaller to larger subproblems.
- Greedy algorithm
  - Make the greedy choice.
  - Solve the subproblem arising after the choice is made.
  - The choice we make may depend on previous choices, but not on solutions to subproblems.
  - Top down solution, problems decrease in size.
- Common: Optimal substructure

Experiment for Activity Selection Problem

Table 7.1 Running-time comparison of algorithms for activity selection problem

n	100	200	400	600	800	1000
Dynamic programming(DP)	0.000	0.015	0.189	0.656	1.656	3.250
RecursiveTaskSelect(RTS)	0.000	0.000	0.000	0.000	0.000	0.000
GreeyTaskSelect(GTS)	0.000	0.000	0.000	0.000	0.000	0.000
Maximum task number	19	24	30	40	46	52



厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学 计算机科学系 Computer Science Department of Xiamen University

Image source: 图7.10, 张德富, 算法设计与分析, 国防工业出版社, 2009.

Classroom Exercise

Use greedy algorithm to solve the following problem:

- Given n integers, concatenate them in a row to constitute a maximum integer.
- For example:
  - $n = 3$, 34331213 is the maximum integer to concatenate 13, 312, 343.
  - $n = 4$, 7424613 is the maximum integer to concatenate 7, 13, 4, 246.

Classroom Exercise

Solution 1:

- Simply select the integer with largest first bits.
- How about the following cases:
  - 12, 121
  - 12, 123

Solution 2:

- For the numbers with the same first bits, compare $a + b$ with $b + a$ and use the one with maximum value.

FRACTIONAL KNAPSACK PROBLEM

Fractional Knapsack Problem

Fractional knapsack (部分背包) problem:

- There are n items: the ith item is worth v_i dollars and weights w_i kg.
- The capacity of knapsack is W kg.
- ~~Items must be taken entirely or left behind.~~
- Items can be taken fractionally.
- Which item fractions should we take to maximize the total value?

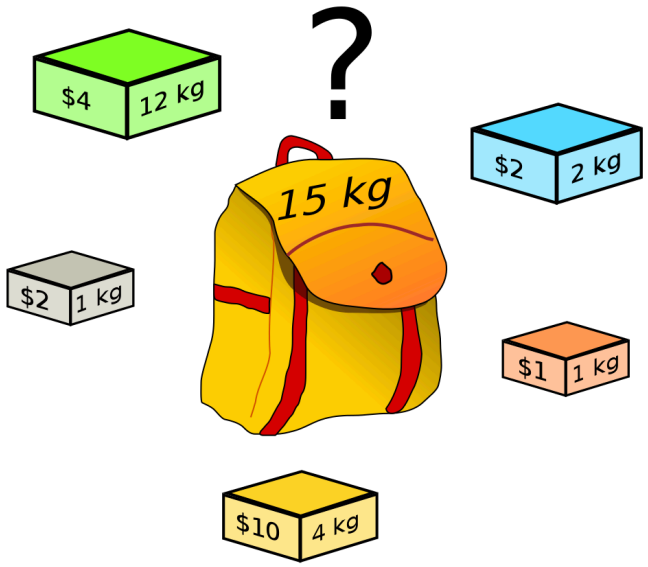


Image source: https://en.wikipedia.org/wiki/Knapsack_problem

Example

- Weight capacity $W = 50\text{kg}$.
- Using the solution of 0/1 knapsack problem, we choose item 2 and 3 with total value $100 + 120 = 220$.
- However, the solution of fractional knapsack problem is to choose item 1 and 2 plus 2/3 of item 3, with total value $60 + 100 + 120 \times 2/3 = 240$.

i	v_i	w_i
1	\$60	10kg
2	\$100	20kg
3	\$120	30kg

Greedy Strategy

$$W = 50\text{kg}$$

- Greedy strategy 1: Pick the item with the maximum value.
- This greedy strategy is obviously not optimal.

i	v_i	w_i
1	\$60	10kg
2	\$100	20kg
3	\$120	30kg

Greedy Strategy

- Greedy strategy 2: Pick the item with the maximum value per kg v_i/w_i.
- If the supply of that element is exhausted and we can carry more, take as much as possible from the item with the next greatest value per kg.
- It is good to order items based on their value per kg:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

$$W = 50\text{kg}$$

i	v_i	w_i	v_i/w_i
1	\$60	10kg	\$6/kg
2	\$100	20kg	\$5/kg
3	\$120	30kg	\$4/kg

It seems ok! But how to prove?

Pseudocode

```
Greedy2Knapsack(W, v)
1  order items based on their value per kg: v_1/w_1 ≥ v_2/w_2 ≥ ··· ≥ v_n/w_n
2  i ← 1; w ← W
3    while w > 0 and as long as there are items remaining do
4        x_i ← min{1, w/w_i}
5        remove item i from list
6        w ← w − x_i w_i
7        i ← i + 1
```

Running time: $\Theta(n)$ if items already ordered; else $\Theta(nlgn)$

Greedy Choice Property

- Now we need to prove that the fractional knapsack problem has greedy choice property.
- Assume the items are ordered based on their value per kg:

$$1 \;\; 2 \;\; 3 \;\; \ldots \;\; i \;\; \ldots \;\; n$$

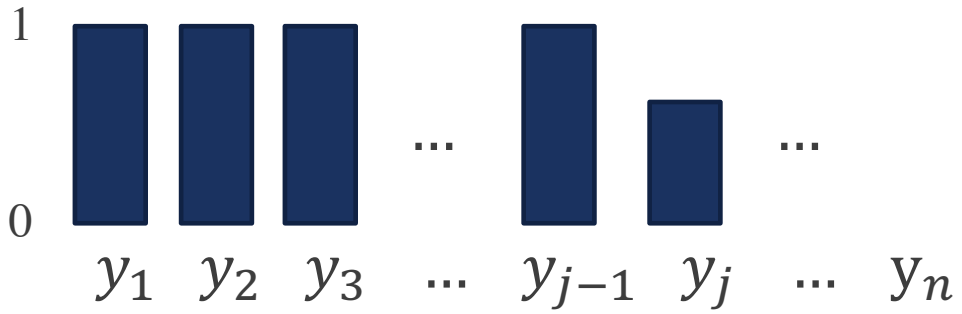
- The greedy solution is

$$x_1 \;\; x_2 \;\; x_3 \;\; \ldots \;\; x_i \;\; \ldots \;\; x_n$$

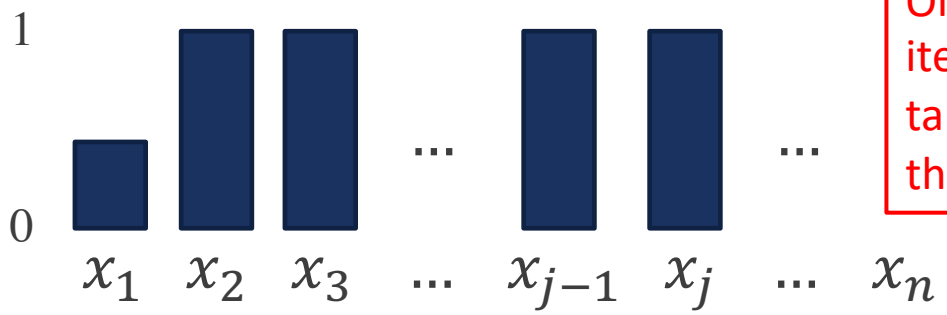
where $x_i \in [0,1]$ is the fraction to take item i.

Greedy Choice Property

- The greedy solution will always look like this:



- Now, if the greedy solution is not optimal, there must exists an optimal solution that looks like this:

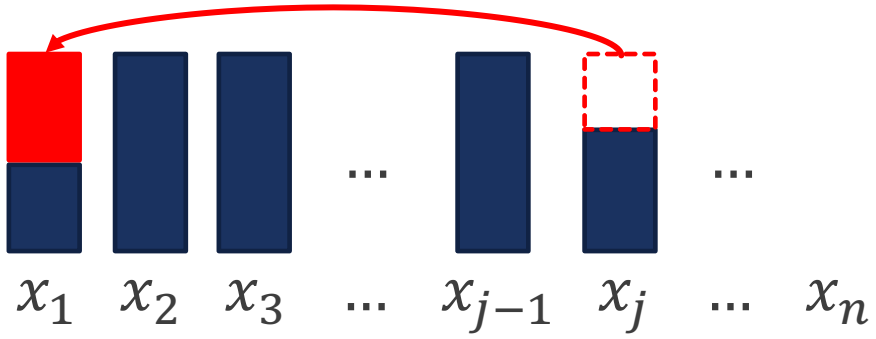


One of the first $j-1$ items is not fully taken, simply assume that it is the first item.

Greedy Choice Property

- Now, we can do some transformation to the optimal solution: Increase the fraction of item 1 and by decreasing item j.
  - Moving weight: $(y_1 - x_1)w_1$.
  - Value increased for item 1: $(y_1 - x_1)w_1 \times v_1/w_1$.
  - Value decreased for item j: $(y_1 - x_1)w_1 \times v_j/w_j$.

Which one is larger?



Greedy Choice Property

- Therefore, given any solution, we can transform it to the greedy solution for larger value.
- It means the greedy solution has the largest value.

Optimal Substructure

- Consider the most valuable load that weights at most W kg.
- If we remove a weight w of item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from the remaining $n - 1$ items plus $w_j - w$ pounds of item j.

Classroom Exercise

Can greedy algorithm obtain optimal solution for the coin change problem?

Classroom Exercise

The greedy algorithm:

1. Select the largest coin.
2. Check if adding the coin makes the change exceed the amount.
   a. No, add the coin.
   b. Yes, set the largest coin as the second largest coin and go back to step 1.
3. Check if the total value of the change equals the amount.
   a. No, go back to step 1.
   b. Yes, problem solved.

Classroom Exercise

- Successful example:
  - For $N = 86$ (cents) and $d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 25, d_6 = 50, d_7 = 100$.
  - The greedy approach is optimal: 50, 25, 10, 1.
- Failed example:
  - For $N = 6$ (cents) and $d_1 = 1, d_2 = 3, d_3 = 4$.
  - The greedy approach is not optimal: 4, 1, 1.
  - The optimal solution: 3, 3.
- For this problem, the success of greedy approach depends on the coin currency.
  - It works for canonical coin systems like US, but not for arbitrary coin system.

HUFFMAN CODE

Data Compression by Binary Code

- The problem of data compression (数据压缩) is to find an efficient method for encoding a data file.
  - Compress string S into S', which can be restored to S, such that $|S'| < |S|$.
- A common way to represent a file is to use a binary code (二进制编码).
- In such a code, each character is represented by a unique binary string, called the codeword (码字).

Data Compression by Binary Code

There are two ways to use binary code:

- A fixed-length code (固定长度编码) represents each character using the same number of bits.
  - For example, suppose our character set is $\{a, b, c\}$.
  - Then we could use 2 bits to code each character: a: 00, b: 01, c: 11.
  - Given this code, if our file is $ababcbbbc$, our encoding will be 000100011101010111.
- We can obtain a more efficient coding using a variable-length code (可变长度编码).
  - Such a code can represent different characters using different numbers of bits.

Data Compression by Binary Code

- For example, a data file of 100 characters contains only the characters a-f, with the frequencies:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- How much do we save in this case?
  - Fixed-length code: $100 \times 3 = 300$ bits.
  - Variable-length code: $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) = 224$ bits

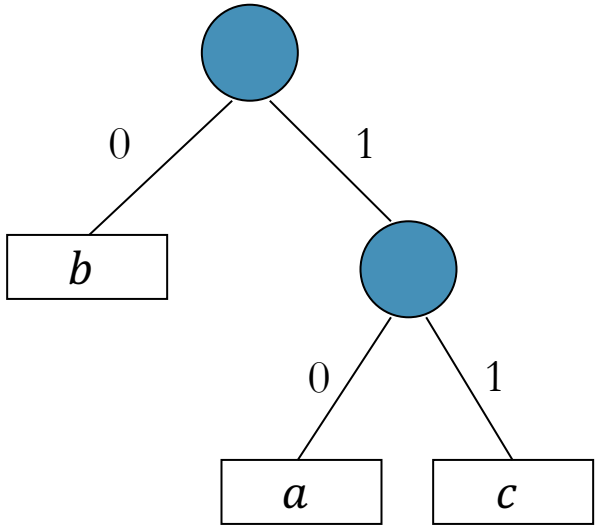
Prefix Codes

- One particular type of variable-length code is a prefix code (前缀编码).
- No codeword is also a prefix of some other codeword.
- The advantage of a prefix code is that there is no ambiguity when interpreting the codes.
- For example:
  - $abc \rightarrow 0 \cdot 101 \cdot 100 \rightarrow 0101100$
  - $010110000 \rightarrow 0 \cdot 101 \cdot 100 \cdot 0 \cdot 0 \rightarrow abcaa$

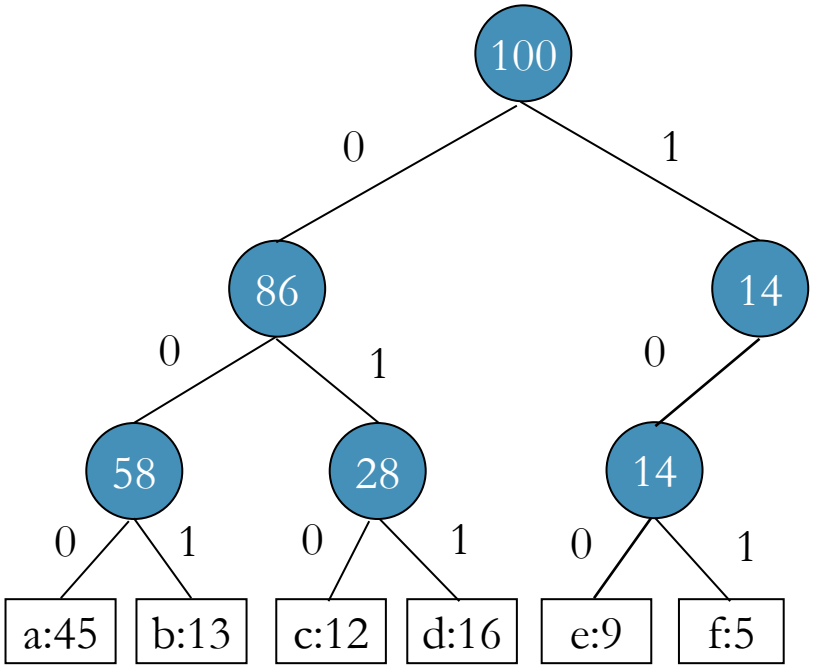
a	b	c
0	101	100

Binary Tree Representation

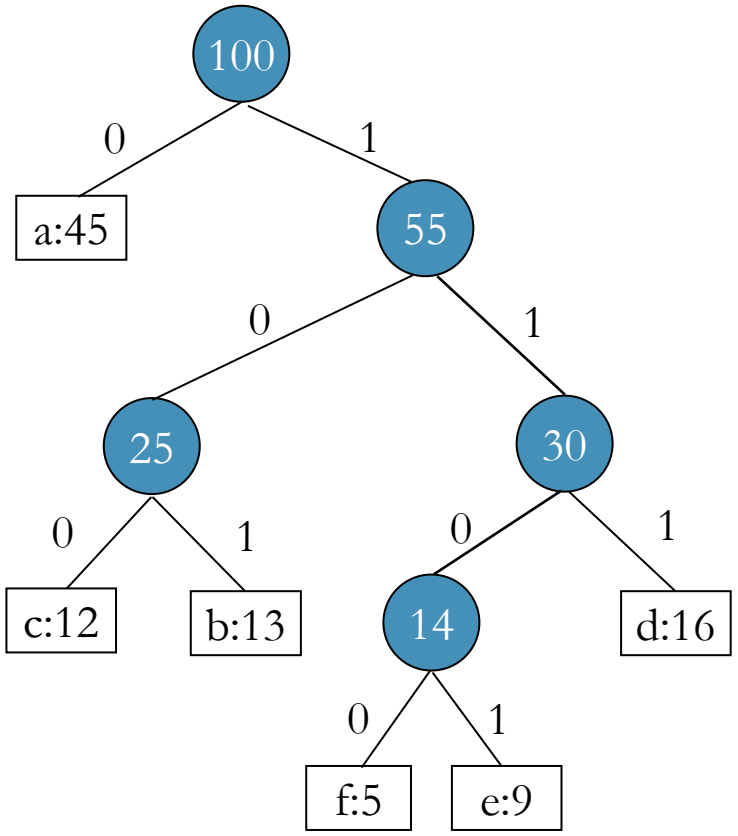
- The prefix coding can be represented by a binary tree, where we put all characters on leaves.
- Interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child.", then we can construct a binary tree corresponding to the coding schemes.



Binary Tree Representation



Fixed-length code



Variable-length code

Binary Tree Representation

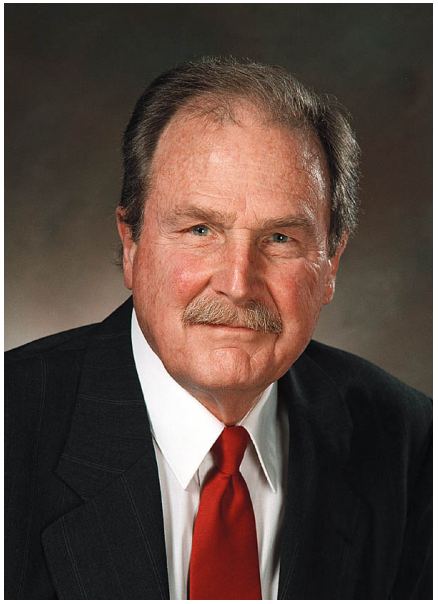
- Given a tree T corresponding to a prefix code, for each character c in the alphabet C, let
  - $f(c)$ denote the frequency of c in the file;
  - $d_T(c)$ denote the depth of c's leaf in the tree, also the length of the codeword for character c.
- The number of bits required to encode a file is thus

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

  which we define as the cost of the tree T.
- It is similar to the optimal BST problem, but with no constraint of being a search tree ($k_{left} \leq k_{node} \leq k_{right}$).

Huffman Code

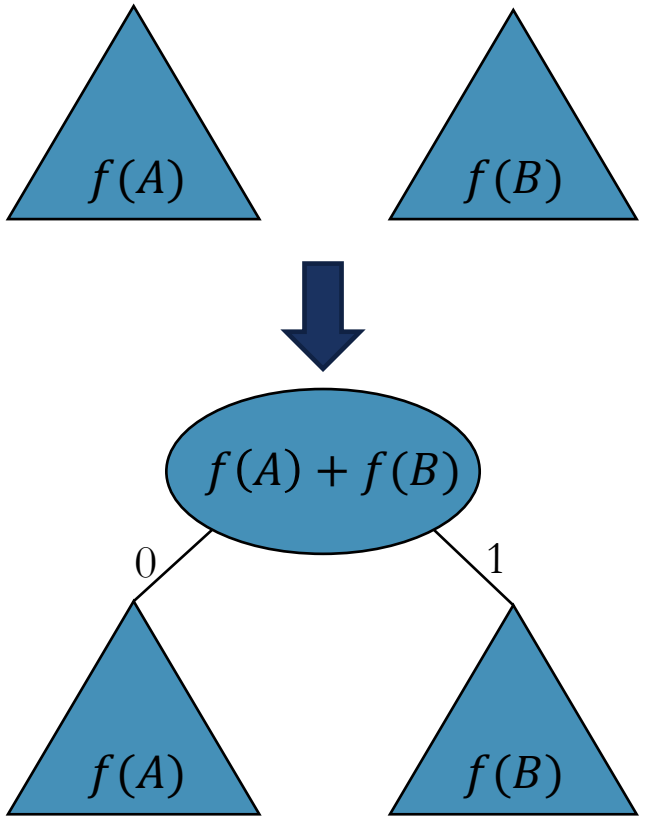
- Huffman code (哈夫曼编码/霍夫曼编码) was developed by David A. Huffman while he was a Sc.D. student at MIT in 1952.
- Huffman code efficiently builds the optimal prefix code, given the frequency of each character.



David Albert Huffman
(1925 – 1999)

Huffman's Algorithm

- Huffman's algorithm builds the codeword bottom up.
- Consider a forest of trees:
  - Initially, one separate node for each character.
  - In each step, join two trees into a larger tree.
  - Repeat this until only one big tree remains.
- Which trees to join? Greedy choice the trees with the lowest frequencies!



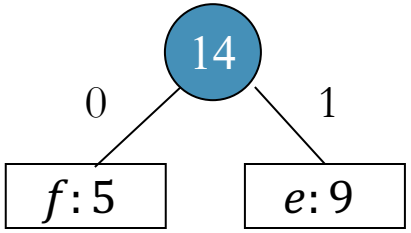
Example

- Sort with character frequency.

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

Example

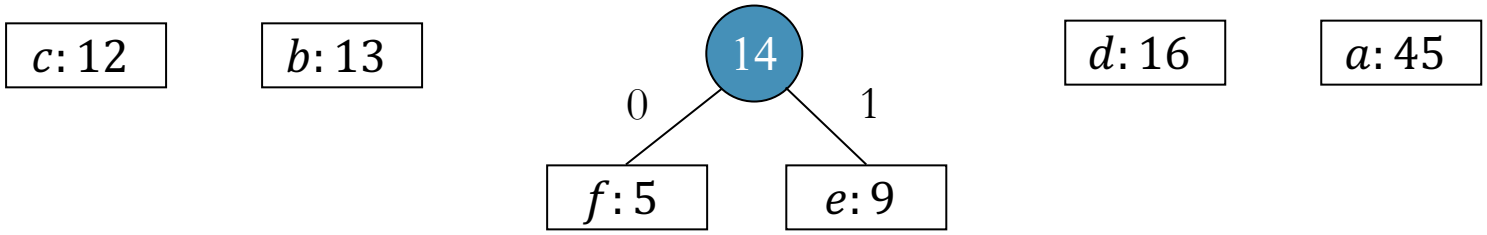
- Join two nodes with lowest frequency, and sum up the frequency.



$c:12$ $b:13$ $d:16$ $a:45$

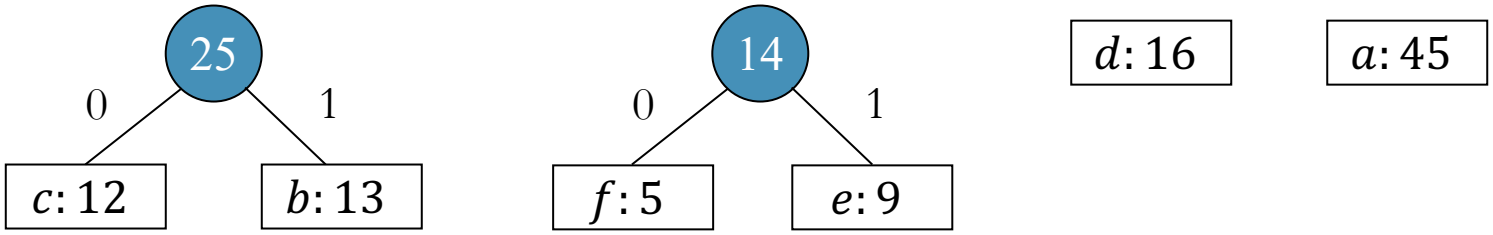
Example

- Insert into proper position.



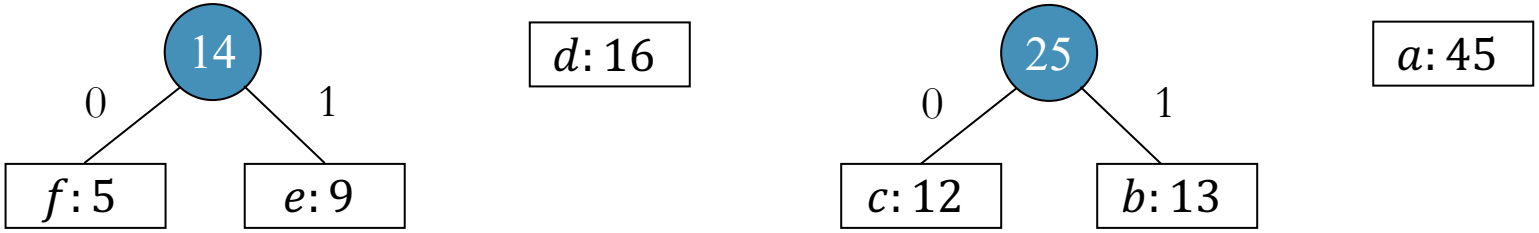
Example

- Join two nodes with lowest frequency, and sum up the frequency.



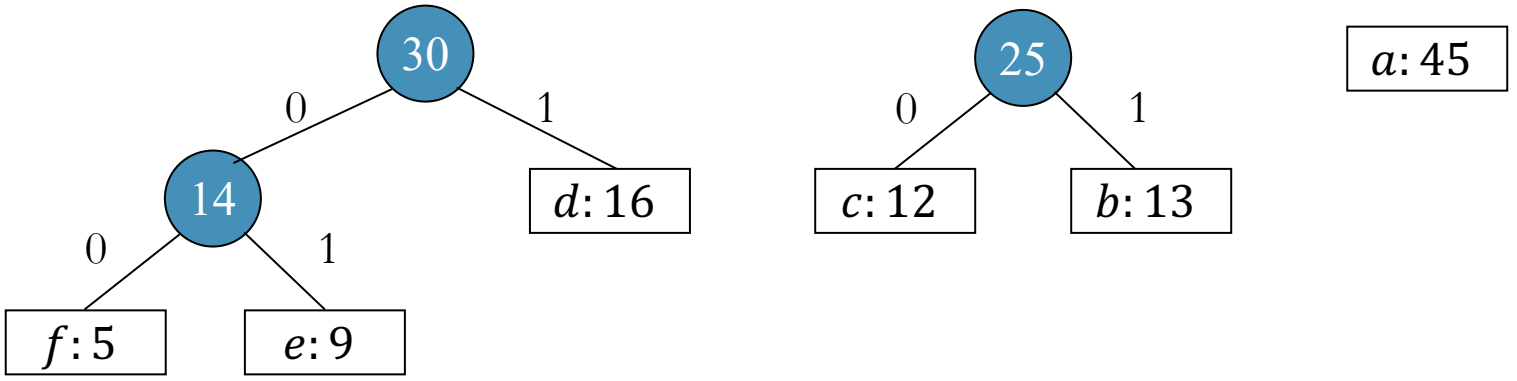
Example

- Insert into proper position.



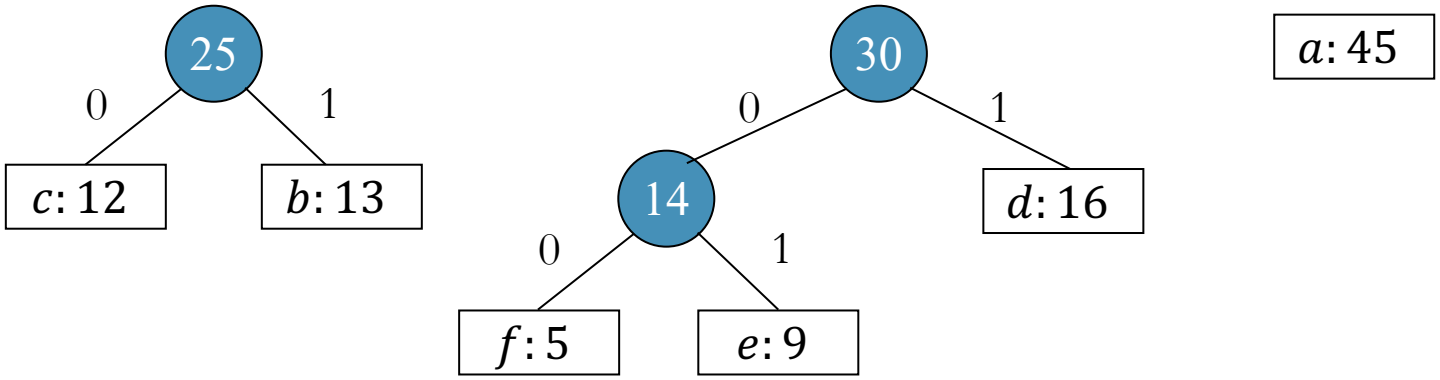
Example

- Join two nodes with lowest frequency, and sum up the frequency.



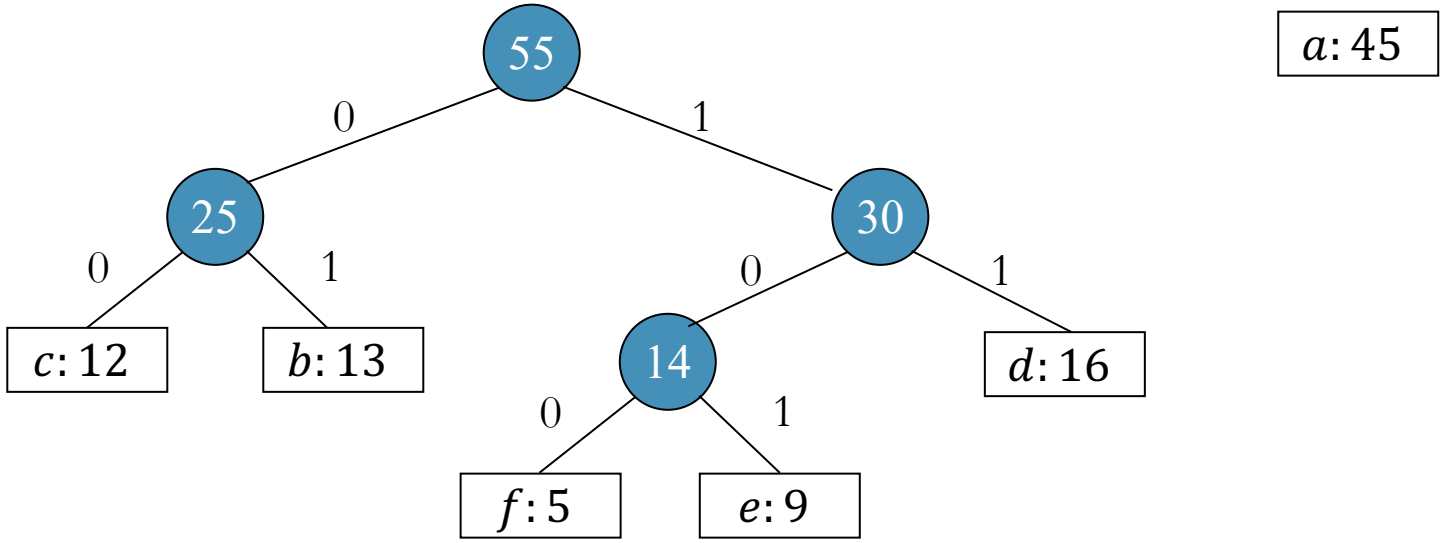
Example

- Insert into proper position.



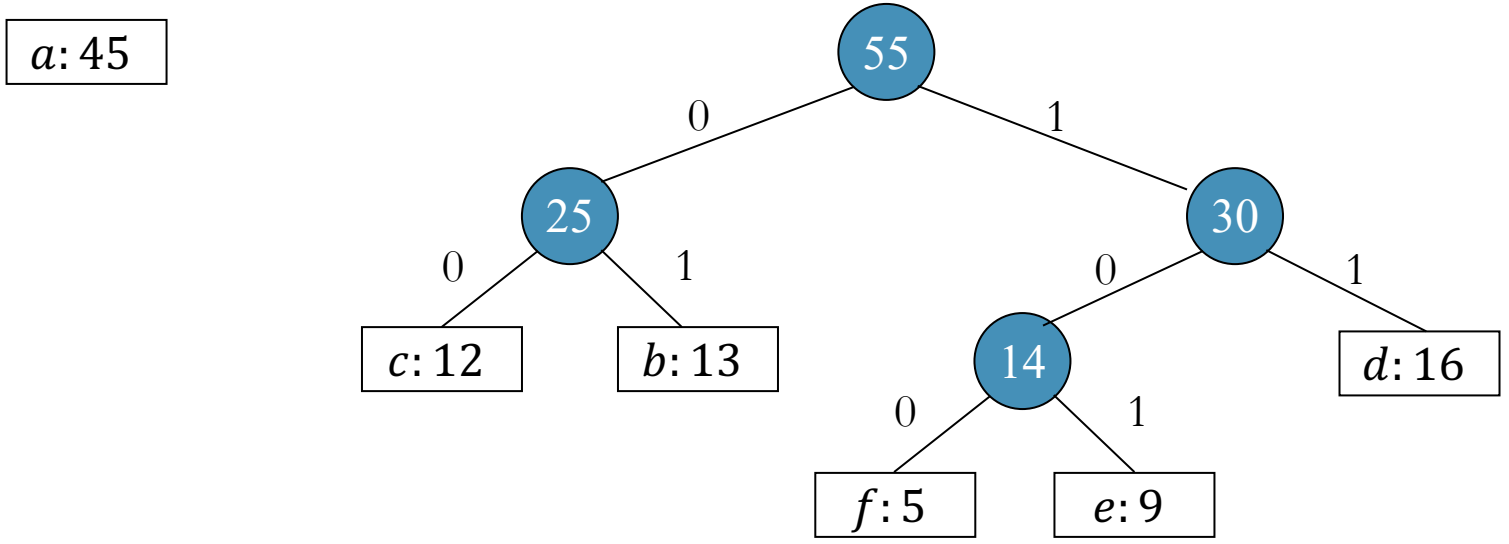
Example

- Join two nodes with lowest frequency, and sum up the frequency.



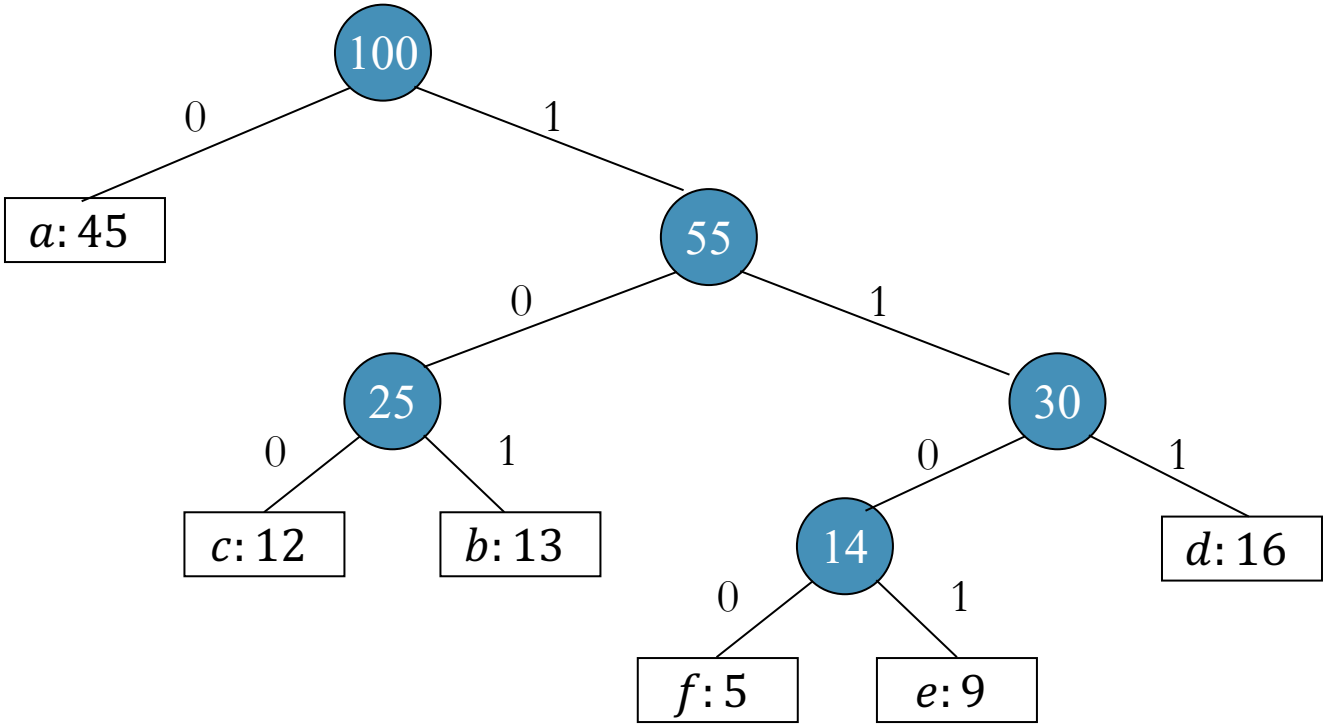
Example

- Insert into proper position.



Example

- Join two nodes with lowest frequency, and sum up the frequency.



Pseudocode

Minimum priority queue

```
HuffmanCode(C)
1  Q ← C
2  for i ← 1 to n − 1 do
3       allocate a new node z
4       x ← ExtractMin(Q)
5       y ← ExtractMin(Q)
6       f(z) ← f(x) + f(y)
7       Insert(Q, z)
8  return ExtractMin(Q)
```

ExtractMin and Insert are heap operators, which take $O(\lg n)$. Therefore, the total cost is $O(n \lg n)$.

- Greedy algorithm is always easy. What is the difficult part?

Correctness of Huffman's Algorithm

We are going to prove two things:

- Greedy choice property: Select the lowest frequency characters is always correct.
- Optimal substructure property: Greedy choice + optimal solution of subproblem = optimal solution of original problem.

Correctness of Huffman's Algorithm

- We first prove the greedy choice property.

Theorem 7.4

Let C be an alphabet in which each character $c \in C$ has frequency $f(c)$.

Let x and y be two characters in C having the lowest frequencies.

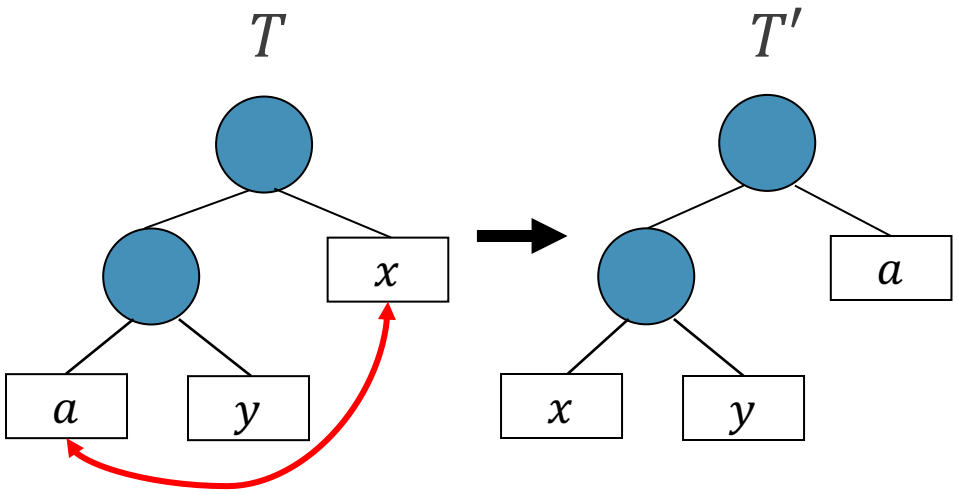
Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

- In one sentence: To join the lowest frequency characters is always correct!

Correctness of Huffman's Algorithm

Proof:

- Idea: If x and y are not in the deepest level of an optimal tree, we can always switch them to the deepest level, to obtain a better tree.



$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= \big(f(x)d_T(x) + f(a)d_T(a)\big) - \big(f(x)d_{T'}(x) + f(a)d_{T'}(a)\big)$$

$$= \big(f(x)d_T(x) + f(a)d_T(a)\big) - \big(f(x)d_T(a) + f(a)d_T(x)\big)$$

$$= \big(f(a) - f(x)\big)\big(d_T(a) - d_T(x)\big) \geq 0$$

Correctness of Huffman's Algorithm

- Then we prove that the problem of constructing optimal prefix codes has the optimal substructure property.

Theorem 7.5

The conditions is the same as Theorem 7.4 .

Let C' be the alphabet C with characters x, y removed and a new character z added, namely $C' = \{C - \{x, y\}\} \cup \{z\}$.

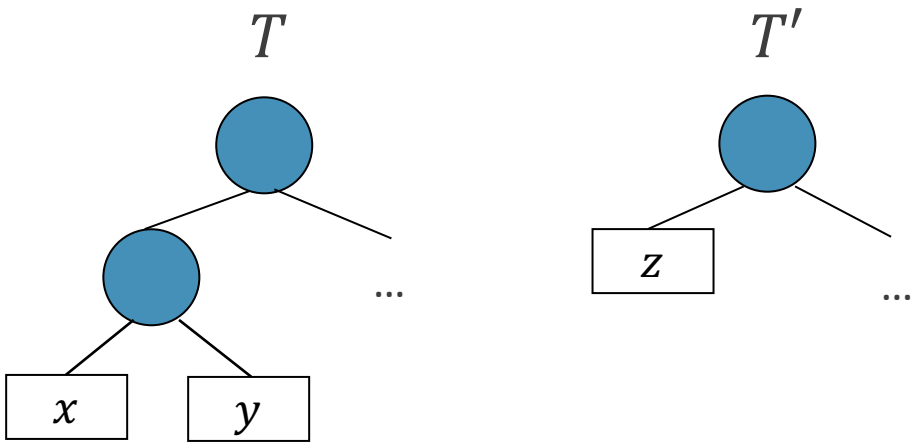
Let $f(z) = f(x) + f(y)$ and other frequencies in C and C' are same.

Let T' be an optimal prefix code for C'.

Then the tree T for an optimal prefix code for C can be obtained from T' by replacing the leaf node for z with an internal node having x and y as children.

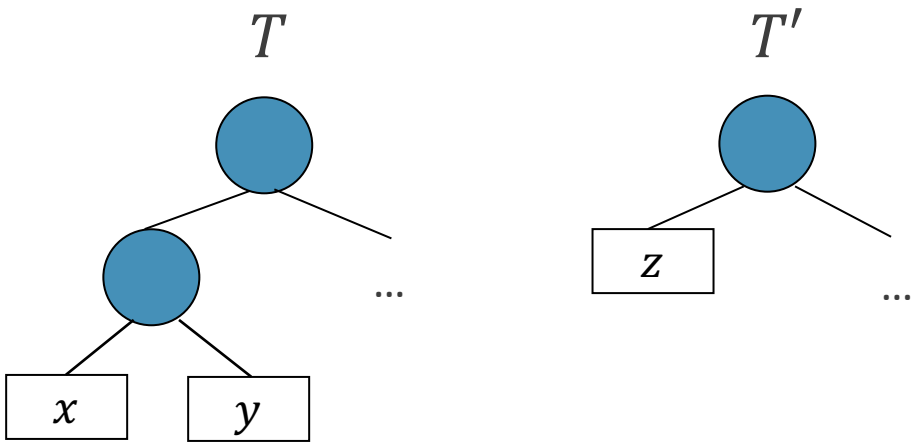
- In one sentence: Merge x and y + optimal solution of C' = optimal solution of C.

Correctness of Huffman's Algorithm



- T' is the optimal solution of the subproblem.
- We need to prove: Given the greedy choice and T', we can obtain the optimal solution T for the original problem.

Correctness of Huffman's Algorithm



Proof:

- Let $B(T)$ and $B(T')$ be the cost of tree T and T', then

$$\begin{aligned} & f(x)d_T(x) + f(y)d_T(y) \\ = & \big(f(x) + f(y)\big)(d_{T'}(z) + 1) \\ = & f(z)d_{T'}(z) + \big(f(x) + f(y)\big) \end{aligned}$$

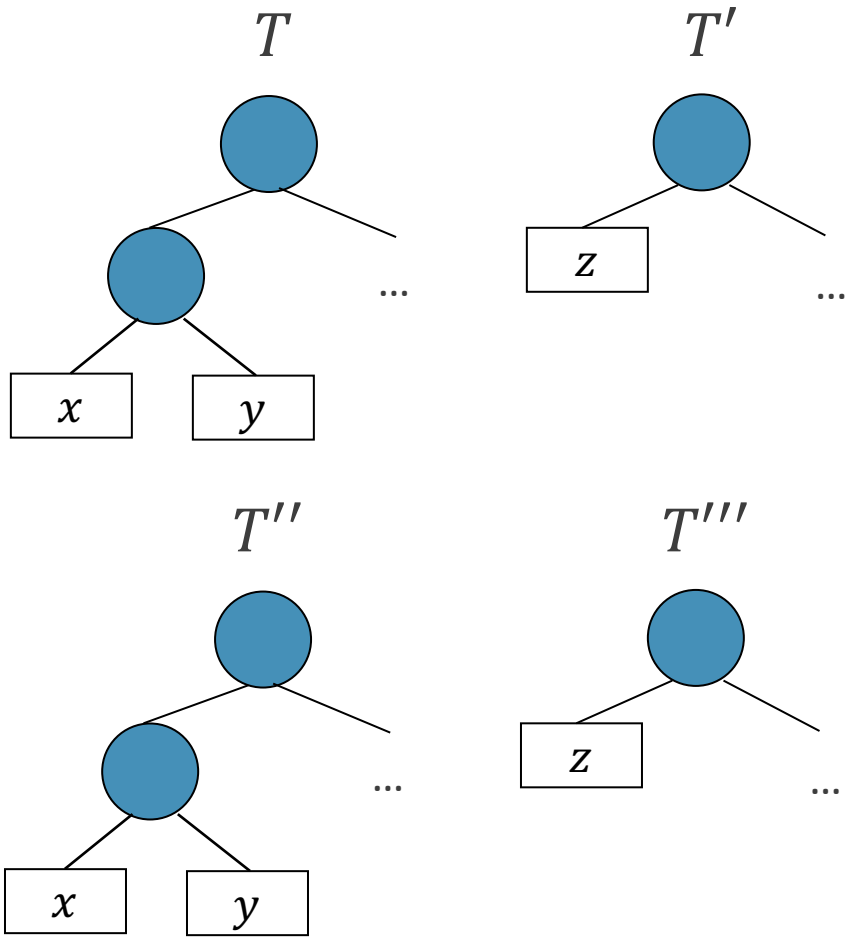
$d_T(x) = d_T(y)$
$= d_{T'}(z) + 1$

- So $B(T) = B(T') + f(x) + f(y)$.

Correctness of Huffman's Algorithm

Proof (cont'd):

- We prove it by contradiction: If T' is optimal but T is not optimal.
- If T is not optimal, there must exist an optimal tree T'' such that $B(T'') < B(T)$, and x and y are also at the deepest level of T'' (Theorem 7.4).
- Then we can construct a tree T''' from T'', by removing x and y and adding z to T''.



Correctness of Huffman's Algorithm

Proof (cont'd):

- By the formula we derive before:

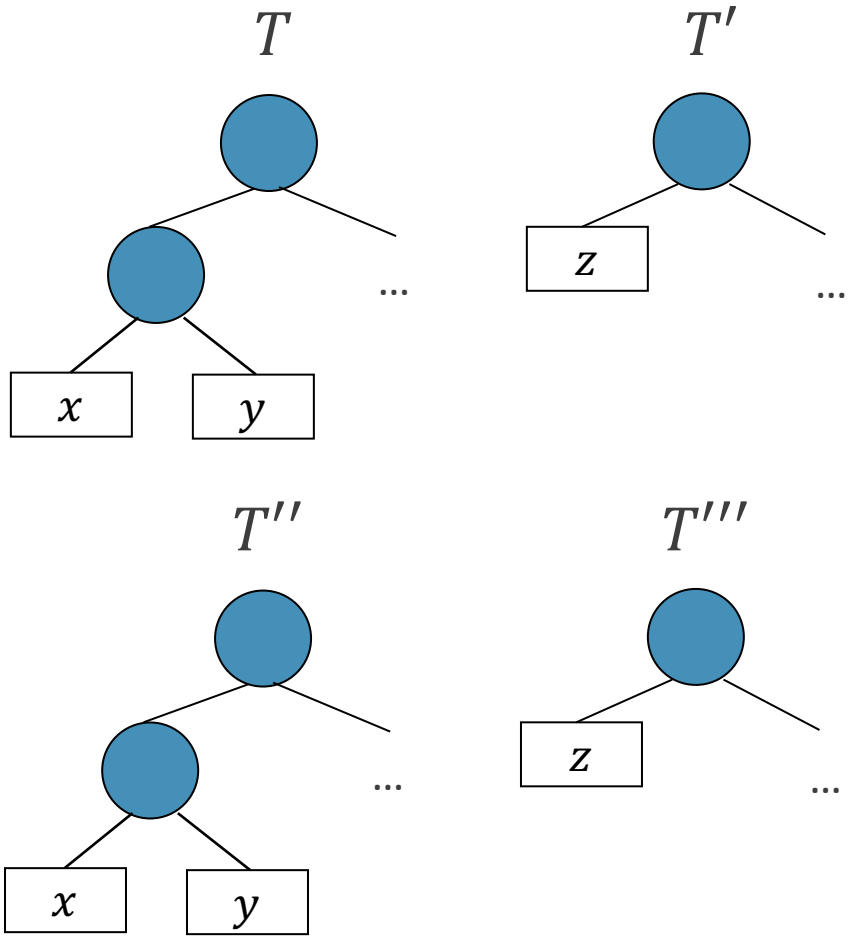
$$B(T'') = B(T''') + f(x) + f(y)$$

$$B(T) = B(T') + f(x) + f(y)$$

$$B(T'') < B(T)$$

we obtain $B(T''') < B(T')$, yielding a contradiction to that T' is optimal.

- Thus T represents an optimal prefix code for C.



Correctness of Huffman's Algorithm

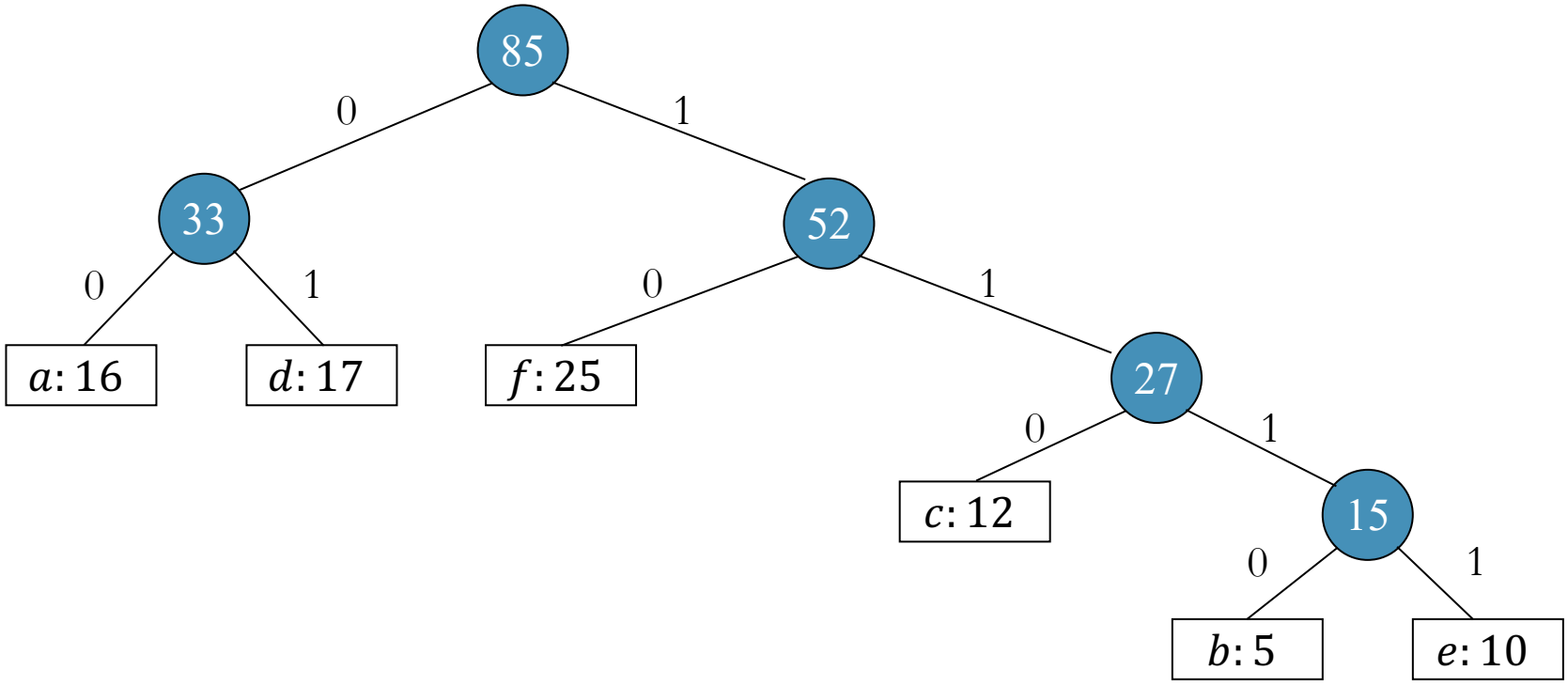
- Theorem 7.4: The greedy choice property.
- Theorem 7.5: The optimal substructure property
- Therefore, the Huffman algorithm produces an optimal prefix code.

Classroom Exercise

Build the Huffman code for the following character frequencies.

Character	Frequecy
a	16
b	5
c	12
d	17
e	10
f	25

Classroom Exercise



Conclusion

After this lecture, you should know:

- What is greedy approach.
- What is greedy choice property.
- How to prove the correctness of a greedy algorithm.
- What is the difference between dynamic programming and the greedy approach.

Homework

- Page 109-111

  7.2

  7.5

  7.8

  7.11

Experiment 1

如何使广告的收益最大化?

- Google的AdWords,或者其他搜索引擎的关键定广告, 使用的基本都是“关键字竞价” (或者称“关键字拍卖”) 的机制, 对每个用户搜索的关键定, 挑选为它竞价的广告来显示.
- 用户搜索的关键字到达搜索引擎次序无法预知, 每个竞价者为一个关键字出的价钱也千差万别, 竞价者还会对每天的花费总额有一个封顶的预算, 超过了这个预算, 即使有合适的关键字, 竞价者也不希望为它多花钱了.
- 搜索引擎们是通过什么样的规则来安排哪个广告给哪个关键字, 以最大化当天的收益的呢?

Experiment 1

- 有N个竞价者, 每个竞价者指定了一个最大预算b_i. 一共有M个关键字. 每个竞价者i对第j个关键字指定一个出价C_{ij}. 竞价开始后, 关键字序列$1, 2, \ldots, M$实时到达, 每个关键字j必须实时分配给某个竞价者i的广告以赚取收益C_{ij}.
- 问题的目标是: 在满足竞价者对关键字匹配要求的基础上, 使总收益最大.

$$\max \sum_{i=1}^{N} \sum_{j=1}^{M} q_{ij} C_{ij}$$

$$\text{s.t.} \sum_{j=1}^{M} q_{ij} C_{ij} \leq b_i \quad \text{for } i = 1, \ldots, N$$

其中$q_{ij} = 1$表示将关键字j实时分配给竞价者i, 否则为0.

- 请设计出一个贪心算法: 任给一个输入实例, 能输出总收益.
  - 测试案例自行设计, 尽可能覆盖各种情形.

Experiment 2

- 用贪心算法求解石材切割问题

谢谢

有问题欢迎随时跟我讨论