Lecture 6:
Greedy algorithm (Part. 2)

Algorithm

Jeong-Hun Kim

Remind

- Greedy algorithm
  - Number selection problem
  - Minimum spanning tree (MST) problem

- MST problem
  - Kruskal's algorithm
    - Improved algorithm: union-find data structure
  - Prim's algorithm

Table of Contents

Part 1

KNAPSACK PROBLEMS

Knapsack Problems

- 0-1 knapsack problem
  - There are n items and a knapsack with weight limit W; the i-th item is worth v_i and weighs w_i, where v_i and w_i are integers
  - Pack the items into the knapsack as valuable as possible, where each item must be either entirely accepted or rejected

- Fractional knapsack problem
  - The same setup as in 0-1 knapsack problem
  - The items can be fractioned





Knapsack Problems

- Optimal substructure
  - 0-1 knapsack problem
    - Suppose that item j is included in an optimal solution $S_n(W)$
    - $S_{n-1}(W-w_j)$: an optimal solution to the subproblem with n-1 items excluding the item j and weight limit $W-w_j$
    - $S_n(W) = S_{n-1}(W-w_j) \cup \{j\}$

- Fractional knapsack problem
  - Suppose that item j of weight w is included in an optimal solution $F_n(W)$
  - $F'_n(W-w)$: an optimal solution to the subproblem with n-1 items and item j of weight w_j-w and weight limit W-w
  - $F_n(W) = F'_n(W-w) \cup \{(j, w)\}$

Knapsack Problems

- Greedy solution
  - Greedy choice
    - Choose first the item with the largest value per weight v_i / w_i
  - Greedy-choice property
    - Proof.
      - Suppose that there are two items i and j such that

        $x_i < w_i$, $x_j > 0$, and $v_i / w_i > v_j / w_j$
      - where x_i, x_j are the amount of item i and j to be put into the knapsack
      - Let $y = \min\{w_i - x_i, x_j\}$
      - We could replace an amount y of item j with an equal amount of item i, thus increasing the total value without changing the total weight
      - Therefore, we can correctly compute optimal amounts for the items by greedily choosing item with the largest value index

Knapsack Problems

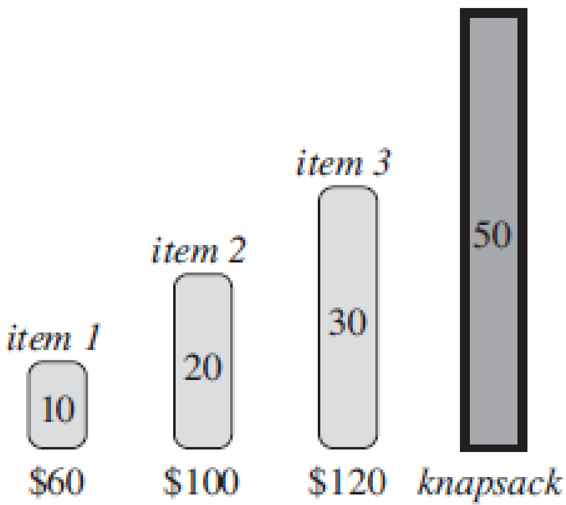
- Greedy solution

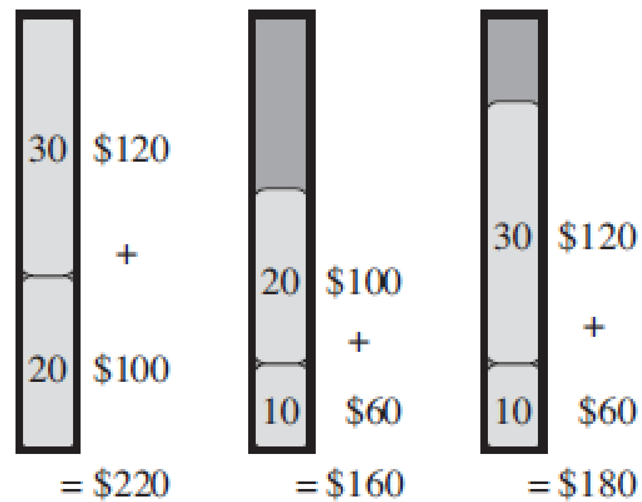
```
GREEDY_FRACTIONAL_KNAPSACK(n, v, W, x)
   for i = 1 to n
      do p_i = v_i / w_i             // the value per weight
         x_i = 0                     // the amount of allocated weight
Sort the items according to p_i      // take O(n lg n) time
cw = W
i = 1
while cw > 0 and i <= n
   do x_i = min{w_i, cw}             // packed into the knapsack
      cw = cw – min{w_i, cw}
      i++
```

- Time complexity: O(n log n)

Knapsack Problems

❖ Can we apply the greedy method to 0-1 knapsack problem?





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

$W = 50.$

How do you solve the 0-1 knapsack problem?

Part 2

INTERVAL SCHEDULING

Interval Scheduling

❖ Problem details

There are n jobs, and each job is already scheduled with the format [start, end]

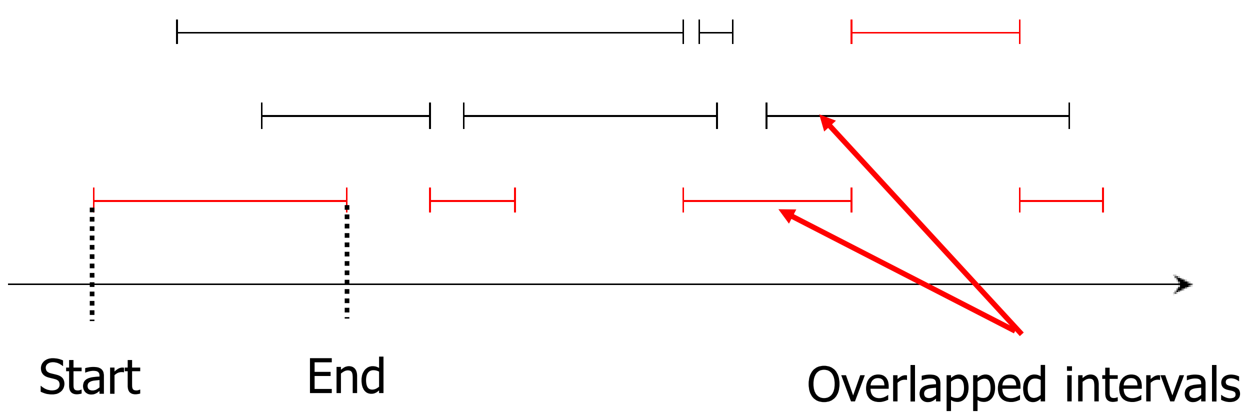
At any given point in time, only one job can be executed

When two jobs are transitioning, it is possible to execute the jobs simultaneously

Two jobs i and j cannot be executed simultaneously, then both tasks are considered to overlap

❖ Goal

Schedule as many jobs as possible within the given time limit
(selection of the optimal jobs)



Interval Scheduling

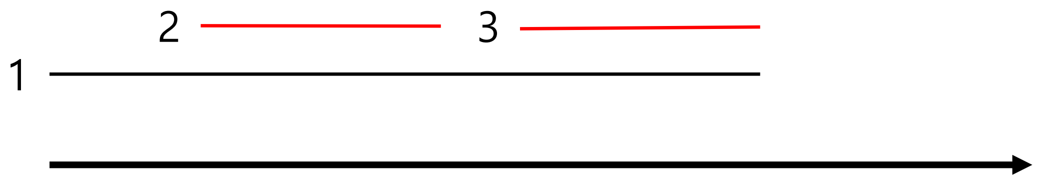
❖ Pseudo code

```
Initially R is the set of all jobs
A is empty        *A will store all the jobs that will be scheduled*
while R is not empty
    choose job i ∈ R
    add i to A
    Remove from R all jobs that overlap with i
Return the set A
```

What is the greedy method for selecting job i ?

Interval Scheduling

- Greedy algorithm 1
  - Select the job with the earliest start time in R

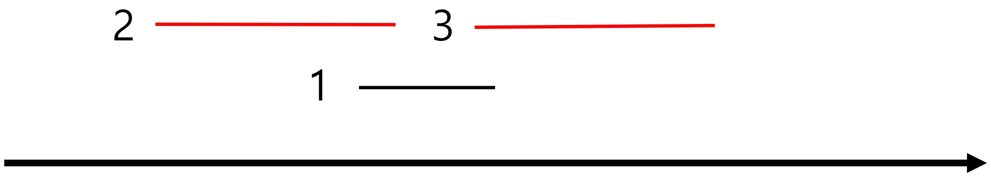


Selected: 1

Optimal solution: 2, 3

Interval Scheduling

- Greedy algorithm 2
  - Select the job with the shortest duration in R

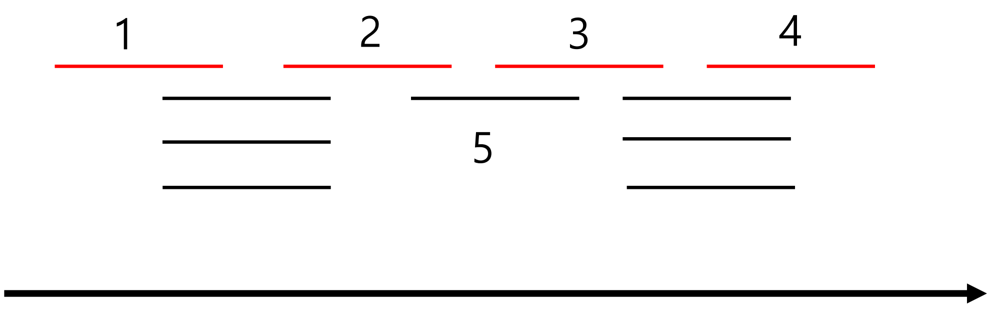


Selected: 1

Optimal solution: 2, 3

Interval Scheduling

- ❖ Greedy algorithm 3
  - Select the job with the fewest overlaps with other jobs in R

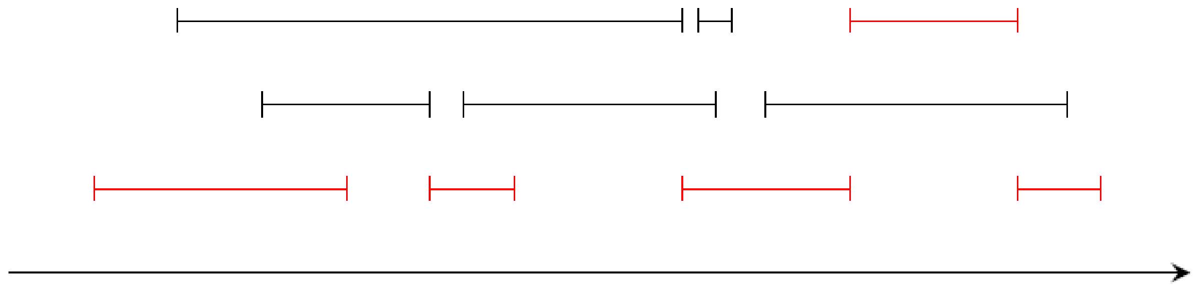


Selected: 5, 1, 4

Optimal solution: 1, 2, 3, 4

Interval Scheduling

- Greedy algorithm 4
  - Select the job with the earliest end time in R



Theorem: greedy algorithm 4 returns an optimal solution

Interval Scheduling

- Greedy algorithm 4 (cont'd)
  - Select the job with the earliest end time in R
  - **Theorem:** it returns an optimal solution
  - **Proof:**
    - According to the previous pseudo code, the jobs selected in Algorithm 4 do not overlap with each other
    - Sets A and O are defined as follows:
      - A: the set of jobs selected by Algorithm 4
      - O: the set of jobs that make up the optimal solution
    - Claim: |A| = |O|
      - When proving a claim, the theorem is also proven

Interval Scheduling

- Greedy algorithm 4 (cont'd)
  - Claim: $|A| = |O|$
    - Let $A=\{i_1, i_2, \ldots, i_k\}$ and $O=\{j_1, j_2, \ldots, j_m\}$
    - Let $s(i)$ and $f(i)$ be the start and end time of job i, respectively
    - Then, we need to prove that $k=m$
    - $f(i_r) \leq f(j_r)$ $(i \leq r \leq k)$ can be proved using induction on k
    - If $m > k$, $f(i_k) \leq f(j_k)$

      $f(i_k) < s(j_{k+1})$

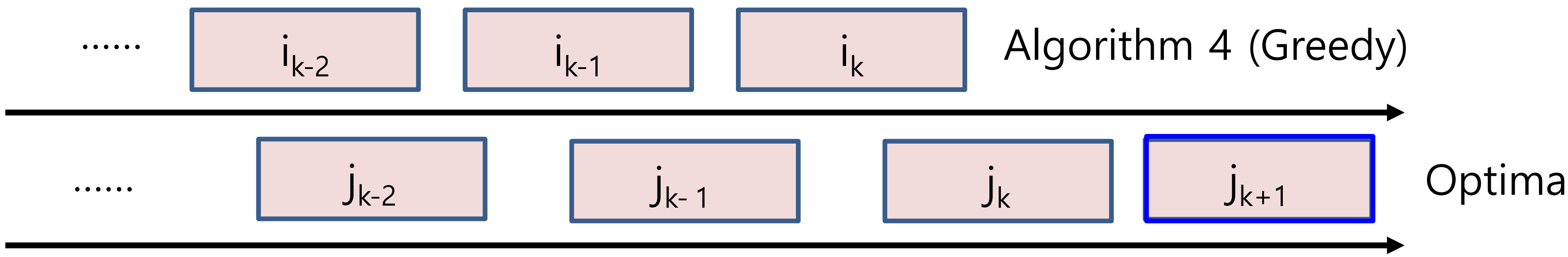
      A can include j_{k+1}

      Therefore, $m > k$ is contradiction

Interval Scheduling

❖ Greedy algorithm 4 (cont'd)

- Claim: |A| = |O|



Interval Scheduling

- Greedy algorithm 4 (cont'd)
  - Pseudo code

```
Initially R is the set of all jobs
A is empty
Sort all the jobs in R based on their finishing time
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in A
        add i to A
    remove from R all jobs that overlap with i
return the set A
```

Interval Scheduling

- Greedy algorithm 4 (cont'd)
  - Time complexity
    - Sort all jobs in order of their end(finish) times: O(n log n)
    - Select the job in R with the earliest end time: O(1)
    - Check for overlaps on a per-job basis: O(1)
    - Total complexity: O(n log n)

Part 3

SCHEDULING ALL REQUESTS

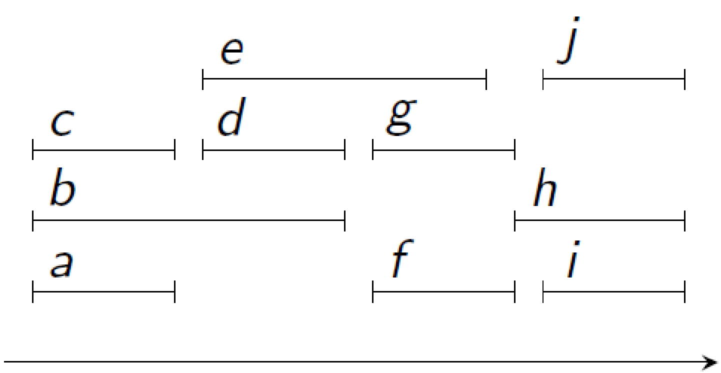
Scheduling All Requests

❖ Problem details

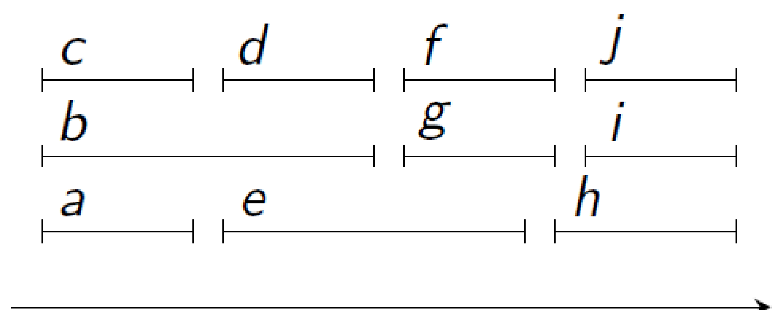
There are n lectures
Each lecture has start and end time (lecture = interval)
At a specific time, only one lecture can be conducted in a classroom

❖ Goal

Determine the minimum number of classrooms required to conduct n lectures



Four classrooms required



Three classrooms required
(optimal solution)

Scheduling All Requests

- Greedy algorithm
  - Select the lecture with the earliest start time among the lectures that have not been assigned a classroom yet
  - Assign the lecture to an available classroom where it can be conducted
  - If there are no available classrooms, allocate the lecture to a new classroom

Scheduling All Requests

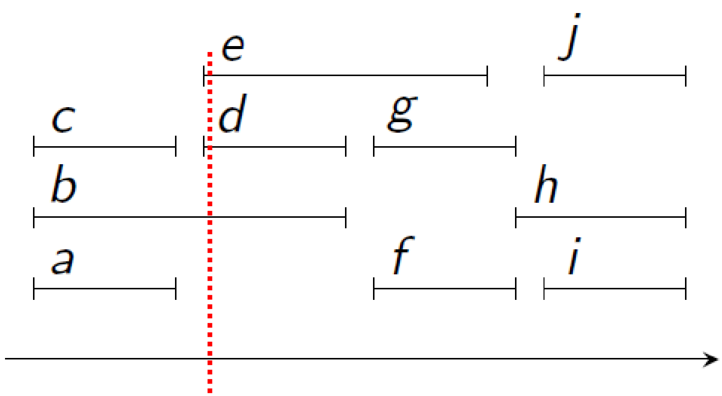
- Greedy algorithm (cont'd)
  - Pseudo code

```
Initially R is the set of all requests
d = 0 /* number of classrooms */
Sort all the lectures in R based on their starting time
while R is not empty
   choose i ∈ R such that start time of i is earliest
   if i can be scheduled in some classroom k ≤ d
      schedule lecture i in classroom k
   else
      allocate a new classroom d+1 and schedule lecture i in d+1
      d = d+1
```

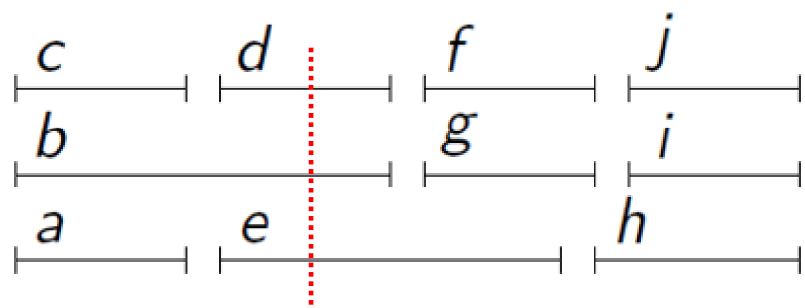
Will the above greedy algorithm return an optimal solution?

Scheduling All Requests

- Greedy algorithm (cont'd)
  - Let's define the depth of R as the maximum number of lectures from within R that are scheduled to be conducted at any specific time
  - Key observation:
    - To conduct all lectures in R, it would need at least the depth of R classrooms



Depth = 3



Depth = 3

Scheduling All Requests

- Greedy algorithm (cont'd)
  - Theorem:
    - The greedy algorithm for R requires the depth of R classrooms (therefore, it results in an optimal solution)
  - Proof:
    - Assume that the greedy algorithm requires more than d classrooms
    - Let j be the first lecture to be conducted in (d+1)-th classroom
    - The greedy algorithm selects lectures in order of their start times
      - Therefore, there are at least d lectures that overlap with j
      - d lectures have start times earlier than j
    - At the starting time of lecture j, the depth of R must be at least d+1
      - It is contradiction

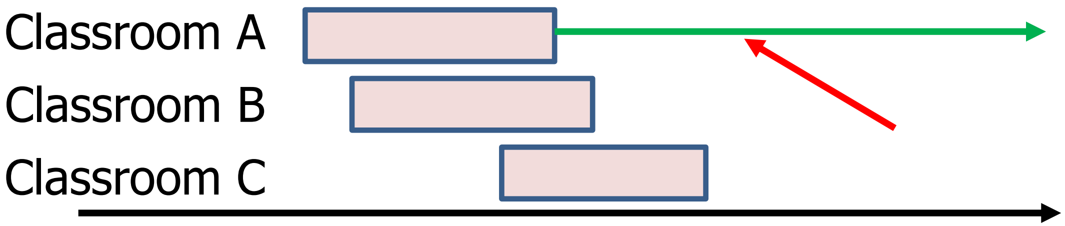
Scheduling All Requests

- Greedy algorithm (cont'd)
  - Time complexity
    - Sort all lectures in order of their start times: O(n log n)
    - Select the lecture in R with the earliest start time: O(1)
    - Assign the selected lecture to a classroom: O(d)
      - Preserve the time when the last lecture ends in each classroom
    - Total complexity: O(n log n + nd)
      - In worst case, d = n, and thus O(n^2)

Is there a way to improve the time complexity?

Scheduling All Requests

- Improved greedy algorithm
  - Time complexity
    - Sorting: O(n log n)
    - Selection: O(1)
    - Allocation:
      - Use a priority queue
        - Priority: the time when the last lecture ends in the classroom
      - Key observation:
        - If it is not possible to allocate a new lecture to the classroom with the highest priority, a new classroom must be created



Scheduling All Requests

- Improved greedy algorithm
  - Time complexity
    - Allocation:
      - Use a priority queue
        - Priority: the time when the last lecture ends in the classroom
        - Allocate a new lecture to a classroom: heapify O(log d)
        - n iteration: O(n log d)
    - Total complexity: O(n log n) + O(n log d) = O(n log n)

Part 4

HUFFMAN ENCODING

Huffman Encoding

❖ Encoding

- Representing a string consisting of an alphabet set Σ as a binary string composed of 0s and 1s
  - Alphabet set of the binary string: {0, 1}

❖ Decoding

- Reconstructing the encoded binary string back into the original string

Welcome to algorithms

Encoding ↓ ↑ **Decoding**

001001001110101010101011101

Huffman Encoding

❖ Simplest encoding

- A method for defining a binary string (code) corresponding to each alphabet (mapping table)
- Duplicate codes are not allowed (due to decoding)

	Binary code
A	00
B	01
C	10
D	11

ABB**C**C**D**A

000101**10**10**11**00

Huffman Encoding

- Simplest encoding (cont'd)
  - If each code has a different length,

	Binary code
A	0001
B	00
C	01
D	001

ABBCCDA

0001000001010010001

- Encoding is suitable, but what about decoding?

0001000001010010001

It is impossible to distinguish between A and BC

Huffman Encoding

- Simplest encoding (cont'd)
  - When no code is a prefix of another code, the decoding is unique

(A is a prefix of B: string B = A + (B - A), e.g., A=00, B=0001)

Concatenation

	Binary code
A	0
B	100
C	101
D	11

ABBCCDA

0100100101101110

By scanning from the left and converting whenever you find an alphabet corresponding to a code, that specific part is uniquely decoded as AB

The binary code described above is known as a prefix-free code

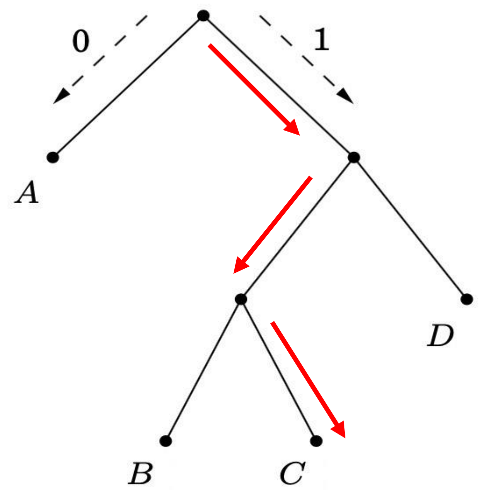
Huffman Encoding

- Simplest encoding (cont'd)
  - It is possible to represent the mapping table of a prefix-free code as a full binary tree, where each node has either 0 or 2 children
    1. Each leaf node of the tree corresponds to an alphabet in the original string
    2. The code for alphabet A is determined by the path from the root of the tree to A
    3. Starting with an empty code, when moving from the root to the left child, concatenate 0 to the code, and when moving to the right child, concatenate 1 to the code to create the code
    4. In the same manner, decoding can also be done solely using the tree

Huffman Encoding

❖ Simplest encoding (cont'd)

	Binary code
A	0
B	100
C	**101**
D	11



Huffman Encoding

❖ Fixed-length encoding vs. variable-length encoding

- Fixed-length encoding
  - Length of the code corresponding to each alphabet is the same
- Variable-length encoding
  - Length of the code corresponding to each alphabet is different
- When is variable-length code useful?

AAAADDBC

	Binary code
A	0
B	100
C	101
D	11

00001111100101 **Length : 14**

Encoding 1 (variable-length)

	Binary code
A	00
B	01
C	10
D	11

0000000011110110 Length : 16

Encoding 2 (fixed-length)

Huffman Encoding

- Fixed-length encoding vs. variable-length encoding (cont'd)
  - A shorter length of the encoded result is preferable

- When using a prefix-free code for encoding, the length of the encoded result can be calculated as follows

The length of result = $\sum_{i=1}^{n} f_i \cdot$ (depth of alphabet i in tree)

Here, f_i is the frequency of the i-th alphabet

AAAADDBC $\longrightarrow$ $f_A = 4$, $f_B = 1$, $f_C = 1$, $f_D = 2$

Goal: constructing a mapping table to minimize the length of the result

Huffman Encoding

- Huffman tree

Goal: constructing a mapping table to minimize the length of the result

  - Variable-length encoding that satisfies the goal is known as Huffman encoding
  - Full binary tree corresponding to Huffman encoding is called a Huffman tree

- Basic idea
  - To minimize the depth of leaf nodes corresponding to high-frequency alphabets

  → Construct subtrees recursively in ascending order of frequency

Huffman Encoding

- Constructing a Huffman tree
  - Construct subtrees recursively in ascending order of frequency
  - Algorithm with example
    - Original string: BDBBEACDEEAEEDBDCD
    - Step 1)
      - calculating the frequency of each alphabet in the original string

f_A	f_B	f_C	f_D	f_E
2	4	2	5	5

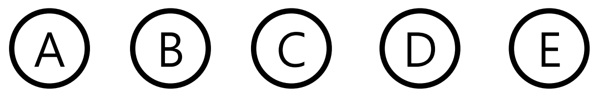
Frequency table

Huffman Encoding

- Constructing a Huffman tree
  - Algorithm with example (cont'd)
    - Step 2)

f_A	f_B	f_C	f_D	f_E
2	4	2	5	5

– Creating a leaf node for each alphabet



– **Lemma**

» Two leaf nodes with the smallest frequencies must have the largest height in the Huffman tree

– **Proof**

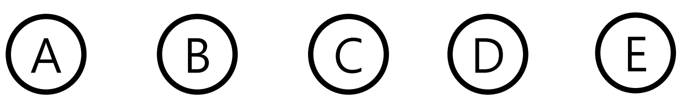
» Assume that there exists a leaf node j with a frequency greater than leaf node i, where leaf node i has the largest height

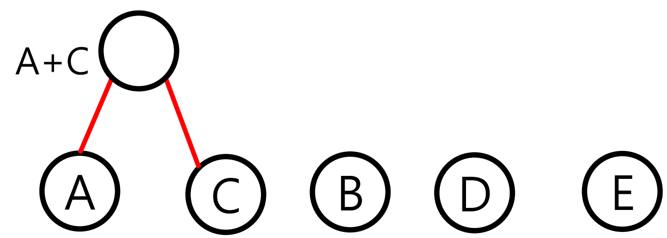
» Swapping i and j in the Huffman tree reduces the length of the encoding result

» The assumption leads to a contradiction

Huffman Encoding

- Constructing a Huffman tree
  - Algorithm with example (cont'd)
    - Step 3) Greedy construction
      - Select the two leaf nodes with the smallest frequencies and construct a binary tree with them as the two children of the root
      - When nodes i and j are selected, the root node of the resulting tree corresponds to the alphabet (i+j), and the frequency of this alphabet becomes $f_i + f_j$



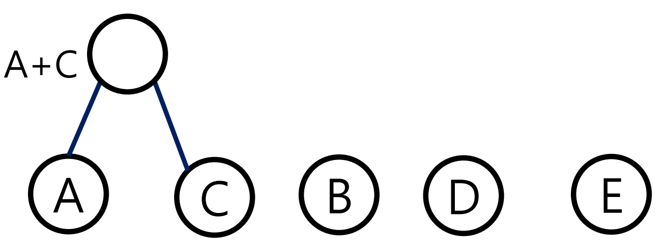


f_A	f_B	f_C	f_D	f_E
2	4	2	5	5

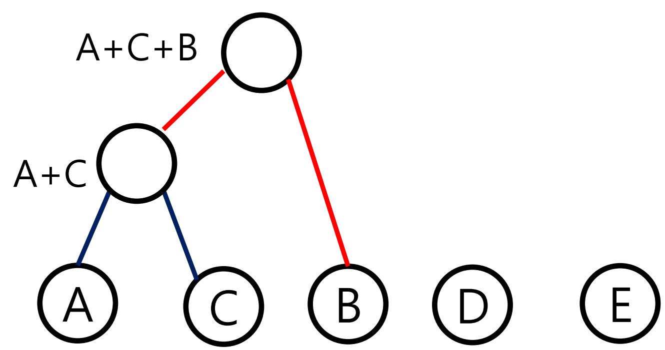
f_{A+C}	f_B	f_D	f_E
4	4	5	5

Huffman Encoding

- Constructing a Huffman tree
  - Algorithm with example (cont'd)
    - Step 4) Recursive greedy construction
      - Repeat step 3 for the remaining alphabets in the current frequency table
      - Until there is only one alphabet left in the frequency table (when the root node of the Huffman tree is generated)



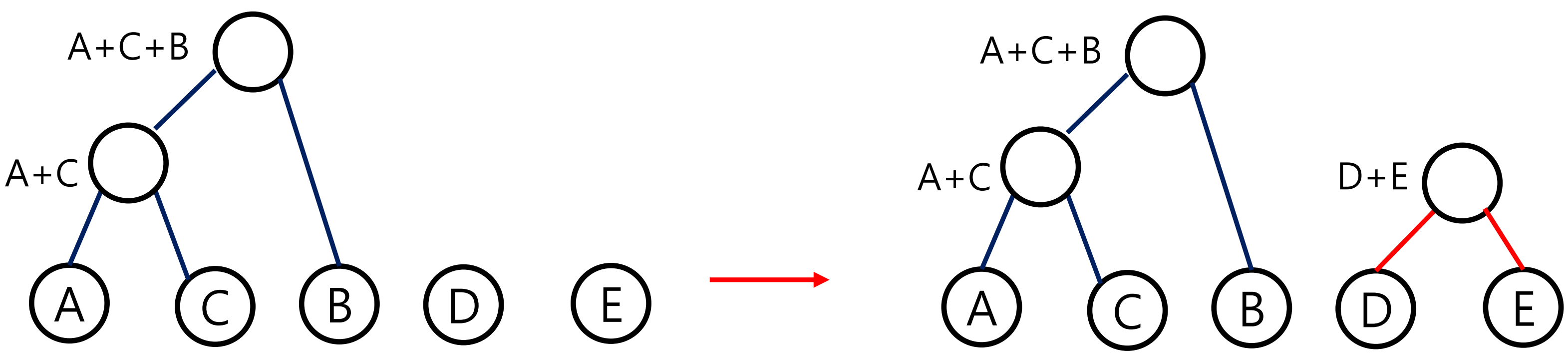
f_{A+C}	f_B	f_D	f_E
4	4	5	5



f_{A+C+B}	f_D	f_E
8	5	5

Huffman Encoding

- Constructing a Huffman tree
  - Algorithm with example (cont'd)
    - Step 4) Recursive greedy construction

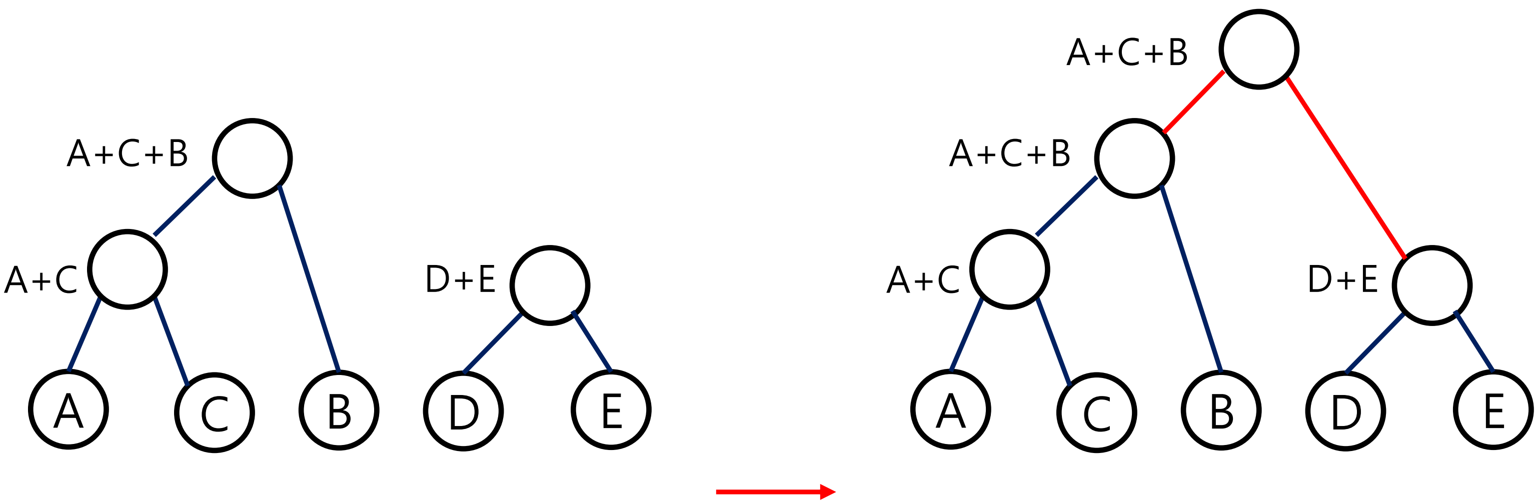


f_{A+C+B}	f_D	f_E
8	5	5

f_{A+C+B}	f_{D+E}
8	10

Huffman Encoding

- Constructing a Huffman tree
  - Algorithm with example (cont'd)
    - Step 4) Recursive greedy construction



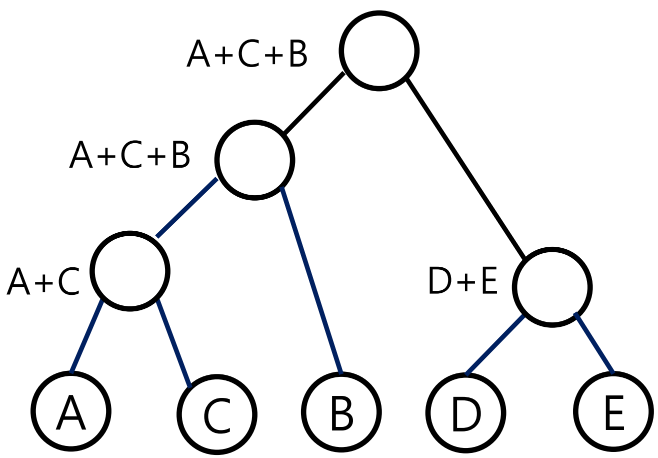
f_{A+C+B}	f_{D+E}
8	10

$f_{A+C+B+D+E}$
18

Huffman Encoding

❖ Constructing a Huffman tree

- Algorithm with example (cont'd)
  - Input string: BDBBEACDEEAEEDBDCD



Huffman tree

	Binary code
A	000
B	01
C	001
D	10
E	11

Codeword table

Huffman encoding: 0110010111000001101111000111110011000110

Huffman Encoding

- Constructing a Huffman tree
  - Pseudo code

```
procedure Huffman(f)
Input: An array f[1…n] of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i=1 to n: insert(H, i)
for k=n+1 to 2n-1:
   i = deletemin(H), j = deletemin(H)
   create a node numbered k with children i, j
   f[k] = f[i] + f[j]
   insert(H, k)
```

deletemin(): select the two alphabets with the smallest frequencies

insert(): add the alphabet to the frequency table

Huffman Encoding

- Constructing a Huffman tree
  - Time complexity
    - Measure the frequency of each alphabet in the original string: O(n)
    - Generate a frequency table: O(n log n) (n times heapify)
    - deletemin(): O(n log n)
    - insert(): O(n log n)
    - Total complexity: O(n log n)

Questions?

SEE YOU NEXT TIME!