CSD2183:
Data Structures

Greedy Algorithms

Bingjie Xu

Contact: bingjie.xu@singaporetech.edu.sg

23 March 2026

Information

- Plan time well for Homework Project 2, clarify early, more analytical thinking and reflection.
- Week 13: Consultation on mock exam during the lecture; office hours on 2 April.
- **Quiz 2: 7 April Tuesday, 16:00 – 18:00**.
- Lecture Quiz 1 & 8 will not be counted into final grade.

Overview

	Basics (Week 1 - 6)	**Advanced (Week 8 - 13)**
1	Foundations	Graph Foundations and Traversal
2	Running Times	Disjoint Sets and Minimum Spanning Trees
3	Sorting	Shortest Paths (Single-Source)
4	List and Hash Tables	Dynamic Programming
5	Trees	**Greedy Algorithms**
6	Consultation	Consultation

Cormen, Thomas H., et al. Introduction to algorithms 4th edition. MIT press, 2022.

Learning Objectives

By end of this lecture, you will be able to:

- Understand the concept of greedy algorithms.
- Identify problems where the **greedy choice property** and **optimal substructure** hold.
- Develop and implement greedy algorithms for common problems (activity selection, Huffman codes).
- Analyze the time complexity of greedy algorithms.

Introduction to Greedy Approach

Greedy Approach

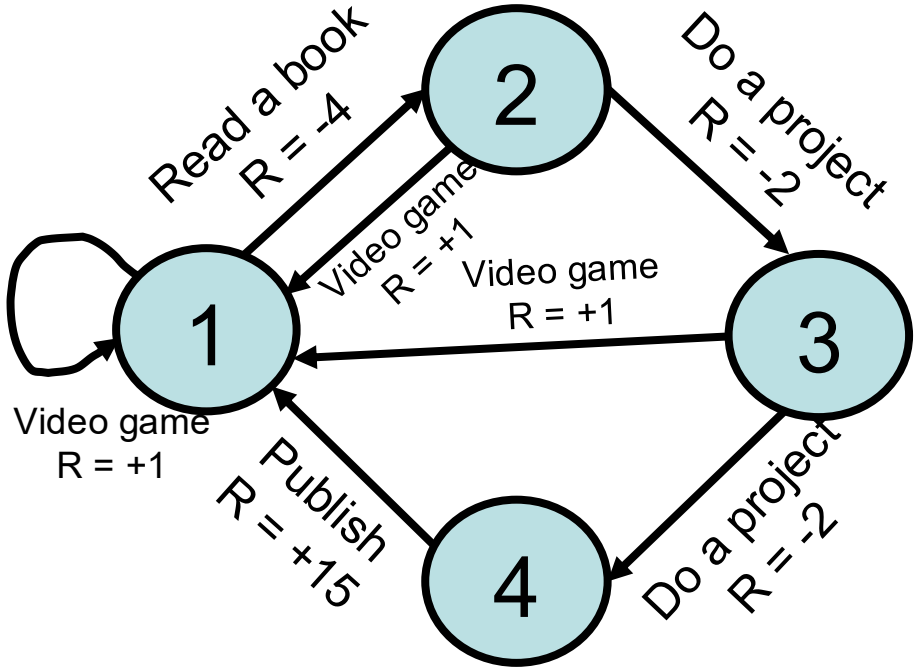
Goal: Start vertex is 1. What is the best path to maximum the reward?

$$R(p) = r_{v_0 v_1} + \gamma \cdot r_{v_1 v_2} + \gamma^2 \cdot r_{v_2 v_3} + \cdots$$

DP approach (lecture 11): the optimal substructure

$$R(v) = \max_u [r_{vu} + \gamma \cdot R(u)]$$

Greedy approach: Always picks the immediate best +1 from the current vertex 1,1,1,1,1,… May miss the better long-term payoff.



Discounted reward problems usually need Bellman/dynamic programming, not plain greedy.

- If $\gamma < 0.576$, greedy self-looping at node 1 is better.
- If $\gamma > 0.576$, (1,2,3,4,1) is better.

Greedy Approach

Goal: Given currency denominations: 5¢, 10¢, 20¢, 50¢, $1, give change to customer using the *fewest* number of coins.

Cashier's algorithm: At each iteration, give the *largest* coin valued ≤ the amount to be paid.

Ex: $2.8.

Greedy Is Not Always Optimal

Observation: Greedy algorithm is sub-optimal for postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

Greedy: 100, 34, 1, 1, 1, 1, 1, 1.

Optimal: 70, 70.

Lesson: Greedy is short-sighted. Always chooses the most attractive choice at the moment. But this may lead to a dead-end later.

Greedy Algorithms

- Hard to define exactly but can give general properties
  - Solution is built in small steps
  - Decisions on how to build the solution are made to maximize some criterion without looking to the future
  - Want the 'best' current partial solution as if the current step were the last step
- May be more than one greedy algorithm using different criteria to solve a given problem

Greedy Algorithms

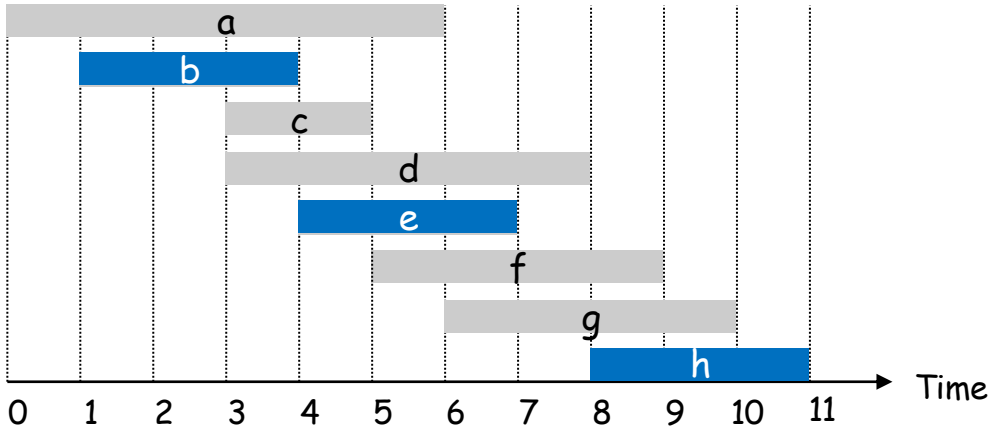
- Greedy algorithms
  - Easy to produce
  - Fast running times
  - Work only on certain classes of problems
    - Hard part is showing that they are correct

- Two methods for proving that greedy algorithms do work
  - Greedy algorithm stays ahead
    - At each step any other algorithm will have a worse value for some criterion that eventually implies optimality
  - Exchange argument
    - Can transform any other solution to the greedy solution at no loss in quality

Classical Problems using Greedy

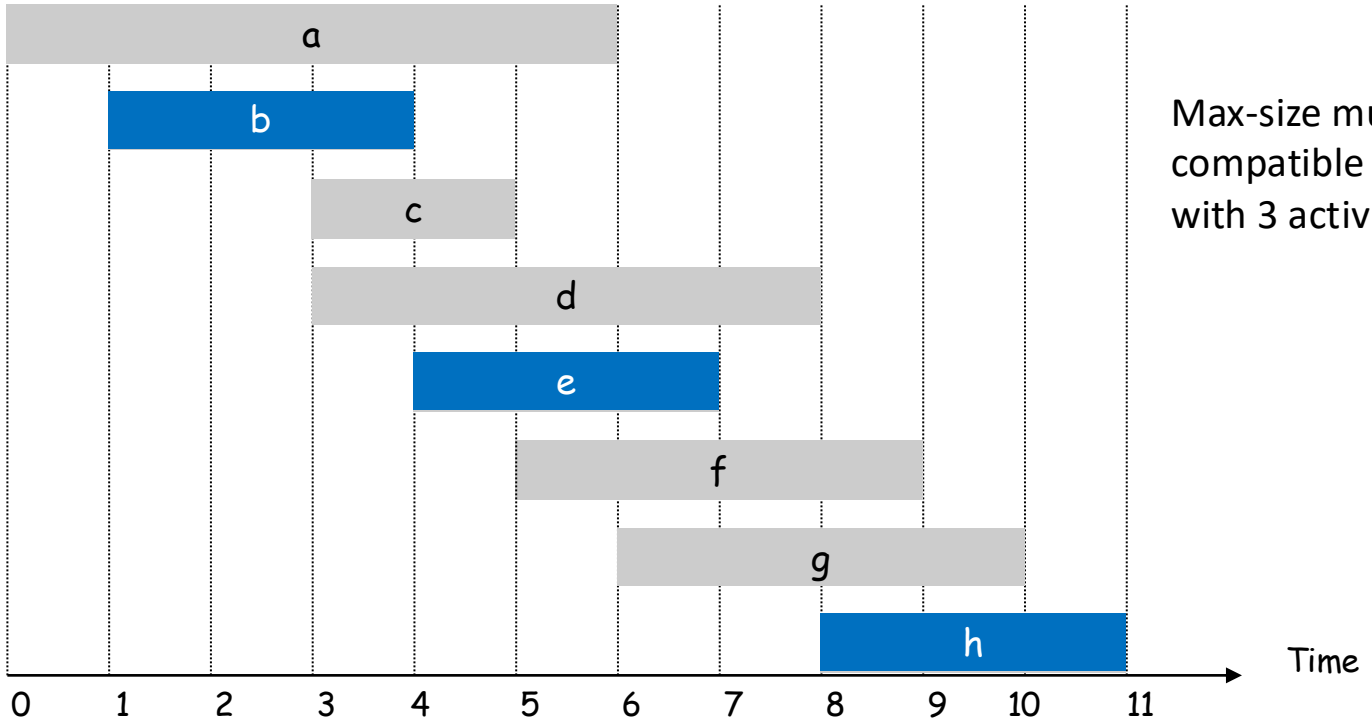
- Activity Selection

- Huffman Codes



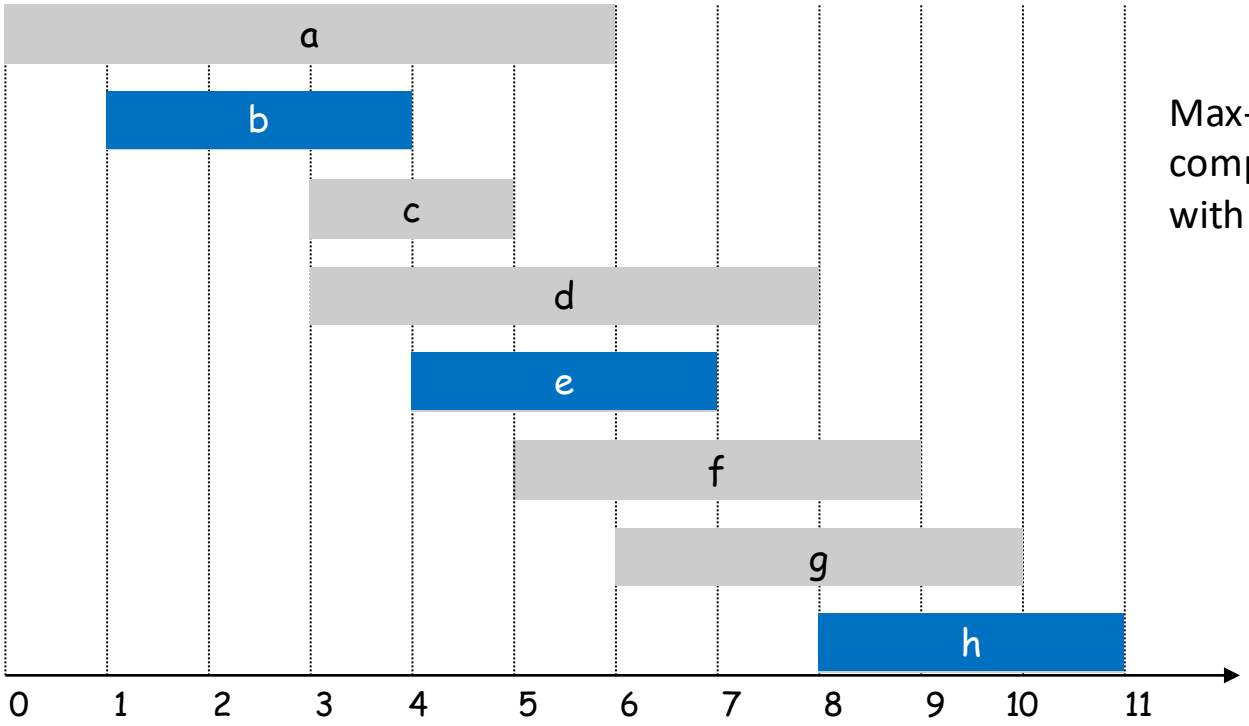
Interval Scheduling

- Activity j starts at $s(j)$ and finishes at $f(j)$, duration $[s(j), f(j))$.
- Two activities compatible if they don't overlap.
- Goal: Find the maximum subset of mutually compatible activities.
- Example:



Interval Scheduling

activity	a	b	c	d	e	f	g	h
s_i	0	1	3	3	4	5	6	8
f_i	6	4	5	8	7	9	10	11



Max-size mutually compatible set: {b, e, h} with 3 activities

Greedy Approach

Sort the activities in some order. Go over the activities and take as many as possible provided it is compatible with the activities already taken.

Main questions:

- What order?
- Does it give the optimum answer?
- Why?

Possible Approaches for Inter Sched

Sort the activities in some order. Go over the activities and take as many as possible provided it is compatible with the activities already taken.

[Shortest interval] Consider activities in ascending order of interval length $f(j) - s(j)$.

Counter example for shortest interval

```
|----------| |----------|
      |----|        <--- Will select this one
```

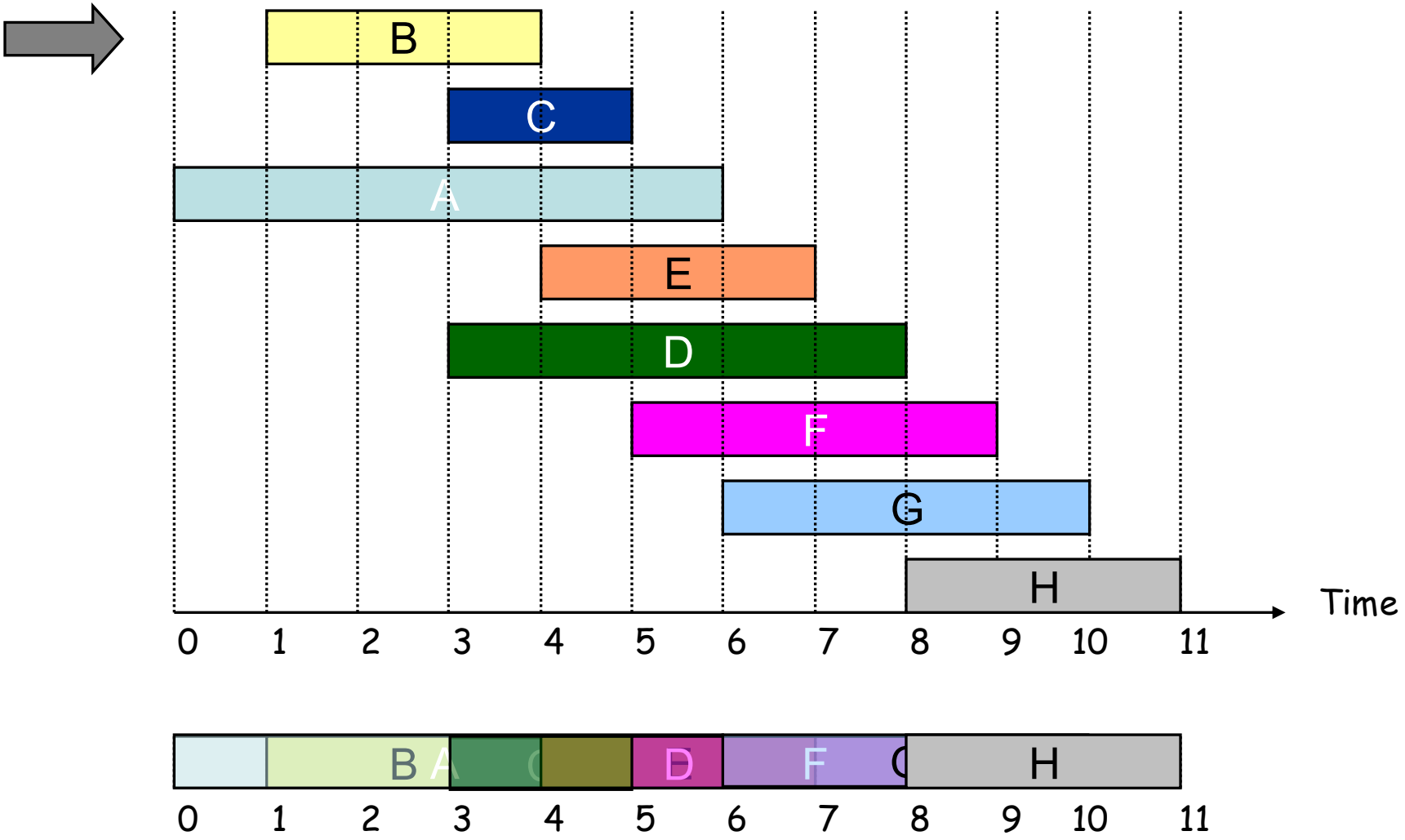
[Earliest start time] Consider activities in ascending order of start time $s(j)$.

Counter example for earliest start time

```
|-------------------------|   <--- Will select this one
  |--------| |---------|
```

[Earliest finish time] Consider activities in ascending order of finish time $f(j)$.

Greedy Alg: Iterative



Greedy Alg: Iterative

Consider activities in increasing order of finish time. Take each activity provided it's compatible with the ones already taken.

```
Sort activities by finish times so that f(1)≤ f(2)≤ ... ≤ f(n)   O(nlogn)
A ← {a_1}
for j = 2 to n {                                                 Θ(n)
    if (job j compatible with A)
        A ← A ∪ {j}                                              O(1)
}
return A
```

Implementation. $O(n\log n) + O(n) \Rightarrow O(n\log n), O(n\log n)$ grows faster than $O(n)$.

- Remember activity j^* that was added last to A.
- Activity j is compatible with A if $s(j) \geq f(j^*)$.

Greedy Alg: Iterative

Consider activities in increasing order of finish time. Take each activity provided it's compatible with the ones already taken.

Sort activities by finish times so that f(1)≤ f(2)≤...≤ f(n). $O(nlogn)$

GREEDY-ACTIVITY-SELECTOR(s, f, n)

```
1  A = {a_1}
2  k = 1
3  for m = 2 to n                                   => Check if compatible
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

Greedy Alg: Recursive

Consider activities in increasing order of finish time. Take each activity provided it's compatible with the ones already taken.

Sort activities by finish times so that f(1)≤ f(2)≤...≤ f(n). $O(nlogn)$

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n) // the k-th activity

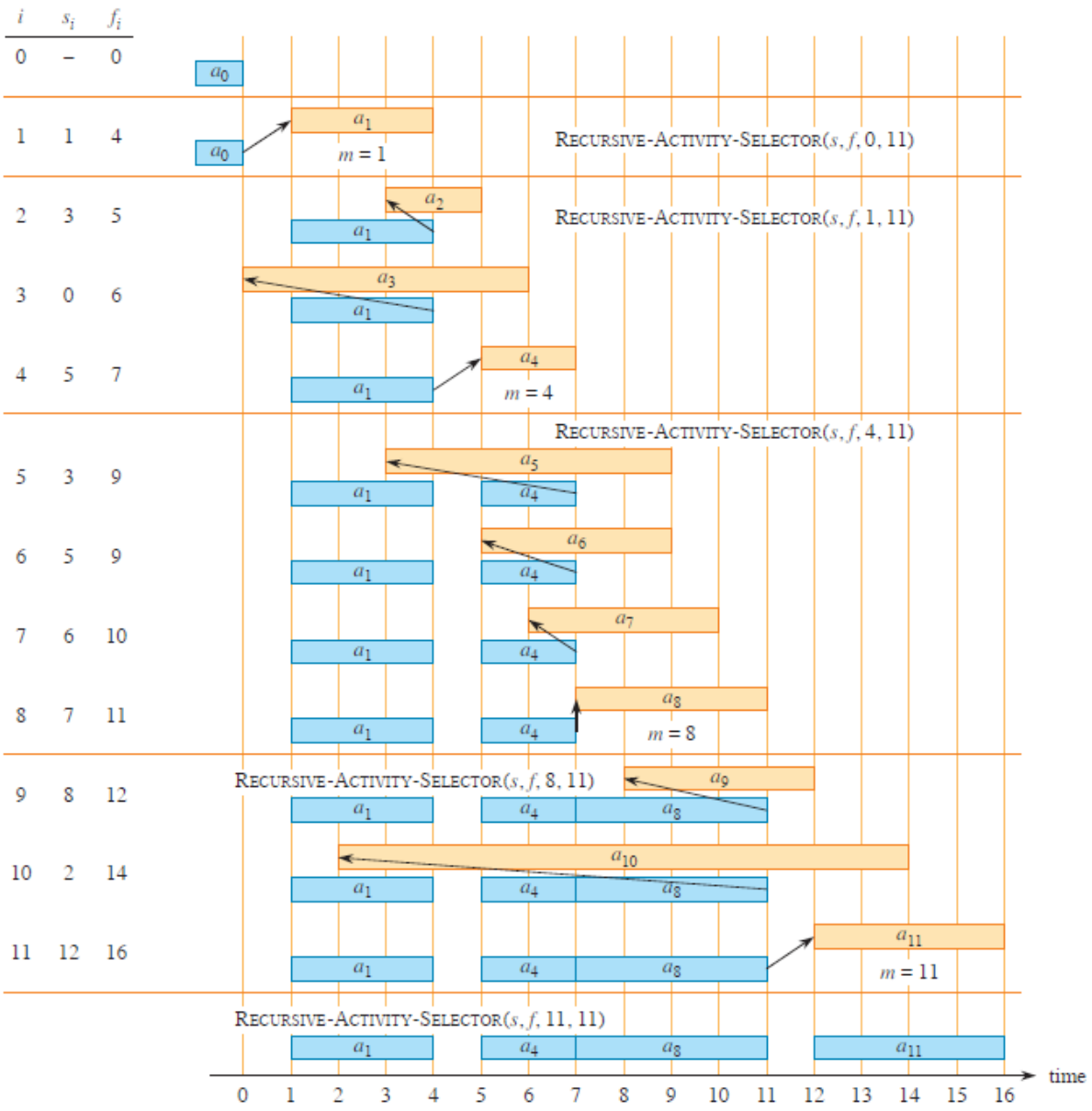
$\Theta(n)$, each activity is examined exactly once

```
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3        m = m + 1                    exit until m > n or s[m] ≥ f[k], i.e., s[k + 1] ≥ f[k]
4  if m ≤ n
5        return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Initial call: RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$

Implementation. $O(n\log n)$.

Greedy Alg: Recursive



Correctness

- The output is compatible. (This is by construction.)

How to prove it gives maximum number of activities?

Let $i_1, i_2, i_3, \cdots$ be activities picked by greedy (ordered by finish time)

Let $j_1, j_2, j_3, \cdots$ be an optimal solution (ordered by finish time)

How about proving $i_k = j_k$ for all k?

No, there can be multiple optimal solutions.

Idea: Prove that greedy picks at least as many activities as any optimal solution ("best")

Given two compatible orders, which is better?

The one finishes earlier.

How to prove greedy gives the "best"?

Induction: it gives the "best" during every iteration.

Correctness

Theorem: Greedy algorithm here is optimal.

Proof: Greedy stays ahead.

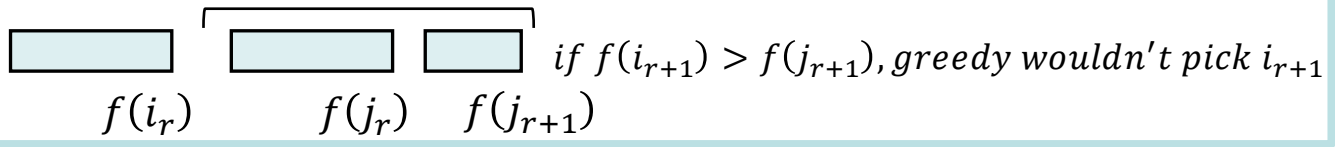
Let $i_1, i_2, i_3, \cdots, i_k$ be activities picked by greedy, $j_1, j_2, j_3, \cdots, j_m$ those in some optimal solution in order.

We show $f(i_r) \leq f(j_r)$ for all r, by induction on r.

Base Case: i_1 chosen to have min finish time, so $f(i_1) \leq f(j_1)$.

IH: $f(i_r) \leq f(j_r)$ for some r

IS: Since $f(i_r) \leq f(j_r) \leq s(j_{r+1})$, j_{r+1} is among the candidates considered by greedy when it picked i_{r+1}, & it picks min finish, so $f(i_{r+1}) \leq f(j_{r+1})$



Observe that we must have $k \geq m$, otherwise there exists an activity j_{k+1} in the optimal solution that could be picked by greedy.

Greedy vs. Dynamic Programming

- Dynamic programming
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Greedy
  - Make a choice at each step.
  - Make the choice *before* solving the subproblems. It will not be revisited.

Elements of Greedy

If greedy can't guarantee optimal, how to decide when to use greedy?

- **Greedy-choice property**
  - Can always make a local best choice that leads to a globally optimal solution
  - Get efficiency gains:

    Preprocess input to put it into greedy order

    Or if dynamic data, use a priority queue/heap (Prim's, Dijkstra)

- **Optimal substructure**

  Show that optimal solution to subproblem and greedy choice => optimal solution to problem

Substructure in activity selection:
$S_k = \{a_i \in S: s_i \geq f_k\}$ activities that start after a_k finishes

What if the activities are weighted?

You can't solve it using greedy, use DP

$$dp[i] = \max(dp[i-1], dp[j] + w[i])$$

Suppose each task has a weight.
Goal: Maximum sum of weights of finished tasked.

[Shortest interval] Consider activities in ascending order of interval length $f(j) - s(j)$.

[Earliest start time] Consider activities in ascending order of start time $s(j)$.

[Earliest finish time] Consider activities in ascending order of finish time $f(j)$.

[Highest Rate] Consider activities in descending order of $w(j)/(f(j) - s(j))$.

Exercise

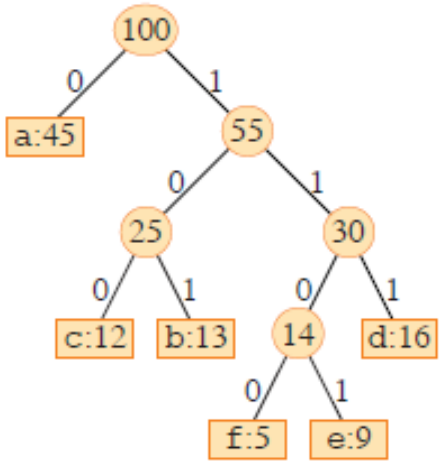
In the GenAI era, as a developer, you must be able to read code and debug mistakes. Debug below code and submit to xSITe Quiz for lecture 12.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<pair<int, int>> activitySelection(vector<pair<int, int>> &activities) {
    // Handle empty input case
    if (activities.empty()) return {};

    // Sorting (here using library, recall sorting algos)
    sort(activities.begin(), activities.end(), [](pair<int, int> a, pair<int, int> b) {
        return a.second < b.first; // Fix: a.second < b.second sort by finish time
    });

    // Greedy choice
    vector<pair<int, int>> selected;
    selected.push_back(activities[0]);  // Select the first activity
    int lastFinishTime = activities[0].second;
    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].second >= lastFinishTime) { // Fix: activities[i].first, compare start time >= previous finish time
            selected.push_back(activities[i]);
            lastFinishTime = activities[i].second;
        }
    }
    return selected;
}

int main() {
    vector<pair<int, int>> activities = {{1, 3}, {2, 5}, {3, 9}, {6, 8}, {5, 7}, {8, 9}};
    vector<pair<int, int>> selectedActivities = activitySelection(activities);
    cout << "Selected Activities: " << endl;
    for (auto activity : selectedActivities) {
        cout << "(" << activity.first << ", " << activity.second << ") ";
    }
    cout << endl;
    return 0;
}
```

Expected output:
(1,3) (5,7) (8,9)

Classical Problems using Greedy

- Activity Selection
- **Huffman Codes**



ASCII Mystery:

01000010 01000001 01000111

- A character encoding standard maps characters to numerical values.
- Here's a small segment from the ASCII encodings for characters, store in 8 bits (1 byte) per character.
- What is the mystery word in the title of this slide page?
- In the computer's eyes, "BAG" is equivalent to the bit sequence 010000100100000101000111

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

Drawbacks of fixed-length codes

- Wasted space
- Same number of bits used to represent all characters

  'a' and 'e' occur more frequently than 'q' and 'z'
- **Potential solution**: use variable-length codes
  - variable number of bits to represent characters when frequency of occurrence is known
  - short codes for characters that occur frequently

But with variable-length codes, how do we know where one character ends and another begins?

Prefix property

- A code has the **prefix property** if no character code is the prefix (start of the code) for another character
- Example:

Symbol	Code
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- 000 is not a prefix of 11, 01, 001, or 10
- 11 is not a prefix of 000, 01, 001, or 10 …

Task

- Design a variable-length prefix-free code such that the message **DEAACAAAAABA** can be encoded using 22 bits.
- Possible solution:
  1. A occurs eight times while B, C, D, and E each occur once
  2. represent **A** with a one bit code, say 0

     - remaining codes cannot start with 0
  3. represent **B** with the two bit code 10

     - remaining codes cannot start with 0 or 10
  4. represent **C** with 110
  5. represent **D** with 1110
  6. represent **E** with 11110

Encoded message and code

DEAACAAAAABA

Symbol	Code
A	0
B	10
C	110
D	1110
E	11110

111011110001100000100

22 bits

Another possible code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	1101
E	1111

1101111100101000001000 22 bits

Better code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	110
E	111

11011100101000001000 20 bits

What code to use?

- Question: Is there a variable-length code that makes the most efficient use of space?

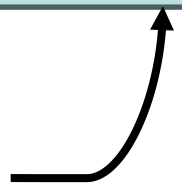
Answer: Yes! Huffman Codes.

Huffman Codes

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a provably minimal encoding for a given pattern of letter frequencies.
- Different data (text, images, etc.) will each have their own personalized Huffman coding tree.
- The Huffman coding algorithm is a flexible, powerful, adaptive algorithm for data compression, from GZIP, PKZIP (winzip etc) and BZIP2, to image formats such as JPEG and PNG.

Huffman Codes

Swapping 0/1 on edges still produces valid prefix-free codes.



- Binary tree
  - each leaf contains symbol (character)
  - label edge from node to left child with 0
  - label edge from node to right child with 1
- Code for any symbol by following path from root to the leaf containing symbol
- Code has prefix property
  - leaf node cannot appear on path to another leaf
  - *note*: fixed-length codes are represented by a complete Huffman tree and clearly have the prefix property

Building a Huffman Tree

Steps:

- Find frequencies of each symbol occurring in message
- Begin with a forest of single node trees
  - each contain symbol and its frequency
- Do recursively
  - select two trees with smallest frequency at the root
  - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- Recursion ends when there is one tree
  - this is the Huffman coding tree

Example

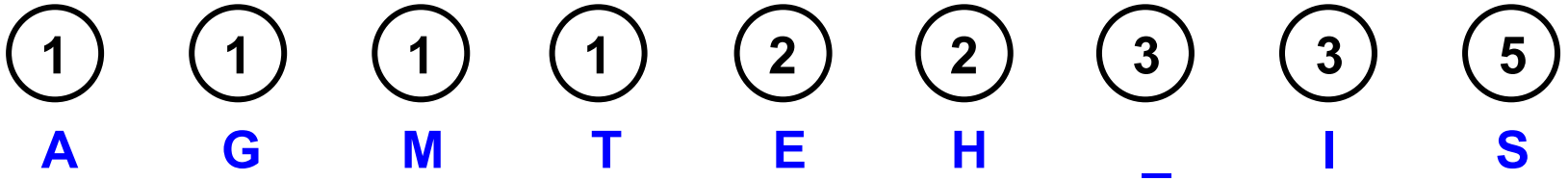
- Build the Huffman coding tree for the message

This is his message

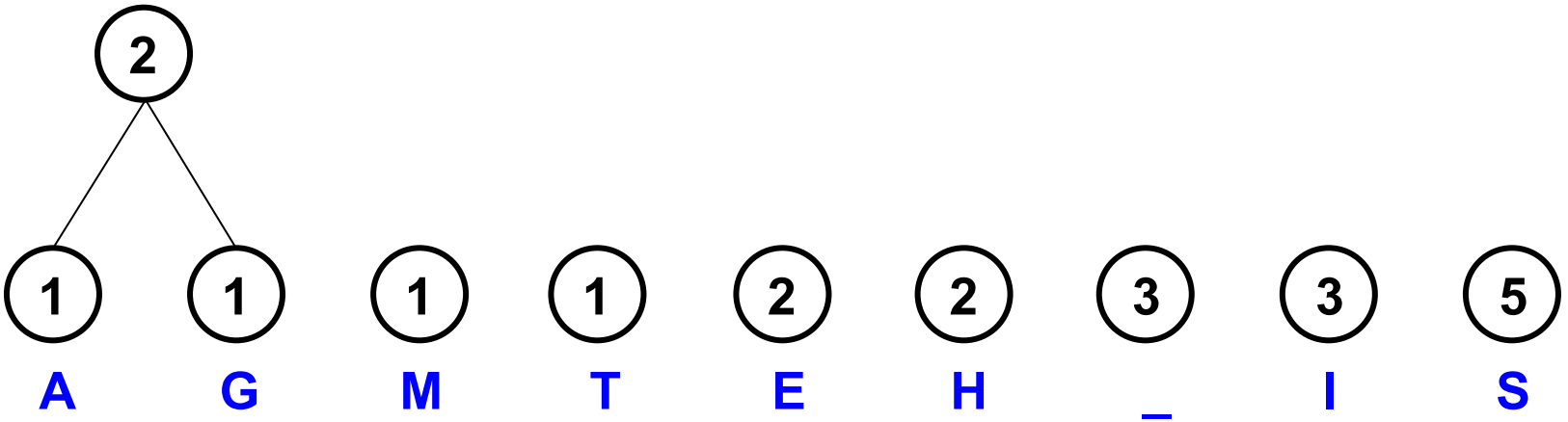
- Character frequencies

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

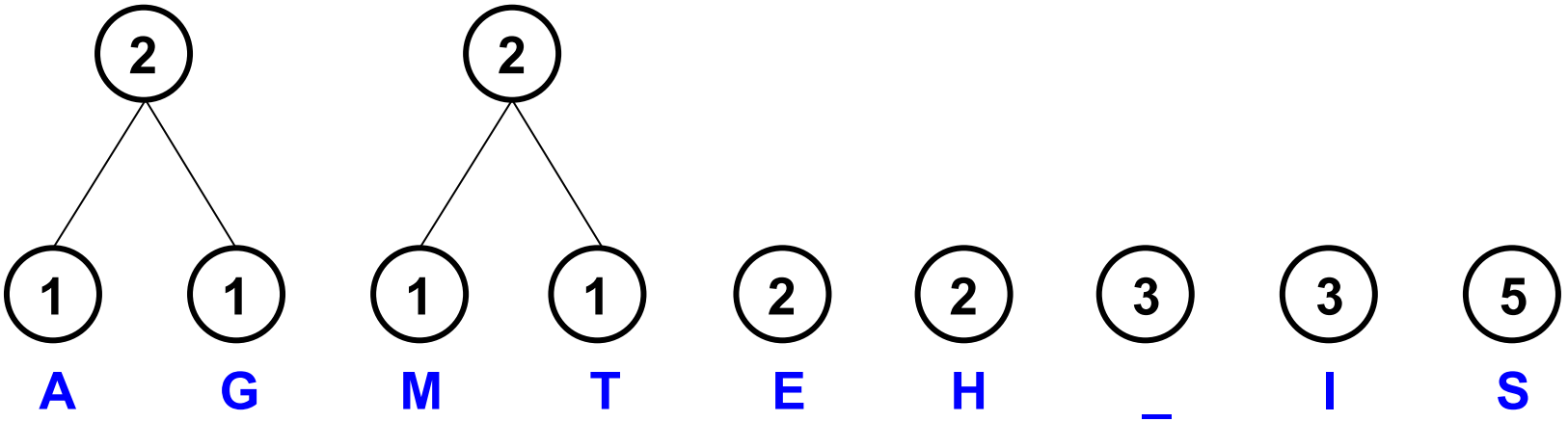
- Begin with forest of single trees



Step 1

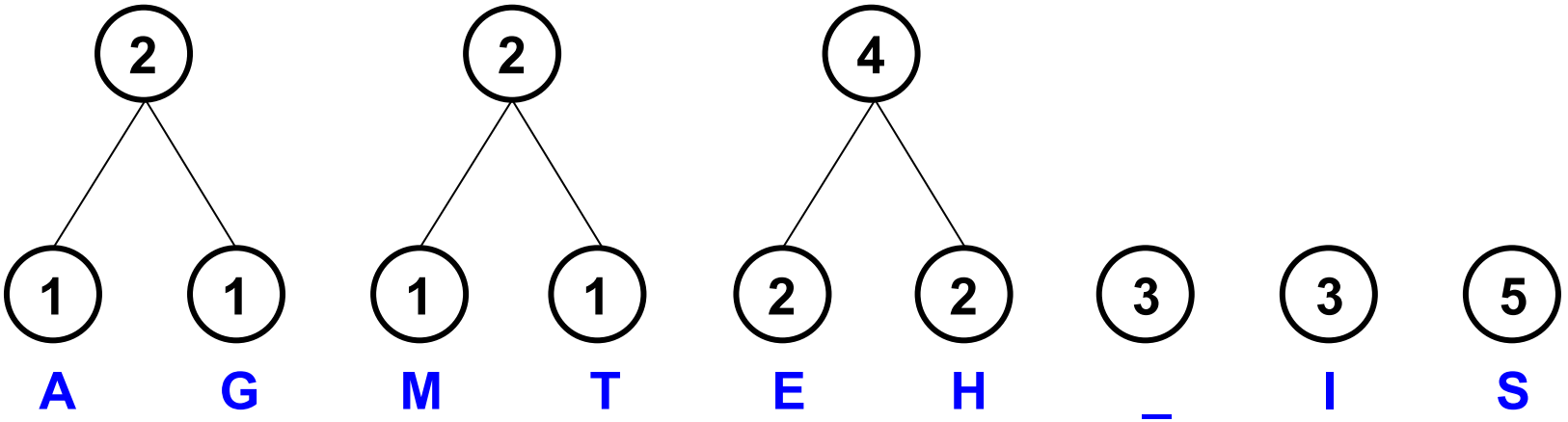


Step 2

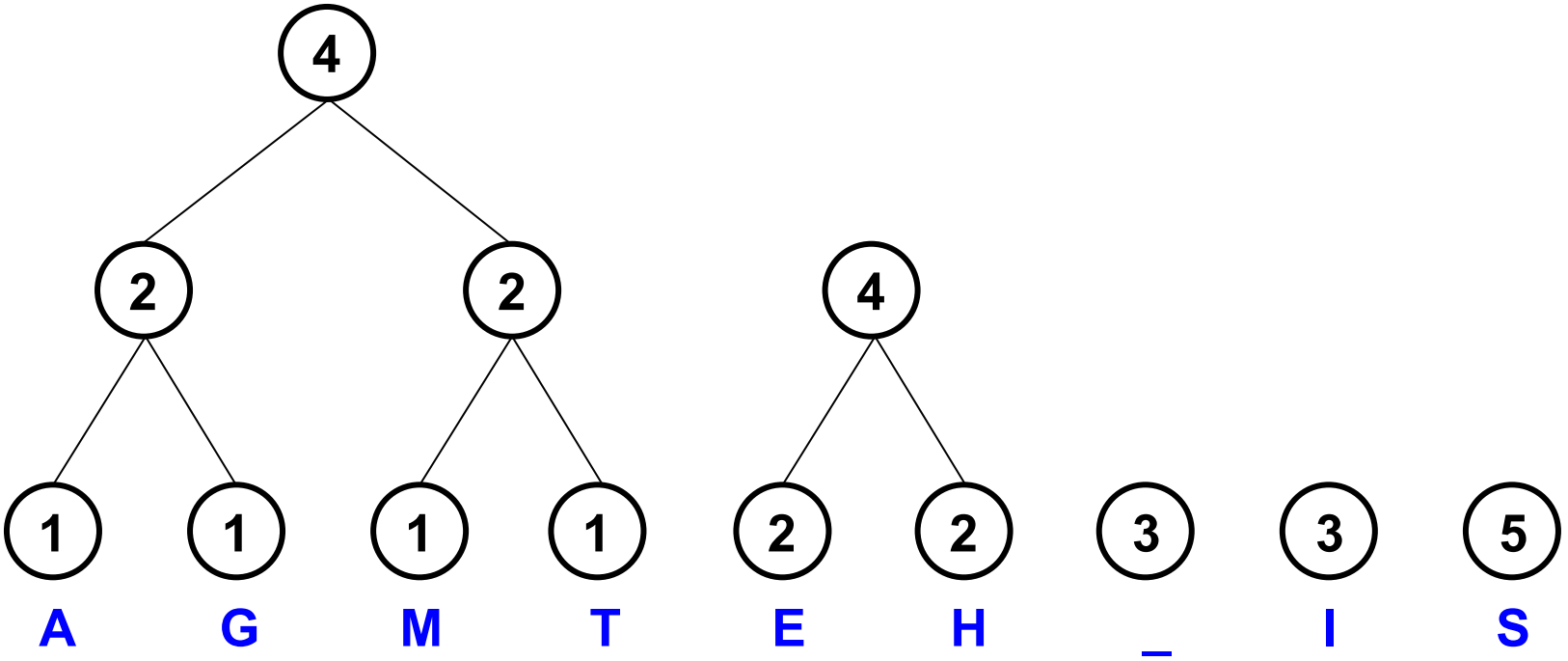


Step 3

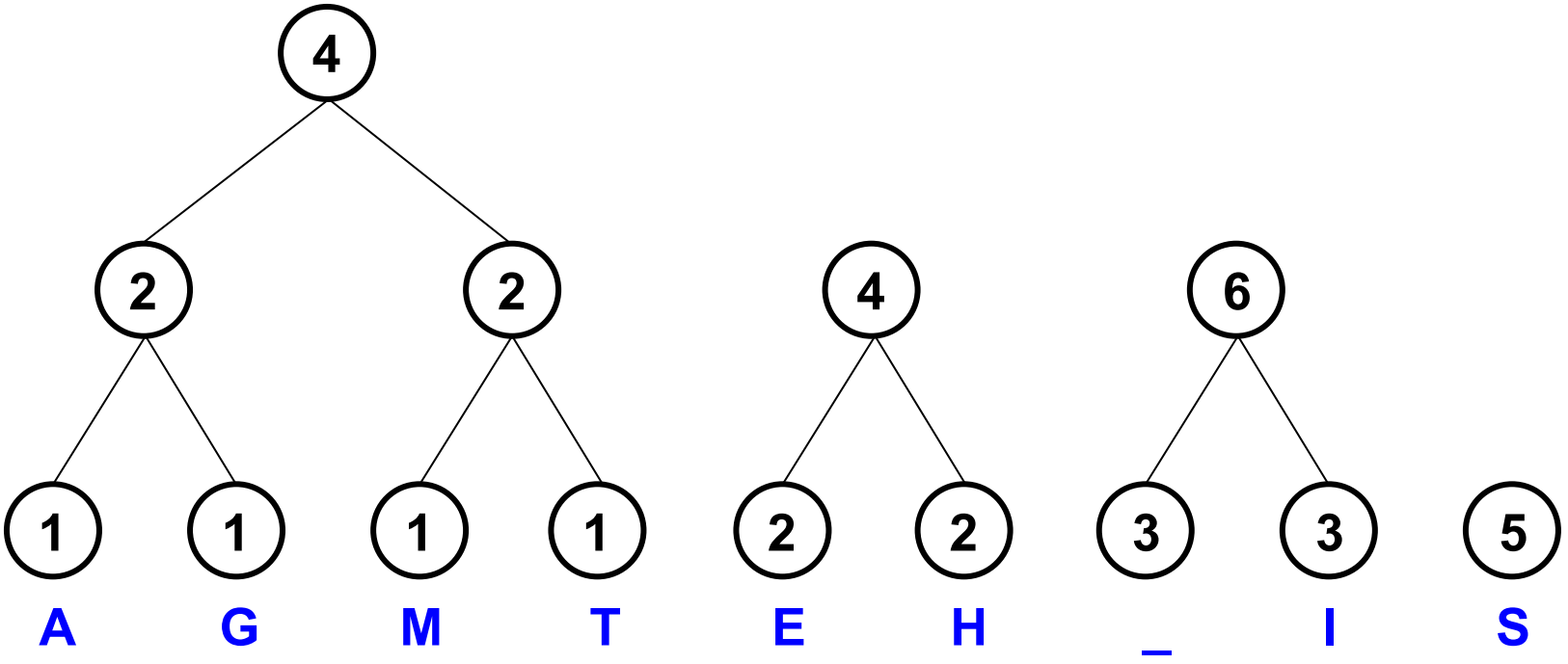
Multiple solutions are valid, as long as the two smallest nodes are selected and combined at each step.



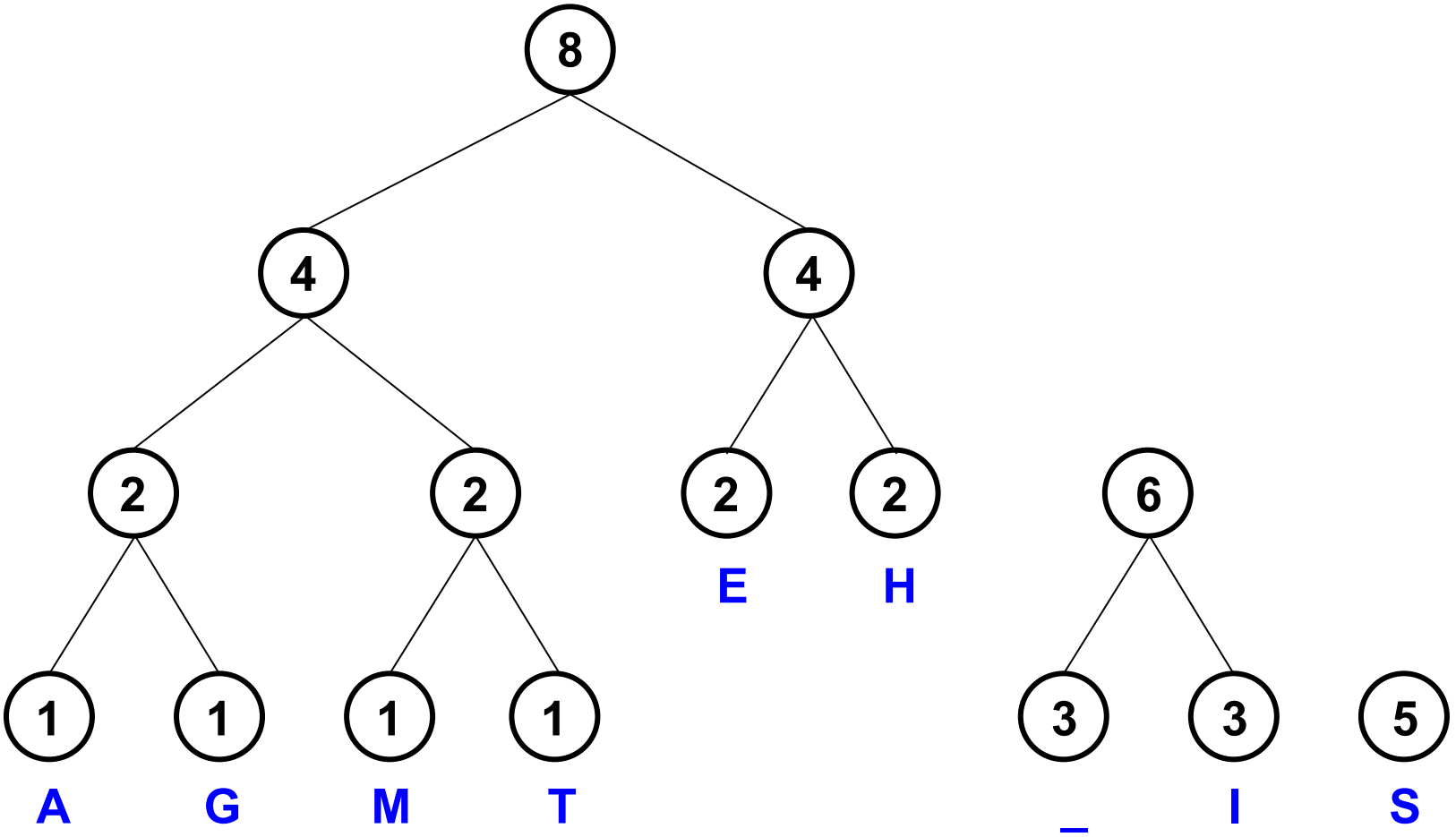
Step 4



Step 5

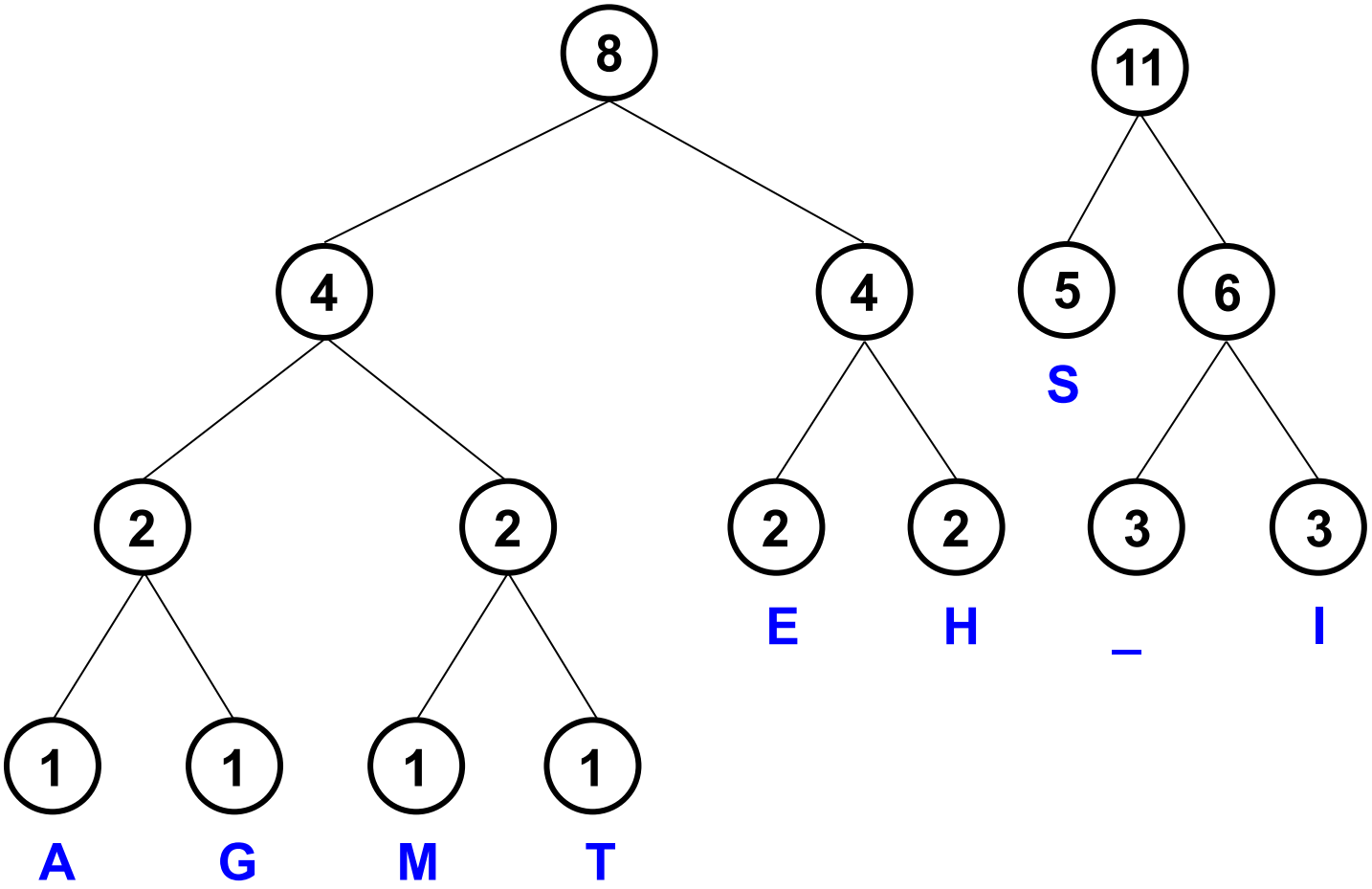


Step 6

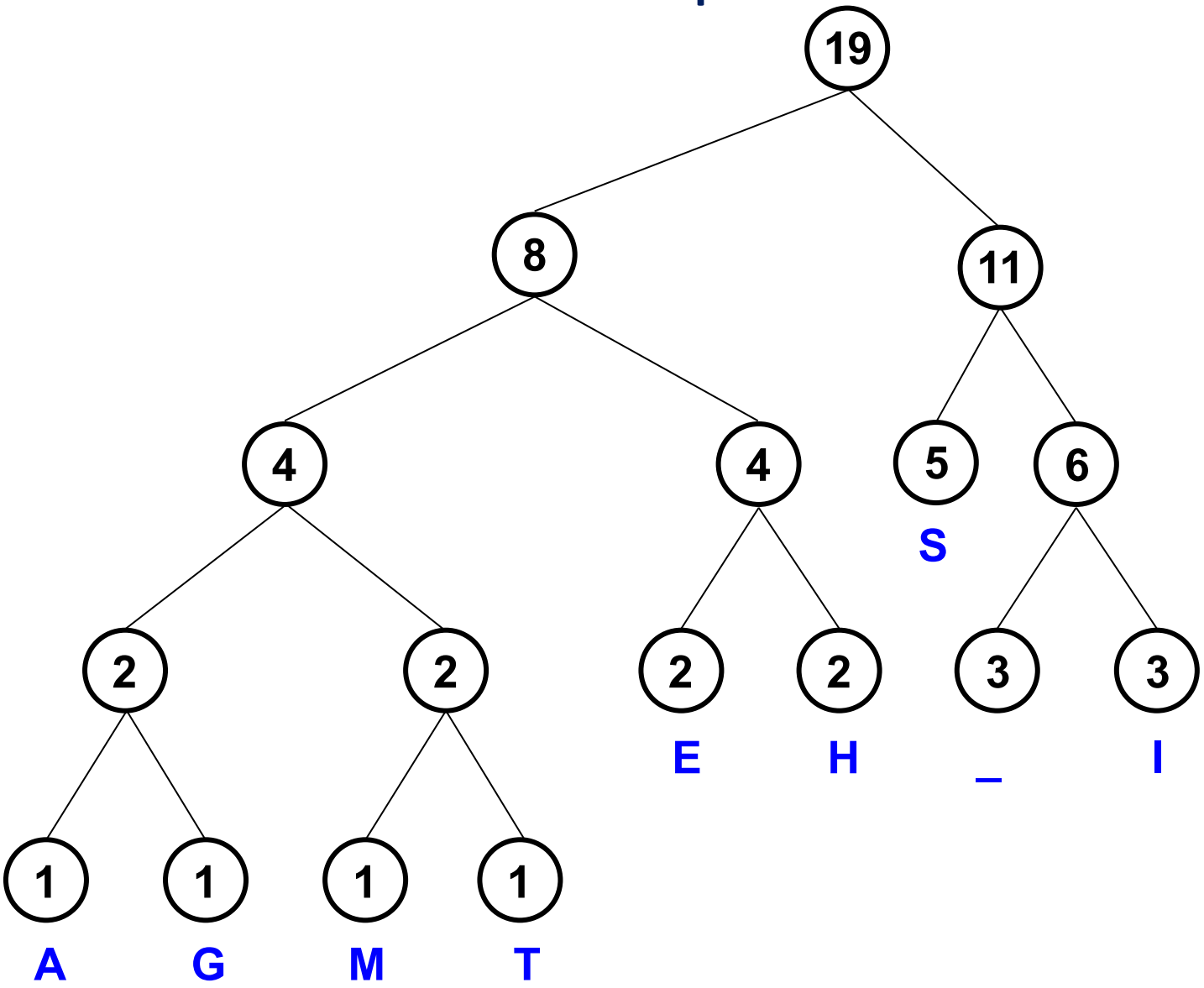


Step 7

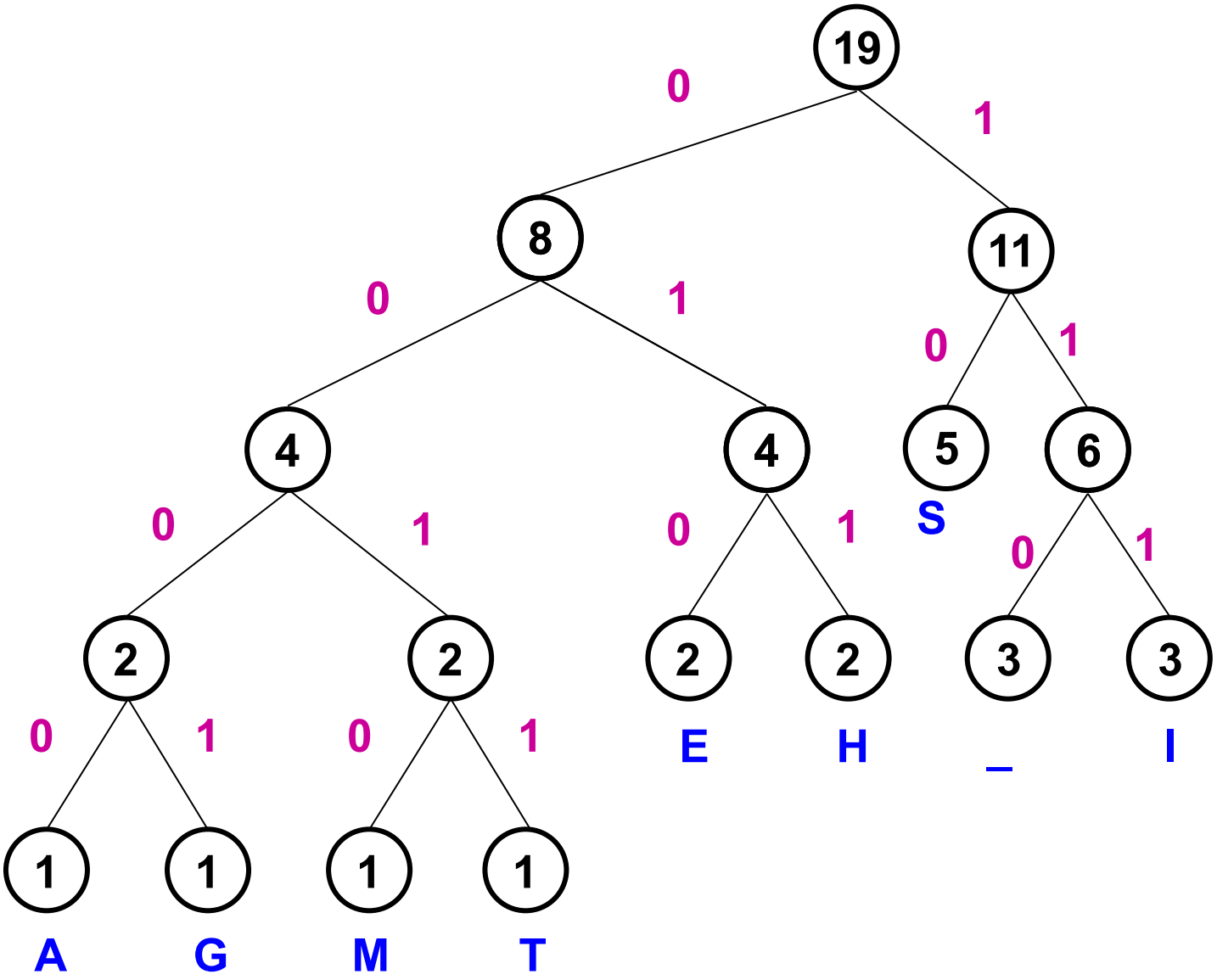
Left/right placement of root is arbitrary.



Step 8



Label edges



Huffman codes and encoded message

This is his message

S	10
E	010
H	011
_	110
I	111
A	0000
G	0001
M	0010
T	0011

00110111111011011111011001111111011000100101010000000001010

Implementation

```
HUFFMAN(C)    // C is a set of n characters with frequencies
1  n = |C|
2  Q = C      // min-priority queue Q, keyed on the freq attribute, takes O(n)
3  for i = 1 to n − 1      // Θ(n)
4      allocate a new node z
5      x = EXTRACT-MIN(Q)  // identify the two least-frequent objects to merge together
6      y = EXTRACT-MIN(Q) // O(log n), every min-heap operation
7      z.left = x
8      z.right = y
9      z.freq = x.freq + y.freq
10     INSERT(Q, z)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Time complexity: $O(n) + O(nlogn) \Rightarrow O(nlogn)$,
$O(nlogn)$ grows faster than $O(n)$.

Recap on Greedy Algorithms

Revisit Greedy Approaches in Graph Algorithms

Prim's for MST: Always select **the smallest cost edge** connected to already selected vertices – always a tree (no cycle).

Dijkstra's for SSSP: Once vertex with **shortest distance** is selected, relax for connected vertices.

```
MST-PRIM(G, w, r)
1  for each vertex u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = ∅
6  for each vertex u ∈ G.V
7      INSERT(Q, u)
8  while Q ≠ ∅
9      u = EXTRACT-MIN(Q)          // add u
10     for each vertex v in G.Adj[u]   // update
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

Dijkstra's: Correctness of Greedy Approach

Theorem: For any $u \in S$, the path P_u on the tree is the shortest path from s to u on G. (For all $u \in S, d(u) =$ true shortest path from s to u.)

Proof: Induction on $|S| = k$. Show it holds for $|S| = k + 1$.

Base Case: This is always true when $S = \{s\}$.

Inductive Step: Say v is the $(k+1)^{st}$ vertex that we add to S. Let (u, v) be last edge on P_v.

If P_v is not the shortest path, there is a shorter path P to S.

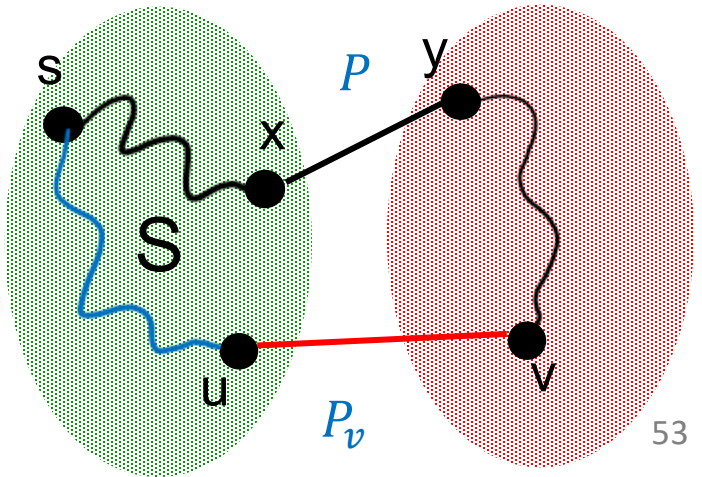
Consider the first time that P leaves S with edge (x, y).

So, $c(P) \geq d(x) + c_{x,y} \geq d(u) + c_{u,v} = d(v) = c(P_v)$.

P is the shorter path.

Due to the choice of v

A contradiction.



Compare DP and Greedy Approaches for SSSP

Algorithms	Bellman-Ford	Dijkstra
Technique	Dynamic Programming	Greedy
Relaxation	All edges, repeated	Greedy (smallest $d(u)$)
Order	No fixed order	Priority queue
Negative weights	Works	Fails

Summary

Technique	Dynamic Programming	Greedy
Decision	Explores **all** possible choices and often uses memoization or tabulation to store intermediate results. $\boldsymbol{d(v)} = \min_{(u,v)\in E}(\boldsymbol{d(u)} + w(u,v))$	Makes a sequence of choices, each of which looks the best at the moment. Once a choice is made, **it is not revisited!** $\boldsymbol{d(v)} = \min(\boldsymbol{d(u)} + w(u,v))$
Optimal Solution	Ensures a global optimal solution	May not guarantee the global optimal
Complexity	Slower but systematic due to overlapping	Faster in many cases, as it makes a single pass or a few passes over the data
Use Cases	Bellman-Ford's SSSP, rod cutting, matrix-chain multiplication, longest common sequence, Fibonacci series, 0/1 knapsack…	Prim's MST, Dijkstra's SSSP, activity selection, Huffman codes, fractional knapsack (not guarantee global optimal)…

Additional Materials

- NotebookLM shared at xSITe: here.
- Interesting reading: Greedy vs. Minimax Algorithms in AI Game Bots.
- Greedy vs. DP version for Activity Selection: here.
- Prim's MST for game development: here.
- Dijkstra's SSSP for turn-based tactical game: here.