Chapter 2: Greedy Algorithms

Advanced Algorithms

The greedy approach | Making change | Activity selection | Job sequencing | Fractional knapsack | MST and the cut property | Huffman coding

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_02_Greedy_Algorithms.html

Learning Outcomes

Approach

Define the greedy strategy; explain when the greedy-choice property and optimal substructure guarantee a correct result.

Classic problems

Solve making change, activity selection, job sequencing, and fractional knapsack greedily; give counterexamples where greedy fails.

Spanning trees

Explain why the cut property makes the greedy choices of Kruskal, Prim, and Borůvka correct; compare the three by selection rule, data structure, and running time.

Data compression

Build a Huffman tree, derive prefix-free codes, encode and decode a text, and compute the savings in bits.

Winning Step by Step

- **Greedy algorithms** proceed by choosing, at each step, the option that **looks best right now**.
- Like a **treasure hunter**: at every stage, take the option with the largest immediate gain.
- It works in stages: instead of looking at the problem as a whole, it focuses on **the best option at hand**.
- A decision is **never undone**; there is no backtracking.
- Like planning a trip by taking the road that looks shortest at every junction.

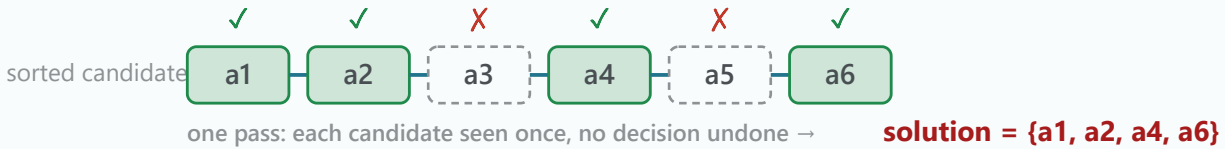
Gain and risk

Fast and simple; the code is usually short and clear.
However, it **does not guarantee an optimal solution**; in many cases it yields a good approximate one.

The Greedy Paradigm: General Template

```
Solution greedy(Candidate[] candidates) {
  Solution s = new Solution(); // empty
  sort(candidates);          // by criterion
  for (Candidate a : candidates) {
    if (s.isFeasible(a))     // constraints hold
      s.add(a);              // never undone
  }
  return s;
}
```

1. Define the set of **feasible solutions** (which choices keep the constraints satisfied?).
2. Pick a **greedy criterion**: cheapest, earliest finishing, most profitable, most frequent…
3. Choose the **locally best** candidate by the criterion; add it to the solution if feasible.
4. Repeat until the solution is complete.



The cost of most greedy algorithms comes from **sorting** or **priority queue** operations: typically **O(n log n)**.

When Is Greedy Correct?

Greedy-choice property

There is **at least one** optimal solution that contains the locally best choice. The first greedy choice never leads us "down the wrong path".

Optimal substructure

Once a choice is made, an optimal solution of the remaining subproblem, together with that choice, gives an optimal solution of the whole problem.

Proof technique: the exchange argument

Take any optimal solution O. If O does not contain the greedy choice, **exchange** one of its elements for the greedy choice; the solution stays feasible and does not get worse. So an optimal solution containing the greedy choice exists; induction extends this to every step.

If neither property holds, the greedy algorithm is **only a heuristic**: fast, but its result may be worse than optimal.

Advantages and Disadvantages

Advantages

+ Easy to understand and implement; short code.
+ Fast: usually a sort plus one pass, O(n log n).
+ Little memory: only the current solution is kept.
+ Optimal on the right problems (MST, Huffman, activity selection).
+ On hard problems it gives a good approximation or a starting point.

Disadvantages

– **No guarantee** of optimality (making change with {1, 3, 4}, 0/1 knapsack).
– Correctness **must be proved** separately for each problem.
– The criterion is critical: the wrong one gives the wrong result.
– Decisions are never undone; there is no chance to fix a mistake.

The Change-Making Problem (Coin Change)

Pay a given amount with the **fewest** coins.

Greedy criterion: take the **largest** coin not exceeding the remaining amount; repeat until the amount is zero.

```
// coins[] sorted in decreasing order
static int makeChange(int[] coins, int amount) {
  int count = 0;
  for (int p : coins) {
    while (amount >= p) {
      amount -= p;
      count++;
    }
  }
  return count;
}
```

Is it always correct?

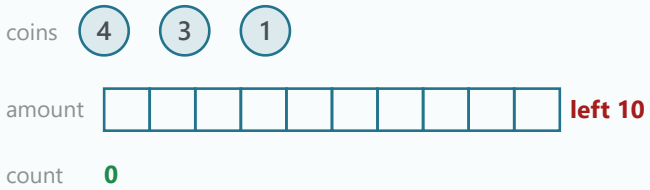
Yes for **canonical** coin systems such as {50, 25, 10, 5, 1}: 68 = 50 + 10 + 5 + 1 + 1 + 1.

With {4, 3, 1}, greedy pays 6 as 4 + 1 + 1 (3 coins); the optimum is 3 + 3 (2 coins). In general, **dynamic programming** is needed.

Time: O(k + count), where k is the number of coin types.

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
▶    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
6        amount -= p;
7        count++;
8      }
9    }
10   return count;
11 }
```



The coins {4, 3, 1} are sorted in decreasing order; the amount is **10**. This system is not canonical, so the greedy result may not be optimal.

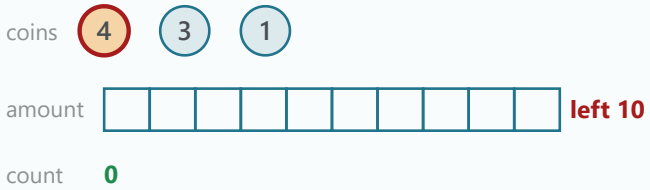
Step 1 / 9

VARIABLES

`p = –` `amount = 10` `count = 0`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
▶    for (int p : coins) {
▶      while (amount >= p) {
6        amount -= p;
7        count++;
8      }
9    }
10   return count;
11 }
```



p = 4: 10 ≥ 4 → this coin can be used.

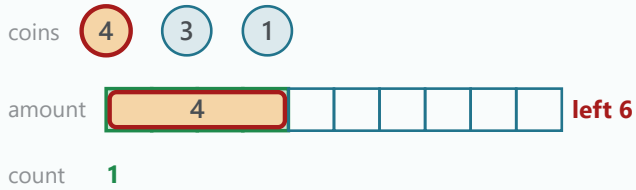
Step 2 / 9

VARIABLES

`p = 4` `amount = 10` `count = 0`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
▶        amount -= p;
▶        count++;
8      }
9    }
10   return count;
11 }
```



amount −= 4 → **6**, count = 1. The loop continues.

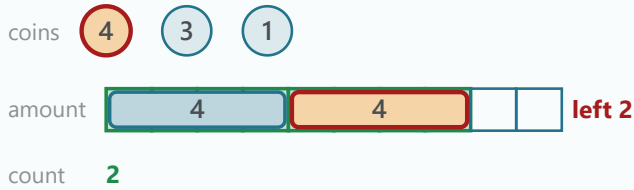
Step 3 / 9

VARIABLES

`p = 4` `amount = 6` `count = 1`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
▶        amount -= p;
▶        count++;
8      }
9    }
10   return count;
11 }
```



amount −= 4 → **2**, count = 2. 2 < 4: the while loop ends.

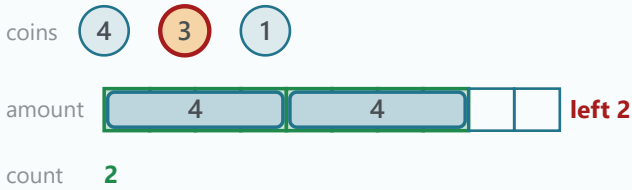
Step 4 / 9

VARIABLES

`p = 4` `amount = 2` `count = 2`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
▶    for (int p : coins) {
▶      while (amount >= p) {
6        amount -= p;
7        count++;
8      }
9    }
10   return count;
11 }
```



p = 3: 2 < 3 → the 3-coin is **never used**; move on to the next coin.

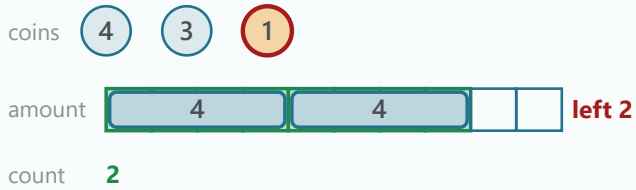
Step 5 / 9

VARIABLES

`p = 3` `amount = 2` `count = 2`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
▶    for (int p : coins) {
▶      while (amount >= p) {
6        amount -= p;
7        count++;
8      }
9    }
10   return count;
11 }
```



p = 1: 2 ≥ 1 → this coin can be used.

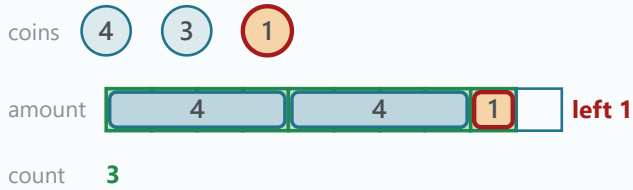
Step 6 / 9

VARIABLES

`p = 1` `amount = 2` `count = 2`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
▶        amount -= p;
▶        count++;
8      }
9    }
10   return count;
11 }
```



amount −= 1 → **1**, count = 3. The loop continues.

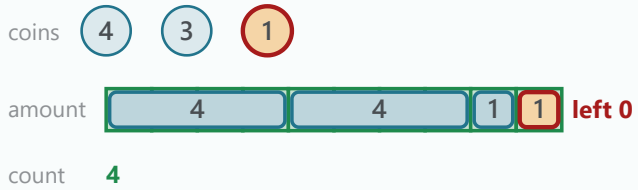
Step 7 / 9

VARIABLES

`p = 1` `amount = 1` `count = 3`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
▶        amount -= p;
▶        count++;
8      }
9    }
10   return count;
11 }
```



amount −= 1 → **0**, count = 4. 0 < 1: the while loop ends.

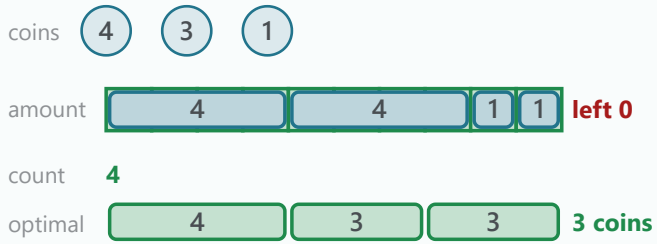
Step 8 / 9

VARIABLES

`p = 1` `amount = 0` `count = 4`

makeChange({4, 3, 1}, 10): Code Trace

```
1  // coins[] sorted in decreasing order
2  static int makeChange(int[] coins, int amount)
   {
3    int count = 0;
4    for (int p : coins) {
5      while (amount >= p) {
6        amount -= p;
7        count++;
8      }
9    }
   ▶ return count;
11 }
```



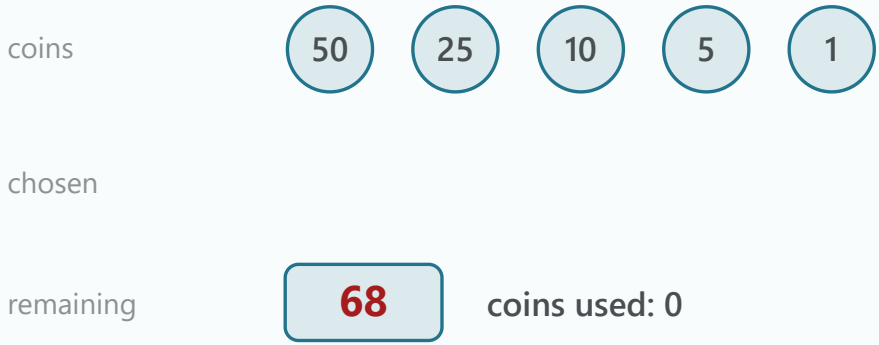
Greedy returns 4 + 4 + 1 + 1 = **4 coins**, yet 4 + 3 + 3 = **3 coins** suffice (found by DP). Taking two 4-coins first blocked the use of the 3-coins.

Step 9 / 9

VARIABLES

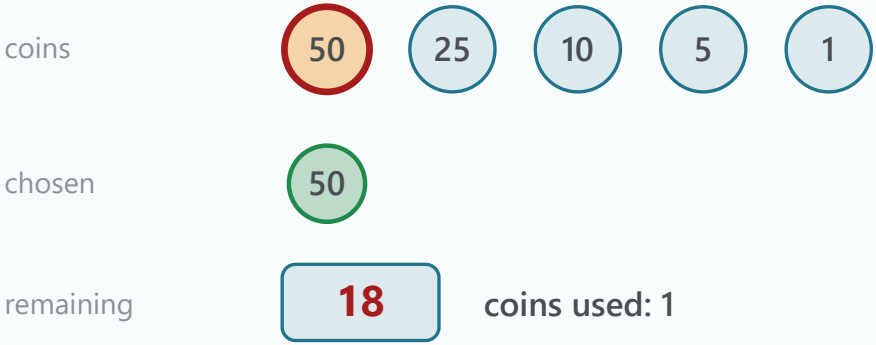
`p = –` `amount = 0` `count = 4`

Making Change: Step by Step



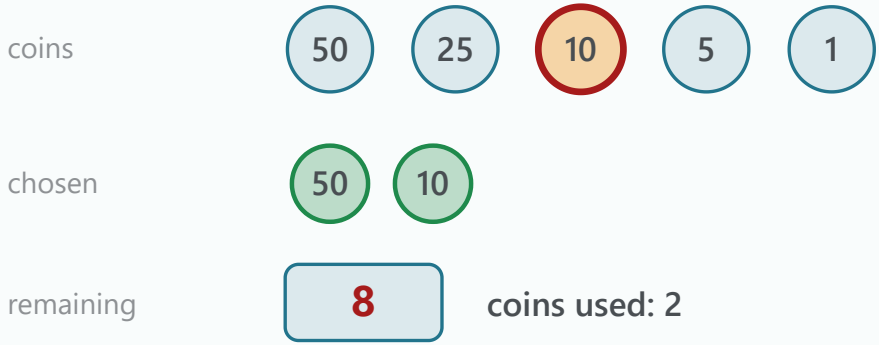
Example 1: coins {50, 25, 10, 5, 1}, amount **68**. Start with the largest coin.

Making Change: Step by Step



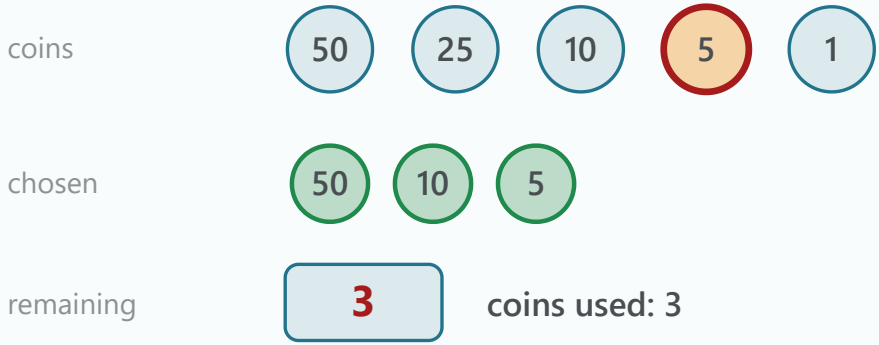
The largest coin that fits is **50** → 68 − 50 = **18** left.

Making Change: Step by Step



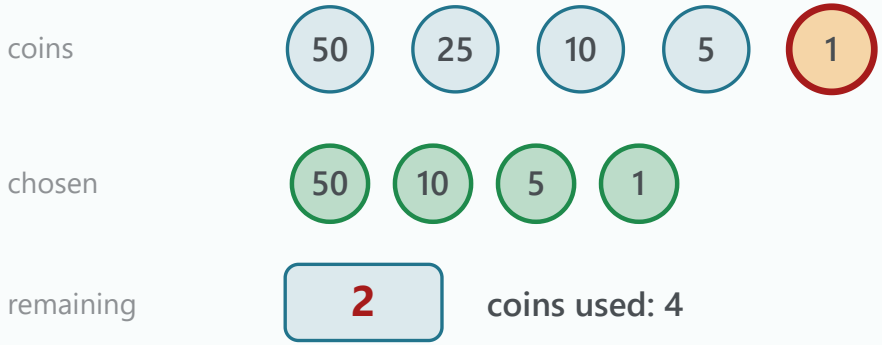
The largest coin that fits is **10** → 18 − 10 = **8** left.

Making Change: Step by Step



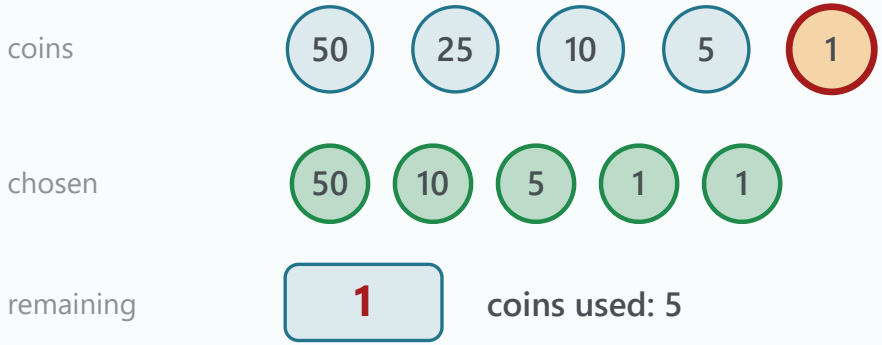
The largest coin that fits is **5** → 8 − 5 = **3** left.

Making Change: Step by Step



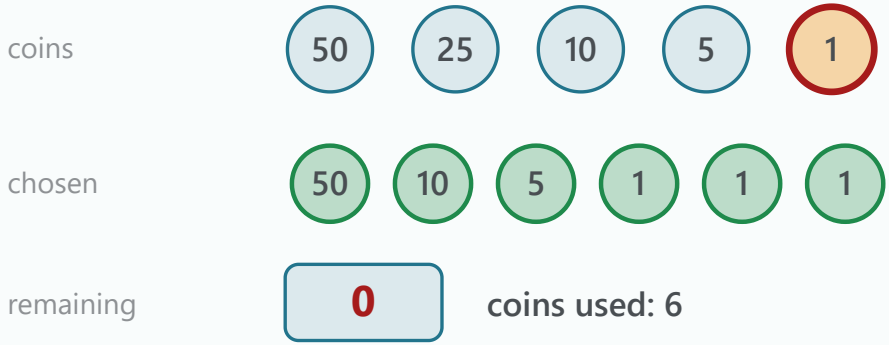
The largest coin that fits is **1** → 3 − 1 = **2** left.

Making Change: Step by Step



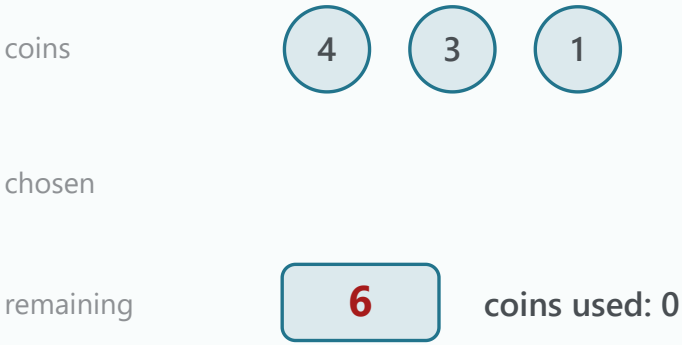
The largest coin that fits is **1** → 2 − 1 = **1** left.

Making Change: Step by Step



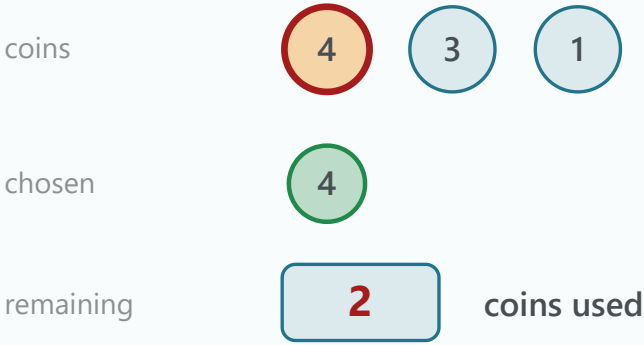
The largest coin that fits is **1** → 1 − 1 = **0** left. **6 coins** in total — optimal for this coin system.

Making Change: Step by Step



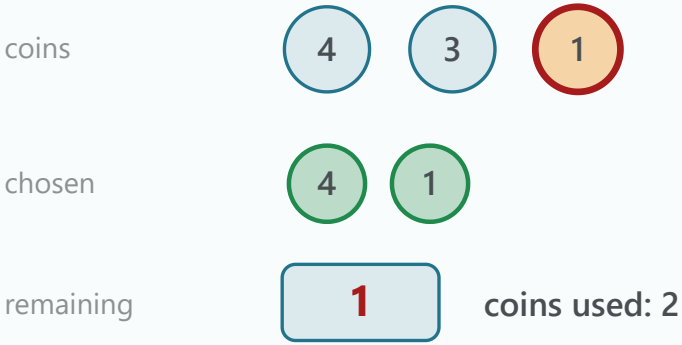
Example 2: coins {4, 3, 1}, amount **6**. Start with the largest coin.

Making Change: Step by Step



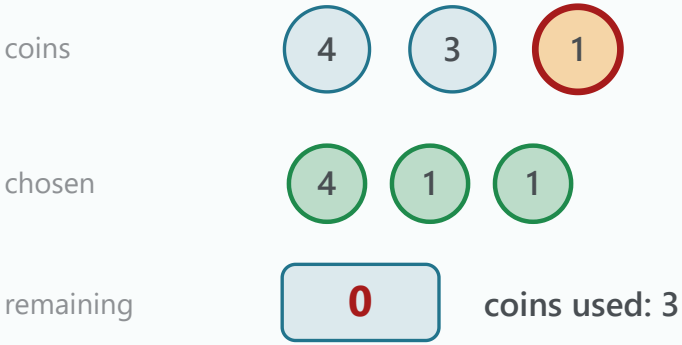
The largest coin that fits is **4** → 6 − 4 = **2** left.

Making Change: Step by Step



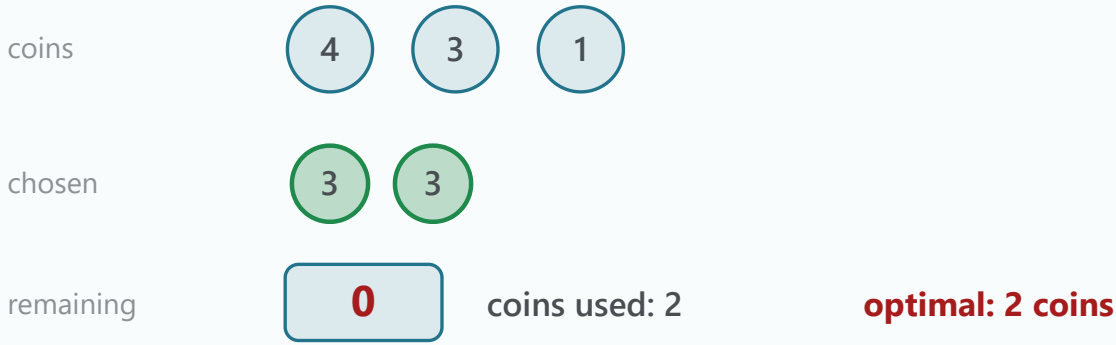
The largest coin that fits is **1** → 2 − 1 = **1** left.

Making Change: Step by Step



The largest coin that fits is **1** → 1 − 1 = **0** left.

Making Change: Step by Step



Greedy gave 4 + 1 + 1 = **3 coins**, yet **3 + 3 = 2 coins** suffice. The locally best choice (4) missed the global optimum.

Try It: Greedy vs. Optimal Change

Coins `1, 3, 4` Amount `6` **compute** **TL coins** **1, 7, 10**

Greedy

4 1 1

3 coins

Optimal (DP)

3 3

2 coins

Greedy is **not optimal**: 3 coins, optimal 2 coins (3 + 3).

The optimum is found by dynamic programming: minCoins[t] = 1 + min minCoins[t − p]. Coins are integers from 1 to 999, at most 8 kinds; the amount is at most 999. Enter = compute.

The Greedy Approach vs. Divide and Conquer

	Greedy	Divide and conquer
Decision	a single locally best choice per step; never undone	split by a fixed rule; no choice is made
Subproblems	one subproblem remains after the choice	several independent subproblems
Correctness	greedy-choice property + optimal substructure must be proved	depends on the correctness of the combine step
Cost	usually O(n log n): sort + one pass	via a recurrence and the master theorem
Example	activity selection, Kruskal, Huffman	merge sort, Karatsuba

Both shrink the problem: divide and conquer solves **all** the pieces and combines them; greedy makes **one** choice and continues with the single remaining subproblem.

Activity Selection

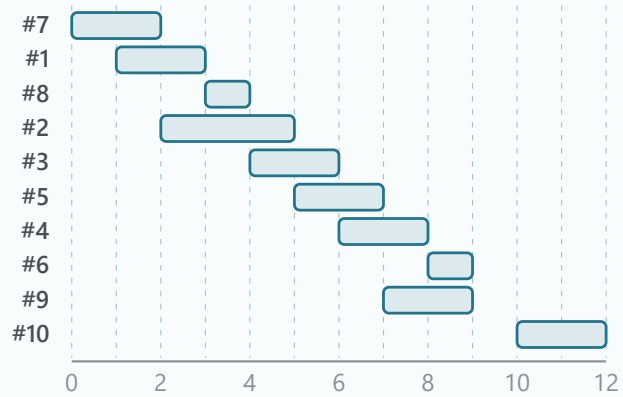
- Each activity has a **start** time s_i and a **finish** time f_i; only one activity can run at a time (one hall).
- Goal: select the **maximum number** of mutually compatible activities. $[s_i, f_i)$ and $[s_j, f_j)$ are compatible $\Leftrightarrow s_j \geq f_i$ or $s_i \geq f_j$.
- **Greedy criterion:** select the activity that **finishes earliest**, discard those that overlap it, and repeat with the rest.

Activity	1	2	3	4	5	6	7	8	9	10
start	1	2	4	6	5	8	0	3	7	10
finish	3	5	6	8	7	9	2	4	9	12

Time: sorting O(n log n) + one pass O(n). If the activities are already sorted by finish time, **O(n)**.

select(e): Code Trace

```
1  static void select(Activity[] e) {
▶    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
6      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



The activities are sorted **by finish time** (top to bottom). Dashed line: finish of the last selected activity.

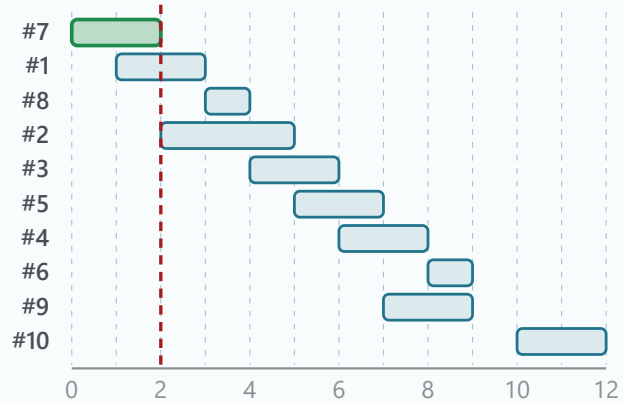
Step 1 / 11

VARIABLES

`i = –` `j = –` `e[i].finish = –` `output = –`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
▶    int i = 0;                    // last selected
▶    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
6      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



The earliest-finishing activity (#7, [0, 2)) is **always** selected.

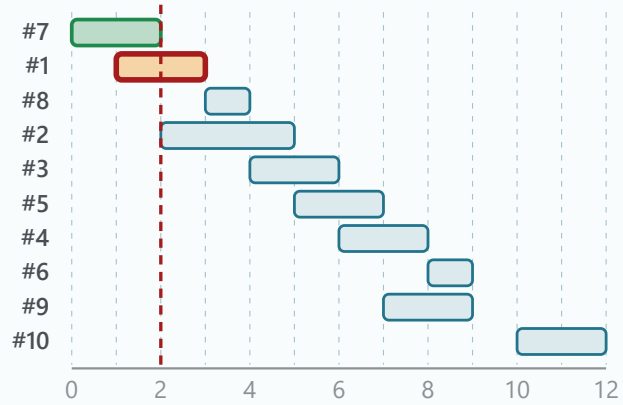
Step 2 / 11

VARIABLES

`i = 0` `j = –` `e[i].finish = 2` `output = 7`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#1 [1, 3): start 1 < 2 → overlaps, **skipped**.

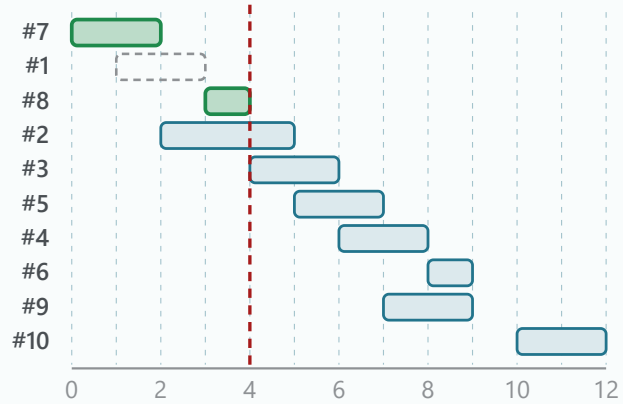
Step 3 / 11

VARIABLES

`i = 0` `j = 1` `e[i].finish = 2` `output = 7`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#8 [3, 4): start 3 ≥ 2 → **selected**, i = 2.

Step 4 / 11

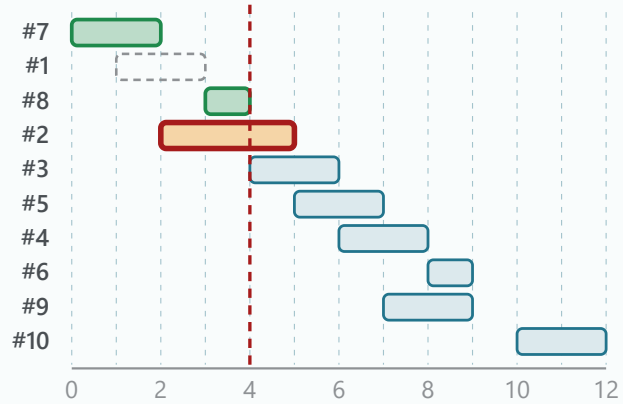
VARIABLES

`i = 2` `j = 2` `e[i].finish = 4`

`output = 7 8`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#2 [2, 5): start 2 < 4 → overlaps, **skipped**.

Step 5 / 11

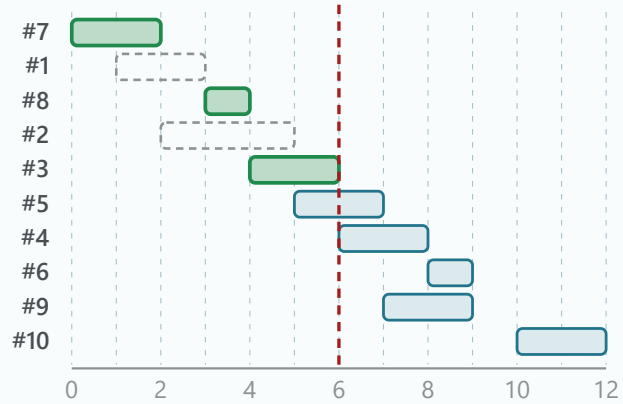
VARIABLES

`i = 2` `j = 3` `e[i].finish = 4`

`output = 7 8`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#3 [4, 6): start 4 ≥ 4 → **selected**, i = 4.

Step 6 / 11

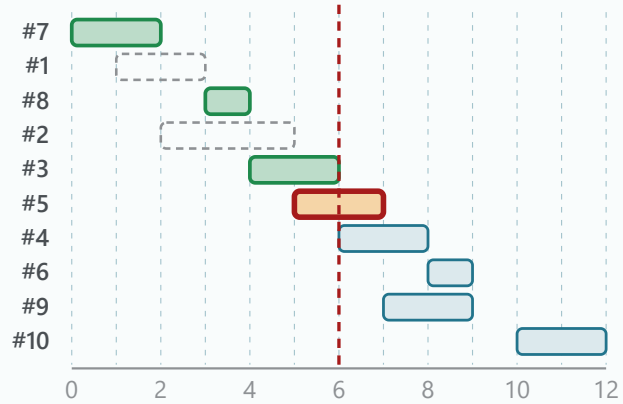
VARIABLES

`i = 4` `j = 4` `e[i].finish = 6`

`output = 7 8 3`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#5 [5, 7): start 5 < 6 → overlaps, **skipped**.

Step 7 / 11

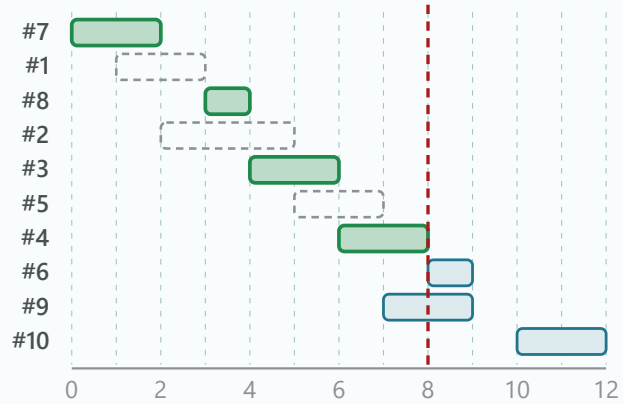
VARIABLES

`i = 4` `j = 5` `e[i].finish = 6`

`output = 7 8 3`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                    // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#4 [6, 8): start 6 ≥ 6 → **selected**, i = 6.

Step 8 / 11

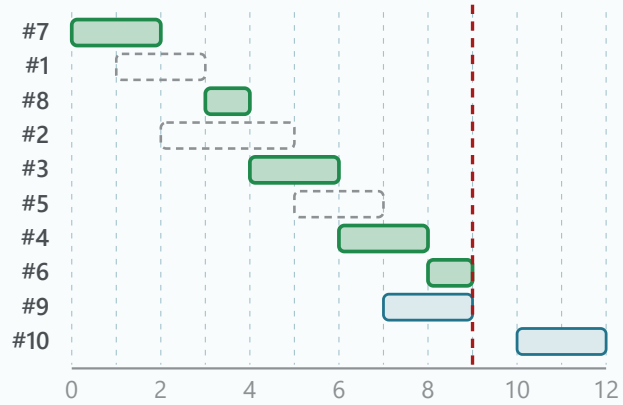
VARIABLES

`i = 6` `j = 6` `e[i].finish = 8`

`output = 7 8 3 4`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                    // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#6 [8, 9): start 8 ≥ 8 → **selected**, i = 7.

Step 9 / 11

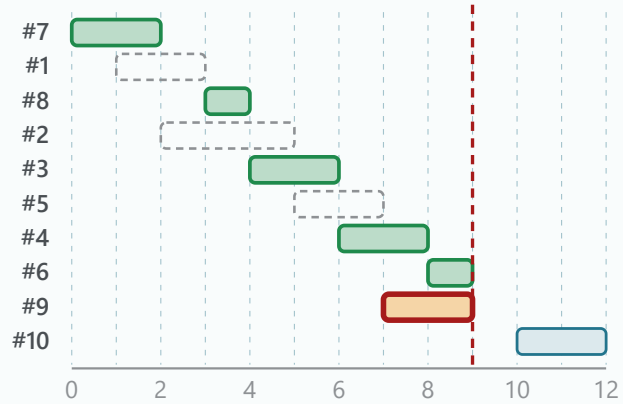
VARIABLES

`i = 7` `j = 7` `e[i].finish = 9`

`output = 7 8 3 4 6`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                  // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
7        System.out.print(e[j].id + " ");
8        i = j;
9      }
10   }
11 }
```



#9 [7, 9): start 7 < 9 → overlaps, **skipped**.

Step 10 / 11

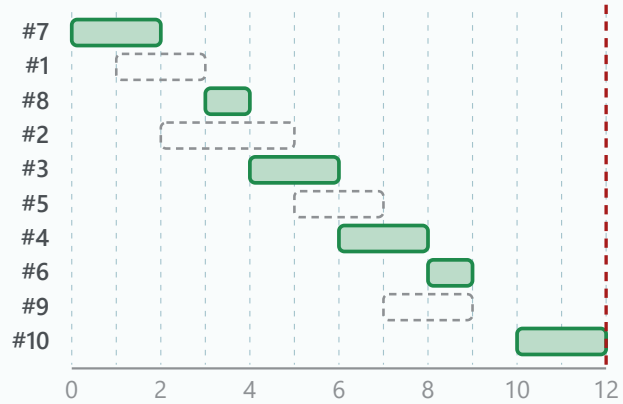
VARIABLES

`i = 7` `j = 8` `e[i].finish = 9`

`output = 7 8 3 4 6`

select(e): Code Trace

```
1  static void select(Activity[] e) {
2    Arrays.sort(e, (x, y) -> x.finish -
   y.finish);
3    int i = 0;                    // last selected
4    System.out.print(e[i].id + " ");
5    for (int j = 1; j < e.length; j++) {
▶      if (e[j].start >= e[i].finish) {
▶        System.out.print(e[j].id + " ");
▶        i = j;
9      }
10   }
11 }
```



#10 [10, 12): start 10 ≥ 9 → **selected**, i = 9. Result: **6 activities** (7, 8, 3, 4, 6, 10).

Step 11 / 11

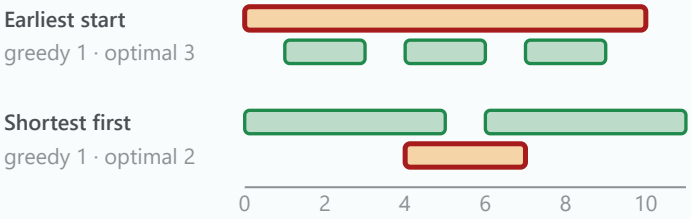
VARIABLES

`i = 9` `j = 9` `e[i].finish = 12`

`output = 7 8 3 4 6 10`

Why Earliest Finish?

Other natural criteria **do not guarantee** an optimal result (orange: greedy's choice, green: optimal solution):



The "fewest overlaps" criterion has a counterexample too.

Exchange argument

Let a_1 be the earliest-finishing activity and O an optimal solution. If the first activity of O is $b \neq a_1$, replace b with a_1: since $f(a_1) \leq f(b)$, it overlaps no other activity of O, and the count stays the same.

So some optimal solution contains a_1. After discarding the activities that overlap a_1, the remaining problem has the same structure (optimal substructure).

Job Sequencing with Deadlines

- Each job takes **1 unit** of time and has a **deadline** and a **profit**.
- The profit is earned only if the job finishes by its deadline; one job runs at a time.
- Goal: maximize the **total profit**.
- **Greedy criterion:** take the jobs in **decreasing** order of profit; put each in the **latest free** slot before its deadline; if no slot is free, skip the job.
- Choosing the latest slot leaves the early slots for jobs with closer deadlines.

Job	deadline	profit
1	4	20
2	1	10
3	1	40
4	1	30
5	3	50
6	2	30

Time: sorting $O(n \log n)$ + slot search $O(n \cdot d_{max})$ → worst case $\mathbf{O(n^2)}$. With Union-Find, the slot search drops to nearly $O(n)$.

Sorting jobs by **deadline** is wrong: for (profit 10, deadline 1) and (profit 40, deadline 1), deadline order earns 10, profit order earns 40. Decreasing profit order is optimal: the feasible job sets form a **matroid**.

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
▶    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
▶    int[] slot = new int[n];
▶    Arrays.fill(slot, -1);
▶    int total = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, jobs[i].deadline) -
   1;
8      for (; j >= 0; j--) {
9        if (slot[j] == -1) {
10         slot[j] = jobs[i].id;
11         total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit

Job 5	Job 3	Job 4	Job 6	Job 1	Job 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
profit 50	profit 40	profit 30	profit 30	profit 20	profit 10

slot (time slot)

1	2	3	4	5	6
—	—	—	—	—	—

The jobs are sorted in **decreasing order of profit** (Job 4 and Job 6, with equal profit, keep their original order). All slots are empty (−1).

Step 1 / 8

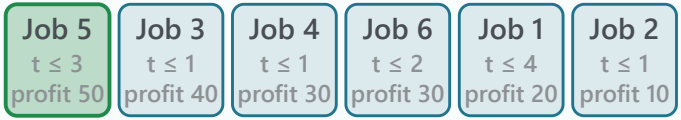
VARIABLES

`i = –` `j = –` `total = 0`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, jobs[i].deadline) -
   1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = jobs[i].id;
▶          total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit



slot (time slot)



Job 5 (deadline 3, profit 50): the latest feasible slot **3** is free → placed. total = 50.

Step 2 / 8

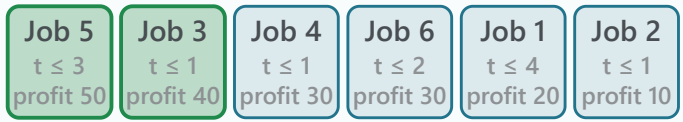
VARIABLES

`i = 0` `j = 2` `total = 50`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, jobs[i].deadline) -
   1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = jobs[i].id;
▶          total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit



slot (time slot)



Job 3 (deadline 1, profit 40): the latest feasible slot **1** is free → placed. total = 90.

Step 3 / 8

VARIABLES

`i = 1` `j = 0` `total = 90`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, jobs[i].deadline) -
   1;
8      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
10         slot[j] = jobs[i].id;
11         total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit

Job 5	Job 3	Job 4	Job 6	Job 1	Job 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
profit 50	profit 40	profit 30	profit 30	profit 20	profit 10

slot (time slot)

1	2	3	4	5	6
Job 3	—	Job 5	—	—	—

Job 4 (deadline 1, profit 30): slot 1 is **taken** (Job 3) and there is no earlier slot → the job is **skipped**.

Step 4 / 8

VARIABLES

`i = 2` `j = 0` `total = 90`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, jobs[i].deadline) -
   1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = jobs[i].id;
▶          total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit



slot (time slot)



Job 6 (deadline 2, profit 30): the latest feasible slot **2** is free → placed. total = 120.

Step 5 / 8

VARIABLES

`i = 3` `j = 1` `total = 120`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
▶      int j = Math.min(n, jobs[i].deadline) -
   1;
▶      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
▶          slot[j] = jobs[i].id;
▶          total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit



slot (time slot)



Job 1 (deadline 4, profit 20): the latest feasible slot **4** is free → placed. total = 140.

Step 6 / 8

VARIABLES

`i = 4` `j = 3` `total = 140`

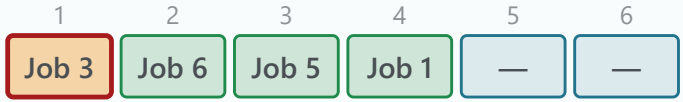
jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, jobs[i].deadline) -
   1;
8      for (; j >= 0; j--) {
▶        if (slot[j] == -1) {
10         slot[j] = jobs[i].id;
11         total += jobs[i].profit;
12         break;
13       }
14     }
15   }
16   return total;
17 }
```

jobs sorted by profit



slot (time slot)



Job 2 (deadline 1, profit 10): slot 1 is **taken** (Job 3) and there is no earlier slot → the job is **skipped**.

Step 7 / 8

VARIABLES

`i = 5` `j = 0` `total = 140`

jobSequencing(jobs, 6): Code Trace

```
1  static int jobSequencing(Job[] jobs, int n)
   {
2    Arrays.sort(jobs, (a, b) -> b.profit -
   a.profit);
3    int[] slot = new int[n];
4    Arrays.fill(slot, -1);
5    int total = 0;
6    for (int i = 0; i < n; i++) {
7      int j = Math.min(n, jobs[i].deadline) -
   1;
8      for (; j >= 0; j--) {
9        if (slot[j] == -1) {
10         slot[j] = jobs[i].id;
11         total += jobs[i].profit;
12         break;
13       }
14     }
15   }
▶    return total;
17 }
```

jobs sorted by profit

Job 5	Job 3	Job 4	Job 6	Job 1	Job 2
t ≤ 3	t ≤ 1	t ≤ 1	t ≤ 2	t ≤ 4	t ≤ 1
profit 50	profit 40	profit 30	profit 30	profit 20	profit 10

slot (time slot)

1	2	3	4	5	6
Job 3	Job 6	Job 5	Job 1	—	—

Result: slots 1–4 = Job 3, Job 6, Job 5, Job 1 → **total profit 140**. Job 4 and Job 2 cannot be scheduled.

Step 8 / 8

VARIABLES

`i = 6` `j = –` `total = 140`

Fractional Knapsack

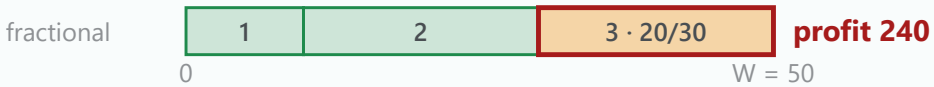
Capacity **W = 50**. If items can be split, the greedy criterion is: take the one with the largest **profit/weight ratio**.

Item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4

Fractional: items 1 + 2 in full, 20/30 of item 3 → 60 + 100 + 80 = **240** (optimal). Time O(n log n).

Why optimal?

Each unit of the highest-ratio item earns at least as much as any other option of the same weight. If an optimal solution does not take as much of this item as it can, swap in an equal weight of it for another item; the profit does not drop (exchange argument). If items cannot be split (0/1 knapsack), the ratio criterion does not guarantee the optimum; that problem is solved with **dynamic programming**.



fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4

knapsack W = 50



cap = 50 **profit = 0**

Items sorted by decreasing profit/weight ratio: 1 (6), 2 (5), 3 (4). The knapsack is empty; remaining capacity **50**.

Step 1 / 8

```
1  static double fractional(Item[] s, int W) {
▶    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
▶    double profit = 0;
▶    int cap = W;
5    for (Item x : s) {
6      if (x.weight <= cap) {           // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
9      } else {                         // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = –` `cap = 50` `profit = 0`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4



Item 1: weight 10 ≤ cap 50 → fits **entirely**.

Step 2 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
▶    for (Item x : s) {
▶      if (x.weight <= cap) {          // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
9      } else {                        // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 1` `cap = 50` `profit = 0`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4



cap −= 10 → 40, profit += 60 → **60**.

Step 3 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
5    for (Item x : s) {
6      if (x.weight <= cap) {           // fits entirely
▶        cap -= x.weight;
▶        profit += x.profit;
9      } else {                         // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 1` `cap = 40` `profit = 60`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4



Item 2: weight 20 ≤ cap 40 → fits **entirely**.

Step 4 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
▶    for (Item x : s) {
▶      if (x.weight <= cap) {           // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
9      } else {                         // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 2` `cap = 40` `profit = 60`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4

knapsack W = 50



cap = 20 **profit = 160**

cap −= 20 → 20, profit += 100 → **160**.

Step 5 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
5    for (Item x : s) {
6      if (x.weight <= cap) {           // fits entirely
▶        cap -= x.weight;
▶        profit += x.profit;
9      } else {                         // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 2` `cap = 20` `profit = 160`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4



Item 3: weight 30 > cap 20 → does not fit entirely, **else** branch.

Step 6 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
▶    for (Item x : s) {
▶      if (x.weight <= cap) {          // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
▶      } else {                        // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 3` `cap = 20` `profit = 160`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4

knapsack W = 50



knapsack full **profit = 240**

For the remaining 20 units, **20/30** of item 3: 120 · 20/30 = 80 → profit = **240**. The knapsack is full, `break`.

Step 7 / 8

```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
5    for (Item x : s) {
6      if (x.weight <= cap) {           // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
9      } else {                         // fractional part
▶        profit += x.profit * (double) cap / x.weight;
▶        break;
12     }
13   }
14   return profit;
15 }
```

VARIABLES

`x = 3` `cap = 20` `profit = 240`

fractional(s, 50): Code Trace

item	profit	weight	ratio
1	60	10	6
2	100	20	5
3	120	30	4

knapsack W = 50



knapsack full **profit = 240**

Result **240**: optimal for the fractional problem. The cost comes from sorting: **O(n log n)**; the scan is O(n).

Step 8 / 8

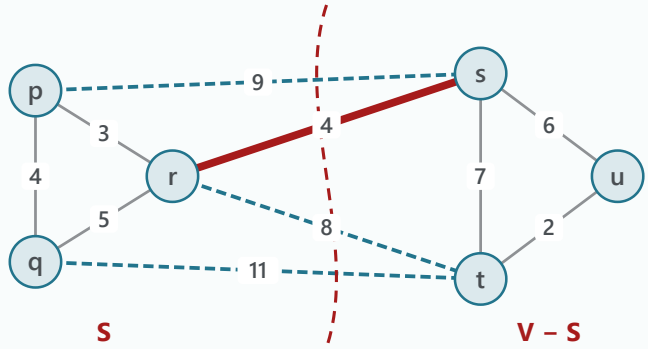
```
1  static double fractional(Item[] s, int W) {
2    Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
3    double profit = 0;
4    int cap = W;
5    for (Item x : s) {
6      if (x.weight <= cap) {           // fits entirely
7        cap -= x.weight;
8        profit += x.profit;
9      } else {                         // fractional part
10       profit += x.profit * (double) cap / x.weight;
11       break;
12     }
13   }
▶    return profit;
15 }
```

VARIABLES

x = – cap = 20 profit = 240

Minimum Spanning Trees and the Cut Property

- **Minimum spanning tree** (MST): a tree connecting all vertices of a connected, weighted, undirected graph with the **minimum total weight**; V − 1 edges.
- **Cut property**: if the vertices are split into S and V − S, the **lightest edge** crossing this cut belongs to an MST.
- Kruskal, Prim, and Borůvka are **greedy** algorithms that use this property with different cuts: every edge added is the lightest edge crossing some cut, so no choice is ever undone.
- If the weights are distinct, the MST is **unique**.



Edges crossing the cut: 9, 4, 8, 11. The lightest, r–s (4), is in the MST.

Kruskal, Prim, and Borůvka

	Kruskal	Prim	Borůvka
Greedy choice	lightest edge in the whole graph that forms no cycle	lightest edge leaving the tree	lightest edge leaving each component
Growth	forest → tree	a single tree, vertex by vertex	a forest, merging in rounds
Cut used	the two components separating the edge's endpoints	the tree vs. the remaining vertices	each component vs. the rest
Data structure	sorting + Union-Find	priority queue	Union-Find
Time	O(E log V)	O(E log V) (binary heap)	O(E log V)
Best for	sparse graphs, edge list	dense graphs, adjacency	parallel / distributed

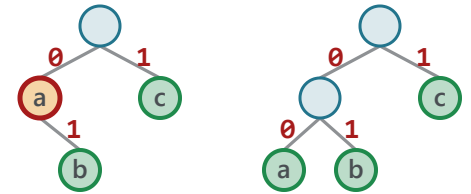
All three find the same MST on the same graph (it is unique when weights are distinct); they differ in the order they consider edges and in the data structure they need.

Huffman Coding

- A **lossless data compression** method (David Huffman, 1952).
- Uses **variable-length** codes: **frequent** characters get **short** codes, rare ones get long codes.
- Produces **prefix-free** codes: no code is a prefix of another → decodes unambiguously without separators.
- Greedy step: merge the **two lowest-frequency** trees.

Why prefix-free?

Let a = 0, b = 01, c = 1. Is "01" "ab" or "b"?
Ambiguous: 0 is a prefix of 01.
Huffman codes sit at the **leaves** of a binary tree, so no code can be a prefix of another.



✗ a is on b's path ✓ all leaves: 00, 01, 1

Used in: the entropy-coding stage of ZIP/DEFLATE, gzip, PNG, JPEG, and MP3.

Huffman Coding: Steps

1. **Step 1:** Count the **frequency** of each character.
2. **Step 2:** Build a **Huffman tree** from the frequencies: each character is a leaf; repeatedly merge the two lowest-frequency nodes into a new internal node.
3. **Step 3:** Read each character's code from the tree: left edge 0, right edge 1.
4. **Step 4:** Encode the data with these Huffman codes.

Sample frequencies (25 characters in total):

character	F	O	R	G	E	T
frequency	2	3	4	4	5	7

n − 1 merges for n characters; **O(n log n)** with a priority queue.

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
▶    for (int k = 0; k < ch.length; k++)
▶      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```

F	O	E	T	G	R
2	3	5	7	4	4

Create a leaf with its frequency for each character and add it to the priority queue (min-heap).

Step 1 / 12

VARIABLES

`pq = [F2, O3, G4, R4, E5, T7]` `left = –` `right = –`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
▶      Node left = pq.poll();
▶      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```



`poll()` twice: the smallest frequencies are **F2** (left) and **O3** (right).

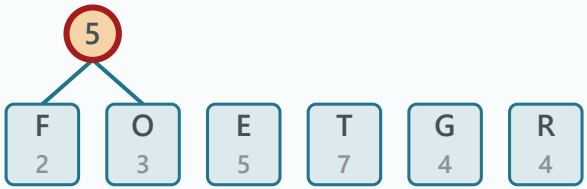
Step 2 / 12

VARIABLES

`pq = [G4, R4, E5, T7]` `left = F2` `right = O3`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
▶      Node parent = new Node('\0', left.f +
   right.f);
▶      parent.left = left;
▶      parent.right = right;
▶      pq.add(parent);
13   }
14   return pq.poll();
15 }
```



New internal node **5 = 2 + 3** with children F2 and O3, added back to the queue.

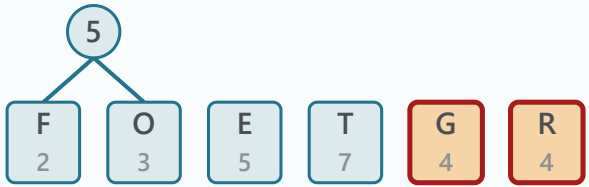
Step 3 / 12

VARIABLES

`pq = [G4, R4, (5), E5, T7]` `left = F2` `right = O3`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
▶      Node left = pq.poll();
▶      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```



`poll()` twice: the smallest frequencies are **G4** (left) and **R4** (right). G4 and R4 tie; which comes out first depends on the heap.

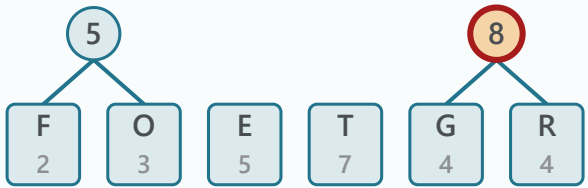
Step 4 / 12

VARIABLES

`pq = [(5), E5, T7]` `left = G4` `right = R4`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
▶      Node parent = new Node('\0', left.f +
   right.f);
▶      parent.left = left;
▶      parent.right = right;
▶      pq.add(parent);
13   }
14   return pq.poll();
15 }
```



New internal node **8 = 4 + 4** with children G4 and R4, added back to the queue.

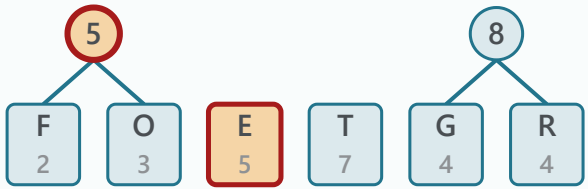
Step 5 / 12

VARIABLES

`pq = [(5), E5, T7, (8)]` `left = G4` `right = R4`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
▶      Node left = pq.poll();
▶      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```



`poll()` twice: the smallest frequencies are **(5)** (left) and **E5** (right). (5) and E5 tie; which comes out first depends on the heap, and the code lengths do not change.

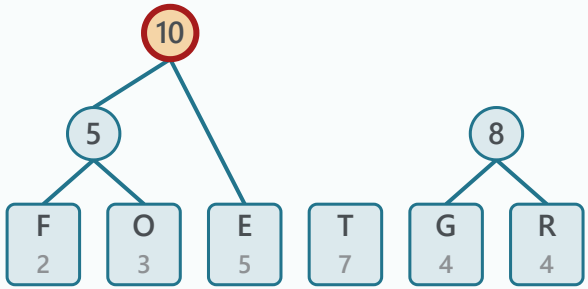
Step 6 / 12

VARIABLES

`pq = [T7, (8)]` `left = (5)` `right = E5`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
▶      Node parent = new Node('\0', left.f +
   right.f);
▶      parent.left = left;
▶      parent.right = right;
▶      pq.add(parent);
13   }
14   return pq.poll();
15 }
```



New internal node **10 = 5 + 5** with children (5) and E5, added back to the queue.

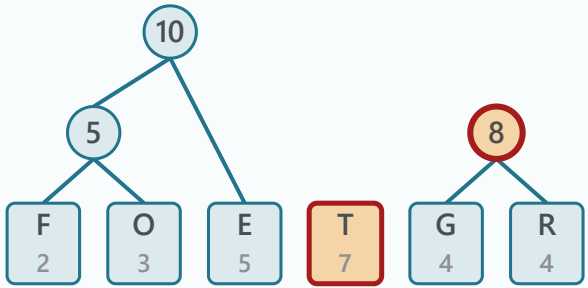
Step 7 / 12

VARIABLES

`pq = [T7, (8), (10)]` `left = (5)` `right = E5`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
▶      Node left = pq.poll();
▶      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```



`poll()` twice: the smallest frequencies are **T7** (left) and **(8)** (right).

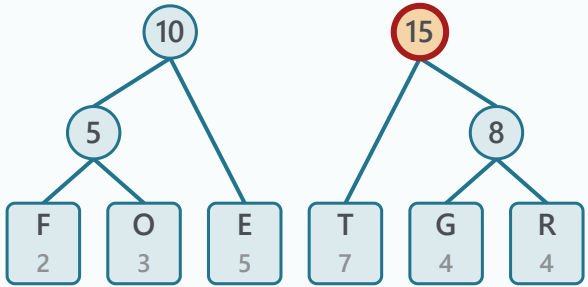
Step 8 / 12

VARIABLES

`pq = [(10)]` `left = T7` `right = (8)`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
▶      Node parent = new Node('\0', left.f +
   right.f);
▶      parent.left = left;
▶      parent.right = right;
▶      pq.add(parent);
13   }
14   return pq.poll();
15 }
```



New internal node **15 = 7 + 8** with children T7 and (8), added back to the queue.

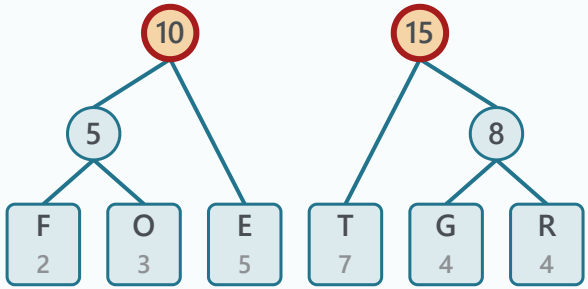
Step 9 / 12

VARIABLES

`pq = [(10), (15)]` `left = T7` `right = (8)`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
▶      Node left = pq.poll();
▶      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
14   return pq.poll();
15 }
```



`poll()` twice: the smallest frequencies are **(10)** (left) and **(15)** (right).

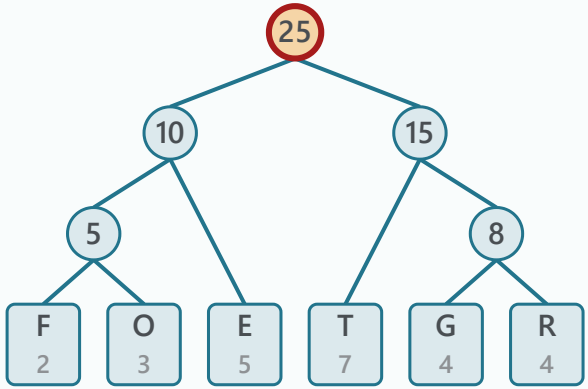
Step 10 / 12

VARIABLES

`pq = []` `left = (10)` `right = (15)`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
▶      Node parent = new Node('\0', left.f +
   right.f);
▶      parent.left = left;
▶      parent.right = right;
▶      pq.add(parent);
13   }
14   return pq.poll();
15 }
```



New internal node **25 = 10 + 15** with children (10) and (15), added back to the queue. One node is left in the queue: the root.

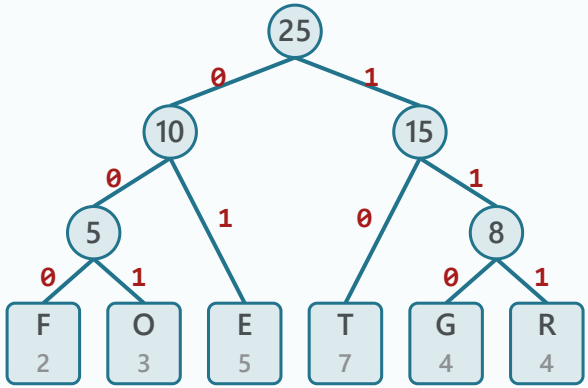
Step 11 / 12

VARIABLES

`pq = [(25)]` `left = (10)` `right = (15)`

Building the Huffman Tree: Code Trace

```
1  static Node huffman(char[] ch, int[] f) {
2    PriorityQueue<Node> pq =
3      new PriorityQueue<>((a, b) -> a.f - b.f);
4    for (int k = 0; k < ch.length; k++)
5      pq.add(new Node(ch[k], f[k]));
6    while (pq.size() > 1) {
7      Node left = pq.poll();
8      Node right = pq.poll();
9      Node parent = new Node('\0', left.f +
   right.f);
10     parent.left = left;
11     parent.right = right;
12     pq.add(parent);
13   }
▶  return pq.poll();
15 }
```



The loop ends and the root (25) is returned. Left edge 0, right edge 1: F=000, O=001, E=01, T=10, G=110, R=111.

Step 12 / 12

VARIABLES

`pq = [(25)]` `left = (10)` `right = (15)`

Codes, Encoding, and Decoding

char	frequency	code	length	bits
F	2	`000`	3	6
O	3	`001`	3	9
E	5	`01`	2	10
T	7	`10`	2	14
G	4	`110`	3	12
R	4	`111`	3	12
total				**63**

- Fixed length: 6 characters → 3 bits, 25 × 3 = **75 bits**. Huffman: **63 bits** (16% savings, 2.52 bits on average).
- **Encoding:** "FORTE" → 000 001 111 10 01 = 13 bits (15 with a fixed code).
- **Decoding:** start at the root, 0 → left, 1 → right; at a leaf, output the character and return to the root: `1100110` → 110 | 01 | 10 → "GET".
- Which child goes left or right is arbitrary (e.g., T could be the right child): the codes change, but the code **lengths** and the 63 bits do not.

Generating Codes from the Tree: Code Trace

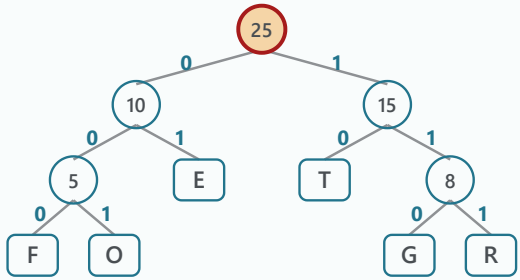
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {      // leaf
3     System.out.println(d.ch + " = " + code);
4     return;
5   }
▶   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

`genCodes((25), "")`

`main()`

OUTPUT



`genCodes(root, "")` is called on the root (25) of the Huffman tree: the root is not a leaf, so we descend to the left child, appending **"0"**.

Step 1 / 11

VARIABLES

`d = (25)` `code = ""`

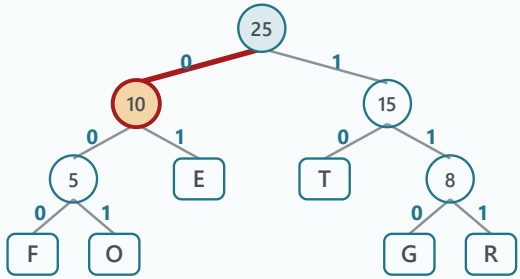
Generating Codes from the Tree: Code Trace

```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {      // leaf
3     System.out.println(d.ch + " = " + code);
4     return;
5   }
▶   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes((10), "0")`
- `genCodes((25), "")`
- `main()`

OUTPUT



Internal node (10), code = "0" (reached via a left edge) → not a leaf; descend left with "00".

Step 2 / 11

VARIABLES

`d = (10)` `code = "0"`

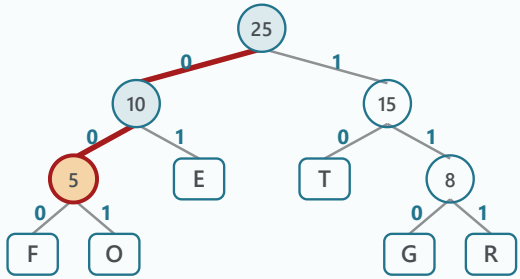
Generating Codes from the Tree: Code Trace

```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {      // leaf
3     System.out.println(d.ch + " = " + code);
4     return;
5   }
▶   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes((5), "00")`
- `genCodes((10), "0")`
- `genCodes((25), "")`
- `main()`

OUTPUT



Internal node (5), code = "00" (reached via a left edge) → not a leaf; descend left with "000".

Step 3 / 11

VARIABLES

`d = (5)` `code = "00"`

Generating Codes from the Tree: Code Trace

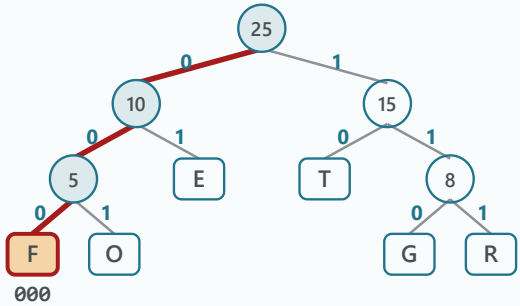
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(F, "000")`
- `genCodes((5), "00")`
- `genCodes((10), "0")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
```



Leaf **F** (frequency 2): print the bits on the path → **F = 000**.

Step 4 / 11

VARIABLES

`d = F` `code = "000"`

Generating Codes from the Tree: Code Trace

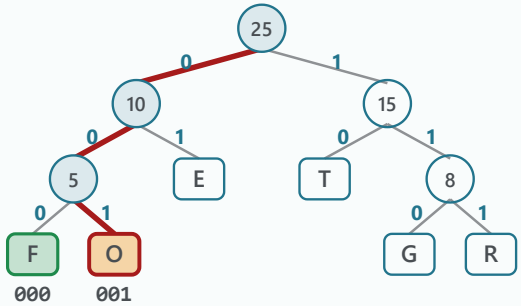
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(O, "001")`
- `genCodes((5), "00")`
- `genCodes((10), "0")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
```



Leaf **O** (frequency 3): print the bits on the path → **O = 001**.

Step 5 / 11

VARIABLES

`d = O` `code = "001"`

Generating Codes from the Tree: Code Trace

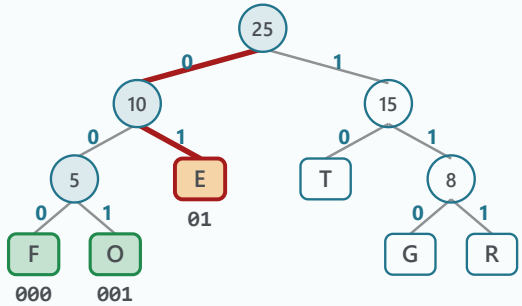
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(E, "01")`
- `genCodes((10), "0")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
```



Leaf **E** (frequency 5): print the bits on the path → **E = 01**.

Step 6 / 11

VARIABLES

`d = E` `code = "01"`

Generating Codes from the Tree: Code Trace

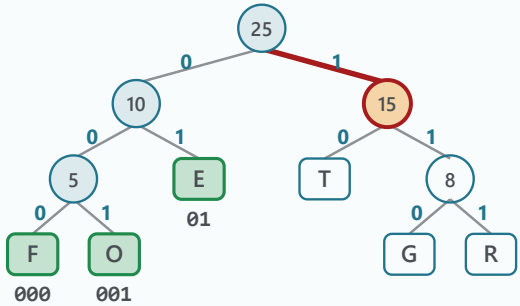
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {      // leaf
3     System.out.println(d.ch + " = " + code);
4     return;
5   }
▶   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes((15), "1")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
```



Internal node (15), code = "1" (reached via a right edge) → not a leaf; descend left with "10".

Step 7 / 11

VARIABLES

`d = (15)` `code = "1"`

Generating Codes from the Tree: Code Trace

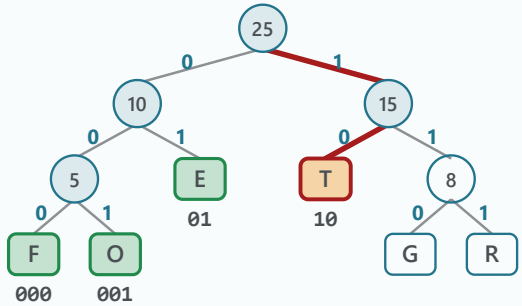
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(T, "10")`
- `genCodes((15), "1")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
T = 10
```



Leaf **T** (frequency 7): print the bits on the path → **T = 10**.

Step 8 / 11

VARIABLES

`d = T` `code = "10"`

Generating Codes from the Tree: Code Trace

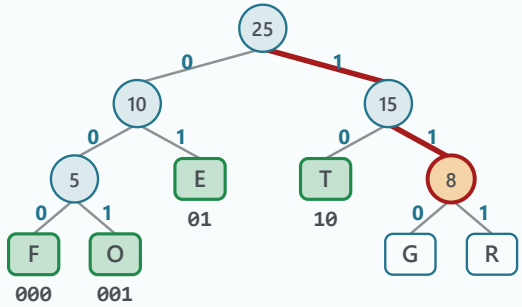
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {      // leaf
3     System.out.println(d.ch + " = " + code);
4     return;
5   }
▶   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes((8), "11")`
- `genCodes((15), "1")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
T = 10
```



Internal node (8), code = "11" (reached via a right edge) → not a leaf; descend left with "110".

Step 9 / 11

VARIABLES

`d = (8)` `code = "11"`

Generating Codes from the Tree: Code Trace

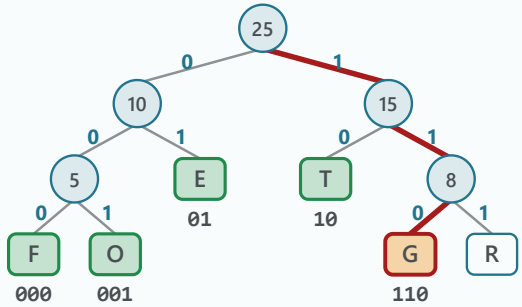
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(G, "110")`
- `genCodes((8), "11")`
- `genCodes((15), "1")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
T = 10
G = 110
```



Leaf **G** (frequency 4): print the bits on the path → **G = 110**.

Step 10 / 11

VARIABLES

`d = G` `code = "110"`

Generating Codes from the Tree: Code Trace

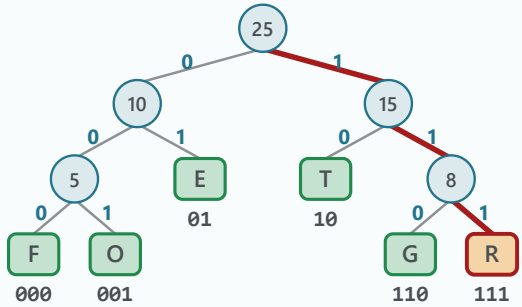
```
1 static void genCodes(Node d, String code) {
▶   if (d.left == null) {     // leaf
▶     System.out.println(d.ch + " = " + code);
4     return;
5   }
6   genCodes(d.left, code + "0"); // left: 0
7   genCodes(d.right, code + "1"); // right: 1
8 }
```

CALL STACK

- `genCodes(R, "111")`
- `genCodes((8), "11")`
- `genCodes((15), "1")`
- `genCodes((25), "")`
- `main()`

OUTPUT

```
F = 000
O = 001
E = 01
T = 10
G = 110
R = 111
```



Leaf **R** (frequency 4): print the bits on the path → **R = 111**. All leaves are printed: the frequent characters (E, T) get 2 bits, the rare ones 3. The tree is traversed once: **O(n)**.

Step 11 / 11

VARIABLES

`d = R` `code = "111"`

Why Is Huffman Optimal?

Greedy choice

There is an optimal code tree in which the two lowest-frequency characters x and y are **siblings** at the deepest level: swapping with a deeper character does not increase the total bits (exchange argument).

Optimal substructure

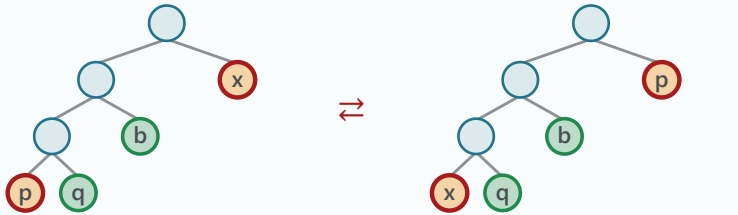
Merge x and y into a single character "z" with frequency f(x) + f(y); expanding z in an optimal tree of the smaller problem gives an optimal tree of the original problem.

Cost

Total bits = Σ freq(c) · depth(c). Among prefix-free codes, Huffman makes this **minimal**.

Complexity

Counting frequencies O(m) (m = text length); tree O(n log n); encoding O(m). The tree (or the code table) must also be sent to the decoder.



x: least frequent, shallow

x deepest: no extra bits

APPLICATIONS

Other Greedy Algorithms

Dijkstra and Prim

At each step, permanently select the vertex with the smallest distance / connection cost. Dijkstra finds shortest paths in graphs without negative edges; Prim finds an MST for any weights.

Graph coloring

Color the vertices in order, each with the smallest color its neighbors do not use. Fast, but the number of colors may not be minimal (NP-hard).

Bin packing

First fit: put each item in the first bin it fits (best fit: the bin that leaves the least space). First fit decreasing uses at most $\approx 11/9 \cdot \mathrm{OPT} + 1$ bins.

Maximum coverage

At each step, pick the set that covers the most new elements: guarantees at least $(1 - 1/e) \approx 63\%$ of the optimum.

For NP-hard problems, greedy algorithms are valuable as **approximation algorithms** with proven ratios.

Summary

Problem	Greedy criterion	Time	Optimal?
Making change	largest coin that fits	O(k + count)	if canonical
Activity selection	earliest finish	O(n log n)	yes
Job sequencing	most profitable job, latest free slot	$O(n^2)$	yes
Fractional knapsack	largest profit/weight	O(n log n)	yes (0/1: no)
MST	lightest edge crossing a cut (Kruskal, Prim, Borůvka)	O(E log V)	yes
Huffman	two lowest-frequency trees	O(n log n)	yes

SUMMARY

Common Mistakes

Assuming greedy is correct without proof

6 with coins {1, 3, 4}: greedy uses 3 coins, the optimum 2. The greedy-choice property must be shown for each problem.

The wrong criterion

In activity selection, the earliest **start** or the **shortest** activity is wrong; the right criterion is the earliest **finish**.

Ratio in the 0/1 knapsack

The ratio criterion, optimal for the fractional problem, does not guarantee the optimum for indivisible items.

Earliest free slot

In job sequencing, placing a job in the **earliest** free slot before its deadline leaves no room for jobs with tight deadlines; choose the **latest** free slot.

Misreading the cut property

The safe edge is the lightest edge **crossing** a cut. Taking the lightest edges of the whole graph in order can create a cycle; that is why Kruskal discards edges that form one.

Decoding Huffman without the tree

Without the code table or the tree, the data cannot be decoded. For a one-character text the root is a leaf: its code still needs at least 1 bit.

SUMMARY

Sample Problems

1. How many coins does greedy use to pay 63 with {1, 5, 10, 25}? What are the greedy and optimal results for 14 with {1, 7, 10}?
2. Activities (s, f): (1,4), (3,5), (0,6), (5,7), (3,9), (5,9), (6,10), (8,11), (8,12), (2,14), (12,16). What is the maximum number of activities that can be selected?
3. Frequencies a:5, b:9, c:12, d:13, e:16, f:45. Find the Huffman code lengths and the total bits.
4. The vertices are split into {A, B} and {C, D}; the edges crossing the cut are A–C 5, B–C 3, B–D 4. Which edge is in every MST?
5. Can Kruskal, Prim, and Borůvka find different spanning trees on the same graph?

▶ **Show answers**

Exercises

1. Jobs (id, deadline, profit): (a,2,100), (b,1,19), (c,2,27), (d,1,25), (e,3,15). Apply greedy job sequencing.
2. W = 10, items (profit, weight): (10,5), (40,4), (30,6), (50,3). What is the fractional knapsack value?
3. How many bits does the Huffman encoding of "ABRACADABRA" take? Compare with a fixed-length code.
4. Explain what Kruskal produces when all edges have the same weight, and which step accounts for its O(E log V) cost.
5. Find a coin system with the smallest amount for which greedy change is not optimal, and justify it.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapter 15: Greedy Algorithms; Chapter 21: MST).
- Kleinberg and Tardos, *Algorithm Design*, 2006 (Chapter 4: Greedy Algorithms).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (4.3 MST, 5.5 Data Compression).
- D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE* 40(9), 1952.
- J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. AMS* 7(1), 1956.

The End

Chapter 2: Greedy Algorithms

Previous chapter: Chapter 1: Divide and Conquer · Next chapter: Chapter 3: Dynamic Programming