Chapter 5: Branch and Bound

Advanced Algorithms

State-space tree | Bounding and pruning | FIFO, LIFO, least cost | 0/1 knapsack | Traveling salesman | Integer programming | Maximum clique | Comparing paradigms

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_05_Branch_and_Bound.html

Learning Outcomes

Paradigm

Explain the state-space tree, branching, bounding and pruning steps of branch and bound; state how it differs from backtracking.

Bound design

Build an upper bound for maximization and a lower bound for minimization; justify why the bound must be valid (optimistic).

Search strategies

Trace FIFO, LIFO and least-cost (best-first) strategies on the same example; compare them by memory and node count; tell the design paradigms apart.

Applications

Solve 0/1 knapsack, traveling salesman, integer linear programming and maximum clique step by step with branch and bound.

Weighing the Best Options

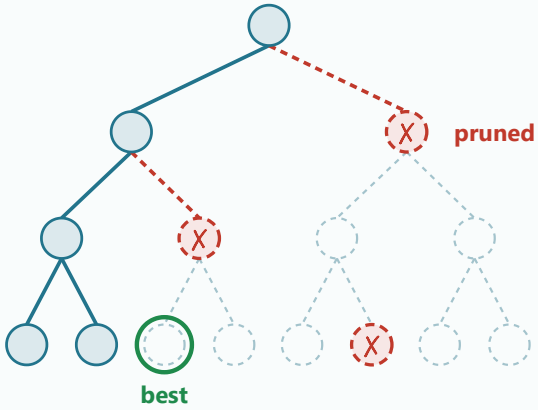
- Solving a hard problem is sometimes like picking the right way at a **fork in the road**.
- **Branch and bound** algorithms explore the possible solution paths systematically to find the **best option**.
- Goal: an **exact** (optimal) solution to **combinatorial optimization** problems. Trying every candidate takes exponential or factorial time.
- Idea: **estimate in advance** how good a branch can possibly be; if that is not good enough, never enter the branch.

n	2^n subsets	$(n-1)!$ tours
5	32	24
10	1,024	362,880
20	1,048,576	$\approx 1.2 \cdot 10^{17}$
40	$\approx 1.1 \cdot 10^{12}$	$\approx 2.0 \cdot 10^{46}$

Knapsack tries 2^n subsets, the traveling salesman $(n-1)!$ tours: brute force quickly becomes impossible.

Branch and Bound Algorithms

- Think of it as a **tree that branches and gets pruned**.
- First the problem is split into smaller **subproblems** (**branching**).
- For each subproblem the best reachable value is **estimated** (the **bound**).
- Options whose bound is no better than the best solution found so far are **discarded** (pruning).
- The process repeats for all remaining subproblems; as useless branches are pruned, the search gets shorter.
- Applications: the traveling salesman, knapsack, scheduling and more.



The subtree of a pruned node (X) is never generated.

Basic Concepts

State-space tree

Root = no decisions made yet; each level is one decision (e.g. "take item i?"); leaves are complete solutions.

Branching

Generating a node's subproblems (children). The node being expanded is the **E-node** (expanding node).

Bound

An optimistic value that **no** solution below the node can beat: an upper bound for maximization, a lower bound for minimization.

Live node

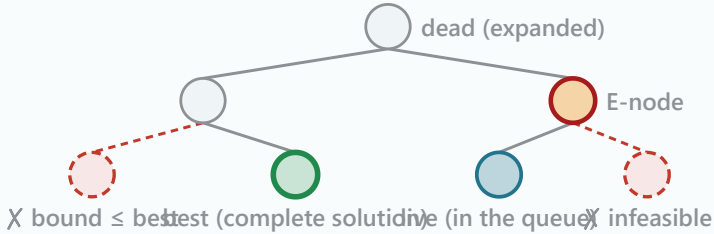
A promising node that has been generated but not yet expanded. Live nodes wait in a **queue**.

Dead node

A node that has been expanded, or pruned as infeasible or with an insufficient bound; it is never considered again.

best (incumbent)

The value of the best complete solution found so far. Pruning decisions are made against it.

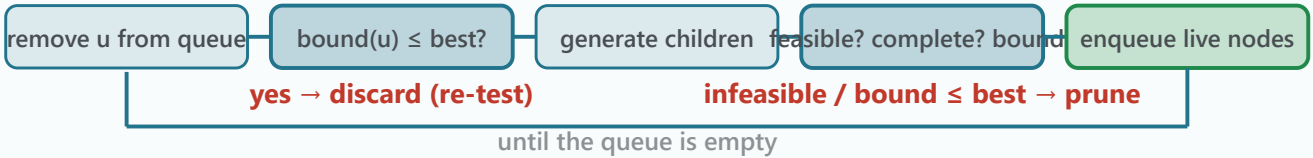


PARADIGM

General Scheme (Maximization)

```
int best = -∞;
queue.add(root);
while (!queue.isEmpty()) {
  Node u = queue.remove();     // strategy
  if (bound(u) <= best) continue;
  for (Node c : u.children()) {
    if (!isFeasible(c)) continue;
    if (isComplete(c) && value(c) > best)
      best = value(c);         // new incumbent
    if (bound(c) > best)
      queue.add(c);            // live node
  }
}
return best;
```

1. **Select:** remove a live node from the queue. The **strategy** decides which one (FIFO, LIFO, best bound).
2. **Re-test:** best may have grown while the node was waiting; if its bound is no longer enough, discard it.
3. **Branch:** generate the children; drop infeasible ones.
4. **Update:** if a complete solution is better, raise best.
5. **Bound:** enqueue only children whose bound can beat best.



For minimization the directions flip: use a lower bound and prune when `lb(c) ≥ best`.

PARADIGM

How to Choose a Bound

It must be valid (optimistic)

For maximization bound(u) ≥ every solution in the subtree; for minimization lb(u) ≤ every solution. Otherwise the optimum may be pruned and the result is **wrong**.

It should be tight

The closer it is to the true value, the more branches are pruned. An infinitely large "bound" is valid but prunes nothing.

It should be cheap

The bound is computed at every node. A very expensive bound can cost more time than the nodes it prunes.

Build it by relaxation

Drop a hard constraint and solve the easy problem: fractional knapsack, LP relaxation, cheapest outgoing edges.

Success depends heavily on a good bound. In minimization the **lower bound** is the minimum achievable in the best case; a weak lower bound makes the search longer.

PARADIGM

When Is a Node Pruned?

1. Infeasibility

The node violates a constraint: weight > capacity, a second visit to the same city, an LP with no solution. No valid solution lies below it.

2. Bound

bound(u) ≤ best (maximization) or lb(u) ≥ best (minimization). No solution below it can beat the best.

3. Completeness

The node is a complete solution (a leaf), or the relaxation's solution is already feasible (e.g. the LP solution is integral). Its value is compared with best; no branching is needed.

Backtracking uses only rule 1 (and completeness). What sets branch and bound apart is **rule 2**: a node is cut even when feasible if it is "not good enough".

PARADIGM

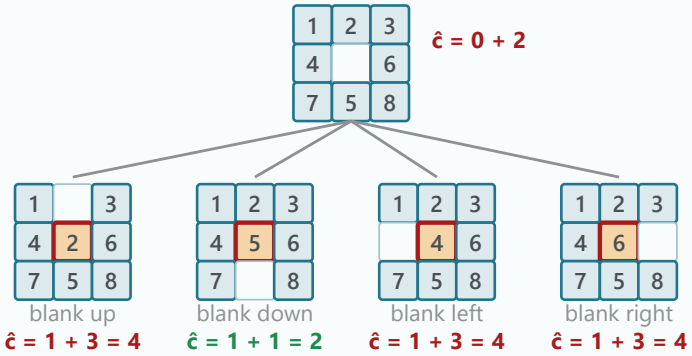
Search Strategies: Which Live Node Next?

Strategy	Container	How it explores	Pros / cons
FIFO branch and bound	queue (`ArrayDeque`)	level by level, breadth-first (BFS)	finds shallow solutions early; the number of live nodes can grow exponentially
LIFO branch and bound	stack (or recursion)	depth-first (DFS)	memory O(depth); an early complete solution raises best quickly
Least cost (LC, best-first)	priority queue (`PriorityQueue`)	node with the most promising bound first	usually expands the fewest nodes; the queue consumes memory, each operation is O(log k)

- All three strategies find **the same optimum**; what changes is the number of generated nodes and the memory.
- In LC search the cost function $\hat{c}(u)$ is the bound value (or the estimate $f(u) + \hat{g}(u)$); A* search is the path-finding version of this idea.

Least-Cost (LC) Search: ĉ(x) = f(x) + ĝ(x)

- **f(x)**: distance from the root (cost paid so far); **ĝ(x)**: estimate of the remaining cost from x to a solution.
- Expand the live node with the smallest ĉ. ĝ = 0 → only the cost paid: uniform-cost search (FIFO/BFS with unit steps when f is the depth); f = 0 → greedy best-first search.
- In the 8-puzzle ĝ = **number of misplaced tiles**: each misplaced tile needs at least one move, so ĝ never overestimates.
- Initially 5 and 8 are misplaced (ĝ = 2). The child with the smallest ĉ is "blank down"; its child "blank right" reaches the goal: **2 moves**, only 2 expansions.



Goal: 1 2 3 / 4 5 6 / 7 8 blank. Orange: the tile that moved.

PARADIGM

Backtracking vs. Branch and Bound

Feature	Backtracking	Branch and bound
Problem type	decision / search: one or all valid solutions (n-queens, Sudoku)	optimization: the best solution (largest profit, shortest tour)
Pruning criterion	feasibility: is a constraint violated?	feasibility + **bound**: can it beat best?
Traversal	always DFS (recursion)	DFS, BFS or best-first
Extra information	only the partial solution	partial solution + bound function + best
Memory	O(depth)	O(depth) with DFS; queue size with BFS/LC
Worst case	exponential	still exponential; far fewer nodes in practice

A TSP or maximum-clique search with no bound test is really backtracking; add a single bound line and it becomes branch and bound.

Pros and Cons

Pros

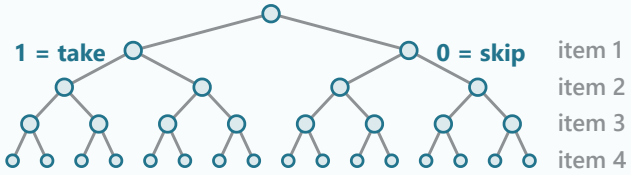
+ Finds the **exact optimum** of hard optimization problems.
+ Narrows the search space and discards useless options.
+ Ideal for problems that ask for the best route or selection.
+ Applies to a wide range of problems whose solutions can be represented as a tree.
+ If stopped early, it still holds a good solution and its bound (a quality guarantee).

Cons

– Solution time can grow when there are very many options: **exponential** time in the worst case.
– With FIFO and best-first strategies, memory can also grow exponentially.
– Success depends on **a good bound**; a weak bound prunes almost nothing.
– Needs a problem-specific bound and branching rule; harder to implement than backtracking.

Example: The 0/1 Knapsack Problem

- Notation: profit p_i, weight w_i, capacity W, decision $x_i \in \{0, 1\}$; **maximize** $\Sigma\, p_i x_i$ subject to $\Sigma\, w_i x_i \le W$.
- Items are sorted by **decreasing p_i/w_i ratio**; the bound computation relies on this order.
- The state-space tree is binary: at level i the left branch is "item i taken" (1), the right one "not taken" (0). The full tree has $2^5 - 1 = 31$ nodes.



Item	p_i	w_i	p_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

Capacity W = 16.

Optimum: items 1 and 3, profit **90** (weight 12). Branch and bound gets there without generating all 31 nodes.

Upper Bound: Fractional Knapsack Relaxation

- Relaxation: drop the "items cannot be split" constraint. The greedy method solves the **fractional** knapsack optimally.
- At node u (profit k, weight a, level i): add the remaining items in ratio order **whole** while they fit, then a **fraction** of the first item that does not fit:

$$\text{bound}(u) = k + \Sigma_{j=i+1}^{m-1} p_j + (W - a - \Sigma w_j) \cdot p_m / w_m \quad \text{(m: first item that does not fit)}$$

Why valid?

Every 0/1 solution is also a fractional solution. Fractional optimum ≥ 0/1 optimum, so the bound never understates the true best.

For the root

40 + 30 = 70 (weight 7); item 3 does not fit, 9 units left → 9 · 50/10 = 45. **bound(root) = 115**. If a > W the node is infeasible (bound 0).

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



no items yet

Root node: level −1, profit 0, weight 0.
Since 0 ≤ 16, the node is **feasible**.

Step 1 / 9

VARIABLES

`u = A (root)` `j = –` `totW = –` `result = –`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;    // infeasible
▶    double result = u.profit;
▶    int totW = u.weight;
▶    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



j = 0: next is item 1

The bound starts from the current profit; `j` points to the next item (index 0 = item 1).

Step 2 / 9

VARIABLES

`u = A (root)` `j = 0` `totW = 0` `result = 0`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
▶    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



0 + 2 ≤ 16 ?

0 + w[0] = 2 ≤ 16 → item 1 fits **entirely**.

Step 3 / 9

VARIABLES

`u = A (root)` `j = 0` `totW = 0` `result = 0`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
▶      totW += w[j];
▶      result += p[j];
▶      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



item 1 added

totW = 2, result = 40, j = 1.

Step 4 / 9

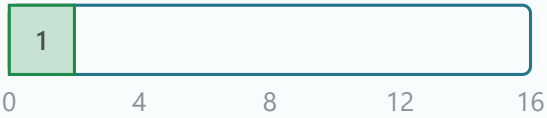
VARIABLES

u = A (root) j = 1 totW = 2 result = 40

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
▶    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```

capacity W = 16



2 + 5 ≤ 16 ?

2 + w[1] = 7 ≤ 16 → item 2 fits entirely too.

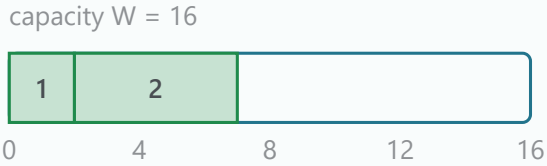
Step 5 / 9

VARIABLES

`u = A (root)` `j = 1` `totW = 2` `result = 40`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
▶      totW += w[j];
▶      result += p[j];
▶      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



item 2 added

totW = 7, result = 70, j = 2.

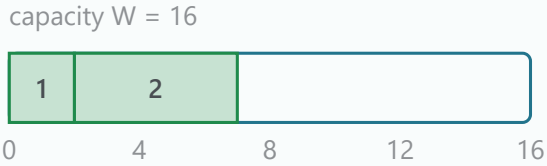
Step 6 / 9

VARIABLES

u = A (root) j = 2 totW = 7 result = 70

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
▶    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



7 + 10 = 17 > 16

7 + w[2] = 17 > 16 → item 3 does not fit entirely; the loop ends.

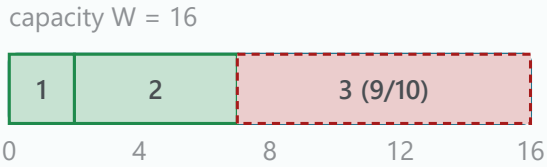
Step 7 / 9

VARIABLES

`u = A (root)` `j = 2` `totW = 7` `result = 70`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;   // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
▶    if (j < n)                  // fractional part
▶      result += (W - totW) * (double) p[j] / w[j];
13   return result;
14 }
```



9 units left: 9/10 of item 3

Remaining capacity 9: **9/10** of item 3 → 9 · 50/10 = 45 is added, result = 115.

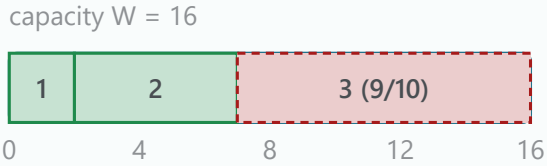
Step 8 / 9

VARIABLES

`u = A (root)` `j = 2` `totW = 7` `result = 115`

bound(root): Code Trace

```
1  double bound(Node u) {
2    if (u.weight > W) return 0;    // infeasible
3    double result = u.profit;
4    int totW = u.weight;
5    int j = u.level + 1;
6    while (j < n && totW + w[j] <= W) {
7      totW += w[j];
8      result += p[j];
9      j++;
10   }
11   if (j < n)                  // fractional part
12     result += (W - totW) * (double) p[j] / w[j];
▶    return result;
14 }
```



bound(A) = 115

bound(A) = 115: no solution below this node can earn more than 115.

Step 9 / 9

VARIABLES

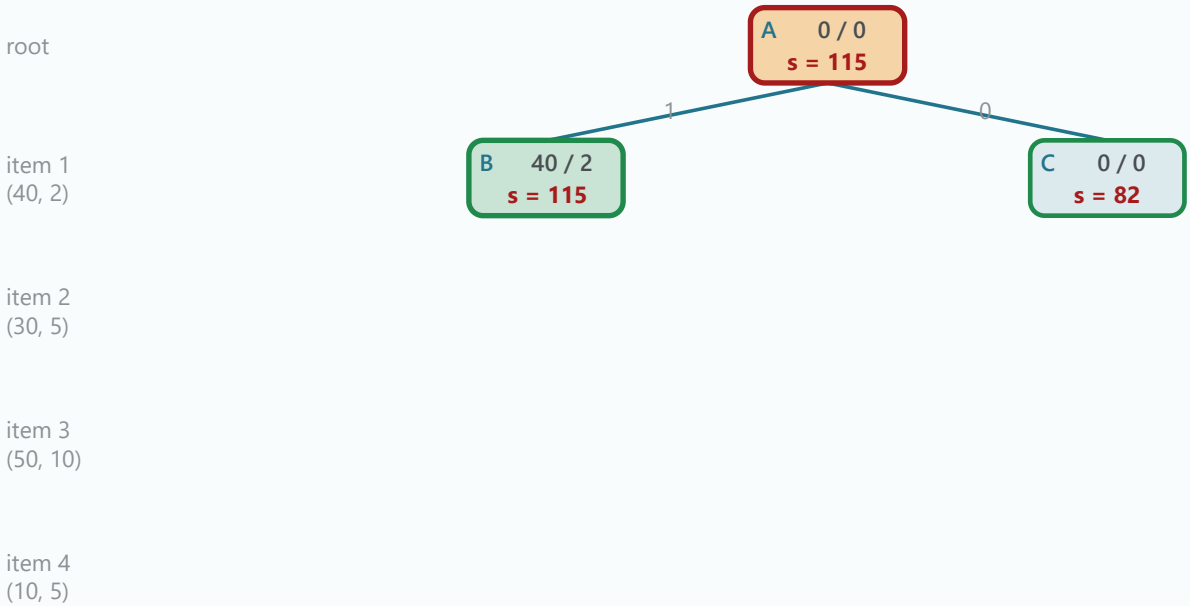
`u = A (root)` `j = 2` `totW = 7` `result = 115`

Best-First (LC) Branch and Bound: State-Space Tree



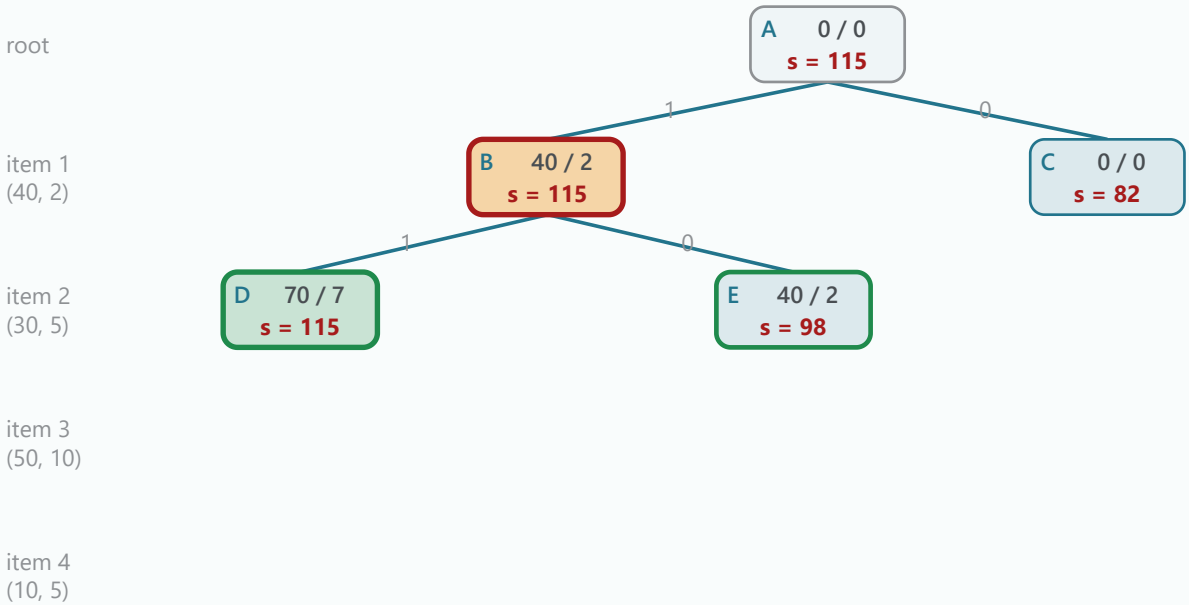
Root **A**: no item chosen (profit 0, weight 0). Fractional bound **b = 115**; best = 0. Queue: A (115).

Best-First (LC) Branch and Bound: State-Space Tree



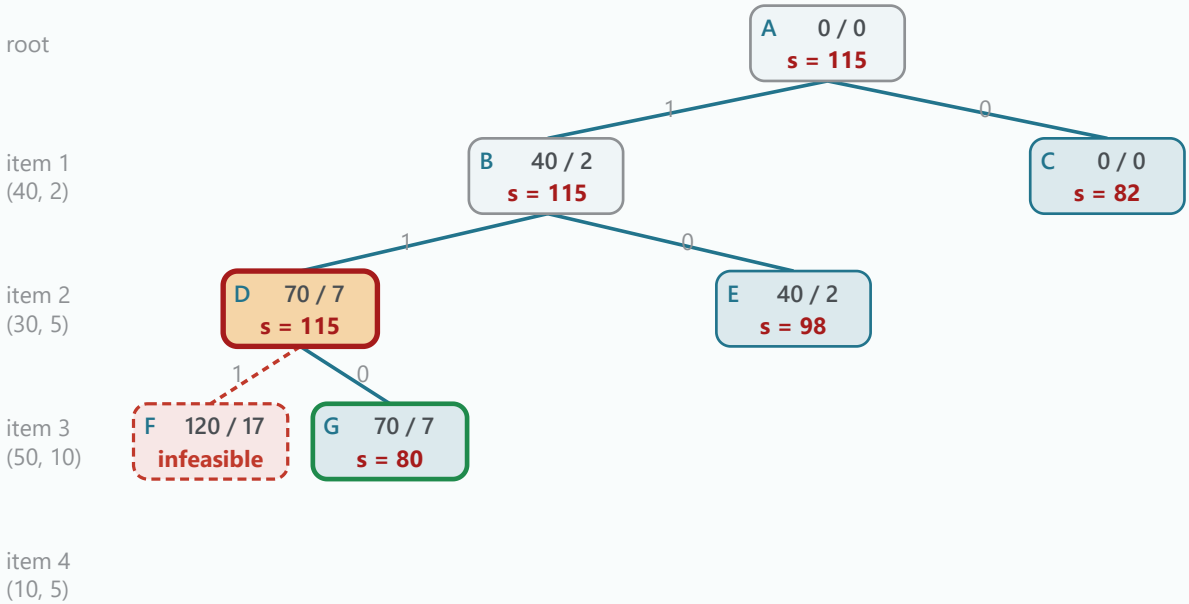
A is removed (largest bound) and branched. **B** (item 1 taken): **best = 40**, b = 115 → enqueued. **C** (no item 1): b = 82 → enqueued. Queue: B (115), C (82).

Best-First (LC) Branch and Bound: State-Space Tree



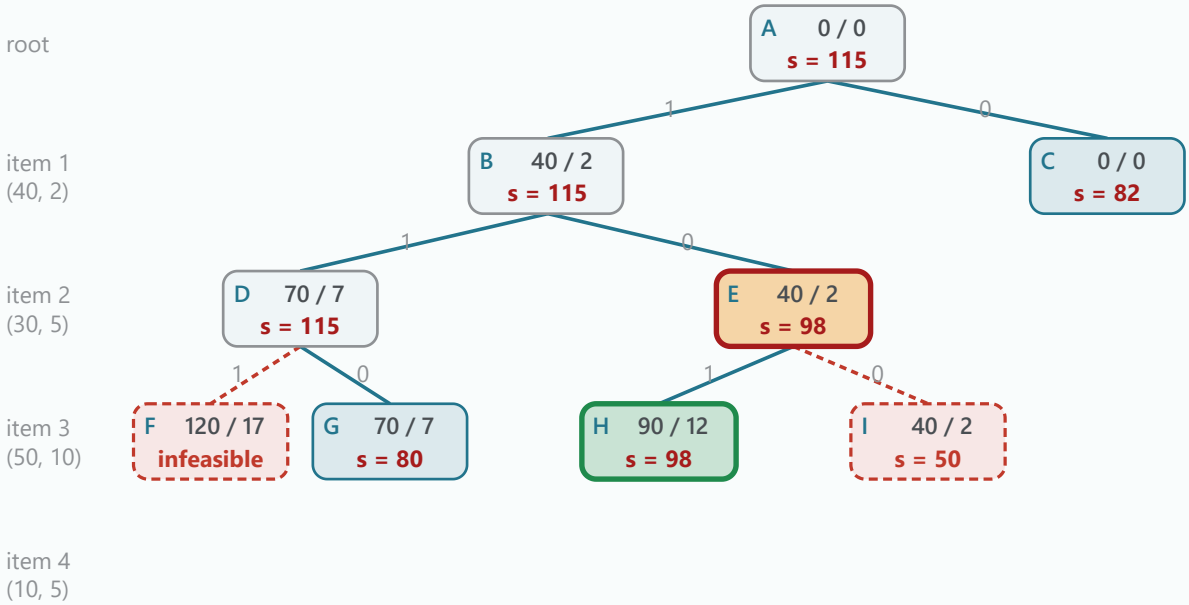
B is removed (largest bound) and branched. **D** (item 2 taken): **best = 70**, b = 115 → enqueued. **E** (no item 2): b = 98 → enqueued. Queue: D (115), E (98), C (82).

Best-First (LC) Branch and Bound: State-Space Tree



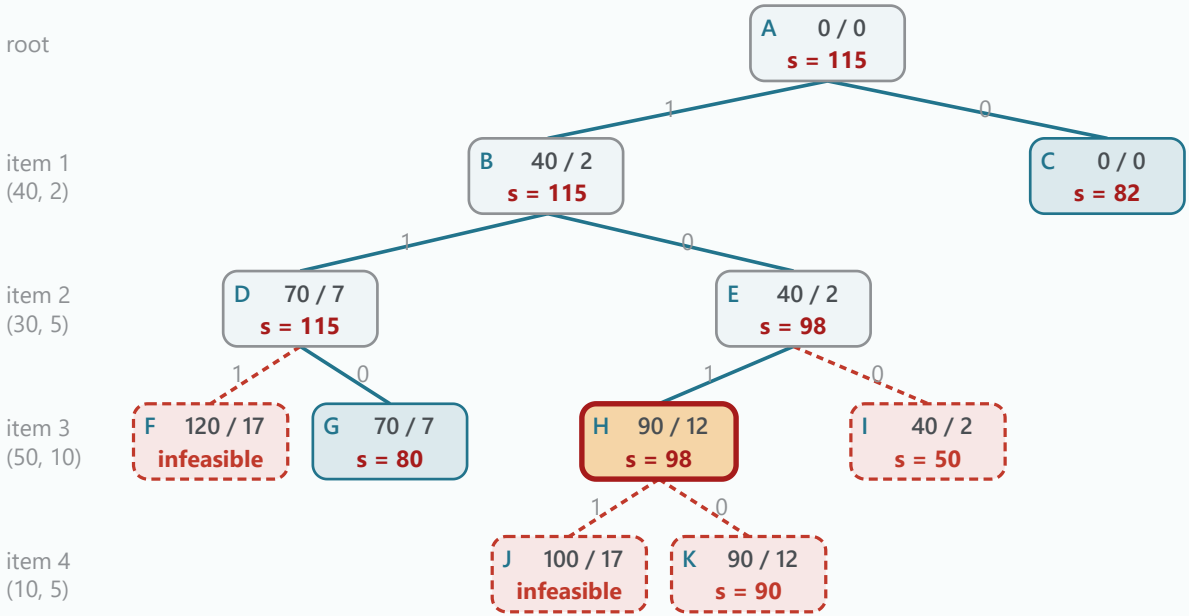
D is removed (largest bound) and branched. **F** (item 3 taken): weight 17 > 16 → **infeasible**. **G** (no item 3): b = 80 → enqueued. Queue: E (98), C (82), G (80).

Best-First (LC) Branch and Bound: State-Space Tree



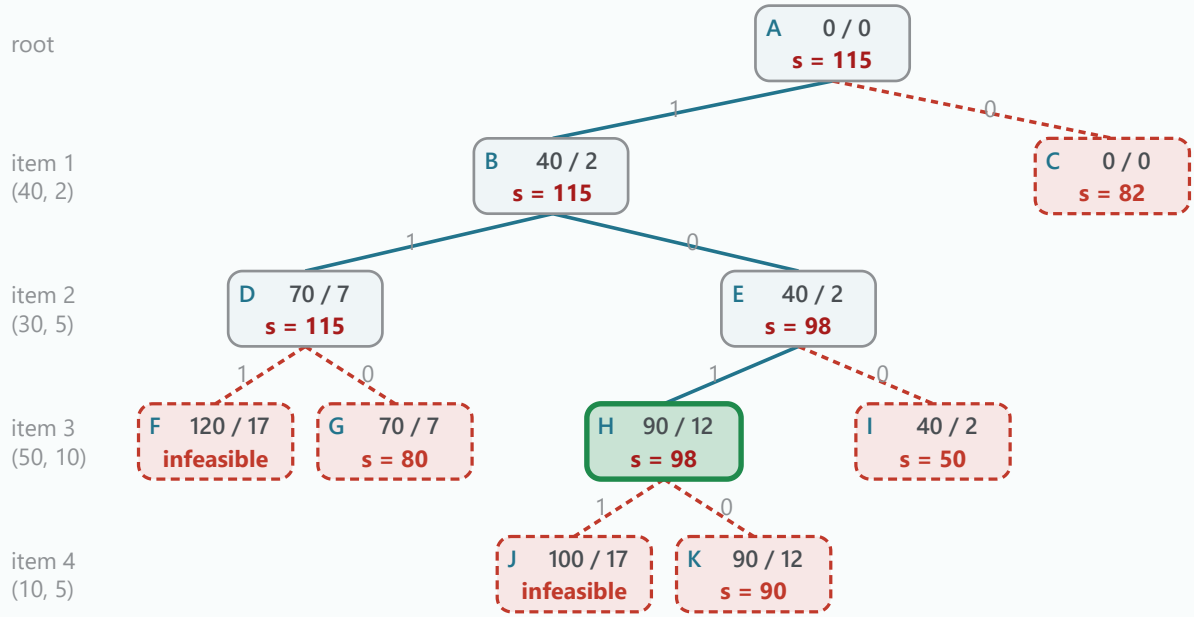
E is removed (largest bound) and branched. **H** (item 3 taken): **best = 90**, b = 98 → enqueued. **I** (no item 3): b = 50 ≤ 90 → pruned. Queue: H (98), C (82), G (80).

Best-First (LC) Branch and Bound: State-Space Tree



H is removed (largest bound) and branched. **J** (item 4 taken): weight 17 > 16 → **infeasible**. **K** (no item 4): b = 90 ≤ 90 → pruned. Queue: C (82), G (80).

Best-First (LC) Branch and Bound: State-Space Tree



C is removed: b = 82 ≤ best = 90 → pruned. **G** is removed: b = 80 ≤ best = 90 → pruned. Queue empty: **best profit 90** (H: items 1 and 3). Only 11 of the full tree's 31 nodes were generated.

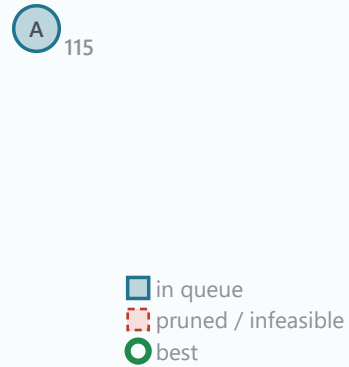
Best-First Branch and Bound: Java

```
int branchAndBound() {
  PriorityQueue<Node> queue = new PriorityQueue<>(
      (a, b) -> Double.compare(b.bound, a.bound));
  Node root = new Node(-1, 0, 0);
  root.bound = bound(root);
  queue.add(root);  int best = 0;
  while (!queue.isEmpty()) {
    Node u = queue.poll();              // largest bound
    if (u.bound <= best || u.level == n - 1) continue;
    int i = u.level + 1;
    // left child: take item i
    Node take = new Node(i, u.profit + p[i],
                         u.weight + w[i]);
    if (take.weight <= W && take.profit > best)
      best = take.profit;
    take.bound = bound(take);
    if (take.bound > best) queue.add(take);
    // right child: skip item i
    Node skip = new Node(i, u.profit, u.weight);
    skip.bound = bound(skip);
    if (skip.bound > best) queue.add(skip);
  }
  return best;
}
```

- `Node` : level, profit, weight, bound (a `boolean[]` or bitmask can be added for the chosen items).
- The comparator is reversed: the **largest bound** comes out first.
- The test after `poll` discards nodes that went **stale** while waiting in the queue (C and G in the example).
- For FIFO, `ArrayDeque` + `add()` / `poll()` is enough; for LIFO, `push()` / `pop()` ; the rest of the code is the same.
- Worst case $O(2^n)$ nodes, O(n) bound computation per node.

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
▶    ArrayDeque<Node> queue = new ArrayDeque<>();
▶    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
5      Node u = queue.poll();            // first in
6      if (u.bound <= best || u.level == n - 1) continue;
7      for (int take = 1; take >= 0; take--) { // take first
8        Node c = child(u, take);        // computes
   bound(c)
9        if (c.weight <= W && c.profit > best)
10         best = c.profit;                // new incumbent
11       if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



The same knapsack instance (W = 16), this time with a **FIFO** queue. Root A: profit 0, bound 115; only A is in the queue.

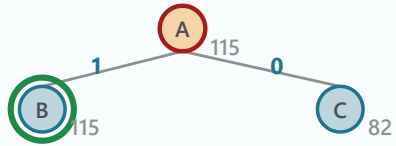
Step 1 / 9

VARIABLES

`u = –` `best = 0` `queue = [A]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
▶      for (int take = 1; take >= 0; take--) { // take first
▶        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;                // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



in queue
pruned / infeasible
best

A leaves the front of the queue and branches: B (take: b = 115 → enqueued, **best = 40**); C (skip: b = 82 → enqueued).

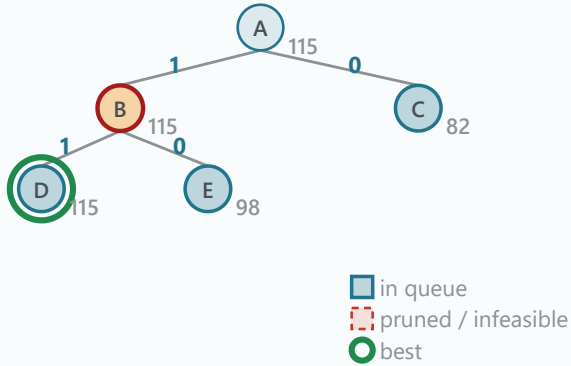
Step 2 / 9

VARIABLES

`u = A` `best = 40` `queue = [B, C]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
▶      for (int take = 1; take >= 0; take--) { // take first
▶        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;                // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



B leaves the front of the queue and branches: D (take: b = 115 → enqueued, **best = 70**); E (skip: b = 98 → enqueued).

Step 3 / 9

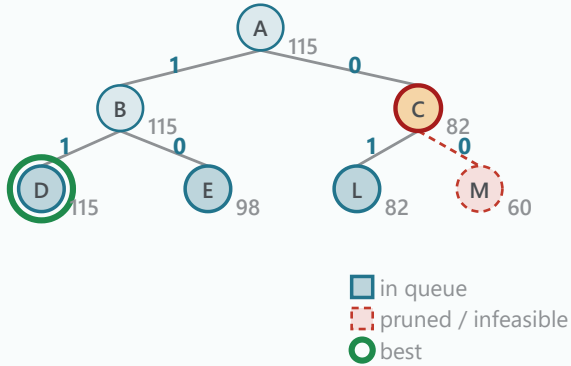
VARIABLES

`u = B` `best = 70`

`queue = [C, D, E]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
▶      for (int take = 1; take >= 0; take--) { // take first
▶        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;                // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



C leaves the front of the queue and branches: L (take: b = 82 → enqueued); M (skip: b = 60 ≤ 70, pruned).

Step 4 / 9

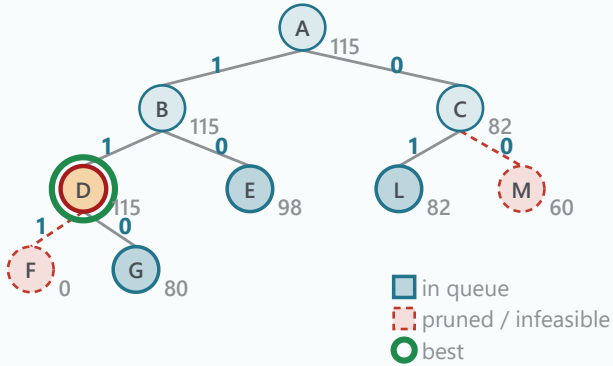
VARIABLES

`u = C` `best = 70`

`queue = [D, E, L]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
▶      for (int take = 1; take >= 0; take--) { // take first
▶        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;                // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



D leaves the front of the queue and branches: F (take: weight 17 > 16, infeasible); G (skip: b = 80 → enqueued).

Step 5 / 9

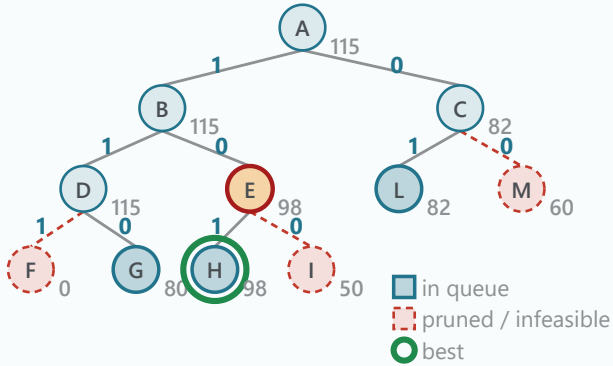
VARIABLES

`u = D` `best = 70`

`queue = [E, L, G]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
▶      for (int take = 1; take >= 0; take--) { // take first
▶        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;                // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



E leaves the front of the queue and branches: H (take: b = 98 → enqueued, **best = 90**); I (skip: b = 50 ≤ 90, pruned).

Step 6 / 9

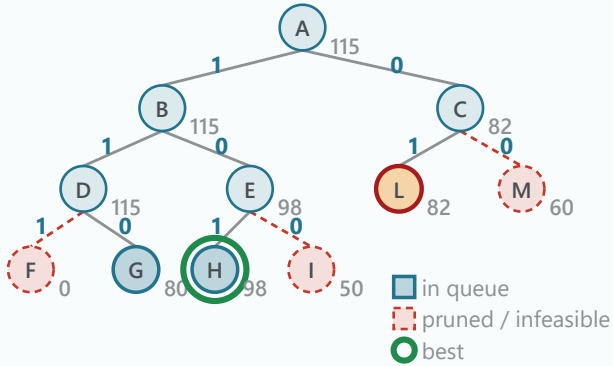
VARIABLES

`u = E` `best = 90`

`queue = [L, G, H]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
7      for (int take = 1; take >= 0; take--) { // take first
8        Node c = child(u, take);        // computes
   bound(c)
9        if (c.weight <= W && c.profit > best)
10         best = c.profit;                // new incumbent
11       if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



L is removed: its bound 82 ≤ best = 90 → **discarded by the re-test** (it looked promising when enqueued, but best grew while it waited).

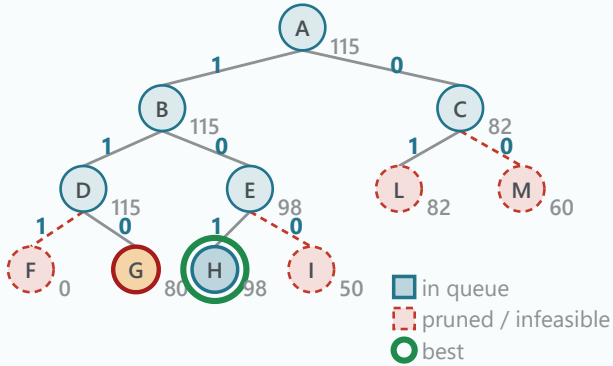
Step 7 / 9

VARIABLES

`u = L` `best = 90` `queue = [G, H]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
▶      Node u = queue.poll();            // first in
▶      if (u.bound <= best || u.level == n - 1) continue;
7      for (int take = 1; take >= 0; take--) { // take first
8        Node c = child(u, take);        // computes
   bound(c)
9        if (c.weight <= W && c.profit > best)
10         best = c.profit;                // new incumbent
11       if (c.bound > best) queue.add(c);
12     }
13   }
14   return best;
15 }
```



G is removed: its bound 80 ≤ best = 90 → **discarded by the re-test** (it looked promising when enqueued, but best grew while it waited).

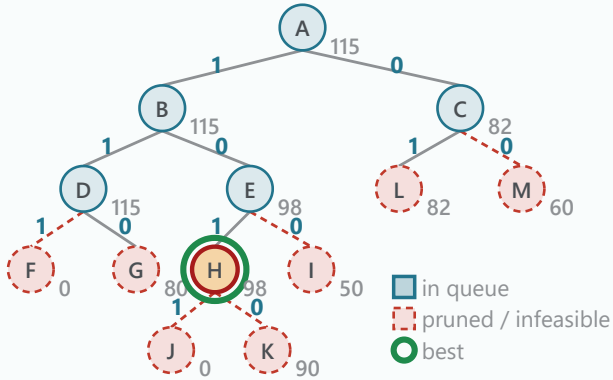
Step 8 / 9

VARIABLES

`u = G` `best = 90` `queue = [H]`

FIFO Branch and Bound: Code Trace

```
1  int fifoBranchAndBound() {
2    ArrayDeque<Node> queue = new ArrayDeque<>();
3    queue.add(root);  int best = 0;
4    while (!queue.isEmpty()) {
5      Node u = queue.poll();            // first in
6      if (u.bound <= best || u.level == n - 1) continue;
7      for (int take = 1; take >= 0; take--) { // take first
8        Node c = child(u, take);        // computes
   bound(c)
▶        if (c.weight <= W && c.profit > best)
▶          best = c.profit;               // new incumbent
▶        if (c.bound > best) queue.add(c);
12     }
13   }
▶    return best;
15 }
```



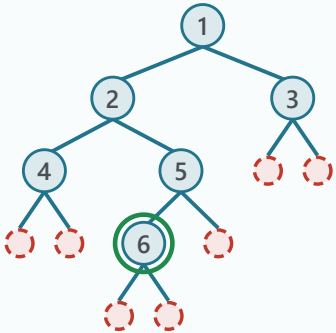
H leaves the front of the queue and branches: J (take: weight 17 > 16, infeasible); K (skip: b = 90 ≤ 90, pruned). Queue empty: **best = 90** (items 1 and 3). 13 nodes generated, 6 expansions, at most 3 in the queue.

Step 9 / 9

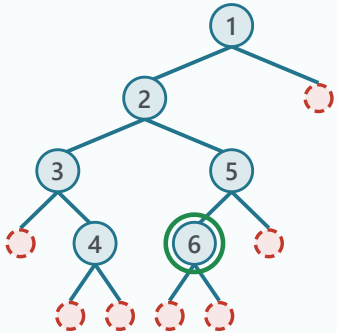
VARIABLES

`u = H` `best = 90` `queue = []`

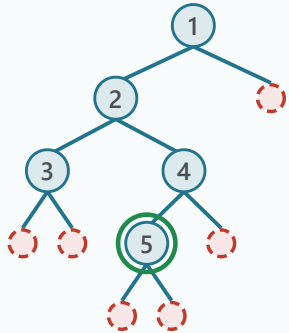
Three Strategies on the Same Example



FIFO: 13 nodes, 6 expansions



LIFO: 13 nodes, 6 expansions



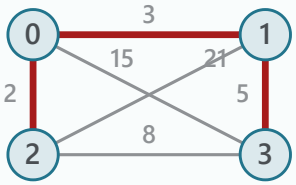
Best-first: 11 nodes, 5 expansions

Numbers give the expansion order; small circles are nodes generated but not expanded (red dashed: pruned); ring: the optimum (90).

Method	Nodes generated	Expanded	Largest queue
Brute force (full tree)	31	15	—
FIFO branch and bound	13	6	3
LIFO branch and bound ("take" first)	13	6	3
Best-first (LC)	11	5	3

Traveling Salesman Problem (TSP): Lower Bound

- Minimization: we want the shortest tour; node bounds are **lower bounds**.
- Node = a partial path starting at 0; children = the next unvisited city.
- **Lower bound:** every city **not yet left** (the current city + the unvisited ones) must be left by at least one edge:



bold: each city's cheapest edge

2 + 3 + 2 + 5 = 12

	0	1	2	3	min
0	—	3	2	15	**2**
1	3	—	21	5	**3**
2	2	21	—	8	**2**
3	15	5	8	—	**5**

Symmetric 4-city distance matrix d.

$$\text{lb(path)} = \text{cost(path)} + \text{minOut[last city]} + \Sigma_{\text{v unvisited}}\ \text{minOut[v]}$$

Root: 2 + 3 + 2 + 5 = 12. Path 0 → 1: 3 + 3 + (2 + 5) = 13. No real tour can cost less than this sum, so the bound is valid.

tsp(0, 0, 1): Bounded Depth-First Search

```
▶ void tsp(int city, int cost, int count) {
2   if (count == n) {            // tour
  complete
3     int tour = cost + d[city][0];
4     if (tour < best) best = tour;
5     return;
6   }
7   for (int s = 0; s < n; s++) {
8     if (visited[s]) continue;
9     int next = cost + d[city][s];
10    visited[s] = true;
11    if (lowerBound(s, next) < best) // bound
12      tsp(s, next, count + 1);
13    visited[s] = false;       // undo
14  }
15 }
```



Start at city 0. The root's lower bound = 2 + 3 + 2 + 5 = **12**.

Step 1 / 13

CALL STACK

`tsp(0, 0, 1)`

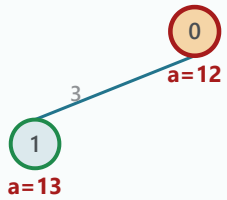
VARIABLES

`city = 0` `cost = 0` `count = 1` `s = –`

`next = –` `lb = –` `best = ∞`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



0 → 1: next = 0 + 3 = 3; lb = 13 < ∞ → **branch**.

Step 2 / 13

CALL STACK

`tsp(0, 0, 1)`

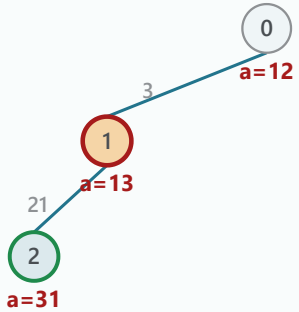
VARIABLES

`city = 0` `cost = 0` `count = 1` `s = 1`

`next = 3` `lb = 13` `best = ∞`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



1 → 2: next = 3 + 21 = 24; lb = 31 < ∞ → **branch**.

Step 3 / 13

CALL STACK

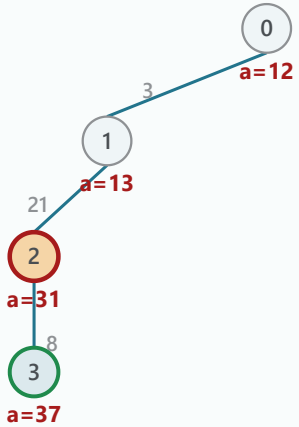
- `tsp(1, 3, 2)`
- `tsp(0, 0, 1)`

VARIABLES

`city = 1` `cost = 3` `count = 2` `s = 2` `next = 24` `lb = 31` `best = ∞`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;        // undo
14   }
15 }
```



2 → 3: next = 24 + 8 = 32; lb = 37 < ∞ → **branch**.

Step 4 / 13

CALL STACK

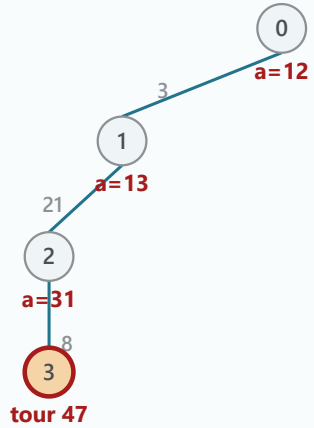
- tsp(2, 24, 3)
- tsp(1, 3, 2)
- tsp(0, 0, 1)

VARIABLES

city = 2 | cost = 24 | count = 3 | s = 3 | next = 32 | lb = 37 | best = ∞

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
▶      int tour = cost + d[city][0];
▶      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
11     if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;          // undo
14   }
15 }
```



Tour 0 → 1 → 2 → 3 → 0 = 32 + 15 = **47** < ∞ → best = 47.

Step 5 / 13

CALL STACK

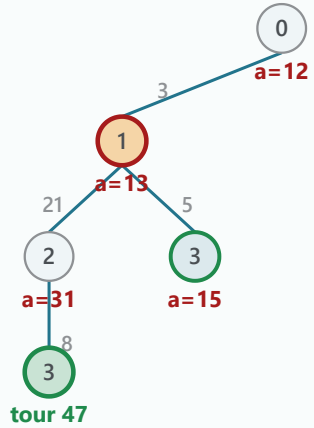
- **tsp(3, 32, 4)**
- tsp(2, 24, 3)
- tsp(1, 3, 2)
- tsp(0, 0, 1)

VARIABLES

`city = 3` `cost = 32` `count = 4` `s = –`
`next = –` `lb = –` `best = 47`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



1 → 3: next = 3 + 5 = 8; lb = 15 < 47 → **branch**.

Step 6 / 13

CALL STACK

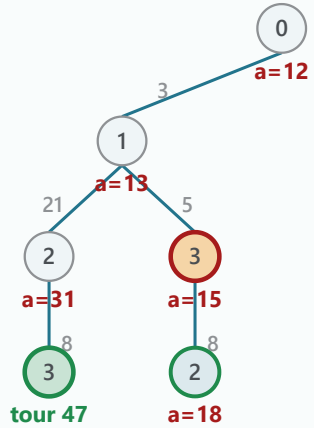
- **tsp(1, 3, 2)**
- tsp(0, 0, 1)

VARIABLES

`city = 1` `cost = 3` `count = 2` `s = 3` `next = 8` `lb = 15` `best = 47`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



3 → 2: next = 8 + 8 = 16; lb = 18 < 47 → **branch**.

Step 7 / 13

CALL STACK

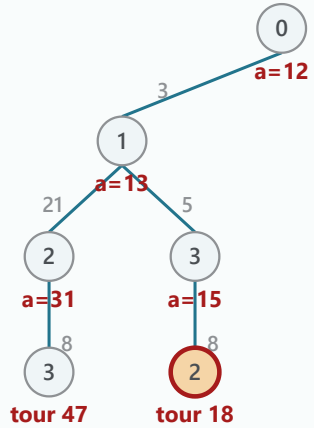
- tsp(3, 8, 3)
- tsp(1, 3, 2)
- tsp(0, 0, 1)

VARIABLES

city = 3 | cost = 8 | count = 3 | s = 2 | next = 16 | lb = 18 | best = 47

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
▶      int tour = cost + d[city][0];
▶      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
11     if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;          // undo
14   }
15 }
```



Tour 0 → 1 → 3 → 2 → 0 = 16 + 2 = **18** < 47 → best = 18.

Step 8 / 13

CALL STACK

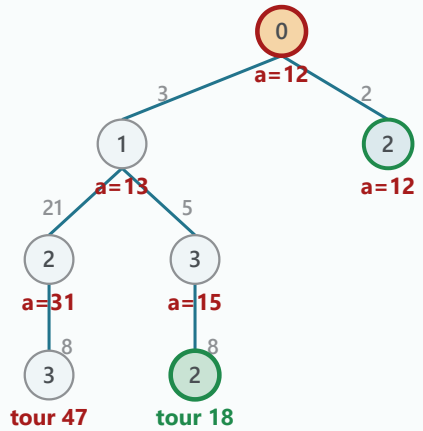
- **tsp(2, 16, 4)**
- tsp(3, 8, 3)
- tsp(1, 3, 2)
- tsp(0, 0, 1)

VARIABLES

`city = 2` `cost = 16` `count = 4` `s = –`
`next = –` `lb = –` `best = 18`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
▶        tsp(s, next, count + 1);
13     visited[s] = false;        // undo
14   }
15 }
```



0 → 2: next = 0 + 2 = 2; lb = 12 < 18 → **branch**.

Step 9 / 13

CALL STACK

`tsp(0, 0, 1)`

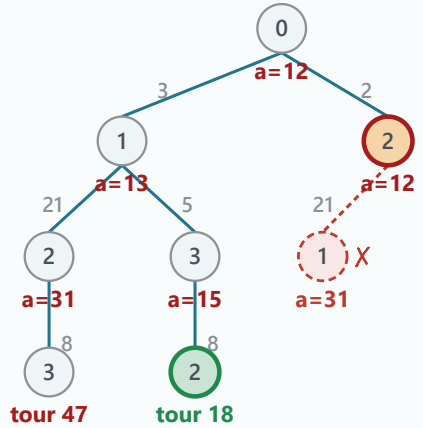
VARIABLES

`city = 0` `cost = 0` `count = 1` `s = 2`

`next = 2` `lb = 12` `best = 18`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



2 → 1: next = 23; lb = 31 ≥ best = 18 → **pruned**.

Step 10 / 13

CALL STACK

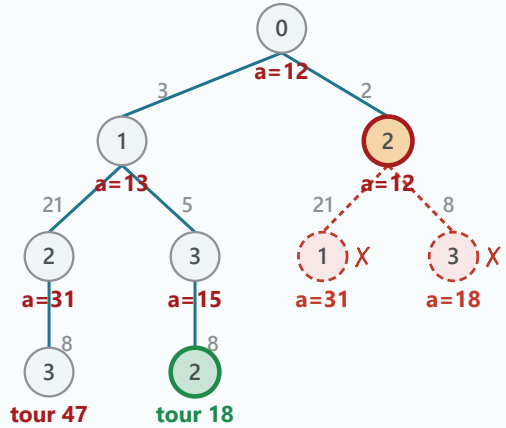
- **tsp(2, 2, 2)**
- tsp(0, 0, 1)

VARIABLES

city = **2** · cost = **2** · count = **2** · s = **1** · next = **23** · lb = **31** · best = **18**

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;        // undo
14   }
15 }
```



2 → 3: next = 10; lb = 18 ≥ best = 18 → **pruned**.

Step 11 / 13

CALL STACK

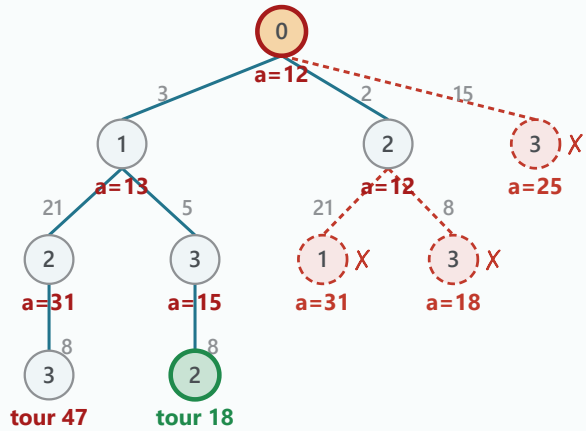
- **tsp(2, 2, 2)**
- tsp(0, 0, 1)

VARIABLES

`city = 2` `cost = 2` `count = 2` `s = 3` `next = 10` `lb = 18` `best = 18`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
▶      if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
15 }
```



0 → 3: next = 15; lb = 25 ≥ best = 18 → **pruned**.

Step 12 / 13

CALL STACK

`tsp(0, 0, 1)`

VARIABLES

`city = 0` `cost = 0` `count = 1` `s = 3`

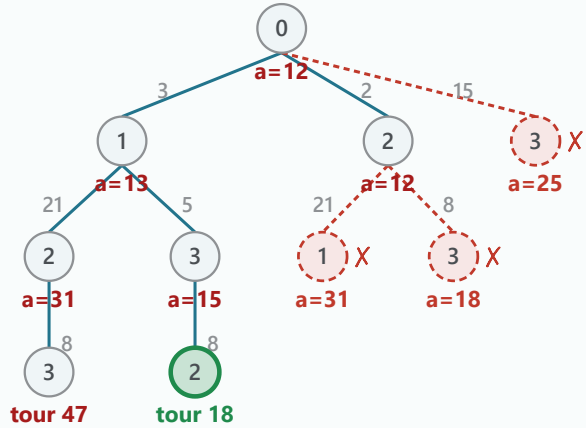
`next = 15` `lb = 25` `best = 18`

tsp(0, 0, 1): Bounded Depth-First Search

```
1  void tsp(int city, int cost, int count) {
2    if (count == n) {            // tour
   complete
3      int tour = cost + d[city][0];
4      if (tour < best) best = tour;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (visited[s]) continue;
9      int next = cost + d[city][s];
10     visited[s] = true;
11     if (lowerBound(s, next) < best) // bound
12       tsp(s, next, count + 1);
13     visited[s] = false;       // undo
14   }
▶ }
```

CALL STACK

empty



Done: the shortest tour is **0 → 1 → 3 → 2 → 0 = 18**; 10 nodes instead of 16.

Step 13 / 13

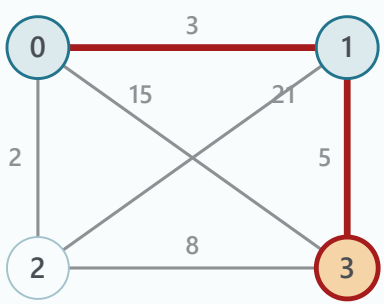
VARIABLES

`city = 0` `cost = 0` `count = 1` `s = –`

`next = –` `lb = –` `best = 18`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
▶   int lb = cost + minOut[last];
3   for (int v = 0; v < n; v++)
4     if (!visited[v])        // not visited
5       lb += minOut[v];
6   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

Step 1 → 3 of the TSP trace: path 0 → 1 → 3, cost 3 + 5 = 8; visited = {0, 1, 3}. The last city 3 must be left once more: lb = 8 + minOut[3] = 8 + 5 = **13**.

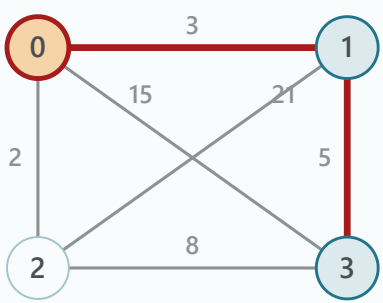
Step 1 / 6

VARIABLES

`last = 3` `cost = 8` `v = –` `lb = 13`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
2   int lb = cost + minOut[last];
▶   for (int v = 0; v < n; v++)
▶     if (!visited[v])        // not visited
5       lb += minOut[v];
6   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

v = 0: visited (on the path) → no contribution; its outgoing edge is already in the cost.

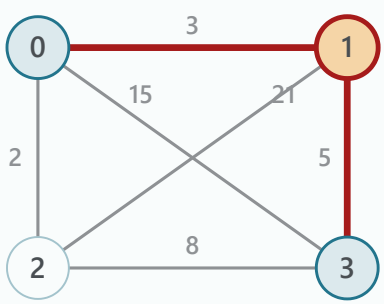
Step 2 / 6

VARIABLES

`last = 3` `cost = 8` `v = 0` `lb = 13`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
2   int lb = cost + minOut[last];
▶   for (int v = 0; v < n; v++)
▶     if (!visited[v])        // not visited
5       lb += minOut[v];
6   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

v = 1: visited (on the path) → no contribution; its outgoing edge is already in the cost.

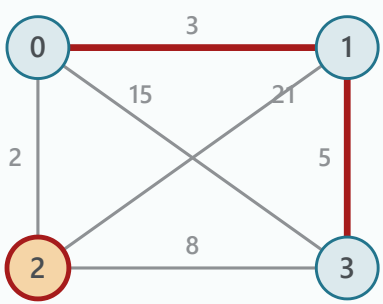
Step 3 / 6

VARIABLES

`last = 3` `cost = 8` `v = 1` `lb = 13`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
2   int lb = cost + minOut[last];
3   for (int v = 0; v < n; v++)
▶     if (!visited[v])        // not visited
▶       lb += minOut[v];
6   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

v = 2: not yet visited → must be left by at least minOut[2] = 2: lb = **15**.

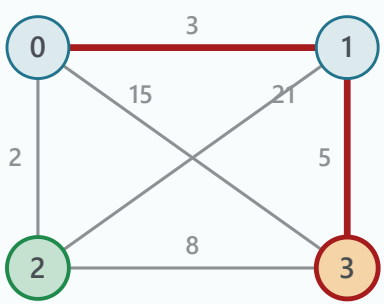
Step 4 / 6

VARIABLES

`last = 3` `cost = 8` `v = 2` `lb = 15`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
2   int lb = cost + minOut[last];
▶   for (int v = 0; v < n; v++)
▶     if (!visited[v])        // not visited
5       lb += minOut[v];
6   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

v = 3: visited (on the path) → no contribution; 3's exit was added at the start.

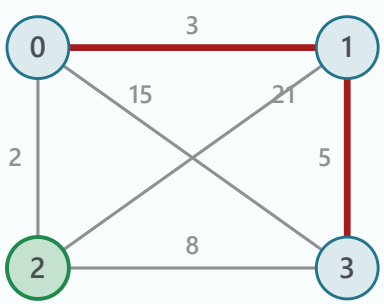
Step 5 / 6

VARIABLES

`last = 3` `cost = 8` `v = 3` `lb = 15`

lowerBound(3, 8): Code Trace

```
1 int lowerBound(int last, int cost) {
2   int lb = cost + minOut[last];
3   for (int v = 0; v < n; v++)
4     if (!visited[v])        // not visited
5       lb += minOut[v];
▶   return lb;
7 }
```



v	minOut	vis.
0	2	✓
1	3	✓
2	2	—
3	5	✓

path 0 → 1 → 3, cost 8

Returns lb = **15**. Every tour in this branch is ≥ 15; the real tour 0 → 1 → 3 → 2 → 0 = 18. Since 15 < best = 47, the TSP trace entered the branch.

Step 6 / 6

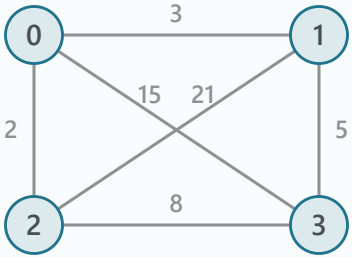
VARIABLES

`last = 3` `cost = 8` `v = –` `lb = 15`

A Tighter Lower Bound: Two Shortest Edges

- A tour **enters each city by one edge and leaves it by another**: every city touches exactly two tour edges.
- These two edges cannot be shorter than the two shortest edges touching the city. Summed over all cities, each edge is counted twice:

$$\text{tour} \geq \tfrac{1}{2} \cdot \Sigma_v \text{ (v's two shortest edges)}$$



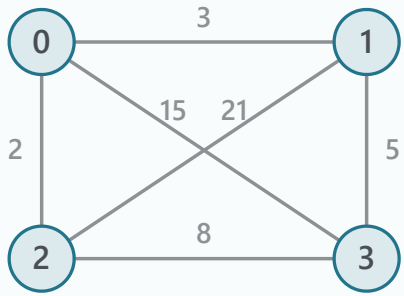
v	two smallest	sum
0	2 + 3	= 5
1	3 + 5	= 8
2	2 + 8	= 10
3	5 + 8	= 13

36 / 2 = 18

On the same matrix the “cheapest outgoing edge” bound gave 12; this bound gives 18, equal to the optimal tour 0 → 1 → 3 → 2 → 0 = 3 + 5 + 8 + 2 = 18. Since bound = optimum at the root, all remaining branches are pruned as soon as the first complete tour is found.

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
▶    int sum = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // two smallest
5      for (int u = 0; u < n; u++) {
6        if (u == v) continue;
7        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = — m2 = — sum = 0

Symmetric 4-city matrix. sum = 0; we look for the two shortest edges of each city.

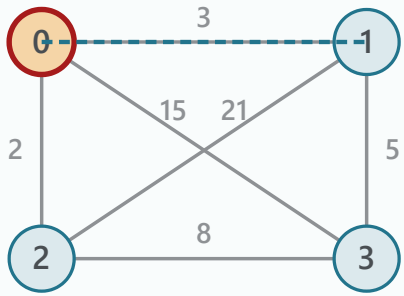
Step 1 / 8

VARIABLES

v = – | u = – | m1 = – | m2 = – | sum = 0

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // two smallest
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 3 m2 = ∞ sum = 0

v = 0, u = 1: d = 3 < m1 = ∞ → m1 = 3 (the old m1 shifts to m2).

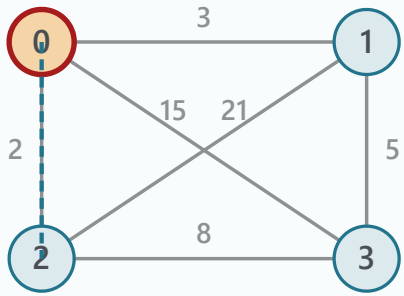
Step 2 / 8

VARIABLES

v = 0 | u = 1 | m1 = 3 | m2 = ∞ | sum = 0

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // two smallest
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 2 m2 = 3 sum = 0

u = 2: d = 2 < m1 = 3 → m2 = 3, m1 = 2. The two minima shift along.

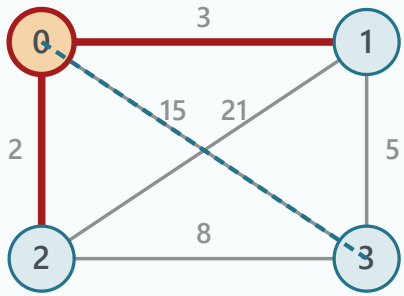
Step 3 / 8

VARIABLES

v = 0 | u = 2 | m1 = 2 | m2 = 3 | sum = 0

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // two smallest
       for (int u = 0; u < n; u++) {
         if (u == v) continue;
         if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
         else if (d[v][u] < m2) m2 = d[v][u];
9      }
       sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 2 m2 = 3 sum = 5

u = 3: 15 is larger than both, no change. sum += 2 + 3 = **5**.

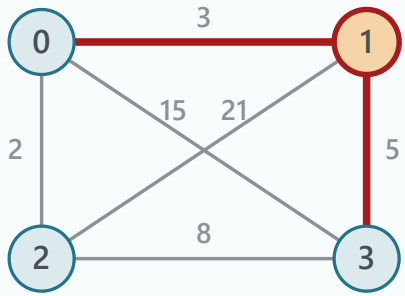
Step 4 / 8

VARIABLES

v = 0 | u = 3 | m1 = 2 | m2 = 3 | sum = 5

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // two smallest
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 3 m2 = 5 sum = 13

v = 1: edges 3, 21, 5 → m1 = 3, m2 = 5. sum = **13**.

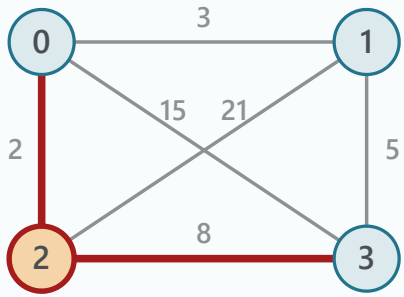
Step 5 / 8

VARIABLES

`v = 1` `u = 3` `m1 = 3` `m2 = 5` `sum = 13`

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // two smallest
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 2 m2 = 8 sum = 23

v = 2: edges 2, 21, 8 → 2 + 8. sum = **23**.

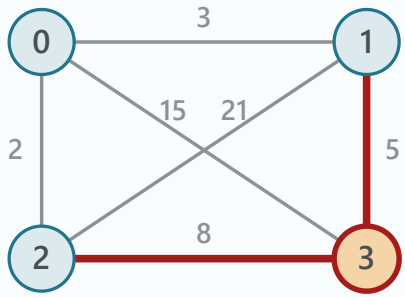
Step 6 / 8

VARIABLES

`v = 2` `u = 3` `m1 = 2` `m2 = 8` `sum = 23`

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // two smallest
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      sum += m1 + m2;
11   }
12   return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 5 m2 = 8 sum = 36

v = 3: edges 15, 5, 8 → 5 + 8. sum = **36**.

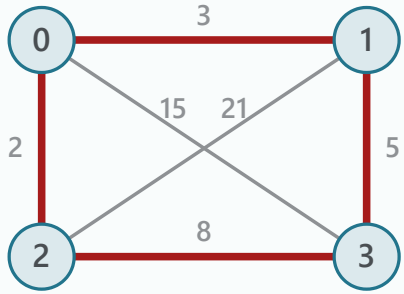
Step 7 / 8

VARIABLES

v = 3 | u = 2 | m1 = 5 | m2 = 8 | sum = 36

twoEdgeBound(): Code Trace

```
1  double twoEdgeBound() {
2    int sum = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // two smallest
5      for (int u = 0; u < n; u++) {
6        if (u == v) continue;
7        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     sum += m1 + m2;
11   }
▶    return sum / 2.0;  // each edge counted twice
13 }
```



m1 = 5 m2 = 8 sum = 36

Returns 36 / 2 = **18**. The bold edges form exactly the optimal tour (0-1-3-2-0), which is why the bound is tight. On a partial path the same idea applies with the path's edges fixed.

Step 8 / 8

VARIABLES

`sum = 36` `result = 18.0`

Lower Bound from the Reduced Cost Matrix

- Set the diagonal to ∞. Subtract the minimum of each **row**, then the minimum of each **column**.
- Every tour uses exactly one cell in each row and each column; the total subtracted comes off every tour's cost → a **lower bound**.
- Example: rows 2 + 3 + 2 + 5 = 12, 2 from column 3 → root bound **14**.
- **Branching** on edge i → j: set row i and column j to ∞, set (j, i) to ∞, and reduce the matrix again. Child bound = parent + reduced c[i][j] + the new reduction.

Branch from the root	bound
0 → 1	14 + 1 + 3 = 18
0 → 2	14 + 0 + 4 = 18
0 → 3	14 + 11 + 3 = 28

	0	**1**	**2**	**3**
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

Reduced matrix (total 14). Every row and column contains at least one 0.

Branch 0 → 3 with 28 is pruned as soon as the optimum 18 is found. The method also works for the asymmetric TSP (d[i][j] ≠ d[j][i]) (Little et al., 1963).

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
▶    int sum = 0;
3    for (int i = 0; i < n; i++) {        // rows
4      int mn = rowMin(m, i);
5      if (mn > 0 && mn < INF) {
6        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```

	0	1	2	3
0	∞	3	2	15
1	3	∞	21	5
2	2	21	∞	8
3	15	5	8	∞

sum = 0

4 × 4 distance matrix with ∞ on the diagonal; sum = 0.

Step 1 / 8

VARIABLES

`i = –` `j = –` `mn = –` `sum = 0`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
▶    for (int i = 0; i < n; i++) {        // rows
▶      int mn = rowMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```

	0	1	2	3
0	∞	1	0	13
1	3	∞	21	5
2	2	21	∞	8
3	15	5	8	∞

sum = 2

mn = 2

Row 0: minimum 2 → subtract 2 from the row: 1, 0, 13. sum = 2.

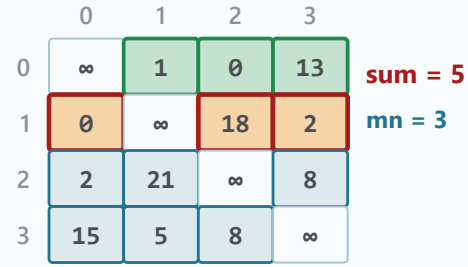
Step 2 / 8

VARIABLES

`i = 0` `mn = 2` `sum = 2`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
▶    for (int i = 0; i < n; i++) {        // rows
▶      int mn = rowMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```



Row 1: minimum 3 → 0, 18, 2. sum = 5.

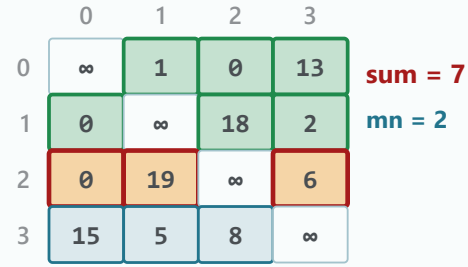
Step 3 / 8

VARIABLES

`i = 1` `mn = 3` `sum = 5`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
▶    for (int i = 0; i < n; i++) {        // rows
▶      int mn = rowMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```



Row 2: minimum 2 → 0, 19, 6. sum = 7.

Step 4 / 8

VARIABLES

`i = 2` `mn = 2` `sum = 7`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
▶    for (int i = 0; i < n; i++) {        // rows
▶      int mn = rowMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```

	0	1	2	3	
0	∞	1	0	13	sum = 12
1	0	∞	18	2	mn = 5
2	0	19	∞	6	
3	10	0	3	∞	

Row 3: minimum 5 → 10, 0, 3. Row reduction done: sum = 12 (the same as the cheapest-exit bound).

Step 5 / 8

VARIABLES

`i = 3` `mn = 5` `sum = 12`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
3    for (int i = 0; i < n; i++) {        // rows
4      int mn = rowMin(m, i);
5      if (mn > 0 && mn < INF) {
6        subtractRow(m, i, mn); sum += mn;
7      }
8    }
▶    for (int j = 0; j < n; j++) {        // columns
▶      int mn = colMin(m, j);
▶      if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```

	0	1	2	3
0	∞	1	0	13
1	0	∞	18	2
2	0	19	∞	6
3	10	0	3	∞

sum = 12

mn = 0

Column 0: minimum 0 → no change. Columns 1 and 2 already contain a 0.

Step 6 / 8

VARIABLES

`j = 0` `mn = 0` `sum = 12`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
3    for (int i = 0; i < n; i++) {        // rows
4      int mn = rowMin(m, i);
5      if (mn > 0 && mn < INF) {
6        subtractRow(m, i, mn); sum += mn;
7      }
8    }
▶    for (int j = 0; j < n; j++) {        // columns
▶      int mn = colMin(m, j);
▶      if (mn > 0 && mn < INF) {
▶        subtractCol(m, j, mn); sum += mn;
13     }
14   }
15   return sum;
16 }
```

	0	1	2	3
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

sum = 14

mn = 2

Column 3: values 13, 2, 6 → minimum 2; subtracted: 11, 0, 4. sum = **14**. The column step adds the fact that “each city is entered once” to the bound.

Step 7 / 8

VARIABLES

`j = 3` `mn = 2` `sum = 14`

reduce(): Row and Column Reduction

```
1  int reduce(int[][] m) {
2    int sum = 0;
3    for (int i = 0; i < n; i++) {        // rows
4      int mn = rowMin(m, i);
5      if (mn > 0 && mn < INF) {
6        subtractRow(m, i, mn); sum += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // columns
10     int mn = colMin(m, j);
11     if (mn > 0 && mn < INF) {
12       subtractCol(m, j, mn); sum += mn;
13     }
14   }
▶    return sum;
16 }
```

	0	1	2	3
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

sum = 14

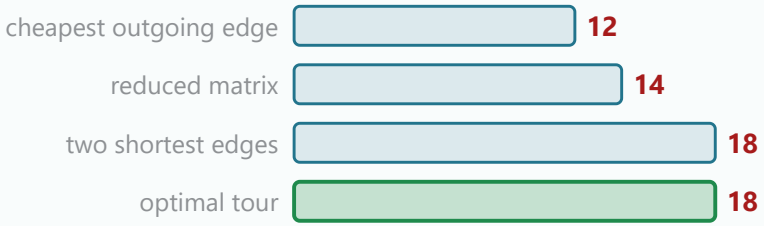
Returns bound 14: tighter than 12, looser than the two-edge bound (18). The matrix is kept for computing the children's bounds.

Step 8 / 8

VARIABLES

`sum = 14`

Three TSP Bounds: Tightness and Cost



Bound	Cost per node	What it uses	Note
Cheapest outgoing edge	O(n) (minOut[] precomputed)	one exit from each city	cheapest, loosest
Reduced matrix	$O(n^2)$	one entry + one exit per city	also valid for asymmetric TSP
Two shortest edges	$O(n^2)$	two edges touching each city	for symmetric TSP

A tighter bound prunes more but costs more at each node. Even tighter bounds (1-trees, LP relaxation) are used on large instances; the choice balances nodes pruned against work per node.

A Larger Example: 6 Cities

	0	1	2	3	4	5	min
0	—	29	20	21	16	31	**16**
1	29	—	15	29	28	40	**15**
2	20	15	—	15	14	25	**14**
3	21	29	15	—	12	18	**12**
4	16	28	14	12	—	22	**12**
5	31	40	25	18	22	—	**18**

- Backtracking without pruning tries all (n−1)! = 120 tours; with the `lowerBound` test the same answer is found with far fewer nodes.
- Root lower bound: 16 + 15 + 14 + 12 + 12 + 18 = **87**; optimum **115**.
- Best tour: **0** → **1** → **2** → **3** → **5** → **4** → **0** = 29 + 15 + 15 + 18 + 22 + 16 = 115.

Version	Nodes generated	tsp calls
No pruning (backtracking)	326	326
Cheapest-exit bound	210	130

Tighter bounds

For the symmetric TSP, half the sum of each city's **two** shortest edges; or the row/column **reduced cost matrix** (Little et al., 1963). A tighter bound prunes more but costs more at each node.

Assignment Problem: n People, n Jobs

- c[i][j]: the cost of person i doing job j. Give each person a different job so that the total is minimized; there are n! possible assignments.
- State space: at level i, pick a free job for person i.
- **Lower bound:** each remaining person pays at least the minimum of their own row (job clashes are ignored):

$$lb = next + restMin[i + 1], \quad restMin[i] = \Sigma_{r \geq i} \min_j c[r][j]$$

Cut the branch if lb ≥ best. Row minima 4, 3, 2 → restMin = [9, 5, 2, 0]: the root bound is 9.

person \ job	0	1	2	min
0	5	8	4	**4**
1	7	3	6	**3**
2	2	9	5	**2**

The assignment problem can also be solved in polynomial time (the Hungarian method, $O(n^3)$). Here branch and bound just shows the method on a small, easy-to-trace example; it is truly needed for variants with extra constraints (e.g. some jobs cannot be done together).

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete
   assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
7      if (taken[j]) continue;
8      int next = cost + c[i][j];
9      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
11     taken[j] = true;
12     assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```

	job 0	job 1	job 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

best = ∞

restMin = [9, 5, 2, 0]; best = ∞. assign(0, 0) is called: jobs are tried in order for person 0.

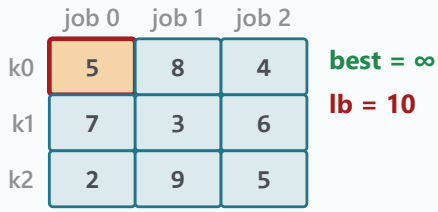
Step 1 / 10

VARIABLES

`i = 0` `j = –` `next = –` `lb = –` `best = ∞`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p0 → job 0: next = 5, lb = 5 + restMin[1] = 10 < ∞ → branch.

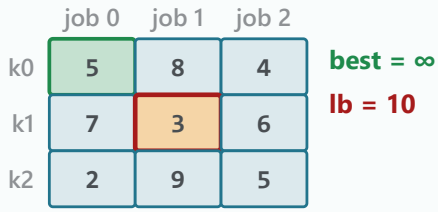
Step 2 / 10

VARIABLES

`i = 0` `j = 0` `next = 5` `lb = 10` `best = ∞`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p1: job 0 is taken, skipped. Job 1: next = 8, lb = 8 + 2 = 10 → branch.

Step 3 / 10

VARIABLES

`i = 1` `j = 1` `next = 8` `lb = 10` `best = ∞`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
▶    if (i == n) {                    // complete assignment
▶      best = Math.min(best, cost);
▶      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p2: only job 2 is free: next = 13 → i = 3, complete assignment: best = **13**.

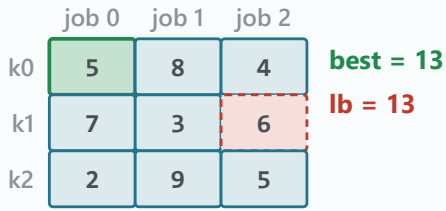
Step 4 / 10

VARIABLES

`i = 3` `cost = 13` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
▶        continue;                    // bound
11     taken[j] = true;
12     assign(i + 1, next);
▶      taken[j] = false;              // undo
14   }
15 }
```



Back up. p1 → job 2: next = 11, lb = 11 + 2 = 13 ≥ 13 → **cut**.

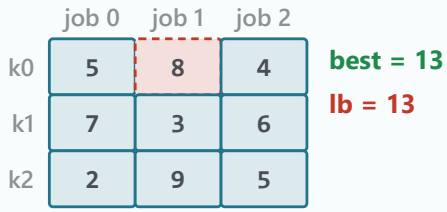
Step 5 / 10

VARIABLES

`i = 1` `j = 2` `next = 11` `lb = 13` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
▶        continue;                    // bound
11     taken[j] = true;
12     assign(i + 1, next);
▶      taken[j] = false;              // undo
14   }
15 }
```



p0 → job 1: next = 8, lb = 8 + 5 = 13 ≥ 13 → cut; the 2 assignments below it are never tried.

Step 6 / 10

VARIABLES

`i = 0` `j = 1` `next = 8` `lb = 13` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p0 → job 2: next = 4, lb = 9 < 13 → branch.

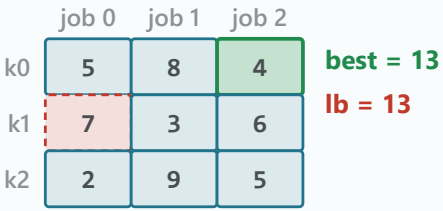
Step 7 / 10

VARIABLES

`i = 0` `j = 2` `next = 4` `lb = 9` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete
   assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
▶        continue;                    // bound
11     taken[j] = true;
12     assign(i + 1, next);
13     taken[j] = false;                // undo
14   }
15 }
```



p1 → job 0: next = 11, lb = 13 ≥ 13 → cut.

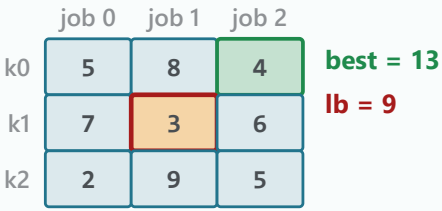
Step 8 / 10

VARIABLES

`i = 1` `j = 0` `next = 11` `lb = 13` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
2    if (i == n) {                    // complete assignment
3      best = Math.min(best, cost);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p1 → job 1: next = 7, lb = 9 → branch.

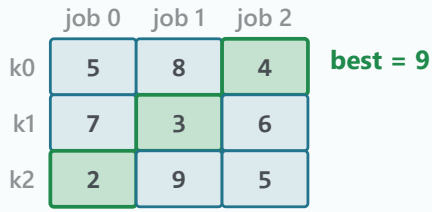
Step 9 / 10

VARIABLES

`i = 1` `j = 1` `next = 7` `lb = 9` `best = 13`

assign(0, 0): Code Trace

```
1  void assign(int i, int cost) {
▶    if (i == n) {                    // complete assignment
▶      best = Math.min(best, cost);
▶      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (taken[j]) continue;
▶      int next = cost + c[i][j];
▶      if (next + restMin[i + 1] >= best)
10       continue;                    // bound
▶      taken[j] = true;
▶      assign(i + 1, next);
13     taken[j] = false;              // undo
14   }
15 }
```



p2 → job 0: next = 9 → complete assignment, best = **9** (4 + 3 + 2). It equals the bound, so nothing better exists. 7 calls; a search without pruning would visit all 3! = 6 assignments (16 calls).

Step 10 / 10

VARIABLES

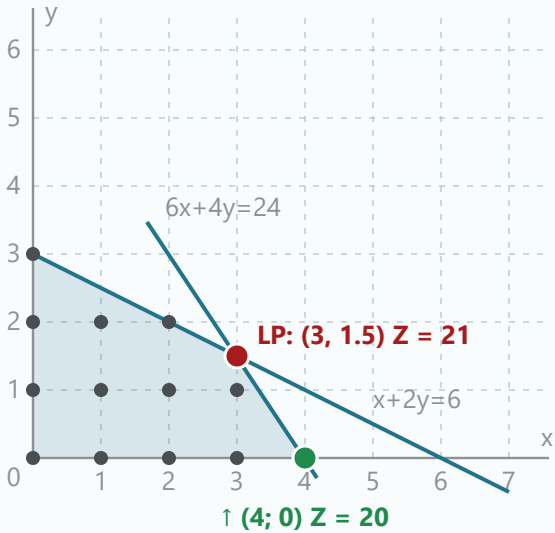
`i = 3` `cost = 9` `best = 9`

Integer Linear Programming (ILP)

- Every variable of the linear program must be an **integer**. This is the most common industrial use of branch and bound (Land and Doig, 1960).
- **Bound:** drop the integrality condition and solve the **LP relaxation**. The LP optimum is an upper bound on the ILP optimum.
- **Branching:** for a fractional variable (y = 1.5), two subproblems: y ≤ 1 and y ≥ 2. The fractional region in between is discarded; no integer point is lost.

Example: maximize Z = 5x + 4y; 6x + 4y ≤ 24, x + 2y ≤ 6, x, y ≥ 0 integers.

Second example: Z = 3x + 2y; x + y ≤ 4, x ≤ 3, y ≤ 2. Brute force tries all 12 points of the grid 0 ≤ x ≤ 3, 0 ≤ y ≤ 2 → (3, 1), Z = 11. Here the LP relaxation also gives (3, 1): branch and bound ends **at the root**.



Shaded: the LP feasible region; dots: integer solutions.

ILP: Branching on LP Relaxations

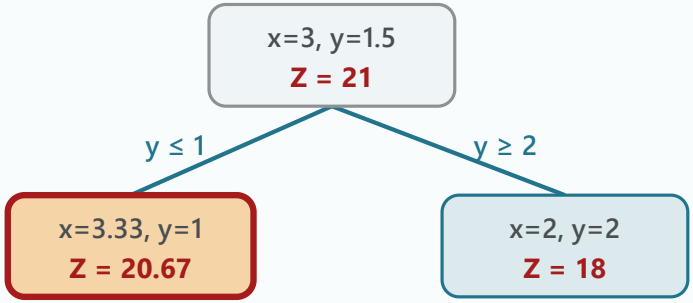
x=3, y=1.5
Z = 21

Root: the integrality condition is relaxed (LP relaxation). The LP optimum (3, 1.5), **Z = 21** → upper bound. y is fractional → branch on y.

Step 1 / 5

Rounding the LP optimum (3, 1.5) gives (3, 1) → Z = 19, or (3, 2) → infeasible. The true optimum (4, 0) cannot be found by rounding.

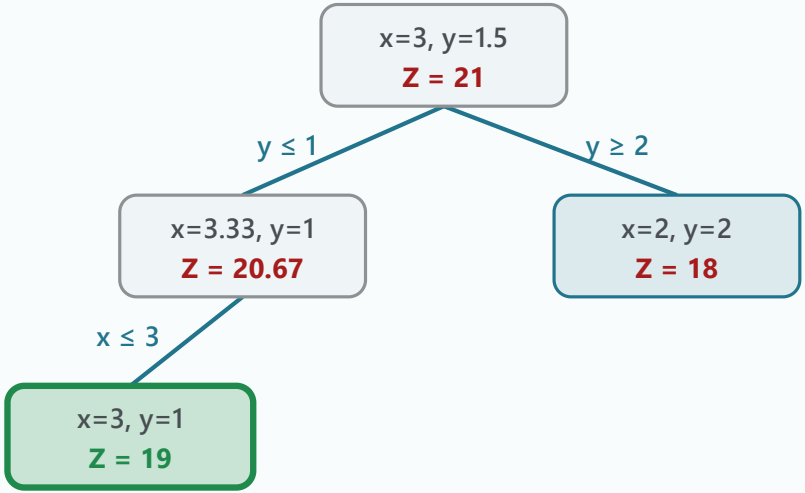
ILP: Branching on LP Relaxations



Branch y ≤ 1: LP optimum (3.33, 1), Z = 20.67. x is fractional → open branches x ≤ 3 and x ≥ 4.

Rounding the LP optimum (3, 1.5) gives (3, 1) → Z = 19, or (3, 2) → infeasible. The true optimum (4, 0) cannot be found by rounding.

ILP: Branching on LP Relaxations

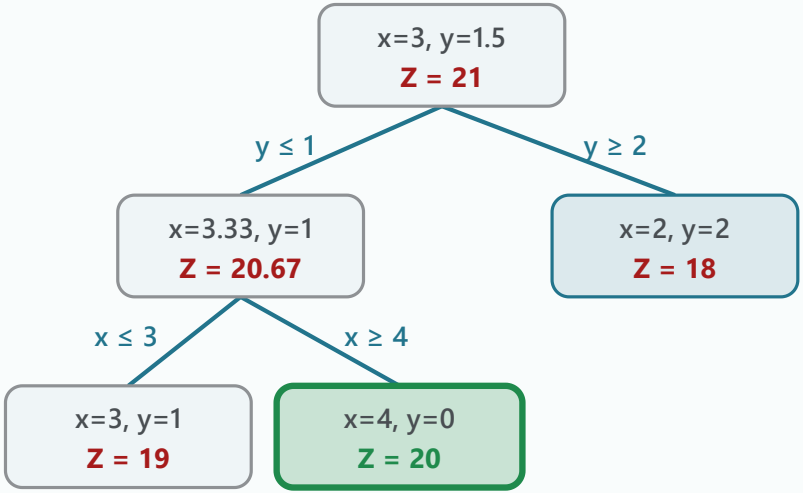


Branch x ≤ 3: the LP solution (3, 1) is **integral** → first incumbent, best = 19.

Step 3 / 5

Rounding the LP optimum (3, 1.5) gives (3, 1) → Z = 19, or (3, 2) → infeasible. The true optimum (4, 0) cannot be found by rounding.

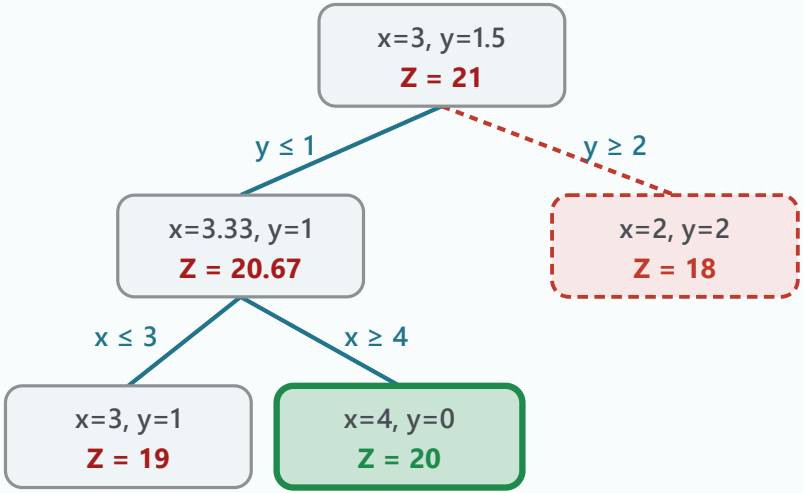
ILP: Branching on LP Relaxations



Branch x ≥ 4: the LP solution (4, 0) is integral, Z = 20 > 19 → **best = 20**.

Rounding the LP optimum (3, 1.5) gives (3, 1) → Z = 19, or (3, 2) → infeasible. The true optimum (4, 0) cannot be found by rounding.

ILP: Branching on LP Relaxations

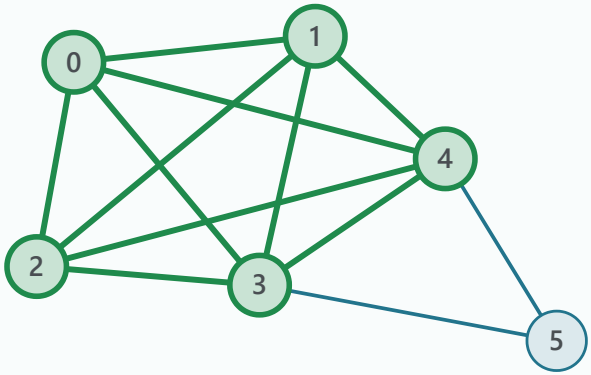


Branch y ≥ 2: LP bound Z = 18 ≤ 20 → **pruned**. Optimum: x = 4, y = 0, **Z = 20**.

Step 5 / 5

Rounding the LP optimum (3, 1.5) gives (3, 1) → Z = 19, or (3, 2) → infeasible. The true optimum (4, 0) cannot be found by rounding.

Maximum Clique



A 6-vertex example graph; the largest clique is {0, 1, 2, 3, 4}.

- **Clique:** a set of vertices in which every two are adjacent.
- **Bound:** if |K| + |cand| ≤ |best|, the current clique cannot beat best however much it grows.

```
void clique(List<Integer> K, int start) {
  if (K.size() > best.size())
    best = new ArrayList<>(K);
  List<Integer> cand = new ArrayList<>();
  for (int v = start; v < n; v++)
    if (adjToAll(v, K)) cand.add(v); // adjacent to K
  if (K.size() + cand.size() <= best.size())
    return;                          // bound
  for (int v : cand) {
    K.add(v);
    clique(K, v + 1);
    K.remove(K.size() - 1);          // undo
  }
}
```

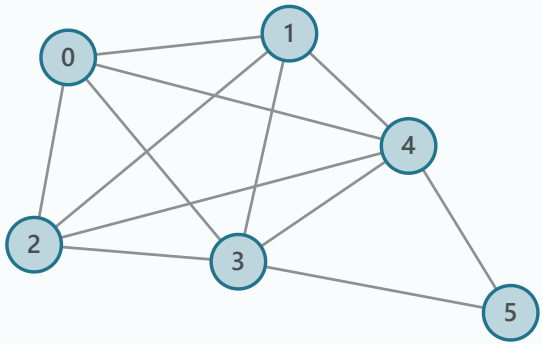
Version	clique calls
Backtracking without a bound	36
With the |K| + |cand| bound	17

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 ▶   List<Integer> cand = new ArrayList<>();
 ▶   for (int v = start; v < n; v++)
 ▶     if (adjToAll(v, K)) cand.add(v); // adjacent to
   K
 ▶   if (K.size() + cand.size() <= best.size())
 8     return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({})`
- `main()`



|K| + |cand| = 0 + 6 = 6 ; |best| = 0

Six-vertex graph; K is empty, best is empty. All of 0–5 are candidates: 0 + 6 > 0, the bound passes; the first candidate 0 is added and we go deeper.

Step 1 / 10

VARIABLES

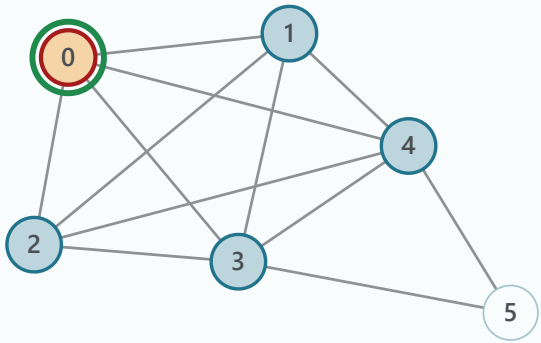
`K = {}` `start = 0` `cand = {0, 1, 2, 3, 4, 5}` `best = {}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 ▶   if (K.size() > best.size())
 ▶     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 ▶     if (adjToAll(v, K)) cand.add(v); // adjacent to
       K
 7   if (K.size() + cand.size() <= best.size())
 8     return;                          // bound
 ▶   for (int v : cand) {
 ▶     K.add(v);
 ▶     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 1 + 4 = 5 ; |best| = 1

K = {0} is larger than best → **best = {0}**. Candidates adjacent to all of K: {1, 2, 3, 4}; 1 + 4 > 1 → continue, the first candidate 1 is added.

Step 2 / 10

VARIABLES

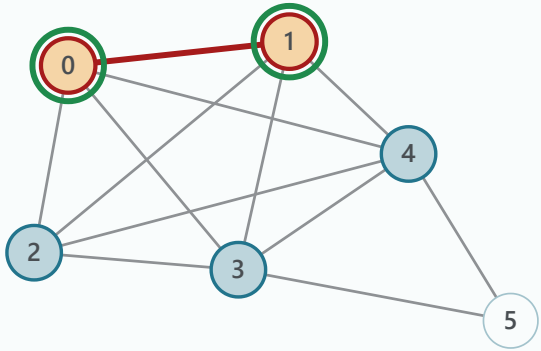
`K = {0}` `start = 1` `cand = {1, 2, 3, 4}` `best = {0}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to K
 7   if (K.size() + cand.size() <= best.size())
 8     return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 2 + 3 = 5 ; |best| = 2

K = {0, 1} is larger than best → **best = {0, 1}**. Candidates adjacent to all of K: {2, 3, 4}; 2 + 3 > 2 → continue, the first candidate 2 is added.

Step 3 / 10

VARIABLES

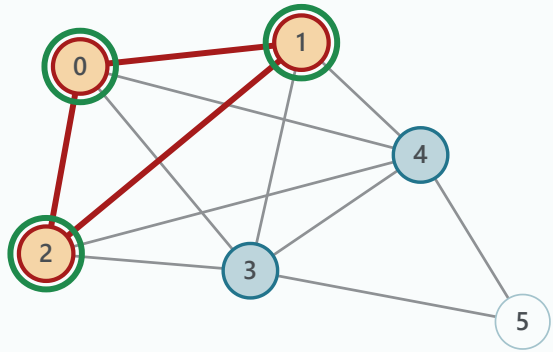
`K = {0, 1}` `start = 2` `cand = {2, 3, 4}` `best = {0, 1}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 ▶   if (K.size() > best.size())
 ▶     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 ▶     if (adjToAll(v, K)) cand.add(v); // adjacent to
       K
 7   if (K.size() + cand.size() <= best.size())
 8     return;                          // bound
 ▶   for (int v : cand) {
 ▶     K.add(v);
 ▶     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1, 2})`
- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 3 + 2 = 5 ; |best| = 3

K = {0, 1, 2} is larger than best → **best = {0, 1, 2}**. Candidates adjacent to all of K: {3, 4}; 3 + 2 > 3 → continue, the first candidate 3 is added.

Step 4 / 10

VARIABLES

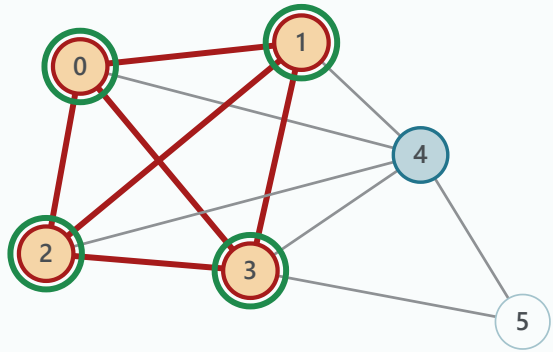
`K = {0, 1, 2}` `start = 3` `cand = {3, 4}` `best = {0, 1, 2}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 ▶   if (K.size() > best.size())
 ▶     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 ▶     if (adjToAll(v, K)) cand.add(v); // adjacent to
       K
 7   if (K.size() + cand.size() <= best.size())
 8     return;                          // bound
 ▶   for (int v : cand) {
 ▶     K.add(v);
 ▶     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1, 2, 3})`
- `clique({0, 1, 2})`
- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 4 + 1 = 5 ; |best| = 4

K = {0, 1, 2, 3} is larger than best → **best = {0, 1, 2, 3}**. Candidates adjacent to all of K: {4}; 4 + 1 > 4 → continue, the first candidate 4 is added.

Step 5 / 10

VARIABLES

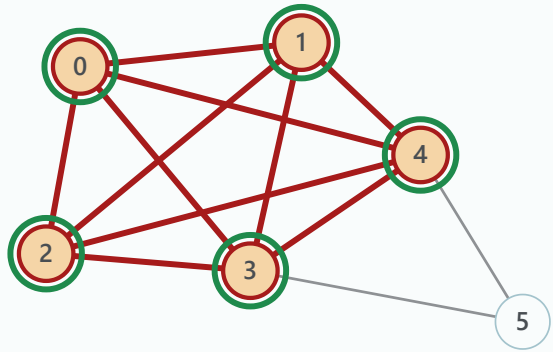
`K = {0, 1, 2, 3}` `start = 4` `cand = {4}` `best = {0, 1, 2, 3}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 ▶   if (K.size() > best.size())
 ▶     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to
     K
 ▶   if (K.size() + cand.size() <= best.size())
 ▶     return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
12     K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1, 2, 3, 4})`
- `clique({0, 1, 2, 3})`
- `clique({0, 1, 2})`
- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 5 + 0 = 5 ; |best| = 5

K = {0, 1, 2, 3, 4}: **best = {0, 1, 2, 3, 4}** (5 vertices). No candidates left: 5 + 0 ≤ 5 → `return` .

Step 6 / 10

VARIABLES

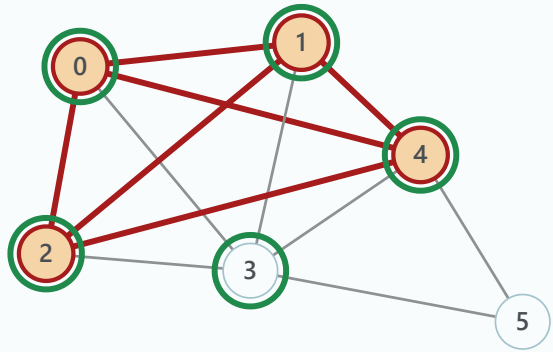
`K = {0, 1, 2, 3, 4}` `start = 5` `cand = {}` `best = {0, 1, 2, 3, 4}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to
   K
▶  if (K.size() + cand.size() <= best.size())
▶    return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
▶    K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1, 2, 4})`
- `clique({0, 1, 2})`
- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 4 + 0 = 4 ; |best| = 5

Undo, then try K = {0, 1, 2, 4}: cand {}, 4 + 0 = 4 ≤ 5 → **pruned by the bound**; this clique cannot beat 5.

Step 7 / 10

VARIABLES

`K = {0, 1, 2, 4}` `start = 5` `cand = {}`

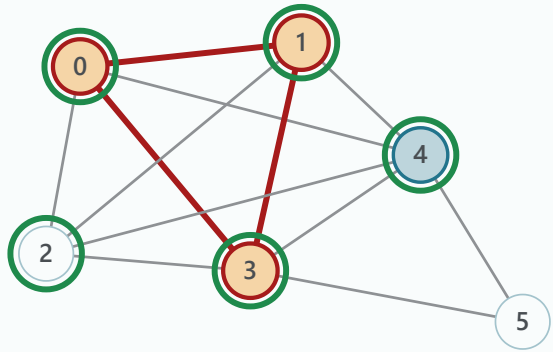
`best = {0, 1, 2, 3, 4}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to
   K
▶  if (K.size() + cand.size() <= best.size())
▶    return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
▶    K.remove(K.size() - 1);          // undo
13   }
14 }
```

CALL STACK

- `clique({0, 1, 3})`
- `clique({0, 1})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 3 + 1 = 4 ; |best| = 5

Undo, then try K = {0, 1, 3}: cand {4}, 3 + 1 = 4 ≤ 5 → **pruned by the bound**; this clique cannot beat 5.

Step 8 / 10

VARIABLES

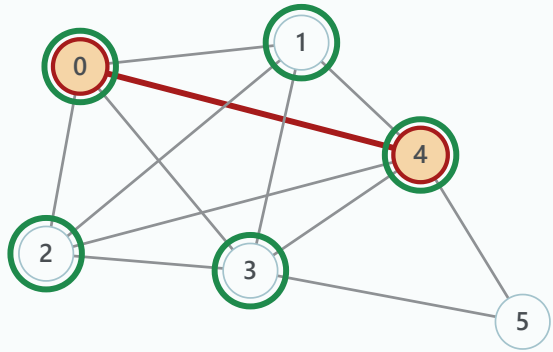
`K = {0, 1, 3}` `start = 4` `cand = {4}` `best = {0, 1, 2, 3, 4}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to
   K
▶  if (K.size() + cand.size() <= best.size())
▶    return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
▶      K.remove(K.size() - 1);         // undo
13   }
14 }
```

CALL STACK

- `clique({0, 4})`
- `clique({0})`
- `clique({})`
- `main()`



|K| + |cand| = 2 + 0 = 2 ; |best| = 5

Likewise {0, 1, 4}: 3 + 0; {0, 2}: 2 + 2; {0, 3}: 2 + 1; {0, 4}: 2 + 0. All ≤ 5, pruned.

Step 9 / 10

VARIABLES

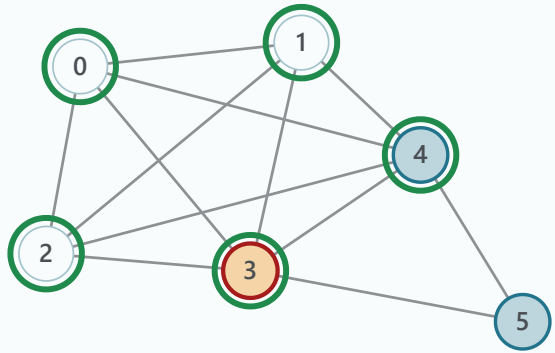
`K = {0, 4}` `start = 5` `cand = {}` `best = {0, 1, 2, 3, 4}`

clique({}, 0): Bounded Search Trace

```
 1 void clique(List<Integer> K, int start) {
 2   if (K.size() > best.size())
 3     best = new ArrayList<>(K);
 4   List<Integer> cand = new ArrayList<>();
 5   for (int v = start; v < n; v++)
 6     if (adjToAll(v, K)) cand.add(v); // adjacent to
   K
▶  if (K.size() + cand.size() <= best.size())
▶    return;                          // bound
 9   for (int v : cand) {
10     K.add(v);
11     clique(K, v + 1);
12     K.remove(K.size() - 1);         // undo
13   }
14 }
```

CALL STACK

- clique({3})
- clique({})
- main()



|K| + |cand| = 1 + 2 = 3 ; |best| = 5

The remaining root candidates 1–5 are tried one by one ({1}: 4, {2}: 3, {3}: 3, {4}: 2, {5}: 1); all ≤ 5 → pruned. Result **{0, 1, 2, 3, 4}**, 17 calls (36 without the bound).

Step 10 / 10

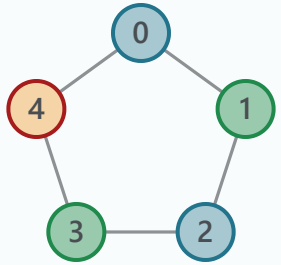
VARIABLES

K = {3} | start = 4 | cand = {4, 5}

best = {0, 1, 2, 3, 4}

A Tighter Bound for Clique: Coloring

- Every pair of vertices in a clique is adjacent, so all vertices of a clique get **different colors**.
- If the candidates can be colored with k colors, at most k of them can join the clique: prune if **|K| + colors(cand) ≤ |best|**.
- Greedy coloring takes O(cand^2) and, even if not optimal, gives a valid upper bound (extra colors only loosen the bound).
- Example: the candidates form a 5-vertex ring, K is empty, best = 3. The old bound 0 + 5 > 3 → the branch is opened; the color bound 0 + 3 ≤ 3 → **pruned** (the largest clique in a ring is 2).



3 colors:

{0, 2} {1, 3} {4}

|cand| = 5

color bound = 3

Most of the fastest exact clique algorithms in practice (Tomita's MCQ / MCS family) rest on this idea: try candidates in color order and prune with the color bound.

OTHER APPLICATIONS

Other Problems Solved by Branch and Bound

Hamiltonian path

A path visiting every vertex exactly once; the weighted version is solved with bounds much like TSP's.

Satisfiability (SAT)

Assign truth values to variables; prune conflicting partial assignments (DPLL). For MAX-SAT the bound is the most clauses that can still be satisfied.

Scheduling

Minimize makespan or tardiness; bound: the best-case load of the remaining jobs.

Bin packing

Pack items into as few bins as possible; lower bound ⌈Σ size / bin capacity⌉.

Maximum coverage

Choose k sets covering the most elements; bound: the elements the largest remaining sets could add.

Assignment problem

Assign n jobs to n people at minimum cost; lower bound: the minimum cost of each unassigned row.

Problems with polynomial-time algorithms, such as maximum flow, do not need branch and bound; the method makes sense for NP-hard problems.

Branch and Bound, A* and Depth-First Branch and Bound

Feature	Best-first branch and bound (LC)	A*	Depth-first branch and bound
Selection	best bound ($\hat{c}$)	smallest f = g + h	stack / recursion
Bound / estimate	valid (optimistic) bound	admissible h (never overestimates)	valid bound + best
Goal	optimization (best leaf)	shortest path from start to goal	optimization
Memory	live nodes: can be exponential	open list: can be exponential	O(depth)
First complete solution	usually late but good	the first goal reached is optimal	early (a solution is always at hand)

- A* is LC branch and bound searching for a path in a weighted graph: h(x) is a lower bound on the remaining cost from x to the goal (misplaced tiles in the 8-puzzle).
- When memory runs out, the depth-first version or IDA* (depth-first search with an iteratively increasing bound threshold) is used.

EVALUATION

Design Paradigms: A Cumulative Comparison

Paradigm	Decision	Subproblems / traversal	When?	Example
Brute force	generate and test every candidate	the full search space	tiny search spaces	8^8 queen placements
Divide and conquer	divide, solve, combine	independent; top-down	non-overlapping subproblems	merge sort, Strassen
Greedy	best-looking choice; never undone	one subproblem after each choice	greedy-choice property holds	Dijkstra, Huffman, Kruskal
Dynamic programming	try all options, keep the best	overlapping; mostly bottom-up	optimal substructure, polynomially many states	Fibonacci, 0/1 knapsack, LCS
Backtracking	extend; undo if invalid	DFS on the state-space tree	constraint satisfaction, all solutions	n-queens, Sudoku
Branch and bound	backtracking + pruning by bound	DFS, BFS or best-first	NP-hard optimization, exact result	0/1 knapsack, TSP

Independent subproblems → **divide and conquer**. Overlapping → **dynamic programming**. Locally best is globally best → **greedy**. NP-hard: **backtracking / branch and bound** for an exact answer, an **approximation algorithm** for a guaranteed one in polynomial time.

Summary

Topic	Key points
Paradigm	state-space tree; select → branch → bound → prune; best = best complete solution so far
Bound	valid (optimistic), tight, cheap; from a relaxation. Upper for max, lower for min
Pruning	infeasible, bound ≤ best (lb ≥ best), complete / integral LP
Strategies	FIFO (queue), LIFO (stack, little memory), LC / best-first (priority queue, few nodes)
Knapsack	fractional upper bound; bound(root) = 115, optimum 90; 11 nodes instead of 31
TSP	lb = cost + cheapest exit edge of each unfinished city; 16 → 10 nodes for 4 cities
ILP / clique	LP relaxation + branch on a fractional variable; clique: prune if |K| + |cand| ≤ |best|
Complexity	exponential in the worst case; speed in practice depends on the bound
Paradigms	independent → D&C; overlapping → DP; greedy choice → greedy; NP-hard exact → backtracking / B&B

SUMMARY

Common Mistakes

Invalid (pessimistic) bound

In knapsack, summing only the items that fit whole is not an upper bound; the branch containing the optimum may be pruned.

Mixing up the direction

Prune when bound ≤ best for maximization and lb ≥ best for minimization. Writing the reverse either misses the optimum or prunes nothing.

Not re-testing after removal

The node was promising when it was added; if best grew while it waited, it is now useless (C and G in the example).

Not sorting the items

The fractional bound is not a valid upper bound unless items are in decreasing order of p_i/w_i.

Updating best only at leaves

In knapsack every feasible node is a solution; updating early gives more pruning.

Confusing it with backtracking

Code without a bound test (e.g. a TSP search with no pruning) is not branch and bound; it just walks the whole tree.

SUMMARY

Sample Problems

Knapsack: (p, w) = (20, 2), (30, 5), (35, 7), (12, 3); W = 9 (ratios 10, 6, 5, 4).

1. Compute the root's fractional upper bound.
2. What is the optimal profit, and which items are taken?
3. What is the bound of the node "item 1 not taken" (0, 0)? Is it pruned once the optimum is found?
4. In the 4-city TSP example, what is the lower bound of the path 0 → 3 → 1?
5. In maximum clique, what happens when K = {3, 4}, cand = {5} and |best| = 5?
6. Which search strategy should be used on a system with very little memory?

▶ **Show answers**

Exercises

1. Write the knapsack code with a FIFO queue; how many nodes are generated on the chapter's example?
2. Add an `int[] bestPath` to the `tsp` method that records the best tour.
3. Compute the "half the sum of the two shortest edges" bound at the root of the 4-city example.
4. For the assignment problem with cost matrix [[9, 2, 7], [6, 4, 3], [5, 8, 1]], what are the root lower bound and the optimum?
5. Prove that the fractional knapsack bound is always valid.
6. In how many nodes does LP-based branch and bound solve the second ILP example ($Z = 3x + 2y$; $x + y \le 4$, $x \le 3$, $y \le 2$)?

▶ **Answers / hints**

References

- E. Horowitz, S. Sahni and S. Rajasekaran, *Computer Algorithms*, 2nd ed., Silicon Press, 2008 (Chapter 8: Branch-and-Bound).
- R. Neapolitan, *Foundations of Algorithms*, 5th ed., Jones & Bartlett, 2015 (Chapter 6).
- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012 (Section 12.2).
- A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems", *Econometrica* 28(3), 1960.
- J. D. C. Little, K. G. Murty, D. W. Sweeney and C. Karel, "An Algorithm for the Traveling Salesman Problem", *Operations Research* 11(6), 1963.
- R. Carraghan and P. M. Pardalos, "An Exact Algorithm for the Maximum Clique Problem", *Operations Research Letters* 9(6), 1990.

The End

Chapter 5: Branch and Bound

Previous: Chapter 4: Backtracking · Next: Chapter 6: Approximation Algorithms