Optimization Methods

Categorization of Optimization Algorithms

Is the obtained solution always optimal?

1: Exact Optimization Algorithms

- Linear Programming
- Dynamic Programming
- Branch-and-Bound Method

2: Approximation Algorithms

- Greedy Algorithms
- Local Search
- Genetic Algorithms
- Almost All Nonlinear Optimization Algorithms
- Learning of Connection Weights in Neural Networks

In the first six weeks

Categorization of Optimization Algorithms

Is the obtained solution always optimal?

1: Exact Optimization Algorithms

- Linear Programming
- Dynamic Programming
- Branch-and-Bound Method

The solution is always optimal.

2: Approximation Algorithms

- Greedy Algorithms
- Local Search
- Genetic Algorithms
- Almost All Nonlinear Optimization Algorithms
- Learning of Connection Weights in Neural Networks

Is the obtained solution always optimal?

Approximation algorithms can find the optimal solution with a high probability especially for small-size problems. However, we do not know whether the obtained solution is optimal or not. Exact optimization algorithms always find the optimal solution with clear theoretical support about the optimality of the obtained solution.

1: Exact Optimization Algorithms

- Linear Programming
- Dynamic Programming
- Branch-and-Bound Method

2: Approximation Algorithms

- Greedy Algorithms
- Local Search
- Genetic Algorithms
- Almost All Nonlinear Optimization Algorithms
- Learning of Connection Weights in Neural Networks

Flow Shop Scheduling

Input: Job set: n jobs ($i = 1, 2, ..., n$)

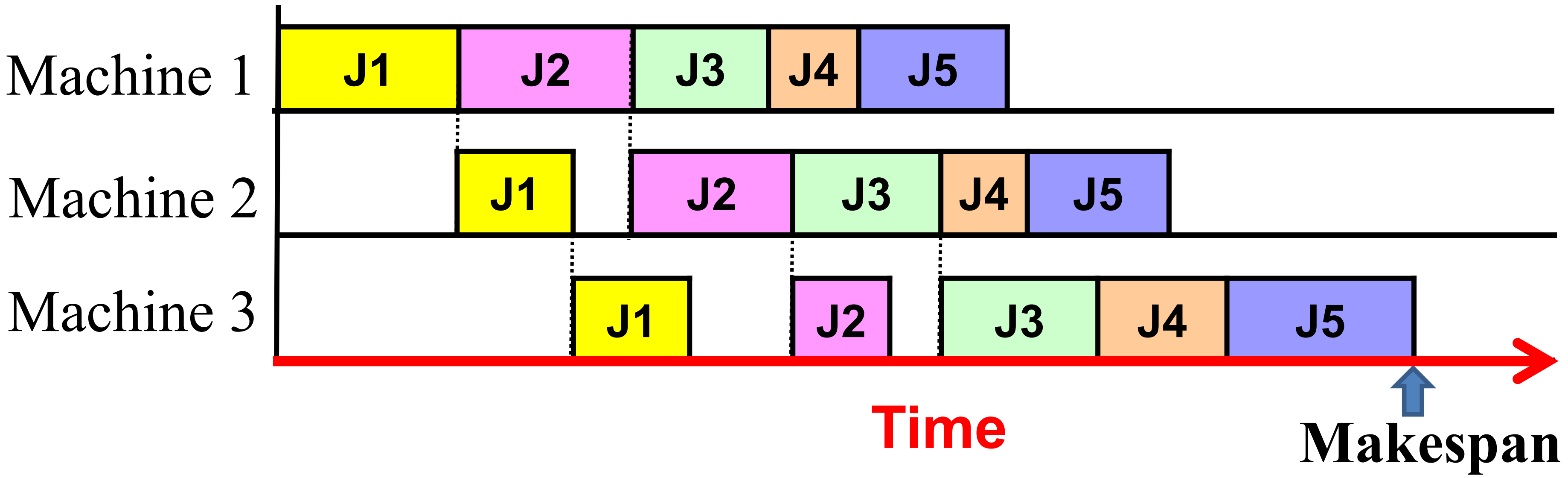
Machine set: m machines ($j = 1, 2, ..., m$)

Processing time: p_{ij} of job i on machine j

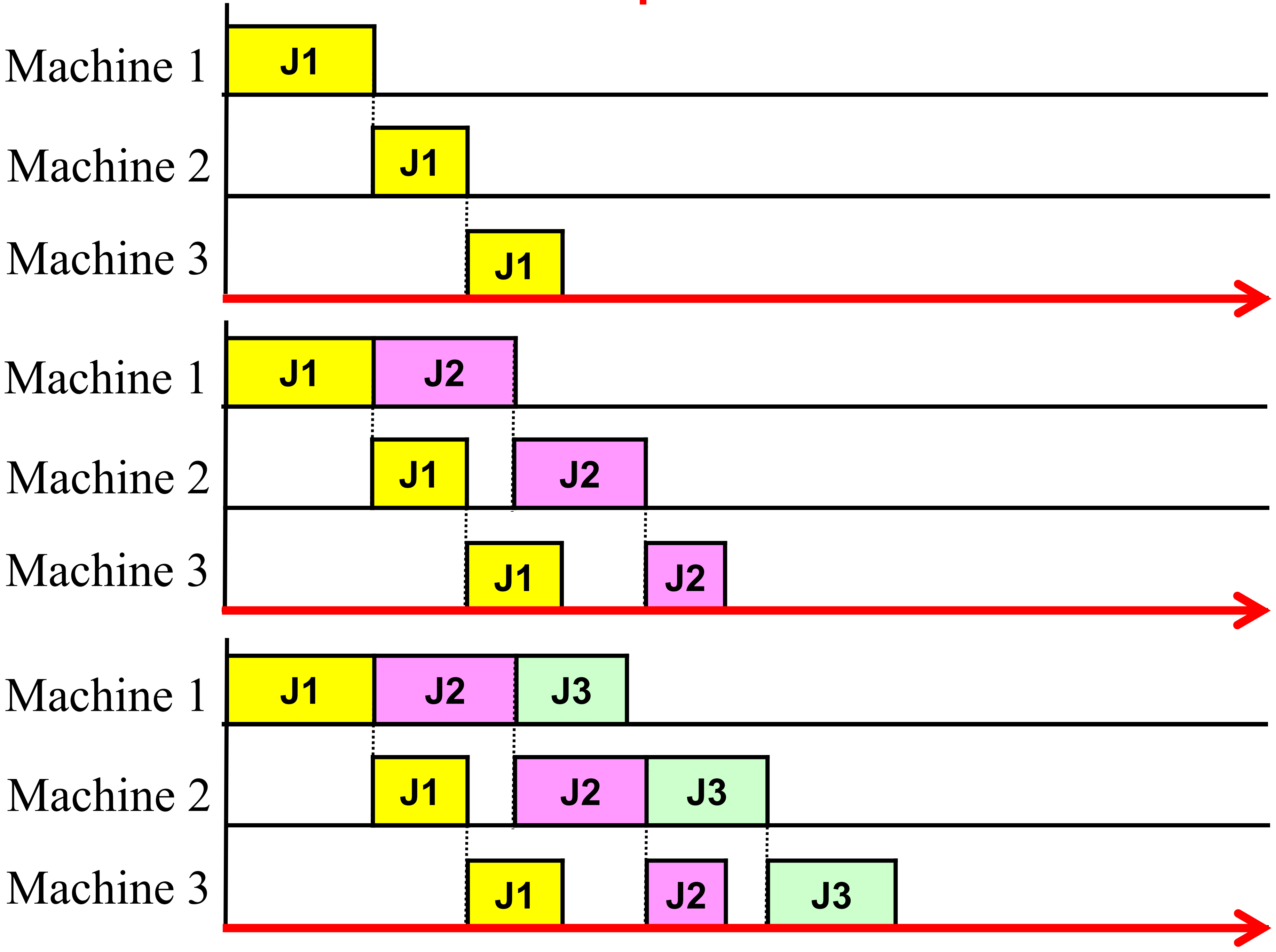
Objective: Minimization of the makespan

Conditions: Each job should be processed by all machines in the same order: machine 1, machine 2, ..., machine m.

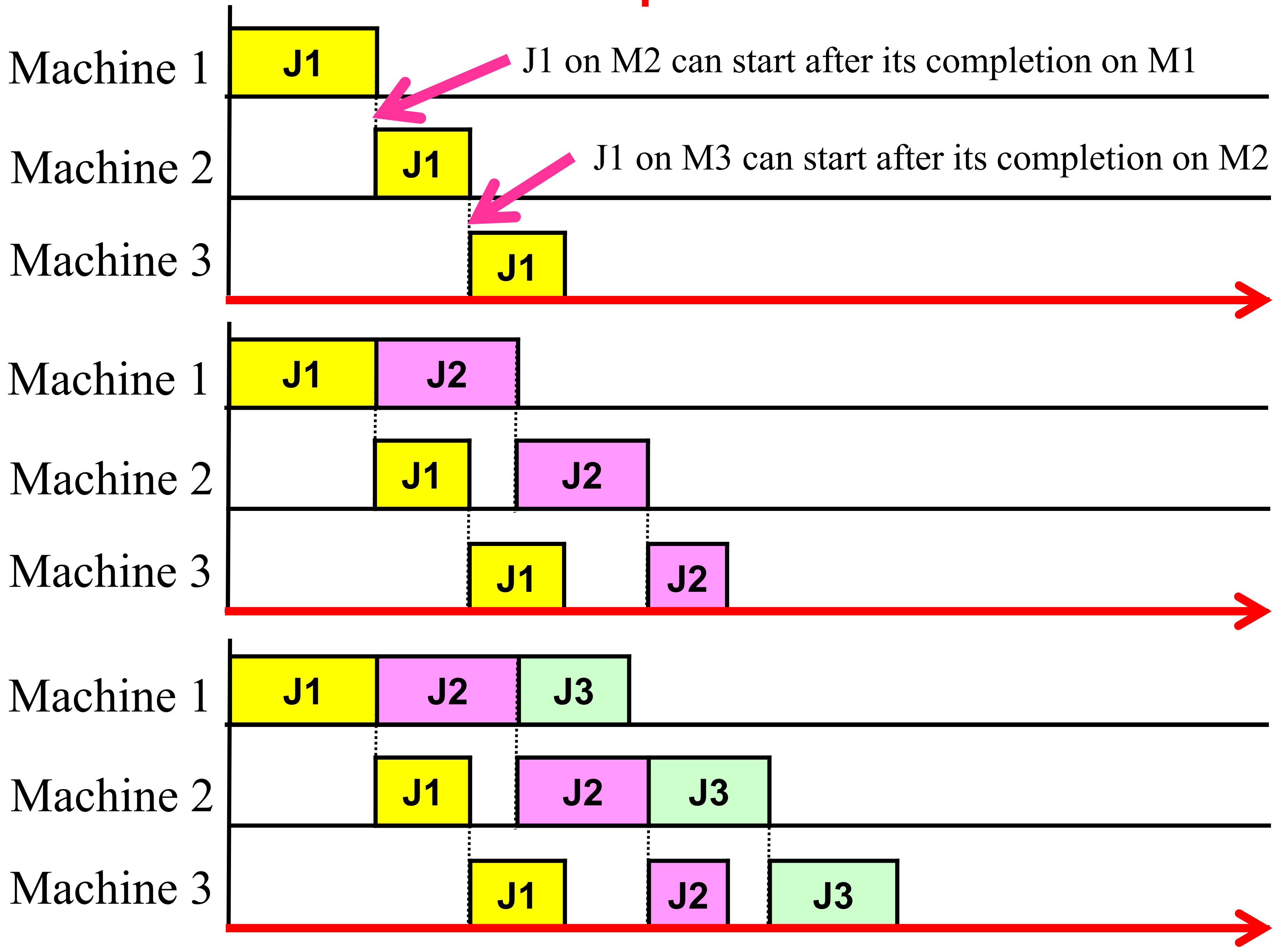
Output: Order of the n jobs



Makespan Calculation

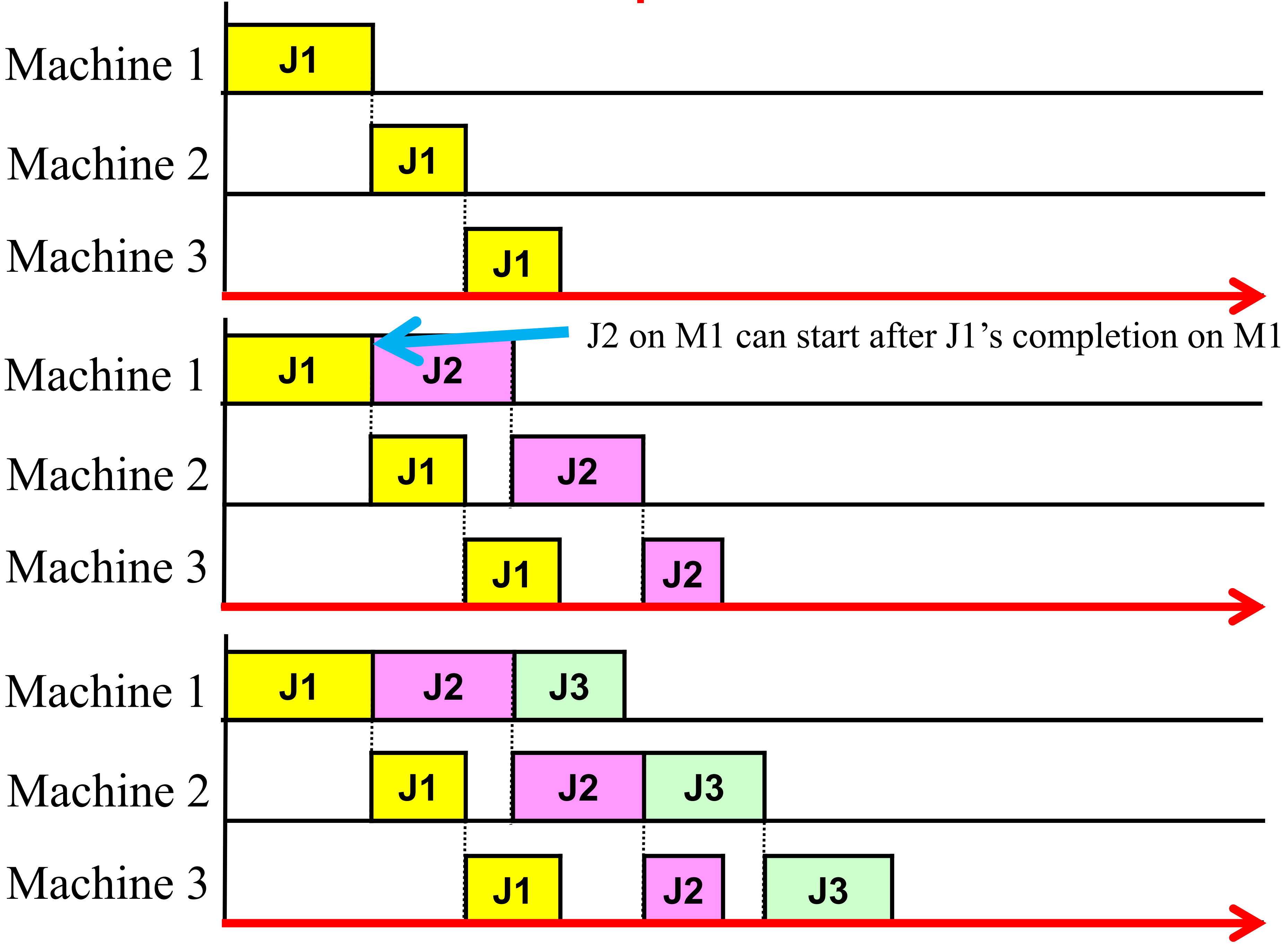


Makespan Calculation

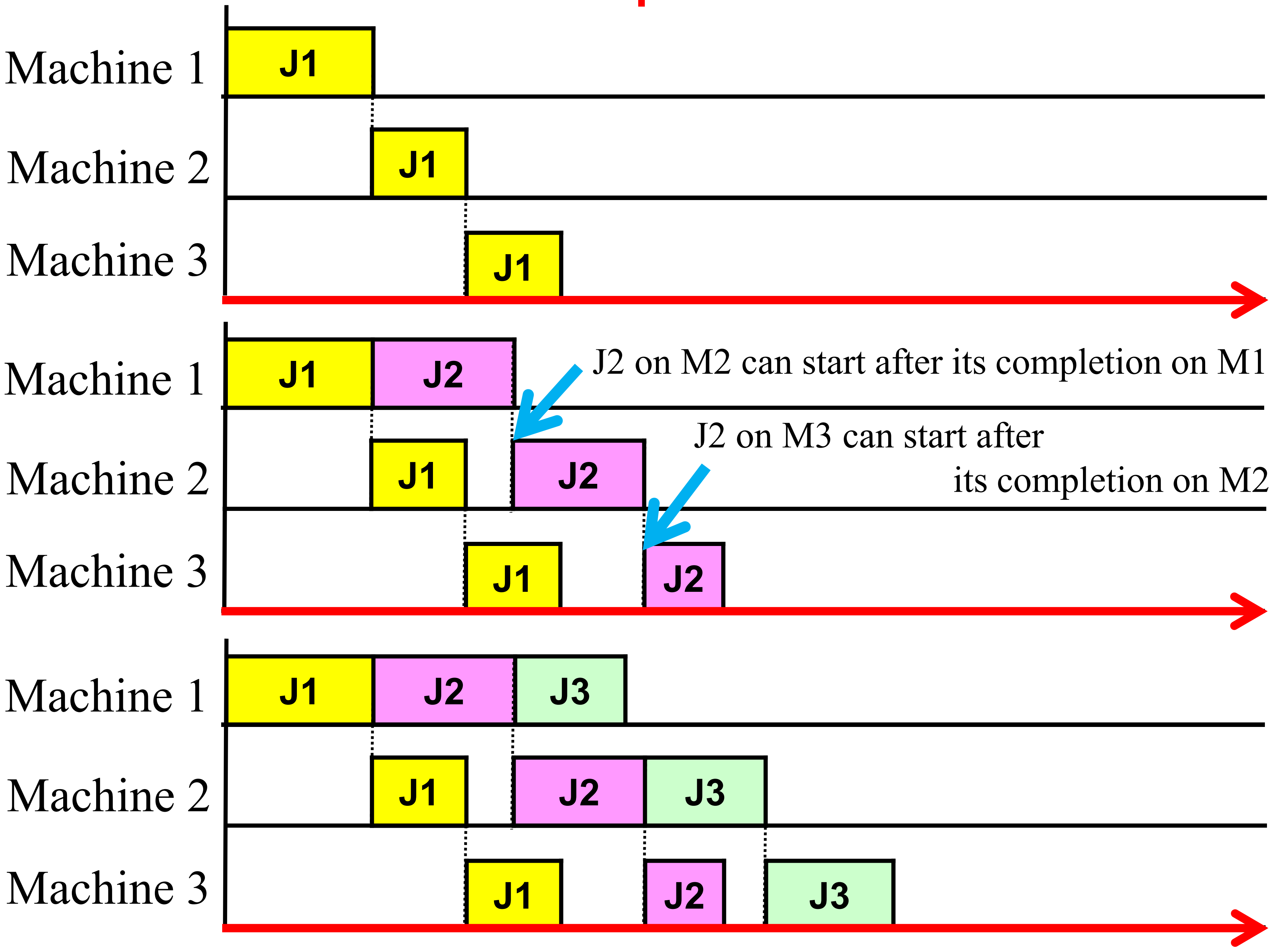


Machine 1
Machine 2
Machine 3
J1
J1 on M2 can start after its completion on M1
J1 on M3 can start after its completion on M2
J2
J3

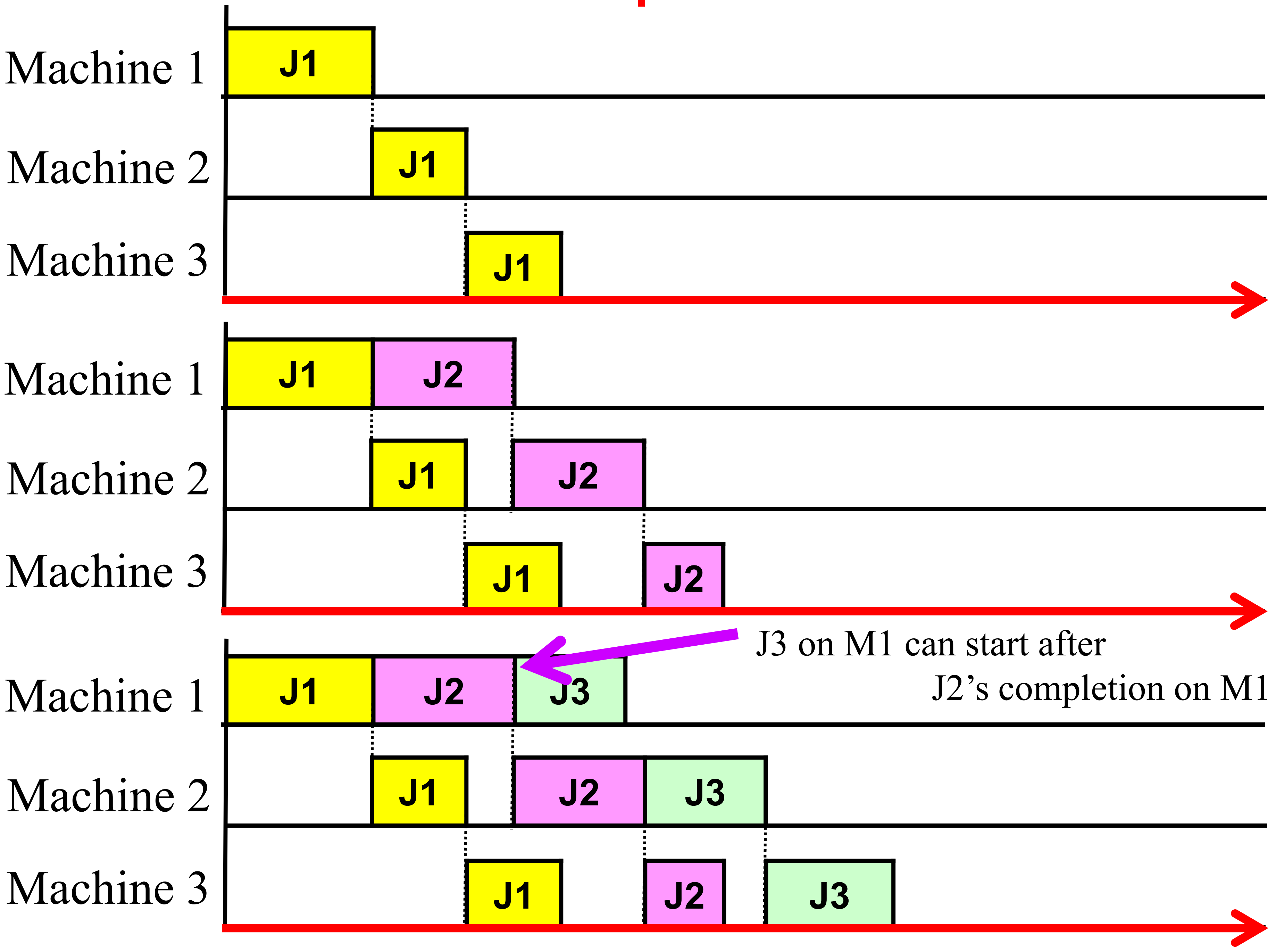
Makespan Calculation



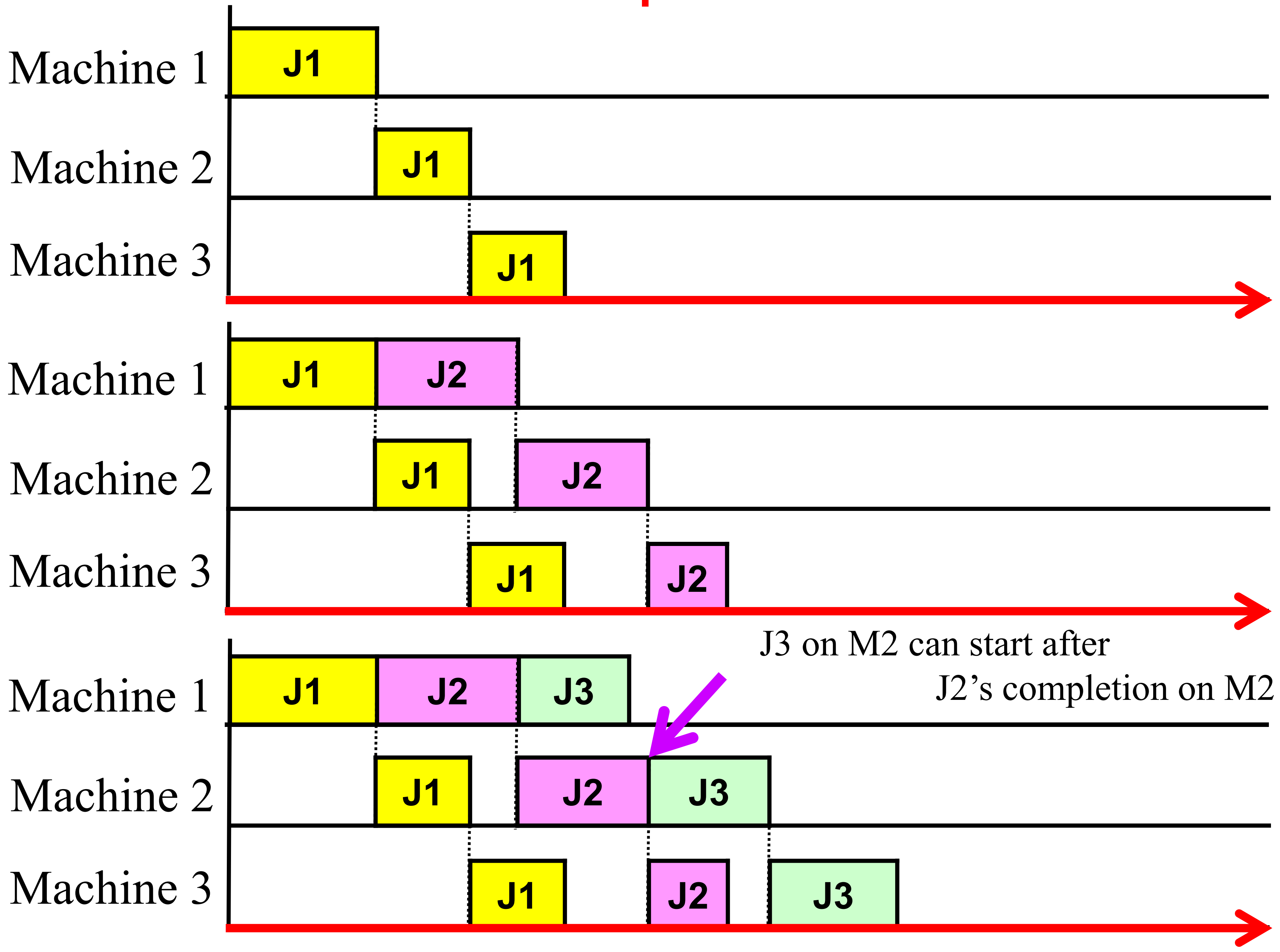
Makespan Calculation



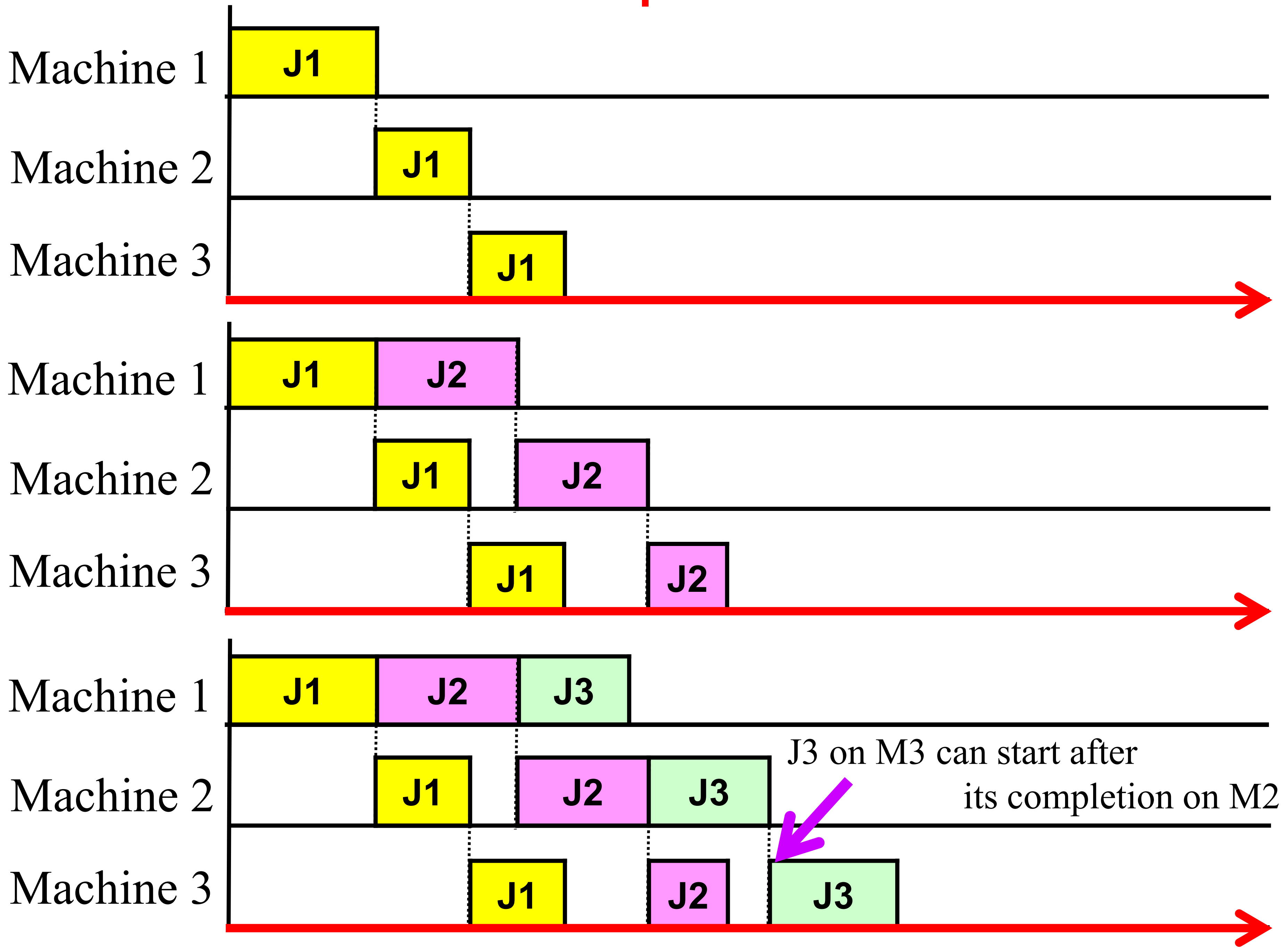
Makespan Calculation



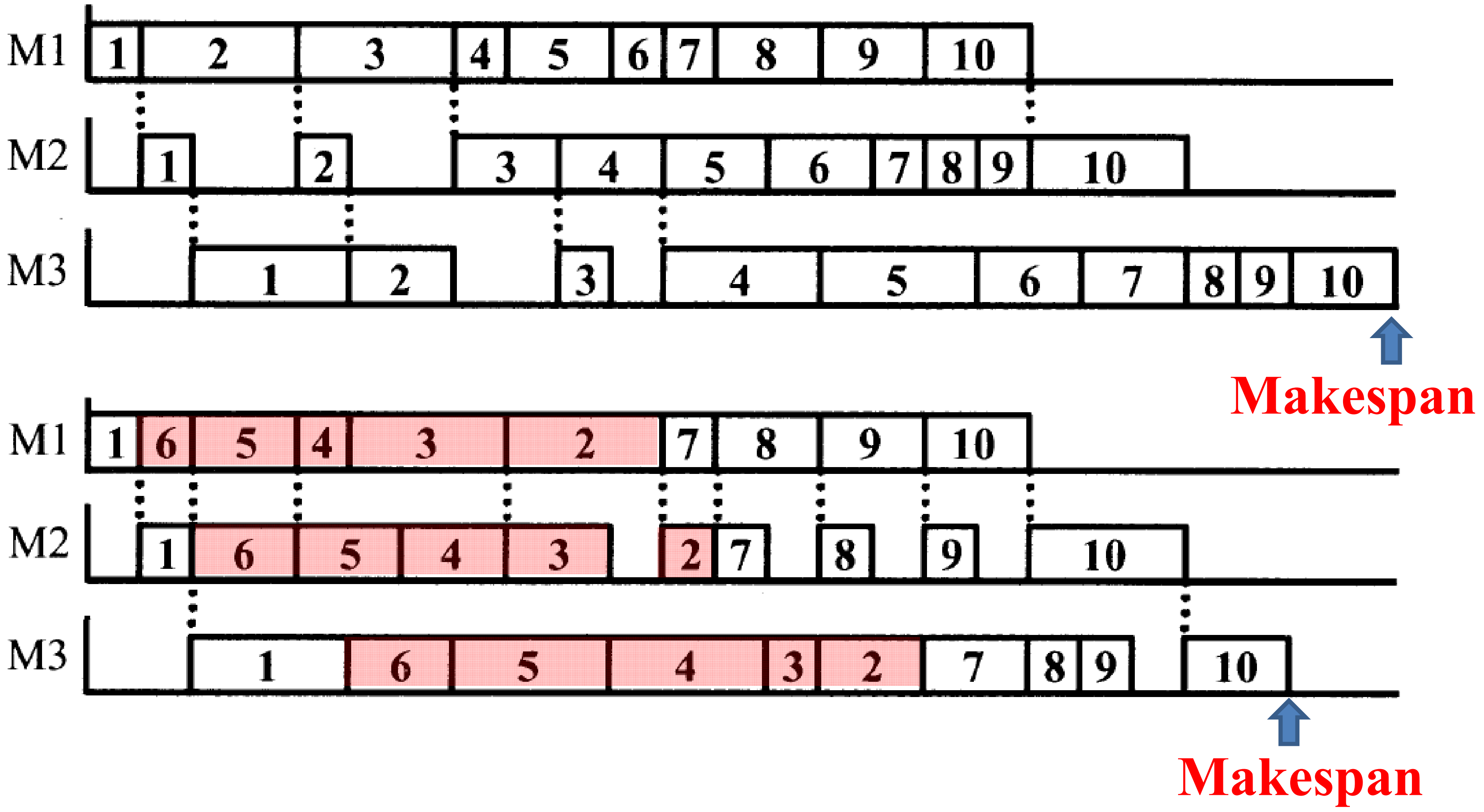
Makespan Calculation



Makespan Calculation

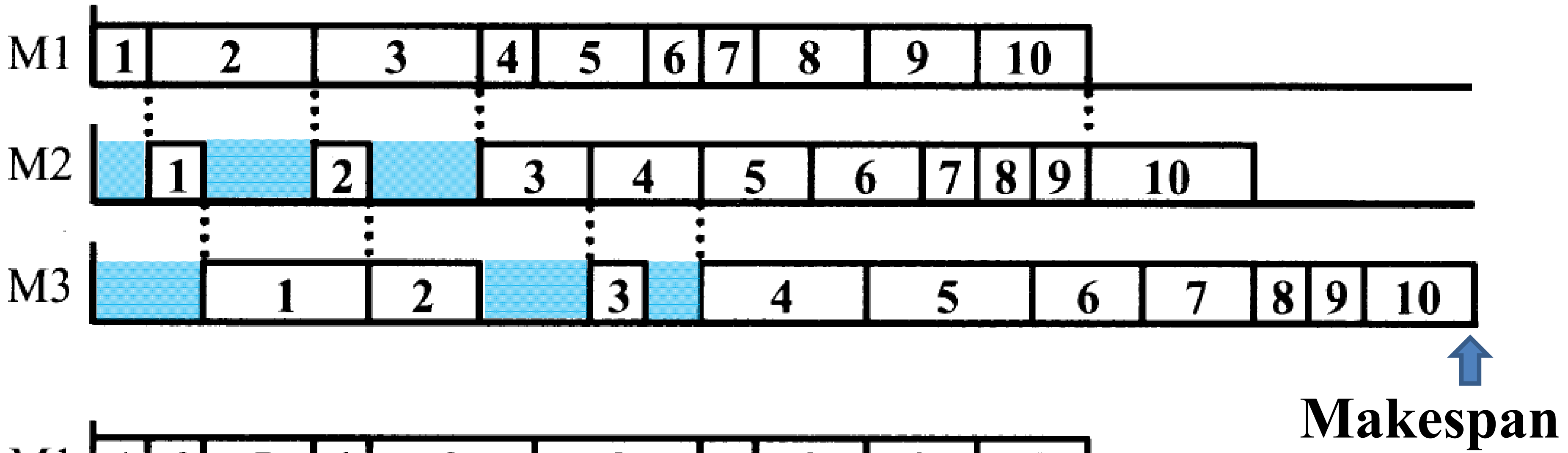


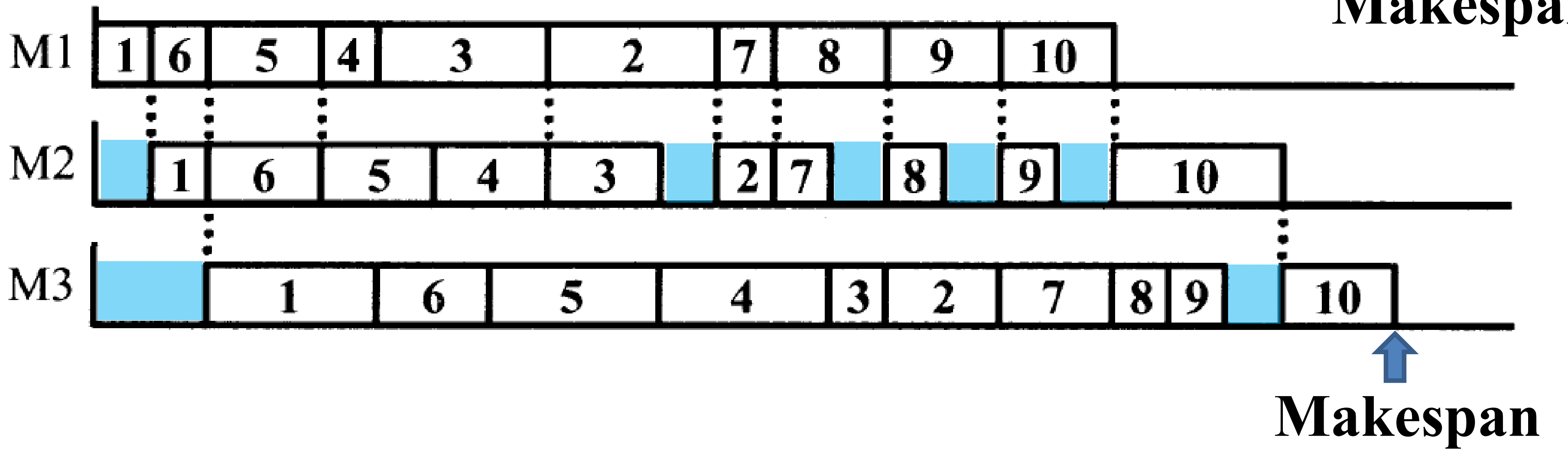
Dependency of the makespan on the processing order of jobs



Point of Scheduling

To decrease the waiting time (idle time)

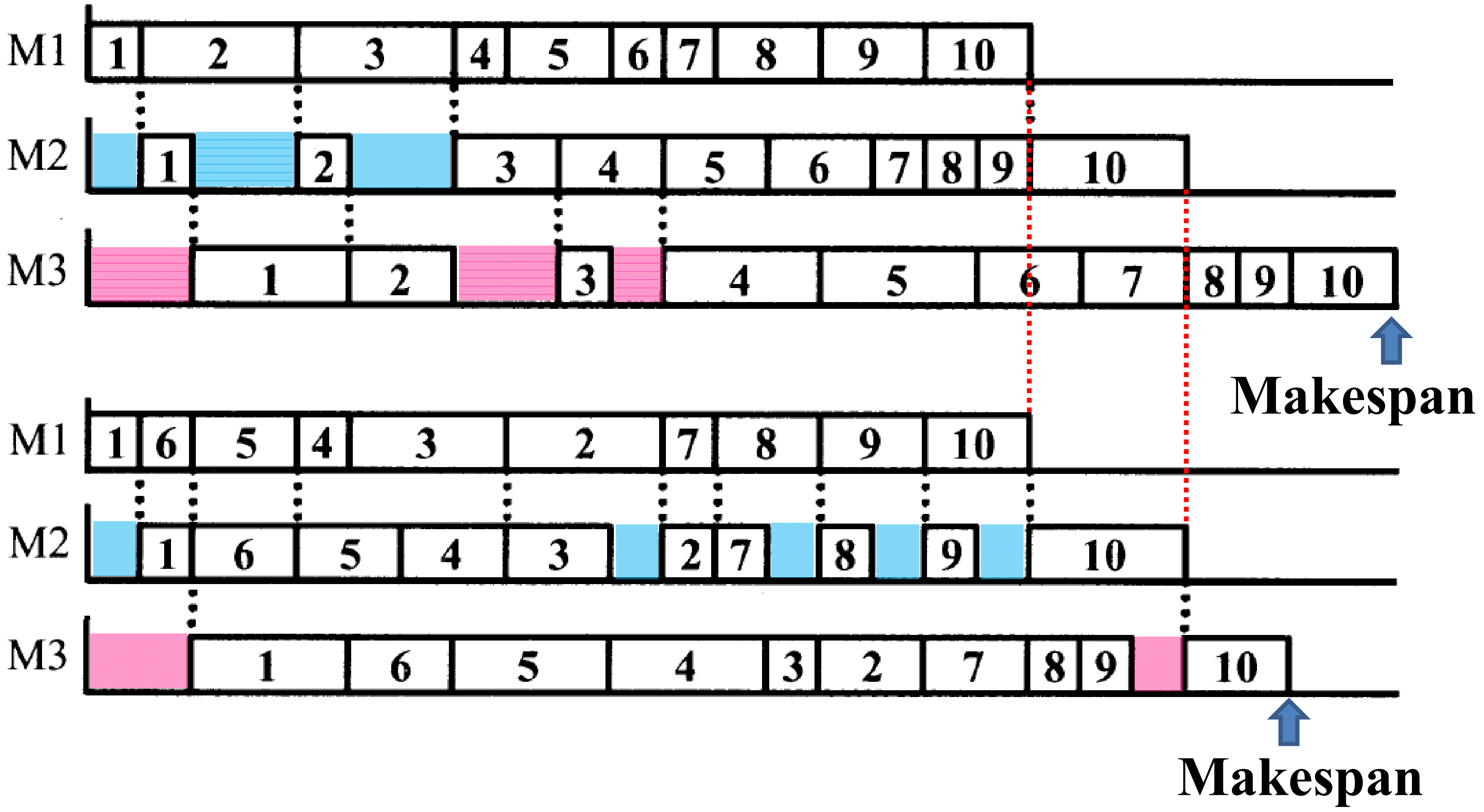




Point of Scheduling

To decrease the waiting time (idle time)

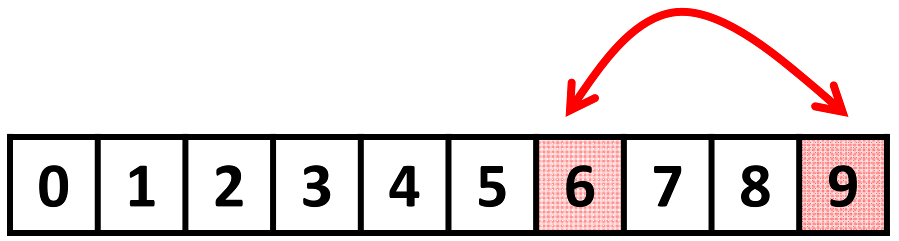
To minimize the completion time of the last job at M3



Neighborhood Structures

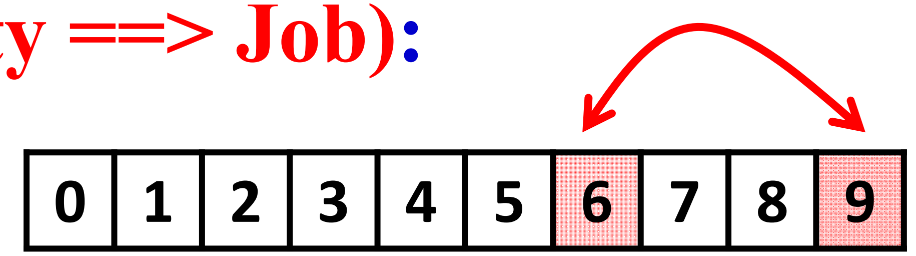
TPS (City):

- Adjacent two-city change
- Arbitrary two-city change
- Insertion (Shift)
- Inversion (Two-edge change)
- Arbitrary three-city change



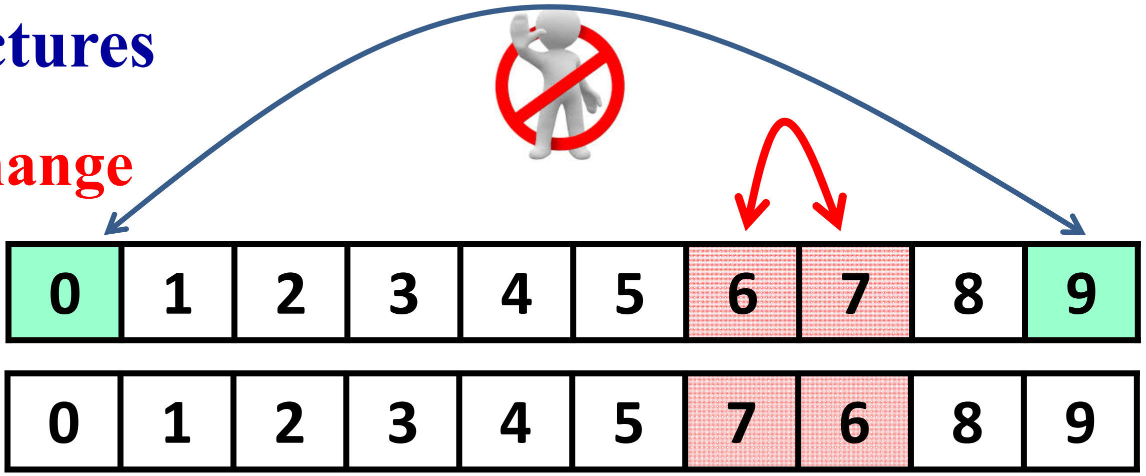
Flowshop Scheduling (City ==> Job):

- Adjacent two-job change
- Arbitrary two-job change
- Insertion (Shift)
- Inversion
- Arbitrary three-job change

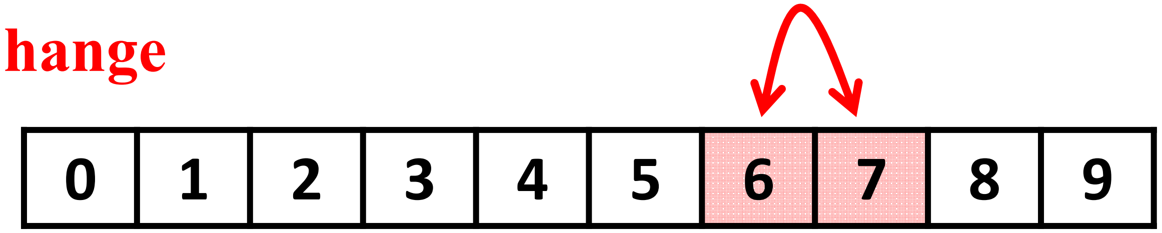


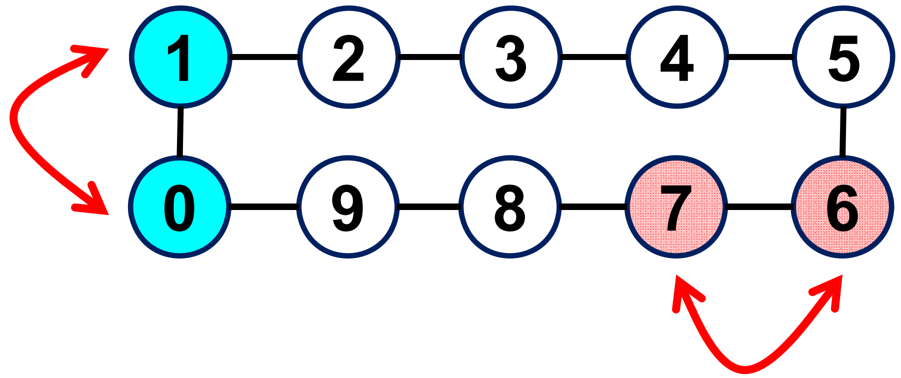
Neighborhood Structures

Adjacent Two-Job Change



Adjacent Two-City Change





Total number of different solutions

An n-job flowshop scheduling problem has $\boxed{n!}$ different solutions.

$$n! = n(n-1)(n-2)\ \dots 1$$

n!: n factorial is equal to the product of all positive integers from 1 to n.

Question: Yes or No.

We assume that we have an algorithm which can find the optimal solution by examining only 0.0000001% of all solutions, i.e., 1/(one billion) of all solutions. Is this a good algorithm ?

Total number of different solutions

An n-job flowshop scheduling problem has $\boxed{n!}$ different solutions.

$$n! = n(n-1)(n-2)\ ...1$$

n!: n factorial is equal to the product of all positive integers from 1 to n.

Question: Yes

We assume that we have an algorithm which can find the optimal solution by examining only 0.0000001% of all solutions, i.e., 1/(one billion) of all solutions. Is this a good algorithm ?

For *n* = 14: This algorithm examines only 87 solutions among 87,178,291,200 solutions. Very good algorithm.

Total number of different solutions

An n-job flowshop scheduling problem has $\boxed{n!}$ different solutions.

$$n! = n(n-1)(n-2)\ ...1$$

n!: n factorial is equal to the product of all positive integers from 1 to n.

Question: Yes and No

We assume that we have an algorithm which can find the optimal solution by examining only 0.0000001% of all solutions, i.e., 1/(one billion) of all solutions. Is this a good algorithm ?

For *n* = 14: This algorithm examines only 87 solutions among 87,178,291,200 solutions. Very good algorithm.

For *n* = 100: This algorithm needs to examine an unrealistically large number of solutions (more than all solutions for $n = 95$)

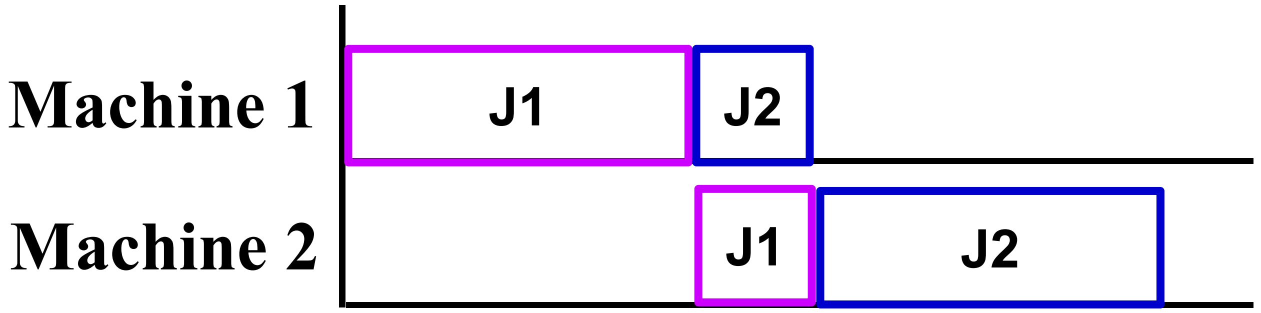
Flowshop Scheduling

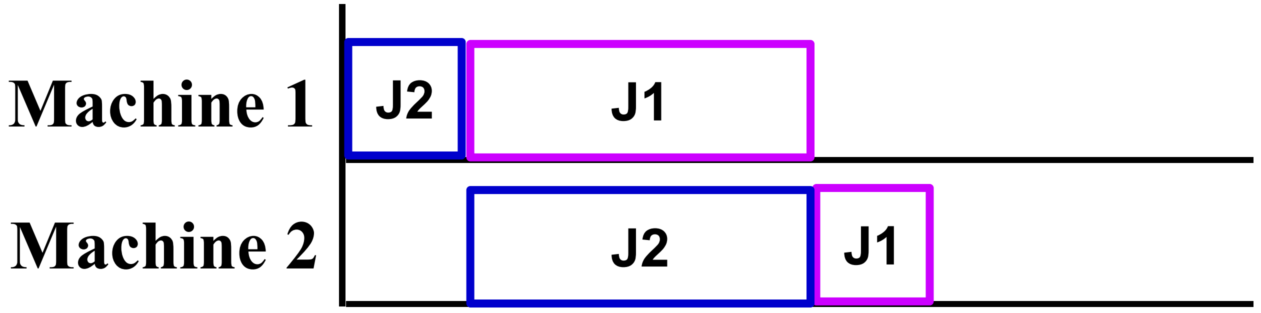
- Simple heuristics
- **Exact optimization algorithms**
- Metaheuristics (local search, iterated local search, SA, GA)

Johnson Method: 2-Machine Flow Shop Scheduling

Simple Example: Order of Two Jobs on Two Machines

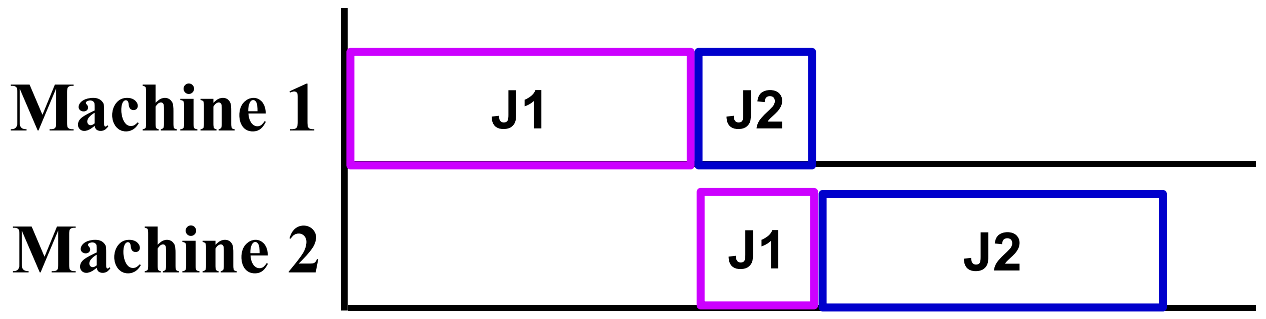
	Job 1	Job 2
Machine 1	30	10
Machine 2	10	30

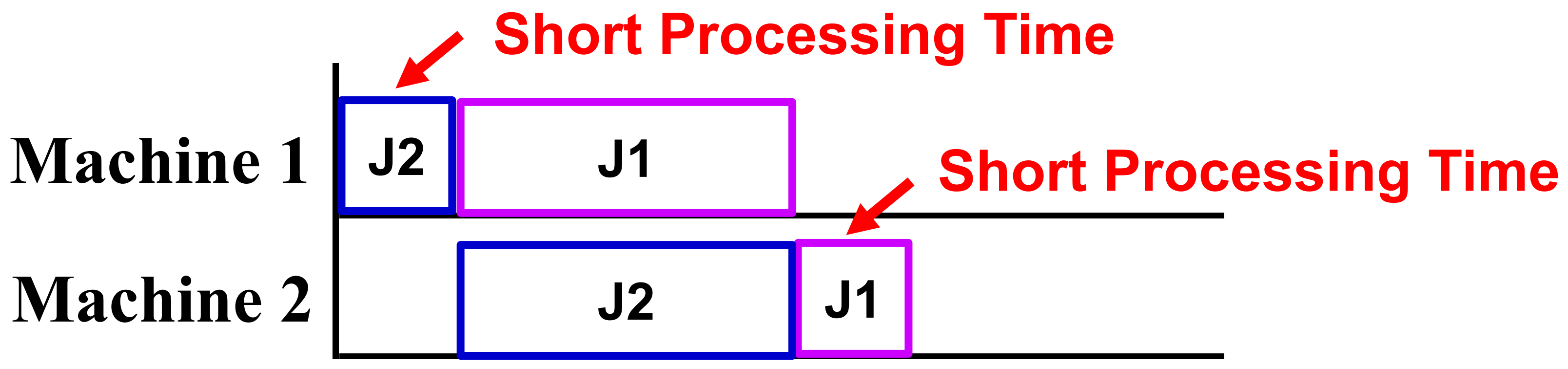




Simple Example: Order of Two Jobs on Two Machines

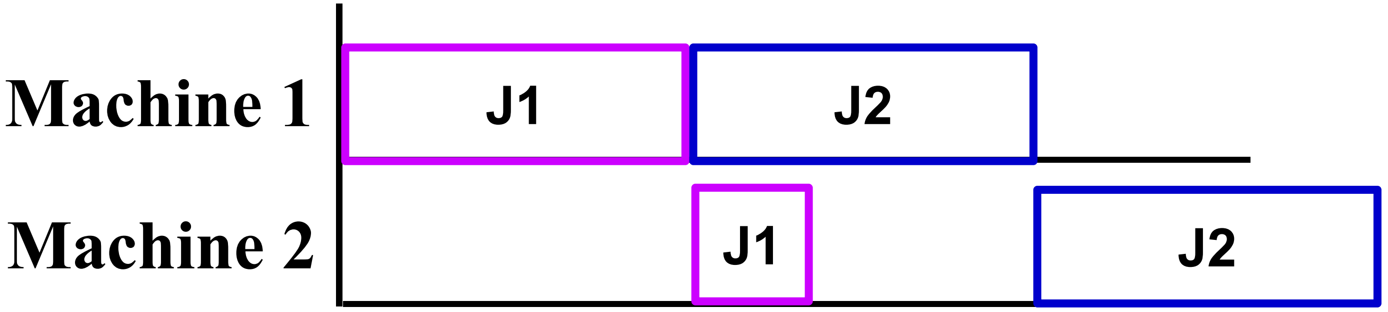
	Job 1	Job 2
Machine 1	30	10
Machine 2	10	30

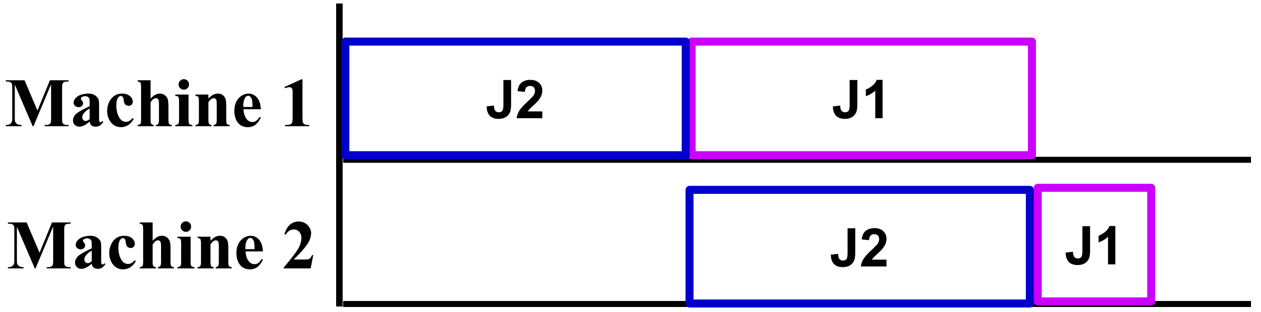




Simple Example: Order of Two Jobs on Two Machines

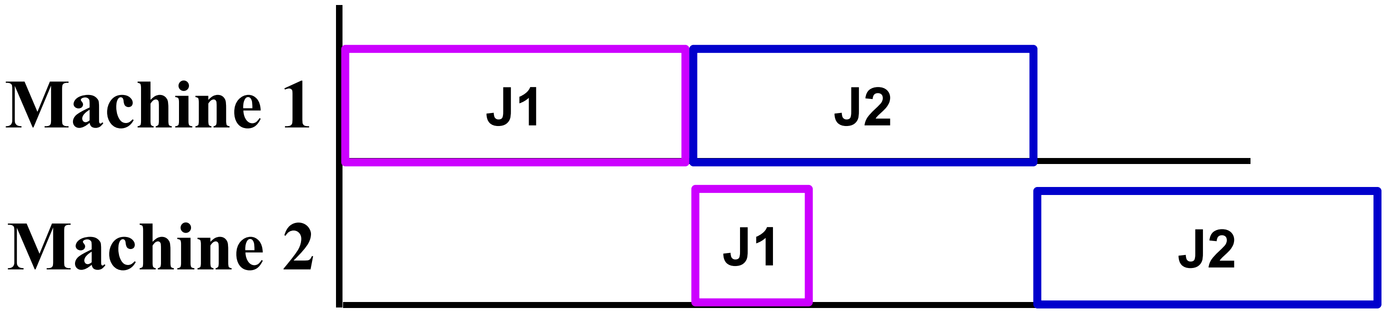
	Job 1	Job 2
Machine 1	30	30
Machine 2	10	30

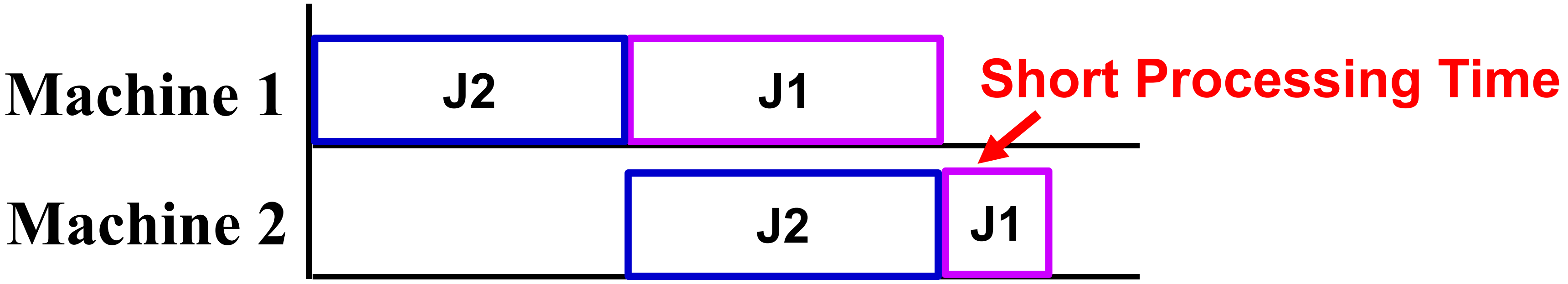




Simple Example: Order of Two Jobs on Two Machines

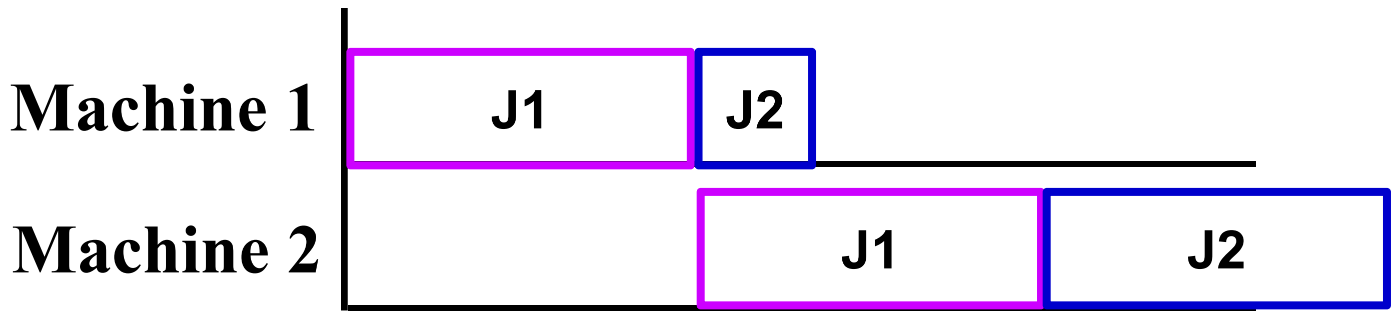
	Job 1	Job 2
Machine 1	30	30
Machine 2	10	30

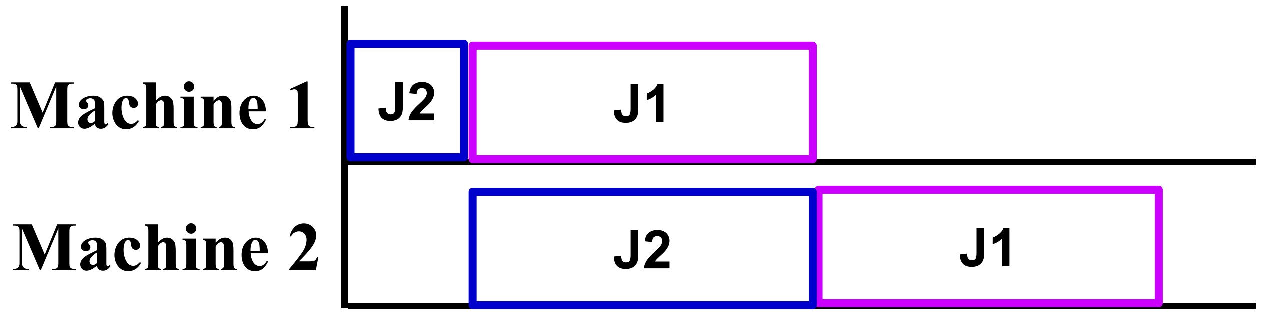




Simple Example: Order of Two Jobs on Two Machines

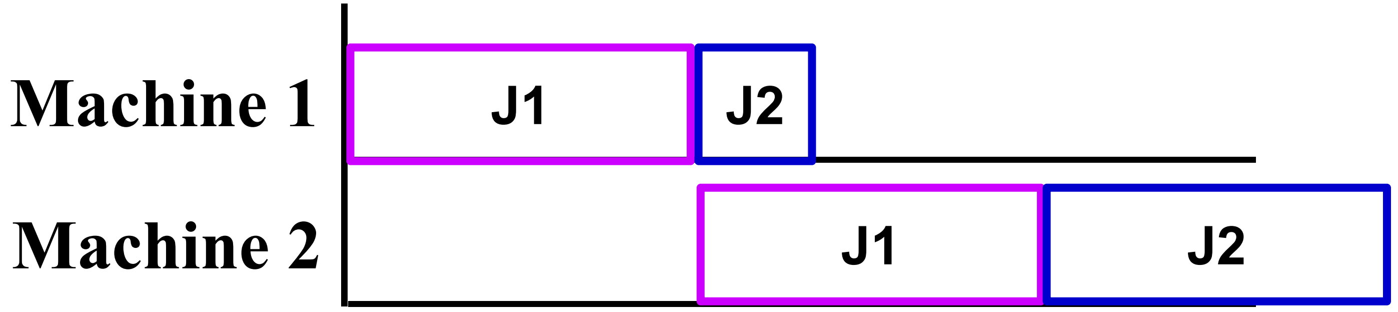
	Job 1	Job 2
Machine 1	30	10
Machine 2	30	30

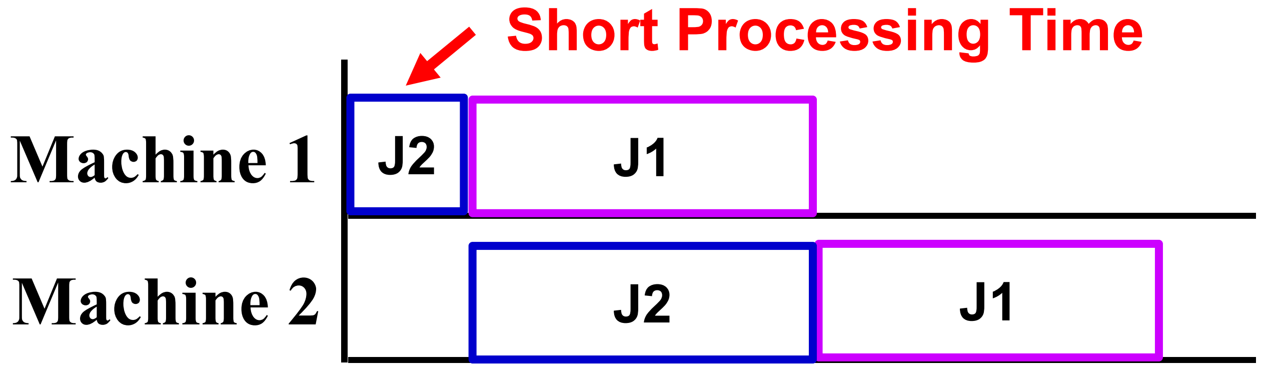




Simple Example: Order of Two Jobs on Two Machines

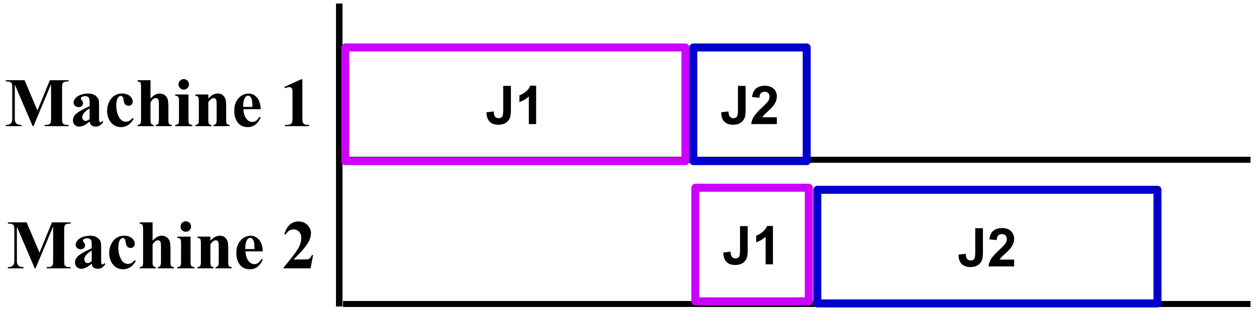
	Job 1	Job 2
Machine 1	30	10
Machine 2	30	30

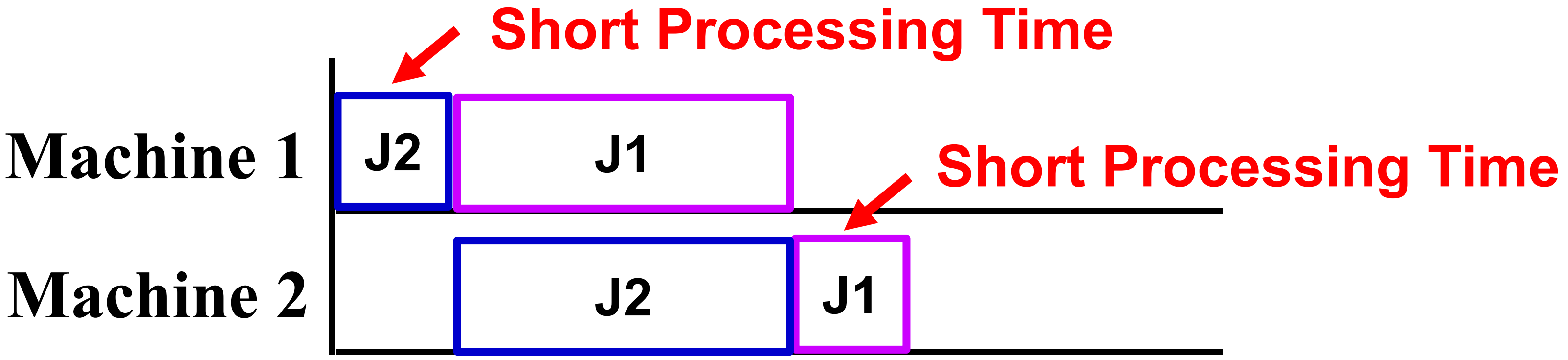




Simple Example: Order of Two Jobs on Two Machines

	Job 1	Job 2
Machine 1	30	10
Machine 2	10	30

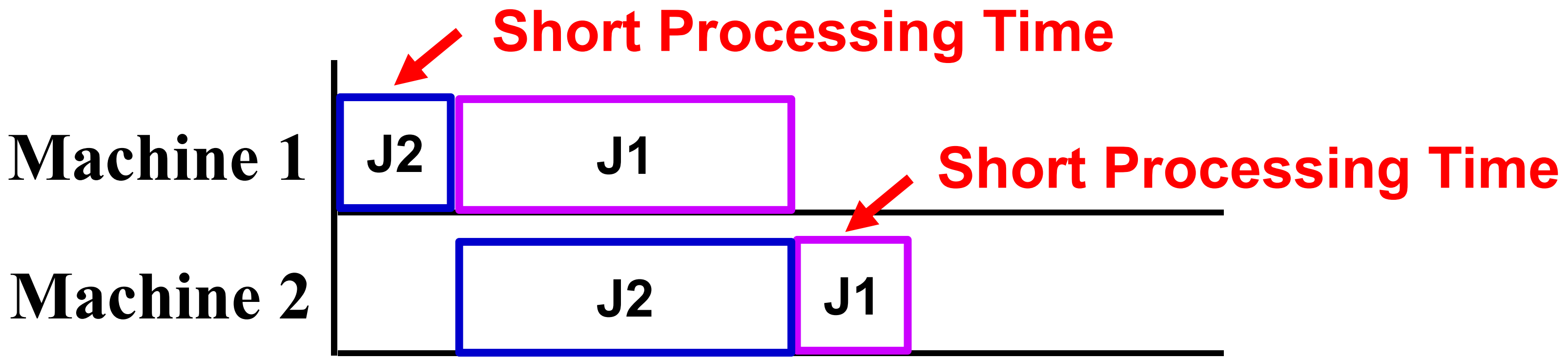




Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

(Exact Optimization Algorithm)



Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.
(Exact Optimization Algorithm)

JOHNSON, S. M., "Optimal Two- and Three-Stage Production Schedules with Setup Times Included," Naval Research Logistics Quarterly, 1954, Vol. 1, pp. 61-68.

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Send your answer

Solution							
	Job ?	Job ?	Job ?	Job ?	Job ?	Job ?	Job ?

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Solution

						Job 4

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Solution							
	Job 1						Job 4

The optimal schedule is obtained!

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Solution							
	Job 1					Job 3	Job 4

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Solution

Job 1	Job 2			Job 5	Job 3	Job 4

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
Machine 1	20	40	60	80	100	120	140
Machine 2	70	50	30	10	40	80	70

Solution							
	Job 1	Job 2	Job6	Job 7	Job 5	Job 3	Job 4

The optimal schedule is obtained!

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

General *m*-machine *n*-job problem

(e.g., 10-machine 20-job, 20-machine 100-job)

- Simple heuristics (utilization of Johnson method)
- **Exact optimization algorithms (branch and bound)**
- Metaheuristics (local search, iterated local search, SA, GA)

Johnson Method: 2-Machine Flow Shop Scheduling

Find the shortest processing time in the remaining jobs. If it is on the first machine, assign the job to the first position among the remaining positions. If it is on the second machine, assign the job to the last position among the remaining positions.

General *m*-machine *n*-job problem

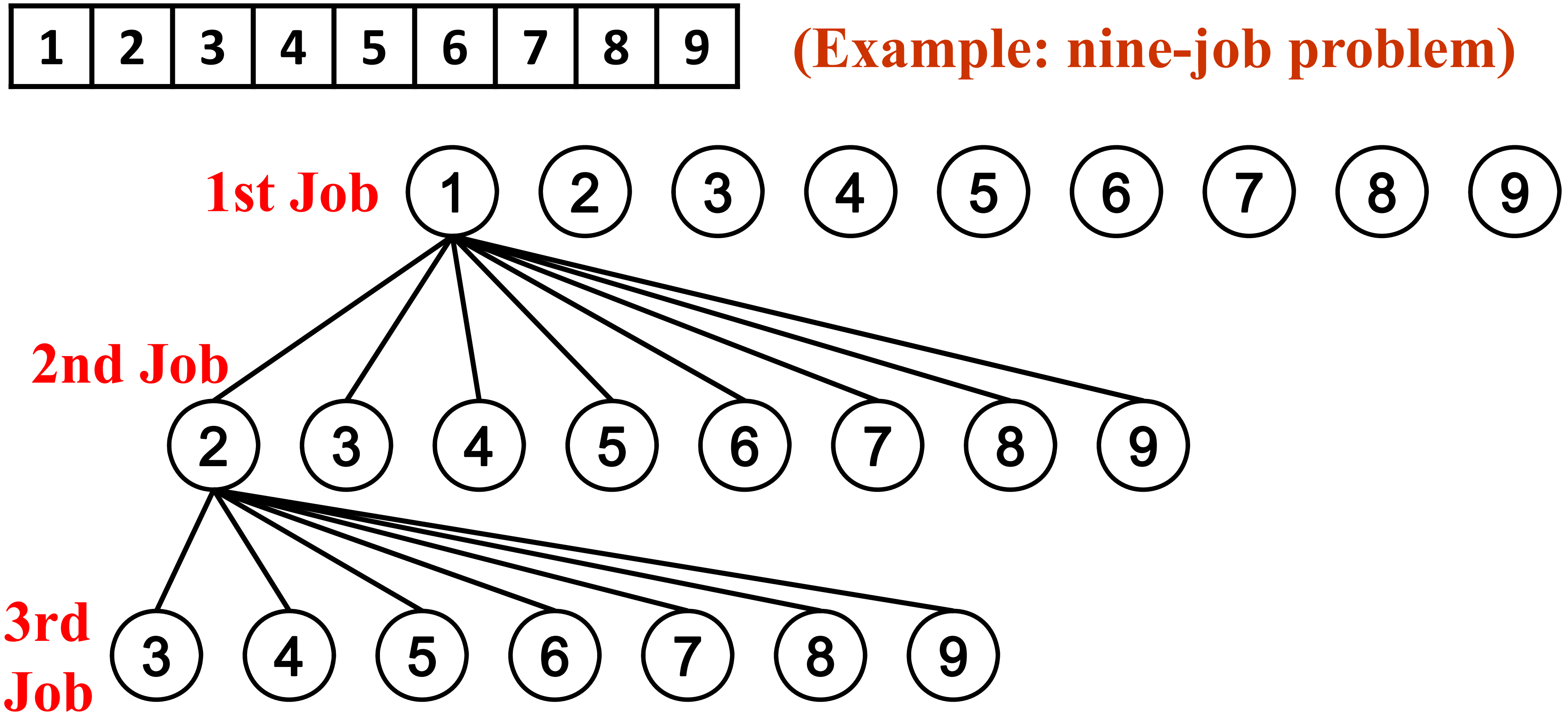
(e.g., 10-machine 20-job, 20-machine 100-job)

- Simple heuristics (utilization of Johnson method)
- **Exact optimization algorithms (branch and bound)**
- Metaheuristics (local search, iterated local search, SA, GA)

Exact Optimization Algorithms:

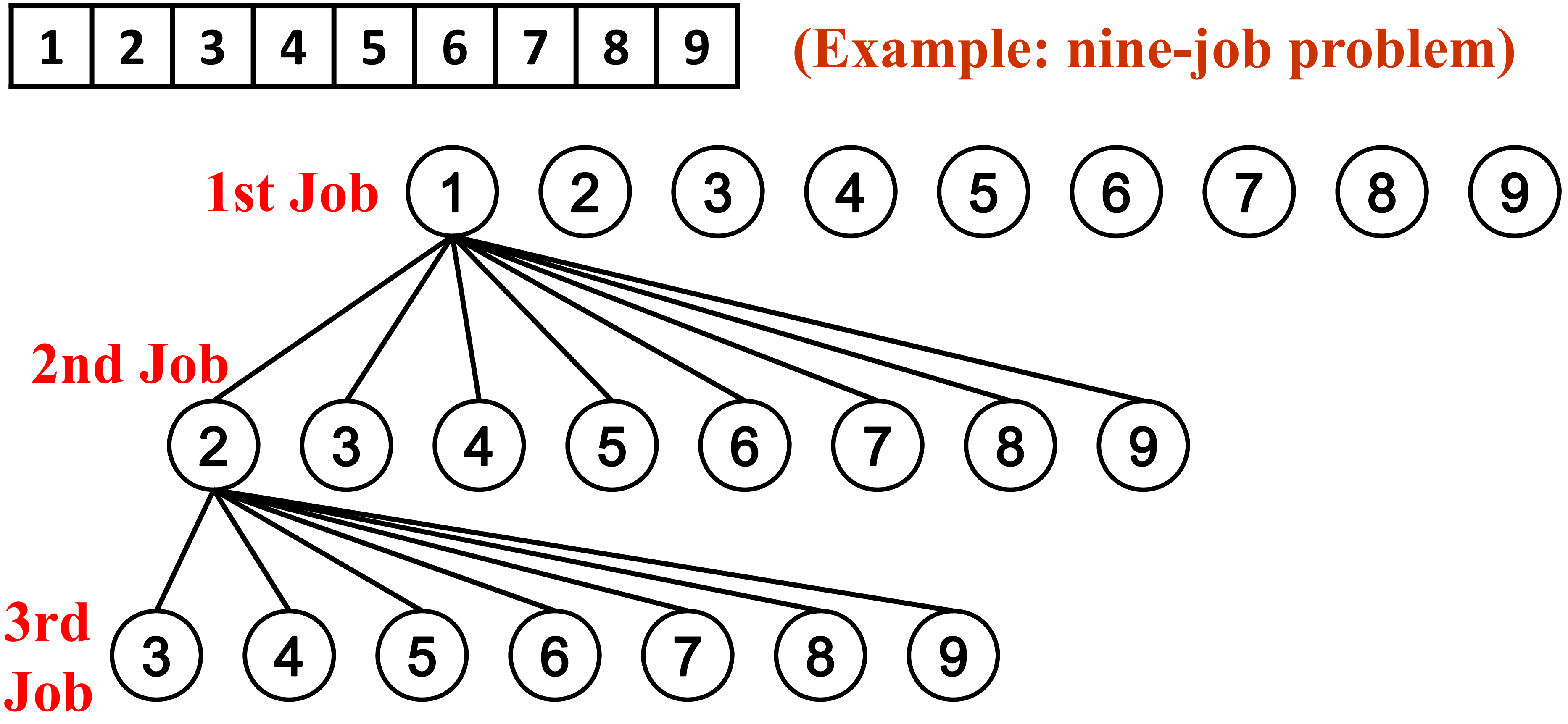
Basically we need to examine all solutions (we need to confirm that the obtained solution is the best solution in all solutions).

Branch and Bound



Basic Idea: Create a tree to generate all possible solutions. Then we can always choose the optimal solution.

Branch and Bound



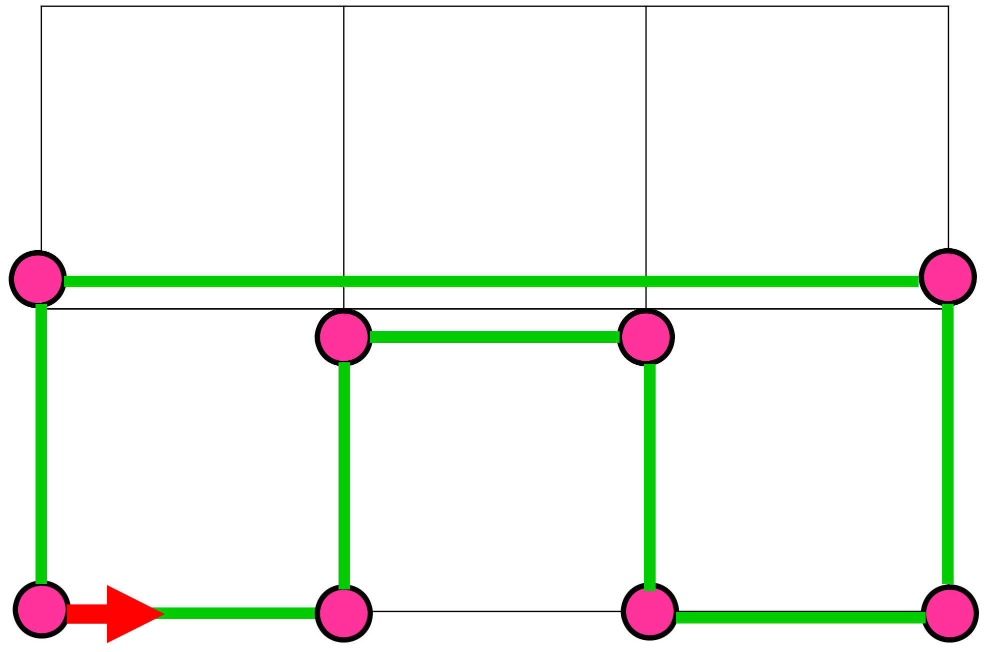
Basic Idea: Create a tree to generate all possible solutions. Then we can always choose the optimal solution.

Important Idea: Avoid creating nodes from which the optimal solution cannot be generated. Then we can find the optimal solution without examining all possible solutions.

Example: 8-City TSP

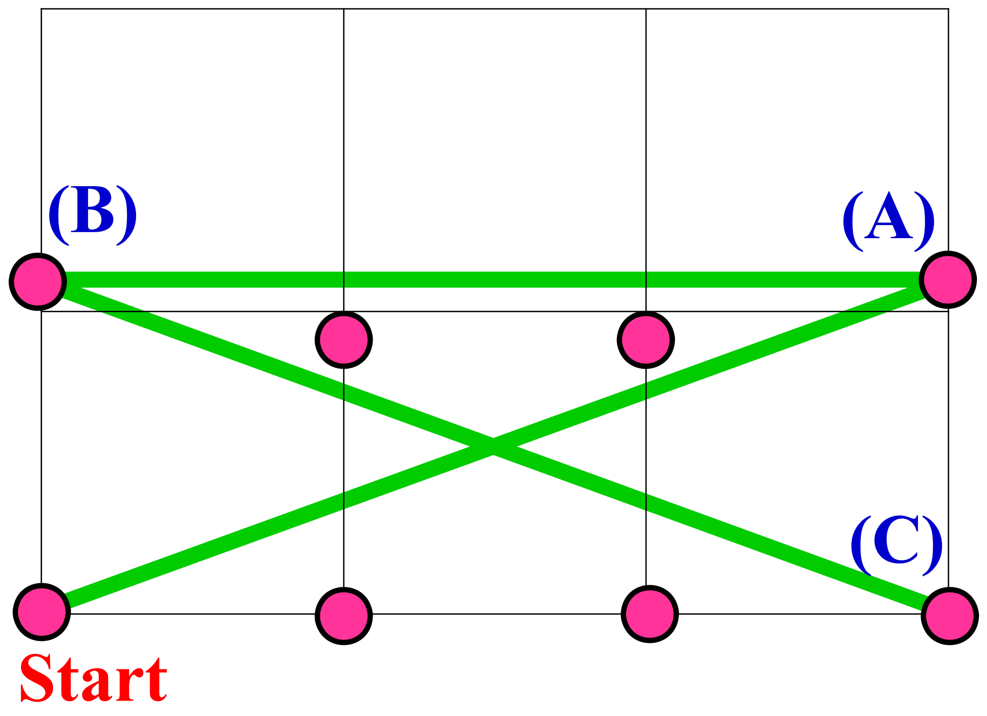
Greedy solution

Tour length = 10



Termination node:

Current Tour length > 10



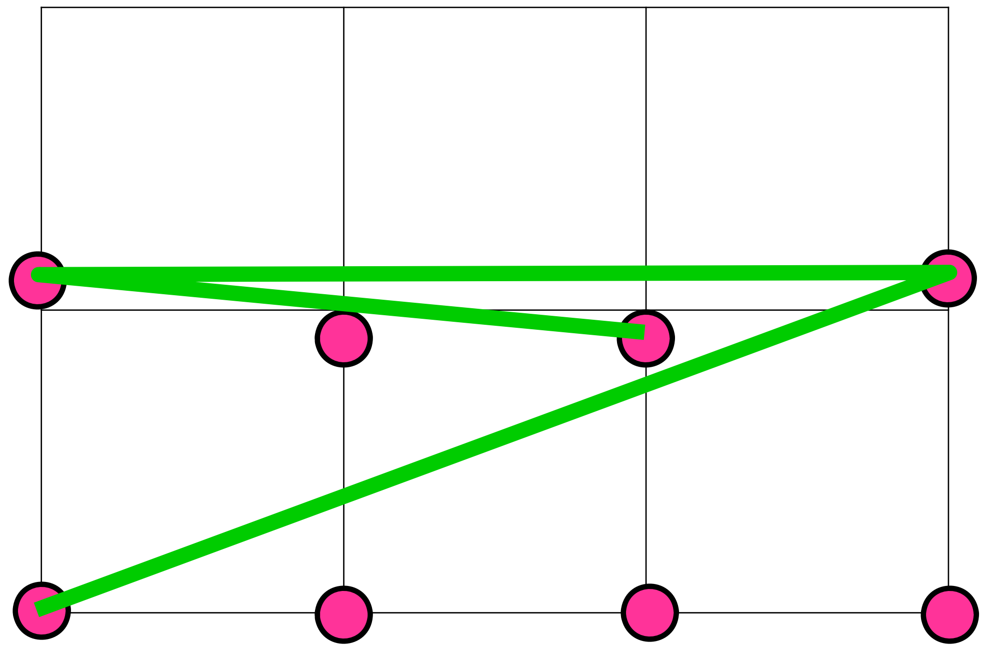
After visiting A, B, C, it is clear that the tour length will be larger than 10 independent of the order of the other four cities (since we need to go back to the start city after visiting the other cities, which is longer than the direct distance from C to the start city)

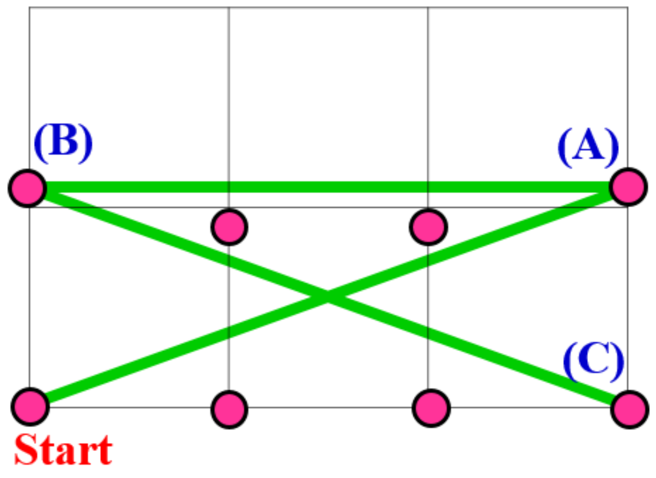
Other termination nodes

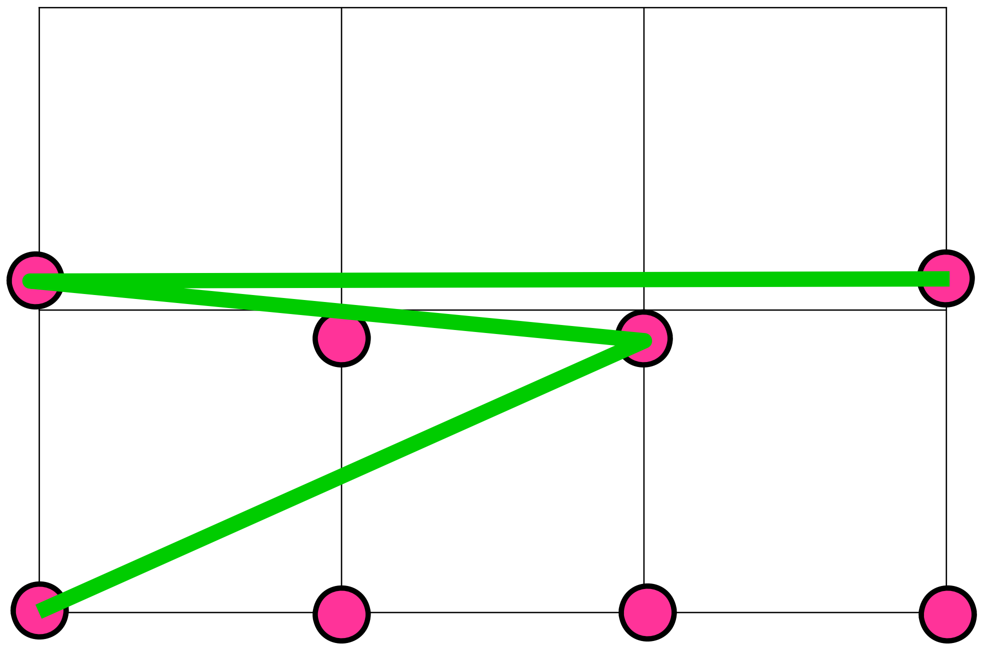
Current Tour length > 10

Many nodes can be terminated.

The tree size becomes very small.
(very efficient search)





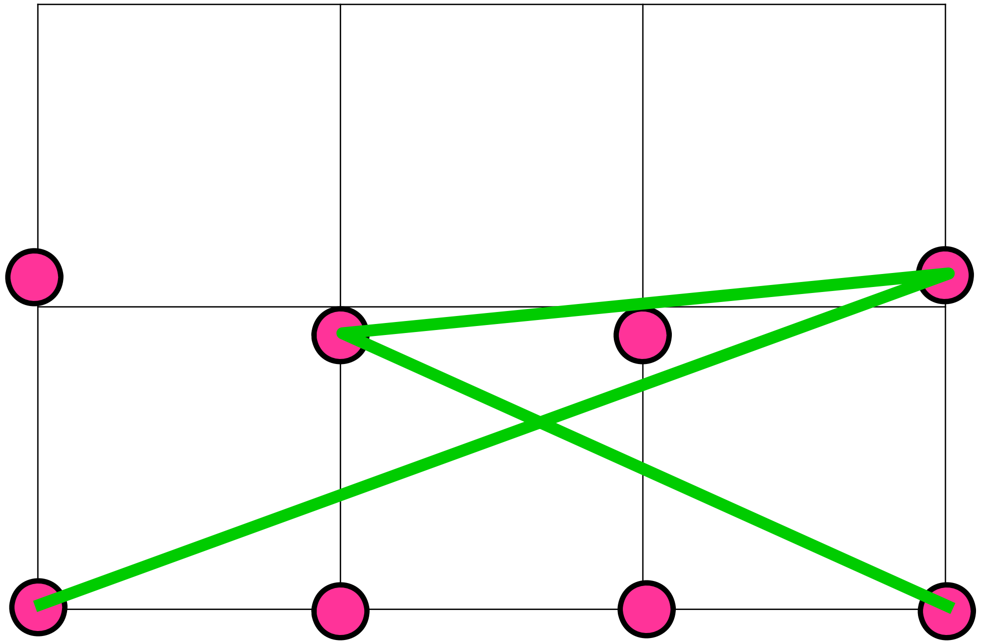


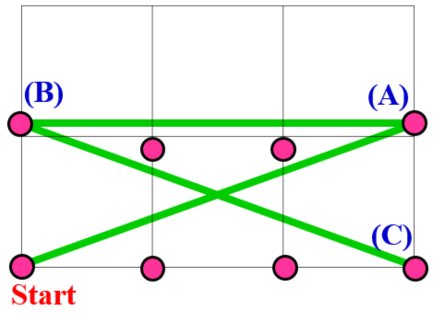
Other termination nodes

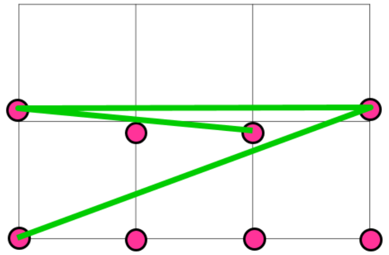
Current Tour length > 10

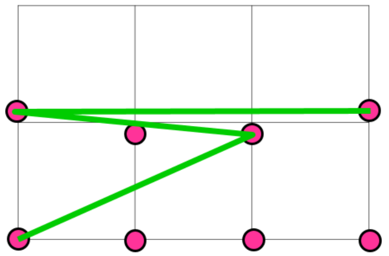
Many nodes can be terminated.

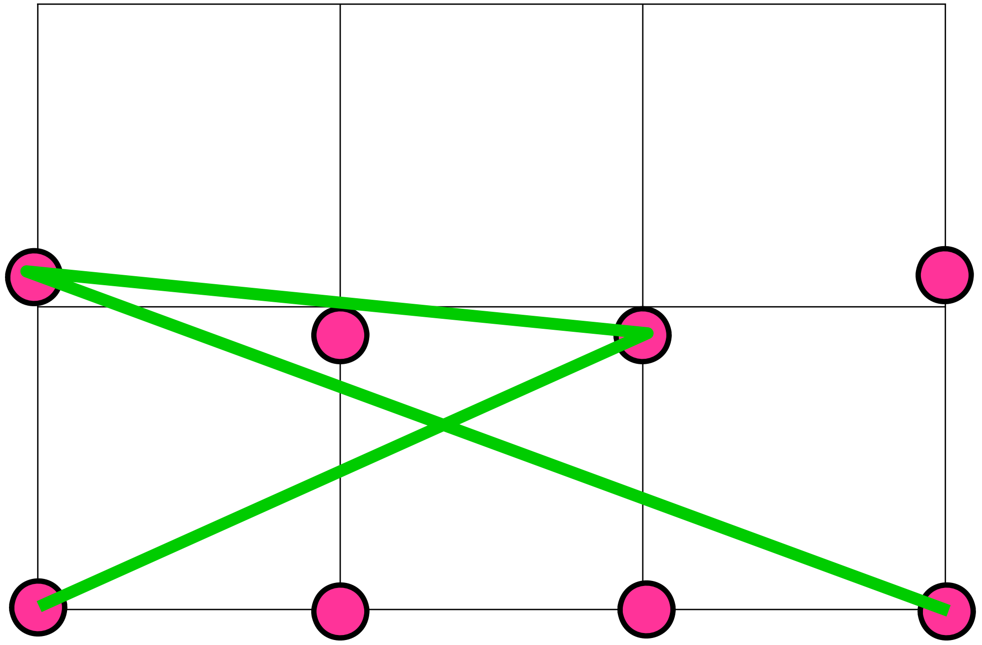
The tree size becomes very small.
(very efficient search)









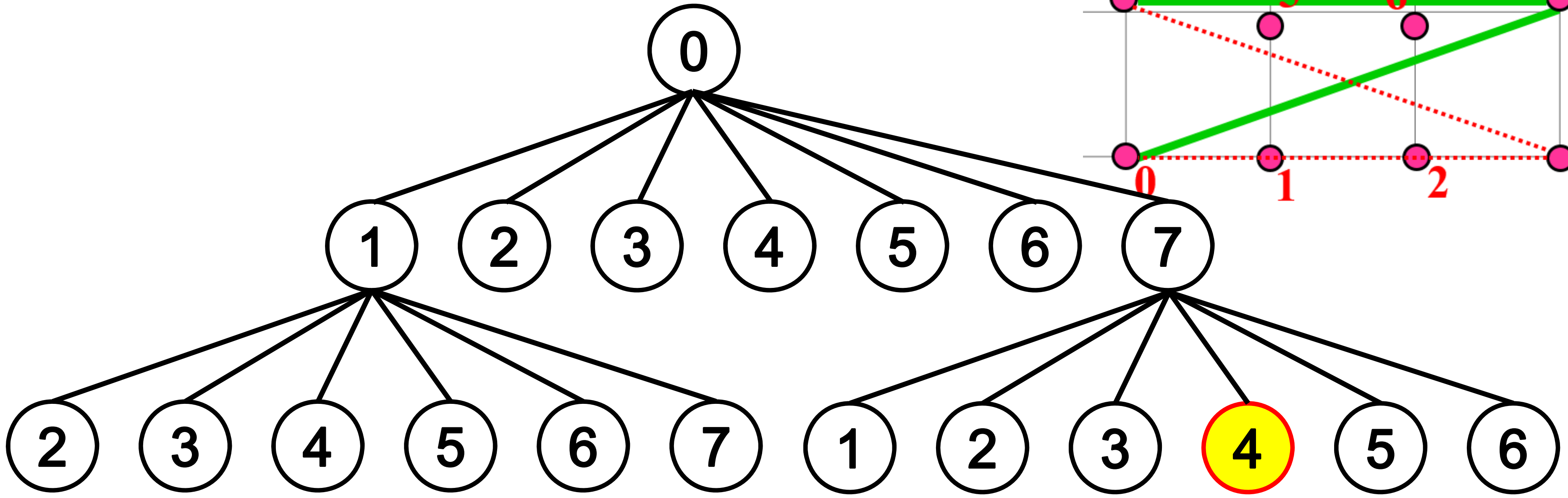


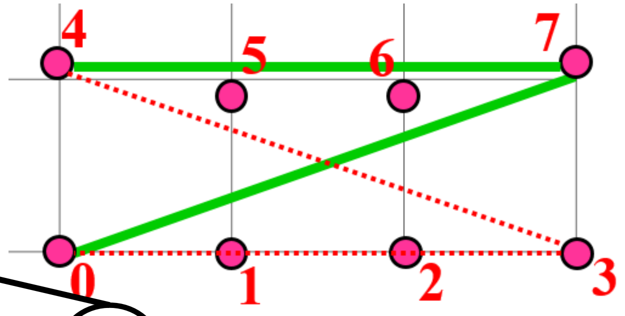
Implementation Issues.

(1) How to obtain the initial upper bound (i.e., how to obtain an initial approximate solution). Greedy solution for TSP

Node termination condition:

Better solutions than the current best solution cannot be generated from this node. (Greedy Solution Tour Length = 10)



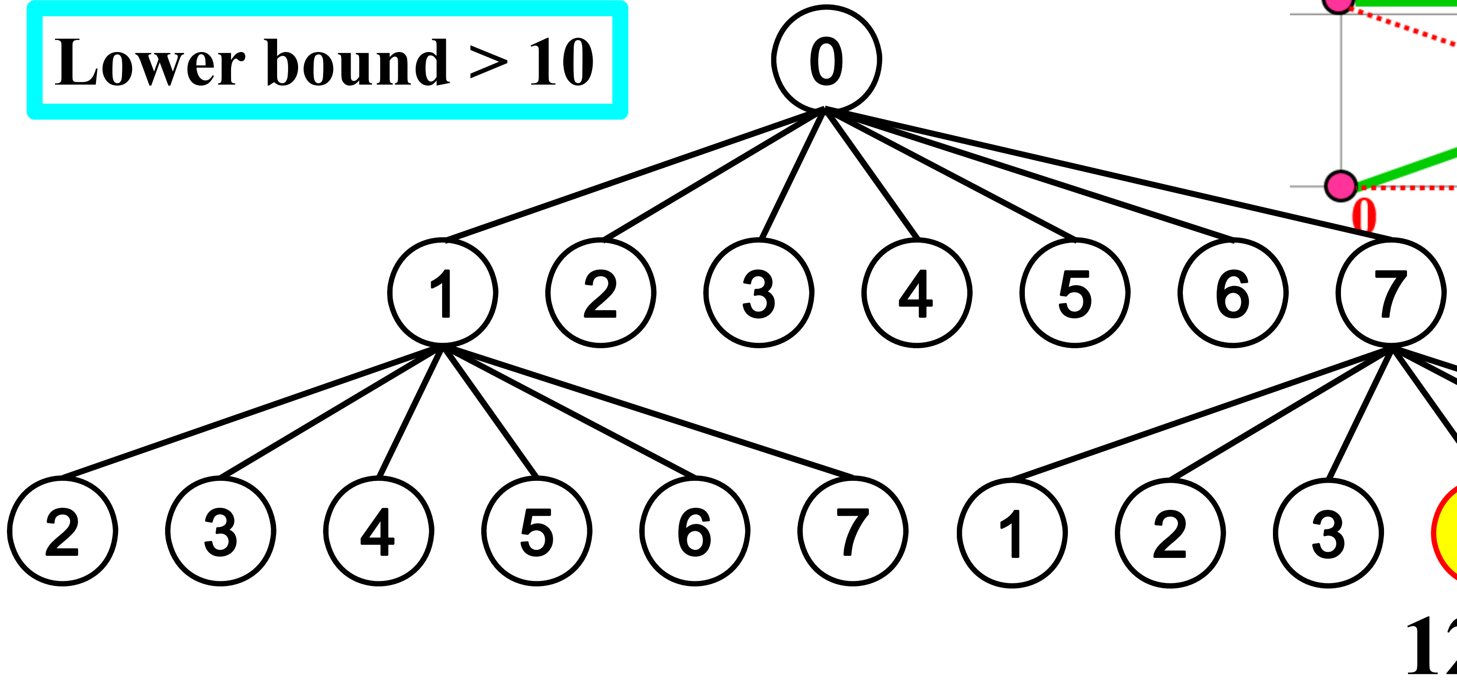


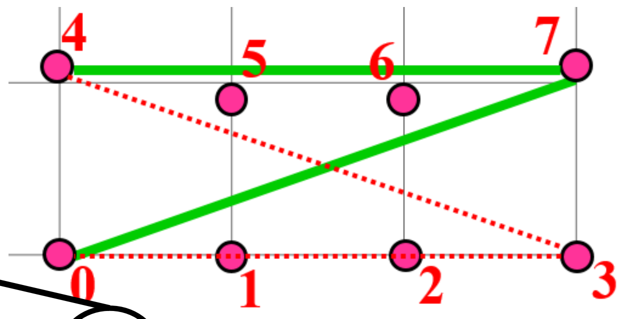
Implementation Issues.

(1) How to obtain the initial upper bound (i.e., how to obtain an initial approximate solution). Greedy solution for TSP

(2) How to calculate the lower bound of each node (i.e., the lower bound about the best possible objective value among all children from that node). Better solutions than the lower bound cannot be generated from that node (for minimization problems).

Lower bound > 10





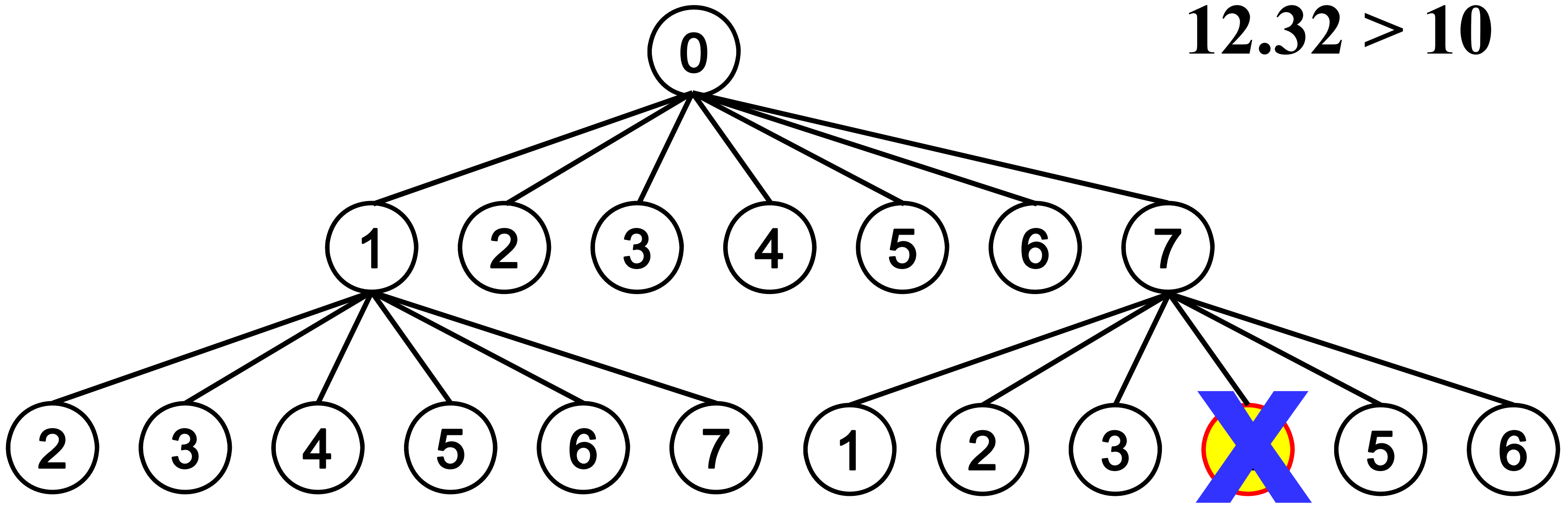
Implementation Issues.

(1) How to obtain the initial upper bound (i.e., how to obtain an initial approximate solution). Greedy solution for TSP

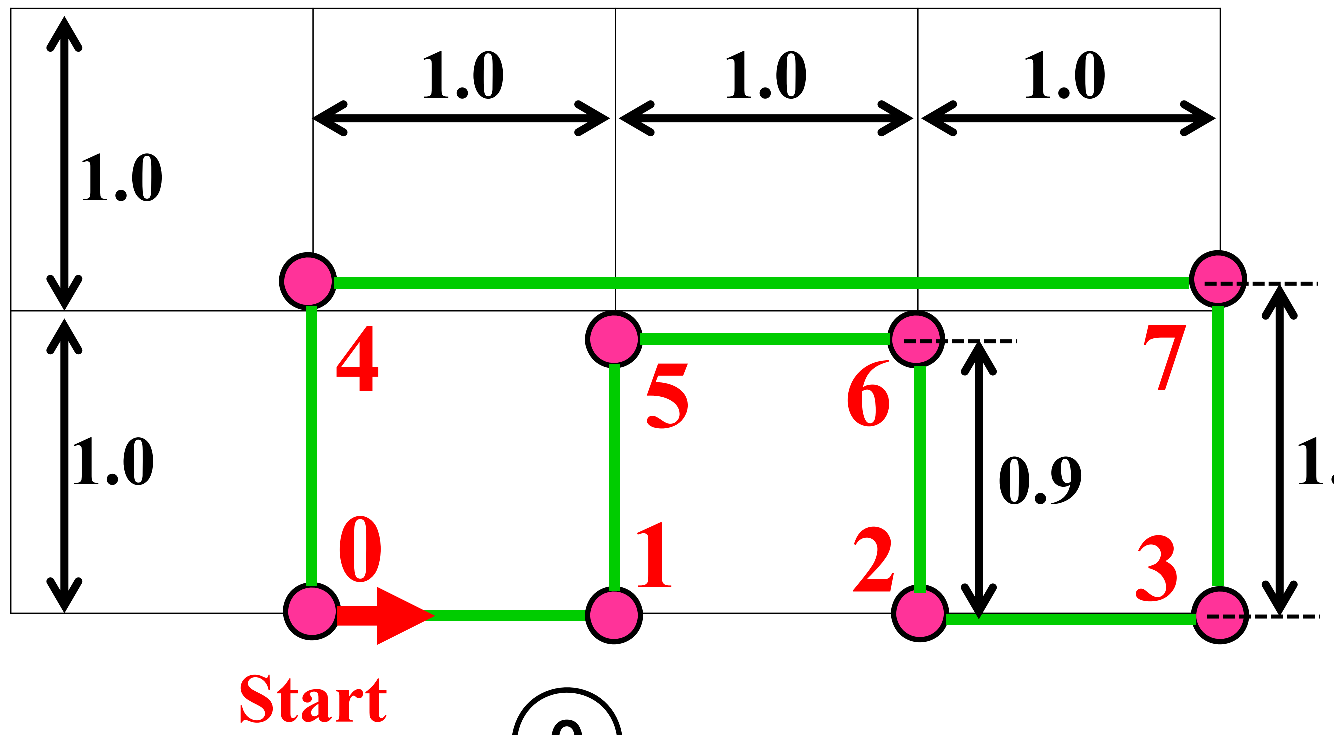
(2) How to calculate the lower bound of each node (i.e., the lower bound about the best possible objective value among all children from that node). Better solutions than the lower bound cannot be generated from that node (for minimization problems).

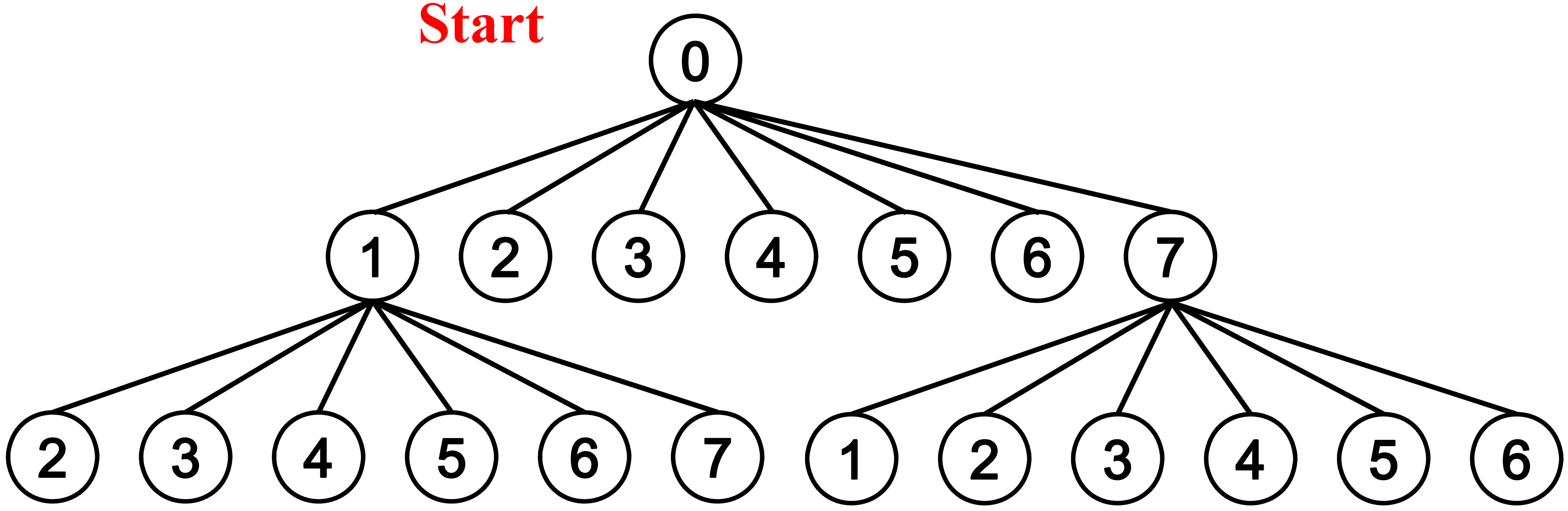
If "the lower bound > the current best", terminate the node.

12.32 > 10

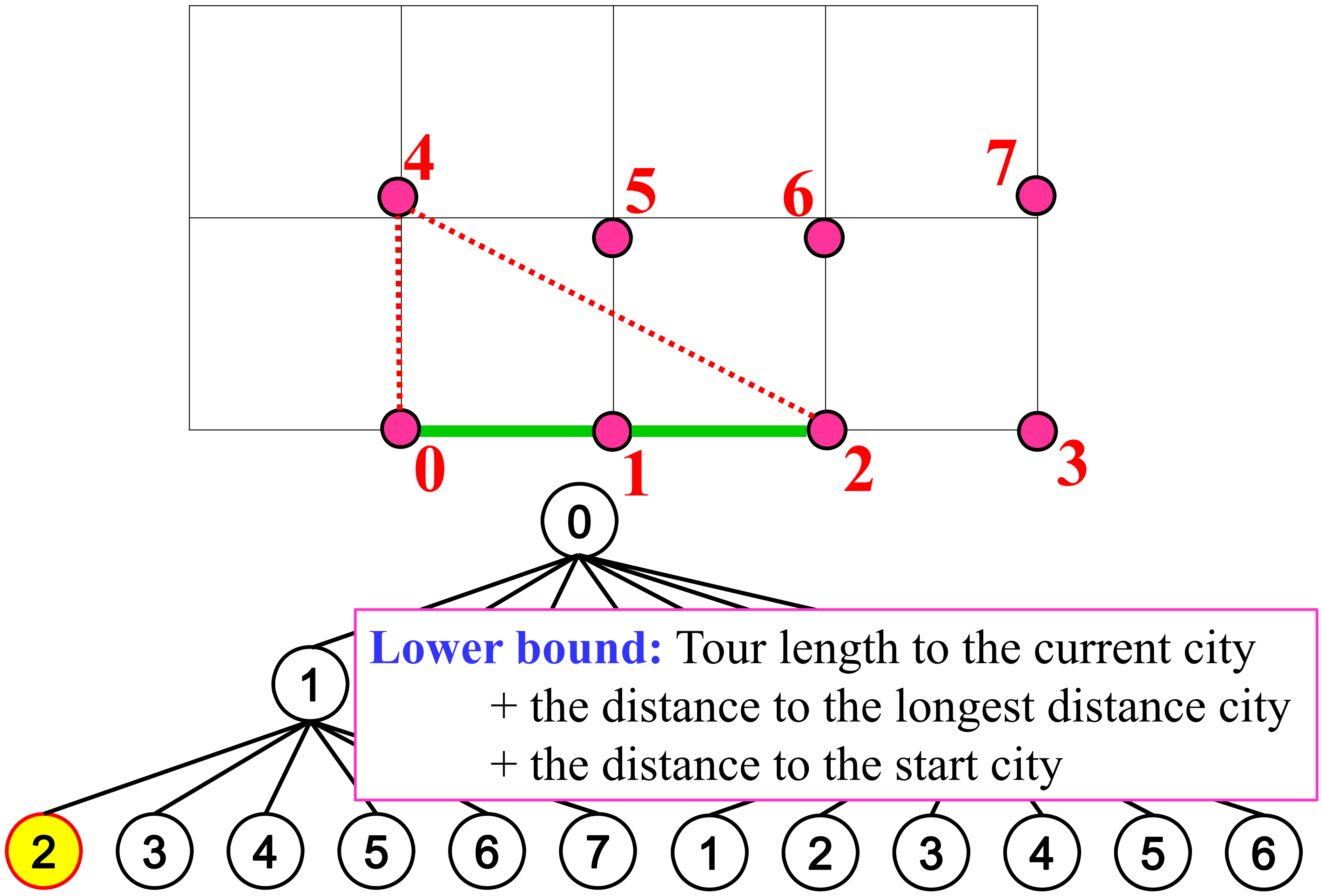


Example: 8-City TSP   Greedy solution: **Tour length = 10**



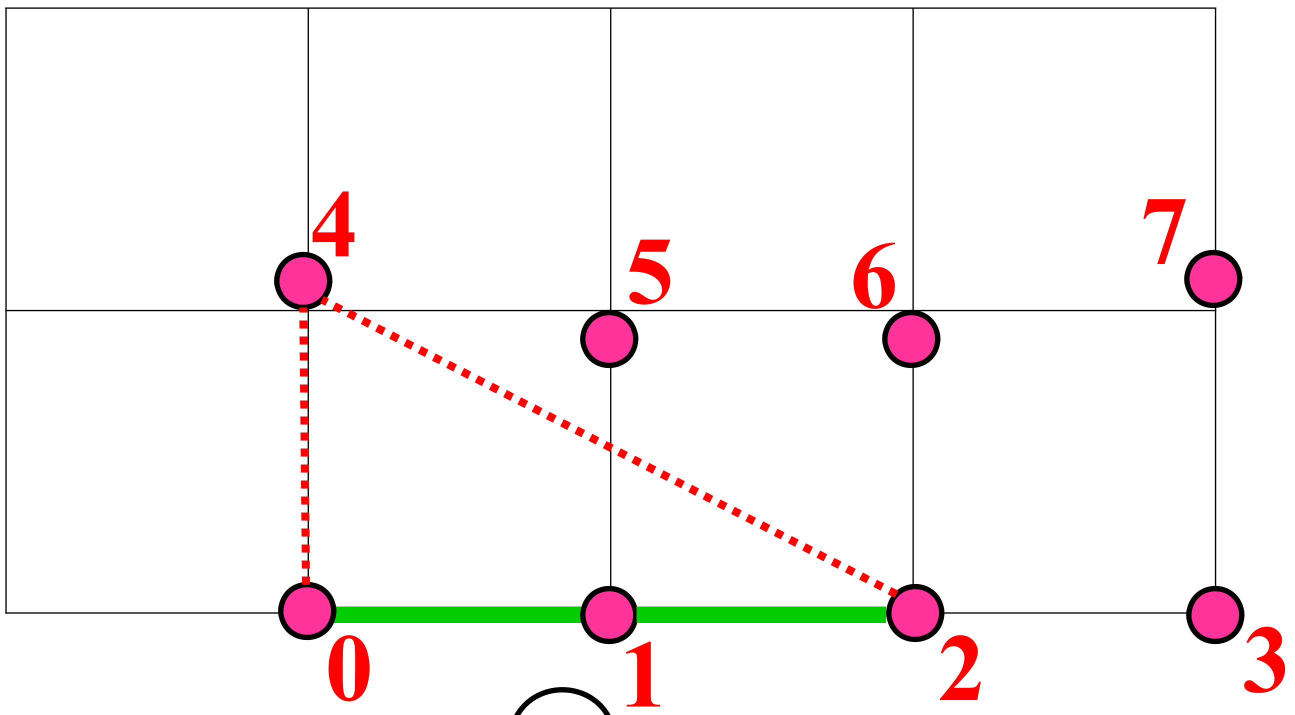


Example: 8-City TSP   Greedy solution: **Tour length = 10**

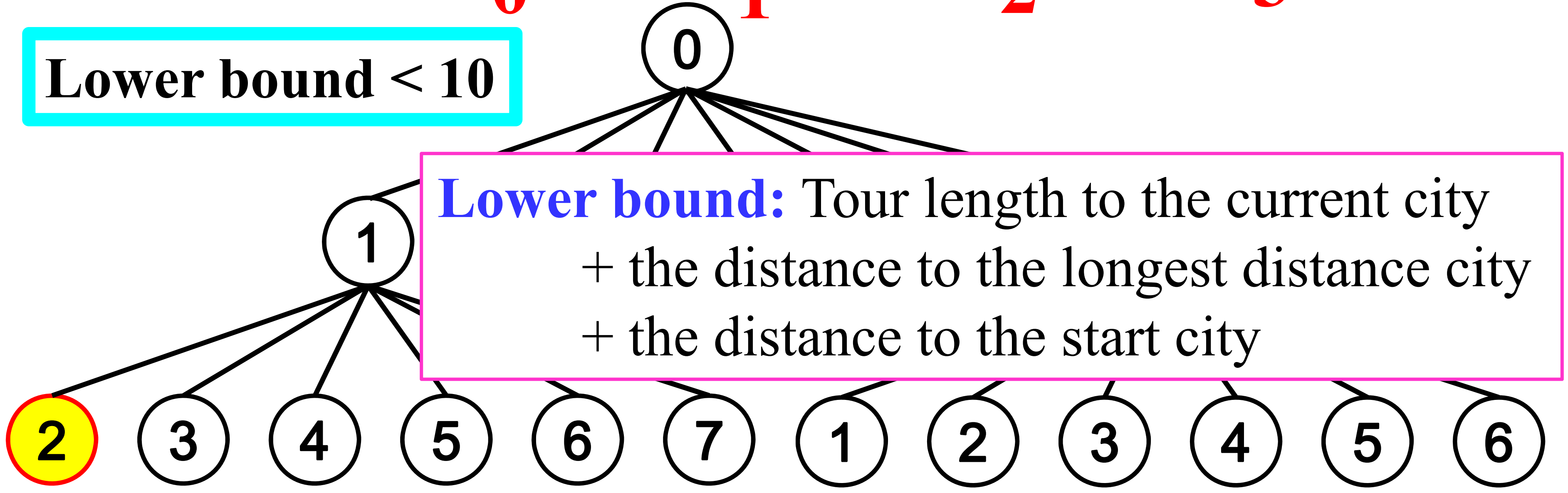


Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP Greedy solution: **Tour length = 10**

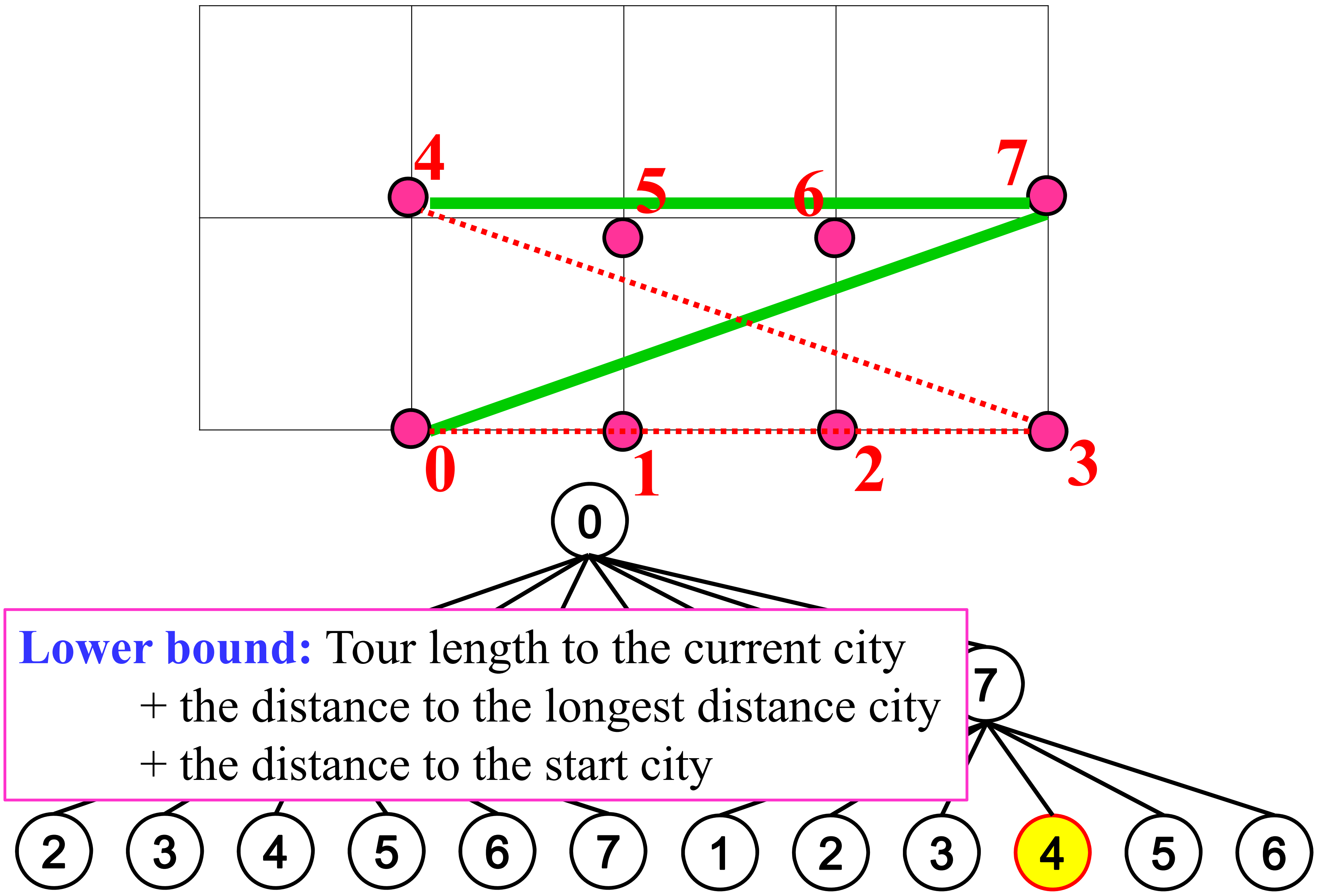


Lower bound < 10



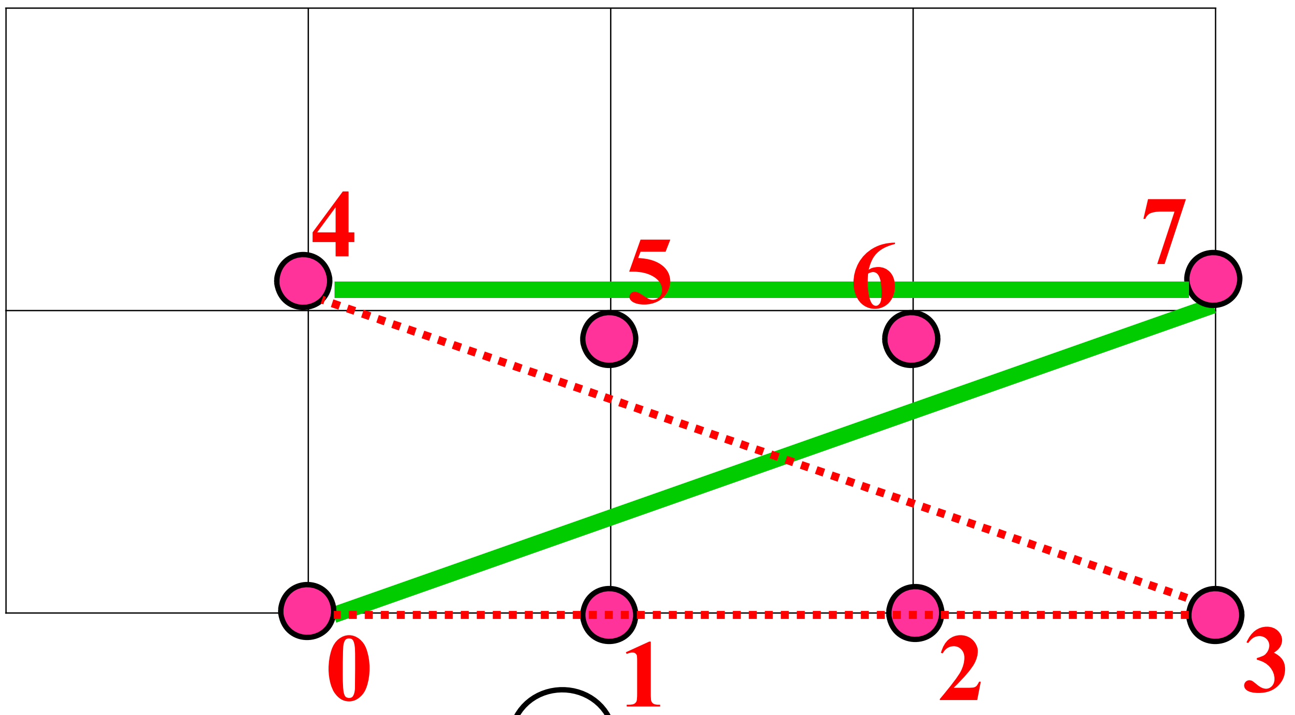
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP Greedy solution: **Tour length = 10**



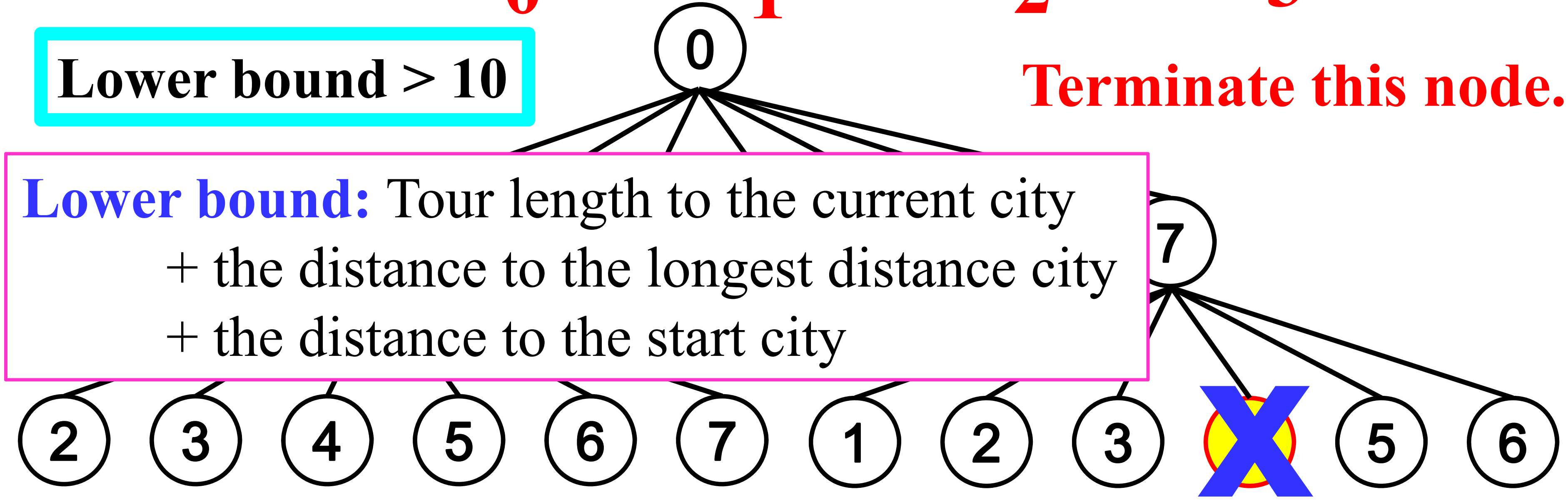
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP Greedy solution: **Tour length = 10**



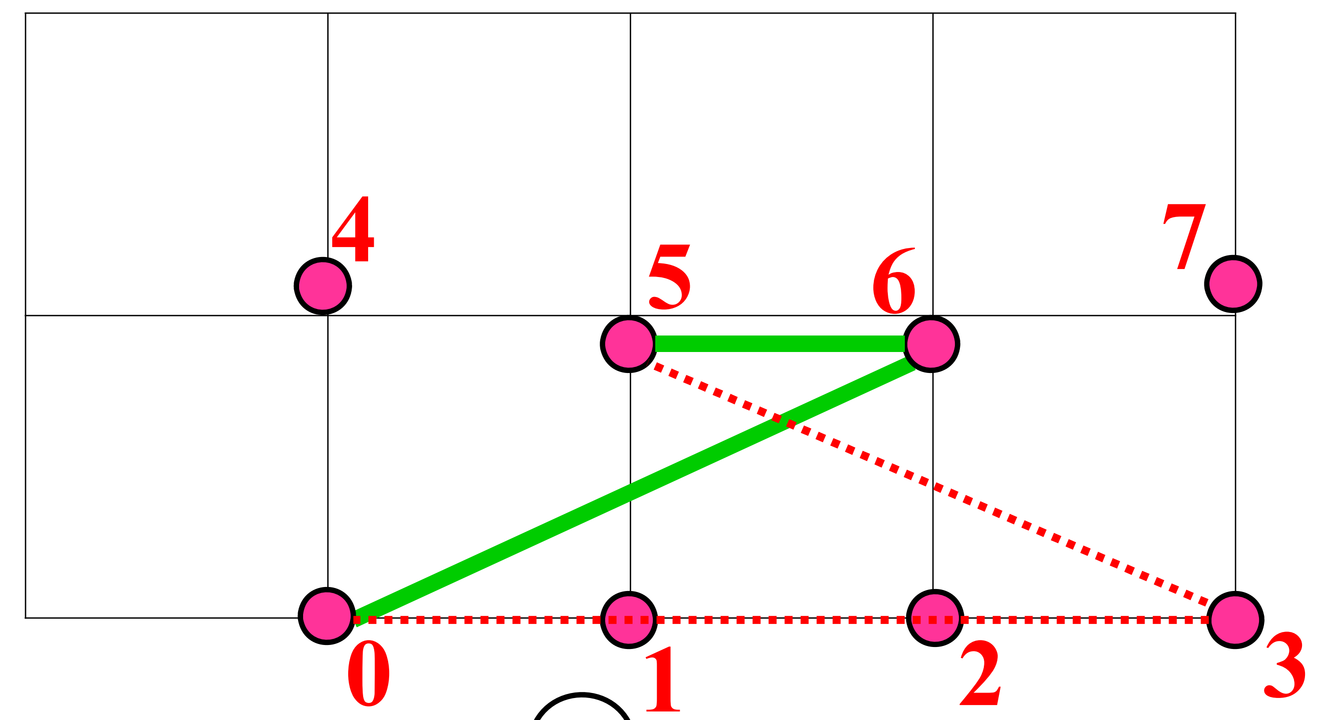
Lower bound > 10

Terminate this node.



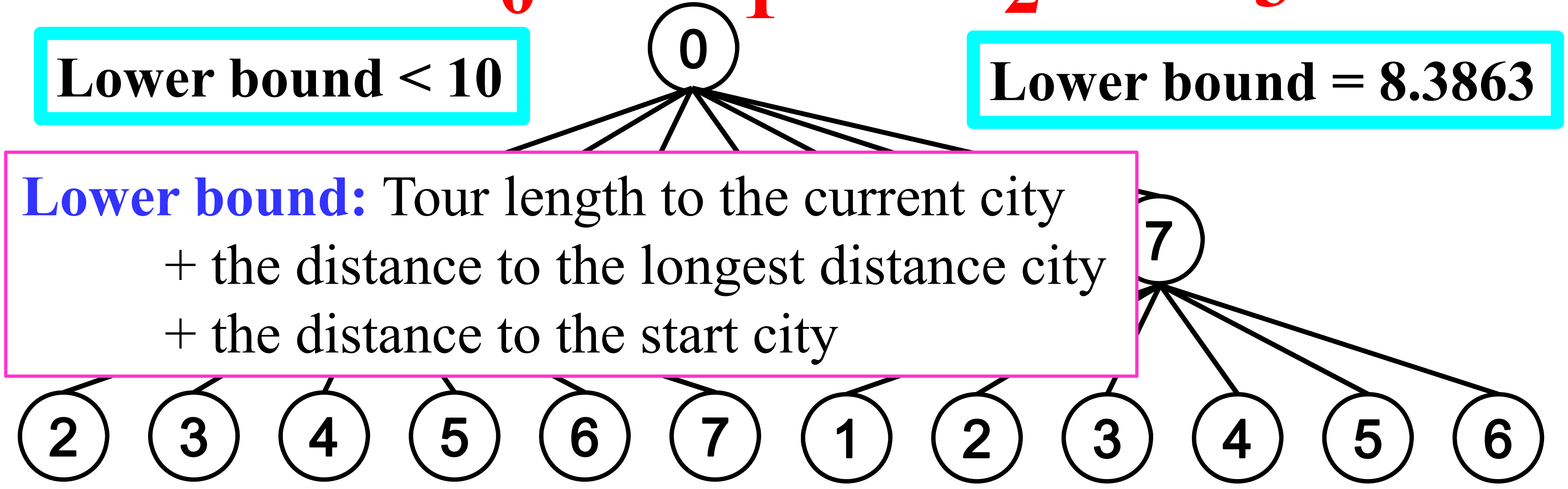
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP   Greedy solution: **Tour length = 10**



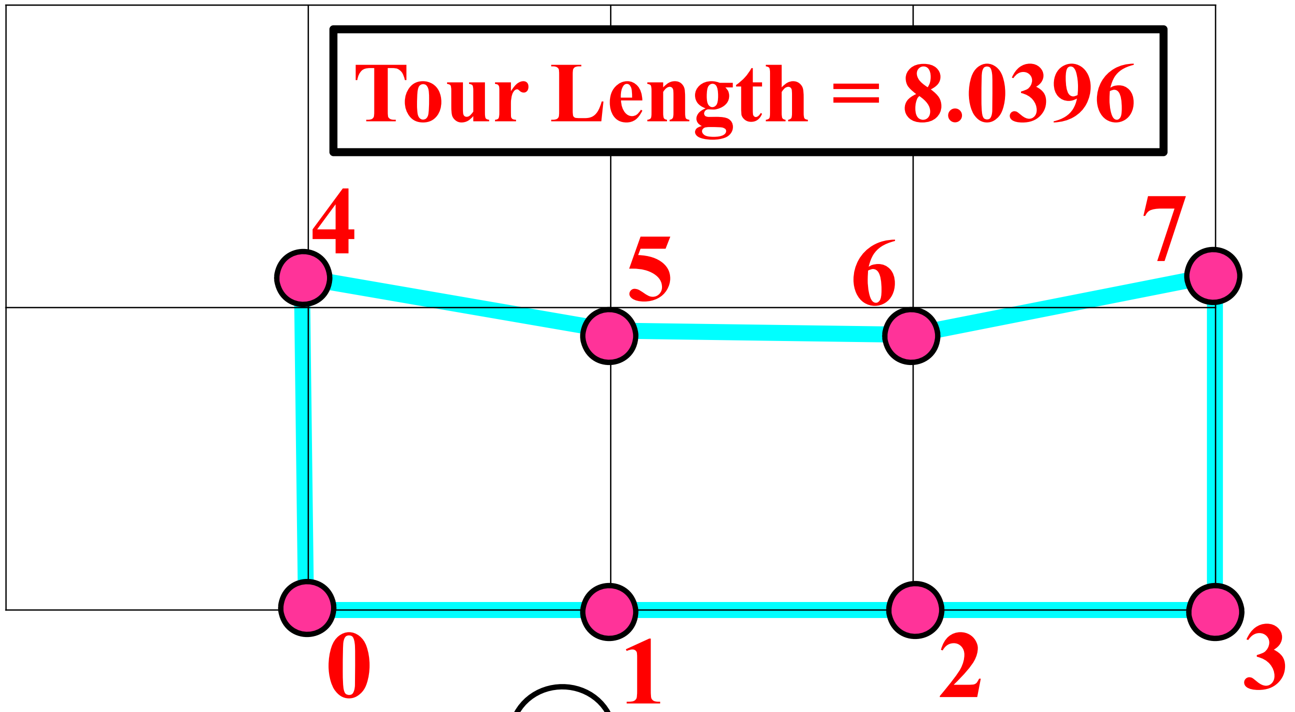
Lower bound < 10

Lower bound = 8.3863



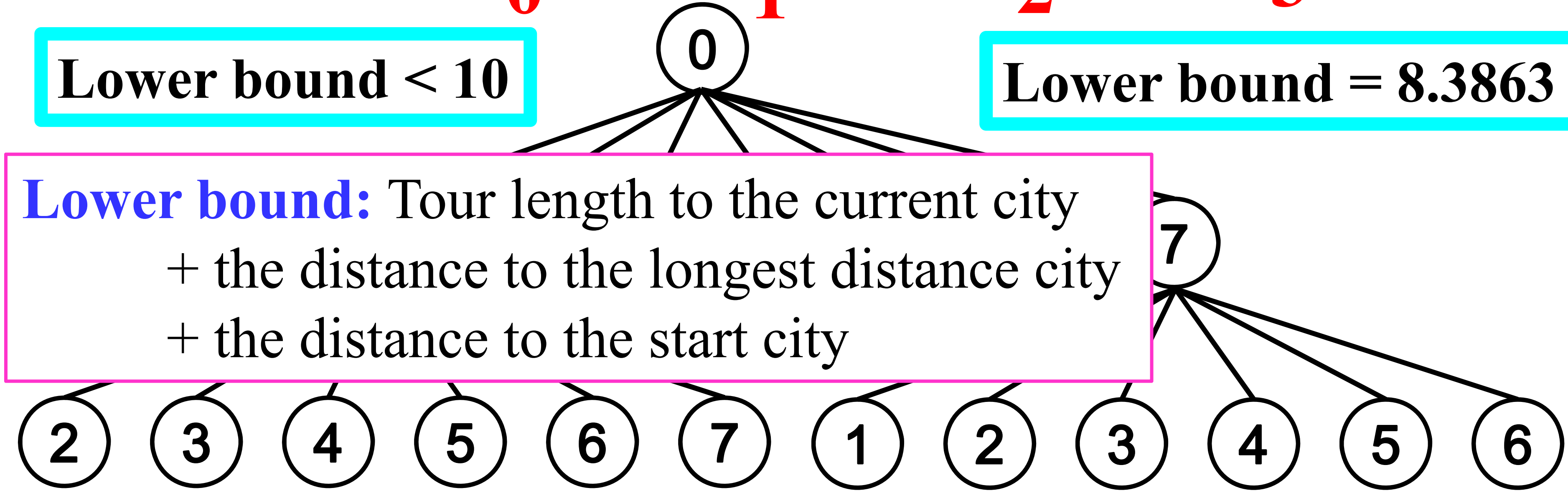
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP   Greedy solution: **Tour length = 10**



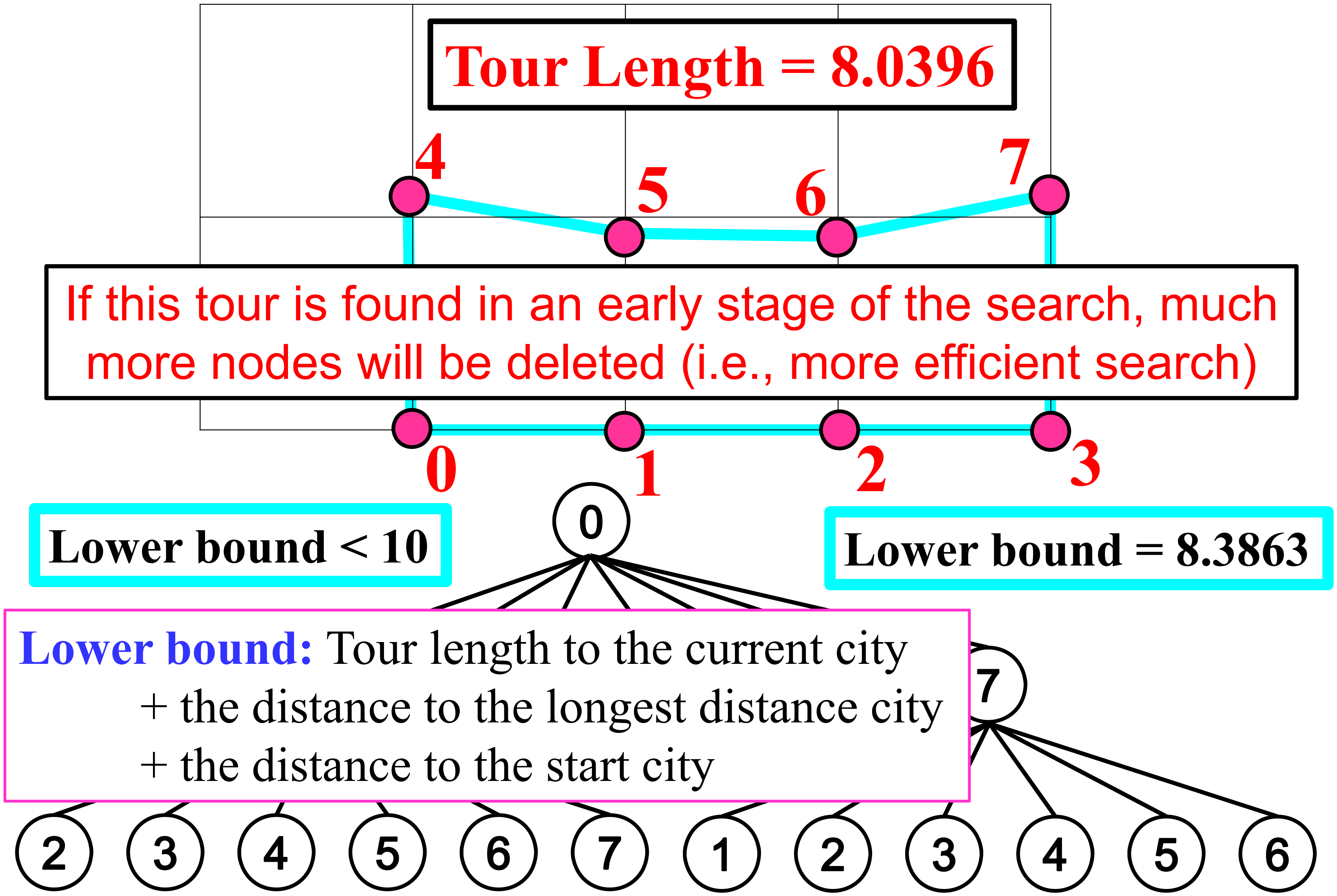
Lower bound < 10

Lower bound = 8.3863



Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Example: 8-City TSP   Greedy solution: **Tour length = 10**



Tour Length = 8.0396

If this tour is found in an early stage of the search, much more nodes will be deleted (i.e., more efficient search)

Lower bound < 10

Lower bound = 8.3863

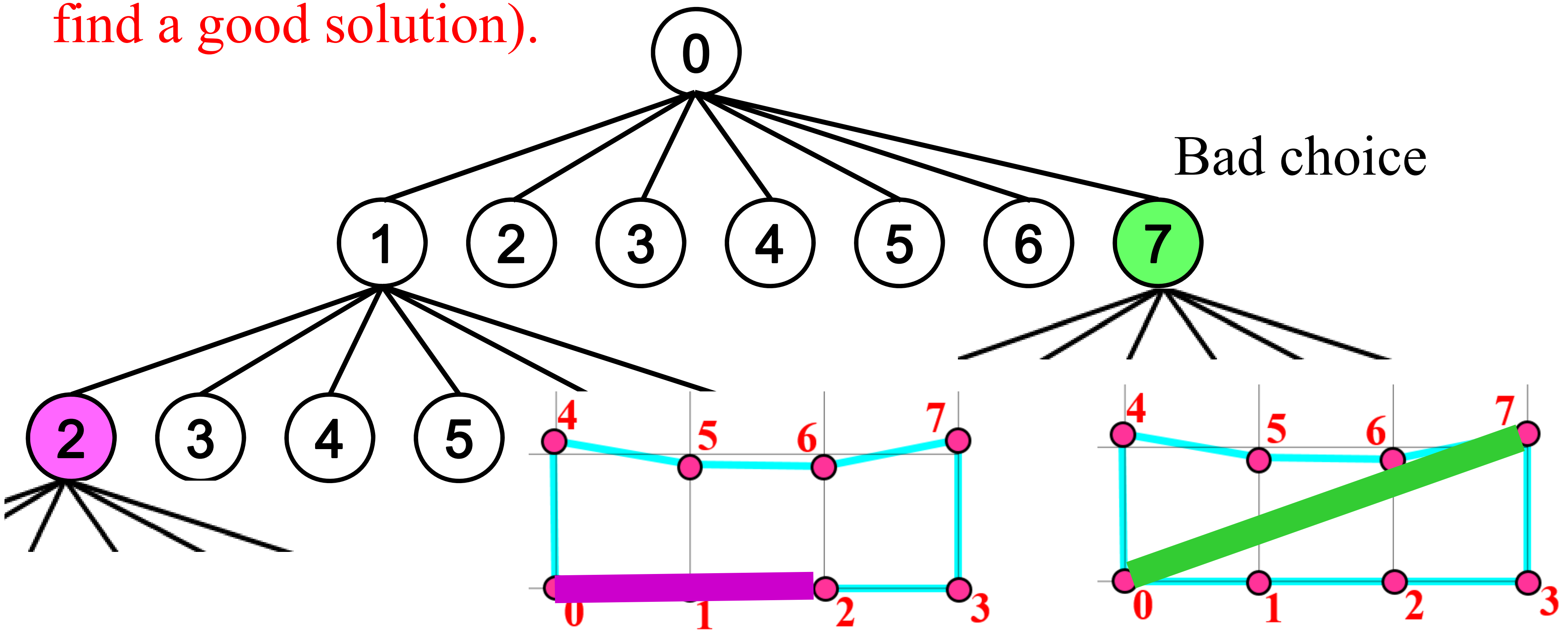
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city

Implementation Issues.

(1) How to obtain the initial upper bound (i.e., how to obtain an initial approximate solution). Greedy solution for TSP

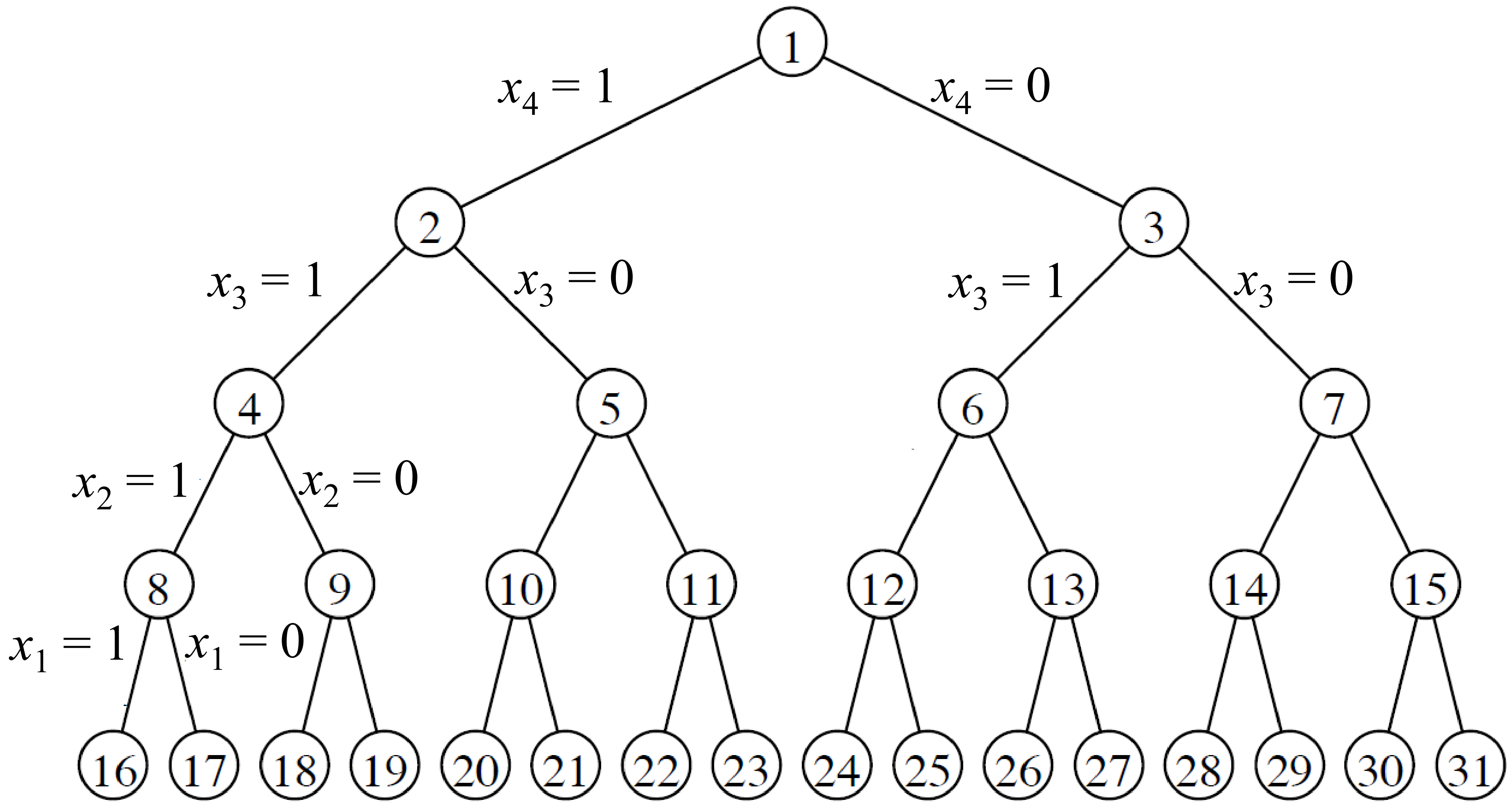
(2) How to calculate the lower bound of each node (i.e., the lower bound about the best possible objective value among all children from that node).

(3) How to choose the node for the next branching (to quickly find a good solution).



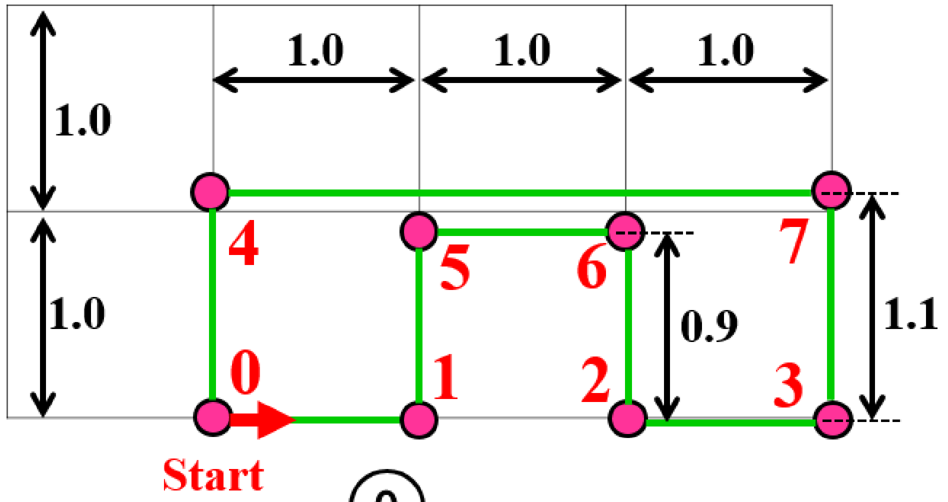
Tree Structure for Binary Strings of Length 4

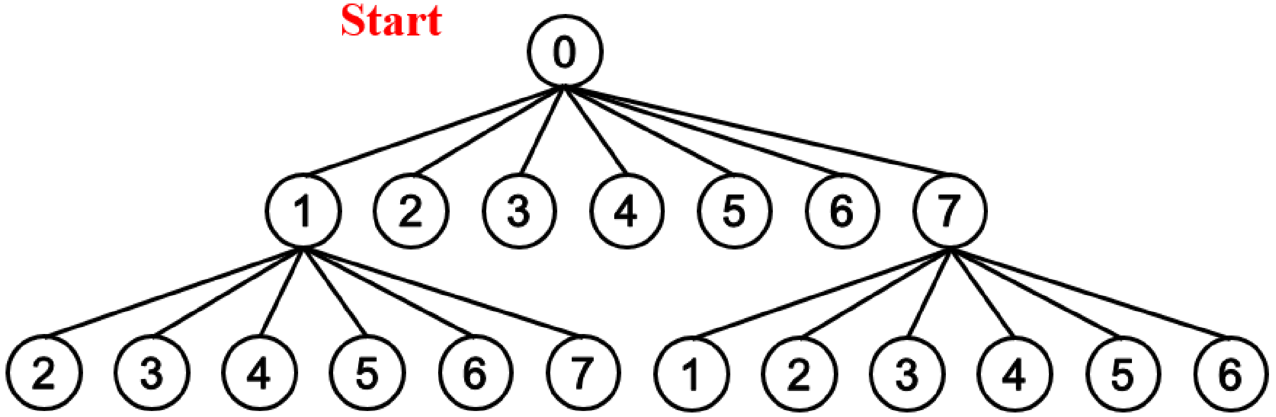
(From 1111 to 0000)



Lab Session

Example: 8-City TSP Greedy solution: **Tour length = 10**

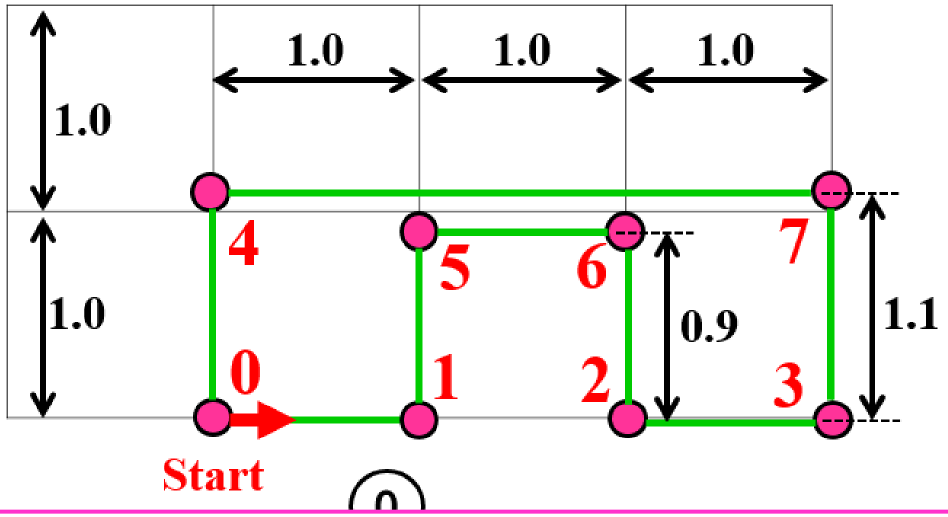




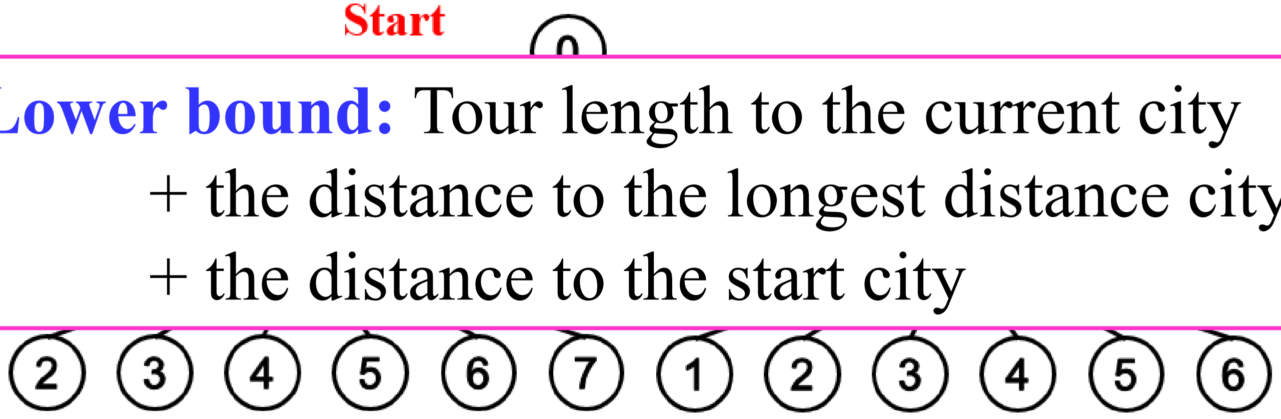
Q1. This tree has 42 nodes at the depth 2 level. How many depth 2 nodes can be terminated using the greedy solution of tour length 10?

Lab Session

Example: 8-City TSP Greedy solution: **Tour length = 10**



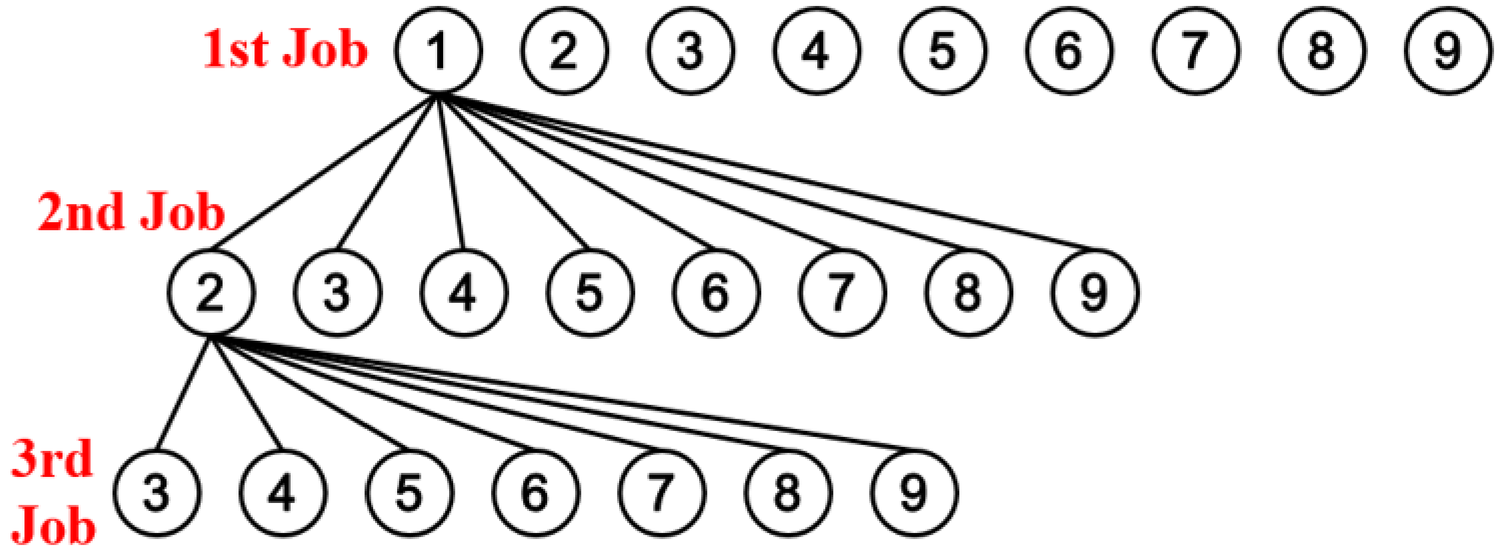
Lower bound: Tour length to the current city
+ the distance to the longest distance city
+ the distance to the start city



Q1. This tree has 42 nodes at the depth 2 level. How many depth 2 nodes can be terminated using the greedy solution of tour length 10?

Lab Session

Q2. For *m*-machine *n*-job flowshop scheduling problems (e.g., $m = 10$, $n = 100$), explain how we can specify the lower bound for each node. (Your idea)



Q3. For 10-machine *n*-job flowshop scheduling problems, discuss the largest problem size (i.e., the largest value of *n*) to which a blanch and bound algorithm can be applied in a practically acceptable computation time (e.g., one hour, one day). (Your idea)