Programming for Business Computing

Lists and (a little) Strings

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Reading many values in a line

- In many cases we need to read a lot of values into our program.
- An easy way to express an input is:
  - Values will be **put in a line**, **separated by** white spaces (or commas or other **delimiters**).
  - For example, five student grades may be put in these ways:

```
98 65 57 48 80
```

```
98,65,57,48,80
```

- One type of plain-text file, **CSV** (**comma-separated values**), uses commas to separate each row into multiple columns.
  - This is why it can be opened by MS Excel (or something similar).
- How to read these data into our program?

Reading many values as a string

- Python is very good at string processing.
- To read in a line of data, simply invoke **`input()`**.

```
gradeStr = input()
print(gradeStr)
```

- But gradeStr is a **string**, not five numbers!
- We need to do three things:
  - **Splitting** the string into five pieces (substrings).
  - Converting the five substrings into five numbers.
  - Put the five numbers into a **list**.

String splitting

- A string can be split by invoking **`split()`**.

```
gradeStr = input()      # 1 2 3 4 5
grades = gradeStr.split()
print(grades)           # ['1', '2', '3', '4', '5']
```

- We may choose the delimiter when invoking split().

```
gradeStr = input()      # 1,2,3,4,5
grades = gradeStr.split(',')
print(grades)           # ['1', '2', '3', '4', '5']
```

- What is **`grades`**?

Lists

- The outcome of invoking **`split()`** is a **list**.
  - A list is an **ordered container** that stores items.
  - An item may be an integer, a float, a string, or of other types.
  - Each item can be accessed by the **indexing operator `[]`**.
  - The **first** item is indexed at **0**!

```
gradeStr = input()                # 1 2 3 4 5
grades = gradeStr.split()
print(grades[0], grades[2] * 2)   # 1 33
```

- The length of a list can be obtained by invoking **`len()`**.

```
gradeStr = input()          # 1 2 3 4 5
grades = gradeStr.split()
print(len(grades))          # 5
```

List declaration

- We may **declare an empty list** as follows:

```
aList = []
print(aList, len(aList)) # [] 0
```

- We may **declare a list of three 0s** as follows:

```
aList = [0] * 3
print(aList, len(aList)) # [0, 0, 0] 3
```

- We may declare a list of items with various data types:

```
aList = [0, "hi", True]
print(aList) # [0, 'hi', True]
```

Putting items into a list

- We may add items into a list by invoking **append()**.

```
gradeStr = input() # 1 2 3 4 5
grades = gradeStr.split()
grades.append(-1)
print(grades) # ['1', '2', '3', '4', '5', -1]
```

  - Note that the last item is an integer, not a string.
- We may even put **a list** into a list:

```
gradeStr = input() # 1 2 3
grades = gradeStr.split()
grades.append([9, 7, 5])
print(grades) # ['1', '2', '3', [9, 7, 5]]
```

Traversing a list in a loop

- What does the following programs do?

```
print(range(1, 101))

sum = 0

for i in range(1, 101):
  sum = sum + i

print(sum)
```

```
gradeStr = input()                  # 1 2 3 4 5
gradeList = gradeStr.split()
print(gradeList)                    # ['1', '2', '3', '4', '5']
grades = []

for g in gradeList:
  grades.append(int(g))

print(grades)                       # [1, 2, 3, 4, 5]
```

List modification

- What does the following program do?

```
gradeStr = input()                 # 1 2 3 4 5
grades = gradeStr.split()
print(grades)                      # ['1', '2', '3', '4', '5']

for i in range(len(grades)):
  grades[i] = int(grades[i]) * 2

print(grades)                      # [2, 4, 6, 8, 10]
```

Example: month names

- What does the following program do?

```
# months is a list used as a lookup table
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

n = int(input("Enter a month number (1-12): "))

print("The month is", months[n - 1] + ".")
```

- What will you do if you cannot use a list?

Example: tic-tac-toe

- Let’s write a program to detect the winner of a tic-tac-toe game:

```
game = [[1, 0, 1], [1, 1, 0], [0, 0, 1]] # 2-dim list

for i in range(3):
  if game[i][0] == game[i][1] and game[i][1] == game[i][2]:
    print("winner:", game[i][0])
    break

# then check for columns and diagonals
```

X	○	X
X	X	○
○	○	X

List operations

Method	Meaning
`<list>.append(x)`	Add element **`x`** to end of list.
`<list>.sort()`	Sort (order) the list. A comparison function may be passed as a parameter.
`<list>.reverse()`	Reverse the list.
`<list>.index(x)`	Returns index of first occurrence of **`x`**.
`<list>.insert(i, x)`	Insert **`x`** into list at index **`i`**.
`<list>.count(x)`	Returns the number of occurrences of **`x`** in list.
`<list>.remove(x)`	Deletes the first occurrence of **`x`** in list.
`<list>.pop(i)`	Deletes the **`i`**th element of the list and returns its value.

Examples of list operations

- What does the following program do?

```
lst = [3, 1, 4, 1, 5, 9]
lst.append(2)
print(lst) # [3, 1, 4, 1, 5, 9, 2]

lst.sort()
print(lst) # [1, 1, 2, 3, 4, 5, 9]

lst.reverse()
print(lst) # [9, 5, 4, 3, 2, 1, 1]

print(lst.index(4)) # 2
```

```
lst.insert(4, "Hi")
print(lst) # [9, 5, 4, 3, 'Hi', 2, 1, 1]

print(lst.count(1)) # 2

lst.remove(1)
print(lst) # [9, 5, 4, 3, 'Hi', 2, 1]

print(lst.pop(3)) # 3

print(lst) # [9, 5, 4, 'Hi', 2, 1]
```

List copying

- Consider the following program:

```
aList = [1, 2, 3]
anotherList = aList

anotherList[0] = 5

print(aList) # [5, 2, 3]
```

- Why **`aList`** is modified?
- In Python, a list variable is a "**reference**" referring to a set of values.
  - Copying a list is just **copying the reference**, not those values.
  - Modifying the values through different references has the same effect.

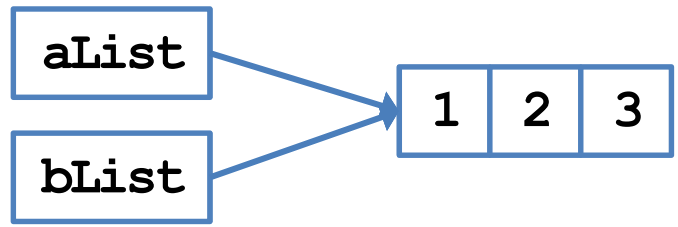
List copying

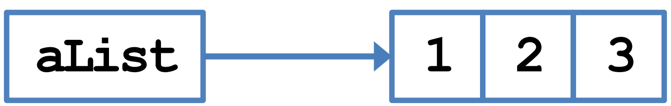
- Visualization:

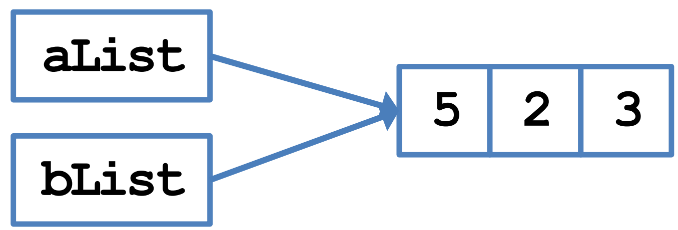
```
aList = [1, 2, 3] # (1)
bList = aList     # (2)

bList[0] = 5      # (3)

print(aList) # [5, 2, 3]
```

(2)


(1)


(3)


- In short, **`aList`** and **`bList`** are not really **two lists**; they are **two names** of **one single list**.

Programming for Business Computing
Applications in Operations Management

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Operations Management

- **Operations Management** (OM) deals with "operations."
  - Operations are activities inside a company.
  - It was **Production Management** at the beginning.
  - Today it includes (or is highly related to) Service Management, Decision Analysis, Supply Chain Management, etc.
- Typically OM issues:
  - Facility location.
  - Production planning and scheduling.
  - Inventory control.
  - Logistics and transportation.
- We will solve some OM problems by writing computer programs.
  - Basically we implement **algorithms**.

Algorithms

- An **algorithm** is a **step-by-step** procedure for solving a problem.
  - For a given task, it precisely describes **what to do** at each moment to complete that task.
- As an example, suppose that I want to sort Poker cards on my hands:
  - First put one card at the first position.
  - Look at the second card. Leave it there is it is bigger than the first one; exchange it with the first one otherwise.
  - Look at the third card and put it as the first, second, or third card to make the first three cards sorted.
  - …
  - For the *i*th card, "insert" it to a position that makes the first *i*th card sorted.
- In short, it is a **detailed description of actions** such that each action is doable.

Programming for Business Computing
Applications in Scheduling

Ling-Chieh Kung

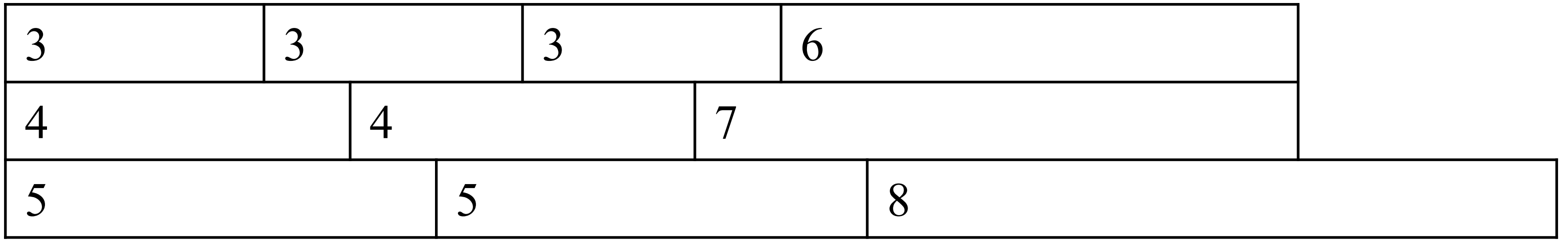
Department of Information Management
National Taiwan University

Scheduling: Makespan minimization

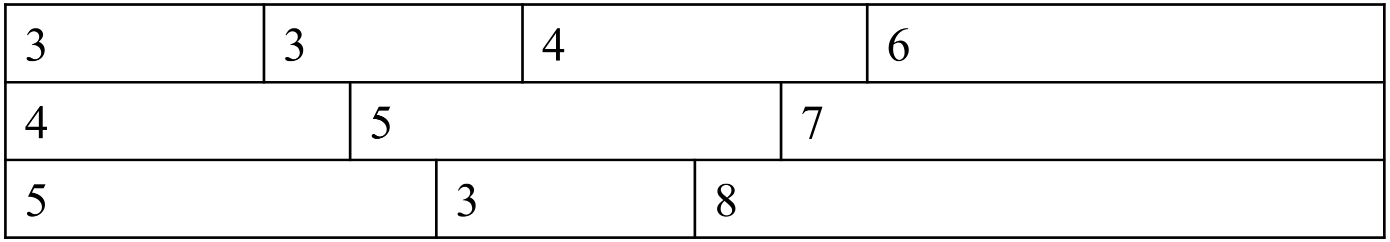
- There are m **machines** in a factory.
- There are n **jobs** to be processed.
- Job j has a given **processing time** p_j:
  - For a machine to complete job j, it needs to spend p_j amount of time.
  - We say job j is completed at its completion time C_j if $C_j = S_j + p_j$, where S_j is its starting time (the time that it is started to be processed).
- A typical scheduling problem is to schedule these jobs to machines to **minimize the *makespan***, i.e., the latest completion time among all jobs.

Example

- There are ten jobs with processing times 3, 3, 3, 4, 4, 5, 5, 6, 7, and 8.
- There are three machines.
- Schedule 1: Makespan = 18.



- Schedule 2: Makespan = 16.



Makespan minimization

- Makespan minimization is for **load balancing**.
  - To fairly allocation jobs.
  - To save utility fees.
  - To go home as early as possible.
- It turns out that this problem is “hard.”
  - Most researchers believe that an optimal schedule is too time-consuming to obtain in general.
- Several **heuristic algorithms** have been developed.
  - Typically easy to implement, easy to execute, and time-efficient.
  - Typically not too bad (near optimal).

Longest processing time first (LPT)

- One well-known algorithm for makespan minimization is the **longest processing time first** (**LPT**) rule.
  - First, **sort** jobs in the descending order of their processing times.
  - Then in each iteration schedule a job to the machine that is currently **the least loaded** (having the earliest completion time).
- We call LPT an **iterative algorithm**.
  - It runs in iterations.
  - In each iteration, it performs a similar action.
- We call LPT a **greedy algorithm**.
  - In each iteration, it makes the choice that is **the best at that moment**.
- In fact, even if we skip the sorting step, the algorithm still performs "well."

LPT implementation (without sorting)

- Let's implement LPT without the sorting step.
- First, read inputs and do the preparations:

```
# read and prepare n, m, and p
n = int(input("Number of jobs: "))
m = int(input("Number of machines: "))
pStr = input("Processing times: ")

p = pStr.split(' ')
for i in range(n):
  p[i] = int(p[i])

# machine completion times
loads = [0] * m
assignment = [0] * n
```

LPT implementation (without sorting)

- Second, we do the iterative assignment:

```
# in iteration j, assign job j to the least loaded machine
for j in range(n):

  # find the least loaded machine
  leastLoadedMachine = 0
  leastLoad = loads[0]
  for i in range(1, m):
    if loads[i] < leastLoad:
      leastLoadedMachine = i
      leastLoad = loads[i]

  # schedule a job
  loads[leastLoadedMachine] += p[j]
  assignment[j] = leastLoadedMachine + 1
```

LPT implementation (without sorting)

- Finally, we check the result:

```
# the result
print("Job assignment: " + str(assignment))
print("Machine loads: " + str(loads))
```

Remarks

- LPT has been shown to have a worst-case performance guarantee.
  - Let z^* be the makespan associated with an optimal solution.
  - Let z^{LPT} be the makespan obtained by LPT.
  - For any instance, we have $\frac{z^{\text{LPT}}}{z^*} \leq \frac{4}{3}$.
  - The approximation factor of LPT is $\frac{4}{3}$.
- Even if we do not sort jobs first, the approximation factor is 2.
- Analysis is hard, but implementation (and problem solving) is easy!

Programming for Business Computing
Applications in Inventory Control

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Inventory control

- **Inventory** are commonly needed in practice.
  - Retailers keep inventory of products.
  - Manufacturers keep inventory of raw materials.
- Why inventory?
  - To be a buffer between supply and demand.
  - To balance between fixed ordering cost and variable holding cost.
- To control inventory, people develop **inventory policies**.
  - When to order.
  - How many to order.
  - From whom to order (if there are multiple suppliers).

Common inventory policies

- A common inventory policy is **the $(\boldsymbol{Q}, \boldsymbol{R})$ policy**.
  - Regularly check the amount of inventory level I.
  - If $I < R$, order Q units. Otherwise, do nothing.
- Another common policy is **the $(\boldsymbol{s}, \boldsymbol{S})$ policy**.
  - Regularly check the amount of inventory I.
  - If $I < s$, order up to S. Otherwise, do nothing.

Automatic ordering

- Obviously, today people may implement **automatic ordering**.
- If we build a **continuous** review system:
  - Whenever one item is sold through the POS (point-of-sales) system, check the inventory level.
  - If the reorder point is reached, place an order.
- If we build a **periodic** review system:
  - Check the inventory level at the end of each “period,” (e.g., a day).
  - If the reorder point is reached, place an order.
- Let’s implement a periodic review system.

Implementation of the (Q, R) policy

- Let's implement the (Q, R) policy:

```
Q = int(input("Order quantity Q: "))
R = int(input("Reorder point R: "))
I = int(input("Initial inventory I: "))
print("Inventory: " + str(I))

while True:
  sales = int(input("Sales in a day: "))
  I = I - sales if I - sales >= 0 else 0
  if I < R:
    I = I + Q
  print("Inventory: " + str(I))
```

- How to implement the (s, S) policy?

Optimizing the (Q, R) policy

- How to choose the policy parameters Q and R?
- Three relevant costs:
  - **Inventory cost**: Cash generates investment returns, but inventory does not.
  - **Ordering cost**: The fixed cost incurred for each order (e.g., shipping cost).
  - **Shortage cost**: The loss sales and goodwill upon shortage.
- Objective: Minimize the sum of inventory, ordering, and shortage cost.

An instance

- Suppose that your boss asks you to optimize the (Q, R) policy:
  - Q is fixed to 30 due to a requirement set by the supplier.
  - The unit purchasing cost is \$1000 and the annual interest rate is 7.3%.
  - The per order shipping cost paid to the supplier is \$200.
  - If a customer comes but there is no on-hand inventory, she waits while getting \$2 off.
  - Daily demands for the past twenty days are given. Twenty units were on hand twenty days ago.

```
14,23,26,17,17,12,24,19,10,18,22,31,19,16,22,28,20,27,20,32
```

- Note that as Q is fixed, the ordering cost does not matter.
- Which R minimizes the total cost?

What if…

- What if we have chosen $R = 10$:

Sales		14	23	26	17	17	…
Before replenishment	20	6	13	–13	0	13	
After replenishment		36	13	17	30	13	
Inventory cost		\$7.2	\$2.6	\$3.4	\$6	\$2.6	
Shortage cost		\$0	\$0	\$26	\$0	\$0	

  - The total cost (for the past twenty days) with $R = 10$ is \$191.
  - Is it good or bad?

Finding the "optimal" *R*

- To find R that minimizes the **expected total cost** for the future, we need to estimate/forecast/predict future demands.
- A proxy is to find R that minimizes the total cost **for the past twenty days**.
  - For each value of $R = 0, 1, 2, 3, \ldots$, suppose that we have implemented the (Q, R) policy, what is the cost for the past twenty days?
  - If the demand pattern is going to remain unchanged, this should be good.
  - This works only if we have past demand, not only past sales.

Implementation

- First, we get the given information ready:

```
# past sales
salesStr = "14,23,26,17,17,12,24,19,10,18,22,31,19,16,22,28,20,27,20,32"
sales = salesStr.split(',')
for i in range(len(sales)):
  sales[i] = int(sales[i])

# given information
stgCost = 2
invCost = 1000 * 0.073 / 365
Q = 30
I = 20
```

Implementation

- We then search for the best *R* and print it out.

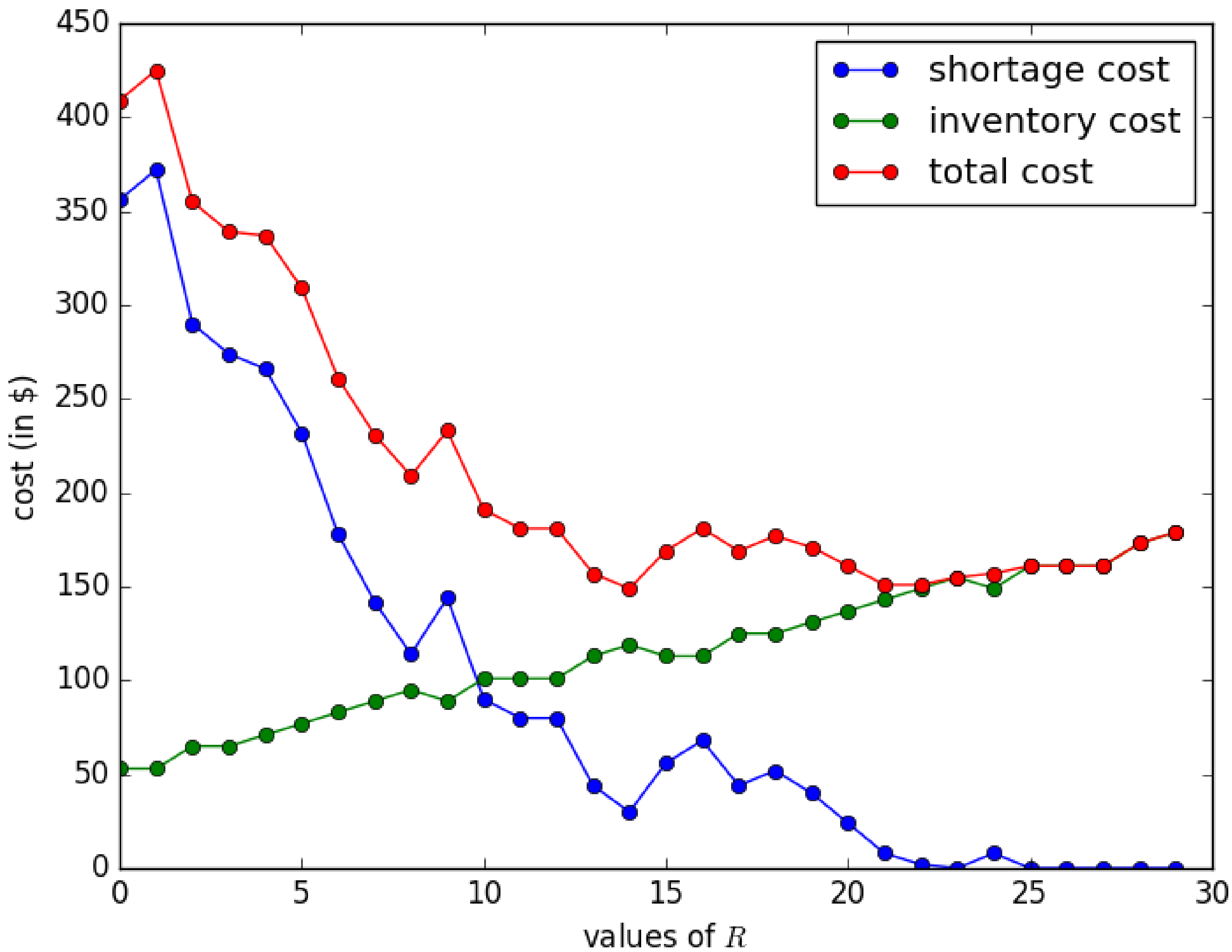
```
# finding the best R
bestR = 0
costOfBestR = 100000000
for R in range(Q):
  totalCost = 0

  # finding the total cost of this R
  for s in sales:
    I -= s
    if I < 0:
      totalCost += -I * stgCost
      I += Q
    elif I < R:
      I += Q
    totalCost += I * invCost

  # update bestR when necessary
  if totalCost < costOfBestR:
    bestR = R
    costOfBestR = totalCost

print(bestR)
```

Visualization



Further questions

- What if we also want to optimize Q?
- What if we have multiple products with independent ordering operations?
- What if we have multiple products, and it is preferred to combine their purchasing orders?
- What if sales are lost rather than backlogged?

Programming for Business Computing
Applications in Logistics and Transportation

Ling-Chieh Kung

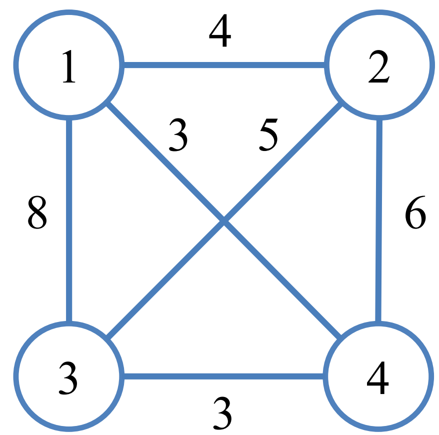
Department of Information Management
National Taiwan University

Logistics and transportation

- **Logistics** and **transportation** problems are common in supply chain management and service management.
  - Route, quantity, timing, and transportation mode selection.
  - Shipping goods among suppliers, manufacturers, and retailers.
  - Shipping goods to consumers.
  - Sending passengers here and there.
- Many of these problems are solved by **algorithms**.
  - To save cost.
  - To deliver better service.
  - To increase profit.

Travelling salesperson problem

- In many cases, we need to deliver/collect items to/from customers in the most efficient way.
- E.g., consider a post officer who needs to deliver to three addresses.
  - The shortest path between any pair of two addresses can be obtained.
  - This is a **routing** problem: To choose a route starting from the office, passing each address exactly once, and then returning to the office.
- This is a **sequencing** problem; in total there are 3! = 6 feasible routes (some are identical). Which route minimizes the total distance (or travel time)?
- The problem described above is the **traveling salesperson problem** (TSP).

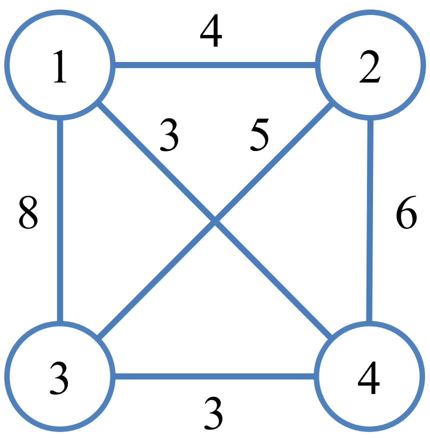


Vehicle routing problem

- Sometimes the **vehicle capacity** is also an issue.
- Consider the truck towing bicycles in NTU.
  - It must start at the car pound, pass several locations in NTU, and then return to the origin.
  - However, the truck capacity is quite limited (because too many people violate the parking regulation).
  - The driver needs to find multiple routes to cover all the locations.
- The traveling salesperson problem is a special case of vehicle routing problems.
  - The vehicle capacity is unlimited.

A greedy algorithm for TSP

- Let's consider **a greedy algorithm** for TSP.
  - At the origin, go to the next location that is **closest**.
  - At each location, go to the next location among all unvisited ones that is **closest** to the current location.
  - Repeating this until we get back to the origin.
- In this example, we will travel in the route (1, 4, 3, 2, 1). The total distance is 15.
  - (1, 4, 3, 2, 1) is optimal for this instance.
  - Does the greedy algorithm always find an optimal route?



Implementation of the greedy algorithm

- Let's implement the greedy algorithm.
- First, we get the given data ready:

```
# set up the distance matrix
numLoc = 5
dst = [[0, 9, 6, 7, 4], # 2-dim list
       [9, 0, 5, 9, 6], # dst[i][j] is the
       [6, 5, 0, 3, 1], # distance between
       [7, 9, 3, 0, 4], # locations i and j
       [4, 6, 1, 4, 0]]

# set up the origin
origin = 0
```

Implementation of the greedy algorithm

- Second, we prepare two lists **`tour`** and **`unvisited`**
  - In each iteration, we move one location from **`unvisited`** and to **`tour`**.

```
# tour: a list that will contain the solution
# tourLen: the total distance of the solution
# unvisited: a list that contains those
#            unvisited locations at any time
tour = [origin]
tourLen = 0
unvisited = []
for i in range(numLoc):
  unvisited.append(i)
unvisited.remove(origin)
```

Implementation of the greedy algorithm

- The algorithm consists of **numLoc** – 1 iterations.
- In each iteration, we check the distance between the current location and each unvisited location.
  - To find the closest unvisited location.
- We move to the next location to initiate the next iteration.

```
# The algorithm
cur = origin
for i in range(numLoc - 1):
  # find the next location to visit
  next = -1
  minDst = 999
  for j in unvisited:
    if dst[cur][j] < minDst:
      next = j
      minDst = dst[cur][j]

  # move "next" from unvisited to tour
  unvisited.remove(next)
  tour.append(next)
  tourLen += minDst

  # run the next iteration from the next location
  cur = next
```

Implementation of the greedy algorithm

- Finally, we complete the tour and print out the solution.

```
# complete the tour
tour.append(origin)
tourLen += dst[cur][origin]

# print out the solution
print(tour, tourLen)
```

Improving the input process

- Our program should allow a user to input her/his instance information.

```
# set up the distance matrix
numLoc = 5
dst = [[0, 9, 6, 7, 4],
       [9, 0, 5, 9, 6],
       [6, 5, 0, 3, 1],
       [7, 9, 3, 0, 4],
       [4, 6, 1, 4, 0]]

# set up the origin
origin = 0
```

➔

```
# set up the distance matrix
numLoc = int(input())
dst = []
for i in range(numLoc):
  dst.append(input().split())
  for j in range(numLoc):
    dst[i][j] = int(dst[i][j])

# set up the origin
origin = 0
```

- How to avoid entering a lot of numbers for each trial?
  - First, save your input in a plain text file. Give it a file name (e.g., "in.txt").
  - Open the console/terminal window to execute your program.
  - Use `<` (or something similar) like "`python TSP.py < TSP_in.txt`" to load the input data.

Remarks

- Many (quantitative) decisions may be supported by computers and programs.
  - As long as we have a good **algorithm** and **know how to program**.
- To solve real problems, we need **domain knowledge**.
  - Being able to program allows one to **implement a solution** (by herself/himself or collaborating with engineers).
  - Being able to program allows one to **try** some solutions.
  - Being able to program facilitates "**learning by doing**."