Programming Design

Advanced Topics

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Road map

- **Python**
- C++ vs. Python
- The power of data structures

C++ vs. Python

- In this course, we study the C++ programming language.
  - It is a **compiled language**.
  - It is a **statically typed language**.
- Basically this is why C++, C, Java, etc., are "**fast**."
  - Let's feel how fast C++ is by comparing it to **Python**.
- Python is one of the most popular language nowadays.
  - It is an **interpreted language**.
  - It is a **dynamically typed language**.
  - It is great for beginner; it is great for writing "small" programs.
  - However, it can be "slow."

Python

- Python was invented by Guido van Rossum around 1996: “Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas.”
  – The latest version (in August, 2017) is **3.6.2**.
- Python is very good for beginners.
  – It is simple.
  – It is easy to start.
  – It is powerful.

(https://en.wikipedia.org/wiki/Python_(programming_language)

How to run Python

- To taste Python online:
  - https://www.python.org/ or other similar websites.
- To get the Python interpreter:
  - Go to https://www.python.org/downloads/, download, double click, and then click and then click… and then you are done.

Interpreting a program

- An **interpreter** translates programs into assembly programs.
  - For other high-level programs (including C, C++, Java, etc.), a **compiler** is used.
  - Python uses an interpreter.
- An interpreter interpret a program **line by line**.
- We may write Python in the **interactive mode**.
  - Input one line of program, then see the result.
  - Input the next line, then see the next result.
  - The statements should be entered after the **prompt**.

```
>>> 3 + 6
9
>>> 4 - 2
2
>>> a = 100
>>> b = 50
>>> c = a - b
>>> print(c)
50
```

Interpreting a program

- We may also write Python in the **script mode**.
  - Write several lines in a file (with the extension file name .py), and then interpret all the lines one by one at a single execution.
- A programming language using an interpreter is also called a **scripting language**.
  - E.g., R.

```
for i in xrange(0, bingo):
    a = random.randint(start, end) - 1
    temp = seqNo[a]
    seqNo[i] = temp

seqNoSorted = sorted(seqNo[0:bingo])
#print(seqNoSorted)

for i in xrange(0, bingo):
    print(seqNoSorted[i])
```

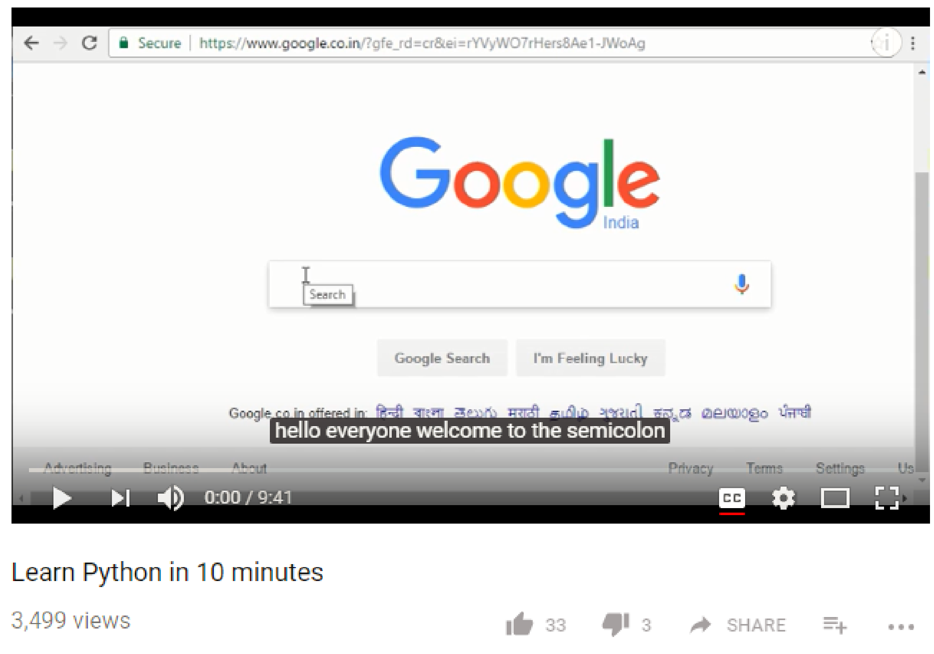
How to run Python

- To try the interactive mode:
  - Open your console (the command line environment) and type **`python`** to initiate the interactive mode.
  - You may need to set up your "PATH" variables.
- To write Python on an **editor** and interpret a script with the interpreter:
  - Open a good text editor (e.g., Notepad++), write a script, save it (.py).
  - Open the **console**, locate your script file (.py), interpret it with the instruction **`python`**, and see the result.

(Figure 1.1, *Think Python*)

Let write Python!

- Let's "learn Python in ten minutes."
  - https://www.stavros.io/tutorials/python/
  - https://www.youtube.com/watch?v=a5Y3e9aqMg8
  - https://leanpub.com/learn-python/read



Let’s write Python: “Hello world!”

- Recall our first C++ program:

```cpp
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

- In python:

```python
print("Hello World!")
```

- So easy!

Let's write Python: input and sum

- Recall our second C++ program:

```
#include <iostream>
using namespace std;

int main()
{
  int num1 = 0, num2 = 0;
  cin >> num1 >> num2;

  cout << "The sum is "
       << num1 + num2;

  return 0;
}
```

- In python:

```
num1 = int(input())
num2 = int(input())

print("The sum is",
      num1 + num2)
```

- No need to **declare a variable**.
  – No need to specify a **type**.
  – But still can do **casting**.

Let's write Python: if and while

- Recall our third C++ program:

```
#include <iostream>
using namespace std;

int main()
{
  int num1 = 0, num2 = 0;
  cin >> num1 >> num2;

  while(num1 > num2)
  {
    cout << "number 1 is "
         << num1 << "\n";
    num1 = num1 - 1;
  }

  return 0;
}
```

- In python:

```
num1 = int(input())
num2 = int(input())

while num1 > num2:
  print("number 1 is", num1)
  num1 = num1 - 1
```

- No semicolon.
  - Python uses **line breaks** to separate statements.
  - Python uses **indentions** to determine blocks.

Let's write Python: array, list, function

- In C++, we may create static or dynamic arrays:

```
#include <iostream>
using namespace std;

int print(int** arr, int r)
{
  for(int i = 0; i < r; i++)
  {
    for(int j = 0; j <= i; j++)
      cout << arr[i][j] << " ";
    cout << "\n";
  }
}
```

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++) {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r);
  for(int i = 0; i < r; i++)
    delete [] array[i];
  delete [] array;
  return 0;
}
```

Let’s write Python: array, list, function

- In Python, we use **lists**:

```
def printList(arr, r):
  for i in range(r):
    for j in range(i + 1):
      print(arr[i][j], end = " ")
    print()
```

```
r = 3
array = []
for i in range(r):
  array.append([])
  for j in range(i + 1):
    array[i].append(j + 1)
printList(array, r)

# print(array)
```

- `def` defines a **function**.
  - `range(r)` creates a list of integers 0, 1, …, `r` – 1.
- All **function parameters** are not declared with types.

Let’s write Python: bubble sort

- Recall our bubble sort in C++:

```
void bubbleSort(const int unsorted[], int sorted[], int len)
{
  for(int i = 0; i < len; i++)
    sorted[i] = unsorted[i];

  for(int i = len - 1; i > 0; i--) {
    for(int j = 0; j < i; j++) {
      if(sorted[j] > sorted[j + 1]) {
        int temp = sorted[j];
        sorted[j] = sorted[j + 1];
        sorted[j + 1] = temp;
      }
    }
  }
}
```

Let’s write Python: bubble sort

- In Python:

```
def bubbleSort(unsorted, sorted, len):
  for i in range(len):
    sorted[i] = unsorted[i]

  for i in range(len - 1, 0, -1):
    for j in range(i):
      if sorted[j] > sorted[j + 1]:
        temp = sorted[j]
        sorted[j] = sorted[j + 1]
        sorted[j + 1] = temp
```

- **`range(len - 1, 0 -1)`** creates a list **`len`** – 1, **`len`** – 2, …, 1.

Good!

- Now you know **two** programming languages!
- You may **learn more by yourself** in the future.
  - As long as you want.
  - As long as you have Internet access.
  - As long as you have a solid foundation.

- But do not get confused by the word “language”…

A little joke

- 作者 XXXXX (只願上天的成全) 看板 NTUIM-11
  標題 請問
  時間 Tue Nov 5 22:54:40 2002

  想要買本資料結構的書來看

  不知道那一種板本寫的比較詳細

  或是那一種板[本的書比較容易懂的

  知道的人可以回一下嗎?

A little joke

- 作者 rrro (小傑) 看板 NTUIM-11
  標題 Re: 請問
  時間 Wed Nov 6 08:03:37 2002

※ 引述《gentle (只願上天的成全)》之銘言：
: 想要買本資料結構的書來看
: 不知道那一種板本寫的比較詳細
: 或是那一種板[本的書比較容易懂的
: 知道的人可以回一下嗎?
你要用什麼語言的啊？

A little joke

- 作者 XXXXX (只願上天的成全) 看板 NTUIM-11
  標題 Re: 請問
  時間 Wed Nov 6 22:17:33 2002

  ※ 引述《rrro (小傑)》之銘言：
  : ※ 引述《gentle (只願上天的成全)》之銘言：
  : : 想要買本資料結構的書來看
  : : 不知道那一種板本寫的比較詳細
  : : 或是那一種板[本的書比較容易懂的
  : : 知道的人可以回一下嗎?
  : 你要用什麼語言的啊？

  中文的~

  謝謝

Road map

- Python
- **C++ vs. Python**
- The power of data structures

C++ vs. Python

- Let's use C++ and Python as examples to compare:
  – Compilation and interpretation.
  – Static typing and week typing.
  – Without and with initial values.
  – "Old" and "new" languages.

Compilation and interpretation

- To write a program in a **compiled language**, **the whole program** must be compiled before an executable file is generated.
  - There should be no syntax error in the whole program.
  - There is a program; there is an executable file.
- To write a program in an **interpreted language**, **each line of code** may be interpreted by itself.
  - That statement must have no syntax error; syntax errors in later statements do not matter (at least in the interactive mode).
  - There can be no program; there is no executable file.

```
>>> 3 + 6
9
>>> 4 - 2
2
>>> a = 100
>>> b = 50
>>> c = a - b
>>> print(c)
50
```

Compilation and interpretation

- **Executing** an executable file is typically **faster** than interpreting and running an interpreted program.
- However, **developing** a program in an interpreted language is typically **faster**.
  - We may just write a small piece and then test it.
- A good combination:
  - Design a way to solve your problem.
  - Write a program in an interpreted language to validate your solution.
  - After you confirm the effectiveness of your solution, write a program in a compiled language (if needed) to generate an executable file. Run the executable file in the future.

Static typing and dynamic typing

- **Static typing** means the types of variables must be determined during the **compilation time**.
  - In C++, we declare variables by specifying types.
  - A variable's type does not change during the run time (though its value's type may change).
- **Dynamic typing** means variable types will be determined during the **run time**.
  - In Python, we do not declare a variable's type.
  - A variable's type will change during the run time (depending on the type of the value assigned to it).
- They are also called strong typing and weak typing.

```
r = 3
array = [4, 1, 3]

print(type(r))
print(type(array))

array = 1.4
r = "this is a string"

print(type(r))
print(type(array))
```

Static typing and dynamic typing

- A program written with static typing typically **runs faster**.
  – No need to change a variable's type during the run time.
- Developing a program with static typing may **take more time**.
  – Need a clear understanding about types and casting.
  – Typically **more syntax errors**; may have **fewer run-time errors**.

Initial values

- In C++, a newly declared variable (typically) does not get its **initial value** automatically.
  - The programmer must assign an initial value to the variable manually.
  - This is called **initialization**.
- In Python, a new variable (typically) gets an initial value automatically.
  - In fact, because there is no need to "declare a variable," almost always we assign a value to a variable when creating it.
- Running a C++ program will take more time if C++ assigns initial values to all variables..

```
a = int()
b = float()
c = ""
d = list()

print(a, b, c, d)
```

A numerical experiment

- Let’s do a numerical experiment to test the speeds of C++ and Python.
  - We use C++ and Python to implement the same algorithms “bubble sort” and “insertion sort.”
  - We then use each implementation to sort 2000, 4000, 6000, …, or 20000 randomly generated integers.
  - For each scenario (number of integers), we run each implementation and record the time.
- Ideally, we should include multiple (say, 50) instances in each scenario for us to calculate an average across all instances. For simplicity, below we will generate only one instance for each scenario.

C++ implementations: two functions

```
void bubbleSort(const int unsorted[],
                int sorted[], int len)
{
  for(int i = 0; i < len; i++)
    sorted[i] = unsorted[i];

  for(int i = len - 1; i > 0; i--) {
    for(int j = 0; j < i; j++) {
      if(sorted[j] > sorted[j + 1]) {
        int temp = sorted[j];
        sorted[j] = sorted[j + 1];
        sorted[j + 1] = temp;
      }
    }
  }
}
```

```
void insertionSort(const int unsorted[],
                   int sorted[], int len)
{
  for(int i = 0; i < len; i++)
    sorted[i] = unsorted[i];

  for(int i = 0; i < len; i++) {
    for(int j = i; j > 0; j--) {
      if(sorted[j] < sorted[j - 1]) {
        int temp = sorted[j];
        sorted[j] = sorted[j - 1];
        sorted[j - 1] = temp;
      }
      else
        break;
    }
  }
}
```

C++ implementations: time counting

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

const int LEN_BASE = 2000;
const int MAX = 10000;

void bubbleSort(const int unsorted[], int sorted[], int len);
void insertionSort(const int unsorted[], int sorted[], int len);
void setRN(int rn[], int len) {
  srand(time(nullptr));
  for(int i = 0; i < len; i++)
    rn[i] = rand() % MAX;
}

// continue to the next page
```

C++ implementations: time counting

```
int main()
{

  cout << "n Bubble Insertion\n";

  for(int expSeq = 0; expSeq < 10; expSeq++)
  {
    const int LEN = LEN_BASE * (expSeq + 1);
    int* rn = new int[LEN];
    int* sorted = new int[LEN];

    setRN(rn, LEN);
    cout << LEN << " ";

    // continue to the next page
```

C++ implementations: time counting

```
    clock_t startTime = clock();
    bubbleSort(rn, sorted, LEN);
    clock_t endTime = clock();
    cout << static_cast<float>(endTime - startTime) / CLOCKS_PER_SEC << " ";

    startTime = clock();
    insertionSort(rn, sorted, LEN);
    endTime = clock();
    cout << static_cast<float>(endTime - startTime) / CLOCKS_PER_SEC << "\n";

    delete [] sorted;
    delete [] rn;
  }

  return 0;
}
```

Python implementations: two functions

```python
def bubbleSort(unsorted, sorted, len):
  for i in range(len):
    sorted[i] = unsorted[i]

  for i in range(len - 1, 0, -1):
    for j in range(i):
      if sorted[j] > sorted[j + 1]:
        temp = sorted[j]
        sorted[j] = sorted[j + 1]
        sorted[j + 1] = temp
```

```python
def insertionSort(unsorted, sorted, len):
  for i in range(len):
    sorted[i] = unsorted[i]

  for i in range(len):
    for j in range(i, 0, -1):
      if sorted[j] < sorted[j - 1]:
        temp = sorted[j]
        sorted[j] = sorted[j - 1]
        sorted[j - 1] = temp
      else:
        break
```

Python implementations: time counting

```
import random
import time

LEN = 10000;
MAX = 10000;
BIN_CNT = 10;

def setRN(rn, len):
  for i in range(LEN):
    rn[i] = random.randrange(32767) % MAX

LEN_BASE = 2000
print("n Bubble Insertion")

# continue to the next page
```

Python implementations: time counting

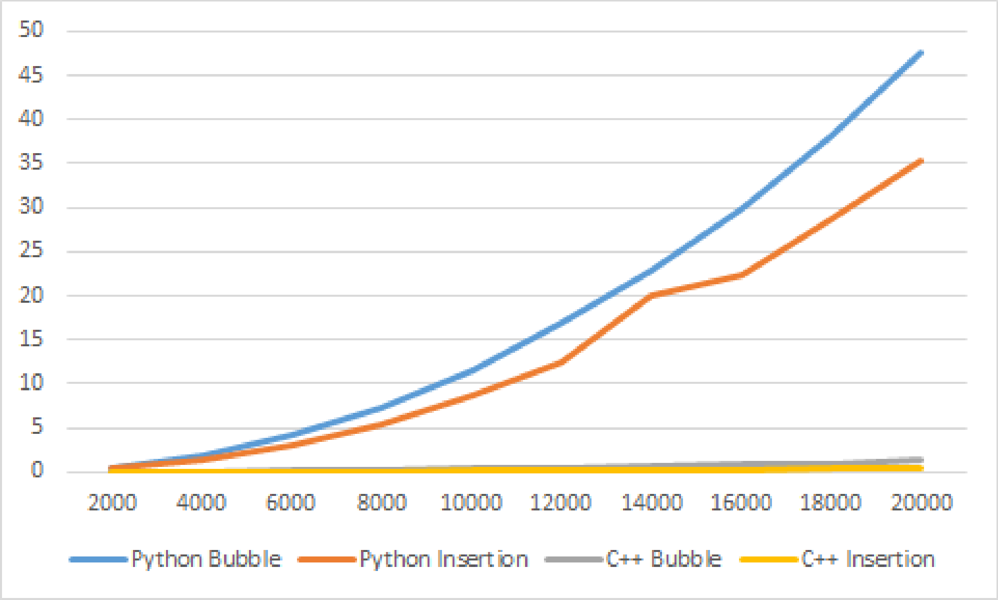
```
for expSeq in range(10):
  LEN = LEN_BASE * (expSeq + 1)
  rn = [0] * LEN
  sorted = [0] * LEN
  setRN(rn, LEN)
  print(LEN, end = " ")

  startTime = time.clock()
  bubbleSort(rn, sorted, LEN)
  endTime = time.clock()
  print(round((endTime - startTime) * 1000) / 1000, end = " ")

  startTime = time.clock()
  insertionSort(rn, sorted, LEN)
  endTime = time.clock()
  print(round((endTime - startTime) * 1000) / 1000)
```

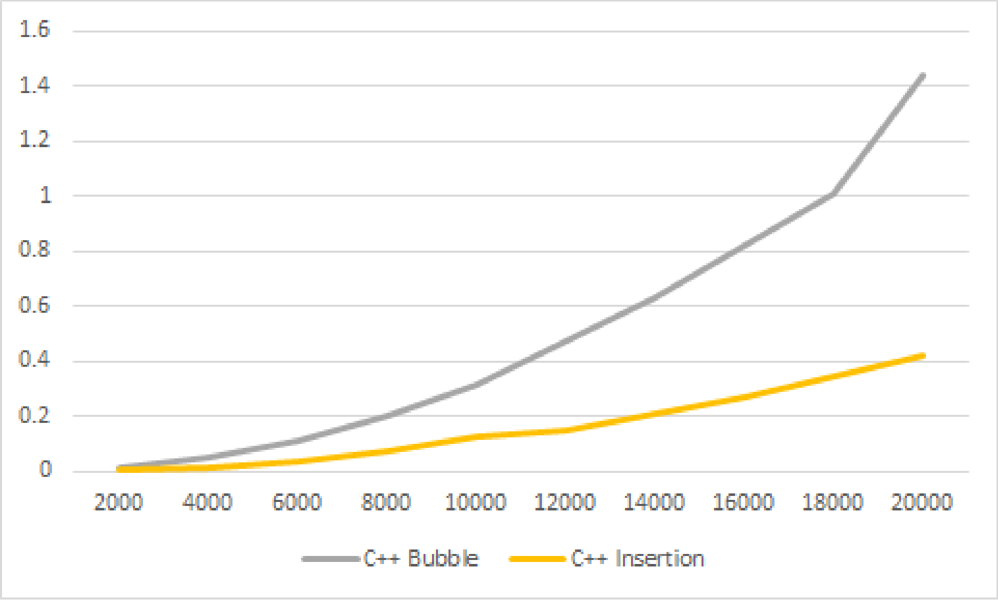
Comparisons

Number of integers	Python Bubble	Python Insertion	C++ Bubble	C++ Insertion	Ratio Bubble	Ratio Insertion
2000	0.470	0.339	0.013	0.004	36.15	84.75
4000	1.826	1.375	0.051	0.017	35.80	80.88
6000	4.162	3.106	0.113	0.038	36.83	81.74
8000	7.345	5.460	0.201	0.075	36.54	72.80
10000	11.598	8.750	0.317	0.126	36.59	69.44
12000	16.983	12.460	0.471	0.150	36.06	83.07
14000	22.787	19.960	0.632	0.208	36.06	95.96
16000	29.864	22.385	0.820	0.267	36.42	83.84
18000	38.174	28.737	1.010	0.348	37.80	82.58
20000	47.573	35.245	1.438	0.419	33.08	84.12

Comparisons

- C++ is indeed (much) faster!





“Old” and “new” languages

- C++ is older than Python.
- A new language typically has better design.
  - E.g., Indention vs. curly brackets.
- Nevertheless, both languages are still evolving.

Road map

- Python
- C++ vs. Python
- **The power of data structures**

The power of data structures

- In the next semester, you will take the course “Data Structures and Advanced Programming.”
  - To better manage and protect your data.
  - To improve the efficiency of your program.
  - To allow a higher complexity of your system.
- We already see some different data structures:
  - E.g., an adjacency matrix and an adjacency list.
- To motivate your study in the next semester, let’s use one example to illustrate the power of data structures.

Case study: makespan minimization

- n jobs should be allocated to m machines. It takes p_j hours to complete job j.
  - p_j is called the **processing time** of job j.
- When a machine is allocated several jobs, its **completion time** is the sum of all processing times of allocated jobs.
- We want to **minimize** the completion time of the machine whose completion time is the **latest**.
  - This is called "**makespan**" in the subject of scheduling.
  - The problem is called "makespan minimization among identical machines."

Heuristics for makespan minimization

- Makespan minimization among identical machines is NP-hard.
- Two well-known heuristic algorithms were proposed by Graham (1966, 1969).
  - Both algorithms are iterative and greedy.
- Algorithm 1:
  - Let the jobs be ordered in any way.
  - In each iteration, assign the next job to the machine that is **currently having the earliest completion time**.
- Algorithm 2 (longest processing time first, **LPT**):
  - Let the jobs be ordered in the **descending order of processing times**.
  - In each iteration, assign the next job to the machine that is currently having the earliest completion time.

Examples

- Suppose that we have ten jobs and three machines.
  - Processing times are 9, 4, 7, 1, 6, 3, 5, 4, 3, and 8.
- Algorithm 1:
  - Machine 1: 9, 5, 8 (total: 22).
  - Machine 2: 4, 1, 6, 3 (total: 14).
  - Machine 3: 7, 3, 4 (total: 14).
- Algorithm 2 (LPT):
  - Machine 1: 9, 4, 4 (total: 17).
  - Machine 2: 8, 5, 3, 1 (total: 17).
  - Machine 3: 7, 6, 3 (total: 16).
- For this example, LPT happens to find an optimal solution.

Worst-case performance guarantees

- While the two algorithms are simple, they also possess a great theoretical property: their **worst-case performance** is guaranteed.
- Let P be a minimization **problem**, I be an **instance** of P.
  - "Makespan minimization among identical machines" is a problem.
  - $n = 10$, $m = 3$, and $p = (9, 4, 7, 1, 6, 3, 5, 4, 3, 8)$ define an instance.
- For an instance I:
  - Let $z^{\text{OPT}}(I)$ be the objective value of an optimal solution.
  - Let $z^{\text{ALG}}(I)$ be the objective value of a solution obtained by the algorithm.

Worst-case performance guarantees

- An algorithm is a factor-α **approximation algorithm** if

$$\frac{z^{\text{ALG}}(I)}{z^{\text{OPT}}(I)} \leq \alpha \text{ for all } I.$$

  – α is called the **approximation factor** of the algorithm.
  – This must be true for **all** possible instances, including the weirdest instance in the world.
- For the two heuristic algorithms:
  – Algorithm 1 is a factor-2 approximation algorithm.
  – Algorithm 2 (LPT) is a factor-$\frac{4}{3}$ approximation algorithm.
  – The proofs are beyond the scope of this course.

Time complexity

- Having a worst-case performance guarantee is great, but how about **worst-case time complexity**?
- Algorithm 1:
  - Let the jobs be ordered in any way: **do nothing**.
  - In each iteration, assign the next job to the machine that is currently having the earliest completion time.
- Algorithm 2 (LPT):
  - Sort jobs in the descending order of processing times: $\boldsymbol{O(n \log n)}$.
  - In each iteration, assign the next job to the machine that is currently having the earliest completion time.
- Let's analyze the common step.

Time complexity: the common step

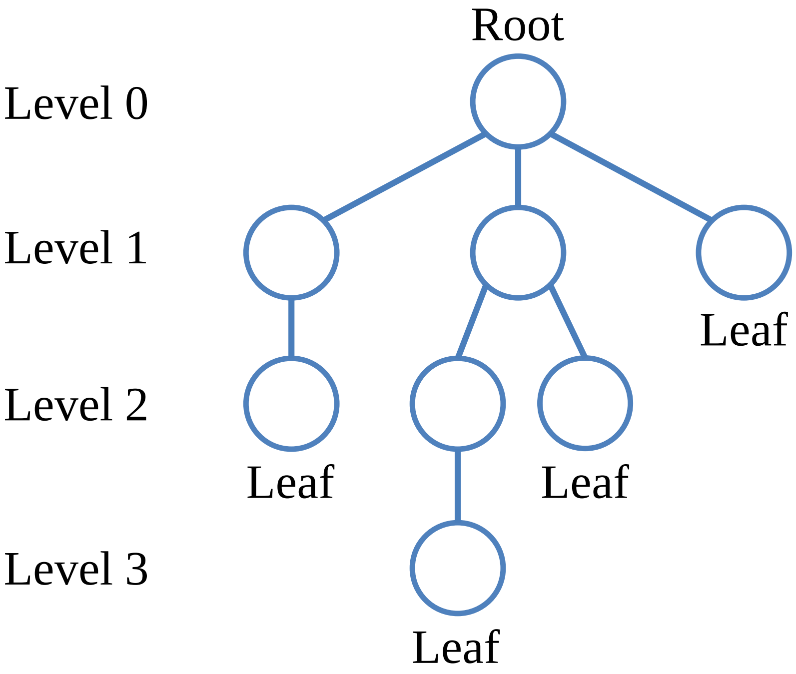
- The pseudocode:

> Let p be a vector of processing times of the n jobs.
> Initialize C_i to 0 for all $i = 1, \ldots, m$. // accumulated completion times
> ***for*** j from 1 to n
>   Find i^* such that $C_{i^*} \leq C_i$ for all $i = 1, \ldots, m$. // how to implement?
>   Assign job j to machine i^*; update C_{i^*} to $C_{i^*} + p_j$.

- Method A: **sort** all completion times to find a smallest one.
  - Sorting: $O(m \log m)$. The whole step: $O(nm \log m)$.
- Method B: do a **linear search** to find a smallest one.
  - Sorting: $O(m)$. The whole step: $O(nm)$.
- May we do better?

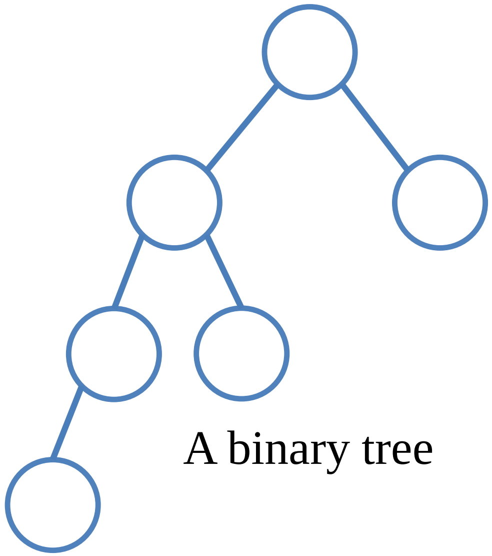
A min heap: preparation

- To further improve our algorithm, we introduce a data structure called **heap**.
  - There are **min heaps** and **max heaps**. Below we will introduce min heaps. Max heaps may be defined and used in a similar way.
- Let's start with a **tree**.
  - A tree is a graph with no cycle.
  - In a tree, we may specify a node to be the **root** and some nodes to be **leafs**. Others are **internal** nodes.
  - The root is at **level** 0; the root's neighbors are at level 1; the neighbors of the root's neighbors are at level 2, etc.

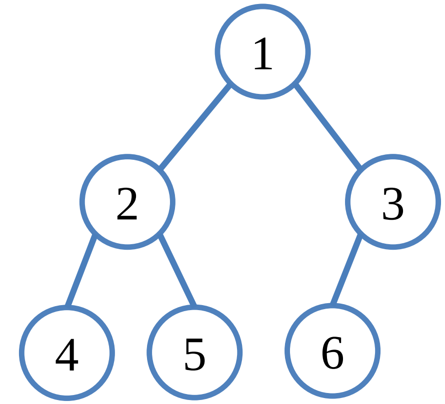


A min heap: preparation

- For two nodes are levels k and $k+1$, the one at level k is the **parent** and that at level $k+1$ is the **child**.
- A **binary tree** is a tree in which each node has at most two children.
  - Level k has at most 2^k nodes.
- A binary tree is **complete** if the existence of a level-k node implies that there are 2^{k-1} nodes in level $k-1$.
- For a complete binary tree, we may label each node level by level, from left to right.
  - Nodes in level k are labeled from 2^k to $2^{k+1}-1$.
- A complete binary tree of n nodes has **$\lceil \log n \rceil$ levels**.



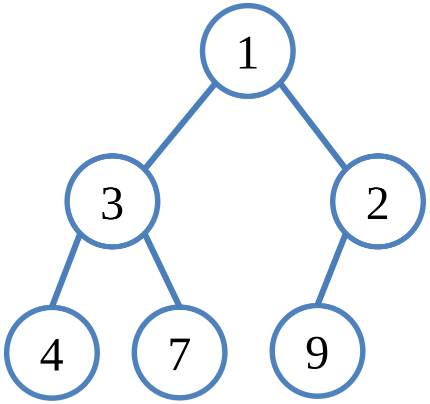
A binary tree

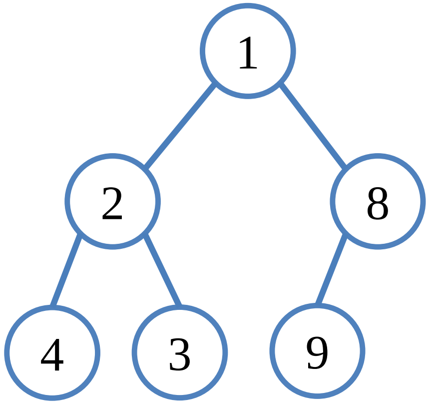


A complete binary tree

A min heap

- A **min heap** is a complete binary tree where a parent is **no greater than** any of its children.

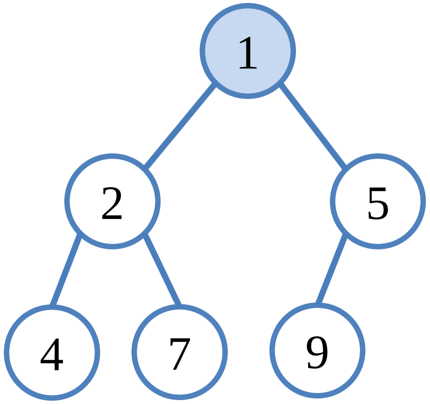




- For each **subtree**, the root contains the **minimum value** in the subtree.
  - The root of the whole tree contains the minimum value in the tree.
  - There is no restriction on values in different subtrees.

A min heap for completion times

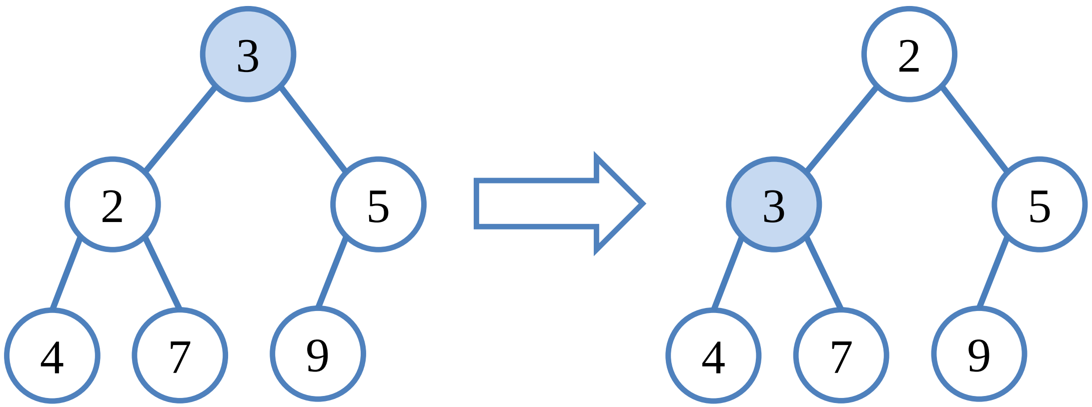
- Now, let's put the m completion times into a min heap.
- Find the **minimum completion time** is simple: Just look at the root.



- We then update that completion time by **adding a job's processing time** to it.
  - How to **update the tree** to make it still a min heap?

Keeping the tree as a min heap

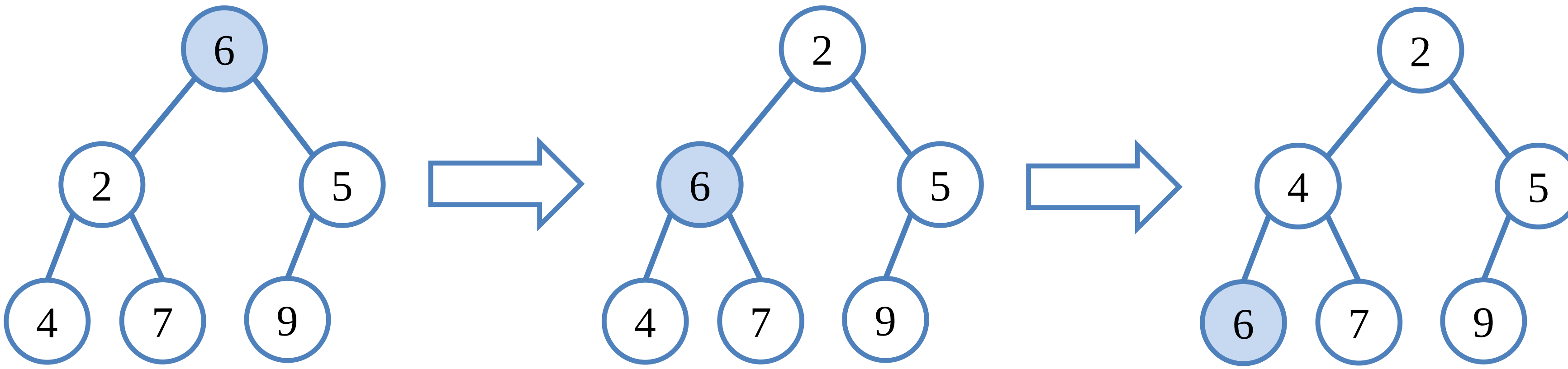
- Suppose that we add 2 into the minimum completion time. 1 becomes 3.
  - We then **exchange 3 with 2**, the **smaller** one of its children.
  - The resulting tree then becomes a min heap.



- Why exchanging 3 with its smaller child?

Keeping the tree as a min heap

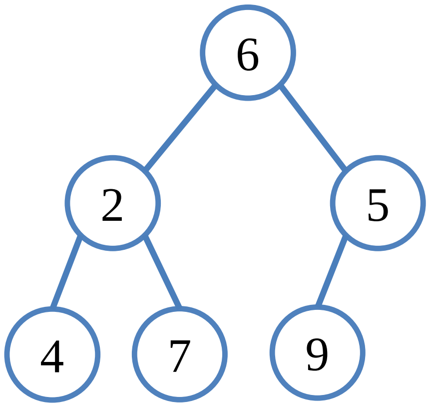
- Suppose that we add 5 into the minimum completion time. 1 becomes 6.
  - We then exchange 6 with 2, the **smaller** one of its children.
  - We **keep doing so** if needed.

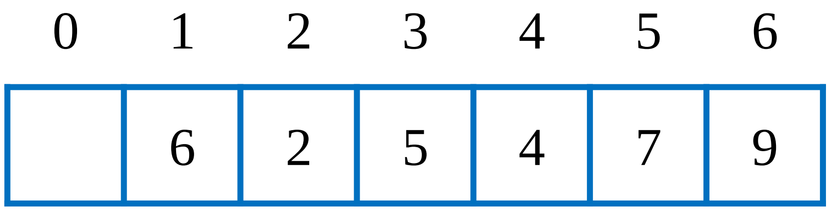


- To do an adjustment, the maximum number of exchange is roughly $\log m$.
- Doing this n **times** takes only $O(n \log m)$.

Implementing a min heap

- An n-nodes min heap may be easily **implemented** with a size-$(n + 1)$ **array**.
  - Intentionally leave the 0th element unused.
  - Put the value in node i in the **ith element** of the array.





- For node i, its **children** are **nodes $2i$ and $2i + 1$**.
  - Just compare **`array[i]`** with **`array[2 * i]`** and **`array[2 * i + 1]`**.

Time complexity: the common step

Let p be a vector of processing times of the n jobs.
Initialize C_i to 0 for all $i = 1, \ldots, m$. // accumulated completion times
for *j* from 1 to *n*
  Find i^* such that $C_{i^*} \leq C_i$ for all $i = 1, \ldots, m$. // how to implement?
  Assign job j to machine i^*; update C_{i^*} to $C_{i^*} + p_j$.

- One algorithm, three methods:
  - Method A: **sort** to find a smallest one: $O(nm \log m)$.
  - Method B: **linear search** to find a smallest one: $O(nm)$.
  - Method C: use a **min heap** to find a smallest one: $O(n \log m)$.
- A and B are different in **algorithms**; B and C are different in **data structures**.
  - Both B and C use a size-$O(m)$ array. Only the way of storing values differ.

Conclusions

- Regarding time complexity:
  - To complete a task, different algorithms may perform differently.
  - To realize an algorithm, different data structures may perform differently.
- Data structures affect more than time complexity:
  - Space complexity.
  - Flexibility.
  - Safety.
- This is why we need courses for data structures and algorithms!