第一讲
算法导论
Lecture 1
Introduction to Algorithm

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

编程模式
Programming Paradigm

- 显式编程（Explicit Programming）

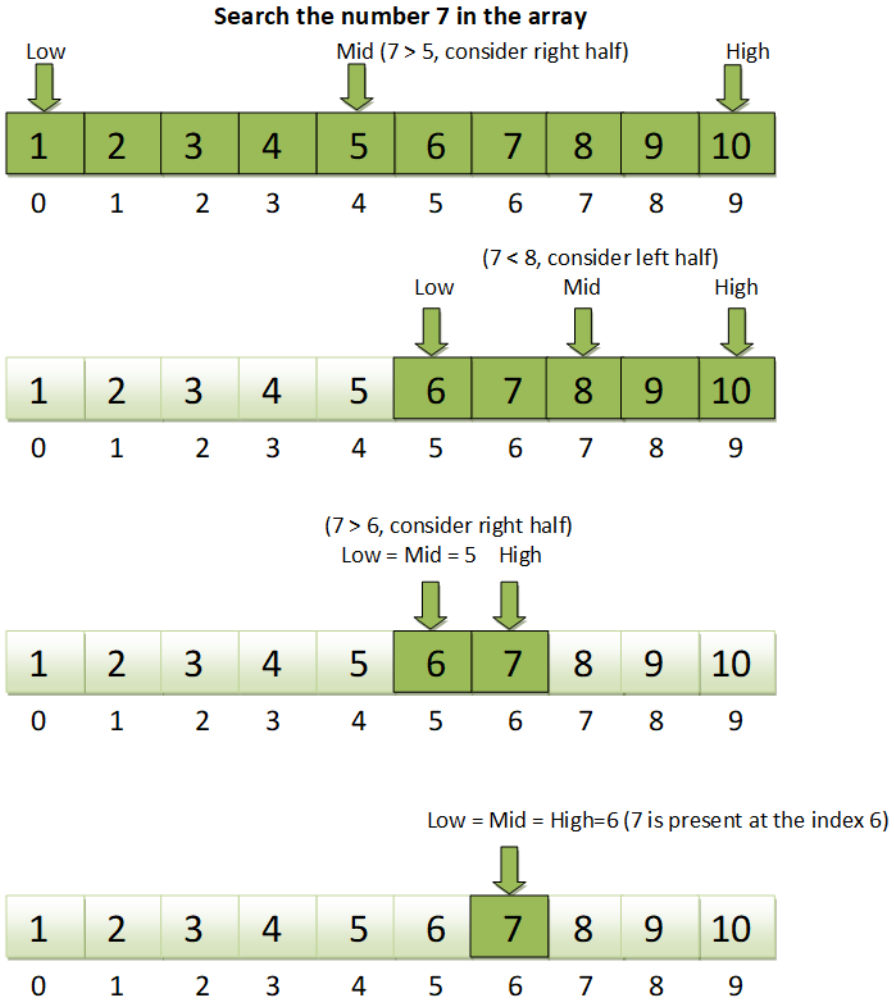
  指定算法的每一步骤

  Specify each step of the algorithm

  - 以查找二分查找算法为例

    Take binary search algorithms for example

```
BinarySearch(list[], min, max, key)
while min ≤ max do
   mid = (max+min) / 2
   if list[mid] >key then
      max = mid-1
   else if list[mid] <key then
      min = mid+1
   else
      return  mid
   end if
end while
return  false
```



编程模式
Programming Paradigm

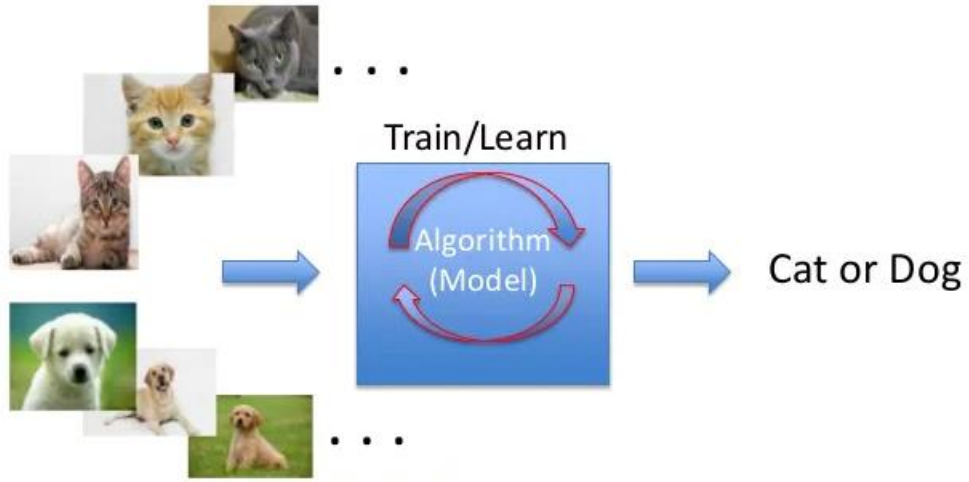
- 隐式编程（Implicit Programming）

  由模型或算法自动学习解决问题的步骤，机器学习常用

  The procedures to solve the problem are learned by the model or the generic algorithm via data-driving.

  - 以图像分类为例子

    Exemplified by image classification



算法定义
Definition

- 算法是为了解决定义良好的问题，而制定的定义良好的步骤，或者说，计算序列；此序列将输入转化为输出，此输出正为解决问题所需。

  An algorithm is a well-defined computational procedure, or a sequence of computational steps, that transform the input into the output, which are cater to the problem itself.

- 正确性是算法的一个重要特征；通常我们假设算法是正确的，即对任何符合问题定义的输入，都能在合适的时机停止并产生正确的输出；不正确的算法一般不能处理某些特殊情况

  Accuracy is a critical characteristic of algorithm; Generally, we always assume the underlying algorithm is correct. That is, for any input in accordance with the problem description, the algorithm will halt at a proper occasion with proper output. For faulty algorithm, it can fail to copy some extreme situation.

正确性证明
Proof of correctness

- 程序语言分为命令式语言，如C，Java等，及函数式语言，如Erlang，Lisp等；函数式编程语言与命令式编程语言的一个最大的不同是变量不变性。

  The programming languages can be divided into imperative language, for example, C, Java, etc., and functional language, for example, Erlang, Lisp. The most eminent difference between imperative language and functional language is the invariance of variables.

- 在命令式语言如C中，赋值操作会导致一个变量的值的改变，这是合理的，因为其中隐含了时序，如$x = x + 1$。但形式化的逻辑验证中不太容易捕捉到这种时序，形而上地说，$x = x + 1$从数学上来讲是不对的。

  In imperative language such as C, the assignment will cause the original value of a variable updated to a new value. This is valid due to the implicit time sequential operations, for example, $x = x + 1$. However, in formal validation, it is hard to catch the sequential implication. So, from the math perspective, $x = x + 1$ is incorrect.

正确性证明
Proof of correctness

- 回忆：数学归纳法，推理与证明中的常见方法。应用数学归纳法主要分三步：
  - 第一步：证明当n取第一个值n_0时结论正确;
  - 第二步：假设当$n = k$（k属于正整数且大于或等于n_0）时结论正确（归纳假设），写出关于$n = k$时的结论表达;
  - 第三步：证明当$n = k + 1$时结论也正确。

- Recall the mathematical induction method, which is very common in proof and reasoning. The application of MI is generally in three steps:
  - Step 1: Prove the conclusion for initial value $n = n_0$;
  - Step 2: Assume the conclusion for $n = k$ (k is an integer and greater than n_0), this is called induction assumption; and express the conclusion for $n = k$;
  - Step 3: Prove the conclusion for $n = k + 1$.

正确性证明
Proof of correctness

- 示例：证明首项为a_1，公差为d的等差数列的通项和公式为$S_n = a_1 n + \frac{n(n-1)}{2} d$。

  Example: Proof the sum of an arithmetic sequence with the first term a_1 and common difference d has the form $S_n = a_1 n + \frac{n(n-1)}{2} d$

  - 第一步：$n = 1$，则$S_1 = a_1 1 + \frac{1(1-1)}{2} d = a_1$。因为仅有一项，显然成立；

    Step one: if $n = 1$, then $S_1 = a_1 1 + \frac{1(1-1)}{2} d = a_1$. It is manifest the sum of single term conveys itself.

  - 第二步：假设当$n = k$时公式成立，即$S_k = a_1 k + \frac{k(k-1)}{2} d$；

    Step two: suppose the formula holds for $n = k$, that's $S_k = a_1 k + \frac{k(k-1)}{2} d$

正确性证明
Proof of correctness

- 第三步：则当$n = k + 1$时，$S_{k+1} = S_k + a_{k+1}$。注意到$a_{k+1} = a_1 + k \cdot d$，则

$$S_{k+1} = a_1 k + \frac{k(k-1)}{2} d + a_1 + k \cdot d = a_1(k+1) + \frac{k(k-1)+2k}{2} d$$
$$= a_1(k+1) + \frac{(k+1)\big((k+1)-1\big)}{2} d$$

Step three: when $n = k + 1$, we have $S_{k+1} = S_k + a_{k+1}$. Note $a_{k+1} = a_1 + k \cdot d$, then we further have:

$$S_{k+1} = a_1 k + \frac{k(k-1)}{2} d + a_1 + k \cdot d = a_1(k+1) + \frac{k(k-1)+2k}{2} d$$
$$= a_1(k+1) + \frac{(k+1)\big((k+1)-1\big)}{2} d$$

正确性证明
Proof of correctness

- 循环不变式：有一大类算法，均可以用循环不变式证明其正确性，其证明过程类似于数学归纳法

  Loop invariant: there are algorithms belonging to the class of which the correctness can be verified by loop invariant. Such a process resembles the mathematical induction.

- 初始（第一步）：在迭代或循环初始，算法作用于输入时，操作正确

  Initialization (The first step): By applying the initial iteration of loop to the input, the state is manifested as true.

- 保持（第二步）：若在中间次的循环前处于正确的状态，则在下次循环前（即当次循环结束后），仍处于正确的状态。

  Maintenance (The second step): It can be demonstrated that if it is true before an iteration of the loop, it remains true after it.

正确性证明
Proof of correctness

- 终止（第三步）：当算法终止时，其输出为可验证的期望输出

  Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

- 以插入排序来说明上述方法

  We demonstrate the above method via insertion-sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

正确性证明
Proof of correctness

- 我们对最外层循环使用循环不变性来说明算法设计的正确性

  We prove the correctness of the designed algorithm via loop invariance for the outer loop.

- 初始：当$J = 2$时，数组$A[1 \quad \cdots \quad j-1]$显然仅有一个元素$A[1]$；一个元素显然无所谓顺序问题，即在循环初始阶段，结论是正确的。

  Initialization: when $J = 2$, it occurs that $A[1 \quad \cdots \quad j-1]$ is just with one element $A[1]$. The order for one element is trivial, interpreted from any perspectives. Nonetheless, the conclusion holds.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

正确性证明
Proof of correctness

- 保持：在保持阶段，首先，$A[j]$赋值给变量key后，不用担心覆盖问题；注意内层循环，其所做工作是从后向前，将$A[j-1]$、$A[j-2]$、$A[j-3]$等等与$A[j]$分别比较，若其比key大，则依次后移。直到找到合适的位置插入$A[j]$，此时，$A[i] \leq A[j] < A[i+1]$，数组亦是有序的。

  Maintenance: during the maintenance stage, the assignment of value of $A[j]$ to the variable key at the first place eliminates the overwrite concern of $A[j]$. Then, notably, for the inner loop, which backwardly compare $A[j-1]$, $A[j-2]$, $A[j-3]$ etc. with $A[j]$ respectively, to shift them one slot back if any one them is greater than key. Upon find the appropriate position, then $A[j]$ will be inserted. Now since $A[i] \leq A[j] < A[i+1]$. So, the array is sorted before the next outer iteration.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

正确性证明
Proof of correctness

- 终止：若$A.length == n$，则终止时$j = n + 1$。在循环不变式的标示中，$j = n + 1$对应的子数组为$A[1 \quad \cdots \quad n]$，但此数组即整个数组。由于在保持阶段已经说明，在新的一轮循环开始前，结论正确，即$A[1 \quad \cdots \quad n]$是排序好的数组，而$j = n + 1$又能恰当终止，即算法是正确的。

  Termination: if $A.length == n$, when the algorithm terminates, we have $j = n + 1$. During the wording of loop invariant, the corresponding subarray for $j = n + 1$ is $A[1 \quad \cdots \quad n]$. But this is already the overall array. During the maintenance stage, it is already shown that $A[1 \quad \cdots \quad n]$ is already sorted before the new iteration. Plus, the algorithm can terminate properly, so the algorithm is correct.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

正确性证明
Proof of correctness

- 许多算法的证明可能没有固定的范式，需要具体问题具体分析，依靠其它不变性来达到目的。

  Not all algorithm follow some paradigm to verify their correctness. It might be other invariances to be sought for the proof of correctness

- 考虑取火柴游戏：总共有15根火柴，A与B两名游戏参与着轮流取；每次最少取1根，最多取三根，不可以不取；最后取者为输。问必胜策略是什么？

  Consider the matchstick game, which starts with 15 matchstick. 2 participants take in turns to fetch the sticks. The rule states that each turn at least 1 matchstick and at most 3 sticks must be retrieved. No action is not allowed. The question is what is the always win strategy or algorithm.

正确性证明
Proof of correctness

- 最后取者为输，即一定要为对手剩余一根火柴，按游戏规则，对手不得不取，因此判输。

  The last fetcher loses the game, which means the remaining matchstick for the opponent must be one. According to the rule of the game, which enforce the opponent to act and lost the game.

正确性证明
Proof of correctness

- 必胜策略意味着取火柴过程必须出现不变性因素，维持此种不变量，以达到控制游戏，可视为算法执行的流程。因此要分析出这种不变性。单纯看每个选手的动作，似乎具有随机性，但同时看两个参赛者的动作，便可以看出此种不变性。

  The always-win strategy indicates some invariance must be sought during the stick fetching. By maintaining such kind of invariance, to have the game well-controlled as proceeds. This process can be analogous to the algorithm execution. Hence, to find out the invariance is critical for correctness proof. If solely inspect the actions of each opponent, they render some kind of randomness. However, if jointly study the actions of two participants, to figure out the invariance is conceivable.

正确性证明
Proof of correctness

- 同时看两个参赛者的动作，因为每个选手最少必须取1根，最多只能取3根，则一轮结束后，最少有2根火柴被取走，最多有6根火柴最取走；那么就要看，2至6这四个火柴中，哪个最有可能成为不变量。显然2不行，因为只要先手取2根及以上，则此轮结果不会只取2根；同理3也不行。5不行，因为先手取1根的话，无论后手怎么取，均不能达到；同理6也不行。4是否具有不变量潜质呢？答案是肯定的。先手取1，则后手取3；先手取2，则后手取2；先手取3，则后手取1。

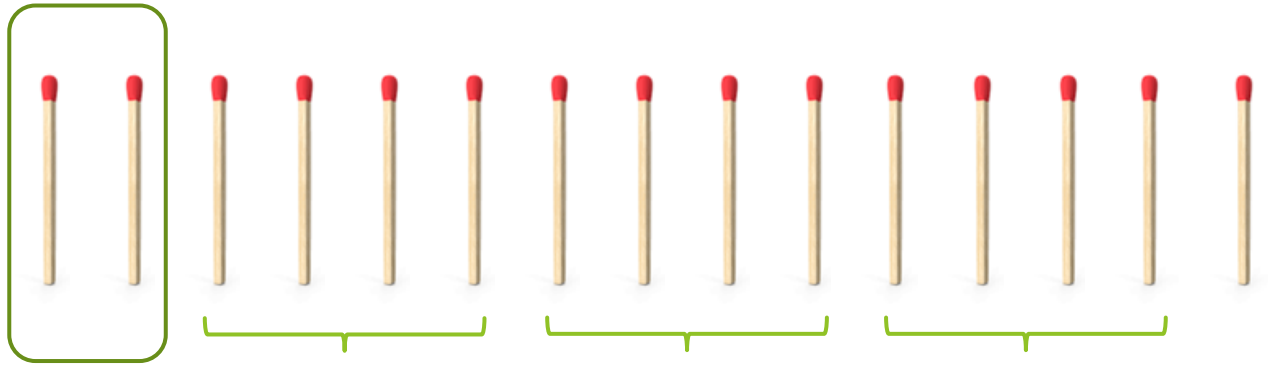
  To jointly study the actions of two participants, due to the fact each time at least 1 matchstick must be retrieved whilst at most 3 matchsticks must be fetched, therefore, upon one round finishes, at least 2 sticks have been removed and at most 6 sticks have been pulled out. So, it is necessary to verify the numbers from 2 to 6, which could be candidate for invariance. Obviously 2 is not eligible. Once the first participant fetches more than 2, then 2 is un-attainable. The same argument applies to 3. 5 is not eligible as well, if the first participant only fetches 1 stick. The same reasoning applies to 6. Let's check the number 4. The answer is positive. If the first participant fetches 1, the second can fetch 3 for compensation. If the first participant fetches 2, then second 2 for compensation. Then 3 and 1 respectively.

正确性证明
Proof of correctness

- 通过上述分析，在每轮取中，后手可以控制以不变量的形式取火柴（可以类比于循环不变量）；因此必胜策略意味着成为每轮循环的后手；不过由于要剩余1根火柴，所以要合理地构造初始状态，以便成为后手并且恰余1根。分析可知，一旦剩余根数为$4n+1$，则游戏便进入可控的循环不变状态。显然，若玩家A先取2走两根，并且保持循环不变性，则其必赢。

  The above analysis reveals that in each round, the second player can tactically control the way of fetching to maintain some invariance (analogous to loop invariant). So, the always win strategy means to be the next turn in each round. Considering the remaining matchstick must be 1, it is necessary to construct the initial state so as to satisfy the conditions mentioned above. As aforementioned, once the remaining matchsticks becomes $4n+1$, the game can fall into loop invariant state. So, it is obvious if the player A acts first and fetches 2 sticks, and intents to maintain the invariance, he dooms to win.

正确性证明
Proof of correctness



时间复杂度
Time Complexity

- 对于常见的算法，我们通常要分析算法的时间复杂度与空间复杂度。一般对空间复杂度的分析，整体上可能会相对明确，特别是当操作可以是原地操作时。当然对于递归操作，可能需要需要额外的空间存储栈上变量，不太好估。一般情况上，比如算法运行需要多少内存，是可以有一个大概的估计的。对于时间复杂度，可能需要逐语句分析了。但时间复杂度依赖于算法的执行路径，有时候，理想情况与最差情况相差会很多，所以大多数时候，我们只需要关注平均情况即可。

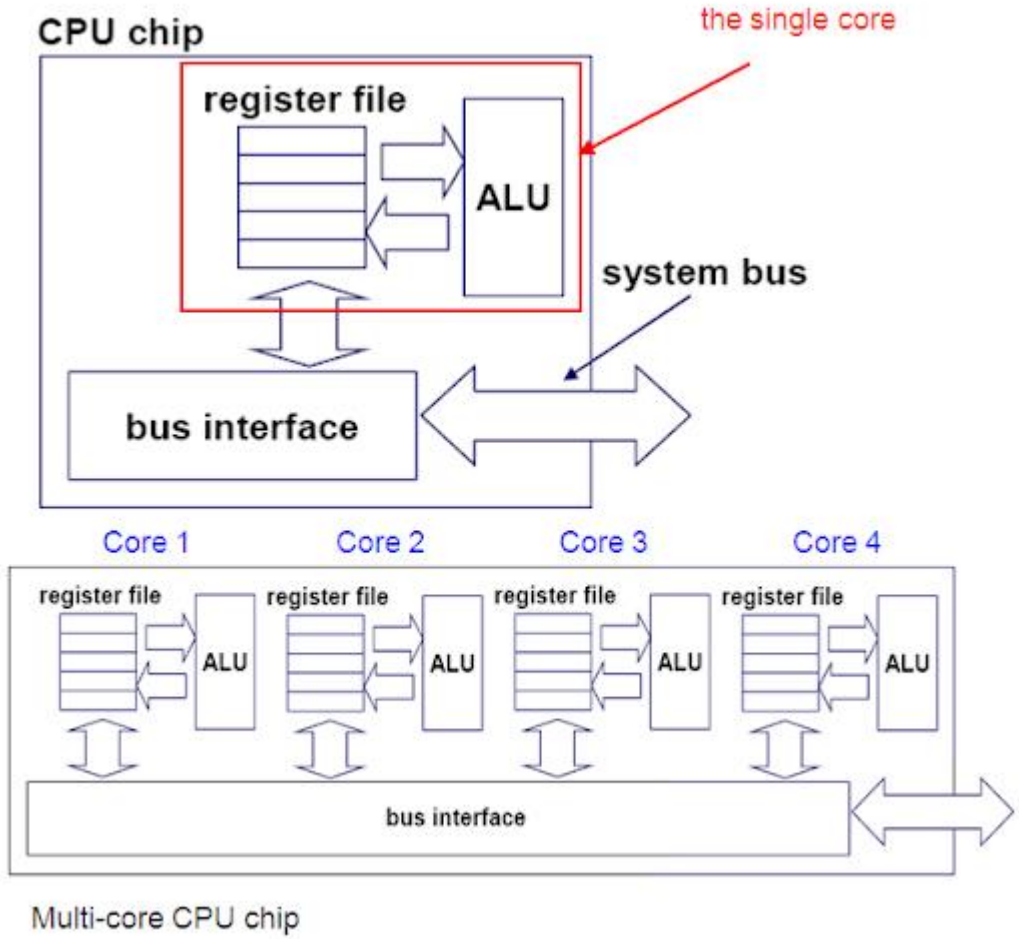
  For the commonly-used algorithms, in most cases we should have a sense of the time complexity and space complexity. For the space complexity, it might be self-evident usually, especially the manipulations involved in the procedures of the algorithm are in-place operations. Of course, for some algorithm such as recursion, because more memories are required to store the stack variables, the estimation can become hard. But generally, how many memories shall be occupied by the algorithm can be estimated. However, the time complexity is up to the analyze of code piece by piece. But the time complexity relies on the path of execution, which renders huge difference between the ideal case and worst case. Most of the time, we just care about the average case.

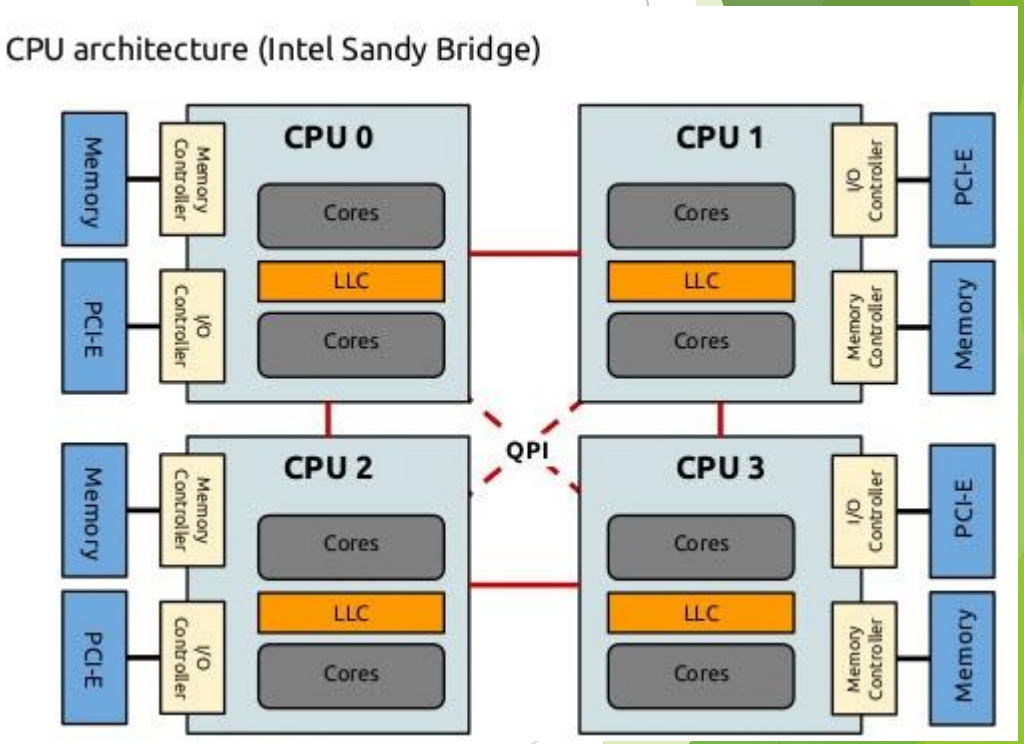
时间复杂度
Time Complexity

- 在上面关于分析的论述中，我们都假设一种单处理器、随机内存访问的计算模型。这个假设是必要的，比如，假设是多处理器，非均匀内存访问的计算模型，可能为了处理器的亲和性，对单一数据也需要多个内存拷贝，这个内存需求实际上会加倍，而由于多处理器，处理时间可能会减半。这样会增加分析的复杂性。另外，算法分析也和指令集相关，对于某些操作，复杂指令系统可能通过指令完成，但简单指令系统可能通过微码完成。

  The above explanation about algorithm analysis assumes a single central processing unit and random-access memory model of computation. This assumption is necessary. For example, considering a multi-processor, non-uniform memory access model of computation, the same data might have several copies considering the affinity of CPU, which increases the demand on memory. Instead, the multiple processor will cut down the execution time of the algorithm. Notably, algorithm analysis should also put the instruction set into consideration. For some operations, complex instruction set might provide the instruction execute on the hardware level, whilst simple instruction set has to simulate the execution via microcode at software level.

时间复杂度
Time Complexity





时间复杂度
Time Complexity

- 一般来说，时间复杂度与输入的规模有关。对于同一个算法，处理一个输入与处理一亿个输入，花费的时间一般是很不相同的；对于处理的数字而言，处理一个二进制位数为10的，与处理一个二进制位数为100的，花费的时间一般也是很不相同的。无论是输入的数字的个数，还是输入数字的大小，均可视为输入数字的规模。规模越大，耗时越多。对于算法分析而言，我们需要大概地定量刻画这种相关性，即输入与耗时而言，是线性相关的关系，还是多项式关系，抑或指数关系的。

  Generally, the time complexity is closely related to the scale of input. For the same algorithm, the time consuming on an input with a single value is almost surely different from the processing on an input with 100 million elements. Further, to process an input with 10 digits is practically less costly than an input with 100 digits. No matter the number of the input or digits of the input, it is all related to the scale of the input. Higher scale incurs greater time complexity. For algorithm analysis, we need at least an imprecise quantification of the relationship. For example, they are linearly corelated, or in a polynomial relation, or of exponential relationship.

时间复杂度
Time Complexity

- 在下面的分析中，我们假设单次执行一行（假设为第i行）伪代码的时间是一定的，记为c_i。以插入排序为例子，我们来分析该算法的时间复杂性。

  During the algorithm analysis below, we assume that the amount of time to execute each line of the pseudo-code is constant (c_i for the i^{th} line). We exemplify this by using insertion-sort algorithm

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

```
void insertion_sort(vector<int>& v)
{
    for (int i = 1; i < v.size(); i++)
    {
        auto key = v[i];
        int j = i - 1;

        for (; j >= 0 && v[j] > key; j --)
        {
            v[j + 1] = v[j];
        }
        v[j + 1] = key;
    }
}
```

时间复杂度
Time Complexity

$$T(n)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- 下面我们先分析该算法的最好情况与最差情况下时间复杂度。最好情况显然是当数组已经排序时，最差情况是初始数组逆序排列时。

  Now we analyze the time complexity for in the ideal case and worst case. The ideal case is array is already sorted and the worst case is array is already sorted.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_1 + c_2 + c_4 + c_5 + c_8)$$

时间复杂度
Time Complexity

$$
\begin{aligned}
&T(n)\\
&= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right)\\
&+ c_8(n-1)\\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)\cdot n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)\cdot n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

- 一般说来，我们对算法的复杂度的最差情况要有一个估计，特别是涉及到控制的一些算法。考虑平均情况，其时间复杂度大概是最差情况的四分之一，有时，就用最差情况的时间复杂度作为算法耗时的度量。

  Generally, it demands an estimate of the time complexity of the algorithm, especially for in the control domain. Considering the average case, in which the time consuming is about a quarter of the worst case. Sometimes, the worst case is adopted as the time cost measurement.

时间复杂度
Time Complexity

- 时间复杂度与输入规模的关系有时又称时间增长率，或增长的量级，用Θ表示。通常情况下，仅考虑时间表达式的最高次项；当谈及量级时，一般不考虑系数。综上，对于插入排序算法而言，其时间复杂度为即$\Theta(n^2)$。

  The time complexity correlating input scale is sometimes called rate growth, or order of growth, denoted by Θ. Generally, time complexity only need take the highest order term into consideration. And when talk about order, the coefficients are usually ignored. For example, the time complexity for insertion-sort is denoted as $\Theta(n^2)$.

习题
Problems

- 描述算法都有哪些特征

  Describe the characteristics of algorithms

- 将所示的插入排序伪代码用Python或C/C++实现。

  Implement the insertion-sort in Python or C/C++