INT102

> 内容完全来自于**LearningMall**的PPT。仅供学习和交流使用。任何机构或个人不得将其用于商业用途或未经授权的传播。如需引用或分享，请注明出处并保留此声明。
>
> GITHUB REPOSITORY
>
> VISIT NOTES AUTHOR'S PAGE CLICK ME

CPT102 **Data Structures and Algorithms** 数据结构和算法

算法基础与问题求解 Algorithmic Foundations And Problem Solving

■ [Week 1 Lecture 1]

> Week 1 Lecture 1 + Week 1 Lecture 2 (Tutorial)

[0] 课程情况

教师信息

Teacher Name	Office Room	Email	Office Hour
Dr. Jia Wang (M. Leader)	SD537	jia.wang02@xjtlu.edu.cn	14:00-16:00, Tuesday
Dr. Yushi Li	SD561	yushi.li@xjtlu.edu.cn	14:00-16:00, Tuesday
Dr. Pengfei Fan	SD559	pengfei.fan@xjtlu.edu.cn	14:00-16:00, Tuesday

学分组成

评估项	占比	其他
Assessment 1	10%	week 6 - week 7
Assessment 2	10%	week 12 – week 13
Final Examination	80%	Date TBA.

总内容

Methodology/Problems	Asymptotic Idea 渐进思想	Brute Force 蛮力法（暴力）	Divide & Conquer 分治法	Dynamic Programming （动态规划）	Greedy 贪心	Space/Time	Branch & Bound 分支界定法	Complexity Theory 复杂性理论
Efficiency 效率	Big-O							
Sorting 排序		Selection/Bubble/insertion	Mergesort		Count sorting			
Searching 搜索			Binary-searching					
String Matching 串匹配						Horspool algorithm		
Graph 图		DFS/BFS		Bellman-ford/Warshal/Assembly-line	MST/Dijkstra’s/Shortest path		Traveling salesman, Job assignment	
Complexity 复杂性								P/NP

程序 = 数据结构 + 算法

[1] 伪代码简述

> 伪代码（Pseudo）

```
p = 1 # 赋值
______________________________
for i = 1 to n do # 循环
    p = p * x
______________________________
for i = 1 to n do # 循环 2
begin
    statement
end
______________________________
output p # 输出
______________________________
if p < 0 then   # if 语句
    output -p
______________________________
while a > 10 do # while 语句
    statement
______________________________
while condition do # while 语句2
begin
    a = ask for a number # 输入数字
end
______________________________
repeat          # repeat语句
    statement
until a<0
______________________________
```

小练习

1. Find the product of all integers in the interval [x, y], assuming that x and y are both integers

> 寻找连续区间$[x,y]$的乘积。例如 $[4,10] = 4 \times 5 \times \ldots \times 10$

```
sum = 1
for i = x to y do
begin
    sum *= i
end
# sum *= i 即 sum = sum * i； 同时begin和end可省去（无歧义）；
```

2. List all factors of a given positive integer x

> 列出所有`x`的因数

```
for i = 1 to x do
    if x % i == 0 then
        output i
```

能否把他们转为`while`和`repeat`呢？尝试一下吧。

为什么我们需要更好的算法？接下来是一个直观的例子。

> 假设计算机的速度是：5 次操作/s
>
> 现在有一个算法的速度如下：比较 n 个数，需要 n^2 次操作
>
> 因此，处理 5 个数据需要 $5 \times 5 = 25$ 次操作，也就是 5 s
>
> 假设把计算机速度提高100倍，也就是 500 次操作 / s
>
> 如果不优化算法，在相同时间 5 s内，只能完成 $\sqrt{5 \times 500} = 50$个数据。
>
> 可见，即便把计算机速度提高 100 倍，如果不改变算法，也只能处理比原来多 10 倍的数据。

[2] 多项式

涉及n的幂的函数（例如，n，$nlog(n)$，n^2，称为**多项式函数**）与n作为指数的函数（例如，2^n，称为**指数函数**）之间存在巨大差异。简而言之就是n在上面还是下面差别很大。

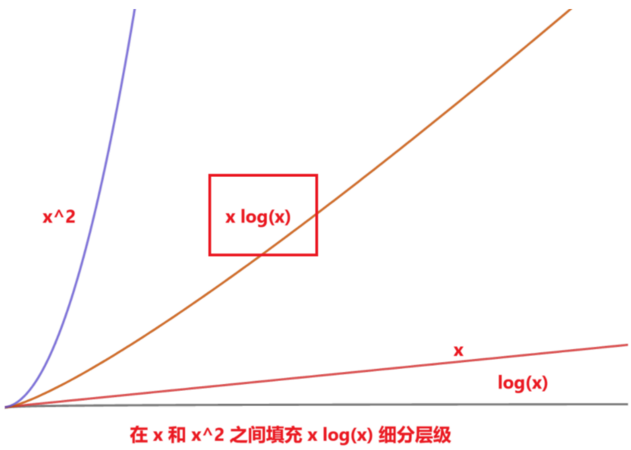
> 在算法分析和时间复杂度中，$log(n)$ 通常表示的是以 2 为底的对数，即$log_2(n)$。

> 多项式函数 polynomial functions
> 指数函数 exponential functions

对于多项式函数而言，齐次幂的更具有可比性。例如 $f_3(n) = n^2 - 3^n + 6$ 和 $f_4(n) = 2n^2$

我们可以定义一个函数的**层级（Hierarchy）**，每个函数的数量级都比其前一个函数要大。对于较大的数据量而言，层级小的函数所用次数少：c（常数）$<\ log(n)\ <\ n <\ n^2\ <\ ...$ 。

我们可以通过在 n 和 n^2 之间插入 $nlog(n)$，在 n^2 和 n^3 之间插入 $n^2log(n)$ 等等来进一步细化这个层级。



[3] 大O符号

现在，当我们有一个函数时，我们可以将这个函数归类到**层级**中的某个函数。

> 例如：$f(n) = 2n^3 + 5n^2 + 4n + 7 \to$ 归类到 n^3 因为主要受 n^3 的影响。

这个概念通过大 `o` 符号（Big-O notation） 表示。

$$f(n) = O(\ g(n)\ )$$

其中 $g(n) = f(n)$ **的最高次项(除常数)**。例如$2n^3 = O(n^3)$、$n^3 + n^2 = O(n^3)$

左右两侧的函数被认为具有**相同数量级(Same order of magnitude)**。

正式定义

$f(n) = O(g(n))$: 存在一个常数 c 和 n_0，使得对于所有 $n > n_0$，$f(n) \le c * g(n)$。

> 这个难以理解没关系，会用就可以。例如函数 $f(n) = 2n^3$ ，我们可以找到一个常数 c 比如 $c = 2 + 1 = 3$ ，和一个 n_0 比如 1，使得对于所有 $n > 1$，$2n^3$ 始终比 $3n^3$ 小。这时候，我们就可以说 $f(n) = O(n^3)$。
>
> 如果说不是那么明显的话，例如函数是$f(n) = n^6 + n^5 + ... + n$，我们就可以找到一个常数，例如 $c = 1 + 1 = 2$，然后再找一个非常大的 $n = 1,000,000$。使得对于所有 $n > 1,000,000$ 总有 $f(n) = n^6 + ... + n < 2n^6$
>
> ```
> >>> c = 1000000
> >>> pow(c,5)+pow(c,4)+pow(c,3)+pow(c,2)+pow(c,1) < pow(c,6)
> True
> # 1000001000001000001000001000000 < 1000000000000000000000000000000000000
> ```
>
> 简而言之，我们总可以找到一个 $c = f(n)$ 的最高次幂项的系数 $+1$，以及一个 $n \to \infty$ 满足：$n > n_0$总有 $f(n) \le c * g(n)$。

通常我们只关心**渐近时间复杂度(Asymptotic time complexity)**，也就是说，**当 n 很大时**:

$$O(\ log(n)\ ) < O(\sqrt{n}) < O(n) < O(nlogn) < O(n^2) < O(2^n)$$

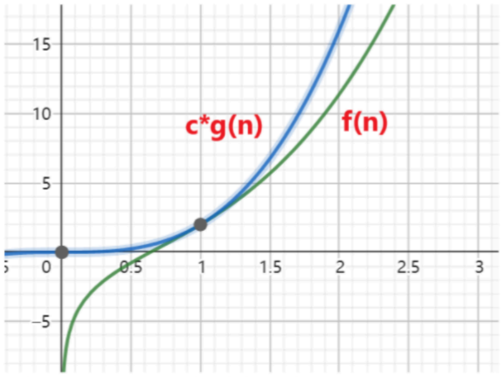
PPT习题

证明 $2n^2 + 4n$ **是** $O(n^2)$

- 因为对于所有 $n_0 > 4$，$c = 3$，$n \le n^2$， 我们有 $2n^2 + 4n \le 3n^2$ 对于所有$n > n_0$
- 因此，根据定义，$2n^2 + 4n$ 是 $O(n^2)$。

证明 $n^3 + n^2 logn + n$ **是** $O(n^3)$

- 因为对于所有 $n_0 > 1, c = 2$，我们有$n^3 + n^2 log(n) + n \le 2n^3$，对于所有$n > n_0$
- 因此，根据定义，n³ + n² log n + n 是 O(n³)。



f(n)<=c*g(n) holds when n>1, c = 2

这些证明展示了如何通过不等式分析，将多项式表达式归约为主导项，从而证明其渐近复杂度。

一、证明以下函数的数量级（Prove the order of magnitude）

1. $n^3 + 3n^2 + 3$

该函数的数量级是$O(n^3)$。证明如下：

假设$c = 1 + 1 = 2$。

求解出n_0：$n_0^3 + 3n_0^2 + 3 \leq 2 * n_0^3 \implies 3n_0^2 + 3 \leq n_0^3$。

可以取$n_0 = 4$。因此，$\exists c = 2, n_0 = 4$ 使得 $n^3 + 3n^2 + 3 \leq n^3$ 对于所有 $n > n_0$ 成立。

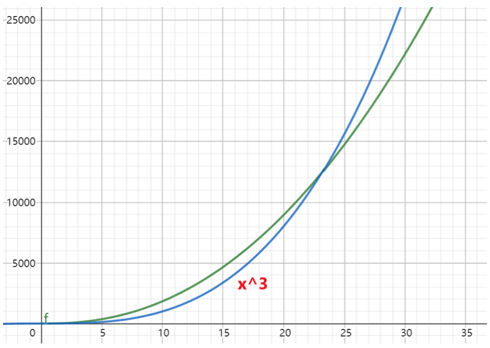
2. $4n^2 log(n) + n^3 + 5n^2 + n$

该函数的数量级是$O(n^3)$。证明如下：

假设$c = 1 + 1 = 2$。

求解出n_0：$4n_0^2 log(n_0) + 5n_0^2 + n_0 \leq n_0^3$。如果以2为底，大约取$n_0 = 25$即可。

因此，$\exists c = 2, n_0 = 25$使得 $4n^2 log(n) + n^3 + 5n^2 + n \leq 2 * n^3$ 对于所有 $n > n_0$成立



3. $2n^2 + n^2 log(n)$

该函数的数量级是$O(n^2 log(n))$。证明如下：

假设$c = 1 + 1 = 2$。

求解出n_0：$2n_0^2 \leq n_0^2 log(n_0) \implies n_0 \geq 4$。可选$n_0 = 4$。

因此，$\exists c = 2, n_0 = 4$ 使得 $2n^2 + n^2 log(n) \leq 2 * n^2 log(n)$ 对于所有 $n > n_0$成立

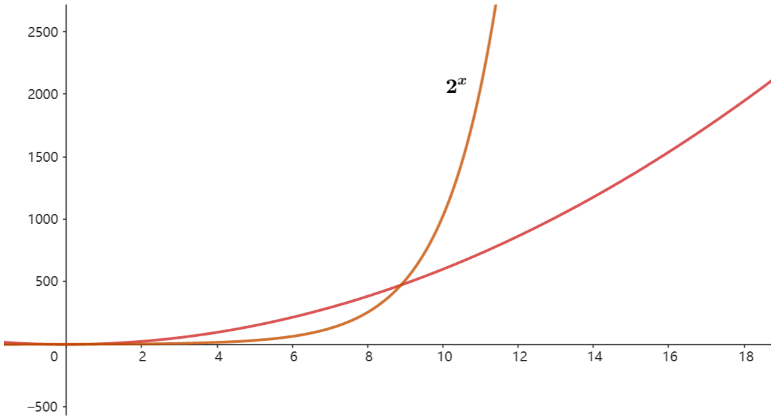
4. $6n^2+2^n$

该函数的数量级是$O(2^n)$。证明如下：

假设$c=1+1=2$。

求解出n_0：$6n_0^2 \le 2^{n_0}$，可选$n_0=10$。

因此$\exists c=2, n_0=10$ 使得 $6n^2+2^n \le 2*2^n$ 对于所有 $n > n_0$ 成立

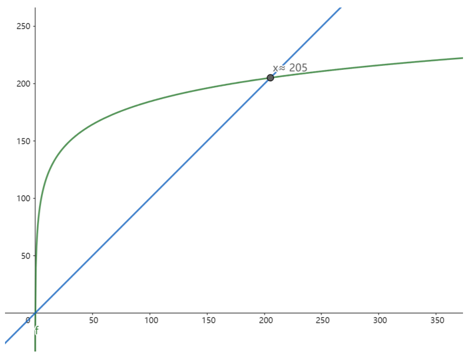


二、写出 $6n^{20}+2^n$ 的数量级

$6n^{20}+2^n$ 的数量级是 $O(2^n)$。为什么不是 n^{20}？

不妨假设 $c=1+1=2$。先求出n_0：$6n_0^{20} \le 2^{n_0}$。如果说直接看的话，就很难看出来结果。但是我们可以试着取对数：

$$\begin{aligned} 6n_0^{20} &\le 2^{n_0} \\ 20*log(6n_0) &\le n_0 log(2) \\ 20*log(6n_0) &\le n_0 \\ n_0 &\approx 205 \end{aligned}$$



也就是，$\exists c=2, n_0=205$ 使得 $6n^{20}+2^n \le 2*2^n$ 对于所有$n > n_0$ 成立。

■ [Week 2 Lecture 2]

> 算法效率 + 搜索/排序
>
> - [ ] 了解一些多项式时间和指数时间算法的例子
> - [ ] 能够对算法进行简单的渐进分析
> - [ ] 能够应用搜索/排序算法并推导其时间复杂度

[1] 线性搜索

```java
public static int linearSearch(int[] array, int number)
{
    for(int i =0;i<array.length;i++)
        if(array[i]==number)
            return i;       // 找到了，返回index
    return -1;              // 没找到index，返回-1
}
```

最好情况：$O(1)$，最坏情况，$O(n)$。平均而言 $O(\frac{n}{2})$ 也就是 $O(n)$。

[2] 二分搜索

如果说数据有序（例如升序），那么我们就可以使用二分搜索的方法来减少时间复杂度。

递归方法：

```java
public static int binarySearch(int[] array, int number){
    if(array == null || array.length==0) return -1; // 如果数组是空的，返回-1
    else return binarySearch(array,number,0,array.length-1);// 开始二分搜索
}
private static int binarySearch(int[] array,int number, int left, int right){
    if(left>right) return -1; // 如果左边大于右边，直接返回-1。因为越界代表找不到
    int middleValue = array[(left+right)/2]; // 定义中间值
    if(number == middleValue) return (left+right)/2; // 如果相等就返回
    if(number < middleValue)  return binarySearch(array,number,left,(left+right)/2-1);
// 如果number比中间值小，就往左边找
    return binarySearch(array,number,(left+right)/2+1,right); // 如果number比中间值大，就
往右边找
}
```

循环方法（更简单）：

```java
public static int binarySearchLoop(int[] array, int number)
{
    if(array == null || array.length==0) return -1; // 如果数组是空的，返回-1
    int left = 0;
    int right = array.length-1;
    while(left<=right)
    {
        int middleValue = array[(left+right)/2];
        if(number > middleValue){ // 往右边找
            left = (left+right)/2 +1;
            continue;
        }
        if(number < middleValue){ // 往左边找
            right = (left+right)/2-1;
            continue;
        }
        if(number == middleValue)
```

```
            return (left+right)/2;
        }
        return -1;
}
```

最好情况 $O(1)$。现在计算最坏情况。假设我们有n个数据。那么假设这个数据恰好要对比到最后一次，也就是需要比较 $\lceil log_2(n) \rceil + 1$次。

如果一个数组 `arr = =[0,1,2,3,4,5,6,7,8,9]`。我们想找到数字 `6`。

第一次：`middle` = `arr[(0+9)/2]` = `arr[4] = 4`。继续搜索下标 `5`~`9`。

第二次：`middle` = `arr[7]` = `7`。继续搜索下标 `5`~`6`。

第三次：`middle` = `arr[5]` = `5`。继续搜索下标 `6`~`6`。

第四次：找到了 `6`。

一共计算了 $log_2(10) + 1 = 4$ 次。

因此最坏的时间复杂度是$O(log_2(n) + 1)$。平均而言就是$O((log_2(n) + 2)/2) = O(\frac{1}{2}log_2(n) + 1)$。最后，时间复杂度是 $O(log_2(n))$

对比而言，二分搜索更有效率！

> 经过测试，如果用线性搜索 $400,000,000$ 个数据，平均每次查找需要 81.34 ms。而使用二分搜索，平均每次查询只需要 5.578×10^{-4} ms。

[3] 字符串匹配

给定一个文本`text`(String型)和一个**模式**`pattern`。要求找到这个文本`text`内是否含有`pattern`。

```java
public static boolean exist(String text, String pattern)
{
    if(text==null||pattern==null|| text.isEmpty() || pattern.isEmpty()) return false;
    if(pattern.length()>text.length()) return false;


    for(int i = 0;i<text.length();i++)                    // 第一层循环
    {
        for(int j = 0; j<pattern.length();j++)            // 第二层循环
        {
            if(pattern.charAt(j)!=text.charAt(j+i)) break;
            if(j==pattern.length()-1) return true;
        }
    }
    return false;
}
```

假设`text`的长度是`n`，`pattern`的长度是`m`。这里嵌套了两层循环。最好的情况是`pattern`就在开头，也就是$O(m)$时间。

最坏的情况是找到末尾才结束。也就是大约遍历了$m \times n$次，即$O(nm)$。总时间复杂度就是 $O(nm)$。

[4] 选择排序

步骤如下：

原始数组 `originalArray= {5,1,3,2,4,0}`。

找出`originalArray= {5,1,3,2,4,0}`最小的，得到`0`，放入新数组`sortedArray = {0}`

找出`originalArray= {5,1,3,2,4}`最小的，得到`1`，放入新数组`sortedArray = {0,1}`

找出`originalArray= {5,3,2,4}`最小的，得到`2`，放入新数组`sortedArray = {0,1,2}`

...

这样就能排序好了。

实际上，并不需要创建两个数组。只需要进行**交换操作**即可

```
public static void selectionSort(int[] array){
    if(array == null||array.length==0) return;
    for(int i = 0; i<array.length-1;i++)
    {
        for(int j=i+1;j<array.length;j++)
        {
            if(array[j]<array[i])
                swap(array,i,j);
        }
    }
}
private static void swap(int[] array, int indexA, int indexB)
    {
        int temp = array[indexA];
        array[indexA] = array[indexB];
        array[indexB] = temp;
    }
```

计算时间复杂度：

这个算法没有极端情况和最好情况。

当`i = 0`的时候，`j`遍历`n-1`次。

当`i = 1`的时候，`j`遍历`n-2`次。

...

因此总共需要的执行数是 $(n-1)+(n-2)+\cdots+1=\dfrac{n\times(n-1)}{2}$

因此时间复杂度是 $O(n^2)$。

[5] 冒泡排序

冒泡排序的思路和选择排序差不多。可以看这个视频。

```
public static void bubbleSort(int[] array){
    if(array == null||array.length==0) return;
    for(int end = array.length-1;end>=0;end--)
    {
        for(int begin = 0;begin<end;begin++)
        {
            if(array[begin]>array[begin+1]) swap(array,begin,begin+1);
        }
    }
}
```

同样的，它的时间复杂度仍然是 $O(n^2)$。

> 选择排序交换的是起始索引（一直往后移动）和动索引，界限是前面。而冒泡排序则是交换两个相邻索引，界限是在后面。

[6] 插入排序(Optional)

插入排序，可以看这个视频。

```
public static void insertSort(int[] array){
    if(array == null||array.length==0) return;
    for(int i = 1;i<array.length;i++)
    {
        int j = i-1;
        while(j>=0 && array[j]>array[j+1])
        {
            swap(array,j,j+1);
            j--;
        }
    }
}
```

总比较数（最坏情况）：$1+2+\cdots+(n-1)=\dfrac{n\times(n-1)}{2}$，时间复杂度为 $O(n^2)$。

[-] 小结

选择排序、冒泡排序、插入排序：这三种算法在最坏情况下的时间复杂度均为 $O(n^2)$。

是否存在更高效的排序算法？**是的**，我们后续会学习它们。

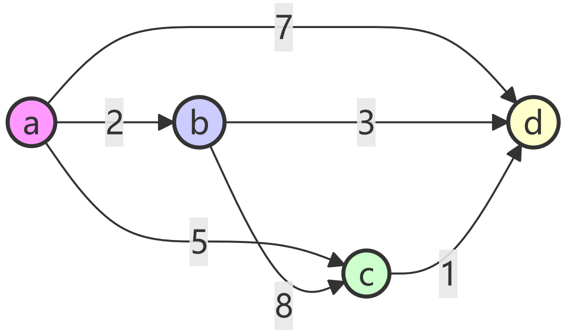
基于比较的最快排序算法的时间复杂度是多少？ $O(n\ \ log(n)\ \ )$

一些**指数时间复杂度**的算法包括：

- **旅行商问题**（Traveling Salesman Problem）
- **背包问题**（Knapsack Problem）

旅行商问题

输入：共有 n 个城市。
输出：找到从某一特定城市出发的最短路径，要求恰好访问每个城市一次，并最终返回到起点城市。
这个问题被称为**哈密顿回路**（Hamiltonian Circuit）。



路径	计算方式	总距离
a -> b -> c -> d -> a	2 + 8 + 1 + 7	18
a -> b -> d -> c -> a	**2 + 3 + 1 + 5**	**11**
a -> c -> b -> d -> a	5 + 8 + 3 + 7	23
a -> c -> d -> b -> a	**5 + 1 + 3 + 2**	**11**
a -> d -> b -> c -> a	7 + 3 + 8 + 5	23
a -> d -> c -> b -> a	7 + 1 + 8 + 2	18

但是，随着城市的增加，可能会出现指数级别的爆炸。如果城市数量为 n，那么需要考虑的数量是 $(n-1)!$

时间复杂度为$O(\ (n-1)!\ )$。比如上图一共四个城市，总共可能的回路数量是 $(4-1)! = 6$

背包问题

输入：给定 n 个物品，每个物品有重量 $w_1, w_2, \ldots, w_n$和价值 $v_1, v_2, \ldots, v_n$，以及一个容量为 w 的背包。

输出：找到一个可以放入背包且总价值最大的物品子集。

应用：一架运输机需要将最有价值的物品集运送到一个偏远地点，同时不超出其容量限制。

如果有四个物品，也就是$n = 4$，这种情况可能的数量是 $2^4 = 16$。时间复杂度：$O(2^n)$，

Q&A

假设你忘记了由5个字符组成的密码。你只记得：

- 这5个字符都是不同的。
- 这5个字符分别是 B、D、M、P、Y。

如果你想尝试所有可能的组合，总共有多少种？

> $5!$

如果这5个字符可以是26个大写字母中的任意一个，那么总共有多少种?

> 26^5（如果重复）
>
> A_{26}^5（如果不重复）

假设密码还包含2个数字，即总共有7个字符:

- 所有字符都是**不同的。**
- 5个字母分别是 B、D、M、P、Y。
- 数字只能是 0 或 1。

总共有多少种组合?

> $C_7^2 A_2^2 A_5^5$

假设密码的形式是 `adaaada`，其中:

- `a` 表示字母，`d` 表示数字。
- 所有字符都是不同的。
- 5个字母分别是 B、D、M、P、Y。
- 数字只能是 0 或 1。

总共有多少种组合?

> $A_2^2 A_5^5$

Tutorial

1. Give the trace table and the output of the following algorithm for $m = 16$.

   ```
   input m
   count = 0
   x = 1
   while x < m do
       begin
           x = x * 2
           count = count + 1
       end
   output count
   ```

   What is the output for general m that is a positive power of 2 (i.e., $m = 2, 4, 8, 16, 32, \ldots$)?

   > $log_2(m) = log_2(16) = 4$

2. Rewrite the following for-loop into a while loop.

```
input m
x = 0
for i = 1 to m do
    begin
        x = x + i
    end
output x
```

```
input m
x = 0
while m>0 do
    begin
    x = x + i
     m--;
 end
output x
```

3. Rewrite the above for-loop into a repeat loop.

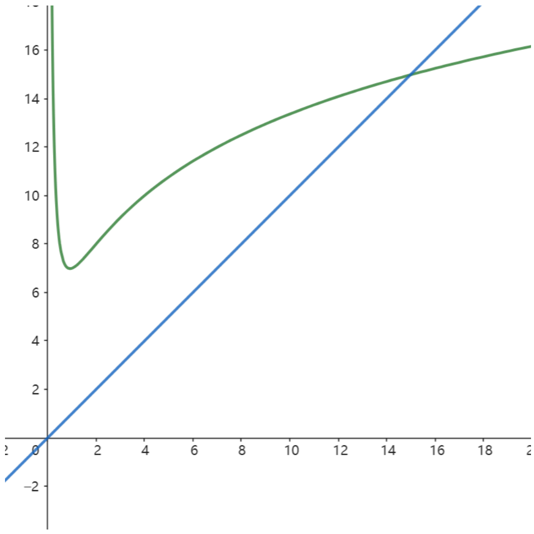
```
input m
x = 0
repeat
    x = x + i;
 m--;
until m = 0
```

4. List the following functions from lowest order to highest order of magnitude:

> 将以下函数从最低阶到最高阶的量级进行排序:
>
> $O(1)$、$O(\ log(n)\ )$、$O(\ log^2(n)\ )$、$O(\sqrt{n})$、$O(n)$、$O(\ nlog(n)\ )$、$O(n^{\frac{3}{2}})$、$O(n^2)$、$O(n^2log^4(n))$、$O(n^3)$、$O(n^{\frac{10}{3}})$、$O(n^4)$

5. Prove that the function $f(n) = 3n^2log\, n + 2n^3 + 3n^2 + 4n$ is $O(n^3)$.

> 令 $c = 2 + 1 = 3$。解不等式 $3n_0^2log\, n_0\ + 2n_0^3 + 3n_0^2 + 4n_0 < 3n_0^3$ 得到 $n_0 \approx 15$ 取 $n_0 = 16$
>
> 所以 $\exists c = 3, n_0 = 16$ 使得 $f(n) \le c \cdot g(n), g(n) = n^3$ 对于所有的 $n > n_0$成立



6. Consider a string T of n characters $T[0..(n-1)]$. Design and write a pseudo code of an algorithm to determine if a given character, denoted by C, appears uniquely in $T[0..(n-1)]$, i.e., whether the character C appears exactly once in T.

   For example, if T is “Hello, how are you?” and C is `H`, then the algorithm should report “Yes, the character appears uniquely.”. However, if C is `e`, then the algorithm should report “No, the character does not appear uniquely.”

   > 考虑一个由 n 个字符组成的字符串 T，其索引范围为 $T[0..(n-1)]$。设计并编写一个伪代码算法，用于判断给定的字符 C 是否在 $T[0..(n-1)]$ 中唯一出现，即字符 C 是否在 T 中恰好出现一次。
   >
   > 例如，如果 T 是 “Hello, how are you?”，且字符 C 是 H，那么算法应报告 “Yes, the character appears uniquely.”（“是的，该字符唯一出现。”）。然而，如果 C 是 e，那么算法应报告 “No, the character does not appear uniquely.”（“不，该字符并未唯一出现。”）。

```java
public static void isAppearAtOnce(String text, String pattern){
 if(countAppearTime(text,pattern)>1)
     System.out.println("No, the character does not appear uniquely.");
 else
     System.out.println("Yes, the character appears uniquely.");
}
private static int countAppearTime(String text,String pattern){
 if(text==null||pattern==null|| text.isEmpty() || pattern.isEmpty()) return 0;
 if(pattern.length()>text.length()) return 0;
 int count = 0;
 for(int i = 0;i<text.length();i++)
 {
     for(int j = 0; j<pattern.length();j++)
     {
         if(pattern.charAt(j)!=text.charAt(j+i)) break;
         if(j==pattern.length()-1) count++;
     }
 }
 return count;
}
```