第一讲 课程简介

本课程针对数学科学学院计算数学系的专业同学。目标是把他们培养成同时具备严密的逻辑思维和丰富编程技能的复合型人才。由于这一特殊目的，反映到课程讲授上，会有以下一些和标准的数据结构课程不太一致的地方：

- 算法和数据结构的逻辑合理性部分，主要靠学生自己阅读课本和视频完成。课堂上不会详细讲述，特别是其证明和推理的细节。因为这一部分恰好是数学院同学的强项，他们已经和即将接受更加复杂的逻辑证明，不需要在这个课程上突出训练。
- 重点放在如何在具体的系统（Linux）上，用一种具体的语言（C++），严格实现一个具体的算法，并确保其拥有正确的效率和特性。这一点，根据我在数学院的体会，恰好是大部分数学院学生需要巩固和提高的。
- 本课程不培养程序员。不会刻意强调代码的工业级别效率和严格性。允许采用较为自由（但必须自我封闭）的代码风格。大部分专题讨论在同阶复杂度下（比如多线程明显就是个例外），不追求小常数级别的效率改进。
- 本课程重视平时作业和项目作业，会有较大的编程量和文献阅读量。
- 课程学习要求学生已具备基本编程能力，基本的计算机操作能力，以及对至少虚拟机级别的Linux操作系统和操作环境有基本了解和掌握。不满足要求的同学可以通过自学QQ群相关视频获得相关技能。并在课程中同步学习。
- 课本采用 Introduction to Algorithm，在QQ群中能找到中英文版本和全部 MIT03 年课程视频。尽管在各课程网站能找到此课程更新的版本，但个人认为 MIT03 是此课程讲解最精彩的一轮。之前没有任何数据结构基础的同学，务必认真看完全部视频。课本上能找到全部内容，算法和相关伪代码。我们授课按视频和课本的次序展开，但内容和课本、视频未必一致。

以上内容供参考，**特别是非数学院同学**，请再次慎重考虑是否要继续学习本课程。

插入排序

排序问题（Sorting） 是一个经典算法问题。它要求对一个输入序列

$$< a_1, a_2, \cdots, a_n >$$

给出输出该序列的一个重排（permutation）

$$< a'_1, a'_2, \cdots, a'_n >$$

使得

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n.$$

一个经典的思路是插入排序，书上给出的伪代码如下：

```
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
```

课本和视频在这里都花了大量的篇幅介绍什么是算法效率，什么是算法本质等等。而我这里假定大家都已经完美掌握了这些精髓。我们现在需要完成的任务是，如何在 Linux 环境的 C++ 编译器下实现这一算法。

首先，我们毫不怀疑这段伪代码的正确性，因此我们可以采用一种非常朴素（naive）的做法，即将上面的代码直接复制过去，然后修改成 C++ 语法能接受的形式。

```
int INSERTION_SORT(double A[], int length)
{
    for (int j = 1; j < length; j++)
    {
        double key = A[j];
        // Insert A[j] into the sorted sequence A[1...j-1].
        int i = j - 1;
        while (i >= 0 && A[i] > key)
        {
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
    return 0;
};
```

在很多时候，我们在阅读文献的时候，都喜欢用这个方式先验证算法的正确性，或者仅仅为了获得一些成就感。命令：

```
git clone https://gitee.com/wang_heyu/Data-Structure-and-Algorithm.git
```

可以获得本课程全部相关讲义和程序的源代码。不停更新中，也欢迎同学参与维护、更新和捉虫。对于表现优异和首次提交并接受的捉虫同学，会有适当的平时成绩分和物质奖励。

正如其名，上面版本的插入排序，先不管其效率如何，首先就是一个“朴素”的版本。这是一个典型的“课堂程序”，是大家熟悉的风格，而且几乎就只是一段 C 语言代码，并没有体现出 C++ 的固有风格。在我们继续之前，首先解决一个“小问题”——批量数据处理和动态内存分配。

我们考虑用户以一个文本文件，空格分隔的形式，向我们提供了 N 个浮点数，该文件的第一个数据是一个 `int`，存放 N 值，然后是连续的 N 个浮点数，以空格分隔。我们将现有的程序框架，主要是输入输出接口作适当的修改以适应这种形式的工作。整个算法流程并没有什么变化，但数据结构稍微有一点小变化，主要是内存的长度需要动态分配，而对于文件的读取，在 Linux 系统中，并没有刻意的区分，你当然可以专门写一个文件读取接口，但实际上，`iostream` 就足够以标准输入输出的方式接管文件读取。先看 `main` 函数的代码：

```
int main(int argc, char *argv[])
{
    int N;
    std::cin >> N;
    double *A = NULL;
    if ((A = (double *)std::malloc(sizeof(double) * N)) == NULL)
    {
        std::cerr << "No enough memory" << std::endl;
        std::exit(-1);
    }
    for (int i = 0; i < N; i++)
        std::cin >> A[i];

    INSERTION_SORT(A, N);
    std::cout << "A: ";
    for (int i = 0; i < N; i++)
        std::cout << A[i] << " ";
    std::cout << std::endl;
    return 0;
};
```

尽管已经用了一些 C++ 的元素，但仍然是标准 C 的风格。不过至少在此框架下，我们已经可以展开一些实际工作了。比如我们可以利用 Linux 系统的 Shell 操作，将数据文件 data 重定向为标准输入，从而实现文本文件读取。大家可以尝试一下：

```
./main < data
```

我们不会专门讨论系统和 C++ 编程，而是在整个授课过程中逐渐熟悉和学习。

现在回头讨论，什么是 C++ 风格的编程？当年开发 C++ 的一个重要原因是为了代码的积累和重用。而我们使用 C++ 的一个理由也是充分利用前人积累的优秀代码，比如 STL 库。事实上我们已经在使用一些 STL 的标准，比如我们已经标明了名空间（`std::`）。我们两个版本的 InsertionSort，一个重要区别是动态数组。但即便是第二个版本，仍然需要程序员主动来控制和管理数组长度 。STL中的 vector 模版类则提供了更为安全和方便的线性存储结

构。在深入了解其机理之前，我们可以将 vector 看作一个数组，而将对应的 iterator 看作是一个自动指针。以下是一个利用了 vector 的 InsertionSort 版本。

```
    int insertionSort(std::vector<double> &_arr)
    {
        for (int j = 1; j < _arr.size(); j++)
        {
            double key = _arr[j];
            // Insert A[j] into the sorted sequence A[1...j-1].

            int i = j - 1;
            while (i >= 0 && _arr[i] > key)
            {
                _arr[i + 1] = _arr[i];
                i = i - 1;
            }
            _arr[i + 1] = key;
        }
        return 0;
    };
```

这里由于仍然采用了 `[i]` 这样的运算形式，并没有完全消除数组越界的危险。不是不可以用 iterator 避免这一问题，但那样会导致代码可读性严重降低，因此，并没有什么原则是不可改变的。这里我宁可牺牲一点稳健性来增加可读性。可读性是一个比较重要的因素，注意这里我的代码风格也开始转变，适当的有一些命名规则出现。我并不强调命名规则的重要性，只是提醒：

- 有比没有好；
- 保持一个自我一致的风格。

在主流程中，适当的运用了 iterator 以从源头上尽量避免潜在的数组越界风险。

```
    int main (int argc, char *argv[])
    {
        int n = -1;
        std::cin >> n;
        std::vector<double> arr(n);
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cin >> *arr_iterator;
        insertionSort(arr);
        std::cout << "A: ";
        for (std::vector<double>::iterator arr_iterator = arr.begin();
```

```
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cout << *arr_iterator << " ";
        std::cout << std::endl;
        return 0;
    };
```

到此，我们借着插入排序，初步讨论了和程序编写有关的一些细节问题。这种对细节的认真考虑会体现在整个课程教学中。

时间效率测试

MIT03 花了大量时间讨论时间复杂度，这确实是一个算法的核心问题，大家务必认真学习视频。而为了不重复讨论，我们这里将关注对象放在如何在实际的工作环境中测试我们的程序效率。MIT03 中的渐进分析方法讨论的都是算法的相对复杂度。你可以认为它把一个 $\Theta(n)$ 复杂度算法作为一个标准，然后评估各算法在问题规模为 n 时和 $\Theta(n)$ 相比的相对阶。这和我们在数学分析中学习过的“无穷大分析”是一致的，而且更简单。所以我有充分的信心大家都会掌握的很好。

现在我们讨论另一种相对时间效率评估，也就是固定测试环境不变，观察不同的算法在这一具体环境下的时间表现。由于涉及实际的具体时间，我们需要掌握一些技术，同时也不能忘记渐进分析的原理。尽管严格意义上说，一个完善的理论证明不需要任何其他手段来加强其正确性，但由于算法是和机器有关的分支学科，而机器、操作系统、编译器和程序员并没有形成严密的逻辑闭环，因此在很多研究工作中，一个实际评估能极大增强研究结果的说服力。

在 Linux 的 shell 下，比如 bash，最简单的时间测试就是使用 `time` 命令：

```
time ./main < data
```

这里 data 文件不妨搞的大一点以便于观察。为此我们编写了一个名为 `generate_data.cpp` 的程序来完成这个任务：

```
#include <iostream>
#include <cstdlib>
#include <random>
#include <vector>
int main(int argc, char *argv[])
{
    int n = -1;
    if (argc != 2)
        n = 1;
    else
        n = std::atoi(argv[1]);
    std::vector<double> arr(n);
```

```
        std::default_random_engine generator;
        std::uniform_real_distribution<double> dist(0.0, 1.0);
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            *arr_iterator = dist(generator);
        std::cout << n << " ";
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cout << *arr_iterator << " ";
        std::cout << std::endl;
        return 0;
    };
```

该程序使用了 C++11 的新提供的 `random` 库来提供随机数，因此在编译的时候需要增加选项 `-std=c++11` ：

```
g++ -o main main.cpp -std=c++11
```

现在我们在 `generate_data` 目录下有产生 `data` 的程序，在 `vector` 目录下有进行排序的程序。我们当然可以将 `generate_data` 产生在标准输出的内容转成一个文件 `data` 再拿去排序：

```
./main 10 > data
```

然而事实上没必要这么做，我们只要 `generate_data/main` 产生的数据“管道（pipe）”给 `vector/main` 即可。也就是：

```
./main 10 | ../vector/main
```

就会自动随机生成10个浮点数，然后再排序。现在我们可以观察整个过程的时间代价：

```
time ./main 10 | ../vector/main
```

不同的系统的输出会有一点点不同，但我们基本上都会得到三个值（Mac 上只有两个，以及一个系统实际时间的百分比）：

- real：实际开销的时间，又称 wall time，墙上的钟过去的时间；
- user：用户 CPU 时间，从用户获得 CPU 运行权限开始计时；
- sys：系统 CPU 时间，从实际程序（进程）获得 CPU 运行资源开始计时。

这里我们真正关心的是 sys 时间。不过我们这里这样做仍然有一点粗糙，因为我们把随机数生成，以及一切转换过程都算进去了。严格意义上说，我们应该只计时 `inserttionSort()` 函数的实际执行时间，因此我们需要在程序中增加断点来完成。

传统的 C 语言提供了 `time()` 函数，它仍然保留在 `cstdlib` 库中。它的功能比较简单，大家可以自己查阅资料。我们这里介绍 C++ 11 标准引入的一个新的专门处理各种时间计算的库 `chrono` 。目前 C++ 社区基本上会把一些前瞻性的工作放在 `boost` 这个库集合中，然后将一些成熟的工作纳入到 C++ 的标准库，也就是 Standard Template Library （STL）中。`chrono` 就曾经是 `boost` 的一部分，现在被吸收到了 STL 中。由于使用了 C++ 11 标准，所以接下去的编译都必须增加参数 `-std=c++11` ，直到有一天编译器升级默认支持 C++ 11 以上标准为止。

这一段完整的测试代码为：

```
    /// Time the sort by chrono.
    int main (int argc, char *argv[])
    {
        /// The unit seems be one nano second.
        typedef std::chrono::high_resolution_clock myclock;
        int n = -1;
        std::cin >> n;
        std::vector<double> arr(n);
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cin >> *arr_iterator;
        /// Start the timer.
        myclock::time_point beginning = myclock::now();
        insertionSort(arr);
        /// Finish the timer.
        myclock::duration d = myclock::now() - beginning;
        std::cout << "A: ";
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cout << *arr_iterator << " ";
        std::cout << std::endl;
        std::cout << "time: " << d.count() << std::endl;
        return 0;
    };
```

对它的编译命令为：

```
    g++ -o main -std=c++11 main.cpp
```

如果我们在 `timer` 目录下，调用 `vector` 目录下的 `data` 来测试这个程序，完整的命令为：

```
./main < ../vector/data
```

我们之前已经完成了随机生成指定长度随机数据的程序 `generate_data`，现在可以通过 shell 的管道运算 `|` 将二者连接，成为一个完整的测试插入排序效率的脚本：

```
../generate_data/main 10 | ./main
```

这样就是测试10个随机输入的插入排序的时间。这实际上是一个 shell 的脚本。shell 作为操作系统和用户间的一个字符交互环境，具有丰富的功能。我们自己设计的 C++ 程序，在这种环境下自然形成了操作系统的扩展。shell 本身也是一种编程语言，有自己的条件和循环，具体请自行参阅文献（比如菜鸟教程），Ubuntu 的默认 shell 是 bash，而 Mac 的默认 shell 则是 zsh。不过各种不同版本的 shell，基本命令都差不多。

我们多运行几次上面的脚本，首先出现的问题是我们期待每一次的输入都是随机的，但事实上并不是。这个原因是在 `generate_data` 中，我们产生的随机数实际上是伪随机数，因此当我们不调整随机数参数时（又称“种子”，seed），就会每次产生一样的随机序列。其次是即便对同样的序列，每一次运行的时间都不太一样。这是因为机器运行的周期造成时间上会有偏差。一个认真的时间统计，一般要做到：随机输入，多次测试，再进行统计分析（此时均值，方差分别代表什么意义？）。我们这里可以利用机器周期的不稳定，用计时工具产生一个近乎真随机数的某段代码的实际运行时间，然后用它做随机数种子，产生伪随机数列。这种做法能得到质量非常高的随机序列。为此我们将 `generate/main.cpp` 的代码改为：

```
#include <iostream>
#include <cstdlib>
#include <random>
#include <vector>
#include <chrono>
int main(int argc, char *argv[])
{
    typedef std::chrono::high_resolution_clock myclock;
    myclock::time_point beginning = myclock::now();
    // Just do something ...
    int n = -1;
    if (argc != 2)
        n = 1;
    else
        n = std::atoi(argv[1]);
    std::vector<double> arr(n);
    // obtain a seed from the timer
    myclock::duration d = myclock::now() - beginning;
    unsigned seed = d.count();
    std::default_random_engine generator(seed);
```

```
        std::uniform_real_distribution<double> dist(0.0, 1.0);
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            *arr_iterator = dist(generator);
        std::cout << n << " ";
        for (std::vector<double>::iterator arr_iterator = arr.begin();
             arr_iterator != arr.end();
             ++arr_iterator)
            std::cout << *arr_iterator << " ";
        std::cout << std::endl;
        return 0;
    };
```

我们手工运行多次可以发现这个程序确实可以产生随机的指定长度序列：

```
./main 10
```

接下去我们在 `timer` 目录下用 shell 脚本完成用户指定 `$1` 次数的输入规模为 `$2` 的随机测试：

```
#!/bin/bash
echo $1   # 用户给出的第一个参数
for i in $(seq 1 10)   # $后为数据集，从 1 到 10 的整数
do
        ../generate_data/main $2 | ./main
done
```

这个脚本可以存储为一个文本文件 `test`，执行的方式有两种，一种直接在 shell 下输入：

```
bash test 10 10
```

另一种先将 `test` 设置为可执行文件：

```
chmod a+x test
```

然后它就可以直接运行：

```
./test 10 10
```

当然所有这些功能我们都可以用 C++ 或任何其他编程语言实现。但相对而言，shell 是一种更加轻量级的编程。熟练以后能极大提高工作和开发效率。我们以后的讲述将混合两种模式。比如这里，我们就已经利用脚本完成指定长度随机输入和多次测试两个任务。但是输出的内容非常复杂，而我们只需要其中的 time 部分，解决的方法要么修改 `timer.cpp`，要么直接使用 shell 命令 grep：

```
$./test 10 10 | grep time
```

你会看到所有的需要的数据被干净地列了出来。接下去我们需要对数据进行统计，shell 也可以完成这个任务，但是明显不如 C++ 这样一个强语言擅长，所以这里不如转回 C++ 代码。我们另外开设一个目录 `stat` 专门处理数据统计，里面的 `main.cpp` 为：

```
#include <iostream>
#include <vector>
#include <string>
int main (int argc, char *argv[])
{
    int n_test = 0;
    std::string dump;
    double time = 0;
    double total_time = 0;
    while (true)
    {
        if (!(std::cin >> dump))
            break;
        std::cin >> time;
        total_time += time;
        n_test++;
    }
    std::cout << "average time: " << total_time / n_test << std::endl;
    return 0;
}
```

现在注意，我们进入 `timer` 目录，然后执行：

```
./test 10 100 | grep time | ../stat/main
```

就能够得到 10 次长度为 100 的随机数组排序的平均时间。仔细回顾一下这个过程，看看我们是如何充分利用计算机系统来自动化我们的工作的。有了这个工作环境，我们很容易能做一个插入排序相对时间效率的测试，也即时间 t 关于输入数组规模 n 的增长趋势。注意这里我们采用的是随机数组，因此实际测试的是平均时间，而不是书上强调的最坏时间。如何测试最坏时间，留作作业。

最后我们写一个批处理 `batch_test`，来一口气完成一系列测试。当然一个 C++ 程序也可以完成，但对于这种轻量级的过程，有时用 shell 效率更高：

```
#!/bin/bash
# $1 测试总组数
# $2 测试的起始规模
# $3 每组测试的增量
# $4 每组测试的重复次数
echo $1 $2 $3 $4
n=$2            # 不能有空格
for i in $(seq 1 $1)    # $后为数据集，从1到10的整数
do
    echo "size: $n"
    ./test $4 $n | grep time | ../stat/main
    n=$((n+$3))       # 不能有空格
done
```

现在我们在 `timer` 目录下输入：

```
./batch_test 20 20 20 1000
```

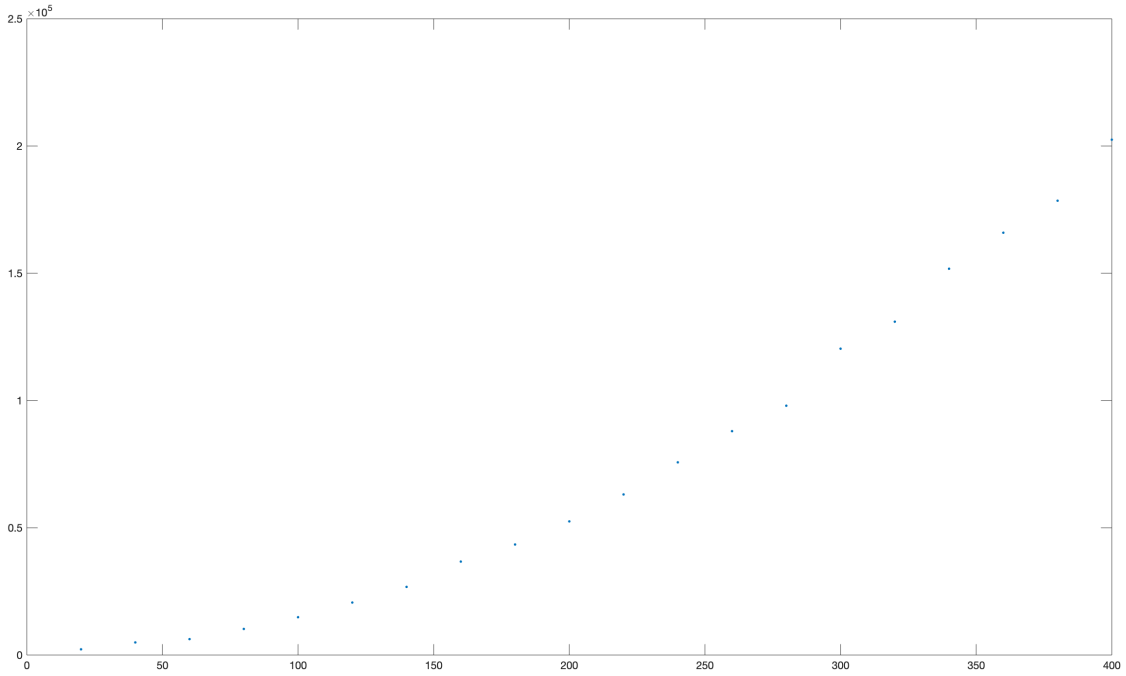
就能得到一组输入规模从 20 开始，每组增加 20，一共 20 组，每组重复测试 1000 次再取均值的测试结果。如果能掌握这种工作方式，相信一定会在未来的学习和工作中提高利率。比如我们想将此结果输入到 Matlab 画一张图，那么我们可以将上面的 shell 脚本稍加修改成 `test4matlab`：

```
#!/bin/bash
n=$2
echo "re = ["
for i in $(seq 1 $1)
do
    re=`./test $4 $n | grep time | ../stat/main`
    echo $n ${re#*:}   # re#*: 表示忽略re中:之前的部分
    n=$((n+$3))
done
echo "];"
echo "plot(re(:, 1), re(:, 2), '.');"
```

现在只要运行

```
./test4matlab 20 20 20 100 > result.m
```

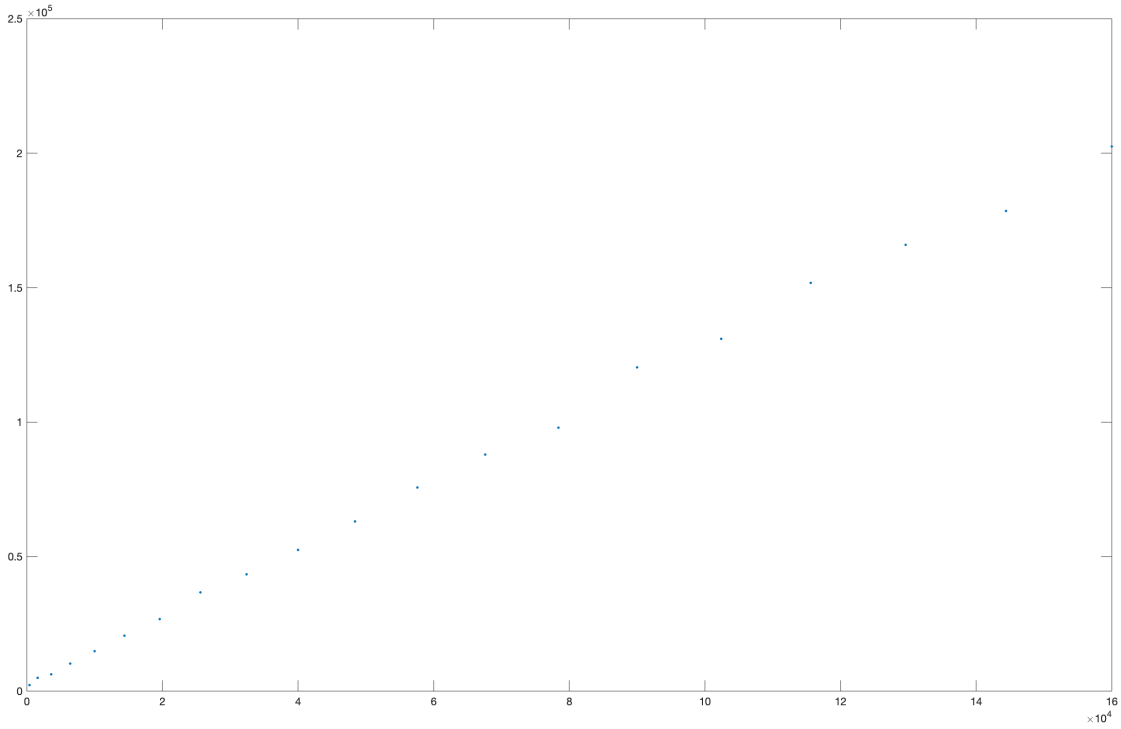
就会得到一个 `result.m` 文件，在 Matlab 中运行后即可看到一个图形化的结果。



从 MIT03 视频我们知道插入排序的效率是 $\Theta(n^2)$，如何验证这一点？尽管我们现在看到的点的分布很像一条二次曲线，但“很像”不是一个好的结论。办法有很多，比如，将绘图命令改成：

```
plot(re(:, 1).^n, re(:, 2), '.');
```

这样我们能看见所有的点基本分布在一条直线上。也就是 (n^2, t_n) 应该呈线形关系，这里 t_n 表示输入规模为 n 时问题的计算时间。



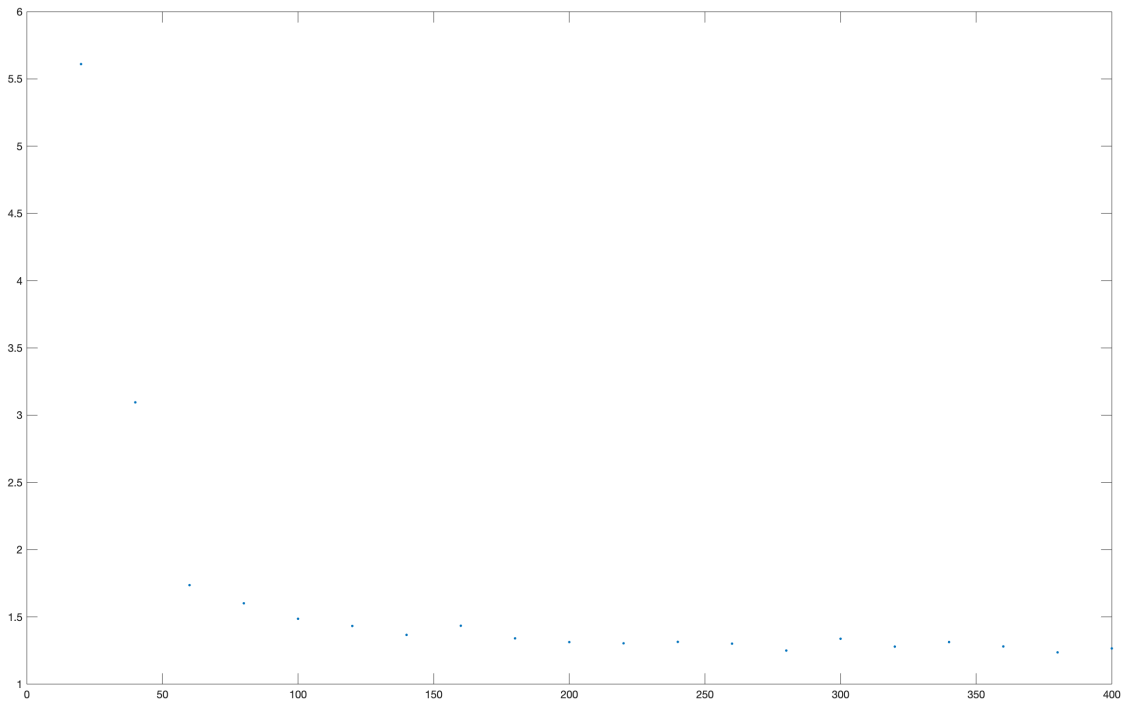
更好的办法是，直接绘制

```
plot(re(:, 1), re(:, 2)./(re(:, 1).^2), '.');
```

这里我们看到点列逐渐趋于一稳定常数，这表明

$$\frac{t_n}{n^2} \to c, n \to \infty.$$

这里 c 是常数，而这才是 $t_n = \Theta(n)$ 的本意。

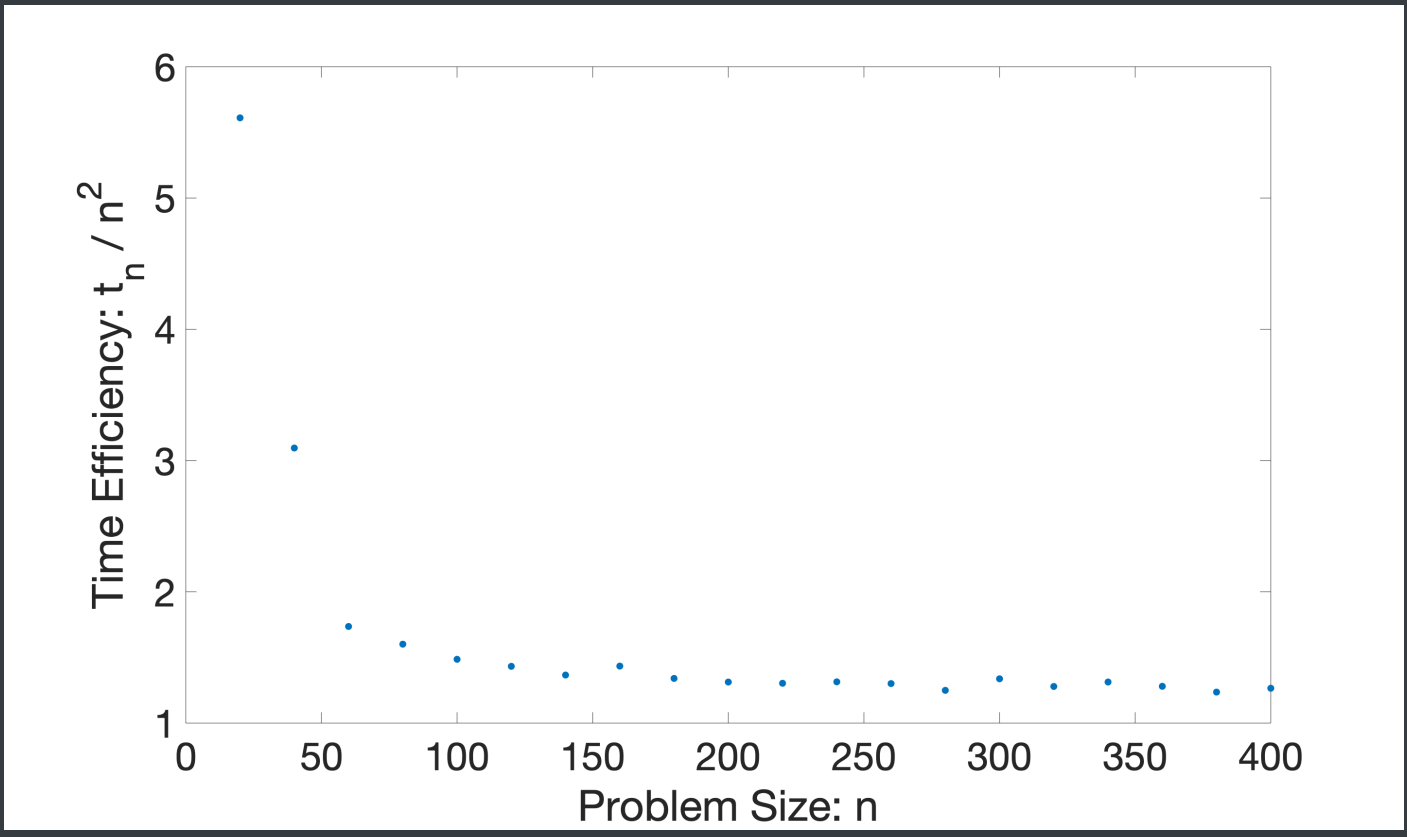


实际工作时，图形也是很重要的描述手段，大家可以用 Matlab，Python，gnuplot，或任何你熟悉的绘图软件。但不论用那种软件绘图，有几个基本原则必须遵守：

1. 严禁为了任何理由，篡改实际数据；
2. 图像是有版权和著作权的，不得直接复制正式出版或发表的图像（必要时可以重绘），如果在绘图时使用了别人独创的想法，必须标注引用；
3. 图像必须编号，其下方或上方的显著位置必须有图像内容的标题（caption）和间接，对图像中的有意义的内容，必须有说明，必要时，应该增加图示（legend）；
4. 在报告或文章的证明中，对图像的制作动机、内容和显示结果的意义必须有必要的分析和说明，不怕与图像标题说明部分重复；
5. 图像的具体内容、标注和数据都必须尽可能清晰，明确。图像的正确、清晰、直观永远优先于美观。比如图像中的数字和标记都必须足够大且清晰。
   因此以上各幅图像并不严格符合上述原则，特别是3，4，5（1，2则是更严重的错误，一般我们的同学极少有意去犯，但因为其后果极其严重，要杜绝侥幸和疏忽大意）。我们用下面的命令重绘一下图像：

```
plot(re(:, 1), re(:, 2)./re(:, 1).^2,'.', 'MarkerSize', 20);
xlabel('Problem Size: n');
ylabel('Time Efficiency: t_n / n^2');
set(gca,'FontSize',20);
```

这样得到的图像要好得多。



然而归根到底，我们讲义采用的是一种轻量级的笔记排版工具：Markdown。用它来写作业或报告，缺少一些必要的排版功能如编号，引用等等。我们推荐使用 latex 作为作业工具，在 doc 目录下，我们提供了一个中文的作业模版，供参考，如果大家愿意用英文完成作业，也可以接受。在 Ubuntu 下的 pdf 阅读工具我推荐 okular，安装办法是：

```
sudo apt-get install okular
```

而 Markdown 的工具可以选择的有很多，推荐 Typora 或 vs code。由于效率问题，不建议在虚拟机中直接大量排版 Markdown 文档，应该在你的主系统中做这件事。我们在未来的学习和工作时，可以根据自己的喜好来选择编辑工具和排版工具，只要最终能生成具有专业质量的文档、代码和结果即可。但工具的选择在学习初期是重要的，如果选择了错误的工具和系统，会极大影响未来的学习和工作效率。这里我规定：不得使用除 vs code 以外的微软提供的工具软件来完成作业。特别是 Word！当然你可以使用某个微软工具来处理中间数据，比如用 Excel 来绘图，但实际上并不鼓励这么做。Linux 不止是一个操作系统，也是一整套工作习惯，代表了以 C/C++ 为核心的社区和文化。本课程的一个重要目的也是引导大家进入这个社区，因为这里才是真正的计算机世界。

作业

MIT03 视频中，更加重视的是最坏时间效率。请自己设计实验方案，测试 C++ STL 版本的插入排序的最坏时间效率。讲义所提供的程序和脚本都可以直接使用，也可以根据自己的需要做适当的调整和改写。提交一份最多不超过五号中文字体（或11pt英文字体）3页 A4 篇幅的 pdf 实验报告。程序和脚本作为附件提交，并且不计入篇幅。参考文献不计入篇幅。