实验内容

排序n个元素，元素为随机生成的0到2^{15} − 1之间的整数，n的取值为：$2^3, 2^6, 2^9, 2^{12}, 2^{15}, 2^{18}$。

实现以下算法：直接插入排序，堆排序，快速排序， 归并排序，计数排序。

截图：

- 五个排序算法n=2^3时排序结果的截图。
- 任一排序算法六个输入规模运行时间的截图。

根据不同输入规模时记录的数据，画出各算法在不同输入规模下的运行时间曲线图。比较你的曲线是否与课本中的算法渐进性能是否相同，若否，为什么，给出分析。

比较不同的排序算法的时间曲线，分析在不同输入规模下哪个更占优势?

实验设备和环境

macOS Mojave 10.14.6

MacBook Pro (Retina, 13-inch, Early 2015)

处理器 2.9 GHz Intel Core i5

实验方法和步骤

思路：

- `main.c` 包含生成随机数(../input/input.txt)、五个算法的实现、精确度够高的时间函数、每个算法的六种测试量级（5个对应output下文件夹内会有6个result文件和1个综合的5行time文件）

  注意1：测试情况都打印出来，截图。

  注意2：写英文注释（防止乱码）。
- 作图，图片要有单位，横纵坐标等信息。

具体实现：

头文件

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
```

宏定义

为了某些数组可以静态创建，也为了区分 pow(2, n) 和这两个特殊值（分别是输入规模上限和数据大小上限）。

```c
#define POW2_18 262144
#define POW2_15 32768
```

为了让五个算法统一处理，需要将五个函数放入数组。声明封装好的、统一输入输出形式的五个函数，并且利用 typedef 定义这种函数类型。

```c
typedef void (*SORT)(int);

void INSERTION_SORT(int n);
void HEAP_SORT(int n);
void QUICK_SORT(int n);
void MERGE_SORT(int n);
void COUNTING_SORT(int n);

// function list
SORT function_list[] = {INSERTION_SORT,\
                        HEAP_SORT,\
                        QUICK_SORT,\
                        MERGE_SORT,\
                        COUNTING_SORT};
```

为了不让每次排序都从 input.txt 中读取，用一个全局数组 A 保存数据，本程序中不修改。所以排序前需要将对应输入规模的数据拷贝到数组 B，结果也将在数组 B 中。

```c
int A[POW2_18]; // input array
int B[POW2_18]; // output array
```

随机生成输入的函数 rand_15_18()

为了严谨性，添加了 `srand((unsigned)time(NULL));`。

```c
/* write the txt and generate A[] */
void rand_15_18(){
    //  rand() between 0 and 2^15-1
    //printf("%d\n", time(NULL));
    srand((unsigned)time(NULL));
```

```
    FILE *fp = fopen("../input/input.txt", "w");
    char buf[20];
    int num;
    for(int i = 0; i < POW2_18; i++){
        num = rand() % POW2_15;
        A[i] = num;
        memset(buf, '0', 20* sizeof(char));
        sprintf(buf, "%d\n", num);
        //printf("%s", buf);
        fprintf(fp, "%s", buf);
    }
    fclose(fp);
    printf("successfully closed fp!\n");
    return;
}
```

接下来是五个算法的实现，共同点是将 B 数组的初始化等等处理与算法分离（考虑到计时的准确性）并且要注意数组开始地址为 0 不是 1

直接插入排序 INSERTION_SORT()

```
void INSERTION_SORT(int n){
    int key, i;
    for(int j = 1; j < pow(2, n); j++){
        key = B[j];
        i = j-1;
        while(i >= 0 && B[i] > key){
            B[i+1] = B[i];
            i = i-1;
        }
        B[i+1] = key;
    }
    return;
}
```

堆排序 HEAP_SORT

LEFT() 和 RIGHT() 因为开始地址为 0 需要做调整。

```
int LEFT(int i){
    return 2*i+1;
}
int RIGHT(int i){
    return 2*i+2;
}

void MAX_HEAPIFY(int i, int heapsize){
    int l, r, largest, t;
    l = LEFT(i);
```

```
    r = RIGHT(i);
    if(l < heapsize && B[l]>B[i])
        largest = l;
    else
        largest = i;
    if(r < heapsize && B[r]>B[largest])
        largest = r;
    if(largest != i){
        t = B[i];
        B[i] = B[largest];
        B[largest] = t;
        MAX_HEAPIFY(largest, heapsize);
    }
}

void BUILD_MAX_HEAP(int heapsize){
    for(int i = heapsize/2; i >= 0; i--)
        MAX_HEAPIFY(i, heapsize);
}

void HEAP_SORT(int n){
    int heapsize = pow(2, n);
    BUILD_MAX_HEAP(heapsize);
    int t;
    for(int i = pow(2, n)-1; i >= 1; i--){
        t = B[0];
        B[0] = B[i];
        B[i] = t;
        heapsize--;
        MAX_HEAPIFY(0, heapsize);
    }
}
```

快排 QUICK_SORT()

为了统一，额外封装了QUICK_SORT()。

```
int PARTITION(int p, int r){
    int x = B[r];
    int i = p-1;
    int t;
    for(int j = p; j < r; j++){
        if(B[j] <= x){
            i++;
            t = B[i];
            B[i] = B[j];
            B[j] = t;
        }
    }
```

```
    t = B[i+1];
    B[i+1] = B[r];
    B[r] = t;
    return i+1;
}

void QUICKSORT(int p, int r){
    if(p<r){
        int q = PARTITION(p, r);
        QUICKSORT(p, q-1);
        QUICKSORT(q+1, r);
    }
    return;
}

void QUICK_SORT(int n){
    QUICKSORT(0, pow(2, n)-1);
    return;
}
```

归并排序 MERGE_SORT()

为了统一，额外封装了MERGE_SORT()。

因为输入最大为 POW2_15-1，所以令 POW2_15 为“无穷”。

```
void MERGE(int p, int q, int r){
    int m = q-p+1;
    int n = r-q;
    int *ptr_m = (int*)malloc(sizeof(int)*(m+1));
    int *ptr_n = (int*)malloc(sizeof(int)*(n+1));
    int i, j;

    for(i = 0; i < m; i++)
        ptr_m[i] = B[p+i];
    for(j = 0; j < n; j++)
        ptr_n[j] = B[q+j+1];
    ptr_m[m] = POW2_15;
    ptr_n[n] = POW2_15;
    //printf("%d %d\n", ptr_m[m], ptr_n[n]);
    i = 0;
    j = 0;
    for(int k = p; k <= r; k++){
        if(ptr_m[i] <= ptr_n[j]){
            B[k] = ptr_m[i];
            i++;
        }
        else{
            B[k] = ptr_n[j];
            j++;
```

```
        }
    }
    free(ptr_m);
    free(ptr_n);
    return;
}

void MERGESORT(int p, int r){
    if(p < r){
       int q = (p+r)/2;
       MERGESORT(p, q);
       MERGESORT(q+1, r);
       MERGE(p, q, r);
    }
    return;
}

void MERGE_SORT(int n){
    MERGESORT(0, pow(2, n)-1);
    return;
}
```

计数排序 COUNTING_SORT()

第 14 行要写上C[A[j]]**-1** 我通过打印 C[A[j]] 的信息才意识到。

```
void COUNTING_SORT(int n){
    /*
     * input n = pow(2,n) from 0 to k = 2^15-1
     */
    int C[POW2_15];
    memset(C, 0, sizeof(int)*POW2_15);
    int length = pow(2, n);
    for(int j = 0; j < length; j++)
        C[A[j]] = C[A[j]]+1;
    for(int i = 1; i < POW2_15; i++)
        C[i] += C[i-1];
    for(int j = length-1; j >= 0; j--){
        //printf("%d", C[A[j]]);
        B[C[A[j]]-1] = A[j];
        C[A[j]] = C[A[j]]-1;
    }
    //printf("\n");
    return;
}
```

考虑到写文件的复杂情况，使用专门的函数处理更加干净。

```
const char * string_list[] = {"INSERTION_SORT",\
```

```
                                "HEAP_SORT",\
                                "QUICK_SORT",\
                                "MERGE_SORT",\
                                "COUNTING_SORT"};

void write_output_time(char *array, int len, int i){
    memset(array, '\0', sizeof(char)*len);
    strcpy(array, "../output/");
    strcat(array, string_list[i]);
    strcat(array, "/time.txt");
    //printf("%s\n", array);
    return;
}

void write_output_result(char *array, int len, int i){
    memset(array, '\0', sizeof(char)*len);
    strcpy(array, "../output/");
    strcat(array, string_list[i]);
    strcat(array, "/result_");
    //printf("%s\n", array);
    return;
}
```

接下来介绍主函数 main()

先调用 rand_15_18() 创建输入，文件指针 fp1 是 time.txt 的指针，rp 是 result_n.txt 的指针。

clock() 计时的单位是时钟周期数，需要通过 CLOCKS_PER_SEC 换算为秒。

fclose() 要做到一一对应。

```
int main(){
    printf("Let's start!\n");
    //printf("RAND_MAX = %d\n", RAND_MAX);
    // generate random numbers
    rand_15_18();

    int n, num;
    clock_t start_t, end_t;
    double total_t;
    char buf[40], string[40];
    for(int i = 0; i < 5; i++){
        write_output_time(string, 40, i);
        FILE *fp1 = fopen(string, "w");

        for(n = 3; n <= 18; n = n+3) {
            // initialization
            memcpy(B, A, pow(2, n) * sizeof(int));
            // time
            start_t = clock();
```

```
            function_list[i](n);
            end_t = clock();
            total_t = (double) (end_t - start_t) / CLOCKS_PER_SEC;
            printf("n=%d, sort%d time=%fs\n", n, i, total_t);
            // fill output time.txt
            memset(buf, '0', 40 * sizeof(char));
            sprintf(buf, "%f\n", total_t);
            fprintf(fp1, "%s", buf);
            // generate output result.txt
            memset(buf, '0', 40 * sizeof(char));
            write_output_result(string, 40, i);
            sprintf(buf, "%s%d%s", string, n, ".txt");
            FILE *rp = fopen(buf, "w");
            for(int i = 0; i < pow(2, n); i++){
                num = B[i];
                memset(buf, '0', 40 * sizeof(char));
                sprintf(buf, "%d\n", num);
                fprintf(rp, "%s", buf);
            }
            fclose(rp);
            printf("successfully closed rp!\n");
        }
        fclose(fp1);
        printf("successfully closed fp1!\n");
    }
}
```

实验结果与分析

进入src/ 运行

```
gcc main.c -o main
./main
```

默认OUTPUT/ 文件夹下已经建好对应五个空文件夹，否则会 segmentation fault。

默认会进入 `src/` 所以文件使用的是相对路径。

五个排序算法n=2^3时排序结果的截图

计数排序

result_3.txt

```
572
8194
12919
13348
14190
14877
17471
20564
```

堆排序

result_3.txt

```
572
8194
12919
13348
14190
14877
17471
20564
```

插入排序

result_3.txt

```
572
8194
12919
13348
14190
14877
17471
20564
```

归并排序

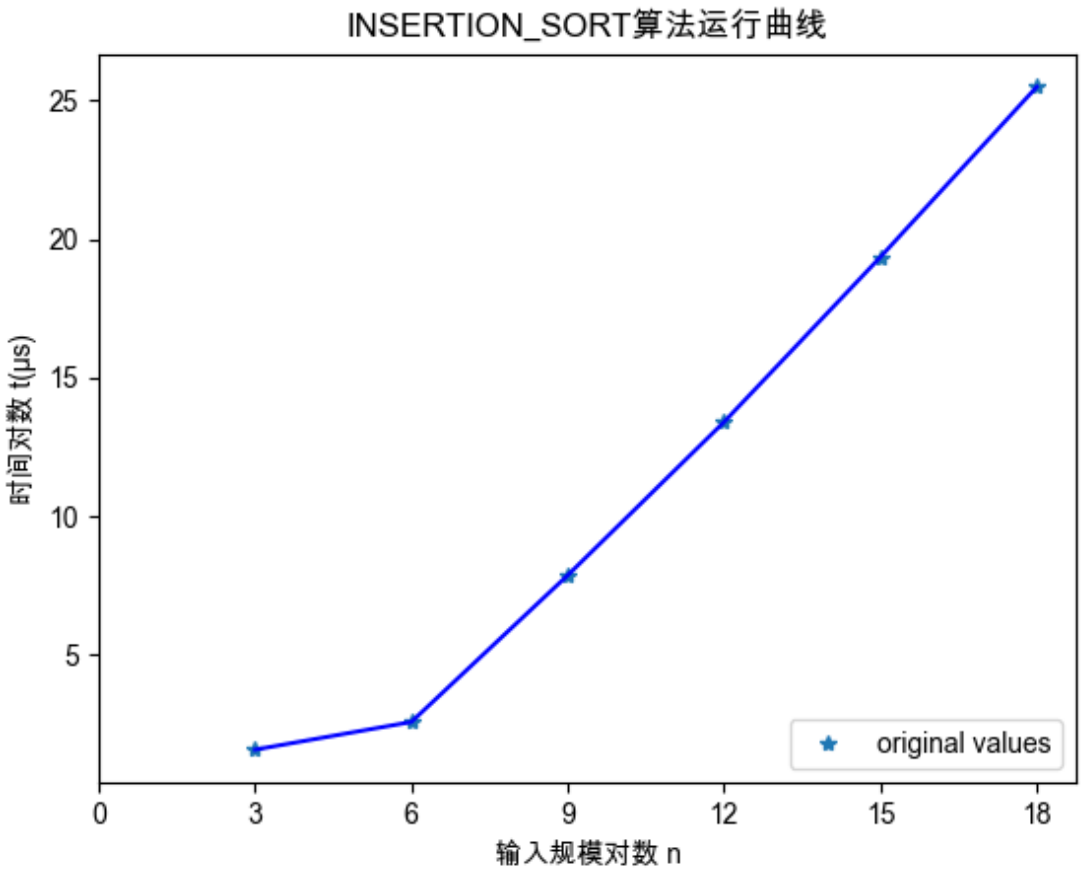
快速排序

任一排序算法六个输入规模运行时间的截图

基数排序

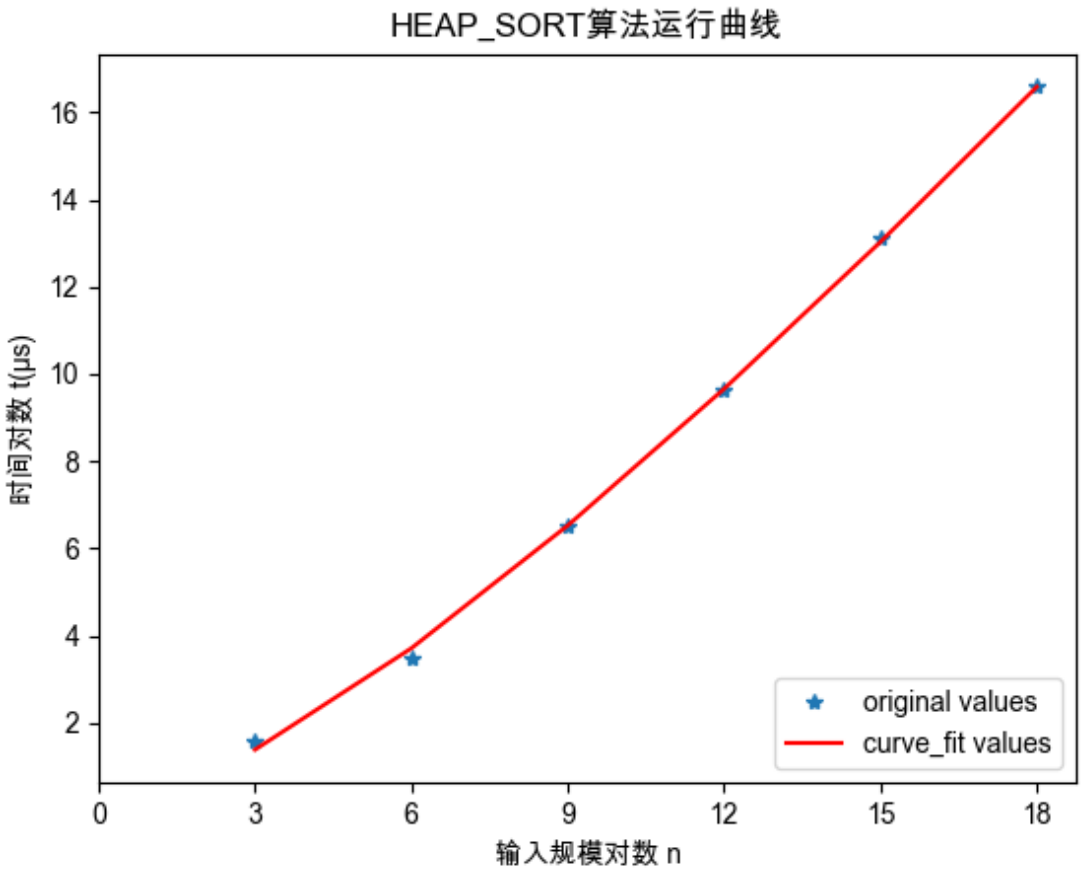
运行时间曲线图

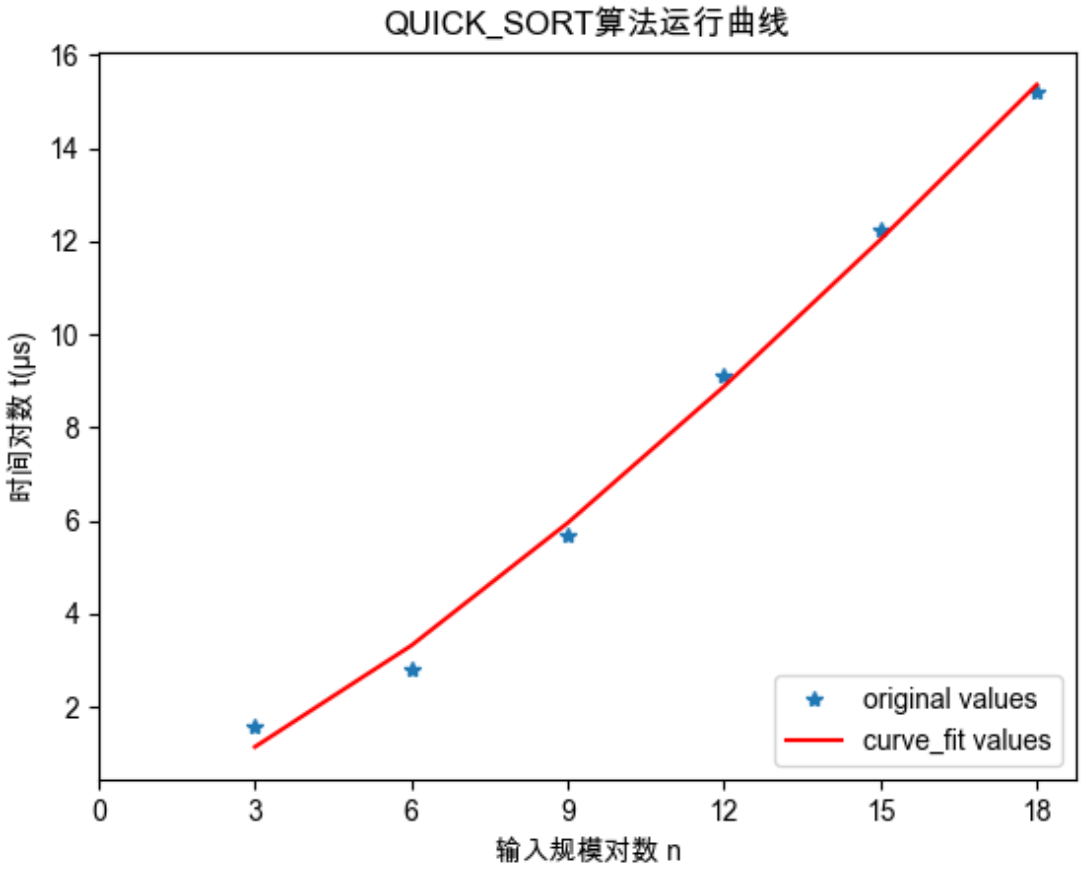
比较你的曲线是否与课本中的算法渐进性能是否相同，若否，为什么，给出分析。

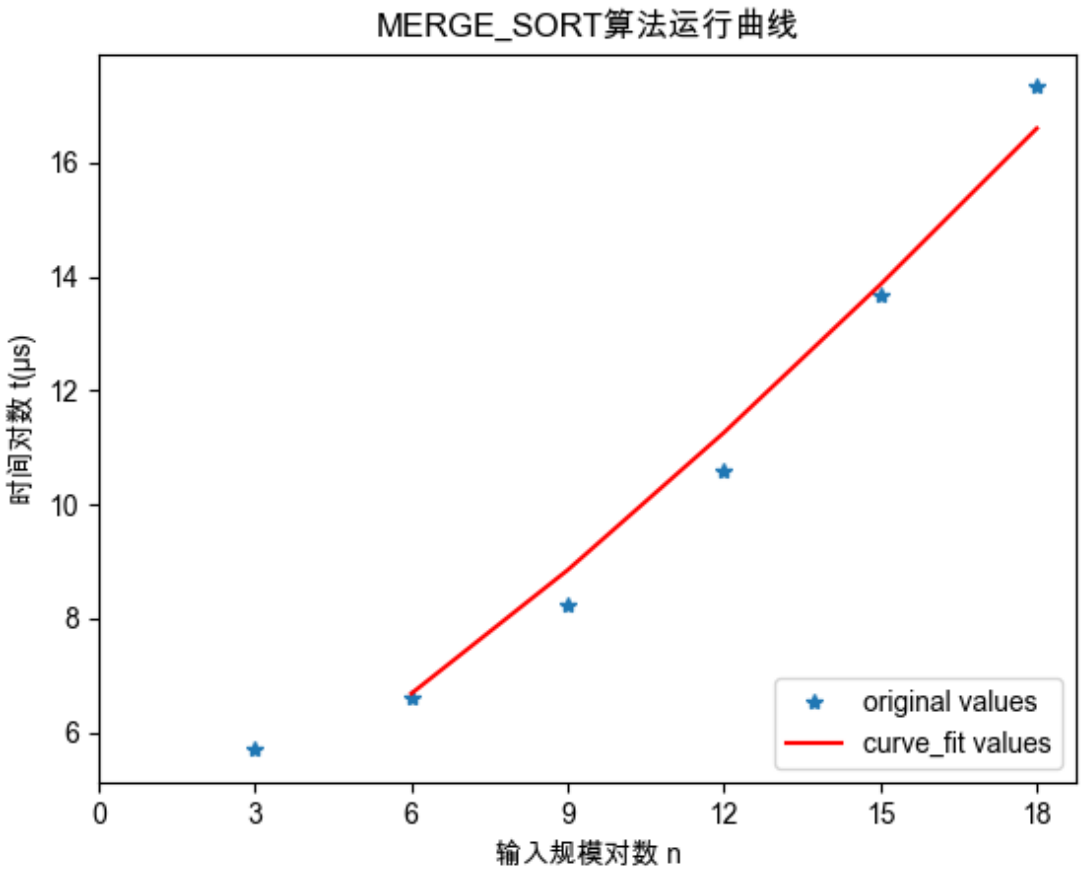


插入排序渐近性能是 $\Theta(n^2)$，在图中预期是斜率为2的直线。

除去 n = 3 处的点的拟合函数为 1.91 x - 9.178，很接近预期，可以认为与课本中的算法渐进性能相同。

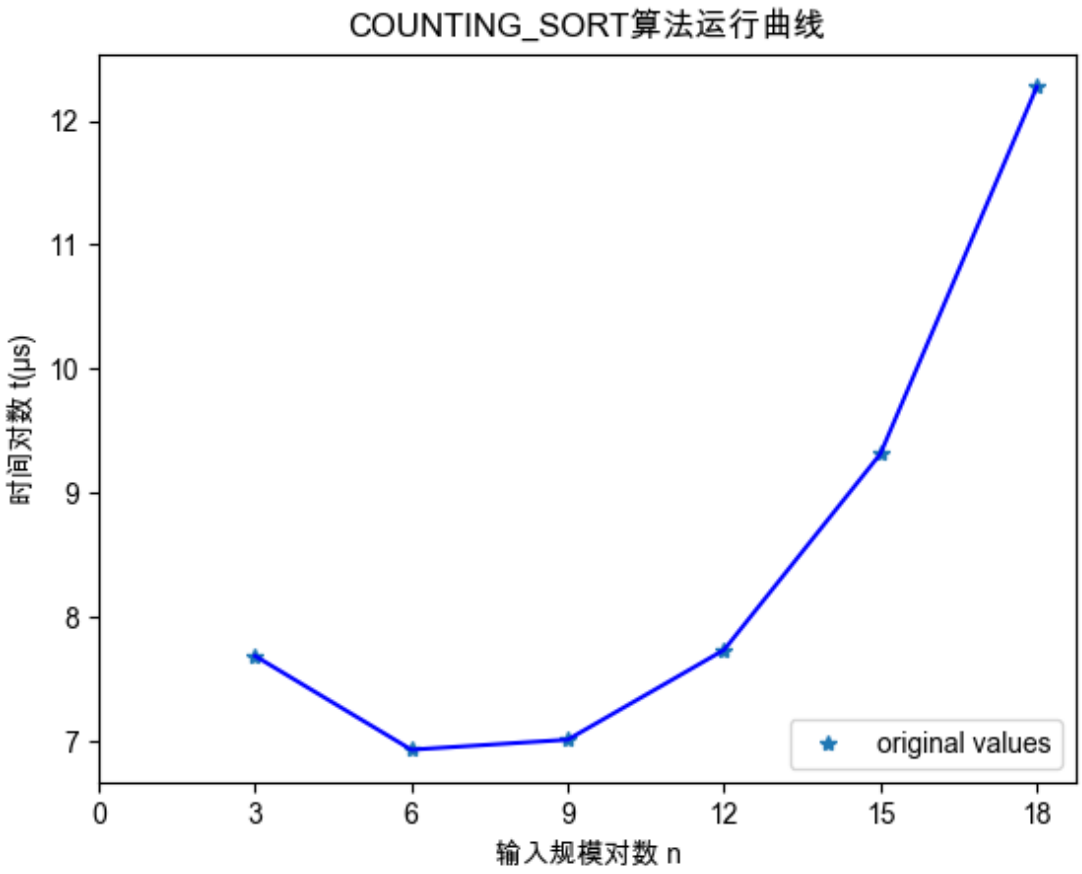

HEAP_SORT算法运行曲线
时间对数 t(μs)
输入规模对数 n
original values
curve_fit values
0
3
6
9
12
15
18
2
4
6
8
10
12
14
16


QUICK_SORT算法运行曲线
时间对数 t(μs)
输入规模对数 n
original values
curve_fit values
0
3
6
9
12
15
18
2
4
6
8
10
12
14
16



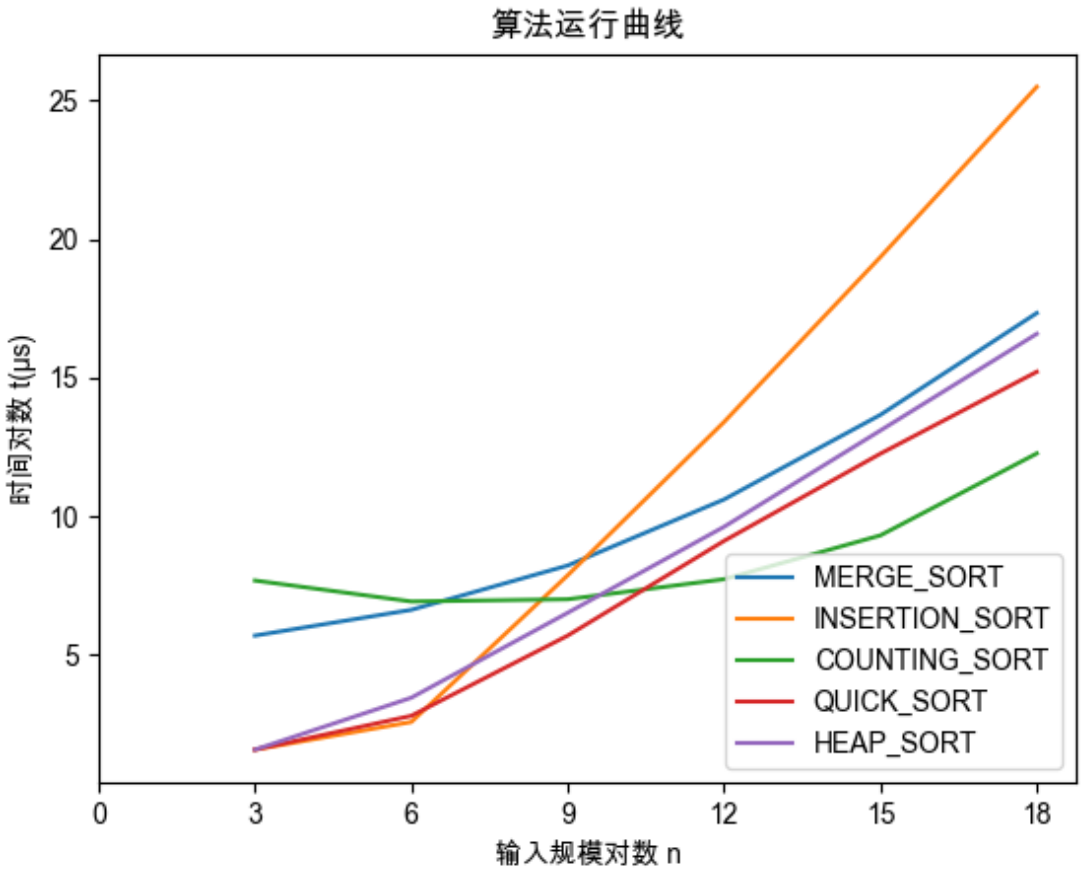
堆排序、快速排序、归并排序 $\Theta(nlg(n))$

观察n 较大的情况，结合折线与拟合，可以认为与算法渐近性能相同。



计数排序复杂度其实为 O(k+n)，所以在 n << k 时，算法耗时的对数都在8~10左右，当 n=18 时，时间与规模才成正比。符合课本的讨论。

比较不同的排序算法的时间曲线，分析在不同输入规模下哪个更占优势?



n = 3,6,9,12的情况都是快速排序效果较好（插入排序和堆排序都有优势）；

n = 15,18的时候计数排序更占优势。