数据结构 第一次作业

徐浩博 2020010108

1. 请写出下列操作的时间复杂度

(1) $a+b(a,b\in R)$

$O(1)$

(2) 二分搜索

$O(\log n)$

(3) 数组最大值

$O(n)$

(4)

```
for ( i = 0; i < n; i++ )
  for ( j = 0; j <= i; j++ )
    sum += a[i][j];
```

$O(n^2)$

(5) 选择排序

$O(n^2)$

(6) 矩阵乘法

$n\times m$ 与 $m\times k$ 矩阵相乘 $O(nmk)$

2. 对一个初始为空的向量依次执行 insert(0, 4), insert(1, 2), put(0, 1), remove(1), insert(0, 8) 后的结果是 8,1 。

3. 请自行举例分析并判断下列结论的正确性

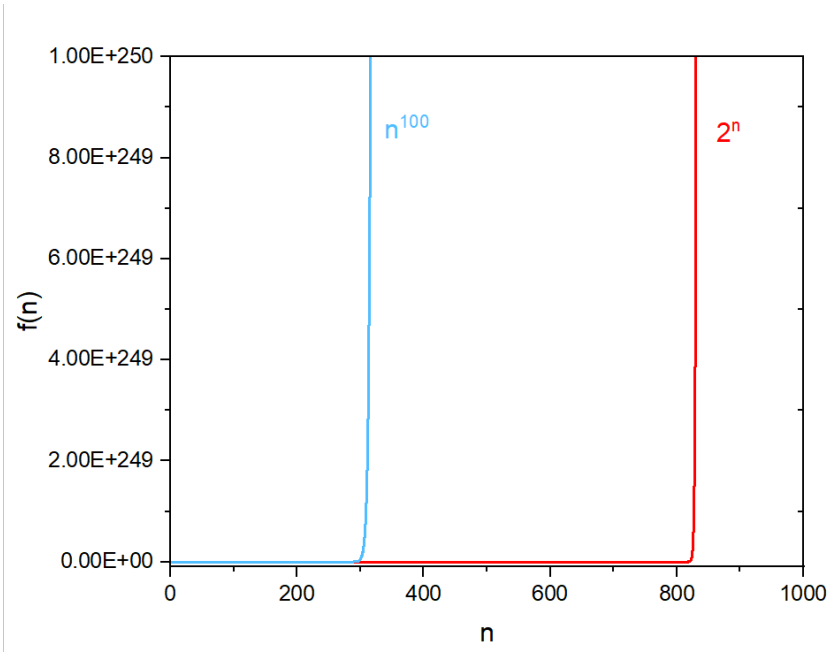
(1) 2^n 的复杂度高于 n 的任意系数的多项式

解：可以证明 $\lim_{n\to\infty}\frac{2^n}{p(n)}=+\infty$，其中 $p(n)$ 是关于n的多项式 ，因此存在 n_0，使得对任意 n_0，有 $2^n\geqslant p(n)$，因此由渐进复杂度的定义，2^n 的渐进复杂度高于 n 的任意系数的多项式。

然而实际上，当 n 的系数、指数较大且 n 较小时，在一定范围内，仍有 n 多项式时间开销大于 2^n。下图对比了 2^n 与 n^{100} 的函数图像，虽然通过数学的方法可以得出在 $n\geqslant 1e6$ 时恒有 $2^n\geqslant n^{100}$ 的关系 [1]，但由图像能够明显看出，在 $n\leqslant 800$ 的范围内时间开销上有 $2^n\leqslant n^{100}$。

[1] 欲使 $2^n\geqslant n^{100}$，只需 $n\ln 2\geqslant 100\ln(n)$，而 $0.5\ln(n)<n^{0.5}$，故只需 $n^{0.5}\geqslant 200/\ln 2$，因此可取 $n\geqslant 1e6$，此时必有 $2^n\geqslant n^{100}$.

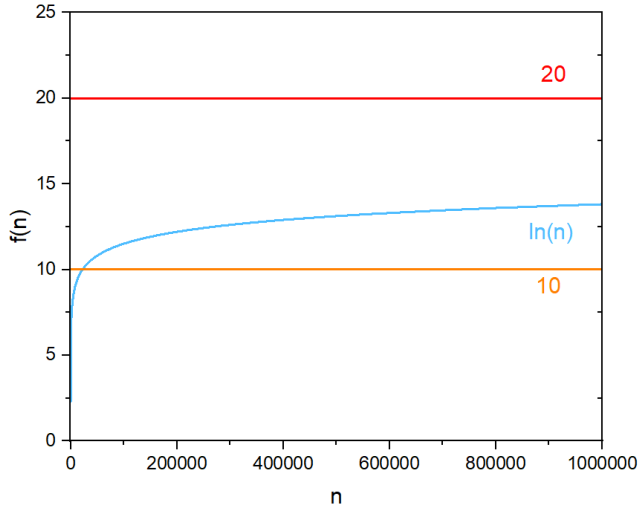
因此，虽然 2^n 的渐进复杂度高于 n 的任意系数的多项式，但在实际问题中，时间开销是否一定更高，仍需要再具体分析。



(2) $\log(n)$ 的渐进复杂度显著高于 1

解：由渐进复杂度定义，对于任何实数 $c \in \mathbf{R}$，均存在 n_0 使得任意 $n \geq n_0$ 均有 $\log(n) \geq c$，因此 $\log(n)$ 渐进复杂度大于 $O(1)$。

但本题询问“是否显著大”，而这一点需要具体问题具体分析。下图绘制的是 $\ln(n)$ 的图像，虽然能够看出当 $n \geq 5e4$ 时即有 $\ln(n) \geq 10$，但即使到了 $n \approx 1e6$ 的情况下，仍有 $\ln(n) \leq 20$，可以认为时间开销仍没有显著大于 1。然而，若将 $\ln(n)$ 换作 $10000\ln(n)$，那么在一定程度上可以认为时间开销显著大于 1。因此，针对“logn 渐进复杂度是否显著高于 1”这个问题，仍需要结合具体情况具体分析。



4. 请判断下示程序的复杂度，并说明其功能

```
__int64 func(int n){
    return (2 > n) ? (__int64)n : func(n-1) + func(n-2);
}
```

解：

时间复杂度　递归方程：

$$T_n = T_{n-1} + T_{n-2} \tag{1}$$

$$T_2 = T_1 = 1 \tag{2}$$

利用待定系数法将(1)式变形:

$$T_n + \lambda T_{n-1} = c(T_{n-1} + \lambda T_{n-2})$$

其中$c\lambda = 1, c - \lambda = 1$，可解得$c_{1,2} = \frac{1\pm\sqrt{5}}{2}, \lambda_{1,2} = \frac{-1\pm\sqrt{5}}{2}$，于是可得：

$$T_n + \lambda_1 T_{n-1} = c_1(T_{n-1} + \lambda_1 T_{n-2}) = c_1^{n-2}(T_2 + \lambda_1 T_1) = c_1^{n-2}(1 + \lambda_1) \tag{3}$$

$$T_n + \lambda_2 T_{n-1} = c_2(T_{n-1} + \lambda_2 T_{n-2}) = c_2^{n-2}(T_2 + \lambda_2 T_1) = c_2^{n-2}(1 + \lambda_2) \tag{4}$$

由$\lambda_2 \times (3) - \lambda_2 \times (4)$得：

$$(\lambda_2 - \lambda_1)T_n = \lambda_2 c_1^{n-2}(1 + \lambda_1) - \lambda_1 c_2^{n-2}(1 + \lambda_2)$$

从而

$$T_n = \frac{\lambda_2 c_1^{n-2}(1 + \lambda_1) - \lambda_1 c_2^{n-2}(1 + \lambda_2)}{\lambda_2 - \lambda_1} = O(c^n)$$

即程序为指数级别的时间复杂度$O(c^n)(c \in \mathbb{R}_+)$.

5. 对长度为N的常规向量做顺序查找，其中αN次查找成功（0≤α≤1），请计算平均查找长度。

解：

考虑到向量共有N个元素，假设查找成功时查找到每个元素的概率是等可能的，即$p_i = \frac{1}{N}, \forall i = 1, 2, ..., n$，因此查找成功时的平均查找长度为

$$ASL = \sum_{i=1}^{N} i \times p_i = \sum_{i=1}^{N} \frac{i}{n} = \frac{N+1}{2}$$

则对于查找成功的αN个元素来说，平均查找长度为$\frac{N+1}{2}$，而对于查找不成功的$(1-\alpha)N$个元素来说，查找长度均为N. 综合二者，总查找长度为$\frac{1}{2}\alpha N(N+1) + (1-\alpha)N^2$，平均查找长度$\frac{1}{2}\alpha(N+1) + (1-\alpha)N$.

6. 下示 mergeSort() 算法即使在最好情况下仍需要$O(n\log n)$时间运行。试修改代码，使之在（子）序列已有序时仅需线性时间，并简单解释。

```
template <typename T>
void Vector<T>::mergeSort ( Rank lo, Rank hi ) {
if ( hi - lo < 2 ) return;
   int mi = ( lo + hi ) / 2;
   mergeSort ( lo, mi ); mergeSort ( mi, hi );
   merge ( lo, mi, hi );
}

template <typename T>
void Vector<T>::merge ( Rank lo, Rank mi, Rank hi ) {
   T* A = _elem + lo;
   int lb = mi - lo; T* B = new T[lb];
   for ( Rank i = 0; i < lb; B[i] = A[i++] );
   int lc = hi - mi; T* C = _elem + mi;
   for ( Rank i = 0, j = 0, k = 0; (j<lb)||(k <lc);) {
      if ( ( j < lb ) && ( ! ( k < lc ) || ( B[j] <= C[k] ) ) ) A[i++] = B[j++];
      if ( ( k < lc ) && ( ! ( j < lb ) || ( C[k] <  B[j] ) ) ) A[i++] = C[k++];
   }
   delete [] B;
}
```

解：修改 mergeSort 函数为：

```
1  template <typename T>
2  void Vector<T>::mergeSort ( Rank lo, Rank hi ) {
3      if ( hi - lo < 2 ) return;
4      int mi = ( lo + hi ) / 2;
5
6      bool sortFlag = false;
7      for(Rank i = lo + 1; i < hi; i++){
8          if(_elem[i - 1] > _elem[i]){
9              sortFlag = true;
10             break;
11         }
12     }
13     if(sortFlag == false)return;
14
15     mergeSort ( lo, mi ); mergeSort ( mi, hi );
16     merge ( lo, mi, hi );
17 }
```

具体修改内容为：**在 mergeSort 内新增上图 6-13 行的内容**。当程序递归运行到 lo-hi 部分的子序列时，加入一个判断该序列是否有序的机制，如果序列有序，则不

再进行该序列子列的排序，直接退出函数。

首先，加入这个机制使得有序（子）序列只消耗线性时间：这是因为判断有序只需要遍历子序列内容；发现有序则不进行后续排序，因此如果有序，时间开销只有线性的遍历。

其次，该机制没有提高算法原先的复杂度。算法原先的 merge 函数需要合并两个子序列的元素，时间开销是线性的；而判断是否有序的机制最差的时间复杂度也是线性的，因此并没有提高算法原先的复杂度，反而可能因为减少了有序序列的排序这一时间开销而降低时间复杂度。

另外还有一种做法，修改 merge 函数为：

```
1  template <typename T>void Vector<T>::merge ( Rank lo, Rank mi, Rank hi )
2  {
3      if(_elem[mi - 1] <= _elem[mi])return;
4      T* A = _elem + lo;
5      int lb = mi - lo;
6      T* B = new T[lb];
7      for ( Rank i = 0; i < lb; B[i] = A[i++] );
8      int lc = hi - mi; T* C = _elem + mi;
9      for ( Rank i = 0, j = 0, k = 0; (j<lb)||(k <lc);)
10     {if ( ( j < lb ) && ( ! ( k < lc ) || ( B[j] <= C[k] ) ) ) A[i++] = B[j++];
11     if ( ( k < lc ) && ( ! ( j < lb ) || ( C[k] < B[j] ) ) ) A[i++] = C[k++];}
12     delete [] B;
13 }
```

具体修改内容为：**在 merge 内新增上图第 3 行的内容**，这样可以直接合并子列。在该序列有序时（假设该序列长度为 n），若把递归过程化作一棵二叉树，则第 k 层有 2^k 个子列，则此层计算次数为 2^k；计算总递归过程，一共的计算量为 $1+2+2^2+\cdots\cdots+2^{(\log_2 n)}=O(n)$，为线性复杂度。