

**1. OOP****02**

如果你没有专门学习过面向对象编程，可重点针对以下关键词查阅资料自学：

对象/属性/方法、类/构造/析构、继承/派生、重载、重写、虚方法、模板

**2. STL Vector****02A2**

- 试仿照讲义中的示例，学习使用STL中的vector，并逐步了解它的更多接口，乃至更多的ADT；
- 试设计若干计算任务，对vector的计算效果做一测试。

**3. DSA Vector****02A3**

- 在实例代码包中找到Vector对应的项目，编译并运行对应的测试程序；
- 阅读主测试程序的代码，了解测试的**逻辑与过程**；
- 对照测试的每个环节，阅读对应算法的代码，进一步理解测试所报告的结果。

**4. Amortization****02B[1-2]**

expand()扩容的比例如果换成其他比例，比如1.5倍，是否依然可以保证分摊的 $\mathcal{O}(1)$ 效率？

**5. Amortization****02B[1-2]**

我们知道，对expand()的调用可分为两种情况：**无需**扩容时直接返回，以及**真地**进行了扩容。而且，“expand()分摊时间复杂度为 $\mathcal{O}(1)$ ”的结论，是相对于对expand()的**所有**调用来做均摊而得的。

那么，为什么不能仅相对于**后一类**情况来做均摊？

**6. shrink()****02B[1-2]**

- 对照讲义的分析，阅读并理解Vector::expand()算法；
- 进一步阅读和理解Vector::shrink()算法（尽管讲义中并未对它做更多讲解）；
- shrink算法为何将**装填因子**的阈值设为1/4，而不是与expand算法**相对应**地取作1/2？

**7. remove()****02C1**

在讲义中，我们选择了先实现向量的**区间删除**接口，然后再借用它来实现**单元素删除**接口。

我们为何不倾向于反过来，也就是说：先实现后者，再通过反复地调用它来实现前者？

**8. remove(lo, hi)****02C1**

在向量的**区间删除**算法中，我们特殊处理了lo == hi的情况。

- 试确认，即便不做这一特判，算法依然能够应对这种特殊情况；
- 既如此，我们为何还是倾向于对其做特判处理？

**9. memcpy() & memmove()****02C1**

在Vector::insert()及Vector::remove()中，时间成本主要消耗于后缀的移动。

- 示例代码是通过显式的循环来完成移动的，如果颠倒循环的方向，可能会有什么问题？何时没问题？
- 如果改为调用memcpy()或memmove()来实现上述移动，需要注意什么方面？

**10. find()****02C2**

我们看到，Vector::find(e, lo, hi)算法约定，返回lo-1当且仅当查找失败。

- a) 那么特别地, 当 $lo=0$ 时, 返回的秩就是 $0 - 1 = -1$ ?
- b) 作为unsigned int的Rank, 如何才能与-1做比对?

### 11. find() Using Sentinel

02C2

在`Vector::find(e, lo, hi)`算法中, while循环的每一步迭代都会涉及两次比较: 一次是在**秩与秩**之间( $lo < hi--$ ), 另一次是在**元素与元素**之间 ( $e != \_elem[hi]$ )。实际上, 通过整体增加1次后一类比较, 可以省略掉**所有**的前一类比较。具体的思路是, 在进入循环之前先备份`_elem[lo-1]`, 然后将e的副本放入`_elem[lo-1]` (称作**哨兵**), 待退出循环后再予复原——这样, 同样能保证返回 $lo-1$ 当且仅当查找失败。

- a) 为此实现这一思路, 向量结构的总体设计还有哪些方面需要调整?
- b) 试实现这一新方法, 并通过测试验证实际的改进效果。

### 12. Vector::dedup()

02C3

本节介绍的无序向量去重该算法, 计算的成本主要来自于迭代循环中对每个元素的**find()**及 (可能的) **remove()**操作, 我们分别称之为**F类**、**R类**成本。

- a) 试确认, 再最坏情况下每一步迭代中的这两类成本都是 $O(n)$ , 故总体为 $O(n^2)$ 复杂度;
- b) 试确认: 尽管在单步循环中, F类成本有可能只是 $O(1)$ , 但对应的R类成本却必是线性的;
- b) 对称地试确认: 若某步循环中的R类成本为 $O(1)$  (即`remove()`并未执行), 则对应的F类成本必是线性的;
- c) 试改进该算法, 在不改变**简而治之**的整体策略与逻辑、仍通过调用**find()**来检测重复的前提下, 使得无论如何, R类成本累计都不会超过 $O(n)$ 。
- d) 如此改进之后尽管F类成本总计依然是 $O(n^2)$ , 但**实际**的运行速度为何会有极大提高?
- e) 如何**实质地**提高该算法的效率呢?

### 13. Function Object

02C4

- a) 针对函数对象Increase, 阅读示例代码包Vector项目中对应的代码, 理解**函数对象**的形式与机制;
- b) 试仿照示例代码中的形式, 编写更多的函数对象, 实现对向量**不同功能**的遍历 (比如减一、加倍、求和等)。

### 14. uniquify()

02D1

在本节我们看到, 通过精巧的设计, 可以在 $O(n)$ 时间内完成**有序**向量的去重操作, 且空间成本仅为 $O(1)$ 。

- a) 试确认: 在遇到**第一个**重复元素之前, 循环中的移动操作都是**徒劳无益**的——不过是自己到自己的赋值;
- b) 针对这一问题, 你有什么**改进**的思路? 试实现之。      c) 这种改进是否值得?

### 15. binSearch() & fibSearch()

02D[2~5]

考查二分查找、Fibonacci查找算法的各种版本。

- a) 它们都以搜索区间收缩至1甚至0作为迭代退出的条件 (对应于递归版本, 即相当于递归基)。试分别验证, 无论如何迭代退出时, 它们**最终**的区间宽度的确为1或0;
- b) 试证明: 无论何种算法, 失败情况总是比成功情况多出一**种**;
- c) 对照讲义中的实例, 验证成功、失败、平均情况下的**查找长度** (Search Length, SL, 即元素比较的次数);
- d) 试自己设计若干实例, 熟悉**平均查找长度** (Average Search Length, ASL) 的计算。

**16. Sorted By Probability****02D[2-5]**

讲义中主张，在二分查找中将概率最低的分支——在当前轴点 $[mi]$ 处命中——放在测试判断序列的最后。

- 试以32位无符号整数为例，估计出在长度为 $10^6$ 的查找表中查找成功的概率；
- 如果将这一分支挪到最前端，平均查找的时间会增加多少倍？——尽管渐近复杂度并无影响。

**17. fibSearch()****02D3**

- 试确认，本节实现的fibSearch()算法，在有多个命中元素时，不能保证返回秩最大者；在失败时，只是简单地返回-1，而不能指示失败的位置；
- 试参照稍后02D5的思路与技巧来改进fibSearch()，使其返回值的**语义**与binSearch()版本C完全一致。

**18. binSearch(): Version B****02D4**

本节指出，只需将二分搜索算法**版本B**的返回值调整如下，即可使之符合统一的语义：

```
return e < S[lo] ? lo-1 : lo;
```

- 试选择若干实例，验证这一结论；
- 试证明该结论的确成立；
- 如此调整之后，调用者应如何判定查找失败的情况？相应地，为此需要额外花费多少时间？

**19. binSearch(): Version C: Correctness****02D5**

本节通过建立循环不变性，证明了该算法的正确性。你还有其他证明方法吗？

**20. binSearch(): Version C: Average Search Length****02D5**

本节指出，该算法总是满足： $ASL_{succ} = ASL_{fail} = O(1.00 \cdot \log n)$

- 试证明，这一结论对于任何 $n = 2^k - 1$ 都成立；
- 试证明，这一结论对任何 $n > 0$ 都成立。

**21. intSearch()****02D6**

即便是有序序列，数据的分布也可能很畸形。试举出这样的实例，导致插值查找算法的复杂度过高。

**22. intSearch()****02D6**

本节针对均匀分布的数据，给出了轴点的估算公式。如果换成其它的分布形式，比如高斯分布呢？

**23. Interpolation/Binary/Sequential Search****02D[2-6]**

课上指出，长度不同的查找表，适用的查找算法也不同。大体来说，随着有效范围的长度递减，应当以接力的方式，依次采用插值查找、二分查找、顺序查找。当然，这只是一个定性的趋势和原则，应当结合具体的应用与硬件平台来确定它们之间的准确分界线。

- 试在你的电脑，设计实验并通过实验数据确认，何时应当从二分查找切换至顺序查找；
- 如果有可能，可针对更多平台及应用数据进行实验，得出对应的结果。

**24. bubblesort()提前终止版****02E**

试在示例代码包中找到该算法对应的项目：

- 适当修改测试程序并自行设计测例，了解其中sorted标志的演化过程，验证该算法的**逻辑**正确无误；
- 对足够大的数据做实测，估计出这一改进的平均效果（比如，**平均**能节省多少趟扫描）。

**25. bubblesort()跳跃版****02E**

试在示例代码包中找到该算法对应的项目：

- a) 适当修改测试程序并自行设计测例，了解其中last标志的演化过程，验证该算法的**逻辑**正确无误；
- b) 对足够大的数据做实测，估计出这一改进的平均效果（比如，**平均**能节省多少趟扫描）。

**26. mergesort() ~ merge()****02F2 + 02C3 + 02D1**

该算法的实质计算无非是二路归并，讲义及示例代码中该算法的实现虽然表面上很简洁，但由于反复地申请（new）和释放（delete）空间，时间复杂度的常系数其实颇高。

- a) 你认为可以如何减少这类**动态**内存操作，以降低复杂度的常系数？
- b) 试编程实现你的办法，并做实测对比。
- c) 试确认：至此，不难照如下策略将Vector::dedup()的复杂度改进至 $\mathcal{O}(n \log n)$ ：  
先调用mergesort()将无序向量转化为有序向量，再调用uniquify()完成去重。
- d) Vector::dedup()的复杂度，可否进一步优化？如果可以，试给出具体算法；否则，试说明理由。

**27. merge()****02F2**

示例代码中实现的二路归并算法，似乎只处理了**A**先耗尽的情况，**B**先耗尽呢？

**28. merge()****02F2**

我们注意到，即便待归并的两个序列已经**完全有序**，示例代码中实现的merge()仍是墨守成规，花费线性的时间以完成“归并”。

- a) 试改进merge()算法，使之能在上述情况下及时返回；
- b) 为此你的merge()需要多花多少时间？
- c) 如此多出的时间，相对于所得收益孰大孰小？

**29. In-place Mergesort In STL****02F3**

STL中的归并排序对merge()算法做了调整，从而彻底避免使用更多的辅助空间。

- a) 试查阅相关资料，理解其原理及实现细节；
- b) 这种调整付出了什么代价？
- c) 相对于空间方面的收益，这些代价是大是小？为什么？

**30. Bitmap****02G**

- a) 试在示例代码包中找到该算法对应的项目，着重阅读并理解其中利用位运算（bitwise operation）实现比特位定位的原理及过程；
- b) 对照02B1中的Vector::expand()算法，阅读Bitmap::expand()的代码并理解其实现原理；
- c) 为保证常数时间内完成Bitmap的整体复位，02G3节介绍了Hopcroft的校验环技巧。试从算法演示包中找到对应的Excel版演示，通过实践体会和理解其原理及过程；
- d) 如果只是着眼于在常数时间内整体复位，其实还有其它方法，你能想出来一种吗？
- e) 相对于Hopcroft的方法，你的方法有何优势和劣势？
- f) 自学C++模板类bitset，并从接口于功能的角度，与课上讲授的Bitmap做一对比。

**31. Vector<int>::dedup() With A Bitmap****02G2**

- a) 试借助Bitmap, 实现一个对整数向量的去重新算法;
- b) 设整数的取值范围为 $[0, m)$ , 向量长度为 $n$ , 你的新算法的时间、空间复杂度各是多少?
- c) 试验证, 你的算法不仅能够实现去重, 同时也能完成排序。

**32. Bitmap::clear() By Rotate Or Shift****02G3**

在Hopcroft改进的Bitmap中, 清除一个元素对 $F[]$ 数组而言直截了当, 但对 $T[]$ 来说却比较复杂。我们在课上看到, Hopcroft以一种**旋转**的方式, 将末元素**前置并顶替**被删除者, 从而在 $\mathcal{O}(1)$ 时间内实现了删除。

- a) 试按照常规的后缀**前移**的思路来编程实现`Bitmap::clear()`;
- b) 这种实现方式的复杂度如何? 包括程序的运行时间, 以及编写和调试程序所消耗的时间;
- c) 对比两种思路, 反过来体会Hopcroft技巧的高明之处。

**33. Vector<int>::dedup() With Hopcroft's Bitmap****02G3**

- a) 试说明, 当整数的取值范围很大 ( $m > n$  甚至  $m \gg n$ ) 时, 借助Hopcroft的Bitmap可将向量去重算法的效率从 $\mathcal{O}(n + m)$ 优化到 $\mathcal{O}(n)$ , 也就是说运行时间与取值范围无关 (尽管仍与空间复杂度相关)。
- b) 相对于使用常规的Bitmap, 作为补偿, 这种方法有何倒退?