向量

概念：ADT

Abstract Data Type vs. Data Structure

抽象数据类型 = 数据模型 + 定义在该模型上的一组操作			
	抽象定义	外部的逻辑特性	操作&语义
	一种定义	不考虑时间复杂度	不涉及数据的存储方式
数据结构 = 基于某种特定语言，实现ADT的一整套算法			
	具体实现	内部的表示与实现	完整的算法
	多种实现	与复杂度密切相关	要考虑数据的具体存储机制

Application = Interface * Implementation

向量是对数组的抽象与泛化。

换句话说，我们用数组实现向量的 ADT 。

核心

call by rank 的访问模式的封装。

动态空间管理——扩容方法

在即将上溢的时候适当扩大【扩大一倍】内部数组的容量

复杂度分摊下来是 $O(1)$ 的。

代价：装填因子有所下降

基本操作

元素访问

重载 `[]` 运算符，$O(1)$ 。

插入

若有必要，扩容；

从后往前依次后移；

置入新位置；

更新容量与秩，$O(n)$

区间删除

依次前移动，$O(n)$

单元素删除

一个前移动，$O(n)$

无序向量的操作

查找 $O(n)$ 、去重 $O(n^2)$ 、遍历 $O(n)$

有序向量的操作

唯一化（去重）

image-20230205111645831

算法复杂度为O(n)

查找

评估查找的性能

平均查找长度.

二分查找

复杂度：$O(\log n)$.

版本A：和中间元素比一次小于，再比一次等于， $O(1.5\log n)$ 。但是失败的话可能会更长一点。

版本B：不跟中间比一次等于，直接划分成两半。

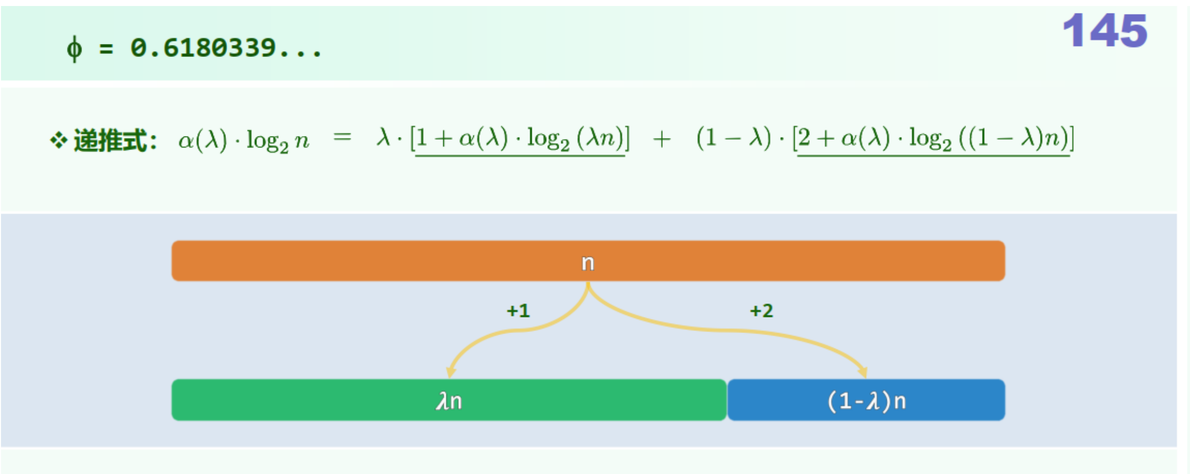
版本C：确保是 lower_bound，区间宽度缩减到 0 的时候才结束递归。这种情况下的循环不变性体现在 `A[0, lo) <= e < A[hi,n)` 。

对于版本A，设查找成功情况下平均比较次数为$\mathcal{S}$，失败情况下平均比较次数为$\mathcal{F}$，二者满足关系$(\mathcal{S}+1)\cdot n=\mathcal{F}\cdot(n+1)$，该结论可以通过对树高进行归纳证明。（习题2-18）

Fibonacci 查找

往左侧走只需要一次比较，往右侧走却需要两次比较，因此适当的将左侧的长度调长一些来缩小总代价。

最后用数学方法可以算出来应该是黄金分割，$1.618:1$。



插值查找

在字长意义下的折半查找。根据分布律查找。

复杂度分析：没经过依次查找，待查找区间的宽度从 n 缩短到 $\sqrt{n}$ 。也就是，有效字长 $\log n$ 每次折半，所以复杂度 $O(\log\log n)$ 。

但是，会引入乘法和除法，所以会不好。

最好的方法应该是：

- 首先插值查找，快速缩短查找区间；
- 然后二分查找，进一步缩短；
- 最后顺序查找，充分利用缓存等优势。

应用

位图(bitset)

就是一个形式上的 bool 的 vector 啦，在 C++ 中使用 `vector<bool>` 即可获得。

可以使用位运算“加速”。

快速初始化——校验环

不能保证初始内容的情况下，如何不需要初始化？ J. Hopcraft, 1974.

把 bitset 拆成两个等长的 rank 向量 `F` 和 `T`，原来的 bitset 中使用的位置 `k` 均满足 `T[F[k]] = k, F[T[k]] = k` 。

这样：

- `reset` 就只需要把 `T` 的 `top` 置作 0 。
- `set` 就是把 `T` 的 `top++`；
- `clear` 就是把 `top` 位置的环覆盖到 `k` 对应的环，然后给 `--top`

排序

起泡排序

时间 $O(n^2)$ ，额外空间 $O(1)$ ，稳定（只有相邻元素才能交换，但也得看实现）。

交换次数就是序列逆序对数目，因为一次交换逆序对数目恰好减一

思路

从左往右扫描，逐个交换，将最大的元素到最右侧。

归并排序

J.von Neumann, 1945.

思路

- 序列一分为二；
- 子序列递归排序；
- 合并有序子序列。

时间复杂度根据 主定理 可以计算出为 $O(n\log n)$ ，额外空间复杂度 $O(n)$ ，稳定。

优势：

- 完全顺序访问，不需要随机读写；
- 可扩展性好；
- 可并行计算；

缺点：

- 非就地
- 最好情况不好。

逆序对计算

分治之后，递归计算内部的；

复杂度分析

比较/代数判定树

针对比较-判定类型算法的计算模型。

给定输入的规模，将所有可能的输入所对应的一系列判断表示出来；

代数判定，使用某一常次数代数多项式，将任意一组关键码作为变量，对多项式求值；根据结果的方向，确定算法的推进方向。

Comparison Tree 是最简单的 Algebra Decision Tree，使用了二元一次多项式 $K_i - K_j$；比较树是三叉树；

每一个叶子节点各对应于

- 起自根节点的一条道路
- 某一可能的运行过程
- 运行所得到的输出

叶节点深度 ~ 比较次数 ~ 计算成本；树高 ~ 最坏情况时的计算成本

树高的下界 ~ 所有 compare based algorithm 复杂度的下界?

对于排序算法所对应的 代数判定树，必然有 $N \geq n!$；(因为每一个叶子都要对应着一种情况！)

所以，$\log_3 N \geq \log_3 n! = \log_3 e \cdot [n \ln n - n + O(\ln n)] = \Omega(n \cdot \log n)$

这里用到了一个逼近公式，也就是 $n! \sim \sqrt{2\pi n} \cdot (n/e)^n$