1. Contructing An Interval Tree **07A**

在`Interval Tree`的构造过程中，我们也需要反复地找到端点中的**Median**。不难看出，为了保证整体$\mathcal{O}(n\log n)$的时间复杂度，每个Median都必须在线性时间内找到。除了调用14章将要介绍的（在线性时间内找到Median的）算法，你在此处还有其他什么更为**简明**的解决方案？

2. Contructing An Interval Tree **07A**

讲义中指出，可以在$\mathcal{O}(n\log n)$时间内构造`Interval Tree`，你认为是否有渐近复杂度**更低**的构造算法？试**设计**这么一个算法，或者**证明**这样的算法不存在。

3. Balance Of An Interval Tree **07A**

课上指出，既然在构造`Interval Tree`的过程中我们总是在Median处做划分，子树中存放的区间数量至多为父亲的一半，故知树的高度不会超过$\mathcal{O}(\log n)$，`Interval Tree`必然是渐近平衡的。那么进一步地，这种平衡性是否如AVL树那样意味着，任意一对**兄弟**子树中所存放区间的数量之差，不会超过某个**上限**？

4. Querying An Interval Tree **07A**

我们看到，对于任何查询位置，`Interval Tree`的查询算法都能**确定**$\mathcal{O}(\log n)$个节点，并分别从它们各自所预存的线段中**分拣**出一个子集，这些子集之**并**恰好就是所有命中的线段。为此，需要遍历各节点的**左**端点有序列表或**右**端点有序列表：在此过程中的每一步都能**报告**一条命中线段，除了最后一步因越界而**失败**。那么，**这类**失败的访问，为何没有在**整体**$\mathcal{O}(r+\log n)$的复杂度中体现出来？

5. Segment Tree **07B**

我们知道，为构造`Segment Tree`，需要将各条线段依次插入其中，而插入过程可以**递归**地描述和实现。试证明：只有在第一个存入该线段的节点处，才可能发生**二分**递归；而在其余位置，必然都是**线性**递归。

6. Segment Tree **07B**

试证明：在`Segment Tree`的每一层上，每一条输入线段至多会在**两个**节点中存放各一份——从而每一条线段至多占用$\mathcal{O}(\log n)$空间，整体的空间复杂度不过$\mathcal{O}(n\log n)$。

7. Segment Tree **07B**

试证明：沿着在**线段树**中的每一条搜索路径，每一条输入线段至多在某一个节点存放一份——从而在每次`search()`的过程中，途经各节点所报告的线段**子集**彼此**无交**；在这方面消耗的时间确实为$\mathcal{O}(r)$。

8. Stabbing Counting **07[A+B]**

穿刺**计数**是穿刺查询的特例：只需要统计命中线段的**总数**，而不必逐一枚举或统计。

试一个算法，在经过$\mathcal{O}(n\log n)$时间的**预处理**后，得到一个占用$\mathcal{O}(n)$空间的数据结构，此后便可以在$\mathcal{O}(\log n)$时间内完成每一次穿刺计数查询——与命中点集的规模**无关**。

9. Range Counting **07C[1-2]**

范围**计数**是范围查询的特例：只需要统计落在查询范围内的**点数**，而不必逐一枚举或统计。

a) 试针对一维点集设计一个算法，在经过$\mathcal{O}(n\log n)$时间的**预处理**后，得到一个占用$\mathcal{O}(n)$空间的数据结构，此后便可以在$\mathcal{O}(\log n)$时间内完成每一次范围计数查询——与命中点集的规模**无关**；

b) 对于**二维**点集，是否也有类似的算法？

10. Range Sum 07C[1-2]

范围**求和**也是范围查询的特例：如果每个点都有一个**数值**属性，需要统计查询范围内所有点的总和。

a） 试针对一维点集设计一个算法，在经过$\mathcal{O}(n\log n)$时间的**预处理**后，得到一个占用$\mathcal{O}(n)$空间的数据结构，便可以在$\mathcal{O}(\log n)$时间内完成每一次范围求和——与命中点集的规模**无关**；

b） 对于二维点集，是否也有类似的算法？

11. Lower Bound Of 1D Range Query 07C1

讲义中针对一维点集描述了一个算法，在经过$\mathcal{O}(n\log n)$时间的**预处理**后，得到一个占用$\mathcal{O}(n)$空间的数据结构，即可在$\mathcal{O}(r+\log n)$时间内完成每一次范围查询。试说明，这样的性能已经是**最优**的了。

12. 2D Range Counting/Sum By Inclusion-Exclusion Principle 07C2

本节描述了一种基于**容斥原理**，借助经预处理得到的一个矩阵来支持范围查询的方案。

a） 该算法的预处理需要多少**时间**？

b） 试确认，该方案确实需要使用$\mathcal{O}(n^2)$空间；

c） 试确认，该方案可以在$\mathcal{O}(\log n)$内支持范围计数（counting）；

d） 试确认，该方案可以在$\mathcal{O}(\log n)$内支持范围合计（summing）；

e） 如果范围查询需要列举出（report）命中的所有点，该方案是否依然可行？

13. Binary Search vs. BST 07D1

本节所使用的BST，与此前第06章介绍的版本有所不同：其中的每个内部节点，都是某个叶节点的复本。

a） 试确认，该版本的BST与此前的版本实质上彼此**等效**；

b） 试确认，这些版本的BST，分别对应于第02章所介绍二分查找的**某一**版本；

c） 试确认，这棵BST可在$\mathcal{O}(n\log n)$时间内构造出来。

14. Range Query By BBST 07D1

本节基于BBST描述了一个一维范围查询算法，试确认：

a） 沿横向介乎x_1、x_2所对应查找路径之间的所有子树，彼此**无交**；

b） 它们的**并集**，即是对应于$[x_1,x_2)$的查询输出；

c） 无论输出的规模多大，这些子树总共不会超过$\mathcal{O}(\log n)$棵。

15. Node ~ Canonical Subset 07D1

一维点集在被组织为BBST后，其中的每个节点x都自然对应于一个**覆盖子集**$Int(x)$——叶节点的**覆盖子集**仅包含其对应的那个**点**；父节点的覆盖子集是其左、右孩子之覆盖子集的**并集**。试确认：

a） 同深度节点所对应的覆盖子集，彼此**无交**；

b） 在任何一棵子树中，同深度所有节点所对应区间的**并集**都是一样的——即子树之根节点的覆盖子集。

16. Duplicates In A BST 07D1

在第06章引入BST结构时，出于简化的考虑，我们暂且假定了所有关键码互异。在了解了如何用BST来解决一维范围查询之后，试给出一种方法，无需对BST的节点和整体结构作任何改造，即可直接而有效地支持（关键码）**相等**的节点。具体地：

a) 试增加BST::search**First**(e)接口，在$\mathcal{O}(\log n)$时间内，从关键码为e的所有节点中，找出**最先**插入者；

b) 试增加BST::remove**First**(e)接口，在$\mathcal{O}(\log n)$时间内，从关键码为e的所有节点中，删除**最先**插入者；

c) 试增加BST::search**Last**(e)接口，在$\mathcal{O}(\log n)$时间内，从所有关键码为e的节点中，找出**最后**插入者；

d) 试增加BST::remove**Last**(e)接口，在$\mathcal{O}(\log n)$时间内，从所有关键码为e的节点中，删除**最后**插入者；

e) 试增加BST::count**All**(e)接口，在$\mathcal{O}(\log n)$时间内，统计出**所有**关键码为e的节点总数；

f) 试增加BST::search**All**(e)接口，在$\mathcal{O}(r + \log n)$时间内，列出**所有**关键码为e的节点（r为其总数）；

g) 试增加BST::remove**All**(e)接口，在$\mathcal{O}(r + \log n)$时间内，删除**所有**关键码为e的节点（r为其总数）；

h) 为配合上述接口，BST::insert(e)算法相应地需做什么调整？

17. Range Query ~ Node/Interval Classification 07D1

调用本节所描述算法每做一次范围查询，都等效于对BBST中所有节点做了**分类**。试确认：

a) 无非四类：Accepted、Rejected、Recursion、Unvisited；

b) 在任一深度上，前三类节点都不超过**常数**个——全树累计不过$\mathcal{O}(\log n)$个；

c) 所有Accepted节点所对应的覆盖子集，彼此无交；它们的**并集**，便是查询的输出。

18. Construction Of MLST 07D2

本节介绍了借助**多层搜索树**的范围查询算法。

a) 试描述二维MLST（包括X-树及所有关联Y-树）的**构造**过程；

b) 试证明，构造过程只需$\mathcal{O}(n \log n)$时间；

c) 试用**两种**方法证明，MLST占用的**空间**不超过$\mathcal{O}(n \log n)$。

19. Worst Case Of MLST::search() 07D2

a) 本节证明了二维MLST的查找时间不超过$\mathcal{O}(\log^2 n)$，试举（最坏）例说明这个界是**紧**的；

b) 本节也指出，d维MLST的查找时间不超过$\mathcal{O}(\log^d n)$，试证明这一结论，并举（最坏）例说明这个界是**紧**的。

20. Dynamic Range Tree 07E

本节介绍了如何借助**Fractional Cascading**技巧，将Range Tree的查找时间降至$\mathcal{O}(\log n)$。

a) 如果Range Tree还要支持数据点集的**插入**、**删除**，父子节点之间的快捷链接应当如何更新？

b) 这类**更新**需要耗费多少时间？

21. 2d-Tree ~ Quadtree 07F1

查阅资料自学四叉树（Quadtree）。

a) 确认该结构确实可以视作2d-Tree的一种**特例**；

b) 四叉树的构造及查询算法，相应地可以做哪些**简化**？

22. Median In kd-Tree 07F1

在构造kd-Tree的过程中，每次递归之前都要找到当前子集的中位数（Median）。

a) 尽管第14章将会介绍在线性时间内找到Median的算法，但就目前而言，我们可否通过**全局排序**之类的预处理，避免反复地做这种查找？

b) 如果每次都是平凡地通过**排序**来确定中位数，kd-Tree的构造将需要多少时间？

23. Iterative Construction Of kd-Tree **07F1**

本节描述了一个**自顶而下递归**构造kd-Tree的算法，该算法可否改写为**自底而上迭代**式的？

24. Recursion Bases Of bulidKdTree() **07F1**

讲义中的buildKdTree()算法，会一直递归到只剩一个点，但这并不必要。从降低实际运行时间的角度出发，往往是根据具体的硬件环境确定一个更大的阈值M（比如100），一旦子集规模降到M以下，便可不再递归，直接创建一个与该子集对应的叶节点。

a) 试编码实现这样的策略；

b) 在你的计算环境中通过反复测试，确定最佳的阈值M。

25. Storage Cost for kdSearch() **07F2**

我们看到，kdSearch()算法会一批一批地，以reportSubTree()的方式输出查询结果。

a) 为此，是否需要在预处理过程中，将每棵子树对应的部分解预先存放好？

b) 如果要预存这些信息，渐近的空间复杂度是否会增加？如果增加，会增加多少？

26. kd-Tree Using Bounding Box **07F2**

本节简略介绍了在kd-Tree中应用**包围盒**的原理，为此**构造**算法及**查询**算法需相应地做哪些调整？

27. Complexity Of 2d-Query **07F3**

为界定2d-Tree查询算法的时间复杂度，关键在于得到一个可行的**递推式**，试确认讲义中的递推式成立。

28. #Recursions **07F3**

考查与平面点集$\{\ (2i+1, 2j+1)\ \mid\ 0 \leq i, j \leq 7\ \}$对应的2d-树。

若$R\ =\ [4, 14] \times [6, 12]$，则使用该树完成范围查询的过程中总共会发生多少次**递归**（含递归基）？

（与讲义的惯例一致，这里也约定在构造该树时，首先按**x坐标**切分；每个区域都是**左开右闭**、**下开上闭**。）

29. Worst Cast Of 2d-Query **07F3**

a) 本节指出，2d-Tree的每次查询时间不超过$\mathcal{O}(r+\sqrt{n})$。试给出达到这一**上界**的**最坏**情况。

b) 试进一步举例说明，在输出量k很小甚至接近0时，仍有可能需要花费$\mathcal{O}(\sqrt{n})$时间。

30. kd_Tree vs. MLST **07F3**

抛开公共的用于输出的时间$\mathcal{O}(r)$，二维的kd-树、MLST用于搜索的时间分别是$\mathcal{O}(\sqrt{n})$、$\mathcal{O}(\log^2 n)$。就渐近意义而言，前者的确更大；但在n还不算太大时，情况可能**相反**。

a) 如果二者的常系数相同，试确定上述两种情况的**分水岭**；

b) 实际上因为原理清晰、实现简明，kd-树的常系数往往远小于MLST。

比如，如果小**十倍**，分水岭又会在哪里？