Lesson1

> **区间问题**
>
> 思路：
>
> - **排序**：左端点，右端点，双关键字排序
> - 试样例，找算法
> - 尝试证明
>
> **区间选点**
>
> 思路
>
> - 将每个区间按照右端点从小到大排序
> - 从前往后依次枚举每个区间
>   - 若当前区间已经包含点，则pass该区间（左端点 <= ed）
>   - 若当前区间内不包含点，尝试在**右端点**放置一个点（当前最好的情况，短视的行为）
>   - 因此，贪心用函数表示的话，函数图像是单峰的，通过局部最优值可以达到全局最优值
>
> 证明
>
> - 数学上证明两个值相等，分别证明 a<=b 与 a>=b。
> - 按照此选法，算法结束之后，每个区间一定包含一个点，因此当前选择方案一定是一个合法方案
>
>   即ans <= cnt
> - 第一个区间在右端点一定放置一个点，且下一个选择的区间的左端点一定和上一个区间没有交集。若一个选择了cnt个点，代表了cnt个区间，而这些区间两两之间没有交集。而若想把每个区间覆盖掉，至少需要cnt点，所以所有选择ans>=cnt
> - 所以综上所述，答案即为cnt
>
> 具体实现

```java
static int N = 100010;

static int n;
static Interval[] ivals = new Interval[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ivals[i] = new Interval(a, b);
    }

    //按右端点升序排列所有区间，Comparator函数 = sign(o1.r-o2.r)，1交换元素位置，0和-1则不变
    Arrays.sort(ivals, 0, n, (o1, o2) -> o1.r-o2.r);
```

```java

    int res = 0, ed = (int)-2e9;     //ed初始化当前所有区间最右点为-2e9
    for (int i=0; i<n; i++) {
        if (ivals[i].l > ed) {
            res++;
            ed = ivals[i].r;     //更新ed
        }
    }

    out.println(res);

    out.flush();
}

static class Interval {
    int l, r;

    Interval(int ll, int rr) {
        l = ll; r = rr;
    }
}
```

最大不相交区间数量

思路

- 和上述区间选点问题思路一致

证明：（分别证明ans<=cnt与ans>=cnt）

- 按这种方式，选出的区间两两之间一定没有交集，所以该选择方案是合法的，所以ans>=cnt
- （反证法证ans<=cnt）假设ans>cnt，即我们可以选择比cnt更多的两两没有交集的区间。那我们则最少需要ans个点才能覆盖掉所有区间，与已知结果矛盾，故ans <= cnt。
- 综上所述 ans = cnt

具体实现

```java
static int N = 100010;

static int n;
static Interval[] ivals = new Interval[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ivals[i] = new Interval(a, b);
    }

    //按右端点升序排列区间
    Arrays.sort(ivals, 0, n, (o1, o2) -> o1.r-o2.r);

```

```
    int res = 0, ed = (int)-2e9;     //与区间选点贪心思路一致
    for (int i=0; i<n; i++) {
        if (ivals[i].l > ed) {
            res++;
            ed = ivals[i].r;     //更新ed
        }
    }

    out.println(res);

    out.flush();
}

static class Interval {
    int l, r;
    Interval(int ll, int rr) {
        l = ll; r = rr;
    }
}
```

区间分组

思路：

- 将所有区间按照**左端点从小到大**进行排序
- 从前往后处理每一个区间

  判断能否将当前区间放到某个现有的组中。即判断**是否存在**（找最小值 ）某个组，满足组内所有区间右端点的最大值 **小于** 当前区间左端点（动态维护最小值，可以使用堆来做）。

  - 若不存在这样的组，则开一个新的组，并将当前区间放入新组
  - 若存在这样的组，则将其放入任意一个满足条件的组内，并更新当前组的Max_r。

证明： （ans=cnt <=> ans>=cnt, ans<=cnt）

- 按照上述方式得到的划分结果，一定是一种合法方案，因此ans <= cnt
- 假设一共有cnt个组，当新开最后一个组时，当前区间i一定与前面cnt-1个组都存在交集。即每个组的Max_r都大于等于Li，且前面所有组的所有区间的Lj <= 当前区间的左端点Li。故每个组内都至少存在一个区间满足Ljj <= Li，且Rjj > Ri。 因此我们可以找到cnt-1+1即cnt个区间存在公共点Li，因此不管怎么分组，这cnt个区间一定在不同的组内，因此ans >= cnt
- 所有 ans = cnt

具体实现

```
static int N = 100010;

static int n;
static Range[] ranges = new Range[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++) {
        ins.nextToken(); int l = (int)ins.nval;
        ins.nextToken(); int r = (int)ins.nval;
```

```
            ranges[i] = new Range(l, r);
        }

        //将区间按左端点从小到大排序
        Arrays.sort(ranges, 0, n, (o1, o2) -> o1.l-o2.l);

        //定义小根堆，记录各个分组中最大的右端点值，堆首元素即最大右端点的最小值
        PriorityQueue<Integer> heap = new PriorityQueue<>((o1, o2) -> o1-o2);
        for (int i=0; i<n; i++) {
            //堆空或右端点最小值>=range[i].l，新开一个分组
            if (heap.isEmpty() || heap.peek()>=ranges[i].l) heap.add(ranges[i].r);
            else {  // <range[i].l，加入最小值所在分组
                heap.poll();    //弹出堆
                heap.add(ranges[i].r);   //更新堆
            }
        }

        out.println(heap.size());

        out.flush();
    }

    static class Range {
        int l, r;
        Range(int ll, int rr) {
            l = ll; r = rr;
        }
    }
```

区间覆盖

思路：

- 将所有区间按照**左端点从小到大**排序。
- 从前往后以此枚举每个区间，在所有能覆盖start（需覆盖区间左端点）的区间中，选择右端点最大的区间。选完之后将start更新为右端点的最大值。当start >= end时，则选择结束。

证明：（ans=cnt <=> ans<=cnt && ans>=cnt)

- 按照上述选法，得到的结果一定能完整覆盖区间，因此ans <= cnt
- （调整法）将最优解方案与算法得到的方案，按照左端点从小到大排序。并从前往后找到第一个不一样的区间，**可以将算法选择的区间替换到最优解的对应区间上**，且并不会导致最优解结果的变化。进一步继续向后找到不一样的区间，并用算法得到的区间替换掉最优解中的区间。因此，可以通过该种方式，将最优解转换为算法得到的解，即ans = cnt

具体实现：

```
static int N = 100010;

static int n;
static Interval[] ivals = new Interval[N];

```

```java
public static void main(String[] args) throws Exception {
    int st, ed;
    ins.nextToken(); st = (int)ins.nval;
    ins.nextToken(); ed = (int)ins.nval;
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        ivals[i] = new Interval(a, b);
    }

    Arrays.sort(ivals, 0, n, (o1, o2) -> o1.l-o2.l);

    boolean flag = false;
    int res = 0;
    // 双指针算法
    for (int i=0; i<n; i++) {
        int j = i, r = (int)-2e9;
        while (j<n && ivals[j].l <= st) {
            r = Math.max(ivals[j].r, r);
            j++;
        }

        if (r < st) break;

        res++;
        if (r >= ed) {
            flag = true; break;
        }

        st = r;
        i = j-1;
    }

    if (flag) out.println(res);
    else out.println("-1");

    out.flush();
}

static class Interval {
    int l, r;
    Interval(int ll, int rr) {
        l = ll; r = rr;
    }
}
```

Huffman树

合并果子（注意与石子合并区别，该题能够合并任意两堆，没有位置限制）

思路：

- 每次数量最小的两堆果子进行合并，并将合并后的果堆加入堆中

- 重复上述过程，直至只剩下一堆

证明：

- 在所有的数里面，最小的两个数**一定**深度最深，且**可以**互为兄弟。

  （反证法）假设最小的两个数(a, b)不是最深的，则一定可以通过将a，b和深度最深的点进行交换，使树的总权值严格减少，不满足huffman树性质。故最小的两个数深度一定是最深。当最小两点a,b处于深度最深的位置时，交换同一层的结点并不更改结点的深度，即不会影响树的权值，故a，b可以互为兄弟。

- 贪心式合并能否得到全局最优解？ 如何证明？n-1的最优解是否时n的最优解？

  设f(n)表示合并n堆果子的最小值。则有f(n) = f(n-1) + （a + b）（由1知，a一定和b进行合并），由于a, b固定，所以对f(n)求解可以转换为对f(n-1)求解。进而证明贪心能得到全局最优解。

具体实现：

```java
static int n;
static PriorityQueue<Integer> heap = new PriorityQueue<>((o1, o2) -> o1-o2);

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    for (int i=0; i<n; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        heap.add(a);
    }

    int res = 0;
    while (heap.size() > 1) {
        int a = heap.poll();
        int b = heap.poll();
        res += a+b;
        heap.add(a+b);
    }

    out.println(res);

    out.flush();
}
```