倍增、RMQ

倍增

所有能用二分的题也可以用倍增来做。

假设某个闭区间$[l,r]$中左半边部分满足某个性质P，右半部分不满足，我们想找到满足P的最后一个数（这个数可以不存在）。当然可以用二分来做。现在介绍倍增做法：

倍增的思想是这样的：

1. 从$x=l-1$出发，先设步长$p=1$；
2. 如果$x+p$满足性质（当然在验证满足性质之前还要先判断一下有没有出界），我们就将x变为$x+p$，并且将步长加倍，即p变为$2p$；如果$x+p$不满足性质，则让p减半，x不变；
3. 重复2，直到$p=0$；
4. 循环退出时x即为答案。如果不存在满足性质的数，则会有$x=l-1$。

假设"验证性质"的时间复杂度是$O(f)$，那么整个算法的时间复杂度仅仅由步长p变了多少次决定。不妨设起点0，我们要找的位置是y，设k是使得$1+2+...+2^k\le y$的最大整数，那么$k\le \log_2(y+1)-1$，所以总体时间复杂度是$O(f\log y)$。

我们回想到二分的时间复杂度是$O(f\log n)$，看起来倍增似乎要快一点，但是，二分的`while`循环次数是严格的$\log n$的（我们的模板是这样），而倍增里循环次数事实上大致是$2\log y$，因为p会经历一个先变大后变小的过程，所以事实上，倍增只是在答案非常靠近左边的时候才会显著比二分优越。所以如果题目里的搜索范围非常大，并且我们能知道答案大概率出现在很左边的地方，那倍增就是个更优的选择。

例题：https://blog.csdn.net/qq_46105170/article/details/134114544

RMQ

上述的倍增算法用处不大，但是倍增的思想在很多地方都是非常有用的。我们现在介绍一种叫Sparse Table的数据结构，中文叫稀疏表。考虑如下问题：

给定一个长n数组a，再给定若干次询问，每次询问给出两个数$l\le r$，求$a[l,l+1,\dots,r]$的最小值。

可以先求一个二维数组$f[i][k]$，其定义为：$f[i][k] = \min a[i, i+1, \dots, i+2^k-1]$，即表示从$a[i]$开始的$2^k$个数里的最小值。我们考虑如何求$f[i][k]$。显然$f[i][0] = a[i]$，然后有递推式：
$f[i][k] = \min\{f[i][k-1], f[i+2^{k-1}][k-1]\}$。这样所有的$f$都能求出来了。

我们考虑求$a[l, l+1, \dots, r]$的最小值。我们先找最大的k使得$2^k < r-l+1$，那么$a[l, \dots, l+2^k-1]$和$a[r-2^k+1, \dots, r]$就覆盖了$a[l:r]$这一段，从而这一段的最小值就是$a[l, \dots, l+2^k-1]$和$a[r-2^k+1, \dots, r]$这两段的最小值，答案就是$\min\{f[l][k], f[r-2^k+1][k]\}$。

数学上来讲，上面的理论已经足够。代码上还有几点考虑：

1. 怎么迅速求2^k，这是很容易的，可以用位运算`1 << k`。
2. 怎么求最大的k使得$2^k < r-l+1$，这里可以调用`log2`的库函数，该库函数在`cmath`库里。但是细心的读者可能会想，如果直接调用`log2(x)`然后cast成整数，如果x不是2的幂次还好，如果是的话，比如$x = 2^k$，`static_cast<int>(log2(x))`到底会等于k呢还是等于$k-1$呢？需知道计算机里存储浮点数是无法百分百精确的。但是事实上，这里无论等于哪个，对计算答案而言都是没有影响的，所以等于谁其实无所谓。
3. `log2`这个函数的时间复杂度是多少？如果其时间消耗很长，那回答询问的时间将不是$O(1)$。幸运的是，`log2`这个函数时空复杂度确实是$O(1)$的，并且当参数是正整数的时候，确实存在$O(1)$的算法，而且有的编译器可以直接优化为这个$O(1)$算法。事实上，如果n是`int32`类型，`log2(n)`就是31减去n的二进制表示开头有多少个0。读者可以验证。

例题：https://blog.csdn.net/qq_46105170/article/details/119861243

```
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;

const int N = 2e5 + 10, M = 19;
int n, m;
int f[N][M];

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%d", &f[i][0]);

  for (int j = 1; 1 << j < n; j++)
    for (int i = 1; i + (1 << j) - 1 <= n; i++)
      f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);

  scanf("%d", &m);
  while (m--) {
    int l, r;
```

```
    scanf("%d%d", &l, &r);
    int k = log2(r - l + 1);
    printf("%d\n", max(f[l][k], f[r - (1 << k) + 1][k]));
  }
}
```

可以看出构造稀疏表时间复杂度$O(n\log n)$，稀疏表本身空间$O(n\log n)$，每次询问时间$O(1)$。