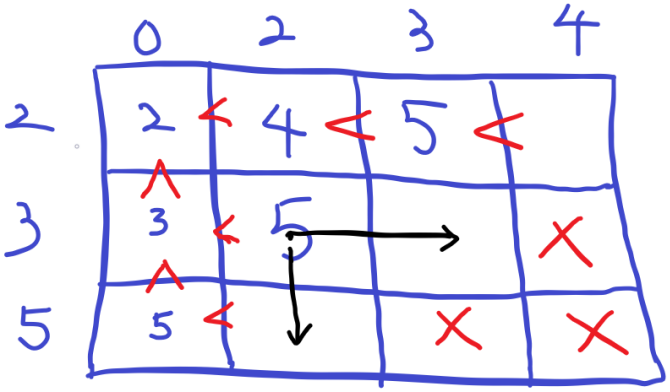
The Lightest Language

有一个简单的想法就是从上往下进行拓展，则每次应当是尽可能拓展最小的一个。因此这样很容易得出一个贪心做法，维护一个优先队列，初始只有根。每次将当前权值最小的点取出，钦定它为一个非叶子节点，然后将它所有的儿子加入优先队列。这样做的正确性在于假如我们选择了另外一个点做非叶节点而没有选择最小的点，那么将最终挂在新点下面的子树放回这个最小的点一定是更优的。

不过这样做的问题在于我们不知道什么时候应该停止。例如一种极端的情况就是 $k=n$，但第 k 个权值特别大，此时如果一旦有了 n 个叶子就停止，那么就会选择那个最大的权值。但如果我们先将 1 设为非叶子节点，那么 1 的权值会被多算几次，但是仍然小于 k 的权值。因此一个想法就是一直重复这个过程，如果存在至少 n 个叶子那可以用它们中最小的 n 个更新一次答案。那什么时候应该停止呢？由于每次更新会增加一个非叶子节点，而又不可能有一个点只有一个儿子，所以 n 个叶子至多对应 $n-1$ 个非叶子节点，那么只需要更新 n 次即可。这样做复杂度为 $O(nk\log nk)$，如果每次不把所有儿子都加入而是只加入最小的儿子和最小的兄弟，那么复杂度 $O(nk\log n)$（因为弹出一个兄弟不算做增加一个非叶子节点）。

考虑进一步优化这个做法。如果我们用一个 $n\times k$ 的表格记录上述这个过程，其中 (i,j) 这个位置记录的是第 i 次更新时用第 j 小的字符插入优先队列的数。记 b_i 为第 i 个非叶节点（即第 i 次更新时弹出的点）的权值。然后可以观察到，若 $ij\geq 2n$，则表格中比 ij 小的数至少有 $2n$ 个，其中至多 n 个被优先队列弹出，那么在任意可以更新答案的时刻都至少存在 n 个比它小的数作为叶子，所以它一定不会有用。这意味着有用的点只可能有至多 $O(n\ln n)$ 个，从而复杂度可以做到和 k 无关，即 $O(n\ln n\log n)$。如下图中格子中的数具有这样的单调关系，更新的时候只可能沿着黑色的线更新，最右下角的几个格子是无用的。



接着我们发现继续优化看起来是比较困难的，因为我们不管怎么优化这个更新的过程，在求答案的时候都需要将 一些数插入一个 set，然后保留其中最小的 n 个，总共要插入 $n\ln n$ 次，所以复杂度的下界就是 $O(n\ln n\log n)$。不过注意到上面这个做法中，第 i 次更新后的答案 s_i 是凸的，这是因为每次被插入的元素值越来越大，但被弹出的元素值越来越小。这样如果我们想要求这个函数的最值，就可以二分来实现。具体来说，我们尝试二分找到第一个 $s_i<s_{i+1}$ 的位置，则将前 i 小的位置弹出之后剩余的部分取前 n 小即可。而一个静态矩形的前 n 小是容易 $O(n\log n)$ 求出的，所以我们得到了一个 $O(n\log^2 n)$ 的做法。

但这个做法还不如之前的做法，其原因是每次二分都必须从头开始一个一个插入，但实际上这是很浪费的。考虑一种分块的思想，按照权值排序后，每 B 个点分一块（所有点的权值 b_i 可以在最一开始的时候 $O(n\log n)$ 直接用堆预处理，并从小到大排序。由于具有单调性且只有 $O(n\ln n)$ 个点，所以这部分复杂度为 $O(n\log n+n\ln n))$ 。然后先二分出答案在哪一块里，最后在这一块里二分找到答案的真实位

置。这样做的复杂度是多少呢？容易发现是 $n\log n\log\frac{n\ln n}{B}+B\log n$，取 $B=n$ 得到最终复杂度为 $O(n\log n\log\log n)$。代码：

```
#include <bits/stdc++.h>
#define INF 1000000000
#define LINF 1000000000000000000
#define MOD 1000000007
#define mod 998244353
#define F first
#define S second
#define ll long long
#define N 10010
using namespace std;
ll n,k,a[N],val[N][30],b[N];
bool vis[N][30];
ll calc(ll x)
{
    if((x+1)*k-x<n)
    {
        return LINF;
    }
    ll i,j;
    for(i=0;i<=x;i++)
    {
        for(j=0;j<k&&j*i<=n*2;j++)
        {
            vis[i][j]=false;
        }
    }
    priority_queue<pair<ll,pair<ll,ll> > > pq;
    vis[0][0]=true;
    pq.push(make_pair(-val[0][0],make_pair(0,0)));
    ll tot=n+x,ret=0;
    while(tot)
    {
        ll w=-pq.top().F,cx=pq.top().S.F,cy=pq.top().S.S;
        pq.pop();
        tot--;
        if(tot<n)
        {
            ret+=w;
        }
        if(cx+1<=x&&(cx+1)*cy<=n*2&&(!vis[cx+1][cy]))
        {
            vis[cx+1][cy]=true;
            pq.push(make_pair(-val[cx+1][cy],make_pair(cx+1,cy)));
        }
        if(cy+1<k&&cx*(cy+1)<=n*2&&(!vis[cx][cy+1]))
        {
            vis[cx][cy+1]=true;
            pq.push(make_pair(-val[cx][cy+1],make_pair(cx,cy+1)));
        }
    }
    return ret;
}
```

```
int main(){
    ll i,j;
    cin>>n>>k;
    for(i=0;i<k;i++)
    {
        cin>>a[i];
    }
    sort(a,a+k);
    k=min(k,n);
    multiset<pair<ll,pair<ll,ll> > > pq;
    for(i=0;i<k;i++)
    {
        vis[0][i]=true;
        val[0][i]=a[i];
        pq.insert(make_pair(a[i],make_pair(0,i)));
    }
    b[0]=0;
    for(i=1;i<=n+1;i++)
    {
        b[i]=pq.begin()->F;
        ll x=pq.begin()->S.F,y=pq.begin()->S.S;
        pq.erase(pq.begin());
        for(j=0;j<k&&j*i<=n*2;j++)
        {
            val[i][j]=b[i]+a[j];
        }
        if((x+1)*y<=n*2&&(!vis[x+1][y]))
        {
            vis[x+1][y]=true;
            pq.insert(make_pair(val[x+1][y],make_pair(x+1,y)));
        }
        if(y+1<k&&x*(y+1)<=n*2&&(!vis[x][y+1]))
        {
            vis[x][y+1]=true;
            pq.insert(make_pair(val[x][y+1],make_pair(x,y+1)));
        }
        while(pq.size()>n)
        {
            pq.erase(--pq.end());
        }
    }
    ll l=0,r=n;
    while(l<r)
    {
        ll mid=(l+r)>>1;
        if(calc(mid)<calc(mid+1))
        {
            r=mid;
        }
        else
        {
            l=mid+1;
        }
    }
    cout<<calc(l)<<'\n';
    return 0;
```

```
}
```

回顾一下这个问题，其本质是对于一个具有单调性的矩阵求第 n 小值的过程。可以这么分块进行二分的关键在于原本动态的问题在静态情形中具有更好的复杂度，但是又没有办法静态处理。此时先在外层二分确定答案具体在哪一层内，然后在内部动态更新答案即可。例如在树链上二分的一个经典做法就是树剖之后先对重链二分，确定答案在哪条链上，然后在这条重链上二分确定具体的答案。这么做本质上就是因为直接在树上二分复杂度较高，但是二分重链复杂度较低，所以平衡一下得到这样的做法。

例如下面这个问题：在平面上依次加入 n 个点，求什么时刻第一次有至少 k 对点有偏序关系。动态做需要树套树，是 $O(n\log^2 n)$ 的，但固定一个时刻可以离线二维数点，是 $O(n\log n)$ 的。如果直接二分那么复杂度仍然是 $O(n\log^2 n)$。考虑分块，设一个 B，每 B 个点分一块，先确定答案在哪一块里，这里做静态的二维数点，复杂度 $O(n\log n\log\frac{n}{B})$。然后在块内用树套树动态维护（或者用 CDQ 分治也可以），可以做到 $O(B\log^2 n)$ 的复杂度。平衡一下，取 $B=\frac{n}{\log n}$，得到复杂度为 $O(n\log n\log\log n)$。