A1.

(1) 每次删除后，只有相邻元素受到影响，因此可以用链表维护序列，用最大堆维护当前能够删除的元素，每次删除元素后检查相邻元素的状态是否发生改变，如果改变了，就在堆中进行相应的操作。初始时的扫描及建堆用时 $\Theta(n)$，至多进行 $n-1$ 次删除操作，每次删除执行一次 EXTRACT-MAX，还可能需要执行一次 INSERT，易验证最坏时间复杂度为 $\Theta(n \lg n)$。代码示例如下：

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <queue>
using namespace std;
int main()
{
    int n;
    cin >> n;
    vector<int> val(n + 2), prev(n + 2), next(n + 2);
    vector<bool> del(n + 2);
    for (int i = 0; i <= n + 1; i++)
    {
        if (i >= 1 && i <= n)
            cin >> val[i];
        prev[i] = i - 1;
        next[i] = i + 1;
    }
    auto chk = [&](int p) {return del[p] = (p >= 1 && p <= n && ((prev[p] != 0 && val[prev[p]] == val[p] - 1) || (next[p] != n + 1 && val[next[p]] == val[p] - 1))); };
    vector<pair<int, int>> init;
    for (int i = 1; i <= n; i++)
        if (chk(i))
            init.emplace_back(val[i], i);
    priority_queue pq(init.begin(), init.end());
    int res = 0;
    while (!pq.empty())
    {
        auto p = pq.top().second;
        pq.pop();
        int l = prev[p], r = next[p];
        next[l] = r;
        prev[r] = l;
        if (!del[l] && chk(l))
            pq.emplace(val[l], l);
        else if (!del[r] && chk(r))
```

```
            pq.emplace(val[r], r);
        res++;
    }
    cout << res;
    return 0;
}
```

(*2)

(*2.1) 假设第一步删除了元素 x_k，如果删除的同时产生了新的可删除元素，那么一定有：$|x_{k-1} - x_{k+1}| = 1$。不妨令 $x_{k-1} = x_{k+1} - 1$，则新可删除元素为 x_{k+1}。

因为 x_k 可删除，故 x_{k-1}, x_{k+1} 中至少有一个等于 $x_k - 1$。若 $x_{k-1} = x_k - 1$，则新可删除元素为 x_k；若 $x_{k+1} = x_k - 1$，则新可删除元素为 $x_k - 1$。无论哪种情况，新可删除元素都比原元素小，这说明无论如何操作，序列中最大的可删除元素一定不会变大。

如果某个操作序列能够使得 x_j 与 x_i 相邻，则最大可删除元素从 x_i 变大到 x_j，这与前述论证矛盾。

(*2.2) 记 d 的长度为 l。令：

$$\varphi(d) = \begin{cases} d & ,若d \in D' \\ (x_i, d_1, d_2, \ldots, d_l) & ,若d中不包含x_i \\ (x_i, d_1, d_2, \ldots, d_{k-1}, d_{k+1}, \ldots, d_l) & ,若d_k = x_i, k > 1 \end{cases}$$

由 (2.1) 知，d 中不存在某个元素依靠 x_i 删除，因此将 x_i 的删除提前并不会干扰 d 中任何其他元素的删除，所得删除序列仍合法。

(*3) 借鉴计数排序的思路，用一个长度为 m 的“数组套链表”的结构取代 (1) 中的最大堆记录当前可删除元素。由 (2.1) 中的论证，删除过程中最大可删除元素不增，因此可以从大到小遍历上述数组并在删除过程中维护，最坏时间复杂度为 $O(m+n)$。代码示例如下：

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int m, n;
    cin >> m >> n;
    vector<int> val(n + 2), prev(n + 2), next(n + 2);
    vector<bool> del(n + 2);
```

```
    for (int i = 0; i <= n + 1; i++)
    {
        if (i >= 1 && i <= n)
            cin >> val[i];
        prev[i] = i - 1;
        next[i] = i + 1;
    }
    auto chk = [&](int p) {return del[p] = (p >= 1 && p <= n && ((prev[p] !=
         0 && val[prev[p]] == val[p] - 1) || (next[p] != n + 1 && val[next[
        p]] == val[p] - 1))); };
    vector pq(m + 1, vector<int>());
    for (int i = 1; i <= n; i++)
        if (chk(i))
            pq[val[i]].push_back(i);
    int res = 0;
    for (int e = m; e >= 1; e--)
        while (!pq[e].empty())
        {
            int p = pq[e].back();
            pq[e].pop_back();
            int l = prev[p], r = next[p];
            next[l] = r;
            prev[r] = l;
            if (!del[l] && chk(l))
                pq[val[l]].push_back(l);
            if (!del[r] && chk(r))
                pq[val[r]].push_back(r);
            res++;
        }
    cout << res;
    return 0;
}
```

A2.

(1) 维护一个长度为 k 的滑动窗口，用 $O(n)$ 时间求出 $\mathbf{x}$ 中长度为 k 的最大子串和 $S = \max\limits_{m\in[1,n-k+1]} \sum_{i=m}^{m+k-1} x_i$。能缴获的电脑数量的最大值即为 $S + (0 + 1 + \cdots + k - 1) = S + \frac{k(k-1)}{2}$

(2) 将能够缴获的电脑数量 N 分为两部分：$N = I + P$，I 表示初始时即在宿舍中，之后被缴获的电脑数量，P 表示晚上被带入宿舍，之后被缴获的电脑数量。显然，$I \le S, P \le \frac{k(k-1)}{2}$，因此 N 的一个上界是 $S + \frac{k(k-1)}{2}$

现构造一种具体的方案，使得恰能够收缴这么多台电脑，从而证明该上界确实是最优解：记 S 的最大值在 m' 处取到，第一天，舍管进入第 m' 间

宿舍，之后每天都进入右边相邻的那间宿舍，易验证此时 $N = S + \frac{k(k-1)}{2}$

(*3) 记 $S = \sum_{i=1}^{n} x_i$，最大值为 $S + (k-1)n - \frac{n(n-1)}{2}$，这显然可以在 $O(n)$ 时间内求出。

(*4) 同样地，我们考虑 $N = I + P$。显然，$I \leq S$。对于 P，从反面进行考虑：在 k 天中，n 间宿舍一共新藏了 $(k-1)n$ 台电脑，记最后一天收缴结束时，第 i 间宿舍内未被收缴的新藏的电脑数量为 e_i，则 $P = (k-1)n - \sum_{i=1}^{n} e_i$。$e_i$ 的值等于“上次对该间宿舍的收缴距离现在的天数”，因为每天只能收缴一间宿舍，因此 e_i 是互不相同的自然数，所以 $\sum_{i=1}^{n} e_i \geq 0+1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$，即 $P \leq (k-1)n - \frac{n(n-1)}{2}$，所以 N 的一个上界是 $S + (k-1)n - \frac{n(n-1)}{2}$

具体方案的构造：前 $k-n+1$ 天都停留在第一间宿舍，第 $k-n+2$ 天进入第二间宿舍，第 $k-n+3$ 天进入第三间宿舍，以此类推，直到第 k 天进入最后一间宿舍结束。易验证此时 $N = S + (k-n) + \sum_{i=2}^{n}(k-n+i-1) = S + (k-1)n - \frac{n(n-1)}{2}$

A3.

(1) 对 $\mathbf{x}$ 从小到大排序即得到所求的 $\mathbf{y}$

(2) 对 $\mathbf{x}$ 从大到小排序即得到所求的 $\mathbf{y}$

(*3)

(*3.1) 任取一个能够取到最大值的 $\mathbf{z}^\star$，如果其中有相邻元素组成了 $\lesssim$ 意义下的逆序对，则将它们调换位置，直到得到一个 $\lesssim$ 意义下的顺序排列 z'。由 $\lesssim$ 的性质，每次这样的调换都不会使结果变差，因此 $f(\mathbf{z}') \geq f(\mathbf{z}^\star)$。由全预序的性质，任意两个顺序排列都可以通过若干次相邻等价元素间的对换进行相互转化，而这样的对换不会改变结果的大小，因此任意顺序排列 $\mathbf{z}$，$f(\mathbf{z}) = f(\mathbf{z}') \geq f(\mathbf{z}^\star)$。因为 $z^\star$ 处能够取得最大值，因此上式中“$\geq$”一定取等，所以 $f(\mathbf{z}) = f(\mathbf{z}^\star)$。

对于 (1) 中的问题，取 $\lesssim$ 为一般意义上的“小于等于”。

$$f(\mathbf{y}') - f(\mathbf{y}) = iy_{i+1} + (i+1)y_i - (iy_i + (i+1)y_{i+1}) = y_i - y_{i+1}$$

因此 $y_{i+1} \leq y_i$ 是 $f(\mathbf{y}') \geq f(\mathbf{y})$ 的充分条件，因此当 $\mathbf{y}$ 是 $\mathbf{x}$ 的顺序排列时能够取到最大值。

对于 (2) 中的问题，取 $\lesssim$ 为一般意义上的“大于等于”

$$f(\mathbf{y}') - f(\mathbf{y}) = \max\{0, y_{i+1} - i\} + \max\{0, y_i - i - 1\} - \max\{0, y_i - i\} - \max\{0, y_{i+1} - i - 1\}$$

当 $y_{i+1} \lesssim y_i$，即 $y_{i+1} \geq y_i$ 时，分情况讨论：

- 若 $y_i \geq i+1$，此时 $y_{i+1}-i$、y_i-i-1、y_i-i、$y_{i+1}-i-1$ 均非负，$f(\mathbf{y}')-g(\mathbf{y})=0$
- 若 $y_i \leq i$，此时 y_i-i、y_i-i-1 均非正，$f(\mathbf{y}')-f(\mathbf{y})=\max\{0,y_{i+1}-i\}-\max\{0,y_{i+1}-i-1\}\geq 0$

因此 $y_{i+1}\geq y_i$ 是 $f(\mathbf{y}')\geq f(\mathbf{y})$ 的充分条件，因此当 $\mathbf{y}$ 是 $\mathbf{x}$ 的逆序排列时能够取到最大值。

(*3.2) 记 $\mathbf{y}'$ 是将 $\mathbf{y}$ 中 y_i, y_{i+1} 交换得到的排列，如果能够使 $t_{i+1}(\mathbf{y}')\leq t_{i+1}(\mathbf{y})$，那么由归纳法一定有 $t_n(\mathbf{y}')\leq t_n(\mathbf{y})$。

记 $S=\sum_{j=1}^{i-1}a_j$，$\tau=t_{i-1}(\mathbf{y})$，则：

$$
\begin{aligned}
t_i(\mathbf{y}) &= \max\{\tau, S+a_i\}+b_i \\
t_{i+1}(\mathbf{y}) &= \max\{\max\{\tau, S+a_i\}+b_i, S+a_i+a_{i+1}\}+b_{i+1} \\
t_i(\mathbf{y}') &= \max\{\tau, S+a_{i+1}\}+b_{i+1} \\
t_{i+1}(\mathbf{y}') &= \max\{\max\{\tau, S+a_{i+1}\}+b_{i+1}, S+a_{i+1}+a_i\}+b_i
\end{aligned}
$$

可以看出 $t_{i+1}(\mathbf{y}), t_{i+1}(\mathbf{y}')$ 分别在三个数中取最大值，适当变形得到：

$$
\begin{aligned}
t_{i+1}(\mathbf{y}) &= \max\{\tau+b_i+b_{i+1}, S+a_i+b_i+b_{i+1}, S+a_i+a_{i+1}+b_{i+1}\} \\
t_{i+1}(\mathbf{y}') &= \max\{\tau+b_i+b_{i+1}, S+a_{i+1}+b_i+b_{i+1}, S+a_i+a_{i+1}+b_i\}
\end{aligned}
$$

两者的第一项是相同的，因此 $t_{i+1}(\mathbf{y}')\leq t_{i+1}(\mathbf{y})$ 的一个充分条件是：

$$\max\{S+a_{i+1}+b_i+b_{i+1}, S+a_i+a_{i+1}+b_i\}\leq \max\{S+a_i+b_i+b_{i+1}, S+a_i+a_{i+1}+b_{i+1}\}$$

提取出公共部分并消去 S，移项整理后，不等式等价于：

$$a_{i+1}+b_i-\max\{a_{i+1},b_i\}\leq a_i+b_{i+1}-\max\{a_i,b_{i+1}\}$$

从两个数中减去较大的数，留下的即为较小数，因此不等式等价于：

$$\min\{a_{i+1},b_i\}\leq \min\{a_i,b_{i+1}\}$$

经检验，这一关系不具有传递性，不符合全预序的要求，因此无法用于排序。某种程度上可以理解为：这一关系对“等价”的界定过于宽松了，元素之间的“差异”会在“等价链”中逐渐累积，从而有可能使得链的两端不

再“等价”，进而破坏传递性。考虑将原关系中“等价”的情况进行细化，构造全预序 $\lesssim$（构造方法不止一种），定义如下：

$$\langle s,t\rangle \lesssim \langle u,v\rangle \iff \min\{s,v\} < \min\{u,t\} \vee (\min\{s,v\} = \min\{u,t\} \wedge s \le u)$$

$\lesssim$ 的完全性显然，传递性可通过分类讨论证明或直接编写程序遍历所有情况进行检验。

由以上论证可知：

$$y_{i+1} \lesssim y_i \Rightarrow \min\{a_{i+1},b_i\} \le \min\{a_i,b_{i+1}\} \Rightarrow t_{i+1}(\mathbf{y}') \le t_{i+1}(\mathbf{y}) \Rightarrow t_n(\mathbf{y}') \le t_n(\mathbf{y})$$

仿照 (3.1) 中的论证，可知当 $\mathbf{y}$ 是 $\lesssim$ 意义下的顺序排列时，$t_n(\mathbf{y})$ 能够取到最小值。排序用时 $O(n \lg n)$

验证传递性的代码：

```cpp
#include <iostream>
#include <algorithm>
#include <utility>
using namespace std;
bool lesssim(int s, int t, int u, int v)
{
    return make_pair(min(s, v), s) <= make_pair(min(u, t), u);
}
int main() {
    for (int s = 1; s <= 6; s++) for (int t = 1; t <= 6; t++)
    for (int u = 1; u <= 6; u++) for (int v = 1; v <= 6; v++)
    for (int x = 1; x <= 6; x++) for (int y = 1; y <= 6; y++)
    if (lesssim(s, t, u, v) && lesssim(u, v, x, y) && !lesssim(s, t, x, y))
    {
        cout << "No transitivity";
        return 0;
    }
    return 0;
}
```

A4.

(1)

(1.1) 考虑“(⋆)”，如果先匹配星号，则第一遍扫描后字符串变为“(”，在第二遍扫描中，算法会输出`false` 的错误结果。

(1.2) 考虑“(⋆()”，如果先匹配左边的左括号，则第一遍扫描后字符串变为“⋆(”，在第二遍扫描中，算法会输出`false` 的错误结果。

(1.3) 考虑 “(⋆(⋆”，如果先匹配左边的左括号，则在第二遍扫描中，字符串会变为 “⋆(”，随后输出`false` 的错误结果。

(1.4) 用两个栈分别记录已扫描部分中剩下的左括号和星号的位置，这样匹配删除操作可以用栈的弹出进行模拟，从而达到 $O(1)$ 的单次操作时间，总运行时间为 $O(n)$

(*2) 严格的论证平凡而繁琐，因此这里仅给出主要思路。

- 若算法输出`true`，借助算法运行过程可以得到一个 “交错匹配”，可证明任意一个 “交错匹配” 都是合法的 “嵌套匹配”
- 如果算法在第一遍扫描中输出`false`，可证明即使将左侧所有星号都替换为左括号，也无法让当前的右括号被平衡
- 如果算法在第二遍扫描中输出`false`，可证明即使将右侧所有星号都替换为右括号，也无法让当前的左括号被平衡