最大连续子数列和一道很经典的算法问题，给定一个数列，其中可能有正数也可能有负数，我们的任务是找出其中连续的一个子数列（不允许空序列），使它们的和尽可能大。我们一起用多种方式，逐步优化解决这个问题。

为了更清晰的理解问题，首先我们先看一组数据：

```
8

-2 6 -1 5 4 -7 2 3
```

第一行的8是说序列的长度是 8 ，然后第二行有 8 个数字，即待计算的序列。

对于这个序列，我们的答案应该是 14 ，所选的数列是从第 2 个数到第 5 个数，这 4 个数的和是所有子数列中最大的。

最暴力的做法，复杂度$O(N^3)$

暴力求解也是容易理解的做法，简单来说，我们只要用两层循环枚举起点和终点，这样就尝试了所有的子序列，然后计算每个子序列的和，然后找到其中最大的即可， C 语言代码如下：

In [1]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void maxsum01(){
        //N是数组长度，num是待计算的数组，放在全局区是因为可以开很大的数组
        //int N, num[1024];   //每一个函数都有一个num数组，函数多了，内存就不够用了。
        int N;
        int *num=(int *)malloc(sizeof(int) * 1024);  //改成动态数组，当函数执行结束，释放内存。jupyter环境下么办法
        freopen("linemaxsum.in","r",stdin);
        //输入数据
        scanf("%d", &N);
        for(int i = 1; i <= N; i++){
            scanf("%d", &num[i]);
        }


        int ans = num[1]; //ans保存最大子序列和，初始化为num[1]能保证最终结果正确
        //i和j分别是枚举的子序列的起点和终点，k所在循环计算每个子序列的和
        for(int i = 1; i <= N; i++) {
            for(int j = i; j <= N; j++) {
                int s = 0;
                for(int k = i; k <= j; k++) {
                    s += num[k];
                }
                if(s > ans) ans = s;
            }
        }
        printf("%d\n", ans);
        fclose(stdin);
        free(num);
        return ;
    }
}
#endif
```

Out[1]:

In [2]:
```c
maxsum01();
```

```
14
```

Out[2]: (void) nullptr

这个算法的时间复杂度是$O(N^3)$，复杂度的计算方法可参考《算法导论》第一章，如果我们的计算机可以每秒计算一亿次的话，这个算法在一秒内只能计算出 500 左右长度序列的答案。

一个简单的优化

如果你读懂了刚才的程序，我们可以来看一个简单的优化。 如果我们有这样一个数组 sum ， sum[i] 表示第1个到第i个数的和。那么我们如何快速计算第i个到第j个这个序列的和？对，只要用 sum[j] - sum[i-1] 就可以了！这样的话，我们就可以省掉最内层的循环，让我们的程序效率更高！C语言代码如下：

In [3]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void maxsum02(){
        //N是数组长度，num是待计算的数组，sum是数组前缀和，放在全局区是因为可以开很大的数组
        int N;
        int *num=(int *)malloc(sizeof(int) * 16384);
        int *sum=(int *)malloc(sizeof(int) * 16384);
        freopen("linemaxsum.in","r",stdin);
        //输入数据
        scanf("%d", &N);
        for(int i = 1; i <= N; i++){
            scanf("%d", &num[i]);
        }


        //计算数组前缀和
        sum[0] = 0;
        for(int i = 1; i <= N; i++) {
            sum[i] = num[i] + sum[i - 1];
        }

        int ans = num[1]; //ans保存最大子序列和，初始化为num[1]能保证最终结果正确
        //i和j分别是枚举的子序列的起点和终点
        for(int i = 1; i <= N; i++) {
            for(int j = i; j <= N; j++) {
                int s = sum[j] - sum[i - 1];
                if(s > ans) ans = s;
            }
        }
        printf("%d\n", ans);
        fclose(stdin);
        free(num);
        free(sum);
        return ;
    }
}
#endif
```

Out[3]:

In [4]:
```
maxsum02();
```

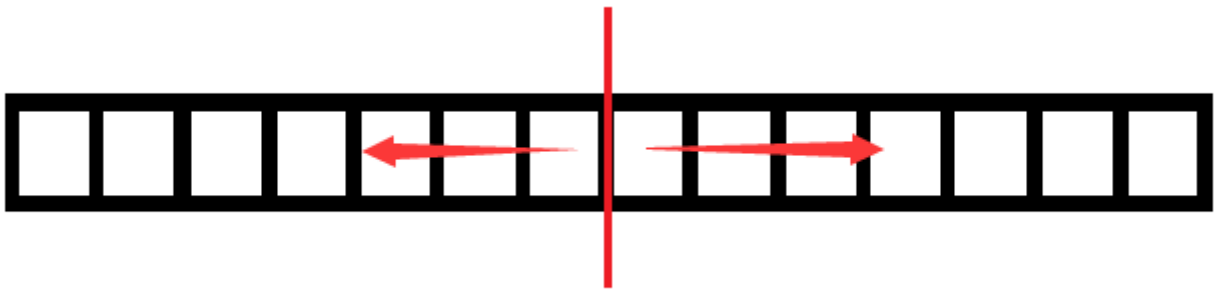
```
14
```

Out[4]: (void) nullptr

这个算法的时间复杂度是$O(N^2)$。如果我们的计算机可以每秒计算一亿次的话，这个算法在一秒内能计算出 10000 左右长度序列的答案，比之前的程序已经有了很大的提升！此外，我们在这个程序中创建了一个 sum 数组，事实上，这也是不必要的，我们也可以把数组前缀和直接计算在 num 数组中，这样可以节约一些内存。

换个思路，继续优化 你应该听说过分治法，正是：分而治之。我们有一个很复杂的大问题，很难直接解决它，但是我们发现可以把问题划分成子问题，如果子问题规模还是太大，并且它还可以继续划分，那就继续划分下去。直到这些子问题的规模已经很容易解决了，那么就把所有的子问题都解决，最后把所有的子问题合并，我们就得到复杂大问题的答案了。可能说起来简单，但是仍不知道怎么做，接下来分析这个问题： 首先，我们可以把整个序列平均分成左右两部分，答案则会在以下三种情况中：

1. 所求序列完全包含在左半部分的序列中。
2. 所求序列完全包含在右半部分的序列中。
3. 所求序列刚好横跨分割点，即左右序列各占一部分。 前两种情况和大问题一样，只是规模小了些，如果三个子问题都能解决，那么答案就是三个结果的最大值。我们主要研究一下第三种情况如何解决：



我们只要计算出：以分割点为起点向左的最大连续序列和、以分割点为起点向右的最大连续序列和，这两个结果的和就是第三种情况的答案。因为已知起点，所以这两个结果都能在$O(N)$的时间复杂度能算出来。

递归不断减小问题的规模，直到序列长度为 1 的时候，那答案就是序列中那个数字。 综上所述， C 语言代码如下，递归实现：

In [5]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int divN,divnum[16777216];

    int maxsumdiv(int left, int right)
    {
        //序列长度为1时
        if(left == right)
            return divnum[left];

        //划分为两个规模更小的问题
        int mid = (left + right) >> 1;   //除2
        int lans = maxsumdiv(left, mid);
        int rans = maxsumdiv(mid + 1, right);

        //横跨分割点的情况
        int sum = 0, lmax = divnum[mid], rmax = divnum[mid + 1];
        for(int i = mid; i >= left; i--) {   //算一个起点在左边，终点为mid的一个最大的区间和
            sum += divnum[i];
            if(sum > lmax) lmax = sum;
        }
        sum = 0;
        for(int i = mid + 1; i <= right; i++) {  //算一个起点为mid+1，终点在右边的一个最大区间和
            sum += divnum[i];
            if(sum > rmax) rmax = sum;
        }

        //答案是三种情况的最大值
        int ans = lmax + rmax;
        if(lans > ans) ans = lans;
        if(rans > ans) ans = rans;

        return ans;
    }
    void maxsum03(){
        freopen("linemaxsum.in","r",stdin);
        //输入数据
        scanf("%d", &divN);
        for(int i = 1; i <= divN; i++)
            scanf("%d", &divnum[i]);

        printf("%d\n", maxsumdiv(1, divN));

        return ;
    }
}
#endif
```

Out[5]:

In [6]:
```
maxsum03();
```

```
14
```

Out[6]: (void) nullptr

不难看出，这个算法的时间复杂度是$O(N\log N)$的（想想归并排序）。它可以在一秒内处理百万级别的数据，甚至千万级别也不会显得很慢！这正是算法的优美之处。对递归不太熟悉的话可能会对这个算法有所疑惑，那可就要仔细琢磨一下了。

动态规划的魅力，$O(N)$解决！

很多动态规划算法非常像数学中的递推。我们如果能找到一个合适的递推公式，就能很容易的解决问题。

我们用 dp[n] 表示以第 n 个数结尾的最大连续子序列的和，于是存在以下递推公式：

```
dp[n] = max(1, dp[n-1]) + num[n]
```

仔细思考后不难发现这个递推公式是正确的，则整个问题的答案是$max(dp[m])\mid m\in[1,N]$。 C 语言代码如下：

In [1]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void maxsum04(){
        freopen("linemaxsum.in","r",stdin);
        //N是数组长度，num是待计算的数组，放在全局区是因为可以开很大的数组
        int N;
        int *num=(int *)malloc(sizeof(int) * 134217728);
        //输入数据
        scanf("%d", &N);

        for(int i = 1; i <= N; i++)
            scanf("%d", &num[i]);

        num[0] = 0;
        int ans = num[1];
        for(int i = 1; i <= N; i++) {
            if(num[i - 1] > 0) num[i] += num[i - 1];  //这个if就是求后缀和
            else num[i] += 0;
            if(num[i] > ans) ans = num[i];
        }

        printf("%d\n", ans);
        fclose(stdin);
        free(num);
        return ;
    }
}
#endif
```

Out[1]:

In [2]:
```
maxsum04();
```

```
14
```

Out[2]: (void) nullptr

这里我们没有创建 dp 数组，根据递归公式的依赖关系，单独一个 num 数组就足以解决问题，创建一个一亿长度的数组要占用几百 MB 的内存！这个算法的时间复杂度是$O(N)$的，所以它计算一亿长度的序列也不在话下！不过你如果真的用一个这么大规模的数据来测试这个程序会很慢，因为大量的时间都耗费在程序读取数据上了！

另辟蹊径，又一个O(N)的算法

考虑之前$O(N^2)$的算法，即一个简单的优化一节，还有没有办法优化这个算法呢？答案是肯定的！

已知一个 sum 数组， sum[i] 表示第 1 个数到第 i 个数的和，于是 sum[j]-sum[i-1] 表示第 i 个数到第 j 个数的和。

那么，以第 n 个数为结尾的最大子序列和有什么特点？假设这个子序列的起点是m，于是结果为 sum[n]-sum[m-1] 。并且， sum[m] 必然是 sum[1],sum[2]...sum[n-1] 中的最小值！这样，我们如果在维护计算 sum 数组的时候，同时维护之前的最小值， 那么答案也就出来了！为了节省内存，我们还是只用一个 num 数组。 C 语言代码如下：

0 -2 6 -1 5 4 -7 2 3

0 -2 4 3 8 12 5 7 10

最小值没变过， -2

In [1]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void maxsum05(){
        freopen("linemaxsum.in","r",stdin);
        //N是数组长度，num是待计算的数组
        int N;
        int *num=(int *)malloc(sizeof(int) * 134217728);
        //输入数据
        scanf("%d", &N);
        for(int i = 1; i <= N; i++)
            scanf("%d", &num[i]);

        //计算数组前缀和，并在此过程中得到答案
        num[0] = 0;
        int ans = num[1], lmin = 0;
        for(int i = 1; i <= N; i++) {   //通过观察，num[i]，i一直在增大，且不会再次计算，所以。。。。。。。。下一个优化
            num[i] += num[i - 1];
            if(num[i] - lmin > ans)  //如果找到了一个最大的，则更新最大值
                ans = num[i] - lmin;
            if(num[i] < lmin)   //记录最小的和
                lmin = num[i];
            //printf("%d\n",lmin);
        }

        printf("%d\n", ans);
        fclose(stdin);
        free(num);
        return ;
    }
}
#endif
```

Out[1]:

In [2]:
```
maxsum05();
```

```
14
```

Out[2]: (void) nullptr

看起来我们已经把最大连续子序列和的问题解决得很完美了，时间复杂度和空间复杂度都是$O(N)$，不过，我们确实还可以继续!

大道至简，最大连续子序列和问题的完美解决

很显然，解决此问题的算法的时间复杂度不可能低于$O(N)$，因为我们至少要算出整个序列的和，不过如果空间复杂度也达到了$O(N)$，就有点说不过去了，让我们把 num 数组也去掉吧!

In [1]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void maxsum06(){
        freopen("linemaxsum.in","r",stdin);
        //N是数组长度，num是待计算的数组
        int N, n, s, ans, m = 0;
        scanf("%d%d", &N, &n); //读取数组长度和序列中的第一个数
        ans = s = n; //把ans初始化为序列中的的第一个数
        for(int i = 1; i < N; i++) {
            if(s < m) m = s;       //m用来存最小和
            scanf("%d", &n);
            s += n;
            if(s - m > ans)   //找到更好的答案
                ans = s - m;
        }

        printf("%d\n", ans);
        fclose(stdin);
        return ;
    }
}
#endif
```

Out[1]:

In [2]:
```
maxsum06();
```

```
14
```

Out[2]: (void) nullptr

In [ ]: