1. 找到两个和为固定值的数

2.3-7 ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

法一： 快速排序+二分查找

```cpp
#include<iostream>
#include<vector>
using namespace std;

bool searchTwoSum(const vector<int>& vec, int target) {
    int left = 0;
    int right = vec.size() - 1;
    int mid;

    while (left < right) {
        mid = (left + right) >> 1;
        if (vec[mid] == target) return true;
        if (vec[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return vec[left] == target;
}

// 左闭右闭
void quickSort(vector<int>& vec, int bg, int ed) {
    if (bg < ed) {
        int left = bg, right = ed, pivot = vec[bg];

```

```
        while (left < right) {
            while (left < right && vec[right] >= pivot) right--;
            vec[left] = vec[right]; // 将小于pivot的换到左边

            while (left < right && vec[left] <= pivot) left++;
            vec[right] = vec[left]; // 将大于pivot的换到右边
        }

        vec[left] = pivot;

        quickSort(vec, bg, left-1);
        quickSort(vec, left+1, ed);
    }
}

int main() {
    vector<int> vec = {1,2,5,6,3,2,1,10,9,18,22};
    int target1 = 31;
    int target2 = 0;

    quickSort(vec, 0, vec.size() - 1);

    //for (auto i : vec) cout<<i<<" ";
    for (int i :vec) {
        if (searchTwoSum(vec, target1 - i)) {
            cout<<"There exists "<< target1 <<endl;
            break;
        }
    }

     for (int i :vec) {
        if (searchTwoSum(vec, target2 - i)) {
            cout<<"There exists "<< target2 <<endl;
            break;
        }
    }
```

```
}
```

简述：首先将输入数组进行快速排序，复杂度为$O(nlogn)$，然后遍历数组中的数（可以优化为只遍历$\lceil n \rceil$次），并对每个数进行二分查找，遍历与查找复杂度为$O(nlogn)$，故总复杂度$O(nlogn)$。

法二：快速排序+双指针

```
// 左闭右闭
void quickSort(vector<int>& vec, int bg, int ed) {
    if (bg < ed) {
        int left = bg, right = ed, pivot = vec[bg];

        while (left < right) {
            while (left < right && vec[right] >= pivot) right--;
            vec[left] = vec[right]; // 将小于pivot的换到左边

            while (left < right && vec[left] <= pivot) left++;
            vec[right] = vec[left]; // 将大于pivot的换到右边
        }

        vec[left] = pivot;

        quickSort(vec, bg, left-1);
        quickSort(vec, left+1, ed);
    }
}

int main() {
```

```cpp
    vector<int> vec = {1,2,5,6,3,2,1,10,9,18,22};
    int target1 = 31;

    quickSort(vec, 0, vec.size() - 1);

    int i = 0, j = vec.size() - 1, flag = 0;
    while (i < j) {
        if (vec[i] + vec[j] < target1) i++;
        else if (vec[i] + vec[j] > target1) j--;
        else {
            flag = 1;
            cout<<"There exists "<< target1 <<endl;
        }
    }
    if (!flag) cout<<"There doesn't exists "<< target1 <<endl;
}
```

简述：首先将输入数组进行快速排序，复杂度为$O(nlogn)$，然后使用两个指针i, j来查找有序数组中是否有符合要求的数，复杂度为$O(n)$，故总复杂度$O(nlogn)$。

2. 最大子数组

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \,.\,.\, j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1 \,.\,.\, j+1]$ is either a maximum subarray of $A[1 \,.\,.\, j]$ or a subarray $A[i \,.\,.\, j+1]$, for some $1 \le i \le j+1$. Determine a maximum subarray of the form $A[i \,.\,.\, j+1]$ in constant time based on knowing a maximum subarray ending at index j.

算法如下：

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main() {
    vector<int> vec = {1,3,-5,-10,8,9,-1,2,-10,18,9,-6,2};
    int s = vec.size();

    int submax = 0;
    int sublow = 0;

    int max = INT_MIN;
    int low = 0, high = 0;

    for (int i = 0;i < s;i++) {
        submax += vec[i];
        if (max < submax) {
            max = submax;
            low = sublow;
            high = i;
        }
        else if (submax < 0){
            submax = 0;
            sublow = i + 1; // 重新开始
        }
    }
    cout<< low << "-" << high << ": " << max;
}
```

简述：使用max保存当前全局最大的情况，并不断更新submax，当submax<0时，没有继续维护的必要，直接清0。只需遍历一次数组，复杂度$O(n)$。

3. 堆

6.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

使用反证法证明，假设有$\lceil n/2^{h+1} \rceil + 1$个节点：

设$C(h)$表示高度为h的堆所含有的元素数量, 有$C(h) \in [2^h, 2^{h+1} - 1]$。

若存在$N(h) = \lceil n/2^{h+1} \rceil + 1$个高度为$h$的节点，有$N(h) \geq n/2^{h+1} + 1$，那么所有高度为$h$的堆的子堆元素总数

$$S(h) = C(h)N(h) \geq 2^h(n/2^{h+1} + 1) = n/2 + 2^h \quad (1)$$

因为容量为n的堆的高度为$\lfloor lgn \rfloor$，所以：$S(\lfloor lgn \rfloor) = n$。将$h = \lfloor lgn \rfloor$带入式$(1)$，可得

$$S(\lfloor lgn \rfloor) \geq n/2 + 2^{\lfloor lgn \rfloor} > n/2 + 2^{lgn-1} = n/2 + n/2 = n$$

与前提矛盾，故不可能有超过$\lceil n/2^{h+1} \rceil$个节点。

4. 分析复杂度

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$.

b. $T(n) = T(7n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

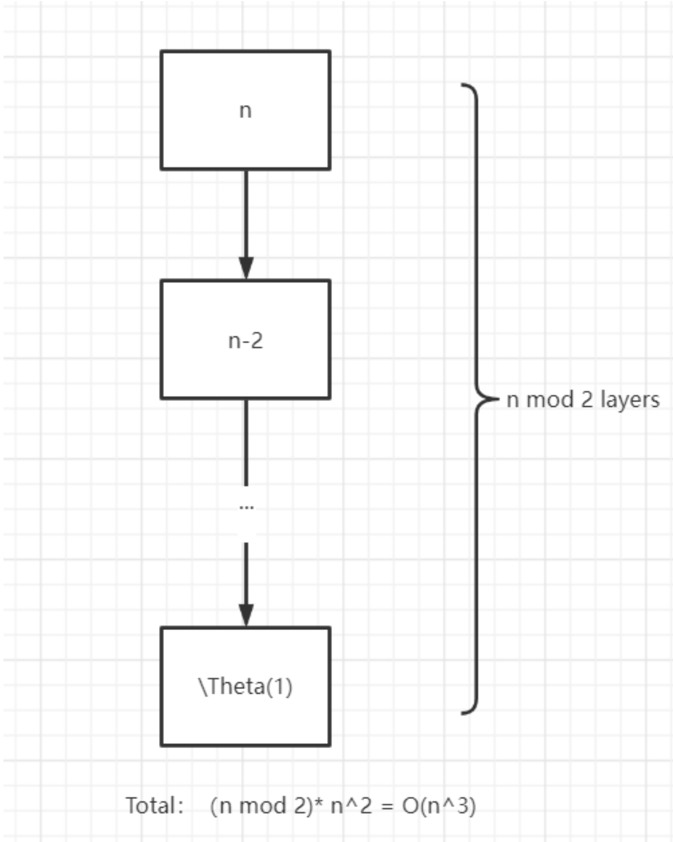
f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-2) + n^2$.

根据主方法公式，与$log_b a$比较得到：

$$\begin{aligned}
&1.\Theta(n^4) \\
&2.\Theta(n) \\
&3.\Theta(n^2 logn) \\
&4.\Theta(n^2) \\
&5.\Theta(log7) \\
&6.\Theta(\sqrt{n} logn) \\
&7.\Theta(n^3)
\end{aligned} \tag{1}$$

其中7不能通过主方法求解，可以画递归树得到：



5. 线性复杂度

8.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

因为比较排序的过程都能抽象为一棵决策树，树的高度即为复杂度。假设为线性复杂度，则树高$h = O(n)$。

当叶子节点为$n!/2$个时，有$2^h \geq n!/2$，故有$h \geq lgn! - 1 > (n/2)lgn - 1$，又$h = O(n)$，得到的不等式与前提矛盾。

当叶子节点为$(n-1)!$个时，同样的，有$h \geq lg(n-1)! > (n-1)/2 * lg(n-1)$，同理可推出与前提矛盾。

当叶子节点为$n!/2^n$个时，同样的，有$2^h \geq n!/2^n$即$h + n \geq lgn! > (n/2)lgn$，同理可推出与前提矛盾。

故不存在为线性复杂度的比较算法满足如上情况。

6. 计数排序

> ***8.3-4***
> Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

因为$range \in [0, n^3 - 1]$，无法直接使用计数排序，**可以做一次转化，令**$a = lgn$，对a构成的数组A进行计数排序，时间复杂度为$O(n)$。

7. 桶排序

> ***8.4-2***
> Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

在最坏情况下，输入并不随机分布，而是集中在一个桶中，因此插入排序的时间复杂度为$O(n^2)$，总的时间复杂度为$O(n^2)$。

可以通过改变排序算法来改善最坏情况复杂度，如：将插入排序修改为归并排序。

8. 快速排序

9.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

为了在最坏情况也保证$O(nlgn)$的复杂度，需要改变对pivot的选择。可以每次都选择序列中的中位数作为pivot，对中位数的选择只需要$O(n)$，因此不需额外复杂度。