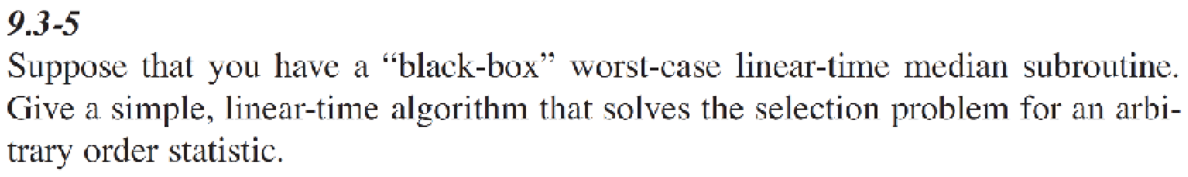
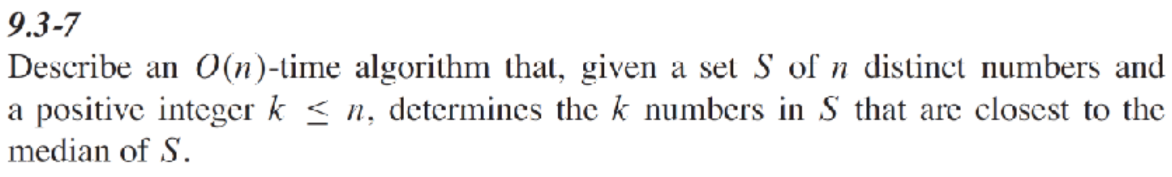
Order

9.3.5线性选择

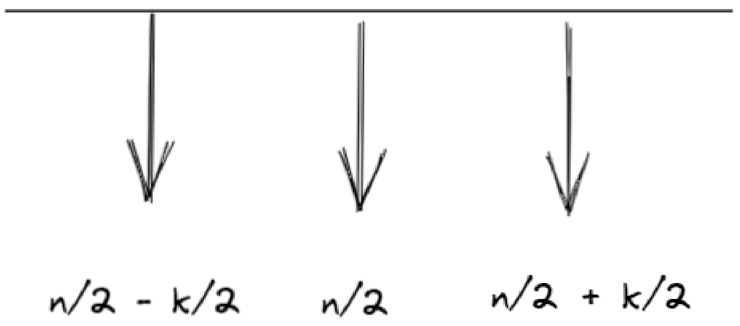

9.3-5
Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

参考章节中讲的算法，首先利用$O(n)$时间算法找出中位数，并用中位数做partition。如果期望的$i = 1/2\ len$，直接返回中位数。若$i < 1/2\ len$递归的在前半个数组中寻找第i个数。若$i > 1/2 len$，递归的在后半个数组中寻找第$i - 1/2\ len$个数。

9.3.7找范围数

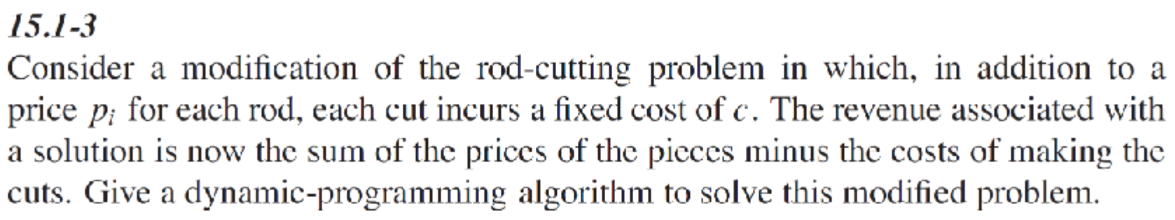

9.3-7
Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

首先使用$O(n)$复杂度算法找到集合中第$n/2 - k/2$大的元素并做partition，此时该元素左侧和右侧已经被划分好。再从该元素右侧（既所有大于该元素的集合中）找到第k大的元素并做partition。则可以得到第二次查找后，该元素左侧集合的元素就是我们所需的元素。



DP

15.1.3考虑cost

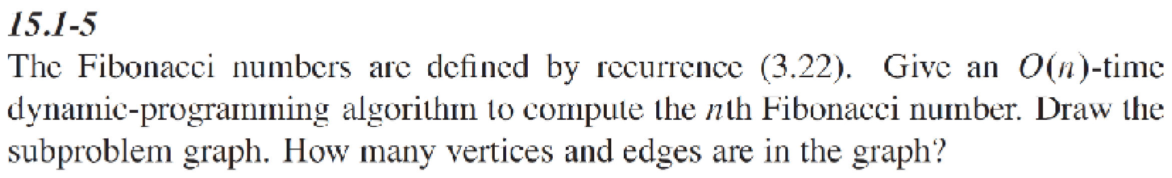


15.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

```
const int SIZE = 5;

int main() {
    vector<int> prices = {2,4,5,6,9};
    int cost = 1;

    vector<int> dp(SIZE + 1, INT_MIN);
    dp[0] = 0;

    for (int i = 0;i < SIZE;i++) {
        for (int j = i + 1;j <= SIZE;j++) {
            dp[j] = max(dp[j], prices[i] + dp[j - i - 1] - cost);
        }
    }
    cout << dp[SIZE] << endl;

    return 0;
}
```

15.1.5斐波那契

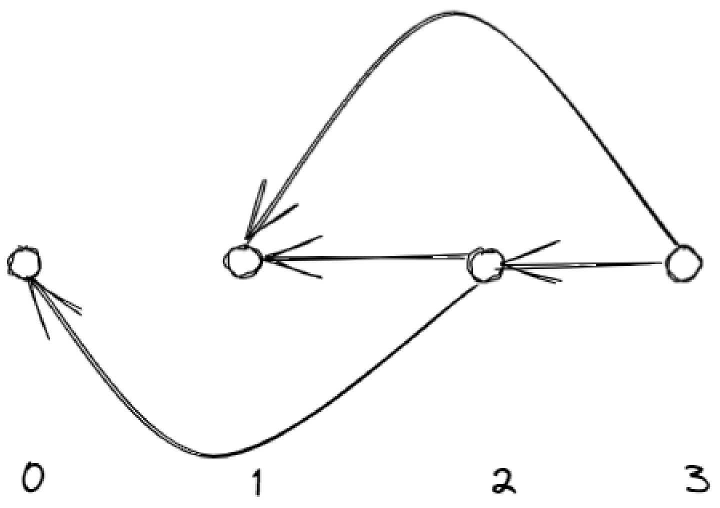


15.1-5

The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

```
int main() {
    vector<int> fib;
    fib.push_back(0);
    fib.push_back(1);

    for (int i = 2;i < SIZE;i++) {
        fib.push_back(fib[i-1] + fib[i-2]);
        cout <<  i <<": " << fib[i] << endl;
    }
    return 0;
}
```

求解第n项时，图上共有$n+1$个顶点，从第二项开始，每个顶点与前两个顶点形成联系，故有$2n-2$条边。



15.2.1求解矩阵链

15.2-1
Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

求解最优解并打印的代码：

```
public int matrixChain_UpBottom(int p[]){
        if(p.length <= 1) {
            return 0;
        }

        // 存储最小乘积次数
        int m[][] = new int[p.length][p.length];
        for (int i = 0; i < m.length; i++) {
            Arrays.fill(m[i], Integer.MAX_VALUE);
            m[i][i] = 0;
        }

        // 存储分隔矩阵结果
```

```java
        int s[][] = new int[p.length][p.length];
        int res = matrixChain_UpBottom_helper(m, s, p, 1, p.length - 1);

        System.out.print("Minimum multiply count:  " + res + ", ");

        printSolution(s,1,p.length-1);

        return res;
}

private int matrixChain_UpBottom_helper(int m[][], int s[][], int[] p, int i, int j){
    for (int k = i; k < j; k++) {
        // 递归求值
        int temp = matrixChain_UpBottom_helper(m, s, p, i, k)
            + matrixChain_UpBottom_helper(m, s, p, k + 1, j)
            +p[i -1]*p[k]*p[j];
        if(m[i][j] > temp){
            m[i][j] = temp;
            s[i][j] = k;
        }
    }

    return m[i][j];
}

public void printSolution(int s[][], int i, int j) {
    if (i == j) {
        System.out.print("A" + i);
    }
    else {
        System.out.print("(");
        printSolution(s, i, s[i][j]);
        printSolution(s, s[i][j] + 1, j);
        System.out.print(")");
    }
}

public static void main(String[] args) {
    Main m = new Main();
    m.matrixChain_UpBottom(new int[]{5, 10, 3, 12, 5, 50, 6});
}
```

运行程序得到结果为：$((A1A2)((A3A4)(A5A6)))$。

15.4.5LCS

15.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

因为求解过程中使用了两重循环，可以得到复杂度为$O(n^2)$。代码如下：

```
int main() {
    vector<int> input = {5,10,3,12,5,50,6};
    int s = input.size();

    vector<int> dp(s+1, INT_MIN);
    // 记录前一个索引位置的数组
    vector<int> tmp(s+1, INT_MIN);
    int res = INT_MIN;
    int res_idx = -1;

    // 遍历构造每个位置最长递增序列长度
    for (int i = 0;i < s;i++) {
        dp[i] = 1;
        for (int j = 0;j < i;j++) {
                if (input[j] < input[i]) {
                    if (dp[j] + 1 > dp[i]) {
                        dp[i] = dp[j] + 1;
                        tmp[i] = j;
                    }
                }
            }
            if (dp[i] > res) {
                res = dp[i];
                res_idx = i;
            }
        }

        // 根据保存数据输出结果
        vector<int> r;
        while (res_idx >= 0 && res_idx < s) {
            r.push_back(input[res_idx]);
            if (res_idx == tmp[res_idx]) break;
            res_idx = tmp[res_idx];
        }

        cout<< res <<endl;

        for (auto i = r.rbegin(); i != r.rend(); i++) {
            cout<< *i << " ";
        }

        return 0;
    }
```

15.4.6LCS改进

15.4-6 ★
Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i-1$. Maintain candidate subsequences by linking them through the input sequence.)

思路：遍历数组，并维护一个列表s，其中$s[i]$代表长度为i的上升子序列的最大末尾元素**的索引**。遍历数组的每一个值时，都通过二分搜索找到$s[i]$中不小于当前值的第一个元素对应的索引，并用当前元素索引替换$s[i]$中找到的元素索引。同时维护一个列表tmp，$tmp[i]$代表下标为i的数在序列中对应的前一个下标。

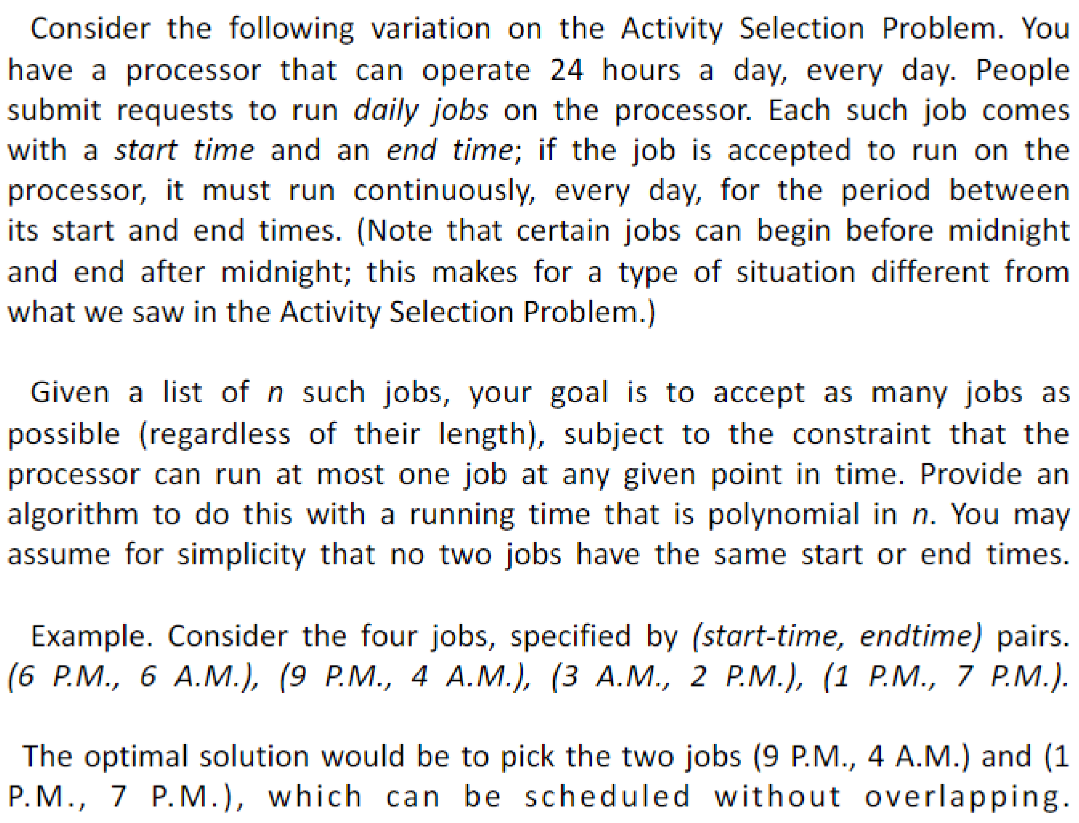
遍历复杂度为$O(n)$，二分查找复杂度$O(lgn)$。总的算法复杂度为$O(nlgn)$。

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

vector<int> input = {5,10,3,12,5,50,6,10,3,12,5};

void findAndReplace(vector<int>& s, vector<int>& tmp, int idx, int s_max_idx) {
    int left = 1, right = s_max_idx;

    while (left < right) {
        int mid = (left + right) >> 1;

        // 小于目标值，不符合条件
        if (input[s[mid]] < input[idx]) {
            left = mid + 1;
        }
        else {
            right = mid;
        }
    }

    // 找到大于等于input[idx]的第一个值，替换下标为idx
    s[left] = idx;

    // 保存idx的前一个数的下标
    tmp[idx] = s[left - 1];
}

int main() {
    int len = input.size();

    // 保存长度为i的子序列的最小末尾元素序号
    vector<int> s(len + 1, INT_MAX);
    s[0] = INT_MIN;

    // 当前s有合法数值的长度
    int s_max_idx = 0;

    // 保存子序列的回溯结果
    vector<int> tmp(len, INT_MAX);

    for (int i = 0;i < len;i++) {
        // 如果当前输入比s中最大的数都大
        if (s_max_idx == 0 || input[i] > input[s[s_max_idx]]) {
            tmp[i] = s[s_max_idx];
            s_max_idx++;
            s[s_max_idx] = i;
            continue;
        }

        findAndReplace(s, tmp, i, s_max_idx);
    }

    int res = 0;
    int cur = 0;
    for (int i = 1;i < len + 1;i++) {
        if (s[i] != INT_MAX) {
```

```cpp
                res++;
            } else {
                cur = s[i-1];
                break;
            }
        }

        cout << res << endl;

        // 打印结果（倒序打印出来的，偷懒没reverse）
        while (cur != INT_MAX) {
            cout << input[cur] << " ";
            cur = tmp[cur];
        }

        cout << endl;
    }
```

Greedy

活动选择环



Consider the following variation on the Activity Selection Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Activity Selection Problem.)

Given a list of *n* such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in *n*. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the four jobs, specified by *(start-time, endtime)* pairs. *(6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.).*

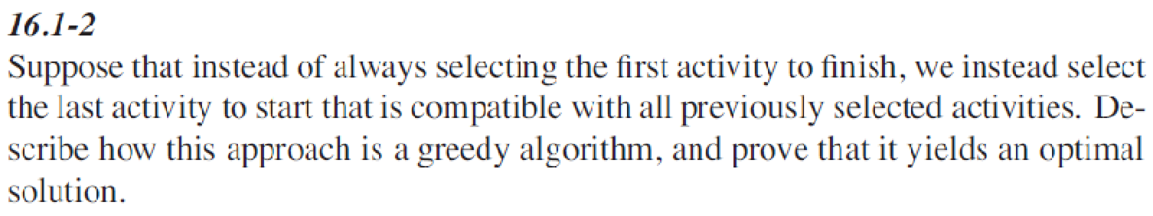
The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

此问题将可选择空间变成了一个环形。为了方便处理，首先将输入转化为24小时制，如题中的输入变为：$(18,6),(21,4),(3,14),(1,7)$。

首先，最优解一定在两种情况内：包含凌晨的工作，不包含凌晨的工作。

- 不包含凌晨工作时，只需要从输入中去掉所有凌晨的工作，问题退化为普通的活动选择问题，直接求解即可。
- 包含凌晨工作时，因为凌晨的工作两两之间冲突，因此只可能包含其中的一个工作，首先一一列举可能包含的凌晨工作，再求解一天除去凌晨工作时段之后的活动安排最优解，最后比较所有情况，得出最优解。

16.1.2活动选择变体

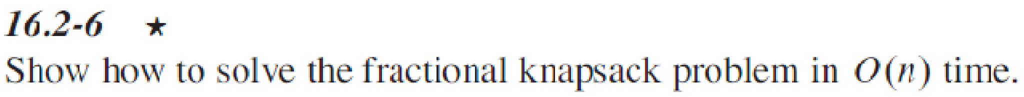


16.1-2
Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

算法：首先按照最晚开始时间对输入进行排序，依次选择与前一个活动时间兼容的数组。

证明：考虑任意非空子问题S_k，令a_m是S_k中开始时间最晚的活动，则a_m在S_k的某个最大兼容活动子集中。因为若S_k的所有最大兼容活动子集都不包含a_m，令a_j为其中开始时间最晚的活动，则a_m的开始时间晚于a_j，因此一定可以将a_j替换为a_m，且活动之间都不相交。这与前提矛盾，因此一定有一个S_k的最大兼容活动子集且它包含a_m。

16.2.6分数背包


16.2-6 ★
Show how to solve the fractional knapsack problem in $O(n)$ time.

若物品数目不超过两个，可以直接计算出结果。

若物品数目大于两个，首先找出单价的中位数m，设单价高于m的物品总量为w，若w不超过背包容量，则在剩下的$n/2$中递归寻找$W-w$的物品。

若超过背包容量，则在此$n/2$范围内寻找W重的物品。

得出时间复杂度为$T(n)=T(n/2)+O(n)$，由主方法计算得到$T(n)=O(n)$。