习题总结

第一二三次作业

A1-Q1　　排序算法+分治法

1. Please write the pseudocode of ***Quick Sort***.

```
ALGORITHM quicksort(A, lo, hi)   //lo 最左边值，hi 最右边值，A 是数组
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1 ) //中间值p 左边进行排序
        quicksort(A, p + 1, hi)  //中间值p 右边进行排序
END

// 以最后一个元素作为主元素，遍历前面的元素，判断主元素最后的位置，最后交换到这个位置上
// 当然，以第一个元素为主元素，或者前后指针一起移动都可以
partition(A, lo, hi)
    pivot := A[hi] //暂定基准是最右边的值
    i := lo - 1    //i初始值是lo -1
    for j := lo to hi - 1 do //从最左边到最右边第二个数开始循环
        if A[j] <= pivot then //如果A[j] 小于等于中间值即最右边的值
            i := i + 1   //i加1
            swap A[i] with A[j]  //交换A[i] 和 A[j]的位置
    swap A[i + 1] with A[hi]  //循环结束，交换A[i+1] 和 A[hi]
    return i + 1
```

A1-Q2　　算法复杂度

2. Which of the following statements is/are valid? (more than one answer is possible)

A. Time Complexity of Quick Sort is $\Theta(n^2)$

B. Time Complexity of Quick Sort is $O(n^2)$

C. For any two functions $f(n)$ and g(n), we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

D. Time complexity of all computer algorithms can be written as $\Omega(1)$

快排的时间复杂度可以写成：$\Theta\,(n\ logn)$ 、$O\,(x)$ 其中$x \geqslant n\ logn$ 、$\Omega\,(x)$ 其中$x \leqslant n\ logn$ ，所以A错，B对

C选项，如果$f(x)$ 是g(x) 的精确渐进边界，那么当且仅当$f(x)$ 既是g(x) 的上确界，又是g (x) 的下确界 （高数常识）

D选项，任何算法的下限都可以写成 $\Omega(1)$

所以最终的答案是 BCD

A1-Q3 算法复杂度

(a) The 1-D Discrete Fourier Transform (**DFT**) of N samples of a signal $f(x)$ sampled at $x = 0,1,2 \ldots N-1$ is:

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{\frac{-j2\pi ux}{N}}$$

for $u = 0, 1, \ldots, N-1$, and $j = \sqrt{-1}$. If $f(x)$ is generally complex, how many complex multiplications are needed to compute the Fourier Transform of the given sample?

(b) A significantly more efficient algorithm for computing the DFT is called the Fast Fourier Transform **(FFT)**. The FFT algorithm has the following recurrence relation:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n,\ for\ n > 1\ with\ T(1) = 1,\ n\ is\ a\ power\ of\ 2$$

Find the complexity of the FFT.

(a) 要计算N次F(u)，而对于每个F(u)又要计算N次复数乘法，最终也就是N^2次

(b) $T(n) = 2\,T\left(\frac{n}{2}\right) + n,\ for\ n > 1\ with\ T(1) = 1,\ n\ is\ a\ power\ of\ 2$

$$T(2^m) = 2\,T(2^{m-1}) + 2^m = \cdots = (m+1)2^m$$

$$\rightarrow T(n) = O(\log n + 1)n = O(n \log n)$$

A1-P 动态规划

Problem:

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount, including the amount 0 which can be made up of 0 coin. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Case 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Case 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Case 3:

```
Input: coins = [1], amount = 0
Output: 0
```

1. 定义 dp 数组/函数的含义，dp[n]表示总金额是n的情况下所需要的最少硬币数
2. 找出递推关系式
3. 确定初始值

$$dp(n) = \begin{cases} 0, n = 0 \\ -1, n < 0 \\ min\{dp(n - coin) + 1 | coin \in coins\}, n > 0 \end{cases}$$

A1-P 动态规划

```java
int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    // 数组大小为 amount + 1，初始值也为 amount + 1
    Arrays.fill(dp, amount + 1);

    // base case
    dp[0] = 0;
    // 外层 for 循环在遍历所有状态的所有取值
    for (int i = 0; i < dp.length; i++) {
        // 内层 for 循环在求所有选择的最小值
        for (int coin : coins) {
            // 子问题无解，跳过
            if (i - coin < 0) {
                continue;
            }
            dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
        }
    }
    return (dp[amount] == amount + 1) ? -1 : dp[amount];
}
```

dp数组迭代解法（自底向上）

```java
int[] memo;

int coinChange(int[] coins, int amount) {
    memo = new int[amount + 1];
    // 备忘录初始化为一个不会被取到的特殊值，代表还未被计算
    Arrays.fill(memo, -666);

    return dp(coins, amount);
}

int dp(int[] coins, int amount) {
    if (amount == 0) return 0;
    if (amount < 0) return -1;
    // 查备忘录，防止重复计算
    if (memo[amount] != -666)
        return memo[amount];

    int res = Integer.MAX_VALUE;
    for (int coin : coins) {
        // 计算子问题的结果
        int subProblem = dp(coins, amount - coin);
        // 子问题无解则跳过
        if (subProblem == -1) continue;
        // 在子问题中选择最优解，然后加一
        res = Math.min(res, subProblem + 1);
    }
    // 把计算结果存入备忘录
    memo[amount] = (res == Integer.MAX_VALUE) ? -1 : res;
    return memo[amount];
}
```

带备忘录的递归（自顶向下）

A2-Q1　　算法复杂度/主定理

对于 $T(n) = aT(\frac{n}{b}) + f(n)$ ，有三种情况：

$f(n)$ 与 $n^{\log_b(a)}$ 进行比较

1. 存在ε>0，且 $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$，则有 $T(n) = \Theta\left(n^{\log_b a}\right)$

2. 存在ε≥0，且 $f(n) = \Theta\left(n^{\log_b a}\log^{\epsilon} n\right)$ ，则有 $T(n) = \Theta\left(n^{\log_b a}\log^{\epsilon+1} n\right)$

特殊情况：$f(n) = \Theta\left(n^{\log_b a}\right)$

3. 存在ε>0，且 $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$，同时存在c<1且满足 $af(n/b) \leq cf(n)$ ，则有 $T(n) = \Theta(f(n))$

1. Calculate the time complexity of the following recurrence relation using the master theorem.

(a)

$$T(N) = 2T\left(\frac{N}{2}\right) + NlogN, \qquad T(1) = 1$$

(b)

$$T(N) = 4T\left(\frac{N}{2}\right) + N, \qquad T(1) = 0$$

(c)

$$T(N) = T\left(\frac{N}{2}\right) + 2^N, \qquad T(1) = 1$$

(a) 满足情形二 $T(n) = \Theta(nlog^2 n)$

(b) 满足情形一 $T(n) = \Theta(n\text{^}log_b(a)) = \Theta(n^2)$

(c) 满足情形三 $T(n) = \Theta(2\text{^}n)$

A2-Q2 排序算法+分治法

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

Divide

Conquer

Combine

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \le C[j]$
    $A[k] \leftarrow B[i];\ i \leftarrow i+1$
  else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
  $k \leftarrow k+1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Time Complexity: $T(n) = O(nlogn)$

Space Complexity: $S(n) = O(n)$

A2-Q3　　分支限界法

3. Suppose there are six items with weights of (6, 2, 4, 3, 9, 12) and values of (9, 4, 6, 5, 14, 20). The capacity of the backpack is 16. You cannot break an item, either pick the complete item or don't pick it. Can you use ***branch and bound method*** to determine which items you should pick in order to maximize the value of the items without surpassing the capacity of your backpack? You should write the pseudocode (return the maximum value) and draw a solution space tree for this problem.

物品	重量(w)	价值(v)	价值密度(w/v)
1	2	4	2.0
2	3	5	1.67
3	12	20	1.67
4	9	14	1.56
5	6	9	1.5
6	4	6	1.5

优先队列分支限界法

$$ub = v + (W - w) \times (v_{i+1} / w_{i+1})$$

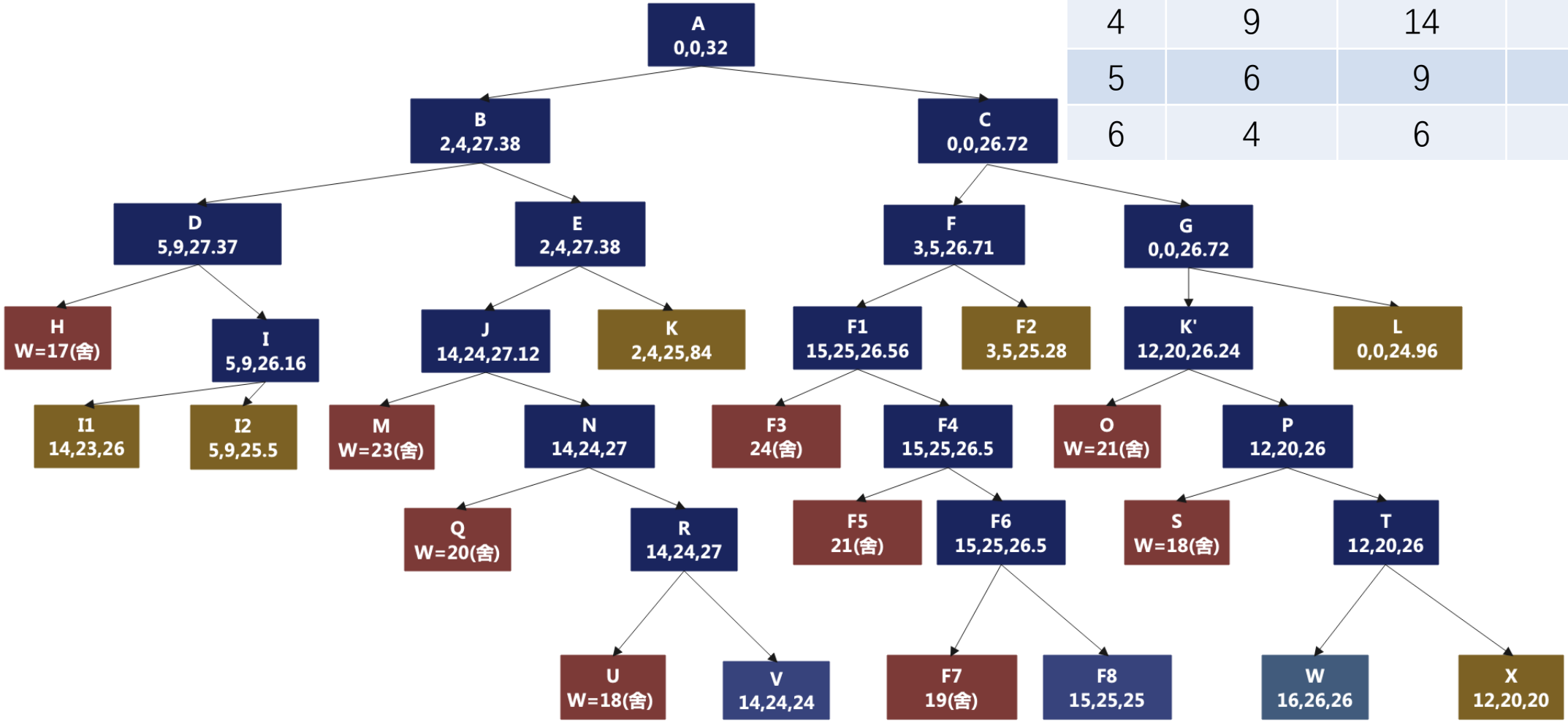
限界函数

v：前i个物品的总价值
w：前i个物品的总重量

限界函数表示，将第i+1个物品放入背包时最大能获得的价值

A2-Q3 分支限界法

物品	重量(w)	价值(v)	价值密度(v/w)
1	2	4	2.0
2	3	5	1.67
3	12	20	1.67
4	9	14	1.56
5	6	9	1.5
6	4	6	1.5



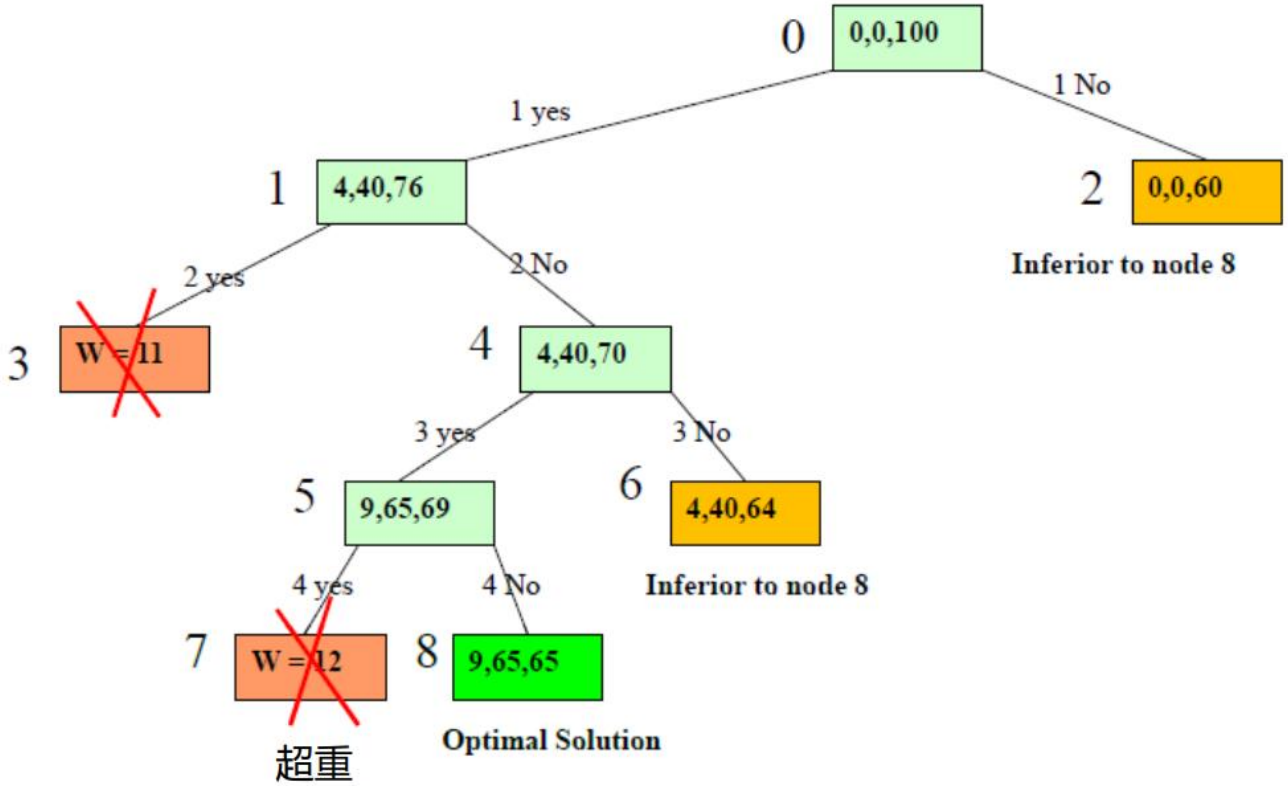
最优解

例题

分支限界法

$\mathbf{W} = 10, n = 4$

Item	Weight	Value	Value Density
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4



A2-Q3 分支限界法

```
//优先队列式分支限界法，返回最大价值，n为物品数目，c为背包容量，w为物品重量，p为物品价值
ALGOTRITHM MaxKnapsack( n, c, w[], p[] ) {
    //算法开始之前，已经按照物品单位价值率按照降序顺序排列好了
    cw = 0, cp = 0; //cw为当前装包重量，cp为当前装包价值
    bestp = 0; //当前最优值
    i=1, up = Bound(1); //函数Bound(i)计算当前结点相应的价值上界
    while( i != n+1 ) { //非叶子结点
        //首先检查当前扩展结点的左儿子结点为可行结点
        if( cw + w[i] <= c ) { //左孩子结点为可行结点
            if( cp + p[i] > bestp ) bestp = cp + p[i];
            //将左孩子结点插入到优先队列中
            AddLiveNode( up, cp + p[i] + cw + w[i], true, i + 1);
        }
        up = Bound(i+1);
        //检查当前扩展结点的右儿子结点
        if( up >= bestp ) //右子树可能包含最优解
            //将右孩子结点插入到优先队列中
            AddLiveNode( up, cp, cw, false, i + 1);
        //从优先级队列（堆数据结构）中取下一个扩展结点N
        H->DeleteMax(N) ;
        i = N.level;
    }
}
```

```
//计算结点所对应的价值的上界
Bound(i) {
    cleft = c - cw; //剩余背包容量
    b = cp; //价值上界
    //以物品单位重量价值递减顺序装填剩余容量
    while( i <= n && w[i] <= cleft ){
        cleft -= w[i]; //w[i]表示i物品的重量
        b += p[i]; //p[i]表示i物品的价值
        i++;
    }
    //装填剩余容量装满背包
    if(i<=n) b += p[i]/w[i] * cleft;
    return b;
}
```

A2-P 回溯法

Problem:

Please Write a program to solve the Sudoku puzzle. A sudoku solution should follow all of the following three rules:

1. Each of the digits 1-9 can only appear once in each row.
2. Each of the digits 1-9 can only appear once in each column.
3. Each of the digits 1-9 can only appear once in each of the 3x3 sub-boxes of the grid.

Explanation:

1. 路径：也就是已经做出的选择，即已经填好数字的空格列表。
2. 选择列表：也就是当前可以做的选择。此时需要写一个函数判断当前选择是否合法。
3. 结束条件：也就是到达决策树底层的条件。这里遍历完棋盘即可。

A2-P 回溯法

```
void solveSudoku(vector<vector<char>>& board) {
    backtracking(board);
}


bool backtracking(vector<vector<char>>& board) {
    for (int i = 0; i < board.size(); i++) {        // 遍历行
        for (int j = 0; j < board[0].size(); j++) { // 遍历列
            if (board[i][j] != '.') continue;
            for (char k = '1'; k <= '9'; k++) {     // (i, j) 这个位置放k是否合适
                if (isValid(i, j, k, board)) {
                    board[i][j] = k;                // 放置k
                    if (backtracking(board)) return true; // 如果找到合适一组立刻返回
                    board[i][j] = '.';              // 回溯，撤销k
                }
            }
            return false;                           // 9个数都试完了，都不行，那么就返回false
        }
    }
    return true; // 遍历完没有返回false，说明找到了合适棋盘位置了
}
```

```
bool isValid(int row, int col, char val, vector<vector<char>>& board) {
    for (int i = 0; i < 9; i++) { // 判断行里是否重复
        if (board[row][i] == val) {
            return false;
        }
    }
    for (int j = 0; j < 9; j++) { // 判断列里是否重复
        if (board[j][col] == val) {
            return false;
        }
    }
    int startRow = (row / 3) * 3;
    int startCol = (col / 3) * 3;
    for (int i = startRow; i < startRow + 3; i++) { // 判断9方格里是否重复
        for (int j = startCol; j < startCol + 3; j++) {
            if (board[i][j] == val ) {
                return false;
            }
        }
    }
    return true;
}
```

A3-Q2　　动态规划

2. You are given an array *prices* where prices[i] is the price of a given stock on the i-th day. Find the maximum profit you can achieve. You can only hold at most one share of the stock at any time. After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day). You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

```
Example:
Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

Please write the pseudocode or C++ code of ***dynamic programming*** to solve this problem

求解：能够获取的最大利润

约束条件　一：冷冻期为一天（卖出股票后，你无法在第二天买入股票）

　　二：必须在再次购买前出售掉之前的股票

A3-Q2 动态规划

1. 定义 dp 数组/函数的含义，dp[i]表示金额第i天结束之后的累积最大收益

每天存在三种情况：

- 目前持有一只股票，记为$dp[i][0]$
- 目前不持有任何股票，处在冷冻期中，记为$dp[i][1]$
- 目前不持有任何股票，不处在冷冻期中，记为$dp[i][2]$

2. 找出递推关系式

$dp[i][0] = \max(dp[i-1][0], dp[i-1][2] - prices[i])$

$dp[i][1] = dp[i-1][0] + prices[i]$

$dp[i][2] = \max(dp[i-1][1], dp[i-1][2])$

最终目标：$\max(dp[n-1][1], dp[n-1][2])$

3. 确定初始值

$$\begin{cases} f[0][0] & = -prices[0] \\ f[0][1] & = 0 \\ f[0][2] & = 0 \end{cases}$$

A3-Q2 动态规划

```cpp
int maxProfit(vector<int>& prices) {
    if (prices.empty()) {
        return 0;
    }

    int n = prices.size();
    // f[i][0]: 手上持有股票的最大收益
    // f[i][1]: 手上不持有股票，并且处于冷冻期中的累计最大收益
    // f[i][2]: 手上不持有股票，并且不在冷冻期中的累计最大收益
    vector<vector<int>> f(n, vector<int>(3));
    f[0][0] = -prices[0];
    for (int i = 1; i < n; ++i) {
        f[i][0] = max(f[i - 1][0], f[i - 1][2] - prices[i]);
        f[i][1] = f[i - 1][0] + prices[i];
        f[i][2] = max(f[i - 1][1], f[i - 1][2]);
    }
    return max(f[n - 1][1], f[n - 1][2]);
}
```

dp数组迭代解法，时间复杂度O(n)，空间复杂度O(n)

```cpp
int maxProfit(vector<int>& prices) {
    if (prices.empty()) {
        return 0;
    }

    int n = prices.size();
    int f0 = -prices[0];
    int f1 = 0;
    int f2 = 0;
    for (int i = 1; i < n; ++i) {
        int newf0 = max(f0, f2 - prices[i]);
        int newf1 = f0 + prices[i];
        int newf2 = max(f1, f2);
        f0 = newf0;
        f1 = newf1;
        f2 = newf2;
    }

    return max(f1, f2);
}
```

优化空间复杂度，时间复杂度O(n)，空间复杂度O(1)

A3-Q3 贪心算法

3. The background is the same as the previous question. On each day, you may decide to buy and/or sell the stock. However, you can buy it then immediately sell it on the same day (no cooling time). Find and return the maximum profit you can achieve.

```
Example:
Input: [7,1,5,3,6,4]
Output: 7
Explanation: [do nothing, buy, sell, buy, sell, do nothing]
```

Please write the pseudocode or C++ code of ***greedy algorithm*** to solve this problem

```
int maxProfit(vector<int>& prices) {
    int ans = 0;
    int n = prices.size();
    for (int i = 1; i < n; ++i) {
        ans += max(0, prices[i] - prices[i - 1]);
    }
    return ans;
}
```

策略：只要后一天比当天股价高就买入，并在后一天卖出；否则就不买入。

“贪心”在于，对于“后一天的股价 – 当天的股价”，得到的结果有正数、负数、0三种可能，

贪心算法的决策是只加正数。

A3-P 动态规划

Problem:

Matrix chain multiplication is an optimization problem that to find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.

The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))

There is a sequence of N matrices $A_1, A_2, \ldots A_n$, where A_i is a $P_{i-1} \times P_i$ matrix. You are given a input vector $P = [P_0, P_1 \ldots P_n]$ of the matrix chain above , and should determine the order of multiplication that minimize the total number of basic operations. Return the minimum number of basic operations.

Case 1:

```
Input: P = [10, 30, 5, 60]
Output: 4500
Explanation:
(AB)C needs (10×30×5) + (10×5×60) = 4500 operations,
A(BC) needs (30×5×60) + (10×30×60) = 27000 operations.
```

1. 定义 dp 数组/函数的含义，$dp[i][j]$表示计算$A_iA_{i+1}\cdots A_j$的最优代价。

2. 找出递推关系式

$$dp[i][j] = \min_{i \le k \le j}\{dp[i,k] + dp[k+1,j] + P_{i-1}P_kP_j\}$$

$P_{i-1}P_kP_j$ 是$(A_iA_{i+1}\cdots A_k)(A_{k+1}\cdots A_j)$的代价

3. 确定初始值，$dp[i][i] = 0$

最后需要的是$dp[1][n\text{-}1]$

A3-P 动态规划

```
int matrixChainMultiplication(vector<int> const &dims)
{
    int n = dims.size();
    int dp[n + 1][n + 1];

    for (int i = 1; i <= n; i++) {
        dp[i][i] = 0;
    }

    for (int len = 2; len <= n; len++) {
        for (int i = 1; i <= n - len + 1; i++)
        {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; j < n && k <= j - 1; k++)
            {
                int cost = dp[i][k] + dp[k + 1][j] + dims[i - 1] * dims[k] * dims[j];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }
    return dp[1][n - 1];
}
```