

# Assignment Two

Name	Student Number
胡锦涛	2053300

**Calculate the time complexity of the following recurrence relation using the master theorem**

- $T(N) = 2T(\frac{N}{2}) + N \log N, T(1) = 1$

解答:

对于  $T(N) = aT(\frac{N}{b}) + f(N)$

记  $d = \log_b a = 1$ , 且  $f(N) = N \log N = \Theta(N^c \log^k N)$  解得  $c = 1, k = 1$

而  $d = c = 1$ , 满足master theorem 的 case 2, 则

$$T(N) = \Theta(N^d \log^{k+1} N) = \Theta(N \log^2 N)$$

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

解答:

对于  $T(N) = aT(\frac{N}{b}) + f(N)$

记  $d = \log_b a = 2$ , 且  $f(n) = N = O(N^c)$  解得  $c = 1$

而  $c = 1 < d = 2$ , 满足master theorem 的 case 1, 则

$$T(n) = \Theta(n^d) = \Theta(n^2)$$

- $T(N) = T(\frac{N}{2}) + 2^N$

解答:

对于  $T(N) = aT(\frac{N}{b}) + f(N), T(1) = 1$

记  $d = \log_b a = 0$ , 且  $f(n) = 2^N = \Omega(N^c)$  解得  $c = 1$

而  $c = 1 > d = 0$ , 满足master theorem 的 case 3, 则

$$T(n) = \Theta(f(n)) = \Theta(2^N)$$

**Write the pseudocode of merge sort, and analyze the time complexity and space complexity**

解答：

```

Merge(A, p, q, r) {
    n1 = q - p + 1
    n2 = r - q
    创建长度为n1的数组L, 和长度为n2的数组R
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    i = 1
    j = 1
    for k = p to r
        if L[i] <= R[j]
            A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
    }

MergeSort(A, p, r) {
    if p < r
        q = (p + r) / 2
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)
    }

```

复杂度分析：

记问题的输入规模为 $n$ ，即这里需要对 $n$ 个数进行归并排序。

- 时间复杂度：记归并排序 $n$ 个数的最坏情况所需要的运行时间为  $T(n)$

当 $n = 1$ 时，归并排序一个数需要常量时间，即  $T(1) = \Theta(1)$

当 $n > 1$ 时，根据伪代码可以将运行时间分解为以下三部分：

- 计算子数组的中间位置，需要常量时间，即  $A(n) = \Theta(1)$
- 递归的求解两个规模为  $\frac{n}{2}$  的子问题，需要  $2T(\frac{n}{2})$  的运行时间
- 合并操作需要  $B(n) = \Theta(n)$  的运行时间，具体分析如下：

对照上面伪代码中的  $\text{MERGE}(A, p, q, r)$ ，合并长度为  $n = r - p + 1$  长度的子数组需要的运行时间可以分解为以下两部分：

- 创建两个长度分别为  $n_1 = q - p + 1, n_2 = r - q$  的子数组需要  $\Theta(n_1 + n_2) = \Theta(n)$  的运行时间
- 将两个子数组合并需要  $n$  次迭代, 每次迭代需要常数时间, 共需要  $\Theta(n)$  的运行时间, 相较于合并操作的  $B(n) = \Theta(n)$ , 第一步的  $A(n) = \Theta(1)$  可以忽略, 则有  $T(n)$  的递归式:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

这里我们采用master theorem:

对于  $T(n) = aT(\frac{n}{b}) + f(n)$ , 有  $a = b = 2$

记  $d = \log_b a = 1$ , 且  $f(n) = \Theta(n^c) = \Theta(n)$ , ( $c = 1$ )

而  $d = c = 1$ , 满足master theorem 的 case 2, 则

$$T(n) = \Theta(n^d \log n) = \Theta(n \log n)$$

**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 any 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**

解答:

```

Constraint(c) {
    if c > 16
        return false
    return true
}

Bound(P, v, t) {
    if (v + P[size] - P[t] < best) {
        return false
    }
    return true
}

Backtrack(W, V, P, c, v, t) {
    if (t > size) {
        best = max(best, v)
        return
    }
    if (Constraint(c) && Bound(P, v, t)) {
        Backtrack(W, V, P, c + W[t], v + v[t], t + 1)
        Backtrack(W, V, P, c, v, t + 1)
    }
}

Calc() {
    size = 6
    best = 0
    W = {6, 2, 4, 3, 9, 12}
    V = {9, 4, 6, 5, 14, 20}
    创建前缀和数组P = {9, 0, 0, 0, 0, 0}
    for i = 2 to size {
        P[i] = P[i - 1] + V[i]
    }
    Bactrack(W, V, P, 0, 0, 0)
    return best
}

```

已通过如下代码在Windows11, c++17环境下进行过正确性验证

```

#include <iostream>
#include <cstdio>
#include <fstream>
#include <algorithm>
#include <cmath>
#include <deque>
#include <vector>
#include <queue>
#include <string>
#include <cstring>
#include <map>
#include <unordered_map>
#include <stack>
#include <set>
#include <numeric>
#include <iomanip>
typedef long long ll;
using namespace std;

int capacity, best;

vector<int> w, v, p;

bool constraint(int c) {
    return c <= capacity;
}

bool bound(int cntv, int t, int size) {
    return cntv + p[size - 1] - p[t] >= best;
}

void backtrack(int cntc, int cntv, int size, int t) {
    if (constraint(cntc)) {
        if (t == size) {
            best = max(best, cntv);
            return;
        }
        if (bound(cntv + v[t], t + 1, size)) {
            backtrack(cntc + w[t], cntv + v[t], size, t + 1);
        }
        if (bound(cntv, t, size)) {
            backtrack(cntc, cntv, size, t + 1);
        }
    }
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

```

```

cout.tie(nullptr);
int size;
cin >> size >> capacity;
for (int i = 0; i < size; i++) {
    int num;
    cin >> num;
    w.push_back(num);
}
for (int i = 0; i < size; i++) {
    int num;
    cin >> num;
    v.push_back(num);
    if (i == 0) {
        p.push_back(num);
    }
    else {
        p.push_back(p[i - 1] + num);
    }
}
backtrack(0, 0, size, 0);
cout << best << '\n';

return 0;
}


// test data
// 6 16
// 6 2 4 3 9 12
// 9 4 6 5 14 20

```

解空间树：

图例：x->(c, v)

其中x代表是第x次执行backtrack方法，c代表当前背包中所有物品的体积之和，v代表当前背包中所有物品的价值之和

 解空间树