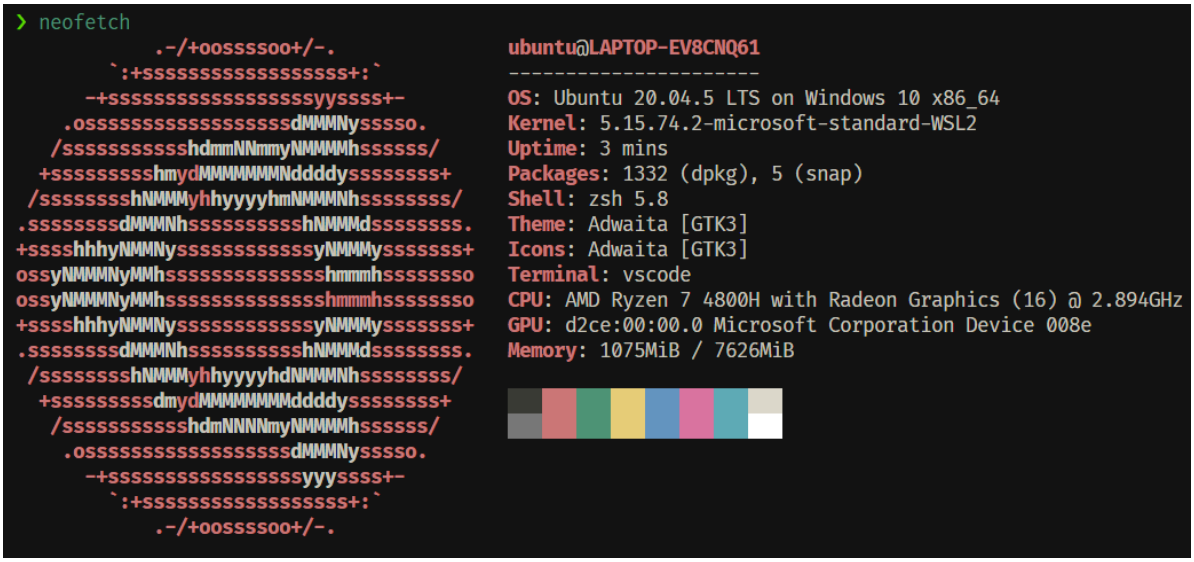
算法基础 动态规划实验报告

实验设备和环境

如图所示。



实验内容及要求

矩阵链乘最佳方案

- n 个矩阵链乘，求最优链乘方案，使链乘过程中乘法运算次数最少。
- n 的取值 5, 10, 15, 20, 25，矩阵大小见 2_1_input.txt。
- 求最优链乘方案及最少乘法运算次数，记录运行时间，画出曲线分析。
- 仿照 P214 图 15-5，打印 n=5 时的结果并截图。

最长公共子序列

- 给定两个序列 X、Y，求出这两个序列的最长公共子序列（某一个即可）。
- X, Y 序列由 A、B、C、D 四种字符构成,序列长度分别取 10、15、20、25、30，见 2_2_input.txt。
- 打印最长公共子序列，记录运行时间，画出曲线分析。

方法和步骤

矩阵链乘最佳方案

方便起见，定义一个模拟的二维数组（C++ 不能很好地初始化一片连续的动态大小的二位数组）。

```
template <typename T> class Table {
  public:
    explicit Table(size_t n) : n(n), data(std::vector<T>(n * n)) {}
    inline T get(size_t x, size_t y) const { return data[x * n + y]; }
    inline void set(size_t x, size_t y, T value) { data[x * n + y] = value; }
    inline size_t size() const { return n; }

  private:
    size_t n;
    std::vector<T> data;
};

using u64 = uint64_t;
```

需要实现两个函数，一个用于计算矩阵链乘的最佳方案，返回 m 和 s 两个二维数组。

```
/**
 * @brief Compute the best matrix chain multiplication plan
 *
 * @param p Dimensions for matrices
 * @return std::tuple<u64, Table<u64>, Table<u64>>
 *         min_cost, m, s
 */
inline std::tuple<u64, Table<u64>, Table<u64>>
matrix_chain_multiply(const std::vector<u64> &p) {
    const int n = p.size() - 1;

    // Initialize n * n matrix m & s (with all 0)
    Table<u64> m(n);
    Table<u64> s(n);

    // do a bottom-up search
    // iter over chain length
    for (size_t l = 2; l ⩽ n; l++) {
        // iter over (i, j)
        for (size_t i = 0; i ⩽ n - l; i++) {
            const size_t j = i + l - 1;
            // set to +\inf
            m.set(i, j, UINT64_MAX);
            // split point k
            // (A[i] ... A[k]) * (A[k+1] ... A[j])
            for (size_t k = i; k < j; k++) {
                const u64 q =
                    m.get(i, k) + m.get(k + 1, j) + p[i] * p[k + 1] * p[j + 1];
                if (q < m.get(i, j)) {
                    // better solution
                    m.set(i, j, q);
                    s.set(i, j, k);
                }
            }
        }
    }
```

```
    return {m.get(0, n - 1), m, s};
}
```

上面的函数在实现上考虑了 C++ 的数组下标从 0 开始，因此与书上的算法略有不同。

另一个函数用于根据二维数组 s，返回矩阵链乘某个方案的括号表示。

```
/**
 * @brief Return the paren repr of matrix chain multiplication plan in string
 *
 * @param s Table s of matrix chain multiplication
 * @return std::string paren repr in string
 */
inline std::string paren_repr(const Table<u64> &s) {
    auto recur_paren_repr = [&](auto &&self, u64 i, u64 j) {
        if (i == j)
            return std::string("A");
        auto repr = std::string("(");
        repr += self(self, i, s.get(i, j));
        repr += self(self, s.get(i, j) + 1, j);
        repr += ")";
        return repr;
    };
    return recur_paren_repr(recur_paren_repr, 0, s.size() - 1);
}
```

最长公共子序列

用枚举类型约定 b 数组所表示的方位，同上用一维数组模拟二维数组。

```
enum CHOICE { L, T, LT };

template <typename T> class Table {
  public:
    explicit Table(size_t m, size_t n)
        : m(m), n(n), data(std::vector<T>(m * n)) {}
    inline T get(size_t x, size_t y) const { return data[x * n + y]; }
    inline void set(size_t x, size_t y, T value) { data[x * n + y] = value; }
    inline size_t size() const { return n; }

  private:
    size_t m;
    size_t n;
    std::vector<T> data;
};
```

最长公共子序列的计算分为两部分，计算 b, c 数组以及根据 b 数组还原最长公共子序列。

```
/**
 * @brief Return LCS of two strings
 *
 * @param x The first string
 * @param y The second string
 * @return std::string The LCS
 */
```

```cpp
inline std::string longest_common_sequence(std::string &x, std::string &y) {
    auto m = x.length();
    auto n = y.length();

    // b and c are initialized with 0
    auto b = Table<CHOICE>(m, n);
    auto c = Table<unsigned>(m + 1, n + 1);

    // bottom up iteration
    for (auto i = 0; i < m; i++) {
        for (auto j = 0; j < n; j++) {
            if (x[i] == y[j]) {
                c.set(i + 1, j + 1, c.get(i, j) + 1);
                b.set(i, j, CHOICE::LT);
            } else if (c.get(i, j + 1) ≥ c.get(i + 1, j)) {
                c.set(i + 1, j + 1, c.get(i, j + 1));
                b.set(i, j, CHOICE::T);
            } else {
                c.set(i + 1, j + 1, c.get(i + 1, j));
                b.set(i, j, CHOICE::L);
            }
        }
    }

    // collect final string
    auto i = m - 1;
    auto j = n - 1;
    auto result = std::string();
    while ((i + 1 ≠ 0U) && (j + 1 ≠ 0U)) {
        if (b.get(i, j) == CHOICE::LT) {
            result += x[i];
            i -= 1;
            j -= 1;
        } else if (b.get(i, j) == CHOICE::T) {
            i -= 1;
        } else {
            j -= 1;
        }
    }
    std::reverse(result.begin(), result.end());
    return result;
}
```

输入输出和时间测量

利用 `sys/time.h`。

```cpp
// test according to specified methods and scales
struct timeval t1;
struct timeval t2;
double timeuse_ms;

auto time_file = std::ofstream("output/time.txt", std::ios_base::out);
auto result_file = std::ofstream("output/result.txt", std::ios_base::out);
```

```
for (const auto &testcase : testcases) {

    gettimeofday(&t1, nullptr);
    // 100 times
    for (auto i = 0; i < 99; i++)
        auto result = matrix_chain_multiply(testcase);
    auto result = matrix_chain_multiply(testcase);
    gettimeofday(&t2, nullptr);

    timeuse_ms = static_cast<double>(t2.tv_sec - t1.tv_sec) * 1000.0 +
                 static_cast<double>(t2.tv_usec - t1.tv_usec) / 1000.0;

    // print info
    // ......

    // save to files
    time_file << timeuse_ms << std::endl;
    result_file << min_cost << std::endl;
    result_file << paren_repr(s) << std::endl;
}

result_file.close();
time_file.close();
```

将文件内容读入到 `testcases`，然后遍历，测量每一个样例需要的时间，并打印、保存到文件即可。

因为样例比较小，这里我们每个样例都重复了 100 次。

结果与分析

矩阵链乘最佳方案

`n = 5` 时，结果如图所示。

```
❯ ./build/main
    154865959097238     138766801119366     183439291324068     120958281818244                   0
    128049683226820     105723424955724     119490227350806                   0
     74062781976714      43981152513978                   0
     15903764653528                   0
                  0

                  1                   4                   4                   4
                  1                   3                   3
                  1                   2
                  1
```

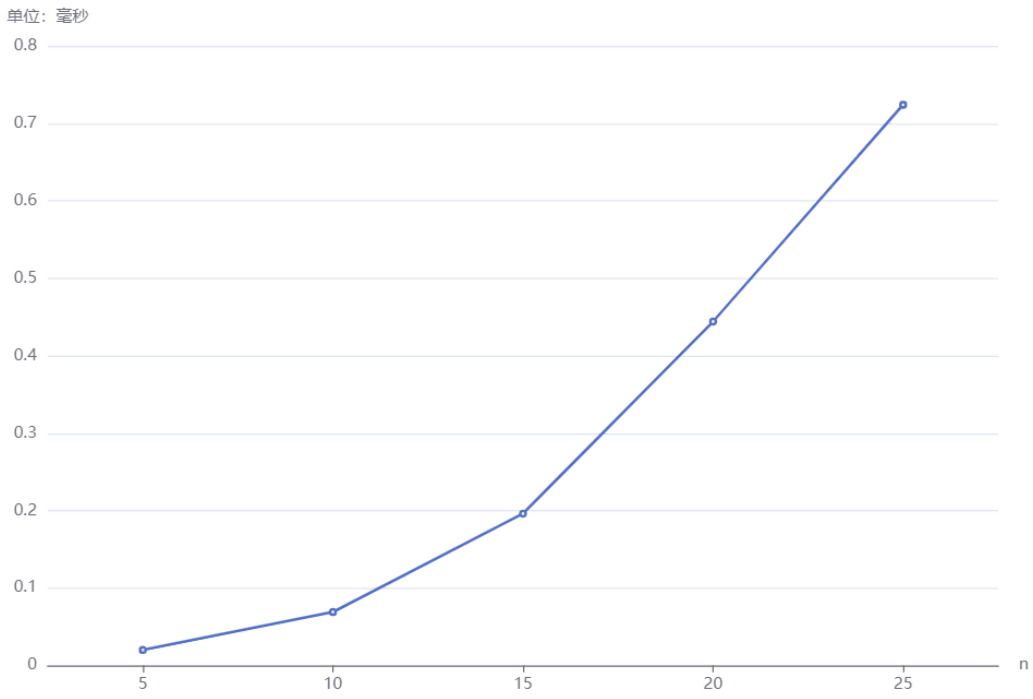
输出 `result.txt` 如下。

```
154865959097238
(A(((AA)A)A))
42524697503391
((AA)(((((((AA)A)A)A)A)A)A))
5400945319618
((((((((((((AA)A)A)A)A)A)A)A)A)A)A)A)A)
319329979644400
((A(A(A(A(A(A(A(A(A(A(A(AA)))))))))))))((((AA)A)A)A))
574911761218280
((A(A(A(A(A(A(A(A(A(AA)))))))))))((((((((((((AA)A)A)A)A)A)A)A)A)A)A)A)A))
```

`time.txt` 如下（单位：毫秒）。

```
0.02
0.069
0.196
0.444
0.724
```

画出对应曲线图如下。



复杂度根据理论推导应该是 $O(n^3)$ 级别，实际从图可以看出，应该是符合多项式级别的。

最长公共子序列

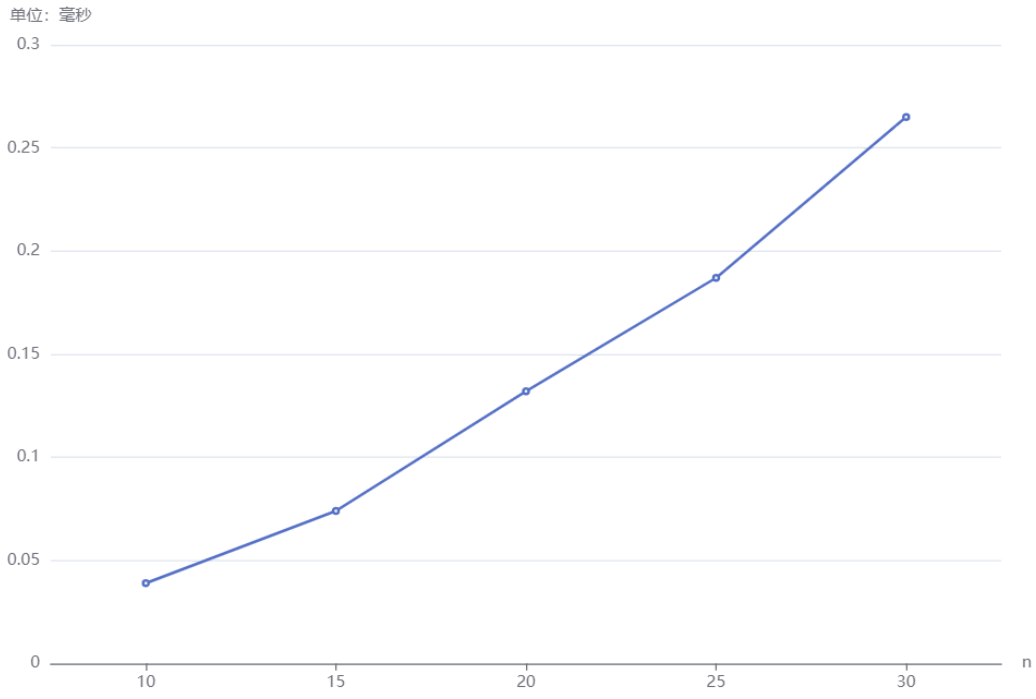
打印最长公共子序列如下。

```
 5: CAABA
 8: BADBCCCD
12: BACAAADCABAA
14: DCBABDDBDCCBDD
16: ADDBBCDBBCDDDCBD
```

`time.txt` 如下。

```
0.039
0.074
0.132
0.187
0.265
```

画出对应曲线图如下。



复杂度根据理论推导应该是 $O(n^2)$ 级别，实际从图可以看出，应该是符合多项式级别的。