

实验目标

- 1、熟悉动态规划算法实现的基本方法和步骤；
- 2、学会动态规划算法的实现方法和分析方法。

实验题目

问题 1: 最大公共子序列问题, 测试数据

$X=\{A B C B D A B\}$ $Y=\{B D C A B A\}$

$X=\{\text{zhejiang university of technology}\}$ $Y=\{\text{zhejiang university college}\}$

采用不记录 b 方案实现。

问题 2: 最大字段和问题, 比较三重循环, 分治法和动态规划算法的效果, 测试数据:

1) $(-2, 11, -4, 13, -5, -2)$

2) 过去大约三百年间, 太阳黑子数的时间数据如该链接所示, 请问, 历史上何时太阳黑子迎来了最大爆发?

第一个实验 (最长公共子序列问题)

算法问题:

最大公共子序列问题, 测试数据

$X=\{A B C B D A B\}$ $Y=\{B D C A B A\}$

$X=\{\text{zhejiang university of technology}\}$ $Y=\{\text{zhejiang university college}\}$

采用不记录 b 方案实现。

算法原理

初步分析:

最长公共子序列的递归方程为 $\max(a, b) = \begin{cases} \max(a-1, b-1) + 1 & s1[a] = s2[b] \\ \max(\max(a-1, b), \max(a, b-1)) & \text{others} \end{cases}$

因此可以使用递归和动态规划的方法来实现。循环的顺序为外层 $i = 0 \rightarrow s1.length$, 内层 $j = 0 \rightarrow s2.length$ 。

查找的字符串可以基于递归方程来递推。从二维数组的右下角开始, 如果次数比它的左上角大 1, 则该字符串属于公共子序列。否则, 向左侧或者上侧较大的位置继续查找。直到找到边界位置。

时间复杂度:

进行循环的次数为 $O(s1.length * s2.length)$, 循环内部的时间复杂度为 $O(1)$, 因此算法的复杂度约为 $O(m * n)$ 。

伪代码:

获取最大公共子序列的表格

```
def get_map(a:str, b:str):array<int,int>
    arr[a.length][b.length]:array<int,int>
    for i in 0 -> a.length:
        for j in 0 -> b.length:
            if a[i] == b[j]:
                arr[i][j] = arr[i-1][j-1] + 1
            else:
                arr[i][j] = max(arr[i-1][j], arr[i][j-1])
    return arr
```

反向寻找字符串

```
def search(table:array<int,int>, a:str, b:str):str
    str result
    i = a.length - 1
    j = b.length - 1
    while i > 0 and j > 0: # 没有到边界继续查找
        if table[i][j] = table[i-1][j-1] + 1:
            result += a[i] # 将此字符串加入到公共字符串
        else:
            if table[i-1][j] > table[i][j-1]:
                i -= 1
            else:
                j -= 1
    if table[i][j] == 1: # 边界环境
        result += a[i]
    return result.reverse() # 倒置字符串
```

结果:

```
C:\Users\t1542\Program\C++\an_dy\cmake-build-debug\an_dy.exe
please input string a:
ABCBDAB
please input string b:
BDCABA
arr:
  0  0  0  1  1  1
  1  1  1  1  2  2
  1  1  2  2  2  2
  1  1  2  2  3  3
  1  2  2  2  3  3
  1  2  2  3  3  4
  1  2  2  3  4  4
str:
BCAB
Process finished with exit code 0
```

图 3-1 最大公共子序列实验结果（1）

```
C:\Users\t1542\Program\C++\an_dy\cmake-build-debug\an_dy.exe
please input string a:
zhejiang university of technology
please input string b:
zhejiang university college
str:
zhejiang university oolg
Process finished with exit code 0
```

图 3-2 最大公共子序列实验结果 2（1）

图 3-1 到 3-2 为最长公共子序列测试样例的实验结果，实验结果正确，符合预期。

分析:

最长公共子序列问题是一个经典的利用动态规划的算法。动态规划相对于传统的递归方法来说，计算步骤会更加少。（一方面是递归需要堆栈来存储函数状态，另一方面是递归可能造成大量的重复运算），因此一般来说动态规划比递归有着更好的效果。在最长公共子序列中，使用递归方法会存在重复计算的问题，因此动态规划可以大大降低时间复杂度。另外一种常用来代替动态规划的方法是备忘录方法。备忘录相对于动态规划算法的优点是如果中间的计算过程是跳跃的（比如说(5,4)不会计算到），那么备忘录方法就会剩下这一步的时间消耗，另外一个优点是设计简单（结构仍使用传统的递归，使用空间换时间的方法来解决）。但是备忘录方法额外的判断步骤则会增加算法的时间消耗。在实际过程中，这两种方法常常是可以相互替换的。

第二个实验（最大字段和问题）

算法问题：

问题 2：最大字段和问题，比较三重循环，分治法和动态规划算法的效果，测试数据：

1) (-2, 11, -4, 13, -5, -2)

2) 过去大约三百年间，太阳黑子数的时间数据如该链接所示，请问，历史上何时太阳黑子迎来了最大爆发？

算法原理：

算法原理：

最大字段和的二重循环法： $i = 0 \rightarrow n, j = i \rightarrow n$ ，可以枚举所有的情形，时间复杂度为 $O(n^2)$ 。
最大字段和的分治法，最大字段和只能存在以下 3 种情况中的一个①最大字段包含中间的元素②最大字段在左边的字段中③最大字段在右边的字段中。这样就把一个大问题分解成了三个小问题。算法时间复杂度的递归式为 $F(n) = 2F\left(\frac{n}{2}\right) + O(n)$ ，整个算法的时间复杂度为 $O(n \log n)$ 。

最大字段和的动态规划方法，设 $0..n-1$ 的最大字段和为 s ，记当前的数字为 k ，则 $0..n$ 的最大字段和为 $\begin{cases} s+k, k > 0 \\ \max(s, k), k \leq 0 \end{cases}$ ，基于此我们可以从 $0 \rightarrow n$ 使用动态规划方法。时间复杂度为 $O(n)$ 。

伪代码：

```
def max_number_loop(numbers: list<int>):
    max: int = numbers[0]
    max_start: int = 0
    max_end: int = 1

    for i = 0 -> len(numbers):
        temp = 0
        for j = i -> len(numbers):
            temp += numbers[j]
            if temp > max:
                max = temp
                max_start = i
                max_end = j + 1
    return max, max_start, max_end

def max_number_division(numbers: list<int>, start = 0, end =
len(numbers)):
    middle: int = numbers[0]
```

```

max_start = middle
max_end = middle + 1
max = numbers[middle]

temp = max
for i: middle - 1 -> 0
    temp = temp + numbers[i]
    if temp > max:
        max = temp
        max_start = i
temp = max
for j = middle + 1 -> n
    temp = temp + numbers[j]
    if temp > max:
        max = temp
        max_end = j + 1

if start + 1 <= middle:
    temp_max, temp_start, temp_end = max_number_division(numbers,
start, middle)
    if temp_max > max:
        max = temp_max
        max_start = temp_start
        max_end = temp_end
if middle + 1 <= end:
    temp_max, temp_start, temp_end = max_number_division(numbers,
start, middle)
    if temp_max > max:
        max = temp_max
        max_start = temp_start
        max_end = temp_end
return max, max_start, max_end

def max_number_dy(numbers: list<int>):
    max = numbers[0]
    max_start = 0
    max_end = 1

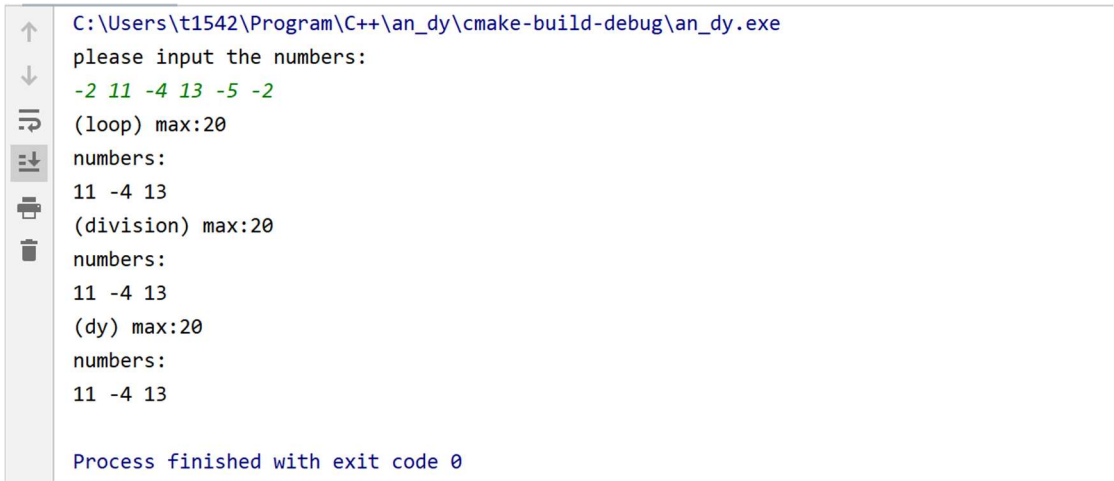
    temp = max
    for i = 1 -> numbers.length:
        if numbers[i] < 0

```

```
        if numbers[i] > max:
            max = numbers[i]
            max_start = i
            max_end = i + 1
        temp = max
    else:
        temp += numbers[i]
        if temp > max:
            max_end = i + 1

    return max, max_start, max_end
```

结果：



```
C:\Users\t1542\Program\C++\an_dy\cmake-build-debug\an_dy.exe
please input the numbers:
-2 11 -4 13 -5 -2
(loop) max:20
numbers:
11 -4 13
(division) max:20
numbers:
11 -4 13
(dy) max:20
numbers:
11 -4 13

Process finished with exit code 0
```

图 3-3 最大字段和问题实验结果。

9100:	137	0	0
9200:	147	0	0
9300:	132	0	0
9400:	153	0	0
9500:	146	0	0
9600:	155	1	1
9700:	145	1	1
9800:	148	1	1
9900:	153	1	1

图 3-4 最大字段和问题算法比较（二重循环 分支，动态规划）。

分析：

最大字段和的三种算法中，动态规划的时间复杂度最低，主要是动态规划是自底向上的算法，

其构建过程是线性的，而分治法则是树状的。因此相对来说动态规划的算法更优。