【数据结构】Day2

Class	Advanced Data Structures
Date	@December 2, 2021
Material	
Series Number	
Summary	

【Ch2】运行时间计算

2.4 运行时间计算

为了简化分析运行时间，我们采用如下约定：不存在特定的时间单位。因此，我们抛弃低阶项，而从计算大O运行时间。

由于大O是一个上界，因此我们必须仔细，绝不要低估程序的运行时间。实际上，分析的结果为程序在一定的时间范围内能够终止运行提供了保障。程序可能提前结束，但绝不可能延后

2.4.1 一个简单的例子

```
        int
        Sum( int N )
        {
            int i, PartialSum;

/* 1*/      PartialSum = 0;
/* 2*/      for( i = 1; i <= N; i++ )
/* 3*/          PartialSum += i * i * i;
/* 4*/      return PartialSum;
        }
```

分析：

1. 声明不计时间。第1行和第4行各占一个时间单元。

2. 第3行每执行一次占用4个时间单元（两次乘法，一次加法和一次赋值），而执行N次共占用4N个时间单元
3. 第二行在初始化i、测试i≤N和对i的自增运算中隐含着开销。所有这些的总开销为：
    a. 初始化1个时间单元
    b. 测试N+1个时间单元
    c. 自增运算N个时间单元，共2N+2
4. 忽略调用函数和返回值的开销，得到总量是6N+4。

因此，我们说该函数是O(N)

2.4.2 一般法则

1. 法则一：For循环

    一次for循环的运行时间至多是该for循环内语句（包括测试）的运行时间乘以迭代的次数

2. 法则二：嵌套的for循环

    从里向外分析这些循环。在一组嵌套循环内部的一条语句总的运行时间为该语句的运行时间乘以该组所有的for循环的大小的乘积

3. 法则三：顺序语句

    将各个语句的运行时间求和即可

4. 法则四：If/Else语句

    一个if/else语句的运行时间从不超过判断再加上statement运行时间长者总的运行时间

分析的基本策略是从内部（或最深层部分）向外展开的。如果有函数调用，那么这些调用要首先分析。如果有递归过程，那么存在几种选择。

若递归实际上只是for循环，则分析通常很简单，

如下面的函数实际上就是一个简单的循环，因此运行时间为O(N)

```
long int Factorial(int N) {
  if (N <= 1)
    return 1;
```

```
    else
      return N * Factorial(N - 1);
}
```

2.4.3 最大子序列和问题的解

```
            static int
            MaxSubSum( const int A[ ], int Left, int Right )
            {
                int MaxLeftSum, MaxRightSum;
                int MaxLeftBorderSum, MaxRightBorderSum;
                int LeftBorderSum, RightBorderSum;
                int Center, i;

/* 1*/          if( Left == Right )  /* Base Case */
/* 2*/              if( A[ Left ] > 0 )
/* 3*/                  return A[ Left ];
                    else
/* 4*/                  return 0;

/* 5*/          Center = ( Left + Right ) / 2;
/* 6*/          MaxLeftSum = MaxSubSum( A, Left, Center );
/* 7*/          MaxRightSum = MaxSubSum( A, Center + 1, Right );

/* 8*/          MaxLeftBorderSum = 0; LeftBorderSum = 0
/* 9*/          for( i = Center; i >= Left; i-- )
                {
/*10*/              LeftBorderSum += A[ i ];
/*11*/              if( LeftBorderSum > MaxLeftBorderSum )
/*12*/                  MaxLeftBorderSum = LeftBorderSum;
                }

/*13*/          MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/          for( i = Center + 1; i <= Right; i++ )
                {
/*15*/              RightBorderSum += A[ i ];
/*16*/              if( RightBorderSum > MaxRightBorderSum )
/*17*/                  MaxRightBorderSum = RightBorderSum;
                }

/*18*/          return Max3( MaxLeftSum, MaxRightSum,
/*19*/                  MaxLeftBorderSum + MaxRightBorderSum );
            }

            int
            MaxSubsequenceSum( const int A[ ], int N )
            {
                return MaxSubSum( A, 0, N - 1 );
            }
```

令T(N)是求解大小为N的最大子序列和问题所花费的时间。

- 如果N=1，则算法3执行程序第1行到第4行花费某个时间常量。于是，T(1)=1。

- 否则，程序必须运行两侧递归调用，即在第9行到17行之间的两个for循环，还需要某个小的薄记量，如在第5行和第8行。
- 这两个for循环总共接触从A_0到A_{N-1}的每一个元素，而在循环内部的工作量是常量，因此，在第9到17行花费的时间为O(N)。
- 其余就是第6、7行的工作，这两行求解大小为N/2的子序列问题（假设N是偶数）。因此，这两行每行花费T(N/2)个时间单元，共花费2T(N/2)个时间单元。算法总花费的是减肥2T(N/2)+O(N)。我们得到方程组

$$T(1) = 1$$
$$T(N) = 2T(N/2) + O(N)$$

为了简化，我们用N代替O(N)；*由于T(N)最终还是要由大O来表示，所以这么做不会影响最终答案。*

如果T(N)=2T(N/2)+N，且T(1)=1，那么T(2)=4=2*2，T(4)=12=4*3，T(8)=32=8*4。其形式是显然的并且可以得到，即

若$N = 2^k$，则T(N)=N*(k-1)=NlogN+N=O(NlogN)