Data Structure and Algorithms

B.W. Sc44964

[Introduction]

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every' enterprise application uses various types of data structures in one or the other way.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Here in the club,we need to learn by ourself .Though I have found it's hard,difficult,tough.....But nothing of great significance is easy to get .All we need to do is to put all our will against here ,and it's amazing what can be achieved .

So let's start the hard way .

Please remember,

所谓无底深渊，下去也是前程万里。

B.W.

2017.4.1

[Reference]:*Data Structures and Algorithm Analysis in C*

[Website Resources]:

https://www.tutorialspoint.com/data_structures_algorithms/data_structure_overview.htm

https://www.hackerrank.com/dashboard

基础知识

第1章 引言

对于连通性问题(*Dynamic Connectivity*)，书中首先提出的是快速查找算法(*quick-find algorithm*)。大意是把连通的id值赋值为同一个，用id值的等价来表示它的连通。

过程大致如下：

1. 初始化数组
2. 获取输入
    - 如果输入的两个值的id相等，则跳出（已连通）
    - 否则，就把所有id值为id[p]的元素变为id[q]，相同的id表示它们在同一个集合（标志连通属性）并在下一次循环中输出上一次的结果（未连通则输出p-q，连通则不输出）

快速查找算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, p, q, t, id[N];
  // 初始化数组
  for(i = 0; i < N; i++)
    id[i] = i;
  while(scanf(" %d %d\n", &p, &q) == 2){
    // 如果输入的两个值得id相等，则跳出（已连通）
    if(id[p] == id[q])
    {
          printf(" %d %d之前已连通\n", p, q);
          continue;
    }

    // 检查所有id值为id[p]的元素，并将其值置为id[q]
    for(t = id[p], i = 0; i < N; i++)
      if(id[i] == t)
        id[i] = id[q];
    printf(" %d %d之前未连通\n", p, q);
  }
  return 0;
}
```

优点：查找快速

缺点：慢速合并，对于每个合并操作，for循环迭代N次，N个对象，至少执行MN次，在大量的对象和输入时不可取。

但也可按照根(*root*)查找，根一定是指向自身的，每一次输入，都是“枝”的生长、合并，当查找的时候，找的是位于“树根”（枝的尽头）处的值，只要“树根”处的值位于一个集合中，则它们在一个“树”上，那我们就可以说，它们是连通的，这就是快速合并算法。

那么此时的数组就比第一种更加抽象了，数组的id值其实是该节点的“上一个值”，即父节点，通过数组的不停迭代上溯查找，来找到“树根”。

则有代码：

快速合并算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N];
  // 初始化数组,每一个元素都是自己的"根"
  for(i = 0; i < N; i++)
    id[i] = i;
  while(scanf(" %d %d\n", &p, &q) == 2){
    // 两个for循环都是为了找到各自的"根"。如果i和id值相等，那么
    // 自然有i == id[i]，即找到了"根"，退出循环；如果i和id值不
    // 等，则此值一定会通过"上溯"的方法找到"根"的位置，即，把其
id
    // 值作为新的查找值，直至找出其"根"。
    for(i = p; i != id[i]; i = id[i])
      ;
    for(j = q; j != id[j]; j = id[j])
      ;
    // 如果"根"相同，则说明它们在同一个集合中；否则，就将其归入
    // 另外一个集合中。
    if(i == j){
      printf(" %d %d之前已连通\n", p, q);
      continue;
    }
    id[i] = j;
    printf(" %d %d之前未连通\n", p, q);
  }
}
```

但是考察一下第二个算法，当进行树的合并($id[i] = j$)的时候，如果把较大的树合并到较小的树上，就会造成查找的耗时。而我们可以通过增加一个判断，使较小的树总是连接到较大的树上，从而达到优化算法的目的。

这样的话，就需要一个size数组来记录每一个树的大小，在连接时进行比较，由此，将较小的树连接到较大的树上。

则有代码：

加权快速合并算法

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N], sz[N];
  for(i = 0; i < N; i++){
    id[i] = i;
    sz[i] = 1; // 每一个独立单元都是一个size为1的树
  }
  while(scanf(" %d %d\n", &p, &q) == 2){
    for(i = p; i != id[i]; i = id[i])
      ;
    for(j = q; j != id[j]; j = id[j])
      ;
    if(i == j){
      printf(" %d %d之前已连通\n", p, q);
      continue;
    }
    // 加权合并
    if(sz[i] < sz[j]){
      id[i] = j;
      sz[j] += sz[i];
    }else{
      id[j] = i;
      sz[i] += sz[j];
    }
    printf(" %d %d之前未连通\n", p, q);
  }
}
```

优点：树中每个节点到根节点的距离变小，因而查找效率更高，非常高。

但对于这个算法而言，我们可以把一个树进行路径压缩，使树“平扁化”。

等分路径压缩

```c
#include <stdio.h>
#define N 10000
int main(){
  int i, j, p, q, t, id[N], sz[N];
  for(i = 0; i < N; i++){
    id[i] = i;
    sz[i] = 1; // 每一个独立单元都是一个size为1的树
  }
  while(scanf(" %d %d\n", &p, &q) == 2){
    for(i = p; i != id[i]; i = id[i])
      id[i] = id[id[i]];
    for(j = q; j != id[j]; j = id[j])
      id[j] = id[id[j]];
    if(i == j){
      printf(" %d %d之前已连通\n", p, q);
      continue;
    }
    // 加权合并
    if(sz[i] < sz[j]){
      id[i] = j;
      sz[j] += sz[i];
    }else{
      id[j] = i;
      sz[i] += sz[j];
    }
    printf(" %d %d之前未连通\n", p, q);
  }
}
```

第一章通过对连通性问题求解的逐步求精给我们展示了一个算法从雏形到完善的整个过程，这一章讲算法的目的并不在于解决这一问题，而在于展现出设计算法的基础步骤：

1. 确定完整、明确的问题描述，包括确定问题固有的基本抽象操作。
2. 仔细设计一个简单算法的简明实现。
3. 通过逐步求精的过程开发改进后的实现，经过实验分析、数学分析或两者共同验证改进思想的效率。
4. 找出数据结构或者算法操作的高级抽象表示，能够使改进版本的设计性能高效。
5. 可能时尽量保证最坏情况下的性能，但实际数据可用时接受好的性能。

而对于第一章的学习，我们可以知道数组的抽象形式。一个整型数组不仅仅只是内存中占有等长字节的连续的一组值，它可通过位-值得关系来进行排序（桶排序），也可以抽象成集合、树（如上），甚至还可以对树进行加权、压缩！

And.....

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

所以我们下面再开始分析算法吧。。。。（前言还有一些超级变态的课后题，一起来做做吧。。。）

第2章 算法分析的原理

PRE READINGS

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

```
step 1 - START
step 2 - declare three integers a, b & c
step 3 - define values of a & b
step 4 - add values of a & b
step 5 - store output of step 4 to c
step 6 - print c
step 7 - STOP
```

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

```
step 1 - START ADD
step 2 - get values of a & b
step 3 - c ← a + b
step 4 - display c
step 5 - STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- **A Priori Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A Posterior Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

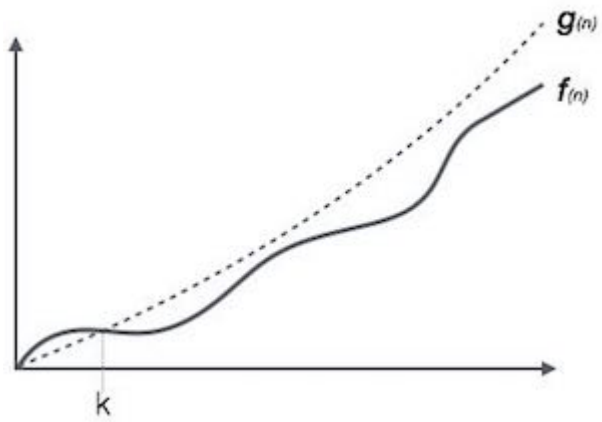
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- $\mathcal{O}$ Notation
- Ω Notation

- Θ Notation

Big Oh Notation, $\mathcal{O}$



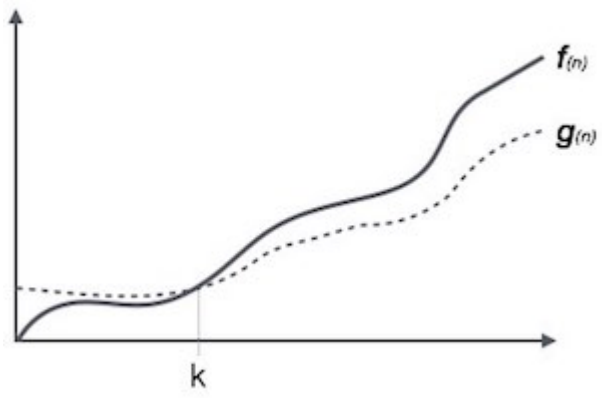
The notation $\mathcal{O}$ (n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

For example, for a function **f(n)**

```
O(f(n)) = { g(n) : there exists c > 0 and n0 such that
f(n) ≤ c.g(n) for all n > n0. }
```

$$\mathcal{O}(f(n)) = \{g(n) : \text{ there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) < cg(n) \text{ for all } n > n_0\}$$

Omega Notation, Ω

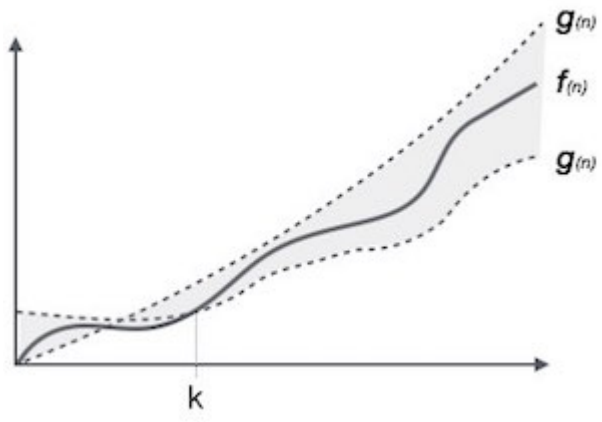


The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

For example, for a function **f(n)**

$$\Omega(f(n)) \geq \{g(n) : \text{ there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq cf(n) \text{ for all } n > n_0\}$$

Theta Notation, Θ



The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows,

$$\Theta(f(n)) = \{g(n) \text{ if and only if } g(n) = \mathcal{O}(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0.\}$$

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

CONSTANT	− $\mathcal{O}(1)$
logarithmic	− $\mathcal{O}(\log n)$
linear	− $\mathcal{O}(n)$
n log n	− $\mathcal{O}(n \log n)$
quadratic	− $\mathcal{O}(n^2)$
cubic	− $\mathcal{O}(n^3)$
polynomial	− $n^{\mathcal{O}(1)}$
exponential	− $2^{\mathcal{O}(n)}$

鉴于这部分内容基础部分大部分可以自行理解，觉得https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm的Data Structures - Asymptotic Analysis部分梳理的很好,整理在这里，阅读一遍即可。

$\mathcal{O}$(*O-notation*)：【渐进上界】如果存在常数c_0和N_0，对于所有$N > N_0$，有$g(N) < c_0 f(N)$,则称函数$g(N)$是$\mathcal{O}(f(N))$的。

- 限制忽略数学公式中的低阶项时产生的误差。
- 限制由于忽略对程序的总运行时间贡献较小的某些部分时产生的错误。
- 允许我们按照算法总运行时间的上界对算法进行分类。

基本递归方程

- 程序的循环通过输入每次减少一项

$$C_N = C_{N-1} + N, N \geq 2, C_1 = 1$$
$$C_N = \frac{N(N+1)}{2}$$

- 程序每次使输入减半

$$C_N = C_{N/2} + 1, N \geq 2, C_1 = 1$$
$$C_{2^n} = n + 1$$

- 程序每次使输入减半，但需检查输入的每一项

$$C_N = C_{N/2} + N, N \geq 2, C_1 = 0$$
$$C_N \approx 2N$$

- 程序把输入分成两半，但在划分之前、划分之中以及划分之后需要线性遍历输入

$$C_N = 2C_{N/2} + N, N \geq 2, C_1 = 0$$
$$C_{2^n} = n2^n$$

Extend Readings

Time complexity

Big O nonation

A beginner's guide to Big O notation

LAST UPDATE ON 4.2.2017