UM–SJTU Joint Institute

VE477 Intro to Algorithms

Homework 5

Wang, Tianze
515370910202

Question 1 Partition Problem

(1) Definition

It is the task of deciding whether a given multiset S of positive integers can be partitioned into several subsets S_1, S_2,..., S_k such that the maximum of the sum of each set is as small as possible.

(2) Simple Solution

No, it is not a good decision. For example, suppose we have a set like

$$\{1,\ 2,\ 97,\ 99\}$$

And we wish to partition it into two sets. Using this method will lead to a max set of 196, however if we partition it into

$$\{1, 2, 97\}, \{99\}$$

This will lead to a better solution, which is 100.

(3) Recursive Algorithm

The recursive algorithm is defined as follows. Starting from the last partition, we place a divider, which will yields to two sets for best position: **the last partition** and **the first $k-2$ partitions**. To minimize the total cost, we should try to make the rest $k-2$ partitions as equal as possible. So on and so forth.

To make the long story short, we calculate all the possible solutions, and we find the minimum result.

(4) Complexity

The calculating process will cost $k \cdot n$ total combinations.

To find the minimum path, it will cost n^2, since we need to check each entry. Thus the total complexity is

$$\mathcal{O}(kn^3)$$

(5) Stored Quantities

We should store all the sum from the first element to every element $p[k] = \sum_{i=1}^{k} s[i]$, which will save time when we calculate the block size. E.g.

$$s_3 + \cdots + s_5 = \sum_{i=1}^{5} s_i - \sum_{i=1}^{3} s_i = p[5] - p[3]$$

(6) Pseudocode

Here we store everything in the DP matrix.

(7) Correctness

The algorithm will always yield to a correct result in that it will calculate the minimized result after calculating all the basis for the calculation. Also, it follows the idea that we formed in the very beginning.

Algorithm 1: Refer to CSE417 in Washington

```
   Input  : multiset S, integer k
   Output : linear partition with smallest max size
1  Function partition(S,k):
2      M[1,1] = s_1 ;
       /* First consider two base cases                    */
3      for i=1 to n do
4          M[i,1] = M[i-1,1] + s_i ;
5      end for
6      for j=1 to k do
7          M[1,j] = s_1;
8      end for
9      for i=2 to n do
10         for j=2 to k do
11             M[i,j] = ∞ ;
12             for pos = 1 to i-1 do
13                 s = max{M[pos,j-1], p[i] - p[pos]};
14                 if M[i,j] > s then
15                     M[i,j] = s;
16                 end if
17             end for
18         end for
19     end for
20 end
```

(8) Complexity

The complexity is decided by line 9 to 21 in Algorithm 1, that we need at most $k \cdot n$ for each iteration of i, so the total complexity is then

$$\mathcal{O}(kn^2)$$

(9) Path

Each time when we update M we store the position of the partition simultaneously. And finally by reading the position, we can find the partition directly.

Question 2 Critical Thinking

Here we use the idea as: The binary representative of a decimal number.

0-4 to 0-7

7 can be represented as 111 in binary. Thus 0 to 7 is 000_b to 111_b.

Algorithm 2: 0-4 to generate 0-7

```
Input  : 7
Output : a random integer between 0-7
Function generator(3):
    i ← 3 ;
    while i ≠ 0 do
        j ← get an output from black box ;
        /* << means the operation of shift left */
        if j = 0 or 1 then
            b = 0 ;
            a = (a << 1) + b ;
            i − − ;
        else if j = 2 or 3 then
            b = 1 ;
            a = (a << 1) + b ;
            i − − ;
        else
            Continue;
        end if
    end while
    return a
end
```

(1) 0-4 to common case

For the common case, we apply the same idea, but we need a judging condition to represent whether our result falls into the range of acceptance.

Algorithm 3: 0-4 to generate 0-n

```
Input  : n
Output : a random integer between 0-n
Function generator(n):
    A ← a binary number ;
    while a>n do
        i ← ⌈log_2 n⌉ ;
        while i ≠ 0 do
            j ← get an output from black box ;
            /* << means the operation of shift left */
            if j = 0 or 1 then
                b = 0 ;
                a = (a << 1) + b ;
                i − − ;
            else if j = 2 or 3 then
                b = 1 ;
                a = (a << 1) + b ;
                i − − ;
            else
                Continue;
            end if
        end while
    end while
end
```

Question 3 Bellman-Ford Algorithm

To detect the negative cycle, we apply the relax operation on all edges in a graph for n times, if it can be uploaded on the $n-th$ iteration, it means that there exists a negative cycle.

Algorithm 4: Relax

Input : an edge e starting from u, ending at v
Output : Relaxed info of u and v

Function relax(E)**:**
  if *v.d (distance from source vertex to v) $>$ u.d (distance from source vertex to u) + E.weight* **then**
    $v.d = u.d + w(u, v)$;
    $v.p = u$;
  end if
end

Algorithm 5: Detect negative cycle

Input : Graph G with n nodes
Output : Whether there is a negative cycle

Function detect(G)**:**
  for *i = 1 to n-1* **do**
    for *All edges in graph* **do**
      relax();
    end for
  end for
  for *All edges in graph* **do**
    relax();
    if *any edge is relaxed* **then**
      return *Exists negative cycle*;
    end if
  end for
  return *Does not exists negative cycle*;
end

Question 4 Augmenting Path

Question 5 Wifi Network

1. First check whether the number of clients exceeds the maximum capacity of all connections. If this fails, then return no.
2. Then decide each cell phone can connect to which hot spots.
3. Then for each hotspot, we decide whether it can accommodate all the possible connections. Here we apply the greedy method to calculate.

Since we have tried out every case, we will definitely get the correct answer on whether hostspots can hold the clients.

```
Input   : k hotspots with each l, r; n clients
Output  : whether user can all connect to Internet
Function wifi_connection(k hotspots, n clients):
    if Client number > sum of maximum connection number for all hotspots then
        return False
    end if
    for All n clients do
        for All k hot spots do
            Decide whether the client can connect to the hotspot. if Can connect then
                Append client to hotspot.available_user;
            end if
        end for
    end for
    /* Greedy Method.                                                    */
    while Not all cases exhausted do
        for all hotspots do
            for The rest clients that not connected and in hotspot.available_user list do
                Connect as many clients as possible. The client must be different from last iteration.
            end for
        end for
        if All clients are connected then
            return True
        end if
    end while
    return False
end
```

The method yields to a maximum time complexity as:

$$\mathcal{O}(n^{kl})$$

which is a polynomial time.