Belgian Olympiad in Informatics

Easter Training 17 April 2015

String algorithms, Segment Tree and Shortest Path algorithms (Revision)

Trainer:
Floris Kint
Notes:
Floris Kint

April 16, 2015

Contents

1 String algorithms

1.1 Longest common substring

Given two strings a and b, determine the longest string s such that s is a substring of a and s is a substring of b.

We can use Dynamic Programming to solve this problem. Suppose a 2-dimensional array *cache* that contains k at position (i, j) iff k is the largest integer number such that $a[i-x] = b[j-x], \forall x < k$. To compute the value at a certain index in *cache*, we just need to check whether the a and b have the same value at position i and j respectively. If this is the case, we add one to the value at $cache[i-1][j-1]$ and store it at $cache[i][j]$. If this is not the case, we store 0. If we loop over the strings as in Listing 1, we only need the values at $i-1$, so we only need $O(N)$ instead of $O(N \cdot M)$ space. The result of the algorithm is the heighest value that has ever been stored in *cache*.

Listing 1: Longest common substring

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 using namespace std;
5
```

```
6  int lcs(string a, string b){
7    int best = 0;
8    vector<int> cache(a.size()+1, 0);
9    for(int i = 0; i < b.size(); ++i){
10     for(int j = a.size()-1; j >= 0; --j){
11       if(a[i] == b[j]){
12         cache[j+1] = cache[j] + 1;
13         best = max(cache[j+1], best);
14       }else{
15         cache[j+1] = 0;
16       }
17     }
18   }
19   return best;
20 }
21
22
23 int main(){
24   string a, b;
25   cin >> a >> b;
26   cout << lcs(a, b) << endl;
27   return 0;
28 }
```

1.2 Trie

A Trie is a useful datastructure to store a set of strings. Instead of storing each string on its own (e.g. in a map), it stores the path to reach a string. This can be useful when searching for all words in a dictionary starting with a certain prefix. The following code example illustrates this by calculating the length of the longest prefix of a new string that also appears in the words that had already been added.

Listing 2: Trie

```
1  #include <map>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  struct node{
8    map<char, node*> children;
9  };
10
11 void add_to_prefix_tree(node *n, string s){
12   node* current_node = n;
13   int current_index = 0;
14   while(current_index < s.size() && current_node->
         children.find(s[current_index]) != current_node->
```

```
        children.end()){
15      current_node = current_node->children[s[current_index
            ]];
16      current_index ++;
17    }
18    while(current_index < s.size()){
19      current_node->children[s[current_index]] = new node()
            ;
20      current_node = current_node->children[s[current_index
            ]];
21      current_index ++;
22    }
23  }
24  int find_longest_prefix(node *n, string s){
25    node* current_node = n;
26    int current_index = 0;
27    while(current_index < s.size() && current_node->
          children.find(s[current_index]) != current_node->
          children.end()){
28      current_node = current_node->children[s[current_index
            ]];
29      current_index++;
30    }
31    return current_index;
32  }
33
34  int main(){
35    std::ios::sync_with_stdio(false);
36    int N;
37    cin >> N;
38    node *root = new node();
39    for(int i = 0; i < N; ++i){
40      string s;
41      cin >> s;
42      cout << find_longest_prefix(root, s) << endl;
43      add_to_prefix_tree(root, s);
44    }
45    return 0;
46  }
```

1.3 String matching

A common problem is to find if a pattern appears in another text. Standard libraries provide this functionality for strings, but it is still worth understanding how this can be achieved. These techniques can be applied to other problems (see exercises).

1.3.1 Naive approach

A first naive approach is illustrated in the following code. Note that this algorithm has a complexity of $O(N \cdot M)$.

Listing 3: Naive string matching

```
#include <string>
#include <iostream>

using namespace std;

int matching(string text, string pattern){
  for(int i = 0; i <= text.size() - pattern.size(); ++i){
    bool found = true;
    for(int j = 0; j < pattern.size(); ++j){
      if(text[i+j]!=pattern[j]){
        found = false;
        break;
      }
    }
    if(found){
      return i;
    }
  }
  return -1;
}

int main(){
  string text, pattern;
  cin >> text >> pattern;
  cout << matching(text, pattern) << endl;
  return 0;
}
```

1.3.2 Knuth-Morris-Pratt algorithm

A better algorithm that can solve this problem in $O(N + M)$, is the Knuth-Morris-Pratt algorithm (often referred to as KMP).

The KMP algorithm consists of two steps.

First it analyzes the pattern that is being searched for. It constructs a table *back_table* of the same length as the pattern. Suppose $text[a : a+j] = pattern[0 : j]$, but $text[a + j] \neq pattern[j]$, then $back\_table[j]$ contains the last index in *pattern* that will match with *text* until the previous position $text[a + j - 1]$. That is, $back\_table[j]$ contains the largest number $x < j$ such that $text[i : i+x] = pattern[0 : x]$.

During the second step the algorithm uses that information to iterate over *text* mucht faster than the naive approach. It can be shown that the complexity of this algorithm is $O(N + M)$.

Listing 4: KMP algorithm

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

vector<int> preprocess(string pattern){
  vector<int> back_table(pattern.size()+1, 0);
  int j = -1;
  back_table[0] = -1;
  for(int i = 0; i < pattern.size(); ++i){
    while(j >= 0 && pattern[i] != pattern[j]){
      j = back_table[j];
    }
    ++j;
    back_table[i+1] = j;
  }
  return back_table;
}

int matching(string text, string pattern){
  vector<int> back_table = preprocess(pattern);
  int j = 0;
  for(int i = 0; i < text.size(); ++i){
    while(j >= 0 && text[i] != pattern[j]){
      j = back_table[j];
    }
    j++;
    if(j == pattern.size()){
      return i+1-j;
    }
  }
  return -1;
}

int main(){
  string a, b;
  cin >> a >> b;
  cout << matching(a, b) << endl;
  return 0;
}
```

1.4 Exercises

1. Trie: `https://www.facebook.com/hackercup/problems.php?pid=313229895540583&round=344496159068801`

2. Trie: `http://uva.onlinejudge.org/external/115/p11590.pdf`

3. KMP: `http://uva.onlinejudge.org/external/114/p11475.pdf`

4. KMP: `http://codeforces.com/contest/536/problem/B`

2 Segment Tree

A Segment Tree is a datastructure that can be used to speed up range queries. Range queries are queries like "What is the sum of all elements starting at index 10 up to index 15?".

2.1 Concise version

The book Competitive Programming 3 provides the following code for a Segment Tree that can be used to find the minimum value in a range.

Listing 5: Segment Tree (from Competitive Programming 3)

```
1  #include <vector>
2  #include <iostream>
3
4  using namespace std;
5  //Competitive Programming 3, p. 57
6  class SegmentTree{
7    private:
8      vector<int> st, A;
9      int n;
10     int left(int p){ return p << 1; }
11     int right(int p){ return (p << 1) + 1; }
12
13     void build(int p, int L, int R){
14       if(L == R)
15         st[p] = L;
16       else{
17         build(left(p), L, (L+R)/2);
18         build(right(p), (L+R)/2 + 1, R);
19         int p1 = st[left(p)], p2 = st[right(p)];
20         st[p] = (A[p1] <= A[p2])?p1:p2;
21       }
22     }
23
24     int rmq(int p, int L, int R, int i, int j){
25       if(i > R || j < L) return -1;
26       if(L >= i && R <= j) return st[p];
27
28       int p1 = rmq(left(p), L, (L+R)/2, i, j);
29       int p2 = rmq(right(p), (L+R)/2 + 1, R, i, j);
30
31       if(p1 == -1) return p2;
32       if(p2 == -1) return p1;
33
```

```
34        return (A[p1] <= A[p2])? p1 : p2;
35      }
36    public:
37      SegmentTree(const vector<int> &A){
38        this->A = A;
39        n = (int)A.size();
40        st.assign(4*n, 0);
41        build(1, 0, n-1);
42      }
43      int rmq(int i, int j){ return rmq(1, 0, n-1, i, j); }
44  };
45
46  int main(){
47    int N;
48    cin >> N;
49    int t;
50    vector<int> v;
51    for(int i = 0; i < N; ++i){
52      cin >> t;
53      v.push_back(t);
54    }
55    SegmentTree st(v);
56    int M;
57    cin >> M;
58    for(int i = 0; i < M; ++i){
59      int a, b;
60      cin >> a >> b;
61      cout << st.rmq(a, b) << endl;
62    }
63    return 0;
64  }
```

2.2 Intuitive approach

However, you might find it more convenient to build a segment tree from scratch as opposed to remembering every detail by heart. The version below is longer to code, but is more intuitive to the writer. It also includes methods to update elements in the array. That way you can answer queries, update elements, answer questions about the modified array.

Listing 6: More advanced Segment Tree

```
1  #include <cstdio>
2  #include <algorithm>
3  #include <utility>
4  #include <assert.h>
5  using namespace std;
6
7  #define MAX 999999
8
```

```
 9 struct node{
10   node* right;
11   node* left;
12   int value;
13   int index;
14   int right_bound;
15   int left_bound;
16   int minimum_index;
17   int minimum_value;
18   node(int index, int left_bound, int right_bound){
19     this->minimum_index = index;
20     this->minimum_value = MAX;
21     this->left_bound = left_bound;
22     this->right_bound = right_bound;
23     this->index = index;
24     this->left = NULL;
25     this->right = NULL;
26     if(left_bound < this->index){
27       this->left = new node((left_bound+this->index)/2,
             left_bound, this->index-1);
28     }
29     if(right_bound > this->index){
30       this->right = new node((right_bound+this->index+1)
             /2, this->index+1, right_bound);
31     }
32   }
33   bool overlaps(int start, int end){
34     if(start > this->right_bound)
35       return false;
36     if(end < this->left_bound)
37       return false;
38     return true;
39   }
40   pair<int, int> find_minimum_range(int start, int end){
41     if(!overlaps(start, end))
42       return make_pair(MAX, this->index);
43     if(start<=this->left_bound && end >= this->
          right_bound){
44       return make_pair(this->minimum_value, this->
             minimum_index);
45     }
46     pair<int, int> best = make_pair(MAX, this->index);
47     if(start <= this->index && end >= this->index){
48       best = make_pair(this->value, this->index);
49     }
50     if(this->left != NULL)
51       best = min(best, this->left->find_minimum_range(
             start, end));
52     if(this->right != NULL)
```

```
53        best = min(best, this->right->find_minimum_range(
              start, end));
54      return best;
55    }
56    void update_minimum_index(){
57      this->minimum_index = this->index;
58      this->minimum_value = this->value;
59      if(this->left && this->left->minimum_value < this->
          value){
60        this->minimum_value = this->left->minimum_value;
61        this->minimum_index = this->left->minimum_index;
62      }
63      if(this->right && this->right->minimum_value < this->
          value){
64        this->minimum_value = this->right->minimum_value;
65        this->minimum_index = this->right->minimum_index;
66      }
67    }
68    void set_value_at(int index, int v){
69      if(this->index == index){
70        this->value = v;
71        update_minimum_index();
72      }else if(index < this->index){
73        this->left->set_value_at(index, v);
74      }else{
75        this->right->set_value_at(index, v);
76      }
77    }
78  };
79
80  int main(){
81    int N;
82    scanf("%d", &N);
83    node *root = new node(N/2, 0, N);
84    for(int i = 0; i < N; ++i){
85      int v;
86      scanf("%d", &v);
87      root->set_value_at(i, v);
88    }
89    int M;
90    scanf("%d", &M);
91    for(int i = 0; i < M; ++i){
92      int left, right;
93      scanf("%d%d", &left, &right);
94      pair<int, int> result = root->find_minimum_range(left
          , right);
95      printf("index: %d, value: %d\n", result.second,
          result.first);
96    }
97  }
```

2.3 Complexity

The segment tree updates elements and answers queries in $O(\log(N)$.

2.4 Exercises

1. `http://uva.onlinejudge.org/external/112/11297.html`

3 Shortest path algorithms (revision)

3.1 Dijkstra

Dijkstra's algorithm has already been explained in previous trainings. Below is an example implementation.

Listing 7: Single-source shortest path using Dijkstra's algorithm

```
1  #include <cstdio>
2  #include <queue>
3  #include <list>
4  #include <vector>
5
6
7  using namespace std;
8
9  struct node{
10   int index;
11   int distance;
12   bool operator<(const node &n) const{
13     return this->distance > n.distance;
14   }
15   node(int index, int distance){
16     this->index = index;
17     this->distance = distance;
18   }
19 };
20
21 int dijkstra(vector<list<pair<int, int> > > &neighbours,
       int from, int to){
22   priority_queue<node> pq;
23   pq.push(node(from, 0));
24   vector<bool> visited(neighbours.size(), false);
25   while(!pq.empty()){
26     node n = pq.top();
27     pq.pop();
28     if(to == n.index)
29       return n.distance;
30     if(visited[n.index])
31       continue;
32     visited[n.index] = true;
```

```
33      for(list<pair<int, int> >::iterator it = neighbours[n
            .index].begin(); it != neighbours[n.index].end();
            ++it){
34        if(visited[it->first])
35          continue;
36        pq.push(node(it->first, it->second + n.distance));
37      }
38    }
39    return -1;
40  }
41
42  int main(){
43    int N;
44    scanf("%d", &N);
45    vector<list<pair<int, int> > > neighbours(N);
46    int M;
47    scanf("%d", &M);
48    for(int i = 0; i < M; ++i){
49      int a, b, d;
50      scanf("%d%d%d", &a, &b, &d);
51      neighbours[a].push_back(make_pair(b, d));
52    }
53    int start, end;
54    scanf("%d%d", &start, &end);
55    printf("%d\n", dijkstra(neighbours, start, end));
56    return 0;
57  }
```

3.2 Floyd-Warshall

Sometimes one needs to compute the path lengths for all pairs of nodes. Or sometimes the input constraints may allow a slower algorithm than Dijkstra's. In both cases, Floyd-Warshall comes to the rescue.

For every pair of nodes (i, j), the path checks all other nodes k to see if going from i to k and then from k to j is shorter than going from i to j via the shortest path so far. It then updates the best path length from i to j and uses this new path length to improve other paths.

The algorithm is very simple and it can easily be seen that the time complexity is $O(N^3)$.

Listing 8: All pairs shortest path using Floyd Warshall

```
1  #include <iostream>
2  #include <vector>
3  #include <iomanip>
4  using namespace std;
5
6  #define MAX 999999
7  void floyd_warshall(vector<vector<int> > &distances){
8    for(int k = 0; k < distances.size(); ++k){
```

```
 9       for(int i = 0; i < distances.size(); ++i){
10         for(int j = 0; j < distances.size(); ++j){
11           distances[i][j] = min(distances[i][j], distances[
                 i][k] + distances[k][i]);
12         }
13       }
14     }
15   }
16   int main(){
17     int N;
18     cin >> N;
19     vector<vector<int> > distances;
20     for(int i = 0; i < N; i++){
21       distances.push_back(vector<int>(N, MAX));
22     }
23     int M;
24     cin >> M;
25     for(int i = 0; i < M; ++i){
26       int a, b, c;
27       cin >> a >> b >> c;
28       distances[a][b] = c;
29     }
30     floyd_warshall(distances);
31     for(int i = 0; i < N; ++i){
32       for(int j = 0; j < M; ++j){
33         cout << setw(8) << distances[i][j];
34       }
35       cout << endl;
36     }
37     return 0;
38   }
```

3.3 Exercises

1. Dijkstra: `http://uva.onlinejudge.org/external/113/11338.html`
2. Floyd-Warshall: `http://uva.onlinejudge.org/external/8/821.html`

4 Quiz results

The feedback of the Quiz can be found in a separate document.