HW3

Created	@November 28, 2021 4:19 PM
Tags	

Problem 1

input 파일의 처리

config_LPS.txt 파일의 첫번째 줄을 읽어 실행해야하는 테스트 케이스를 test_num 으로 받은 뒤, `test_num`만큼 for문을 돌려 처리해야 하는 입력 문자열이 저장되어 있는 bin파일(`input_f`)과 결과를 출력해야하는 bin파일의 파일 이름(`output_f`)을 읽어들여 input_f에 해당하는 파일을 읽어 처리해야하는 문자열의 개수(`m`) 과 문자열(`X`)를 읽어 LPS 함수를 실행한다.

LPS

길이가 m인 문자열 X가 가지는 LPS의 길이(`LPS_length`)와 문자열(`LPS_string`)을 구하기 위한 함수로 dynamic programming을 이용해 LPS를 구하였다.

optimal substructure

x_i 부터 x_j 에 해당하는 X의 subsequence에서 구한 LPS의 길이를 $L[i][j]$ 라고 하면 optimal substructure는 아래와 같다.

$$L[i][j]_{i\leq j} = \begin{cases} L[i+1][j-1]+2 & \text{if } x_i == x_j \\ max(L[i+1][j], L[i][j-1]) & \text{if } x_i ! = x_j \end{cases}$$

x_i 와 x_j 가 같을 경우, 두 문자를 포함해 회문을 만들 수 있으므로 두 문자열을 제외한 x_{i+1} 에서 x_{j-1} 사이의 LPS 값에 2를 더하고

다를 경우, 두 문자는 회문에 포함되지 않으므로 x_i를 제외한 경우 또는 x_j를 제외한 경우 중 더 큰 값을 취하도록 하였다.

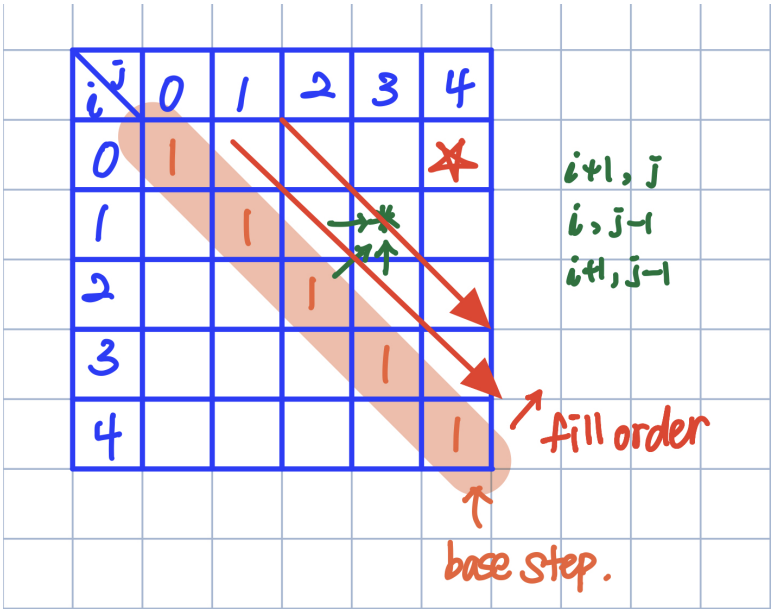
base step

i와 j가 같을 경우 길이가 1인 회문이므로 $L[i][i]$를 1로 초기화 해주는 과정을 진행하였다.

또한 j가 $i+1$과 일치하는데 두 문자가 같을 경우, 해당 문자열은 길이가 2인 회문이므로 $L[i][i+1]$의 경우 optimal structure대로 계산을 진행하지 않고 2를 할당해주었다.

filling the table

$m * m$의 크기를 가지는 테이블에서 $i \leq j$인 영역의 테이블만 채우고 우리가 궁극적으로 구해야하는 값은 $L[0][m-1]$이기 때문에 대각 순으로 테이블을 채워주었다.



m = 5 인 경우의 테이블

코드상으로는 해당 대각선의 `i`와 `j`의 차이인 `g`를 1부터 `m`까지 for문을 돌린 뒤, `i`를 0부터 `m-g`까지 돌면서 `j=i+g`에 대해 `L[i][j]`를 채워주었다.

문자열 구하기

길이 뿐만 아니라 실제 문자열을 구하기 위해서 추가로 P테이블을 만들어 두 문자가 같을 경우, 2를 (i+1, j)에서 왔을 경우 1, (i, j-1)에서 왔을 경우 -1을 저장해 테이블을 채웠다.

채워진 P테이블을 바탕으로 (0, m-1)에서부터 값을 확인하면서 2일 경우, LPS의 양 끝에 해당 문자를 넣은 뒤 i와 j를 각각 +1, -1 해주었고, 나머지의 경우 값에 따라 움직이며 i에 +1또는 j에 -1을 해 $i \leq j$를 만족할 때까지 while문을 돌았다.

Problem2

input 파일의 처리

commands_3_2.txt 의 `fscanf` 결과 값이 -1이 나오기 전까지 while문을 돌면서 해결해야하는 파일의 이름(`input_f`)를 읽은 뒤, 해당 파일의 각 줄을 읽으면서 왼쪽 카드의 개수와 카드 값을 각각 `int L`, `int* arr_l`에 저장하고 오른쪽 카드의 개수와 카드 값을 각각 `int R`, `int* arr_r`에 저장하였다.

마지막 두 줄에 들어오는 true or false를 판단해야 하는 input의 경우 2번의 iteration을 통해 섞인 카드 묶음의 개수와 카드 값 배열을 `int N`과 `int *arr_n`에 저장하였다.

dynamic programming

L과 R의 크기를 가지는 왼쪽 카드와 오른쪽 카드에 대해 각각 l_i까지와 r_j 까지의 카드를 고려했을 때 섞인 카드의 $i+j$번째까지가 유효한 순서인지를 판단하는 $O[i][j]$를 subproblem으로 정의하였다.

optimal substructure

$$O[i][j] = \begin{cases} True & \text{if } O[i-1][j] \text{ and } l_i == n_{i+j} \\ True & \text{if } O[i][j-1] \text{ and } r_j == n_{i+j} \\ False & \text{if the rest of the case} \end{cases}$$

섞인 카드 묶음의 $i+j$번째 숫자가 r_j와 같고 그 바로 전 단계인 $(i, j-1)$에서 true 값을 가질 경우 $O[i][j]$는 True 값을 가진다.

또한 섞인 카드 묶음의 $i+j$번째 숫자가 l_i와 같은 경우에도 바로 전 단계인 $(i-1, j)$가 true 값을 가진다면 $O[i][j]$는 True 값을 가진다.

나머지의 경우, False를 가진다.

base step

$O[0][0]$ 의 경우 아무것도 고려하지 않는 경우이고 (0, 1)이나 (1, 0)에서 고려할 앞 단계가 없으므로 해당 값을 True로 해 $l_1 == n_1$ 또는 $r_1 == n_1$ 일 경우 해당 값을 True로 세팅할 수 있도록 한다.

i=0인 경우와 j=0인 경우에 대해 미리 초기화를 해주는데 두 경우는 optimal substructure와 달리 고려해야할 이전 단계가 하나 적기 때문이다.

i=0인 경우는 j-1에 대해서만, j=0인 경우는 i-1인 경우만 고려해준다.

filling the table

L*R의 크기를 가지는 테이블에 대해 i부터 for문을 돌려 row 순서대로 테이블을 채워주었다.

Problem3

input 파일 처리

입력파일에 ASCII 코드를 huffman coding하는 것이므로 아래 구조체를 원소로 가지는 ascii_table[128] 배열을 선언해 입력 파일에 저장된 각 character의 빈도수(`freq`)를 저장하였다.

```
typedef struct _info {
  int freq;
  int code;
  unsigned int bits;
} INFO;
```

이때 전체 character 개수를 char_cnt에 저장해 각 character의 빈도 백분율을 구하였다.
(`(float)ascii_table[i].freq / char_cnt * 100.0`)

이렇게 작성한 ascii_table을 바탕으로 freq가 0이 아닌 원소에 대해 아래 구조체를 가지는 single node tree를 생성해 트리들을 linked list로 연결해주었다.

```
typedef struct _node {
  int key;
  INFO* t_ptr;
  struct _node* left;
  struct _node* right;
  struct _node* next;
} NODE;
```

tree의 root들은 `next` 멤버 변수를 통해 연결되어 있으며 tree의 depth는 `left`또는 `right`를 이용해 이동하였다.

`t_ptr`의 경우 `ascii_table` 포인터로 `key` 값은 해당 테이블에 저장된 빈도수(`freq`)를 `key`값으로 한다.

void insert_node(NODE** list, NODE* new_node)

트리들의 root들로 linked list 를 생성할 때 sorted linked list를 생성하였으며 `freq`가 0이 아닌 table 원소에 대해 `NODE` 변수를 생성한 뒤, `insert_node`함수를 실행해 노드가 `key`에 대한 non-increasing 순서로 정렬될 수 있도록 했다.

greedy approach

single node tree가 저장되어 있는 root에 대해 greedy approach로 하나의 트리로 합쳐 주었다.

NODE* pop_min(NODE** list)

sorted linked list인 root에 대해 pop_min 함수를 실행해 가장 앞에 있는 노드를 가져오고 list의 head를 바꿔주는 함수를 선언하였따.

pop_min 함수를 이용해 root가 가장 작은 tree를 가지는 두 트리를 가져와 두 트리의 root의 key의 합을 새로운 root로 하는 트리를 생성해 다시 sorted_linked_list에 넣어주었다.

```
for (i = 0; i < tree_cnt-1; i++) {
    min1 = pop_min(&root);
    min2 = pop_min(&root);
    tmp_node = (NODE*)malloc(sizeof(NODE));
    tmp_node->key = min1->key + min2->key;
    tmp_node->t_ptr = NULL;
    tmp_node->left = min1;
    tmp_node->right = min2;
    tmp_node->next = NULL;
    insert_node(&root, tmp_node);
  }
```

위의 과정을 통해 Max heap을 구현되고 트리의 leaf node는 ascii_table과 연결된 포인터를 가진 노드들로 구성된다.

void fill_code(NODE* root, int b, int code, int bits)

구현된 Max heap 트리에 대해 각 트리를 search하면서 ascii_table과 연결된 노드에 대해 테이블로 접근해 해당 테이블의 `code`(huffman code의 10진수 값) 과 `bits`(총 bit수)를 저장했다.

재귀적으로 함수를 구현하였으며 왼쪽 자식으로 갈 경우 0, 오른쪽 자식으로 갈 경우 1을 넣어 각 node의 code를 업데이트할 수 있도록 했다.

자식으로 넘어갈 때마다 code의 경우 왼쪽으로 shift 연산 한 뒤, b값을 더하고 bits는 1 추가해 넘겨주었다.

```
code <<= 1;
code += b;
```

함수의 종료 조건은 넘어온 노드가 NULL값일 떄이다.

encoding

greedy approach를 통해 구한 huffman code를 바탕으로 입력 파일을 encoding해주었다.

입력 파일을 읽어 해당 character 의 table에 저장되어있는 bits값으로 인코딩하는 데에 필요한 total bits를 구하고 해당 값을 round up해서 필요한 total bytes 값을 구한다.

0과 1값을 저장해주는 bits_arr 함수를 선언해 index 0부터 차례로 encoding 결과를 저장해준다.

```
bits_cnt = total_bytes * 8 - total_bits;
for (i = 0; i < total_bits; i++) {
  encoding[j] <<= 1;
  encoding[j] += bits_arr[i];
  bits_cnt++;
  if (bits_cnt % 8 == 0) {
    j++;
  }
```

위에서 구한 total bytes 값을 바탕으로 해당 크기를 가지는 char 배열을 할당해 8번씩 끊어 bit_arr에 저장된 값을 더해준다. 해당 값은 LSB에 저장되므로 left shift연산을 1칸 해준 뒤 값을 더해줘야 한다.