```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
#define ALLOC_LENGTH    (2)
#define SOLUTION_1 1
#define SOLUTION_2 0
#define SOLUTION_3 0
#define HASH_SIZE       (5000)
#if(SOLUTION_1)


typedef struct node
{
    void* prev;
    void* next;
}node_t;


typedef struct hash_item
{
    struct node node;
}hash_item_t;

typedef struct str_hash_item
{
    struct hash_item item;
    int len;
    char* word;
    bool* ref;
}str_hash_item_t;


typedef struct hash_map
{
    struct hash_item* item;
    struct hash_item* item_tail;
}hash_map_t;

typedef struct hash_item* (*create_hash_item_t)(struct hash_item*);
typedef uint32_t (*hash_code_t)(struct hash_item*, uint32_t);
typedef bool (*hash_cmp_t)(struct hash_item* , struct hash_item* );

struct hash_map* create_hash(uint32_t entries)
{
    struct hash_map* map;

    map = (struct hash_map*)calloc(entries, sizeof(struct hash_map));

    return map;
}

struct hash_item* create_hash_str_item(struct hash_item* item)
{
    struct str_hash_item* copy_item;

    copy_item = (struct str_hash_item*)calloc(1, sizeof(struct str_hash_item));
    copy_item->item.node.prev = NULL;
    copy_item->item.node.next = NULL;
    copy_item->word = ((struct str_hash_item*)item)->word;
    copy_item->len = ((struct str_hash_item*)item)->len;
    copy_item->ref = ((struct str_hash_item*)item)->ref;

    return (struct hash_item*)copy_item;
}

uint32_t str_hash_code(struct hash_item* item, uint32_t entries)
{
    struct str_hash_item* str_hash_item;
    char *c;
    uint32_t index;
    int word_len;

    unsigned int hc = 5381;
    str_hash_item = (struct str_hash_item*)item;
    c = str_hash_item->word;

    word_len = str_hash_item->len;
    index = 0;

    while(index < word_len)
    {
        hc = hc * 33 + *c;
        c++;
        index++;
    }


    return hc % entries;
}

bool str_cmp(struct hash_item* origin_item, struct hash_item* cmp_item)
{
    struct str_hash_item* o_item;
    struct str_hash_item* c_item;
    int word_len;

    o_item = (struct str_hash_item*)origin_item;
    c_item = (struct str_hash_item*)cmp_item;
    word_len = o_item->len;

    if( (*(o_item->ref) == false) && (strncmp(o_item->word, c_item->word, word_len) == 0) )
    {
        return true;
    }else
    {
        return false;
    }
}

void hash_put(struct hash_map* map, struct hash_item* item, hash_code_t hash_code, create_hash_item_t create_hash_item, uint32_t entries)
{
    uint32_t code;
    struct hash_item* create_item;
    struct hash_item* item_ptr;

    code = hash_code(item, entries);
    create_item = create_hash_item(item);

    if(map[code].item == NULL)
    {
        map[code].item = create_item;
        map[code].item_tail = create_item;
        create_item->node.next = NULL;
        create_item->node.prev = NULL;
    }else
    {
        create_item->node.next = NULL;
        map[code].item_tail->node.next = create_item;
        create_item->node.prev = map[code].item_tail;
        map[code].item_tail = create_item;
    }
}

struct hash_item* hash_get(struct hash_map* map, struct hash_item* item, hash_code_t hash_code, hash_cmp_t hash_cmp, uint32_t entries)
{
    uint32_t code;
    int index;
    struct hash_item* item_ptr;
    struct hash_item* prev_ptr;
    struct hash_item* first_ptr;
    struct str_hash_item* tmp;

    code = hash_code(item, entries);
    first_ptr = NULL;
    prev_ptr = NULL;
    item_ptr = map[code].item;

    while(item_ptr != NULL)
    {
        tmp = (struct str_hash_item*)item_ptr;
        if(*(tmp->ref) == true)
        {
            return NULL;
        }

        if(hash_cmp(item_ptr, item) == true)
```

```
        {
            tmp = (struct str_hash_item*)item_ptr;


            if(prev_ptr && map[code].item_tail != item_ptr)
            {
                prev_ptr->node.next = item_ptr->node.next;
            }

            if(map[code].item == item_ptr && item_ptr->node.next != NULL)
            {
                map[code].item = item_ptr->node.next;
                 tmp = (struct str_hash_item*)map[code].item;
            }

            if(map[code].item_tail != item_ptr)
            {
                map[code].item_tail->node.next = item_ptr;
                map[code].item_tail = item_ptr;
            }

            item_ptr->node.next = NULL;

            return item_ptr;
        }

        prev_ptr = item_ptr;
        item_ptr = item_ptr->node.next;
    }

    return NULL;
}

void reset_ref(bool* ref, int size)
{
    int index;

    for(index = 0; index < size; index++)
    {
        ref[index] = false;
    }
}

int* findSubstring(char * s, char ** words, int wordsSize, int* returnSize){
    int index;
    int s_len;
    int replace_index;
    int s_index;
    int word_len;
    int found;
    int count;
    int count_index;
    int* result;
    int alloc_length;
    int inner_index;
    int section_len;
    int str_len;
    struct hash_map* hash_map;
    struct str_hash_item  str_hash_item;
    struct str_hash_item* found_str_item;
    bool* ref;

    s_len = strlen(s);
    word_len = strlen(words[0]);
    section_len = word_len*wordsSize;
    replace_index = 0;
    s_index = 0;
    alloc_length = ALLOC_LENGTH;
    result = (int*)malloc(alloc_length*sizeof(int));
    ref = (bool*)calloc(wordsSize, sizeof(bool));
    *returnSize = 0;
    hash_map = create_hash(HASH_SIZE);

    for(index = 0; index < wordsSize; index++)
    {
        str_hash_item.word = words[index];
        str_hash_item.ref = &ref[index];
        str_hash_item.len = word_len;
        //printf("index:%d\n", index);
        hash_put(hash_map, (struct hash_item*) &str_hash_item, str_hash_code, create_hash_str_item, HASH_SIZE);
    }

     str_hash_item.ref = (bool*)calloc(wordsSize, sizeof(bool));

    for(index = 0; index < word_len; index++)
    {
        s_index = index;

        while( (s_index + section_len) <= s_len )
        {
            count = 0;
            reset_ref(ref, wordsSize);

            for(inner_index = 0; inner_index < wordsSize; inner_index++)
            {
                count_index = s_index + inner_index*word_len;

                str_hash_item.word = &s[count_index];
                str_hash_item.len = word_len;
                found_str_item = hash_get(hash_map, &str_hash_item, str_hash_code, str_cmp, HASH_SIZE);

                if(found_str_item != NULL)
                {
                    count++;
                    *(found_str_item->ref) = true;

                }else
                {
                    break;
                }
            }

            if(count == wordsSize)
            {

                result[*returnSize] = s_index;
                (*returnSize)++;
                if((*returnSize) == alloc_length)
                {
                    alloc_length*=2;
                    result = (int*)realloc(result, alloc_length*sizeof(int));
                }
            }


            s_index+=word_len;

        }
    }

    return result;

}
#elif(SOLUTION_2)
int* findSubstring(char * s, char ** words, int wordsSize, int* returnSize){
    int index;
    int s_len;
    int replace_index;
    int s_index;
    int word_len;
    int found;
    int count;
    int count_index;
    int* result;
    int* map;
    int alloc_length;
    int* limit;
    uint8_t* limit_count;
    int inner_index;
    int section_len;

    s_len = strlen(s);
    word_len = strlen(words[0]);
    section_len = word_len*wordsSize;
    replace_index = 0;
    s_index = 0;
```

```
alloc_length = ALLOC_LENGTH;
result = (int*)malloc(alloc_length*sizeof(int));
map = (int*)calloc(s_len, sizeof(int));
limit = (int*)calloc(wordsSize, sizeof(int));
limit_count = (uint8_t*)calloc(wordsSize, sizeof(uint8_t));
*returnSize = 0;


for(index = 0; index < wordsSize; index++)
{
    if(limit[index] == 0)
    {
        limit[index] = 1;
        for(inner_index = index+1; inner_index < wordsSize; inner_index++)
        {
            if(strcmp(words[index], words[inner_index]) == 0)
            {
                limit[inner_index] = -1;
                limit[index]++;
            }
        }
    }

}

while(s[s_index] != '\0')
{
    found = false;

    for(index = 0; index < wordsSize; index++)
    {
        if (limit[index] != -1 && strncmp(words[index], &s[s_index], word_len) == 0)
        {
            map[s_index] = index;
            found = true;
            break;
        }
    }

    if(found)
    {
        s_index++;
    }else
    {
        map[s_index] = INT_MAX;
        s_index++;
    }

}


for(index = 0; index < word_len; index++)
{
    s_index = index;

    while( (s_index + section_len) <= s_len )
    {
        count = 0;
        memset(limit_count,0x0, sizeof(uint8_t)*wordsSize);

        for(inner_index = 0; inner_index < wordsSize; inner_index++)
        {
            count_index = s_index + inner_index*word_len;

            if(map[count_index] != INT_MAX)
            {
                if( limit_count[map[count_index]] < limit[map[count_index]] )
                {
                    limit_count[map[count_index]]++;
                    count++;

                }else
                {
                    break;
                }
            }else
            {
                break;
            }
        }

        if(count == wordsSize)
        {
            result[*returnSize] = s_index;
            (*returnSize)++;
            if((*returnSize) == alloc_length)
            {
                alloc_length*=2;
                result = (int*)realloc(result, alloc_length*sizeof(int));
            }
        }


        s_index+=word_len;

    }
}

/*
count = 0;
s_index = 0;
count_index = INT_MIN;
while(s[s_index] != '\0')
{
    if(map[s_index] != INT_MAX)
    {
        if( limit_count[map[s_index]] < limit[map[s_index]] )
        {
            if(INT_MIN == count_index)
            {
                count_index = s_index;
            }

            limit_count[map[s_index]]++;
            count++;
            //printf(" %d, %d, %d\n", count, s_index, count_index);
        }else
        {
            count = 0;
            s_index = count_index+1;
            count_index = INT_MIN;
            memset(limit_count,0x0, sizeof(int)*wordsSize);
            continue;
        }

        if(count == wordsSize)
        {
            result[*returnSize] = count_index;
            (*returnSize)++;
            if((*returnSize) == alloc_length)
            {
                alloc_length*=2;
                result = (int*)realloc(result, alloc_length*sizeof(int));
            }
            s_index = count_index+1;
            count_index = INT_MIN;
            count = 0;
            memset(limit_count,0x0, sizeof(int)*wordsSize);
            continue;
        }

        s_index += word_len;
    }else
    {

        count = 0;
        memset(limit_count,0x0, sizeof(int)*wordsSize);
        if(count_index != INT_MIN)
        {
            s_index = count_index;
            count_index = INT_MIN;
        }

        s_index++;
    }
}
```

```
    */
    return result;

}
#elif(SOLUTION_3)
/*
30. Substring with Concatenation of All Words

You are given a string, s, and a list of words, words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exa



For example, given:
s: "barfoothefoobarman"
words: ["foo", "bar"]



You should return the indices: [0,9].
(order does not matter).
*/

/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
typedef struct item_s {
    char *word;
    int idx;
    struct item_s *next;
} item_t;

typedef struct {
    item_t *p;
    int n;
} buff_t;

#define HF 1021

unsigned int hash_code(const char *s, int len) {
    unsigned int hc = 5381;
    char c;
    while (len -- > 0) {
        hc = hc * 33 + s[len];
    }
    return hc % HF;
}

item_t *lookup(item_t **ht, unsigned int hc, const char *w, int len) {
    item_t *p = ht[hc];
    while (p) {
        if (!strncmp(p->word, w, len)) {
            return p;
        }
        p = p->next;
    }
    return NULL;
}

int* findSubstring(char* s, char** words, int wordsSize, int* returnSize) {
    // 1. sort all words
    // 2. caculate each uniq words appears how many times
    /* for (i = 0; i < length of string; i ++) {
        substr = s[i ... length of one word]
        if (substr is not one of the words or it appears more than what we have) continue;
        increase the count of this word
        if all words are found, add i into result
    } */

    item_t *ht[HF] = { 0 }, **sp, *p;
    int *counts, *counts2, total, i;
    char *w;
    unsigned int hc;

    int total_len, word_len;
    int left, mid, right;

    int *results;

    if (wordsSize == 0) return NULL;

    buff_t buff = { 0 };
    buff.p = malloc(wordsSize * sizeof(item_t));
    //assert(buff->p);
    counts = calloc(wordsSize * 2, sizeof(int));
    //assert(counts);
    counts2 = &counts[wordsSize];
    total = 0;

    word_len = strlen(words[0]);
    total_len = strlen(s);

    sp = malloc(total_len * sizeof(item_t *));
    //assert(sp);

    results = malloc(total_len * sizeof(int));
    //assert(results);
    *returnSize = 0;

    for (i = 0; i < wordsSize; i ++) {
        w = words[i];
        hc = hash_code(w, word_len);
        p = lookup(ht, hc, w, word_len);
        if (p) {
            counts[p->idx] ++;
        } else {
            p = &buff.p[buff.n];

            p->idx = buff.n ++;
            p->word = w;

            p->next = ht[hc];
            ht[hc] = p;

            counts[p->idx] = 1;
        }
    }

    left = mid = right = 0;
    while (right <= total_len - word_len) {
        w = &s[right];
        hc = hash_code(w, word_len);
        p = lookup(ht, hc, w, word_len);
        if (!p) {
            total = 0;
            memset(counts2, 0, buff.n * sizeof(int));   // reset all counts
            left ++;            // shift one character from left
            mid = left;
            right = left;       // reset right
        } else {
            sp[right] = p;
            i = p->idx;
            counts2[i] ++;
            total ++;
            while (counts2[i] > counts[i]) {
                p = sp[mid];
                mid += word_len;   // push out a word
                counts2[p->idx] --;
                total --;
            }
            if (total == wordsSize) {   // all are found
                results[*returnSize] = mid;
                (*returnSize) ++;
                total = 0;
                memset(counts2, 0, buff.n * sizeof(int));   // reset all counts
                left = mid + 1;
                mid = left;
                right = left;
            } else {
                right += word_len;
            }
        }
    }

    free(buff.p);
    free(counts);
```

```
    free(sp);

    return results;
}


/*
Difficulty:Hard
Total Accepted:82.6K
Total Submissions:376.3K


Related Topics Hash Table Two Pointers String
Similar Questions Minimum Window Substring

*/
#endif
```