[CSE3081(2반)] 알고리즘 설계와 분석 과제 1 보고서

20151623 한상구

1. 5개의 알고리즘에 대한 요약

A. Algorithm 1, find max-sum subarray with $O(n^2)$

Subarray의 처음과 끝을 각각 i, j라고 하자. 이들의 가능한 위치는 given input size가 n개 일 때 $\frac{n(n-1)}{2}$개 이고, 이들을 전부 고려한다면 위에서 주어진 시간 복잡도가 나올 것이다.

Brute force.

B. Algorithm 2, find max-sum subarray with $O(n\,log\,n)$

$log\,n$ 의 시간복잡도를 가지기 위해서는 각 step에 대하여 $\frac{1}{k}, where\;k \in \mathbb{Z}$ 의 크기로 분할되어야 한다. 각 단계를 거칠 때 마다, 반으로 분할하며 진행하였고 주어진 과제에서 원하는 시간 복잡도를 얻을 수 있었다. $Max(left,\;right,\;cross)$ 의 진행이었다.

Divide and Conquer.

C. Algorithm 3, find max-sum subarray with $O(n)$

Linear한 시간복잡도를 위하여 수학적 접근을 시도했으며, 다음과 같은 점화식을 얻어낼 수 있었다.

$max(i) = max(0, max(i-1)) + A[i], where\;max(k)\;stands\;for\;maximum\;sum\;at\;k$

이 점화식을 구현 한 결과, 시간 복잡도를 linear하게 줄일 수 있었다.

Dynamic programming.

D. Algorithm 4, find max-sum subrectangle with $O(n^4)$

A와 같이, Subrectangle의 top-left와 bottom-right을 각각 (i,k), (j,l) 이라고 하자. 이들의 가능한 위치는 given input size가 n개 일 때 $\left(\frac{n(n-1)}{n}\right)^2$개 이고, 이들을 전부 고려한다면 위에서 주어진 시간 복잡도가 나올 것이다.

각 Subrectangle의 크기를 요청하는 쿼리에 대해 $O(1)$으로 응답해야 위의 시간 복잡도를 유지할 수 있으므로, row-major partial sum을 응용하였다.

Brute force & partial sum.

E. Algorithm 5, find max-sum subrectangle with $O(n^3)$

D에서 n을 줄이기 위해서, row, 혹은 col을 고정시키고 그 사이에서 C를 적당히 변형한

알고리즘을 적용시킬 수 있었다.

2. 시간 복잡도

A. $O(n^2)$

```
s = 1 << 31;      // set s as least value in integer
for (i = 0; i < n; i++)       // find max-sum subarray
{
    tmp = 0;
    for (j = i; j < n; j++)
    {
        tmp += A[j];
        if (s < tmp)     // if current sum of subarray is bigger than s, update the info of answer
        {
            a[0] = s = tmp;
            a[1] = i;
            a[2] = j;
        }
    }
}
```

i와 j는 각각 subarray의 처음과 끝 인덱스를 가리키고 있다.

$[0, n-1]X[i, n-1]$ 의 반복 수행 끝에 정답을 찾아낼 것이며, 모든 경우의 수를 보기 때문에 정답은 보장되어있다.

위에 기술한 범위와 같이 시간복잡도는 $O(n\text{^}2)$가 될것이다.

B. $O(n \log n)$

```
a[0] = Algorithm2_recur(A, n, 0, &a[1], &a[2]);
```

```c
int Algorithm2_recur(int *arr, int len, int f, int *s, int *e)
{
    if (len == 1) return arr[0];
    if (!len) return 0;
    int mid, left, right, lmax, rmax, tmp, i, center, to, from;
    mid = len / 2;
    left = Algorithm2_recur(arr, mid, f, s, e);
    right = Algorithm2_recur(arr + mid, len - mid, f + mid, s, e);
    lmax = rmax = -10000000;
    tmp = 0;
    for (i = mid; i >= 0; i--)
    {
        tmp += arr[i];
        if (lmax < tmp)
            lmax = tmp, from = i;
    }
    tmp = 0;
    for (i = mid + 1; i < len; i++)
    {
        tmp += arr[i];
        if (rmax < tmp)
            rmax = tmp, to = i;
    }
    center = lmax + rmax;
    if (left >= right && left >= center)
        return left;
    if (right > left && right > center)
        return right;
    *s = from + f; *e = to + f;
    return center;
}
```

Len이 1 이하인 경우가 base case가 되는 Recursive function으로 구현하였다.

절반으로 나누어 left, right으로 구분하였고, mid를 걸치게 되는 max-sum과의 비교를 통해 max-sum subarray를 찾아내었다. (index는 결국 subarray끼리 합쳐지게 되므로 그 경우에 저장하였음)

Given input size가 n인 경우, maximum depth는 $[\log n]$이 되고, 각 경우 n만큼 살펴보게 되므로 전체적인 시간복잡도는 $O(n \log n)$이 될 것이다.

$T(n) = T\left(\frac{n}{2}\right) + c(n)$ 을 통해서도 복잡도를 이끌어 낼 수 있다.

C. $O(n)$

```
s = 1 << 31;
tmp = 0; i = 0;
for (j = 0; j < n; j++)
{
    fread(&A, sizeof(int), 1, fp);  // get info of array elements
    if (tmp > 0)
        tmp += A;
    else
    {
        tmp = A;
        i = j;
    }
    if (tmp > s)
    {
        a[0] = s = tmp;
        a[1] = i;
        a[2] = j;
    }
}
```

$$max(i) = max(0, max(i-1)) + A[i], where\ max(k)\ stands\ for\ maximum\ sum\ at\ k$$

위 식을 그대로 구현하였고, 이 경우에는 굳이 배열을 선언 할 필요 없이, 전 상태와 현 상태만을 살펴보면 되므로 element를 읽어오며 상태를 갱신하였다.

Given input size가 n인 경우, 시간복잡도는 $O(n)$가 될 것이다.

D. $O(n^4)$

```c
fread(&n, sizeof(int), 1, fp);  // get n
A = (int **)malloc(sizeof(int *)*n);
PS = (int **)malloc(sizeof(int *)*(n + 1));
for (i = 0; i < n; i++)
{
    *(A + i) = (int *)malloc(sizeof(int)*n);
    *(PS + i) = (int *)calloc(n + 1, sizeof(int));
    fread(*(A + i), sizeof(int), n, fp);    // get info of array elements
}
*(PS + n) = (int *)calloc(n + 1, sizeof(int));
fclose(fp);     // close input file

for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        PS[i][j] = A[i - 1][j - 1] - PS[i - 1][j - 1] + PS[i][j - 1] + PS[i - 1][j];

s = 1 << 31;      // set s as least value in integer
for (i = 0; i < n; i++)  // i as top-left
{
    for (j = i; j < n; j++)  // j as bottom-right
    {
        for (k = 0; k < n; k++)  // k as top
        {
            for (l = k; l < n; l++)  // l as bottom
            {
                tmp = PS[l + 1][j + 1] - PS[l + 1][i] - PS[k][j + 1] + PS[k][i];
                if (tmp > s)
                {
                    a[0] = s = tmp;
                    a[1] = k; a[2] = i;
                    a[3] = l; a[4] = j;
                }
            }
        }
    }
}
```

PS는 입력받은 2차원 배열에 대한 row-major partial sum을 담고 있는 배열이고,

(top, left) := (i, k), (bottom, right) := (j, l) 로 가정 한 뒤 모든 경우의 수를 탐색하였다. 시간복잡도에 대한 계산은 1.D.에서 언급하였고, $O(n^4)$가 될 것이다.

추가적으로, partial sum을 구하는 과정은 $O(n^2)$이지만, $O(n^4)$가 dominate하므로 시간복잡도에는 변함이 없다.

Partial sum을 통한 rectangle의 계산은 합집합-교집합을 이용하여 구할 수 있었다.

E. $O(n^3)$

```c
int Algorithm5_sub(int *tmp, int *s, int *e, int r)     // modified from Algorithm3 func
{
    int local = 0, max = 1 << 31, i, start;
    *e = 0xffffffff;
    for (i = start = 0; i < r; i++)
    {
        local += tmp[i];
        if (local < 0)
        {
            local = 0;
            start = i + 1;
        }
        else if (local > max)
        {
            max = local;
            *s = start;
            *e = i;
        }
    }
    if (*e != 0xffffffff)
        return max;

    // when all numbers are negative (in tmp array)
    max = tmp[0];
    *s = *e = 0;
    for (i = 1; i < r; i++)
        if (tmp[i] > max)
            max = tmp[i], *s = *e = i;
    return max;
}
```

주석에서 볼 수 있듯이, C의 알고리즘을 살짝 변형하였다. 시간복잡도는 $O(n)$로 동일.

```c
fread(&n, sizeof(int), 1, fp);  // get n
A = (int **)malloc(sizeof(int *)*n);
tmp = (int *)malloc(sizeof(int)*n);
for (i = 0; i < n; i++)
{
    *(A + i) = (int *)malloc(sizeof(int)*n);
    fread(*(A + i), sizeof(int), n, fp); // get info of array elements
}
fclose(fp);     // close input file

a[0] = 1 << 31;    // set a[0] as least value in integer

for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++) tmp[j] = 0;
    for (j = i; j < n; j++)
    {
        for (k = 0; k < n; k++)
            tmp[k] += A[k][j];      // partial sum array for every row between i-th and j-th col
        local = Algorithm5_sub(tmp, &s, &e, n);
        if (local > a[0])
        {
            a[0] = local;
            a[1] = s; a[2] = i;
            a[3] = e; a[4] = j;
        }
    }
}
```

본 알고리즘 함수를 살펴보자. for문이 총 3번 중첩되어있으며(각각 $O(n)$임을 볼 수 있다), col을 고정하고 탐색을 진행해 나가는 것을 볼 수 있다. 처음으로 중첩 된 for문에서는 partial sum을 사용하기 위해 초기화를 진행하는 $O(n)$, 마지막 for문에서는 partial sum을 위한 $O(n)$, 위에서 C의 알고리즘을 살짝 변형하였고 이를 위한 $O(n)$이 필요함을 볼 수 있고, 취합한다면 총 $O(n^3)$이 됨을 볼 수 있다.

3. 실행 시간

   A. $O(n^2)$

```
Algorithm 1, With size'10'
0.2160 ms 0.1460 ms 0.1620 ms 0.1600 ms 0.1330 ms
Average time: 0.1634
Algorithm 1, With size'100'
0.2750 ms 0.2540 ms 0.1290 ms 0.1220 ms 0.1190 ms
Average time: 0.1798
```

   B. $O(n \log n)$

```
Algorithm 2, With size'10'
0.1560 ms 0.0960 ms 0.0870 ms 0.0760 ms 0.1000 ms
Average time: 0.1030
Algorithm 2, With size'100'
0.1930 ms 0.1540 ms 0.1180 ms 0.0870 ms 0.1180 ms
Average time: 0.1340
```

   C. $O(n)$

```
Algorithm 3, With size'10'
0.1530 ms 0.1200 ms 0.0860 ms 0.0780 ms 0.0690 ms
Average time: 0.1012
Algorithm 3, With size'100'
0.1780 ms 0.1170 ms 0.1010 ms 0.0950 ms 0.1040 ms
Average time: 0.1190
```

   D. $O(n^4)$

```
Algorithm 4, With size'10'
0.2110 ms 0.2450 ms 0.1120 ms 0.1680 ms 0.1650 ms
Average time: 0.1802
Algorithm 4, With size'100'
157.8970 ms 138.5200 ms 192.5180 ms 177.4360 ms 131.9010 ms
Average time: 159.6544
```

E. $O(n^3)$

```
Algorithm 5, With size'10'
0.1900 ms 0.1900 ms 0.1000 ms 0.0820 ms 0.0760 ms
Average time: 0.1276
Algorithm 5, With size'100'
5.3610 ms 6.4880 ms 5.9420 ms 5.4460 ms 5.3290 ms
Average time: 5.7132
```

F. 측정 함수

```c
clock_t start, end;

srand((unsigned int)time(NULL));
printf("Algorithm 5, With size'100'\n");

fscanf(fp, "%d", &tc);  // get the number of test cases
while( tc-- )   // for the given test cases
{
    fscanf(fp, "%d %s %s", &op, input, output); // get info
    start = clock();
    switch(op){
        case 1: Algorithm1(input, output); break;
        case 2: Algorithm2(input, output); break;
        case 3: Algorithm3(input, output); break;
        case 4: Algorithm4(input, output); break;
        case 5: Algorithm5(input, output); break;
        default: break;
    }
    end = clock();
    printf("%.4lf ms ", (double)(end-start)/CLOCKS_PER_SEC*1000);
    sum += (end - start);
}
printf("\nAverage time: %.4lf\n", (double)sum/CLOCKS_PER_SEC*1000/5);
```

4. 실험 환경

A. OS: OS X El Capitan Ver. 10.11.6

B. CPU: 1.7 GHz Intel Core i7

C. RAM: 8.00GB

D. Compiler: gcc compiler Ver. 4.2.1