A warm up!

Chapter 2 of the notes

Maximum Sub-array Sum

Problem: given an array of **n** integers (positive and negative) find the sub-array of maximum sum.

Input: array $A[1,n]$ of positive and negative integers

Output: l, r where $A'[l, r]$ is the sub-array

Output: The value of the sum

Example:

Input -1 5 8 -9 1 1

Output 13

Solution 1

```
max=a[0];
for(i=0; i<n; i++)
{
    for(j=i; j<n; j++)
    {
        sum=0;
        for(k=i; k<=j; k++)
        {
        sum+=a[k];
        }
        if (sum> max) max=sum;
    }
}
```

```
max=a[0];
for(i=0; i<n; i++)
{
    for(j=i; j<n; j++)
    {
        sum=0;
        for(k=i; k<=j; k++)
        {
        sum+=a[k];
        }
        if (sum > max) max=sum;
    }
}
```

```
              i     j
Input   -1 5 8 -9 1 1
```

Solution 1

```
max=a[0];
for(i=0; i<n; i++)
{
    for(j=i; j<n; j++)
    {
        sum=0;
        for(k=i; k<=j; k++)
        {
        sum+=a[k];
        }
        if (sum > max) max=sum;
    }
}
```

$O(1)$

$O(n)$

$O(n^2)$

$O(n^3)$

Time complexity $O(n^3)$

Lower Bound for Solution 1

Instruction somma+=a[k];
(on sub-arrays of length j-i+1) is executed

$$\sum_{i=0,n-1}\sum_{j=i,n-1}(j-i+1)$$ times, or

$$\sum_{i=0,n-1}\sum_{j=1,n-i} j \geq \sum_{i=0,n-1} (n-i)^2/2 \geq \sum_{j=1,n+1} j^2/2$$

$$\Omega(n^3)$$

Solution 2

```
max=a[0];
for(i=0; i<n; i++)
{
   sum=0;
   for(j=i; j<n; j++)
   {
      sum+=a[j];
      if(sum > max) max=sum;
   }
}
```

```
        i         j
Input -1 5 8 -9 1 1
```

Solution2

```
max=a[0];
for(i=0; i<n; i++)
{
    sum=0;
    for(j=i; j<n; j++)
    {
        sum+=a[j];
        if(sum > max) max=sum;
    }
}
```

$O(1)$

$O(n)$

$O(n^2)$

Time complexity: $O(n^2)$

How to do better

Observe 2 properties of the maximum sum sub-array:

1) The sum of the values in each prefix of the maximum sum sub-array is positive, otherwise we could remove this prefix obtaining a sub-array with greater sum (contradiction).
2) The value of the element previous than the first element of the maximum sum sub-array is negative, otherwise it could be added to the sub-array obtaining a sub-array with greater sum (contradiction).

-3 2 3 -1 8 1

Solution 3

```
max = A[0]; sum = 0;
for(i=0; i<n; i++)
   {
   if(sum > 0) sum+=a[i];    extend the segment
       else sum=a[i];        start a new segment
   if(sum > max) max=sum;
   }
```

Time complexity: O(n)

Optimal Algorithm ! Why?

Example linear solution

	SUM	MAX
1 2 -4 1 3 2 -2 1	1	1
1 2 -4 1 3 2 -2 1	3	3
1 2 -4 1 3 2 -2 1	-1	3
1 2 -4 1 3 2 -2 1	1	3
1 2 -4 1 3 2 -2 1	4	4
1 2 -4 1 3 2 -2 1	6	6
1 2 -4 1 3 2 -2 1	4	6
1 2 -4 1 3 2 -2 1	5	6

2-level memory model

- B= block (page) size
- M= internal memory size

How to evaluate the complexity of an algorithm?

number of I/Os operations

- In this model Solution3 takes n/B I/Os operations is optimal.
- It is independent from the block size. Very important feature for an algorithm. Cache-oblivious.

Another linear time algorithm

- Let Sum_D [x,s] the sum of items in positions from x to s

$$Sum_D[x,s] = Sum_D[1,s] - Sum_D[1,x-1]$$

prefix sums until s prefix sums until x-1 O(n)

New algorithm:

$$\max_{s}\Big(\max_{b \le s} Sum_D[b,s]\Big)$$

$$\max_{s}\Big(\max_{b \le s} Sum_D[1,s] - Sum_D[1,b-1]\Big)$$

Another linear time algorithm

Find the positions <l,r> of the subarray.
Compute prefix sums of D in array P.

D	4	-6	3	1	3	-2	3	-4	1	-9	6
P	4	-2	1	2	5	3	6	2	3	-6	0

Note that $P[i] = P[i-1] + D[i]$ pose $P[0] = 0$

Write the sum in terms of P:

$$\max_{s}\left(\max_{b \leq s} Sum_D(1,s) - Sum_D(1,b-1) \right)$$

$$\max_{s}\left(\max_{b \leq s} (P[s] - P[b-1]) \right)$$

Decompose max computation in max-min computation:

$$\max_{s}\left(\max_{b \leq s} (P[s] - P[b-1]) \right) = \max_{s}\left(P[s] - \min_{b \leq s} P[b-1] \right)$$

independent from b can be precomputed in M

Another linear time algorithm

D

4	-6	3	1	3	-2	3	-4	1	-9	6

P

0	4	-2	1	2	5	3	6	2	3	-6	0

M[i] = min (M[i-1], P[i])   M[0] = 0   M[0, 10]

$\min_{b \leq s} P[b-1]$)

M

0	0	-2	-2	-2	-2	-2	-2	-2	-2	-6

P[s]- M[s-1]

P'

4	-2	3	4	7	5	8	4	5	-4	6

Max value is 8 for s=7,

Left extreme is computed as position where P' is min, l=2, +1 :   Return <3, 7>

Another linear time algorithm

Algorithm 2.4 Another linear-time algorithm

```
MaxSum = −∞; b_o = 1;
TmpSum = 0; MinTmpSum = 0;
for (s = 1; s ≤ n; s++) do
    TmpSum += D[s];
    if (MaxSum < TmpSum − MinTmpSum) then
        MaxSum = TmpSum; s_o = s;
    end if
    if (TmpSum < MinTmpSum) then
        MinTmpSum = TmpSum; b_o = s + 1;
    end if
end for
return ⟨MaxSum, b_o, s_o⟩;
```

The discussed algorithm takes three scans over D, in fact can be organized in a single pass and no memory.

Keep the values of P[s] and of M[s-1] (max and min) in 2 variables Tmpsum and MinTmpSum scanning the array and compute formula P[s]- M[s-1] incrementally.

Interesting variants with application to the Bio-informatics

Sub-array → Segment

Maximum-sum segment problem

DNA sequence is a string on 4 letters (A,T, C, G)

AGATA
TAGATÀ
GATTA
GATTACCA
ATTACCA
ACCATTA
TACATACA

Interesting variants with application to the Bio-informatics

Sub-array → Segment

Maximum-sum segment problem

DNA sequence is a string on 4 letters (A,T, C, G)

Problem: Identify segments rich of C and G nucleotides (biologically significant).

Is it possible to exploit our algorithm?

Input: from DNA sequences to arrays of numbers.

Two ways

1. Assign penalty $-p$ to A and T and a reward $1-p$ to C and G. In this way the sum of the values of a segment of length l containing x C or G is $x-pl$.

   $$x(1-p)-(l-x)\,p = x-xp-xl+xp = x-pl$$

Our linear algorithm can be used to solve this problem with this objective function.

Another way

2. Assign value 0 to A and T and 1 to C and G. Compute the density of a segment l, containing x occurrences of C and G as x/l.

   Segments of length 1 containing C or G have max density, but they are not very interesting!

   Often biologists prefer to put limits on the length of the segments, to avoid extremely short or long segments. Now the algorithm cannot be applied!!!

   Another linear solution, however, is possible.

Small changes in the problem:: Big Jump in the complexity

The trap is: no limits to the segment length implies only trivial solutions of length 1.

Circumvent the single output searching:

Problem: Maximize the sum provided that its length is within a given range.

Complementary problem : Given an array D[1,n] of positive and negative integers , find the longest segment in D whose sum is largest of a fixed threshold t.

The structure of the algorithmic solution is the same!

The two problem can be reduced one into the other

$$\frac{Sum_D(x,y)}{y-x+1} = \sum_{k=x,y} \frac{D[k]}{y-x+1} \geq t \iff \sum_{k=x,y} (D[k] - t) \geq 0$$

Subtracting t to all elements in D, the problem of bounded density becomes equivalent to find the longest segment with sum larger or equal to 0.

A sum based problem is equivalent to a density based problem!

Reduction is very important technique to reuse solutions.

Go back to the problem:

Given an array D[1,n] of positive and negative integers , find the longest segment in D whose sum is larger than a fixed threshold s.

Property

Inductively: consider $D[1, i-1]$ and let $D[l_{i-1}, r_{i-1}]$ the longest segment with sum $\geq t$ already computed for $i=1, \ldots, i-1$.

Consider $D[1, i]$: from the solution $D[l_{i-1}, r_{i-1}]$. 2 possibilities:

1. $r_i < i$ the solution does not change
2. $r_i = i$ $D[l_i, r_i]$ is longer than $D[l_{i-1}, r_{i-1}]$.

Observe: l_i occurs to the left of position $L_i = i - (r_{i-1} - l_{i-1})$

We can discard for l_i all positions between Li and i since they generate solutions shorter than $D[l_{i-1}, r_{i-1}]$.

Reformulated problem. Given $D[1, n]$ of positive and negative numbers we want to find at every step the smallest index li such that $Sum_D [l_i, i] \geq t$. (l_i in the interval $[1, Li)$).

Algorithm

Recall to compute the sum compute the prefix sums as before so that $Sum_D[1, i]$- $Sum_D[1,li-1] = P[i]-P[li-1]$.

We look for the smallest index li in [1, Li] such that $P[i]-P[li-1] \geq t$.

Array P is pre-computed in linear time and space.

We have to find a minimum for all values of i, that means $O(n^2)$.

Instead, we identify a set of candidate positions for iteration i: $C_{i,j}$ is the position of the leftmost minimum of the sub-array $P[1, C_{i,j-1} -1]$. C_{i0}=Li.

P

							Li		
4	7	3	8	3	1	6	2	3	2
C_{i3}		C_{i2}			C_{i1}		C_{i0}		

Properties: $C_{i3} < C_{i2} < C_{i1} < C_{i0}=Li$

$P[C_{i3}] > P[C_{i2}] > P[C_{i1}]$

$P[C_{ij}]$ is the leftmost minimum of prefix $P[1, C_{i,j-1} -1]$

hence is smaller than any other values on its left.

Algorithm

P

							Li		
4	**7**	**3**	**8**	**3**	**1**	**6**	**2**	**3**	**2**
C_{i3}		C_{i2}			C_{i1}		C_{i0}		

Properties: $C_{i3} < C_{i2} < C_{i1} < C_{i0} = Li$

$P[C_{i3}] > P[C_{i2}] > P[C_{i1}]$

$P[C_{ij}]$ is the leftmost minimum of prefix $P[1, C_{i,j-1} - 1]$

hence is smaller than any other values on its left.

$Sum_D[C_{i,j}+1, i]$ can be computed as $P[i] - P[C_{ij}]$.

Algorithm

Fact: At each iteration i, the largest index j* such that $Sum_D [C_{i,j^*+1}, i] \geq t$, if any, is the longest segment we are searching for.

Proof: in the lecture notes.

The computation of all candidate positions takes O(n) time. See also lecture notes.

Computing j* does not require constant time at each iteration, but… if at iteration i, we perform $\Theta(si)$ steps we extend the solution by $\Theta(si)$ units. Hence the sum of the extra cost cannot be larger than n.

Algorithm

Amortized complexity:

$O(n)$, since le length of the solution is at most n.