CIS 121

Programming

FALL 2008

Lecture Notes9

© 1999-2006 S.Kannan, S.Guha, V.Tannen & K. Daniilidis

Improving asymptotic running time

Next we look at a problem (finding the maximum contiguous subsequence sum) that can be solved by several algorithms.

We will see that additional thinking and clever ideas can lead to algorithms that run in better asymptotic bounds!

Maximum Contiguous Subsequence Sum: One Problem, Many Solutions

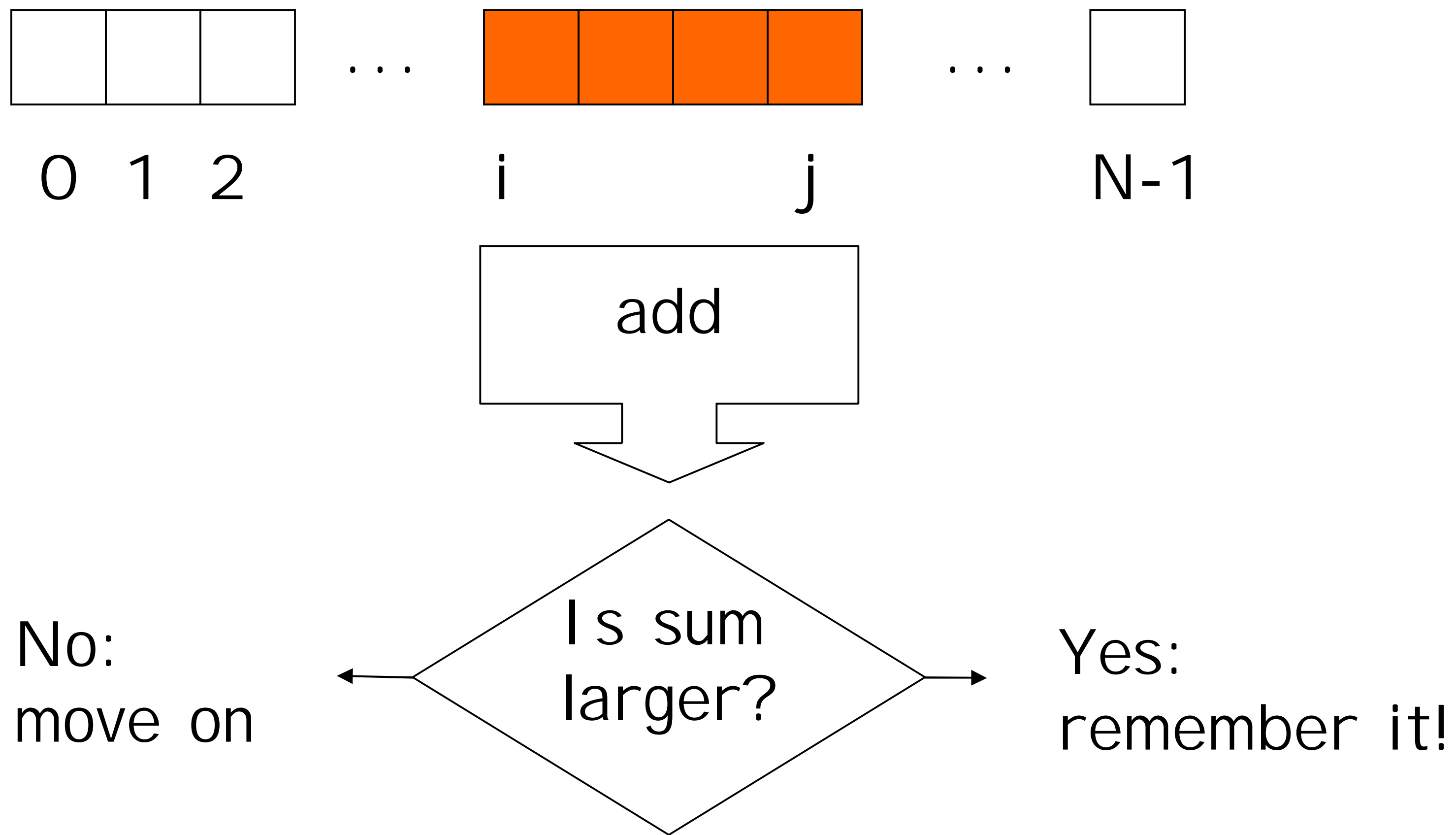
Problem: Given a sequence of (possibly negative) integers $A_1, A_2, \ldots, A_N,$ find the maximum value of the sum of a subsequence: $\sum_{k=i}^{j} A_k$

The maximum subsequence sum is 0 if all the integers are negative.

Example: The input is [-2, 11, -4, 13, -5, 2]
The answer is 20, by summing [11, -4, 13].

Solution 1 --- ruminations...

Consider all possibilities for where the answer might come from...



Analysis of solution 1

When we are sufficiently clear in our ruminations we should be able to analyze the program even before we write it! In fact, we are analyzing the algorithm, not the details of the program.

- To add up the interval of length d takes $\Theta(d)$ steps.
- There are (N-d+1) intervals of length d (starting at positions 0,1,...,N-d).
- Time taken to add up all intervals of length d: $\Theta((N-d+1)*d)$
- Possible values of d: 1,...,N.
- Calculate

$$\sum_{d=1}^{N}(N-d+1)*d$$

(recall $1^2+2^2+\ldots+N^2=?$)

Obtain $\Theta(N^3)$

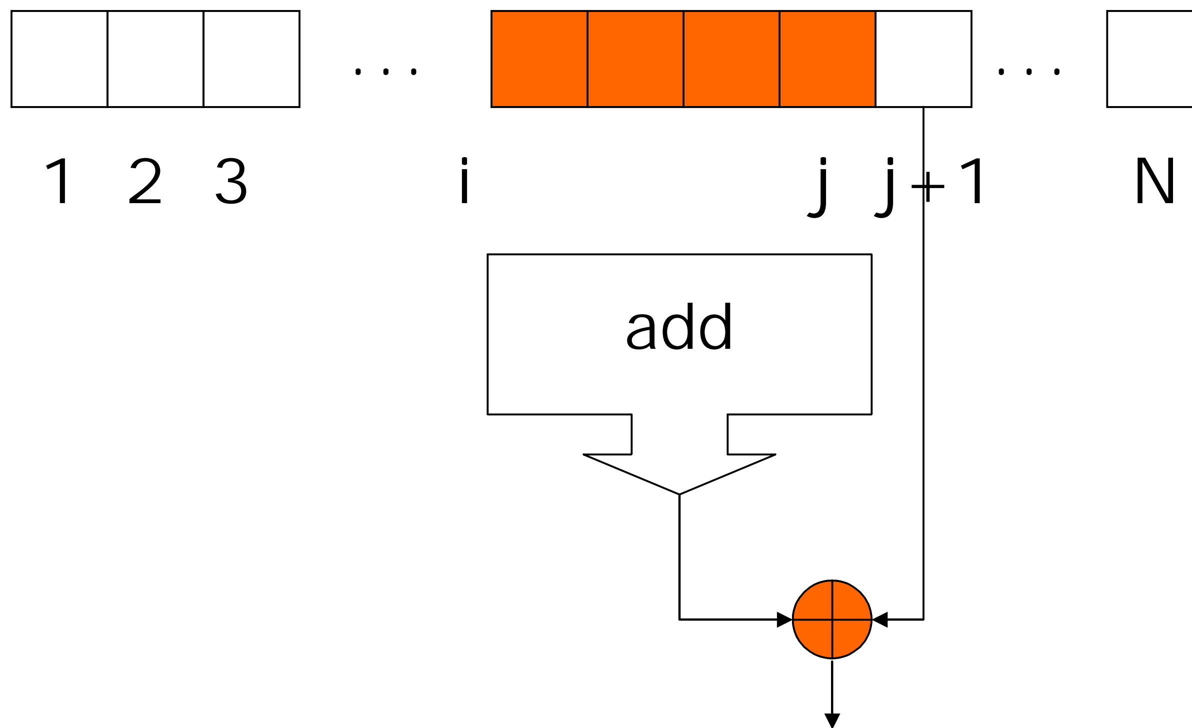
Program 1

```
public static int maxSubSum1( int [] a)  {
  int maxSum = 0;
  for (int i=0; i < a.length; i++) {
    for (int j=i; j < a.length; j++) {
      int thisSum = 0;

      for (int k=i; k <= j; k++)
        thisSum = thisSum + a[k];

      if (thisSum > maxSum)
        maxSum = thisSum;
    }
  }
  return maxSum;
}
```

$\Theta(n^3)$

Solution 2: a simple idea for improvement

We don't need to compute interval sums from scratch.



After adding numbers in the interval [i,j], the interval [i,j+1] takes only one more step.

Program 2

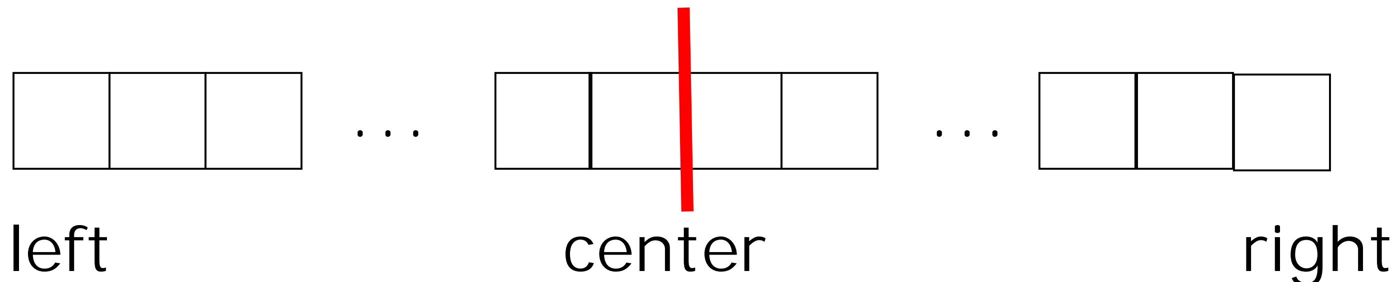
```
public static int maxSubSum2( int[] a)  {
  int maxSum = 0;
  for (int i=0; i < a.length; i++) {
    int thisSum = 0;
    for (int j=i; j < a.length; j++) {
      thisSum = thisSum + a[j];
      if (thisSum > maxSum)
        maxSum = thisSum;
    }
  }
  return maxSum;
}
```

$\Theta(n^2)$

(We've analyzed programs with this structure before.)

Solution 3: divide-and-conquer, using recursion

Consider the center of the sequence:



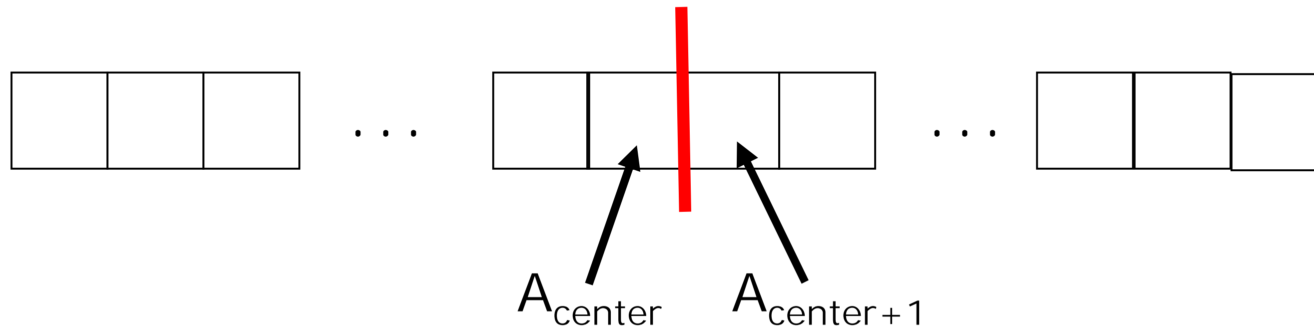
Where is the max subsequence? Three possibilities:

1) All of it is left of center --> find by recursive call.

2) All of it is right of center --> find by recursive call.

3) Across the center (see next slide).

Solution 3: key idea and analysis



Key idea: when the max sum subsequence is across the center, both A_{center} and $A_{center+1}$ must be in it so we can maximize separately the part that is left of center and ends at A_{center} and the part that is right of center and starts at $A_{center+1}$. This part is $\Theta(N)$!

Recurrence relation: $T(N) = 2T(N/2) + cN$

Solving this gives **$T(N)$ is $\Theta(N \log N)$**

Program 3

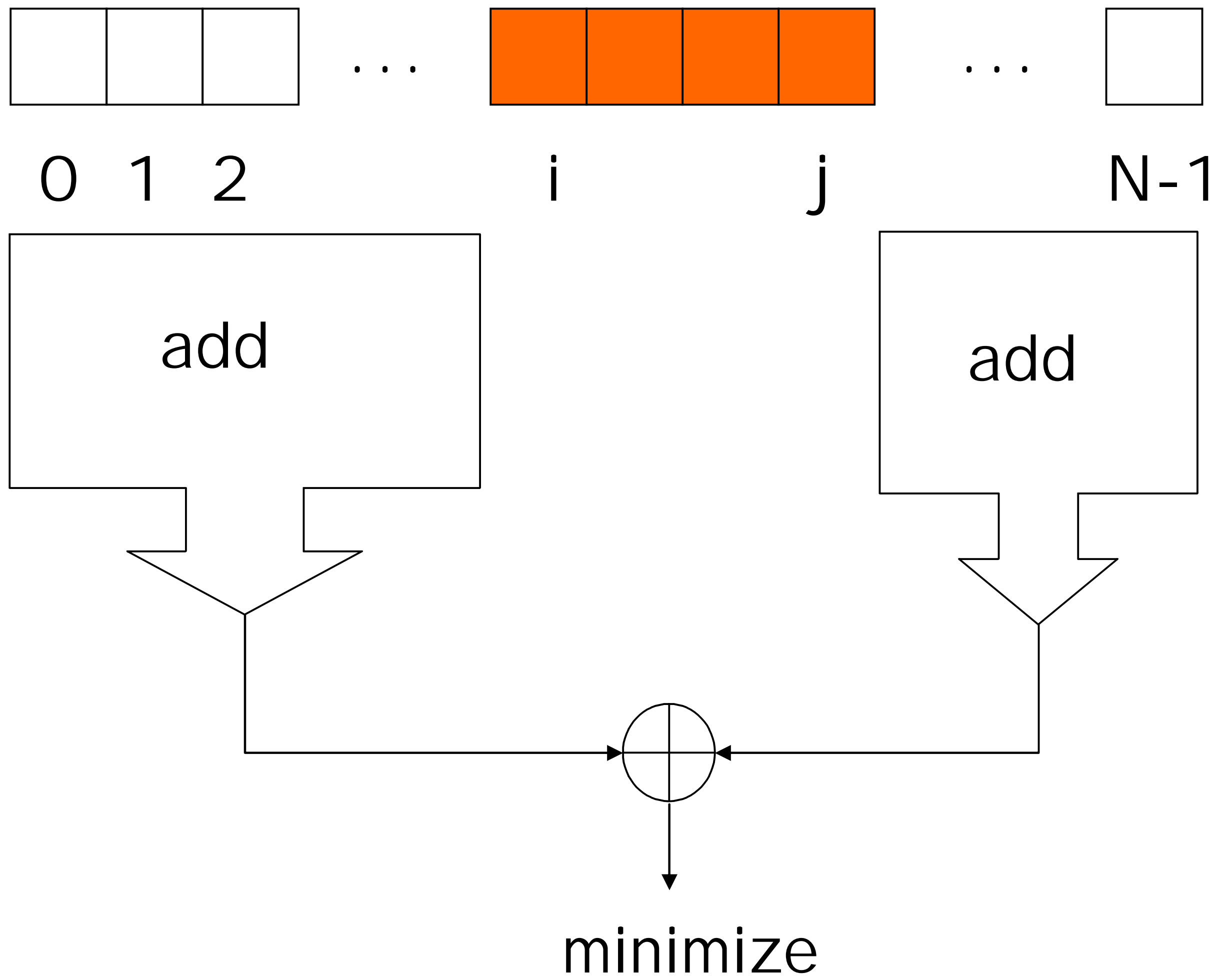
Θ(n logn)

```
public static int maxSubSum3( int[] a ) {
  return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
}
private static int maxSumRec( int [ ] a,
                              int left, int right ) {
  int center = ( left + right ) / 2;

  if( left == right )                          // "Base case"
    return a[ left ] > 0 ? a[ left ] : 0;

  return max3( maxSumRec(a, left, center),       // REC. CALL
               maxAcrossCenter(a, left, right), // next slide
               maxSumRec(a, center+1, right) ); // REC. CALL
}
private static int max3( int a, int b, int c ) {
  return a > b ? a > c ? a : c : b > c ? b : c;
                                        // max of three ints
}
```

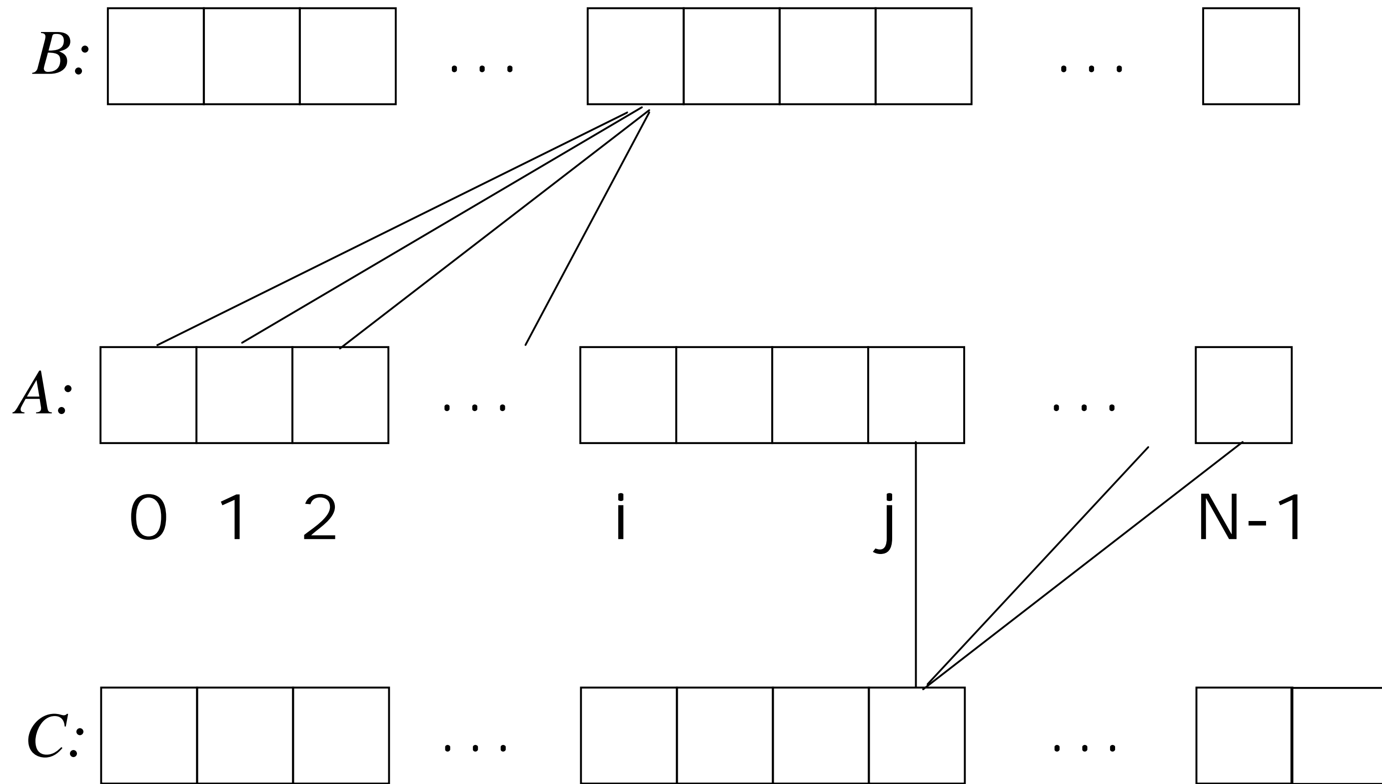
Program 3, cont'd

```
private static int maxAcrossCenter( int[] a,
                                    int left, int right )  {
  int center = ( left + right ) / 2;

  int maxLeft = 0;
  for( int sum=0, int i = center; i >= left; i-- ) {
    sum = sum + a[i];
    if( sum > maxLeft ) maxLeft = sum;
  }

  int maxRight = 0;
  for( int sum=0, int i = center + 1; i <= right; i++ ) {
    sum = sum + a[i];
    if( sum > maxRight ) maxRight = sum;
  }
  return maxLeft + maxRight;
}
```

Solution 4: an entirely different perspective



$$B[0] = 0;\ \ B[i] = \sum_{k=0}^{i-1} A[k]$$



$$C[N] = 0;\ \ C[j] = \sum_{k=j}^{N-1} A[k]$$

To solve the original problem: Find indices

$$i, j : i < j \text{ and } B[i] + C[j] \text{ is minimized.}$$

So, for given $B[i]$:

Need to find least $C[j]$ such that $i < j$.

Compute two new arrays D and E:

$$D[i] = \min_{0 \le k \le i} B[k]$$

$$E[j] = \min_{j < k \le N} C[k]$$

Now, the best solution is obtained by finding:

$$i : D[i] + E[i] \text{ is minimized.}$$

Analysis:

Looks complicated but actually the algorithm is fast.

Each array takes $\Theta(N)$ steps to compute.

Finding i: D[i]+E[i] is minimum takes another $\Theta(N)$steps.

From this information, finding the best interval takes $\Theta(N)$ steps.

Total time $\Theta(N)$

(The book gives another $\Theta(N)$ algorithm (a 5th solution!) which is slicker but perhaps harder to understand.)

Program 4

```
public static int maxSubSum4( int[] a)   {
  int n = a.length;
  int total = 0;
  for (int i =0; i < n; i++)
    total = total + a[i];
  int[] b = new int[n];
  int[] c = new int[n+1];

  for (b[0]=0, int i=1; i < n; i++)
    b[i] = b[i-1] + a[i-1];

  for (c[n]=0, int j=n-1; j != 0; j--)
    c[j] = c[j+1] + a[j];
  ... continued
```

$\Theta(n)$

Program 4, cont'd

```
...
  int[] d = new int[n];
  int[] e = new int[n];

  for (d[0]=0, int i=1; i < n; i++)
    d[i] = (d[i-1] < b[i]) ? d[i-1] : b[i];

  for (e[n-1]=c[n-1], int j=n-2; j != 0; j--)
    e[j] = (e[j+1] < c[j]) ? e[j+1] : c[j];

  int minSum = d[0] + e[0];
  for (int i=1; i < n; i++)
    if (d[i]+e[i] < minSum)
      minSum = d[i]+e[i];

  return total - minSum;
}
```