Asymptotic Behavior

Algorithm : Design & Analysis

[2]

In the last class…

- Goal of the Course
- Algorithm: the Concept
- Algorithm Analysis: the Contents
- Average and Worst-Case Analysis
- Lower Bounds and the Complexity of Problems

Asymptotic Behavior

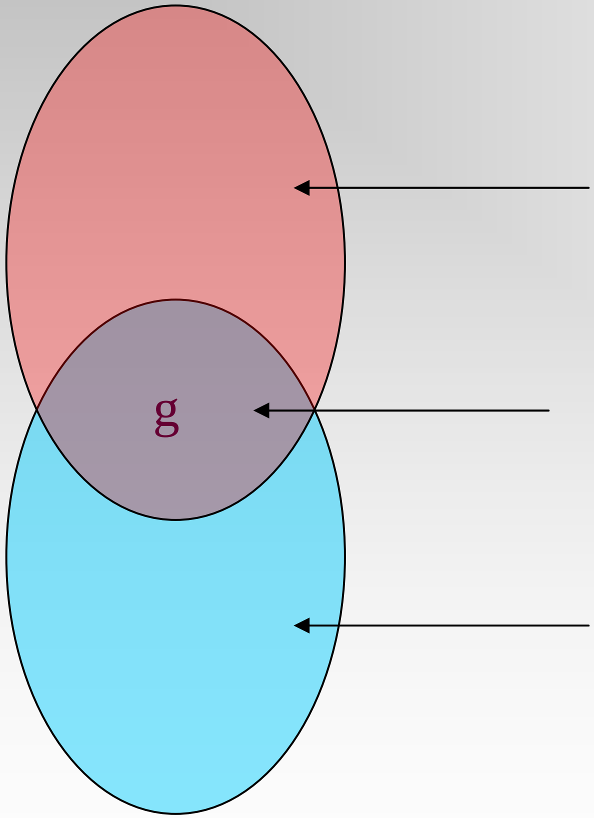
- Asymptotic growth rate
- The Sets O, Ω and Θ
- Complexity Class
- An Example: Maximum Subsequence Sum
  - Improvement of Algorithm
  - Comparison of Asymptotic Behavior
- Another Example: Binary Search
  - Binary Search Is Optimal

How to Compare Two Algorithm?

- Simplifying the analysis
  - assumption that the total number of steps is roughly proportional to the number of basic operations counted (a constant coefficient)
  - only the leading term in the formula is considered
  - constant coefficient is ignored

- Asymptotic growth rate
  - large n vs. smaller n

Relative Growth Rate



Ω (g):functions that grow at least as fast as g

Θ (g):functions that grow at the same rate as g

O (g):functions that grow no faster as g

The Set "Big Oh"

- Definition
  - Giving g:N→R$^+$, then $O(g)$ is the set of f:N→R$^+$, such that for some $c\in R^+$ and some $n_0\in N$, $f(n)\le cg(n)$ for all $n\ge n_0$.
- A function $f\in O(g)$ if $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c<\infty$
  - Note: c may be zero. In that case, $f\in o(g)$, "little Oh"

Example

Using L'Hopital's Rule

- Let $f(n)=n^2$, $g(n)=n\lg n$, then:
  - $f \notin O(g)$, since $\lim_{n\to\infty}\frac{n^2}{n\lg n}=\lim_{n\to\infty}\frac{n}{\lg n}=\lim_{n\to\infty}\frac{n}{\frac{\ln n}{\ln 2}}=\lim_{n\to\infty}\frac{1}{\frac{1}{n\ln 2}}=\infty$
  - $g \in O(f)$, since $\lim_{n\to\infty}\frac{n\log n}{n^2}=\lim_{n\to\infty}\frac{\log n}{n}=\lim_{n\to\infty}\frac{\ln n}{n\ln 2}=\lim_{n\to\infty}\frac{1}{n\ln 2}=0$

For your reference: L'Hôpital's rule

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$

with some constraints

Logarithmic Functions and Powers

Which grows faster?

$$\log_2 n \quad \text{or} \quad \sqrt{n}\ ?$$

So, $\log_2 n \in O(\sqrt{n})$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{\log_2 e}{n}}{\frac{1}{2\sqrt{n}}} = (2\log_2 e)\lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0$$

The Result Generalized

- The log function grows more slowly than ***any*** positive power of *n*

$$\lg n \in o(n^{\alpha}) \text{ for any } \alpha > 0$$

By the way:

The power of *n* grows more slowly than any exponential function with base greater than 1

$$n^{k} \in o(c^{n}) \text{ for any } c > 1$$

Dealing with big-O correctly

We have known that : $\log n \in o(n^{0.0001})$

$$(\text{since } \lim_{n\to\infty} \frac{\log n}{n^{\varepsilon}} = 0 \ \text{ for any } \varepsilon > 0)$$

However, which is larger : $\log n$ and n^{ε}, if $n = 10^{100}$?

Factorials and Exponential Functions

- $n!$ grows faster than 2^n for positive integer n.

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\dfrac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty$$

$$\text{Stirling' s formular}\ : n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

The Sets Ω and Θ

- Definition
  - Giving g:N→R$^+$, then $\Omega(g)$ is the set of f:N→R$^+$, such that for some c∈R$^+$ and some n_0∈N, $f(n) \geq cg(n)$ for all $n \geq n_0$.
- A function $f \in \Omega(g)$ if $\lim_{n\to\infty}[f(n)/g(n)]>0$
  - Note: the limit may be infinity
- Definition
  - Giving g:N→R$^+$, then $\Theta(g) = O(g) \cap \Omega(g)$
- A function $f \in \Theta(g)$ if $\lim_{n\to\infty}[f(n)/g(n)]=c, 0<c<\infty$

Properties of O, Ω and Θ

- Transitive property:
  - If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
- Symmetric properties
  - $f \in O(g)$ if and only if $g \in \Omega(f)$
  - $f \in \Theta(g)$ if and only if $g \in \Theta(f)$
- Order of sum function
  - $O(f+g)=O(\max(f,g))$

Complexity Class

- Let S be a set of f:N→R* under consideration, define the relation ~ on S as following: $f \sim g$ iff. $f \in \Theta(g)$

  then, ~ is an equivalence.
- Each set $\Theta(g)$ is an equivalence class, called complexity class.
- We usually use the simplest element as possible as the representative, so, $\Theta(n)$, $\Theta(n^2)$, etc.

Effect of the Asymptotic Behavior

algorithm		1	2	3	4
Run time in *ns*		$1.3n^3$	$10n^2$	$47n\log n$	$48n$
time for size	10^3 10^4 10^5 10^6 10^7	1.3*s* 22*m* 15*d* 41*yrs* 41*mill*	10*ms* 1*s* 1.7*m* 2.8*hrs* 1.7*wks*	0.4*ms* 6*ms* 78*ms* 0.94*s* 11*s*	0.05*ms* 0.5*ms* 5*ms* 48*s* 0.48*s*
max Size in time	sec min hr day	920 3,600 14,000 41,000	10,000 77,000 6.0×10^5 2.9×10^6	1.0×10^6 4.9×10^7 2.4×10^9 5.0×10^{10}	2.1×10^7 1.3×10^9 7.6×10^{10} 1.8×10^{12}
time for 10 times size		×1000	×100	×10+	×10

on 400Mhz Pentium II, in C

from: Jon Bentley: *Programming Pearls*

Searching an Ordered Array

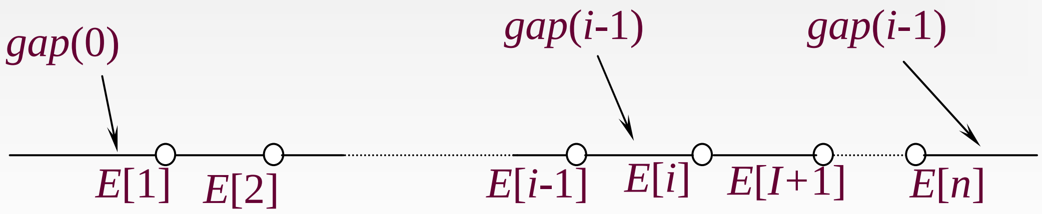
- The Problem: Specification
  - Input:
    - an array E containing n entries of numeric type sorted in non-decreasing order
    - a value K
  - Output:
    - *index* for which $K=E[index]$, if K is in E, or, -1, if K is not in E

Sequential Search: the Procedure

- The Procedure
  - **Int** seqSearch(**int**[] E, **int** n, **int** K)
  - 1. **Int** ans, index;
  - 2. Ans=-1; // Assume failure
  - 3. **For** (index=0; index<n; index++)
  - 4. **If** (K= =E[index]) ans=index;//success!
  - 5. **break**;
  - 6. **return** ans

Searching a Sequence

- For a given *K*, there are two possibilities
  - *K* in *E* (say, with probability *q*), then *K* may be any one of *E*[*i*] (say, with equal probability, that is 1/n)
  - *K* not in *E* (with probability 1-*q*), then *K* may be located in any one of *gap*(i) (say, with equal probability, that is 1/(n+1))



Improved Sequential Search

- Since E is sorted, when an entry larger than K is met, no nore comparison is needed
  - Worst-case complexity: n, unchanged
  - Average complexity

$$A(n) = qA_{succ}(n) + (1-q)A_{fail}(n)$$

$$A_{succ}(n) = \sum_{i=0}^{n-1}\left(\frac{1}{n}\right)(i+1) = \frac{n+1}{2}$$

$$A_{fail}(n) = \sum_{i=0}^{n-1}\left(\frac{1}{n+1}\right)(i+1) + \left(\frac{1}{n+1}\right)n = \frac{n}{2} + \frac{n}{n+1}$$

$$A(n) = \frac{q(n+1)}{2} + (1-q)\left(\frac{n}{2} + \frac{n}{n+1}\right)$$

$$= \frac{n}{2} + \left(\frac{n}{n+1} + q\left(\frac{1}{2} - \frac{n}{n+1}\right)\right) = \frac{n}{2} + O(1)$$

Roughly $n/2$

Note: $A(n) \in \Theta(n)$

Divide and Conquer

- If we compare *K* to every *j*th entry, we can locate the small section of *E* that may contain *K*.
  - To locate a section, roughly n/j steps at most
  - To search in a section, j steps at most
  - So, the worst-case complexity: (n/j)+j, with j selected properly, $(n/j)+j \in \Theta(\sqrt{n})$
- However, we can use the same strategy in the small sections recursively

Choose $j = \sqrt{n}$

Binary Search

```
int binarySearch(int[] E, int first, int last, int K)
    if (last<first)
        index=-1;
    else
        int mid=(first+last)/2
        if (K==E[mid])
            index=mid;
        else if (K<E[mid])
            index=binarySearch(E, first, mid-1, K)
        else if (K<E[mid])
            index=binarySearch(E, mid+1, last, K)
    return index;
```

Worst-case Complexity of Binary Search

- Observation: with each call of the recursive procedure, only at most half entries left for further consideration.
- At most $\lfloor \lg n \rfloor$ calls can be made if we want to keep the range of the section left not less than 1.
- So, the worst-case complexity of binary search is $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$

Average Complexity of Binary Search

- Observation:
  - for most cases, the number of comparison is or is very close to that of worst-case
  - particularly, if $n=2^k-1$, all failure position need exactly k comparisons
- Assumption:
  - all success position are equally likely (1/n)
  - $n=2^k-1$

Aver

- Avera
  - No
    fo
    s_t=

For your reference : Arithmetic - Geometric Series

$$\sum_{i=1}^{k} i2^i = \sum_{i=1}^{k} i\left(2^{i+1} - 2^i\right)$$

$$= (2^2 + 2\cdot 2^3 + 3\cdot 2^4 + \cdots + (k-1)\cdot 2^k + k\cdot 2^{k+1})$$

$$- (2 + 2\cdot 2^2 + 3\cdot 2^3 + \cdots + (k-1)\cdot 2^{(k-1)} + k\cdot 2^k)$$

$$= (k\cdot 2^{k+1} - 2) - \sum_{i=2}^{k} 2^i = (k\cdot 2^{k+1} - 2) - (2^{k+1} - 4)$$

$$= (k-1)\cdot 2^{k+1} + 2$$

$A_q($

$A_1(n$

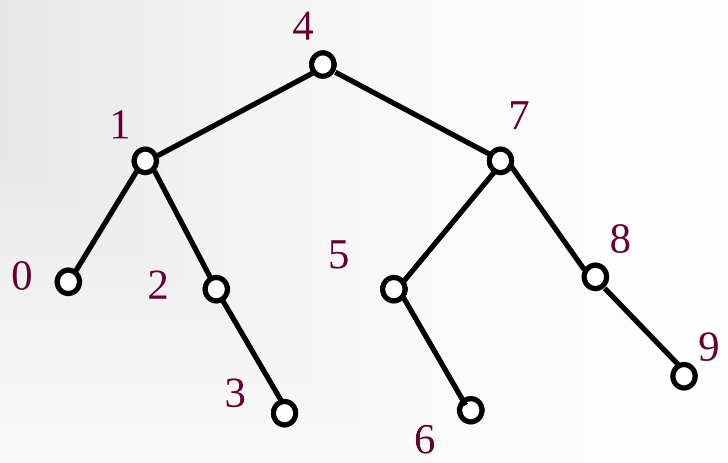
$$\frac{(k-1)(n+1)+1}{n} = \lg(n+1) - 1 + O\left(\frac{\log n}{n}\right)$$

$$A_0 = \lg(n+1)$$

$$A_q(n) = \lg(n+1) - q$$

Decision Tree

- A decision tree for *A* and a given input of size *n* is a binary tree whose nodes are labeled with numbers between 0 and *n*-1
  - Root: labeled with the index first compared
  - If the label on a particular node is *i*, then the left child is labeled the index next compared if $K<E[i]$, the right child the index next compared if $K>E[i]$, and no branch for the case of $K=E[i]$.



Binary Search Is Optimal

- If the number of comparison in the worst case is p, then the longest path from root to a leaf is p-1, so there are at most 2^p-1 node in the tree.
- There are at least n node in the tree.

  (We can prove that For all $i \in \{0,1,\ldots,n\text{-}1\}$, exist a node with the label i.)
- Conclusion: $n \leq 2^p-1$, that is $p \geq \lg(n+1)$

Binary Search Is Optimal

- For all $i \in \{0,1,\ldots,n\text{-}1\}$, exist a node with the label i.
- Proof:
  - if otherwise, suppose that i doesn't appear in the tree, make up two inputs E_1 and E_2, with $E_1[i]=K$, $E_2[i]=K'$, $K'>K$, for any $j \neq i$, $(0 \leq j \leq n\text{-}1)$, $E_1[j]=E_2[j]$. **(Arrange the values to keeping the order in both arrays)**. Since i doesn't appear in the tree, for both K and K', the algorithm behave alike exactly, and give the same outputs, of which at least one is wrong, so A is not a right algorithm.

Home Assignment

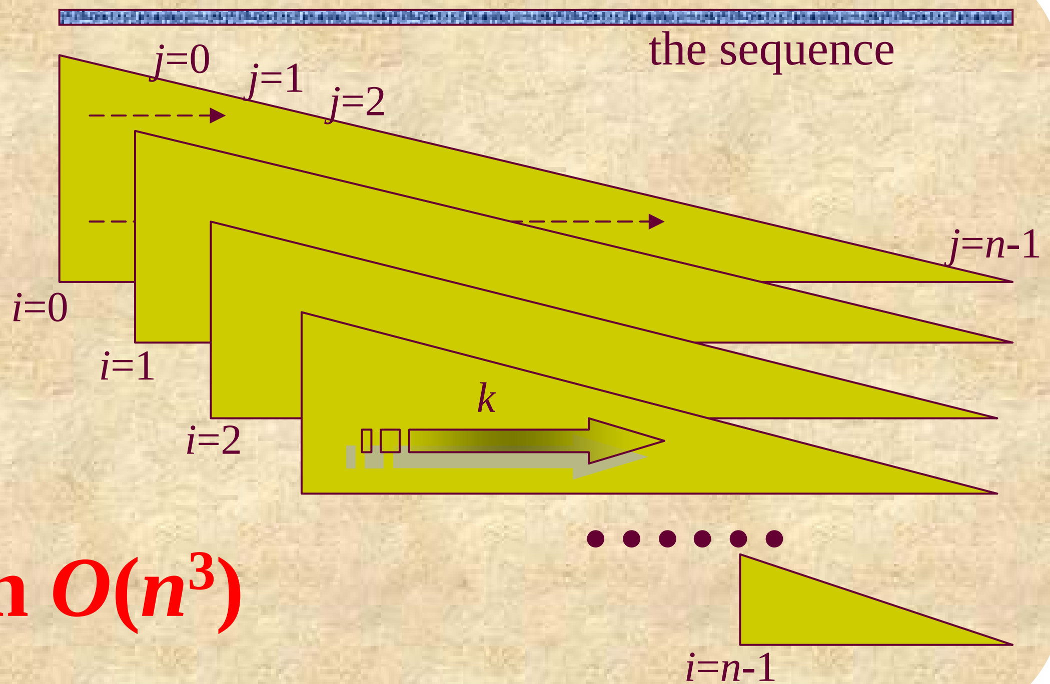
- pp.63 –
  - 1.23
  - 1.27
  - 1.31
  - 1.34
  - 1.37
  - 1.45

Maximum Subsequence Sum

- The problem: Given a sequence *S* of integer, find the **largest sum** of a consecutive subsequence of *S*. (0, if all negative items)
  - An example: -2, 11, -4, 13, -5, -2; the result 20: (11, -4, 13)

A brute-force algorithm:

```
MaxSum = 0;
 for (i = 0; i < N; i++)
  for (j = i; j < N; j++)
  {
   ThisSum = 0;
   for (k = i; k <= j; k++)
   ThisSum += A[k];
   if (ThisSum > MaxSum)
    MaxSum = ThisSum;
  }
 return MaxSum;
```



in $O(n^3)$

More Precise Complexity

The total cost is : $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1$

$$\sum_{k=i}^{j} 1 = j - i + 1$$

$$\sum_{j=i}^{n-1} (j - i + 1) = 1 + 2 + \ldots + (n - i) = \frac{(n - i + 1)(n - i)}{2}$$

$$\sum_{i=0}^{n-1} \frac{(n - i + 1)(n - i)}{2} = \sum_{i=1}^{n} \frac{(n - i + 2)(n - i + 1)}{2}$$

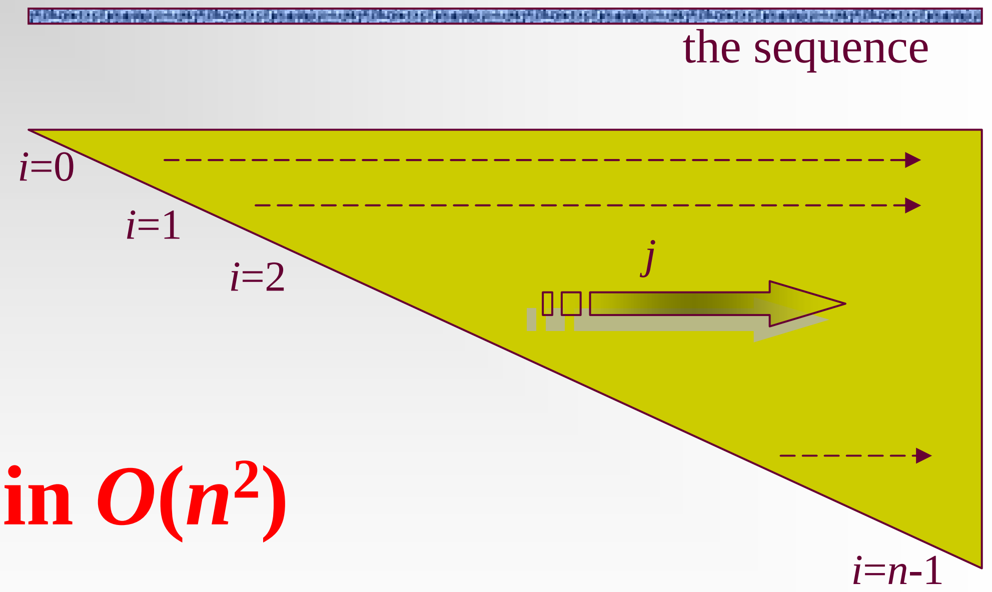
$$= \frac{1}{2}\sum_{i=1}^{n} i^2 - (n + \frac{3}{2})\sum_{i=1}^{n} i + \frac{1}{2}(n^2 + 3n + 2)\sum_{i=1}^{n} 1$$

$$= \frac{n^3 + 3n^2 + 2n}{6}$$

Decreasing the number of loops

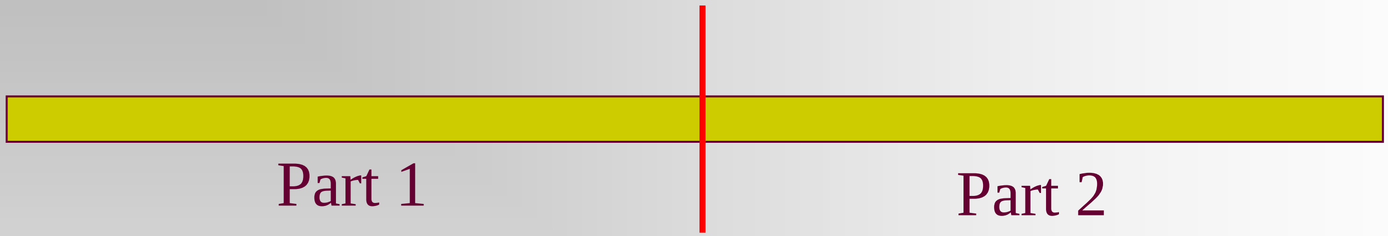
An improved algorithm

```
MaxSum = 0;
 for (i = 0; i < N; i++)
 {
  ThisSum = 0;
  for (j = i; j < N; j++)
  {
   ThisSum += A[j];
   if (ThisSum > MaxSum)
    MaxSum = ThisSum;
  }
 }
 return MaxSum;
```

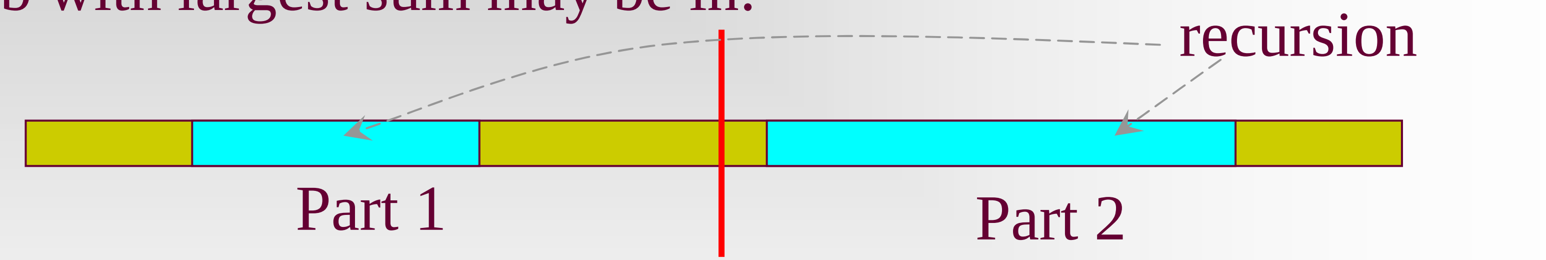


in $O(n^2)$

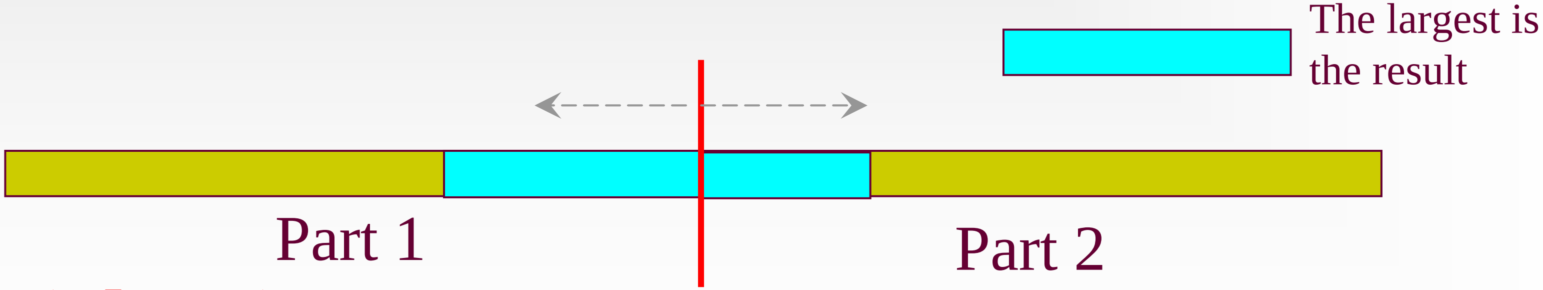
Power of Divide-and-Conquer



the sub with largest sum may be in:



or:



in *O*(*n*log*n*)

Divide-and-Conquer: the Procedure

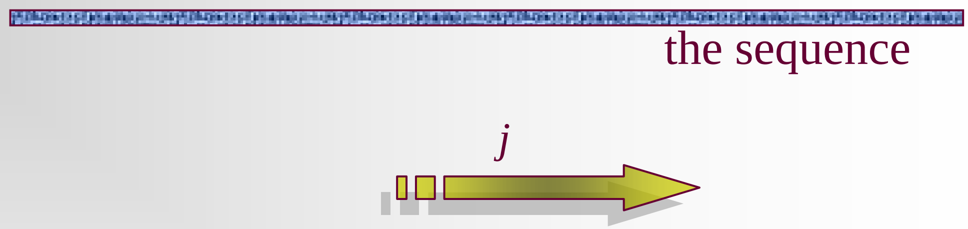
```
Center = (Left + Right) / 2;
 MaxLeftSum = MaxSubSum(A, Left, Center);  MaxRightSum = MaxSubSum(A, Center + 1, Right);

 MaxLeftBorderSum = 0; LeftBorderSum = 0;
 for (i = Center; i >= Left; i--)
 {
   LeftBorderSum += A[i];
   if (LeftBorderSum > MaxLeftBorderSum)    MaxLeftBorderSum = LeftBorderSum;
 }

 MaxRightBorderSum = 0; RightBorderSum = 0;
 for (i = Center + 1; i <= Right; i++)
 {
   RightBorderSum += A[i];
   if (RightBorderSum > MaxRightBorderSum)  MaxRightBorderSum = RightBorderSum;
 }
 return Max3(MaxLeftSum, MaxRightSum,
        MaxLeftBorderSum + MaxRightBorderSum);
```

Note: this is the core part of the procedure, with base case and wrap omitted.

A Linear Algorithm

```
ThisSum = MaxSum = 0;
  for (j = 0; j < N; j++)
  {
   ThisSum += A[j];
   if (ThisSum > MaxSum)
     MaxSum = ThisSum;
   else if (ThisSum < 0)
     ThisSum = 0;
  }
  return MaxSum;
```

in $O(n)$



This is an example of "online algorithm"

Negative item or subsequence cannot be a prefix of the subsequence we want.