CSCI 362 : Data Structure

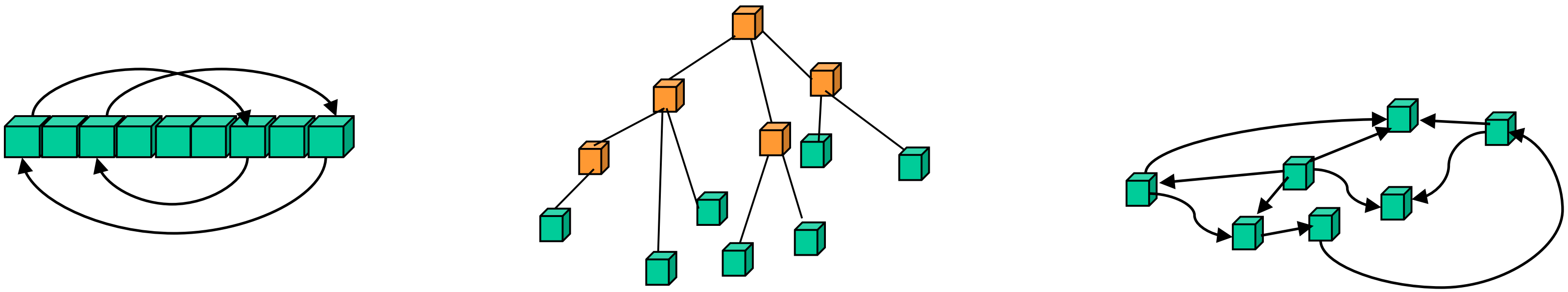
Spring, 2015

Part I: Data Structures & Algorithm Analysis

Chap. 1 Introduction

1.1 Introduction

- Data Structures: methods of organizing large amount of data (input data, output data, run-time data)

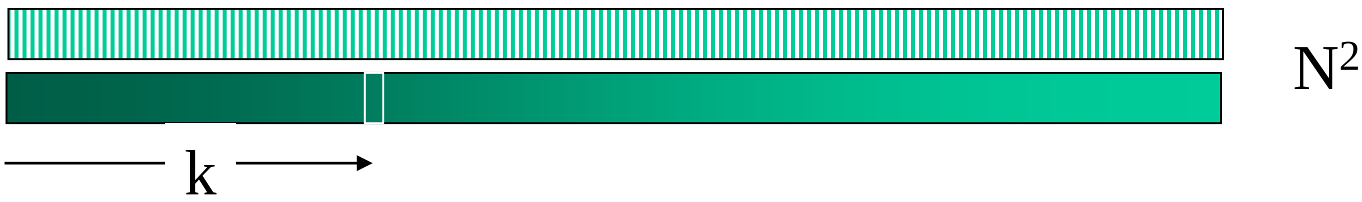


- Algorithm analysis: estimation of running time of algorithms
- Goals
  - Various data structures, their properties, operations, implementations, & applications
  - Data structure design skills for problem solving
  - Algorithm design and analysis skill

– Algorithm analysis examples

- Finding the k^{th} largest number among a group of N numbers

Bubble sort



http://en.wikipedia.org/wiki/Bubble_sort#mediaviewer/File:Bubble-sort-example-300px.gif

1.2 Mathematics review

– Exponents

$$X^A X^B = X^{A+B} \qquad (X^A)^B = X^{AB}$$

$$X^A / X^B = X^{A-B}$$

– Logarithms

$$X^A = B \Leftrightarrow \log_X B = A \qquad (B, X > 0, X \neq 1)$$

$$\log_A B = \log_C B / \log_C A$$

$$\log AB = \log A + \log B$$

$$\log A / B = \log A - \log B$$

$$\log(A^B) = B \log A$$

$$\log X < X \qquad (\text{for all } X > 0)$$

$$\log 1 = 0, \log 2 = 1, \log 1024 = 10, \ldots$$

$$\boxed{\begin{aligned} \log_2 A &= \log A \\ \log_{10} A &= \lg A \\ \log_e A &= \ln A \end{aligned}}$$

– Series $$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}, \quad \sum_{0}^{N} 2^i = 2^{N+1}-1$$

$$\text{If } 0 < A < 1 \;\Rightarrow\; \sum_{0}^{N} A^i \le \frac{1}{1-A}$$

$$\sum_{i=0}^{\infty} A^i = \frac{1}{1-A} \quad (0 < A < 1): \text{ geometric series}$$

$$\sum_{1}^{N} i = \frac{N(N+1)}{2} \approx N^2/2$$

$$\sum_{1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx N^3/3$$

ex. $2+5+8+\ldots+(2k-1)$

$$\sum_{1}^{N} i^k \approx N^{k+1} \big/ |k+1| \quad (k \ne -1)$$

Harmonic number

Euler's Constant:

$$H_N = \sum_{1}^{N} \frac{1}{i} \approx \log_e N = \ln N$$

$$\gamma = \lim_{N\to\infty} |H_N - \ln N| = 0.57721566$$

$$\sum_{i=1}^{N} f(N) = Nf(N)$$

$$\sum_{i=n_0}^{N} f(N) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{n_0-1} f(i)$$

– Modular Arithmetic

- A is congruent to B modulo N $\Leftrightarrow A \equiv B \pmod{N}$

$\Leftrightarrow N$ divides A - B

e.g. $81 \equiv 61 \equiv 1 \pmod{10}$

- If $A \equiv B \pmod{N}$ then $A + C \equiv B + C \pmod{N}$

and $AD \equiv BD \pmod{N}$

– Mathematical proof techniques

- Deriving some statement from

(a) Assumption or hypothesis

(b) Statement that are already derived

(c) Other generally accepted facts

- Proof by construction

  Example: for any integers a & b, if a, b are odd, then ab is also odd.

- Proof by contradiction

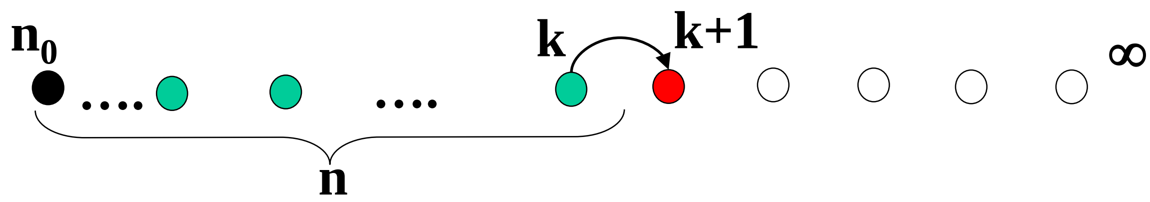
  Example: There is an infinite number of primes. $N=P_1P_2\ldots.P_n+1$

- Proof by counterexample

  Example: $F_k \le k^2$, True?

– Mathematical induction

- Suppose $P(n)$ is a statement involving an integer *n*, then to prove that $P(n)$ is true for every $n \ge n_0$, it is sufficient to show:
  - (a) $P(n_0)$ is true **(base case)**
  - (b) For any $k \ge n_0$, if $P(n)$ is true for any $n_0 \le n \le k$ **(inductive hypothesis)**, then $P(k+1)$ is also true **(induction)**.



Example:

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

Example:

Fibonacci number are defined as:

$$F_0=1,\ F_1=1, \ldots \quad F_k=F_{k-1}+F_{k-2}$$

Prove: $F_i < (5/3)^i$

Example:

for any integer $n \geq 2$, n is either a prime or a product of two or more primes.

1.3 Recursion

- Defining a function in terms of itself
- Fundamental rules of recursion
  1.) Base cases
  2.) Making progress through recursion
  3.) Design rule: assuming all recursive call work (details hidden)
  4.) Compound interest rule: do not duplicate recursive calls
- Fibonacci number is a recursive function

$$F_k=F_{k-1}+F_{k-2}, \quad F_0=1, F_1=1$$

Example:

$$f(x) = 2f(x-1)+x^2, \quad f(0)=0;$$

Example:

$$n! = \begin{cases} 1 & n=0 \\ n(n-1)! & n>0 \end{cases}$$

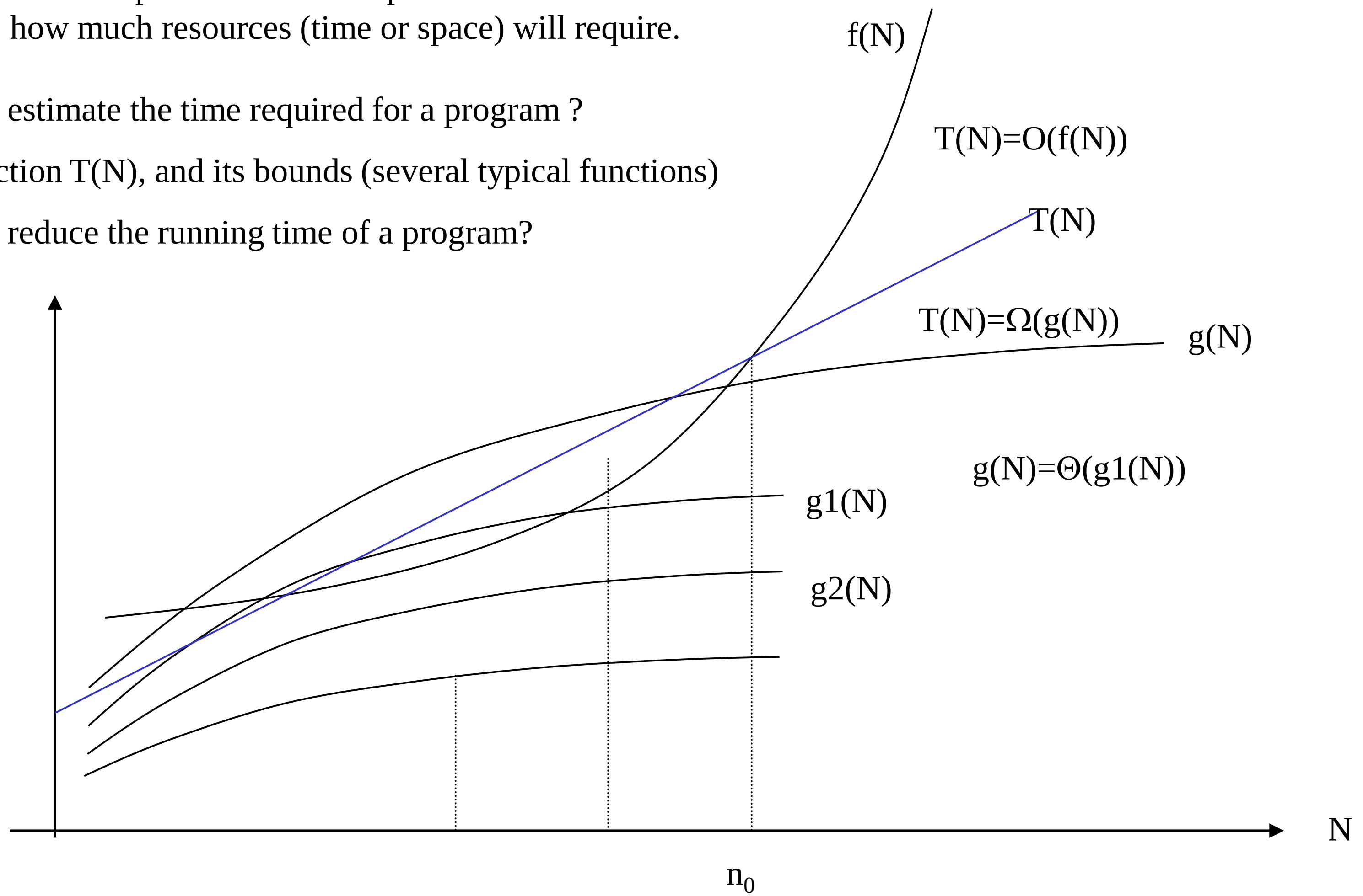
Algorithm Complexity

- An algorithm is specified set of simple instructions
- Totally, how much resources (time or space) will require.

How to estimate the time required for a program ?

→ function T(N), and its bounds (several typical functions)

How to reduce the running time of a program?



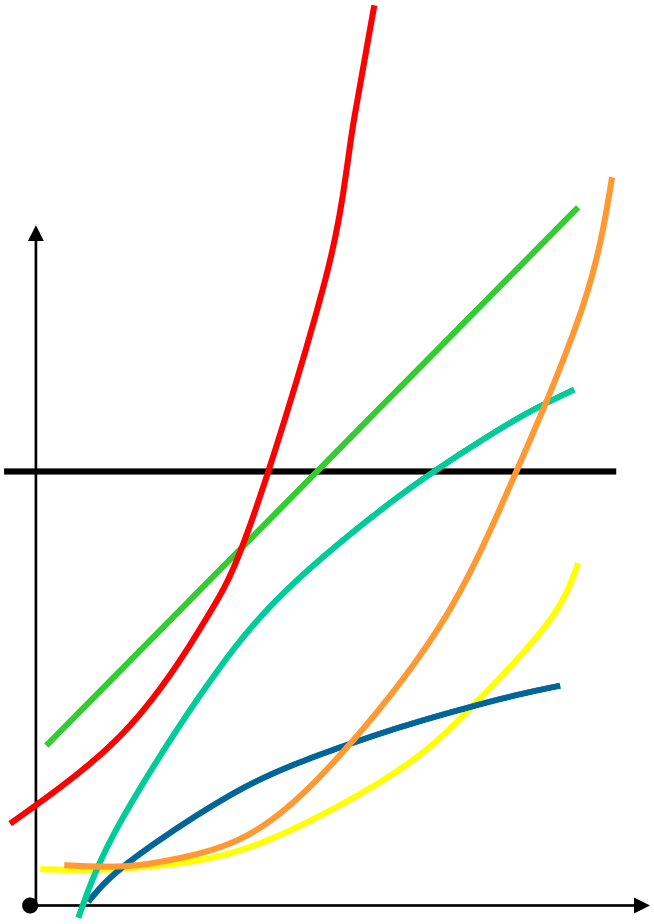
Chap. 2 Algorithm Analysis

2.1 Math. Background

- Some definitions on Computation Order
  - $T(N)=O(f(N))$ if there is constant $c > 0$ and $n_0 > 0$ such that $T(N) \leq c\, f(N)$ when $N \geq n_0$
  - $T(N)=\Omega(g(N))$: if there is constant $c > 0$ and $n_0 > 0$ such that $T(N) \geq cg(N)$ when $N \geq n_0$
  - $T(N)=\Theta(h(N))$: if and only if $T(N)= O(h(N))$ and $T(N)= \Omega(h(N))$
  - $T(N)=o(p(N))$: if $T(N)= O(p(N))$ and $T(N)\neq\Theta(p(N))$
- Function growth rate (relative rate of growth or computation order)
  - establish a relative order among functions
  - only interested in large N. ($N \geq n_0$)
  - constant factor is ignored $Cf(N) = O(f(N))$, $C+f(N) = O(f(N))$
  - $T(N)=O(f(N))$: f is an upper bound on $T(N)$, and T is a lower bound on f (i.e. $f(N)=\Omega(T(N))$)

Typical growth rates bounding running time

- Function Name
  - 2^N Exponential
  - N^3 Cubic
  - N^2 Quadratic
  - NlogN NLogN
  - N Linear
  - $\log^2 N$ Log-squared
  - logN Logarithmic
  - C constant



– Computing growth rates

- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$
  - a) $T_1(N) + T_2(N) = O(max(f(N), g(N)))$
  - b) $T_1(N) * T_2(N) = O(f(N) * g(N))$
- If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$
- $\log^k N = O(N)$, for any constant k.
- Minimal upper-bound and maximal lower-bound
- Let $L = \lim_{N \to \infty} f(N)/g(N)$

  If $L=0$ then $f(N)=o(g(N))$ (also $O(g(N))$)

  If $L \neq 0$ then $f(N) = \Theta(g(N))$

  If $L = \infty$ then $g(N)=o(f(N))$

  If L doesn't exist, then $f(N)$ and $g(N)$ do not have a relationship

Example: $f(N) = N \log N, \quad g(N) = N^{1.5}$

$$\lim_{N\to\infty}\frac{f(N)}{g(N)} = \lim_{N\to\infty}\frac{N\log N}{N\sqrt{N}} = \lim_{N\to\infty}\frac{\log N}{\sqrt{N}} = \lim_{N\to\infty}\frac{1/N}{\frac{1}{2}N^{-\frac{1}{2}}} = 0$$

$$\Rightarrow f(N) = o(g(N)) \text{ and } f(N) = O(g(N))$$

$$\text{or} \quad N = O(N), \quad \log N = O(\sqrt{N})$$

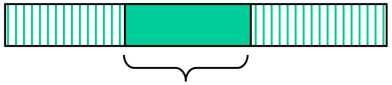
$$\Rightarrow N\log N = O(N^{1.5})$$

2.2 Computational Model

- Turing machine based
- Simple operations (+, -, X, /, =, ==,…)
- All operations have the same execution time (one unit per operation)
- Sequential computation
- Ignore complex issues such as memory paging, caching,I/O, compiler, etc.

2.3 Analysis

- Running time analysis
  - $T_{worst}(N)$: worst case analysis
  - $T_{avg}(N)$: average case analysis
  - The implementation of the algorithm should not change the big Os.
- Example: Maximum subsequence sum problem
  - The problem: given integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$



  - Four algorithms with different running time estimates:

    $O(N^3), O(N^2), O(N \log N), O(N)$

    (Textbook, Weiss, pp.46)

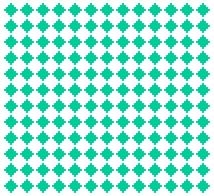
2.4 Running Time Calculations

– General rules

- *for loop*: at most the running time of the loop body times the number of iterations
- *nested loop*: the running time of the loop body multiplied by the product of the sizes of all the loops

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    k++
```

$\Rightarrow O(N^2)$



- *consecutive statement*: adding their running times

```
for (i=0; i<n; i++)
  a[I]=0;
```

$\Rightarrow O(N)$

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[I]+=a[j]+I+j;
```

$\Rightarrow O(N^2)$

$\Rightarrow O(N^2)$

- *If/else*: at most the running time of test plus the larger of the running times of the two branches

```
if (condition)
  S1
else
  S2
```

– Recursion

- convert to loops if possible
- running time is a recursive function
- example:

```
long factorial(int n) {
if (n <= 1)
   return 1;
else
   return n * factorial(n-1); }
```

- example:

$$f(n) = \begin{cases} 1 & n \le 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

```
long fib(int n) {
   if (n <= 1)
      return 1;
   else
      return fib(n-1)+ fib(n-2);}
```

T(N)=T(N-1)+T(N-2)+2

Fib(N) <$(5/3)^N$

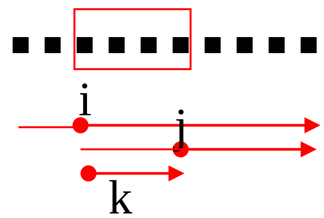
– Solutions for the maximum subsequence sum problem

- Algr. 1: exhaustive searching `(Fig. 2.5, page 50)`

  $\Rightarrow$ running time $T(N) = O(N^3)$
- Algr. 2: `(Fig. 2.6, page 52)` $\Rightarrow T(N) = O(N^2)$
- Algr. 3: `(Fig. 2.7, page 53)` $\Rightarrow T(N) = O(N log N)$
- Algr. 4: `(Fig. 2.8, page 55)` $\Rightarrow T(N) = O(N)$

```
#include <iostream.h>
#include "vector.h"
/* START: Fig02_05.txt */
/* Cubic maximum contiguous
subsequence sum algorithm.*/
      int maxSubSum1( const vector<int> & a )
      {
/* 1*/      int maxSum = 0;
```

$O(N^3)$

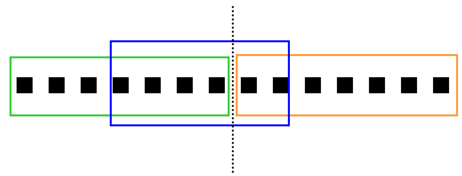



```
/* 2*/      for( int i = 0; i < a.size( ); i++ )
/* 3*/          for( int j = i; j < a.size( ); j++ )
            {
/* 4*/              int thisSum = 0;
/* 5*/              for( int k = i; k <= j; k++ )
/* 6*/                  thisSum += a[ k ];
/* 7*/              if( thisSum > maxSum )
/* 8*/                  maxSum = thisSum;
            }
/* 9*/      return maxSum;
      }
```

```
/* START: Fig02_06.txt       Quadratic maximum
contiguous subsequence sum algorithm. */
int maxSubSum2( const vector<int> & a )
      {
/* 1*/      int maxSum = 0;
/* 2*/      for( int i = 0; i < a.size( ); i++ )
          {
/* 3*/          int thisSum = 0;
/* 4*/          for( int j = i; j < a.size( ); j++ )
              {
/* 5*/              thisSum += a[ j ];
/* 6*/              if( thisSum > maxSum )
/* 7*/                  maxSum = thisSum;
              }
          }
/* 8*/      return maxSum;
      }
/** Return maximum of three integers.*/
      int max3( int a, int b, int c )
      {      return a > b ? a > c ? a : c : b > c ? b : c;
      }
```

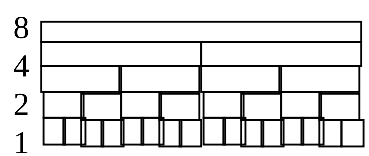
$O(N^2)$

```
/* START: Fig02_07.txt */

    /*** Recursive maximum contiguous subsequence sum algorithm.
      Finds maximum sum in subarray spanning a[left..right].* Does not
      attempt to maintain actual best sequence. */

    int maxSumRec( const vector<int> & a, int left, int right )     {
/* 1*/     if( left == right )  // Base case
/* 2*/        if( a[ left ] > 0 )
/* 3*/            return a[ left ];
           else
/* 4*/            return 0;
/* 5*/     int center = ( left + right ) / 2;
/* 6*/     int maxLeftSum  = maxSumRec( a, left, center );
/* 7*/     int maxRightSum = maxSumRec( a, center + 1, right );
/* 8*/     int maxLeftBorderSum = 0, leftBorderSum = 0;
/* 9*/     for( int i = center; i >= left; i-- )  {
/*10*/         leftBorderSum += a[ i ];
/*11*/         if( leftBorderSum > maxLeftBorderSum )
/*12*/             maxLeftBorderSum = leftBorderSum;
        }
/*13*/     int maxRightBorderSum = 0, rightBorderSum = 0;
/*14*/     for( int j = center + 1; j <= right; j++ ) {
/*15*/         rightBorderSum += a[ j ];
/*16*/         if( rightBorderSum > maxRightBorderSum )
/*17*/             maxRightBorderSum = rightBorderSum;
        }
/*18*/     return max3( maxLeftSum, maxRightSum,
/*19*/                 maxLeftBorderSum + maxRightBorderSum );
    }
```



$T(N)=2T(N/2)+O(N)$

$=O(N\log N)$




```
/* * Driver for divide-and-conquer maximum contiguous
subsequence sum algorithm.  */

int maxSubSum3( const vector<int> & a )
{         return maxSumRec( a, 0, a.size( ) - 1 );
}
```

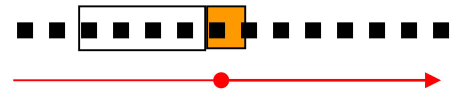
$$
\left.\begin{aligned}
T(N) &= 2\,T(N/2) + O(N)\\
T(N/2) &= 2\,T(N/4) + O(N/2)\\
T(N/4) &= 2\,T(N/8) + O(N/4)\\
T(N/8) &= 2\,T(N/16) + O(N/8)\\
&\ldots\ldots\\
T(4) &= 2\,T(2) + O(4)\\
T(2) &= 2\,T(1) + O(2)
\end{aligned}\right\} K \quad N=2^k
$$

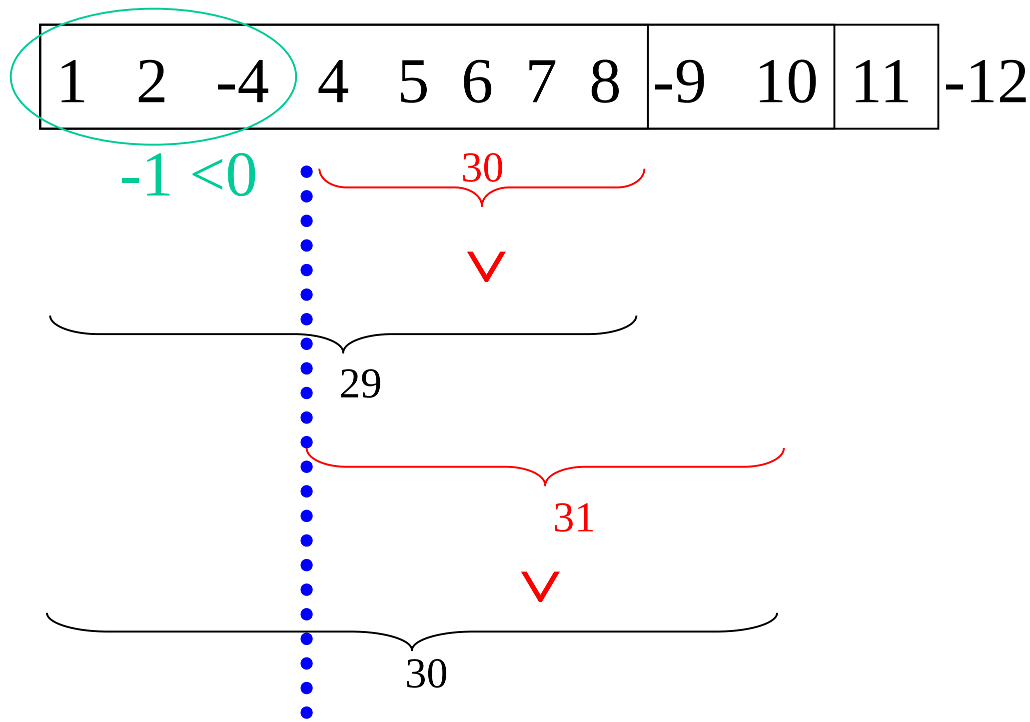
$$
\begin{aligned}
T(N) = {} & O(N)\\
& + 2\,O(N/2)\\
& + 2^2\,O(N/4)\\
& + \ldots\\
& + 2^k\,O(N/2^k)
\end{aligned}
$$

$= O(N)\ k = N\log N$

```
/* START: Fig02_08.txt        Linear-time
maximum contiguous subsequence sum
algorithm. */
 int maxSubSum4( const vector<int> & a )
{
/* 1*/      int maxSum = 0, thisSum = 0;
/* 2*/      for( int j = 0; j < a.size( ); j++ ) {
/* 3*/          thisSum += a[ j ];
/* 4*/          if( thisSum > maxSum )
/* 5*/              maxSum = thisSum;
/* 6*/          else if( thisSum < 0 )
/* 7*/              thisSum = 0;
            }
/* 8*/      return maxSum;
}
/* END */
```

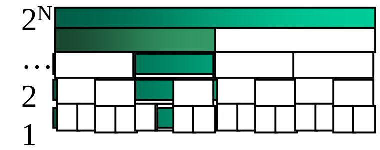
O(N)





– Logarithms

- Divide-and-conquer strategy with a constant dividing cost
- Binary search: (`Fig. 2.9, page 57`) $\Rightarrow T(N) = O(logN)$
- Euclid's algr: GCD problem (`Fig. 2.10, page 58`)
- $X^N = X^{(N-1)/2} X^{(N-1)/2} X \Rightarrow T(N) = O(logN)$



– Checking your analysis

- checking actual program's running time with number of inputs changing as: N, $2N$, $4N$, $8N$,…., and then measure the running times
- If $T(N) = O(f(N))$, let $r = \dfrac{T(N)}{f(N)}$ test program with N, $2N$, $4N$,… Inputs.

If $r \rightarrow constant$, $T(N) = \Theta(f)$

If $r \rightarrow 0$, $T(N) = O(f)$

(`Fig. 2.13, page 60`)