Designing Algorithms with Limited Work Space

省メモリのための新たなアルゴリズム設計技法:
制限された作業用メモリでアルゴリズムを如何に設計するか

NAIST, July 23, 2012

Tetsuo Asano
School of Information Science, JAIST

This talk is based on joint works with

Sergey **Bereg**, University of Texas at Dallas
Lilian **Buzer**, Universit'e Paris-Est,
David **Kirkpatrick**, UBC, Vancouver
Wolfgang **Mulzer**, Freie Universit"at Berlin
G"unter **Rote**, Freie Universit"at Berlin
Jun **Tarui**, Electro-Communication University
Yajun **Wang**, Micrsoft Research, Beijin

CPU is faster and Memory is cheaper than before

Can handle problems of larger sizes
Create new directions of computation
such as Data Streaming, Massive-Parallel, etc.

⇩

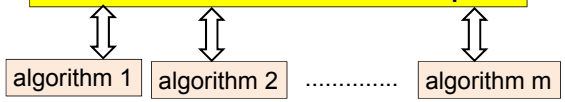
Lack of memory space

Memory Constrained Algorithm

Algorithms using less memory space
in addition to an array storing input data.

Also interested in space-time tradeoff.

**Shared Memory: large scale,
a set of data which can be observed,
or a set of values which can be computed**



A number of algorithms located in different CPUs access a large scale shared memory.
The data on the shared memory may be used for different purposes and thus they should not modified.

➔ Assume that the data are stored in a **read-only array**.
Any element can be read in constant time.

Basic assumption

Three different models for memory-constrained algorithms:

[1] Constant work space algorithm

Can use only a constant number of variables, each of O(log n) bits, where n is the input size.
Such an algorithm has been called **"log-space" algorithms** in complexity theory and has been extensively studied.

[2] Small work space algorithm

Can use only o(n) work space (more exactly, o(n log n) bits).
Typical case: $O(\sqrt{n})$ work space.

[3] Adjustable work space algorithm

Can use work space which is available at execution time.
Work space: O(s), which is between O(1) and o(n).
Space-time tradeoff:
upper and lower bounds on the space-time product.

Memory-Adjustable Data Structures

Conventional setting for data structure:
purpose: to efficiently answer queries on some set **S** of **n** objects.
the whole set is included in the structure
requiring at least as large as the size of the set
(sometimes superlinear)

New setting:
a set **S** of **n** objects is given in a **read-only array**
preprocess it, and store some additional helpful information in a data structure of adjustable size **s**.
Objective:
(1) to design an algorithm that uses this additional information in a way that the query algorithm runs faster.
(2) To achieve a good trade-off between query time and space.

Problem 1: Best time interval
Given a list of monthly profit, find the best time interval that gives the maximum total amount of profit.

Example

month	1	2	3	4	5	6	7	8	9	10	11	12
profit	4	-3	5	-7	9	-2	5	-7	9	-2	-5	6

P[3,6] =5
P[6,10] = 3
P[9,11] = 2
P[5, 9] = 14 : maximum

month	1	2	3	4	5	6	7	8	9	10	11	12
profit	4	-3	5	-7	9	-2	5	-7	9	-2	-5	6

```
Naive Algorithm
maxT = 0
for i=1 to n  // left endpoint
  for j=i to n  // right endpoint
    sum=p[i]
    for k=i+1 to j   // sum in the interval [i, j]
      sum = sum + p[k]
    if sum > maxT then maxT = sum
output maxT.
```

time: $O(n^3)$
space: O(1) in addition to the input read-only array of size n

month	1	2	3	4	5	6	7	8	9	10	11	12
profit	4	-3	5	-7	9	-2	5	-7	9	-2	-5	6

```
Another Naive Algorithm using 2-D array
maxT = 0, iopt= jopt = 0.
for i=0 to n-1
  for j=0 to n-1
    if j<i then s[i]][j] = 0.        // computing s[i][j]
    else if j=i then s[i][j] = p[i].
    else s[i][j] = s[i][j-1] + p[j].
    if s[i][j] > maxT then maxT = s[i][j], iopt=i, jopt=j.
Output [iopt, jopt]
```

time: $O(n^2)$
space: $O(n^2)$ in addition to the input read-only array of size n

Note that only two elements s[i][j] and s[i][j-1] are accessed.
➔ Just replace them with two variables s1 and s2.

```
Another Naive Algorithm using 2-D array
maxT = 0, iopt= jopt = 0.
for i=0 to n-1
  for j=0 to n-1
    if j<i then s[i]][j] = 0.        // computing s[i][j]
    else if j=i then s[i][j] = p[i].
    else s[i][j] = s[i][j-1] + p[j].
    if s[i][j] > maxT then maxT = s[i][j], iopt=i, jopt=j.
Output [iopt, jopt]
```

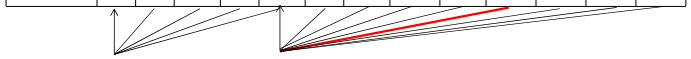
```
Another Naive Algorithm using 2-D array
maxT = 0, iopt= jopt = 0.
for i=0 to n-1
  s1 = 0
  for j=i to n-1
    s2 = s1,  s1 = s1 + p[j]
    if s1 > maxT then maxT = s1, iopt=i, jopt=j.
Output [iopt, jopt]
```

O(1) space,
but still
$O(n^2)$ time

Compute S[i] = p[1] + p[2] + ... + p[i].
Then, P[i,j] = p[i]+p[i+1]+...+p[j]
= p[1]+p[2]+ ... + p[j] - (p[1]+p[2]+ ... + p[i-1])
= S[j] - S[i-1]

Once we have computed the array S[0]=0, S[1], ... , S[n]
for each j, we should find the smallest among S[0], ... , S[j-1]

month	0	1	2	3	4	5	6	7	8	9	10	11	12
profit	0	4	-3	5	-7	9	-2	5	-7	9	-2	-5	6
sum	0	4	1	6	-1	8	6	11	4	13	11	6	12



the smallest element to the left

This leads to O(n)-space and $O(n^2)$-time algorithm
or O(n)-space, O(n)-time algorithm.

Algorithm for maintaining the smallest element to the left

```
sel = 0; sum = 0;
for i=1 to n
  sum = sum + p[i];
  if sum < sel then sel = sum
```

Algorithm for finding the best time interval

```
sel = 0; sum = 0; maxT = 0;
for i=1 to n
  sum = sum + p[i];
  if sum - sel > maxT then maxT = sum - sel;
  if sum < sel then sel = sum;
output maxT;
```

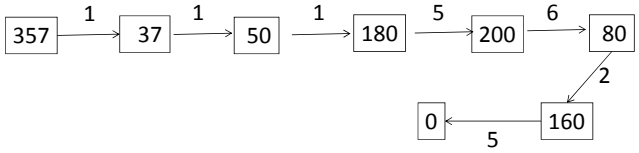
Time: O(n)
Space: O(1) in addition to the read-only input array p[ ]

Problem 2: Compute m/n exactly (using repeated decimals)

two integers **m** and **n** are given

repeat computing quotients and remainders until we get the same remainder again or the remainder is 0.

Example: m = 357, n = 320
357 / 320 = 1.115625

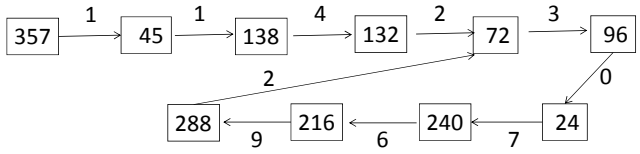


Compute m/n exactly (using repeated decimals)

two integers m and n are given

repeat computing quotient and remainder until we get the same remainder again or the remainder is 0.

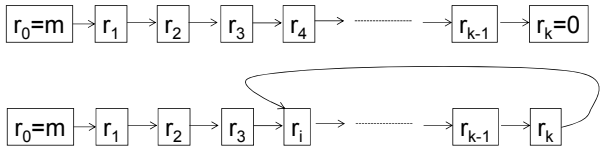
Example: m = 357, n = 312
357 / 312 = 1.144(230769)



Compute m/n exactly (using repeated decimals)

two integers m and n are given

repeat computing quotient and remainder until we get the same remainder again or the remainder is 0.



- Distinguish two cases
- If cycle, locate the entrance point.

Question:
Is it possible to design an algorithm using no array?

D: a data structure

- **Insertion**: Insert a remainder.
- **Query**: Determine whether a query element is in ***D***.

Algorithm for computing m/n

```
Initialize the data structure D.
r = m.
while (r>0 and Query(r) = NO){
   insert(r).
   r = r * 10 mod n.
}
if r = 0 then it is not "repeated".
else it is "repeated".
```

If ***D*** can be implemented by a **constant-space data structure**, then we have a constant-work-space algorithm, which takes quadratic time.

What is a constant-space data structure?

a collection of algorithms executing all the operations required for a data structure using work space of constant number of words of O(log n) bits.

D: a data structure

- Insertion: Insert a remainder
- Query: Determine whether a query remainder is in ***D***.

Insertion:
Just maintain the number of remainders inserted so far.
This is done by incrementing a counter value.

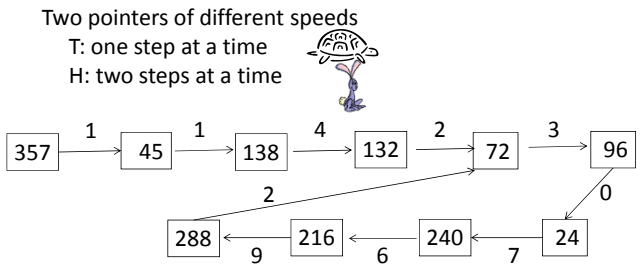
Query:
recompute a remainder sequence of length **C** to see whether the query remainder is encountered.

➔Quadratic time algorithm using work space of O(log n) bits

Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

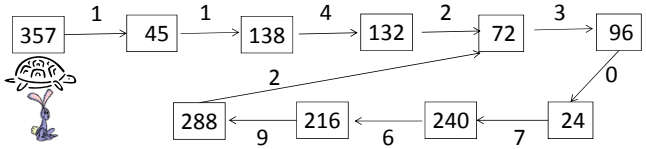
Two pointers of different speeds
T: one step at a time
H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

Two pointers at different speeds
T: one step at a time
H: two steps at a time

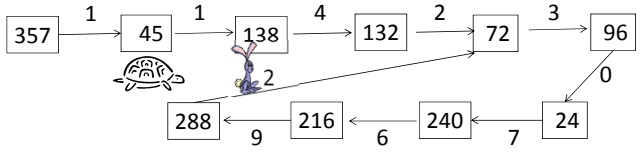


According to the Wikipedia on "Cycle detection"
D. Knuth credits the algorithm Floyd for the algorithm in his book "The art of Computer Programming, vol. II, Seminumerical Algorithms," Addison-Wesley.

Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

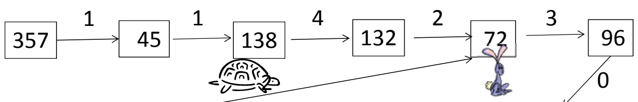
Two pointers at different speeds
T: one step at a time
H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

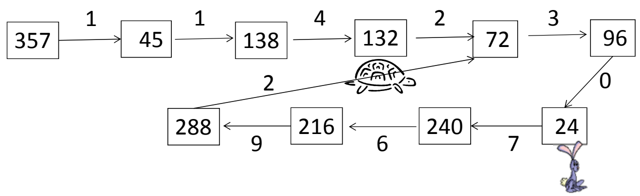
Two pointers at different speeds
- T: one step at a time
- H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

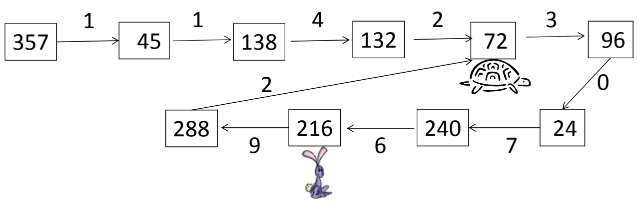
Two pointers at different speeds
- T: one step at a time
- H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

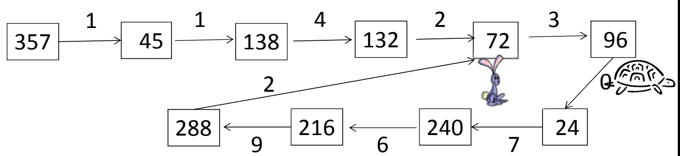
Two pointers at different speeds
- T: one step at a time
- H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

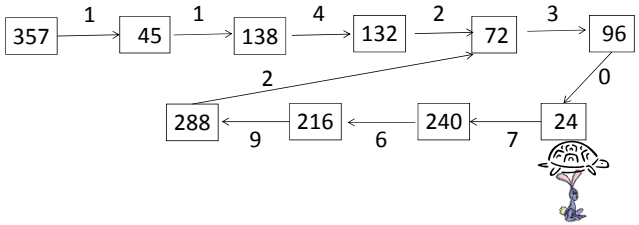
Two pointers at different speeds
- T: one step at a time
- H: two steps at a time



Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm

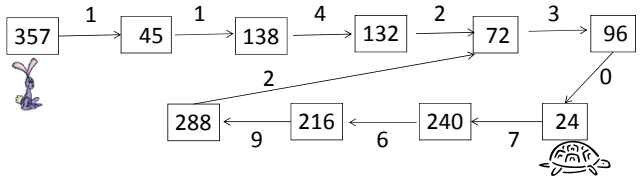
Two pointers at different speeds
- T: one step at a time
- H: two steps at a time



The Tortoise meets the Hare!

Any improvement on the running time?

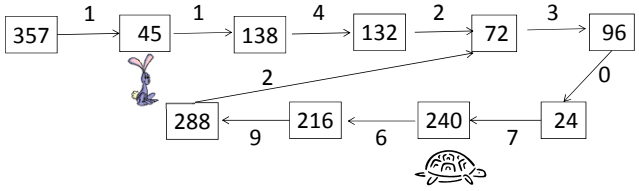
Technique called "**the Tortoise and the Hare**" algorithm



The Hare goes back to the start.

Any improvement on the running time?

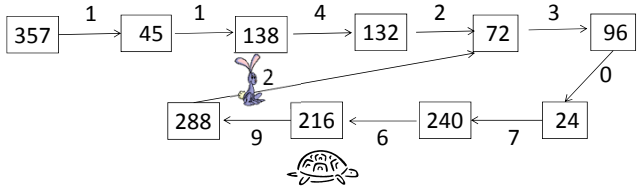
Technique called "**the Tortoise and the Hare**" algorithm



Both of them at the same speed.

Any improvement on the running time?

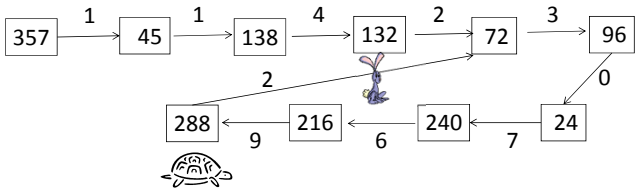
Technique called "**the Tortoise and the Hare**" algorithm



Both of them at the same speed.

Any improvement on the running time?

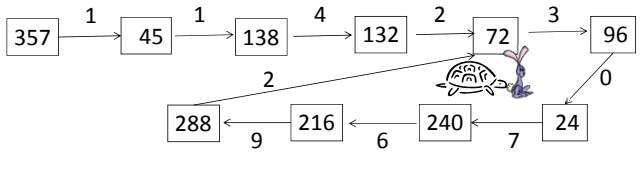
Technique called "**the Tortoise and the Hare**" algorithm



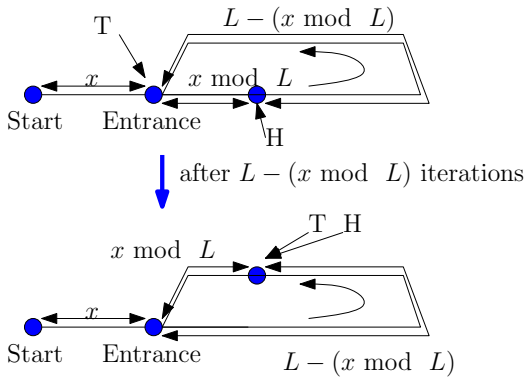
Both of them at the same speed.

Any improvement on the running time?

Technique called "**the Tortoise and the Hare**" algorithm



The Tortoise meets the Hare again.



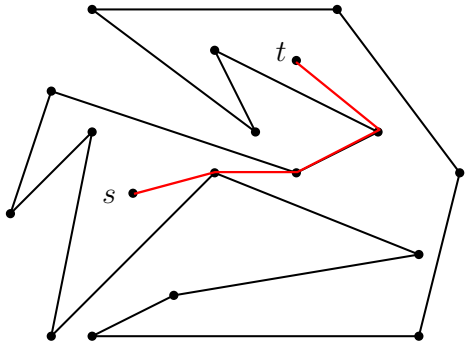

```
#include <stdio.h>
int main(void){
    int n, ra, rb;

    printf("Enter m and n ");  scanf("%d %d, &m, &n);
    printf("¥n %d/%d = %d.", m, n, m/n);
    ra = rb = m % n;
    do{
        ra = ra*10 % n;
        rb = rb*10 % n; rb = rb*10 % n;
    } while(ra != rb && rb != 0);
    if(rb == 0){
        ra = 1;
        do{
            printf("%1d", ra*10/n);
            ra = ra*10 % n;
        } while(ra>0);
    } else { // Case: ra == rb
```

21/44 = 0.4772727272...
27/52 = 0.5192307692...
29/56 = 0.5178571428...

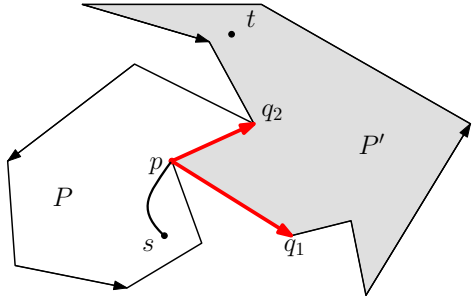
```
} else { // Case: ra == rb
        rb = 1;
        while(ra != rb){
            ra = ra*10 % n;
            rb = rb*10 % n;
        }
        ra = 1;
        while(ra != rb){
            printf("%1d", ra*10/n);
            ra = ra*10 % n;
        }
        printf("(");
        do{
            printf("%1d", ra*10/n);
            ra = ra*10 % n;
        } while(ra != rb);
        printf(")");
    }
    return 1;
}
```

Problem: Given a simple polygon *P* and two points *s* and *t* in its interior, find a shortest path between *s* and *t* within *P*.

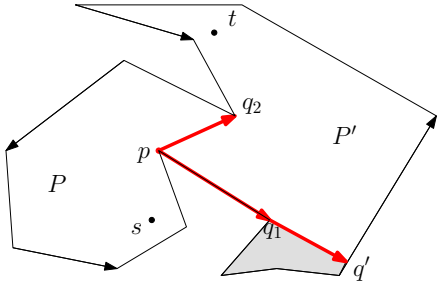


Is it possible to find a shortest path without using any extra array --- **constant work space algorithm**?

Invariant:
(i) The geodesic shortest path from **s** to **t** passes through **p**.
(ii) **t** lies in the subpolygon **P'**.

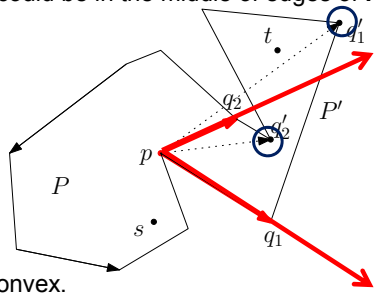


Case 1:
$\mathbf{q_1}$ is a vertex of **P'** which is concave, that is, **p**, $\mathbf{q_1}$, **succ(**$\mathbf{q_1}$**)** is a clockwise turn, where **succ(q)** is the succeeding vertex of **q** on the boundary of **P**.



Extend the ray pq_1 until it hits the boundary of **P'** at **q'**.
The segment $\mathbf{q_1q'}$ splits the **P'** into two parts.
Check which one contains **t.**

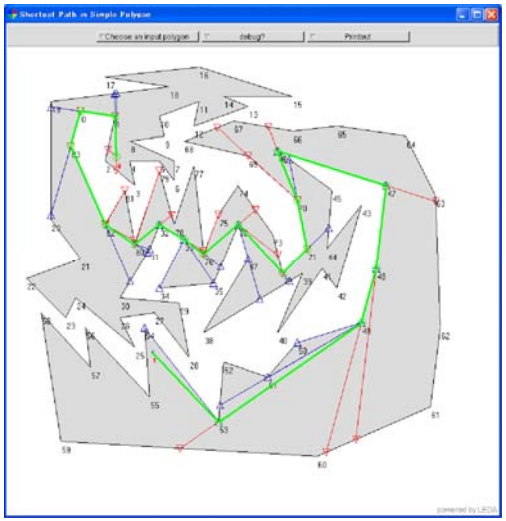
Case 2:
Both $\mathbf{q_1}$ and $\mathbf{q_2}$ are convex vertices of **P'**.
In particular, $\mathbf{q_1}$ and/or $\mathbf{q_2}$ could be in the middle of edges of **P**.



If both of $\mathbf{q_1}$ and $\mathbf{q_2}$ are convex.
Then, one of ($\mathbf{q_1}$, succ($\mathbf{q_1}$)) and ($\mathbf{q_2}$, pred($\mathbf{q_2}$)) lies between $p\mathbf{q_1}$ and $p\mathbf{q_2}$.

otherwise there would be a crossing.

Theorem: [A., Mulzer, Rote, Wang, 2011]
Given a simple polygon of n vertices and two points s and t in its interior, there is an algorithm which finds the shortest path from s to t in $O(n^2)$ time using constant work space.



$O(n^2)$-time algorithm using O(log n) bits

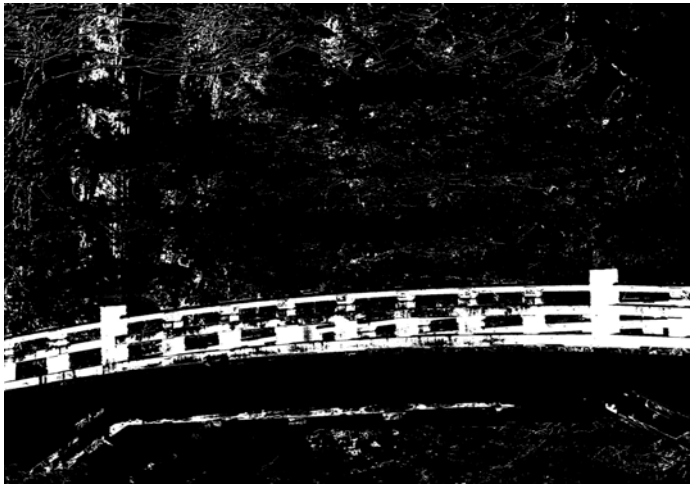
Problem setting:

Let ***G*** be a color image of ***n*** pixels, and
let ***f*** be a function from color space into {0, 1}.

example: red: 80 < r < 255
green: 0 < g < 80
blue: 0 < b < 255
for each pixel ***p*** in ***G***
f(p) = 1 if color values at ***p*** are within the ranges above
= 0 otherwise

Then, the function ***f*** defines a binary image ***f(G).***
So, we assume that the input binary image f(G) is stored in a read-only array.

input image red: 80-255, green: 0-80, blue: 0-255



Binary image f(G): red: 80-255, green: 0-80, blue: 0-255

Input color image filtered by f(G)

Given a color image ***G*** of ***n*** pixels and a function ***f*** from color space into {0, 1}, we can define a binary image ***f(G).***

Problem:

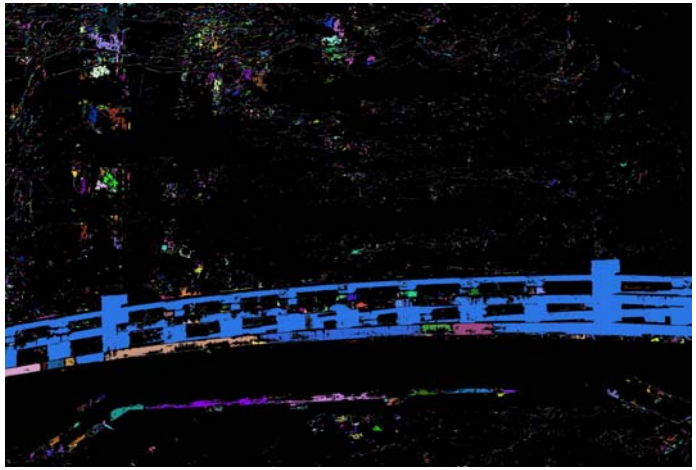
(1) Compute the **number of connected components** in the binary image.

(2) Output the binary image in **raster order** such that
g(***p***) = 1 if and only if ***f***(***p***) = 1 and
p belongs to the **largest component** in ***f(G).***

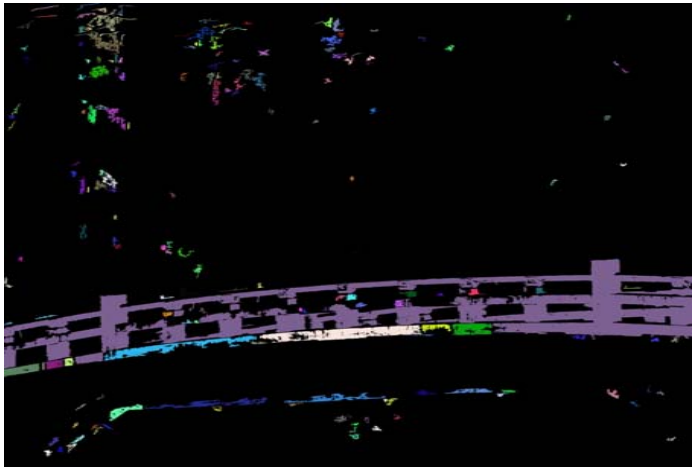
(3) Output the binary image in **raster order** such that
g(***p***) = 1 if and only if ***f***(***p***) = 1 and
p belongs to a **component of size > K** in ***f(G).***

(4) Output a **label matrix *M(f(G))*** in raster order such that
same component ➔ same label
different components ➔ different labels
value 0 ➔ label 0

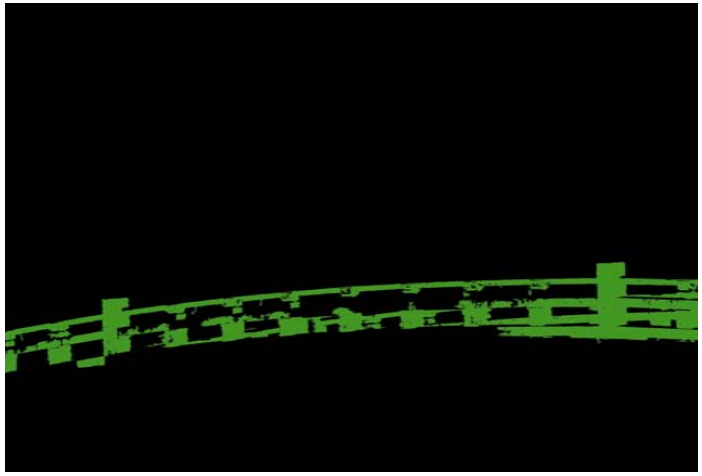
input image red: 80-255, green: 0-80, blue: 0-255



Component labeling (using different colors)



Components of sizes > 200 pixels



Largest component

Connected Components Labeling

What is a connected component?

M: an nxn binary image (matrix)
M(x, y) = 0 or 1
Two 1-pixels are connected if there is a rectilinear sequence of 1-pixels between them.

A connected component is a maximal set of 1-pixels any two of which are connected.

0	0	0	0	0	1	1	0	0	1	0	0	
0	0	0	0	0	0	1	0	1	1	0	0	
0	0	1	1	1	0	0	1	1	0	1	1	
0	1	1	1	1	0	1	1	0	0	1	1	
0	1	1	0	1	0	0	0	0	1	1	1	
0	1	1	0	1	0	0	1	1	1	0	0	
0	1	1	0	1	1	0	0	1	1	0	0	
0	1	1	1	1	0	0	0	0	1	1	1	

⇨

0	0	0	0	0	2	2	0	0	3	0	0	
0	0	0	0	0	0	2	0	3	3	0	0	
0	0	1	1	1	0	0	3	3	0	4	4	
0	1	1	1	1	0	3	3	0	0	4	4	
0	1	1	0	1	0	0	0	0	4	4	4	
0	1	1	0	1	0	0	4	4	4	0	0	
0	1	1	0	1	1	0	0	4	4	0	0	
0	1	1	1	1	0	0	0	0	4	4	4	

FILL Algorithm for Connected Component Labeling

```
L = 1
for each pixel p s.t. g[p]=1 in raster order
   Increment L.
   g[p] = L.
   Push p into a stack.
   while(stack is not empty){
      Pop a pixel r from the stack.
      for each pixel p with g[p]=1 adjacent to r
         Push p into the stack .
         g[p] = L.
   }
```

R. Klette and A. Rosenfeld:
``Digital Geometry: Geometric Methods for Digital Picture Analysis,'' Elsevier, 2004.

Observation:
Given a color image **G** of **n** pixels and a function **f** from color space into {0, 1}, we can compute a labeling matrix for the binary image **f**(**G**) in **O(n)** time using work space of **O(n log n)** bits.

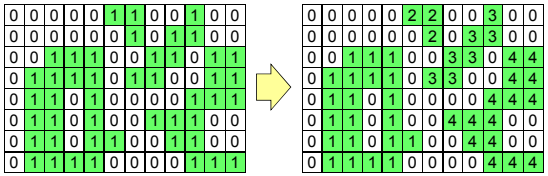
If we need the labeling matrix as the output, the algorithm is optimal.
The labeling matrix requires **Θ(n log n)** bits.

New situation:
don't need to store the labeling matrix.
just need to output the matrix elements in raster order.

How much work space can be saved?

Enumeration of all components together with their areas

Given a binary image G, enumerate all connected components together with their areas (number of pixels) each exactly once.

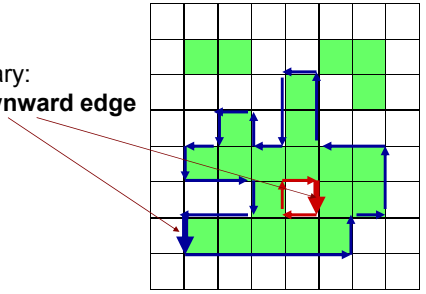


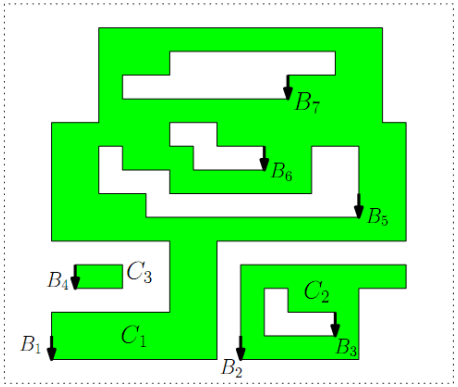
There is an algorithm for enumerating all components together with their areas in a read-only binary image in O(n log n) time using work space of O(log n) bits.

Key Idea for Counting

Basic assumption: Component boundary is directed so that the interior always lies to the left of the boundary
External boundary : counter-clockwise order
Internal boundary : clockwise order

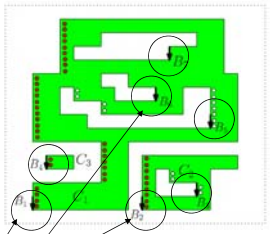
Canonical edge
for each boundary:
the lowest downward edge





Each boundary (internal/external) has a unique canonical edge, which is the lowest downward edge on the boundary.

Checking orientation



Local configuration around a starting edge

0 1
0
External boundary

0 1
1
Internal boundary

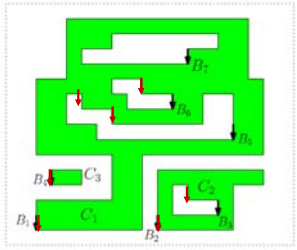
So easy to check the orientation!!

Algorithm for Connected Components Counting

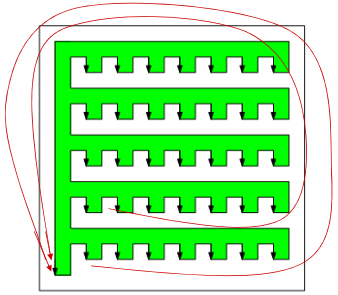
counter = 0.
Perform Raster Scan
  whenever we find a candidate downward edge e.
  Apply **Boundary_trace_from(e).**
  if it returns 1 then increment the count
Report the count.

0 1
? 0

Boundary_trace_from(e){
  e_s = e.
  do{
    e = next edge of e on the same boundary.
    if e is downward and lower than e_s then return 0.
  }while(e != e_s)
  return 1.
}



Theorem: The algorithm Connected-Components-Counting correctly computes the number of connected components in a given binary image of n pixels in $O(n^2)$ time using only work space of O(log n) bits.

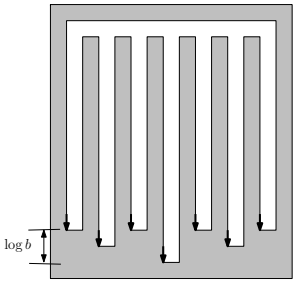


worst-case example

How to improve the time complexity

bidirectional search

whenever we find a candidate edge e, search on the boundary in two opposite directions for any lower edge.



worst-case example
which requires O(n log n)
for an image with n pixels

Theorem:
Given a binary image of n pixels on a read-only array, we can enumerate all components together with their areas in the image in O(n log n) time using work space of O(log n) bits.

Counting can be done in O(n log n) time.

Possible to extend the result to labeling problem?

Given a binary image of n pixels,
is it possible to output labels of pixels in raster order
using work space of O(log n) bits in polynomial time?

A New Algorithm for Connected Components Labeling

Idea: put labels along boundaries and propagate them to the right.

0	0	0	0	0	1	1	0	0	1	0	0
0	0	0	0	0	0	1	0	1	1	0	0
0	0	1	1	1	0	0	1	1	0	1	1
0	1	1	1	1	0	1	1	0	0	1	1
0	1	1	0	1	0	0	0	0	1	1	1
0	1	1	0	1	0	0	1	1	1	0	0
0	1	1	0	1	1	0	0	1	1	0	0
0	1	1	1	→	0	0	0	0	1	1	1

Algorithm for connected components labeling

```
Array: edge[ ], lab[ ]. // label arrays for two rows
L = 0  // the number of labels used so far
Initialize the array edge[.] = 0
for y=1 to h
  for x = 1 to w  // scan pixels in raster order
     p = pixel at (x, y)  // p-1 = pixel at (x-1, y)
     switch(value of B[p-1]B[p]){
     case  '00': lab[p] = 0.
     case  '11': lab[p] = lab[p-1].
     case  '01': Let e be the edge between 0 and 1.
        if edge[e] = 0 then // new component
           create a new label L, and lab[p] = L.
           follow the boundary from e while putting L into edges
     case  '10': Let e be the edge between 1 and 0.
        if edge[e] = 0 then { k = lab[p-1]
           follow the boundary from e while putting k into edges.}
     }
```

Algorithm for connected components labeling

switch(value of B[p-1]B[p]){
case '00': lab[p] = 0.
case '11': lab[p] = lab[p-1].
case '01': Let **e** be the edge between 0 and 1.
if edge[e] = 0 then // new component
create a new label **L**, and lab[p] = L.
follow the boundary from e while putting L into edges
case '10': Let **e** be the edge between 1 and 0.
if edge[e] = 0 then { **k** = lab[p-1]
follow the boundary from e while putting k into edges.}
}

When do we follow a boundary in the case '01'?

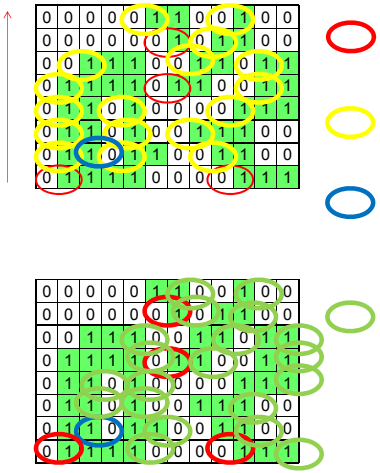
This event happens only at a canonical edge of an **external** boundary.

Algorithm for connected components labeling

switch(value of B[p-1]B[p]){
case '00': lab[p] = 0.
case '11': lab[p] = lab[p-1].
case '01': Let **e** be the edge between 0 and 1.
if edge[e] = 0 then // new component
create a new label **L**, and lab[p] = L.
follow the boundary from e while putting L into edges
case '10': Let **e** be the edge between 1 and 0.
if edge[e] = 0 then { **k** = lab[p-1]
follow the boundary from e while putting k into edges.}
}

When do we follow a boundary in the case '10'?

This event happens only at a canonical edge of an **internal** boundary.



01 event
for a canonical edge

01 event
for a non-canonical edge

10 event
for a canonical edge
for a hole

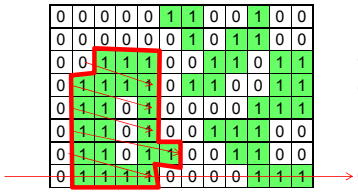
10 event
for a non-canonical edge

Lemma:
Given a binary image of n pixels in a read-only array, we can report the labeling matrix in raster order in O(n) time if O(n) work space is available.

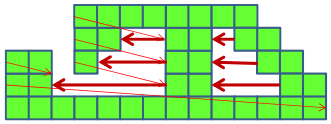
Proof:
At every canonical edge we follow the corresponding boundary, each exactly once, which takes O(n) time in total.
We can put correct labels along boundaries.
Propagating labels to the right takes O(n) time.
Thus, the total time we need is O(n) provided that labels can be stored in edges on boundaries.

Reducing work space into O(√n)

Idea: Just maintain labels in two rows, current and previous.



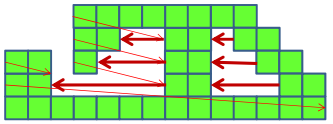
If we start at a non-canonical edge e, then we must encounter an edge e' < e (in the previous row or in the same row but to the left of e). Note that the pixel on the edge has been labeled.



Reducing work space into O(√n)

Idea: Just maintain labels in two rows, current and previous.

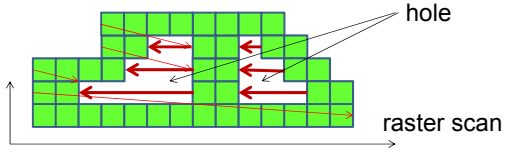
case '01': Let **e** be the edge between 0 and 1.
Determine whether the edge is canonical or not by following the boundary fro e.
If the boundary has no edge < e, then it is canonical.
if canonical then create a new label **L**, and lab[c][x] = L.



Reducing work space into O(√n)

Idea: Just maintain labels in two rows, current and previous.

case '10': Let **e** be the edge between 1 and 0.
Follow the boundary from e
until we encounter an edge e' < e.
k = the label of the pixel just to the left of e'
lab[c][x] = k.



Theorem:
Given a binary image of n pixels in a read-only array, we can report the labeling matrix in raster order in $O(n^2)$ time using $O(\sqrt{n})$ work space.

Proof:
For each edge **e**, we follow the boundary until we see an edge **e' < e** in the raster order.
Once we find such an edge, we get a correct label.
It may take O(n) time for each edge.
Thus, it takes $O(n^2)$ time.
The work space we need is $O(\sqrt{n})$ to maintain labels in two rows.
Q.E.D.

Acceleration of the algorithm

Idea: Just use bidirectional search

This is just the same as the one for finding canonical edges.

Theorem:
Given a binary image of n pixels in a read-only array, we can report the labeling matrix in raster order in O(n log n) time using bidirectional search using O(√n) work space.

Animation: Algorithm 1 using O(n) space

Algorithm 2 using O(√n) space

Algorithm 3 (bidirectional search)

Problem of Computing a Largest Component

Problem: Given a binary image of n pixels in a binary image, compute its largest connected component.

Problem 1: Output a set of pixels (locations) in a largest connected component.

Problem 2: Output a binary image in which a pixel is white if and only if it is white and belongs to a largest connected component.

Which is harder?
Both of them can be solved in O(n) time if O(n) work space is available. What about the case when O(√n) space is available.

Conclusions and Future Works

- Try more problems on computational geometry
- designing more space-adjustable algorithms or characterize those problems for which efficient we can design space-adjustable algorithms.
- Developing general schema for designing space-adjustable algorithms