Sorting & Searching

LI Hao 李颢, Assoc. Prof. SPEIT & Dept. Automation of SEIEE

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Sorting

- **Sorting**
  - *data*
  - *priority*
    - min-prior: E1<E2 (e.g. 3 is prior to 5)
    - max-prior: E1>E2 (e.g. 5 is prior to 3)
    - ascending-aphabet-prior: E1 before E2 in aphabet (e.g. c is prior to e)
    - descending-aphabet-prior: E1 after E2 in aphabet (e.g. e is prior to c)
    - *ad hoc* defined prior
  - *swap*
    - swap elements in wrong prior-order

unsorted	37	11	43	59	89	23	67	71
min-prior	11	23	37	43	59	67	71	89
max-prior	89	71	67	59	43	37	23	11

Sorting

- **Insertion sort**
  - insert current element into correct place in previous elements sorted

unsorted	37	11	43	59	89	23	67	71
insert 37	37							
insert 11	37	11						
	11	37						
insert 43	11	37	43					
insert 59	11	37	43	59				
insert 89	11	37	43	59	89			
insert 23	11	37	43	59	89	23		

Sorting

- **Insertion sort**
  - insert current element into correct place in previous elements sorted

unsorted	37	11	43	59	89	23	67	71
insert 23	11	37	43	59	89	23		
	11	37	43	59	23	89		
	11	37	43	23	59	89		
	11	37	23	43	59	89		
	11	23	37	43	59	89		
insert 67	11	23	37	43	59	89	67	
	11	23	37	43	59	67	89	

Sorting

- **Insertion sort**
  - insert current element into correct place in previous elements sorted

unsorted	37	11	43	59	89	23	67	71
insert 23	11	37	43	59	89	23		
	11	23	37	43	59	89		
insert 67	11	23	37	43	59	89	67	
	11	23	37	43	59	67	89	
insert 71	11	23	37	43	59	67	89	71
	11	23	37	43	59	67	71	89
min-prior	11	23	37	43	59	67	71	89

Sorting

- **Insertion sort**
  - insert current element into correct place in previous elements sorted
  - worst case
    - insertion of k-th element involves k-1 comparison (and swap)
    - complexity: $O(n^2)$
  - best case
    - insertion of k-th element involves only one comparison
    - complexity: $O(n)$
  - average (probabilistic expectation) case
    - insertion of k-th element involves an expectation of (k-1)/2 comparison (and swap)
    - complexity: $O(n^2)$

Sorting

- **Bubble sort**
  - traverse backward & swap adjacent elements that are in wrong prior-order

unsorted	37	11	43	59	89	23	67	71
1st traversal	37	11	43	59	89	23	67	71
	37	11	43	59	23	89	67	71
	37	11	43	23	59	89	67	71
	37	11	23	43	59	89	67	71
	11	37	23	43	59	89	67	71
2nd traversal	11	37	23	43	59	89	67	71
	11	37	23	43	59	67	89	71

Sorting

- **Bubble sort**
  - traverse backward & swap adjacent elements that are in wrong prior-order

unsorted	37	11	43	59	89	23	67	71
1st traversal	11	37	23	43	59	89	67	71
2nd traversal	11	37	23	43	59	67	89	71
	11	23	37	43	59	67	89	71
3rd traversal	11	23	37	43	59	67	89	71
	11	23	37	43	59	67	71	89
4th traversal	11	23	37	43	59	67	71	89

Sorting

- **Bubble sort**
  - traverse backward & swap adjacent elements that are in wrong prior-order

unsorted	37	11	43	59	89	23	67	71
1st traversal	11	37	23	43	59	89	67	71
2nd traversal	11	23	37	43	59	67	89	71
3rd traversal	11	23	37	43	59	67	71	89
4th traversal	11	23	37	43	59	67	71	89
5th traversal	11	23	37	43	59	67	71	89
6th traversal	11	23	37	43	59	67	71	89
7th traversal	11	23	37	43	59	67	71	89

Sorting

- **Bubble sort**
  - traverse backward & swap adjacent elements that are in wrong prior-order
  - worst case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$
  - best case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$
  - average (probabilistic expectation) case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$

Sorting

- **Selection sort**
  - *k*-th traversal selects the *k*-th prior element

unsorted	37	11	43	59	89	23	67	71
1st traversal	11							
2nd traversal	11	23						
3rd traversal	11	23	37					
4th traversal	11	23	37	43				
5th traversal	11	23	37	43	59			
6th traversal	11	23	37	43	59	67		
7th traversal	11	23	37	43	59	67	71	89

Sorting

- **Selection sort**
  - *k*-th traversal selects the *k*-th prior element
  - worst case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$
  - best case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$
  - average (probabilistic expectation) case
    - *k*-th traversal involves (n-*k*) comparison (and potential swap)
    - complexity: $O(n^2)$

Sorting

- **Adjacent exchange sort**
  - insertion sort W:O(n^2) A:O(n^2) B:O(n)
  - bubble sort W:O(n^2) A:O(n^2) B:O(n^2)
  - selection sort W:O(n^2) A:O(n^2) B:O(n^2)

- **Reflection**
  - insertion sort's ability to take advantage of almost sorted status
    - save "routine-administrative" operations & efficient at *handling small sequences*
    - can be integrated into advanced sorts to handle trivial & small "terminal" sub-tasks
  - bubble sort seems to be "stupid", then what is its use?
    - involve purely *local operations* & can be fully parallelized
    - spirit of *parallel processing*

Sorting

- **Shell sort**
  - hierarchical insertion sort with varying increments

unsorted	89	11	43	71	37	23	67	59
insertion sort +4	89				37			
	37				89			
		11				23		
			43				67	
				71				59
				59				71
	37	11	43	59	89	23	67	71

Sorting

- **Shell sort**
  - hierarchical insertion sort with varying increments

unsorted	89	11	43	71	37	23	67	59
insertion sort +4	37	11	43	59	89	23	67	71
insertion sort +2	37		43		89		67	
	37		43		67		89	
		11		59		23		71
		11		23		59		71
insertion sort	37	11	43	23	67	59	89	71
	11	23	37	43	59	67	71	89

Sorting

- **Adjacent exchange sort**
  - insertion sort
  - bubble sort
  - selection sort

```
template <typename T> inline void swap(T s[],int a,int b){
        T tmp=std::move(s[a]);s[a]=std::move(s[b]);s[b]=std::move(tmp);}
template <typename T> inline void swap(T& a,T& b){
        T tmp=std::move(a);a=std::move(b);b=std::move(tmp);}

// adjacent exchange sort: insertion; bubble; selection
// insertion sort | W:O(n^2); A:O(n^2); B:O(n)
template <class T,class P> void insertionsort(T s[],int n){
        for(int i=1;i<n;i++)
                for(int j=i;(j>0)&&(P::p(s[j],s[j-1]));j--)
                        swap(s,j,j-1);}
// bubble sort | W:O(n^2); A:O(n^2); B:O(n^2)
template <class T,class P> void bubblesort(T s[],int n){
        for(int i=0;i<n-1;i++)
                for(int j=n-1;j>i;j--)
                        if(P::p(s[j],s[j-1])) swap(s,j,j-1);}
// selection sort | W:O(n^2); A:O(n^2); B:O(n^2)
template <class T,class P> void selectionsort(T s[],int n){
        for(int i=0;i<n-1;i++){int imin=i;
                for(int j=n-1;j>i;j--)
                        if(P::p(s[j],s[imin])) imin=j;
                swap(s,i,imin);}}
// END adjacent exchange sort: insertion; bubble; selection
```

Sorting

- **Adjacent exchange sort**
  - insertion sort
  - bubble sort
  - selection sort

```
#include <iostream>
#include "sorting.h"
#include "prior.h"
using namespace std; using cptr=const char*; // typedef const char* cptr;

template <class T> void show(const T s[],int n){
        for(int i=0;i<n;i++) cout<<s[i]<<' '; cout<<'\n';}
template <class T> void cp(T s[],const T si[],int n){while(n--) s[n]=si[n];}

int main(){const int ni=16,nc=8;
        int iTab[ni]={42,92,96,79,93,4,85,66,68,76,74,63, 39,17,71,3};int iA[ni];
        cptr cTab[nc] = {"machine", "intelligence", "system", "automation",
                "program", "technique", "computer", "data"};cptr cA[nc];
```

```
cout<<"DEMO : adjacent exchange sorts =>\n";
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"insertion sort: ";insertionsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"insertion sort: ";insertionsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"bubble sort: ";bubblesort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"bubble sort: ";bubblesort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"selection sort: ";selectionsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"selection sort: ";selectionsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
```

Sorting

- **Adjacent exchange sort**
  - insertion sort
  - bubble sort
  - selection sort

```
DEMO : adjacent exchange sorts =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
insertion sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
insertion sort: automation computer data intelligence machine program system technique
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
bubble sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
bubble sort: automation computer data intelligence machine program system technique
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
selection sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
selection sort: automation computer data intelligence machine program system technique
```

```cpp
cout<<"DEMO : adjacent exchange sorts =>\n";
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"insertion sort: ";insertionsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"insertion sort: ";insertionsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"bubble sort: ";bubblesort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"bubble sort: ";bubblesort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"selection sort: ";selectionsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"selection sort: ";selectionsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
```

Sorting

- **Shell sort**
  - hierarchical insertion sort with varying increments

```
// shell sort: hierarchical insertion sort with varying increments | A:O(n^1.5)
// insertion sort with flexible increment i.e. a
template <class T,class P> void insertionsort2(T s[],int n,int a){
        for(int i=a;i<n;i+=a)
                for(int j=i;(j>=a)&&(P::p(s[j],s[j-a]));j-=a) swap(s,j,j-a);}
// division-by-two increments: 1,2,4,8,16,...
template <class T,class P> void shellsort2(T s[],int n,int amin){
        int a=1;while(a<n) a*=2; a/=2;amin=amin<1?1:amin;
        for(;a>=amin;a/=2)
                for(int j=0;j<a;j++) insertionsort2<T,P>(&s[j],n-j,a);}
// division-by-three increments: 1,4,13,40,121,... (recommended)
template <class T,class P> void shellsort(T s[],int n,int amin){
        int a=1;while(a<n) a=3*a+1; a=(a-1)/3;amin=amin<1?1:amin;
        for(;a>=amin;a=(a-1)/3)
                for(int j=0;j<a;j++) insertionsort2<T,P>(&s[j],n-j,a);}
template <class T,class P> void shellsort(T s[],int n){shellsort<T,P>(s,n,1);}
// END shell sort
```

Sorting

- **Shell sort**
  - hierarchical insertion sort with varying increments

```
DEMO : shell sorts =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
shell sort (/2): 42 76 74 63 39 4 71 3 68 92 96 79 93 17 85 66
shell sort (/2): 39 4 71 3 42 17 74 63 68 76 85 66 93 92 96 79
shell sort (/2): 39 3 42 4 68 17 71 63 74 66 85 76 93 79 96 92
shell sort (/2): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
shell sort (/3): 17 71 3 79 93 4 85 66 68 76 74 63 39 42 92 96
shell sort (/3): 17 4 3 63 39 42 74 66 68 71 85 79 93 76 92 96
shell sort (/3): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
shell sort (/3): automation computer data intelligence machine program system technique
```

```
cout<<"DEMO : shell sorts =>\n";int amin;
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);amin=1;while(amin<ni) amin*=2;
cout<<"unsorted sequence: ";show<int>(iA,ni);
for(amin/=2;amin>=1;amin/=2){
cout<<"shell sort (/2): ";shellsort2<int,IntPriorMin>(iA,ni,amin);show<int>(iA,ni);}
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);amin=1;while(amin<ni) amin=3*amin+1;
cout<<"unsorted sequence: ";show<int>(iA,ni);
for(amin=(amin-1)/3;amin>=1;amin=(amin-1)/3){
cout<<"shell sort (/3): ";shellsort<int,IntPriorMin>(iA,ni,amin);show<int>(iA,ni);}
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"shell sort (/3): ";shellsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
```

Sorting

- **Shell sort**
  - hierarchical insertion sort with varying increments
  - complexity (depending on *shell sequence*)
    - Pratt sequence: O(n (*log* n)^2)
      - 1,2,3,4(2^2),6(2x3),9(3^2),8(2^3),12(2^2x3),18(2x3^2),27(3^3),16(2^4),24(2^3x3),36(2^2x3^2),54(2x3^3),81(3^4),...
    - geometric sequence based sequence: O($n^{1.5}$)
      - all-one binary sequence $1_{(2)}$,$11_{(2)}$,$111_{(2)}$,$1111_{(2)}$,... namely 1(2^1-1),3(2^2-1),7(2^3-1),15(2^4-1),31(25-1),...
      - the best constant factor turns out to be roughly between 2 and 4
      - **Knuth's recommended sequence** 1,4,13,40,121,364,1093,3280,... (use relatively few increments & do well in empirical studies)

$$c(n) \approx \sum_{k=1}^{\infty} a^k c(\frac{n}{a^k}) + O(n) \geq \sum_{k=1}^{log_a^n} a^k O(\frac{n}{a^k}) = O(n \log n) \Rightarrow c(n) \approx \sum_{k=1}^{\infty} a^k c(\frac{n}{a^k}) + O(n) \geq \sum_{k=1}^{log_a^n} a^k O(\frac{n}{a^k} \log \frac{n}{a^k}) \approx O(n\,(\log n)^2)$$

$$\Downarrow$$

$$c(n) \approx n^{1+log_a^2}$$

```
// division-by-three increments: 1,4,13,40,121,... (recommended)
template <class T,class P> void shellsort(T s[],int n,int amin){
        int a=1;while(a<n) a=3*a+1; a=(a-1)/3;amin=amin<1?1:amin;
        for(;a>=amin;a=(a-1)/3)
                for(int j=0;j<a;j++) insertionsort2<T,P>(&s[j],n-j,a);}
template <class T,class P> void shellsort(T s[],int n){shellsort<T,P>(s,n,1);}
```

Sorting

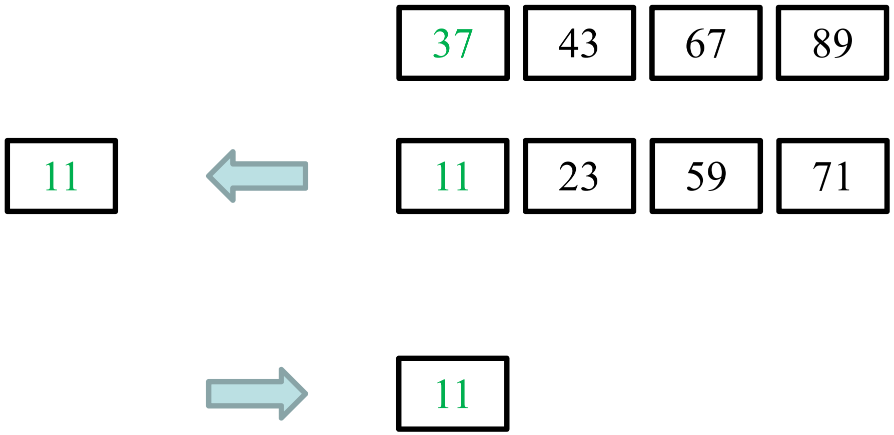
- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

sorted sub-sequence 1	37	43	67	89
sorted sub-sequence 2	11	23	59	71

11	23	37	43	59	67	71	89

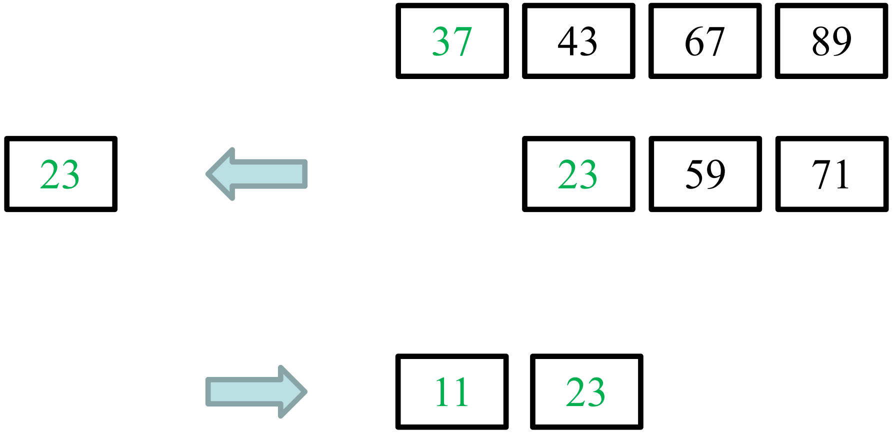
Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one



Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one



Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

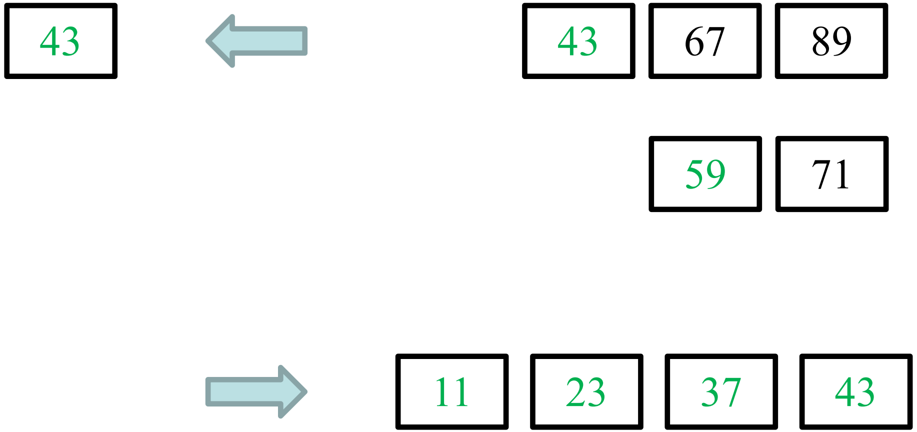
37 ← 37 43 67 89

59 71

→ 11 23 37

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one



Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

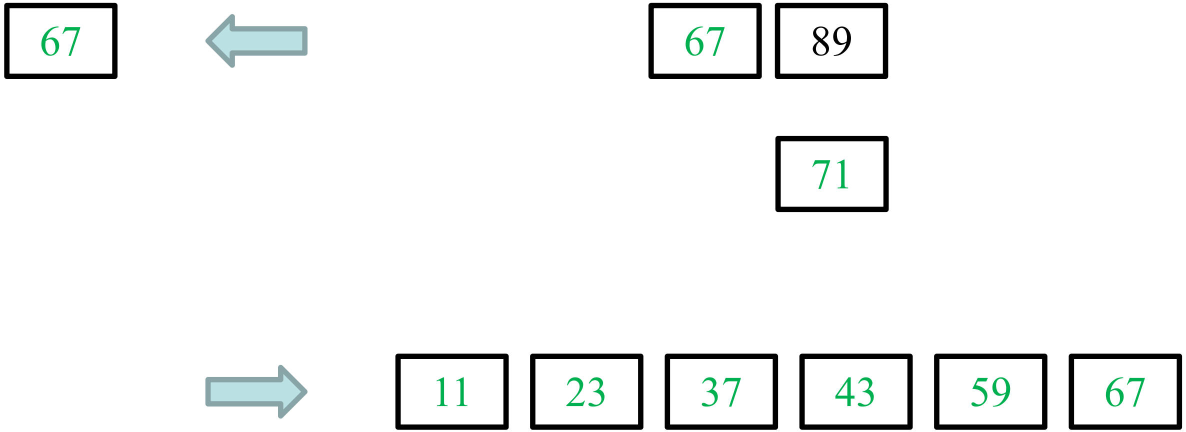
67 89

59 ⇐ 59 71

⇒ 11 23 37 43 59

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be "merged" efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one



Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

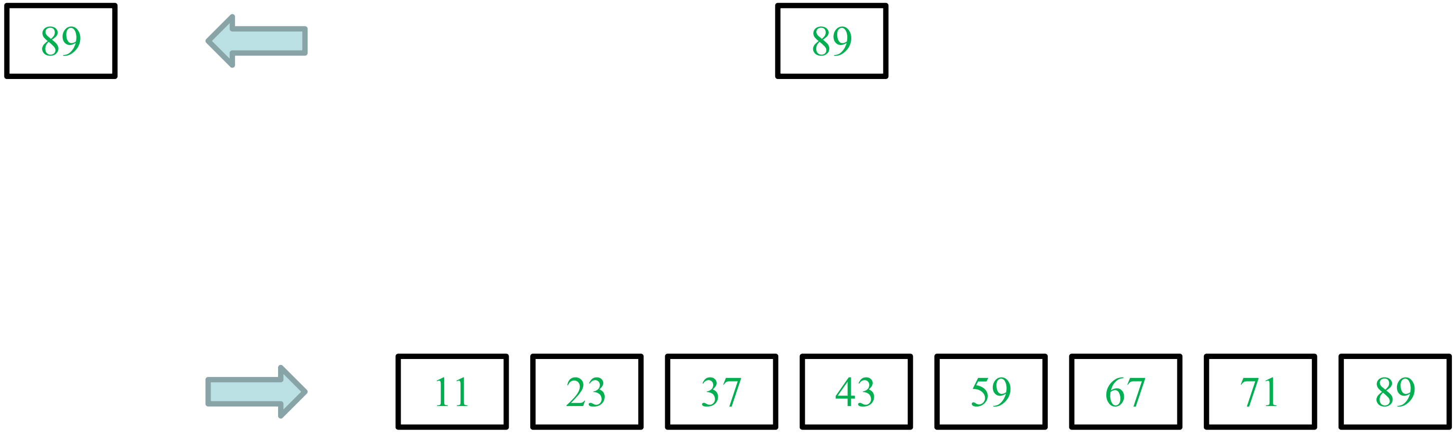
89

71 ⇐ 71

⇒ 11 23 37 43 59 67 71

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one



Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

sorted sub-sequence 1	37	43	67	89
sorted sub-sequence 2	11	23	59	71

a **single traversal** of both **sorted** sub-sequences

11	23	37	43	59	67	71	89

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be "merged" efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **complexity of merging: O(n)**

sorted sub-sequence 1

37	43	67	89

sorted sub-sequence 2

11	23	59	71

a **single traversal** of both **sorted** sub-sequences

11	23	37	43	59	67	71	89

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **complexity of merging: O(n)**
  - divide a sequence of n elements into two sub-sequences of n/2 elements
    - suppose previously introduced adjacent exchange sorting methods are used
    - sort the first sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - sort the second sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - merge the two sub-sequences: O(n)

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be "merged" efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **complexity of merging: O(n)**
  - divide a sequence of n elements into two sub-sequences of n/2 elements
    - suppose previously introduced adjacent exchange sorting methods are used
    - sort the first sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - sort the second sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - merge the two sub-sequences: $O(n)$
    - total complexity: $O(n^2/4)+O(n^2/4)+O(n)=O(n^2/2+n) < O(n^2)$

divide & conquer brings efficiency enhancement

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be "merged" efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **complexity of merging: O(n)**
  - divide a sequence of n elements into two sub-sequences of n/2 elements
    - suppose previously introduced adjacent exchange sorting methods are used
    - sort the first sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - sort the second sub-sequence: $O((n/2)^2)=O(n^2/4)$
    - merge the two sub-sequences: $O(n)$
    - total complexity: $O(n^2/4)+O(n^2/4)+O(n)=O(n^2/2+n) < O(n^2)$
  - divide sub-sequences further into sub-sub-sequences, sub-sub-sub-sequences ... ...

basic spirit of mergesort

Sorting

- **Mergesort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be “merged” efficiently
  - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **complexity of merging: O(n)**
  - divide a sequence of n elements into two sub-sequences of n/2 elements
  - divide sub-sequences further into sub-sub-sequences, sub-sub-sub-sequences ... ...

unsorted	89	11	43	71	37	23	67	59
merge every two	11	89	43	71	23	37	59	67
merge every four	11	43	71	89	23	37	59	67
merge every eight (all)	11	23	37	43	59	67	71	89

Sorting

- **Mergesort - divide & conquer**
  - divide into sub-tasks that are much easier and can be "merged" efficiently

```
// merge sort: divide & conquer
#define MERGESORT_SMALL_T 3
template <class T,class P> void mergesort(T s[],int iL,int iR,T tmp[],int depth){
        if((iR-iL)<=MERGESORT_SMALL_T){ // insertionsort for small sub-sequences
                insertionsort<T,P>(&s[iL],iR-iL);return;}
        int iM=(iL+iR)/2,i,j,k;
        // divide into two sub-sequences for sorting respectively
        mergesort<T,P>(s,iL,iM,tmp,depth-1);mergesort<T,P>(s,iM,iR,tmp,depth-1);
        if(depth>0) return; // not show all but show partial results of mergesort
        for(i=iL;i<iM;i++) tmp[i]=s[i]; // copy the first sub-sequence
        for(i=iM,j=iR;i<iR;i++) tmp[--j]=s[i]; // copy the second sub-sequence
        for(i=iL,j=iR-1,k=iL;k<iR;k++) // merge the two sorted sub-sequences
                if (P::p(tmp[i],tmp[j])) s[k]=tmp[i++]; else s[k]=tmp[j--];
}
template <class T,class P> void mergesort(T s[],int iL,int iR,T tmp[]){ // [iL,iR)
        mergesort<T,P>(s,iL,iR,tmp,0);} // mergesort sequence of s[iL] to s[iR-1]
// END merge sort: divide & conquer
```

Sorting

- **Mergesort - divide & conquer**
  - divide into sub-tasks that are much easier and can be “merged” efficiently

```
cout<<"DEMO : merge sort =>\n";int itmp[ni];cptr ctmp[nc];
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"merge sort: ";mergesort<int,IntPriorMin>(iA,0,ni,itmp,3);show<int>(iA,ni);
cout<<"merge sort: ";mergesort<int,IntPriorMin>(iA,0,ni,itmp,2);show<int>(iA,ni);
cout<<"merge sort: ";mergesort<int,IntPriorMin>(iA,0,ni,itmp,1);show<int>(iA,ni);
cout<<"merge sort: ";mergesort<int,IntPriorMin>(iA,0,ni,itmp);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"merge sort: ";mergesort<cptr,CharsPriorMin>(cA,0,nc,ctmp);show<cptr>(cA,nc);
```

```
DEMO : merge sort =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
merge sort: 42 92 79 96 4 93 66 85 68 76 63 74 17 39 3 71
merge sort: 42 79 92 96 4 66 85 93 63 68 74 76 3 17 39 71
merge sort: 4 42 66 79 85 92 93 96 3 17 39 63 68 71 74 76
merge sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
merge sort: automation computer data intelligence machine program system technique
```

THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY