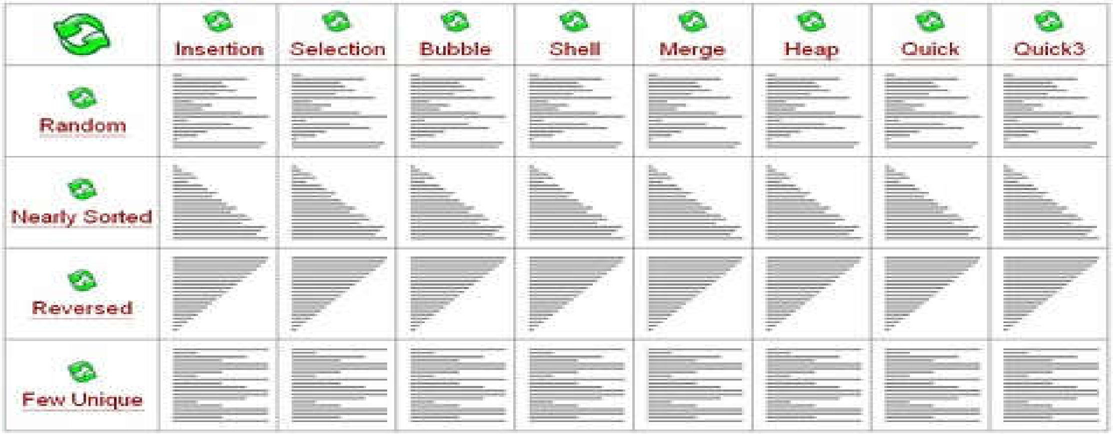
Data Structures and Algorithms

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com



Sorting

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com

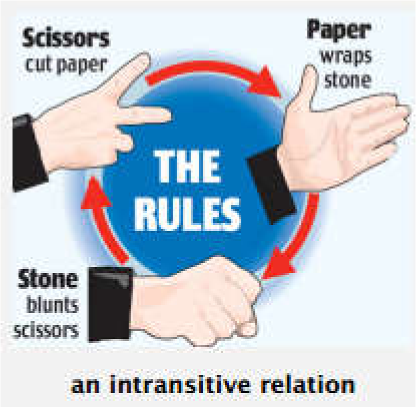
Sorting

- Sorting is one of the most fundamental algorithmic problems within computer science.
- It has been claimed that as many as 25% of all CPU cycles are spent sorting

- One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.
- Sorting is fundamental to most other algorithmic problems, for example binary search.

What is sorting?

- It is the problem of taking an arbitrary permutation of *n* items and rearranging them into the total order.
- A **total order** is a binary relation $\leq$ that satisfies properties below:
  - Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$.
  - Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$.
  - Totality: either $v \leq w$ or $w \leq v$ (comparable)
- Example:
  - Standard order for natural and real numbers.
  - Chronological order for dates or times.
  - Alphabetical order for strings
  - …



Issues in Sorting

- Increasing or Decreasing Order?
  - The same algorithm can be used by both
  - all we need do is to change in the comparison function as we desire.
- What about equal keys?
  - Does the order matter or not?
  - Maybe we need to sort on secondary keys.
- We can assume a comparison function which depends on the application.
  - Compare (a, b) should return '<', '>', or '='.

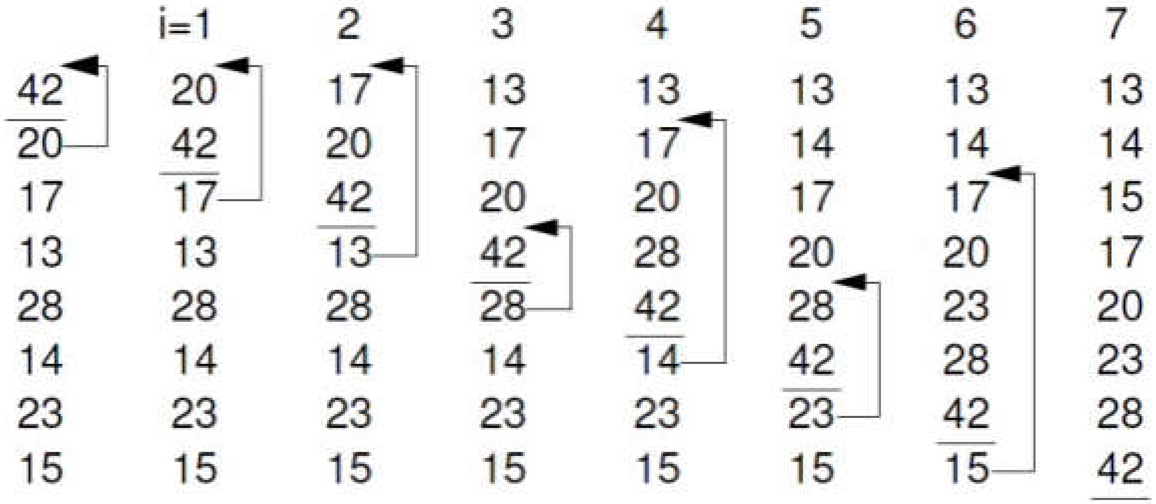
How do you sort? Sort strategies

- There are several different ideas which lead to sorting algorithms:
- Insertion - putting an element in the appropriate place in a sorted list yields a larger sorted list.
- Exchange - rearrange pairs of elements which are out of order, until no such pairs remain.
- Selection - extract the largest element from the list, remove it, and repeat.
- Distribution - separate into piles based on the first letter, then sort each pile. (e.g. contact list in your mobile)
- Merging - Two sorted lists can be easily combined to form a sorted list.

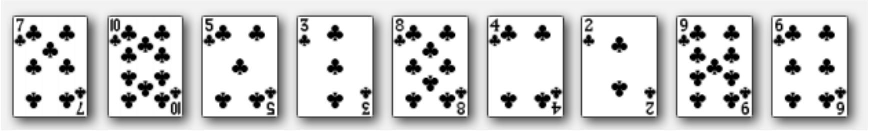
Insertion Sort

Insertion Sort

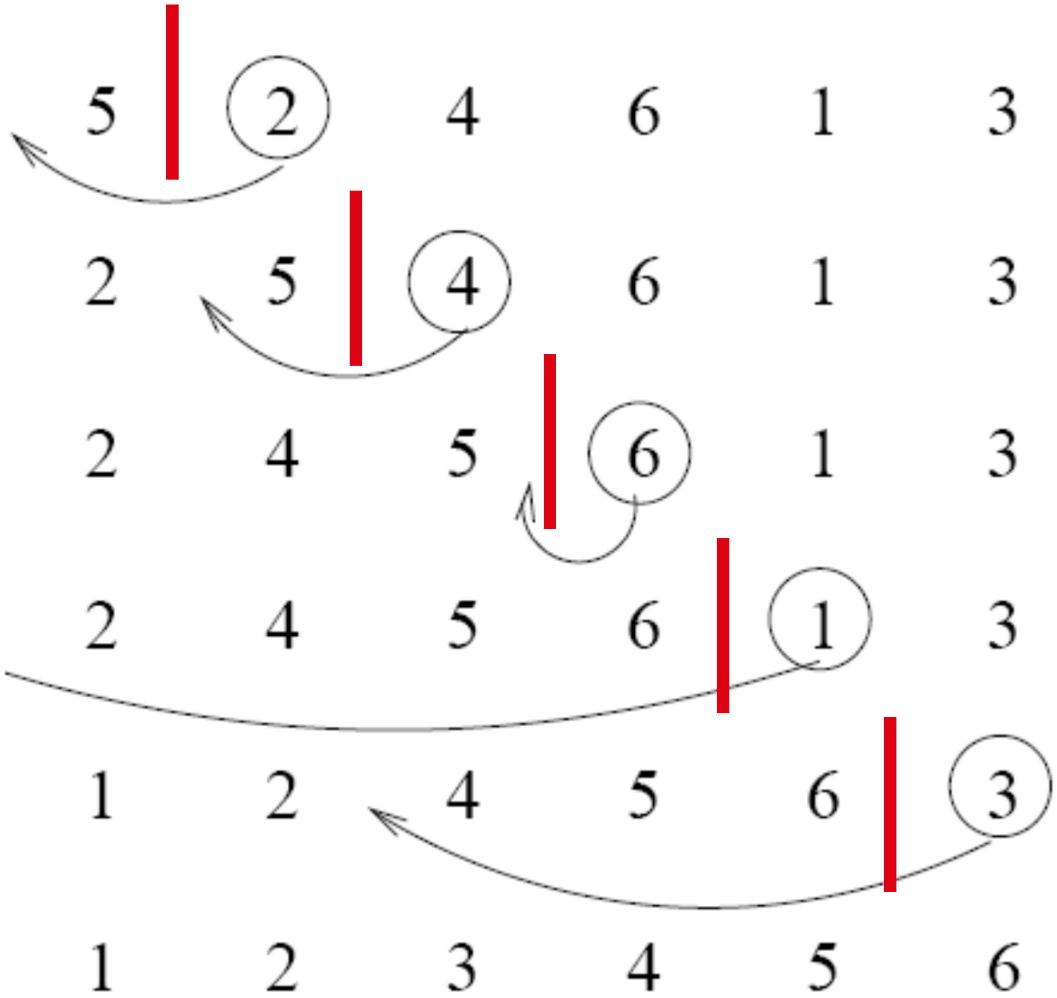
- In insertion sort, we repeatedly add elements to a sorted subset of our data, inserting the next element in order:

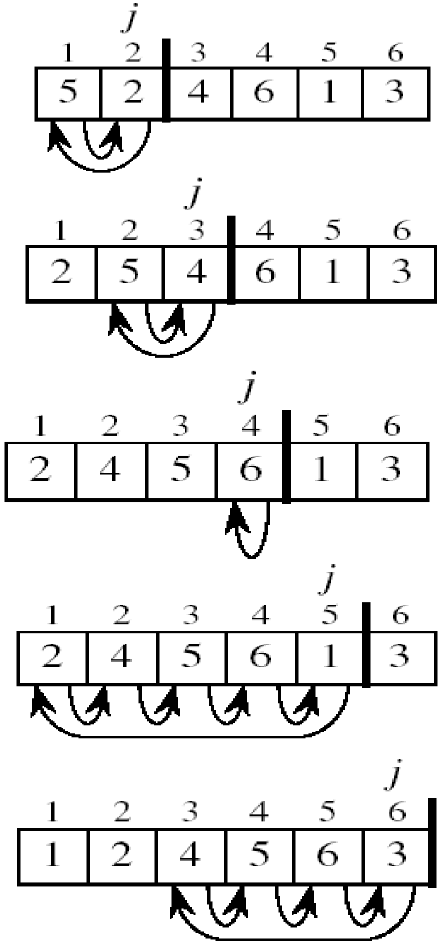


	i=1	2	3	4	5	6	7
42	20	17	13	13	13	13	13
20	42	20	17	17	14	14	14
17	17	42	20	20	17	17	15
13	13	13	42	28	20	20	17
28	28	28	28	42	28	23	20
14	14	14	14	14	42	28	23
23	23	23	23	23	23	42	28
15	15	15	15	15	15	15	42



Insertion Sort





Insertion Sort

- If the elements are in **an array** , we scan from bottom to top until we find the j such that

$$A[j] \leq x \leq A[j+1]$$

then move from j+1 to the end down one to make room.

- If the elements are in **a linked list** , we do the sequential search until we find where the element goes,

  then insert the element there.
  No other elements need move!

Insertion Sort : Complexity

- The best, worst, and average cases for insertion sort all differ.
- We do not necessarily have to scan the entire sorted section of the array.
- **Best case** :
  - the element always gets inserted at the end, so we don't have to move anything, and only compare against the last sorted element.
  - We have (n-1) insertions, each with exactly one comparison and no data moves per insertion!
- What is this best case permutation?
  - It is when the array or list is already sorted!

Worst Case Complexity

- Worst case:
  - The element always gets inserted at the front,
  - so all the sorted elements must be moved at each insertion.
  - The ith insertion requires (i-1) comparisons and moves so:

$$\sum_{i=1}^{n}(i-1) = n(n-1)/2 = O(n^2)$$

- What is the worst case permutation?
  - When the array is sorted in reverse order.

Average Case Complexity

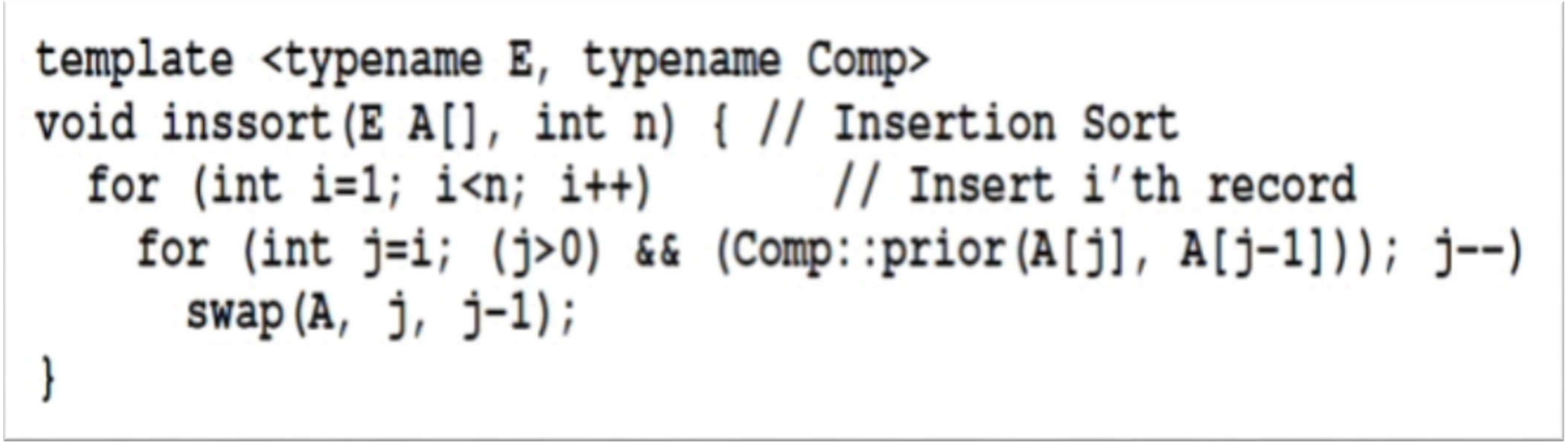
- Average Case :
  - If we were given a random permutation, the chances of the ith insertion requiring 0,1,2,…,(i-1) comparisons are equal, and hence 1/i.
  - The expected number of comparisons is for the ith insertion is:

$$\sum_{k=1}^{i}\frac{(k-1)}{i}=\frac{1}{i}\sum_{k=1}^{i-1}k=\frac{1}{i}\times\frac{i(i-1)}{2}=\frac{i-1}{2}$$

  - Summing up over all n keys,

$$\sum_{i=1}^{n}\frac{i-1}{2}=\frac{1}{2}\left(\frac{n(n-1)}{2}+\frac{n}{2}\right)=\frac{n^2}{4}+O(n)$$

Insertion Sort: Implementation

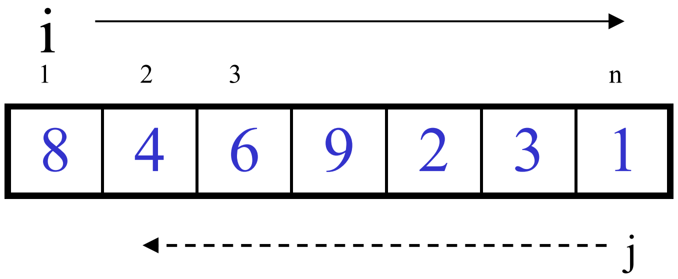


```
template <typename E, typename Comp>
void inssort(E A[], int n) { // Insertion Sort
  for (int i=1; i<n; i++)          // Insert i'th record
    for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
      swap(A, j, j-1);
}
```

Bubble Sort

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



i

1	2	3				n
8	4	6	9	2	3	1

j

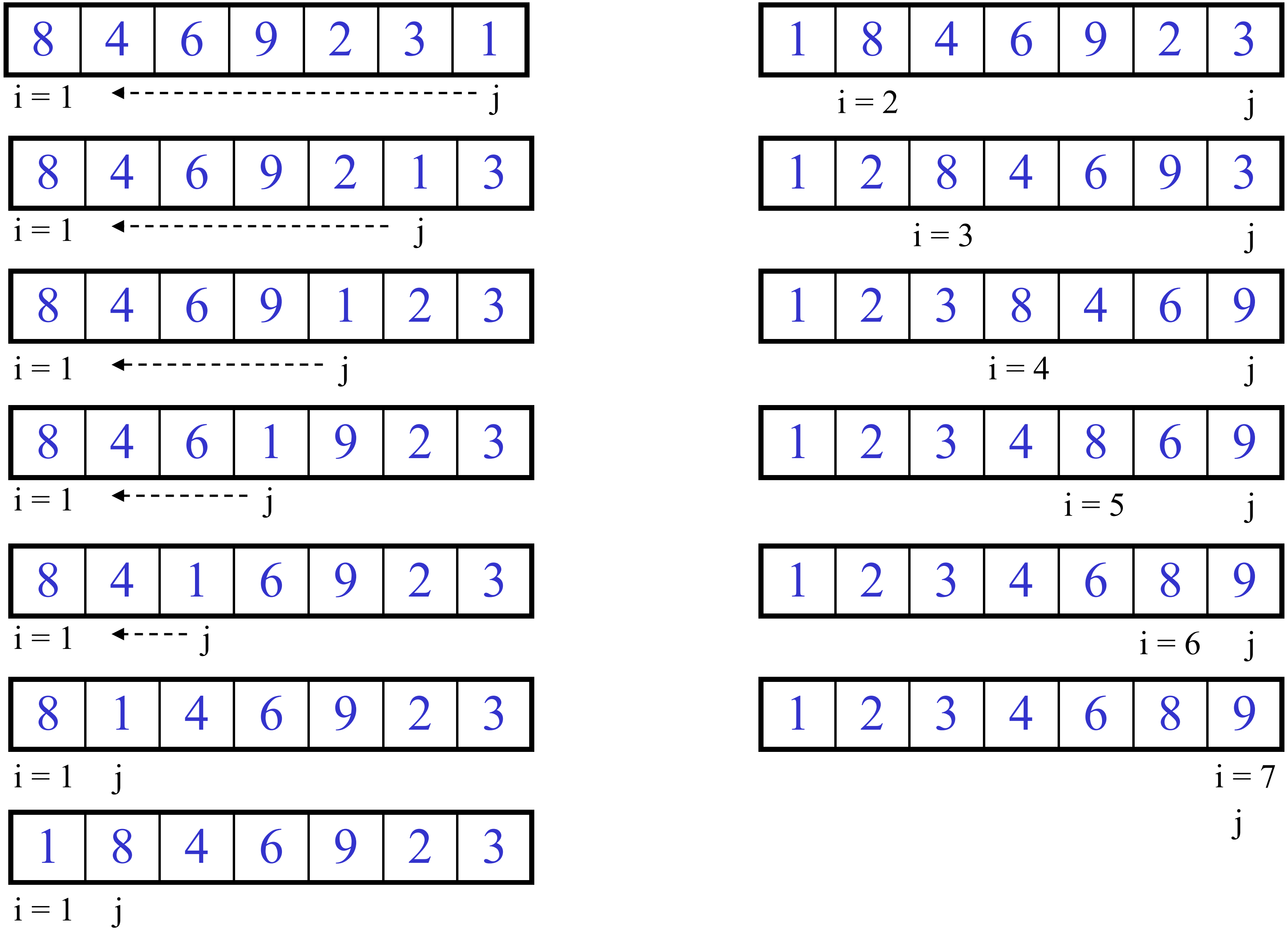
- Easier to implement, but slower than Insertion sort

Bubble Sort

- Simplest sorting algorithm
- **Steps**:

```
1. Set flag = false
2. Traverse the array and compare pairs of two elements
   2.1 If  $E1 \le E2$  - OK
   2.2 If  $E1 > E2$  then
          Swap(E1, E2)  and set flag = true
3. If flag = true goto 1.
```

Bubble Sort: Example



8	4	6	9	2	3	1

i = 1 ←- j

8	4	6	9	2	1	3

i = 1 ←- - - - - - - - - - - - - - - - - - j

8	4	6	9	1	2	3

i = 1 ←- - - - - - - - - - - - - - j

8	4	6	1	9	2	3

i = 1 ←- - - - - - - - - j

8	4	1	6	9	2	3

i = 1 ←- - - - j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

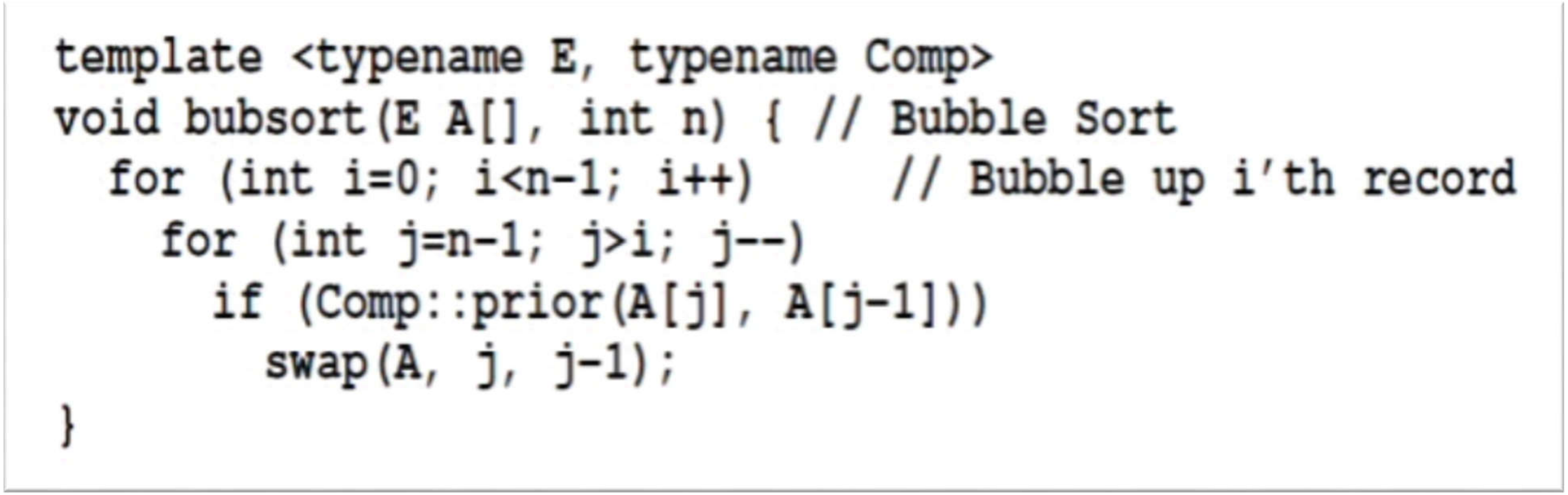
i = 6 j

1	2	3	4	6	8	9

i = 7

j

Bubble Sort: Implementation



```
template <typename E, typename Comp>
void bubsort(E A[], int n) { // Bubble Sort
  for (int i=0; i<n-1; i++)     // Bubble up i'th record
    for (int j=n-1; j>i; j--)
      if (Comp::prior(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

Bubble Sort

- Demo (Bubble-sort with Hungarian (-Csángó-) folk dance) and
- consider its time complexity

Bubble Sort: Complexity

- One traversal = move the maximum element at the front
- Traversal #i: n – i operations

- Running time:

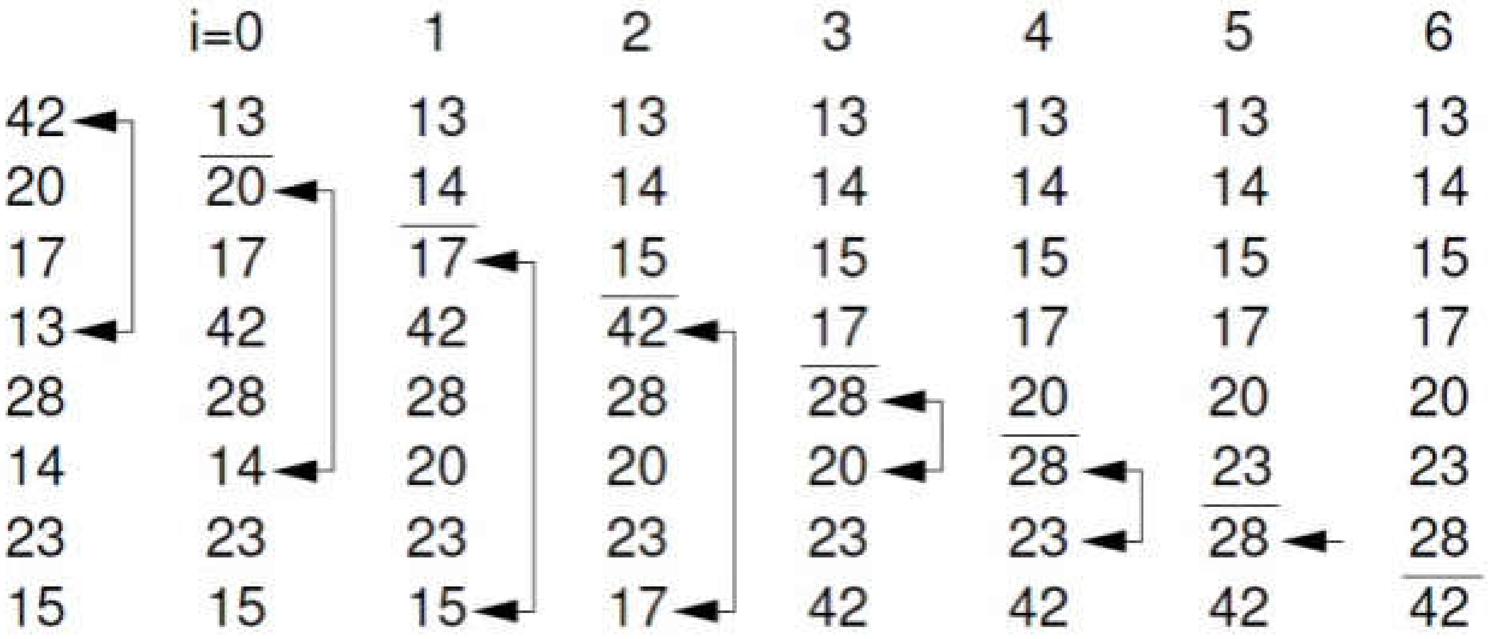
$$(n - 1) + (n - 2) + \ldots + 1 = (n - 1)\, n / 2 = O(n^{2})$$

- Bubble Sort's running time is roughly the same in the best, average, and worst cases

Selection Sort

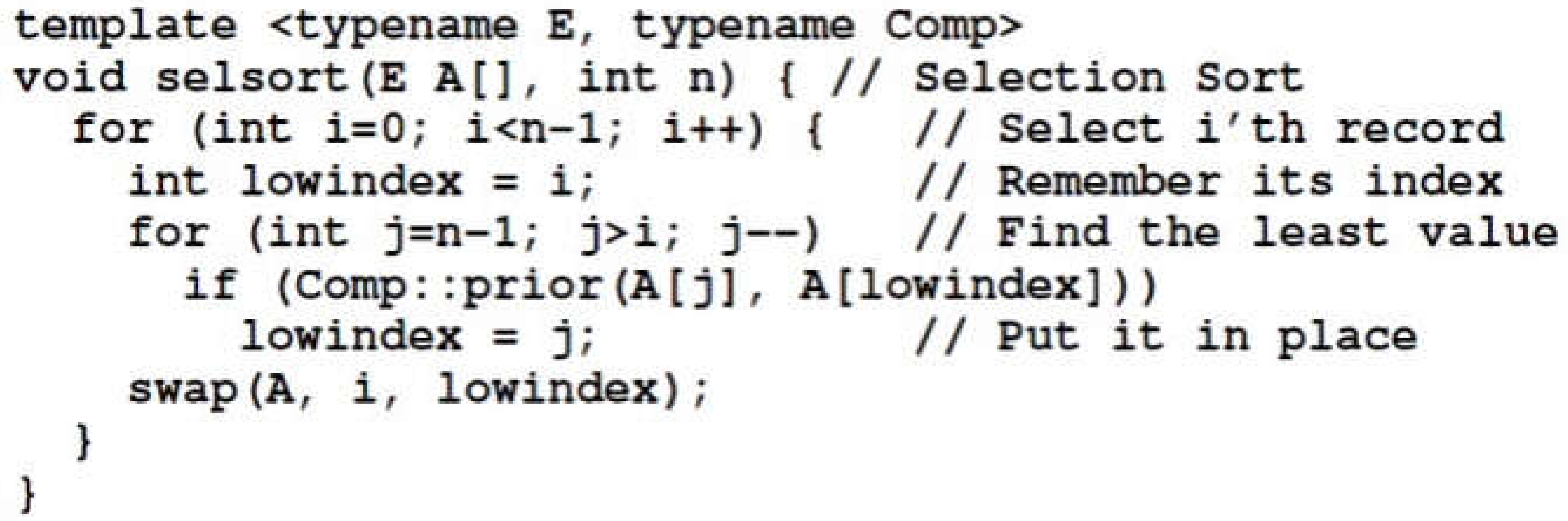
Selection Sort

- The most natural and easiest sorting algorithm is selection sort ,
  - where we repeatedly find the smallest element,
  - move it to the front, then repeat...



	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Selection Sort : Implementation



```
template <typename E, typename Comp>
void selsort(E A[], int n) { // Selection Sort
  for (int i=0; i<n-1; i++) {   // Select i’th record
    int lowindex = i;           // Remember its index
    for (int j=n-1; j>i; j--)   // Find the least value
      if (Comp::prior(A[j], A[lowindex]))
        lowindex = j;           // Put it in place
    swap(A, i, lowindex);
  }
}
```

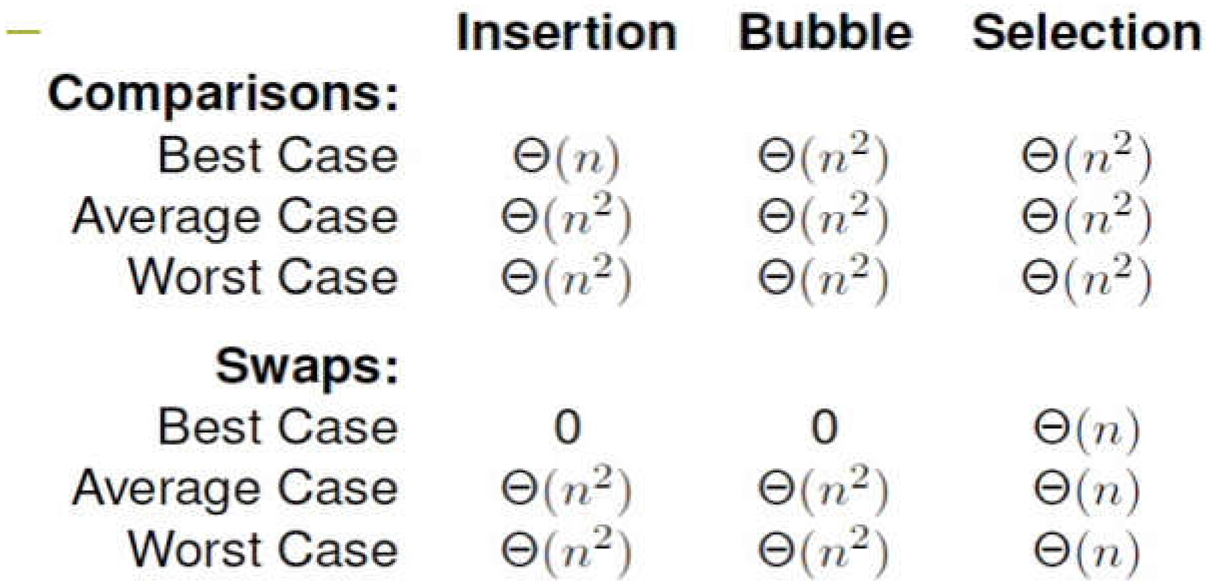
- If elements are **in an array**,
  - swap the first with the smallest element- thus only one array is necessary.
- If elements are **in a linked list**,
  - we must keep two lists, one sorted and one unsorted, and
  - always add the new element to the back of the sorted list.

Selection Sort : Complexity

- The best case, worst case, and average cases are all the same!
- To find the largest takes (n-1) steps, to find the second largest takes (n-2) steps, …, so, takes totally

$$\sum_{i=1}^{n-1} i = n(n-1)/2 = n^2/2 + n/2 = O(n^2)$$

Summary: exchange sorts



	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

- Swapping adjacent records is called an exchange.
- Thus, these sorts are sometimes referred to as **exchange sorts**.

Why exchange sorts are slow?

- The cost of any exchange sort can be at best the **total number of steps** that the records in the array must move to reach their "correct" location (i.e., the number of inversions for each record).
- **Inversion**: a pair (i,j) such that i<j, but Array[i] > Array[j]
- Array of size N
  - average number of inversions in a random set of elements is N(N-1)/4
- Exchange Sort only swaps adjacent elements
  - only removes 1 inversion!

Shellsort

Shellsort (diminishing increment sort)

- Shellsort is also known as *diminishing increment sort*.
- Invented by Donald Shell in 1959.
- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven
- Shellsort works by comparing elements that are distant rather than adjacent elements in an array.

Shellsort (diminishing increment sort)

- The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared

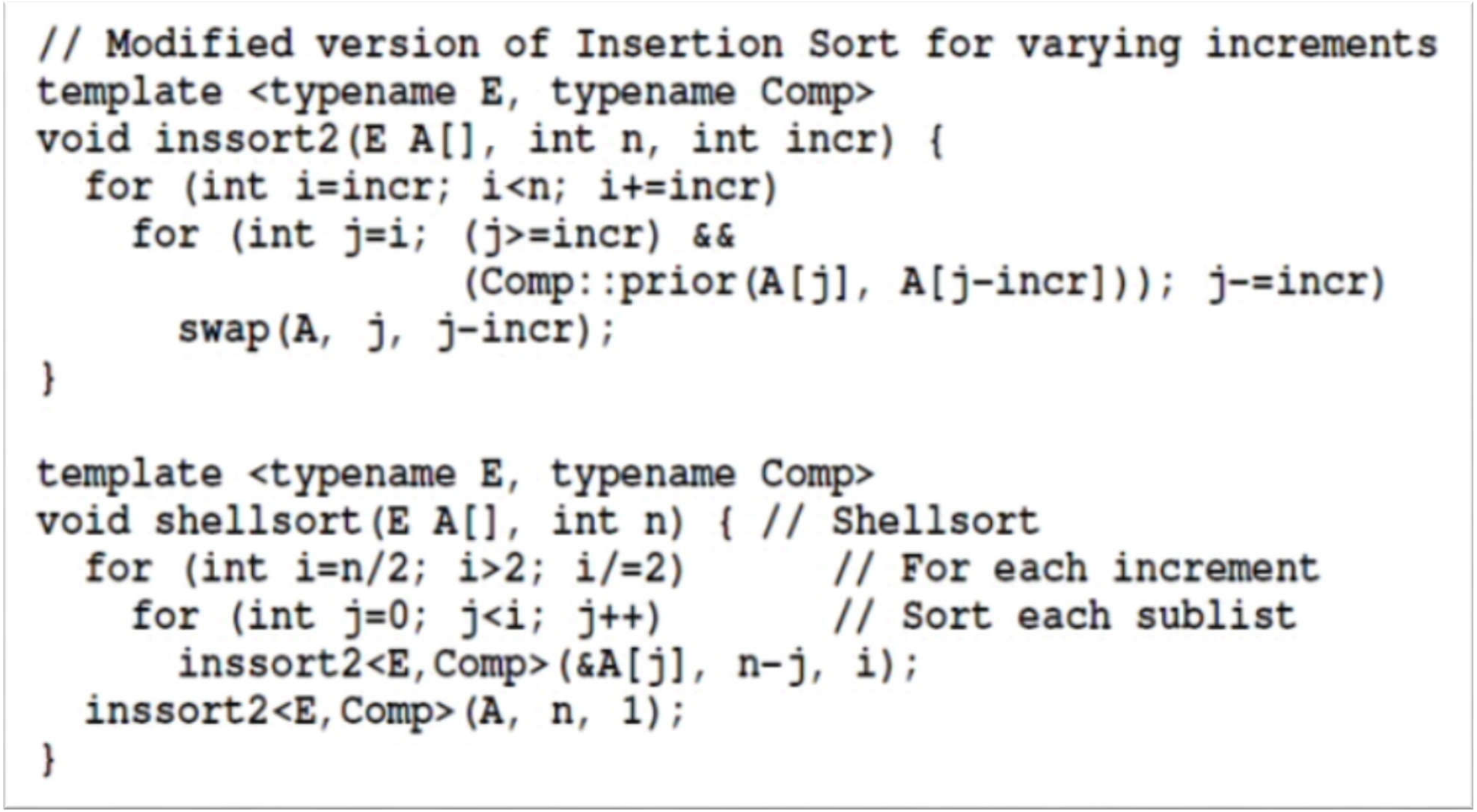
Shellsort (diminishing increment sort)

- Shellsort makes comparisons and swaps between **non-adjacent** elements
- Shellsort ’s strategy is to make the list “mostly sorted ” so that a final Insertion Sort can finish the job
- Idea
  - Shellsort breaks the array of elements into “virtual ” sublists according to the increment.
  - Each sublist is sorted using an Insertion Sort.
  - Another group of sublists is then chosen and sorted,
  - and so on.

Example demo

	1	2	3	4	5	6	7	8	9
Initial list	40	25	49	25*	16	21	08	30	13
d = 4	40	25	49	25*	16	21	08	30	13
	13	21	08	25*	16	25	49	30	40
d = 2	13	21	08	25*	16	25	49	30	40
	08	21	13	25*	16	25	40	30	49
d = 1	08	21	13	25*	16	25	40	30	49
	08	13	16	21	25*	25	30	40	49

Shellsort: Implementation



```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i; (j>=incr) &&
                  (Comp::prior(A[j], A[j-incr])); j-=incr)
      swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2)      // For each increment
    for (int j=0; j<i; j++)       // Sort each sublist
      inssort2<E,Comp>(&A[j], n-j, i);
  inssort2<E,Comp>(A, n, 1);
}
```

Shellsort: Complexity

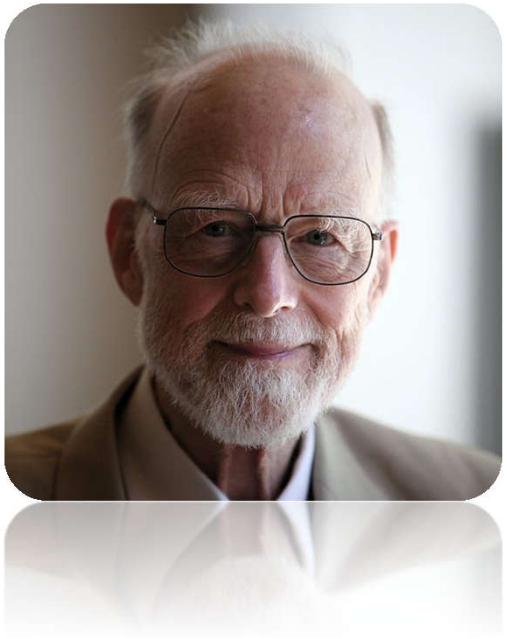
- The average-case performance of Shellsort (for “divisions by three ” increments) is O($n^{1.5}$).

- Shellsort is substantially better than Insertion Sort, or any of exchange sorts (O(n^2)).

Quicksort

Quicksort

- Sir Charles Antony Richard Hoare (born 11 January 1934), commonly known as Tony Hoare or C. A. R. Hoare
- He developed the sorting algorithm Quicksort in 1960 at age 26.
- He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP)
- Won ACM Turing Award for "fundamental contributions to the definition and design of programming languages" on 27 October 1980
- For many years under his leadership his Oxford department worked on formal specification languages such as CSP and Z.

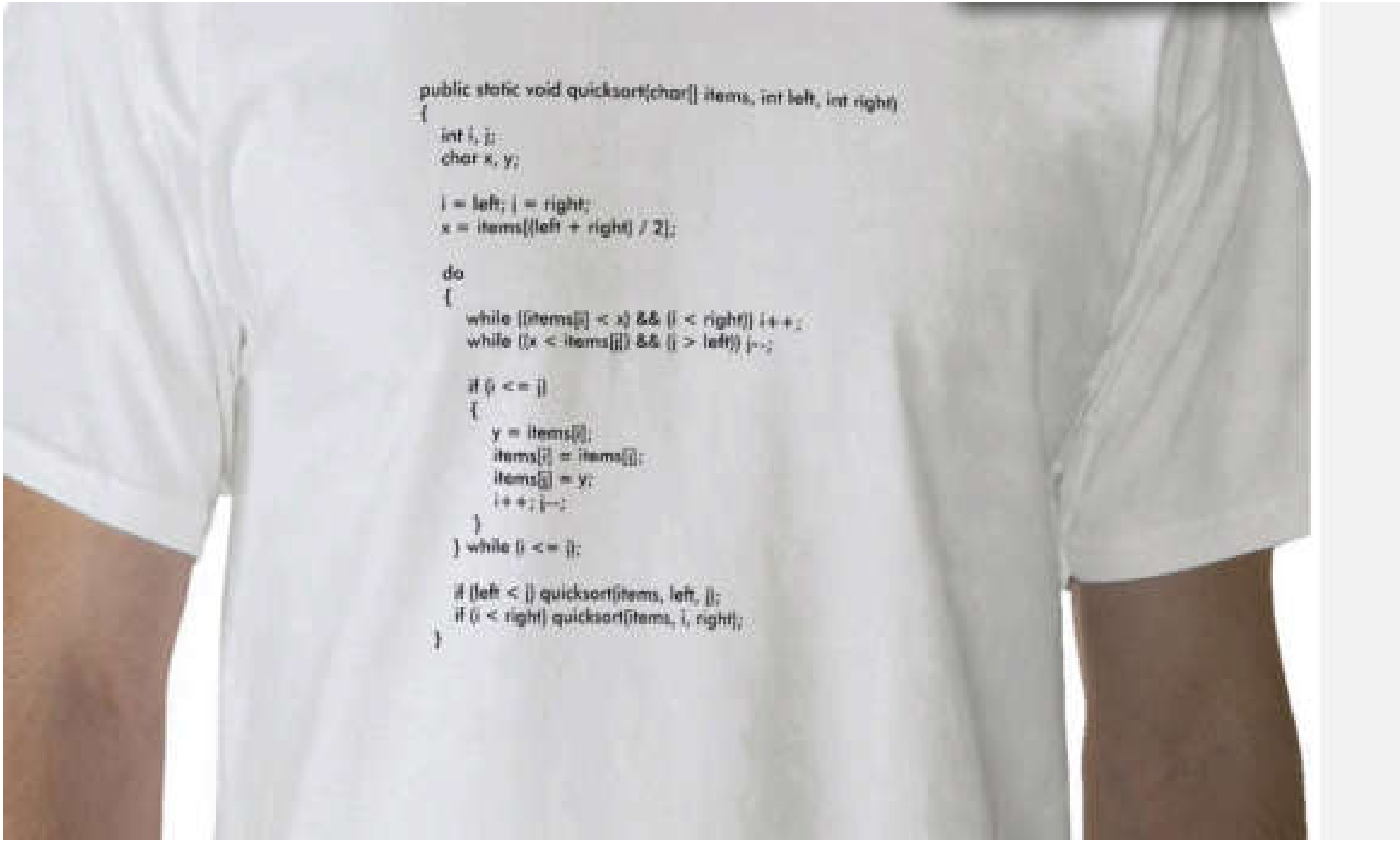


Quicksort

- Quicksort when properly implemented, is the fastest known general-purpose in-memory sorting algorithm in the average case.

- It does not require the extra array needed by Mergesort, so it is space efficient as well.

Quicksort t-shirt

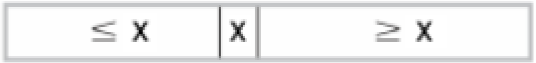


```
public static void quicksort(char[] items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[(left + right) / 2];

   do
   {
      while ((items[i] < x) && (i < right)) i++;
      while ((x < items[j]) && (j > left)) j--;

      if (i <= j)
      {
         y = items[i];
         items[i] = items[j];
         items[j] = y;
         i++; j--;
      }
   } while (i <= j);

   if (left < j) quicksort(items, left, j);
   if (i < right) quicksort(items, i, right);
}
```

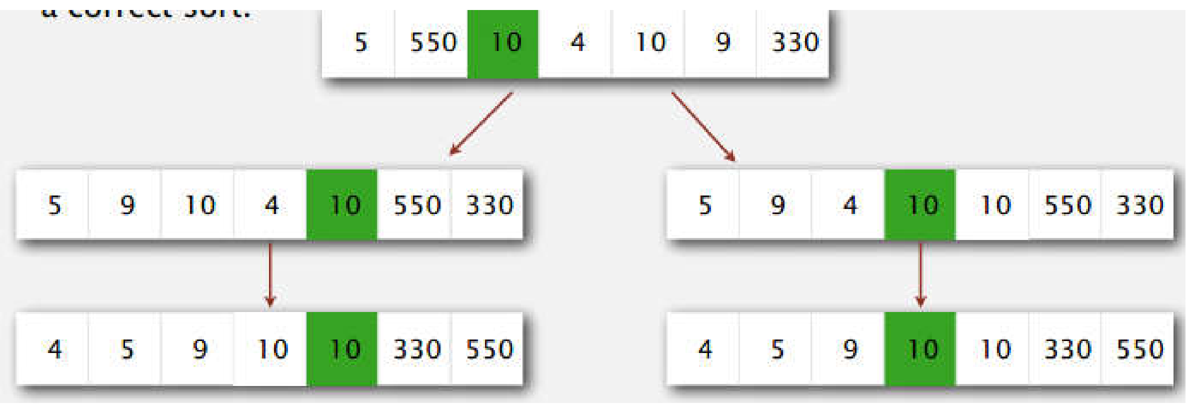
Partitioning - the heart of Quicksort

Partitioning

- To partition an array a[] on element x=a[i] is to rearrange it such that
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x.
  - All entries to the right of x are >= x.



- Observations.
  - O1. After partitioning , x is ‘in place.’ No need to move x in order to reach a correct sort.



a correct sort.

5 550 10 4 10 9 330

5 9 10 4 10 550 330

5 9 4 10 10 550 330

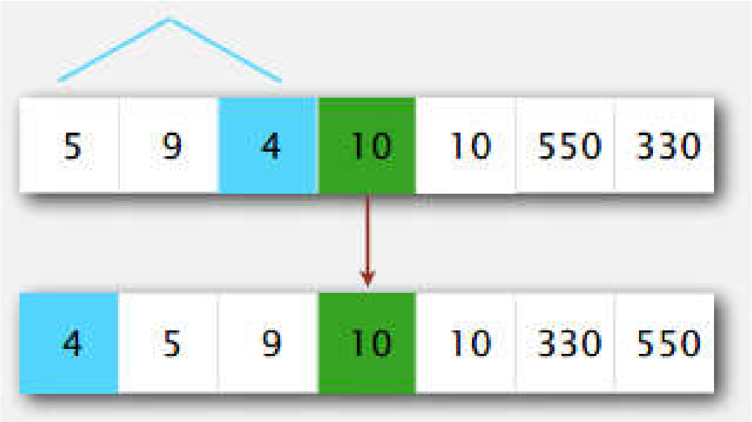
4 5 9 10 10 330 550

4 5 9 10 10 330 550

Partitioning - the heart of Quicksort

Partitioning

- Observations.
  - O1. After Partitioning , x is ‘in place.’ No need to move x in order to reach a correct sort.
  - O2. To Partition on a new pivot y, there’s no need to look beyond the confines of previously partitioned items.



5 9 4 10 10 550 330

4 5 9 10 10 330 550

Quicksort: Idea

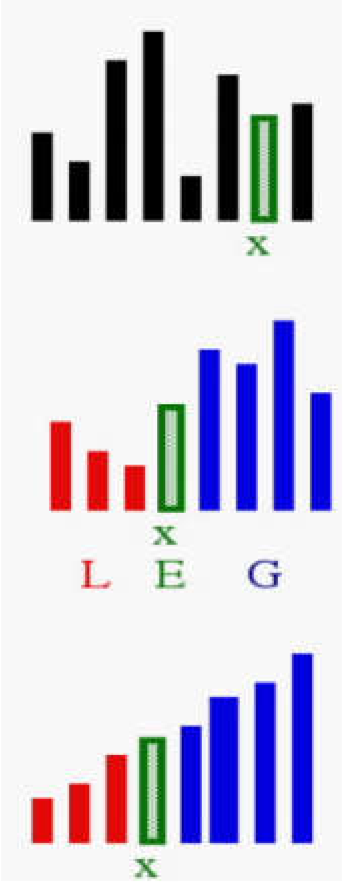
1) Select: pick an element

2) Divide:

rearrange elements so that x goes to its final position E

3) Recurse and Conquer:

recursively sort

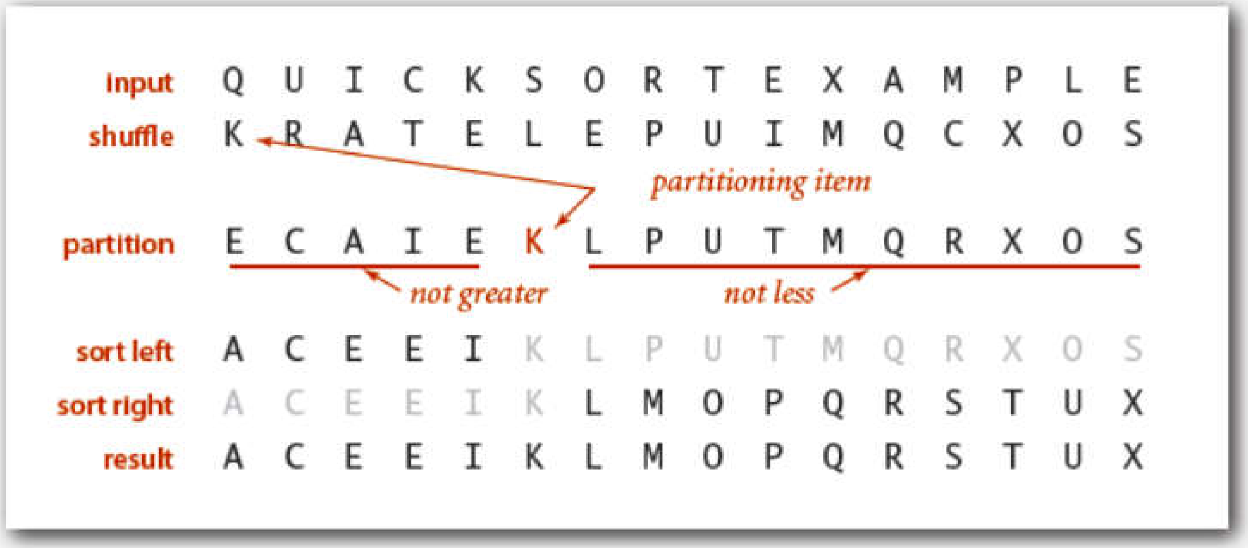


Quicksort: Idea

Basic plan.

- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each piece recursively.

Sir Charles Antony Richard Hoare
1980 Turing Award

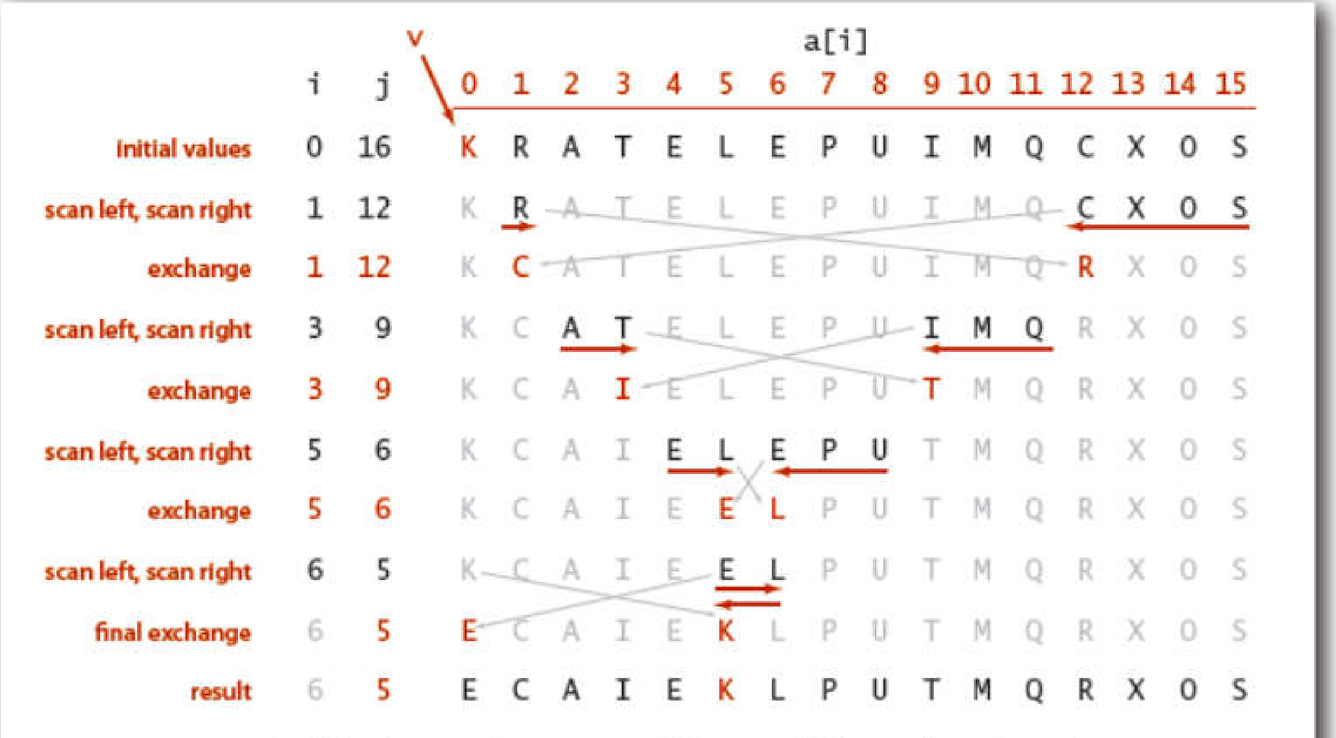


input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicsort partitioning

Basic plan.

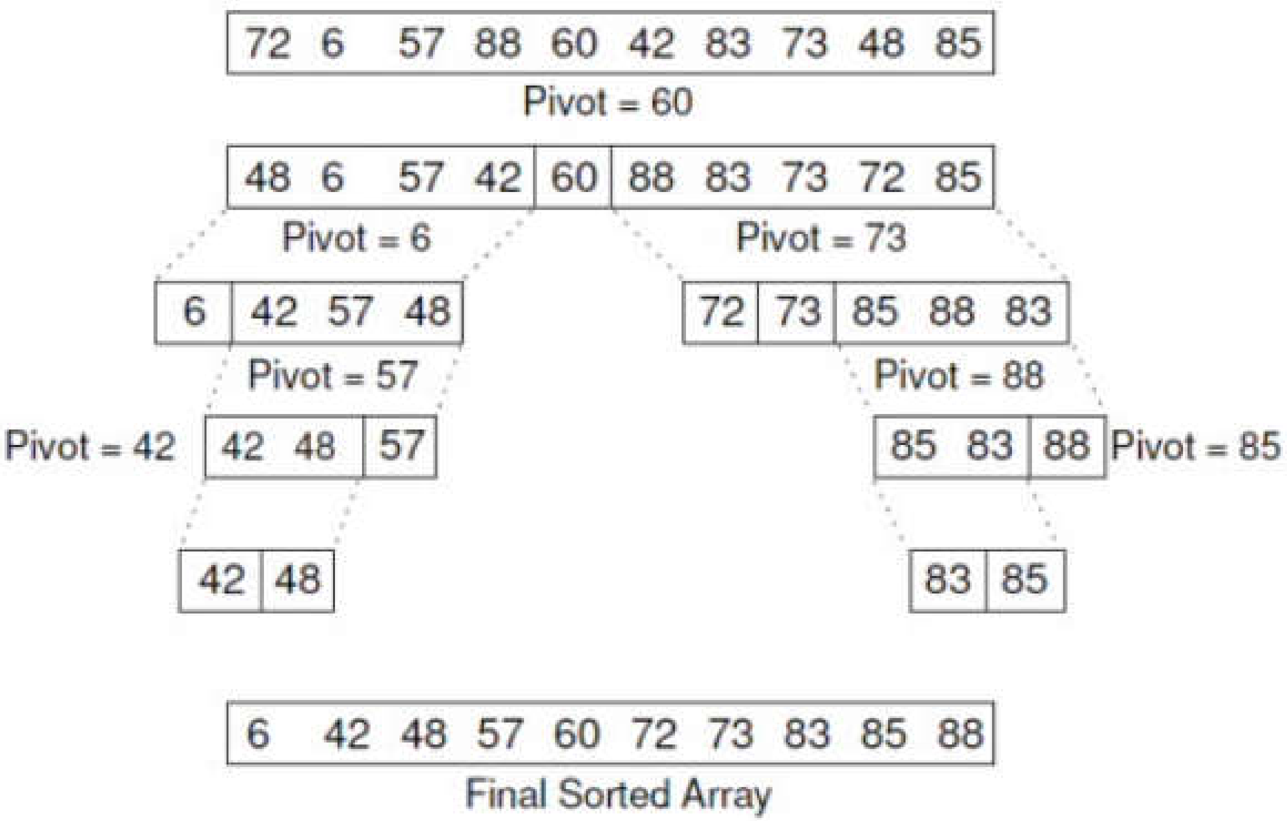
- Scan i from left for an item that belongs on the right.
- Scan j from right for an item that belongs on the left.
- Exchange a[i] and a[j].
- Repeat until pointers cross.



	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Quicksort: Example



72 6 57 88 60 42 83 73 48 85

Pivot = 60

48 6 57 42 | 60 | 88 83 73 72 85

Pivot = 6

Pivot = 73

6 | 42 57 48

72 | 73 | 85 88 83

Pivot = 57

Pivot = 88

Pivot = 42

42 48 | 57

85 83 | 88

Pivot = 85

42 | 48

83 | 85

6 42 48 57 60 72 73 83 85 88

Final Sorted Array

Quicksort Implementatio

```
int   FindPivot( int  i, int  j ) /* A is an extra array */
/* if A[i],…A[j] are the same， then return 0；
else， the bigger one。 */
{    keytype   firstkey = A[i].key ; /* */
     int  k ;
     for ( k=i+1 ;  k<=j; k++ ) /*
            if ( A[k].key > firstkey )
                        return  k ;
            else if ( A[k].key < firstkey )
                        return  i ;
     return  0 ;
}
```

Quicksort Implementation

```
int Partition ( int i , int j , keytype pivot )
/* partition A[i],…,A[j]*/
{   int l , r ;
    do{
        for( l = i ; A[l].key  <  pivot ; l++ ) ;
        for( r = j ; A[l].key >= pivot ; r--)  ;
        if( l < r )  swap(A[l],A[r]);
    } while( l <= r );
    return   l ;
}
```

Quicksort Implementation

```
void QuickSort ( int i , int  j )
{    keytype pivot;
     int k; //
     int pivotindex ; //
     pivotindex = FindPivot ( i , j );
     if( pivotindex != 0 ) {  //
             pivot=A[pivotindex].key;
             k=Partition ( i , j , pivot );
             QuickSort ( i , k-1);
             QuickSort ( k   , j );
   }
}
```

Partitioning - the heart of Quicksort

- Demo 2: comparison bubble sort and quick sort (Visualization of Quick sort.mp4)

- And think about the time complexity of quicksort.

Time complexity

- Best case
  - – Each partition, the length of the sub-lists are the same.

$$T(n) \le 2T(n/2) + n$$

$$\le 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$$

$$\le 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$$

$$\ldots \ldots \ldots$$

$$\le nT(1) + n\log_2 n = O(n\log_2 n)$$

- ***Time complicity $O(n^2)$***
- ***Time complicity $O(n)$***

Time complexity

- Best case
  - Each partition, the length of the sub-lists are the same.

$$
\begin{aligned}
T(n) &\le 2T(n/2)+n \\
&\le 2(2T(n/4)+n/2)+n=4T(n/4)+2n \\
&\le 4(2T(n/8)+n/4)+2n=8T(n/8)+3n \\
&\dots\ \dots\ \dots \\
&\le nT(1)+n\log_2 n=O(n\log_2 n)
\end{aligned}
$$

- ***Time complicity*** $O(n\log_2 n)$
- ***Space complicity*** $O(n)$

Time complexity

- Worst case
  - when the pivot does not divide the sequence in two
  - At each step, the length of the sequence is only reduced by 1
  - Total running time

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

  - Time complicity $O(n^2)$
  - Space complicity $O(n)$

Time complexity

- General/Average case:
  - Time spent at level i in the tree is O(n)
  - Running time: O(n) * O(height)

  - Time complexity $O(n\log_2 n)$
  - Spatial complexity $O(n)$

Mergesort

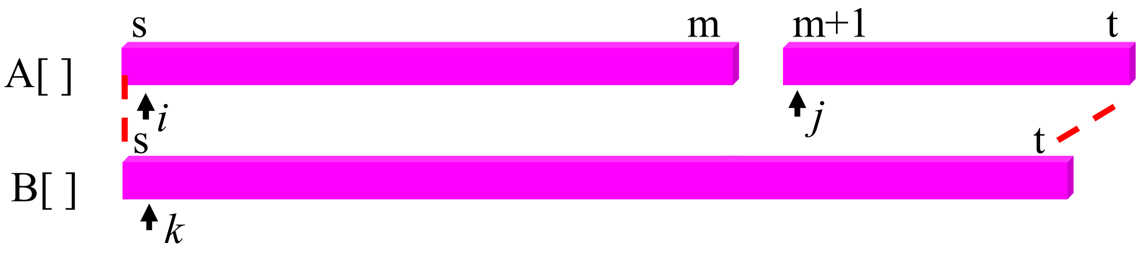
Mergesort

- Mergesort is one of the simplest sorting algorithms conceptually, and
- has good performance both in the asymptotic sense and in empirical running time.

- Basic idea is to merge two or more ordered lists into one ordered list.

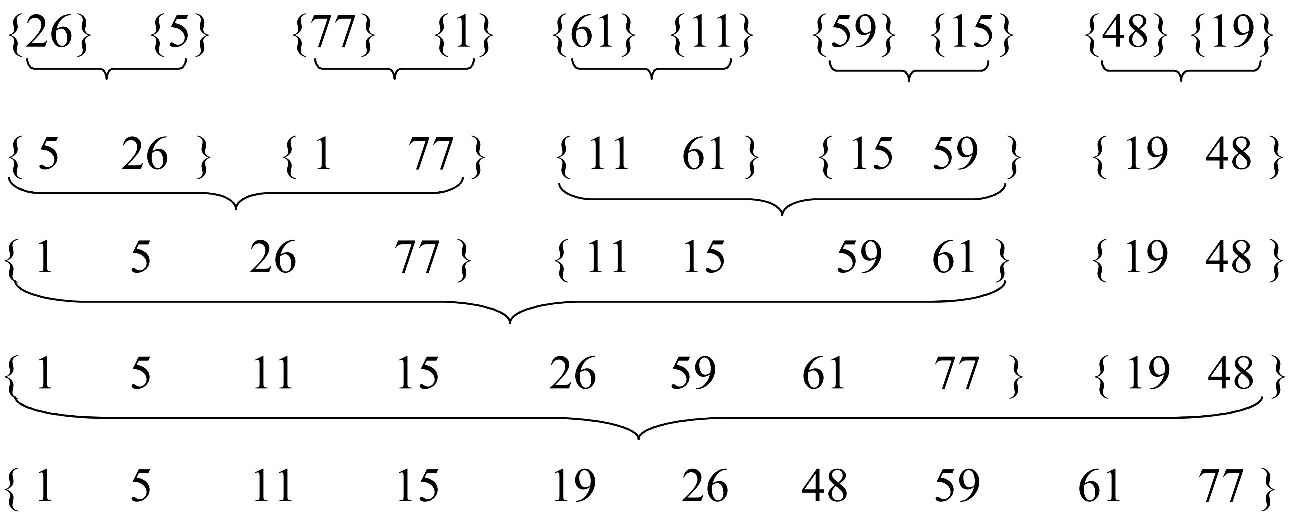
Merge order

```
void Merge (int s , int m , int t , LIST  A , LIST  B)
/*Merge ordered lists of A[s],…,A[m] and A[m+1],…,A[t] into ordered one of
      B[s],…,B[t]*/
{     int i = s ;  j = m+1 , k = s ;//initialization
         /* if not empty, place smaller into B[k] */
      while ( i <= m && j <= t )
           B[k++] = ( A[ i ].key <= A[ j ].key) ? A[i++] : A[j++] ;
         /* if the first list is not empty, copy the left into B */
       while ( i <= m )    B[k++] = A[i++] ;
         /* if the second list is not empty, copy the left into B */
       while ( j <= t )     B[k++] = A[j++] ;
}/*time complex. : O( t - s + 1 ) ; space complex. : O( t - s + 1 ) */
```



Merge order: non recursive

- Idea of two-way mergesort
  - Firstly regard the list of n records as n lists of 1 record
  - Merging each two lists, we have $\lceil n/2 \rceil$ ordered lists of 2 records
  - Merging each two lists, we have $\lceil n/4 \rceil$ ordered lists of 4 records
  - ……
  - At last, we have a ordered list of n records.



{26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

{ 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 }

{ 1 5 26 77 } { 11 15 59 61 } { 19 48 }

{ 1 5 11 15 26 59 61 77 } { 19 48 }

{ 1 5 11 15 19 26 48 59 61 77 }

Merge order: non recursive algorithm

Idea for one pass: merge the two adjacent lists which length is *h* in new ordered list which length is *2h*

```
void  MergePass (int n , int h , LIST A , LIST B)
{    int i , t ;
     for ( i=1 ; i+2*h-1<= n ; i+=2*h )
             Merge(i, i+h-1, i+2*h-1, A, B) ;  /*merge two ordered lists of length h */
     if ( i+h-1< n ) /* there are still two lists left, but the length of the second one is less then h */
             Merge( i, i+h-1, n , A, B) ;  /* merge the last two lists */
     else /* if i<= n and i+h-1>= n, only one list left, make a copy */
             for ( t= i ; t<= n ; t++ )
           B[t] = A[t] ;
} /* Mpass */
```

Merge order: non recursive algorithm

Idea for Control the loop of merge: the length of merged list reach to n

```
void MergeSort ( int n , LIST A )
{   /* two-way merge sort */
    int h = 1 ;/* the length of merging list, initialized as 1 */
    LIST B ;
    while (h< n){
          MergePass (n , h , A , B) ;
          h = 2*h ;
          MergePass (n , h , B , A) ; /* swap A and B */
          h = 2*h ;
    }
}/* MergeSort */
```

Performance analysis

- Time complexity：
    - Each pass $O(n)$ times merge are needed
    - $\lceil \log_2 n \rceil$ passed merge are needed for all.
    - $O(n\log_2 n)$ for the best, worst and average case.
- Space complexity：
    - $O(n)$

- merge sort vs quick sorts

Merge sort: recursion

Idea: divide-and-conquer

- Divide: If S has at least two elements (nothing needs to be done if S has zero or one elements),
  - remove all the elements from S and
  - put them into two sequences, S_1 and S_2, each containing about half of the elements of S. i.e.
  - S_1 contains the first n/2 elements and
  - S_2 contains the remaining n/2 elements.
- Recur : Recursive sort sequences S_1 and S_2
- Conquer: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence

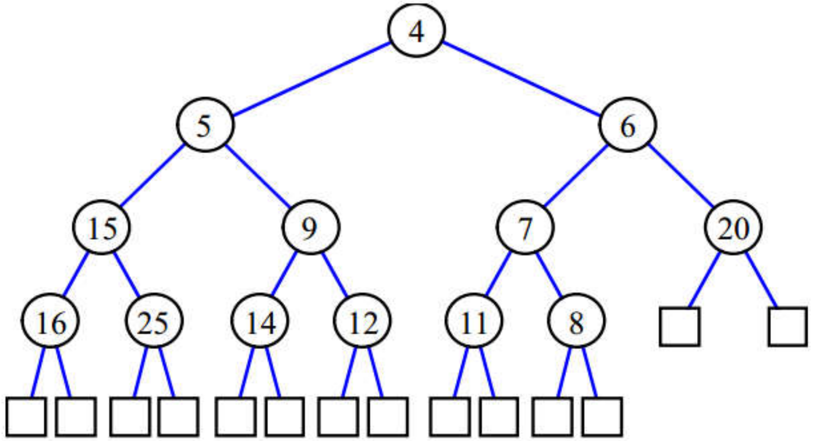
Merge sort: recursive algorithm

```
void MergeSort ( LIST A , LIST B , int low , int high )
/*  divide-and-conqur,  mergesort A[low], …, A[high] */
{    int mid = (low+high)/2 ;
     if (low<high){ /* high-low>0 */
          MergeSort (A , B , low , mid) ;
          MergeSort (A , B , mid+1 , high) ;
          Merge (low , mid , high , A , B) ;
     }
}/* MergeSort */
```

Heap sort

Heaps

- A heap is a binary tree T that stores a key-element pairs at its internal nodes
- It satisfies two properties:
  - MinHeap: key(parent) < =key(child)
  - [OR MaxHeap: key(parent) >=key(child)]
  - all levels are full, except the last one, which is left-filled
- A heap T storing n keys has height h =log(n + 1)



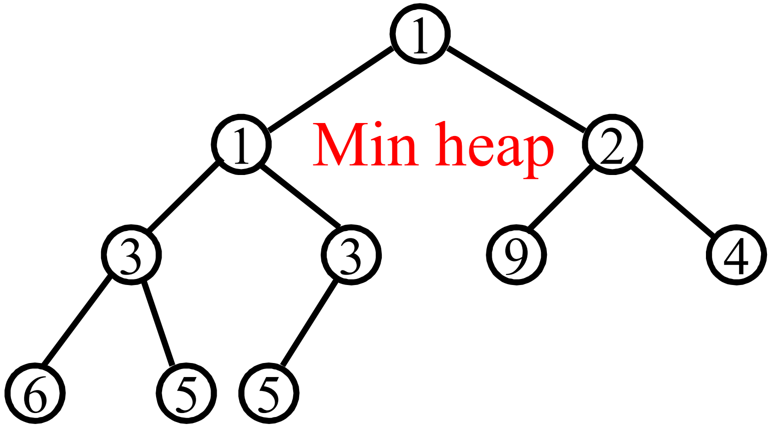
Properties of Heaps

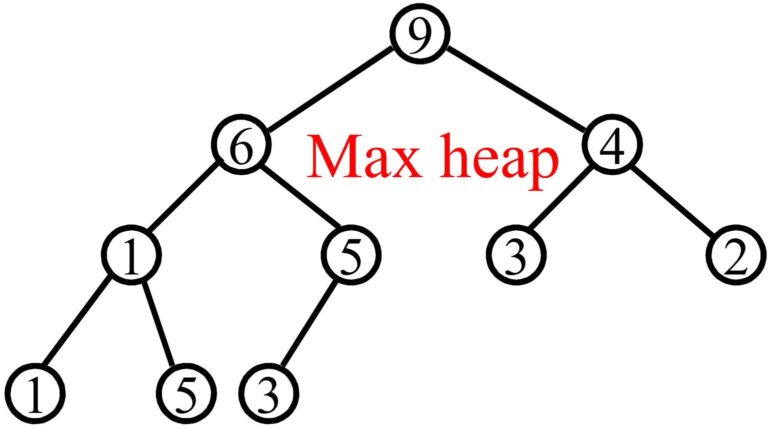
MinHeap: representation of array A

- (1) if $2*i \le n$, then A[i].key $\le$ A[$2*i$].key
- (2) if $2*i+1 \le n$, then A[i].key $\le$ A[$2*i+1$].key

MaxHeap: representation of array A

- (1) if $2*i \le n$, then A[i].key $\ge$ A[$2*i$].key
- (2) if $2*i+1 \le n$, then A[i].key $\ge$ A[$2*i+1$].key





Heap sort

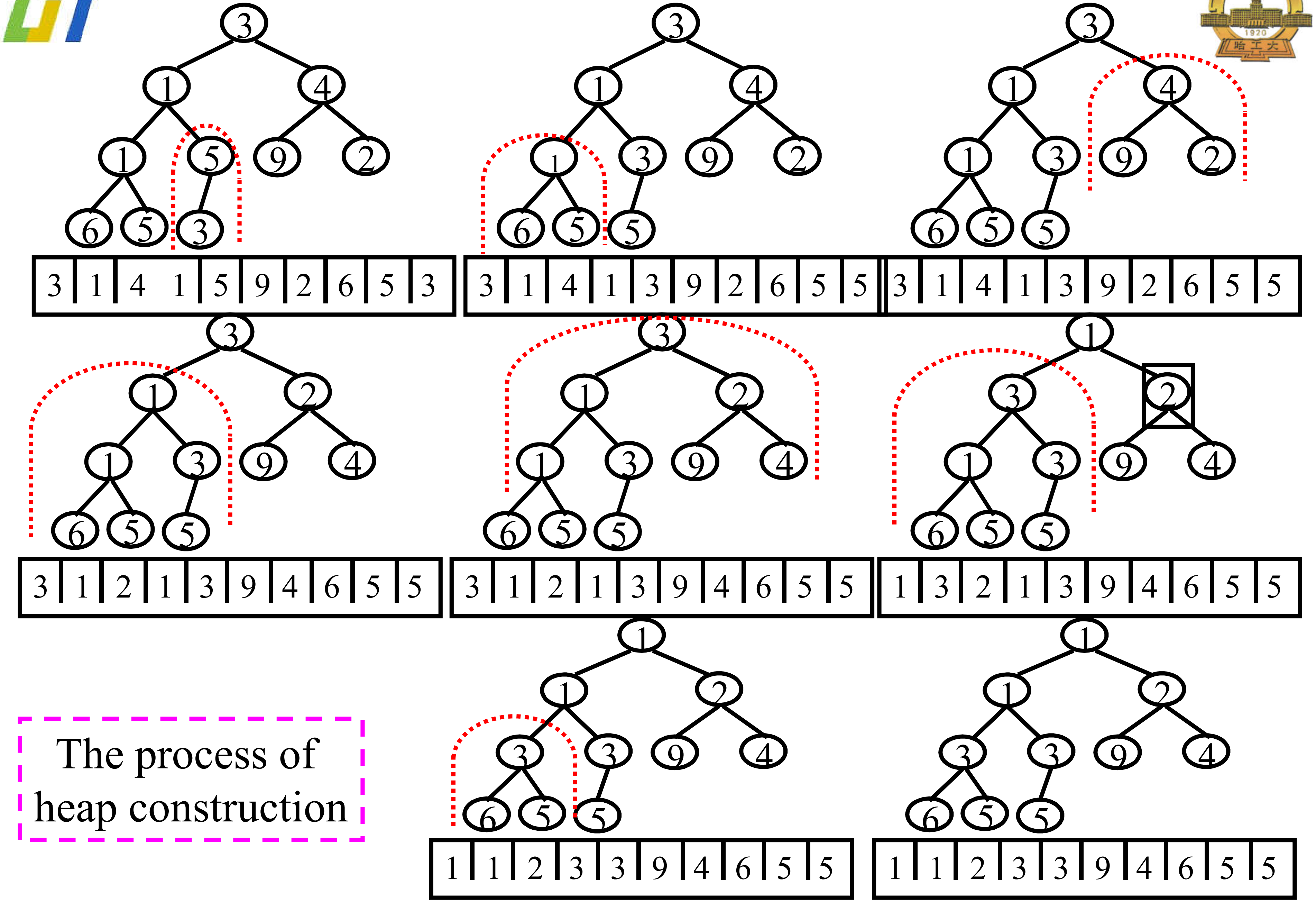
Steps：

- **Initialize** the array of complete binary tree with unsorted records
- **Construct heaps**: let *i*= *n*/2,…,2,1, construct heap with root 1 by PushDown(*i*, *n*)。
- **Heap sort**: let i = *n*, *n*-1 ,…, 2

  1.swap: swap the root (the least value) with *i* element (the largest value), i.e. swap(A[1],A[*i*]) ;

  2.manage: make the left *i*-1 elements a heap by PushDown(1, *i*-1) ;

  3.repeat: until the sequence of A[1],A[2],…,A[*n*] is ordered.



3	1	4	1	5	9	2	6	5	3

3	1	4	1	3	9	2	6	5	5

3	1	4	1	3	9	2	6	5	5

3	1	2	1	3	9	4	6	5	5

3	1	2	1	3	9	4	6	5	5

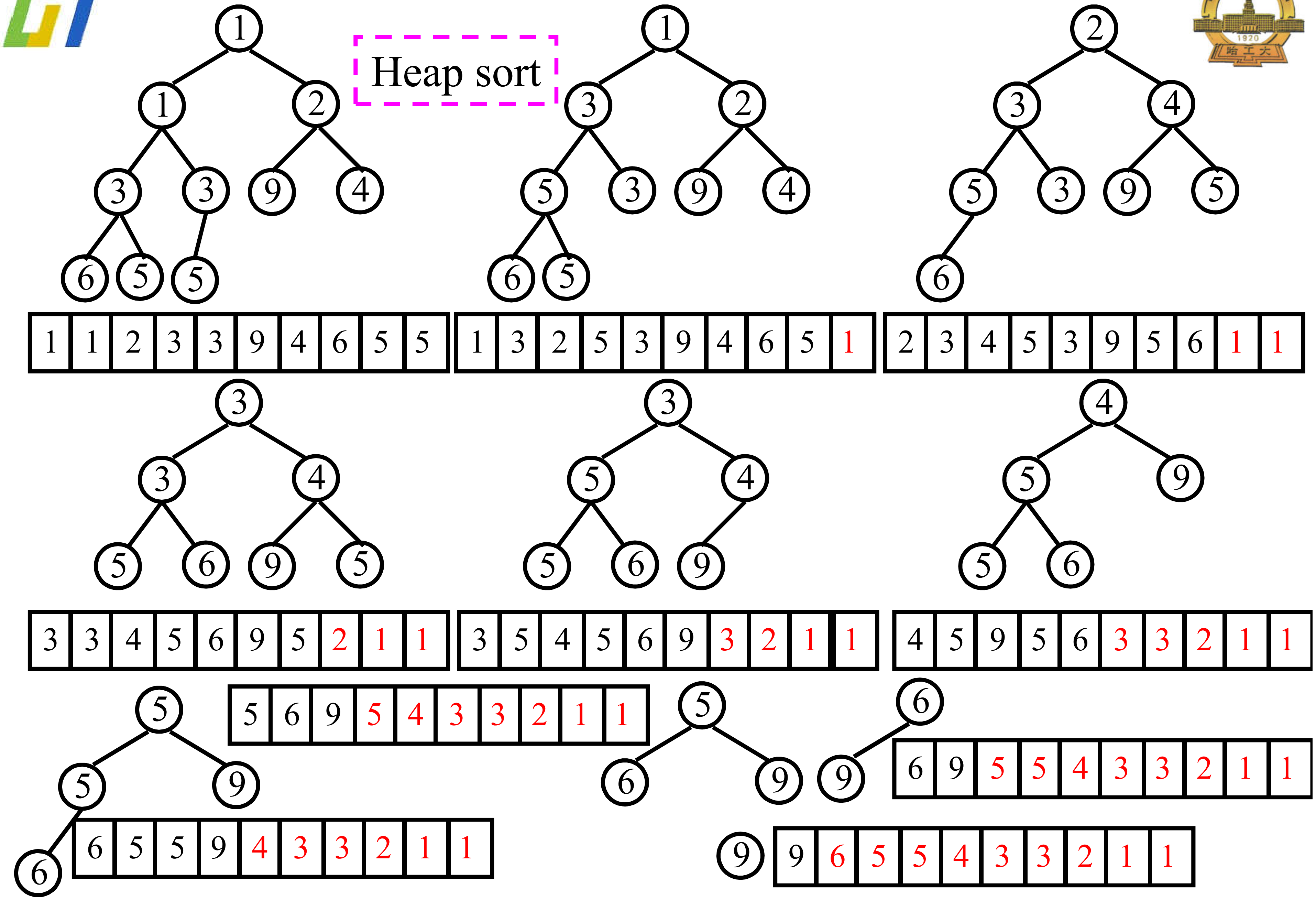
1	3	2	1	3	9	4	6	5	5

1	1	2	3	3	9	4	6	5	5

1	1	2	3	3	9	4	6	5	5

The process of heap construction

Heap sort



1	1	2	3	3	9	4	6	5	5

1	3	2	5	3	9	4	6	5	1

2	3	4	5	3	9	5	6	1	1

3	3	4	5	6	9	5	2	1	1

3	5	4	5	6	9	3	2	1	1

4	5	9	5	6	3	3	2	1	1

5	6	9	5	4	3	3	2	1	1

6	9	5	5	4	3	3	2	1	1

6	5	5	9	4	3	3	2	1	1

9	6	5	5	4	3	3	2	1	1

Implementation of heap sort

```
void HeapSort ( int n , LIST A )
{    int i;
     for( i=n/2; i<=1; i--)   /*construct heap from the most right non-leaf node*/
             PushDown( i, n);
              /*make it a heap, i is the root, n is the index of max leaf node  */
     for( i=n; i<=2; i--) {
             swap(A[1],A[i]); // swap root with i which is index of max leaf node
             PushDown( 1, i-1 );
          /*make a heap, 1 is root*/
     }
}
```

- 20151225   Friday week 15
- The course on Sunday is changed to 3-4 on Monday,28 Dec.

- The last course for the presentation is arranged 3-4 on Thursday, 31 Dec.

Implementation of heap sort

```
void PushDown(int first,int last)
{     /*manage heap: push A[first] down to a proper position*/
      int r=first; /* r is the right position， initialized as first*/
      while(r<=last/2) /* A[r] is not leaf, otherwise it is a heap */
            if((r==last/2) && (last%2==0)) {/* r has one left son of 2*r*/
               if(A[r].key>A[2*r].key)
                     swap(A[r],A[2*r]);/*push down, swap*/
               r=last;  /*A[r].key <=A[2*r].key or >=, if reach the leaf, the loop stop*/
            } else if((A[r].key>A[2*r].key)&&(A[2*r].key<=A[2*r+1].key)) {
               /*root >left son and left son <=right son (select smaller one)*/
               swap(A[r],A[2*r]);        /*swap with the left son*/
               r=2*r;               /* push down to the position*/
            } else if((A[r].key>A[2*r+1].key)&&(A[2*r+1].key<A[2*r].key)) {
               /* root > right son, and the right son <left son (select smaller one) */
               swap(A[r],A[2*r+1]);      /*swap with the right son*/
               r=2*r+1;             /*push down to the position*/
            }else /*A[r] is the heap */
               r=last;
}/*PushDown*/
```

Performance of heap sort

Performance analysis

- Time complexity of PushDown:
  - Within a while, r=first*2^i >last/2， i.e. i > $\log_2$(last/first)-1
  - O(log(last/first))=O($\log_2 n$)
- Time complexity of HeapSort :
  - O($n\log_2 n$) for best, worst and average cases.

How do you sort? Sort strategies

- There are several different ideas which lead to sorting algorithms:
- Insertion - putting an element in the appropriate place in a sorted list yields a larger sorted list.
- Exchange - rearrange pairs of elements which are out of order, until no such pairs remain.
- Selection - extract the largest element from the list, remove it, and repeat.
- Distribution - separate into piles based on the first letter, then sort each pile. (e.g. contact list in your mobile)
- Merging - Two sorted lists can be easily combined to form a sorted list.

Radix sort

Review of the performance of sorting algorithms

- Review: summary of the performance of sorting algorithms
  Frequency of execution of instructions in the inner loop:

algorithm	average	extra space	operations on keys
insertion sort	n^2	no	**compareTo()**
selection sort	n^2	no	**compareTo()**
mergesort	nlgn	n	**compareTo()**
quicksort	1.39 n lg n	c lg n	**compareTo()**

- **lower bound**: nlgn-1.44n compares are required by any algorithm
  //Based on the Stirling's approximation: $\log_2 n! \approx n\log_2 n-1.44n+O(\log_2 n)$
- Q: Can we do better (despite the lower bound)?

Radix sort

- We can do better (despite the lower bound)
  - Yes, if we do not depend on comparisons
  - Linear time complexity O(n)
- Idea of radix sort (bin sort, bucket sort)
  - Radix sort sorts integers by processing individual digits, by comparing individual digits sharing the same significant position.
  - Radix sort is not limited to integers.
    - Because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers.
- Least-significant-digit (LSD) radix sort
- Most-significant-digit (MSD) radix sorts

Radix sort

Ideas:

- There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit.
- Assuming there is sequence of integers, which have *figure* of digits，
- Each digit ranges 0,1,…, to 9, i.e. the radix is 10

1.Initializaton: there are ten empty queues, Q[0],Q[1],…,Q[9].

2.Assignment: for the *pass* pass, take all data in A, and put them in corresponding bucket according to the most right *pass* digit (r) of data.key, insert data into Q[r].

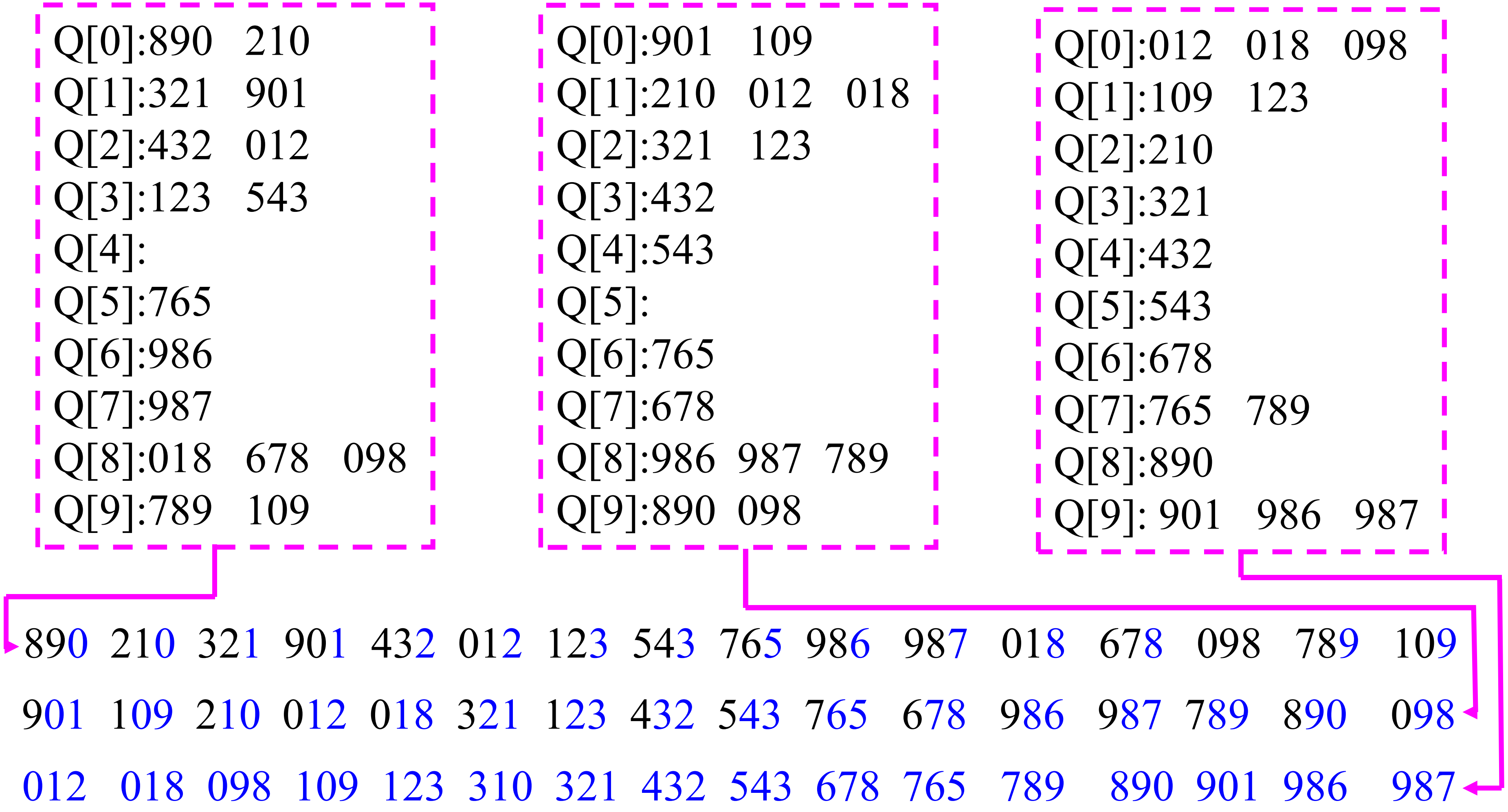
/*The processing of the keys begins at the least significant digit (i.e., the rightmost digit), and proceeds to the most significant digit (i.e., the leftmost digit) */

3.Gathering: starting from Q[0], take the all numbers from Q[0],Q[1],…, to Q[9], and insert them into array A in the taken order.

4.Repeat step 1,2,3, till *figure* passes, we have the array A ordered finally.

Demonstration of radix sort

321 986 123 432 543 018 765 678 987 789 098 890 109 901 210 012



Q[0]:890 210
Q[1]:321 901
Q[2]:432 012
Q[3]:123 543
Q[4]:
Q[5]:765
Q[6]:986
Q[7]:987
Q[8]:018 678 098
Q[9]:789 109

Q[0]:901 109
Q[1]:210 012 018
Q[2]:321 123
Q[3]:432
Q[4]:543
Q[5]:
Q[6]:765
Q[7]:678
Q[8]:986 987 789
Q[9]:890 098

Q[0]:012 018 098
Q[1]:109 123
Q[2]:210
Q[3]:321
Q[4]:432
Q[5]:543
Q[6]:678
Q[7]:765 789
Q[8]:890
Q[9]: 901 986 987

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 109

901 109 210 012 018 321 123 432 543 765 678 986 987 789 890 098

012 018 098 109 123 310 321 432 543 678 765 789 890 901 986 987

Implementation of the algorithm

```
void   RadixSort( int figure, QUEUE &A)
{      QUEUE Q[10]; records  data ;
       int  pass, r, i ;
       for ( pass=1; pass<=figure ; pass++ ){
          for ( i=0 ; i<=9 ; i++ )  /*empty queue*/
                MAKENULL( Q[i] ) ;
        while ( !EMPTY( A ) ){/* assignment*/
             data = FRONT ( A ) ;
             DEQUEUE ( A );
             r = Radix(data.key, pass) ;
             ENQUEUE( data , Q[r] ) ; }
        for ( i=0 ; i <=9 ; i++ ) /*gather*/
             while ( !EMPTY( Q[i] ) ) {
                   data = FRONT ( Q[i] ) ;
                   DEQUEUE( Q[i] ) ;
                   ENQUEUE( data, A );}
    }
}
```

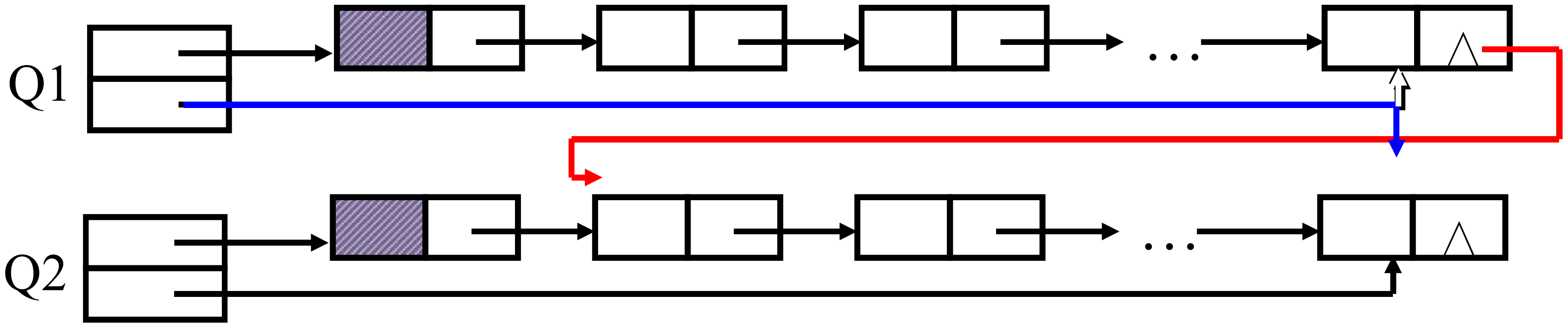
```
/  *return the pth digit of integer k */
int  Radix ( int  k, int   p)
{  int  power= 1 ;
    for ( int i=1; i<=p-1 ; i++ )
          power = power * 10 ;
    return
(( k%(power*10))/power) ;
}
```

```
for (i=0;i<=9;i++) {
     Concatenate(Q[1], Q[i]);
      A=Q[0];
}/*connect queues together */
```

Implementation of the algorithm

If not sure the length of Q[0]，Q[1],…,Q[9], then we use linked queue

```
void Concatenate(QUEUE Q1, QUEUE Q2)
{    if ( !EMPTY( Q2 ) ) {
          Q1.rear->next=Q2.front->next;
          Q1.rear=Q2.rear;
     }
}
```



Performance of radix sort

Performance analysis

- n: the number of records; d: the number of digits of integer key; r: the radix
- Time complexity：
  - Assignment O(n), gathering O(r)， d times of assignment and gathering
  - O(d(n+r))
- Space complexity：
  - The number of head and tail of queue is 2r
  - The number of link field of each record is n
  - O((n+r))

Comparison of Sorting Algorithms

- Performance factors:

(1)Running time;

(2)Space；

(3)Stability；

(4)Simple；

(5)Input distribution

(6) Number of records to be sorted

Comparison of Running Time

Sorting algorithms	Average case	Best case	Worst case
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Shell sort	$O(n\log_2 n)$	$O(n^{1.3})$	$O(n^2)$
Bubble sort	$O(n^2)$	$O\ (n)$	$O(n^2)$
Quick sort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O\ (n\log_2 n)$
Merge sort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
Radix sort	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$

Space and stability

Sorting algorithms	auxiliary space	stability
Insertion sort	$O(1)$	Yes
Shell sort	$O(1)$	No /3,2,2'(d=2,d=1)
Bubble sort	$O(1)$	Yes
Quick sort	$O(\log_2 n)$ ~$O(n)$	No /2,2',1
Selection sort	$O(1)$	No /2,2',1
Heap sort	$O(1)$	No/1,2,2'(min heap)
Merge sort	$O(n)$	yes
Radix sort	$O(n+r)$	yes

Comparison of Simpleness & record key distribution

1) Simple Algorithms:

- Insertion sort
- Selection sort
- Bubble sort

2) Not simple Algorithms:

- Shell sort
- Heap sort
- Quick sort

Not affected by the record key distribution

- Selection sort
- Heap sort
- Merge sort

Affected by the record key distribution

- Insertion sort
- Bubble sort
- Quick sort

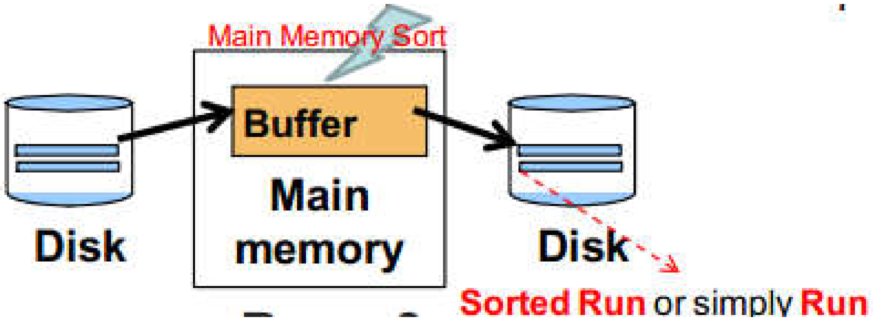
External Sorting

External Sorting Introduction

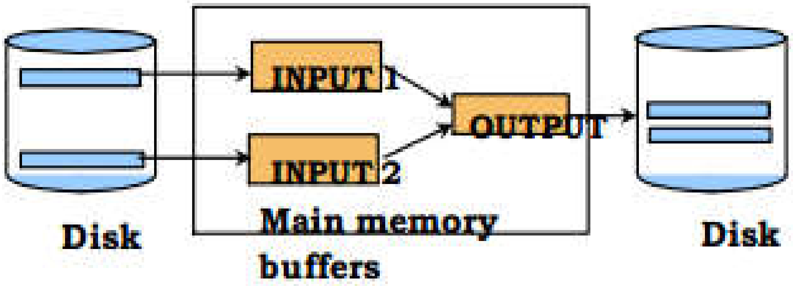
- **Problem**: We can’t sort 1TB of data with 1GB of RAM (i.e., more data than available memory) in main memory
- **Solution**: Utilize an External Sorting Algorithm
  - **External sorting** refers to the sorting of a file that resides on secondary memory (e.g., disk, flash, etc).
  - **Internal sorting** refers to the sorting of an array of data that is in RAM (quick-, merge-, insertion-, selection-, radix-, bucket-, bubble-,heap-, sort algorithms)
- **Objective:** Minimize number of I/O accesses.

Two-Way External Merge-Sort

- Let us consider the simplest idea for external sorting
  - Assumption: Only 3 Buffer pages are available
  - Idea: Divide and Conquer (similarly to MergeSort, Quicksort)
- Idea Outline
  - Pass 0 (Sort Lists): For every page, read it, sort it, write it out
    - Only one buffer page is used!
    - Now we need to merge them hierarchically
  - Pass 1, 2, …, etc. (Merge Lists):
    - For this step we need three buffer pages!
    - see next page for merging concept



Pass 0

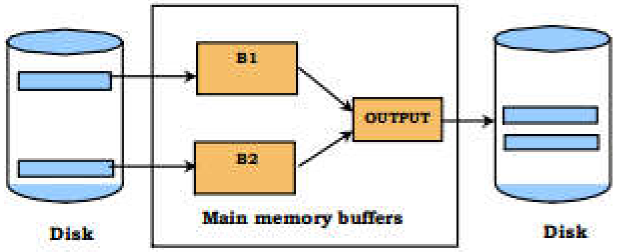


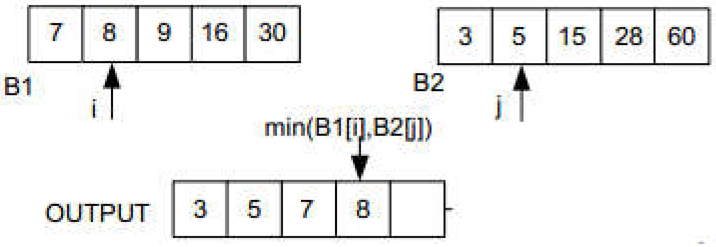
Pass 1, 2, …

Merging Lists

- Merging Lists Outline (Phase 1,2,…)

  1. Load the next sorted *runs* R1and R2 into main memory buffers B1and B2 a page-at-a-time (i.e., initially first page from each run) (see left figure)
     – Obviously R1>=B1 and R2>=B2 (a *Run* might be larger than a Buffer)
     – The rest pages will be loaded to main memory during subsequent steps.
  2. Initialize indices i, j to the head of each list (i.e., i=j=0)
  3. Compare B1[i] with B2[j] and move the smallest item to OUTPUT buffer.
     – If B1[i] was smallest item then i++ else j++ (see right figure)
     – If OUTPUT gets full, it is appended to the end of a file on DISK and cleared in RAM.
  4. Repeat the above until either index i or j reaches the end of its buffer.
     – At this point write the remaining records to OUTPUT, flush it to disk and finish.
  5. Repeat procedure from 1-4 until all *runs* have been traversed.



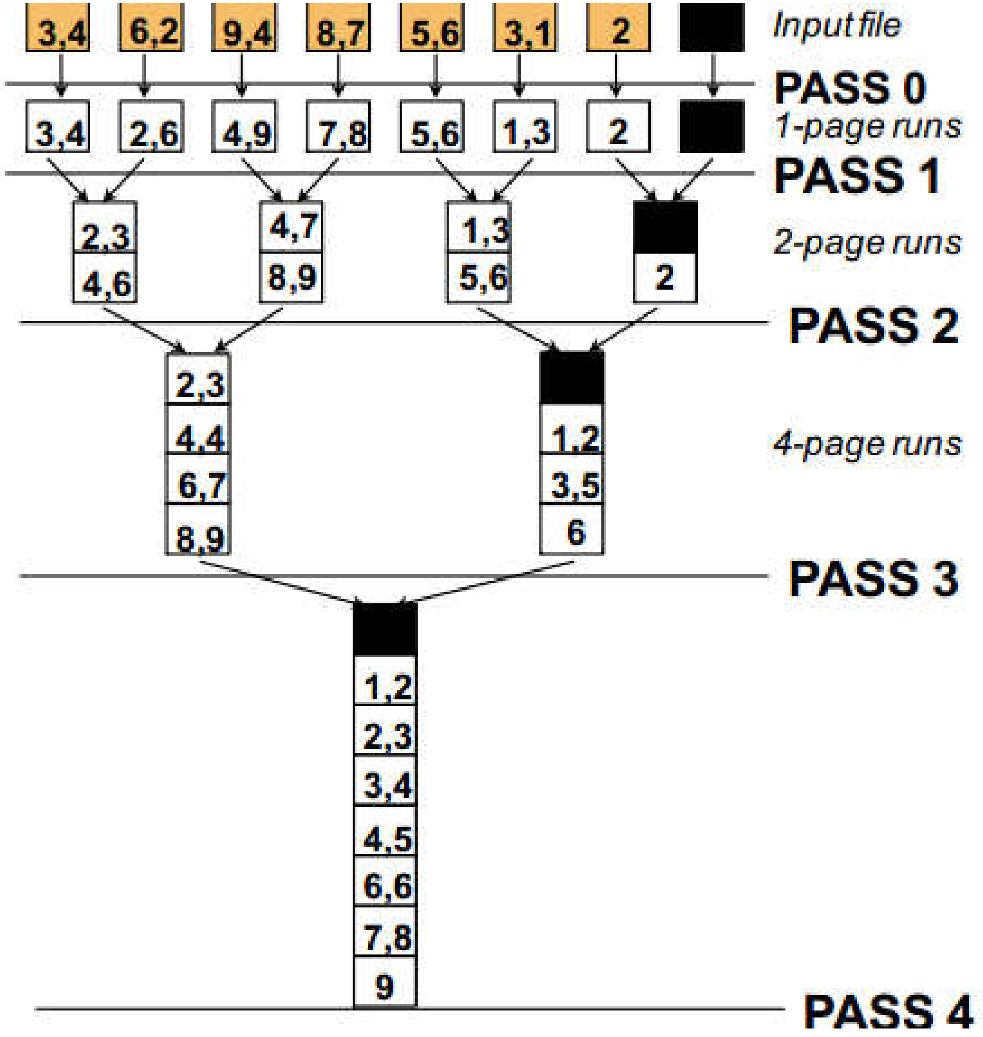

B1: 7 8 9 16 30

B2: 3 5 15 28 60

OUTPUT: 3 5 7 8

Two-Way External Merge Sort Example

Example Execution



Cost of Two-Way External Merge Sort

- **Each pass we read + write each of N pages in file.** Pass 0
- **Number of passes:**

$$= \lceil \log_2 N \rceil + 1$$

e.g., for N=7, N=5 and N=4

$$\lceil \log_2 7 \rceil + 1 = \left\lceil \frac{\log_{10} 7}{\log_{10} 2} \right\rceil + 1 = \lceil 2.8 \rceil + 1 = 4$$

$$\lceil \log_2 5 \rceil + 1 = \lceil 2.3 \rceil + 1 = 4$$

$$\lceil \log_2 4 \rceil + 1 = \lceil 2 \rceil + 1 = 3$$

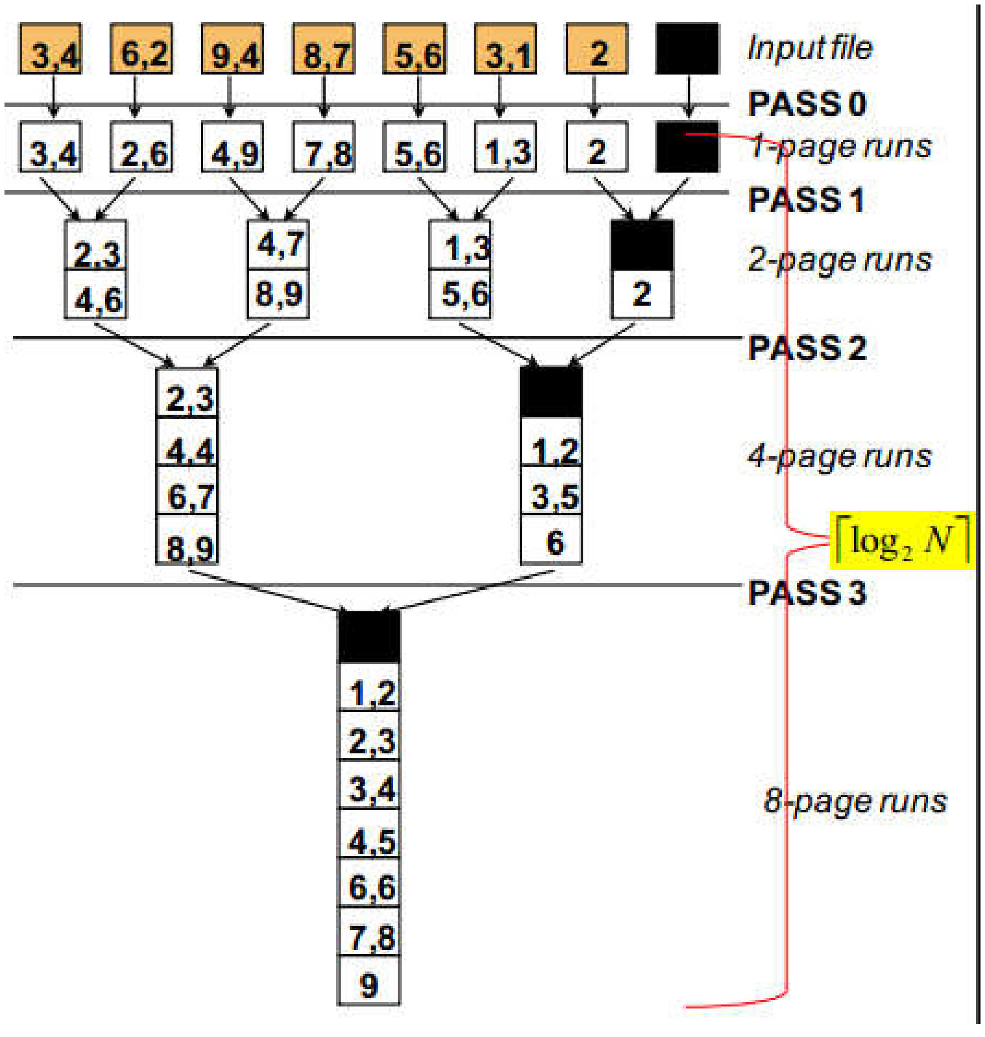
- **Total (I/O) cost is:**

$$2N * (\# \, passes)$$

e.g., for N=7

$$2*7*(\lceil \log_2 7 \rceil + 1) = \boxed{2*7*4} = 56$$

- *i.e., (read+write) * 7 pages * 4 passes*
- *That can be validated on the right figure*
  - *Pass#0=2*7* *Pass#1=2*7*
  - *Pass#2=2*7* *Pass$3=2*7*



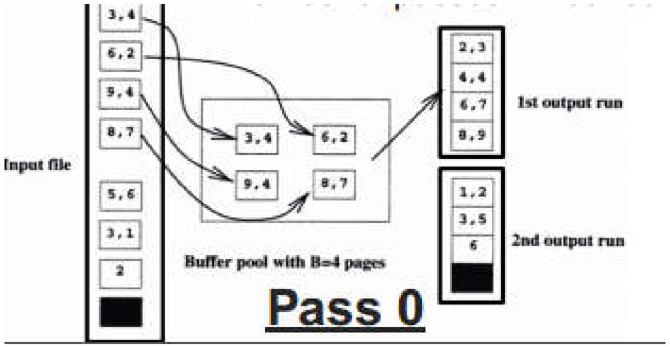
Internal Sorting Remarks

- External Sorting Algorithms utilize some Internal Sorting Algorithm to sort records in main memory.

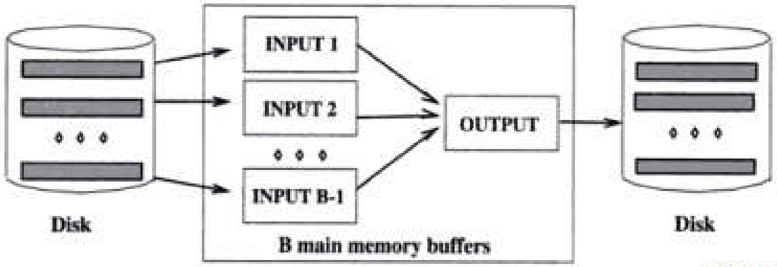
- What In-Memory Algorithms do real DBMSes use?
  - Oracle uses InsertionSort
  - Microsoft uses MergeSort
  - IBM uses RadixSort (quite similar to BucketSort)

General External Merge Sort

- Let’s turn the 2-way Merge-sort Algorithm into a Practical Algorithm.
  - Assumption: B Buffer pages are available
  - Idea: Merge (B-1) pages in each step rather than only 2 (faster!)
- Idea Outline
  - Pass 0 (Sort): Sort the N pages using B buffer pages
    - Use B buffer pages for input
    - That generates $N1 = \lceil N/B \rceil$ sorted runs e.g., N=8 and B=4 => N1=2
  - Pass 1, 2, …, etc. (Merge): Perform a (B-1)-way merge of runs
    - Use (B-1) buffer pages for input , 1 page for output
    - Number of passes will be reduced dramatically!



Pass 0



Passes 1,2,…

Cost of External Merge Sort

	(B-1)-way Sort	2-way Sort
• Number of passes:	$1+\lceil \log_{B-1}\lceil N/B\rceil\rceil$	$1+\lceil \log_2 N\rceil$
• I/O Cost =	2N * (# of passes)	2N * (# of passes)

Example: N=108 pages and B=5 Buffer pages

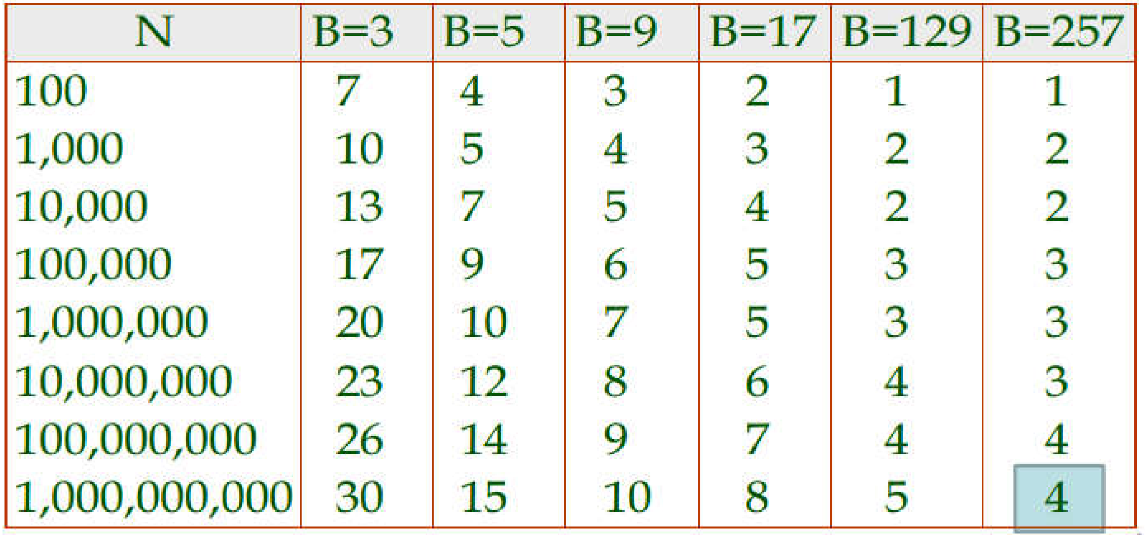
•**#Passes**= $1+\lceil \log_4\lceil 108/5\rceil\rceil=4$

•**IO Cost** =Read+Write108 pages for 4 passes => 2*108*4 = 864 Ios

- •**Pass 0**: 108/5 = 22 sorted runs of 5 pages each (last run is only 8 pages）
- •**Pass 2**: 22/4 = 2 sorted runs, 80 pages (4*20 pages) and 28 pages
- •**Pass 3**: Sorted file of 108 pages

Number of Passes of External Sort

- External Merge Sort is quite efficient!
- With only B=257 (~1MB) Buffer Pages we can sort N=1 Billion records with four (4) passes … in practice B will be larger
- Two-Way Mergesort would require $\lceil \log_2 10^9 \rceil + 1$ passes!



N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Results generated with formula: $1+\lceil \log_{B-1} \lceil N/B \rceil \rceil$

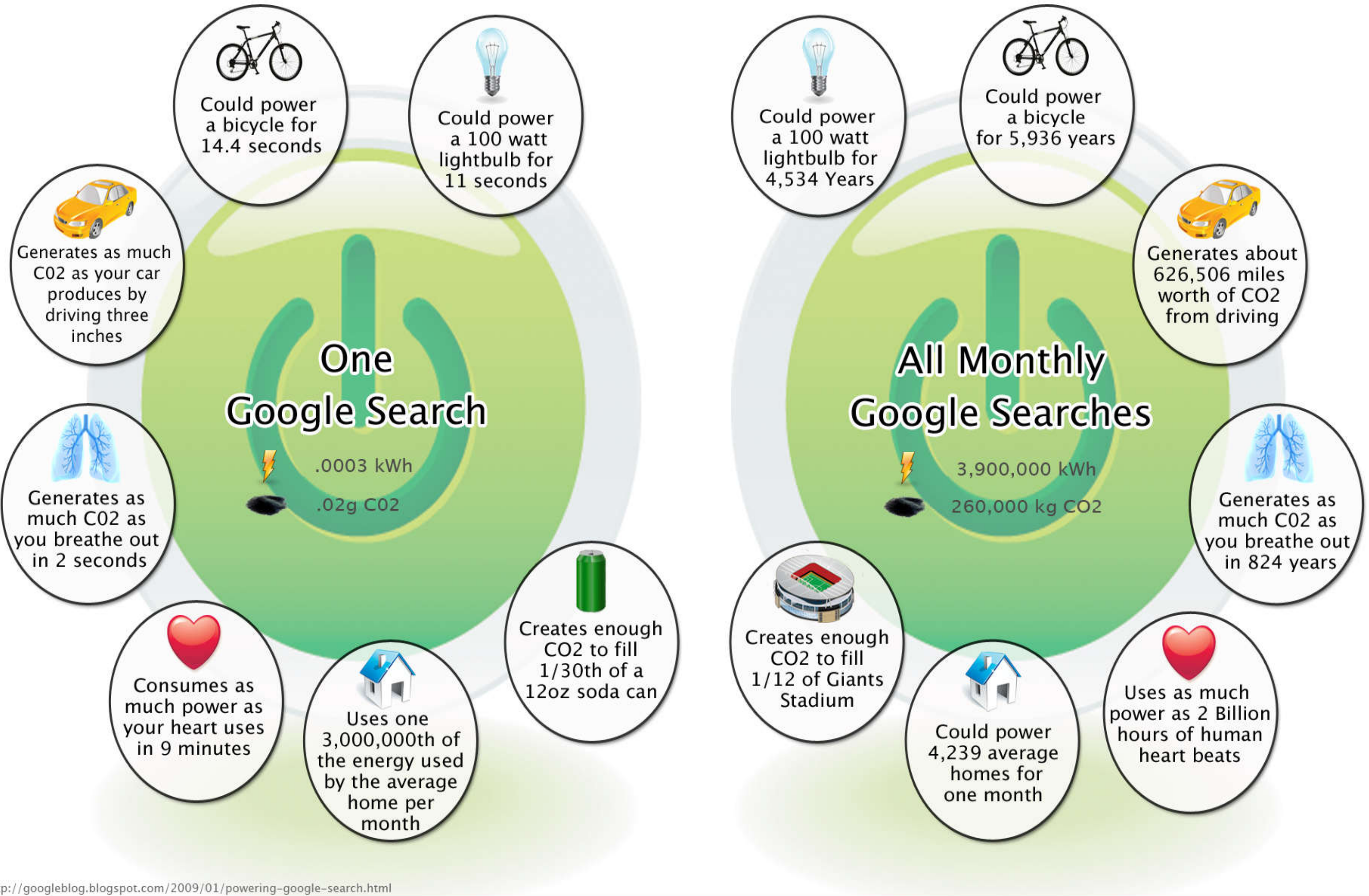
Sorting algorithms visualization

Map/Reduce

The Energy

Used in Google Search

As the world's largest search engine, Google processes nearly 13 Billion monthly searches. They are able to handle such large volumes of data because they have huge datacenters with thousands of servers capable of handling immense capacities. Such large amounts of computing power require a great deal of electricity. This electricity consumption translates directly into carbon emissions. Take a look below:



One Google Search

.0003 kWh

.02g C02

Could power a bicycle for 14.4 seconds

Could power a 100 watt lightbulb for 11 seconds

Generates as much C02 as your car produces by driving three inches

Generates as much C02 as you breathe out in 2 seconds

Consumes as much power as your heart uses in 9 minutes

Uses one 3,000,000th of the energy used by the average home per month

Creates enough CO2 to fill 1/30th of a 12oz soda can

All Monthly Google Searches

3,900,000 kWh

260,000 kg CO2

Could power a 100 watt lightbulb for 4,534 Years

Could power a bicycle for 5,936 years

Generates about 626,506 miles worth of CO2 from driving

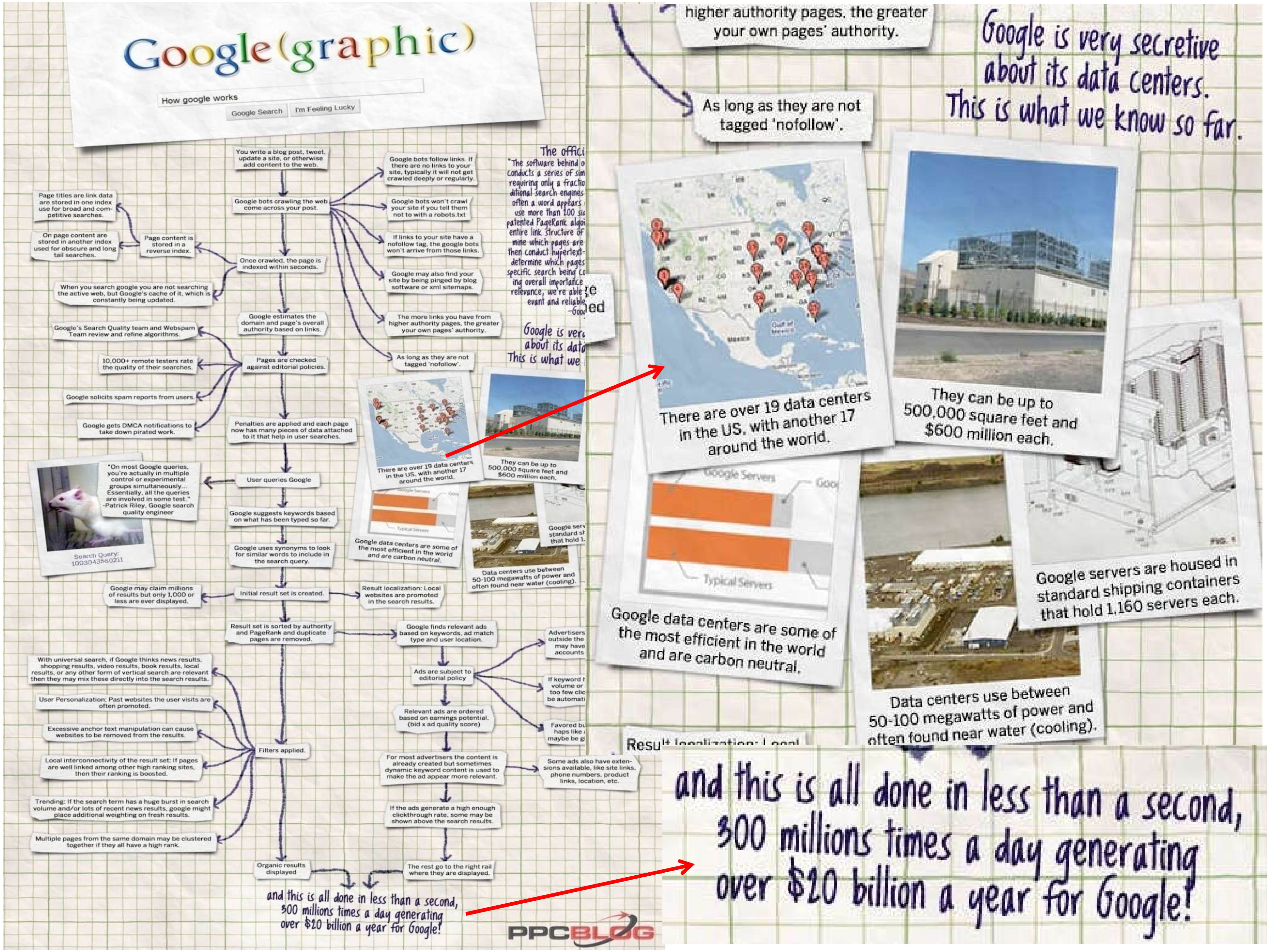
Generates as much C02 as you breathe out in 824 years

Uses as much power as 2 Billion hours of human heart beats

Could power 4,239 average homes for one month

Creates enough CO2 to fill 1/12 of Giants Stadium

ased on data provided at http://googleblog.blogspot.com/2009/01/powering-google-search.html



Google(graphic)

How google works

Google Search | I'm Feeling Lucky

You write a blog post, tweet, update a site, or otherwise add content to the web.

Google bots crawling the web come across your post.

Google bots follow links. If there are no links to your site, typically it will not get crawled deeply or regularly.

Google bots won't crawl your site if you tell them not to with a robots.txt

If links to your site have a nofollow tag, the google bots won't arrive from those links.

Google may also find your site by being pinged by blog software or xml sitemaps.

The more links you have from higher authority pages, the greater your own pages' authority.

As long as they are not tagged 'nofollow'.

Once crawled, the page is indexed within seconds.

Page content is stored in a reverse index.

Page titles are link data are stored in one index use for broad and competitive searches.

On page content are stored in another index used for obscure and long tail searches.

When you search google you are not searching the active web, but Google's cache of it, which is constantly being updated.

Google estimates the domain and page's overall authority based on links.

Pages are checked against editorial policies.

Google's Search Quality team and Webspam Team review and refine algorithms.

10,000+ remote testers rate the quality of their searches.

Google solicits spam reports from users.

Google gets DMCA notifications to take down pirated work.

Penalties are applied and each page now has many pieces of data attached to it that help in user searches.

User queries Google

"On most Google queries, you're actually in multiple control or experimental groups simultaneously... Essentially, all the queries are involved in some test." -Patrick Riley, Google search quality engineer

Search Query: 1003043560211

Google suggests keywords based on what has been typed so far.

Google uses synonyms to look for similar words to include in the search query.

Initial result set is created.

Google may claim millions of results but only 1,000 or less are ever displayed.

Result localization: Local websites are promoted in the search results.

Result set is sorted by authority and PageRank and duplicate pages are removed.

Filters applied.

With universal search, if Google thinks news results, shopping results, video results, book results, local results, or any other form of vertical search are relevant then they may mix these directly into the search results.

User Personalization: Past websites the user visits are often promoted.

Excessive anchor text manipulation can cause websites to be removed from the results.

Local interconnectivity of the result set: If pages are well linked among other high ranking sites, then their ranking is boosted.

Trending: If the search term has a huge burst in search volume and/or lots of recent news results, google might place additional weighting on fresh results.

Multiple pages from the same domain may be clustered together if they all have a high rank.

Organic results displayed

Google finds relevant ads based on keywords, ad match type and user location.

Ads are subject to editorial policy

Advertisers outside the may have accounts

If keyword h volume or too few clic be automati

Favored bu haps like maybe be g

Relevant ads are ordered based on earnings potential. (bid x ad quality score)

For most advertisers the content is already created but sometimes dynamic keyword content is used to make the ad appear more relevant.

Some ads also have extensions available, like site links, phone numbers, product links, location, etc.

If the ads generate a high enough clickthrough rate, some may be shown above the search results.

The rest go to the right rail where they are displayed.

and this is all done in less than a second,
300 millions times a day generating
over $20 billion a year for Google!

The offici
"The software behind o
conducts a series of sim
requiring only a fractio
ditional search engines
often a word appears
use more than 100 si
patented PageRank algo
entire link structure of
mine which pages are
then conduct hypertext-
determine which pages
specific search being co
ing overall importance
relevance, we're able
evant and reliable
-Goo

Google is ver
about its data
This is what we

There are over 19 data centers in the US, with another 17 around the world.

They can be up to 500,000 square feet and $600 million each.

Google data centers are some of the most efficient in the world and are carbon neutral.

Data centers use between 50-100 megawatts of power and often found near water (cooling).

Google serv standard sh that hold 1.

PPCBLOG

higher authority pages, the greater your own pages' authority.

As long as they are not tagged 'nofollow'.

Google is very secretive about its data centers.
This is what we know so far.

There are over 19 data centers in the US, with another 17 around the world.

They can be up to 500,000 square feet and $600 million each.

Google data centers are some of the most efficient in the world and are carbon neutral.

Data centers use between 50-100 megawatts of power and often found near water (cooling).

Google servers are housed in standard shipping containers that hold 1,160 servers each.

and this is all done in less than a second,
300 millions times a day generating
over $20 billion a year for Google!

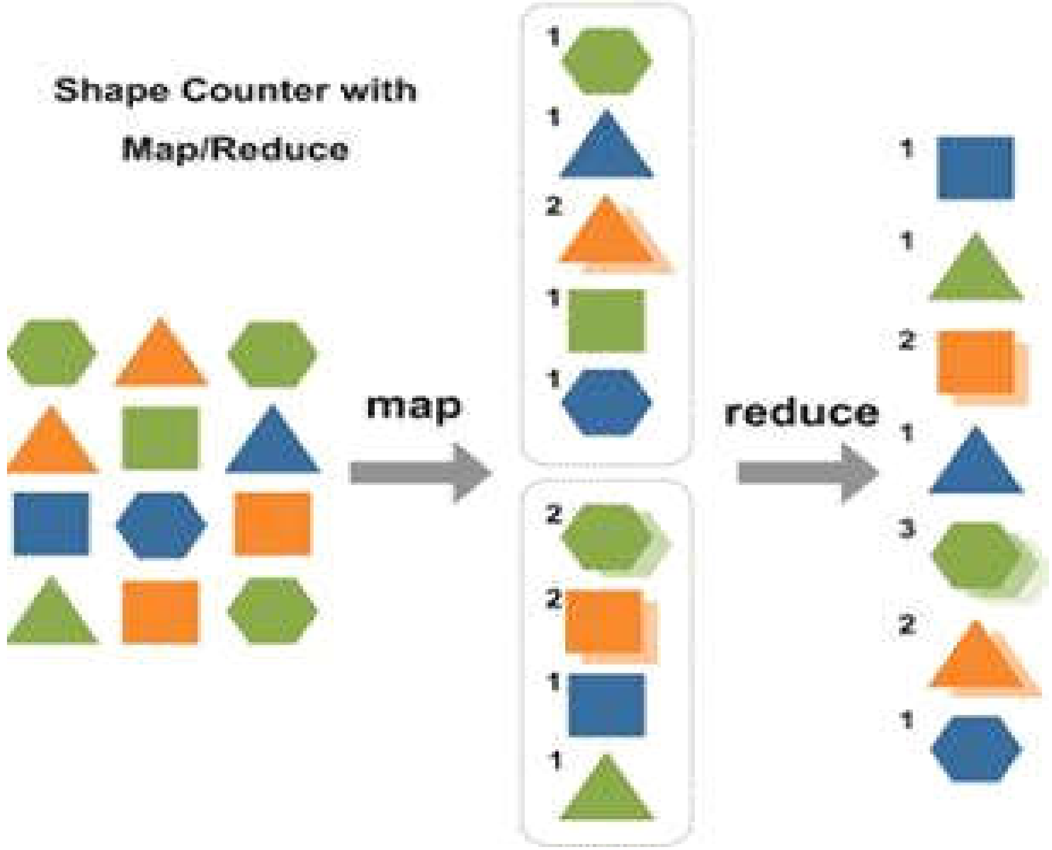
Cloud Computing:
Google five core technologies

- GFS (2003)
  - Google File System
- MapReduce (2004)
  - Simplified data processing on large cluster
- BigTable (2006)
  - A distributed storage system for structured data
- Chubby (2006)
  - Lock service for loosely-coupled distributed systems
- BORG
  - Cluster management and schedule

Concurrent & distributed programming model: Map/Reduce

- A programming paradigm
  - Borrows from functional programming
  - Users implement interface of two functions: map()/reduce()
- For processing large data set (> 1 TB)
- Exploits large set of commodity computers (Clusters: +1k CPUs)
- Executes process in distributed manner
- Offers high degree of transparencies
- In other words:
  - Simple, easy, and maybe suitable for your tasks !!!

Simple example: Shape Counter



MapReduce: Map

Map (input_key, input_value) =>
(output_key, intermediate_value) list

- Records from the data source are fed into the map function as (key, value) pairs:
  - data source: lines out of files, rows of a database, etc
  - (key, value) of data source: (filename, offset)
- ***map()*** produces one or more intermediate values along with an output key from the input.
  - {<key1,1>, <key2,1>, <key3,1>, <key4,1>}

MapReduce: Reduce

Reduce (output_key, intermediate_value list) =>

output list *// usually only one final value per key*

- After the map phase is over, all the intermediate values **for a given output key** are combined together into a list
  - <key → 1, 1, 1>
- ***reduce()*** combines those intermediate values into one or more final values for that **same output key**
  - in practice, usually only one final value per key
  - <key, 3>

Two phases of data processing

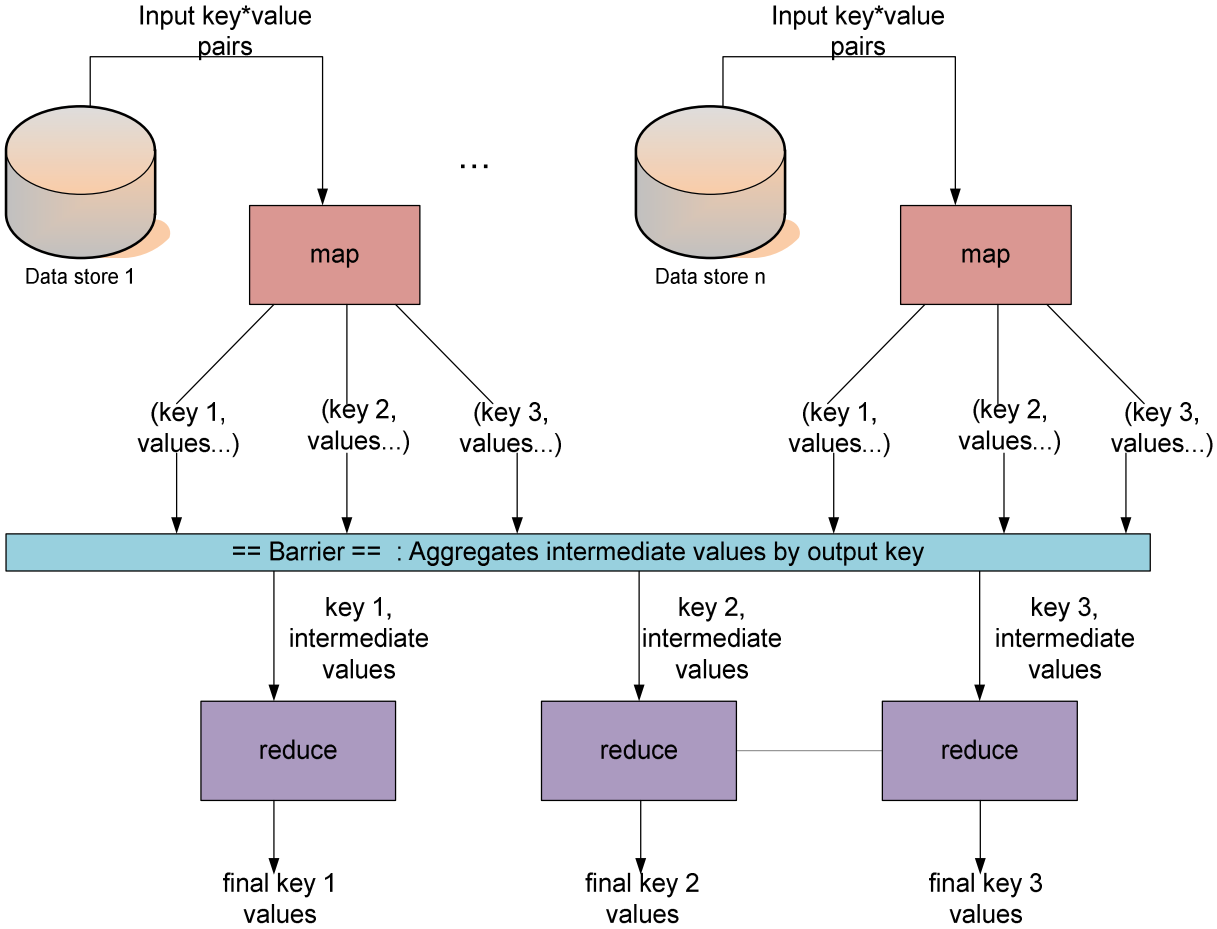
Map (*in_key*, *in_value*) → $\{(key_j, value_j) \mid j = 1 \ldots k\}$

//identify & combine

Aggregates intermediate values by output key

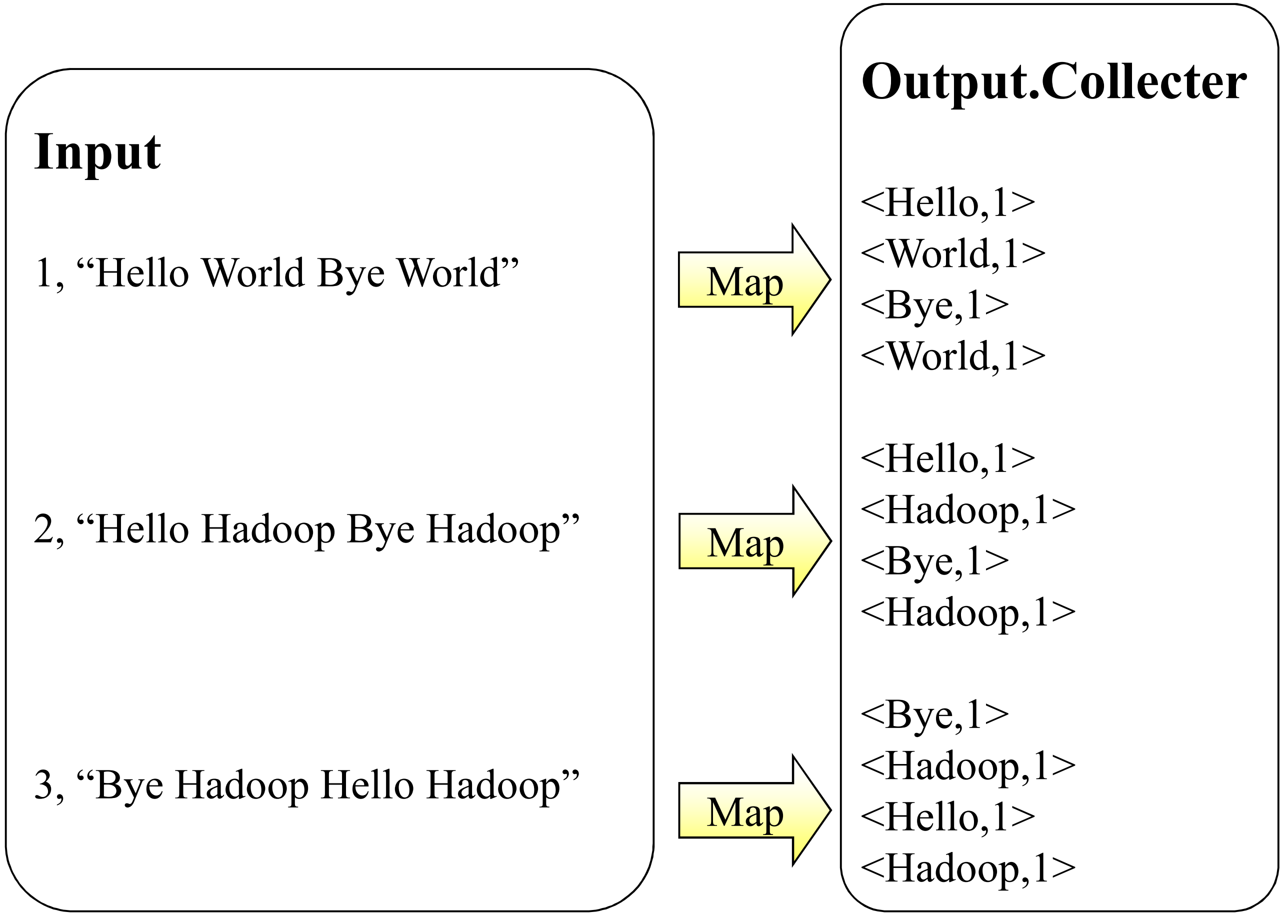
Reduce (*key*, $[value_1, \ldots value_m]$) → (*key*, *f_value*)

// group



Example: WordCount 1/4

$$Map\ (in\_key, in\_value) \rightarrow \{(key_j, value_j) \mid j = 1 \ldots k\}$$



Input

1, “Hello World Bye World”

2, “Hello Hadoop Bye Hadoop”

3, “Bye Hadoop Hello Hadoop”

Output.Collecter

<Hello,1>
<World,1>
<Bye,1>
<World,1>

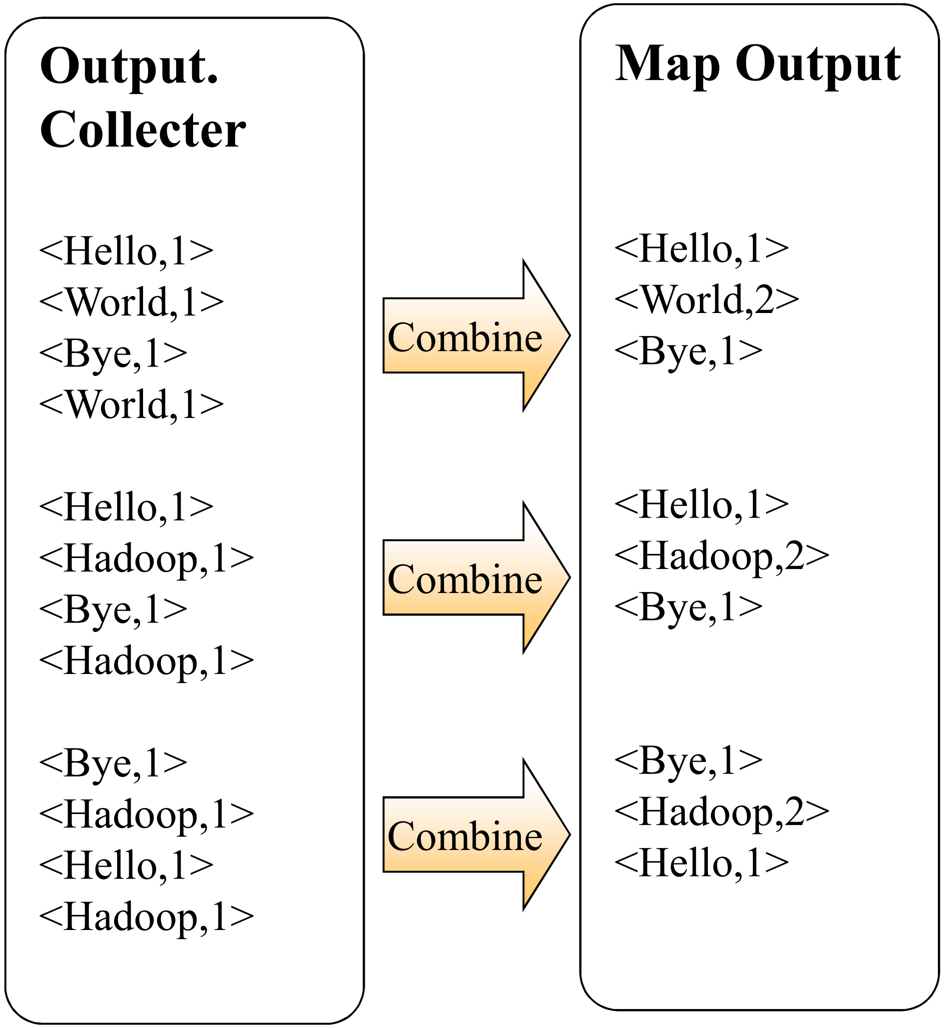
<Hello,1>
<Hadoop,1>
<Bye,1>
<Hadoop,1>

<Bye,1>
<Hadoop,1>
<Hello,1>
<Hadoop,1>

K = document URL
V = document contents

```
Map(K, V) {
  For each word w in V
    Collect(w, 1);
}
```

Example: WordCount 2/4



Output. Collecter

<Hello,1>
<World,1>
<Bye,1>
<World,1>

<Hello,1>
<Hadoop,1>
<Bye,1>
<Hadoop,1>

<Bye,1>
<Hadoop,1>
<Hello,1>
<Hadoop,1>

Combine

Combine

Combine

Map Output

<Hello,1>
<World,2>
<Bye,1>

<Hello,1>
<Hadoop,2>
<Bye,1>

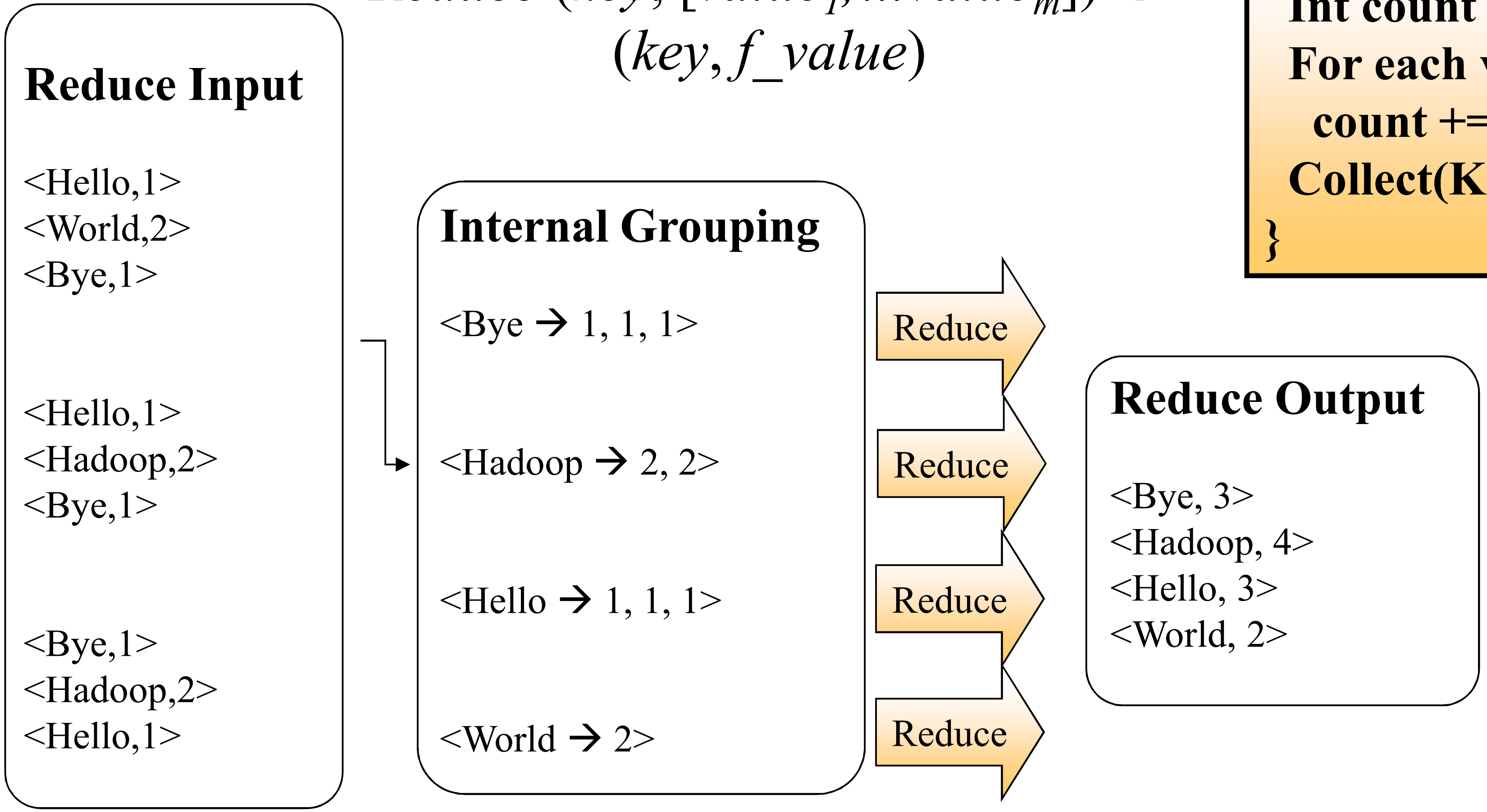
<Bye,1>
<Hadoop,2>
<Hello,1>

```
Combine(K, V[ ]) {
  Int count = 0;
  For each v in V
    count += v;
  Collect(K, count);
}
```

Example: WordCount 3/4

$$\textit{Reduce}\ (\textit{key}, [\textit{value}_1, \ldots \textit{value}_m]) \rightarrow (\textit{key}, \textit{f\_value})$$

```
Reduce(K, V[ ]) {
  Int count = 0;
  For each v in V
    count += v;
  Collect(K, count);
}
```



Reduce Input

<Hello,1>
<World,2>
<Bye,1>

<Hello,1>
<Hadoop,2>
<Bye,1>

<Bye,1>
<Hadoop,2>
<Hello,1>

Internal Grouping

<Bye → 1, 1, 1>

<Hadoop → 2, 2>

<Hello → 1, 1, 1>

<World → 2>

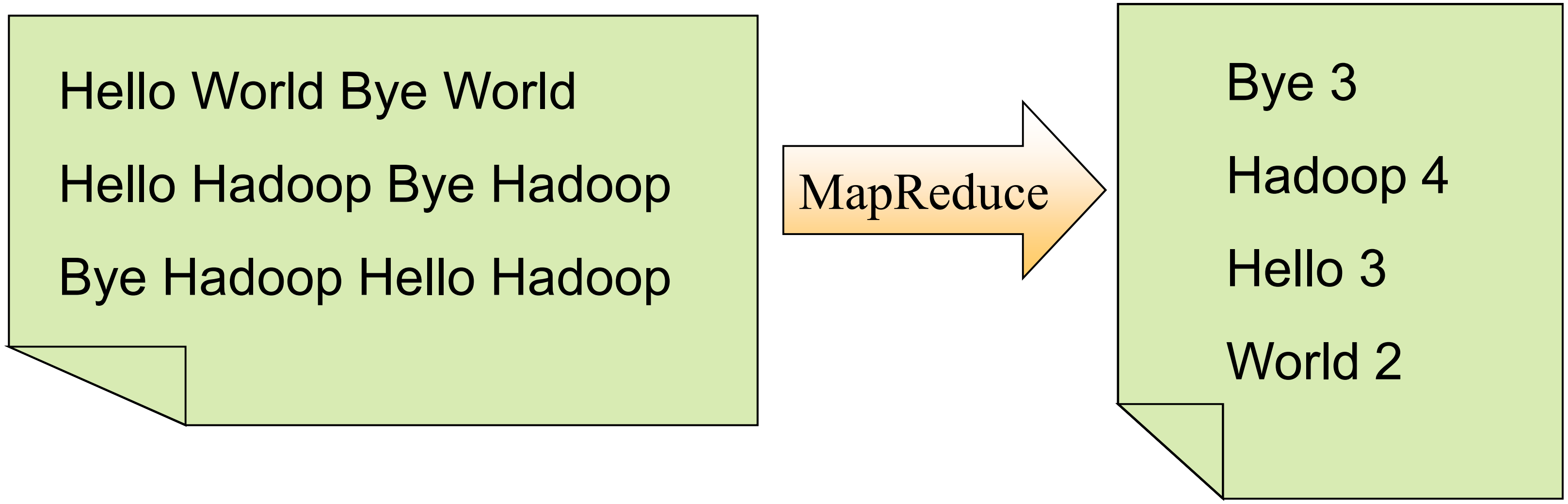
Reduce Output

<Bye, 3>
<Hadoop, 4>
<Hello, 3>
<World, 2>

Example: WordCount 4/4

Input:
File containing words

Output:
Number of occurrences
of each word



Hello World Bye World

Hello Hadoop Bye Hadoop

Bye Hadoop Hello Hadoop

MapReduce

Bye 3

Hadoop 4

Hello 3

World 2

Based on Functional Programming

- Functional operations do not modify data structures
  - They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

More Challenges (1/2)

- Automatic parallelization & distribution
  - All values are processed independently
  - Bottleneck: reduce phase can't start until map phase is completely finished.
- Fault-tolerance: nice retry/failure semantics
  - master/slave: periodical ping/checkpoint
  - re-execution
  - file chunks are automatically distributed and replicated
- Locality
  - mapers on same machine as physical file data

More Challenges (2/2)

- Optimization
  - 64MB blocks: same size as GFS chunks
  - mini-reduce phase to occur for saving bandwidth
  - Combiners can run on same machine as a mapper

- Automatic load balancing & high throughput
- I/O scheduling
- Monitoring & status updates

Example applications: 1. Search

- **Input:** (lineNumber, line) records
- **Output:** lines matching a given pattern

- **Map:**

```
if(line matches pattern):
    output(line)
```

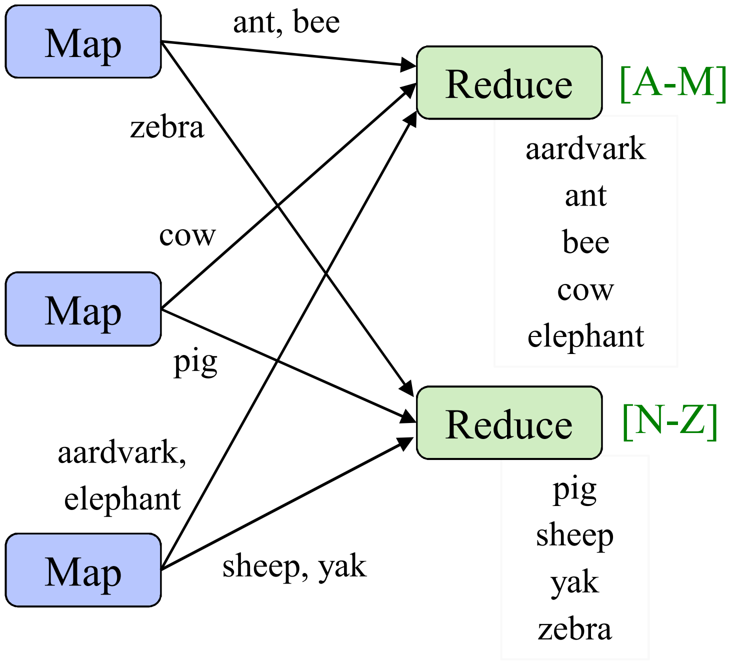
- **Reduce:** identify function
  - Alternative: no reducer (map-only job)

Example applications: 2. Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function h such that $k_1<k_2 \Rightarrow h(k_1)<h(k_2)$



Example applications: 3. Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

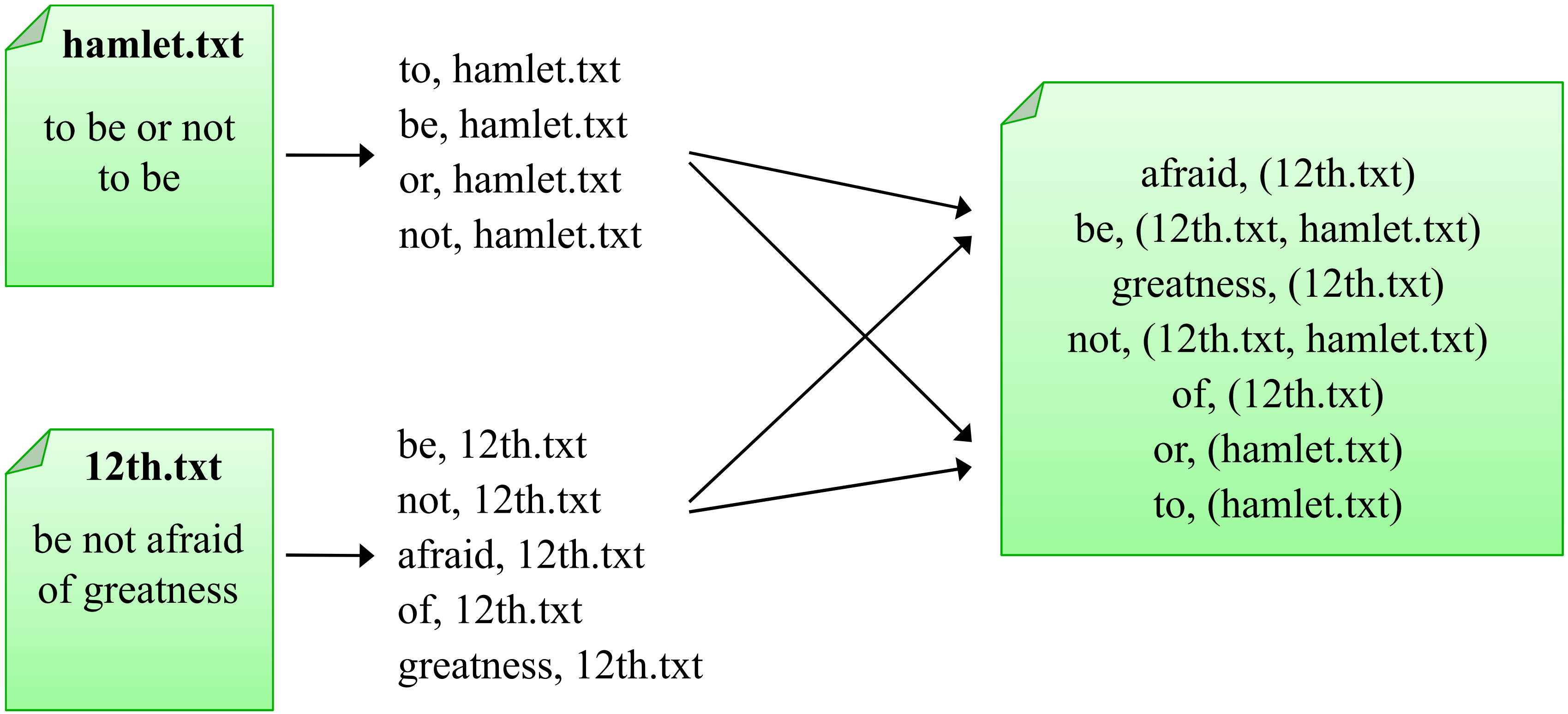
- **Map:**

```
foreach word in text.split():
    output(word, filename)
```

- **Combine:** uniquify filenames for each word
- **Reduce:**

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

Inverted Index Example



hamlet.txt

to be or not
to be

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

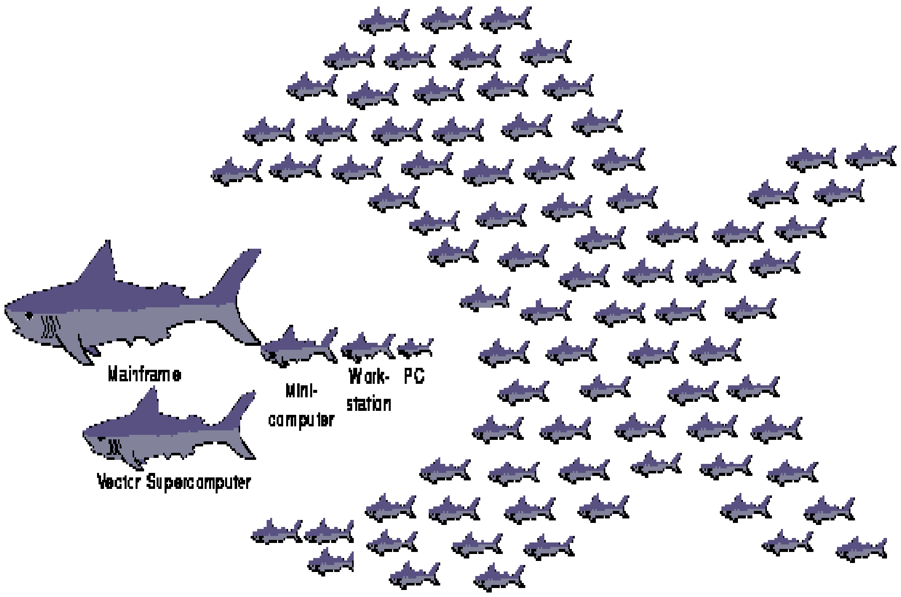
be not afraid
of greatness

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

Summary

- A simple programming model for processing large dataset on large set of computer cluster
- Clean abstraction for large scale data processing
- Very robust in fault tolerance and error handling
- Fun to use,
  - focus on problem, and
  - let the library deal with the messy detail



"Successful people are not gifted; they just work hard, then succeed on purpose." — G. K. Nielson

"Scientists investigate that which already is; Engineers create that which has never been." — Albert Einstein.

" Computers do not solve problems, they execute solutions " — Laurent Gasser

"The real problem is not whether machines think but whether men do.“ — B. F. Skinner

“Your beliefs become your thoughts, Your thoughts become your words, Your words become your action, Your actions become your habits, Your habits become your values, You values become your destiny." — Mahatma Gandhi

You make the world different.

Never give up, Never!