Internal Sorting

Department of Computer Science

HCMC University of Technology, Viet Nam

04, 2014

Outline

Introduction

Sorting is one of the most *important concepts* and *common applications*

Classifying

- Internal Sort: all data in *primary memory*
- External Sort: big data (not fitted in primary memory)

Properties

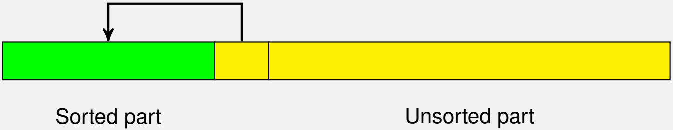
- Stability: data with equal key maintain their relative input order in the output
- Eficiency: number of *comparisons* and number of *moves*

Insertion Sort

- Divide the list into two parts: sorted and unsorted
- For each step, insert the first element in the unsorted part into suitable position the sorted part

Insertion Sort

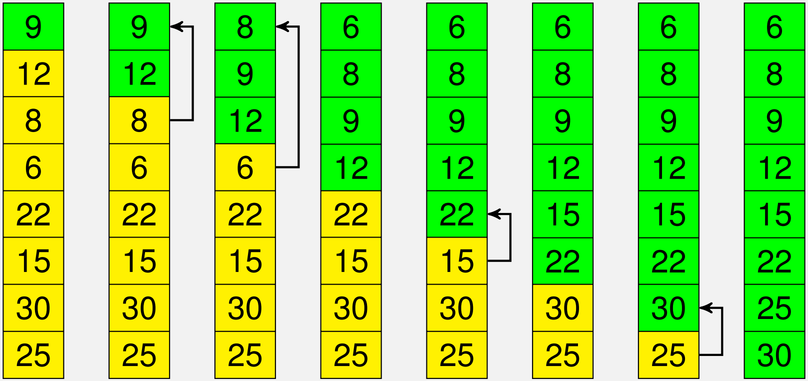
- Divide the list into two parts: sorted and unsorted
- For each step, insert the first element in the unsorted part into suitable position the sorted part



Insertion Sort

- Divide the list into two parts: sorted and unsorted
- For each step, insert the first element in the unsorted part into suitable position the sorted part

Example


9 12 8 6 22 15 30 25
9 12 8 6 22 15 30 25
8 9 12 6 22 15 30 25
6 8 9 12 22 15 30 25
6 8 9 12 22 15 30 25
6 8 9 12 15 22 30 25
6 8 9 12 15 22 30 25
6 8 9 12 15 22 25 30

Insertion Sorting Implementation

Input: Unsorted array **arr**
Output: Sorted array **arr**

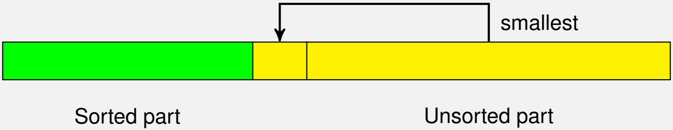
```
for (i = 1; i < n ; i++) {
     tmp = arr[i];
     for (j = i-1;
            j >= 0 && tmp < arr[j];
            j--)
          arr[j+1]=arr[j];
     arr[j+1]=tmp;
}
```

Selection Sort

- Divide the list into two parts: sorted and unsorted
- For each step, select the smallest/largest element in the unsorted part and place it to the sorted part

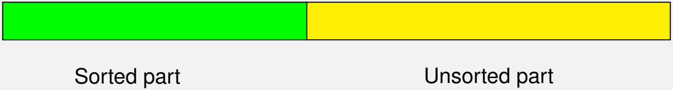
Selection Sort

- Divide the list into two parts: sorted and unsorted
- For each step, select the smallest/largest element in the unsorted part and place it to the sorted part

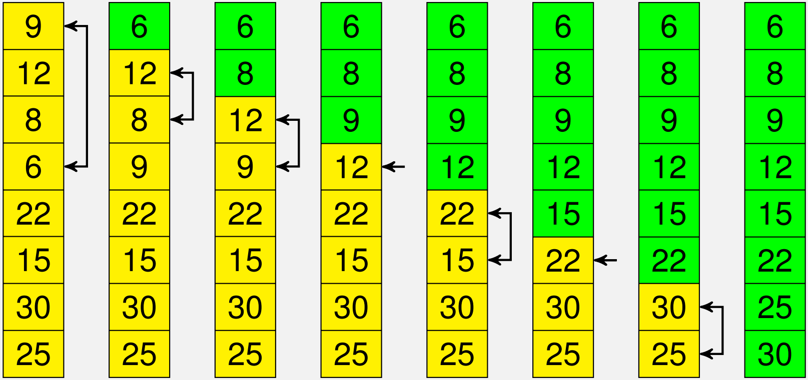


Selection Sort

- Divide the list into two parts: sorted and unsorted
- For each step, select the smallest/largest element in the unsorted part and place it to the sorted part



Example



9 12 8 6 22 15 30 25
6 12 8 9 22 15 30 25
6 8 12 9 22 15 30 25
6 8 9 12 22 15 30 25
6 8 9 12 22 15 30 25
6 8 9 12 15 22 30 25
6 8 9 12 15 22 30 25
6 8 9 12 15 22 25 30

Selection Sorting Implementation

Input: Unsorted array **arr**
Output: Sorted array **arr**

```
for (i=0 ; i<n-1 ; i++) {
    lowindex = i;
    for (j=i+1 ; j<n ; j++)
        if (arr[j] < arr[lowindex])
            lowindex = j;
    swap(arr,i,lowindex);
}
```

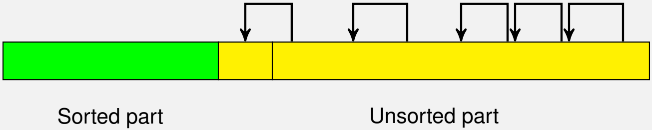
Bubble Sort

- Divide the list into two parts: sorted and unsorted
- For each step, the smallest/largest in the unsorted part is bubbled toward the sorted part

Sorted part Unsorted part

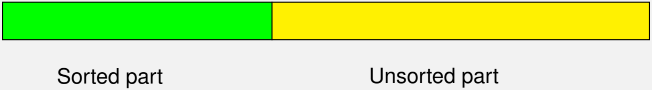
Bubble Sort

- Divide the list into two parts: sorted and unsorted
- For each step, the smallest/largest in the unsorted part is bubbled toward the sorted part

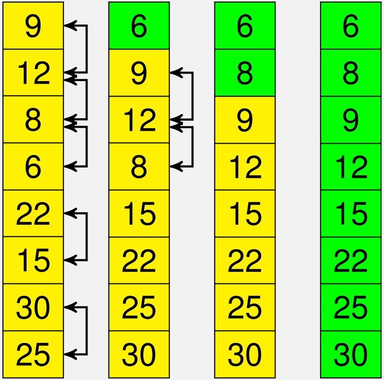


Bubble Sort

- Divide the list into two parts: sorted and unsorted
- For each step, the smallest/largest in the unsorted part is bubbled toward the sorted part

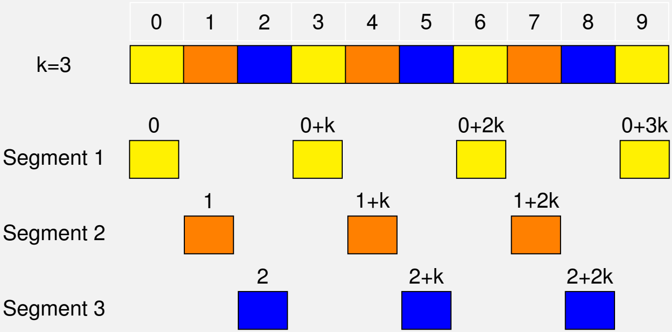


Example


9 12 8 6 22 15 30 25
6 9 12 8 15 22 25 30
6 8 9 12 15 22 25 30
6 8 9 12 15 22 25 30

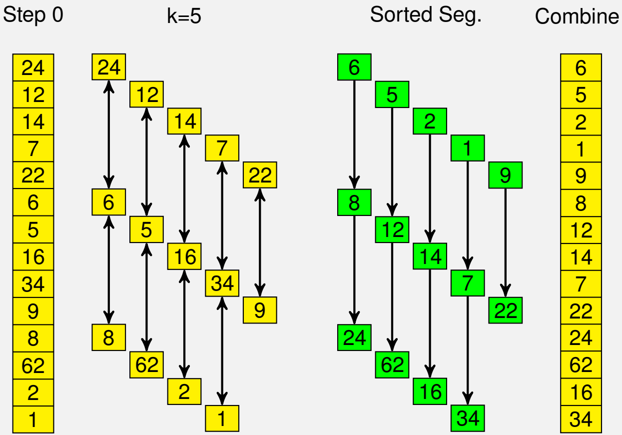
Shell Sort

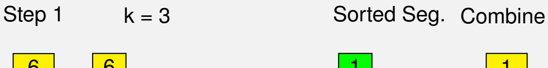
- Invented by Donald L.Shell (1995)
- Also called diminishing-increment sort
- Divide data into K segments (or increment)
- These segments are dispersed through out the data



Shell Sorting Algorithm

- In each iteration, sort K segments using insertion sort
- Reduce K after each iteration until K == 1





Shell Sorting Implementation

```
for (k=first_incremental_value;
     k>=1;
     k=next_incremental_value)
     for (segment=0;
          segment<k;
          segment++)
          segmentSort(segment,k);
```

segmentSort(int segment,int k)

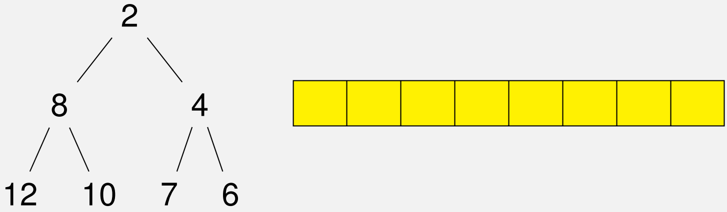
```
for (current=segment+k;
     current<size;
     current=current+k) {
     tmp = data[current];
     for (walker=current-k;
          walker>=0
          && tmp < data[walker];
          walker=walker-k)
          data[walker+k]=data[walker];
      data[walker+k]=tmp;
}
```

Shell Sort Discussion

- incremental values should not be multiple of each other
  - (2k+1): 1, 3, 7, 15, 31,...
  - (3k+1): 1, 4, 13, 40, ...
- Time complexity through experiments: O($n^{1.25}$)

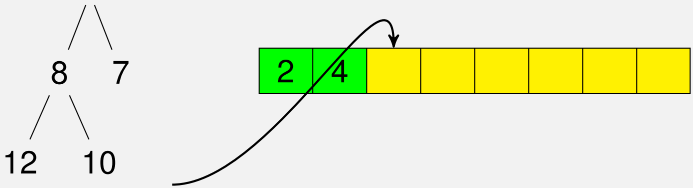
Heap Sort

- Build a heap for data
- For each step, take the root of the heap and put it in the sorted part



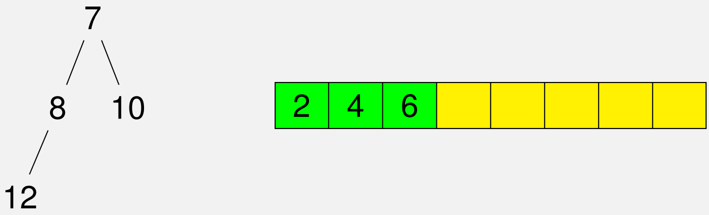
Heap Sort

- Build a heap for data
- For each step, take the root of the heap and put it in the sorted part

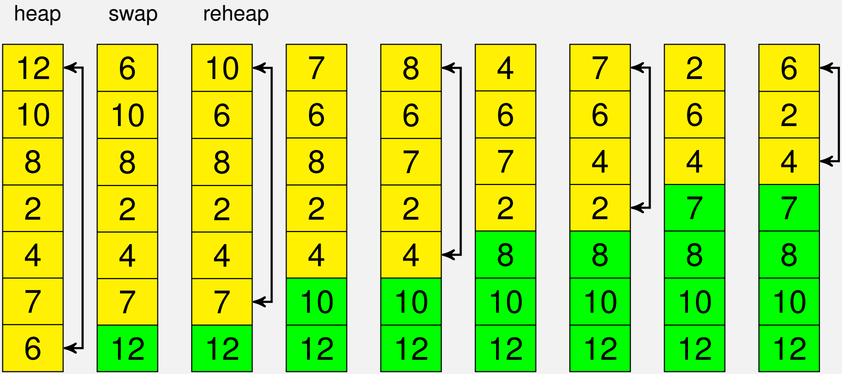


Heap Sort

- Build a heap for data
- For each step, take the root of the heap and put it in the sorted part



Example


heap
swap
reheap
12 10 8 2 4 7 6
6 10 8 2 4 7 12
10 6 8 2 4 7 12
7 6 8 2 4 10 12
8 6 7 2 4 10 12
4 6 7 2 8 10 12
7 6 4 2 8 10 12
2 6 4 7 8 10 12
6 2 4 7 8 10 12

Divide-And-Conquer Meta Algorithm

```
Partition data into many parts
for (each part)
      Divide-And-Conquer on each part
Combine many resulted parts
```

Merge Sort

8	4	12	6	33	42	16	7

- Partition the array into two parts

8	4	12	6

33	42	16	7

- Sort two parts (using rescursive or iterative)

4	6	8	12

7	16	33	42

- Merge two ordered parts

4	6	7	8	12	16	33	42

Quick Sort

8	4	12	6	33	42	16	7	5

- Based on the pivot, partition the array into three parts: less than, pivot, and greater than or equal to.

4	6	7	5

8

12	33	42	16

- Sort the first and the last parts (using rescursive or iterative)

4	5	6	7

8

12	16	33	42

- Append three ordered parts

4	5	6	7	8	12	16	33	42

Pivot Selection

- C. A. Hoare (1962): the first element
  - Simple
  - Unbalanced parts
- R. C. Singleton (1969): the median of the first, last and the middle elements

Partition

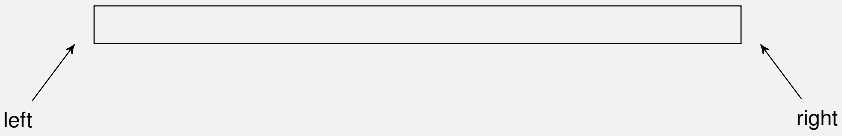
```
int partition(int key[], int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key, left, right);
    } while (left < right);
    return left;
}
```

Partition

```
int partition(int key[],int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key,left,right);
    } while (left < right);
    return left;
}
```

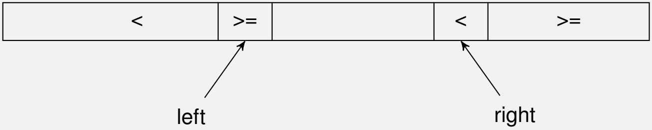
Partition

```c
int partition(int key[],int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key,left,right);
    } while (left < right);
    return left;
}
```



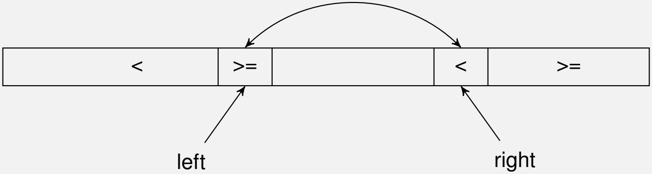
Partition

```
int partition(int key[], int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key, left, right);
    } while (left < right);
    return left;
}
```



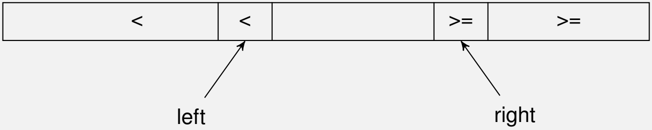
Partition

```c
int partition(int key[],int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key,left,right);
    } while (left < right);
    return left;
}
```



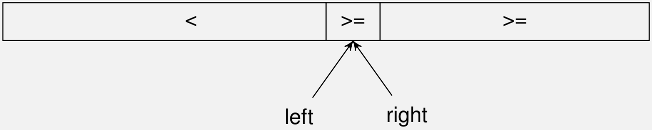
Partition

```
int partition(int key[],int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key,left,right);
    } while (left < right);
    return left;
}
```



Partition

```
int partition(int key[],int left, int right, int pivot){
    do {
        while (key[++left] < pivot);
        while ((left < right) && key[--right] >= pivot);
        myswap(key,left,right);
    } while (left < right);
    return left;
}
```



Radix-Exchange Sort

7	4	1	6	3	2	5	0
①11	①00	⓪01	①10	⓪11	⓪10	①01	⓪00

- Based on the **bit representation** of the keys
- For each step, based on the corresponding bit, partition the array into two parts: bit == 1 and bit == 0.

1	3	2	0
⓪01	⓪11	⓪10	⓪00

7	4	6	5
①11	①00	①10	①01

- Sort these two parts (using rescursive on the next bits or iterative)

0	1	2	3

4	5	6	7

- Append two ordered parts

0	1	2	3	4	5	6	7

Example

7	111	011	001	000	0
4	100	000	000	001	1
2	010	010	010	010	2
6	110	001	011	011	3
5	101	101	101	100	4
1	001	110	100	101	5
0	000	100	110	110	6
3	011	111	111	111	7

Radix-Exchange Sorting Implementation

```
void radixSort(int[] keys,int l, int r,
    int mask)
  int i = l,j = r;
  if (r<=l  mask==0) return;
  while (j!=i) {
    while ((a[i] & mask==0) && (i<j)) i++;
    while ((a[j] & mask==1) && (j>i)) j--;
    swap(a,i,j);
  }
  radixSort(a,l,j-1,mask>>1);
  radixSort(a,j,r,mask>>1);
}
```

Quick Sort vs. Radix-Exchange Sort

Similarities

- partition array
- sort subarray recursively

Differences

- Partitioning Method
  - RE partitions array based on the bit at corresponding position
  - Q partitions array based on the pivot value
- Time complexity
  - RE: O(bn)
  - Q: O($nlog_2n$)

Empirical Comparison

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

Table: Running time in miliseconds

Summary

- Internal Sort requires all elements available on the memory
- Insertion, Selection and Bubble Sort are simple but bad performance
- Shell, Merge, Quick and Heap Sort are more complex but good performance
- Radix Sort is not based on the value of keys but on their radix