data_tructures(&algorithms, lecture03)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

Elementary Sorting Algorithms

Bubble Sort, Insertion Sort, Selection Sort

01

Quick sort

Recursive quick sort, non recursive quick sort

02

Merge Sort

Merging process, recursive merge sort

03

Heap sort

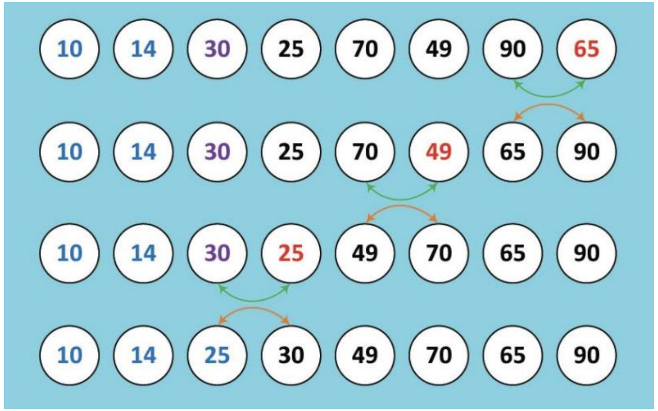
04

Elementary Sorting

Bubble sort, Insertion sort, Selection sort

Bubble Sort

- Key idea: going bottom-up, comparing 2 consecutive elements and let smaller element bubble-up
- After the 1st run, the smallest element will bubble-up to the 1st position and so on.



Bubble Sort

- Bubble sort implementation

```
void bubble_sort(int a[], const int int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > 0; j--)
        {
            if (a[j] < a[j - 1])
                swap(a, j, j - 1);
        }
    }
}
```

Bubble Sort

- Analysis
  - Worst case
    - n(n-1)/2 comparisons
    - n(n-1)/2 exchanges
  - Best case
    - n(n-1)/2 comparisons
    - 0 exchanges
  - => In general, $O(n^2)$

```
void bubble_sort(int a[], const int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > 0; j--)
        {
            if (a[j] < a[j - 1])
                swap(a, j, j - 1);
        }
    }
}
```

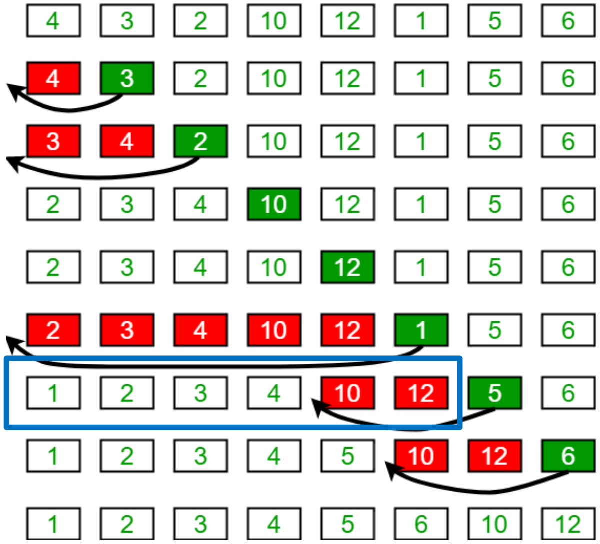
Optimized Bubble Sort

- Analysis
  - Worst case
    - n(n-1)/2 comparisons
    - n(n-1)/2 exchanges
  - Best case
    - n – 1 comparisons
    - 0 exchanges
  - => In best case, O(n)

```c
void bubble_sort(int a[], const int n)
{
    int need_swap;
    for (int i = 0; i < n - 1; i++)
    {
        need_swap = FALSE;
        for (int j = n - 1; j > 0; j--)
        {
            if (a[j] < a[j - 1])
            {
                swap(a, j, j - 1);
                need_swap = TRUE;
            }
        }
        if (!need_swap) break;
    }
}
```

Insertion sort

- Key idea: assume the collection is sorted from element 0 to i^{th}. Insert the $(i+1)^{th}$ element to the right place in the sorted part.



Insertion sort

- Insertion sort implementation.

```
void insertion_sort(int a[], const int n)
{
    for (int i = 1; i < n; i++)
    {
        int temp = a[i];
        int j = i - 1;
        for (; j >= 0 && a[j] > temp; j--)
            a[j + 1] = a[j];
        a[j + 1] = temp;
    }
}
```

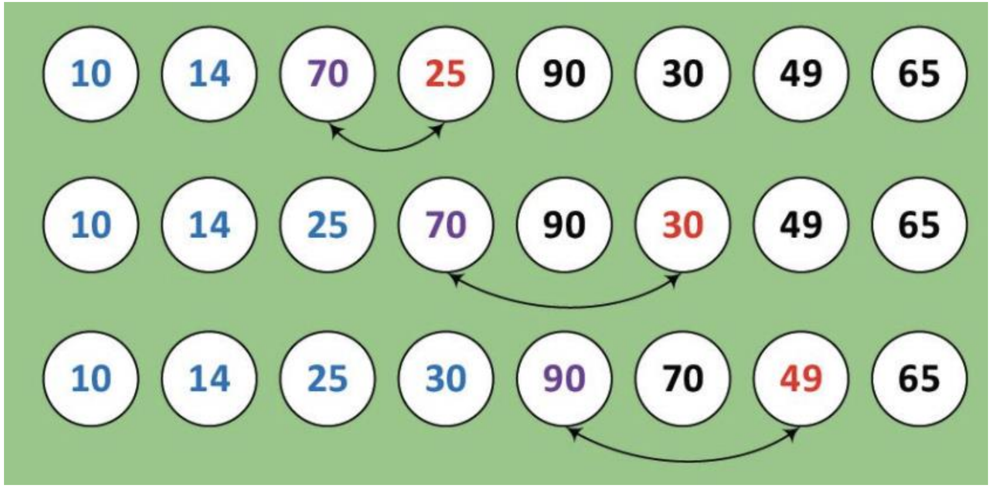
Insertion sort

- Analysis
  - Worst case
    - n(n-1)/2 comparisons
    - (n+1)(n-1)/2 exchanges
    - => $O(n^2)$
  - Best case
    - n-1 comparisons
    - n exchanges
    - => $O(n)$

```
void insertion_sort(int a[], const int n)
{
    for (int i = 1; i < n; i++)
    {
        int temp = a[i];
        int j = i - 1;
        for (; j >= 0 && a[j] > temp; j--)
            a[j + 1] = a[j];
        a[j + 1] = temp;
    }
}
```

Selection sort

- Key idea: Select the smallest element to put in the 1st position, the 2nd smallest element to put in the 2nd position and so on.



Selection sort

- Selection sort implementation

```
void selection_sort(int a[], const int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int imin = i;
        for (int j = i + 1; j < n; j++)
        {
            if (a[j] < a[imin]) imin = j;
        }
        if (imin != i) swap(a, i, imin);
    }
}
```

Selection sort

- Analysis
  - Worst case
    - n(n-1)/2 comparisons
    - n-1 exchanges
  - Best case
    - n(n-1)/2 comparisons
    - 0 exchanges
  - In general: $O(n^2)$

```
void selection_sort(int a[], const int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int imin = i;
        for (int j = i + 1; j < n; j++)
        {
            if (a[j] < a[imin]) imin = j;
        }
        if (imin != i) swap(a, i, imin);
    }
}
```

Comparing Elementary Sort Algorithms

Algorithm	Worst		Best		Average	
	Comp.	Exch.	Comp.	Exch.	Comp.	Exch.
Bubble	$n^2/2$	$n^2/2$	$n^2/2$	0	$n^2/2$	$n^2/2$
Optimized Bubble	$n^2/2$	$n^2/2$	n	0	?	?
Insertion	$n^2/2$	$n^2/2$	n	n	$n^2/4$	$n^2/4$
Selection	$n^2/2$	n	$n^2/2$	0	$n^2/2$	log(n)

[1]. Algorithms in C, Robert Sedgewick

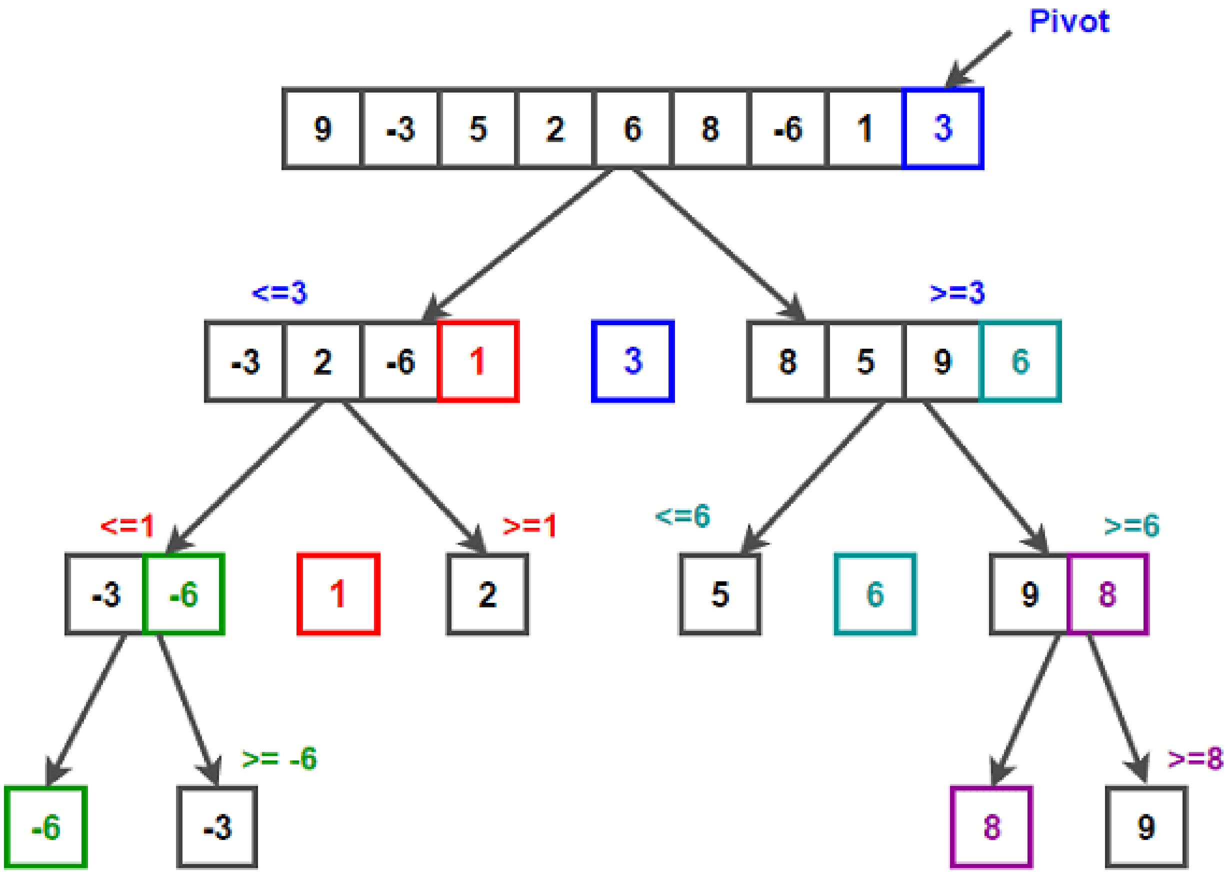
Quick Sort

General ideas, choosing pivot, recursive quicksort, …

QuickSort

- Key idea: rearrange the collection into 2 parts so that the left part is less than some specific element while the right part is greater than it. Continue doing that will sort the collection
- Quicksort is divide by conquer recursive algorithm
  - If collection A has 0 or 1 element, it's already sorted
  - Choose an element v in A (called pivot)
  - Partition A (excluding v) into 2 sub-collections A-left and A-right so that all elements in A-left is $\leq$ v and all elements in A-right is $>$ v
  - Repeat the process on A-left and A-right

Quicksort Illustration



QuickSort Partition

- How to choose pivot?
  - Begin, end, middle or random

```
int qs_partition(int a[], int lo, int hi)
{
    int pivot = lo, i = lo, j = hi;
    while (i < j)
    {
        while (i < j && a[i] <= pivot) i++;
        while (j > i && a[j] > pivot) j--;
        if (i < j) swap(a, i, j);
    }
    if (lo < j) swap(a, lo, j);
    return j;
}
```

Recursive QuickSort

```c
void quick_sort(int a[], const int n)
{
    quick_sort_aux(a, 0, n - 1);
}
void quick_sort_aux(int a[], const int lo, const int hi)
{
    if (lo < hi)
    {
        int pivot_pos = qs_partition(a, lo, hi);
        quick_sort_aux(a, lo, pivot_pos - 1);
        quick_sort_aux(a, pivot_pos + 1, hi);
    }
}
```

QuickSort Analysis

- Best case: pivot is always the middle
  - $T(n) = 2T(n/2) + cn \Rightarrow O(n\log n)$
- Worst case: pivot is always the smallest
  - $T(n) = T(n-1) + cn \Rightarrow O(n^2)$
- Average case: pivot is randomly distributed

$$T(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T(j)\right] + cn$$

  - $\Rightarrow O(n\log n)$

[2]. *Data structures and algorithms in C*, Mark Allen Weiss

Non-recursive QuickSort

```
#define push2(A, B)  push(B); push(A);
void quicksort(Item a[], int l, int r)
  { int i;
    stackinit(); push2(l, r);
    while (!stackempty())
      {
        l = pop(); r = pop();
        if (r <= l) continue;
        i = partition(a, l, r);
        if (i-l > r-i)
          { push2(l, i-1); push2(i+1, r); }
        else
          { push2(i+1, r); push2(l, i-1); }
      }
  }
```

[1]. *Algorithms in C*, Robert Sedgewick

Recursive vs Non-recursive QuickSort

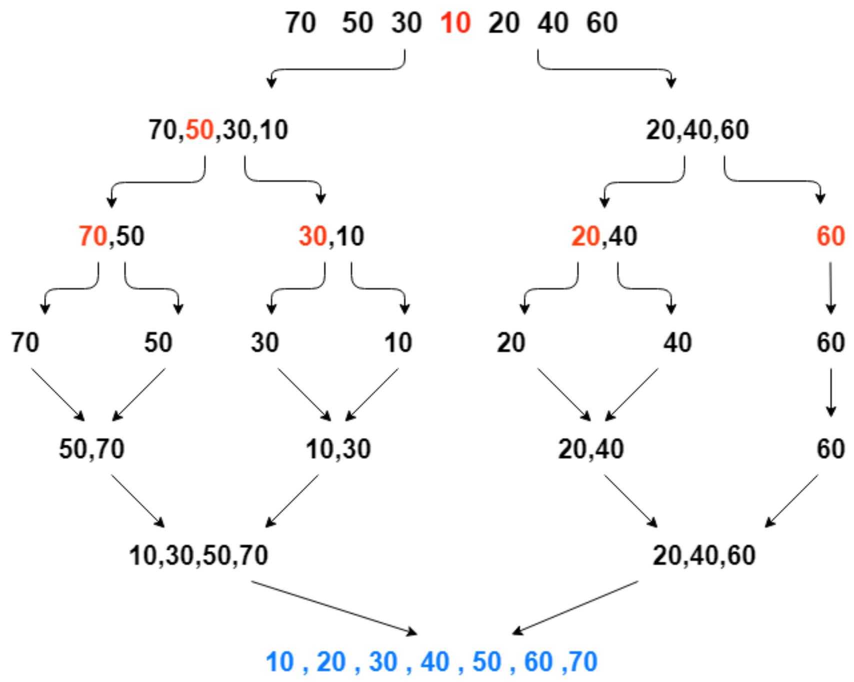
- Normally recursive is slower than iterative because of the overhead of recursive stack
- But QuickSort is tail-recursive so it doesn't really matter
- Some modified version of QuickSort use a threshold and if number of elements is less than it then an elementary sort will be used

Merge Sort

General ideas, merging, recursive mergesort, …

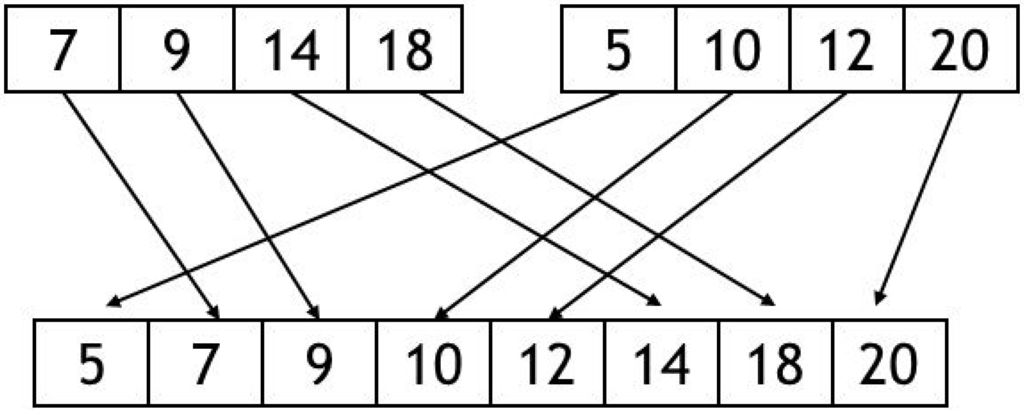
Merge Sort

- Key ideas: if we have 2 sorted collections, we can merge them to have 1 sorted collection.



Merge Sort

- Merge process: Repeatedly compare the 2 least elements and copy the smaller one to the 2nd collection.



Merge Sort

❖ Implementation of merge process

```
void merge(int a[], int lo, int mid, int hi)
{
    int i = lo, j = mid, k = 0, n = hi - lo + 1;
    int temp[n];
    while (i < mid && j <= hi)
    {
        if (a[i] < a[j]) temp[k++] = a[i++];
        else temp[k++] = a[j++];
    }
    if (i == mid) { // 1st part is done, there are
        for (; j <= hi; j++) temp[k++] = a[j];
    }
    else {          // 2nd part is done, there are
        for (; i < mid; i++) temp[k++] = a[i];
    }
    // copy temp -> a
    for (k = 0; k < n; k++) a[lo+k] = temp[k];
}
```

Recursive MergeSort

```
void merge_sort(int a[], const int n)
{
    merge_sort_aux(a, 0, n - 1);
}

void merge_sort_aux(int a[], int lo, int hi)
{
    // base case
    if (lo >= hi) return;
    // recursive case
    int mid = (lo + hi) / 2;
    merge_sort_aux(a, lo, mid);
    merge_sort_aux(a, mid + 1, hi);
    merge(a, lo, mid + 1, hi);
}
```

MergeSort Analysis

- The key is to merge 2 sorted collection so no best case
- $T(n) = 2\ T(n/2) + cn$
- => $O(n\log n)$

- Although mergesort's running time is $O(n \log n)$, it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory, and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably.

Other MergeSort Algorithms

- Non-recursive or bottom-up MergeSort

```
void mergesortBU(Item a[], int l, int r)
  { int i, m;
    for (m = 1; m <= r-l; m = m+m)
      for (i = l; i <= r-m; i += m+m)
        merge(a, i, i+m-1, min(i+m+m-1, r));
  }
```

- Still O(nlogn), still need to use extra space to copy

Other MergeSort Algorithms

- In-place MergeSort

```
left = first;  right = mid+1;
// One extra check:  can we SKIP the merge?
if ( x[mid].compareTo(x[right]) <= 0 )
   return;

while (left <= mid && right <= last)
{  // Select from left:  no change, just advance left
   if ( x[left].compareTo(x[right]) <= 0 )
      left++;
   // Select from right:  rotate [left..right] and correct
   else
   {  tmp = x[right];     // Will move to [left]
      System.arraycopy(x, left, x, left+1, right-left);
      x[left] = tmp;
      // EVERYTHING has moved up by one
      left++;  mid++;  right++;
   }
}
// Whatever remains in [right..last] is in place
```

- No need for extra space but $O(n^2)$