# Yu's Coding Garden

Every geek had a dream...

NEW!!!
I started to post **C++ Solutions for the famous book:**
***Cracking The Coding Interview (5th Edition)***
**Please Look at my gitHub: here**
**(Regularly update)**

**Note:**
***All the code provided for the leetcode questions have been tested and have passed both small and large tests of the online judge system from leetcode.***

**Tuesday, July 30, 2013**

## Common Sorting Algorithms: Concepts and Implementations

**Common Sorting Algorithms: Concepts and Implementations**

### Introduction

Understanding the basic concepts of the popular sorting algorithms, can not only help you better understand the data structure and algorithm from a different perspective, but also helps you make the computational complexity clearer in your mind. Besides, sorting related questions, are also hot questions in your software engineering job interview.

Many books, courses, and websites are providing massive materials of sorting algorithms. From my own experiences, among the long and tedious sources, http://www.sorting-algorithms.com/, and https://en.wikipedia.org/wiki/Sorting_algorithm are two good places you can learn the sorting algorithm intuitively, because there are animations shown in the website. If you have already familiar or have the basic concept of the sorting algorithms, it would help you easily memorize the details of specific algorithm.

In this blog, I'd like to review the common popular sorting algorithms, with the basic concepts, and tries to explain each one intuitively. Also the C++ implementation is the core content and are provided detailed with comments. This blog is more suitable for who have already implemented some of the sorting algorithms (at least you may write the selection sorting or bubble sorting in 1 or 2 minutes), but I will try to explain more for the novices.

### 1. What algorithms are covered ?

1. Bubble sort
2. Selection sort
3. Inset sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Bucket sort

+1

**Search This**

**Pages**

- Home

Useful link: C

**leetcode olc**

**Labels**

bubble sort (1)
insert sort. (1)
(1) O(n) (1) q
algorithm (1) st

**Blog Archiv**

▼ 2013 (10
► Aug 20

**2. Computational Complexity**

| | | |
|---|---|---|
| 1. | Bubble sort | O(n^2 ) |
| 2. | Selection sort | O(n^2) |
| 3. | Inset sort | O(n^2) |
| 4. | Merge sort | O(nlogn) |
| 5. | Quick sort | O(nlogn) |
| 6. | Heap sort | O(nlogn) |
| 7. | Bucket sort | O(n+k) |

How to know these in a fast way? Just memorize them! My experience is that

- If there are 2 loops in the code, it is O(n^2)
- If the algorithm is easily implemented, it is O(n^2)"
- Otherwise it is O(nlogn), expect the bucket sort.

**3. Concepts and Implementations**

The sorting problem is quite straightforward----sort the data array (ascending order). We will not go into the detail that what kind of data structure or data type are used, and not interested in the comparison function.

Here we define the general case:

*Input:*

*int n; //array length*

*int* A[n]; // unsorted int array*

*Output:*

*int* A; // sorted array (ascending order)*

*Function:*

*swap(int a, int b) // swap the value of a and b*

```
void swap(int &a,int &b){
 int tmp;
 tmp = a;
 a = b;
 b = tmp;
}
```

In the following, I'll show the key concept and the way how I remember these sorting algorithms.
NOTE that there may have different forms for one sorting algorithm, here just shows one of them.
NOTE that to better understand the following, I personally suggest read the code directly along with the explanations.

**Bubble sort**

---------------------------------------------------------------------------------

**Concept:**

Scan from start to end, compare every two elements, swap the max to the end.
Scan from start to end-1, compare every two elements, swap the max to the end-1.
Scan from start to end-2, compare every two elements, swap the max to the end-2.
...
Scan from start to start, end.

**Key Point:**

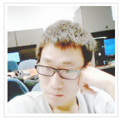
*if A[j]>A[j+1], swap(A[j], A[j+1]);*

**How to Memorize:**

Compare each ***pair*** and ***bubble*** the max value out and move to the last.

▼ Jul 20
Comm
and I
► May 2(
► Apr 20
► Mar 2(
► Feb 2(
► Jan 20
► 2012 (29

**My google +**



**Code:**

```
//Bubble Sort
void bubbleSort(int *A, int n){
  for (int i=n-1;i>0;i--){
    for (int j=1;j<=i;j++){
      if (A[j]<A[j-1]){
        swap(A[j],A[j-1])
      }
    }
  }
}
```

**Selection sort**

--------------------------------------------------------------------------------
**Concept:**
From 1st to last, find the min value,  store to the 1st position.
From 2nd to last, find the min value, store to the 2nd position.
From 3rd to last, find the min value, store to the 3rd position.
...
From last-1 to last, find the smaller one, store to the last-1 position. End.

**Key Point:**
k=i;  // store the current start position
if (A[j]<A[k]) {k=j;} // store the index of min value

**How to Memorize:**
***Select*** the ***min*** value and store to the front.

**Code:**

```
//Selection Sort
void selectSort(int *A, int n){
  for (int i=0;i<n-1;i++){
    int k=i; // k can be viewed as the index of min value
    for (int j=i+1;j<n;j++){ // find the min value
      if (A[j]<A[k]){k=j;}
    }
    swap(A[i],A[k]);  // store the min value to the start
  }
}
```

**Inset sort**

--------------------------------------------------------------------------------
**Concept:**
For each element A[i], the array A[0..i-1] is already sorted. Scan A[0..i-1], find the correct place and insert A[i].

**Key Point:**
Find the correct place and insert the A[i] in sorted A[0..i-1].
Consider A[0..i]:   [1,3,4,5,8,2],
So, A[i]=2.

Store it to a variable: tmp = A[i];
What we need to do now?
[1,3,4,5,8,**2**] ----> [1,**2**,3,4,5,8]
How to do this?
Keep moving each element to its right (A[j]=A[j-1]), until the next element is less than A[i].

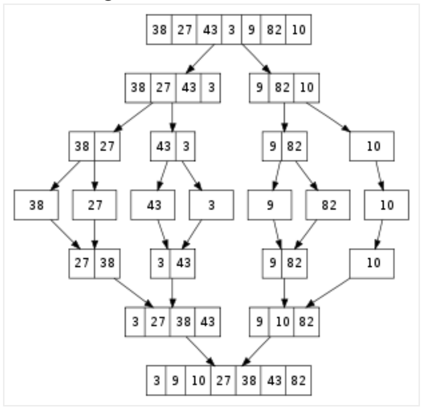
**How to Memorize:**
***Insert*** every element to its previous sorted array.

**Code:**

```
//Insert Sort
void insertSort(int *A, int n){
  for (int i=0;i<n;i++){
    int tmp = A[i];
    int j=i;
    while (j>0 && tmp<A[j-1]){
        A[j]=A[j-1];
        j--;
    }
    A[j]=tmp;
  }
}
```

**Merge sort**

**--------------------------------------------------------------------------------**
**Concept:**
Here I interpret merge sort using the recursion. So hopefully you have the basic idea what recursion is.
The idea is mainly considering the array into smaller subsets, merge the smallest subsets to smaller subset, merge smaller subsets to small subset ... until merge subset to the whole array. Merging process itself handles the sorting.



This figure (from wikipedia) shows the exact process of merge sort. Please go through the whole tree at the same time thinking it as a recursive problem, this will greatly help you understand the implementation of this algorithm.

**Key Point:**
Merge sort consist two parts:
(1) Recursion Part.
(2) Merge Part.

Recursive part, handles divided the current set to two parts, just like the concept of *divide-and-conquer:* Find the middle, divide into left and right subset and continue dividing in each subset. Recursion also keeps the two subset sorted for the merging.

Merge Part, is very very important for this algorithm, which merges two array to make a new sorted array.
How to do it ? Let's take a example.
Assume: A1=[1,5,7,8] and A2=[2,6,9,10,11,12,13]
What we need ?  A new sorted array A = [1,2,5,6,7,8,9,10,11,12,13]
OK, now at least we need a new array
A of length A1+A2, say,  A[ , , , , , , , ,].
How to put element in A and considering the order?
Set 3 pointers i,j,k, for A1, A2, and target array A.

```
A1=[1,5,7,8]
    i
A2=[2,6,9,10,11,12,13]
    j
A =[ , , , , , , , ,].
   k
```

From above one can clearly see, the 1st element in A (A[k]), should be min(A1[i],A2[j]), it is A1[i] = 1. So A1[i] is already in A, then we go to the next element in A1 using i++.  And the 1st element in A is filled, we have to go to the next one, so k++.

```
A1=[1,5,7,8]
      i
A2=[2,6,9,10,11,12,13]
    j
A =[1, , , , , , , ,].
     k
```

Next,  similarly compare A[i] and A[j],  get the smaller one and fill into the array A, and set the pointers properly.

```
A1=[1,5,7,8]
      i
A2=[2,6,9,10,11,12,13]
      j
A =[1,2, , , , , , ,].
       k
```

In such a way, the loop goes until the end of A1. At this time, the merge is NOT finished, we have to combine the rest elements of A2 into A.
Finally,  A = [1,2,5,6,7,8,9,10,11,12,13].

**How to Memorize:**
(1) Recursion Part: divide from the middle
(2) Merge Part:  merge two sorted array into one sorted array

**Code:**

```
//Merge Sort
void mergeSort(int *A, int st, int ed){
  if (st>=ed) {return;}
  int m = st+(ed-st)/2;
  mergeSort(A,st,m);
  mergeSort(A,m+1,ed);

  int *tmp = new int[ed-st];
  int k=0;
  int i=st;
  int j=m+1;

  while (i<m+1 && j<=ed){
    if (A[i]<A[j]){
       tmp[k++]=A[i++];
    }else{
 tmp[k++]=A[j++];
    }
  }
  while (i<m+1){tmp[k++]=A[i++];}
```

```
    while (j<=ed){tmp[k++]=A[j++];}

    for (int ii=0;ii<k;ii++){cout <<tmp[ii] <<" ";}
    cout << endl;

    for (int ii=st; ii<=ed;ii++){ A[ii] = tmp[ii-st];}
```

```
    delete [] tmp;

}
```

**Quick sort**

**-----------------------------------------------------------------------------------**
**Concept:**
This is also a *Divide and Conquer* algorithm. The idea is:  for an element pivot in the array, **place the elements less than pivot to its left, and elements greater than pivot to its right.** Do this same procedure to the two subsets (left and right), until all the elements are sorted.

**Key Point:**
Quick sort mainly consisted two parts:
(1) Recursive part: recursively apply the for the subsets.
(2) Reorder the set, where elements < pivot value are placed to its left, and vice versa.
This is the important part of the algorithm:
Consider the array
A=[8,3,5,6,4,1,9]
Here we choose the middle element as the pivot (also can select the 1st one or randomly select)
What we want to do ?
A[1,3,5,4,**6,**8,9],  then sort[1,3,5,4,6] and [8,9] recursively.

First, put pivot to the front (swap(A[0],A[pivot])):
A=[**6**,3,5,8,4,1,9]
Then set two pointers i and p, start from the 2nd element. **p points to the first element which is bigger than pivot.**

```
A=[6,3,5,8,4,1,9]
     i
     p
```

Compare A[i] with A[0], if A[i] < A[0], swap A[i] and A[p], goto next.

```
A=[6,3,5,8,4,1,9]
       i
       p
```

and

```
A=[6,3,5,8,4,1,9]
         i
         p
```

here A[i]>A[0], no swap, i++

```
A=[6,3,5,8,4,1,9]
           i
         p
```

**4<6, swap A[i] and A[p], because A[p] was found larger than A[0]**

```
A=[6,3,5,4,8,1,9]
             i
           p
```

Still have to swap:

```
A=[6,3,5,4,1,8,9]
               i
```

p
No swap, i goes to the end, and now p is the place where 0..p-1 < pivot, and p..n > pivot.
Last step is to swap A[0] and A[p-1]:
A[1,3,5,4,6,8,9]

**How to Memorize:**
(1) Recursion (divide-and-conquer)
(2) Select a Pivot
(3) Aim: reorder elements<pivot to the left and elements>pivot to the right
(4) Set pivot to front
(5) Set two pinter

**Code:**

```
//Quick Sort
void quickSort(int *A, int st, int ed){
 if(st>=ed){return;}
 int pivot = st+(ed-st)/2;
 swap(A[st],A[pivot]);
 int pos = st+1;
 for (int i=st+1;i<ed;i++){
   if (A[i]<A[st]){
     swap(A[i],A[pos]);
     pos++;
   }
 }
 swap(A[pos-1],A[st]);
 quickSort(A,st,pos-1);
 quickSort(A,pos,ed);

}
```

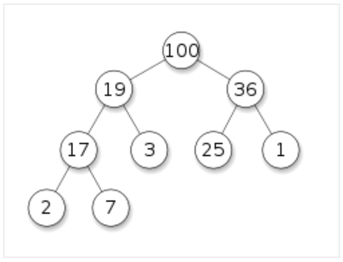
**Heap sort**

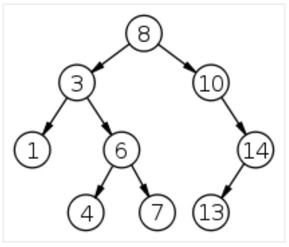
**-----------------------------------------------------------------------------------**
**Concept:**
Heap sort is based on the data structure **heap,** which is a tree structure with a nice ordering property, can be used for sorting. The heap sort algorithm consists of two parts:
(1) Construct the heap
(2) Get the root node each time, update the heap, until all the node are removed.

First let's see what is heap (in my own word):
A heap, briefly speaking, is a tree structure, where the value of each parent node is greater/smaller than its children. Practically in the heap sort, we use the specific kind of heap----binary heap.

Binary heap, is a complete binary tree, also keeps the property that each root value is greater or smaller than its left and right children. Thus, the root of the tree is the biggest (called max heap) or the smallest (called min heap) element in the tree.

Caution!!! A Heap is NOT a binary search tree(BST)! A BST can apply in-order traversal to get the sorted array, heap CANNOT guarantee the ordering within same level, so it does not have a specific requirement of the order for the left and right children. e.g. see the figure below(from wikipedia)

A heap structure: 

A binary search tree structure: 

- How to construct the heap?

Consider a unsorted array, we want to construct a heap. The intuitive way is to obtain every node and add to the tree structure(TreeNode* blablabla...), but a simpler way is just use the array itself, to represent the tree and modify the value to construct the heap.

Tree (Array representation):
Root node: A[0].
Left child of A[i]: 2*i+1
Right child of A[i]: 2*i+2
Parent of A[i]: (i-1)/2

Construct a heap:
An operation downshift is used here.
The idea of downshift is to adjust the current node to its proper position in its downside direction.
Given a node, compare the value to the bigger one of its left and right children, is the value is smaller than the bigger children, then swap them. And keep checking the node (here in the new position after swapping), until it is no less than its children.
To construct the heap, from the end of the array, we downshift every node to the first in the array.

- How to get the sorted array according to the heap?

Given a max heap, the value of root node is the biggest in the heap, each time remove the top node and store to the array. But it's not enough! We have to keep the heap, so there needs a update of the remaining nodes. An efficient way is just swap the root node and the last element of the tree, remove the last node (just let the length of the array -1), downshift the new root node, a heap is updated.

**Key Point:**
**(1) How to construct the heap?**
Use array to represent the tree structure.
Recursively downshift every node.
**(2) How to get the sorted array according to the heap?**
Each time remove the root of the heap, swap the last node to the root and downshift it.
Until all the nodes are removed.

**How to Memorize:**
This algorithm is very particular and requires the skill of heap operations (construct, downshift, update, etc.).

In my opinion, first you get to know the data structure heap, then the heap sort suddenly becomes a piece of cake!

**Code:**

```
//Heap Sort
void downshift(int* A, int n, int parent){
  if (parent<0 ){return;}
  int left = parent*2+1;
  int right = parent*2+2;
  int mxch;
  if (left>=n) {return;}
  if (right>=n) {mxch=left;}
  else{mxch = A[left]>=A[right]?left:right;}
  if (A[parent]<A[mxch]){
    swap(A[parent],A[mxch]);
    downshift(A, n,mxch);
  }
}

void constructHeap(int *A, int n){
  int parent=(n-2)/2;
  for ( ;parent>=0;parent--){
    downshift(A, n, parent);
  }
}

void heapSort(int *A, int n){
constructHeap(A,n);
int i=n-1;
int *B=new int[n];
while (i>=0){
  B[n-i]=A[0]; //get the biggest in the heap
  swap(A[0],A[i]);
  downshift(A,i,0);
  i--;
}
A=B;
}
```

**Bucket sort (coming soon)**

---------------------------------------------------------------------------------
**Concept:**
**Key Point:**
**How to Memorize:**
**Code:**

```
aaa
```

Posted by Yu Zhu at 11:08 AM +1 Recommend this on Google

Labels: bubble sort, bucket sort, heap sort, insert sort., merge sort, quick sort, selection sort, sorting algorithm

## 4 comments:

胡言 August 2, 2013 at 9:54 AM

Is there a memory leak in merge sort code?

Reply

Replies

**Yu Zhu** August 2, 2013 at 10:33 AM

Do you mean there needs a " delete [] tmp" in the function?

胡言 August 6, 2013 at 2:27 PM

Shouldn't it be deleted before return?

**Yu Zhu** August 6, 2013 at 5:19 PM

Yes! It does need to delete when new is used. Thank you very much for your comment. I've correct it.

And it will be a lot easier if using the vector instead of the dynamic array in the code.

:)

**Reply**

Enter your comment...

**Comment as:** Google Account

**Publish** Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)