Data Structure and Algorithm

Class 11

Seongjin Lee

Updated: 2017-03-06
DSA_2017_11

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Sorting

Sorting

Types of Sorting

- Internal Sorting - The list is small enough to sort entirely in main memory
- External Sorting - The list is too large to sort in main memory

Internal sorting algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Shell sort
- Heap sort
- Radix sort

Bubble sort

Bubble sort

```
repeat until no swaps
    for i from 0 to n-2
        if ith and i+1th elements out of order
            swap them
```

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1^{st} run	4	8	3	15	50	9	20	cmp 3 and 4, swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1^{st} run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1^{st} run	4	8	3	15	50	9	20	cmp 4 and 5, no swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1^{st} run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1^{st} run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1^{st} run	4	8	3	15	9	50	20	cmp 5 and 6, swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1^{st} run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1^{st} run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1^{st} run	4	8	3	15	9	50	20	cmp 5 and 6, swap
1^{st} run	4	8	3	15	9	20	50	cmp 6 and 7, swap

Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1^{st} run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1^{st} run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1^{st} run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1^{st} run	4	8	3	15	9	50	20	cmp 5 and 6, swap
1^{st} run	4	8	3	15	9	20	50	cmp 6 and 7, swap
2^{nd} run	4	8	3	15	9	20	50	cmp 1 and 2, no swap
2^{nd} run	4	3	8	15	9	20	50	cmp 2 and 3, swap
2^{nd} run	4	3	8	15	9	20	50	cmp 3 and 4, no swap
2^{nd} run	4	3	8	15	9	20	50	cmp 4 and 5, no swap
2^{nd} run	4	3	8	9	15	20	50	cmp 5 and 6, swap
3^{rd} run	3	4	8	9	15	20	50	1, 2 swap
4^{th} run	3	4	8	9	15	20	50	no swap
5^{th} run	3	4	8	9	15	20	50	no swap
6^{th} run	3	4	8	9	15	20	50	no swap

Bubble sort: the code (codes/bubble.c) I

```
// Unoptimized and Optimized implementation of bubble sort
#include <stdio.h>

void swap(int *x, int *y);
void bubble_unoptimized(int list[], int size);
void bubble(int list[], int size);
void printArray(int list[], int size);

int main(){
    int list[] = {30, 28, 18, 29, 48, 40};
    // get the number of items in the list
    int n = sizeof(list)/sizeof(list[0]);

    printf("initial list: \t");
    printArray(list, n);
```

Bubble sort: the code (codes/bubble.c) II

```
// bubble_unoptimized(list, n);
    bubble(list, n);

    printf("sorted list: \t");
    printArray(list, n);

    return 0;
}

void swap(int *x, int *y){ // swap the two value
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```c

void printArray(int list[], int size){
    for ( int i = 0; i < size; i++)
        printf("%d ", list[i]);
    printf("\n");
}


// unoptimized bubble sort with worst case of O(n^2)
void bubble_unoptimized(int list[], int size){
    int i, j;
    for (i = 0; i < size-1; i++)
        for (j = 0; j < size-i-1; j++) // skip sorted items in the list
            if (list[j] > list[j+1])
                swap(&list[j], &list[j+1]);
}
```

Bubble sort: the code (codes/bubble.c) IV

```c

// optimzed bubble sort
void bubble(int list[], int size){
    int i, j;
    int swapped = 1;
    // loop until no two elements were swapped by inner loop
    for (i = size-1; swapped > 0; i--){
        swapped = 0;
        for (j = 0; j < i; j++)
            if (list[j] > list[j+1]){
                swap(&list[j], &list[j+1]);
                swapped = 1;
            }
    }
}
```

Bubble sort: Time complexity

loop	number of comparison
1	n-1
2	n-2
3	n-3
.	.
.	.
.	.
n-1	1

Time complexity: Worst case $O(n^2)$

Selection sort

Selection sort

```
for i from 0 to n-1
    find smallest element between ith and n-1th
    swap smallest with ith element
```

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap
2^{nd} run	3	4	8	15	50	9	20	midx 4, min 4

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap
2^{nd} run	3	4	8	15	50	9	20	midx 4, min 4
3^{rd} run	3	4	8	15	50	9	20	midx 8, min 8

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap
2^{nd} run	3	4	8	15	50	9	20	midx 4, min 4
3^{rd} run	3	4	8	15	50	9	20	midx 8, min 8
4^{th} run	3	4	8	9	50	15	20	midx 15, min 9, swap

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap
2^{nd} run	3	4	8	15	50	9	20	midx 4, min 4
3^{rd} run	3	4	8	15	50	9	20	midx 8, min 8
4^{th} run	3	4	8	9	50	15	20	midx 15, min 9, swap
4^{th} run	3	4	8	9	15	50	20	midx 50, min 15, swap

Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1^{st} run	3	4	8	15	50	9	20	midx 15, min 3, swap
2^{nd} run	3	4	8	15	50	9	20	midx 4, min 4
3^{rd} run	3	4	8	15	50	9	20	midx 8, min 8
4^{th} run	3	4	8	9	50	15	20	midx 15, min 9, swap
4^{th} run	3	4	8	9	15	50	20	midx 50, min 15, swap
5^{th} run	3	4	8	9	15	20	50	midx 50, min 20, swap

Selection sort: the Code (`codes/selection.c`)

```c
// selection sort
void selection(int list[], int size){
   int i, j, midx;
   for (i = 0; i < size-1 ; i++){
      // find the minimum element in unsorted list
      midx= i;
      for (j = i+1; j < size; j++)
         if (list[j] < list[midx])
            midx = j;

      // swap the minimum element with the first element
      swap(&list[midx], &list[i]);
   }
}
```

time complexity: $O(n^2)$

Insertion sort

Insertion sort

```
for i from 1 to n-1
    call 0th through i-1th elements the ‘‘sorted side’’
    remove ith element
    insert it into sorted side in order
```

Insertion sort

idx	1	2	3	4	5	
	36	28	30	17	23	initial state
1	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side

Insertion sort

idx	1	2	3	4	5	
	36	28	30	17	23	initial state
1	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2	28	30	36	17	23	30 is smaller than 36, move it to the sorted side

Insertion sort

idx	1	2	3	4	5	
	36	28	30	17	23	initial state
1	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2	28	30	36	17	23	30 is smaller than 36, move it to the sorted side
3	17	28	30	36	23	17 is smaller than 36, 30, and 28, move it to the sorted side

Insertion sort

idx	1	2	3	4	5	
	36	28	30	17	23	initial state
1	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2	28	30	36	17	23	30 is smaller than 36, move it to the sorted side
3	17	28	30	36	23	17 is smaller than 36, 30, and 28, move it to the sorted side
4	17	23	28	30	36	23 is smaller than 36, 30, and 28, move it to the sorted side

Insertion sort: the Code (codes/insertion.c)

```c
// insertion sort
void insertion(int list[], int size){
   int i, j, key;
   for (i = 1; i < size; i++){
      key = list[i];

      // if item is greater than the key
      // move the item in list to one position ahead
      for (j = i-1 ; j >= 0 && list[j] > key; j--)
         list[j+1] = list[j];
      list[j+1] = key;
   }
}
```

time complexity: worst case $O(n^2)$, best case $O(n)$

Shell Sort

Shell sort

Shell sort is based on insertion sort algorithm to reduce the number of shifts. For example, if the list in decreasing order and we want to sort in increasing order, the far left record has to be moved to the far left, one by one.

https://www.youtube.com/watch?v=1yDcmjLTWOg

Shell sort: the Code (codes/shell.c)

```c
void shellsort(int list[], int n){
   int i, gap;
   for (gap = n/2; gap > 0 ; gap = gap/2){
      if ( (gap%2) == 0 ) gap++;
      for (i = 0; i < gap; i++)
         incremental_insert(list, i, n-1, gap);
   }
}
```

Shell sort: the Code (codes/shell.c)

```
void incremental_insert(int list[], int first, int last, int
    gap){
  int i, j, key;
  for (i = first + gap; i <= last; i = i + gap){
    key = list[i];
    for ( j = i-gap; j >= first && key < list[j]; j = j - gap
        )
      list[j + gap] = list[j];
    list[j + gap] = key;
  }
}
```

Quick Sort

Quick Sort

Divide and Conquer

- two phase
- split and control

using recursion

- stack is needed

Best average time

- $O(n \cdot log_2 n)$

Quick sort: The principle

1. Pick a "**pivot**" element
2. "**partition**" the array into *three* parts
   - 1^{st} part: all elements in this is less than the pivot
   - 2^{nd} part: the pivot itself (only one element)
   - 3^{rd} part: all elements in this part is greater than or equal to the pivot
3. Recursively apply quick sort to the first and the third part
   - swap the out of order entries from the first and the third part

Quick Sort

Initial list	31	11	41	12	51	90	20	64	53	55	32

31	11	41	12	51	90	20	64	53	55	32

$pivot$=31

- examine the first, and last item of the full list that is 31 (first, pivot), and 32 (last)
- search forward until we find an item larger than the pivot $41 > 31$
- search backward until we find an item smaller than the pivot $20 < 31$
- swap the two items

31	11	20	12	51	90	41	64	53	55	32

$pivot$=31

Quick Sort

$$\underbrace{\boxed{31}\ \boxed{11}\ \boxed{20}\ \boxed{12}\ \boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{pivot=31}$$

- repeat the process until the left and right indices cross each other
- then swap the item with the pivot

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}\ \boxed{31}\ \boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{pivot=31}$$

Quick Sort

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{\color{red}{31}}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

- repeat the quick sort for the left and right part

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{\color{red}{31}}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

Quick Sort

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

- repeat the quick sort for the left and right part

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\underbrace{\boxed{12}}_{left}\ \underbrace{\boxed{11}}_{pivot}\ \underbrace{\boxed{20}}_{right}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

Quick Sort

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

- repeat the quick sort for the left and right part

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\underbrace{\boxed{12}}_{left}\ \underbrace{\boxed{11}}_{pivot}\ \underbrace{\boxed{20}}_{right}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\underbrace{\boxed{11}}_{left}\ \underbrace{\boxed{12}}_{pivot}\ \underbrace{\boxed{20}}_{right}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

Quick Sort

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\quad \underbrace{\boxed{31}}_{pivot}\quad \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

Quick Sort

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{32}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}$$

Quick Sort

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{32}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}$$

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{41}\ \boxed{32}\ \boxed{51}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}$$

Quick Sort

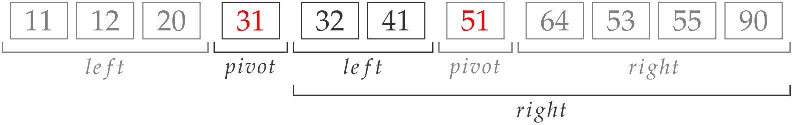
$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{90}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{32}}_{right}$$

$$\underbrace{\boxed{12}\ \boxed{11}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{51}\ \boxed{32}\ \boxed{41}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}$$

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\boxed{41}\ \boxed{32}\ \boxed{51}\ \boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}$$

$$\underbrace{\boxed{11}\ \boxed{12}\ \boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\underbrace{\boxed{41}\ \boxed{32}}_{left}\ \underbrace{\boxed{51}}_{pivot}\ \underbrace{\boxed{64}\ \boxed{53}\ \boxed{55}\ \boxed{90}}_{right}}_{right}$$

Quick Sort

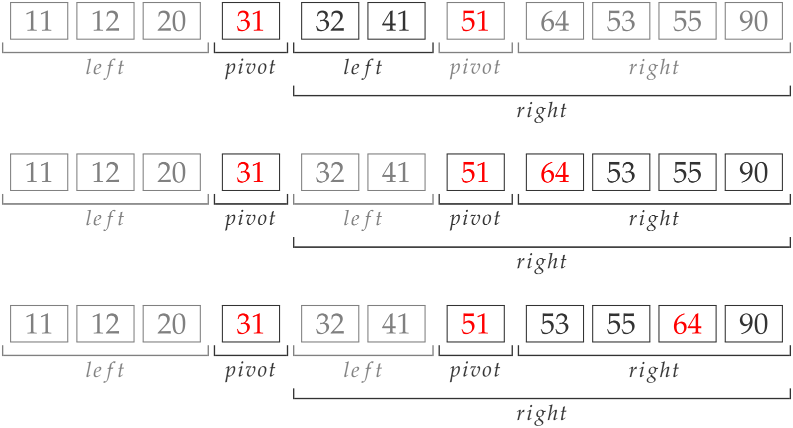

11 12 20
31
32 41
51
64 53 55 90
left
pivot
left
pivot
right
right

Quick Sort

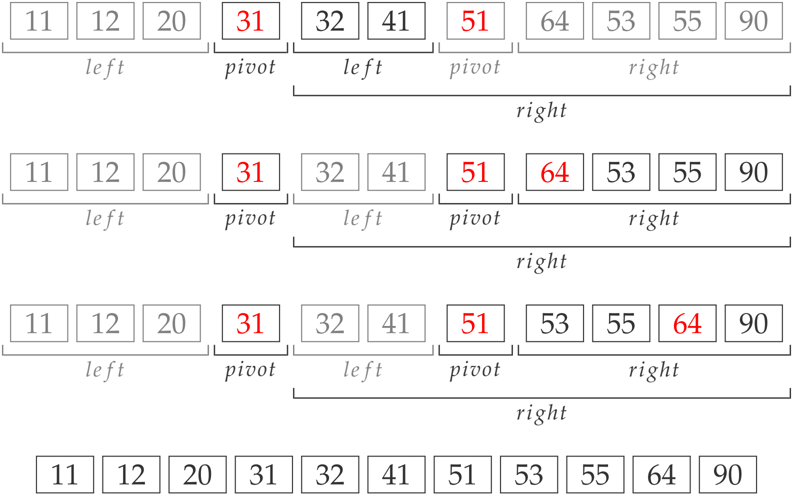
$$\underbrace{\boxed{11}\,\boxed{12}\,\boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\underbrace{\boxed{32}\,\boxed{41}}_{left}\ \underbrace{\boxed{51}}_{pivot}\ \underbrace{\boxed{64}\,\boxed{53}\,\boxed{55}\,\boxed{90}}_{right}}_{right}$$

$$\underbrace{\boxed{11}\,\boxed{12}\,\boxed{20}}_{left}\ \underbrace{\boxed{31}}_{pivot}\ \underbrace{\underbrace{\boxed{32}\,\boxed{41}}_{left}\ \underbrace{\boxed{51}}_{pivot}\ \underbrace{\boxed{64}\,\boxed{53}\,\boxed{55}\,\boxed{90}}_{right}}_{right}$$

Quick Sort


11 12 20 31 32 41 51 64 53 55 90
left
pivot
left
pivot
right
right
11 12 20 31 32 41 51 64 53 55 90
left
pivot
left
pivot
right
right
11 12 20 31 32 41 51 53 55 64 90
left
pivot
left
pivot
right
right

Quick Sort



11 12 20 31 32 41 51 64 53 55 90
left pivot left pivot right
right
11 12 20 31 32 41 51 64 53 55 90
left pivot left pivot right
right
11 12 20 31 32 41 51 53 55 64 90
left pivot left pivot right
right
11 12 20 31 32 41 51 53 55 64 90

Quick Sort: the Code (codes/quick.c)

```
    if (left < right){
                   int pivot = partition(list, n, left, right);
        quicksort(list, n, left, pivot-1);
        quicksort(list, n, pivot+1, right);
    }
}
```

Quick Sort: the Code (codes/quick.c)

```
int pivot, temp;
int low, high;
low = left;
high = right +1;
pivot = list[left];
do{
    do
        low++;
    while(low <= right && list[low] < pivot);
    do
        high−−;
    while (high >= left && list[high] > pivot);
    if (low < high) {
            swap(list[low], list[high], temp);
        printArray(list, n);
    }
} while (low < high);
swap(list[left], list[high], temp);
printArray(list, n);
```

Quick Sort: time complexity

Average case: $O(n \cdot log_2 n)$, when elements can be split into "equal size"

Let $T(n)$ be average time to sort n records

$$
\begin{aligned}
T(n) &= c \cdot n + 2 \cdot T(n/2) \\
&\leq c \cdot n + 2 \cdot (c \cdot n/2 + 2 \cdot T(n/4)) \\
&\leq 2 \cdot c \cdot n + 4 \cdot T(n/4)) \\
&\dots \\
&\leq c \cdot n \cdot log_2 n + n \cdot T(1)) \\
&O(n \cdot log_2 n)
\end{aligned}
$$

Worst case: $O(n^2)$

Quick Sort

Other approaches:

- http://me.dt.in.th/page/Quicksort/
- https://ece.uwaterloo.ca/~cmoreno/ece250/quick-sort-complete-example.pdf

Heap Sort

Heap Sort

Utilize the min heap structure

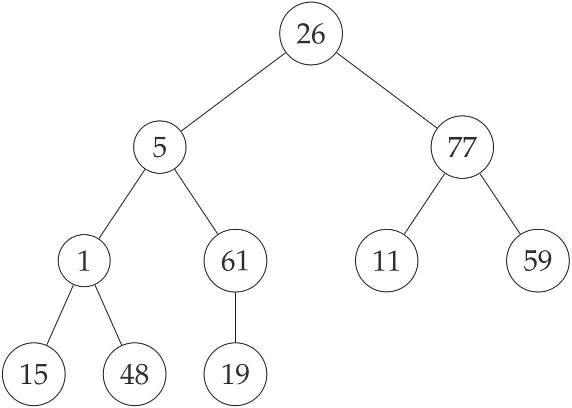
time complexity

- average case: $O(n \cdot log_2 n)$
- average case: $O(n \cdot log_2 n)$

First insert the items to the binary tree and delete the min values from the min heap and put it in the sorted array.

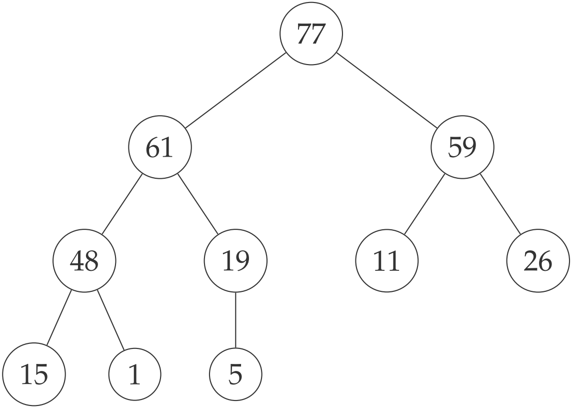
Heap sorting process

- input list: $(26, 5, 77, 1, 61, 11, 59, 15, 48, 19)$



[1]	26
[2]	5
[3]	77
[4]	1
[5]	61
[6]	11
[7]	59
[8]	15
[9]	48
[10]	19

Initial max heap construction



```
// heapify a subtree rooted with node i
// i: an index in list[]; n: is size of heap
void heapify(int list[], int n, int i)
{
    int temp;
    int largest = i; // Initialize largest as root
    int left = 2*i + 1; // left = 2*i + 1
    int right = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (left < n && list[left] > list[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && list[right] > list[largest])
        largest = right;

    // If largest is not root
```

```
    if (largest != i){
        swap(list[i], list[largest], temp);

    // Recursively heapify the affected sub−tree
    heapify(list, n, largest);
    }
}

// heap sort main body
void heapsort(int list[], int n)
{
    int temp;
    for (int i = n / 2 − 1; i >= 0; i−−) // Build heap (rearrange list)
        heapify(list, n, i);
    for (int i=n−1; i>=0; i−−){ // extract an element from heap
        swap(list[0], list[i], temp); // Move current root to end
        heapify(list, i, 0); // call max heapify on the reduced heap
    }
}
```

Heap Sort

Worst case time complexity:

$$log_2 n + log_2(n-1) + \cdots + log_2 2 = O(n \cdot log_2 n)$$

Radix Sort

Radix Sort

- It is a kind of distributive sort
- repeats the following 3 steps
  1. comparison
  2. distribution
  3. merging

Initial unsorted array

110	12	380	2	32	41	151	253

First consider the One's place

11**0**	1**2**	38**0**	**2**	3**2**	4**1**	15**1**	25**3**
11**0**	38**0**	4**1**	15**1**	1**2**	**2**	3**2**	25**3**

Consider the Ten's place

1**1**0	**8**0	**4**1	1**5**1	**1**2	2	**3**2	2**5**3
2	1**1**0	**1**2	**3**2	**4**1	1**5**1	2**5**3	3**8**0

Consider the Hundred's place

2	**1**10	12	32	41	**1**51	**2**53	**3**80
2	12	32	41	**1**10	**1**51	**2**53	**3**80

Radix Sort: Time complexity

Let there be d digits in input integers.

In general, Radix sort takes $O(d \cdot (n + b))$ time where b is the base of the given number. Assume that $k = max(n)$, then $d = O(log_b k)$. The overall

time complexity becomes $O((n + b) \cdot log_b k)$ If we further assume that

$k = n^c$, where c is constant, the time complexity becomes $O(n \cdot log_b n)$