Quick Sort

```
partition(A[], i, j)

QuickSort(A[], i, j) {
    if (i >= j) return;
    pivot_pos = partition(A[], i, j);
    QuickSort(A[], i, pivot_pos-1);
    QuickSort(A[], pivot_pos+1, j); // pos Not included
```

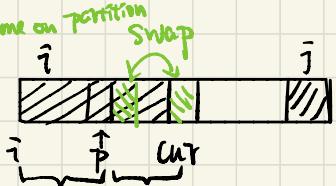
Running Time

if: $T(n) = 2T(\frac{n}{2}) + n \Rightarrow \theta(n \log n)$

$T(n) = T(\frac{n}{5}) + T(\frac{2n}{5}) + n \Rightarrow \theta(n \log n)$

$T(n) = T(\frac{n}{1000}) + T(\frac{999n}{1000}) + n \Rightarrow \theta(n \log n)$

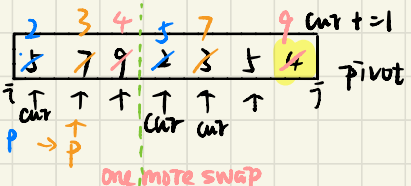
if $T(n) = T(n-1) + n \Rightarrow \theta(n^2)$ — Spending most of the time on partition




```
Partition(A[], i, j) {   O(n)      range (inclusive)
    pivot = A[j];
    p = i-1;  // denote empty range      p: last num ≤ pivot
    for(cur = i, ..., j-1)
        if (A[cur] ≤ pivot) {
            swap(A[cur], A[p+1]);
            p += 1;
        }
    swap(A[j], A[p+1]) //swap pivot
    return p+1;
```

case ① A[cur] > pivot
cur += 1

case ② A[cur] ≤ pivot
swap(A[], p+1, cur)
p += 1
cur += 1

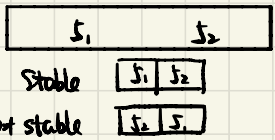


⇓ merge last swap into for loop

```
for(cur=i, ..., j)
    if (A[cur] ≤ pivot) {
        swap(A[cur], A[p+1]);
        p += 1;
    }
```

Stable sort

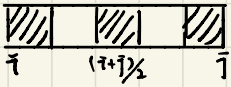
The algo respects the relative order of the same elements.



Pivot selection

① Random selection

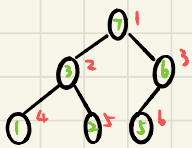
② Median of three (cheaper)



Heap Sort

Operations	Heap	Sorted Array
- insert	O(lgn)	O(n) because of this.
- getMax	O(1)	O(1) We choose heap over sorted array
- deleteMax	O(lgn)	O(1)

logical view



- "Almost full" binary tree
- parent ≥ both children

physical view

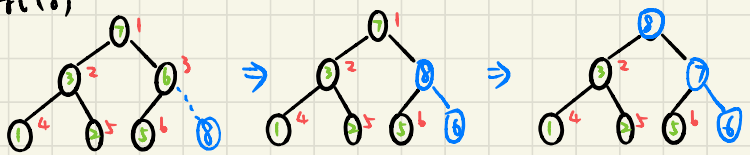
A

7	3	6	1	2	5
1	2	3	4	5	6

parent(i) = i/2 // parent of i
left child(i) = 2*i // --- of i
right child(i) = 2*i+1

- Insertion

insert(8)



O(height)
O(lgn)

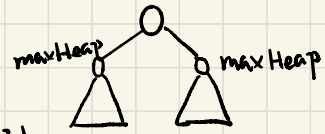
```
insert(int e) {
    n = n+1;
    A[n] = e;
    i = n;
    while(parent(i) > 0 && A[parent(i)] < A[i]) {
        swap(A[parent(i)], A[i]);
        i = parent(i);
    }
}
```

- Delete

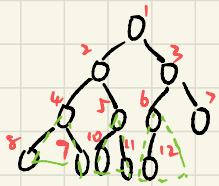
swap the root with last node, then heapify down.

```
deleteMax() {
    swap(A[1], A[n]);
    n = n-1;
    heapify(A[], 1);
}
```

```
heapify(A[], i){   O(logn) Bounded by height
    int largest = i;   // n is size of A
    if (leftChild(i) ≤ n && A[leftchild(i)] > A[largest])
        largest = leftChild(i);
    if (rightChild(i) ≤ n && A[rightChild(i)] > A[largest])
        largest = rightChild(i);
    if (largest == i) return;
    Swap(A[i], A[largest]);
    heapify(A[], largest);
}
```

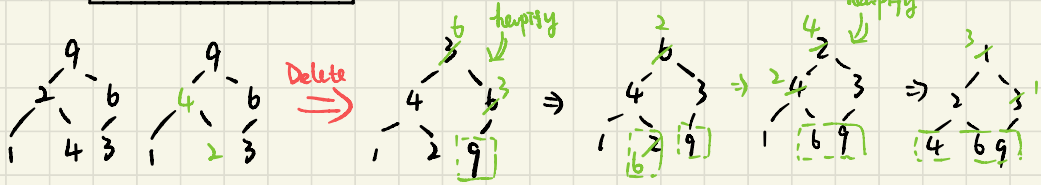


- HeapSort
  - make_heap(A[])
  - for(i=1, ---, n-1){
      deleteMax(A[]) // Not actually delete the node
    }
- make_heap
  #1. insert n times O(nlogn)
  #2. for(i = n/2 --- 1) // 6 is the smallest non-trivial heap
      heapify(A, i) // O(n) since most heap have few height



A

9	2	6	1	4	3



Theorem: comparsion based algo runs in $\Omega(n \log n)$

Counting Sort (Not in-place) (This particular version is stable)

Overall $O(n+k)$ Space $O(k)$?

① for $i = 0, \cdots n-1$) $O(n)$

$B[A[i]] += 1$

② for $(i = 1, \cdots k)$ $O(k)$

$B[i] += B[i-1]$;

③ for $(i = n-1, \cdots, 0)$

$C[--B[A[i]]] = A[i]$

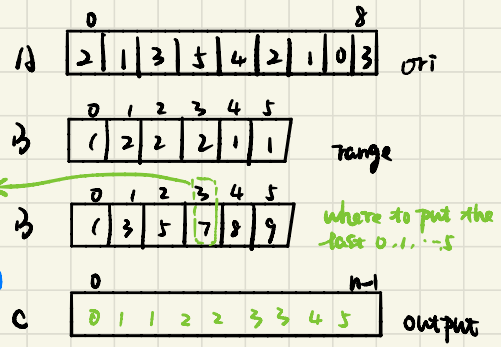
$B[A[i]] -= 1$

return C

decrement first

change to for $(i = 0, \cdots, n-1)$

then it's not stable



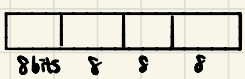
Radix Sort (Digit Sort) positive base 10 integer Stable

$O(d(n+k))$

max # of digit is d

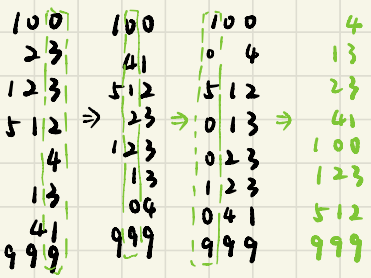
k is the range in counting sort

32 bits



$d = 4$ times counting sort

$k = 2^8 = 256$

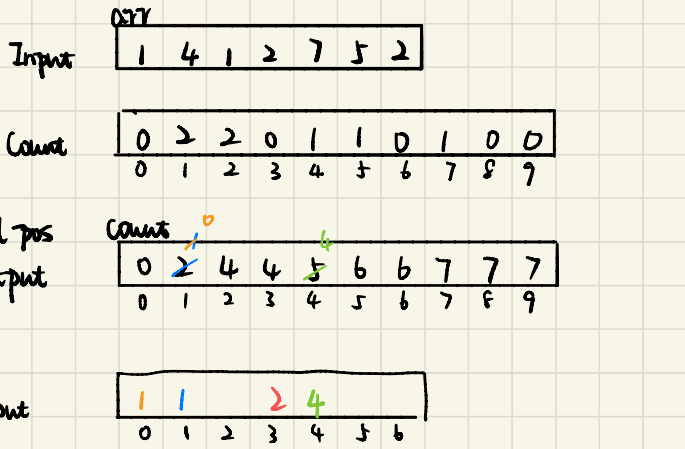


is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n+k)$. There are d passes, and so the total time for radix sort is $\Theta(d(n+k))$. ■

d is the length of digits, in this case: 3

We have to go over n numbers, and every possible digit for 0 to k

Counting Sort



output[count[arr[i]] - 1] = arr[i]

count[arr[i]]--

i = 0, arr[0] = 1, count[1] = 2 - 1 = 1

i = 1, arr[1] = 4, count[4] = 5 - 1 = 4

i = 2 arr[2] = 1, count[1] = 1 - 1 = 0

i = 3 arr[3] = 2, count[2] = 4 - 1 = 3