Big-Oh notation: $T(n)$ is $O(f(n))$ if there exist constants $c>0$ and $n_0>0$ s.t. $T(n) \le c \cdot f(n)$ for all $n \ge n_0$. Upper bounds.

Big-Omega notation: $T(n)$ is $\Omega(f(n))$ if there exist constants $c>0$ and $n_0>0$ s.t. $T(n) \ge c \cdot f(n)$ for all $n \ge n_0$. Lower bounds.

Big-Theta notation: $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1>0$, $c_2>0$ and $n_0>0$ s.t. $c_1 f(n) \le T(n) \le c_2 f(n)$ for all $n \ge n_0$. Tight bounds

Little-o: $\exists\ c>0, n_0>0, \forall n \ge n_0, T(n) < c \cdot f(n)$

Little-ω: $\exists\ c>0, n_0>0, \forall n \ge n_0, T(n) > c \cdot f(n)$

Properties: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Same for O, Ω <
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = \Theta(f(n))$
- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Master's Method: $T(n) = aT(\frac{n}{b}) + f(n)$, where $a \ge 1$, $b > 1$ and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $a f(n/b) \le c f(n)$ for $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Sorting Problem

In Place Sort: The amount of extra space required to sort the data is constant with the input size

Stable Sort: a stable sort preserves relative order of records with equal keys.

Insertion Sort In place

```
Insertion-Sort(A)
for j=2 to A.length        c1
    key = A[j];            c2
    i = j-1;               c3
    while i>0 and A[i]>key c4
        A[i+1] = A[i];     c5
```

Loop Invariant for Insertion Sort

Loop invariant: at the start of each iteration of the loop, the subarray $A[1..j-1]$ consists of elements originally in $A[1..j-1]$, but in sorted order.

Initialization: Just before the first

$i = i-1$; c_6

$A[i+1] = key$; c_7

Iteration: $j = 2$, the subarray $A[1 \cdots j-1] = A[1]$ is sorted.

Maintenance: ---

Termination: ---

Running Time Analysis.

running time $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \cdot (n-1)$

best case: $T(n) = \Theta(n)$

worse case: $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_7(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{(n-1)^2}{2}\right) + c_6\left(\frac{(n-1)^2}{2}\right)$
$= \Theta(n^2)$

Bubble Sort in place, $\Theta(n^2)$ — Easier to implement but slower than insertion sort.

Merge Sort $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \Rightarrow T(n) = \Theta(n \lg n)$

not in place, requires extra space $\sim n$

Heap Sort — Heap { Structural property: all levels are full, except possibly the last one, which is filled from left to right; Order property: for any node x: Parent(x) $\geq x$ }

not stable

Useful Properties: { There are at most 2^l nodes at level l of a binary tree; A binary tree with depth d has at most $2^{d+1} - 1$ nodes; A binary tree with n nodes has depth at least $\lfloor \lg n \rfloor$ }

Can be seen as a complete binary tree

Build-Max-Heap(A); $O(n)$

for i = A.length downto 2: } $n-1$ times

do exchange $A[1] \leftrightarrow A[i]$;

Max-Heapify(A, 1, $i-1$); $O(\lg n)$

$\Rightarrow T(n) = O(n) + (n-1)O(\lg n)$
$= O(n) + O(n \lg n)$
$= O(n \lg n)$

Quick Sort in place, average $O(n \lg n)$, worst $O(n^2)$ $T(n) = T(q) + T(n-q) + n$

(Randomized Quick Sort, Lomuto's Partition)

Average-Case Analysis of Quicksort: $\Rightarrow O(n \lg n)$

① PARTITION is called at most n times.

② The amount of work each PARTITION done is $c + Xk$ } ⇒ The total work is $O(n+X)$, where X = total number of comparisons performed in all calls to PARTITION.

Indicator Random Variable $I\{A\} = \begin{cases} 1, & \text{if } A \text{ occurs} \\ 0, & \text{if } A \text{ doesn't occurs} \end{cases}$

$E(I\{A\}) = Pr\{A\}$

Theorem: all comparison sorts are $\Omega(n \lg n)$

Sorting in Linear Time — Counting Sort (stable, not in place) (no comparison between elements!)

$T(n) = \Theta(n+r)$, where r is the maximum value in the input array

Step 1: find the number of times integer $A[i]$ appears in A

Step 2: find the number of elements $\le A[i]$

Step 3: Insert elements into the output array.

Radix Sort: $\Theta(d(n+k))$. (Represents keys as d-digit numbers in some base-k)

```
RandomizedSelect(A, p, r, i)
  if(p==r) return A[p];
  q = RandomizedPartition(A, p, r);
  k = q-p+1;
  if(i==k) return A[k];
  if(i<k) return RandomizedSelection(A, p, q-1, i);
  else return RandomizedSelection(A, q+1, r, i-k);
```

Worst case: $T(n) = T(n-1) + O(n)$
$= O(n^2)$

Best case: Suppose a 9:1 partition
$T(n) = T(9n/10) + O(n)$
$= O(n)$

Load Factor of a Hash Table: $\alpha = n/m$, where n = # of elements stored in the table, m = # of slots in the table = # of linked list. (α encodes the average number of elements stored in a chain.)

Theorem: An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing. ($Pr(h(x) = h(y)) = 1/m$)

Theorem: A successful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

If m is proportional to n, then $\alpha = n/m = O(1)$, Searching takes constant time.

Hash Function: $h(k) = k \bmod m$, choose m to be a prime, not close to a power of 2.

$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m(kA \bmod 1) \rfloor$ slower, but value of m is not critical.

Inorder tree Walk: left, root, right

Preorder tree walk: root, left, right

Postorder tree walk: left, right, root

Def. successor(x) = y s.t. key(y) is the smallest key > key(x)

Def. predecessor(x) = y s.t. key(y) is the biggest key < key(x)

QUICKSORT(A, p, r)

```
if p<r:
    q = PARTITION(A,p,r);
    QUICKSORT(A, p, q);
    QUICKSORT(A, q+1, r);
```

HOARE-PARTITION(A, p, r)

```
pivot = A[p]; i = p-1; j = r+1;
While(TRUE):
    do repeat j = j-1 until A[j] ≤ pivot
    do repeat i = i+1 until A[i] ≥ pivot
    if i<j: exchange A[i] ⇔ A[j];
    else: return j;
```

Max-Heapify(A, i): O(lgn)

```
l = Left(i); r = Right(i);
if (l < A.length && A[l] > A[i]):
    largest = l;
else: largest = i;
if(r < A.length && A[r] > A[largest]):
    largest = r
if (largest != i):
    SWAP(A, i, largest);
    Max-Heapify(A, largest);
```

Randomized-Partition(A, p, r):

```
i = RANDOM(p, r);
exchange A[p] ⇔ A[i];
return PARTITION(A, p, r);
```

LOMUTO's PARTITION(A, p, r):

```
pivot = A[r]; i = p-1;
for j = p to r-1:
    if A[j] ≤ pivot:
        exchange A[++i] ⇔ A[j]
exchange A[i+1] ⇔ A[r];
return i+1;
```

Build-MaxHeap(A): O(n)

```
for int i = A.length-1 to 0:
    do Max-Heapify(A, i);
```

Logarithmic Function: $a^{\log_b x} = x^{\log_b a}$ $\log_b x = \frac{\log_a x}{\log_a b}$

Common Summation: $\sum_{k=1}^{n} k = 1+2+\cdots+n = \frac{n(n+1)}{2}$

$\sum_{k=0}^{n} x^k = 1+x+x^2+\cdots+x^n = \frac{1-x^{n+1}}{1-x}$ $(x \neq 1)$

Special case: $|x|<1$, $\sum_{k=0}^{n} x^k = \frac{1}{1-x}$

$\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\cdots+\frac{1}{n} \approx \ln n$

$\sum_{k=1}^{n} \log k \approx n\log n - n$

$\sum_{k=1}^{n} k^p = 1^p+2^p+\cdots+n^p \approx \frac{1}{p+1} n^{p+1}$

Stirling approximation: $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$