Recurrence Relations:

If $T(n)$ is a polynomial of degree k then: $T(n) = O(n^k)$
If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then:
$T(n) + S(n) = O(f(n) + g(n))$
If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then:
$T(n)*S(n) = O(f(n)*g(n))$

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

$$T(n) = 2T(n/2) + O(1) = O(n)$$

Quicksort 2 pass partitioning:

2	6	3	6	4	6	5	6	9	9	9	9

index

pivotIndex

advance to left (--) until you find something that is not the pivot.

Quicksort 1 pass partitioning:

< pivot	= pivot	In Progress	> pivot

stricter less

strictly greater

4 pointers to track

Optimizing Quicksort: Recurse until very small arrays are left, then use insertion sort on the whole array.

Trees (Insert): Works like binary search

In order traversal: Left -> node -> Right O(n)

Pre-order traversal: node -> Left -> Right

Post-order traversal: Left -> Right -> Node

Level-order: nodes by level (from root)

Successor(key) query:

If key not in tree: Either find predecessor/successor

Else: Successor is the next biggest element

Delete: Delete the correct node, then:

Case 1: No children: Just delete

Case 2: 1 child: Connect it to parent

Case 3: 2 children: Since successor(node) has at most 1 child, swap the node and its successor, then delete the node.

AVL Trees:

Invariant: A node v is height-balanced if:

Binary Search: (Good for monotonic funcs)

Loop Invar.: A[begin] <= key <= A[end]

Correctness Invar: A[begin] <= key <= A[end]

Perform. Invar: (end-begin) <= $n/2^k$ in iteration k.

is open

Sorted array: A[0..n-1]

2	4	4	5	6	7	8	9	11	17	23	28

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if  key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

What are useful preconditions and postconditions?

Preconditions -
Array is sorted
Array is not null or empty
Begin less than end: condition for loop, not fn.

Postconditions -
Return index it found, otherwise return -1.
A[begin] === key

Iteration k: (end – begin) <= $n/2^k$

Performance invariant

$n/2^k = 1$ ➔ $k = \log(n)$ log base does not usually matter

Peak finding (binary search):

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)
  if A[n/2] is a peak **then return** n/2
  else if A[n/2+1] > A[n/2] **then**
    Search for peak in right half.
  else if A[n/2–1] > A[n/2] **then**
    Search for peak in left half.

Running time:

Time to find a peak in an array of size n

Recursion

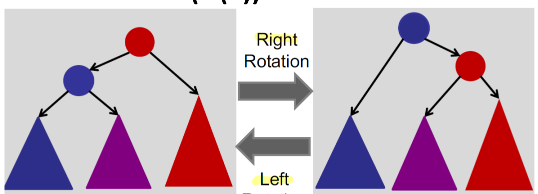
Time for comparing A[n/2] with neighbors

$$T(n) = T(n/2) + \theta(1)$$

Unrolling the recurrence:

$$T(n) = \theta(1) + \theta(1) + \ldots + \theta(1) = O(\log n)$$

Tree Rotations (O(1)):



(Stable) BubbleSort: $O(n^2)$ time

Worst case: Inversely sorted: $O(n^2)$

Best case: Already sorted, O(n)

Loop Invariant: At the end of k iterations, the last k items are sorted and in the correct positions in the final array.

BubbleSort(A, n)
  repeat (until no swaps) **:**
    for j ← 1 **to** n-1
      if A[j] > A[j+1] **then** swap(A[j], A[j+1])

(Not Stable) Selection Sort: $O(n^2)$ time

Best Case: $\Omega(n^2)$ time

Loop invariant: At the end of iteration j: the smallest j items are correctly sorted in the first j positions of the array.

SelectionSort(A, n)
  for j ← 1 **to** n-1**:**
    find minimum element A[j] in A[j..n]
    swap(A[j], A[k])

Find the minimum item, and swap it in place

(Stable) Insertion Sort: Expected $\theta(n^2)$ time

Worst case: Inversely sorted: $O(n^2)$

Best case: Already Sorted $\Omega(n)$ time

Loop Invariant: At the end of iteration j: the first j items in the array are in sorted order.

Insertion-Sort(A, n)
  for j ← 2 **to** n
    key ← A[j]
    i ← j–1
    while (i > 0) **and** (A[i] > *key*)
      A[i+1] ← A[i]
      i ← i–1
    A[i+1] ← *key*

Repeat at most j times.

(Stable) MergeSort: Expected O(nlogn)

MergeSort(A, n)
  if (n=1) **then return;**
  else:
    X ← MergeSort(A[1..n/2], n/2);
    Y ← MergeSort(A[n/2+1, n], n/2);
  return Merge (X,Y, n/2);

Base case

Recursive “conquer” step

Combine solutions

Function	Name
5	Constant
loglog(n)	double log
log(n)	logarithmic
$\log^2(n)$	Polylogarithmic
n	linear
nlog(n)	log-linear
n^3	polynomial
$n^3\log(n)$	
n^4	polynomial
2^n	exponential
2^{2n}	
n!	factorial

Hint: Sterling’s Approximation

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Probability Theory:

Events **A**, **B**:

– Pr(**A**), Pr(**B**)
– **A** and **B** are independent
  (e.g., unrelated random coin flips)

Then:

– Pr(**A** and **B**) = Pr(**A**)Pr(**B**)

Probability of a good pivot:

$p = 8/10$

$(1 - p) = 2/10$

Expected number of times to repeatedly choose a pivot to achieve a good pivot:

$E[\text{\# choices}] = 1/p = 10/8 < 2$

Trees:

Insert, delete, search, predecessor, successor, findMax, findMin: O(n)

Definition: A BST is balanced if h = O(log n)

- A height-balanced tree with n nodes has **at most height h < 2log(n).**

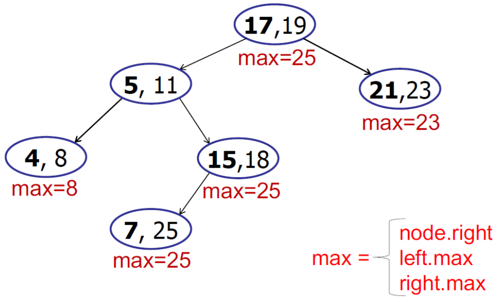
- A height-balanced tree with height h has **at least n > 2h/2 nodes**

|v.left.height – v.right.height| <= 1. A binary search tree is height balanced if every node in the tree is height-balanced.

Interval Trees: Binary tree nodes are compared based on their first digit

Interval search runs in O(logn) time

Augment: maximum endpoint in subtree




```
interval-search(x)
    c = root;
    while (c != null and x is not in c.interval) do
        if (c.left == null) then
            c = c.right;
        else if (x > c.left.max) then
            c = c.right;
        else c = c.left;
    return c.interval;
```

Orthogonal Range Searching (Space O(n)): **O(k + logn),** where k is the no. of point found.

Pre-processing: O(n logn)

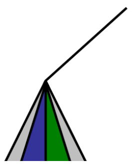
2. Store all points in the leaves of the tree. (Internal nodes store only copies.) nodes are GUIDE POSTS.

3. Each internal node v stores the MAX of any leaf in the left sub-tree.

One Dimensional Range Queries

Algorithm:

- Find "split" node.
- Do left traversal.
- Do right traversal.



Paranoid Select: T(n) <= T(9n/10) + 2n = O(n)

Idea: Pick a random pivot. Partition around it and find its position. Then recurse on the appropriate half and then repeat.

Tree Rotations procedure for AVL trees:

A is left heavy, B is equi-height: Right rotate

A is left heavy, B is left heavy: Right rotate

A is left heavy, B is right heavy: Left rotate B first to get the previous case.

Deleting from AVL trees (O(logn) max rotations):

– Delete key from BST.

– Walk up tree:

- At every step, check for balance.
- If out-of-balance, use rotations to rebalance.
- Continue to root.

Searching for kth largest item: We can now store the weight of each subtree in its root node

2D range tree: Store y-tree in x-node. Space complexity of O(n logn).

Dynamic modifications: Hard, don't do.

Analysis

Query time: $O(\log^2 n + k)$ add. log(n) factor for another dimension

- O(log n) to find split node.
- O(log n) recursing steps
- O(log n) y-tree-searches of cost O(log n) at most O(logn) y-trees, O(log n) time each.
- O(k) enumerating output

Cost of building heap – heapsort O(n)

Height	0	1	2	3	...	$\lfloor \log(n) \rfloor$
Number	$\lceil n/2 \rceil$	$\lceil n/4 \rceil$	$\lceil n/8 \rceil$	$\lceil n/16 \rceil$	...	1

sum < n O(n)

Weighted Union:

Make the taller tree the root: max depth O(logn).

-- weight/rank/size/height of subtree does not change except at root (update root on union).

– weight/rank/size/height only increases when tree size doubles.

Path Compression:

After finding the root: set the parent of each traversed node to the root.

Weighted Union + Path Compression:

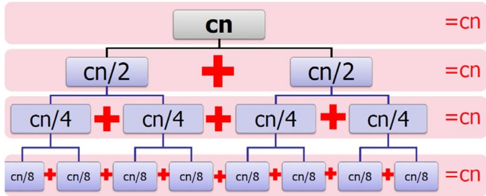
O(n + mα(m, n)) time.

Loop Invariant: After a merge, the merged array is always sorted.

Design a version of MergeSort that **minimizes** the amount of extra space needed: Do not allocate new space during the recursive calls.

Space: O(nlogn) space

T(n) = 2T(n/2) + cn



(Not Stable) Quicksort: O(nlogn) time

Worst case: for normal Quicksort, O(n²)

Worst case: 3-way partition: O(nlogn)

Each partition costs O(n) time.

Loop Invariant: A[high] > pivot at the end of each loop.

Given: n element array $A[1..n]$

1. Divide: Partition the array into two sub-arrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper sub-array.

2. Conquer: Recursively sort the two sub-arrays.
3. Combine: Trivial, do nothing.

3-Way Partitioning

– Option 1: two pass partitioning

1. Regular partition.
2. Pack duplicates.

– Option 2: one pass partitioning Faster

- More complicated.
- Maintain four regions of the array

```
T(n)  →  ParanoidQuickSort(A[1..n], n)
1     →    if (n == 1) then return;
base case of size 1
           else
??  O(n) →   repeat
                 pIndex = random(1,
        O(n) each  p = partition(A[1..n],
             until p > (1/10)n and p < (
recursive instances.
T(k)   →     x = QuickSort(A[1..p–1], p
T(n-k) →     y = QuickSort(A[p+1..n], r
9n/10 > k > n/10
```

Height balanced -> Balanced Tree. But converse is not true.

Tries:

Search String of L: O(L) time,

PRIORITY QUEUES:

Naïve: Sorted Array

Heap: prio[parent] > prio[child]

Every level is full, Last level is as right as possible.

Max height: floor(log n), O(logn)

Insert(k): Put in correct position, bubble up (1< times) if needed.

increaseKey: bubble as needed

decreaseKey: bubble as needed, bubble to the larger child node.

delete(k): swap with last node, then bubble down as necessary.

Heap tree can store in array. Do level-order traversal.

HeapSort (In-place):

Build a heap from unsorted:

Build a binary tree using insert.

Base case: Each leaf is a heap.

Work recursively upwards, keep bubbling.

Run extractMax(). And keep placing max to the end. Time complexity: O(n logn)

Disjoint Set (UnionFind)

Quick-find: Two objects are connected if they have the same component identifier.(Stores CI)

Quick-union: Two objects are connected if they are part of the same tree. (Stores parent)

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	α(m, n)	α(m, n)