Algorithms

Basic DSes

Array vs LL
Arrays: Static data sets, ones where fast index-based access is needed, Slow expansion, insertion, deletion. **LLs:** Dynamic data sets, insertion & deletion more important than random access.

Linked Lists

```
class Node {
    Item item;
    Node next;
    Node prev; // for DLL
}
```

Operations

Search	O(N)
Prepend / Append	O(1)
Delete	O(N)

Runtime Analysis

constant	1
logarithmic	$\log N$
linear	N
linearithmic	$N \log N$
quadratic	N^2
cubic	N^3
exponential	2^N

Sorting Algorithm Info

Algorithm	$O(N)$	$\Omega(N)$	$\Theta(N)$
Selection Sort	$O(N^2)$	$\Omega(N^2)$	$\Theta(N^2)$
Insertion Sort	$O(N^2)$	$\Omega(N)$	$\Theta(N^2)$
Shellsort	$\leq O(N^2)$	$\Omega(N \log N)$	$\Theta(N \log N)$
Mergesort	$O(N \lg N)$	$\Omega(N \lg N)$	$\Theta(N \lg N)$
Quicksort	$O(N^2)$	$\Omega(N \lg N)$	$\Theta(N \lg N)$
Heapsort	$O(N \log N)$	$\Omega(N \log N)$	$\Theta(N \log N)$

Algorithm	Stbl?	In-Place?	Space
Selection Sort	N	Y	O(1)
*Insertion Sort	Y	Y	O(1)
Shellsort	N	Y	O(1)
**Quicksort	N	Y	O(log N)
Mergesort	Y	N	O(N)
Heapsort	N	Y	O(1)

* depends on order of items
** probabilistic guarantee

Tilde Approximation

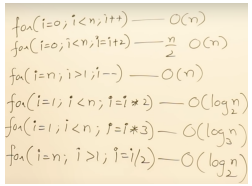
$$\lim_{n\to\infty} \frac{\sim f(n)}{f(n)} = 1$$

TL;DR drop everything but the most significant factor in terms of n.
Order of growth – Just drop the constant from the tilde approximation

Time Complexity

Notation Definitions

- $O(g(n))$ – upper limit
- $\Omega(g(n))$ – lower limit
- $\Theta(g(n))$ – upper and lower limit



Let $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$,

$$0 \leq f(n) \leq c_1 g(n) \quad \Rightarrow f(n) \in O(g(n))$$
$$c_1 g(n) \leq f(n) \quad \Rightarrow f(n) \in \Omega(g(n))$$
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \Rightarrow f(n) \in \Theta(g(n))$$

Limit Definitions

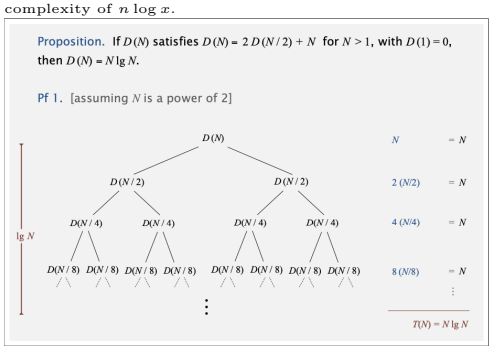
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \quad \Rightarrow f(n) \in O(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \quad \Rightarrow f(n) \in \Omega(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \notin \{0, \infty\} \quad \Rightarrow f(n) \in \Theta(g(n))$$

Memory Complexity

In-place algorithms: $O(N \log N)$
Mergesort:
- Original input array: N.
- Aux array for merging: $O(N)$.
- Local variables: $Const$.
- Function call stack: $O(\log N)$.
- Total: $O(2N \log N)$.

Recurrence Relations

Recurrence Relation – a recursive equation.
Deriving from a Tree – Represent the recurrence equation as a binary tree, where each layer is one layer deeper in the recursion. Eg if the equation is $T(N) = T(N/2) + T(N/2)$, there will be 2 branches on the second layer, each as $T(N/2)$, 4 branches on the third layer, each as $T(N/4)$, and so on. If it's a binary tree, with x layers, it will have a time complexity of $n \log x$.



If you're seeing this, please contribute!

Master Theorem

For an RR with the form $T(n) = aT(n/b) + f(n)$, for constants $a(\geq 1)$ and $b(> q)$ with f asymptotically positive, the following statements are true:
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 $af(n/b) \leq cf(n)$ for some $c < 1$ for all n sufficiently large), then $T(n) = \Theta(f(n))$.

Union-Find

WARNING: !CODE DOES NOT IMPLEMENT COUNT!
Mutually connected nodes are called components.

API

- **Connected(n_id1, n_id2)**: returns true if n_id1 and n_id2 are connected.
- **Union(n_id1, n_id2)**: if not connected, connects two nodes
- **Find(n_id)**: takes node id, returns union n_id is in
- **Count(n_id)**: return the size of n_id's union

Time Complexity

Algo	Q-F	Q-U	WQ-U
Initialization	$O(N)$	$O(N)$	$O(N)$
Find	$O(1)$	$O(N)$	$O(lg(N))$
Connected	$O(1)$	$O(N)$	$O(lg(N))$
Union	$O(N)$	$O(N)$	$O(lg(N))$

Quick-Find

- Associate nodes with unions
- represented with an array
- Indices are node ids
- Values are union ids (we choose one node id for the union id)

Initialization
Initialize an length N array arr with arr[i] = i.
Union(n_id1, n_id2)
Change every occurance of n_id1 to n_id2 in arr.
Find(n_id)
returns arr[n_id].
Connected
returns if arr[n_id1] == arr[n_id2].

Quick-Union

- Associate nodes with other nodes
- nodes point to their parent node, which lead to the root node/union id
- root nodes point to themselves

Initialization
Initialize a length N array arr with arr[i] = i.
Union
Takes indices p and q, finds the root nodes. If they are not equal, make one point to the other.
Find
Takes an index i, finds arr[i] until the root node. Returns the root node id.
Connected
Takes an indices p and q, finds the root nodes. Returns true if they are equal.

```
public class QuickUnion {
  private int[] id;

  public QuickUnion(int N) {
      id = new int[N];
      for(int i = 0; i < N; i++)
          id[i] = i;
  }
  public int find(int i) {
      while (i != id[i]) {
          i = id[i]
      }
      return i;
  }
  public void union(int p, int q) {
      int i = find(p);
      int j = find(q);
      id[i] = j;
  }
  public boolean connected(int p, int q) {
      return find(p) == find(q);
  }
}
```

Weighted Quick-Union

To optimize Quick-Union, we try to create the flattest trees possible. We need another array to count the number of nodes rooted at an index. Union now links the root of the smaller tree to the root of the larger tree. This can be further optimized with path compression, which can be tacked on to find. To do this, we change each node to point at it's grandparent.

```
public class WeightedQuickUnionWithPC {
  private int[] id;
  private int[] sz;

  public void union(int p, int q) {
      int i = find(p);
      int j = find(q);
      id[i] = j;
      if (sz[i] < sz[j])    { id[i] = j; sz[j
          ↪ ] += sz[i] }
      else (sz[i] < sz[j]) { id[j] = i; sz[i
          ↪ ] += sz[j] }
  }
  public int find(int p) {
      while (i != id[i]) {
          id[i] = id[id[i]];
          i = id[i];
      }
      return i;
  }
```

Bad Sorts

Selection Sort

Build up a sorted section of array, by

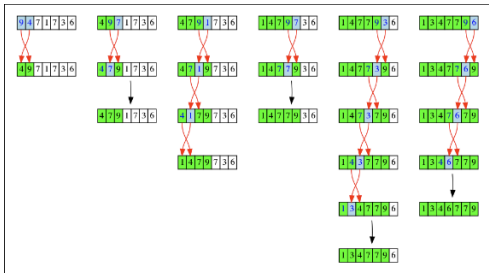
1. Find min element
2. Swap min & first element
3. Examine array, skipping first element.

Pros: Minimal number of write operations (eg tiny RAM?)
Cons: Slow as shit $\Theta(N^2)$

Insertion Sort

- Start from 2 elements, build up sorted subarray, "inserting" a new element each iteration by swapping until it is in the right position.
- # of operations depends on the degree of disorder (how unsorted the array is).

Pros: If the array has a low degree of disorder, it is faster.
Cons: Worst case is still slow as shit.

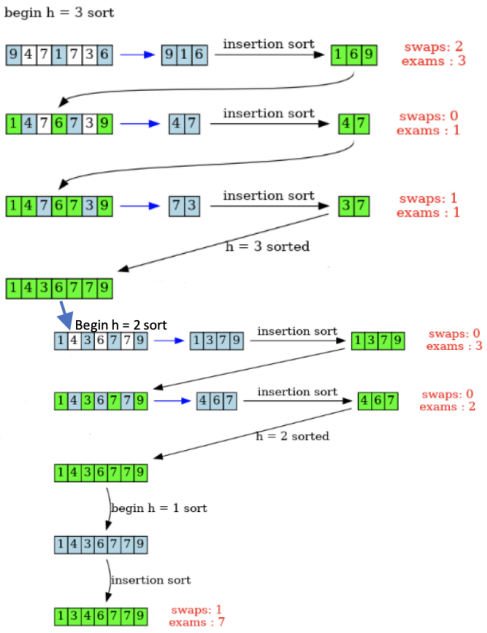


$\Omega(N), O(n^2)$

Shellsort

- Pick every h elements and put them in a subarray.
- Sort the subarrays using insertion sort.
- Repeat until $h = 1$.

Pros: Used when: in embedded systems applications from using small program size and memory efficiency. Reduces large amounts of disorder quickly. **Cons:** Slow



```
public class Shell {
  public static void sort(Comparable[] a) {
    int N = a.length;
    int h = 1;
    while (h < N/3) h = 3 * h + 1;
    while (h >= 1) {
      for (int i = h; i < N; i++) {
        for (int j = i; j >= h && less(a[j],
            ↪ a[j-h]); j -= h)
          exch(a, j, j-h);
      }
      h = h / 3;
    }
  }
}
```

Good Sorts

Mergesort

Divide array in half recursively, until it is down to 1 element. Merge array together like a zipper.
Time Complexity: $O(n \log n)$
Memory Complexity: $O(N)$

Code

```
private static void merge(Comparable[] a,
    ↪ Comparable[] aux, int lo, int mid,
    ↪ int hi) {
  // copy
  for (int k = lo; k <= hi; k++)
    aux[k] = a[k];
  // merge
  int i = lo, j = mid + 1;
  for (int i = lo; k <= hi; k++) {
    if (i > mid)
      a[k] = aux[j++];
    else if
      (j > hi) a[k] = aux[i++];
    else if
      (less(aux[j], aux[i])) a[k] = aux[j
          ↪ ++];
    else
      a[k] = aux(i++);
  }
}

private static void sort(Comparable[] a,
    ↪ Comparable[] aux, int lo, int hi) {
  if (hi <= lo) return;
  int mid = lo + (hi - lo) / 2;
  sort(a, aux, lo, mid);
  sort(a, aux, mid+1, hi);
  merge(a, aux, lo, mid, hi);
}

public static void sort(Comparable[] a) {
  Comparable[] aux = new Comparable[a.length
      ↪ ];
  sort(a, aux, 0, a.length - 1);
}
```

Top-down vs Bottom-up TL;DR

Top-down uses recursion: starts at **top** of tree and proceeds **downwards**. Bottom-up does not use recursion: starts at **bottom** of tree and iterates over pieces moving **upwards**.

Bottom-up

Pass through array, merging as we go to double size of sorted subarrays. Keep performing the passes and merging subarrays, until you do a merge that encompasses the whole array.

```
public class MergeBU {
  private static Comparable[] aux;
  // see above for merge() code
  public static void sort(Comparable[] a) {
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz + sz)
        ↪ // sz: subarray size
      for (int lo = 0; lo < N - sz; lo += sz
          ↪ + sz) // lo: subarray index
        merge(a, lo, lo+sz-1, Math.min(lo+sz
            ↪ +sz-1, N-1));
  }
}
```

Min Top-down Comparisons

Proposition: Top-down mergesort uses between $1/2NlogN$ *and* $NlogN$ *comparisons.*
Let the total number of comparisons be C(N): $C(\lceil N/2 \rceil)$ = left recursion, $C(\lfloor N/2 \rfloor)$ = right recursion, cN = comparisons at this level, $C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + cN$. The smallest number of comparisons made by *MERGE* is $N/2$. If one sub-array contains all the smallest elements, we only walk that array before appending the other. If you sort a sorted array, this would be the case at every level. Solving the above recursion with $c = 1/2$ yields $C(N) = 1/2NlogN$
Max top-down Comparisons:
Similarly, the maximum number of comparisons is made when both sub-arrays must be fully examined. If the input array is of size N , then at most N comparisons are made. If the above happens at every level of the recursion, the total number of comparisons at each level is at most N. By solving the same recurrance equation with $c = 1$, we get $C(N) = NlogN$
Max top-down Accesses:
Proposition. Top-down mergesort uses at most $6NlogN$ *array accesses to sort an array of length* N. Each merge uses at most 6N array accesses $2N$ to copy the sub arrays initially $2N$ to put the values back (in order) At most N comparisons, each accessing two array elements $(2N)$Hence the total number of array accesses after solving the recurrence is most $6NlogN$.
Min/Max Bottom Up Comparisons/Accesses:
Proposition. Bottom-up mergesort uses between $1/2NlogN$ *and* $NlogN$ *compares and at most* $6NlogN$ *array accesses to sort an array of length* N. The number of passes through the array is precisely $logN$. For each pass, by the same argument made as in the case of the top-down mergesort approach, the number of array accesses is exactly $6N$ and the number of compares is at most N and no less than $N/2$. Therefore, bottom-up mergesort uses between $1/2NlogN$ and $NlogN$ compares and at most $6NlogN$ array accesses to sort an array of length N.

Quicksort

1. Shuffle array to reduce impact of order on sorting speed
2. Pick first element of array as pivot
3. Create two sub arrays from remaining elements, one selecting those smaller, one selecting those larger. Put them on either side of the pivot
4. Recurse for each side of the pivot until everything is sorted.

```
partition (arr, lo, hi):
  pivot = hi
  i = lo
  for j from lo+1 to hi:
    if arr[j]<pivot:
      swap(arr[i], arr[j])
      i++

quicksort (arr, lo, hi) {
  if lo<hi:
    pivot = partition(arr, lo, hi)
    quicksort(arr, lo, pivot-1)
    quicksort(arr, pivot+1, hi)
```

- Fastest for disordered arrays, slowest for already sorted arrays
- Randomize array or select a random piviot to prevent worst case. (Best choise of a piviot is the median)
- **Best case**: The partitions are always of equal size : $\Omega(NlogN)$. Recurrence relation is $T(n) = 2T(n/2) + cn$.
- **Worst case**: One partition is always of size 0 (if the array is already sorted and we are picking pivots from the ends) : $O(N^2)$. Recurrence relation is $T(n) = T(n-1) + T(0) + cn$.
- **Average case**: 1.39 $\log N \in \Theta(NlogN)$
- Uses less memory than merge sort. Space complexity $O(n)$

Priority Queues

Supports insertion and removing/popping the priority (largest or smallest) item.
Implementations:

Sorted Array - O(n) insert, O(1) pop

Unsorted Array - O(1) insert, O(n) pop

Binary Heap - O(log n) insert and sort

Binary Heaps & Heapsort

A max binary heap is a complete binary tree where the keys are in the nodes and each parent's key $\geq$ each child's key. This requirement is called the max heap property.
Binary Heaps:

- Can be represented as array or tree/nodes.
- Insertion: We insert at the end of the array, then "swim up" the value.
- Swimming up - exchange a given node with it's parent until the binary max property is fulfilled.
- Popping - swap the first node (the max) with the last node, remove it, then "sink" the first node.
- Sinking - exchange a given node with the max of it's children until the binary max property is fulfilled.

Heapsort relies on the binary heap data structure to sort data. Once a max heap has been constructed, you can perform a single for loop and call $RemoveMax$ to build a sorted array (thus linearizing the heap).

Code

```
public class MaxPQ<Key extends Comparable<
    ↪ Key>> {
  private Key[] pq;
  private int n;

  public MaxPQ(int capacity) {
    // fixed capacity for simplicity
    pq = (Key[]) new Comparable[capacity+1];
  }
  public boolean isEmpty() {
    return n == 0;
  }
  public void insert(Key x) {
    pq[++n] = x;
    swim(n);
  }
  public Key delMax() {
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] == null;
    return max;
  }

  private void swim(int k) {
    // parent of node at k is at k/2
    while (k > 1 && less(k/2, k)) {
      exch(k, k/2);
      k = k/2;
    }
  }
  private void sink(int k) {
    while (2*k <= n) {
      int j = 2*k;
      // children of node at k are 2*k and
          ↪ 2*k+1
      if (j < n && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
    }
  }
}
```

Binary Trees

Binary Tree: A tree where each internal node has at most two children. **Full Binary Tree:** A binary tree where each internal node has exactly two children. **Complete Binary Tree:** A complete binary tree is which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. **Internal Node:** Any node that has children
$L = N - 1$, where L is the number of the internal node in the tree, and N is the number of leaves.
$N = 2^h$, where h is the height of the tree and N is the number of nodes in the tree. To go the other way around, $\log_2(N) = h$.
Maximum height: $n - 1$ **Minimum height:** floor($\lg n$)

Pseudocode

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

Reminders:

- Attributes can be accessed via `x.prev`.
- Class functions **do not exist in pseudocode**. Instead of calling `x.sort()`, call `sort(x)`.
- Use `NIL` instead of `null`.
- State assumptions like passing by value or passing by reference in the pseudocode.
- Multiple values can be passed in a `return` statement.
- Use `error` for throwing an error, but what happens is not specified.