1 Algorithmic Analysis

Asymptotic Notations

Big-O Notation

The upper bound for runtime/space.

$$T(n)\in O(f(n)) \iff$$
$$\exists c, n_0(c>0\wedge n_0>0\wedge \forall n>n_0(T(n)\le cf(n)))$$

Big-Ω notation

The lower bound for runtime/space.

$$T(n)\in \Omega(f(n)) \iff$$
$$\exists c, n_0(c>0\wedge n_0>0\wedge \forall n>n_0(T(n)\ge cf(n)))$$

Note: $f(n)\in O(g(n)) \iff g(n)\in\Omega(f(n))$

Big-Θ notation

The bound for runtime/space.

$$T(n)\in \Theta(f(n)) \iff$$
$$T(n)\in\Omega(f(n))\wedge T(n)\in O(f(n))$$

Comparison

$O(1) < O(\log(\log(n))) < O(\log(n)) < O(\log^k n) < O(n) < O(n^k) < O(2^n) < O(2^{2n}) < O(n!)$

Akra-Bazzi Theorem

$T(n) = g(n) + \sum_{i=1}^{k} a_i T(b_i n + h_i(n))$, with

- Sufficient base cases
- $a_i>0, 0<b_i<1$
- $|g'(x)| = O(x^c), |h_i(x)| = O(\frac{x}{\log^2 x})$

Solve for p : $\sum_{i=1}^{k} a_i b_i^p = 1$

$$T(n)=\Theta(n^p(1+\int_1^n \frac{g(u)}{u^{p+1}}du))$$

Master Theorem

$T(n) = aT(n-b)+O(n^k)$, where $a,b>0, k\ge 0$

$$T(n)=\begin{cases}O(n^k) & a<1\\ O(n^{k+1}) & a=1\\ O(a^{n/b}n^k) & a>1\end{cases}$$

$T(n)=aT(n/b)+\Theta(n^k\log^p n)$, where $a\ge 1, b>1, k\ge 0, p\in\mathbb{R}$. For $c=\log_b a$,

$$T(n)=\begin{cases}\Theta(n^k\log^p n) & c<k, p\ge 0\\ \Theta(n^k) & c<k, p<0\\ \Theta(n^c\log^{p+1} n) & c=k, p>-1\\ \Theta(n^c\log\log n) & c=k, p=-1\\ \Theta(n^c) & c=k, p<-1\\ \Theta(n^c) & c>k\end{cases}$$

Linear Recurrence

$T(n)+c_1T(n-1)+\cdots+c_kT(n-k)+f(n)$

For $f(n)=0$:

$$x_n = c_1x^{n-1}+\cdots+c_k^{n-k}$$

For t distinct roots of multiplicity m_i,

$$T(n) = (\alpha_{1,0}+\alpha_{1,1}n+\cdots+\alpha_{1,m_1}n^{m_1-1})r_1^n +$$
$$\vdots \qquad \vdots$$
$$(\alpha_{t,0}+\alpha_{t,1}n+\cdots+\alpha_{t,m_t}n^{m_t-1})r_t^n$$

For $f(n)\neq 0$, find solution a_h for $f(n)=0$:

$$T(n) = a_p + a_h$$

Guess a_p similar to $f(n)$: Polynomial of degree n, Anc^n, Ac^n

2 Searching

Binary Search

Precondition: Array is sorted, and of size n

Postcondition: If element is in arr, `A[begin]=key`

Invariants: Key is within `A[begin..end]`

`(end-begin)` $\le \frac{n}{2^k}$ after the k-th iteration

```
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
```

Runtime: $O(\log n)$

Peakfinding

Precondition: Array of size n

Postcondition: $A[i-1]\le A[i]\wedge A[i+1]\le A[i]$

Invariants: $\exists$ a peak in `A[begin..end]`

Every peak in `A[begin..end]` is a peak in `A[0..n-1]`

Recurse in the right $\Rightarrow$ Peak in the right half is a peak in the array.

```
int findPeak(A, n)
   if A[n/2] is a peak then return n/2
   else if A[n/2 + 1] > A[n/2] then
      return findPeak(A[n/2 + 1..n], n/2)
   else if A[n/2 - 1] > A[n/2] then
      return findPeak(A[1..n/2 - 1], n/2)
```

Runtime: $O(\log n)$ Steep peakfinding: $O(n)$

Quickselect

Precondition: Array of size n Postcondition: Return the k-th smallest element. Invariants: The k-th eleemnt is on the side that we recurse on.

```
T select(arr, n, k)
   if (n == 1) then return arr[1]
   else
      Choose random pivot index pIndex
      p = partition(arr[1..n], n, pIndex)
      if (k == p) then return arr[p]
      else if (k < p) then
         return select(arr[1..p-1], p-1, k)
      else if (k > p) then
         return select(arr[p+1..n], n-p, k-p)
```

Runtime: $O(n)$

3 Sorting

Problem: Array `A[1..n]` of elements $\Rightarrow$ Permutation `B[1..n]` st. $B[1]\le B[2]\le\cdots\le B[n]$

Algo	Worst case	Best case	Space
Bubble	$O(n^2)$	$O(n)$	$O(1)$
Selection	$O(n^2)$	$\Omega(n^2)$	$O(1)$
Insertion	$O(n^2)$	$O(n)$	$O(1)$
Merge	$O(n\log n)$	$O(n\log n)$	$O(n)$
Quick	$O(n^2)$	$O(n\log n)$	$O(\log n)$
Heap	$O(n\log n)$	$O(n\log n)$	$O(1)$

Invariants

Bubble: Largest j items sorted in the final j positions after the j-th iteration

```
void bubbleSort(arr, n)
   repeat (until no swaps):
      for j = 1 to n-1
         if arr[j] > arr[j + 1] then
            swap(arr[j], arr[j + 1])
```

Bubble sort is stable.

Selection: Smallest j items sorted in the first j positions after the j-th iteration

```
void selectionSort(arr, n)
   for j = 1 to n-1:
      find min element arr[k] in arr[j..n]
      swap(arr[j], arr[k])
```

Selection sort is non-stable.

Insertion: First j positions are sorted after the j-th iteration

```
void insertionSort(arr, n)
   for j = 2 to n:
      key = arr[j]
      insert key into sorted array A[1..j-1]
```

Insertion sort is stable.

Merge: $tmp[j]$ is the minimum of all remaining elements after the j-th iteration

```
void mergeSort(arr, n)
   if (n == 1) then return;
   else
      left = mergeSort(arr[1..n/2], n/2);
      right = mergeSort(arr[n/2+1..n], n/2);
      merge(left, right, n/2);
```

Mergesort is stable.

Quick: B is partitioned around pivot `A[high]>pivot` at the end of each iter

```
void quickSort(arr, n)
   if (n == 1) then return;
   else
      p = partition(arr[1..n], n)
      quickSort(arr[1..p-1], p-1)
      quickSort(arr[p+1..n], n-p)
```

Quicksort [$O(n\log n)$ version] is non-stable.

Duplicates Paritioning: 3 way partitioning: Packing duplicates together!

One pass:

Two passes:

Heap: Largest j items are sorted in the final j positions after the j-th iteration.

```
void heapSort(arr, n)
   for n - 1:
      1. Swap root with last element
      2. Bubble down the root node
```

Heapsort is non-stable.

4 Trees ADT

```
insert(Key k, Value v)  delete(Key k)
search(Key k)           contains(Key k)
successor(Key k)        size()
predecessor(Key k)
```

Runtime: Most methods are $O(height)$

```
K search(K curr, V value)
   if (value == curr) then
      return curr
   else if (value == null) then
      return null;
   else if (value > curr.key) then
      return search(curr.right, value)
   // Symmetrical on the left
void insert(K curr, V value)
   if (value > curr.key) then
      if (curr.right == null) then
         curr.right = value
      else then
         insert(curr.right, value)
   // Symmetrical on the left
K successor(K curr, V value)
   result = search(curr, value)
   if (result > value) then return result;
   else if (result <= value) then
// Return the right child.
// If no right child and result is the
// left child of parent, return parent.
// Otherwise recurse until it is a left
// child.
void delete(K curr, V value)
   result = search(curr, value)
   if (result.hasNoChildren()) then
      result.delete()
   else if (result.hasOneChild()) then
      result.parent.setChild(result.child)
   else if (result.hasTwoChildren()) then
      s = successor(result) // MAX 1 child
      swap(s, result)
      delete(result, result)
```

Balanced Property

BST is balanced if $h\in O(\log n)$

1. Define a "good property" of a tree
2. Prove that the "good property" $\Rightarrow$ balanced tree

   Height balanced BST with height h has at least $n>2^{h/2}$ nodes $\equiv n$ nodes has height $h<2\log n$
3. After every operation, reestablish property as invariant

AVL Tree: $|$`v.left.height` $-$ `v.right.height`$|\le 1$

BST is height balanced $\iff$ $\forall$v $\in$BST, v is height balanced

Proof: $n_h\ge 1+n_{h-1}+n_{h-2}\Rightarrow n_h\ge 2^{h/2}$ Maintain balance: If v is left-heavy:

1. `v.left` is balanced: `rightRotate(v)`
2. `v.left` is left-heavy: `rightRotate(v)`
3. `v.left` is right-heavy: `leftRotate(v.left)` + `right-rotate(v)`

Note: Case 1 does not decrease root height. Only results from deletion, so no need to further decrease root height. Requires up to $O(\log n)$ rotations

(a,b)-Tree: $2\le a\le(b+1)/2$

Node type	#Keys Min	#Keys Max	#Children Min	#Children Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

A non-leaf node must have one more child than its number of keys. The keys specify the range each child falls under.

All leaf nodes must be at the same depth.
`merge, share, split`
Order Statistics Tree: AVL Tree, storing subtree weight

5 Hashing

```
insert(Key k, Value v) contains(Key k)
search(Key k)          size()
delete(Key k)
```

Direct Access Table

Using an array indexed by keys to access values. Suppose English words, with max. 28 letters, and each letter represented in 5 bits. Any word can then be represented in 140 bits. We require a 2^{140} sized direct-access array.

Hash Functions $h : U \mapsto \{1..m\}$

Considers the smaller set of actual keys K, and map the $|K| = n$ keys to $m \approx n$ buckets. By PHP, $\exists h(k_1) = h(k_2) \land k_1 \neq k_2$.
Simple Uniform Hashing Assumption: Every key is equally likely to map to every bucket, and keys are mapped independently.
Let X_i = numBalls in the i-th bucket. $X_i \sim Bin(n, \frac{1}{m})$
For fixed load α, $E(\text{size of max bucket}) = \Theta(\frac{\log n}{\log \log n})$
`int Object::hashCode()`: Must redefine `equals`. If two objects are equal, they have to return the same hash code.
Used: `o.hashCode() ^ (table.size() - 1)`

Dealing with Hash Collisions

- New Hash Function
  "Better"? Need to copy table, and eventually another collision
- Chaining
- Open Addressing

Chaining: Each bucket contains a linked list
Space: $O(m \text{ buckets} + n \text{ items})$
Runtime: Insert $O(1 + \texttt{cost}(h))$ Delete $O(n + \texttt{cost}(h))$
Open Addressing: Probing a sequence of buckets until empty found
Space: $O(m)$ Runtime: Worst case $O(n)$ Practice $O(1)$ for constant load factor and good hash functions

Table Resizing

1. Choose new table size m'
2. Choose new hash function h' such that
   $h : U \mapsto \{1..m\}, h' : U \mapsto \{1..m'\}$
3. $\forall x \in$ currTable, newTable$[h'(x)] = x$

Resizing costs $\Theta(m' + m + n)$ for new table size m', current table size m, n elements.
Proof: Consider sized $2n$ table of n elements, and a sequence of n operations. n inserts or deletions are both $\Theta(n)$.

Cuckoo Hashing

```
insert(Key k, Table T, Hash h)
   slot = h(k)
   displaced = T[slot]
   T[slot] = k
   if (displaced != null) then
      insert(displaced,
             T == A ? B : A,
             T == A ? g : f)
```

6 Binary Heap

```
insert(int pri, Key k) size(Key k)
extractMax()           peekMaxId()
contains(Key k)        peekMaxPriority()
increaseKey(int pri, Key k)
```

Consists: indices $\mapsto$ priorities, id $\mapsto$ indices, indices $\mapsto$ id
Invariants: Priority of each node $<$ Priority of parent
Heap is a complete binary tree
Runtime: Mostly $O(\log n)$
`insert`: Set next = ele, while not root and >parent: bubbleUp
`increase/decreaseKey`: Change priority and bubble upwards/downwards
`extractMax`: Swap root with last element, remove new last element, bubble downwards

Building a Heap

For parent `x` of two subheaps `L`, `R`. Prove by strong induction that if we `bubbleDown(x)`, the resulting heap satisfies invariants.

```
void heapify() {
  for (int idx = size/2; idx >= 1; idx--) {
    bubbleDown(idx);
  }
}
```

Heapify is $O(n)$:

$$\begin{aligned}
1\cdot\frac{n}{4}+2\cdot\frac{n}{8}+\cdots+(h\cdot 1) &= \sum_{k=1}^{h}\frac{kn}{2^{k+1}} = \frac{n}{4}\sum_{k=1}^{h}\frac{k}{2^{k-1}}\\
&< \frac{n}{4}\sum_{k=1}^{\infty}\frac{k}{2^{k-1}}\\
&= \frac{n}{4}\sum_{k=1}^{\infty}kx^{k-1}, x = 1/2\\
&= \frac{n}{4}\frac{d}{dx}\left[\sum_{k=0}^{\infty}x^k\right]\\
&= \frac{n}{4}\frac{d}{dx}\left[\frac{1}{1-x}\right] = n
\end{aligned}$$

7 Graphs

Graph $G = (V, E)$, where
$|V| > 0, E \subset \{(v, w) : (v \in V), (w \in V)\}$
Solving graph problems: Reduce problem to a graph, where each node is a state. Allows for stateless algorithms to run. If possible, use DAG (much more efficient) [Phantom nodes, 0-weight edges]

Graph Representations

If space is limited, $|E| << |V|^2$, use adjacency list.
If require matrix operations, use adjacency matrix.
Adjacency List $|V|$-sized array, each element containing a linked list of neighbours
Space: $O(|V| + |E|)$, Runtime:
Iterating neighbours of specified v: $O(\deg(v))$
Determine if x, y are neighbours: $O(\min\{\deg(x), \deg(y)\})$
Adjacency Matrix $|V|^2$-sized symmetric matrix, $A[v][w] = 1 \iff (v, w) \in E$. A^2 to find out 2-hop neighbours.
Space: $O(|V|^2)$, Runtime:
Iterating neighbours of specified v: $O(|V|)$
Determine if x, y are neighbours: $O(1)$
Edge List $|E|$-sized array of all edges in the graph
Space: $O(|E|)$, Runtime:
Iterating neighbours of specified v: $O(|E|)$
Determine if x, y are neighbours: $O(|E|)$

Searching

Input: Source vertex `S`
Output: Visit destination vertex `D`. OR all nodes in the graph.

```
boolean bfs(source, dest)
   source.isVisited = true
   que.offer(source)
   while (!que.isEmpty())
      curr = que.poll()
      for (nbr : curr.nbrList())
         if (!nbr.isVisited) then
            nbr.isVisited = true
            que.offer(nbr)
            // Set nbr.dist/nbr.parent
   return dest.isVisited
```

DFS is *exactly the same* with a `stack`
BFS & DFS is $O(|V| + |E|)$

Topological Sorting

Ordering of nodes: $(u, v) \in E \Rightarrow u$ before v in the toposort.
Post-order DFS: Runtime: $O(|V| + |E|)$

```
void topo(Node[] nodes)
   for (node : nodes)
     if (!node.visited) then
        node.visited = true
        toposort(node)
        order.prepend(node)
void toposort(Node node)
   for (nbr : node.nbrList())
      if (!nbr.visited) then
         nbr.visited = true
         toposort(node)
         order.prepend(nbr)
```

Tarjan's Algorithm

To find cycles, points which removed disconnects graph (articulation points), strongly connected components ($\forall (u, v) \in V^2, \exists$ path(u, v))

```
stk = new Stack<>()
void tarjan(Node curr, int time)
   stk.push(curr)
   curr.time = time
   lowTime = curr.time
   for (Node nbr : curr.nbrList())
      if (nbr.lowTime is set) then continue
      if (nbr.time is set) then
         lowTime = min(nbr.time, lowTime)
      else if (nbr.time is not set) then
         tarjan(nbr, time + 1)
         lowTime = min(nbr.lowTime, lowTime)
   curr.lowTime = lowTime
   if (lowTime == time) then
      while (stack.peek() != curr)
         stack.pop() // SCC rooted at curr
```

S is an articulation point if

- S is the source and in DFS tree, S has outdegree 2
- S is not the source, and has a neighbour v st. $v.lowTime \geq S.time \lor S.lowTime \geq v.time$

$\exists v \in V(v.lowTime < v.time) \Rightarrow \exists$ a cycle

Dijkstra

Triangle Inequality for (shortest) distance between Nodes:
$$\delta(S, C) \leq \delta(S, A) + \delta(A, C)$$

```
void dijkstra()
   for (node : nodes)
      pq.insert(node, inf)
   pq.decreaseKey(source, 0)
   while (!pq.isEmpty())
      curr, distance = pq.extractMin()
      dist[curr] = distance
      for (nbr : curr.nbrList())
         if (pq.contains(nbr))
            relax(nbr,
                  distance + E[curr, nbr])
```

Runtime: $O(|V|\log|V| + |E|\log|V|)$
Problem Formulation: To ensure that estimates monotonically decrease with relaxation with more nodes, and do not decrease further after it is popped from the priority queue. $\Rightarrow$ No negative edges

Bellman Ford

```
void bf()
   // Set up int[] dist
   for i = 1..|V|-1:
      for edge (u, v) in E:
         relax(dist, u, v)
   for edge (u, v) in E: // +1
      relax(dist, u, v)
      // If no change, no negative cycles
```

Runtime: $O(|V||E|)$ Special Case–DAG:

```
// Set up int[] dist
// Get toposorted nodes topoList
for u in topoList:
   for neighbour v of u:
      relax(dist, u, v)
```

8 MST

Properties of MST

Suppose a connected graph. Otherwise, no spanning tree exists.

- MST is a acyclic graph. Tree.
- If an edge is removed from a MST, the two components are MSTs.
- For every *cycle*, the maximum weighted edge is not in the MST.
- For every *cut* of the nodes, the minimum weighted edge across the cut is in the MST.

Edge colouring for proof of correctness

Red Rule: If C is a cycle with no red arcs, then the max-weighted edge in C is coloured red. Blue Rule: If D is a cut with no blue arcs, then the min-weighted edge in D is coloured blue.

UFDS

Weighted Union: To make the smaller tree the subtree of the larger tree $\Rightarrow O(\log(n))$ height. Runtime: $O(\log n)$
Path Compression: Shrinks the tree whenever `find` is called.
Runtime: m operations on n objects–$O(n + m\alpha(m, n))$

```
findRoot(int p)
   root = p;
   while (parent[root] != root)
      root = parent[root]
   while (parent[p] != p)
      temp = parent[p];
      parent[p] = root;
      p = temp;
   return root;
```

Kruskal's Algorithm

Runtime: $O(E\alpha(E)) + O(E\log E) = O(E\log V)$

- Initialise UFDS for n nodes
- Sort edges by weights
- For each edge $e = (u, v)$:
  - If u, v are in the same component, skip
  - Otherwise, add e, and union u, v

Prim's Algorithm

Runtime: $O((E+V)\log V)$. $O(E+V\log V)$ for fibo-heap.
Basic idea:

- S: set of nodes connected by blue edges
- Initially, $S=\{A\}$
- Repeat:
  - Identify cut: $\{S, V\backslash S\}$
  - Find minimum weighted edge on cut
  - Add new node to S

Variants

Undirected graphs / Equal weighted edges: DFS / BFS!
DAGs: $\forall v \in V$ add minimum weighted incoming edge–$O(E)$
Edges weighted $\{1..10\}$ $\Rightarrow$ use an array of size 10 to act as PQ
Reweighting: Only relative edge weights matter. Addition, Multiplication allowed. To find MaxST, multiply by -1 and run MST.

9 Dynamic Programming

Everything is a table!

Optimal Sub-Structure

Optimal solution can be contructed from optimal solutions to smaller subproblems
Doesn't always exist

DP Recipe

- Identify optimal substructure
- Define subproblems
- Solve problem using subproblems
- Write pseudocode

DP Analysis

- Count subproblems
- Figure out total time to solve all subproblems