Divide & conquer

1. Divide, conquer, & combine
2. $T(n) =$ divide time $+ \; T(n_1) + T(n_2) + ... + T(n_k) \; +$ combine cost
3. Merge sort, peak finding, binary search

Asymptotic complexity

1. $o, O, \Theta, \Omega, \omega$
2. **Sterling's approx.**: $n! \approx \sqrt{2\pi n}(n/e)^n$ so $n! = o(n^n)$ and $n! = \omega(2^n)$ and $\log(n!) = \Theta(n \log n)$
3. Fibonacci numbers: $\varphi = (1 \pm \sqrt{5})/2$ and $F_i = (\varphi^i - \hat{\varphi}^i)/\sqrt{5}$
4. Recurrences: tree method, master theorem, guess
5. **Guess and prove**: $T(n) = 2T(n/2) + n$
    a. Guess $T(n) = O(n \lg n)$ and prove $T(n) \le cn \lg n$
    b. Assume that it holds for $m = n/2$
    c. $T(n/2) \le c(n/2)\lg(n/2)$ and substituting:
    d. $T(n) \le 2(c(n/2)\lg(n/2)) + n$
    e. $T(n) \le cn \lg n - cn \lg 2 + n$
    f. $T(n) \le cn \lg n$
6. **Master theorem**: $T(n) = aT(n/b) + f(n)$
    a. **Case 1**: $f(n) = O(n^{\log_b a - \varepsilon})$ then $T(n) = \Theta(n^{\log_b a})$ (leaf dominate)
    b. **Case 2**: $f(n) = \Theta(n^{\log_b a} \log^k n)$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ where $k \ge 0$
    c. **Case 3**: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ then $T(n) = \Theta(f(n))$ (root dominate)
    d. Cases where $f(n)$ doesn't fit into these include $f(n) = n/\log n$
7. Change of variables: $T(n) = 2T(\sqrt{n}) + \Theta(\lg n)$
    a. $n = 2^m$, $S(m) = T(2^m) = T(n)$
    b. $S(m) = 2S(m/2) + \Theta(m)$ so $S(m) = \Theta(m \lg m)$
    c. $T(n) = \Theta(\lg n \lg \lg n)$
8. Amortized analysis: get total running time for n operations and divide by n

Python cost models

	List (array)	Dict (hash table)
Append	$O(1)$	$O(1)$
Search	$O(n)$	$O(1)$
Remove	$O(n)$	$O(1)$
Order preserved	Yes	No

Sorting

	Sort	**Time**	**In place**	**Stable**
Comparison sort (decision tree with $n!$ leaves)	Insertion	$\Theta(n+d) / O(n^2)$	Yes	Yes
	Merge	$\Theta(n \log n)$	No	Yes
	Heap	$\Theta(n \log n)$	Yes	No
	AVL	$\Theta(n \log n)$	No	Yes
Non-comparison sort	Counting	$\Theta(n+k)$ $\Theta(n)$ if $k = O(n)$	No	Yes
	Radix	$\Theta((n+b)\log_b k)$ $\Theta(n)$ if $b = n$ and $k = n^{O(1)}$	No	Yes

More asymptotic complexity

Heap

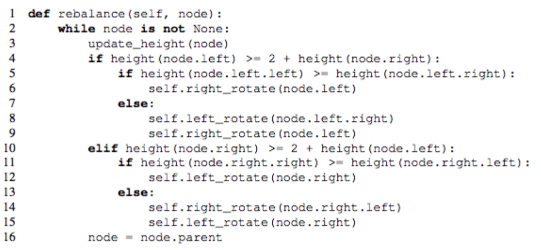
1. Implementation of priority queue; complete binary tree; every parent is max/min of left & right children
2. Implement in array: parent $= i$, left $= 2i+1$, right $= 2i+2$
3. **Find min** $O(1)$, **extract-min** $O(\log n)$, **insert** $O(\log n)$, **decrease key** $O(\log n)$, **min-heapify** $O(\log n)$, **build-min-heap** $O(n)$, **delete** (by value) $O(n)$, **delete** (by ref) $O(\log n)$
4. Extract-min: swap min with last leaf, min-heapify (float down)
5. Build-min-heap: from n/2 to 1, min-heapify (in place)
6. Insertion: add new element as last leaf, bubble up until appropriate
7. Deletion: swap with last leaf & remove, min-heapify
8. **Heapsort**(A): build-max-heap(A); for i=n-1 to 1 {swap A[0] with A[i]; decrease heap size by 1; max-heapify(A,0)}

Binary search tree (BST)

1. Binary tree w/ BST property: every node in left subtree $\le$ node $\le$ every node in right subtree
2. Implementation: node has key, parent pointer, left subtree pointer, right subtree pointer
3. **Search** $O(h)$, **find-min/max** $O(h)$, **pred./successor** $O(h)$, **insert** $O(h)$, **delete** $O(h)$, **inorder traversal** $O(n)$
4. Inorder traversal: inorder(x.left), open(x), inorder(x.right)
5. Search: if x=none or k=x.key {return x}; if k < x.key {return search(x.left,k)}; else {return search(x.right,k)}
6. Successor: if x.right != none {return min(x.right)}; y=x.parent; while y != none and x == y.right {x=y; y=y.parent}; return y
7. Deletion: if no children, remove node; if one child, elevate that child to take node's position; if two children, replace node with its successor
    a. If successor is node's right child, replace node with successor
    b. If successor is within node's right subtree, replace successor with its own right child, then replace node with successor
8. Delete: if z.left == none {transplant(T,z,z.right)}; elseif z.right == none {transplant(T,z,z.left)}; else {y = minimum(z.right); if y.parent != z {transplant(T,y,y.right); y.right=z.right; y.right.parent = y} transplant(T,z,y); y.left=z.left; y.left.parent=y;}

AVL tree

1. Binary tree w/ BST & AVL properties: heights of all left/right subtrees differ by at most 1
2. Implementation: node has key, parent pointer, left subtree pointer, right subtree pointer, and height
3. **Search** $O(\log n)$, **find-min/max** $O(\log n)$, **pred./successor** $O(\log n)$, **insert** $O(\log n)$, **delete** $O(\log n)$, **inorder traversal** $O(n)$, **rotation** $O(1)$
4. To find k^{th} smallest element, augment with subtree size



```
def rebalance(self, node):
    while node is not None:
        update_height(node)
        if height(node.left) >= 2 + height(node.right):
            if height(node.left.left) >= height(node.left.right):
                self.right_rotate(node.left)
            else:
                self.left_rotate(node.left.right)
                self.right_rotate(node.left)
        elif height(node.right) >= 2 + height(node.left):
            if height(node.right.right) >= height(node.right.left):
                self.left_rotate(node.right)
            else:
                self.right_rotate(node.right.left)
                self.left_rotate(node.right)
        node = node.parent
```

5.

$$\log_{\log(5)}(\log(n)^{100}) = \Theta(\log(\log(n))$$
$$\log_{\log(5)}(\log(n^{100})) = \Theta(\log(\log(n))$$

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

$$\begin{aligned}\log\binom{n}{\frac{n}{2}} &= \log\frac{n!}{(\frac{n}{2})!\,(\frac{n}{2})!}\\ &= \log\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left(\sqrt{2\pi n/2}\left(\frac{n}{2e}\right)^{n/2}\right)^2}\\ &= \log\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\pi n\left(\frac{n}{2e}\right)^n}\\ &= \log\left(2^n\sqrt{\frac{2}{\pi n}}\right)\\ &= n\log(2) + 1/2\log\sqrt{2} - 1/2\log\pi n\\ &= \Theta(n)\end{aligned}$$

sorted in order of increasing complexity:

$f_2(n) = 1/n$
$f_6(n) = (\log\log n)^5$
$f_4(n) = \log n$
$f_3(n) = \sqrt{n}$
$f_1(n) = 10^{100}n$
$f_5(n) = n\log n$
$f_7(n) = n^2$
$f_8(n) = 1.001^n$
$f_9(n) = 2^n$

$f_1(n) = \log(n^n) = n\log n$
$f_4(n) = \binom{n}{100} = \Theta(n^{100})$
$f_3(n) = n^{\log n} = \Theta(10^{(\log n)^2})$
$f_6(n) = \binom{n}{n/2} = \Theta(10^n)$
$f_2(n) = (\log n)^n = \Theta(10^{n\log\log n})$
$f_5(n) = n!$

recurrences:

$T(n) = T(\sqrt{n}) + \Theta(\log n) = \Theta(\log n)$, $k = \log n$ and $S(m) = T(2^m)$
$T(n) = 2T(n/2) + n^2 = \Theta(n^2)$ case 3
$T(n) = 8T(n/2) + \log n = \Theta(n^3)$ case 1
$T(n) = T(n/4) + T(3n/4) + n = \Theta(n\log n)$
$T(n) = T(n/3) + T(2n/3) + \Theta(n) = \Theta(n\log n)$
$T(n,k) = kT(n/k,k) + O(n\log k)$ for k-mergesort

peak finding:

- $\Theta(N\log N)$ **algorithm:** Recall the $\Theta(N^2)$ algorithm which first finds the column maximums and then looks for a 1D-peak in that vector. We don't have to find the maximum number in every column. Instead, we just need to calculate the column maximums when we need them, so the number of column maximums that will be calculated should be no more than $\Theta(logN)$.
- $\Theta(N)$ **algorithm:** (This algorithm cuts the array into quarters.)
  - step 1 : if the current search window is a row vector or a column vector, return the global maximum of it and the algorithm terminates.
  - step 2 : finds the global maximum, say $A_{i,j}$, among the center column, center row and the boundaries of the current window.
  - step 3 : if this number is a 2D-peak, the algorithm terminates.
  - step 4 : by the definition of the 2D-peak, there must be at least one neighbor greater than $A_{i,j}$. Pick the quarter containing a greater neighbor of $A_{i,j}$ as the new search window and go to step 1.

  Beware, it must also track best seen! See `crossalgo.py`.

Counting/radix sort

1. Create list C with size k with all elements initialized to 0
2. For every element in A, increment $C[A[i]]$ by 1
3. Iterate through C, setting $C[i] = C[i] + C[i-1]$
4. Iterate backwards through A, setting $R[C[A[i]]] = A[i]$ and decrementing $C[A[i]]$
5. Radix sort is counting sort on digits of elements, first to last

Hashing

1. $h(k)$ maps every key k in the universe U to some index $0 \le i < m$
2. Simple uniform hashing assumption: $P(h[k_1] = h[k_2]) = 1/m$
3. Division method: $h(k) = k$ mod m (m is a prime far from 2^n)
4. Multiplication method: $h(k) =$ floor$(m(kA \bmod 1)$
   a. Works well for any m, but A should not be rational
5. Words can be represented in base 26
6. Chaining is a way of resolving collisions
7. Insert: hash k to an index i in the table; insert in linked list
8. Search: hash k to an index i; iterate through linked list
9. Delete: hash k to an index i; remove from linked list
10. **Load factor**: $\alpha = n/m$, where n is number of elements
11. **Insert** $O(1)$, **delete** (by value) $O(\alpha)$, **delete** (by ref) $O(1)$, **search** $O(\alpha)$

Table doubling/halving

1. Adjust table size so that $m = O(n)$, keep α small
2. When $n = m$, double table size, pick a new h, rehash everything
3. Amortized cost of a single insert is $O(1)$
   a. Cost of i^{th} insert is i if $i - 1 = 2^n$ and 1 otherwise
4. Halve table size when $n \le m/4$
   a. After halving, $n = m/2$ and $m/4$ deletes before rehalving
   b. Cost at next halving is $O(m/2)$
   c. $m/4$ deletes take $O(1)$, amortized

Rolling hash & Rabin-Karp

1. Pattern string search; naive algorithm is $O(nk)$
2. If we have $h(t_{i,k})$ we can compute $h(t_{i+1,k})$ in $O(1)$
3. To get $h("cde")$ from $h("bcd")$
   a. Remove b - obtain a hash for cd
   b. Multiply by 26 (left-shift) - obtain a hash for cda
   c. Add e - obtain a hash for cde
4. So we can compare hashes in $O(1)$ and compute new ones in $O(1)$
5. For n locations, the Rabin-Karp algorithm is $O(n)$
6. Abstract data structure: supports hash, append, skip
   a. Treat it as a multidigit number u in base a
7. **Append**(rh, x): convert x to value val in base a; get rolling hash attributes u (current num), a (base), p (prime), l (length); update $u = ua$ mod $p + val$; $u = u$ mod p; update u and $l = l + 1$ in rh
8. **Skip**(x): convert x to value val in base a; get rolling hash attributes u (current num), a (base), p (prime), l (length); compute part we have to remove $r = ((a^{l-1} \bmod p)\ val) \bmod p$; $u = u - r \bmod p$; update u and $l = l - 1$ in rh
   a. To avoid recomputing a^{l-1}, cache the power of a in $exp = a^{l-1}$ across append and skip operations
9. **Hash**: return u, which is maintained mod p

Open Addressing

- Probe sequence given by $h(k,0),\ h(k,1),\ \dots,\ h(k,m-1)$
- Insertion: $\forall i \in \{0,1,\dots,m-1\}$ try inserting at $h(k,i)$
    - If allowing overriding deletions, search first
- If all m locations exhausted, table is full or malformed h (stuck in a hashing loop)
- Search: look through $h(k,i)$ until you've performed m checks, you've found k, or you've found an *empty* slot
- Deletion: mark with a DELETED constant

Benefits

- Saves space of list pointers, better locality of reference, linked list can wander all around, needs good hash;

Linear probing: $h_m(k,i) = (h'(k) + i)\ mod\ m$

- Creates clusters of size $\Theta(\log k)$

Quadratic probing

- $h_m(k,i) = (h'(k) + ai^2 + bi)\ mod\ m$

Double hashing: $h(k,i) = f(k) + i \times g(k)\ mod\ m$

- $\forall k.\ g(k)$ relatively prime to m (e.g. $m = 2^p, g(k)$ odd
- Ensures probe sequence is exhaustive

UHA: probe sequence is a uniform random permutation of $1 \dots m$

- Expected number of probes is $1/(1-n/m) = 1/(1-\alpha)$

Graph Representation

	Adjacency list	**Adjacency matrix**
Description	Direct access table with linked lists Number all vertices Each linked list contains all neighbors of a specific node	n nodes, matrix is size n*n Each entry in the table is 0 or 1 All 0s on diagonal because no self-loops
Space	$\Theta(n+m)$	$\Theta(n^2)$
Add or remove edge	$\Theta(1)$	$\Theta(1)$
Find edge	$\Theta(\#\ neighbors)$	$\Theta(1)$
Visit neighbors	$\Theta(\#\ neighbors)$	$\Theta(n)$

Breadth First Search

Implementation

- Uses a queue (FIFO); visit node, add all neighbors to queue, pop off next from queue
- May use visited set to keep track of expanded nodes

Runtime

- $O(|V|+|E|)$ - every node and edge is visited at most once
- Becomes $O(|V|^2)$ if an adjacency matrix is used since getting neighbor list for every node takes $O(|V|)$
- # of edges falls between 0 and $|V|^2$ depending on connectivity of graph

Results

- Finds the single-source shortest path between two points
- Example: die hard problem; reachable states

Depth First Search

Implementation

- Uses a stack (LIFO); visit each node recursively
- n two points
- **Tree edge** - edge in DFS tree
- **Back edge** - edge that goes up to a previous level
- **Forward edge** - edge that goes down to a future level
- **Cross edge** - connects branches
- Undirected graphs can't have forward or cross
- Keep track of when each vertex is discovered and finished

Results

- **Topological sort**: reverse order of finishing time
- **Cycle finding**: look for back edges in DFS

Shortest Paths

- BFS may not work on weighted graphs

Variants

- Single-source shortest paths problem: path to every other vertex in the graph (Dijkstra, Bellman-Ford)
- Single-destination shortest paths problem: path from every other vertex in the graph (Dijkstra, Bellman-Ford on reversed graph)
- Single-pair shortest path problem: between two specified vertices (Dijkstra, Bellman-Ford)
- All-pairs shortest path problem: between every pair of vertices (Dijkstra/Bellman-Ford V times, Floyd-Warshall)

Notation/augmentations

- $v.d$ - weight of current shortest path from s to v
    - Initialized to be ∞, by end is the weight of shortest path
- $v.\pi$ - parent of v in the current shortest path
- $w(u,v)$ - weight of edge from u to v
- $\delta(u,v)$ - weight of shortest path from u to v

Structure of shortest path algorithms

```
Initialize:            for $v \in V$:  $v.d \leftarrow \infty$
                                       $v.\pi \leftarrow$ NIL
                       $s.d \leftarrow 0$
Main:                  repeat
                           select edge $(u,v)$   [somehow]
"Relax" edge $(u,v)$         if $v.d > u.d + w(u,v)$ :
                                 $v.d \leftarrow u.d + w(u,v)$
                                 $v.\pi \leftarrow u$
                       until all edges have $v.d \le u.d + w(u,v)$
```

Running time depends on order of edge relaxation

Relaxation

```
RELAX(u, v):
  if v.d > u.d + w(u, v) ## if we find a shorter path to v through u
    v.d = u.d + w(u, v) ## update current shortest path weight to v
    v.pi = u ## update parent of v in current shortest path to v
```

Properties

- **Triangle inequality**: $\delta(s,v) \le \delta(s,u) + w(u,v)$
- **Optimal substructure**: if path is shortest path, subpath is also shortest
- **Upper-bound**: $v.d \ge \delta(s,v)\ \forall v$. Once $v.d = \delta(s,v)$, it never changes.
- **No-path**: if s and v aren't connected, $v.d$ will be ∞
- **Convergence**: if shortest path between s and v contains (u,v) and $u.d = \delta(s,u)$ before relaxing (u,v), then $v.d = \delta(s,v)$ afterward
- **Path-relaxation**: if edges in a path are relaxed in order from v_1 to v_k, then $v_k.d = \delta(v_1, v_k)$ afterward
- **Predecessor subgraph**: subgraph of G that contains all vertices with a finite distance from s & edges connecting v to $v.\pi$
    - Once $v.d = \delta(s,v)$, it's shortest-paths tree rooted at s

Graph transformation

Given a weighted graph $G = (V, E, w)$, suppose we only want to find a shortest path with odd number of edges from s to t. To do this, we can make a new graph G'. For every vertex u in G, there are two vertices u_E and u_O in G': these represent reaching the vertex u through even and odd number of edges respectively. For every edge (u,v) in G, there are two edges in G': (u_E, v_O) and (u_O, v_E). Both of these edges have the same weight as the original. Constructing this graph takes linear time $O(V+E)$. Then we can run shortest path algorithms from s_E to t_O.

Strategies

- graph layering (adding another state)
- adding a node or two
- duplicate vertices
- changing the queue/stack structure
- reweight edges

Dijkstra

- Maintain a frontier of visited vertices

```
DIJKSTRA $(G, W, s)$        //uses priority queue Q
    Initialize $(G, s)$
    $S \leftarrow \phi$
    $Q \leftarrow V[G]$        //Insert into $Q$
    while $Q \neq \phi$
        do $u \leftarrow$ EXTRACT-MIN$(Q)$        //deletes $u$ from $Q$
        $S = S \cup \{u\}$
        for each vertex $v \in$ Adj$[u]$
            do RELAX $(u, v, w)$    $\leftarrow$ this is an implicit DECREASE_KEY operation
```

- Overall runtime $O(V \cdot T_{\text{Extract-Min}} + E \cdot T_{\text{Decrease-Key}})$
- Order of relaxation: from vertex with minimum v_d
- Priority queue implementations

	$T_{extract-min}$	$T_{decrease-key}$	Total
Array	$O(V)$	$O(1)$	$O(V^2 + E)$
Binary min-heap (array)	$O(\log V)$	$O(\log V)$	$O((V+E)\log V)$
Fibonacci	$O(\log V)$ amortiz.	$O(1)$ amortiz.	$O(V \log V + E)$ worst case

- Fibonacci is optimal because of $O(n \log n)$ sorting bound
- Will return incorrect results for negative-weight edges
    - Only if negative edge eventually goes into already processed node and that node has out-edges

Bellman-Ford

- Can handle graphs with negative weights & detect negative cycles
- Runtime $O(|V||E|)$
- Executes with $|V|-1$ iterations; relax edges sequentially

```
Initialize
for $i = 1$ to $|V| - 1$
    for each edge $(u, v) \in E$:
        Relax$(u, v)$
for each edge $(u, v) \in E$
    do if $v.d > u.d + w(u, v)$
        then report a negative-weight cycle exists
```

-

Theorem 5 ***Correctness of Bellman-Ford***
If G contains no negative cycles reachable from s, the algorithm find no negative cycles and $v.d = \delta(s, v)$ for all $v \in V$. If G does contain a negative weight cycle, the algorithm will return that there is a negative cycle in the graph.

Proof. If G contains no negative weight cycles, by the lemmas above, $v.d = \delta(s, v)$ for all $v \in V$ after the termination of the algorithm. Therefore $v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) \leq u.d + w(u, v)$ by the triangle inequality for all edges (u, v). Thus no edge can still be relaxed.

Now assume G contains a negative weight cycle that is reachable from s. Let this cycle be $c =< v_0, v_1, ..., v_j >$ with $v_0 = v_j$, such that $\sum_{i=1}^{j} w(v_{i-1}, w_i) < 0$. We proceed by contradiction. Assume that the last check in BELLMAN-FORD does not return a negative weight cycle. This means that for any edge (u, v) in the graph $u.d \leq v.d + w(u, v)$. Summing these up for all edges in the cycle we get that $\sum_{i=1}^{j} v_i.d \leq \sum_{i=0}^{j-1} v_i.d + w(v_i, v_{i+1})$. But $\sum_{i=1}^{j} v_i.d = \sum_{i=0}^{j-1} v_i.d$ since $v_0 = v_j$, so we get that $\sum_{i=0}^{j-1} w(v_i, v_{i+1}) = \sum_{i=1}^{j} w(v_{i-1}, w_i) \geq 0$ which is a contradiction with the fact that the cycle has negative weight. Thus BELLMAN-FORD does return that the graph has a negative weight cycle if such a cycle exists.

- Can terminate early if all $v.d$ were unchanged in an iteration (means that all shortest paths have been found)
    - DAG-SP is a special case where topological sort order guarantees we can find all shortest paths after one iteration

Dijkstra Modifications

Single-pair shortest path

- Terminate as soon as
- t is extracted from priority queue

```
Initialize()
$Q \leftarrow V[G]$
while $Q \neq \phi$
    do $u \leftarrow$ EXTRACT_MIN(Q)
    if $u == t$: break
    for each vertex $v \in$ Adj$[u]$
        do RELAX$(u, v, w)$
```

-

Bidirectional search

- Use on highly branched graphs (with branching factor
- b)
- Doesn't change worst-case time but optimizes average
- Forward search from s in parallel with search from t
    - Maintain a variable μ = weight of current best-seen path from s to t
- Every time we encounter an edge between forward and reverse search:
    - Update $\mu = \min(\mu, p.s + w(p,q) + q.t)$
    - Find l_s (minimum of $v.s$) over every unexpanded node in forward queue
    - Find l_t (minimum of $u.t$) over every unexpanded node in backwards queue
    - If $l_s + l_t \geq \mu$, all remaining paths on queue can't be smaller, so stop.

Correctness Assume that there exists a path p from s to t with weight smaller than μ. Let (u, v) be an edge on path p s.t. $\delta(s, u) < l_s$ and $\delta(v, t) < l_t$. This means that both u and v have been extracted from their corresponding queues. Without loss of generality, assume that v was processed after u. Then when v was extracted, μ was updated to a value at most the weight of path p. That is, $\mu <= \delta(s, u) + w(u, v) + \delta(v, t)$. However, this contradicts our assumption that the weight of p is smaller than μ.

Performance Assume that a highly branching graph has a branching factor b, and the diameter is roughly $\log_b n$. The forward and backward searches can potentially meet in the middle of this diameter, each exploring a depth of $\frac{\log_b n}{2}$. Thus the number of vertices explored by each search direction is approximately b to the power of depth, which is $n^{\frac{1}{2}}$. Thus in the best case, bidirectional search only explores $O(\sqrt{n})$ vertices.

DFS Cycle Detection

- ***parent*** = {}
- for u in V
    - ***parent***$[u]$ = None; ***ancestor*** = []
    - call ***DFS-cycle-detection*** (V, Adj, u)

```
Define DFS-cycle-detection (V, Adj, $u$):
    ancestor.add($u$)
    for $v$ in Adj[$u$]
        if $v$ not in parent                #not yet seen
            parent[$v$] = $u$
            DFS-cycle-detection (V, Adj, $v$)       #recurse!
        else if $v$ in ancestor             #back edge
            exit("Cycle!")
    ancestor.remove($u$)
```

Summary

- Breadth-first search $O(|V| + |E|)$
- Depth-first search $O(|V| + |E|)$
- DAG-SP $O(|V| + |E|)$
- Dijkstra $O(|V| T_{extract-min} + |E| T_{decrease-key})$
- Bellman-Ford $O(|V||E|)$

Arbitrary Size Integers

- Floating point # with d digits of precision is of the form $y \times B^e$ where $y \in [B^{d-1}, B^d)$ and e small integer (≈ 0)

Addition

- Grade-school adding is $\Theta(d)$

Multiplication/division

- Karatsuba's is $\Theta(d^{\log_2 3})$ from recurrence $T(d) = 3T(d/2) + \Theta(d)$

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

$$x = x_1 B^m + x_0$$
$$y = y_1 B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0)$$
$$xy = z_2 B^{2m} + z_1 B^m + z_0$$

where

$$z_2 = x_1 y_1$$
$$z_1 = x_1 y_0 + x_0 y_1$$
$$z_0 = x_0 y_0$$

These formulae require four multiplications, and were known to Charles Babbage.[4] Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before we can calculate

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

which holds since

$$z_1 = x_1 y_0 + x_0 y_1$$
$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

Square roots

- Use Newton's method
- $x_{i+1} = (1/2)(x_i + a/x_i)$
- Each iteration uses one addition, one easy division by 2 (shift), and one division by x_i
- Brute force is $\Theta(d^\alpha B^d)$; binary search is $\Theta(d^{\alpha+1})$

Termination

- Perform fixed # of iterations and then return estimate
- Perform iterations until estimates change only beyond dth digit

Correctness of Newton's

- Suppose x_i is off from $\sqrt{a}$ by $1 + \epsilon_i$; use definition
- Next step has error $(1/2)\epsilon_i^2/(1+\epsilon_i)$ (quadratic converge.)
- Want $\epsilon_i < 1/2^d$ so each iteration takes us to $2d+1$ digits
- Runtime of $O(d^\alpha \lg d) \rightarrow \Theta(d^\alpha)$ like multiplication

High precision division

- Compute $1/b = R/b$ where $R = 2^k$ (easy to divide)
- Newton's method for computing R/b:
- $x_{i+1} = 2x_i - bx_i^2/R$

Cube roots

$$\begin{aligned} f(x) &= x^3 - a \\ f'(x) &= 3x^2 \\ x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{1}{3}\left(x_i - \frac{a}{x_i^2}\right) \\ &= \frac{2}{3}x_i + \frac{1}{3}\frac{a}{x_i^2} \end{aligned}$$

Dynamic Programming

- Take advantage of optimal substructure of problem
- Runtime is #problems × time/problem
- Need acyclic subproblem dependency

Memoized recursive

- Add a memo dictionary to store outputs for inputs encountered so far
- Easier to understand, don't have to determine ordering

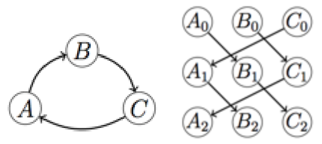
Bottom-up DP

- Avoid recursive overhead by building up subproblems from base case up
- Ordering subproblems is same as topologically sorting DAG defining dependencies of subproblems
- Doesn't have recursive overhead, easier to analyze time

Robotic coin collection: $O(mn)$

$$\begin{aligned} S(i,\,j) &= max(S(i-1,\,j),\,S(i,\,j-1)) + c_{ij},\ i \ge 1,\ j \ge 1 \\ S(0,\,0) &= 0 \\ S(i,\,0) &= S(i-1,\,0) + c_{i0},\ i \ge 1 \\ S(0,\,j) &= S(0,\,j-1) + c_{0j},\ j \ge 1 \end{aligned}$$

Shortest path

- **Single-source:**
  $$\delta(s,v) = \min\{\delta(s,u) + w(u,v) \mid (u,v) \in E\}$$
- **DAG-SP:** use memoized DP recursion ($\Theta(|E| + |V|)$)
- Need to transform graph to be acyclic
- 

Crazy 8s

In the game Crazy 8's, we are given an input of a sequence of cards $C[0], \ldots, C[n-1]$, e.g., 7♣, 7♡, K♣, K♠, 8♡. We want to find the longest subsequence of cards where consecutive cards must have the same value, same suit, or have one of the two cards be an eight. The longest such subsequence in the example is 7♣, K♣, K♠, 8♡.

To solve this, if the cards are stored in array C, we will to keep an auxiliary score array S where $S[i]$ represents the length of the longest subsequence ending with card $C[i]$.

We start with $S[0] = 1$ since the longest subsequence ending with the first card is that card itself and has a length of 1. We iteratively calculate the next score $S[i]$ by scanning all previous scores and set $S[i]$ to be $S[k] + 1$ where $S[k]$ represents the length of the longest subsequence that card $C[i]$ can be appended to.

Longest common subsequence: $O(mn)$

$$S[i][j] = \begin{cases} S[i-1][j-1] + 1 & \text{if } A[i] = B[j] \\ \max\{S[i][j-1], S[i-1][j]\} & \text{otherwise} \end{cases}$$

Knapsack problem: $O(nS)$, n items, S pounds

- $dp[i][j]$ is the max value using items $i+1 \ldots n-1$ (last $n-i$ items) which weighs at most j lbs
- pseudo-polynomial runtime

$$dp[i][j] = \max\begin{pmatrix} dp[i+1][j] & \\ dp[i+1][j-s_i] + v_i & \text{if } j \ge s_i \end{pmatrix}$$

```
KNAPSACK($n, S, s, v$)
for $i$ in $\{n, n-1 \ldots 0\}$
    for $j$ in $\{0, 1 \ldots S\}$
        if $i == n$
            $dp[i][j] = 0$ // initial condition
        else
            $choices = []$
            APPEND($choices, dp[i+1][j]$)
            if $j \ge s_i$
                APPEND($choices, dp[i+1][j-s_i] + v_i$)
            $dp[i][j] = $ MAX($choices$)
return $dp[0][S]$
```

GCD/Euclidean algorithm

- Find GCD d of large numbers a and b st. $d|a$, $d|b$, and any common divisor of a and b divides d
- $a_k = a_{k+1}q_{k+1} + a_{k+2}$ where $0 \le a_{k+2} < a_{k+1}$
- $\Theta(d)$ iterations; each iteration is $\Theta(d^{\alpha})$ so overall is $O(d^{1+\alpha})$

(i) $T(1) = O(1), T(n) = 2T(n/2) + O(n)$ for $n > 1$. Solution: $T(n) = O(n^2)$

(ii) $T(1) = O(1), T(n) = 8T(n/2) + O(n^2)$. Solution: $T(n) = O(n^3)$

(iii) $T(1) = O(1), T(n) = 7T(n/2) + O(n^2)$. Solution: $T(n) = O(n^{\log_2 7})$

(iv) $T(1) = O(1), T(n) = 8T(n/2) + O(n^3)$. Solution: $T(n) = O(n^3 \log n)$

(v) $T(1) = O(1), T(n) = T(n/2) + 2T(n/4) + O(n)$. Solution: $T(n) = O(n^2)$

Answer ii,iii,iv (^)

☐ (i) Merge Sort takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (ii) Insertion Sort takes time $\Theta(n^2)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (iii) Heap Sort based on a max-heap takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted in decreasing order.

☐ (iv) Binary Search Tree Sort takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (v) Radix Sort for base 10 numbers of length at most d takes time $\Theta(dn)$ in the worst case.

Answer ii,v (^)

☐ (i) Finding the median key value, in time $O(\log n)$.

☐ (ii) Returning a sorted list of all the keys in the BST, in time $O(n)$.

☐ (iii) Finding the mean of all the keys in the tree, in time $O(h)$.

☐ (iv) Finding the key that is the $n/4^{th}$ largest, in time $O(h)$.

☐ (v) Finding all keys with values between a and b, for some arbitrary a and b, in time $O(n)$.

Answer ii,iv,v (^)

[5 points] Which of the following are guaranteed to be TRUE for an arbitrary AVL tree?

☐ (i) The median key is guaranteed to appear in the top node.

☐ (ii) The key in any node is at least as great as the keys in both of its children nodes.

☐ (iii) A list of the keys in the tree, in sorted order, can be produced in time $O(n)$.

☐ (iv) Rebalancing the tree after a single insertion can be done in time $O(1)$.

☐ (v) Given any key value k, the smallest key in the tree that is strictly larger than k can be found in time $O(\log n)$.

Answer iii,v (^)

☐ (i) Newton's method for square roots always converges for a positive initial guess.

☐ (ii) Newton's method for square roots always converges for a sufficiently close initial guess.

☐ (iii) Newton's method for reciprocal always converges.

☐ (iv) Newton's method for reciprocal always converges for a sufficiently close initial guess.

☐ (v) Newton's method for reciprocal always converges quadratically for a sufficiently close initial guess.

Answer i,ii,iv,v (^)

[5 points] Which of the following must be TRUE about the structure produced by Depth-First Search for an undirected graph G?

☐ (i) All the edges are tree edges or back edges, and there are no cross edges.

☐ (ii) Two nodes u and v that are reachable from each other in G must be in the same tree.

☐ (iii) If G is a connected graph then the DFS forest consists of a single tree.

☐ (iv) If u is the root of some tree in the DFS forest and u's removal from G disconnects G, then G has more than one child in its DFS tree.

☐ (v) If u is an internal node of some tree in the DFS forest and u's removal from G disconnects G, then no child of u in its DFS tree has a back edge to a proper ancestor of u.

Answer i,ii,iii,iv (^)

It is possible to sort any n integers between 0 and n^{100} in $O(n)$ time

[3 points] Given two integers a, b where $a < m$, we can (using algorithms presented in 6.006) compute $a^b \bmod m$ in $O((\lg m)^{\lg 3} \lg b)$ bit operations.

All-pairs shortest paths

- Floyd-Warshall for weighted graph with no negative cycles
- $dp(i,j,k+1) = min(dp(i,j,k), dp(i,k+1,k) + dp(k+1,j,k))$
- Base case: $dp(i,j,0) = w(i,j)$
- $dp(i,j,k)$ is shortest path from i to j using only first k vertices

☐ (i) Topological Sort sorts an array of numbers of length n in time $O(n)$.

☐ (ii) A Topological Sort algorithm sorts the nodes of an arbitrary directed graph G in an order that is consistent with all the paths in G, that is, if there is a path from u to v in G then u precedes v in the resulting sorted list.

☐ (iii) Topological Sort of a DAG can be implemented easily using Depth-First Search, in time $O(V+E)$, based on the times when the nodes in the DAG are first encountered in the DFS.

☐ (iv) Topological Sort can be used as a subroutine to find shortest paths in a weighted DAG in time $O(V+E)$; in particular, the time does not depend on the magnitudes of the weights on the edges, and the weights on the edges may be negative.

☐ (v) When Topological Sort is used to find shortest paths in a weighted DAG, no relaxation steps are necessary.

Answer iv (^)

Solution:

Let n_h denote the minimum number of nodes in an AVL tree of height h. We know that $n_0 = 1$ and $n_1 = 2$. Furthermore, $n_k \ge n_{k-1} + n_{k-2} + 1$. Because $n_{k-1} \ge n_{k-2}$, $n_k \ge 2n_{k-2}$. By repeated substitution, $n_k \ge 2n_{k-2} \ge 4n_{k-4} \ge 8n_{k-6} \cdots \ge 2^{k/2}$.

Given a AVL tree with n nodes and height h, we know that $n \ge n_h \ge 2^{h/2}$. Taking the log of both sides yields $h \le 2 \log n$, as desired.

Proof that AVL property > balanced

Once items can be repeated, the list of items does not matter as much. The subproblem is $DP(W)$, which represents the max value when capacity W is left in the knapsack. the recurrence becomes the following:

$DP(W) = max_i(DP(W - w_i) + p_i)$

In the above formulation, we choose the best item i to add to the knapsack with capacity W remaining, and select the best of the choices (while also enforcing $w_i \le W$ for the choice). The number of sub-problems is W, and the running time per sub-problem is $O(n)$, so the total time is also $O(nW)$. We keep on going until we run out of space, so one base case is $DP(0) = 0$. The other thing we need to worry about is if all $w_i > W$; that is, all of the remaining weights are greater than the capacity left. We can treat this as another base case: $DP(c) = 0$ if $c < w_i$ for all i.

Knapsack with repetition

$$T(n) = 2T(n/2) + \frac{n}{\log n}$$

Master theorem can't be used for this

Alternate solution: We could also make $|V|^2$ copies of the original graph, where each copy is described by a tuple (i, j), where i is the total number of red edges used thus far and j is the total number of green edges used thus far. We can then run a BFS and compare the paths from $s_{0,0}$ to $t_{i,j}$ for each i and j. This solution incurs a runtime of $O(|V|^2 \cdot |E|)$.

Solution: Build the graph $G = (V, E, w)$ as described above. Based on this, we will build a new graph G'. Informally, G' will consist of four copies of G, where the first copy corresponds to having spent no time talking to relatives, the second to having spent exactly one hour talking to relatives, the third to having spent exactly two hours talking to relatives, and the fourth to having spend at least three hours talking to relatives. Then, we just want to return the longest path from s in the first copy to t in the fourth copy.

Formally, we create the graph as follows:

1. For each $v \in V$, G' contains vertices v_1, v_2, v_3, and v_4.
2. For each edge $(u, v) \in E$ which does not correspond to talking to a relative, add the edges (u_1, v_1), (u_2, v_2), (u_3, v_3), and (u_4, v_4) to G'.
3. For each edge $(u, v) \in E$ corresponding to talking to a relative for at least three hours, add the edges (u_1, v_4) and (u_4, v_4) to G'.
4. For each edge $(u, v) \in E$ corresponding to talking to a relative for exactly two hours, add the edges (u_1, v_3), (u_2, v_4), and (u_4, v_4) to G'.
5. For each edge $(u, v) \in E$ corresponding to talking to ta relative for exactly one hour, add the edges (u_1, v_2), (u_2, v_3), (u_3, v_4), and (u_4, v_4) to G'.

Since G' is four times the size of G, we can still build the graph in $O(n+m)$. Now, any path from s_1 to t_4 must correspond to talking to relatives for at least three hours, and the longest path corresponds to the one that maximizes happiness. As before, we can negate the edges and run DAG-SP to find the longest path in $O(n+m)$.