Contents

[1]This example is Applied Mathematical Programming by Bradley,Hax, Magnanti chapter 11.
[2]From here on down I stole these from https://people.cs.clemson.edu/~bcdean/dp_practice/

Note

Everything is 1 indexed, despite using vaguely Pythonic syntax. This means $A[\text{len}(A)] = A[-1]$. Slicing is $A[a:b] = [A_a, A_{a+1}, \ldots, A_{b-1}]$. Where bounds checking is obviously necessary it is omitted. I assume a different memory model from Python: each entry of $B = [[]]$ is an independent list. Ranges are represented using MATLAB notation $1:n$. In certain place I play fast and loose with what a dictionary is keyed on and whether a label is just a label or a pointer (in particular in the Graph Algorithms section). Also I iterate over a dictionary, which is possible with python's `dict.items()`.

Part 1. Foundations

1. Insertion Sort

Maintains the invariant that $A[1:j-1]$ is sorted by shifting elements right. Insertion sort is *stable*, i.e. two keys already in sorted order remain in the same order at the end. Running time is $O(n^2)$.

```
Insertion-Sort(A)
for j = 2 : len(A):
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key:
        A[i + 1] = A[i]
        i = i − 1
    # either we're one passed the left end
    # or A[i] ≤ key and so
    # A[i + 1] is the proper place for key
    A[i + 1] = key
```

2. Selection Sort

Maintains the same invariant as Insertion Sort but does so by going forward and *selecting* the smallest element each time. Running time is $O\left(n^2\right)$.

```
Selection-Sort(A)
for j = 1 : len(A):
    A[j] = min(A[j + 1 :])
```

3. Bubble Sort

"Bubble up" pair by pair. Stop when no more "bubblings" are possible. Running time is $O\left(n^2\right)$.

```
Bubble-Sort(A)
flips = True
while flips:
    flips = False
    for i = 1 : len(A) − 1:
        if A[i] > A[i + 1]:
            A[i], A[i + 1] = A[i + 1], A[i]
            flips = True
```

4. Merge Sort

Divide and conquer approach. Divide the array in half, recurse, combine results by merging, i.e. taking the smallest entry from each piece in turn. Base case is just an array with one element. Running time is $O(n \lg n)$

```
Merge-Sort(A)
if len(A) == 1:
    return A
else:
    h = ⌊len(A)/2⌋
    L = Merge-Sort(A[h:])
    R = Merge-Sort(A[1:h])
    M = [ ]
    while len(L) > 0 and len(R) > 0:
        # take the minimum of the {L[1], R[1]}
        # and remove it from further contention
        if L[1] < R[1]:
            M.append(L[1])
            del L[1]
        else:
            M.append(R[1])
            del R[1]
    # one of L, R is large by one element.
    if len(L) > 0
        M.append(L[1])
    else:
        M.append(R[1])
        M.append(R[-1])
    return M
```

5. Binary search

If an array is already sorted then you can find an element in it faster than $O(n)$ time; you can find it in $O(\lg n)$ time. Search in either the left side of the middle entry or the right side.

```
Binary-Search(A, x)
if x == A[h]:
    return True
elif x < A[h]:
    return Binary-Search(A[1:h])
else:
    return Binary-Search(A[h:])
```

6. Horner's Rule

Given $A = [a_1, \ldots, a_n]$ the coefficients of a polynomial and a value x a faster way to calculate $p(x)$ is

$$p(x) = \sum_{k=1}^{n} a_k x^k = a_1 + x\left(a_2 + x\left(a_3 + \cdots + x\left(a_{n-1} + x a_n\right)\right)\right)$$

```
Horners-Rule(A, x)
1  y = 0
2  for i = n : 1:
3      y = A[i] + x · y
```

7. Reservoir Sampling

7.1. **Unweighted simple.** Suppose you want to sample k items from n items $A = [a_1, \ldots, a_n]$ fairly, i.e. uniform random, **without replacement**, draws. If you have all n items available immediately then this is simple, but if you're solving the problem *online* it's slightly more involved. For example you might not want to store all n items. Put the first k items into a *reservoir* R then for item $i > k$ draw $j \in \{1, \ldots, i\}$ inclusive. If $i \leq k$ the replace ith item. Running time is $\Theta(n)$.

```
Unweighted-Reservoir-One(A, k)
1  R = [a_0, a_1, ..., a_k]
2  for i = k + 1 : len(A):
3      j = Random(1, i) # both ends inclusive
4      if j ≤ k:
5          R[j] = A[i]
```

7.2. **Unweighted slightly more involved.** Another way to do solve the same problem is to use a priority queue. Why complicate things? This solution generalizes to weighted sampling. Running time takes $O(n \lg k)$ because of potentially n `Extract-Min` operations on a k length priority queue.

```
Unweighted-Reservoir-Two(A, k)
R = Min-Priority-Queue
for i = 1 : k:
    u ~ Uniform(0, 1)
    # priority key is first entry in argument
    H.insert(u, A[i])
for i = k + 1 : len(A):
    u ~ Uniform(0, 1)
    # H.min returns value of minimum without extracting
    if u < H.min:
        H.Extract-Min()
        H.insert(u, A[i])
```

7.3. **Weighted.** Suppose the same sampling problem but each element has a weight associated with it. `Unweighted-Reservoir-Two` extends naturally (sort of).

```
Weighted-Reservoir(A, k)
R = Min-Priority-Queue
for i = 1 : k:
    u ~ Uniform(0, 1)
    u = u^(1/A[i].weight)
    H.insert(u, A[i])
for i = k + 1 : len(A):
    u ~ Uniform(0, 1)
    u = u^(1/A[i].weight)
    if u < H.min:
        H.Extract-Min()
        H.insert(u, A[i])
```

8. Online Maximum

Suppose you wanted to compute a maximum of n items but we can only make the selection once. This is similar to online sampling: fill a reservoir R full of candidates and pick the maximum from the reservoir. Then after finding that maximum pick the next maximum (if one exists) that's higher; this will be the single selection. But what size should the reservoir be? Turns out if $k = n/e$ where e is $\exp(1)$ then we'll pick the true maximum with probability at least e^{-1}. This can be further simplified by realizing you don't need to keep the entire reservoir and you can return after the first forthcoming maximum (if one exists).

```
Online-Max(A, n)
1  m = A[1]
2  # these selections to count against the quota
3  for i = 2 : ⌈n/e⌉ :
4      if A[i] > m :
5          m = A[i]
6  # this one is for keeps
7  for i = k + 1 : len(A) :
8      if A[i] > m :
9          return A[i]
```

9. Stable Matching

The task is given n men and n women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners (a stable matching). One question is does such a stable matching even exist? In fact it does and the algorithm that produces one, the Gale-Shapley algorithm, proves it. It runs in $O\left(n^2\right)$. The next question is the solution optimal. In fact it is not. The algorith is simple: first each man proposes to the woman he prefers best and each woman accepts provisionally, i.e. accepts a proposal but trades up if a more desirable man proposes. Do this for n rounds (or until there are no more unengaged men). Running time is $O\left(n^2\right)$

Matching(P_m, P_w, men)

```
# men is an array of men to be matched
# P_m is an n × n preferences matrix for the men, sorted by increasing priority
# P_w is an n × n a preferences matrix for the women, sorted
matched_M = {}
matched_W = {}
while len(men) > 0:
    m = men[-1]
    w = P_m(m)[-1]
    if w not in matched_W:
        matched_M[m] = w
        matched_W[w] = m
        del P_m(m)[-1]
        del men[-1]
    else: # if w is already matched
        m' = matched_W[w]
        # and prefers m to m'
        if P_w[w][m] > P_w[w][m']:
            # match m with w
            matched_M[m] = w
            matched_W[w] = m
            del P_m(m)[-1]
            del men[-1]
            # unmatch m'
            del matched_M[m']
            matched_M.append(m')
```

Part 2. Sorting and Order Statistics

10. Heaps

Array Heaps[3] are a data structure built on top of an array A, i.e. a structural invariant and a collection of functions that maintain that invariant. Heaps come in two flavors: Min heaps and Max heaps. The invariant for a Max heap is $A[i] \leq A[\lfloor i/2 \rfloor]$. Furthermore each entry has "children": $A[2i]$ is the left child and $A[2i+1]$ is the right child of element $A[i]$.

10.1. **Max Heapify.** To re-establish the heap property we use a procedure that fixes violations by switching the violator with its largest child and then recursing. Running time is $O(\lg n)$.

[3] Heaps can be built on top of trees.

```
Max-Heapify(A, i)
1  largest = i
2  # if the left child exists and is greater then potentially switch
3  if 2i ≤ len(A) and A[2i] > A[i]:
4      largest = 2i
5  # if the right child exists and is greater then switch
6  if 2i + 1 ≤ len(A) and A[2i + 1] > A[largest]:
7      largest = 2i + 1
8  A[i], A[largest] = A[largest], A[i]
9  # potentially fix violation between child and one of its children
10 Max-Heapify(A, largest)
```

10.2. **Build Max Heap.** To build a heap from an array notice that the deepest children/leaves are already legal heaps so there's no need to `Max-Heapify` them, and the children start at $\lfloor \text{len}(A)/2 \rfloor$. Running time is $O(n)$.

```
Max-Heapify(A)
1  for i = ⌊len(A)/2⌋ : 1:
2      Max-Heapify(A, i)
```

10.3. **Extract Min.** $A[1]$ is the maximum element in the heap (by the Max heap invariant), but removing it isn't as simple as just popping it off the top since the invariant might be violated. It's also not as simple as simple as replacing $A[1]$ with it's largest child because. The solution is to replace with the last element in the heap and then re-establish the invariant. Running time is $O(\lg n)$.

```
Extract-Min(A)
1  m = A[1]
2  A[1] = A[−1]
3  A.pop()
4  Max-Heapify(A, 1)
5  return m
```

10.4. **Heap sort.** You can use `Extract-Min` in the obvious way to sort an array. Running time is $O(n \lg n)$.

```
HeapSort(A)
1  s = [ ]
2  while len(A) > 0:
3      s.append(Extract-Min(A))
4  return reversed(s)
```

10.5. **Heap increase key.** In various instances you might want to increase the position of a key in the heap, such as when each key corresponds to the priority of some task. This just involves re-establish the Max heap invariant by "percolating" the entry up the array. Running time is $O(\lg n)$.

```
Heap-Increase-Key(A, i, key)
1  if key < A[i]:
2      throw Exception(key is smaller than current i key)
3  A[i] = key
4  # if child is bigger then parent then swap
5  while i > 1 and A[⌊i/2⌋] < A[i]:
6      A[⌊i/2⌋], A[i] = A[i], A[⌊i/2⌋]
7      i = ⌊i/2⌋
```

10.6. **Heap insert.** Using `Heap-Increase-Key` we can insert into the heap by insert and $-\infty$ element at the end of the heap and then increasing the key to what we want. Running time is $O(\lg n)$

```
Max-Heap-Insert(A, key)
1  A.append(−∞)
2  Heap-Increase-Key(A, len(A), key)
```

11. Quicksort

Quicksort is experimentally the most efficient sorting algorithm. The randomized version runs in $O(n \lg n)$ but is typically faster. It works by dividing and conquering.

```
Quicksort(A)
if len(A) ≤ 1:
    return A
else:
    # randomly pick a pivot
    p = Random(1, len(A))  # inclusive
    # swap so that you can exclude from contention the pivot
    A[p], A[−1] = A[−1], A[p]
    # partition
    A_left = filter(A[: −1], λe : e ≤ A[−1])
    A_right = filter(A[: −1], λe : e > A[−1])
    # recursively sort
    A_left = Quicksort(A_left)
    A_right = Quicksort(A_right)
    # combine
    return A_left + A[−1] + A_right
```

12. Counting Sort

The lower bound on sorting in the comparison model (i.e. using comparisons as an ordering relation) is $\Theta(n \lg n)$. But if one doesn't use comparisons then $\Theta(n)$ is possible. Counting sort is one such $\Theta(n)$ algorithm. If keys range from 1 to k in $A = [a_1, \ldots, a_n]$ then counting sort counts the number of keys less than or equal to each key a_i and then places a_i in that position.

```
Counting-Sort(A)
k = max(A)
C = (k + 1) * [0]
# count how many of values from 1 to k there is
for i = 1 : len(A):
    C[A[i]] = C[A[i]] + 1
# count how entries in A less or equal to i
for i = 1 : k:
    C[i] = C[A[i]] + 1
# now place the items in the correct places
B = (k + 1) * [None]
# go in reverse direction in order for sort to be stable
for i = len(A) : 1:
    # a_i has C[a_i] elements to its left in B
    B[C[A[i]]] = A[i]
    # if there are multiples of a_i then the next
    # should be left of in order for stable
    C[A[i]] = C[A[i]] - 1
```

13. Radix Sort

Radix sort use the same technique that casinos use to sort cards (apparently?): sort stably least significant to most significant digit. For n numbers in base d where each digit ranges from 1 to k the running time is $\Theta(d(n+k))$ if the stable sort runs in $\Theta(n+k)$.

```
Radix-Sort(A, d)
for i = 1 : d:
    # let's pretend i can pass Insertion-Sort a key
    Insertion-Sort(A, key=lambda a : a[-i])
```

14. Bucket Sort

Bucket sort depends on values being uniformly distributed $[0, 1]$. It buckets all the entries and then subsorts. Expected run time is $\Theta(n)$.

```
Bucket-Sort(A)
n = len(A)
B = n · [[ ]]
for i = 1 : n:
    # bucket (imagine n = 10).
    # the +1 is because ⌊10 (0.01)⌋ = 0
    B [⌊nA [i]⌋ + 1] .append (A [i])
for i = 1 : n:
    Insertion-Sort(B [i])
return B
```

15. Order statistics

15.1. **Quickselect.** Any sorting algorithm can be used to compute kth order statistics: simply sort and return the kth element. But using the ideas of `Quicksort` you can get down to expected time $O(n)$: only recurse to one side.

```
Quickselect(A, k)
if len(A) ≤ 1:
    return A [1]
else:
    p = Random (1, len (A)) # inclusive
    A [p] , A [−1] = A [−1] , A [p]
    A_left = filter (A [: −1] , λe : e ≤ A [−1])
    if k == len (A_left + 1)
        # in sorted order A [1 : len (A_left + 1)] = A_left + [A [−1]]
        # and so the pivot is 1 "in front" of A_left
        return A [−1]
    elif k < len (A_left)
        # the kth order statistic in A is still the kth order statistic in A_left
        return Quickselect(A_left, k)
    else:
        A_right = filter (A [: −1] , λe : e > x)
        # the kth order statistic is (k − len (A_left) − 1) th statistic in A_right
        # think about it likes this: A = [1, 2, 3, 4, 5] and we partition on
        # 3 and we look for the 4th order statistic. well obviously it's
        # 4 = A_right [k − len (A_left) − 1] = A_right [1]
        return Quickselect(A_right, k − len (A_left) − 1)
```

15.2. **Quickerselect.** Using *median-of-medians* in order to guarantee good splits we can get down to $O(n)$ worst case (not just expected).

```
Quickerselect(A, k)
if len(A) == 0:
    return A[1]
else:
    # divide into n groups of 5 (except for the last one)
    # and use a sort in order to get medians.
    n = ⌊len(A)/5⌋
    m_1 = Insertion-Sort(A[1 : 5 + 1])[3]
    m_2 = Insertion-Sort(A[5 : 10 + 1])[3]
    ⋮
    m_n = Insertion-Sort(A[5n :])[⌊⌊len(A[5n:])/2⌋⌋]
    # recursively compute median of medians and use it as the pivot
    # after this recursive call the pivot is in position ⌊n/2⌋
    Quickerselect([m_1, m_2, ..., m_n], ⌊n/2⌋)
    A[⌊n/2⌋], A[−1] = A[−1], A[⌊n/2⌋]
    x = A[−1]
    A_left = filter(A[: −1], λe : e ≤ x) + [x]
    if k == len(A_left)
        return x
    elif k < len(A_left)
        return Quickselect(A_left, k)
    else:
        A_right = filter(A[: −1], λe : e > x)
        return Quickselect(A_right, k − len(A_left))
```

Part 3. **Data Structures**

16. Hash Tables

Hash tables are m length arrays keyed on strings instead of numbers.

16.1. **Hash function.** A Hash function is something that "hashes" up strings into numbers. It should uniformly distribute the keys over the hash space, meaning each key k is equally likely to hash to any of the m slots of the hash table. A good hash function according to Knuth is

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

where $A \approx \left(\sqrt{5}-1\right)/2$ and $kA \mod 1$ means the fractional part of kA, i.e. $kA - \lfloor kA \rfloor$.

16.2. **Hashing with chaining.** Hashing with chaining is basically Bucket Sort, except with the $\lfloor \rfloor$ replaced by a Hash function and retrieval.

Hashing with Chaining

```
1  HashInsert(H, k, v)
2  # H is hash table, k is key, v is value
3      m = len(H)
4      hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
5      H[hsh].append(v)
6
7  HashRetrieve(H, k)
8      m = len(H)
9      hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
10     if len(H[hsh]) > 0:
11         return H[hsh][1]
12     else:
13         return None
14
15 HashDelete(H, k, v)
16     m = len(H)
17     hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
18     i = 1
19     while i ≤ len(H[hsh]):
20         if H[hsh][i] == v:
21             del H[hsh][i]
22             return
23         else:
24             i = i + 1
25     return "Error: v not in table"
```

If n is the total number of items in the hash table and m is the length of the hash table then on average (give uniform hashing) each list has $\alpha = n/m$ items. Therefore insertion is $\Theta(1)$, and retrieval/deletion is $\Theta(1+\alpha)$.

16.3. **Hashing with open addressing.** In hashing with open addressing the buckets are "linearized", i.e. just laid out in the table itself: inserts and searches hash and then traverse forward in the table until they find a spot. Deletion is harder so if deletion is necessary then hashing with chaining should be used. Insertion costs at most $1/(1-\alpha)$ and for $\alpha < 1$ retrieval costs

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Integral to these bounds is that α the load factor stay small. In order for the amortized analysis to workout the hash table should be doubled in size (and entries copied) when the table becomes full but halve it only when the load goes down to below 1/4.

Hashing with open addressing

```
1  HashInsert(H, k, v)
2  # H is hash table, k is key, v is value
3      m = len(H)
4      hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
5      hsh_orig = hsh
6      if H[hsh] == NIL:
7          H[hsh] = (k, v)
8      else:
9          hsh = hsh + 1
10         while H[hsh] ≠ NIL and hsh ≠ hsh_orig:
11             # mod so it swings back around and +1
12             # because indexing starts at 1, not 0
13             hsh = (hsh + 1  mod m) + 1
14         if H[hsh] == NIL:
15             H[hsh] = (k, v)
16         else:
17             retun "Error: Hash table full"
18
19 HashRetrieve(H, k)
20     m = len(H)
21     hsh = ⌊m (k (√5 − 1) /2  mod 1)⌋
22     hsh_orig = hsh
23     if H[hsh][1] == k:
24         return H[hsh][2]
25     else:
26         hsh = hsh + 1
27         while H[hsh][1] == k and H[hsh] ≠ NIL and hsh ≠ hsh_orig:
28             # mod so it swings back around and +1
29             # because indexing starts at 1, not 0
30             hsh = (hsh + 1  mod m) + 1
31         if H[hsh] == NIL or hsh == hsh_orig:
32             return "Error: key missing"
33         else:
34             return H[hsh][2]
```

17. Binary Search Tree

A binary tree is a graph where each vertex has at most two children. A binary search tree is a tree with the further constraint that the key of a parent is greater or equal to any of the keys in its left subtree and less than or equal to any of the keys in its right subtree.

The working low-level data structure for trees is dict().

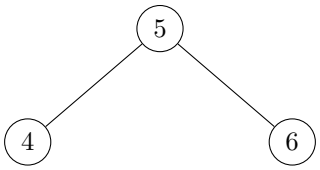
Binary Tree

```
btree = lambda parent, name, val, lchild, rchild:
            {'parent':parent, 'name':name, 'val':val,
             'lchild':lchild, 'rchild':rchild}

# counter generator is for labeling
def counter(x):
    start = x
    while True:
         yield start
         start += 1
c = counter(1)

root = btree(None,next(c),5,None,None)
left = btree(root,next(c),4,None,None)
right = btree(root,next(c),6,None,None)
root['lchild'] = left
root['rchild'] = right
```

Note that the `name is purely` a label and has no relation to `val`. The tree then looks like



17.1. **Inserting into a binary search tree.** Inserting into a binary search tree is easy: the insert vertex just has to obey the binary search constraint: start at the root, if the root value is equal to the key you're inserting then go left, otherwise go right. Once you hit a `None` create a new vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Insert(B, k)
# B is a btree dict as described above, corresponding to the
# root of the tree
prnt = ptr = B
while ptr ≠ None:
    # because of python's memory model we need to keep track
    # of parent since names are references not pointers, i.e.
    # you can't reassign pointers like in C
    prnt = ptr
    if ptr['val'] ≤ k:
        ptr = ptr['lchild']
    else:
        ptr = ptr['rchild']
if prnt['val'] ≤ k
    prnt['lchild'] = btree(prnt, next(c), k, None, None)
    return prnt['lchild']
else:
    prnt['rchild'] = btree(prnt, next(c), k, None, None)
    return prnt['rchild']
```

17.2. **Searching a binary search tree.** Searching is easy because a binary search tree obey the binary search constraint: start at the root, if the root value is equal to the key you're searching for then you're done, otherwise if the key is less than the value go left, otherwise go right. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Search(B, k)
ptr = B
while ptr ≠ None and ptr['val'] ≠ k:
    if ptr['val'] < k:
        ptr = ptr['lchild']
    else:
        ptr = ptr['rchild']
if ptr == None:
    return "Error: key missing"
else:
    return ptr
```

17.3. **Binary search tree min/max.** The minimum of a binary tree is the left-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Min(B)
1 ptr = B
2 while ptr['lchild'] ≠ None:
3     ptr = ptr['lchild']
4 return ptr
```

The maximum of a binary tree is the right-est most vertex. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Max(B)
1 ptr = B
2 while ptr['rchild'] ≠ None:
3     ptr = ptr['rchild']
4 return ptr
```

17.4. **Binary search tree predecessor/successor.** The predecessor of a vertex the maximum of a vertex's left subtree. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Predecessor(B, k)
1 ptr = Binary-Tree-Search(B, k)
2 return Binary-Tree-Max(ptr['lchild'])
```

The successor of a vertex the minimum of a vertex's right subtree. Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Predecessor(B, k)
1 ptr = Binary-Tree-Search(B, k)
2 return Binary-Tree-Min(ptr['rchild'])
```

17.5. **Deleting from a binary search tree.** We need an auxiliary function to wrap up some code that's re-used. Running time is constant.

```
Transplant(u, v)
1  # does not handle case where u is root of tree
2  prnt = u['parent']
3  if prnt['lchild'] == u
4      prnt['lchild'] = v
5  else:
6      prnt['rchild'] = v
7  if v ≠None:
8      v['parent'] = prnt
```

Deleting from a binary search tree is a little more complicated. Since the binary search tree property needs to be always preserved it's unclear what to replace a deleted vertex with. A child? A parent? In fact it should be the successor (or predecessor). The successor is the vertex whose value would follow the vertex you're trying to delete if you listed all the vertices in order. How do you find the successor? It's the minimum of the right subtree or the vertex (and the minimum of a tree is the farthest left of the tree). Then that minimum can be replaced by it's right child (it has no left child). Running time is $O(\lg n)$ if the tree is balanced.

```
Binary-Tree-Delete(B, k)
ptr = Binary-Tree-Search(B, k)
# trivial case, successor is parent
if ptr['rchild'] == None:
    Transplant(ptr, ptr['lchild'])
elif ptr['lchild'] == None:
    Transplant(ptr, ptr['rchild'])
else:
    succ = Binary-Tree-Successor(ptr)
    # if the successor is the right child of ptr then
    # then right child has no left child and task simple
    if succ == ptr['rchild']:
        Transplant(ptr, succ)
        succ['lchild'] = ptr['lchild']
        succ['lchild']['parent'] = succ
    else: # otherwise we have to fix successor subtrees and do the same thing
          # including fixing the right child
        # fix successor
        Transplant(succ, succ['rchild'])
        # don't lose right child of ptr
        succ['rchild'] = ptr['rchild']
        succ['rchild']['parent'] = succ
        # move successor into ptr's position
        Transplant(ptr, succ)
        succ['lchild'] = ptr['lchild']
        succ['lchild']['parent'] = succ
```

17.6. **Pre-order/In-order/Post-order traversal.** A Pre-order/In-order/Post-order traversal of a binary tree is a traversal the manipulates the vertex either before left and right children, after the left child but before the right child, and after both the left and right children. The easiest way to implement any of these is recursion but iterative versions do exist. Running time is $O(n)$ since the traversal visits every vertex. For illustrative purposes we simply print the `val` attribute, but any operation on the vertex could be performed.

```
Pre-order-traversal(B)
print(B['val'])
Pre-order-traversal(B['lchild'])
Pre-order-traversal(B['rchild'])
```

```
In-order-traversal(B)
1  Pre-order-traversal(B['lchild'])
2  print(B['val'])
3  Pre-order-traversal(B['rchild'])
```

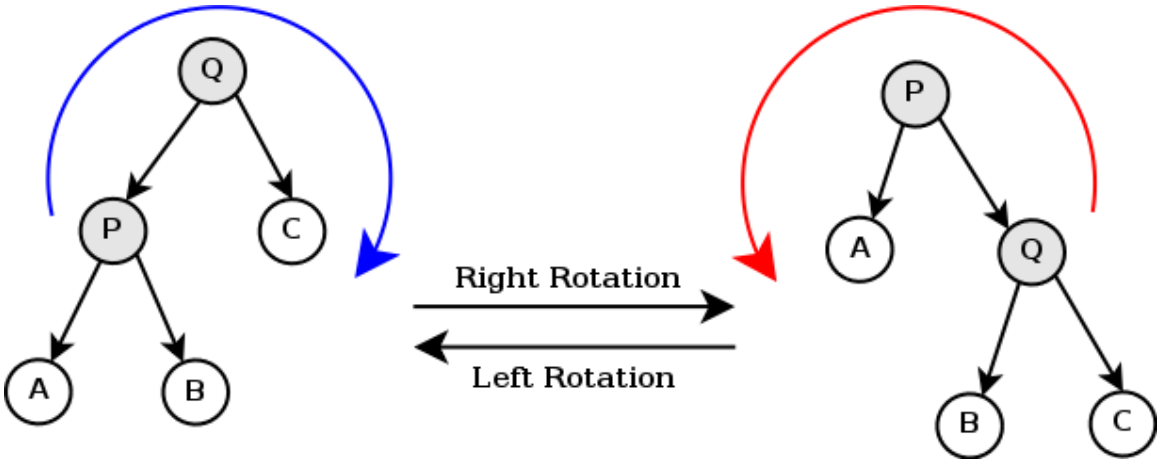
```
Post-order-traversal(B)
1  Pre-order-traversal(B['lchild'])
2  Pre-order-traversal(B['rchild'])
3  print(B['val'])
```

18. Treap

Binary trees have $O(\lg n)$ queries and inserts and deletions if they're balanced. Turns out keep them balanced is tough - a ton of schemes exist. The simpliest is a random binary tree using a treap. A treap combines the invariants of a binary tree *and* and a heap. There are two sets of attributes: priorities and keys. The priorities obey the heap property (children have smaller priority than their parents) and the keys obey the binary search property. In order to get a balanced binary tree, which is the value of treaps, we randomly generate a priority key. This then simulates the generation of a random binary tree which on average has depth $O(\lg n)$. We use a min heap.

18.1. **Treap search.** Just like for binary search tree and hence omitted.

18.2. **Treap insert.** This is easier of the two operations. First we need two auxiliary functions `Left-Rotate` and `Right-Rotate`. The easiest way to remember these is pictures



```
Left-rotate(p)
1  p_prnt = p['parent']
2  a = p['lchild']
3  q = p['rchild']
4  # put q in p's position
5  if p = p_prnt['lchild']:
6      p_prnt['lchild'] = q
7  else:
8      p_prnt['rchild'] = q
9  p['rchild'] = q['lchild']
10 q['lchild'] = p
```

```
Right-rotate(p)
1  q_prnt = q['parent']
2  p = q['lchild']
3  c = q['rchild']
4  # put p in q's position
5  if q = q_prnt['lchild']:
6      q_prnt['lchild'] = p
7  else:
8      q_prnt['rchild'] = p
9  q['lchild'] = p['rchild']
10 p['rchild'] = q
```

To insert into a treap, generate a random priority, and insert the key as if it were a binary search tree (i.e. at the bottom), then rotate up until the heap property is restored.

```
Treap-Insert(T, k)
1  # T is a binary tree that's a treap
2  u = Random(0, 1)  # both ends inclusive
3  ptr = Binary-Tree-Insert(T, (u, k))
4  prnt = ptr['parent']
5  while prnt ≠ None and ptr['val'] < prnt['val']:
6      if ptr == prnt['lchild]:
7          Right-Rotate(prnt)
8      else:
9          Left-Rotate(prnt)
10     ptr = prnt
11     prnt = ptr['parent']
```

18.3. **Treap delete.** To delete a vertex rotate it down until it's a leaf node and then delete the leaf node. Rotate down according to which of the vertex's children have a higher priority: if the left child has a higher priority than the right then rotate right, otherwise rotate left.

```
Treap-Delete(T, k)
# T is a binary tree that's a treap

ptr = Binary-Tree-Search(T, k)
while ptr['lchild'] ≠ None or ptr['lchild'] ≠ None:
    if ptr['lchild'] ≠ None and ptr['lchild']['val'] > ptr['rchild']['val']
        Right-Rotate(ptr)
    else:
        Left-Rotate(ptr)
    if ptr = ptr['parent']['lchild']:
        ptr['parent']['lchild'] = None
    else:
        ptr['parent']['rchild'] = None
    del ptr
```

19. Cartesian Tree

Given a sequence of **distinct** numbers (or any totally ordered objects), there exists a binary min-heap whose inorder traversal is that sequence. This is known as the Cartesian tree for that sequence. A min-treap is an easy way to construct a Cartiesian tree of a sorted sequence. Why? Obviously: it's is heap ordered since it obeys the min heap property and an in order traversal reproduces the sequence in sorted order. How to construct a Cartesian tree for an arbitrary sequence $A = [a_1, \ldots, a_n]$? Process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value x, start at the node representing the value prior to x in the sequence and follow the path from this node to the root of the tree until finding a value y smaller than x. This node y is the parent of x, and the previous right child of y becomes the new left child of x. Running time is $O(n)$.

```
Cartesian-Tree(A)
T = btree(None, next(c), A[1], None, None)
ptr = prnt = T
for i = 2 : len(A):
    while prnt['parent'] ≠ None and A[i] < prnt['val']
        prnt = prnt['parent']
    if prnt == None: # then we're at the root
        # and A[i] is the smallest value we've seen so far
        ptr = btree(None, next(c), A[i], prnt, None)
        prnt['parent'] = ptr
    else:
        ptr = btree(prnt, next(c), A[i], prnt['rchild'], None)
        prnt['rchild'] = ptr
    prnt = ptr
```

20. Interval Trees

An interval tree is built atop your favotire balanced binary tree data structure (treap in our case) and stores left endpoints as key. It also keeps track of maximum right endpoint in the subtree rooted at a vertex. It supports interval intersection tests (very useful). Maintaining the max in insertion and deletion is straightforward during rotations.

20.1. **Interval search.** Interval search works by being optimistic: $i = [a, b]$ and $j = [x, y]$ two intervals overlap if either $a \leq x \leq b$ or $x \leq a \leq y$. Therefore at each interval we test for overlap and whether $x \leq a \leq y$ where y is the maximum right endpoint for any interval in the left subtree. If so we go left. If in fact $y < a$ then no interval in the left subtree could possibly intersect so we go right.

```
Interval-Tree-Search(T, i)
# T is an interval tree, i = [a, b]
a, b = i[1], i[2]
# j['left'] is left endpoint of interval and
# j['right'] is right endpoint
intersect = lamba j: a ≤ j['left'] ≤ b or j['left'] ≤ a ≤ j['right']
# 'int' is interval associated with vertex
while T ≠ None and not intersect(T['int']):
    if T['lchild'] ≠ None and a ≤ T['lchild']['max']:
        T = T['lchild']
    else:
        T = T['rchild']
return T
```

21. Order Statistics Tree

Order statistics trees are probably the simplest thing to build atop a balanced binary search tree. The only extra extra piece of information each vertex stores is the attribute `size` where $x\,['\texttt{size}'] = x\,['\texttt{lchild}']\,['\texttt{size}'] + x\,['\texttt{rchild}']\,['\texttt{size}'] + 1$.

21.1. **Select.** Finding the ith ordered element in the tree works just like Quickselect

```
OS-Select(T, i)
1  # T is an interval tree, i is the rank we're looking for
2  r = T['lchild']['size'] + 1
3  if i == r:
4      return x
5  elif i < r:
6      return OS-Select(T['lchild'], i)
7  else:
8      return OS-Select(T['rchild'], i − r)
```

21.2. **Rank.** We can find the rank of an element by finding how many elements are to its left.

```
OS-Rank(T, x)
1  x is a pointer to a vertex in the tree
2  r = x['lchild']['size'] + 1
3  prnt = x
4  # while prnt is not root of T
5  while prnt ≠ T:
6      if prnt == prnt['parent']['rchild']:
7          r = r + prnt['parent']['lchild']['size'] + 1
8      prnt = prnt['parent']
9  return r
```

21.3. **Maintenance.** Maintaining `size` is easy: for example in `Left-Rotate` add lines

```
13 y['size'] = x['size']
14 x['size'] = x['lchild']['size'] + x['rchild']['size'] + 1
```

and similarly for `Right-Rotate`.

22. Union-Find

A union-find data structure is a data structure suited for taking unions and finding members (duh). The particular units of the data structures are sets (not hash table derivatives), each with a representative. The data structure is very thin, basically a wrapper for the primitive data, except for a pointer to the representative of the set and two heuristics that speed up the operations. The path compression heuristic "compresses" the path to representative of the set by setting it to be equal to that representative (which it might not be after a union). The weighted union heuristic makes it so that the smaller of the two sets unioned is the one whose representative pointers need to be updated.

Amortized complexity of n `Make-Set`, `Find-Set, Union,` operations where m are `Make-Set` is $O(m\alpha(n))$ where $\alpha(n)$ is the Ackermann function and $\alpha(n) \leq 4$ for any realistic application.

```
Make-Set(x)
1  return {'val':x,'rep':x,'rank':0}
```

22.1. Make set.

22.2. Find set.

Find set is interesting: it unwinds the stack in order to reset all the representatives from x to the representative of the set.

```
Find-Set(x)
1  if x['rep'] ≠ x:
2      x['rep'] = Find-Set(x['rep'])
```

22.3. Union.

Find set is interesting: it unwinds the stack in order to reset all the representatives from x to the representative of the set. Running time is $O(m \lg n)$.

```
Union(x, y)
1  x_rep = Find-Set(x)
2  y_rep = Find-Set(y)
3  if x_rep['rank'] > y_rep['rank']
4      y_rep['rep'] = x_rep
5  else:
6      x_rep['rep'] = y_rep
7      if x_rep['rank'] == y_rep['rank']
8          # it's an approximate rank.
9          y_rep['rank'] = y_rep['rank'] + 1
```

23. Euler circuit

An Euler circuit visits each vertex in graph twice - once going past it and once coming back across it. How do you print out an Euler circuit of a tree? Use a modified depth first traversal.

```
Euler-Circuit(u)
# u is a vertex with children
# print it going past
print(u)
for v in u['children']:
    Euler-Circuit(v)
    # print is coming back
    print(u)
```

24. Tarjan's Least Common Ancestor

The least common ancestor w of two vertices u, v in a tree is the ancestor common to both that's of greatest depth. The algorithm is useful for range-minimum querying. It uses the same traversal as `Euler-Circuit` and the Union-Find data structure augmented with a property `ancestor`. The algorithm proceeds by growing "bottom up" sets corresponding to subtrees whose roots are the least common ancestors of any pair of vertices in the tree **which have been completely traversed by the Euler circuit**. Let P be a global with the set of vertices you're interested in finding least common ancestor of and initialize all vertices to have `color` Blue in order to represent unfinishined (i.e. not completely traversed by the Euler circuit).

```
Least-Common-Ancestor(u)
# u is the root of a tree
u_set = Make-Set(u)
#this is the Euler-Circuit transformation (equivalent of print)
u_set['ancestor'] = u
for v in u['children']:
    Least-Common-Ancestor(v)
    # let's pretend there's a big table where i can fetch v_set from
    Union(u_set, v_set)
    u_set['ancestor'] = u
# u_set['val'] = u
u_set['val']['color'] = Red
for each v such that {u, v} ∈ P:
    if v['color'] == Red:
        print("Least common ancestor of {u, v} is " + v_set['ancestor'])
```

25. Range Minimum Queries

Given a sequence of distinct values and a subsequence (specified by it's end points) what is the minimum value of the in that subsequences? It's just the least common ancestor of the end points in the cartesian tree representing the sequence.

Part 4. Advanced Design Techniques

26. Dynamic Programming

Note this is a fairly formal explanation of dynamic programming. Skip if you're not interested in formalism. If you want a good intuitive interpretation of Dynamic programming read chapter 6 of Vazirani (where he says that dynamic programming is simply traversing the topological sort of the problem dependency graph).

Dynamic programming does not means writing code dynamically or changing code dynamically or anything like that. The sense in which programming is being used here is the same sense as setting a program for an festival or something, and dynamic means making decisions in-situ rather than a priori. dynamic programming is an optimization technique - minimizing travel time, minimizing cost, maximizing profits, maximizing utility, et cetera. The central concept is the **Bellman equation** so I'm going to crib here the wikipedia article on the Bellman equation (yes the whole thing).

First, any optimization problem has some objective – minimizing travel time, minimizing cost, maximizing profits, maximizing utility, et cetera. The function that describes this objective is called the **objective function** or **cost function**,i.e. the travel time, cost, or profits as a function of time. Dynamic programming breaks a multi-period planning problem into simpler steps. Therefore, it requires keeping track of how the circumstances, as the concern the decision at each step, change over time. The information about the current situation which is needed to make a correct decision is called the "state". For example, to decide how much to consume and spend at each point in time, people would need to know (among other things) their initial wealth. Therefore, current wealth would be one of their **state variables**. The decisions made at each step are represented by **control variables**. For example, in the simplest case, today's wealth (the state) and how much is consumed (the control) determine tomorrow's wealth (the new state).

The dynamic programming approach describes the optimal plan by finding a rule that tells what the controls should be, given any possible value of the state. For example, if consumption c depends only on wealth W, we would seek a rule $c(W)$ that gives consumption as a function of wealth. Such a rule, determining the controls as a function of the states, is called a **policy function**.

Finally, by definition, the optimal decision rule is the one that achieves the best possible value of the objective. For example, if someone chooses consumption, given wealth (wealth is fixed), in order to maximize happiness (assuming happiness H can be represented by a function, such as a **utility function**), then each level of wealth will be associated with some highest possible level of happiness, $H(W)$. The best possible value of the objective, written as a function of the state, is called the **value function**.

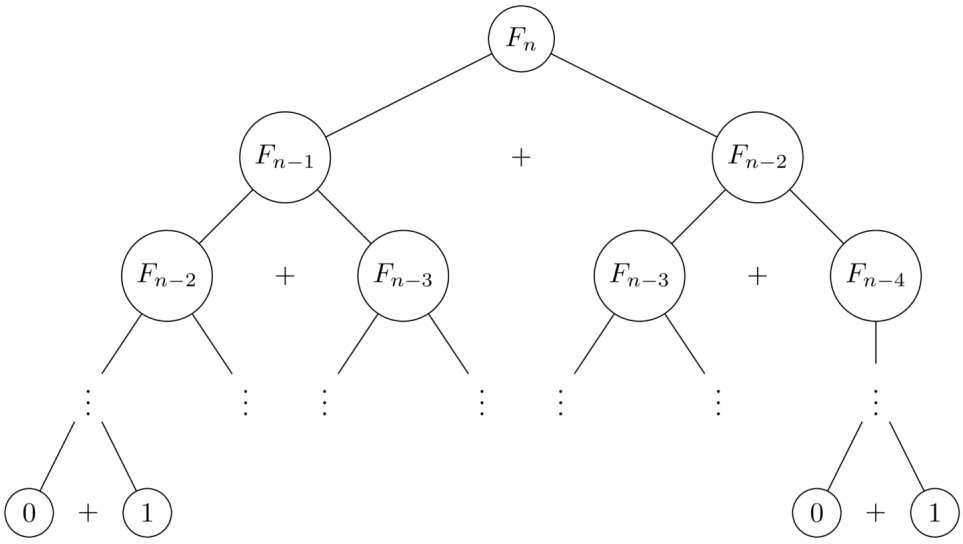
I have no idea what this value function is and how it is distinct from the objective function (but I guess I'll find out).

Richard Bellman showed that a dynamic optimization problem in discrete time can be stated in a recursive, step-by-step form known as **backward induction** by writing down the relationship between the value function[4] in one period and the value function in the next period. The relationship between these two value functions is called the "Bellman equation". In this approach, the optimal policy in the final time period is specified in advance as a function of the state variable's value at that time, and the resulting optimal value of the objective function is thus expressed in terms of that value of the state variable. Next, the next-to-final period's optimization involves maximizing the sum of that period's period-specific objective function and the optimal value of the future objective function, giving that period's optimal policy contingent upon the value of the state variable as of the next-to-last period decision. This logic continues recursively back in time, until the first period decision rule is derived, as a function of the initial state variable value, by optimizing the sum of the first-period-specific objective function and the value of the second period's value function, which gives the value for all the future periods. Thus, each period's decision is made by explicitly acknowledging that all future decisions will be optimally made.

26.1. **Fibonacci Sequence.** The simplest dynamic programming algorithm is computing the nth Fibonacci number faster than using the naive recursive definition

$$F_n = F_{n-1} + F_{n-2}$$

What's the idea? Overlapping substructure: look at the "call stack" for trying to compute F_n



Look at all of the repeated calls to F_j. Rewrite the calculation so that there's no redundant work. There are two ways to do this: using a hashtable to memoize calls or do the computation bottom up. Running time is $O(n)$.

[4]This has gotta mean cost function,or imply that the value function is the cost function.

```
Fibonacci(n)
1  if n == 1 or n == 2:
2      return 1
3  else:
4      F_{k-1} = 1
5      F_k = 1
6      k = 2
7      while k < n:
8          # k is not a parameter, subscripts are labels not indices
9          F_{k+1} = F_k + F_{k-1}
10         F_k = F_{k+1}
11         F_{k-1} = F_k
12         k = k + 1
13 return F_{k+1}
```

26.2. **Rod Cutting.** Given a rod of length n and a table of price $P = [p_1, \ldots, p_n]$ corresponding to cuts at i units of length what's the maximum value r_n obtained by cutting up the rod? The naive solution is to try all 2^{n-1} partitions of the rod. Clearly not efficient. The optimal substructure of the problem is such the maximum value is the sum of maximum value of the potential partitions around a particular cut, and not cutting at all. Therefore the Bellman equation is

$$r_i = \max\{p_i, r_1 + r_{i-1}, r_1 + r_{i-1}, r_2 + r_{i-2}, \ldots, r_{i-1} + r_1\} = \max_{j=1,\ldots,i}\{p_i, r_j + r_{i-j}\}$$

Notice the duplication in the first expression, which corresponds to mirror symmetry of the rod (turning the rod around maps a cut $r_j + r_{i-j}$ to $r_{i-j} + r_j$). Therefore the less redundant Bellman equation is

$$r_i = \max_{j<i-j}\{p_i, r_j + r_{i-j}\}$$

An equivalent formulation (all cuts of this form correspond to all cuts of the prior form) is one solid piece of length i and a potentially further subdivided piece of length $n - i$. Therefore the Bellman equation is (with the $r_0 \equiv 0$)

$$r_i = \max_{j=1,\ldots,i}\{p_j + r_{i-j}\}$$

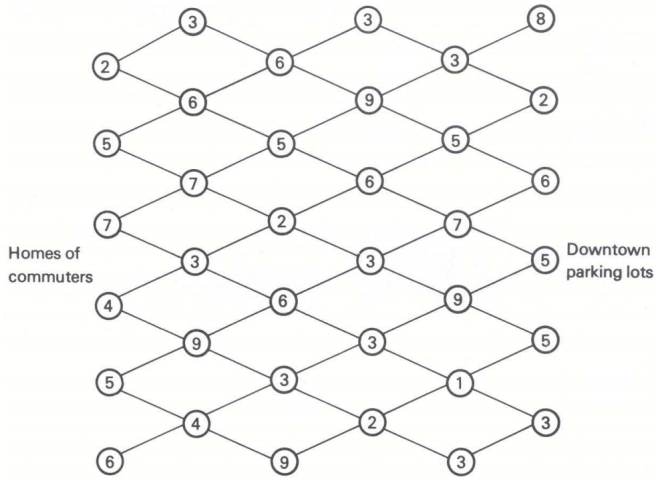
The naive recursive implementation runs in $O(2^n)$ because it recomputes solutions to subproblems several times unnecessarily.

```
Rod-Cut-Rec(P, n)
1  if n == 0:
2      return 0
3  r = -∞
4  for i = 1 : n:
5      r = max{r, P[i] + Rod-Cut-Rec(P, n - i)}
6  return r
```

To speed this up you need to "memoize" redundant calls (hash table with already computed values) *or* compute values bottom up. The more elegant solution solution is the bottom up computation.

```
Bottom-Up-Rod-Cut(P, n)
1  r = n · [0]
2  for i = 1 : n:
3      q = 0
4      for j = 1 : i:
5          # recall r[0] ≡ 0
6          q = max{q, P[i] + r[i − j]}
7      r[i] = q
8  return r[n]
```

26.3. **Getting to work**[5]**.** Given a neighborhood of n commuters and n downtown parking lots what is the fastest way for each commuter to get to work given that intersection have delays?



Imagine instersections are on a rectified grid and that the cost (in time) of getting to intersection i, j from some house is $q(i,j)$ and $c(i,j)$ is the time to get between intersection (left to right - one way streets). Seems like you'd have to start at the end (parking lots) and work backwards right? But no in fact this problem has optimal substructure

$$q(i,j) = \begin{cases} \infty & j < 1 \text{ or } j > n \\ c(i,j) & i = 1 \\ \min\{q(i-1,j-1), q(i-1,j+1)\} + c(i,j) & \text{otherwise} \end{cases}$$

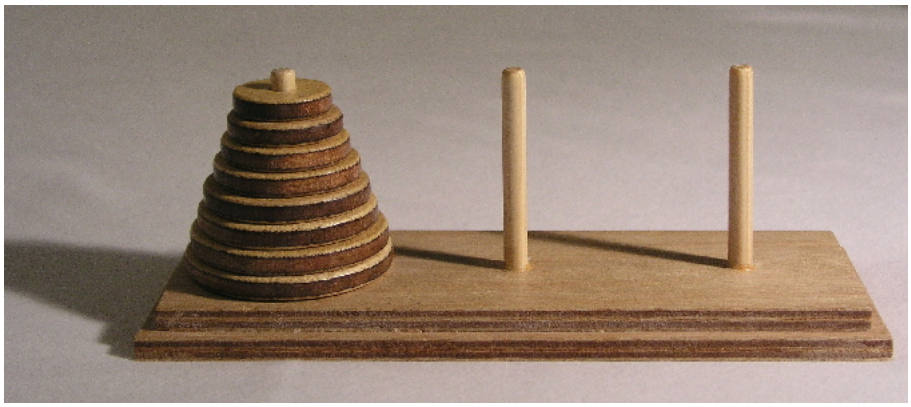
Starting from any particular household $(1, j)$ we can compute the shortest path cost to any parking lot by forward iteration. Running time is $O(nk)$.

[5]This example is Applied Mathematical Programming by Bradley,Hax, Magnanti chapter 11.

```
Shortest-Path-Parking(A, c)
# A is an n × k grid (or something like that)
# and c is the associated costs with getting
# to the intersection
n = len(A)
q = n · [n · [∞]]
for i = 1 : n:
    # initialize first hop intersections
    q[1, i] = c[1, i]
for i = 2 : n:
    for j = 1 : n:
        # i'll let you figure out how to
        # bumper the q array so that
        # when i, j = 1 the entries
        # q[i − 1, j − 1] = ∞
        q[i, j] = min{q[i − 1, j − 1], q[i − 1, j + 1]} + c[i, j]
return q
```

In order to return the actual path to any parking garage just modify the `for` loop to keep track of which of $q(i-1, j-1)$ or $q(i-1, j+1)$ was chosen in the min.

26.4. **Towers of Hanoi.** Move all the disks from the left rod to the far right one only one disk (top disk) at a time.



This problem has optimal substructure in that there's no real difference between any of the rods or disks: so moving n disks from the first rod to the third involves moving $n-1$ disks (from some rod) to the third rod. The solution is purely recursive: let $S(n, h, t)$ be the solution to moving n disks from their "home" rod h to a target rod t. Then

$$S(1, h, t) = \text{just move the disk}$$

and

$$\begin{aligned} S(n,h,t) &= \text{first } S(n-1,h,\text{not}(h,t)) \\ &\quad \text{second } S(1,h,t) \\ &\quad \text{third } S(n-1,\text{not}(h,t),t) \end{aligned}$$

Running time is $O(2^n)$.

26.5. **Egg drop.**

2 eggs 100 floors. Suppose you have 2 eggs and a 100 story building and we want to find out the highest floor *an* can be dropped from safely. The assumptions are

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks when dropped from some floor, then it will break if dropped from higher floors.
- If an egg survives a fall, then it would survive a fall from a lower floor.

Starting from the 14th floor is the best strategy because the number of attempts (in the worst case) is always 14. Why? If the first egg breaks at the 14th floor then you have to check floors 1 through 13 with the second egg for a total of 14 floors. If the egg doesn't break then move to the 27th floor. If it breaks then you have to check floors 15 through 26 for a total of $13+1=14$. Then move to the 39th floor and etc. The sequence of floors is $14, 27, 39, 50, 60, 69, 77, 84, 90, 95, 99, 100$. If the egg breaks at any point before getting to floor 100 you have to test 13 more floors. If it doesn't break until the 100th floor then you have to perform only 12 drops.

2 eggs k floors. Suppose that for the best strategy, the number of drops in the worst case is x. Then, you should start at the xth floor. If the first egg breaks then you have $x-1$ floors to check for a total of $1+(x-1)=x$ floors to check. If it doesn't break then you should check $x+(x-1)$th floor. If it breaks then with the second egg you have to check floors $x+1, x+2, \ldots, (x+(x-1)-1)$, for a total of

$$(x+(x-1)-1)-(x+1)+1 = x-2$$

plus the drop at floor x and floor $x+(x-1)$ makes for a total of x drops.

What's actually happening is assuming that using the best strategy the minimum number of drops is x we are searching for the best strategy that covers all of the floors. Suppose the minimum number of attempts, in the worst case, while using the best strategy is x. How many floors can we cover?

$$x+(x-1)+(x-2)+\cdots+2+1 = \frac{x(x+1)}{2}$$

So we need

$$\frac{x(x+1)}{2} \geq k$$

or

$$x = \left\lceil \frac{-1+\sqrt{1+8k}}{2} \right\rceil$$

which for $k = 100$ implies $x = 14$.

N eggs, k floors. Suppose you have n eggs, h consecutive floors to be tested, and you drop an egg at floor i in this sequence of h floors. If the egg breaks then the problem reduces to $n-1$ eggs and $i-1$ remaining floors. If the egg doesn't break then the problem reduces to n eggs and $h-i$ remaining floors. This is the optimal substructure of the problem: the floors we want to test are irrelevant, only their quantity. Let $W(n,h)$ be the minimum number of drops required to find the threshold floor in the worst case, while using the best strategy. Then

$$W(n,h) = 1 + \min_{i=1,\ldots,h} \left(\max\left\{W(n-1,i-1), W(n,h-i)\right\}\right)$$

If you have only one egg then the minimum number of tests using the best strategy (the one that potentially covers all the floors), if the threshold floor, is the top one is h. So $W(1,h) = h$. If there's only 1 floor we only need 1 egg so $W(n,1) = 1$, and if there are no floors then we need 0 eggs so $W(n,0) = 0$. Running time is $O\left(nh^2\right)$ because of the min over $i = 1,\ldots,h$. Since $W(n-1,i-1)$ is increasing in i and $W(n,h-i)$ is decreasing in i a local min of $g(i) = \max\left\{W(n-1,i-1), W(n,h-i)\right\}$ is a global min and so you can use binary search so speed the min loop to get a running time of $O(nh\log h)$. But there's an even faster solution. Recall that

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

and

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Let $f(d,n)$ be the number of floors we can cover using n eggs and with d remaining drops. If the egg breaks we will be able to cover $f(d-1,n-1)$ floors and otherwise $f(d-1,n)$. Hence

$$f(d,n) = 1 + f(d-1,n-1) + f(d-1,n)$$

Solving for f solves the problem. Let $g(d,n) = f(d,n+1) - f(d,n)$ (an auxiliary function). Then substituting in the recurrence relation for $f(d,n)$ we get

$$g(d,n) = g(d-1,n) + g(d-1,n-1)$$

which is the recurrence relation for the binomial coefficient, and so it seems that $g(d,n) = \binom{d}{n}$. Problem is $f(0,n) = 0$ for all n and so should $g(0,n)$ but $g(0,0) = \binom{0}{0} = 1$. Defining $g(d,n) = \binom{d}{n+1}$ the recursion is still satisfied and no contradictions.

Now to solve the problme: using a telescoping sum for $f(d,n)$

$$\begin{aligned} f(d,n) &= [f(d,n) - f(d,n-1)] \\ &+ [f(d,n-1) - f(d,n-2)] \\ &\vdots \\ &+ [f(d,1) - f(d,0)] \\ &+ f(d,0) \end{aligned}$$

where $f(d,0) = 0$ we get that

$$\begin{aligned} f(d,n) &= g(d,n-1) + g(d,n-2) + \cdots + g(d,0) \\ &= \binom{d}{n} + \binom{d}{n-1} + \cdots + \binom{d}{1} \\ &= \sum_{i=1}^{n} \binom{d}{i} \end{aligned}$$

So we just have to find d such that

$$\sum_{i=1}^{N} \binom{d}{i} \geq k$$

which can be done in linear time using the relation

$$\binom{a}{b+1} = \binom{a}{b} \frac{a-b}{b+1}$$

26.6. **Maximum Positive Subarray/Kidane's algorithm.** Given $A = [a_1, \ldots, a_n]$, how to find the subarray with the maximum positive sum? Use dynamic programming solution called Kidane's algorithm. Change the problem to look at maximum sum subarray ending at some j. Maximum sum subarray ending at j is either empty, i.e. has negative sum, in which case its sum is 0, or includes $A[j]$. The maximum sum subarray in all of A is the maximum of all subarrays ending at all j. Running time is $\Theta(n)$.

`Kidane-Max-Subarray`(A)

```
1  # m_ is max
2  m_here = m_all = A[1]
3  for i = 2 : len(A):
4      m_here = max(0, m_here + A[i])
5      m_all = max(m_all, m_here)
6  return m_all
```

Note that if at $j-1$ the subarray was empty, and hence $m_{here} = 0$ then at j it's the case that $m_{here} = A[j]$. In order to recover the actual subarray you need to keep track of whether counting is reset or subarray is extended. Easiest way to do this is using Python tricks. In general this is calling keeping "back-pointers" and works in all such cases for reconstructing the solution (forthwith omitted).

```
Kidane-Max-Subarray-Mod(A)
m_here = m_all = [[ ], A[1]]
for i = 2 : len(A):
    # take max wrt. first entry of arguments, i.e. max(0, m_here + A[i])
    m_here = max([0, [ ]], [m_here + A[i], m_here.append(A[i])], key=itemgetter(1))
    m_all = max(m_all, m_here, key=itemgetter(1))
return m_all
```

26.7. **Longest increasing subsequence.** A subsequence of a sequence $A = [a_1, a_2, \ldots, a_n]$ need not be contiguous. Just like in Kidane's algorithm you should be looking at subsequences ending at some index i: let $L[i]$ be the longest strictly increasing subsequence ending at index i. What's the "optimal" way to obtain $L[i]$? Extend some smaller optimal subsequence ending at index j. But when can you extend some subsequence ending at position j? Only when $A[j] < A[i]$ since it should be an increasing subsequence! Running time is $O(n^2)$.

```
LIS(A)
n = len(A)
L = n · [∞]
for i = 1 : n:
    L[i] = 1
    for j = 1 : i − 1:
        if A[j] < A[i]:
            L[i] = max{L[i], 1 + L[j]}
return max{L}
```

26.8. **Box stacking**[6]**.** You have n boxes $B = [b_1, \ldots, b_n]$ with dimensions height h_i, width w_i, and depth d_i. What's the tallest stack of boxes you can make? A box b_i can be stacked atop another box b_j if b_i can oriented such that one of its faces is smaller than the upwarding face of b_j. To simplify the problem simply "replicate" the boxes such that one box with dimensions h_i, w_i, d_i corresponds to 3 boxes

$$\begin{aligned} h_i, w_i, d_i &= h_i, w_i, d_i \\ h_i^{'}, w_i^{'}, d_i^{'} &= w_i, d_i, h_i \\ h_i^{''}, w_i^{''}, d_i^{''} &= d_i, h_i, w_i \end{aligned}$$

where without loss of generality (i.e. fix an orientation of the base $w_i \times d_i$) we require $w_i \leq d_i$. For example if we have a box of dimension $1 \times 2 \times 3$ then really we have 3 boxes

$$\begin{aligned} h_i, w_i, d_i &= 1, 2, 3 \\ h_i^{'}, w_i^{'}, d_i^{'} &= 2, 1, 3 \\ h_i^{''}, w_i^{''}, d_i^{''} &= 3, 1, 2 \end{aligned}$$

[6]From here on down I stole these from https://people.cs.clemson.edu/~bcdean/dp_practice/

where the requirement that $w_i \leq d_i$ forces us to define $h_i^{'}, w_i^{'}, d_i^{'} = 2, 1, 3$ instead of $h_i^{'}, w_i^{'}, d_i^{'} = 2, 3, 1$ (which would be the same box). Call $w_i \times d_i$ the base of a box. So box b_i can be stacked atop b_j if the base of box b_i is smaller than the base of box b_j. This is quite similar to the longest increasing subsequence substructure except the relation is geometric rather than simple magnitude: instead of just $A[j] < A[i]$ we have that $w_i < w_j \wedge d_i < d_j$. So what's the algorithm? First sort the boxes (the $3n$ boxes) by decreasing base dimension. Why? We didn't do that for the longest increasing subsequence problem right? Well the natural ordering of the LIS is the order it's given to us in; it's in the statement of the problem that we should look for the longest increasing subsequence of the given sequence. The boxes aren't presented to us in any given order so we must impose one such that we're able to select a longest "increasing" subsequence, where we've redefined increasing. Then the rest is just like longest increasing subsequence (except for base comparison). Running time is $O\left(n^2\right)$ just like longest increasing subsequence.

```
Box-Stacking(B)
n = len(B)
# let's pretend this returns only the second entry, i.e. the b_i
B = sorted([(b['w'] × b['d'], b) for b in B], key = itemgetter(1))
L = n · [∞]
for i = 1 : n:
    L[i] = 1
    for j = 1 : i − 1:
        if B[j]['w'] < B[i]['w'] and B[j]['d'] < B[i]['d']:
            L[i] = max{L[i], 1 + L[j]}
return max{L}
```

26.9. **Bridge crossings.** You have a river crossing a state with n cities on the south bank and n corresponding cities on the north bank (not necessarily in the same order). You want to build as many bridges connecting corresponding cities as possible without building bridges that intersect. Let x_i be the index of the city on the north shore corresponding to the ith city on the south shore. You can figure this out if you're just given the two lists, i.e. integer array $S = [1, 2, \ldots, n]$ to label the southshore cities and integer array $N = [\sigma(1), \sigma(2), \ldots, \sigma(n)]$ for the permutation on the northshore, by sorting the northshore array (while keeping track which index the elements get sorted **from** - think about it and you'll understand). Then you just need to find the longest increasing subsequence of the x_i array. Why? A maximal matching with the already sorted sequence of cities on the southshore is exactly what that is - in fact this is a pretty good model of increasing subsequence period. Running time is $O\left(n^2\right)$ just like longest increasing subsequence.

26.10. **Integer Knapsack.** You're a thief with a knapsack that has a finite capacity C. You break into a store that has n items with integer sizes s_i and values v_i. Which items should you steal? You can only take whole items and you're allowed duplicates. The subproblems here are filling smaller knapsacks duh. So let $M(j)$ be the maximum value obtained by filling a knapsack with capacity exactly j. The maximum value j capacity knapsack that can be constructed is either equal to the

maximum $j-1$ capacity knapsack that can be constructed or it includes item i and all of the items in the $j-s_i$ capacity knapsack. Therefore the Bellman equation is

$$M(j) = \max\left\{M(j-1), \max_i \{M(i-1, j-s_i) + v_i\}\right\}$$

Running time is $O(nC)$ because you compute C entries but each computation considers n items.

```
Integer-Knapsack(S, V, C)
# S is an array of integer sizes, V is an array of values,
# and C is the capacity of the knapsack
M = C · [0]
for j = 1 : C:
    M[i][j] = max{M[j − 1], M[i − 1][j − S[i]] + V[i]}
return M[C]
```

26.11. **0/1 Knapsack.** In this instance you can only take whole items (that's the 0/1) and there are no duplicates. The subproblems here are the optimal value for filling a knapsack with capacity exactly j and with some subset of the items $1, \ldots, i$. $M(i, j)$ either includes items i, in which case it includes all of the items of the optimal knapsack over the items $1, \ldots, i-1$, with capacity $j - s_i$, and in which case it has value $M(i-1, j-s_i) + v_i$, or it does not include item i, in which case it has capacity j and has value $M(i-1, j)$. Hence the Bellman equation is

$$M(i, j) = \max\{M(i-1, j), M(i-1, j-s_i) + v_i\}$$

Then the solution to the whole problem is not $M(n, C)$ but $\max_j \{M(n, j)\}$ because you might not need to use the entire capacity. Running time is still $O(nC)$ because there are $n \times C$ subproblems.

```
0-1Knapsack(S, V, C)
# S is an array of integer sizes, V is an array of integer values,
# and C is the capacity of the knapsack
n = len(S)
M = n · [C · [0]]
for i = 1 : n:
    for j = 1 : C:
        M[i][j] = max{M[i − 1][j], M[i − 1][j − S[i]] + V[i]}
return max{M[n]}
```

26.12. **Balanced Partition.** You get n integers $A = [a_1, \ldots, a_n]$, each in the range $0, \ldots, k$, and the goal is to partition A into two sets S_1, S_2 minimizing $|\text{sum}(S_1) - \text{sum}(S_2)|$. This is similar to the knapsack problem (probably for the same reason that general knapsack is reducible to subset sum). The analogy is between capacity and summing to some j and items from the set A playing the role of store items. Let $P(i, j)$ be a boolean that reports whether some subset of $[a_1, \ldots, a_i]$ sum to j. Then $P(i, j) = 1$ if some subset of $[a_1, \ldots, a_{i-1}]$ sum to j, in which case we don't need

to include item i, or if some subset of $[a_1, \ldots, a_{i-1}]$ sums to $j - a_i$, in which case we include item a_i to get a subset that sums to j. Hence the Bellman equation is

$$P(i,j) = \begin{cases} 1 & \text{if } P(i-1,j) = 1 \text{ or } P(i-1, j-a_i) = 1 \\ 0 & \text{otherwise} \end{cases}$$

More succinctly

$$P(i,j) = \max\{P(i-1,j), P(i-1,j-a_i)\}$$

Note this is just a logical or, i.e. $||$. There are n^2k problems because i range from 1 to n but each a_i could have value k so j ranges from 0 to nk. How do you use this to solve the original problem? Let $S = \sum a_i/2$. Then the subset S_j such that

$$\min_{j \leq S} \left\{ S - j \middle| P(n,j) = 1 \right\}$$

produces the solution. Running time is the same $O\left(n^2k\right)$.

26.13. **Longest common subsequence.** Given two strings $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ what is the longest common subsequence? Let $Z = [z_1, \ldots, z_k]$ be such a longest common subsequence. Working backwards: if $a_n = b_m$ then $z_k = a_n = b_m$ and $[z_1, \ldots, z_{k-1}]$ is a longest common subsequence of $[a_1, \ldots, a_{n-1}]$ and $[b_1, \ldots, b_{m-1}]$. Suppose that the two sequences A and B do not end in the same symbol. Then the longest common subsequence of A and B is the longer of the two sequences $\text{LCS}([a_1, \ldots, a_n], [b_1, \ldots, b_{m-1}])$ and $\text{LCS}([a_1, \ldots, a_{n-1}], [b_1, \ldots, b_m])$. Why? Consider the two following sequences: $A = [123467]$ and $B = [23470]$. The LCS of these two sequences either ends with a 7 (the last element of sequence A) or does not. If the LCS does end with a 7 then it cannot end with 0, thus we can discard the 0 on the end of B and $\text{LCS}(A, B) = \text{LCS}(A, [2347])$. If it does not end in 7 then we can similarly discard 7 and $\text{LCS}(A, B) = \text{LCS}([123467], [23470])$. In either case we're considering either $\text{LCS}([a_1, \ldots, a_n], [b_1, \ldots, b_{m-1}])$ or $\text{LCS}([a_1, \ldots, a_{n-1}], [b_1, \ldots, b_m])$, and in fact the longest of the two. Hence the Bellman equation is

$$\text{LCS}([a_1, \ldots, a_i], [b_1, \ldots, b_j]) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{LCS}([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_{j-1}]) + 1 & \text{if } a_i = b_j \\ \max\{LCS([a_1, \ldots, a_i], [b_1, \ldots, b_{j-1}]), ([a_1, \ldots, a_{i-1}], [b_1, \ldots, b_j])\} & \text{if } a_i \neq b_j \end{cases}$$

Running time is $O(nm)$.

26.14. **Edit distance.** Given two strings $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$ what is minimum the "cost" of transforming one string into the other, where the costs associated with insertion, deletion, and replacement are C_i, C_d, C_r respectively. The subproblems here are similar to those in longest common subsequence. Let $T(i,j)$ be the minimum cost of transforming $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_j]$. There are 4 ways to transform $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_j]$: either delete a_i and transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_j]$, transform $[a_1, \ldots, a_i]$ into $[b_1, \ldots, b_{j-1}]$ then insert b_j, or replace a_i with b_j and then transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_{j-1}]$. Finally if $a_i = b_j$ then just transform $[a_1, \ldots, a_{i-1}]$ into $[b_1, \ldots, b_{j-1}]$. Therefore the Bellman equation is

$$T(i,j) = \min\{C_d + T(i-1,j), T(i,j-1) + C_i, T(i-1,j-1) + C_r, T(i-1,j-1) \text{ if } a_i = b_j\}$$

Running time is $O(nm)$.

26.15. **Counting Boolean Parenthesizations.** Given a boolean expression with n literals and $n-1$ operators how many different ways are there to parenthesize such that the expression evaluates to true. Let $T(i,j)$ be the number of ways to parenthesize literal i through j such that the subexpression evaluates to true and $F(i,j)$ to be the number of ways such that the subexpression evaluates to false. The base cases $T(i,i), F(i,i)$ are just function of the literals. Note that $i<j$ so we then seek to compute $T(i,i+1), F(i,i+1), T(i,i+2), F(i,i+2)$ for all i. How? Well $T(i,j)$ is always a function of two subexpression and the operand between them: the literals from i to k and from $k+1$ to j. For example if the operand is $\wedge$ then $T(i,j) > T(i,k)\cdot T(k+1,j)$ since the expression including the literals from i to j will be true for any values of the subexpression from i to k which evaluate to true and any values of the subexpression $k+1$ to j which evaluate to true. If the operator were $\vee$ then it would be $T(i,k)\cdot T(k+1,j)+T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j)$. And we need to sum over all possible splits k. So the Bellman equation is

$$T(i,j)=\sum_{i\le k\le j-1}\begin{cases} T(i,k)\cdot T(k+1,j) & \text{if } k\text{th operator is } \wedge \\ T(i,k)\cdot T(k+1,j)+T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j) & \text{if } k\text{th operator is } \vee \\ T(i,k)\cdot F(k+1,j)+F(i,k)\cdot T(k+1,j) & \text{if kth operator is xor} \end{cases}$$

Running time is $O\left(n^3\right)$.

26.16. **Coin game.** Given n coins layed out in a row with values $v_1,\ldots,v_n$ you play a game against an opponent where on each turn you pick up one of the two outside coins. The goal is to maximize the sum of the value of the selected coins. Let $V(i,j)$ be the maximum value you can **definitely** win if it's your turn and only the voince $v_i,\ldots,v_j$ remain. The base cases $V(i,i)$ and $V(i,i+1)$ are easily to compute. We seek to compute $V(i,i+2)$ and etc. We need to think two steps ahead to compute arbitrary $V(i,j)$: if we pick v_i then our opponent will either pick the jth coin of the $i+1$th coin. Reasoning conservatively (the opponent will pick the better) we will be presented with the minimum possible scenario of coins $i+1,\ldots,j-1$ and $i+2,\ldots,j$. If we pick v_j then similarly we will be presented with the minimum possible scenario of coins $i,\ldots,j+2$ and $i+1,\ldots,j-1$. Therefore the Bellman equation is

$$V(i,j)=\max\left\{\underbrace{\min\left\{V(i+1,j-1),V(i+2,j)\right\}+v_i}_{\text{pick } v_i},\underbrace{\min\left\{V(i,j+2),V(i+1,j-1)\right\}+v_j}_{\text{pick } v_j}\right\}$$

27. Greedy Algorithms

Greedy algorithms are like dynamic programming algorithms except there's only one subproblem. They're a lot easier to construct than DP algorithms (but take a little arguing to prove they're correct).

27.1. **Activity scheduling.** Suppose you have a set of activities $A=[a_1,\ldots,a_n]$ with sorted start times $S=[s_1,\ldots,s_n]$and sorted finish times $F=[f_1,\ldots,f_n]$. How to schedule the most non-overlapping acitivities? Let S_{ij} be the set of activities that start after a_i finishes and end before

a_j starts. Standard DP argument (partitioning around a parking activity a_k leads to the Bellman equation for $C(i,j)$ the maximal number of activities is

$$C(i,j) = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{C(i,k) + C(k,j) + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Then memoize or bottom-up it and you have an $O(n^2)$ DP algorithm. But there's an obvious greedy algorithm: always pick the job that doesn't overlap with already picked jobs and ends the soonest.

```
Greedy-Activity(A, S, F)
1  n = len(S)
2  # assume a_1 is the job that ends first
3  J = [A[1]]
4  k = 1
5  for m = 2 : n :
6      if F[k] ≤ S[m] :
7          J.append(A[m])
8          k = m
9  return A
```

27.2. **Fractional Knapsack.** This is the same as Integer Knapsack but you can take fractions of items (imagine you broke into a spice shop). The greedy strategy that optimally picks items is one that chooses items that give most bang per weight, a kind of value density: pick as much of the item that has the highest v_i/w_i until it's exhausted. Then continue on to the next most value dense item.

27.3. **Huffman codes.** What's the most optimal way to encode a message using a $\{0,1\}$ code given the distribution over the input alphabet? Letters that appear most often should have the smallest code words and conversely letters that appear rarely should have the longest code words. Using prefix-free codes (codes such that no codeword is a prefix of any other codeword) we can achieve optimal compression so without loss of generality we can use them, and we will since they make things easiest.

Given the frequency distribution C we can construct a binary tree called a Huffman tree whose traversal produces the prefix-free codes using a min-queue.

```
Huffman-Tree(C)
1  n = len(C)
2  # "cast" C to be a min-priority-queue
3  Q = minQueue(C)
4  for i = 1 : n − 1:
5      z = btree(None, next(c), None, None, None)
6      z['lchild'] = x = Extract-Min(Q)
7      z['rchild'] = y = Extract-Min(Q)
8      # 'val' is character frequency
9      z['val'] = x['val'] + y['val']
10     Insert(Q, z)
11 # return root of tree
12 return Extract-Min(Q)
```

Running time is $O(n \log n)$ due to the min-queue operations. Constructing the codes is done by performing a depth-first traversal of the Huffman tree and keeping track of lefts and rights (zeros and ones).

27.4. **Making change with unlimited coins.** Consider the problem of making change for n cents using the fewest number of coins $K = [c_1, \ldots, c_k = 1]$. Assume each coin's value is an integer. If the coins are the US quarters, dimes, nickels, and pennies then a greedy algorith is optimal: change as much for quarters as you can, then as much for dimes, etc. The greedy strategy does not always work: suppose the coins are of denomination $4¢, 3¢, 1¢$ to change $6¢$. In general you need to use dynamic programming - solution is similar to integer knapsack with duplicates. Let $C(i)$ be the optimal number of coins used to make change for $i¢$ using any of the coins. The minimum number of coins needed to change i is 1 plus $C(i - c_j)$ where c_j is the coin denomination that minimizes $C(i - c_j)$ and $c_j < i$. Therefore the Bellman equation is

$$C(i) = \min_j \left\{ C(i - c_j) \,\middle|\, c_j < i \right\} + 1$$

Running time is $O(nk)$.

27.5. **Making change with fixed coins.** Here is another solution. I don't understand why there should be another solution but here it is. Suppose the coins come sorted in decreasing order so $c_1 > c_2 > \cdots > c_k = 1$. Let $C(i, j)$ be the optimal number of coins used to make change for $i¢$ using only coins $j, \ldots, k$. We either use coin c_j or we don't. If we do not then we're solving the problem $C(i, j + 1)$. For example we might not use coin c_j if $c_j > i$. If we do use coin c_j then the rest $(i - c_j)¢$ needs to be changed, potentially using the coin j again.

$$C(i, j) = \begin{cases} C(i, j + 1) & \text{if } c_j > i \\ \min_j \{C(i, j + 1), C(i - c_j, j) + 1\} & \text{if } c_j \leq i \end{cases}$$

Running time is also $O(nk)$.

Part 5. Graph Algorithms

28. Representations of Graphs

There are two ways to represent a graph $G = (E, V)$: **adjacency matrix** and **adjacency list**.

28.1. **Adjaceny matrix.** The former is a table with $n = |V|$ rows and n columns and with an entry in row i column j if there's an edge between vertex i and vertex j. The value of the entry could be anything from simply 1 to indicate an undirected edge, -1 to represent a directed edge, k to represent an edge weight, 0 to represent no edge.

```
Adjacency-Matrix(n, E)
1  # n is the number of vertices, E is the edge list (a list of tuples)
2  # default is directed (so double edges indicate undirected)
3  mat = n · [n · [0]]
4  for u, v, w in E:
5      mat [u] [v] = w
6  return mat
```

28.2. **Adjacency list.** The latter is a list of lists where the i entry in the list is a list containing all j such that edge $(i, j) \in E$. Most algorithms in this section will use the adjacency list representation. Further more we assume that other attributes will be stored in a hash table keyed on the vertex "name", which is a number.

```
Adjacency-list(n, E)
1  # n is the number of vertices, E is the edge list (a list of tuples)
2  # default is directed (so double edges indicate undirected
3  adj = n · [[ ]]
4  for u, v, w in E:
5      adj [u] .append ((v, w))
6  return adj
```

29. Transpose

The transpose graph $G^T = \left(V, E^T\right)$ where $(u, v) \in E^T$ iff $(v, u) \in E^T$, i.e. reverse all the arrows. Computing the transpose graph when a graph is represented by an adjacency matrix amounts to just transposing the matrix. When the original graph is represented by adjacency lists it's a little more complicated but pretty obvious regardless.

30. Breadth-first Search

A bread-first search is exactly what it sounds like: all vertices at a certain breadth (distance) are visited, then the next breadth, then the next breadth, and so on. In order to repeatedly the same vertices we need to keep track of which vertices we've visited. The most elegant way is to "decorate" by constructing tuples $(i, visited)$ and unpacking. An easier way is to just have a hash table that stores that attribute. Running time is $O(V + E)$.

```
Breadth-first-search(s)
# A is an adjacency list and s is a source
# from which to start searching. 'state' = 0 means unvisited,
# = 1 means visited but not explored, = 2 means explored
attrs = {i : {'state':0, 'd':∞, 'prnt':None} for i = 1 : n}
attrs[s]['state'] = 1
attrs[s]['d'] = 0
Q = [s]
while len(Q) > 0:
    u = Q.popleft()
    for each v ∈ A[u]:
        if attrs[v]['state'] == 0:
            attrs[v]['state'] = 1
            attrs[v]['d'] = attrs[u]['d'] + 1
            attrs[v]['prnt'] = u
            Q.append(v)
    attrs[u]['state'] = 2
```

31. Depth-first Search

A depth-first search is exactly what it sounds like: go as deep as possible then back up until you can go deep again, and so on. For depth search we also keep track of what are called opening and closing times; they're useful for other algorithms like topological sort. Running time is $O(V + E)$.

```
Depth-First-Search(s)
attrs = {i : {'state':0, 'opentime': None, 'closetime': None, 'prnt':None} for i = 1 : n}
time = 0
# list of vertices visited
visited = [ ]
S = [s]
while len(S) > 0:
    time = time + 1
    u = S[−1]
    # if we come back to this point in the stack after having
    # explored the vertex that means we've visited all of its children
    # and it's done
    if attrs[u]['state'] == 2:
        attrs[u]['closetime'] == time
        S.popright()
        visited.append(u)
    # otherwise we need to leave it on the stack but explore it
    # really this is superfluous because the only way to get on the stack
    # is to concomitant with having 'state'= 1
    elif attrs[u]['state'] == 1:
        attrs[u]['opentime'] == time
        for each v ∈ A[u]:
            if attrs[v]['state'] == 0:
                attrs[v]['state'] = 1
                attrs[v]['prnt'] = u
                S.append(v)
        attrs[u]['state'] = 2
return visited
```

32. Topological Sort

A topological sort of a directed acyclic graph $G = (V, E)$ is an ordering on V such that if $(u, v) \in E$ then u appears before v in the ordering. Producing a topological sort is easy using `Depth-First-Search`: the `visited` array already returns the topological sort! The vertex at the front of the list is first in topologically sorted order, the second is the second, and so on. Why does this produce a topological sort? The intuition is that the last vertex to close is "behind" all of the others - all of the vertices are descendents of it.

If the graph is connected then some vertices might be unvisited after starting from a particular source. In that case you need to run DFS on every vertex (making sure to not to run twice on a vertex that's already been visited). Running time is $O(V + E)$.

```
Topolgoical-Sort(G)
attrs = {i : {'state':0, 'opentime': None, 'closetime': None, 'prnt':None} for i = 1 : len(G)}
time = 0
S = [ ]
# let's pretend we have a tuple G where V is a hash table {'label' : vertex} where
# vertex is a pointer to the vertex and E is a list of tuples (u, v).
E, V = G
for v ∈ V:
    if attrs[v]['state'] == 0:
        # let's pretend that we changed DFS to take in addition attrs, time
        # and return attrs, time and the list mentioned above
        attrs, time, tsort = Depth-First-Search(v, attrs, time)
        # concatenate to the front because all closing times are later
        S = tsort + S
```

33. Strongly Connected Components

A connected component of a graph $G = (V, E)$ is a subset $V' \subset V$ such that for every $u, v \in V'$ there's a path $u \rightsquigarrow v$ and $v \rightsquigarrow u$. How do you compute all of the connected components of a graph? A topological sort and DFS on the transpose graph G^T. First topological-sort all of the vertices in the graph G. Then DFS the transpose graph G^T in the topologically sorted order produced by the topological sort G. Why does this work?

Theorem 1. *Let $f(C)$ be the latest finishing time of a strongly connected component and C, C' two distinct connected components. Then $f(C) > f(C')$ and in G^T (which has the same strongly connected components as G) $f(C) < f(C')$.*

That means, by the contrapositive, that in G^T if there's an edge between two connected components then $f(C) > f(C')$. That's why we visit strongly connected components in topologically sorted order: after we discover all vertices in some strongly connected component C and move on to the next strongly connected component C' the only edges out will be to C, which has had all of its vertices already marked as visited (and so the second DFS doesn't visit any other connected component's vertices). Running time is $\Theta(V + E)$.

```
Strongly-Connected-Components(G)
# let's pretend tsort just returns labels: 5, 4, 6, 1, ...
tsort = Topological-Sort(G)
G^T = Transpose(G)
V', E^T = G^T
components = [ ]
for i = 1 : len(G):
    components.append(Depth-First-Search(V'[tsort[i]]))
```

34. Minimum Spanning Tree

A spanning tree of an undirected graph $G = (V, E)$ is a tree (no cycles) that covers all the vertices: $n-1$ edges and there's a path there's a path $u \rightsquigarrow v$ and $v \rightsquigarrow u$ for all $u, v \in V$; a minimum spanning tree is defined by edges $A \subset E$. A minimum weight spanning tree is a spanning tree over a weighted graph such that the sum of the weights along the edges is minimal. The two algorithms prestend are greedy algorithms that proceed by adding **safe edges**: an edge such that the tree unioned with that edge is still a minimum spanning tree. The algorithms find these safe edges in different ways.

I'm going to define some things here that will be useful later on. A cut $(S, V \backslash S)$ is a partition of V into two disjoin sets: S and not S. An edge **crosses** a cut if one vertex of the edge is in S and the other in $V \backslash S$. A cut **respects** a subset $A \subset E$ if and no edges in A cross the cut. A **light edge** crossing a cut is an edge that crosses the cut and is of minimum weight.

Theorem 2. *Let A be a subset of E that is included in some minimum spanning tree, let $(S, V \backslash S)$ be any cut that respects A. If (u, v) is a light edge crossing the cut then it's safe for A.*

Corollary 3. *If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component of G_A then (u, v) is safe for A.*

In actuality the two algorithms add light edges.

34.1. **Kruskal's algorithm.** Kruskal's algorithm uses the Union-Find data structure to build the minimum spanning tree. It adds edges that haven't been added to A in increasing order by weight.

```
Kruskal-Minimum-Spanning-Tree(G)
A = [ ]
V, E = G
for v ∈ V:
    vsets.append(Make-Set(v))
# (u, v, w) ∈ E where w is the weight of the edge
E = sorted(E, key = itemgetter(3))
for u, v ∈ E:
    if Find-Set(vset[u]) ≠ Find-Set(vset[v]):
        A.append((u, v))
        Union(vset[u], vset[v])
return A
```

Firstly the sort costs $O(E \lg E)$. The rest of the running time is a function of the running times of the Union-Find data structure operation running times. The second `for` performs $O(E)$ `Find-Set` and `Union` operations. Along with the $O(V)$ `Make-Set` operations in the first `for` the total is $O((V + E)\alpha(V))$, where α is the Ackermann function. Since we assume G is connected (otherwise it could have no spanning tree) it's the case that $E \geq V - 1$ and so the Union-Find operations actually take $O(E\alpha(V))$. Then since $\alpha(V) = O(\lg V) = O(\lg E)$ we get that the run time is $O(E \lg E)$. Finally since $|E| < |V|^2$ we have that $\lg(E) = O(\lg V)$ and therefore the running time is $O(E \lg V)$.

34.2. **Prim's algorithm.** Prim's algorithm uses a min heap to keep the sorted list of "lightest" edges crossing the cut. It functions very similarly to Dijkstra's shortest path algorithm (which is covered in the next section).

```
Prim-Minimum-Spanning-Tree(G)
V, E = G
# initialize some vertex to be the nucleation point of the
# minimum spanning tree
V[1]['key'] = 0
# let's pretend this minqueue is keyed on V[i]['dist']
Q = minQueue(V)
while len(Q) > 0:
    u = Extract-Min(Q)
    for v ∈ V[u]['neighbors']:
            # let's pretend E is a dict like {(u, v) : w}
            # and that we can check membership in Q
            # basically just have a bunch of dicts floating around
            # while it looks like we're keeping a sorted list
            # of vertices this is actually keeping track of lightest
            # edges crossing the cut (think)
            if v ∈ Q and E[(u, v)] < v['key']:
                # crucially you need to be able to adjust
                # the priority key in the min queue
                v['key'] = E[(u, v)]
                v['prnt'] = u
# minimum spanning tree is parent edges back to nucleation point.
return [(v['prnt'], v) for v ∈ V]
```

Running time is $O(E \lg E)$ for a standard implementation of a min heap but can be sped up to $O(E + V \lg V)$ using a Fibonacci heap.

35. Single source Shortest Path

Single source means shortest path from a particular vertex to all other vertices in the graph. The only other thing to mention here is that if a graph has negative weight cycles (a cycle where the sum of the weights around the cycle [not necessarily all weights] is negative) then shortest path is not defined (because just keep going around the cycle to arbitrarily decrease the weight of any path).

For each of these algorithms the actual shortest paths can be reconstructed by following the `prnt` pointer chains back to the source to create a predecessor tree. This is omitted.

35.1. **Bellman-Ford.** Bellman-Ford is kind of stupid simple: just "relax" all of the distance $|V|-1$ times. What does "relax" mean?

```
Relax(u, v, w)
if v['dist'] > u['dist'] + w:
    v['dist'] = w
    v['prnt'] = u
```

The reason it works is a little convoluted but the intuition is that the shortest path to any vertex can have at most $|V|-1$ edges and relaxing them in order, but with any other edges relaxed in between, produces the shortest path/distance.

```
Bellman-Ford(G, s)
V, E = G
for v ∈ V:
    v['dist'] = ∞
    v['prnt'] = None
s['dist'] = 0
for i = 1 : |V| − 1:
    for (u, v, w) ∈ E:
        Relax(u, v, w)
```

Bellman Ford call also check at the end for negative weight cycles: if any distances can be further relaxed then that vertex is on a negative weight cycle (this follows from the fact that any shortest path can undergo at most $|V|-1$ relaxations). Running time is $O(VE)$.

```
Bellman-Ford(G, s)
V, E = G
for v ∈ V:
    v['dist'] = ∞
    v['prnt'] = None
s['dist'] = 0
for i = 1 : |V| − 1:o
    for (u, v, w) ∈ E:
        Relax(u, v, w)
for (u, v, w) ∈ E:
        if v['dist'] > u['dist'] + w:
            return "Error: negative weight cycle"
```

For completeness I'll mention that to actually find a negative weight cycle if one exists run Bellman-Ford twice: the first time finds and edge on a negative weight cycle. The second time run Bellman-Ford with the source vertex being the one that the distance could have been relaxed to and trace the path produced by Bellman-Ford to its parent.

35.2. **Shortest Path in a DAG.** In a dag you can speed up Bellman-Ford because you can figure out exactly the order in which to relax the edges: just do a topological sort.

```
DAG-Shortest(G, s)
tsort = Topolgoical-Sort (G)
V, E = G
for v ∈ V :
    v['dist'] = ∞
    v['prnt'] = None
s['dist'] = 0
for i = 1 : |V| − 1 : o
    u = tsort[i]
    for (u, v, w) ∈ E :
        Relax (u, v, w)
```

35.3. **Dijkstra's.** Dijkstra's shortest path algorithm is very similar to Prim's minimum weight spanning tree algorithm. Just like Prim's it uses a min heap to keep track of objects in the graph, except it's nearest vertices rather than lightest edges crossing a cut. Running time is $O(E + V \lg V)$ using a Fibonacci heap implementation of min queue. Note that Dijkstra does not work for graph with negative weight edges.

```
Dijkstra(G, s)
V, E = G
# initialize some vertex to be the nucleation point of the
# minimum spanning tree
s['dist'] = 0
# let's pretend this minqueue is keyed on s['dist']
Q = minQueue (V)
while len (Q) > 0 :
    u = Extract-Min (Q)
    for v ∈ u['neighbors'] :
        if v ∈ Q and E[(u, v)] + u['dist'] < v['dist'] :
            # note this is shorthand for updating the key in the minQueue
            v['dist'] = E[(u, v)] + u['dist']
            v['prnt'] = u
# shortest path tree is parent edges back to source
```

35.4. **Heuristic search.** Suppose you wanted to use Dijkstra's to find the shortest path to a particular vertex g, for goal. Well you could just run it and recover the path to g but Dijkstra will waste a lot of time searching the rest of the graph. You can hack Dijkstra to be a little faster by using a different priority function (one that encodes a heuristic for most expedient direction). This prompts it to explore in a particular direction more often since the vertices prioritized by the heuristic function will be popped first from the min queue. The code is exactly the same except for

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right]$$

which becomes

$$v\left[\texttt{'dist'}\right] = \texttt{heuristic}\left(v,g\right)$$

To be concrete suppose we're trying to find the shortest path to a vertex on a grid. Then $\texttt{heuristic}\left(v,g\right)$ would just be Manhattan distance (closer Manhattan distance means higher priority).

`Manhattan-Distance`(a,b)

1 **return** $\left|a\left[\mathrm{x}\right] - b\left[\mathrm{x}\right]\right| + \left|a\left[\mathrm{y}\right] - b\left[\mathrm{y}\right]\right|$

35.5. **A* search.** A star search combines heuristic and Dijkstra's to take into account distance from source and some heuristic for distance to goal. The modification to Dijkstra is

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right]$$

becomes

$$v\left[\texttt{'dist'}\right] = E\left[(u,v)\right] + u\left[\texttt{'dist'}\right] + \texttt{heuristic}\left(v,g\right)$$

36. Difference constraints

A set of difference constraints is a set $x_j - x_i \leq b_k$ whose solution is $\mathbf{x}$ such that all of the constraints are satisfied. These can be solved by first constructing a constraint graph $G = \left(\{v_0, v_1, \ldots, v_n\}, E\right)$ where

$$E = \left\{ \left(v_i, v_j\right) \middle| x_j - x_i \leq b_k \text{ is a constraint} \right\} \cup \left\{\left(v_0, v_i\right)\right\}$$

where $w\left(\left(v_i, v_j\right)\right) = b_k$ and $w\left(\left(v_0, v_i\right)\right) = 0$. Then using Bellman-Ford to find the shortest path $\delta\left(v_0, v_i\right)$ from v_0 to every other vertex. If there's a negative weight cycle then no solution exists. The proof that Bellman-Ford produces a solution hinges on the triangle inequality $\delta\left(a,b\right) \leq \delta\left(a,c\right) + \delta\left(b,c\right)$: since the distance to each of the vertices is 0

$$\delta\left(v_0, v_j\right) \leq \delta\left(v_i, v_j\right) + \delta\left(v_i, 0\right)$$

implies

$$\delta\left(v_0, v_j\right) - \delta\left(v_i, v_0\right) \leq \delta\left(v_i, v_j\right) = w\left(\left(v_i, v_j\right)\right)$$

A system of m constraints in n unknowns produces a graph with $n+1$ vertices and $n+m$ edges and hence running time is $O\left(n^2 + nm\right)$.

37. Transitive Closure

The transitive closure of a graph $G = (V, E)$ is a graph $G' = (V, E')$ where $(u, v) \in E'$ if there's a path there's a path $u \rightsquigarrow v$ in G. This problem has optimal substructure: consider all of the paths from i to j where intermediate vertices (vertices in the path not including i, j) come from vertices $\{1, \ldots, k\} \subset \{1, \ldots, n\}$ and a particular path p. Either k is an intermediate vertex of p or not. If k is not an intermediate vertex then all p's intermediate vertices are drawn from $\{1, \ldots, k-1\}$. If k is an intermediate vertex of p then we can further decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ where k is not an intermediate vertex of either p_1 or p_2. Let $t_{ij}^{(k)}$ be 0 or 1 depending on whether i is connected to j in the transitive closure of G or not, then the Bellman equation is

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ or } \left(t_{ik}^{(k-1)} \text{ and } t_{kj}^{(k-1)}\right)$$

with base case

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \end{cases}$$

`Transitive-Closure`(G)

```
1  V, E = G
2  n = len(V)
3  T = n · [n · [0]]
4  for i = 1 : n:
5      for j = 1 : n:
6          if i == j or (u, v) ∈ E:
7              T[i][j] = 1
8  for k = 1 : n:
9      T' = n · [n · [0]]
10     for i = 1 : n:
11         for j = 1 : n:
12             T'[i][j] = T[i][j] or (T[i][k] and T[k][j])
13     T = T'
14 return T
```

---

38. All pairs Shortest Paths

38.1. **Shortest paths by exponentiation.** Shortest paths have optimal substructure: if vertices i and j are distinct, then we can decompose the path p from i to j into $i \overset{p'}{\rightsquigarrow} k \to j$ where p' must be the shortest path from i to k. Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to j that contains at most m edges and w_{uv} be the weight of the edge between vertices u and v, then

the Bellman equation is

$$\begin{aligned} l_{ij}^{(m)} &= \min\left\{l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right\} \\ &= \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \end{aligned}$$

since $w_{jj} = 0$. Base case is

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

The shortest path weight is then $l_{ij}^{(n-1)}$. Given a matrix L that corresponds to the mth iteration we can compute L' corresponding to the $m+1$th iteration using $W = \{w_{ij}\}$.

`Extend-Shortest-Path`(L, W)

```
1  n = len(L)
2  L' = n · [n · [∞]]
3  for i = 1 : n:
4      for j = 1 : n:
5          for k = 1 : n:
6              L'[i][j] = min{L'[i][k], L[k][j] + W[k][j]}
```

The thing to notice is that this is very much like matrix multiplication $L \cdot W$, and hence schematically

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W = W \\ L^{(2)} &= L^{(1)} \cdot W = W^2 \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1} \end{aligned}$$

Therefore we can use exponentiation by repeated squaring to compute $L^{(n-1)}$. In fact it's even simpler because we just need to square and not worry about anything else since $L^{(n+k)} = L^{(n-1)}$ for all k (since shortest paths don't become shorter...). So

---

`Fast-Shortest-Path`(W)

```
1  n = len(W)
2  L = W
3  m = 1
4  while m < n − 1:
5      L' = Extend-Shortest-Path(L, L):
6      L = L'
7      m = 2m
8  return L
```

Running time is $\Theta\left(n^3 \lg n\right)$.

38.2. **Floyd-Warshall.** Floyd-Warshall is very similar to Transitive closure. Consider a subset $\{1,\ldots,k\}$ of vertices. For any two vertices i,j consider all paths whose intermediate vertices (vertices in the path not including i,j) all come from $\{1,\ldots,k\}$ and let p be the minimal weight path from among them. If k is not an intermediate vertex of p then all intermediate vertices of p come from $\{1,\ldots,k-1\}$. If k is an intermediate vertex of p then we can decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ where k is not an intermediate vertex of neither p_1 nor p_2 where both p_1,p_2 have intermediate vertices coming from $\{1,\ldots,k-1\}$. Furthermore both p_1,p_2 are shortest paths themselves. Therefore the Bellman equation is

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k>0 \end{cases}$$

`Floyd-Warshall`(W)

```
1  n = len(W)
2  D = W
3  for k = 1 : n:
4      D' = n · [n · [∞]]
5      for i = 1 : n:
6          for j = 1 : n:
7              D'[i][j] = min{D[i][j], D[i][k] + D[k][j]}
8      D = D'
9  return D
```

Running time is obviously $O\left(n^3\right)$.

38.3. **Johnson's Algorithm.** John's algorithm is slightly faster than Floyd-Warshall on sparse graphs. It works by reweighting all of the vertices so that none are negative using Bellman-Ford and then runs Dijkstra from each vertex. It reweights in a way that doesn't change any of the shortest paths: for any $h(u)$ that maps vertices to real numbers

$$\hat{w}((u,v)) = w((u,v)) + h(u) - h(v)$$

does not alter shortest paths. How to pick $h(u)$ so that $\hat{w}((u,v)) > 0$. Make it a distance function: similar to how a super source is used in difference graphs define a new vertex s with 0 weight edges to every other vertex and let $h(u) = \delta(s,v)$. Since $\delta(s,v)$ is a distance function by the triangle inequality

$$h(v) \le h(u) + w(u,v)$$

and hence

$$\hat{w}((u,v)) = w((u,v)) + h(u) - h(v) \ge 0$$

So just like in difference constraints use Bellman-Ford to compte $\delta(s,v)$ for all $v \in V$. Running time is $O\left(V^2 \lg V + VE\right)$.

```
Johnsons(G)
V, E = G
V' = V ∪ {s}
E' = E ∪ {(s, v, 0) for v ∈ V}
# let's pretend Bellman-Ford returns a distance function
δ = Bellman-Ford((V', E'), s)
for (u, v, w) ∈ E:
    w = E[(u, v)] + δ(s, u) − δ(s, v)
D = n · [n · [∞]]
n = len(V)
for i = 1 : n:
    for j = 1 : n:
        # let's pretend Dijkstra returns a distance function
        δ̂ = Dijkstra(G, i)
        # undo the reweighting to recover the actual distances
        D[i][j] = δ̂(i, j) − δ(s, u) + δ(s, v)
return D
```

39. Min Cut - Max Flow

A flow network $G = (V, E)$ is a directed graph in which each edge has a capacity $c((u, v)) \geq 0$ and if a forward edge exists then no reverse edge exists. Further there are two distinguished vertex, source s and sink t, such that for every vertex v it's the case that $s \rightsquigarrow v \rightsquigarrow t$. A **flow** on a flow network is a real valued function $f : V \times V \to \mathbb{R}$ that satisfies two properties:

(1) Capacity constraint: For all u, v, it's the case that $0 \leq f(u, v) \leq c((u, v))$
(2) Flow conservation: For all $u \in V - \{s, t\}$, it's the case that that flow in equals flow out, i.e.

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

The **value** of the flow $|f|$ is defined as the total flow out of the source minus the total flow into the source, i.e.

$$|f| \equiv \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Typically, a flow network will not have any edges into the source and so $|f| = \sum f(s, v)$. The **maximum-flow** problem is maximizing $|f|$ on a particular flow network G.

39.1. Ford-Fulkerson/Edmond's-Karp.

Given a flow network G we construct a **residual network** G' which models "room to augment" the flow. Then we augment until there's no more room.

Define the **residual capacity** $c_f(u,v)$ to be

$$c_f(u,v) = \begin{cases} c((u,v)) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

The first case is clearly "room to grow". The second case is the flow that's currently going across edge (u,v) but in the reverse direction (think of it "room to squelch" the flow). The residual network $G' = (V, E_f)$ where

$$E_f = \left\{ (u,v) \in V \times V \middle| c_f(u,v) > 0 \right\}$$

An **augmentation** of a flow f by f' is a new flow $(f \uparrow f')$ defined to be

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

i.e. add $f'(u,v)$ units of flow along edge (u,v) but squelch by $f'(v,u)$ units (in the residual network there will be an edge (v,u) with $f'(v,u)$ units of flow on it, think of it as back pressure). Turns out that if f' is a flow in G' then $|f \uparrow f'| = |f| + |f'|$.

An **augmenting path** p is a simple path from s to t in G'. The **residual capacity** $c_f(p)$ of p is defined

$$c_f(p) = \min \left\{ c_f(u,v) \,\middle|\, (u,v) \in p \right\}$$

So define a flow $f_p(u,v)$ that flows this minimum capacity along the path, i.e.

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \in p \\ 0 & \text{otherwise} \end{cases}$$

and then f_p is a flow in G' with $|f_p| = c_f(p) > 0$. As a corollary $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Recall cuts $(S, V \backslash S)$ of a graph. The **capacity of a cut** (S,T) where $s \in S$ and $t \in T$ is defined

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

i.e. the sum of the capacities of all of the edges crossing the cut. The **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network (where the source is in one partition and the sink is in the other). Naturally the net flow across any cut is equal to $|f|$. Something stronger is true.

Theorem 4. *Min cut - Max flow*

The following are equivalent

(1) *f is a maximum flow in G.*
(2) *The residual network G' contains no augmenting paths.*
(3) *$|f| = c(S,T)$ for some cut (S,T) of G. In fact the minimum capacity cut.*

After all that the Ford-Fulkerson/Edmond's-Karp algorithm for finding the max flow on a flow network G is simple: construct the residual network G' and keep looking for augmenting paths and use them to augment the flow on G. When there are no more we know that f is a maximum flow and that $|f|$ is equal to the minimum cut. We use breadth-first search to find the augmenting path. First we need a function that constructs the residual network.

```
Residual-Network(G)
1  V, E = G
2  E' = [ ]
3  for i = 1 : len(V):
4      for j = 1 : len(V):
5          E'[(i, j)] = {}
6          if (i, j) ∈ E:
7              E'[(i, j)]['capacity'] = max{0, E[(i, j)]['capacity'] − E[(i, j)]['flow']}
8          elif (j, i) ∈ E:
9              E'[(i, j)]['capacity'] = E[(j, i)]['flow']
10         else:
11             E'[(i, j)]['capacity'] = 0
12 return (V, E')
```

Finally we implement the plan of search for augment paths and using them to augment the flow on G. Running time is $O\left(VE^2\right)$ because the number of augmentations is $O(VE)$.

```
Ford-Fulkerson-Edmonds-Karp(G, s, t)
V, E = G
for u, v ∈ E:
    E[(u, v)]['flow'] = 0
# let's pretend Breadth-first-search returns a path if one exists
# or None if none exists
G' = Residual-Network(G)
V, E' = G'
p = Breadth-first-search(G', s, t)
while p ≠ None:
    c_f = min{[E'[(u, v)]['capacity'] for (u, v) ∈ p]}
    for (u, v) ∈ p:
        if (u, v) ∈ E:
            E[(u, v)]['flow'] = E[(u, v)]['flow'] + c_f
        else:
            E[(v, u)]['flow'] = E[(v, u)]['flow'] − c_f
    G' = Residual-Network(G)
    V, E' = G'
    p = Breadth-first-search(G', s, t)
```

To find the min cut: do a depth-first search in the *residual* network from the source s marking all vertices which can be reached. The cut is all edges going from marked vertices to unmarked vertices. Those edges are saturated and correspond to the minimum cut. To prove this we need to basically almost prove the Min cut - Max Flow theorem

Proof. Let $G = (V, E)$ be a flow network with s, t being source and sink. Consider the flow f computed for G by Ford-Fulkerson. In the residual network G' define A to be the set of vertices reachable from s and $A^c = V \backslash A$.

Claim. $|f| = c(A, A^c)$

Proof. Since $|f| = \sum_{u \in A} \sum_{v \in A^c} f(u, v) - \sum_{v \in A^c} \sum_{u \in A} f(v, u)$, i.e. flow out of A minus flow into A, for $|f|$ to equal $c(A, A^c)$ we need

(1) All outgoing edges from A to be saturated.
(2) All incoming edges to have zero flow.

Why? Because

$$c(A, A^c) = \sum_{u \in A} \sum_{v \in A^c} c((u, v))$$

and so we should have $c((u, v)) = f(u, v)$ for each out going edge and $f(v, u)$ for each incoming edge. We verify this by considering two cases:

(1) In G there exists an outgoing edge $(x, y), x \in A, y \in A^c$ that is not saturated. This implies there's an edge $(x, y) \in E_f$ and therefore a path from $s \in A$ to $y \in A^c$, which contradicts that A^c is unreachable from s. Therefore $(x, y) \in G$ is saturated.
(2) In G there exists an incoming edge $(y, x), x \in A, y \in A^c$ that is not 0. This implies there's a squelch edge $(x, y) \in E_f$ and therefore a path from $s \in A$ to $y \in A^c$, which contradicts that A^c is unreachable from s. Therefore $(y, x) \in G$ has zero flow.

Therefore $|f| = c(A, A^c)$.

□

Since f was obtained my Ford-Fulkerson it is the maximum flow on the network. Then since any flow in the network is bounded above by every cut, and in particular the minimum cut, the above described cut is the minimum cut.

□

40. Maximum bipartite matching

A bipartite graph is one for which the vertices can be divided into two sets, with edges only going between the two sets. A **maximum bipartite matching** is one which 1-1 matches the largest number of vertices. The matching problem is finding such a matching. The solution is two use Min cut - Max flow: set up a source that connects to all of the vertices in one partition and a sink that connects to all of the vertices in the other. Then set the capacity of all edges to 1 and run Edmonds-Karp. The minimum cut corresponding to the maximum flow gives you the matching. How do we know that the flow will take on an integer value? I.e. that we won't flow 3/4 and 1/4 down some edges?

Theorem 5. *Integrality theorem*

If the capacity function $c((u, v))$ takes on only integral values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer and furthermore $f(u, v)$ is an integer for all u, v.

Running time is $O(VE)$.