

CONTENTS

Note	3
Part 1. Foundations	3
1. Insertion Sort	3
2. Selection Sort	4
3. Bubble Sort	4
4. Merge Sort	4
5. Binary search	5
6. Horner's Rule	6
7. Reservoir Sampling	6
7.1. Unweighted simple	6
7.2. Unweighted slightly more involved	6
7.3. Weighted	7
8. Online Maximum	7
9. Stable Matching	8
Part 2. Sorting and Order Statistics	9
10. Heaps	9
10.1. Max Heapify	9
10.2. Build Max Heap	10
10.3. Extract Min	10
10.4. Heap sort	10
10.5. Heap increase key	11
10.6. Heap insert	11
11. Quicksort	11
12. Counting Sort	12
13. Radix Sort	13
14. Bucket Sort	13
15. Order statistics	14
15.1. Quickselect	14
15.2. Quickerselect	14
Part 3. Data Structures	15

16. Hash Tables	15
16.1. Hash function	15
16.2. Hashing with chaining	15
16.3. Hashing with open addressing	16
17. Binary Search Tree	17
17.1. Inserting into a binary search tree	18
17.2. Searching a binary search tree	19
17.3. Binary search tree min/max	19
17.4. Binary search tree predecessor/successor	20
17.5. Deleting from a binary search tree	20
17.6. Pre-order/In-order/Post-order traversal	22
18. Treap	23
18.1. Treap search	23
18.2. Treap insert	23
18.3. Treap delete	25
19. Cartesian Tree	25
20. Interval Trees	26
20.1. Interval search	26
21. Union-Find	27
21.1. Make set	27
21.2. Find set	27
21.3. Union	27
22. Euler circuit	28
23. Tarjan's Least Common Ancestor	28
24. Range Minimum Queries	29
Part 4. Advanced Design Techniques	29
25. Dynamic Programming	29
25.1. Fibonacci Sequence	30
25.2. Getting to work ¹	31
25.3. Towers of Hanoi	32
25.4. Egg drop	33
25.5. Maximum Positive Subarray/Kidane's algorithm	35

¹This example is Applied Mathematical Programming by Bradley,Hax, Magnanti chapter 11.

25.6. Longest increasing subsequence	36
Part 5. Graph Algorithms	38
26. Representations of Graphs	38
26.1. Adjacency matrix	38
26.2. Adjacency list	38
27. Breadth-first Search	39
28. Depth-first Search	39

Note

Everything is 1 indexed, despite using vaguely Pythonic syntax. This means $A[\text{len}(A)] = A[-1]$. Slicing is $A[a : b] = [A_a, A_{a+1}, \dots, A_{b-1}]$. 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 $1 : n$.

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( $A$ )
1  for  $j = 2 : \text{len}(A)$ :
2       $key = A[j]$ 
3       $i = j - 1$ 
4      while  $i > 0$  and  $A[i] > key$ :
5           $A[i + 1] = A[i]$ 
6           $i = i - 1$ 
7      # either we're one passed the left end
8      # or  $A[i] \leq key$  and so
9      #  $A[i + 1]$  is the proper place for key
10      $A[i + 1] = key$ 

```

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 $O(n^2)$.

```

Selection-Sort( $A$ )
1 for  $j = 1 : \text{len}(A)$ :
2    $A[j] = \min(A[j + 1 :])$ 

```

3. BUBBLE SORT

“Bubble up” pair by pair. Stop when no more “bubbings” are possible. Running time is $O(n^2)$.

```

Bubble-Sort( $A$ )
1  $flips = \text{True}$ 
2 while  $flips$ :
3    $flips = \text{False}$ 
4   for  $i = 1 : \text{len}(A) - 1$ :
5     if  $A[i] > A[i + 1]$ :
6        $A[i], A[i + 1] = A[i + 1], A[i]$ 
7      $flips = \text{True}$ 

```

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( $A$ )
1  if len( $A$ ) == 1:
2      return  $A$ 
3  else:
4       $h = \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor$ 
5       $L = \text{Merge-Sort}(A[h:])$ 
6       $R = \text{Merge-Sort}(A[1:h])$ 
7       $M = []$ 
8      while len( $L$ ) > 0 and len( $R$ ) > 0:
9          # take the minimum of the  $\{L[1], R[1]\}$ 
10         # and remove it from further contention
11         if  $L[1] < R[1]$ :
12              $M.append(L[1])$ 
13             del  $L[1]$ 
14         else:
15              $M.append(R[1])$ 
16             del  $R[1]$ 
17         # one of  $L, R$  is large by one element.
18         if len( $L$ ) > 0:
19              $M.append(L[1])$ 
20         else:
21              $M.append(R[1])$ 
22              $M.append(R[-1])$ 
23     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( $A, x$ )
1  if  $x == A[h]$ :
2      return True
3  elif  $x < A[h]$ :
4      return Binary-Search( $A[1:h]$ )
5  else:
6      return Binary-Search( $A[h:]$ )

```

6. HORNER'S RULE

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

$$p(x) = \sum_{k=1}^n a_k x^k = a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + xa_n)))$$

```
Horner's-Rule( $A, x$ )
```

```
1  $y = 0$ 
2 for  $i = n : 1$ :
3      $y = A[i] + x \cdot y$ 
```

7. RESERVOIR SAMPLING

7.1. Unweighted simple. Suppose you want to sample k items from n items $A = [a_1, \dots, a_n]$ 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 $j \in \{1, \dots, i\}$ inclusive. If $i \leq k$ the replace i th item. Running time is $\Theta(n)$.

```
Unweighted-Reservoir-One( $A, k$ )
```

```
1  $R = [a_0, a_1, \dots, a_k]$ 
2 for  $i = k + 1 : \text{len}(A)$ :
3      $j = \text{Random}(1, i)$  # both ends inclusive
4     if  $j \leq k$ :
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( $A, k$ )
1   $R = \text{Min-Priority-Queue}$ 
2  for  $i = 1 : k$ :
3       $u \sim \text{Uniform}(0, 1)$ 
4      # priority key is first entry in argument
5       $H.\text{insert}(u, A[i])$ 
6  for  $i = k + 1 : \text{len}(A)$ :
7       $u \sim \text{Uniform}(0, 1)$ 
8      #  $H.\text{min}$  returns value of minimum without extracting
9      if  $u < H.\text{min}$ :
10          $H.\text{Extract-Min}()$ 
11          $H.\text{insert}(u, A[i])$ 

```

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( $A, k$ )
1   $R = \text{Min-Priority-Queue}$ 
2  for  $i = 1 : k$ :
3       $u \sim \text{Uniform}(0, 1)$ 
4       $u = u^{1/A[i].\text{weight}}$ 
5       $H.\text{insert}(u, A[i])$ 
6  for  $i = k + 1 : \text{len}(A)$ :
7       $u \sim \text{Uniform}(0, 1)$ 
8       $u = u^{1/A[i].\text{weight}}$ 
9      if  $u < H.\text{min}$ :
10          $H.\text{Extract-Min}()$ 
11          $H.\text{insert}(u, A[i])$ 

```

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  $i = 2 : \lceil n/e \rceil$ :
4     if  $A[i] > m$ :
5          $m = A[i]$ 
6 # this one is for keeps
7 for  $i = k+1 : \text{len}(A)$ :
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 $O(n^2)$. The next question is the solution optimal. In fact it is not. The algorithm 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 $O(n^2)$

```

Matching( $P_m, P_w, men$ )
1  # men is an array of men to be matched
2  #  $P_m$  is an  $n \times n$  preferences matrix for the men, sorted by increasing priority
3  #  $P_w$  is an  $n \times n$  a preferences matrix for the women, sorted
4   $matched_M = \{\}$ 
5   $matched_W = \{\}$ 
6  while  $\text{len}(men) > 0$ :
7       $m = men[-1]$ 
8       $w = P_m(m)[-1]$ 
9      if  $w$  not in  $matched_W$ :
10          $matched_M[m] = w$ 
11          $matched_W[w] = m$ 
12         del  $P_m(m)[-1]$ 
13         del  $men[-1]$ 
14     else: # if w is already matched
15          $m' = matched_W[w]$ 
16         # and prefers m to m'
17         if  $P_w[w][m] > P_w[w][m']$ :
18             # match m with w
19              $matched_M[m] = w$ 
20              $matched_W[w] = m$ 
21             del  $P_m(m)[-1]$ 
22             del  $men[-1]$ 
23             # unmatch m'
24             del  $matched_M[m']$ 
25              $matched_M.append(m')$ 

```

Part 2. Sorting and Order Statistics

10. HEAPS

Array Heaps² 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 $O(\lg n)$.

²Heaps 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  $2i \leq \text{len}(A)$  and  $A[2i] > A[i]$ :
4       $largest = 2i$ 
5  # if the right child exists and is greater then switch
6  if  $2i + 1 \leq \text{len}(A)$  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  $i = \lfloor \text{len}(A)/2 \rfloor : 1$ :
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 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   $A.\text{pop}()$ 
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  $\text{len}(A) > 0$ :
3      $s.\text{append}(\text{Extract-Min}(A))$ 
4 return  $\text{reversed}(s)$ 

```

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  $A[\lfloor i/2 \rfloor] < A[i]$ :
6      $A[\lfloor i/2 \rfloor], A[i] = A[i], A[\lfloor i/2 \rfloor]$ 
7      $i = \lfloor i/2 \rfloor$ 

```

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  $A.\text{append}(-\infty)$ 
2 Heap-Increase-Key( $A, \text{len}(A), key$ )

```

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)
1  if len(A) ≤ 1:
2      return A
3  else:
4      # randomly pick a pivot
5      p = Random(1, len(A)) # inclusive
6      # swap so that you can exclude from contention the pivot
7      A[p], A[-1] = A[-1], A[p]
8      # partition
9      Aleft = filter(A[: -1], λe: e ≤ A[-1])
10     Aright = filter(A[: -1], λe: e > A[-1])
11     # recursively sort
12     Aleft = QuickSort(Aleft)
13     Aright = QuickSort(Aright)
14     # combine
15     return Aleft + A[-1] + Aright

```

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 $A = [a_1, \dots, a_n]$ 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( $A$ )
1  $k = \max(A)$ 
2  $C = (k + 1) * [0]$ 
3 # count how many of values from 1 to k there is
4 for  $i = 1 : \text{len}(A)$ :
5      $C[A[i]] = C[A[i]] + 1$ 
6 # count how entries in A less or equal to i
7 for  $i = 1 : k$ :
8      $C[i] = C[A[i]] + 1$ 
9 # now place the items in the correct places
10  $B = (k + 1) * [None]$ 
11 # go in reverse direction in order for sort to be stable
12 for  $i = \text{len}(A) : 1$ :
13     #  $a_i$  has  $C[a_i]$  elements to its left in B
14      $B[C[A[i]]] = A[i]$ 
15     # if there are multiples of  $a_i$  then the next
16     # should be left of in order for stable
17      $C[A[i]] = C[A[i]] - 1$ 

```

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( $A, d$ )
1 for  $i = 1 : d$ :
2     # let's pretend i can pass Insertion-Sort a key
3     Insertion-Sort( $A, \text{key} = \text{lambda } a : a[-i]$ )

```

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( $A$ )
1  $n = \text{len}(A)$ 
2  $B = n \cdot []$ 
3 for  $i = 1 : n$ :
4     # bucket (imagine  $n = 10$ ).
5     # the +1 is because  $\lfloor 10(0.01) \rfloor = 0$ 
6      $B[\lfloor nA[i] \rfloor + 1].\text{append}(A[i])$ 
7 for  $i = 1 : n$ :
8     Insertion-Sort( $B[i]$ )
9 return  $B$ 

```

15. ORDER STATISTICS

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

```

Quickselect( $A, k$ )
1 if  $\text{len}(A) \leq 1$ :
2     return  $A[1]$ 
3 else:
4      $p = \text{Random}(1, \text{len}(A))$  # inclusive
5      $A[p], A[-1] = A[-1], A[p]$ 
6      $A_{\text{left}} = \text{filter}(A[: -1], \lambda e : e \leq A[-1])$ 
7     if  $k == \text{len}(A_{\text{left}} + 1)$ 
8         # in sorted order  $A[1 : \text{len}(A_{\text{left}} + 1)] = A_{\text{left}} + [A[-1]]$ 
9         # and so the pivot is 1 "in front" of  $A_{\text{left}}$ 
10        return  $A[-1]$ 
11    elif  $k < \text{len}(A_{\text{left}})$ 
12        # the  $k$ th order statistic in  $A$  is still the  $k$ th order statistic in  $A_{\text{left}}$ 
13        return Quickselect( $A_{\text{left}}, k$ )
14    else:
15         $A_{\text{right}} = \text{filter}(A[: -1], \lambda e : e > x)$ 
16        # the  $k$ th order statistic is  $(k - \text{len}(A_{\text{left}}) - 1)$ th statistic in  $A_{\text{right}}$ 
17        # think about it likes this:  $A = [1, 2, 3, 4, 5]$  and we partition on
18        # 3 and we look for the 4th order statistic. well obviously it's
19        # 4 =  $A_{\text{right}}[k - \text{len}(A_{\text{left}}) - 1] = A_{\text{right}}[1]$ 
20        return Quickselect( $A_{\text{right}}, k - \text{len}(A_{\text{left}}) - 1$ )

```

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( $A, k$ )
1  if len( $A$ ) == 0:
2      return  $A[1]$ 
3  else:
4      # divide into  $n$  groups of 5 (except for the last one)
5      # and use a sort in order to get medians.
6       $n = \lfloor \text{len}(A) / 5 \rfloor$ 
7       $m_1 = \text{Insertion-Sort}(A[1 : 5 + 1])[3]$ 
8       $m_2 = \text{Insertion-Sort}(A[5 : 10 + 1])[3]$ 
9      :
10      $m_n = \text{Insertion-Sort}(A[5n :]) \left[ \left\lfloor \frac{\text{len}(A[5n :])}{2} \right\rfloor \right]$ 
11     # recursively compute median of medians and use it as the pivot
12     # after this recursive call the pivot is in position  $\lfloor n/2 \rfloor$ 
13     Quickerselect( $[m_1, m_2, \dots, m_n], \lfloor n/2 \rfloor$ )
14      $A[\lfloor n/2 \rfloor], A[-1] = A[-1], A[\lfloor n/2 \rfloor]$ 
15      $x = A[-1]$ 
16      $A_{\text{left}} = \text{filter}(A[: -1], \lambda e : e \leq x) + [x]$ 
17     if  $k == \text{len}(A_{\text{left}})$ 
18         return  $x$ 
19     elif  $k < \text{len}(A_{\text{left}})$ 
20         return Quickerselect( $A_{\text{left}}, k$ )
21     else:
22          $A_{\text{right}} = \text{filter}(A[: -1], \lambda e : e > x)$ 
23         return Quickerselect( $A_{\text{right}}, k - \text{len}(A_{\text{left}})$ )

```

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

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

where $A \approx (\sqrt{5} - 1) / 2$ and $kA \bmod 1$ 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      $m = \text{len}(H)$ 
4      $hsh = \lfloor m(k(\sqrt{5} - 1)/2 \bmod 1) \rfloor$ 
5      $H[hsh].\text{append}(v)$ 
6
7 HashRetrieve( $H, k$ )
8      $m = \text{len}(H)$ 
9      $hsh = \lfloor m(k(\sqrt{5} - 1)/2 \bmod 1) \rfloor$ 
10    if  $\text{len}(H[hsh]) > 0$ :
11        return  $H[hsh][1]$ 
12    else:
13        return None
14
15 HashDelete( $H, k, v$ )
16      $m = \text{len}(H)$ 
17      $hsh = \lfloor m(k(\sqrt{5} - 1)/2 \bmod 1) \rfloor$ 
18      $i = 1$ 
19     while  $i \leq \text{len}(H[hsh])$ :
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    $m = \text{len}(H)$ 
4    $hsh = \lfloor m(k(\sqrt{5} - 1)/2 \bmod 1) \rfloor$ 
5    $hsh_{orig} = hsh$ 
6   if  $H[hsh] == \text{NIL}$ :
7      $H[hsh] = (k, v)$ 
8   else:
9      $hsh = hsh + 1$ 
10    while  $H[hsh] \neq \text{NIL}$  and  $hsh \neq hsh_{orig}$ :
11      # mod so it swings back around and +1
12      # because indexing starts at 1, not 0
13       $hsh = (hsh + 1 \bmod m) + 1$ 
14    if  $H[hsh] == \text{NIL}$ :
15       $H[hsh] = (k, v)$ 
16    else:
17      return "Error: Hash table full"
18
19 HashRetrieve( $H, k$ )
20    $m = \text{len}(H)$ 
21    $hsh = \lfloor m(k(\sqrt{5} - 1)/2 \bmod 1) \rfloor$ 
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  $H[hsh] \neq \text{NIL}$  and  $hsh \neq hsh_{orig}$ :
28       # mod so it swings back around and +1
29       # because indexing starts at 1, not 0
30        $hsh = (hsh + 1 \bmod m) + 1$ 
31     if  $H[hsh] == \text{NIL}$  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()`.

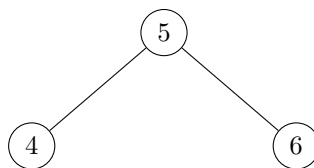
Binary Tree

```

1  btree = lambda parent, name, val, lchild, rchild:
2      {'parent':parent, 'name':name, 'val':val,
3      'lchild':lchild, 'rchild':rchild}
4
5  # counter generator is for labeling
6  def counter(x):
7      start = x
8      while True:
9          yield start
10         start += 1
11  c = counter(1)
12
13  root = btree(None, next(c), 5, None, None)
14  left = btree(root, next(c), 4, None, None)
15  right = btree(root, next(c), 6, None, None)
16  root['lchild'] = left
17  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( $B, k$ )
1  # B is a btree dict as described above, corresponding to the
2  # root of the tree
3   $prnt = ptr = B$ 
4  while  $ptr \neq \text{None}$ :
5      # because of python's memory model we need to keep track
6      # of parent since names are references not pointers, i.e.
7      # you can't reassign pointers like in C
8       $prnt = ptr$ 
9      if  $ptr['val'] \leq k$ :
10          $ptr = ptr['lchild']$ 
11     else:
12          $ptr = ptr['rchild']$ 
13 if  $prnt['val'] \leq k$ 
14      $prnt['lchild'] = \text{btree}(prnt, \text{next}(c), k, \text{None}, \text{None})$ 
15     return  $prnt['lchild']$ 
16 else:
17      $prnt['rchild'] = \text{btree}(prnt, \text{next}(c), k, \text{None}, \text{None})$ 
18     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( $B, k$ )
1   $ptr = B$ 
2  while  $ptr \neq \text{None}$  and  $ptr['val'] \neq k$ :
3      if  $ptr['val'] < k$ :
4           $ptr = ptr['lchild']$ 
5      else:
6           $ptr = ptr['rchild']$ 
7  if  $ptr == \text{None}$ :
8      return "Error: key missing"
9  else:
10     return  $ptr$ 

```

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

Binary-Tree-Min(B)

```

1  $ptr = B$ 
2 while  $ptr['lchild'] \neq \text{None}$ :
3      $ptr = ptr['lchild']$ 
4 return  $ptr$ 

```

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

Binary-Tree-Max(B)

```

1  $ptr = B$ 
2 while  $ptr['rchild'] \neq \text{None}$ :
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  $ptr = \text{Binary-Tree-Search}(B, k)$ 
2 return  $\text{Binary-Tree-Max}(ptr['lchild'])$ 

```

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  $ptr = \text{Binary-Tree-Search}(B, k)$ 
2 return  $\text{Binary-Tree-Min}(ptr['rchild'])$ 

```

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 its right child (it has no left child). Running time is $O(\lg n)$ if the tree is balanced.

```

Binary-Tree-Delete( $B, k$ )
1   $ptr = \text{Binary-Tree-Search}(B, k)$ 
2  # trivial case, successor is parent
3  if  $ptr['rchild'] == \text{None}$ :
4      Transplant( $ptr, ptr['lchild']$ )
5  elif  $ptr['lchild'] == \text{None}$ :
6      Transplant( $ptr, ptr['rchild']$ )
7  else:
8       $succ = \text{Binary-Tree-Successor}(ptr)$ 
9      # if the successor is the right child of ptr then
10     # then right child has no left child and task simple
11     if  $succ == ptr['rchild']$ :
12         Transplant( $ptr, succ$ )
13          $succ['lchild'] = ptr['lchild']$ 
14          $succ['lchild']['parent'] = succ$ 
15     else: # otherwise we have to fix successor subtrees and do the same thing
16           # including fixing the right child
17           # fix successor
18           Transplant( $succ, succ['rchild']$ )
19           # don't lose right child of ptr
20            $succ['rchild'] = ptr['rchild']$ 
21            $succ['rchild']['parent'] = succ$ 
22           # move successor into ptr's position
23           Transplant( $ptr, succ$ )
24            $succ['lchild'] = ptr['lchild']$ 
25            $succ['lchild']['parent'] = succ$ 

```

17.6. Pre-order/In-order/Post-order traversal. A Pre-order/In-order/Post-order traversal of a binary tree is a traversal that 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( $B$ )
1  print( $B['val']$ )
2  Pre-order-traversal( $B['lchild']$ )
3  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( $B$ )
```

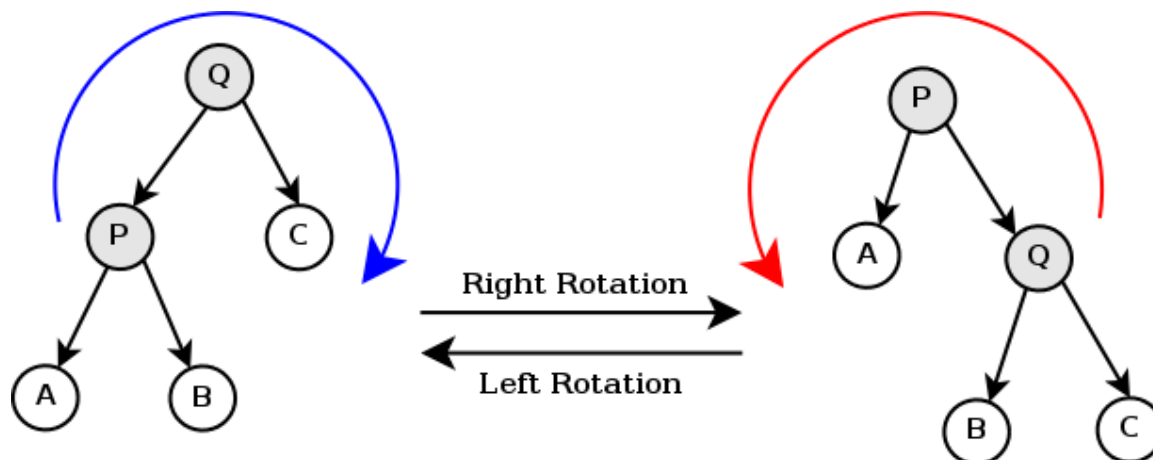
```
1 Pre-order-traversal( $B$ ['lchild'])
2 Pre-order-traversal( $B$ ['rchild'])
3 print( $B$ ['val'])
```

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 simplest is a random binary tree using a treap. A treap combines the invariants of a binary tree *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   $u = \text{Random}(0, 1)$  # both ends inclusive
3   $ptr = \text{Binary-Tree-Insert}(T, (u, k))$ 
4   $prnt = ptr['parent']$ 
5  while  $prnt \neq \text{None}$  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( $T, k$ )
1  # T is a binary tree that's a treap
2
3   $ptr = \text{Binary-Tree-Search}(T, k)$ 
4  while  $ptr['lchild'] \neq \text{None}$  or  $ptr['rchild'] \neq \text{None}$ :
5      if  $ptr['lchild'] \neq \text{None}$  and  $ptr['lchild']['val'] > ptr['rchild']['val']$ 
6           $\text{Right-Rotate}(ptr)$ 
7      else:
8           $\text{Left-Rotate}(ptr)$ 
9      if  $ptr = ptr['parent']['lchild']$ :
10          $ptr['parent']['lchild'] = \text{None}$ 
11      else:
12          $ptr['parent']['rchild'] = \text{None}$ 
13      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 Cartesian 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 $A = [a_1, \dots, a_n]$? 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( $A$ )
1   $T = \text{btree}(\text{None}, \text{next}(c), A[1], \text{None}, \text{None})$ 
2   $ptr = prnt = T$ 
3  for  $i = 2 : \text{len}(A)$ :
4      while  $prnt['parent'] \neq \text{None}$  and  $A[i] < prnt['val']$ 
5           $prnt = prnt['parent']$ 
6      if  $prnt == \text{None}$ : # then we're at the root
7          # and  $A[i]$  is the smallest value we've seen so far
8           $ptr = \text{btree}(\text{None}, \text{next}(c), A[i], prnt, \text{None})$ 
9           $prnt['parent'] = ptr$ 
10     else:
11          $ptr = \text{btree}(prnt, \text{next}(c), A[i], prnt['rchild'], \text{None})$ 
12          $prnt['rchild'] = ptr$ 
13      $prnt = ptr$ 

```

20. INTERVAL TREES

An interval tree is built atop your favorite 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.

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( $T, i$ )
1  #  $T$  is an interval tree,  $i = [a, b]$ 
2   $a, b = i[1], i[2]$ 
3  #  $j['left']$  is left endpoint of interval and
4  #  $j['right']$  is right endpoint
5  intersect = lambda  $j$ :  $a \leq j['left'] \leq b$  or  $j['left'] \leq a \leq j['right']$ 
6  # 'int' is interval associated with vertex
7  while  $T \neq \text{None}$  and not intersect( $T['int']$ ):
8      if  $T['lchild'] \neq \text{None}$  and  $a \leq T['lchild']['max']$ :
9           $T = T['lchild']$ 
10     else:
11          $T = T['rchild']$ 
12  return  $T$ 

```

21. 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}

```

21.1. Make set.

21.2. **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']  $\neq x$ :
2    $x$ ['rep'] = Find-Set( $x$ ['rep'])

```

21.3. **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

```

22. 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)
1  # u is a vertex with children
2  # print it going past
3  print(u)
4  for v in u['children']:
5      Euler-Circuit(v)
6      # print is coming back
7      print(u)

```

23. 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 unfinished (i.e. not completely traversed by the Euler circuit).

```

Least-Common-Ancestor(u)
1  # u is the root of a tree
2  uset = Make-Set(u)
3  # this is the Euler-Circuit transformation (equivalent of print)
4  uset['ancestor'] = u
5  for v in u['children']:
6      Least-Common-Ancestor(v)
7      # let's pretend there's a big table where i can fetch vset from
8      Union(uset, vset)
9      uset['ancestor'] = u
10 # uset['val'] = u
11 uset['val']['color'] = Red
12 for each v such that  $\{u, v\} \in P$ :
13     if v['color'] == Red:
14         print("Least common ancestor of  $\{u, v\}$  is " + vset['ancestor'])

```

24. RANGE MINIMUM QUERIES

Given a sequence of distinct values and a subsequence (specified by its 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

25. DYNAMIC PROGRAMMING

Note this is a fairly formal explanation of dynamic programming. Skip if you're not interested in formalism.

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).

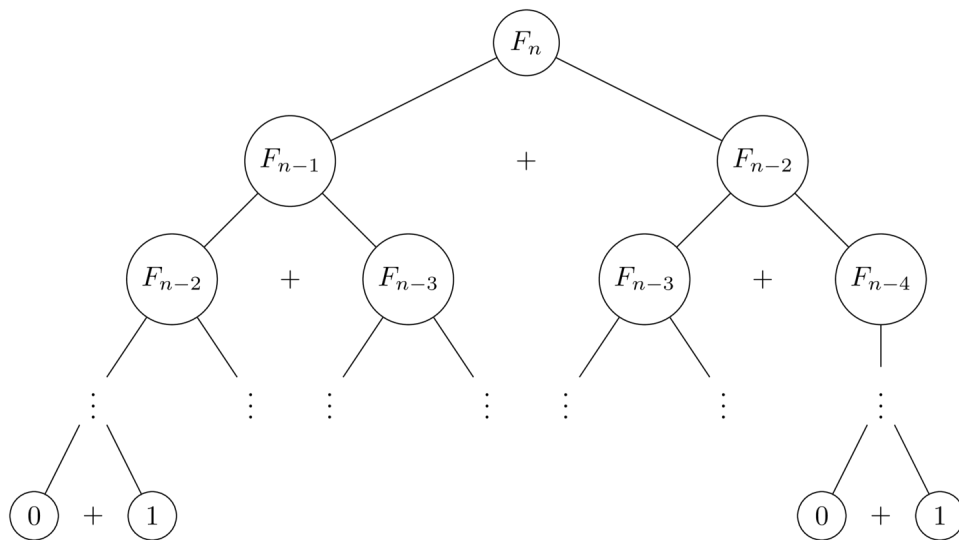
Richard Bellman showed that a dynamic optimization problem in discrete time can be stated in a recursive, step-by-step form known as **backward induction** by writing down the relationship

between the value function³ 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.

25.1. Fibonacci Sequence. The simplest dynamic programming algorithm is computing the n th 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 $O(n)$.

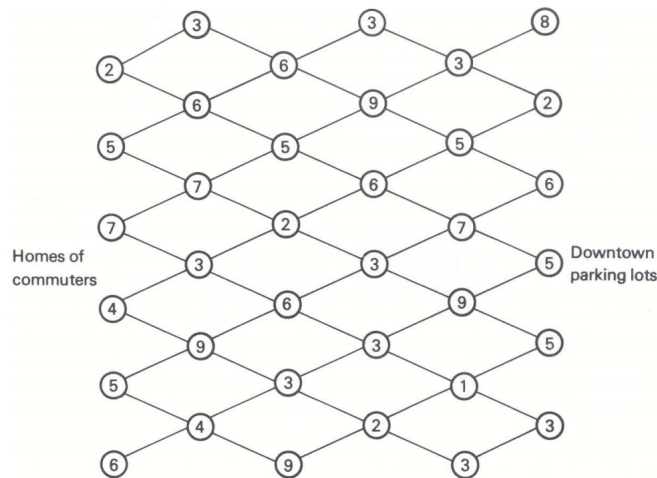
³This has gotta mean cost function, or imply that the value function is the cost function.

```

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  $F_{k+1}$ 

```

25.2. **Getting to work**⁴. 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 intersections 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

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

⁴This example is Applied Mathematical Programming by Bradley, Hax, Magnanti chapter 11.

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

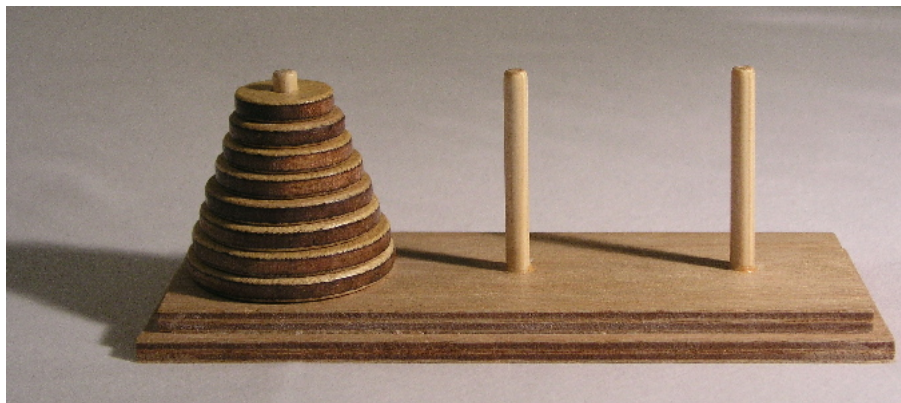
```

Shortest-Path-Parkin( $A, c$ )
1  # A is an  $n \times k$  grid (or something like that)
2  # and c is the associated costs with getting
3  # to the intersection
4   $n = \text{len}(A)$ 
5   $q = n \cdot [n \cdot [\infty]]$ 
6  for  $i = 1 : n$ :
7      # initialize first hop intersections
8       $q[1, i] = c[1, i]$ 
9  for  $i = 2 : n$ :
10     for  $j = 1 : n$ :
11         # i'll let you figure out how to
12         # bumper the q array so that
13         # when  $i, j = 1$  the entries
14         #  $q[i - 1, j - 1] = \infty$ 
15          $q[i, j] = \min \{q[i - 1, j - 1], q[i - 1, j + 1]\} + c[i, j]$ 
16  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.

25.3. 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

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

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

25.4. 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 x th 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 $x + 1, x + 2, \dots, (x + (x - 1) - 1)$, 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?

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

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

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

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 $O(nh^2)$ because of the min over $i = 1, \dots, h$. Since $W(n-1, i-1)$ is increasing in i and $W(n, h-i)$ is decreasing in i a local min of $g(i) = \max \{W(n-1, i-1), W(n, h-i)\}$ 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 problem: 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

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

25.5. Maximum Positive Subarray/Kidane's algorithm. Given $A = [a_1, \dots, a_n]$, 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 $m_{\text{here}} = 0$ then at j it's the case that $m_{\text{here}} = A[j]$. 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.

Kidane-Max-Subarray-Mod(A)

```

1  $m_{here} = m_{all} = [0], A[1]$ 
2 for  $i = 2 : \text{len}(A)$ :
3     # take max wrt. first entry of arguments, i.e.  $\max(0, m_{here} + A[i])$ 
4      $m_{here} = \max([0, [ ]], [m_{here} + A[i], m_{here}.append(A[i])], \text{key}=\text{itemgetter}(1))$ 
5      $m_{all} = \max(m_{all}, m_{here}, \text{key}=\text{itemgetter}(1))$ 
6 return  $m_{all}$ 

```

25.6. Longest increasing subsequence. A subsequence of a sequence $A = [a_1, a_2, \dots, a_n]$ need not be contiguous.

Naive solution. Check every subsequence. Running time is $O(n2^n)$.

Naive-LIS(A)

```

1 # m is max
2  $m = 0$ 
3  $seq = []$ 
4 for each subsequence  $B$  of  $A$ :
5     if  $B$  is increasing and  $\text{sum}(B) > m$ :
6          $m = \text{sum}(B)$ 
7          $seq = B$ 
8 return  $m, seq$ 

```

Recursive 1. What's a recursive definition of $\text{LIS}([a_1, \dots, a_n])$? If $\text{LIS}([a_1, \dots, a_n])$ does not contain a_n then $\text{LIS}([a_1, \dots, a_n]) = \text{LIS}([a_1, \dots, a_{n-1}])$. If $\text{LIS}([a_1, \dots, a_n])$ does contain a_n then it's a little bit unclear. One observation is that if a_n is contained in the longest sequence then all of the elements preceeding it must be smaller than it. Running time is $O(n2^n)$.

Rec1-LIS(A)

```

1  $n = \text{len}(A)$ 
2 if  $n == 0$ :
3     return 0
4 else:
5      $m = \text{Rec1-LIS}(A[1:n-1])$ 
6      $B = \text{filter}(A[1:n-1], \text{lambda } x : x \leq a_n)$ 
7     # if  $a_n$  concatenated on the end of some subset of  $B$  is bigger than
8     # just the longest increasing subsequence  $A[1:n-1]$ 
9      $m = \max\{m, 1 + \text{Rec1-LIS}(B)\}$ 
10    return  $m$ 

```

Recursive 1.1. Refactoring a little bit - call $\text{Rec1.1-LIS}(A, \infty)$ to get the solution. The run time is the same as above.

```

Rec1.1-LIS( $A, x$ )
1   $n = \text{len}(A)$ 
2  if  $n == 0$ :
3      return 0
4  else:
5       $m = \text{Rec1.1-LIS}(A[1 : n - 1], x)$ 
6      if  $A[n] < x$ :
7           $m = \max\{m, 1 + \text{Rec1.1-LIS}(A[1 : n - 1], A[n])\}$ 
8      return  $m$ 

```

Recursive 1.2. The number of subproblems produced by $\text{Rec1.1-LIS}(A, \infty)$ is $O(n^2)$. There's hope! Let $\text{LISE}([a_1, \dots, a_n])$ be the longest increasing subsequence ending at a_n . Then

$$\text{LISE}([a_1, \dots, a_n]) = 1 + \max_{i: A[i] < A[n]} (\text{LISE}([a_1, \dots, a_i]))$$

Then $\max_{i=1}^{\text{len}(A)} \text{Rec1.2-LIS}(A[1 : i])$ is the answer.

```

Rec1.2-LIS( $A$ )
1   $n = \text{len}(A)$ 
2  if  $n == 0$ :
3      return 0
4  else:
5       $m = 1$ 
6      for  $i = 1 : n - 1$ :
7          if  $A[i] < A[n]$ :
8               $m = \max\{m, 1 + \text{Rec1.2-LIS}(A[1 : i])\}$ 
9      return  $m$ 

```

Iterative with memoization. Standard thing. Running time is $O(n^2)$.

```

Iter-LIS( $A$ )
1  $n = \text{len}(A)$ 
2  $L = n \cdot [\infty]$ 
3 for  $i = 1 : n$ :
4      $L[i] = 1$ 
5     for  $j = 1 : i - 1$ :
6         if  $A[i] < A[j]$ :
7              $L[i] = \max\{L[i], 1 + L[j]\}$ 
8 return  $\max\{L\}$ 

```

Part 5. Graph Algorithms

26. REPRESENTATIONS OF GRAPHS

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

26.1. Adjacency 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  $mat = n \cdot [n \cdot [0]]$ 
4 for  $u, v, w$  in  $E$ :
5      $mat[u][v] = w$ 
6 return  $mat$ 

```

26.2. 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  $adj = n \cdot [[]]$ 
4 for  $u, v, w$  in  $E$ :
5      $adj[u].append((v, w))$ 
6 return  $adj$ 

```

27. BREADTH-FIRST SEARCH

A breadth-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 visit 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.

```

Breadth-first-search( $A, s$ )
1  # A is an adjacency list and s is a source
2  # from which to start searching. 'state' = 0 means unvisited,
3  # = 1 means visited but not explored, = 2 means explored
4   $attrs = \{i : \{'state': 0, 'd': \infty, 'prnt': None\} \text{ for } i = 1 : n\}$ 
5   $attrs[s]['state'] = 1$ 
6   $attrs[s]['d'] = 0$ 
7   $Q = [s]$ 
8  while  $\text{len}(Q) > 0$ :
9       $u = Q.\text{popleft}()$ 
10     for each  $v \in A[u]$ :
11         if  $attrs[v]['state'] == 0$ :
12              $attrs[v]['state'] = 1$ 
13              $attrs[v]['d'] = attrs[u]['d'] + 1$ 
14              $attrs[v]['prnt'] = u$ 
15              $Q.\text{append}(v)$ 
16      $attrs[u]['state'] = 2$ 

```

28. 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.

```

Depth-first-search( $A, s$ )
1   $attrs = \{i : \{\text{'state':0, 'opentime': None, 'closetime': None, 'prnt':None}\} \text{ for } i = 1:n\}$ 
2   $time = 0$ 
3   $S = [s]$ 
4  while  $\text{len}(S) > 0$ :
5       $time = time + 1$ 
6       $u = S[-1]$ 
7      # if we come back to this point in the stack after having
8      # explored the vertex that means we've visited all of its children
9      # and it's done
10     if  $attrs[u][\text{'state'}] == 2$ :
11          $attrs[u][\text{'closetime'}] == time$ 
12          $S.popright()$ 
13     # otherwise we need to leave it on the stack but explore it
14     # really this is superfluous because the only way to get on the stack
15     # is to concomitant with having 'state'=1
16     elif  $attrs[u][\text{'state'}] == 1$ :
17          $attrs[u][\text{'opentime'}] == time$ 
18         for each  $v \in A[u]$ :
19             if  $attrs[v][\text{'state'}] == 0$ :
20                  $attrs[v][\text{'state'}] = 1$ 
21                  $attrs[v][\text{'prnt'}] = u$ 
22                  $S.append(v)$ 
23      $attrs[u][\text{'state'}] = 2$ 

```
