Contents

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

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 to 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] ≤ 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 $O\left(n^2\right)$.

```
Selection-Sort(A)
1  for j = 1 to len(A):
2      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)
1  flips = True
2  while flips:
3      flips = False
4      for i = 1 to len(A) − 1:
5          if A[i] > A[i + 1]:
6              A[i], A[i + 1] = A[i + 1], A[i]
7              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 downto 1:
3      y = A[i] + x · y
```

7. 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 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  mHere = mAll = A[1]
2  for i = 2 to len(A):
3      mHere = max(0, mHere + A[i])
4      mAll = max(mAll, mHere)
5  return mAll
```

Note that if at $j-1$ the subarray was empty, and hence $maxHere = 0$ then at j it's the case that $maxHere = A[j]$. 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  mHere = mAll = [[ ], A[1]]
2  for i = 2 to len(A):
3      # take max wrt. first entry of arguments, i.e. max(0, mHere + A[i])
4      mHere = max([0, [ ]], [mHere + A[i], mHere.append(A[i])], key=itemgetter(1))
5      mAll = max(mAll, maxHere, key=itemgetter(1))
6  return mAll
```

8. Reservoir Sampling

8.1. **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 to len(A):
3      j = Random(1, i) # both ends inclusive
4      if j ≤ k:
5          R[j] = A[i]
```

8.2. **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 = Min-Priority-Queue
2  for i = 1 to k:
3      u ~ Uniform(0, 1)
4      # priority key is first entry in argument
5      H.insert(u, A[i])
6  for i = k + 1 to len(A):
7      u ~ Uniform(0, 1)
8      # H.min returns value of minimum without extracting
9      if u < H.min:
10         H.Extract-Min()
11         H.insert(u, A[i])
```

8.3. **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 to k:
    u ~ Uniform(0, 1)
    u = u^(1/A[i].weight)
    H.insert(u, A[i])
for i = k + 1 to len(A):
    u ~ Uniform(0, 1)
    u = u^(1/A[i].weight)
    if u < H.min:
        H.Extract-Min()
        H.insert(u, A[i])
```

9. 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)
m = A[1]
# these selections to count against the quota
for i = 2 to ⌈n/e⌉:
    if A[i] > m:
        m = A[i]
# this one is for keeps
for i = k + 1 to len(A):
    if A[i] > m:
        return A[i]
```

10. 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)

```
1  # men is an array of men to be matched
2  # P_m is an n × n preferences matrix for the men, sorted by increasing priority
3  # P_w is an n × n a preferences matrix for the women, sorted
4  matched_M = {}
5  matched_W = {}
6  while 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')
```
