Contents

Note

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

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 $j = 2$ to 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 $A[i] \leq$ 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 $O(n^2)$.

```
Selection-Sort($A$)
for $j = 1$ to 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 $O(n^2)$.

```
Bubble-Sort($A$)
$flips =$ True
while flips:
    $flips =$ False
    for $i = 1$ to 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:
    $h = \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor$
    $L =$ 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 $A = [a_1, \dots, a_n]$ the coefficients of a polynomial and a value x a faster way to calculate $p(x)$ is

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

```
Horners-Rule($A, x$)
$y = 0$
for $i = n$ downto 1:
    $y = A[i] + x \cdot y$
```

7. 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 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($A$)
$mHere = mAll = A[1]$
for $i = 2$ to len($A$):
    $mHere = \max(0, mHere + A[i])$
    $mAll = \max(mAll, mHere)$
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($A$)
$mHere = mAll = [[\ ], A[1]]$
for $i = 2$ to len($A$):
    # take max wrt. first entry of arguments, i.e. $\max(0, mHere + A[i])$
    $mHere = \max([0, [\ ]], [mHere + A[i], mHere$.append$(A[i])]$, key=itemgetter$(1))$
    $mAll = \max(mAll, maxHere$, key=itemgetter$(1))$
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($A, k$)
$R = [a_0, a_1, \ldots, a_k]$
for $i = k + 1$ to len($A$):
    $j =$ Random$(1, i)$ # both ends inclusive
    if $j \leq k$:
        $R[j] = A[i]$
```

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($A, k$)
$R =$ Min-Priority-Queue
for $i = 1$ to $k$:
    $u \sim$ Uniform$(0,1)$
    # priority key is first entry in argument
    $H$.insert$(u, A[i])$
for $i = k + 1$ to len$(A)$:
    $u \sim$ Uniform$(0,1)$
    # $H$.min returns value of minimum without extracting
    if $u < H$.min:
        $H$.Extract-Min()
        $H$.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 $k$:
    $u \sim$ Uniform$(0,1)$
    $u = u^{1/A[i].\texttt{weight}}$
    $H$.insert$(u, A[i])$
for $i = k + 1$ to len$(A)$:
    $u \sim$ Uniform$(0,1)$
    $u = u^{1/A[i].\texttt{weight}}$
    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 $\lceil n/e \rceil$:
    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($P_m, P_w, men$)
# $men$ is an array of men to be matched
# $P_m$ is an $n \times n$ preferences matrix for the men, sorted by increasing priority
# $P_w$ is an $n \times n$ 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**

11. Heaps

Array Heaps[1] 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 $A[i]$.

11.1. **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 $O(\lg n)$.

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

```
Max-Heapify($A, i$)
$largest = i$
# if the left child exists and is greater then potentially switch
if $2i \leq$ len$(A)$ and $A[2i] > A[i]$:
    $largest = 2i$
# if the right child exists and is greater then switch
if $2i+1 \leq$ len$(A)$ and $A[2i+1] > A[largest]$:
    $largest = 2i+1$
$A[i], A[largest] = A[largest], A[i]$
# potentially fix violation between child and one of its children
Max-Heapify($A, largest$)
```

11.2. **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($A$)
for $i = \lfloor$len$(A)/2\rfloor$ downto 1:
    Max-Heapify($A, i$)
```

11.3. **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($A$)
$m = A[1]$
$A[1] = A[-1]$
$A$.pop()
Max-Heapify($A, 1$)
return $m$
```

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

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

11.5. **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($A, i, key$)
if $key < A[i]$:
    throw Exception($key$ is smaller than current $i$ key)
$A[i] = key$
# if child is bigger then parent then swap
while $i > 1$ and $A[\lfloor i/2 \rfloor] < A[i]$:
    $A[\lfloor i/2 \rfloor], A[i] = A[i], A[\lfloor i/2 \rfloor]$
    $i = \lfloor i/2 \rfloor$
```

11.6. **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($A, key$)
$A$.append($-\infty$)
Heap-Increase-Key($A$, len($A$), $key$)
```

12. 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($(A) > 1$):
    # randomly pick a pivot
    $r = $ Random$(1,$ len$(A))$ # inclusive
    $A[r], A[-1] = A[-1], A[r]$
    $x = A[-1]$
    # partition
    $A_{left} = $ filter$(A[:-1], \lambda e : e \leq x) + [x]$
    $A_{right} = $ filter$(A[:-1], \lambda e : e > x)$
    # combine, i.e. write back to $A$
    $A[1:$ len$(A_{left}) + 1] = A_{left}$
    $A[-$len$(A_{right}) :] = A_{right}$
    # recurse
    Quicksort($A[1:$ len$(A_{left}) + 1]$)
    Quicksort($A[-$len$(A_{right}) :]$)
```

13. Order statistics

13.1. **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$(A) == 0$:
    return $A[1]$
else:
    $r = $ Random$(1, $ len$(A))$ # inclusive
    $A[r], A[-1] = A[-1], A[r]$
    $x = A[-1]$
    $A_{left} = $ filter$(A[: -1], \lambda e : e \leq x) + [x]$
    if $k == $ len$(A_{left})$
        return $x$
    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$(A[: -1], \lambda e : e > x)$
        # the kth order statistic is $(k - $ len$(A_{left}))$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$(A_{left}))] = A_{right}[1]$
        return Quickselect$(A_{right}, k - $ len$(A_{left}))$
```

13.2. **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.
    $n = \lfloor \texttt{len}(A)/5 \rfloor$
    $m_1 = \texttt{Insertion-Sort}(A[1:5+1])[3]$
    $m_2 = \texttt{Insertion-Sort}(A[5:10+1])[3]$
    $\vdots$
    $m_n = \texttt{Insertion-Sort}(A[5n:])\left[\left\lfloor \frac{\texttt{len}(A[5n:])}{2} \right\rfloor\right]$
    # recursively compute median of medians and use it as the pivot
    # after this recursive call the pivot is in position $\lfloor n/2 \rfloor$
    $\texttt{Quickerselect}([m_1, m_2, \ldots, m_n], \lfloor n/2 \rfloor)$
    $A[\lfloor n/2 \rfloor], A[-1] = A[-1], A[\lfloor n/2 \rfloor]$
    $x = A[-1]$
    $A_{left} = \texttt{filter}(A[:-1], \lambda e : e \leq x) + [x]$
    if $k == \texttt{len}(A_{left})$
        return $x$
    elif $k < \texttt{len}(A_{left})$
        return Quickselect$(A_{left}, k)$
    else:
        $A_{right} = \texttt{filter}(A[:-1], \lambda e : e > x)$
        return Quickselect$(A_{right}, k - \texttt{len}(A_{left}))$
```