Contents

Note

I have variously stolen, plagiarized, copied, etc. from many places. Rarely I cite. This is out of pure laziness on my part. For the sake of intellectual honesty consider that absolutely none of this is my own work (it's simply a collection). Maybe eventually I'll go back and cite but probably not.

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 places 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`().

The layout has large gaps intentionally. This is so pictures and diagrams follow their introductions/allusions/references in the text. That means if a picture/diagram is introduced and isn't on the page then it leads on the following page.

Part 1. Foundations

4. Divide-and-Conquer

Exercise (4.1-5)**.** Given $A = [a_1, \ldots, a_n]$, how to find the subarray with the maximum positive sum? Write a linear-time, nonrecursive algorithm for the maximum-subarray problem.

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

Problem (4-4)**.** Fibonacci numbers. Given the generating function for Fibonacci numbers

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$$

where F_i is the ith Fibonacci number

(a) Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

Solution. Let $\mathcal{F}(z) = (0, 1, 1, 2, 3, 5, 8, 13, \dots)$ the coefficients of the terms. Then multiplication by z

$$z\mathcal{F}(z) = (0, 0, 1, 1, 2, 3, 5, 8, \dots)$$

and

$$z^2\mathcal{F}(z) = (0, 0, 0, 1, 1, 2, 3, 5, \dots)$$

Hence

$$\begin{aligned} z = &\quad (0, 1, 0, 0, 0, 0, 0, 0, \dots) \\ z\mathcal{F}(z) = &\quad (0, 0, 1, 1, 2, 3, 5, 8, \dots) \\ \underline{+z^2\mathcal{F}(z)} = &\quad (0, 0, 0, 1, 1, 2, 3, 5, \dots) \\ \mathcal{F}(z) = &\quad (0, 1, 1, 2, 3, 5, 8, 13, \dots) \end{aligned}$$

(b) Show that

$$\mathcal{F}(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right)$$

where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Solution. Since

$$F(z) = z + zF(z) + z^2F(z)$$

we have that

$$F(z)\left(1 - z - z^2\right) = z$$

or

$$F(z) = \frac{z}{1 - z - z^2}$$

Factoring the denominator

$$\begin{aligned} F(z) &= \frac{z}{-\left(z + \frac{(1-\sqrt{5})}{2}\right)\left(z + \frac{1+\sqrt{5}}{2}\right)} \\ &= \frac{z}{\left(1 - z\left(\frac{1+\sqrt{5}}{2}\right)\right)\left(1 - z\left(\frac{1-\sqrt{5}}{2}\right)\right)} \\ &= \frac{z}{(1-\phi z)\left(1-\hat{\phi} z\right)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{(1-\phi z)} - \frac{1}{\left(1-\hat{\phi} z\right)}\right) \end{aligned}$$

(c) Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}\left(\phi^i - \left(\hat{\phi}\right)^i\right) z^i$$

Solution. Using the Taylor series

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n$$

we have by above

$$
\begin{aligned}
F(z) &= \frac{1}{\sqrt{5}}\left(\frac{1}{(1-\phi z)}-\frac{1}{\left(1-\hat{\phi} z\right)}\right) \\
&= \frac{1}{\sqrt{5}}\left(\sum_{n=0}^{\infty}(\phi z)^{n}-\sum_{n=0}^{\infty}\left(\hat{\phi} z\right)^{n}\right) \\
&= \sum_{n=0}^{\infty} \frac{1}{\sqrt{5}}\left(\phi^{n}-\hat{\phi}^{n}\right) z^{n}
\end{aligned}
$$

(d) Use part (c) to prove that $\{F_i\} = \phi^i/\sqrt{5}$, where $\{\}$ is rounding to the nearest integer.

Solution. By comparing coefficients in the the original generating function and the re-expression

$$F(z)=\sum_{n=0}^{\infty} F_{n} z^{n}=\sum_{n=0}^{\infty} \frac{1}{\sqrt{5}}\left(\phi^{n}-\hat{\phi}^{n}\right) z^{n}$$

we see that

$$F_{n}=\frac{1}{\sqrt{5}}\left(\phi^{n}-\hat{\phi}^{n}\right)$$

Since $\left|\hat{\phi}\right| < 1$ it's the case that $\left|\hat{\phi}^n\right| < 1$ and hence is fractional.

Problem (4-5)**.** Chip testing.

(a) Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Solution. Let g be the number of good chips and $n - g \geq g$ be the number of bad chips. Then there exists a set of good chips G and a set of bad chips B such that $|G| = |B|$. The bad chips can conspire to fool the professor in the following way: they call themselves good and the actually good chips bad. The good chips of course report exactly antisymmetrically that they're good and the bad chips are bad. Therefore these two sets of chips are indistinguishable.

(b) Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

Solution. Note that if a test is $(\text{good}, \text{good})$ then either both chips are bad or both good. Otherwise at least one is bad. Here's the Divide-and-Conquer algorithm:

(1) If there's only one chip, then it must be good.

(2) Split the chips into two-chip pairs. If the number of chips is odd let c denote the odd one out.

(3) Test each pair. If the result if $(\text{good}, \text{good})$, then throw one away, otherwise throw away both.

(4) Repeat.

The algorithm performs $\lfloor n/2 \rfloor$ pairwise tests, and keeps at most $\lceil n/2 \rceil$ chips. Now to show that at least half of the remaining chips are good each time: at a particular iteration, assume x pairs consist of two good chips, y pairs are mixed, z pairs consist of bad chips. Then there are possibilities:

- If n is even, then $g = 2x + y \geq y + 2z = b$ and $x \geq y$ implies more at least as many good chips as bad chips remain.

- If n is odd, and c is bad then $g = 2x + y \geq y + 2z + 1 = b$ and $x \geq z + 1$ (since x, z are integers). Since in fact x good chips and $z+1$ bad chips remain, it is the case that more good chips than bad chips remain.
- If n is odd, and c is good then $g = 2x + y + 1 \geq y + 2z = b$ and $x + 1 \geq z$ (since x, z are integers). Since in fact $x+1$ good chips and z bad chips remain, it is the case that more good chips than bad chips remain.

Therefore more good chips than bad chips remain always.

(c) Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Solution. Use the result of (b) to find a good chip in $\Theta(\lg n)$ time and then use it to perform the other $n-1$ comparisons.

Problem (4-6)**.** Monge arrays.

(a) Prove that an array is Monge iff for all $i = 1, \dots m-1$ and $j = 1, \dots, n-1$ we have that

$$A[i,j] + A[i+1, j+1] \leq A[i, j+1] + A[i+1, j]$$

Solution. If an array is Monge then it the property holds by defintion. Conversely suppose an $m \times n$ array has the property. We prove that

$$A[i,j] + A[i+x, j+y] \leq A[i, j+y] + A[i+x, j]$$

for all x, y such that $1 \leq x \leq m-1$ and $1 \leq y \leq n-j$, i.e. the array is Monge. For $x = y = 1$ the property holds by assumption. First suppose $x' < m$ and $y' \leq n$ and the property holds for all x, y such that $1 \leq x \leq x'$ and $1 \leq y \leq y'$. Then it holds for $x = x' + 1$ and $y =' y$: consider $i < m - x'$ and $j \leq n - y'$. By assumption we have that

$$A[i,j] + A[i+x', j+y'] \leq A[i, j+y'] + A[i+x', j]$$

and

$$A[i+x', j] + A[i+x'+1, j+y'] \leq A[i+x', j+y'] + A[i+x'+1, j]$$

Summing these two implies

$$A[i,j] + A[i+x'+1, j+y'] \leq A[i, j+y'] + A[i+x'+1, j]$$

Similarly we can argue the case for $x' \leq m$ and $y' < n$ and thus it holds for $x = x'$ and $y = y' + 1$.

(c) Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \leq f(2) \leq \cdots \leq f(m)$ for any $m \times n$ Monge array.

Solution. By contradiction: assume the inequality is false. Then there is some i such that $f(i) > f(i+1)$ such that $A[i, f(i+1)] > A[i, f(i)]$. Then

$$A[i, f(i+1)] + A[i+1, f(i)] > A[i, f(i)] + A[i+1, f(i+1)]$$

a contradiction.

(d) Here is a description of a divide-and-conquer algorithm that computes the left- most minimum element in each row of an $m \times n$ Monge array A: Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum, and its index, of the even-numbered rows is known) in $O(m+n)$ time.

Solution. Using part (c), if we know the minimum elements $f(i)$ for the even rows then for each odd $2i+1$ row we only need to check columns between $f(2i)$ and $f(2i+2)$. Hence we can compute the minima of the odd rows in time

$$\sum_{i=1}^{m/2} f(2i+2) - f(2i) + 1 = O(m+n)$$

(e) Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n\lg m)$.

Solution. The recurrence is

$$\begin{aligned} T(m) &= T(m/2) + O(m+n) \\ &= O\left(\sum_{k=0}^{\lg m}\left(\frac{m}{2^k}+n\right)\right) \\ &= O(m+n\lg m) \end{aligned}$$

Part 2. **Sorting and Order Statistics**

6. Heapsort

Exercise (6.5-7)**.** Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.

Solution. Run a timer. To construct a FIFO make the priority key the insertion time. To construct a LIFO make the priority key 1/insertion time.

Exercise (6.5-9)**.** Give an $O(n\lg k)$ algorithm for constructing a sorted array from k already sorted arrays (where the total number of elements is n).

Solution. Use a MinHeap with the extract min property: construct a MinHeap from first elements in each array. Pop the the minimum element and add to a surrogate array. Replace with the next element of the array that that one came from. This way the smallast element of each of the k arrays is always in direct competition. Constructing the initial array is $O(k)$ and then each of the Extract-min operations costs $\lg k$, hence $O(n\lg k)$.

Problem (6-3)**.** Young tableuax.

(c) Give an algorithm to implement `Extract-Min` on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time.

Solution. $Y[1,1]$ is clearly the minimum. Pop it and replace it with the bottom right element, then “percolate down”.

(d) Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

Solution. Insert at the bottom right, then “percoluate up”.

(e) Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

Solution. Repeatedly `Extract-Min`.

(f) Give an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Solution. Start at the top right, then you know everything below you is greater and everything to the left is smaller. If the number you’re looking for is smaller than the current entry then go left, and if the number is greater than the current entry then go down.

7. Quicksort

Problem (7-6)**.** Fuzzy sorting of intervals.

(a) Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.).

Solution. The key is that two intervals intersect (overlap) then they don't need to be sorted. That's where the speedup comes from. To that end here's code to compute the intersection (if any exists) of a set $I = ((a_1, b_1), \ldots, (a_n, b_n))$ of intervals

```
Intersection(I)
    i = random()
    I[-1], I[i] = I[i], I[-1]
    a, b = I[-1][1], I[-1][2]
    for i = 1 : len(I) - 1:
        if a ≤ I[i][1] ≤ b or a ≤ I[i][2] ≤ b:
            if a < I[i][1]:
                a = I[i][1]
            if B[i] < b:
                b = I[i][2]
    return a, b
```

This computes the intersection of all intervals if one exists; it does not find one! Note that $a \le I[i][1] \le b$ or $a \le I[i][2] \le b$ can be simplified down to $I[i][1] \le b \wedge I[i][1] \ge a$, since $a_i \le b_i$. Running time is clearly $\Theta(n)$.

Now using the model of `Quicksort` we can build a `Fuzzy-sort`: partition the input array into "left","middle", and "right" subarrays, where the "middle" subarray contains intervals that overlap the intersection of all of them [the intervals] and don't need to be sorted any further. Running time is $O(n \lg n)$ in general but if all of the intervals overlap then the recursion returns without executing anything and so only the `filter`s run (which are $O(n)$).

```
Fuzzy-Sort(I)
    if len(I) ≤ 1:
        return I
    else:
        a, b = Intersection(A, B)
        # first partition for similar reasons to Quicksort,
        # in order to actually sort, i.e. everything in I_right
        # follows everything in I_left in the final ordering
        # but use a as the pivot in order for the second
        # partition to be effective
        I_left = filter(I, λi : i[1] ≤ a)
        I_right = filter(I, λi : i[1] > a)
        # find all the intervals in I_left that overlap [a,b], but
        # since [a,b] is an intersection it should be
        # contained in these intervals
        # therefore everything in I_middle is such that [a,b] ⊆ [a_i, b_i]
        I_middle = filter(I, λi : b ≤ i[2])
        # and I_left-left is everything else.
        I_left-left = filter(I, λi : i[2] < b)
        return Fuzzy-Sort(I_left-left) + I_middle + Fuzzy-Sort(I_right)
```

8. Sorting in Linear Time

Exercise (8.2-4)**.** Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \ldots b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

Solution. Suppose the the cumulates array constructed by `Counting-Sort` is C. Then $C[b] - C[a]$ is the answer to the query.

Exercise (8.3-4)**.** Show how to sort n integers in the range 0 to n^3-1 in $O(n)$ time.

Solution. n base n the number $n^3 - 1$ are two digits numbers e.g. $1000_n = 1 \times n^3 + 0n^2 + 0n + 0 \times 1$. So we make 4 passes using radix sort

$$\Theta(4(n+n)) = O(n)$$

Exercise (8.4-4)**.** We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin.

Solution. A differential ring of area on the unit circle is $dA = 2\pi r dr$ so using bucket sort we can divide up the buckets according this scaling.

Exercise (8.4-5)**.** Suppose that we draw a list of n random variables $X_1, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Solution. $Y = P(X_i)$ is uniformly distributed.

Problem (8-5)**.** Jugs.

(a) Describe a deterministic algorithm that uses $\Theta\left(n^2\right)$ comparisons to group the jugs into pairs.

Solution. Test every blue jug against every red jug.

(b) Skip.

(c) Show how to match the jugs in $O(n \lg n)$ time.

Solution. You could match up the jugs by sorting each set and lining them up. Too bad you can't compare red jugs against red jugs right? But you can just use the `Quicksort` model and with blue jugs being pivots for red jugs and red jugs being pivots for blue jugs.

9. Medians and Order Statistics

Exercise (9.1-1)**.** Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Solution. Tournament style to determine minimum: comparing all pairs costs $n/2$, compare all winners of the first round costs $n/4$, etc. In total this is $n-1$ comparisons. The only way in which the second smallest element is not in the final round is it was eliminated in an earlier round. Therefore keep track of all of the elements that the smallest element "played" against, which is $\lceil \lg n \rceil$, and find the smallest of them. This costs $\lceil \lg n \rceil - 1$ comparisons.

Exercise (9.3-5)**.** Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Solution. Suppose the rank you're looking for is r and the number of elements is n. Use binary search: find the median, then if the order statistic is higher than the median find the $r - \lfloor n/2 \rfloor$ order statistic of the elements larger than the median, and if the order statistic is lower then find the rank r statistic of the elements smaller than the median, and so on.

Exercise (9.3-6)**.** The kth quantiles of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

Solution. If k is even then there are $k-1$ (an odd number) of "pivots" and one of them is the median. Find the median, partition, then solve the subproblems. If k is odd do the same thing but be more careful.

Exercise (9.3-7)**.** Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

Solution. Find the median, then subtract the median from every element, then find the kth order statistic (and in doing so partition).

Exercise (9.3-8)**.** Let $X[1 \ldots n]$ and $Y[1 \ldots n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Solution. The median all of $2n$ elements is always in between the median of each array (by value). Compute the medians in $O(1)$ time. If they're equal return them. Otherwise recurse to either the leftside or rightside of each array depending on which median is larger than which.

Exercise (9.3-9)**.** Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Solution. The median is the element that minimizes the L_1 norm, i.e. the sum of distances.

Problem (9-2)**.** Weighted median.

(a) Argue that the median of $x_1, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, \ldots, n$.

Solution. This is trivially true (algebraically).

(b) Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

Solution. Sort then sum weights, in order of increasing elements, until you exceed $1/2$.

(c) Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as `SELECT` from Section 9.3.

Solutio n. Call $t = 1/2$ the target. Find the median (and in doing so partition around it) and compute the sum of the weights in the "lower" half. If they sum to t then return the median. If the exceed then compute the median in the "lower" half. If the sum is less than t then compute the median in the "top half" but with target being t minus the sum you just computed.

(d) Argue that the weighted median is a best solution for the 1-dimensional post- office location problem, in which points are simply real numbers and the dis- tance between points a and b is $d(a, b) = |a - b|$.

Solution. This is true for the same reason the median minimizes the L_1 norm.

(e) Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_{,1})$ and $b = (x_2, y_2)$ is the Manhattan distance given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Solution. Since the components of the distance "vector" are decoupled you can just do median in each coordinate, i.e. take the median of all of the x_i and the median of all of the y_i.

10. Elementary Data Structures

Exercise (10.1-2)**.** Explain how to implement two stacks in one array $A[1 \ldots n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The `PUSH` and `POP` operations should run in $O(1)$ time.

Solution. Have one stack grow from the left side of the array and the other stack from the right side of the array (store their ends). When they collide then they're both "full".

Exercise (10.1-5)**.** Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a deque (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

Solution. Either use a circular list with the head and tail linked or an array using mod to update indices that keep track of the back and front and checking for collision of the indices.

Exercise (10.1-6)**.** Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Solution. Call the two stacks "inbox" and "outbox". Push to one stack and pop from the other. If the "outbox" stack is empty then pop everything from the inbox stack and push to outbox stack. They'll be pushed in reverse order and pops will produce the correct behavior. Amortized time is $O(n)$.

Exercise (10.1-7)**.** Show how to implement a stack using two queues. Analyze the running time of the stack operations.

Solution. Exactly like 10.1-6.

Exercise (10.4-2)**.** Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

Solution. DFS (or BFS).

Exercise (10.4-3)**.** Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

Solution. This is of course DFS but written iteratively. Assume the tree is represented by a `dict()` with `'leftchild'` and `'rightchild'` keys (whose corresponding values are the nodes). Since we're traversing a tree we don't need to check for backpointers (i.e. don't need to mark visited).

```
DFS(T)
    stk = [T]
    while len(stk) > 0:
        next = stk.pop()
        if next is not None:
            print(next)
            stk.append(next['leftchild'])
            stk.append(next['rightchild'])
```

11. Hash Tables

Exercise (11.1-2)**.** A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

Solution. Assign an index to each element. Use the bit vector to indicate membership by settings bits by initializing all bits to 0 and then setting to 1 when inserting (deletion is reseting to 0).

Exercise (11.1-3)**.** Suggest how to implement a direct-address table in which the keys of stored el- ements do not need to be distinct and the elements can have satellite data. All three dictionary operations (`INSERT`, `DELETE`, and `SEARCH`) should run in $O(1)$ time. Don't forget that `DELETE` takes as an argument a pointer to an object to be deleted, not a key.

Solution. Use chaining. For `DELETE` set the pointer to a `NULL`.

Exercise (11.1-4)**.** We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations `INSERT`, `DELETE`, and `SEARCH` should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. Hint: Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.

Solution. Use the stack to store the actual values. Let *huge* be the array and *stack* be the stack. To insert an element with key x append x to the stack and store in $huge[x]$ the length of the stack (i.e. the index of x in the stack). To search for an element y (i.e. verify membership) check that $huge[y] \leq \texttt{len}(stack)$ and that $stack[huge[y]] = y$. To delete an element x swap the top of the stack with the element to be deleted and update relevant entries in *huge*:

$$\begin{aligned} stack[huge[x]] &= stack[-1] \\ huge[stack[-1]] &= huge[x] \\ huge[x] &= \texttt{None} \end{aligned}$$

and pop the stack.

Exercise (11.2-5)**.** Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining $\Theta(n)$.

Solution. Pigeonhole principle.

Exercise (10.4-2)**.** Suppose we have stored n keys in a hash table of size m, with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$, where $\alpha = n/m$.

Solution. The keyword being randomly (not just any). Pick a random bucket, which will have k elements, then pick an index i from $i, \dots, L$. Reject if $i > k$ and draw i again. This is essentially rejection sampling the array, i.e. how to uniformly random pick an element from $1, \dots, k$ if you can only generate random numbers from 1 to L. The expected number of elements k is equal to the load and so

$$P(i \leq k) = \frac{(n/m)}{L}$$

and hence expected number of times before success is

$$\frac{1}{\frac{(n/m)}{L}} = \frac{L \cdot m}{n}$$

(since "success" is a geometric random variable with probability of success being $P(i \leq k)$). Picking the initial bucket doesn't "cost" anything, hence combined with time L to traverse we get

$$O\left(L + L\frac{m}{n}\right) = O\left(L \cdot \left(1 + \frac{1}{\alpha}\right)\right)$$

expected running time.

Exercise (10.4-3)**.** Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

Solution. This is of course DFS but written iteratively. Assume the tree is represented by a `dict ()` with `'leftchild'` and `'rightchild'` keys (whose corresponding values are the nodes). Since we're traversing a tree we don't need to check for backpointers (i.e. don't need to mark visited).

```
DFS(T)
    stk = [T]
    while len(stk) > 0:
        next = stk.pop()
        if next is not None:
            print(next)
            stk.append(next['leftchild'])
            stk.append(next['rightchild'])
```

Problem (9-2)**.** Weighted median.

(a) Argue that the median of $x_1, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, \ldots, n$.

Solution. This is trivially true (algebraically).

(b) Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

Solution. Sort then sum weights, in order of increasing elements, until you exceed $1/2$.

(c) Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as `SELECT` from Section 9.3.

Solutio n. Call $t = 1/2$ the target. Find the median (and in doing so partition around it) and compute the sum of the weights in the "lower" half. If they sum to t then return the median. If the exceed then compute the median in the "lower" half. If the sum is less than t then compute the median in the "top half" but with target being t minus the sum you just computed.

(d) Argue that the weighted median is a best solution for the 1-dimensional post- office location problem, in which points are simply real numbers and the dis- tance between points a and b is $d(a, b) = |a - b|$.

Solution. This is true for the same reason the median minimizes the L_1 norm.

(e) Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_{,1})$ and $b = (x_2, y_2)$ is the Manhattan distance given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

Solution. Since the components of the distance "vector" are decoupled you can just do median in each coordinate, i.e. take the median of all of the x_i and the median of all of the y_i.