Exercises 1

2.4

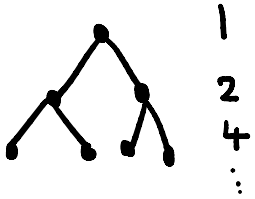
$A: T(n) = 5T(n/2) + \Theta(n)$

By the Master Theorem, we have that 5 > 2^1, so

$T(n) \in \Theta(n^{\log_2(5)})$

$B: T(n) = 2T(n-1) + \Theta(1)$

We draw the recursion tree for this problem.



The amount of work at level k is 2^k. The tree has depth n. Hence we have that

$T(n) = \sum_{k=0}^{n} 2^k = 2^{n+1} - 1 \in \Theta(2^n)$

$C: T(n) = 9T(n/3) + O(n^2)$

By the Master Theorem, we have that 9 = 3^2, so

$T(n) \in O(n^2 \log n)$

I would choose **Algorithm C**.

2.17

The following pseudocode describes our algorithm (similar to binary search):

```
function findIndex(A, start, end) {
    if (start > end) return false
    if (start == end)
        if (A[start] == start) return true
        else return false

    mid = (start + end)/2
    if (mid == A[mid])
        return true
    else if (mid < A[mid])
        return findIndex(A, start, mid-1)
```

```
        else
                return findIndex(A, mid+1, end)
}
```

Runtime: We look at the runtime for binary search. In each iteration, we cut into one subproblem of size n/2. Other operations take O(1) time. Hence we have the recurrence relation

$$T(n) = T(n/2) + O(1)$$

By the Master Theorem, we have that 1 = 2^0. Hence we have that

$$T(n) \in O(\log n)$$

Correctness: We prove correctness of this algorithm using strong induction on n (end-start), the size of the range of array we are searching on.

Base case: We have that if there is one element in the array, start == end and we check the element A[0]. If it is equal, we return true, otherwise we return false.

Inductive step: Assume that the algorithm returns correct inputs for all k < n. Consider the algorithm with an input of size n (end-start == n).

There are 3 cases:

For mid == A[mid], we must be returning the correct answer because we have found an index i where A[i] == i.

For mid < A[mid], since the array is sorted and distinct, for all indices i >= mid, we must have that A[i] > i. Then the solution cannot be in the right half of the array, so it must be in the left half. We search the left half of the array, and by the inductive hypothesis, since n/2 < n, return the correct solution.

For mid > A[mid], since the array is sorted and distinct, for all indices i <= mid, we must have that A[i] < i. Then the solution cannot be in the left half of the array, so it must be in the right half. We search the right half of the array, and by the inductive hypothesis, since n/2 < n, return the correct solution.

2.28 We first split v into two vectors v1 and v2, each of length n/2 = 2^(k-1), where

$$v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

Then we have that

$$H_k v = \begin{pmatrix} H_{k-1} v_1 + H_{k-1} v_2 \\ H_{k-1} v_1 - H_{k-1} v_2 \end{pmatrix}$$

It takes T(n/2) time to compute Hk-1v1, and T(n/2) time to compute Hk-1v2. The addition and subtraction takes O(n) operations. Hence we have

$$T(n) = 2T(n/2) + O(n)$$

By the Master Theorem, we hav that 2 = 2^1, so

$$T(n) \in O(n \log n)$$

2.29 a) We use induction on n to show that this algorithm evaluates the polynomial correctly. [I don't really like this proof but couldn't think of a better way to word it]

Base case: n=0. Then we have p(x) = a0. For this case, the loop is not executed and Horner's rule evaluates this correctly.

Inductive step: Assume the algorithm evaluates the polynomial correctly for n. Consider the polynomial with degree n+1. Then we have that the for loop will execute n+1 times. In the first n iterations, the algorithm computes z = a1 + a2x + … anx^(n-1). On the last iteration, the algorithm computes a0 + zx = a0 + a1x + … anx^n, which is the desired result. Hence we have proven the statement.

b) This routine uses n additions and n multiplications, since in each iteration of the for loop we have 1 addition and 1 multiplication, and we iterate n times. This is optimal.

2. We use the same algorithm as counting inversions, except when we are counting transverse inversions, we multiply every number in the second array by 2 before using mergesort to count inversions. Our algorithm is exactly the same except we multiple half the array by 2 in each iteration, which takes O(n) time. Hence the recurrence is still T(n) = T(n/2)+O(n), so the runtime is O(nlogn).

Thus our algorithm works as follows. Given the array, we split the array in half. We find the number of significant inversions in the left half, then the number of significant inversions in the right half. Now we want to find the number of transverse significant inversions. We first sort the left half, and sort the right half of the array. We multiply each number in the right half of the array by 2. Then we use mergesort to count the number of significant inversions, as demonstrated in the following pseudocode:

```
mergeAndCount(A) {
    result = 0
    S = A // Copy A into new array S
    i = 0, j = n/2
    for (k = 0, k < n, k++)
        if (i > n/2) A[k] = S[j++]
        else if (j > n) A[k] = S[i++]; result += n/2
        else if (S[i] < S[j]) A[j] = S[i++]; c += j - n/2
        else A[k] = S[j++]
}
```

2.22 We use a divide and conquer approach. The pseudocode below describes the algorithm.

```
kthSmallest(arr1, arr2, k) {
    if (arr1.length == 0) return arr2[k]
    if (arr2.length == 0) return arr1[k]

    mid1 = arr1.length / 2
    mid2 = arr2.length / 2

    if (mid1 + mid2 < k)
        if (arr1[mid1] > arr2[mid2])
            return kthSmallest(arr1, arr2[mid2+1 : end], k - mid2 - 1)
        else
            return kthSmallest(arr1[mid1+1 : end], arr2, k - mid1 - 1)
    else
        if (arr1[mid1] > arr2[mid2])
            return kthSmallest(arr1[0 : mid1], arr2, k)
        else
            return kthSmallest(arr1, arr2[0 : mid2], k)
}
```

Runtime: In each iteration, we halve the size of either arr1 or arr2. Hence the runtime is in O(logm + logn).

Correctness: In each iteration, there are 2 cases:

mid1 + mid2 < k:
- arr[mid1] > arr2[mid2]:
  - None of the elements in the left half of array2 can be the kth smallest, since k is larger than the sum of the two middle indices, and arr[mid1] > arr[mid2].
- arr[mid1] <= arr2[mid2]:
  - None of the elements in the left half of array1 can be the kth smallest, since k is larger than the sum of the two middle indices, and arr[mid1] <= arr[mid2].

mid1 + mid2 >= k:
- arr[mid1] > arr2[mid2]:
  - None of the elements in the right half of array1 can be the kth smallest, since k is smaller than the sum of the two middle indices, and arr[mid1] > arr[mid2].
- arr[mid1] <= arr2[mid2]:
  - None of the elements in the right half of array2 can be the kth smallest, since k is smaller than the sum of the two middle indices, and arr[mid1] <= arr[mid2].

2.23 a) We split the array into two halves A1 and A2, and compute the majority elements of A1 and A2: let these be m1 and m2. If m1 = m2, we have that this must the majority element of the entire array, so we return it. Otherwise, we calculate the frequencies of m1 and m2 in the array to determine if either is the majority element. The recurrence for this is thus $T(n) = 2T(n/2) + O(n)$, which resolves to O(nlogn).

b) We pair up the elements of A arbitrarily to get n/2 pairs. For each pair, if the elements are different, we discard both; if they are the same, we keep just one. Since each pair is either completely discarded or only one element is kept, there are at most n/2 elements left.

We show that if A has a majority element, the majority element of the remaining elements has the same majority element.

Suppose A has a majority element m. There must be at least one pair where both of the elements are m, since the frequency of m is greater than n/2. For all pairs (p, q), there are 4 possibilities:

p = q != m
p = q = m
p != q and p != m and q != m
p = m and q != m (WLOG)

Each pair that satisfies one of the latter 3 equations maintains the majority of m, since we either have a pair of ms, remove two non-m elements, or remove one m and one non-m element. For each pair that satisfies the first equation, we keep one non-m element. However, every pair of non-m elements creates one m pair, so we still maintain majority of m in the remaining array.

We can just run a check on the majority element of the remaining elements for its frequency: if it's not a majority element of the original array we return that A has no majority element.

9-2 a) Let m = ceil(n/2) - 1. Then xm is the median of x1, x2,…xn. Then we have

$$\sum_{x_i < x_m} w_i = \sum_{i=1}^{m} \frac{1}{n} = \frac{m}{n} < \frac{1}{2}$$

$$\sum_{x_i > x_k} w_i = \sum_{i=m}^{n} \frac{1}{n} = \frac{n-m-1}{2} \leq \frac{1}{2}$$

Hence the median of the elements is also the weighted median.

b) We sort the elements by their values. This takes O(nlogn) time. We then sum the weights of starting from the first element of this sorted array until we reach 1/2. Let k be the index of this element. Then we have that xk is the weighted median.

c) We modify SELECT as follows, where we are trying to find the weighted median such that

$$\sum_{x_i < x} w_i < x_1 \qquad \sum_{x_i > x} w_i \geq x_2$$

Initially, we pass in x1 = 1/2 and x2 = 1/2.

function weightedMedian(A, x1, x2) {

```
        m = medianOfMedians(A)   // Approximate median
        partition(A, m)          // Partition array A at pivot m

        a = sum(A[0 : m])        // sum of elements in A on left half of pivot
        b = sum(A[m : end])      // sum of elements in A on right half of pivot

        if (a > x1)
            return weightedMedian(A[0 : m], x1, x2 - b)     // Recurse in left half
        else if (b > x2)
            return weightedMedian(A[m : end], x1 - a, x2)  // Recurse in right half
        else
            Return m
}
```

Runtime: It takes O(n) time to find the median of medians, partition, and calculate the sums of the left and right halves. In each iteration, we reduce to a subproblem of at most 7n/10. Hence, the runtime is the same as the SELECT algorithm, which is O(n)

d) Suppose that the weighted median is not the best solution. Let point p be the best solution for some set of points p1…pn. Let pm be the weighted median of this set of points. Then we have that

$$\sum_{i=1}^{n} w_i d(p_m, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + (p_m - p)\left(\sum_{p_i<p} w_i - \sum_{p_i>p} w_i\right) < \sum_{i=1}^{n} w_i d(p, p_i)$$

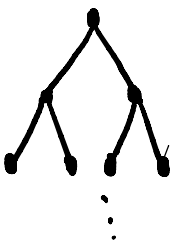
This is a contradiction, since we stated point p is the best solution. Hence the weighted median must be the best solution.

e) We can split the sum as follows:

$$\sum_{i=1}^{n} w_i d(p, p_i) = \sum_{i=1}^{n} w_i |p_x - p_{ix}| + \sum_{i=1}^{n} w_i |p_y - p_{iy}|$$

From part d, we have that the best solution to the 1D problem is the weighted median. Since our 2D sum is a sum of two 1D problems, the best solution must be (px, py), where px is the weighted median of the x-coordinates and py is the weighted median of the y-coordinates.

4.4-5



n

$n-1+\frac{n}{2}$

$n-2+\frac{n-1}{2}+\frac{n}{2}-1+\frac{n}{4}$

Our recursion tree has depth n. The amount of work on level k is

$$n2^k + \frac{4^k - 3^k}{2^{k-1}}$$

Hence we have that

$$T(n) = \sum_{k=0}^{n} n2^k + \frac{4^k - 3^k}{2^{k-1}}$$

$$= (2^{n+1} - 1)n - 2^{1-n} \cdot 3^{n+1} + 2^{n+2} + 2$$

$$\in O\left(n\left(\frac{3}{2}\right)^n\right)$$

[courtesy of Evan]