CS 240 Tutorials

1.1 We find the upper bound:

$$\sum_{i=1}^{n} \frac{1}{(2i)!} \leq \sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^3}{6} \in O(1)$$

We find the lower bound:

$$\sum_{i=1}^{n} \frac{1}{(2i)!} \geq \sum_{i=1}^{n} \frac{1}{(2n)!} = \frac{n}{(2n)!} \in \Omega(1)$$

1.2 We first show that n >= csqrt(n) for n >= c^2.

$$n \geq c\sqrt{n}$$

$$\sqrt{n} \geq c$$

$$n \geq c^2$$

We then compare csqrt(n) and c2^sqrt(logn)

$$c\sqrt{n} \geq c\,2^{\sqrt{\log n}}$$

$$\sqrt{n} \geq 2^{\sqrt{\log n}}$$

$$\log \sqrt{n} \geq \log\left(2^{\sqrt{\log n}}\right)$$

$$\frac{1}{2}\log n \geq \sqrt{\log n}$$

$$\sqrt{\log n} \geq 2$$

$$\log n \geq 4$$

$$n \geq 16$$

Hence we have that for all constants c, n >= c2^sqrt(logn) for all n >= max(c, 16).

1.3 We have that for some constants c and n0, for all n >= n0,

$$f(n) \leq c\, h_1(n)$$

We also have that for all constants c1, there exists an n1 such that for all n >= n1,

$$g(n) \geq c_1 h_2(n)$$

Hence we have that for constants c and n0, for all n >= n0 and c1 > 0,

$$\frac{f(n)}{g(n)} \leq \frac{c\, h_1(n)}{g(n)} \leq \frac{c\, h_1(n)}{c_1 h_2(n)}$$

Let c2 = c/c1. Hence we have that for all constants c2 > 0 and n0 >= 0, for all n >= n0,

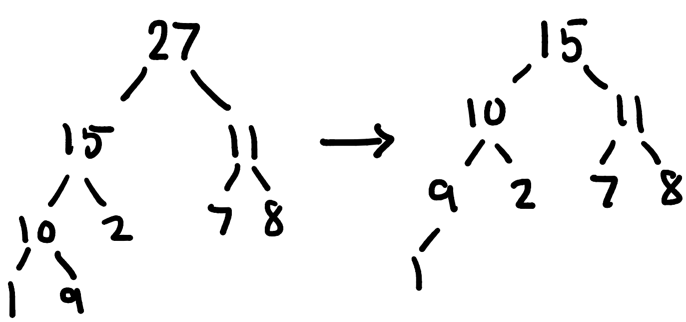
$$\frac{f(n)}{g(n)} \leq c_2 \frac{h_1(n)}{h_2(n)}$$

1.4 We have that the for loop runs n times and the while loop runs n/k + 1 times for each k=2^(i-1). Assume all constant-time lines take 1 time. Hence the runtime of the pseudocode is

$$\sum_{i=1}^{n} \frac{n}{k} + 1 = n + \sum_{i=1}^{n} \frac{n}{2^{i-1}} = n + n\sum_{i=1}^{n} \frac{1}{2^{i-1}} = n + n\sum_{i=0}^{n-1} \frac{1}{2^{i}}$$

$$= n + n\frac{1-\left(\frac{1}{2}\right)^n}{1-\frac{1}{2}} \in \Theta(n)$$

2.1



2.2 Whenever we push an item, we its priority will be higher than any other elements in the stack. We can implement this by making the priority of elements we insert 1 + biggest current priority (which is at the root). Inserting it is the same as inserting into a heap normally, which takes O(logn) time. Popping from the stack is just removing the element with the highest priority, which is at the root, and takes O(logn) time.

2.3

L[i]/L[j] must be the kth smallest fraction, so we must have that 0 <= i < k and n-k <= j < n. There are hence k choices for the numerator and k choices for the denominator.

We can build a min-heap of the k smallest elements. We start with all the fractions of numerator L[0] with every possible denominator L[n-k] to L[n-1]. Building this takes O(k) time. Now, we repeatedly call deleteMin. Each time we delete the fraction L[i]/L[j], we replace it with L[i+1]/L[j] (the next smallest element yet to be inserted), and fix-down to restore the heap-order property. Each iteration of this takes O(log k) time. We iterate k times, to find the kth smallest fraction, so the runtime is O(klogk).

```
heap = Minheap()

for i=0 to min(k, n)
    heap.insert(key=L[0]/L[n-i], (numeratorIndex=0, denominatorIndex = n-i))

for i=0 to k-1
    smallest = heap.deleteMin()
    i = smallest.numeratorIndex+1
    j = smallest.denominatorIndex

    if (i < j)
        heap.insert(key=L[i]/L[j], (numeratorIndex=i, denominatorIndex = j))
```

3.1 The shuffling done in the first step makes all permutations of A equally likely. We have that (i,j) is an inversion if A[i] > A[j]. Hence swapping an adjacent pair of positions that are out of order decreases the number of inversions by exactly 1. Hence the number of swaps performed in line 8 is the number of inversions in A after shuffling. Let Xij be an indicator variable that is 1 if A[i] and A[j] form an inversion, and 0 otherwise. There are (n choose 2) pairs i,j. We have that the probability that A[i] and A[j] are an inversion is 1/2. Hence we have

$$E\left(\sum X_{ij}\right) = \sum E(X_{ij}) = \sum \frac{1}{2} = \frac{1}{2}\binom{n}{2} = \frac{1}{4}n(n-1) \in \Theta(n^2)$$

3.2 The first n-n^e items are sorted. We just need to sort the latter n^e items. We can use mergesort or heapsort. We have that

$$n^{\epsilon}\log(n^{\epsilon}) = \epsilon n^{\epsilon}\log(n) \in O(\epsilon n^{\epsilon} n^{1-\epsilon})$$ since log(n)^d in o(n^d) for any c>0 and d>0

$$\in O(n)$$

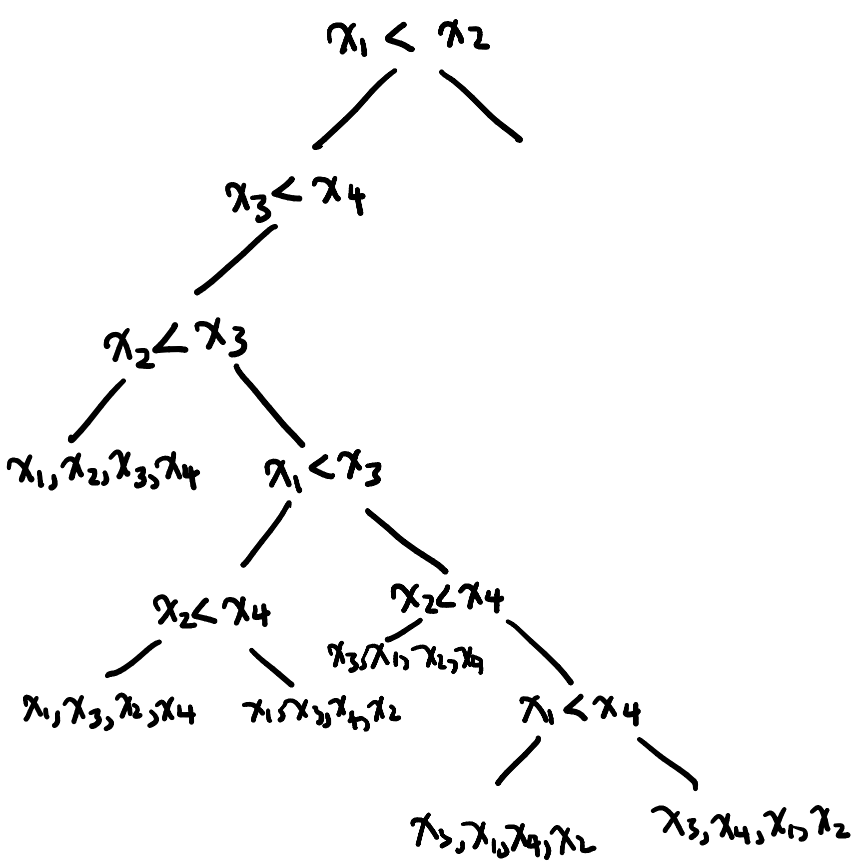
3.3

We calculate the expected runtime by enumerating the first index of A and B that is different. There are 2^n * 2^n possible combinations of strings A and B. For each possible index 0 <= i <= n, we have that the first i bits of A and B must be fixed. There are 2^n possible strings for A. For that A, there are 2^(n-i-1) possible bits for the last n-i bits of B (since the first i bits are the same as A). Assume all constant-time lines take 1 time. Hence we have that

$$T(n) = \frac{1}{2^n \cdot 2^n}\left[\left(\sum_{i=0}^{n-1} (i+1)\, 2^n \cdot 2^{n-i-1}\right) + 2^n(n+1)\right]$$

$$= \left[\sum_{i=0}^{n-1} \frac{i+1}{2^{i+1}}\right] + \frac{n+1}{2^n} \leq \frac{n+1}{2^n} + \sum_{i=0}^{\infty} \frac{i}{2^i} \in \Theta(1)$$

4.1



4.2

We can use LSD radix sort. We start by applying bucket sort on the least significant digit of each number. We repeat this for each digit, treating a non-existent digit as 0. Since bucket sort is stable, this will sort all the integers. Each digit in each integer is examined exactly once, so the problem is O(n+l), and since n < l, we have that this takes O(l) time.

4.3 The best-case runtime for MonkeySort is O(n) - when the first iteration produces a sorted array.

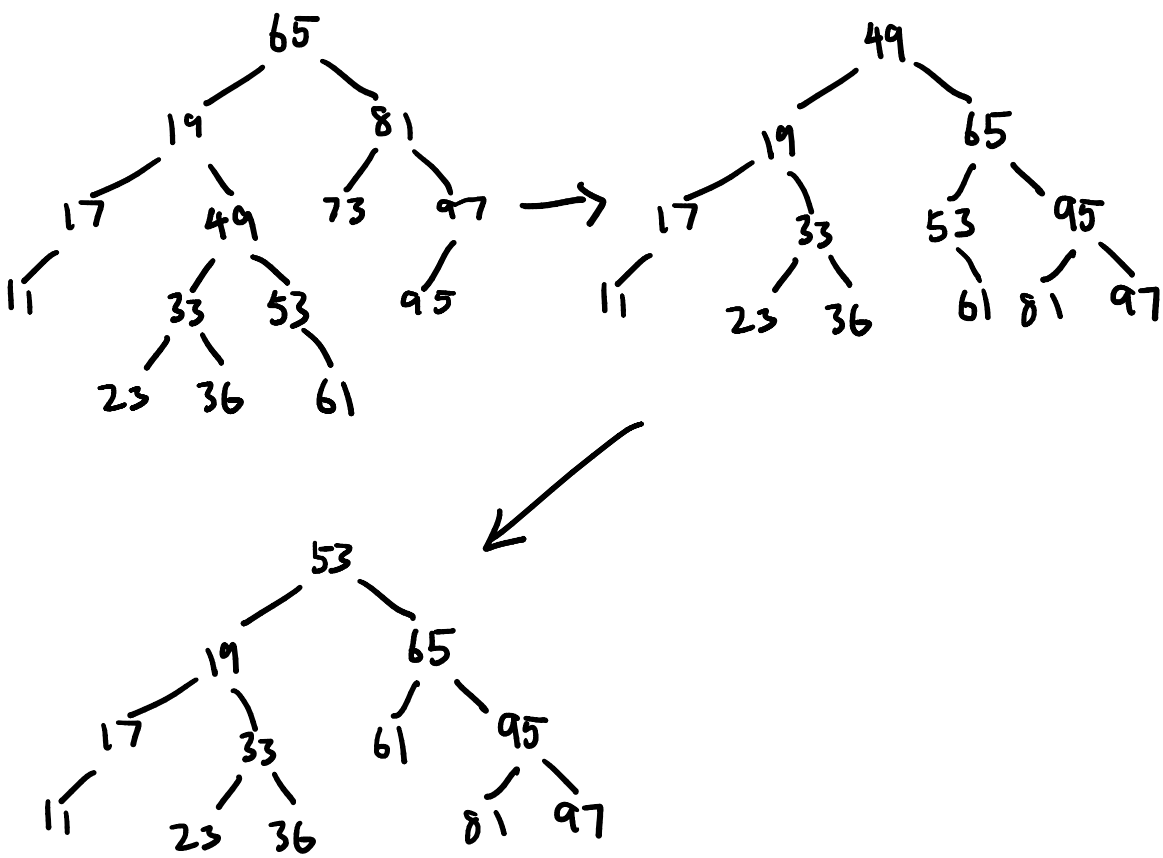
There are n! possible permutations of the array, and only 1 produces the sorted order. Assume all constant-time lines take 1 time. We can hence describe the expected runtime as

$$T(n) = 2n + \frac{1}{n!} + \frac{n!-1}{n!} T(n) = 2n + \frac{1}{n!} + T(n) - \frac{1}{n!} T(n)$$

$$\frac{1}{n!} T(n) = 2n + \frac{1}{n!}$$

$$T(n) = 2n \cdot n! + 1 \in O(n \cdot n!)$$

5.1



5.2

$-\infty$						∞
$-\infty$					20	∞
$-\infty$			12		20	∞
$-\infty$		11	12	13	20	∞
$-\infty$	10	11	12	13	20	∞

$-\infty$				∞
$-\infty$			12	∞
$-\infty$		11	12	∞
$-\infty$	10	11	12	∞

5.3

Let h represent the height of the tree and n represent the number of nodes. For h to be in O(log n), we need there to exist constants c and n0 such that h <= clog(n) for all n >= n0. Instead of showing this, we can let N(h) denote the minimum number of nodes in a tree of height h, and show that N(h) >= 2^(h/c) for all N(h) >= n0.

We have that N(h) = 0 for h = -1 (empty tree), N(h) = 1 for h = 0 (tree with one node), N(h) = 2 for h = 1.

For an AVL tree with more than 2 nodes, we have a root node and two subtrees. At least one of the subtrees must have height h-1 for the tree to be of height h - the minimum number of nodes in this subtree is N(h-1). The smallest possible height for the other subtree is h-3, so the minimum number of nodes in it is N(h-3). Hence we have

N(h) = 1 + N(h-1) + N(h-3), h >= 2

We prove that there exists some c > 0 and n0 > 0 such that N(h) >= 2^(h/c) for all N(h) >= n0 by induction. For c=3 and n0 = 1:

Base cases:

$$N(0) = 1 \geq 1 = 2^{0/3}$$
$$N(1) = 2 \geq 2^{1/3}$$
$$N(2) = 1 + N(1) + N(-1) = 1 + 2 + 0 = 3 \geq 2^{2/3}$$

Assume for all k < h, we have that N(k) >= 2^(k/c). Then we have

$$\begin{aligned} N(h) &= 1 + N(h-1) + N(h-3) \\ &\geq 1 + 2^{\frac{h-1}{c}} + 2^{\frac{h-3}{c}} = 1 + 2^{h/c}(2^{-1/c} + 2^{-3/c}) \\ &\geq 1 + 2^{h/c}(2^{-3/c} + 2^{-3/c}) = 1 + 2^{h/c}(2^{1-3/c}) \\ &\geq 2^{h/c}(2^{1-3/c}) \end{aligned}$$

$$\geq 2^{h/c}(2^{1-3/c})$$

↳ $2^{h/c}(2^{1-3/c}) \geq 2^{h/c}$ if $2^{1-3/c} \geq 1$

$c \geq 3$

^^ This is how we got c=3.

$$\geq 2^{h/c}$$

Hence we have h <= 3log(n) for all n >= 1, and hence h is in O(logn).