Problem Set 1

1. 1. $\frac{1}{n^3}$
   2. 0.0003
   3. $\log \log n$
   4. $\sqrt{n} \log n$
   5. $\frac{n}{\log n}$
   6. n^7
   7. $2^{n^{\frac{2}{3}}}$
   8. $2^{\frac{n}{\log n}}$
   9. $(\sqrt{n})^n$
   10. n^n

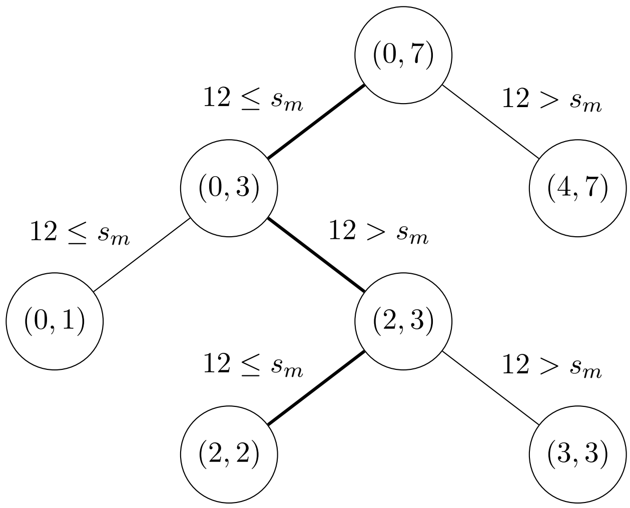
2. Let s be the sequence of integers, and s_i be the i^{th} element of s.

   Each node of the binary search tree works with a tuple, (ℓ, h), representing the highest and lowest index of the array, respectively.

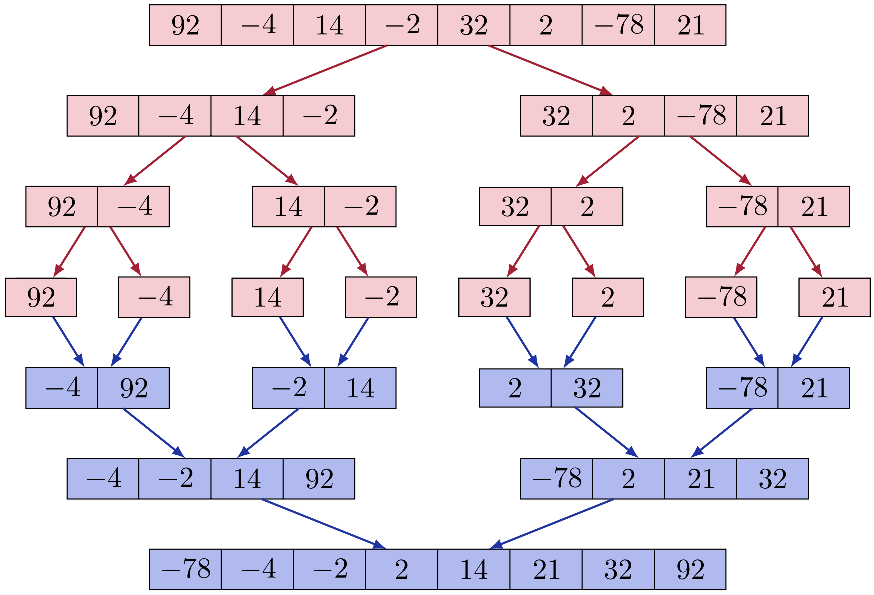
   At each node, there is an implicit halting check that ensures $h \neq \ell$.

   The midpoint, m, is calculated as follows:

$$m = \left\lfloor \frac{h + \ell}{2} \right\rfloor$$



3. The sequence of operations in `mergesort` is as follows:



4. 1. **Proposition.** $(f = O(g) \implies \exists a > 0 \mid \log_a f = O(\log_a g)) = \text{TRUE}.$

   Proof. By definition,

   $$f = O(g) \implies \exists c > 0 \mid f \leq c \cdot g.$$

   Since f and g are asymptotically positive and $a > 0$, we can take $\log_a$ on both sides and preserve the inequality,

   $$\begin{aligned} \log_a f &\leq \log_a(c \cdot g) \\ &\leq \log_a c + \log_a g. \end{aligned}$$

   Again, by definition, we can say,

   $$\begin{aligned} \log_a f &= O(\log_a c + \log_a g) \\ &= O(\log_a g). \end{aligned}$$ □

   2. **Proposition.** $\big(f = O(g) \implies 2^f = \Omega(2^g)\big) = \text{FALSE}.$

   Proof. Counterexample.
   Let $f = 1$ and $g = n$. Thus,

   $$2 \neq \Omega(2^n).$$ □

   3. **Proposition.** $\big(f = O(g) \implies \sqrt{f} = O(\sqrt{g})\big) = \text{TRUE}.$

   Proof. By definition,

   $$f = O(g) \implies \exists c > 0 \mid f \leq c \cdot g.$$

   Since f and g are asymptotically positive, we can take $\sqrt{\cdot}$ on both sides and preserve the inequality,

   $$\begin{aligned} \sqrt{f} &\leq \sqrt{c \cdot g} \\ &\leq \sqrt{c} \cdot \sqrt{g}. \end{aligned}$$

   Again, by definition, we can say,

   $$\begin{aligned} \sqrt{f} &= O(\sqrt{c} \cdot \sqrt{g}) \\ &= O(\sqrt{g}). \end{aligned}$$ □

4. PROPOSITION. $\big(f = O(g) \implies 10^{100} f = O(10^{-100} g)\big) = \text{TRUE}.$

Proof. By definition,

$$f = O(g) \implies \exists c > 0 \mid f \leq c \cdot g.$$

We can rewrite this as,

$$\begin{aligned} f &\leq 10^{100} \cdot 10^{-100} \cdot c \cdot g \\ &\leq 10^{100} c \cdot 10^{-100} g. \end{aligned}$$

Now, we multiply both sides by 10^{100},

$$\begin{aligned} 10^{100} f &\leq 10^{100} \cdot 10^{100} c \cdot 10^{-100} g \\ &\leq 10^{200} c \cdot 10^{-100} g. \end{aligned}$$

Again, by definition, we can say,

$$\begin{aligned} 10^{100} f &= O(10^{200} c \cdot 10^{-100} g) \\ &= O(10^{-100} g). \end{aligned}$$ □

5. PROPOSITION. $\Big(f = O(g) \implies 2^{\frac{n}{1+\log f}} \neq O\big(2^{\frac{n}{1+\log g}}\big)\Big) = \text{FALSE}.$

Proof. Counterexample.
Let $f = 1$ and $g = 10^{n-1}$. Thus,

$$2^{\frac{n}{1+\log f}} = 2^n,$$

and

$$\begin{aligned} 2^{\frac{n}{1+\log g}} &= 2^{\frac{n}{1+\log 10^{n-1}}} \\ &= 2^{\frac{n}{1+n-1}} \\ &= 2^{\frac{n}{n}} \\ &= 2 \\ &= O(1). \end{aligned}$$

We know,

$$2^n \neq O(1).$$ □

5. **Algorithm 1:** `localMinimum(`T`)`

   Input: A binary tree, T

   Output: A *local minimum* node in T

   $n \leftarrow$ root of T

   return `recursiveLocalMinimum(`T, n`)`

   Algorithm 2: `recursiveLocalMinimum(`T, n`)`

   Inputs: A binary tree, T; a node, n, in T

   Output: A *local minimum* node in a subtree of T with root node, n

   $r \leftarrow$ right child of n

   $\ell \leftarrow$ left child of n

   $x_n \leftarrow$ `probe(`n`)`

   $x_r \leftarrow$ `probe(`r`)`

   $x_\ell \leftarrow$ `probe(`ℓ`)`

   if $x_n = \min(x_n, x_r, x_\ell)$ **then**
      return n

   else
      if $x_\ell < x_r$ **then**
         return `recursiveLocalMinimum(`T, ℓ`)`
      else
         return `recursiveLocalMinimum(`T, r`)`

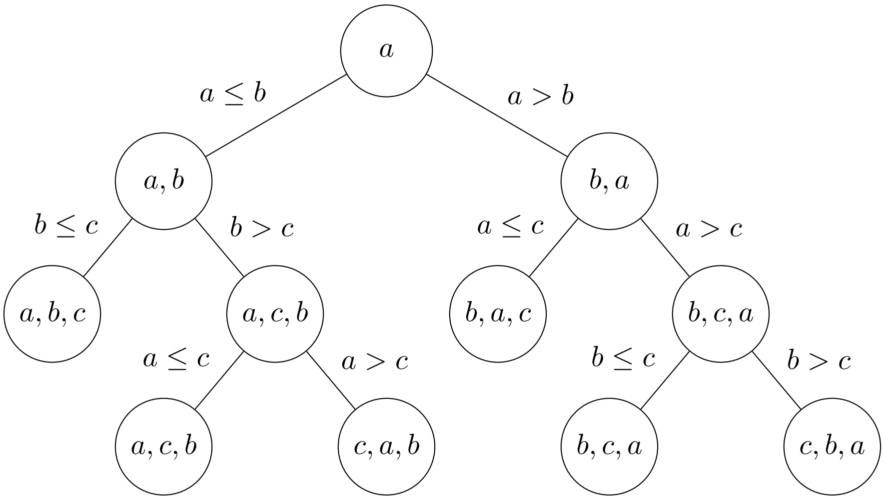
   We start at the root, r, of T and probe it, along with both its children. If x_r is smaller than both its children, then r is a local minimum. We return r and halt. Otherwise, we visit the child node with the smaller probed value and repeat.

   Either we will reach a leaf node, ℓ, with parent, p. We know it will be a local minimum because we would have only reached it if $x_\ell < x_p$.

   Since, at each step, we are making only one subproblem with size $\frac{n}{2}$ and doing constant work, we can use the Master Theorem to calculate the running time,

$$T(n) = 1 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n^0) \implies T(n) = O(\log n).$$

6. Let a, b, and c be the three elements.



7. Master Theorem. $\forall a > 0,\ b > 1,\ d \geq 0,$

$$T(n) = a\,T\left(\left\lceil \frac{n}{b} \right\rceil\right) + \Theta(n^d) \implies T(n) = \begin{cases} \Theta(n^d), & d > \log_b a, \\ \Theta(n^d \log n), & d = \log_b a, \\ \Theta(n^{\log_b a}), & d < \log_b a. \end{cases}$$

(a) We have the following recurrence relation:

$$\begin{aligned} T(n) &= 7\,T\left(\left\lceil \frac{n}{7} \right\rceil\right) + (3n + 20) \\ &= 7\,T\left(\left\lceil \frac{n}{7} \right\rceil\right) + \Theta(n^1). \end{aligned}$$

Let $a = 7$, $b = 7$ and $d = 1$. From the Master Theorem,

$$\log_b a = 1 = d \implies T(n) = \Theta(n \log n).$$

(b) We have the following recurrence relation:

$$\begin{aligned} T(n) &= 16\,T\left(\left\lceil \frac{n}{4} \right\rceil\right) + 100 \\ &= 16\,T\left(\left\lceil \frac{n}{4} \right\rceil\right) + \Theta(n^0). \end{aligned}$$

Let $a = 16$, $b = 4$ and $d = 0$. From the Master Theorem,

$$\log_b a = 2 > d \implies T(n) = \Theta\left(n^2\right).$$

(c) We have the following recurrence relation:

$$\begin{aligned} T(n) &= 2\,T\left(\left\lceil \frac{n}{2} \right\rceil\right) + (5n^2 + 2n + 3) \\ &= 2\,T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n^2). \end{aligned}$$

Let $a = 2$, $b = 2$ and $d = 2$. From the Master Theorem,

$$\log_b a = 1 < d \implies T(n) = \Theta(n^2).$$

8. Algorithm 1: `frequency`(A, b)

Input: An n-sized array of integers, A; an integer, b

Output: The *frequency* of b in A

return `recursiveFrequency`(A, b, 0, $n-1$)

Algorithm 2: `recursiveFrequency`(A, b, ℓ, h)

Input: An n-sized array of integers, A; an integer, b

Output: The *frequency* of b in A between the indices ℓ and h

```
if ℓ = h then
    if A[ℓ] = b then
        return 1;
    else
        return 0;
else
    m ← ⌊(ℓ+h)/2⌋
    return recursiveFrequency(A, b, ℓ, m) +
     recursiveFrequency(A, b, m + 1, h)
```

We are making two subproblems with size $\frac{n}{2}$ each, and doing constant work. We can use the Master Theorem to calculate the running time,

$$T(n) = 2 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n^0) \implies T(n) = O(n).$$