Asymptotics

1. **(Fun with asymptotics)** Using the definitions of Big-O, Big-Ω, and Big-Θ, formally prove the following statements.

   [We are expecting: For each part, a rigorous (but short) proof, using the definitions of Big-O, Big-Ω, and Big-Θ.]

   (a) $\log\log(n) = O(\log^2(n))$

   (b) $n^2 = \Omega(6n\sqrt{n} + 4n)$

   (c) $n^2 + 5n\sqrt{n}$ is **not** $\Theta(n^3)$

   (d) $\log(n!) = \Theta(n\log(n))$

2. **(A hairy proof)** What's wrong with this proof?

> Here, we prove that $n^3 = O(n^2)$. By definition of Big-O, $T(n) = O(g(n))$ iff there exists a c and n_0 such that $0 \leq T(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
>
> First, we apply log to both sides of $n^3 \leq O(n^2)$, which gives: $3\log(n) \leq O(2\log(n))$. Consider the values $c = 2$ and $n_0 = 1$. For the chosen value of c, clearly $3\log(n) \leq 2 \cdot 2\log(n) = 4\log(n)$ for all $n \geq 1$. Since there exists a c and n_0, we conclude that $n^3 = O(n^2)$.

[We are expecting: A brief English description.]

3. **(n-naught not needed?)** Suppose that $T(n) = O(n^d)$, and that $T(n)$ is never equal to ∞. Prove rigorously that there exists a c such that $0 \leq T(n) \leq c \cdot n^d$ for all $n \geq 1$. That is, the definition of Big-O holds with $n_0 = 1$.

   [We are expecting: A rigorous proof using the definition of Big-O.]

4. **(Worst-case and best-case vs. upper-bound and lower-bound)** Recall that while we often care most about the upper-bound of the worst-case runtime and the lower-bound of the best-case runtime, it's not required by definition that a worst-case runtime be expressed as Big-O or a best-case runtime be expressed as Big-Ω. Consider the following implementations and fill in the tables with *tight* asymptotic bounds.

   [We are expecting: Fill in the table with asymptotic bounds using Big-O, Big-Ω, and Big-Θ notation.]

   (a) Here's a standard implementation of linear search, which accepts a list `A` of distinct elements and a value `needle` to search for and returns the index of the value in the list if it exists; otherwise it raises an exception.

```
def linear_search(A, needle):
    for idx in range(len(A)):
        if A[idx] == needle:
            return idx
    raise Exception("needle does not exist!")
```

   (b) Here's a standard implementation of binary search, which accepts a sorted list `A` of distinct elements and a value `needle` to search for and returns the index of the value in the list if it exists; otherwise it raises an exception.

```
def binary_search(A, needle):
    if len(A) == 0:
        raise Exception("needle does not exist!")
    mid = len(A) / 2
    if A[mid] == needle:
        return mid
    elif A[mid] > needle:
        # needle must be in the left half, if it exists
        return binary_search(A[:mid], needle)
    else:
        # needle must be in the right half, if it exists
        return mid + 1 + binary_search(A[mid+1:], needle)
```

	`linear_search`	`binary_search`
Lower-bound of best-case		
Upper-bound of best-case		
Lower-bound of worst-case		
Upper-bound of worst-case		

Recurrences

---

1. **(Fun with recurrences)** Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$, and give a short justification. Be aware that some parts might be slightly more involved than others. Unless otherwise stated, assume $T(1) = 1$.

   [To see the level of detail expected, we have worked out the first one for you.]

   (z) $T(n) = 6T(n/6) + 1$. We apply the Master Theorem with $a = b = 6$ and with $d = 0$. We have $a > b^d$, so the runtime is $O(n^{\log_6(6)}) = O(n)$.

   (a) $T(n) = 3T(n/4) + \sqrt{n}$

   (b) $T(n) = 7T(n/2) + \Theta(n^3)$

   (c) $T(n) = 2T(\sqrt{n}) + 1$, where $T(2) = 1$

Problems

---

1. **(Selection sort)** Here is a Python implementation of selection sort, an iterative comparison-based sorting algorithm similar to insertion sort. Instead of inserting arbitrary elements into its growing sorted list, however, selection sort specifically "selects" the next largest unsorted element and appends it to the end of its growing sorted list.

```
1 def selection_sort(A):
2     for i in range(len(A)-1):
3         min_idx = i
4         for j in range(i+1, len(A)):
5             if A[j] < A[min_idx]:
6                 min_idx = j
7         # Swaps elements occupying min_idx and i in place
8         swap(min_idx, i)
```

   (a) The proof of correctness for selection sort, similar to the one for insertion sort, involves two loop invariants. What is the loop invariant associated with the outer `for` loop?
   [We are expecting: A brief English description.]

   (b) What is the loop invariant associated with the inner `for` loop?
   [We are expecting: A brief English description.]

   (c) Now let's put it all together. Fill in the following information for the outer `for` loop.
   [We are expecting: Brief English descriptions.]
      i. **Inductive Hypothesis**

ii. **Base case (initialization)**

iii. **Inductive step (maintenance)**

iv. **Conclusion (termination)**

(d) Fill in the following information for the inner `for` loop.
[We are expecting: Brief English descriptions.]

i. **Inductive Hypothesis**

ii. **Base case (initialization)**

iii. **Inductive step (maintenance)**

iv. **Conclusion (termination)**

2. **(Why not Select with groups of 3 or 7?)** In the `select` algorithm from class, in order to find a pivot, we divided our list of length n into $m = \lceil n/5 \rceil$ groups of at most length 5. Why 5? In this question, we explore this decision.

   Here is a Python implementation of `select`.

```python
def select(A, k, group_length=5, c=100):
    if len(A) <= c:
        return naive_select(A, k)
    pivot = median_of_medians(A, group_length)
    left, right = partition_about_pivot(A, pivot)
    if len(left) == k:
        # The pivot is the kth smallest element!
        return pivot
    elif len(left) > k:
        # The kth smallest element is left of the pivot
        return select(left, k, group_length, c)
    else:
        # The kth smallest element is right of the pivot
        return select(right, k-len(left)-1, group_length, c)

def select_3(A, k):
    select(A, k, group_length=3)

def select_7(A, k):
    select(A, k, group_length=7)
```

   (a) Prove that the recursive call inside of `select_3` gets passed a list of length at most $2n/3 + 2$. Include the values within the same group as the median of medians as elements that are guaranteed to be greater or less than it.

   [We are expecting: A convincing algebraic proof.]

(b) Write a recurrence relation for `select_3`.
[We are expecting: A recurrence relation.]

(c) Is `select_3` $O(n)$? Justify your answer.
[We are expecting: A convincing argument, such as analyzing a tree, unraveling the recurrence relation to get yield a summation, or attempting substitution method.]

(d) Prove that the recursive call inside of `select_7` gets passed a list of length at most $5n/7 + 4$. Make the same assumptions as part (a).

[We are expecting: A convincing algebraic proof.]

(e) Write a recurrence relation for `select_7`.

[We are expecting: A recurrence relation.]

(f) Is `select_7` $O(n)$? Justify your answer.

[We are expecting: A convincing argument, such as analyzing a tree, unraveling the recurrence relation to get yield a summation, or attempting substitution method.]