Full Name: ________________________ **Student ID:** ________________________

CMPS 2200: Introduction to Algorithms

Exam I

Oct 6-7, 2025

- There are 8 pages in this examination, comprising 3 questions worth a total of 120 points.
- You are allowed to use any relevant class materials (book, lectures, notes, etc.). You are not to use any other source of information, or to communicate with anyone about this exam (either portion) other than the instructors.
- You should turn in your solutions to Canvas on **Tuesday Oct 7th at 11:59 PM**.
- It will make grading much easier if you place your answers directly on this PDF, either by printing out and scanning, or using another PDF annotation method. If you must use separate paper, please clearly mark each question number. *If scanning, we recommend either Genius Scan or Scannable mobile apps to make formatting easier.*
- Please list all the detailed steps for full credit. Note that only showing the final solution will give you partial credits. If you are not confident with the solution, you can comment your thoughts for partial credits.

Question	Points	Score
Comparing Algorithms	40	
Longest Gap	40	
Efficient Exponentiation	40	
Total:	120	

Question 1: Comparing Algorithms (40 points)

Suppose you are given three algorithms, $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ that have the following recurrences for the amount of work they do:

$$\begin{aligned} W_{\mathcal{A}}(n) &= W_{\mathcal{A}}(n/2) + \log(n) \\ W_{\mathcal{B}}(n) &= W_{\mathcal{B}}(\sqrt{n}) + \log(n) \\ W_{\mathcal{C}}(n) &= 3W_{\mathcal{C}}(n/3) + 1 \end{aligned}$$

(a) (20 points) In the exercises we've seen so far, we haven't paid much attention to any necessary assumptions on n to make sure that a recurrence precisely "hits" the base case. For each recurrence above, state the assumptions needed for n as well a base case for each.

Solution 1:

For $\mathcal{A}$ and $\mathcal{C}$, we need to have $n = 2^k$ and can set $W_{\mathcal{A}}(1) = 1$. For $\mathcal{B}$, we can have $n = 2^{2^k}$ and set $W_{\mathcal{B}} = 2$.

(b) (20 points) Which algorithm performs the least work asymptotically? Show your work using the tree method or the brick method. Please put necessary details. **Don't use Master theorem to analyze.**

Solution 2:

$W_{\mathcal{A}}(n)$ is balanced, and its tree depth is $\log n$, $W_{\mathcal{A}}(n) = \log^2 n$,

$W_{\mathcal{B}}(n)$ is root dominated, $W_{\mathcal{B}}(n) = \log n$,

$W_{\mathcal{C}}(n)$ is leaf-dominated, and the commonality of node at level i is 2^i, its tree depth is $\log n$, $W_{\mathcal{C}}(n) = n$.

In this sense, Algorithm B is the most efficient.

Question 2: Longest Gap (40 points)

Consider the following problem: Given an input sequence a of n integers and a target integer key, determine the maximum distance between two occurrences of key in the sequence.

For example, if $key = 3$, the answer for the following sequence is 3:

$$[2, 3, \underline{5, 1, 6}, 3, 8, 4, 3, 9, 12, 3, 7]$$

For simplicity, we will assume that key appears at least twice in a.

We will solve this problem using `map`, `reduce`, and `scan`. The main observation is that we can first map the input to a sequence where position i is 1 if the value at i matches key, and is 0 otherwise. E.g., for the example above:

$[0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]$

We will next run `scan` on this sequence. With the resulting prefix sequence, we will run map-reduce to compute something that we can finally use to solve the original problem.

Below is a skeleton of the implementation. Your job is to fill in the missing blanks (marked with "...").

```
def longest_gap(a, key):
    a_prime = list(map(lambda v: my_map(key, v), a))
    prefixes, last = scan(...)
    mr_result = run_map_reduce(map_f, reduce_f, prefixes)
    mr_result_2 = [v[1] for v in mr_result]
    result = reduce(...)
    return ...
```

In the below, you are welcome to reference any functions we have defined in lecture, labs, or assignments. Assume that we are using our most efficient discussed versions of `map`, `reduce`, `scan`, and `run_map_reduce`, which can be found also in Lab 4.

```
def run_map_reduce(map_f, reduce_f, prefixes):
    # done. do not change me.
    """
    Params:
      map_f......the mapping function
      reduce_f...the reduce function
      prefixes.......list of input records
    """
    # 1. call map_f on each element of prefixes and flatten the results
    # e.g., [('i', 1), ('am', 1), ('sam', 1), ('i', 1), ('am', 1), ('sam', 1), ('is', 1), (
    pairs = flatten(list(map(map_f, prefixes)))
    # 2. group all pairs by their key
    # e.g., [('am', [1, 1]), ('ham', [1]), ('i', [1, 1]), ('is', [1]), ('sam', [1, 1])]
    groups = collect(pairs)
    # 3. reduce each group to the final answer
    # e.g., [('am', 2), ('ham', 1), ('i', 2), ('is', 1), ('sam', 2)]
    return [reduce_f(g) for g in groups]
```

(a) (8 points) Write the `my_map` implementation in Python

```
def my_map(key, v):

    if ________________________________

        return ______________________________
    else:
        return _______________________________
```

Solution 3:

```
def my_map(key, v):
    if key == v:
        return 1
    else:
        return 0
```

(b) (8 points) Complete the call to `scan` in line 3

```
prefixes, last = scan(______________________________________________________)
```

Solution 4:

```
    prefixes, last = scan(plus, 0, a_prime)
```

(c) (8 points) Specify what the `map_f` and `reduce_f` functions should be in the call to `run_map_reduce`

```
def map_f(value):

    return ________________

def reduce_f(group):

    return (group[0], reduce(__________________________________,

                             ___________________________________,

                             ____________________________________))
```

Solution 5:

```
def map_f(value):
    return [(value, 1)]

def reduce_f(group):
    return (group[0], reduce(plus, 0, group[1]))
```

(d) (8 points) Complete the call to reduce and return the final result in lines 6 and 7

```
    result = reduce(___________________________________,

                    ____________________________________,

                    _____________________________________)

    return ________________________________________________________________
```

Solution 6:

```
    return reduce(max, -math.inf, mr_result_2)-1
```

(e) (8 points) What is the work and span of the final algorithm? Write recurrences for each and provide closed-form solutions, showing all of your work.

Solution 7:

map: $W(n) = O(n), S(n) = O(1)$ scan: $W(n) = O(n), S(n) = O(\lg n)$ run map reduce: $W(n) = O(n), S(n) = O(\lg n)$ final reduce: $W(n) = O(n), S(n) = O(\lg n)$

Summing up, we have $W(n) = O(n)$ and $S(n) = O(\lg n)$

Question 3: Efficient Exponentiation (40 points)

Suppose $a > 0$ is a real number and x is a positive integer. We wish to compute a^x. Assume that $x = 2^n$ for some integer $n \geq 0$, and that each multiplication takes constant time.

- **Work** $W(n)$ = total number of operations performed
- **Span** $S(n)$ = length of the longest chain of dependencies.

(a) (5 points) Design an recursive algorithm that iteratively computes a^x. Give your algorithm specification by using the fact that $a^x = a \cdot a^{x-1}$.

Solution 8:

```
def Power(a, x):
    if x == 0:
        return 1
    else:
        return a * Power(a, x - 1)
```

(b) (5 points) What is the work and span of this algorithm as a function of x, and then as a function of n? Write recurrences for each and provide explicit solutions.

Solution 9:

$work/span : O(x), work : O(2^n), span : O(n),$

(c) (10 points) Give a **divide-and-conquer** approach that makes use of a recursive calls on a and $x/2$. Do you notice a glaring inefficiency that can be easily removed? Give your algorithm specification.

Solution 10:

```
def Power(a, x):
    if x == 0:
        return 1
    else:
        half = Power(a, x / 2)
        return half * half
```

(d) (10 points) What is the work and span of this algorithm as a function of x, then as a function of n? Write recurrences for each and provide explicit solutions, showing all of your work.

Solution 11:

$work/span : O(\log x), O(n)$

(e) (10 points) So far, we have assumed that $x = 2^n$. Now suppose that x is an *arbitrary positive integer*. For example, if $x = 13$, then

$$x = (1101)_2 = 8 + 4 + 1, \qquad a^{13} = a^8 \cdot a^4 \cdot a^1.$$

Consider a generalized exponentiation algorithm that:

- Computes the sequence of powers $a^{2^0}, a^{2^1}, \ldots, a^{2^{n-1}}$ sequentially or recursively.
- Selects only those powers where the corresponding bit of x is 1.
- Multiplies the selected terms together using a balanced binary tree, allowing parallel multiplications.

Question: What is the work $W(x)$ and $S(x)$? Assume that each multiplication takes constant time. Be explicit about what parts of the computation happen sequentially and which can be parallelized.

Solution 12:

- Let $\ell = \lfloor \log_2 x \rfloor + 1$ be the number of bits in x.
- Let $t = \text{popcount}(x)$ be the number of 1 bits in x (the number of terms to multiply at the end).

Algorithm Steps:

1. **Sequential phase:** Compute the powers
$$a^{2^0}, a^{2^1}, \ldots, a^{2^{\ell-1}}$$
sequentially. Each power is obtained by squaring the previous one. This requires $\ell - 1$ sequential multiplications.
2. **Selection phase:** Inspect the bits of x and select only the t powers whose corresponding bit is 1. (Work $O(\ell)$, negligible span.)
3. **Parallel phase:** Multiply the t selected powers together using a balanced binary tree. Total of $t - 1$ multiplications, arranged in $\lceil \log_2 t \rceil$ parallel rounds.

Work $W(x)$:

$$W(x) = (\ell - 1) + (t - 1) = \ell + t - 2 = \Theta(\ell) = \Theta(\log x)$$

Span $S(x)$:

$$S(x) = (\ell - 1) + \lceil \log_2 t \rceil = \Theta(\ell) = \Theta(\log x)$$

Sequential vs Parallel:

- **Sequential:** Building the sequence $a^{2^0} \to a^{2^1} \to \cdots \to a^{2^{\ell-1}}$ (chain of $\ell - 1$ squarings).
- **Parallel:** Multiplying the t selected terms in a balanced binary tree of height $\lceil \log_2 t \rceil$.