CMPS 6610 Problem Set 03

Answers

Name:___ Aaron Dumont_____

Place all written answers from `problemset-03.md` here for easier grading.

- **1a.** See main.py

Output: All isearch tests passed!

- **1b.** This function is sequential. It makes a single recursive call on a list that is one element smaller. This creates a dependency chain where each step must complete before the next one can begin.

Work: The function is called for each of the n elements in the list. Therefore, the total work is proportional to the length of the list.

$W(n) = W(n-1) + 1$; balanced. Continues shrinking list until $n - k = 0$. $k = n$ (n levels). Therefore, n x 1 so:

Work: $O(n)$

Span: The algorithm is sequential so work and span are the same. $S(n) = S(n-1) + 1$; balanced. Continues shrinking list until $n - k = 0$. $k = n$ (n levels). Therefore, n x 1 so:

Span: $O(n)$

- **1c.** See main.py

```
Output:[False, False, False, False, True, False, False]
[False, False, False]
[False]
[False, False]
[False]
[False]
[False, True, False, False]
[False, True]
[False]
[True]
[False, False]
[False]
[False]
[False, False, False, False, False, True]
[False, False, False]
[False]
[False, False]
[False]
[False]
[False, False, True]
[False]
[False, True]
[False]
[True]
[False, False, False, False, False, False]
[False, False, False]
[False]
[False, False]
[False]
[False]
```

```
[False, False, False]
[False]
[False, False]
[False]
[False]
[]
All rsearch tests passed!
```

- **1d. Work:** Rsearch has 2 stages: the initial mapping and the reduction. Let n be the length of the list L.

1. Mapping L to bool_list: This step involves one operation per element.

Work: $O(n)$

Span: This is a parallel operation so, assuming enough processors, it can be done in constant time. Span: $O(1)$.

2. Reduce call: The reduce function splits the list in half at each step.

Work: $W(n) = 2W(n/2) + O(1)$. Leaf-dominated. #leaves = $n^{log_2(2)} = n$. Therefore, **Work =** $O(n)$

Span: $S(n) = S(n/2) + O(1)$. Balanced. Work per level=1. #levels, $k = log_2(n)$. Therefore $S(n) = O(logn)$

Combining the above:

Work: $W(n) = O(n) + O(n) = O(n)$

Span: $S(n) = O(1) + O(logn) = O(logn)$

- **1e.** The ureduce function asymmetrically splits the list into 1/3 and 2/3. It then calls the original reduce function on these 2 smaller lists.

Work: $W(n) = W(n/3) + W(2n/3) + O(1)$ Leaf-dominated Total number of leaves is n. At each level the sum of the subproblems is n (eg. level 0 = n, level 2 $n/3 + 2n/3$. Level 3 $n/9 + 2n/9 + 2n/9 + 4n/9$). **Work:** $O(n)$

Span: $S(n) = max(S(n/3), S(2n/3)) + O(1) = S(2n/3) + O(1)$ Recurrnece stops after k steps when $(2/3)^k \cdot n = 1$. $n = (3/2)^k$ $k = log_{3/2}(n)$

So $S(n) = O(logn)$

- **2a.**

```
Python like:
def dedup(A):
    seen = set()
    result = []
    for item in A:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result

SPARC like:
dedup(A) =
  let
    // f is the function that processes one item.
    // It takes the current state (seen, result) and an item,
    // and returns the next state.
    let f((seen, result), item) =
      if Set.member(item, seen) then
        (seen, result) // Item already seen, state is unchanged
```

```
    else
      // Item is new, add it to both the set and the result list
      (Set.insert(item, seen), result ++ <item>)
  in
    // Run the iteration, starting with an empty set and empty list.
    let
      (_, final_result) = iterate(f, (Set.empty, <>), A)
    in
      final_result
    end
  end
```

- Work: $W(n) = W(n-1) + O(1)$; Balanced. Continues until n-k=0. n=k. n levels x work per level (1) $= n$.

  So $W(n) = O(n)$ — each element is checked once, with O(1) expected membership tests.

- Span: $S(n) = S(n-1) + O(1)$; Balanced. Continues until n-k=0. n=k. n levels x work per level (1) $= n$. so $S(n) = O(n)$ — recursion is sequential due to order preservation.

- Result: Returns the list of distinct elements of A, preserving their first occurrence order.

- **2b.**

Algorithm: - Flatten all lists into one sequence. - Group elements by value (using hashing). - Output one representative per group.

SPARC Specification:

```
fun multi_dedup(A : list of lists) : list =
    let
        val flat = flatten(A)
        val grouped = groupBy(flat, key = id)
    in
        keys(grouped)
    end
```

Work:

Flattening: $O(N)$.

Grouping: $O(N)$ expected (hash partitioning).

Extracting keys: $O(U)$, where U = number of unique elements.

Total Work = $O(N)$.

Span:

Flattening can be done in $O(logm)$ span (parallel concatenation).

Grouping can be done in $O(logN)$ span (parallel hashing).

Extracting keys is $O(1)$ span per group.

Total Span = $O(logN)$.

Comparison: - Part (a): Work $O(n)$,$ Span $O(n)$ (sequential, order-preserving) - Part (b): Work $O(N)$, Span $O(logN)$ (parallel, order not required)

I also looked at a multi_dedup using Sort instead of groupBy. multi_dedup_sort(A_list : list of lists) : list = let // 1. Combine all lists into one large list. val flat_A = flatten(A_list) val N = |flat_A|

```
// 2. Sort the list. This is the key step that groups
//    all identical elements next to each other.
val sorted_A = sort(flat_A)

// 3. Create a boolean flag for each element. The flag is true
//    if the element is the first in the list or different
//    from the element right before it.
val is_unique_flag = map(fn (i) =>
  if i == 0 then
    true
  else
    sorted_A[i] != sorted_A[i-1],
  tabulate(fn (i) => i, N)
)

// 4. Filter the sorted list, keeping only the elements
//    that were flagged as unique.
val result = filter(is_unique_flag, sorted_A)
```

in result end

Work and Span Analysis: For this analysis, let N be the total number of elements across all lists (i.e., |flatten(A_list)|). We assume a parallel sort with $O(NlogN)$ work and $O(log^2N)$ span.

Work: $W(N)$

The total work is the sum of the work for each sequential step. The sort operation is the most computationally expensive part.

flatten: $O(N)$

sort: $O(NlogN)$

map (to flag): $O(N)$

filter: $O(N)$

The recurrence relation is dominated by the sort:

$W(N) = W_{sort}(N) + W_{flatten}(N) + W_{map}(N) + W_{filter}(N)$

$W(N) = O(NlogN) + O(N) + O(N) + O(N) = O(NlogN)$

Span: $S(N)$

The total span is the sum of the spans of each step. Again, the parallel sort is the bottleneck. Let m be the number of lists in the original A_list.

flatten: $O(logm)$

sort: $O(log^2N)$

map (to flag): $O(1)$

filter: $O(logN)$

The recurrence relation is:

$S(N) = S_{flatten}(m) + S_{sort}(N) + S_{map}(N) + S_{filter}(N)$ $S(N) = O(logm) + O(log^2N) + O(1) + O(logN) = O(log^2N)$

This sort-based approach has more total work than the ideal hashing method but offers more predictable performance.

–

- **2c.**

Useful operations: - map: to transform elements into key–value pairs - flatten/concat: to merge multiple lists - groupBy/reduce: to collect duplicates and keep one representative - filter: to remove already-seen elements in sequential dedup - sort: allows us to organize and structure data which facilitates solving problems in parallel

These operations are especially powerful in the distributed setting, where they enable parallelism and reduce span from $O(n)$ to $O(logn)$ (although sort has $log^2 n$ span).

Not useful operations for parallelism: - iterate: this function is not helpful for designing an efficient parallel algorithm because it is inherently sequential. Using it, as shown in my dedup algorithm (in 2a), results in a linear span ($O(n)$), which offers no parallelism. It is useful, however, for the sequential implementation in my dedup algorithm.

–

- **3a.** See main.py

```
All parens_match_iterative tests passed!
```

–

- **3b.** Work: At each step, the algorithm performs a constant amount of work (calling the parens_update function) and then recurses on a subproblem of size n-1. The recurrence of the work is:

$W(n) = W(n-1) + O(1)$ This is balanced. Work per level =1. Base case is defined as $n - k$ so $k = n$ levels. Therefore solves to $O(n)$.

Span: The next step of the iteration (iterate on the rest of the list) depends directly on the result of the current step (f(x, a[0])), the operations must happen in sequence. There is no opportunity for parallelism.

Therefore, the span recurrence is identical to the work recurrence:

$S(n) = S(n-1) + O(1)$ which solves to $O(n)$ as above.

- **3c.**

See main.py

```
[1]
[1, -1]
[1]
[-1]
[1, -1]
[1]
[-1]
[1, 0]
[1]
[0]
[1]
[1]
[1]
[-1]
[-1]
[-1]
[1]
[1, 0]
[1]
[0]
```

```
[1, 0, -1]
[1]
[0, -1]
[0]
[-1]
[1, 0, -1, 1]
[1, 0]
[1]
[0]
[-1, 1]
[-1]
[1]
[1, 0, -1, 1, -1]
[1, 0]
[1]
[0]
[-1, 1, -1]
[-1]
[1, -1]
[1]
[-1]
[1, 0, -1, 1, -1]
[1, 0]
[1]
[0]
[-1, 1, -1]
[-1]
[1, -1]
[1]
[-1]
[1, 1, 0, 1, 0]
[1, 1]
[1]
[1]
[0, 1, 0]
[0]
[1, 0]
[1]
[0]
[1]
[1, 1]
[1]
[1]
[1, 1, 1]
[1]
[1, 1]
[1]
[1]
[1, 1, 1, -1]
[1, 1]
[1]
[1]
[1, -1]
[1]
```

```
[-1]
[1, 1, 1, -1, -1]
[1, 1]
[1]
[1]
[1, -1, -1]
[1]
[-1, -1]
[-1]
[-1]
[1, 1, 1, -1, -1, -1]
[1, 1, 1]
[1]
[1, 1]
[1]
[1]
[-1, -1, -1]
[-1]
[-1, -1]
[-1]
[-1]
[1, 1, 1, -1, -1, -1]
[1, 1, 1]
[1]
[1, 1]
[1]
[1]
[-1, -1, -1]
[-1]
[-1, -1]
[-1]
[-1]
[1, 2, 3, 2, 1, 0]
[1, 2, 3]
[1]
[2, 3]
[2]
[3]
[2, 1, 0]
[2]
[1, 0]
[1]
[0]
[1]
[1, 1]
[1]
[1]
[1, 1, -1]
[1]
[1, -1]
[1]
[-1]
[1, 1, -1]
[1]
```

```
[1, -1]
[1]
[-1]
[1, 2, 1]
[1]
[2, 1]
[2]
[1]
[1]
[1, 0]
[1]
[0]
[1, 0, -1]
[1]
[0, -1]
[0]
[-1]
[1, 0, -1, -1]
[1, 0]
[1]
[0]
[-1, -1]
[-1]
[-1]
[1, 0, -1, -1, 1]
[1, 0]
[1]
[0]
[-1, -1, 1]
[-1]
[-1, 1]
[-1]
[1]
[1, 0, -1, -1, 1]
[1, 0]
[1]
[0]
[-1, -1, 1]
[-1]
[-1, 1]
[-1]
[1]
[1, 1, 0, -1, 0]
[1, 1]
[1]
[1]
[0, -1, 0]
[0]
[-1, 0]
[-1]
[0]
All parens_match_scan tests passed!
```

- **3d.**

The parens_match_scan solution has three main, sequential steps:

A parallel map to convert parentheses to numbers.

An efficient parallel scan to get the running totals.

A parallel reduce to find the minimum of the running totals.

We examined fastscan in class which uses contraction. To create the subproblem of half the size the work and span recurrences are:

$W(n) = W(n/2) + cn$. This is root dominated so $W(n) = O(n)$

$S(n) = S(n/2) + 1$ (combination can be done in parallel in constant time with infinite processors). This recurrence is balanced. Work per level is 1. Number of levels: $n/2k = 1$,m $k = log_2 n$. So $S(n) = 1 * log_2 n$. So $S(n) = O(logn)$

The parallel map is done with $O(n)$ and span $O(1)$, and reduce is performed with $O(n) and $O(logn)$. Namely,:

$W(n) = O(n) + O(n) + O(n) = O(n)$

$S(n) = O(1) + O(logn) + O(logn) = O(logn)$

–

- **3e.**

See main.py

```
All parens_match_dc tests passed!
```

- **3f.**

The parens_match_dc_helper function follows the divide and conquer pattern: 1. Divide: The list of size n is split into two halves of size $n/2$.

2. Conquer: Two recursive calls are made to solve the subproblems for each half. The problem indicates these calls can be performed in parallel.
3. Combine: The results from the two halves (R1, L1) and (R2, L2) are combined using a constant number of operations (a min, a few additions and subtractions). The combine step is O(1).

Work Recurrence: We solve two subproblems of size $n/2$ and do a constant amount of work to combine them. So,

$W(n) = 2W(n/2) + O(1)$. Work of first level is 1. Work of second level is 2. This is leaf dominated. Number of leaves is $n^{log_2(2)}$ = n. Therefore $W(n) = O(n)$

$S(n) = S(n/2) + O(1)$ as the 2 recursive calls are done in parallel. This is balanced. Work per level is 1. Number of levels is $k = log_2 n$. So, $S(n) = O(logn)$.