

CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 17: Parallel Algorithms

Announcements

- Read parallel computing notes by Dan Grossman 3.5-5.4

Recap

- Last lectures
 - simple parallel programs
 - fork-join/thread programming
 - common patterns: reduce, map
- Now
 - analysis tools (work, span, parallelism)
 - Amdahl's Law
 - useful building blocks: prefix, pack
 - parallel quicksort, merge sort

Analyzing Parallel Programs

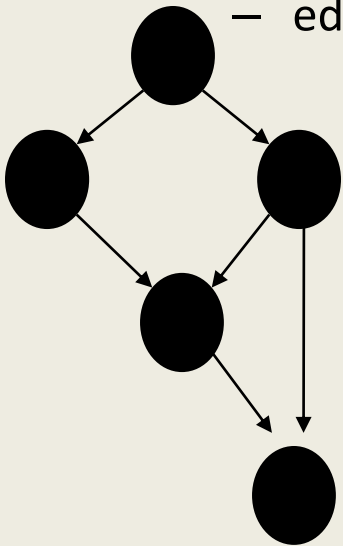
Let T_P be the running time on P processors

Two key measures of run-time:

- **Work**: How long it would take 1 processor = T_1
- **Span**: How long it would take infinity processors = T_∞
 - The hypothetical ideal for parallelization
 - This is the longest “dependence chain” in the computation
 - Example: $O(\log n)$ for summing an array
 - Also called “critical path length” or “computational depth”

The DAG

- Fork-join programs can be modeled with a DAG
 - nodes: pieces of work
 - edges: order dependencies



A **fork** creates two children

- new thread
- continuation of current thread

A **join** makes a node with two incoming edges

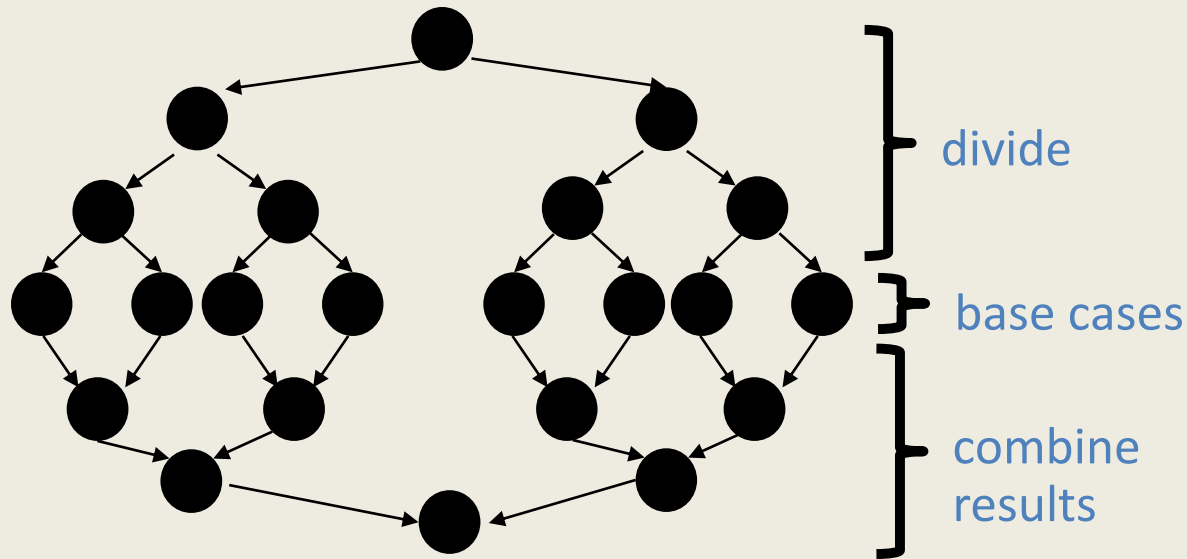
- terminated thread
- continuation of current thread

What's T_1 (work):

What's T_∞ (span):

Divide and Conquer Algorithms

Our **fork** and **join** frequently look like this:



In this context, the span (T_∞) is:

- The longest dependence-chain; longest ‘branch’ in parallel ‘tree’
- Example: $O(\log n)$ for summing an array; we halve the data down to our cut-off, then add back together; $O(\log n)$ steps, $O(1)$ time for each
- Also called “critical path length” or “computational depth”

Parallel Speed-up

- **Speed-up** on **P** processors: T_1 / T_P
- If speed-up is **P**, we call it **perfect linear speed-up**
 - e.g., doubling **P** halves running time
 - hard to achieve in practice
- **Parallelism** is the maximum possible speed-up: T_1 / T_∞
 - if you had infinite processors

Estimating T_p

- How to estimate T_p (e.g., $P = 4$)?
- Lower bounds on T_p (ignoring memory, caching...)
 1. T_∞
 2. T_1 / P
 - which one is the tighter (higher) lower bound?
- The ForkJoin Java Framework achieves the following expected time asymptotic bound:

$$T_p \text{ is } O(T_\infty + T_1 / P)$$

- this bound is optimal

Amdahl's Law

- Most programs have
 1. parts that parallelize well
 2. parts that don't parallelize at all
 - 3.
- The latter become bottlenecks

Amdahl's Law

- Let $T_1 = 1$ unit of time
- Let S = proportion that can't be parallelized

$$1 = T_1 = S + (1 - S)$$

- Suppose we get perfect linear speedup on the parallel portion:

$$T_p =$$

- So the overall speed-up on P processors is (Amdahl's Law):

$$T_1 / T_p =$$

$$T_1 / T_\infty =$$

- If 1/3 of your program is parallelizable, max speedup is:

Take Aways

- Parallel algorithms can be a big win
- Many fit standard patterns that are easy to implement
- Can't just rely on more processors to make things faster (Amdahl's Law)

Parallelizable?

- Prefix-sum:

input	6	3	11	10	8	2	7	8
output								

- $\text{output}[j] = \sum_{i=0}^j \text{input}[i]$

Parallel prefix-sum

- The parallel-prefix algorithm does two passes
 - Each pass has $O(n)$ work and $O(\log n)$ span
 - So in total there is $O(n)$ work and $O(\log n)$ span
 - So like with array summing, parallelism is $n/\log n$
- First pass builds a tree bottom-up: the “up” pass
- Second pass traverses the tree top-down: the “down” pass

Parallel Prefix: The Up Pass

We build want to build a binary tree where

- Root has sum of the range $[x,y)$
- If a node has sum of $[lo,hi)$ and $hi > lo$,
 - Left child has sum of $[lo,middle)$
 - Right child has sum of $[middle,hi)$
 - A leaf has sum of $[i,i+1)$, which is simply $input[i]$

It is critical that we actually create the tree as we will need it for the down pass

- We do not need an actual linked structure
- We could use an array as we did with heaps

Analysis of first step: Work = Span =

The algorithm, part 1

1. Propagate 'sum' up: Build a binary tree where
 - Root has sum of `input[0] .. input[n-1]`
 - Each node has sum of `input[lo] .. input[hi-1]`
 - Build up from leaves; `parent.sum=left.sum+right.sum`
 - A leaf's sum is just its value; `input[i]`

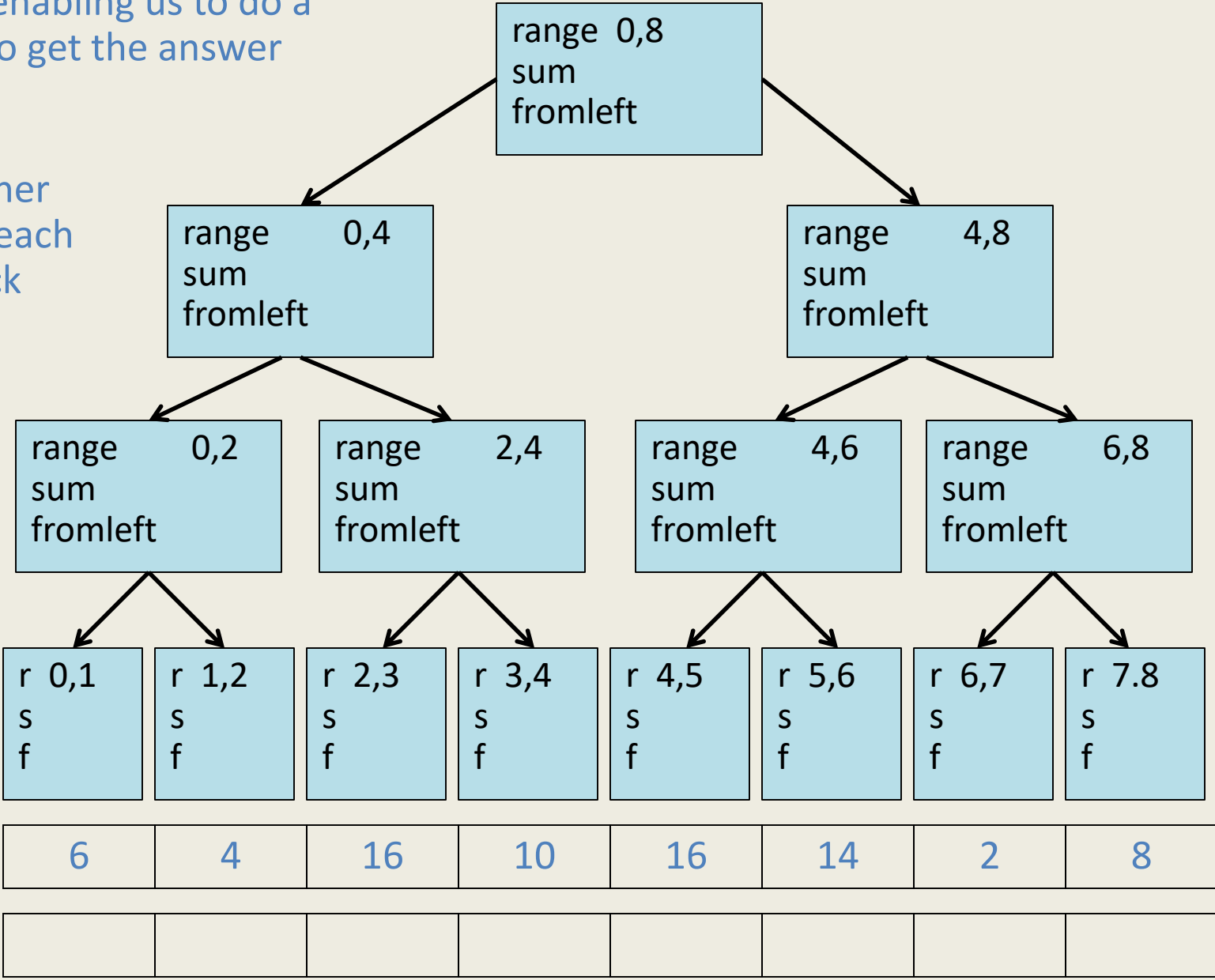
This is an easy fork-join computation: combine results by actually building a binary tree with all the sums of ranges

- Tree built bottom-up in parallel
- Could be more clever; ex. Use an array as tree representation like we did for heaps

Analysis of first step: $O(n)$ work, $O(\log n)$ span

Do an initial pass to gather information, enabling us to do a second pass to get the answer

First we'll gather the 'sum' for each recursive block



The algorithm, part 2

2. Propagate 'fromleft' down:

- Root given a **fromLeft** of 0
- Node takes its **fromLeft** value and
 - Passes its left child the same **fromLeft**
 - Passes its right child its **fromLeft** plus its left child's **sum** (as stored in part 1)
- At the leaf for array position **i**,
output[i] = fromLeft + input[i]

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result (the leaves assign to **output**)

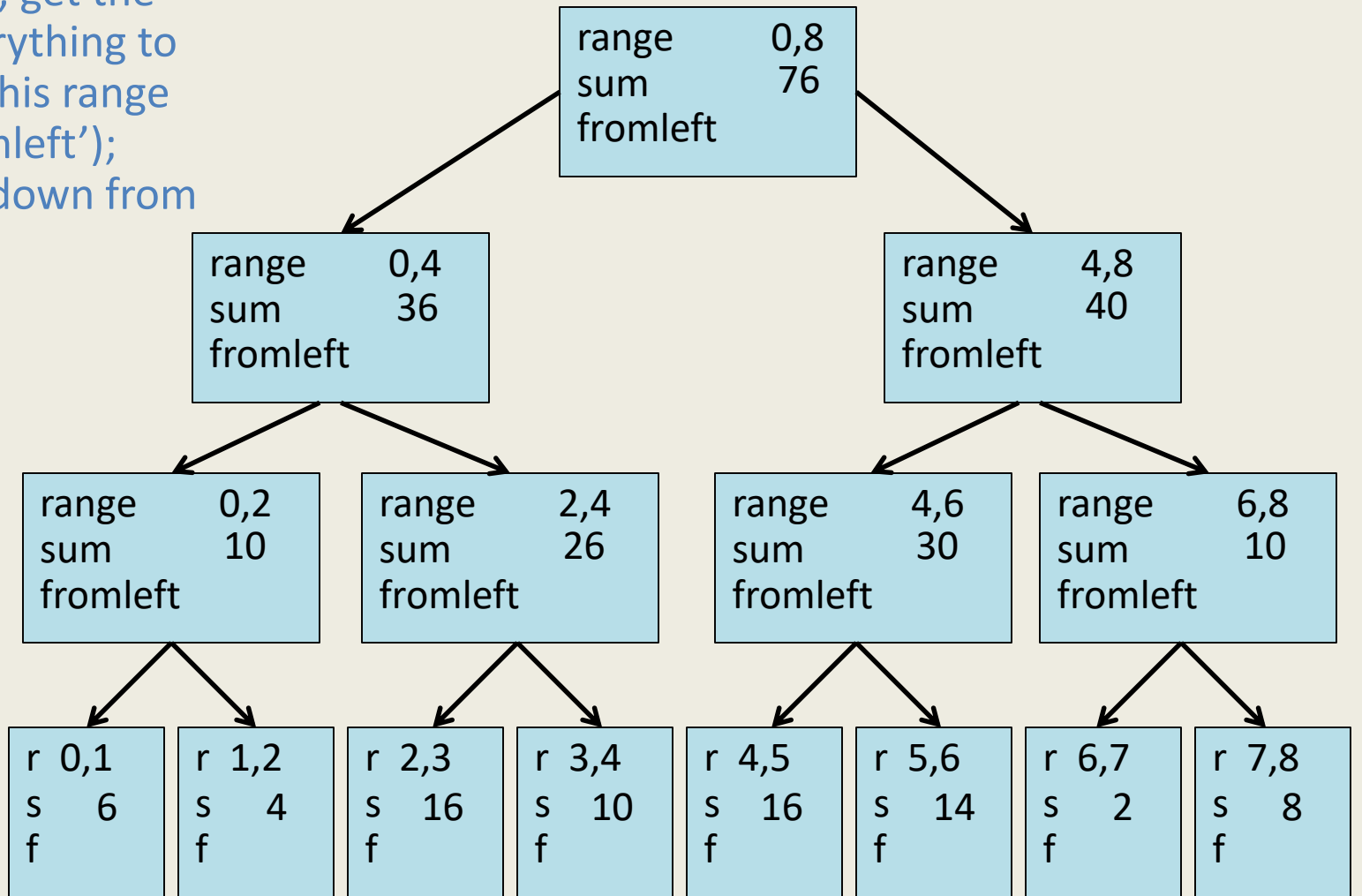
- Invariant: **fromLeft** is sum of elements left of the node's range

Analysis of first step: $O(n)$ work, $O(\log n)$ span

Analysis of second step: $O(n)$ work, $O(\log n)$ span

Total for algorithm: $O(n)$ work, $O(\log n)$ span

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root



input

6	4	16	10	16	14	2	8
---	---	----	----	----	----	---	---

output

--	--	--	--	--	--	--	--

Sequential cut-off

Adding a sequential cut-off isn't too bad:

- **Step One:** Propagating Up the **sums**:
 - Have a leaf node just hold the sum of a range of values instead of just one array value (Sequentially compute sum for that range)
 - The tree itself will be shallower
- **Step Two:** Propagating Down the **fromLefts**:
 - Have leaf compute prefix sum sequentially over its [lo,hi):
`output[lo] = fromLeft + input[lo];`
`for (i=lo+1; i < hi; i++)`
`output[i] = output[i-1] + input[i]`

Parallel Prefix, Generalized

- Prefix-sum is another common pattern (prefix problems)
 - maximum element **to the left of i**
 - is there an element **to the left of i** satisfying some property?
 - count of elements **to the left of i** satisfying some property
 - ...
- We can solve all of these problems in the same way

Pack

- Pack:
input

6	3	11	10	8	2	7	8
---	---	----	----	---	---	---	---

test: $x < 8?$

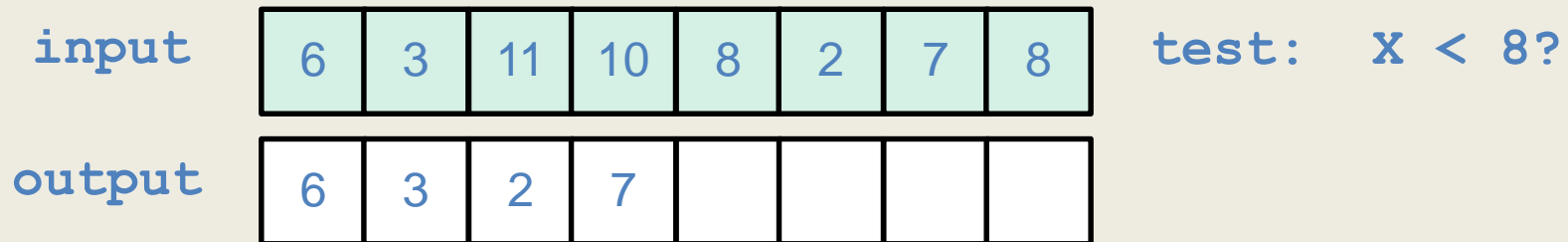
output

--	--	--	--	--	--	--	--

- Output array of elements satisfying **test**, in original order

Parallel Pack?

- Pack



- Determining **which** elements to include is **easy**
- Determining **where** each element goes in output is **hard**
 - seems to depend on previous results

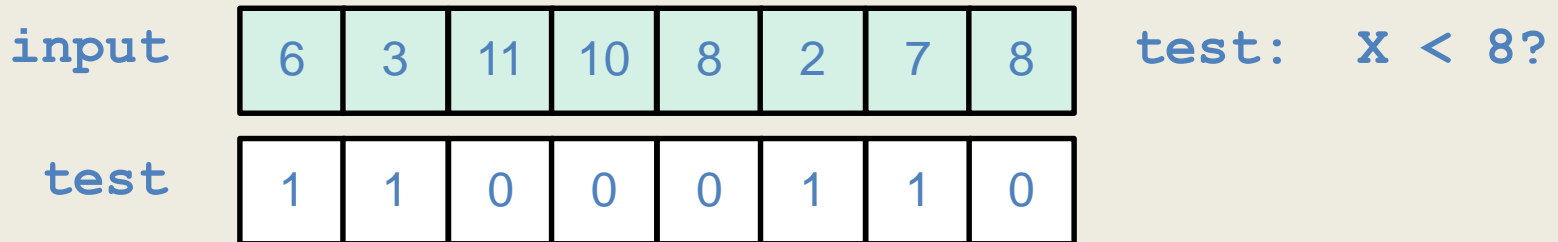
Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8	test:	$x < 8?$
test	1	1	0	0	0	1	1	0		

Parallel Pack

1. map test input, output [0,1] bit vector

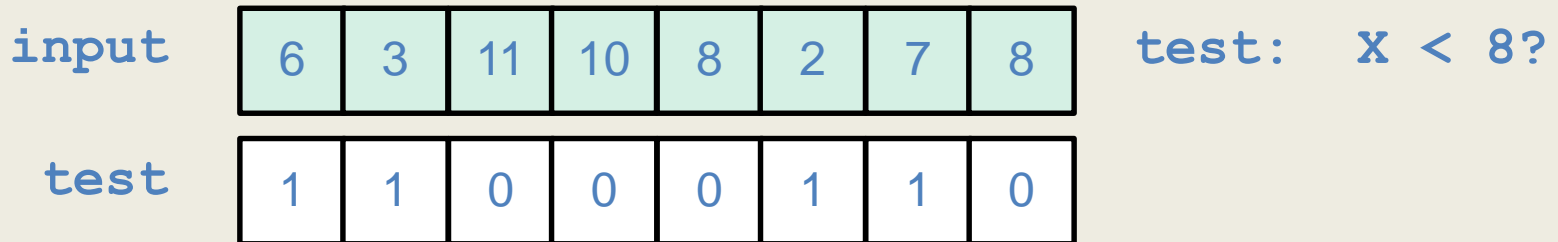


2. transform bit vector into array of indices into result array



Parallel Pack

1. map test input, output [0,1] bit vector



2. prefix-sum on bit vector



3. map input to corresponding positions in output



- `if (test[i] == 1) output[pos[i]] = input[i]`

Parallel Pack Analysis

- Parallel Pack
 1. map: $O(\quad)$ span
 2. sum-prefix: $O(\quad)$ span
 3. map: $O(\quad)$ span
- Total: $O(\quad)$ span