Multidimensional Included and Excluded Sums

Authors: Helen Xu, Sean Fraser, Charles E. Leiserson

Presenter: Derrick Liang

Weak Included-Sums Problem

Problem: Given a 1D array ***A*** of size N, compute range sum queries of size k

Problem: Given a 1D array ***A*** of size N, compute range sum queries of size k

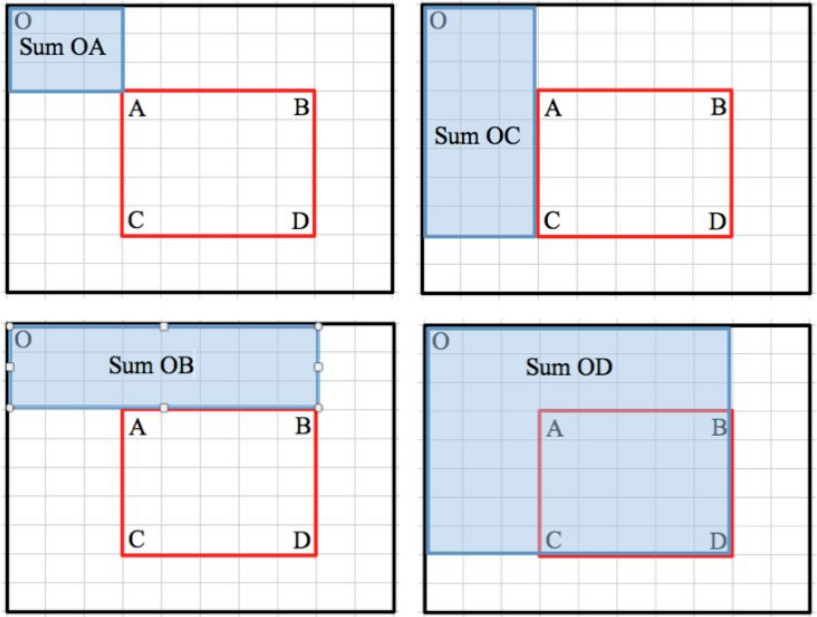
Solution: precompute prefix sums

Problem: Given a 2D array ***A*** of size N, compute box sum queries of size k x k

Problem: Given a 2D array ***A*** of size N, compute box sum queries of size k x k

Solution:

- Apply the inclusion-exclusion principle
- We can extend this to d-dimensions to answer queries in $O(2^d)$ time
- "Summed-area table" (SAT) algorithm



Strong Included-Sums Problem

Problem: Given a 1D array ***A*** of size N, compute range max queries of size k.

Why does the previous approach fail?

Problem: Given a 1D array ***A*** of size N, compute range max queries of size k

Why does the previous approach fail?

- Previously, we were solving the **weak** included-sums problem, meaning the operator has an inverse.
- A solution to the **strong** included-sums problem cannot rely on inverses

Motivation

Motivation for Problem: Strong Included-Sums

- Solving the strong problem allows us to avoid subtraction and catastrophic cancellation or round-off errors
- Real-time image processing/filtering can require rectangular sum queries
- Note: the included-sums problem specifically asks us to compute the range query of size k *over all* positions in the array

Problem: Given a 1D array ***A*** of size N, compute range max queries of size k

Solution:

- Let's try to stay close to the idea we had before.
- The specific issue is that we do not have an inverse (cannot subtract prefixes)
- **Insight 1: In order to use a prefix sums, the starting point of a query must be anchored at the starting point of the prefix computation**
  - (e can't be floating between two endpoints)

- **Insight 2: We haven't used the extra constraint of *fixed* query size.**

Problem: Given a 1D array ***A*** of size N, compute range max queries of size k

Solution:

- **"Bidirectional box-sum" (BDBS)** algorithm
  - Use "sum" as a stand-in for a general aggregating operation
- Create N/k chunks of size k subarrays, and precompute prefix and suffix maxima over each.
- **Any range of size k now can be decomposed into ranges which have an endpoint at the starting point of some subarray's prefix/suffix computation**
  - (We are guaranteed to be anchored now)

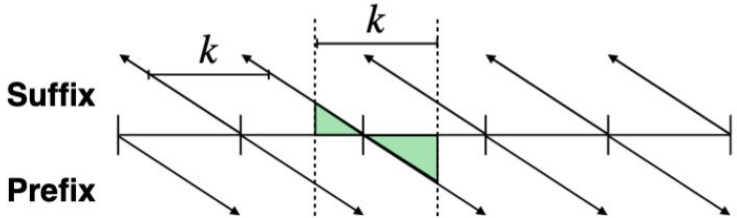
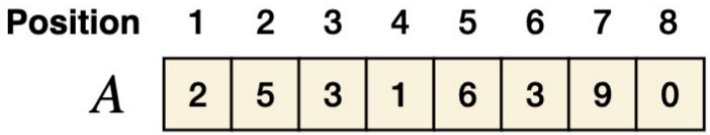
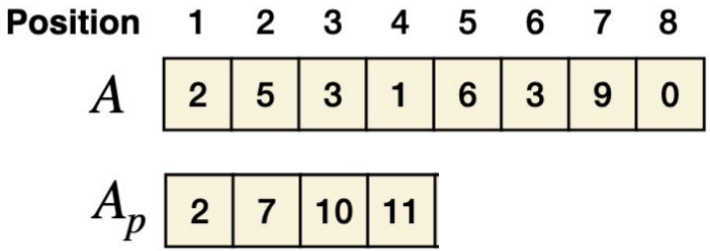


Figure 5: An illustration of the computation in the bidirectional box-sum algorithm. The arrows represent prefix and suffix sums in runs of size k, and the shaded region represents the prefix and suffix components of the region of size k outlined by the dotted lines.

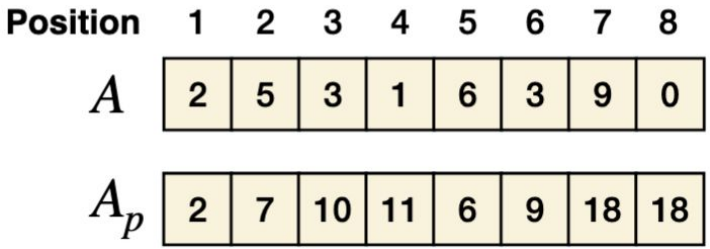
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



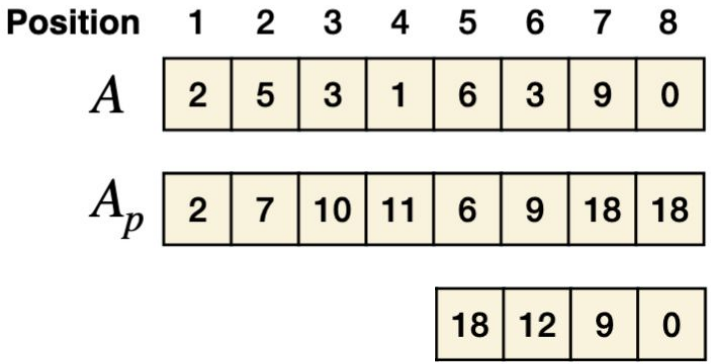
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



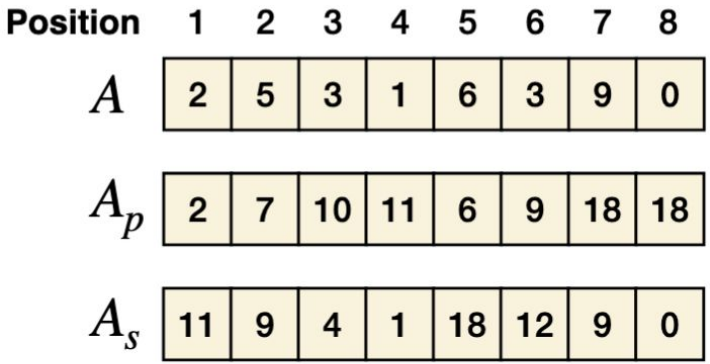
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



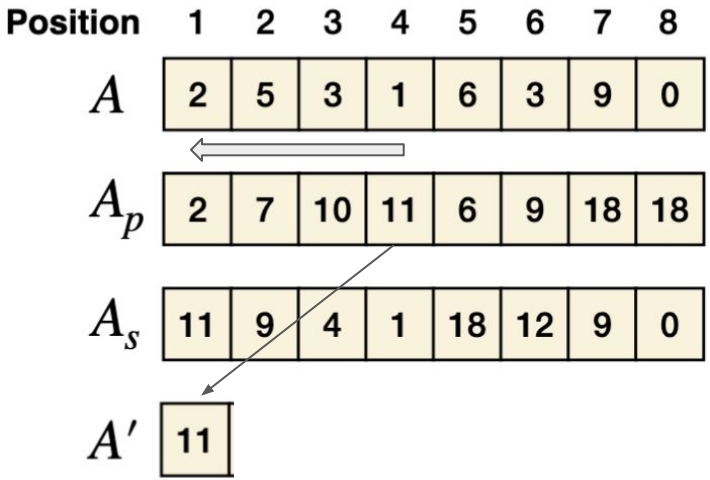
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



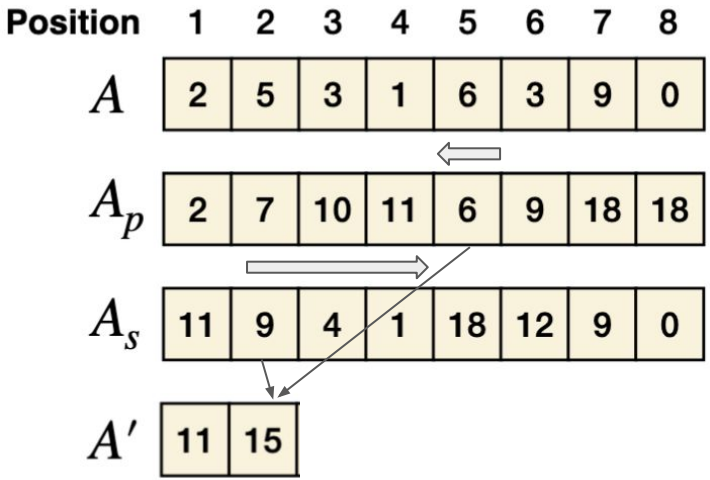
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



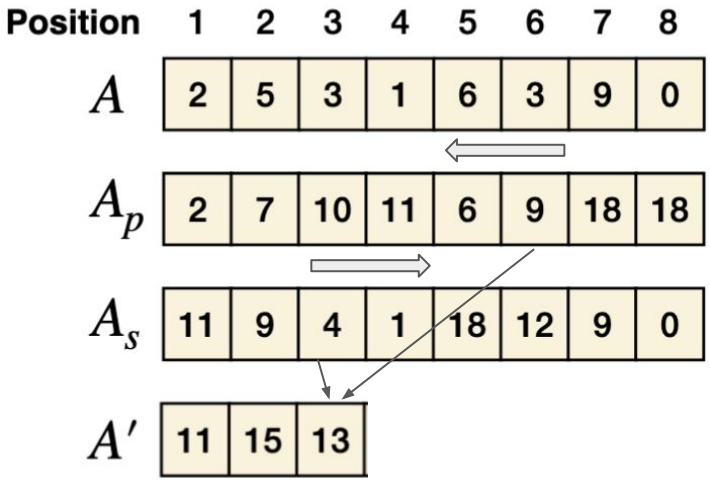
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



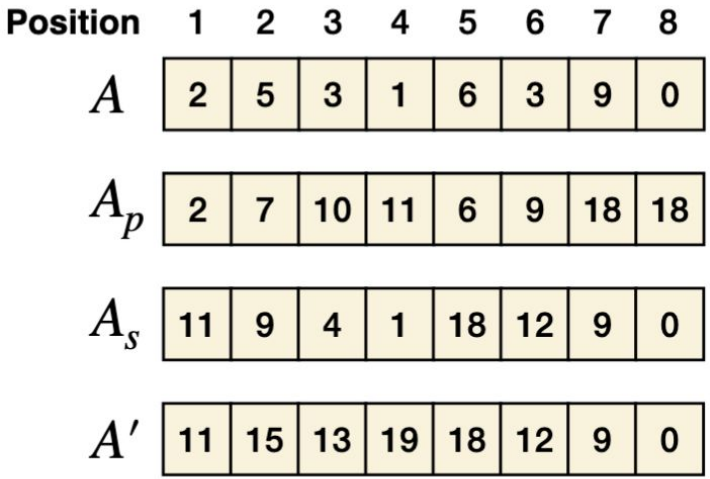
Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



Problem: Given a 1D array ***A*** of size N, compute range max queries of size k



Problem: Given a 2D array ***A*** of size N, compute box max queries of size k x k

Problem: Given a 2D array ***A*** of size N, compute box max queries of size k x k

Solution:

- Apply the BDBS algorithm along one dimension to create $\mathbf{A_1}$, where $\mathbf{A_1}[x_1,x_2] = \max(\mathbf{A}[x_1:x_1+k,\ x_2])$
- Then, apply the BDBS algorithm along the second dimension to yield $\mathbf{A_2}$, where $\mathbf{A_2}[x_1,x_2] = \max(\mathbf{A_1}[x_1,\ x_2:x_2+k]) = \max(\mathbf{A}[x_1:x_1+k,\ x_2:x_2+k])$, as desired.
- **This can be generalized to an d-dimensional array by induction.**

Excluded-Sums Problem

Problem: Given a 1D array ***A*** of size N, compute excluded range max queries of size k

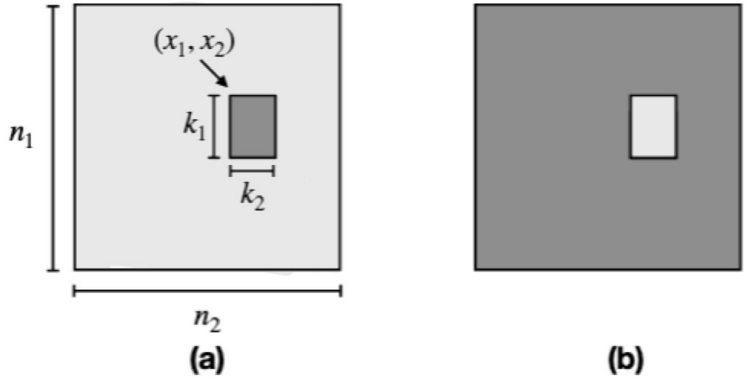


Figure 1: An illustration of included and excluded sums in 2 dimensions on an $n_1 \times n_2$ matrix using a (k_1, k_2)-box. **(a)** For a coordinate (x_1, x_2) of the matrix, the included-sums problem requires all the points in the $k_1 \times k_2$ box cornered at (x_1, x_2), shown as a grey rectangle, to be reduced using a binary associative operator $\oplus$. The included-sums problem requires that this reduction be performed at *every* coordinate of the matrix, not just at a single coordinate as is shown in the figure. **(b)** A similar illustration for excluded sums, which reduces the points outside the box.

Problem: Given a 1D array ***A*** of size N, compute excluded range max queries of size k

Solution: Prefix and suffix sums

Problem: Given a 2D array ***A*** of size N, compute excluded box max queries of size k x k

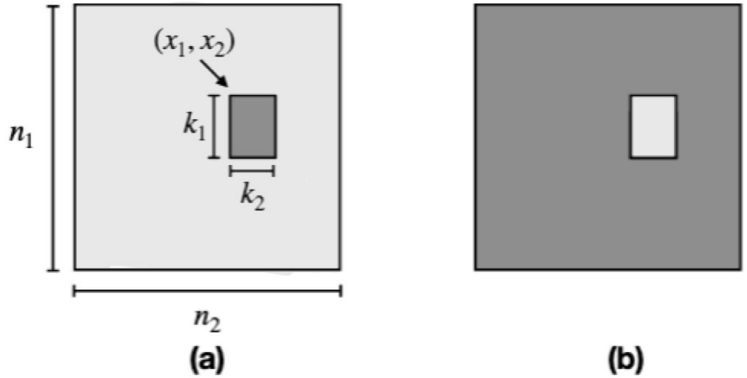


Figure 1: An illustration of included and excluded sums in 2 dimensions on an $n_1 \times n_2$ matrix using a (k_1, k_2)-box. **(a)** For a coordinate (x_1, x_2) of the matrix, the included-sums problem requires all the points in the $k_1 \times k_2$ box cornered at (x_1, x_2), shown as a grey rectangle, to be reduced using a binary associative operator $\oplus$. The included-sums problem requires that this reduction be performed at *every* coordinate of the matrix, not just at a single coordinate as is shown in the figure. **(b)** A similar illustration for excluded sums, which reduces the points outside the box.

Problem: Given a 2D array ***A*** of size N, compute excluded box max queries of size k x k

Existing Solution:

- In general, use all 2^d combinations of prefix/suffix sums with respect to each dimension, yielding $O(2^d)$ time per query
- "Corners" algorithm (Demaine *et al.*)

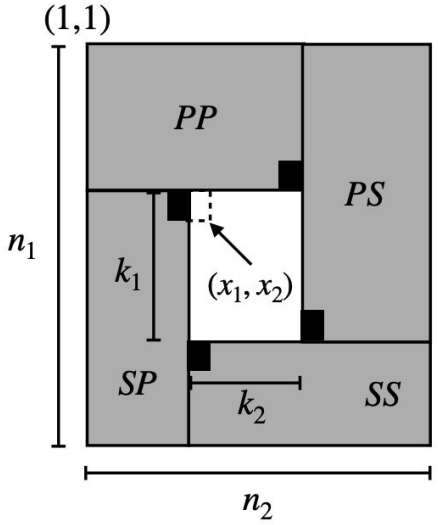
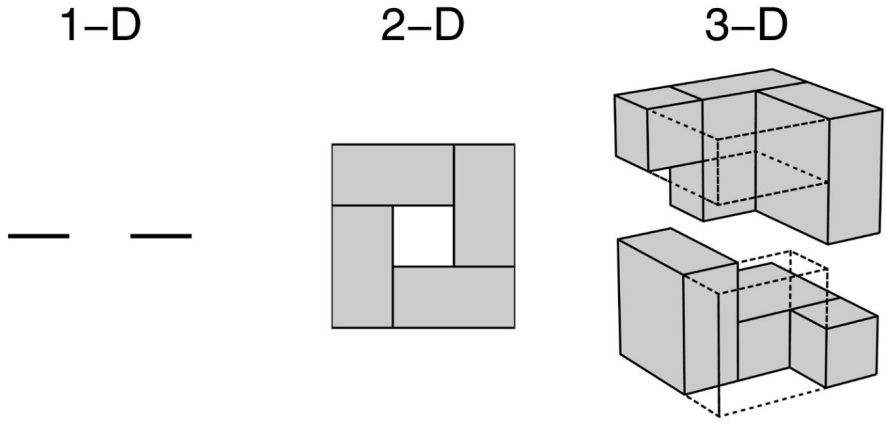


Figure 11: An example of the corners algorithm in 2 dimensions on an $n_1 \times n_2$ matrix using a (k_1, k_2)-box cornered at (x_1, x_2). The grey regions represent excluded regions computed via prefix and suffix sums, and the black boxes correspond to the corner of each region with the relevant contribution. The labels PP, PS, SP, SS represent the combination of prefixes and suffixes corresponding to each vertex.

Problem: Given a 2D array ***A*** of size N, compute excluded box max queries of size k x k

Existing Solution:

- General combinatorial construction exists for higher dimensions



Improved Excluded Sums

Can we do better than exponential time in number of dimensions?

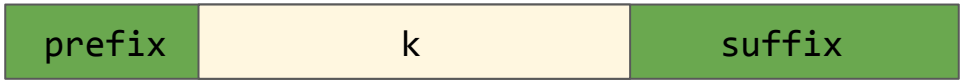
Motivation for Strong Excluded-Sums

- Solving the strong problem again helps avoid round-off errors
- N-particle simulations
  - The fast multipole method (FMM) is often used in this context for approximating long-ranged forces (excluding interactions with neighbors that are too close) and demands a similar computation

- Note: solving the strong included-sums problem (efficiently) means we also solved the weak included-sums problem.
- By taking the complement, we also solve the weak excluded-sums problems (this is the **BDBS-complement** algorithm)

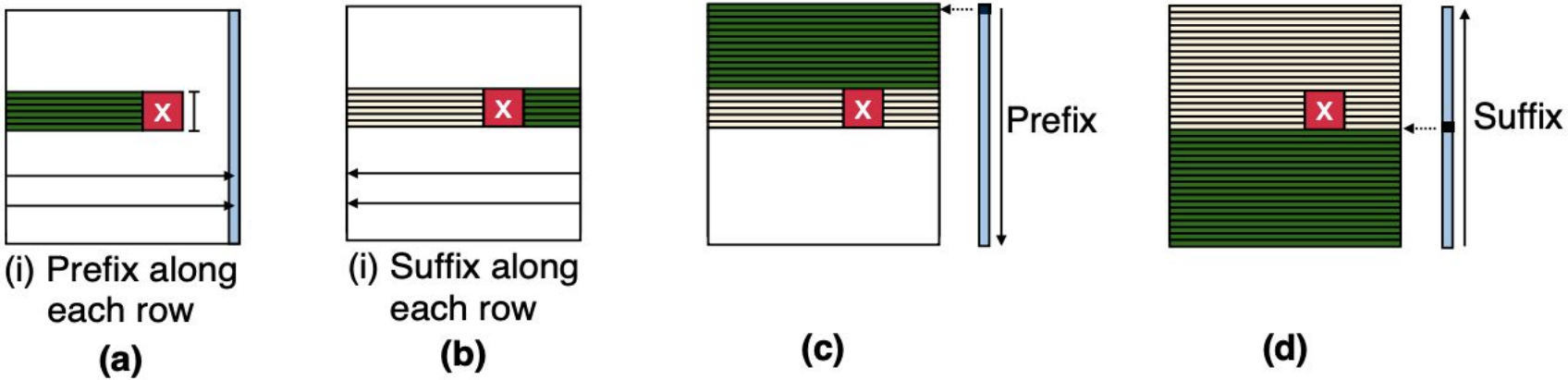
Key Insight - Partitioning the Space

- In 1D, there are two disjoint spaces to combine
- $A'[x] = A_p[x] + A_s[x+k]$



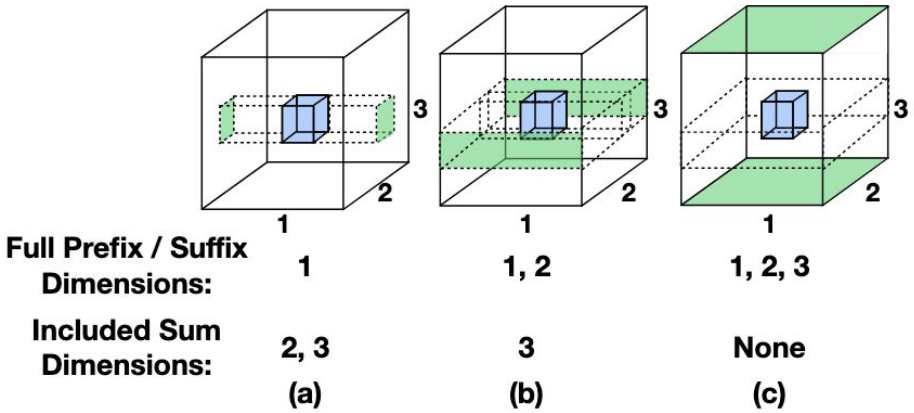
Key Insight - Partitioning the Space

- In 2D, there are four disjoint spaces



Key Insight - Partitioning the Space

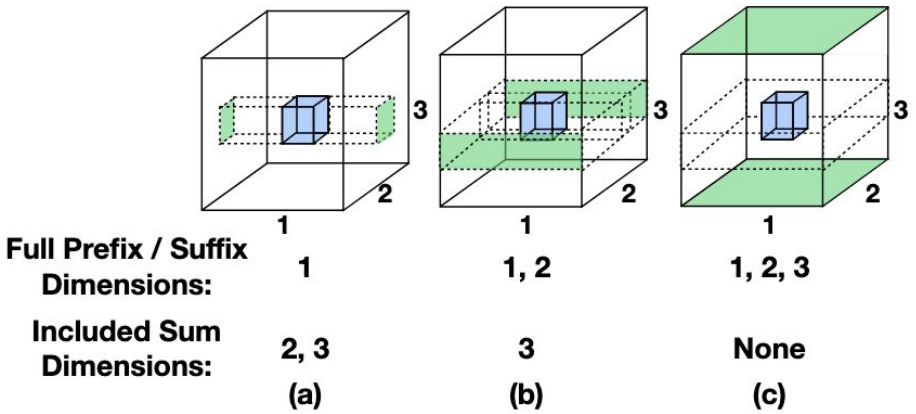
- In 3D, there are eight disjoint spaces. In general there are 2d for a d-dimensional space.
- We can see successively less tightly bound regions.
- **Observation: we can either have d-1, d-2, …, 0 of the coordinates lie within the bounds of the box's coordinates in that dimension**



Consider the spaces bounded by the green planes

Key Insight - Partitioning the Space

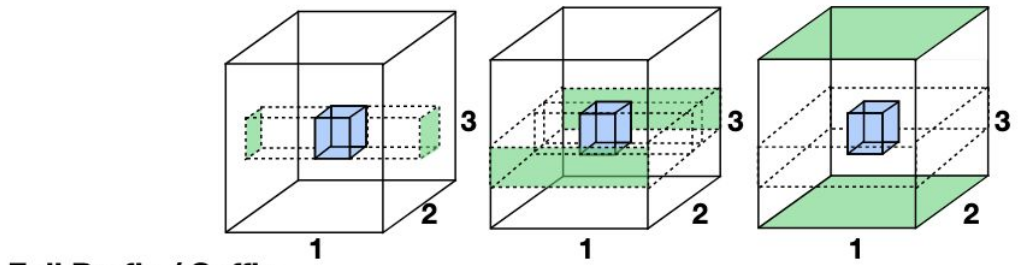
- Can we somehow compute the contribution of the two spaces, then recurse onto a subproblem with d-1 dimensions?
  - Loosely, we're taking out the top and bottom "buns" and then recursing on the "patty"
  - **Box-Complement** algorithm



Consider the spaces bounded by the green planes

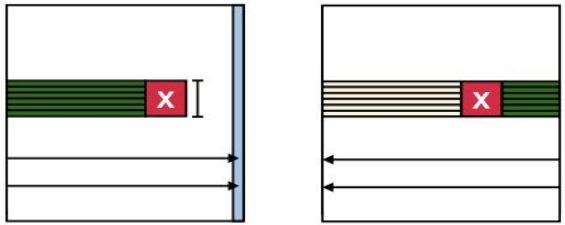
Box-Complement

- Definition: the **i-complement** of a box is all elements that is "out of range" in *some* dimension j ≤ i and "in range" for *all* dimensions > i.
  - The first condition ensures that it doesn't intersect with the excluded box
  - Notice: the (i+1)-complement contains the i-complement



Understanding The Recursive Step

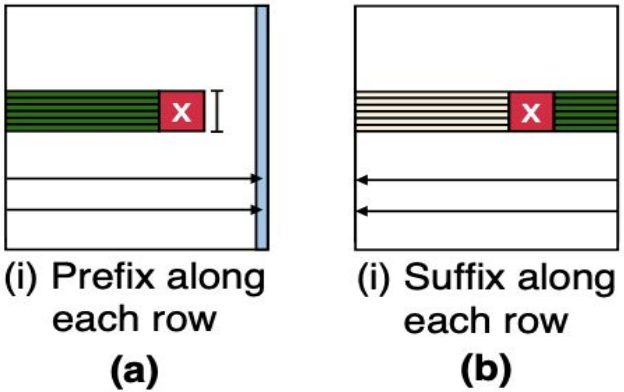
- Recall the 1D solution the problem.
  - $A'[x] = A_p[x] + A_s[x+k]$
- For the 2D problem, let's first try to compute the 1-complement
  - These are all elements within the vertical bound, and outside the horizontal bound of a given excluded box
- First, compute the prefixes and suffixes along one dimension.
- $A_p[x_1, x_2] = A[:x_1, x_2]$
- $A_s[x_1, x_2] = A[x_1:, x_2]$



Here, we consider an arbitrary point p at the top left corner of the red box

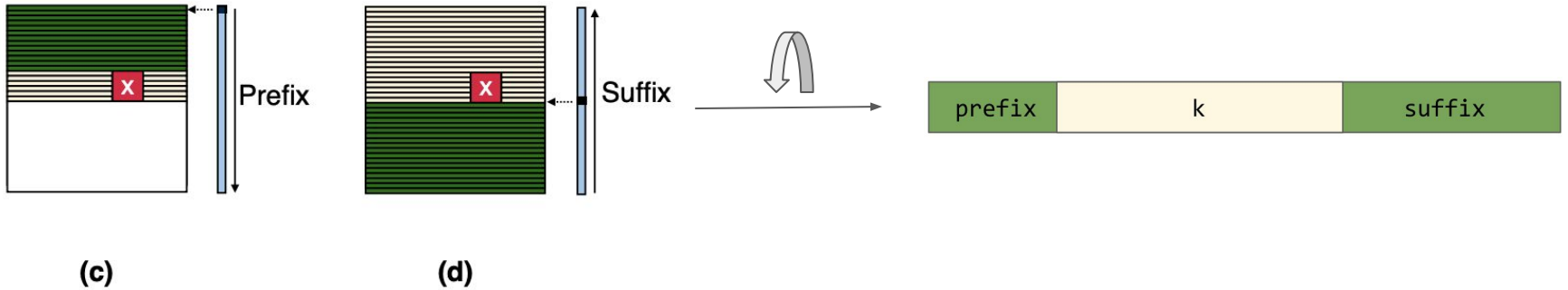
Understanding The Recursive Step

- Next, we use the included sum algorithm (BDBS) to aggregate the horizontal strips of prefixes and suffixes
- $A[x_1, x_2] = A_p[x_1, x_2 : x_2+k] + A_s[x_1+k, x_2 : x_2+k]$
  - At this point, all of the green area in the figure below is stored into the top left corner of the red box



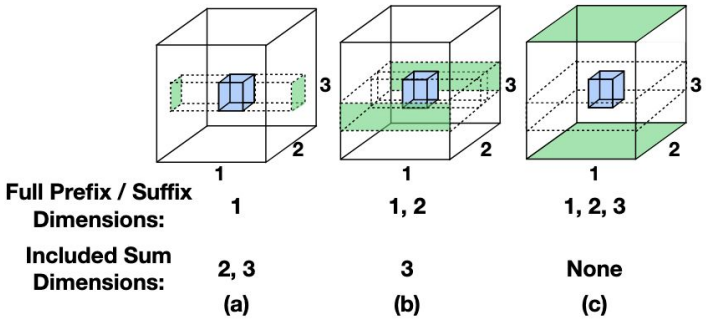
Understanding The Recursive Step

- Now, we need to add the contribution of the 2-complement that wasn't added by the 1-complement.
  - This is the remaining green area in the diagram
- We can construct A', where $A'[x_2] = A_p[n_1, x_2]$
  - This reduces one dimension down (projecting it all onto the last coordinate of dimension 1 which is n_1)
- We can now pass this onto the 1-dimensional solver!



General Algorithm

- For d steps, compute the contribution of the new elements of the current i-complement
  - Apply prefix and suffix sums on the ith dimension
  - Apply the BDBS algorithm **d-i** times and collect the contributions of the two disjoint spaces
  - Use the nth tensor on the ith dimension as the new tensor
- Time: O(dN), Space: O(N)
  - Intuitively, the space and work exponentially shrinks on each level, of the recursion, so the runtime is dominated by the root
  - The implementation reuses space to achieve linear complexity



Pseudocode

```
BOX-COMPLEMENT(A, k)
1  // Input: Tensor A with d-dimensions, box size k
   // Output: Tensor A' with size and dimensions
   // matching A containing the excluded sum.
2  init A' with the same size as A
3  A_p ← A; A_s ← A
4  // Current dimension-reduction step
5  for i ← 1 to d
6  // Saved from previous dimension-reduction step.
7        A_p ← A reduced up to dimension i − 1
8        A_s ← A_p    // Save input to suffix step
9        // PREFIX STEP
         // Reduced up to i dimensions.
10       PREFIX-ALONG-DIM along
11         dimension i on A_p.
12       A ← A_p    // Save for next round
13       // Do included sum on dimensions [i + 1, d].
14       for j ← i + 1 to d
15             // A_p reduced up to i dimensions
16             BDBS-ALONG-DIM on
17               dimension j in A_p
18       // Add into result
19       ADD-CONTRIBUTION from A_p into A'
20
21       // SUFFIX STEP
         // Do suffix sum along dimension i
22       SUFFIX-ALONG-DIM along
23         dimension i in A_s
24       // Do included sum on dimensions [i + 1, (
25       for j ← i + 1 to d
26             // A_s reduced up to i dimensions
27             BDBS-ALONG-DIM on
28               dimension j in A_s
29       // Add into result
30       ADD-CONTRIBUTION from A_s into A'
31 return A'
```

Comparison of Current Algorithms

Algorithm	*Source*	*Time*	*Space*	*Included or Excluded?*	*Strong or Weak?*
Naive included sum	[This work]	$\Theta(KN)$	$\Theta(N)$	Included	Strong
Naive included sum complement	[This work]	$\Theta(KN)$	$\Theta(N)$	Excluded	Weak
Naive excluded sums	[This work]	$\Theta(N^2)$	$\Theta(N)$	Excluded	Strong
Summed-area table (SAT)	[6, 15]	$\Theta(2^d N)$	$\Theta(N)$	Included	Weak
Summed-area table complement (SATC)	[6, 15]	$\Theta(2^d N)$	$\Theta(N)$	Excluded	Weak
Corners(c)	[8]	$\Theta((d+1/c)2^d N)$	$\Theta(cN)$	Excluded	Strong
Corners spine	[8]	$\Theta(2^d N)$	$\Theta(dN)$	Excluded	Strong
Bidirectional box sum (BDBS)	[This work]	$\Theta(dN)$	$\Theta(N)$	Included	Strong
Bidirectional box sum complement (BDBSC)	[This work]	$\Theta(dN)$	$\Theta(N)$	Excluded	Weak
Box-complement	[This work]	$\Theta(dN)$	$\Theta(N)$	Excluded	Strong

Table 1: A summary of all algorithms for excluded sums in this paper. All algorithms take as input a d-dimensional tensor of N elements. We include the runtime, space usage, whether an algorithm solves the included- or excluded-sums problem, and whether it solves the strong or weak version of the problem. We use K to denote the volume of the box (in the runtime of the naive algorithm).

Note: these are the complexities to compute the query answers at all possible starting positions in the input array.

Experimental Results

- We can see the approaches in this paper show linearity in time with dimension number, while previous approaches are exponential.

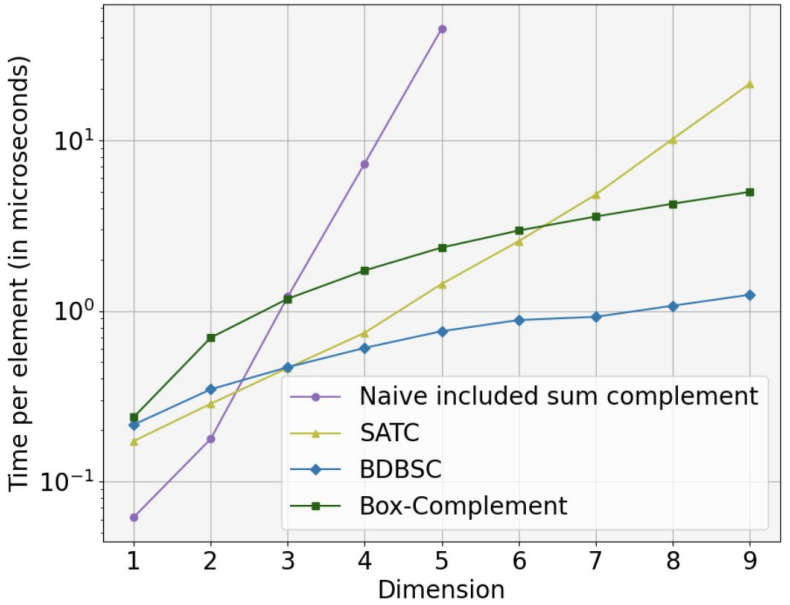


Figure 4: Time per element of algorithms for excluded sums in arbitrary dimensions. The number of elements N of the tensor in each dimension was in the range $[2097152, 134217728]$ (selected to be a exact power of the number of dimensions). For each number of dimensions d, we set the box volume $K = 8^d$.

Experimental Results

- The box-complement beats other existing algorithms as the computation of the operator increases.
  - Box-complement performs ~12 operations per element while the best corners algorithm performs ~22.

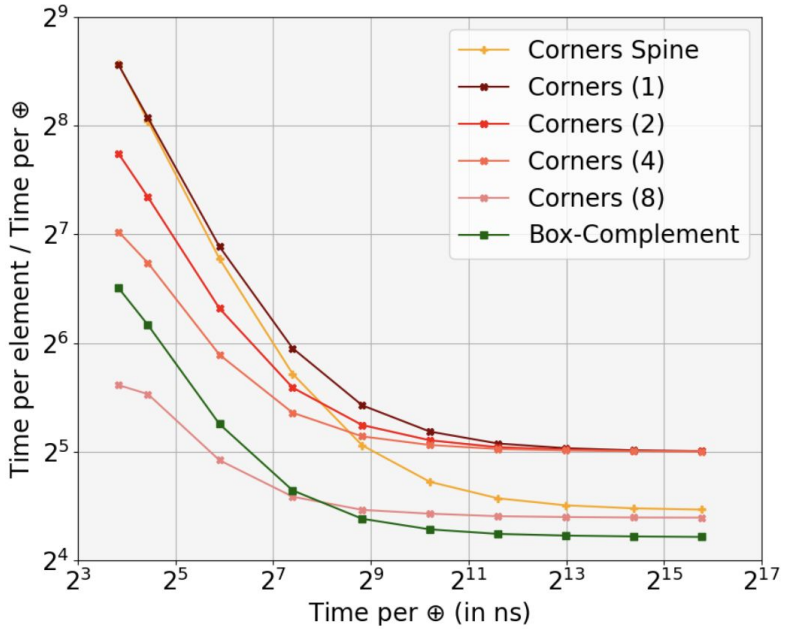


Figure 10: The scalability of excluded-sum algorithms as a function of the cost of operator $\oplus$ on a 3D domain of $N = 4096$ elements. The horizontal axis is the time in nanoseconds to execute $\oplus$. The vertical axis represents the time per element of the given algorithm divided by the time for $\oplus$. We inflated the time of $\oplus$ using increasingly large arguments to the standard recursive implementation of a Fibonacci computation.

Paper Review

- The problem was really interesting and had lots of layers.
- Diagrams in the paper were helpful, but if it showed more examples to work up to the full generalized algorithm it would've helped make it easier to understand.
- Ideas for future directions:
  - Proving these are optimal
  - Loosening the restrictions on box size to a range of desired sizes
  - Creating parallel implementations

References

https://epubs.siam.org/doi/epdf/10.1137/1.9781611976830.17

http://persson.berkeley.edu/pub/demaine05blocks.pdf