

Lecture 5 Notes (Oct 14, 2026)

Sam Westrick

s.westrick@nyu.edu

Partitioning and Counting Sort

Suppose every element x of a sequence has an integer key in the range $[0, K)$, and we want to **partition** the sequence into K buckets: bucket k should contain all elements with key k , in their original order.

```
partition: int -> ('a -> int) -> 'a seq -> 'a seq seq
```

(For example, with $K = 3$ and each key being the element itself:) `partition 3 (fn x => x) <2, 0, 2, 1, 0, 2> = <<0, 0>, <1>, <2, 2, 2>>`

Flattening the buckets sorts the input by key. Because each bucket preserves the original order, this sort is **stable**: elements with equal keys keep their relative order.

Attempt #1: many calls to filter

Using `filter` from last lecture, the solution is one line:

```
fun partition K key s =  
  Seq.tabulate (fn k => Seq.filter (fn x => key x = k) s) K
```

This has $O(\log n + \log K)$ span, but $O(nK)$ work, because every one of the K filters looks at the entire input. We'd like to look at each element only $O(1)$ times.

Attempt #2: a block-based filter, specialized to K keys

Recall that `filter` is a scan over 0/1 flags, followed by a pass that writes out the survivors. In practice, we would implement this with blocks (like the three-phase blocked scan from Lecture 3): break the input into n/B blocks of size B , sequentially count the survivors in each block, scan over the n/B block counts, and then have each block sequentially write its survivors to the output starting at its offset.

Now, rather than running K separate filters, we can do all of them at once. Each block keeps K **counters**, one for each key, for a total of $K \cdot \frac{n}{B}$ counters. Then we do K scans (one per key, each across the blocks), and finally make one more pass where each block writes each of its elements into the output for that element's key.

```
fun partition B K key s =  
  let  
    val n = Seq.length s  
    val numBlocks = (n + B - 1) div B  
    fun idx (b, k) = b * K + k    (* block-major *)  
    val counts = ForkJoin.alloc (numBlocks * K)  
  
    (* Phase 1: each block counts its keys *)  
    val _ = Parallel.parfor (0, numBlocks) (fn b =>  
      ( Util.for (0, K) (fn k => Array.update (counts, idx (b, k), 0))  
      ; Util.for (b*B, Int.min (n, (b+1)*B)) (fn i =>  
        let val j = idx (b, key (Seq.nth s i))  
        in Array.update (counts, j, Array.sub (counts, j) + 1)  
        end) ))
```

```

(* Phase 2: K scans, one per key, across the blocks *)
val offsets = Seq.tabulate (fn k =>
  Parallel.scan (op+) 0 (0, numBlocks) (fn b => Array.sub (counts, idx (b, k))))
  K
val outputs = Seq.tabulate (fn k =>
  ForkJoin.alloc (Seq.nth (Seq.nth offsets k) numBlocks)) K

(* Phase 3: each block writes out its elements, reusing counts
 * to track the next free slot for each key *)
val _ = Parallel.parfor (0, numBlocks) (fn b =>
  ( Util.for (0, K) (fn k =>
    Array.update (counts, idx (b, k), Seq.nth (Seq.nth offsets k) b))
  ; Util.for (b*B, Int.min (n, (b+1)*B)) (fn i =>
    let
      val x = Seq.nth s i
      val j = idx (b, key x)
      val pos = Array.sub (counts, j)
    in
      Array.update (Seq.nth outputs (key x), pos, x);
      Array.update (counts, j, pos + 1)
    end) ))
in
  Seq.map ArraySlice.full outputs
end

```

Each block does $O(B + K)$ work in phases 1 and 3, and the scans touch Kn/B counters in total:

$$W(n) = O\left(\frac{n}{B}(B + K)\right) = O\left(n + \frac{Kn}{B}\right)$$

$$S(n) = O(B + K + \log n)$$

Choosing $B = K$ gives $O(n)$ work and $O(K + \log n)$ span. (In practice we'd pick something like $B = \max(K, 1000)$.)

Attempt #3: transposing the counters (counting sort)

The counters form a $\frac{n}{B} \times K$ table. Above, we stored it block-major, and ran K separate scans down its columns, producing K separate output arrays.

Suppose instead we store the table **transposed** (key-major): all of key 0's counters (one per block), then all of key 1's, etc. Now we can replace the K scans with a **single scan** across the whole table. The offset for (key k , block b) then counts every element with a smaller key, plus every element with key k in an earlier block. This is exactly the position, in one flat output array, where block b 's first key- k element belongs.

Only a few lines change:

```

fun idx (k, b) = k * numBlocks + b    (* key-major *)
...
val offsets = Parallel.scan (op+) 0 (0, K * numBlocks) (fn j =>
  Array.sub (counts, j))
val output = ForkJoin.alloc n
... (* phase 3: all blocks write into the same output array *)
(* bucket k is the slice from offset (k, 0) to offset (k+1, 0) *)

```

```

val output = ArraySlice.full output
fun start k = Seq.nth offsets (idx (k, 0))
in
  Seq.tabulate (fn k => Seq.subseq output (start k, start (k+1) - start k)) K
end

```

The cost bounds are the same, but all buckets now live in one physical array. It is already sorted, with no extra flattening needed. This final algorithm is called **counting sort**. For example, with $K = 3$, $B = 4$, and input $\langle 2, 0, 1, 2, 1, 0, 2, 1 \rangle$ (two blocks):

(k, b)	(0,0)	(0,1)	(1,0)	(1,1)	(2,0)	(2,1)	
counts	1	1	1	2	2	1	
offsets	0	1	2	3	5	7	8

The output is $\langle 0, 0, 1, 1, 1, 2, 2, 2 \rangle$; e.g., bucket 1 is the slice from offset 2 to offset 5. The accompanying code (`countsort/`) implements and benchmarks all three versions (`-impl filters`, `-impl filters-opt`, `-impl counting-sort`).

Computational Geometry in 2D

We'll work in $\mathbb{R}^2$. A **point** (or vector) is a pair of reals, $p = (x, y)$. We'll write $q - p$ for the vector from p to q .

Parallelogram area and the cross product

For two vectors $u = (u_x, u_y)$ and $v = (v_x, v_y)$, the **cross product** $u \times v = u_x v_y - u_y v_x$ is the **signed area** of the parallelogram with sides u and v . It is positive if v is counterclockwise from u (less than a half-turn), negative if clockwise, and zero if u and v are parallel.

For three points p, q, r , we define `triArea` as the cross product of the two vectors leaving p . (The name is conventional: its magnitude is twice the area of triangle pqr .)

```

type point = real * real
fun sub ((x1, y1), (x2, y2)) = (x1 - x2, y1 - y2)
fun cross ((ux, uy), (vx, vy)) = ux * vy - uy * vx
fun triArea (p, q, r) = cross (sub (q, p), sub (r, p))

```

Two useful facts follow directly.

Line-side test. The sign of `triArea(p, q, r)` tells us which side of the directed line $p \rightarrow q$ the point r is on: positive means r is **above** the line, negative means below, and zero means r is on the line. (Here, “above” is relative to the direction of the line: picture $p \rightarrow q$ pointing left-to-right.)

```

fun above (p, q) r = triArea (p, q, r) > 0.0

```

Distance from a line. A parallelogram's area is its base times its height. With base $q - p$, the height is the distance from r to the line through p and q :

$$\text{dist}(r, \overline{pq}) = \frac{|\text{triArea}(p, q, r)|}{|q - p|}$$

In particular, for a *fixed* p and q , comparing distances to the line $\overline{pq}$ is the same as comparing `triAreas`. No division or square root needed.

Convex hull

Imagine the points as nails in a board, and stretch a rubber band around all of them. When released, the band snaps to the **convex hull**: the smallest convex shape that encloses all the points. We usually describe the hull by the points that the band touches, in order around the boundary. (For simplicity, we'll assume no three input points are collinear.)

An equivalent characterization: a point is on the hull exactly if it is the **furthest point in some direction**. That is, for some direction vector d , it maximizes the dot product $d \cdot p$ over all input points. (For example, the leftmost point is the furthest in direction $(-1, 0)$.)

Quickhull

This gives a divide-and-conquer algorithm, similar in spirit to quicksort. The leftmost and rightmost points, l and r , are on the hull. The line $\overline{lr}$ splits the points into an upper and lower half, and we compute the hull of each half separately with `semihull`.

`semihull` takes points S that are all strictly above $p \rightarrow q$ (where p and q are known hull points). It finds the furthest point m , which must be on the hull. Then, any point inside the triangle pmq cannot be on the hull, so we discard it. The remaining points are either above $p \rightarrow m$ or above $m \rightarrow q$, and we recurse on each in parallel.

```
fun semihull (S: point seq) (p: point, q: point) : point seq =
  if Seq.length S = 0 then Seq.empty ()
  else
    let
      (* the furthest point from the line pq *)
      fun further (a, b) =
        if triArea (p, q, a) >= triArea (p, q, b) then a else b
      val m = Seq.reduce further (Seq.nth S 0) S
      val (L, R) =
        ForkJoin.par (fn () => semihull (Seq.filter (above (p, m)) S) (p, m),
                      fn () => semihull (Seq.filter (above (m, q)) S) (m, q))
    in
      Seq.append (L, Seq.append (Seq.singleton m, R))
    end

  (* assumes S is nonempty *)
fun quickhull (S: point seq) : point seq =
  let
    val l = Seq.reduce (fn (a, b) => if #1 a <= #1 b then a else b) (Seq.nth S 0) S
    val r = Seq.reduce (fn (a, b) => if #1 a >= #1 b then a else b) (Seq.nth S 0) S
    val (upper, lower) =
      ForkJoin.par (fn () => semihull (Seq.filter (above (l, r)) S) (l, r),
                    fn () => semihull (Seq.filter (above (r, l)) S) (r, l))
  in
    Seq.append (Seq.append (Seq.singleton l, upper),
                Seq.append (Seq.singleton r, lower))
  end
```

The output lists the hull points in clockwise order, starting from l . (As with quicksort last lecture, the appends can be avoided with tree flattening.)

Cost. Each call does one `reduce` and two `filters`, which is linear work and $O(\log n)$ span. The calls at any one level of the recursion operate on disjoint subsets of the input, so each level costs $O(n)$ work

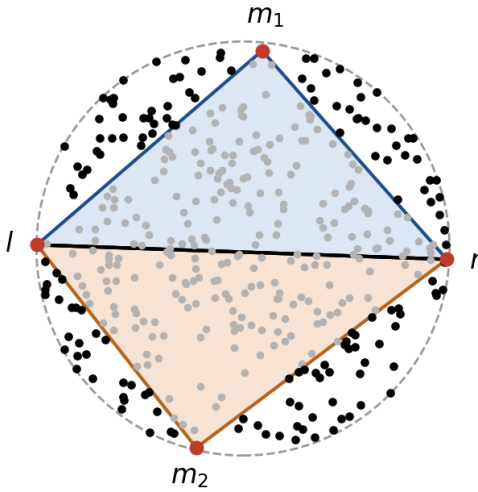
in total. Each level also finds at least one new hull point, so the depth of the recursion is at most h , the number of hull points. This gives $O(nh)$ work and $O(h \log n)$ span, which is $O(n^2)$ work in the worst case. But in practice, most points are discarded early (everything inside the triangle pmq), and quickhull is typically very fast.

An important observation is that, often, the size h of the output hull is very small relative to the input size n . Since quickhull guarantees that it finds a hull vertex on every recursive call, the recursion depth is usually very shallow.

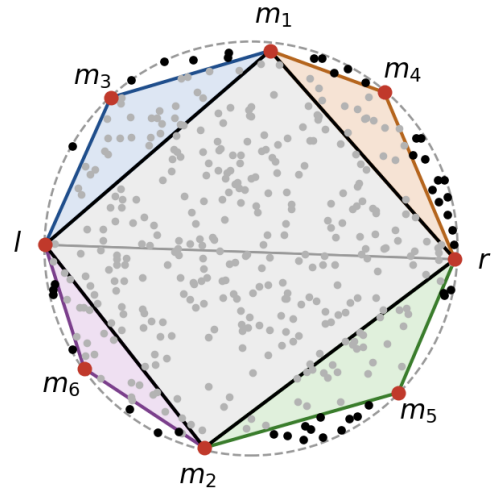
For “nice” datasets (where there are lots of internal points, the hull is small, and the hull vertices themselves are roughly equally spaced around the perimeter), we can argue that quickhull requires at most $O(n \log h)$ work, or even less. Write $W(n, h)$ for the work of `semihull` on an input of size n and output of size h . On a “nice” dataset, we typically manage to split the problem as $W(n, h) = W(n_1, h/2) + W(n_2, h/2) + O(n)$ where both subproblems are less than half the size ($n_1 < n/2$ and $n_2 < n/2$) and the output hull vertices are evenly distributed amongst the subproblems. This gives a recursion depth of $O(\log h)$ with $O(n)$ work per level, for $O(n \log h)$ work in total.

If the subproblem sizes n_1 and n_2 are *significantly* smaller than n (say, where $n_1 + n_2 < n/2$), then the $O(n)$ terms in the recurrence are geometrically decreasing and the total work is $O(n)$.

For example, imagine points distributed uniformly within a circle. The first calls to `semihull` are likely to form (approximately) a square inscribed within that circle, which already eliminates $\sim 2/3$ of the inner area of the circle, as depicted below (where m_1 and m_2 are the first hull points identified by the top-most calls to `semihull`). On deeper calls in the recursion, the subproblem sizes continue to decrease rapidly in a similar way. **On such an input, quickhull is essentially a linear-work algorithm.**



First two `semihull` calls, in parallel



Next four `semihull` calls, in parallel