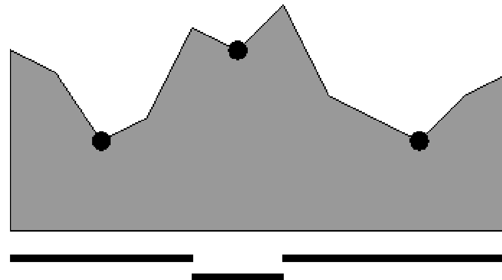
A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

Watersheds

A *watershed* is the area of land that drains into a particular water system - the Seneca Lake watershed, for example, is the area from which water drains into Seneca Lake. The picture below shows a 2D version of this concept - the landscape is shown in gray, the points where water falling on the landscape would collect are shown with black dots, and the watershed for each of the collection points is shown by the span of the black lines below the landscape.



Given an array of n elevation values, report the collection points (as the array index of that point) and their watersheds (as a span of array indexes).

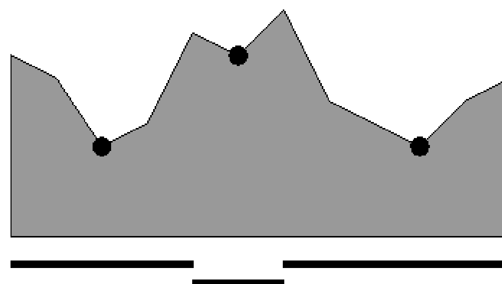
Specifications. State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, distinguish between legal solutions and optimal ones.

Task: Compute the collection points and watersheds for a collection of elevation values.

Input: array of n elevation values

Output: list of collection points (array indexes) and watersheds (spans of array indexes)

Examples.



Examples should illustrate the specifications and/or help with working out an algorithm. An important case to consider here is whether a low point on the edge is a collection point or not.

Boundary cases: a point on the edge is considered a collection point if it is lower than its neighbor.
That point will also be on the watershed boundary.

The other option, that it isn't a collection point, is also valid. Either way, though, the definition used should be stated.

What about special cases like a span of two or more elevations at the same height? Examples of special cases also fall under the rubric of "illustrating the specifications and/or helping with working out an

algorithm" and can be included here. However, from the perspective of an algorithm development process, special cases are not necessarily identified up front — having to consider all legal inputs up front can complicate the algorithm-development process, and that something is a special case isn't necessarily apparent until you have something that handles the other cases. So, there is no harm in including examples of things that occur to you that you'll need to handle, but it isn't essential to identify special cases now.

(What about the boundary case example? It's not really a special case, as every input with $n < 3$ involves boundary cases and boundary cases are common with larger n — the algorithm can only work in very limited circumstances if boundary cases aren't handled. By contrast, there are plenty of inputs which don't involve a span of equal elevations, and an input which doesn't have same-elevation spans at the beginning isn't going to develop them as the problem is divided.)

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

> n is the number of elevation values provided. The smallest non-trivial problem is $n = 1$.

An alternative could be $n = 3$, with the idea that a collection point has to be a low point between two higher points. But nothing in the problem says every elevation profile has to have at least one collection point, and "report the collection points" doesn't exclude "there are none" as an answer.

The smallest problem size should be determined by the smallest input for which there is a sensible answer. For collections, this is often $n = 1$. For a problem involving pairs of distinct elements (such as the stock buy-sell problem), it is $n = 2$ — it's not possible to report a buy-sell pair for fewer than two days.

It is especially important not to conflate "no sensible answer" and "there are none" when such cases are not cleanly delineated by input size. In this case, there are four[1] possible configurations for $n = 3$: three values decreasing, three values increasing, middle value lowest, and middle value highest. Only one of these meets the definition of a collection point as having higher points on either side, so declaring the smallest problem size to be 3 doesn't prevent having to handle "there are none" cases. Furthermore, unless n is known to be limited to evenly-divisible values, additional base cases will be required because $n = 4$ and $n = 5$ can't be subdivided without resulting in a problem smaller than 3.

Targets. What are the time and space requirements for your solution?

For divide-and-conquer, the goal is often to beat the brute force algorithm. A brute force algorithm is a straightforward way of solving the problem that isn't necessarily efficient — the correctness should be obvious and it is generally not too difficult to come up with.

> A collection point is a local minimum and its watershed is the increasing-elevation area on each side — for each point in the array, if the points on either side are higher, it is a collection point. When a collection point is found, scan backward and forward from its position to find the boundaries of the watershed.
>
> A quick assessment is that this is $O(n^2)$ — there are n points to check and, for each collection point, the whole array might be scanned to find the boundaries for its watershed. However, only the points on the watershed boundaries will be included in more than one watershed (and there are only as many watersheds as collection points), so while there may be a lot of array examined to find the watershed for one collection point, the total number of points looked out to find all the watersheds is only $O(n)$. So the running time of this brute force algorithm is actually $O(n)$.

You might realize that collection points are local minima and watershed boundaries are local maxima, so you could solve the problem by simply scanning through the array once, looking for those points. This algorithm is a little more clever than the brute force approach since it requires some reasoning about what watershed boundary points look like — it's not wrong to give it instead, but a key point is that this step is *not* about figuring out a clever way to solve the problem but to provide an algorithm that is clearly correct as a benchmark to try to improve on with a divide-and-conquer approach.

[1] excluding cases where there are two or more consecutive elevations with the same value

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

The paradigm — in this case, divide-and-conquer — is biggest shaper of the algorithm, so there's not really anything else to say here.

Approaches. Consider specific patterns — what would they look like if applied to this problem?

> Process input: split the array in half, have friends find the collection points and watersheds in each half, then combine those results (dealing with the middle, where watersheds may span halves)
>
> Produce output: the goal is to simply append the second friend's collection points and watersheds to the first's, so we'd have to split the elevations between watersheds so that no watershed spans the two halves

Generalize / define subproblems. Friends get smaller versions of the original problem. Define the task, input, and output for the subproblems.

Our friends solve the same problem we are tasked with solving, just a smaller version of it. "Generalizing" means the subproblem can be solved for just a portion of the input instead of only all of it.

> Original problem: `watersheds(A)` — the watersheds in `A`
>
> Subproblem / generalized problem: `watersheds(A,low,high)` — find the collection points and watersheds in `A[low..high]`
>
> Input: n-element array `A` and indexes `low` $\leq$ `high`
>
> Output: the collection points and their watersheds

Main case. Split the problem, solve the subproblems, combine the subproblem solutions into the solution for this problem: the main case addresses how to solve a typical large problem instance. Specify how many friends are needed and what is handed to each friend, as well as how to generate what is handed to the friends and how the results the friends hand back are combined to solve the problem.

If a produce output approach doesn't seem obviously simpler, a process input approach is a good way to start. In this case, combining the solutions seems more complicated for process input, but finding a watershed boundary to split on is not O(1) and there's no guarantee that there will be a (roughly) even split, resulting in a worse worst-case running time. So let's stick with process input.

Process input splits the input array in half, finds the collection points and watersheds in each half, and then combines those results into the collection points and watersheds for the whole. But how? Pictures and examples can help...

Observe that all of the points in the region are included in exactly one of the two halves, and so all of the collection points in the region will be in either the first half or the second half. For the last point in the first half and the first point in the second half, however, the friends can only assess one of the point's neighbors and so those points might be reported as collection points (the neighbor in the half is higher) when they actually aren't (the neighbor in the other half is lower). Thus, when concatenating the collection points returned by the two friends, it is necessary to check the last point reported in the first half and the first point reported in the second half to determine if they belong in the combined list of collection points.

For the watersheds, one of the watersheds for the full region will span the split between halves. (The rest are contained entirely in the first half or the second half.) Thus, when concatenating the watersheds returned by the two friends, it is necessary to determine how to combine the last watershed of the first half and the first watershed of the second half. Let $w = [w_s, w_f]$ be the last watershed in the first half and $w' = [w'_s, w'_f]$ be the first watershed in the second half. There are two cases: if either w_f or w'_s is a local maximum, it's the dividing point between watersheds — w' or w, respectively, should be extended to cover

the gap between w_f and w'_s. Otherwise neither is a local maximum and w and w' are a single watershed that was split because of the boundary — replace both watersheds with $[w_s, w'_f]$.

split A[low..high] in half: mid = (low+high)/2
find the collection points and watersheds for A[low..mid] and A[mid+1..high]
concatenate the list of collection points for each half, including the last one in
  the first half and the first one in the second half only if they are local minima
concatenate the list of watersheds for each half, handling the last watershed in
  the first half and the first watershed in the second half as follows:
    let $w = [w_s, w_f]$ be the last watershed in the first half
    let $w' = [w'_s, w'_f]$ be the first watershed in the second half
    if w'_s is a local maximum, replace w with $[w_s, w'_s]$
    else if w_f is a local maximum, replace w' with $[w_f, w'_f]$
    otherwise replace both watersheds with the single watershed $[w_s, w'_f]$

Base case(s). Address how to solve the smallest problem(s).

The smallest problem is $n = 1$ i.e. `low=high`. Report `low` as the index for the collection point and [`low`,`high`] as the watershed.

Top level. The top level puts the context around the recursion. Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem. Setup covers whatever must happen before the initial subproblem is solved, and wrapup covers whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

Initial subproblem. `watersheds(A,0,n-1`

Setup. Nothing else is needed.

Wrapup. Nothing is needed — the initial subproblem returns the collection points and watersheds as desired.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for special cases as needed.

What if there are two or more consecutive points with the same elevation? "Local minimum" and "local maximum" assume that a point is smaller or bigger than both of its neighbors.

When addressing a special case, an important question is what *should* happen. Only if there's no reasonable solution should the specifications be updated to disallow that input.

Treat a span of equal-elevation points the same way as a single point — a span with larger values on either end is a collection area (rather than a collection point), a span with smaller values on either end forms the boundary between adjacent watersheds (the span itself is not part of any watershed), and a span with a larger value on one side and a smaller value on the other is fully contained within a watershed.

To handle this, add a preprocessing step (in the setup) where spans of equal elevation are replaced with a single value and a postprocessing step (in the wrapup) where those condensed values are expanded back out — a collection point is replaced by the span, watershed boundaries are replaced by the point on the appropriate end of the span, and everywhere the indexes of the points are updated to match the original array.

Algorithm. Assemble the algorithm from the previous steps and state it.

`watersheds(A)` — find the collection points and watersheds in `A`

Input: array `A` of n elevation values

Output: the collection points and their watersheds

```
return watersheds(A,0,n-1)
```

`watersheds(A,low,high)` — find the collection points and watersheds in `A[low..high]`

Input: n-element array `A` and indexes `low` $\leq$ `high`

Output: the collection points and their watersheds in `A[low..high]`

```
 if n = 1
   return (low,[low,high]) as the collection point and watershed
else
   split A[low..high] in half:  mid = (low+high)/2
   find the collection points and watersheds for A[low..mid] and A[mid+1..high]
   concatenate the list of collection points for each half, including the
     last one in the first half and the first one in the second half only
     if they are local minima
   concatenate the list of watersheds for each half, handling the last watershed
     in the first half and the first watershed in the second half as follows:
       let w = [w_s, w_f] be the last watershed in the first half
       let w' = [w'_s, w'_f] be the first watershed in the second half
       if w'_s is a local maximum, replace w with [w_s, w'_s]
       else if w_f is a local maximum, replace w' with [w_f, w'_f]
       otherwise replace both watersheds with the single watershed [w_s, w'_f]
```

Termination. Show that the recursion ends. For making progress, explain why each subproblem is smaller. For reaching the end, show that a base case is always reached.

Making progress. At each step the range `low..high` is cut in half.

Termination. As long as there's at least two points, the interval can be cut in half. The only time no split is possible is when there's only one point, which means `low` equals `high` — and that is the base case.

Correctness. Show that the algorithm is correct.

Establish the base case(s). Explain why the solution is correct for each base case.

From the specifications and examples — a point on the boundary is a collection point and forms the endpoint of its watershed. Thus a single point is a collection point and its watershed contains only it.

Show the main case. Assume that the friends return the correct results for their subproblem, and explain why the correct answer is then produced from those results.

Splitting into `A[low..mid]` and `A[mid+1..high]` means that every point is in exactly one of the two halves.

Observe that all of the points in the region are included in exactly one of the two halves, and so all of the collection points in the region will be in either the first half or the second half. For the last point in the first half and the first point in the second half, however, the friends can only

assess one of the point's neighbors and so those points might be reported as collection points (the neighbor in the half is higher) when they actually aren't (the neighbor in the other half is lower). Thus, when concatenating the collection points returned by the two friends, it is necessary to check the last point reported in the first half and the first point reported in the second half to determine if they belong in the combined list of collection points.

For the watersheds, one of the watersheds for the full region will span the split between halves. (The rest are contained entirely in the first half or the second half.) Thus, when concatenating the watersheds returned by the two friends, it is necessary to determine how to combine the last watershed of the first half and the first watershed of the second half. Let $w = [w_s, w_f]$ be the last watershed in the first half and $w' = [w'_s, w'_f]$ be the first watershed in the second half. There are two cases: if either w_f or w'_s is a local maximum, it's the dividing point between watersheds — w' or w, respectively, should be extended to cover the gap between w_f and w'_s. Otherwise neither is a local maximum and w and w' are a single watershed that was split because of the boundary — replace both watersheds with $[w_s, w'_f]$.

Final answer. Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

The subproblem is defined as covering the span `A[low..high]` inclusive, so `A[0..n-1]` is the full array.

Time and space. Assess the running time and space requirements of the algorithm.

- Base case: $O(1)$
- Main case: $T(n) = 2T(n/2) + O(1)$ — with the right representation (such as a linked list), concatenating two lists can be done in $O(1)$ time. There are only two collection points and two watersheds to handle specially, and they are at the end or the beginning of a list, so $O(1)$ to find and then $O(1)$ to handle.
- Setup: scan through the array, condensing spans of equal elevation. $O(n)$
- Wrapup: scan through the collection points and watersheds returning, updating the spans/indexes to un-condense spans of equal elevation. $O(n)$

Solving the recurrence relation yields $O(n)$. Setup and wrapup are only done once (not once per subproblem!), so the total time is $O(n) + O(n) + O(n) = O(n)$.