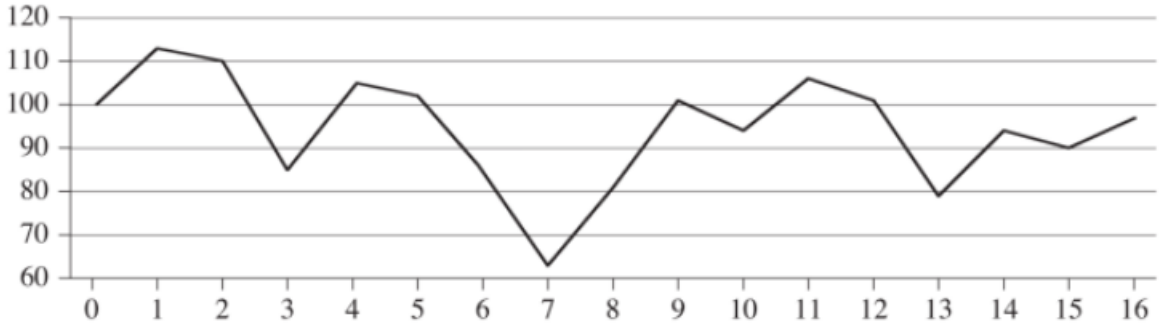
A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

Stocks

Given the price of a stock over an n-day period, determine the best time to have bought and sold 1000 shares of that stock. (Buy and sell once, on different days.)



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

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: Given the price of a stock over an n-day period, determine the best time to have bought and sold 1000 shares of that stock. (Buy and sell once, on different days.)
>
> Input: array of n stock prices
>
> Output: a pair of buy and sell dates
>
> Legal output: buy date is before the sell date
>
> Optimization goal: the buy/sell pair date that maximizes profit

Examples. If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

> For the figure above: buy on day 7, sell on day 11 — profit is 106-73 = 33 per share.

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 days. A smaller problem means fewer days. The smallest problem is 2 days, because buy and sell must be different days.

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 this problem), it is $n = 2$ — it's not possible to report a buy-sell pair for fewer than two days.

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.

> The brute force algorithm is to try all possible buy/sell pairs and pick the one with the max profit. This is $\Theta(n^2)$.

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.

> Beating $O(n^2)$ means that it is not possible to compare every pair of elements directly — and avoiding that by splitting into separate subproblems is exactly what divide-and-conquer is about.

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

> A *process input* approach would mean splitting the input in half, having friends produce the solution for each half — i.e. the buy/sell pairs that maximize profit for each half — and then compute the overall buy/sell pair.

A *produce output* approach has each friend producing part of the solution, but that's not applicable here — the solution isn't a collection of things, and the only parts would be a buy date and a sell date, but the friends have to solve the same problem we are.

Narrowing the search space isn't applicable in max-finding cases — while we are looking for element(s) that are part of the input, a max can't be identified without the context of the rest of the input.

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" typically means adding parameters so that the subproblem can be solved for just a portion of the input instead of only all of it.

> Original problem: Find the buy/sell dates that maximize the profit over all n days.
>
> Subproblem / generalized problem: `maxprofit(A,low,high)` — find the buy/sell dates that maximize the profit over an interval (`low`,`high`) of the days.
>
> Input: array `A` of stock prices and indexes `low` and `high` (inclusive) defining the span of days
>
> Output: the buy/sell date pair in `A[low..high]` which maximizes the profit

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

> The smallest problem is 2 days i.e. `high=low+1` — buy on day `low`, sell on day `high`.

We also need to handle 3 days, because that cannot be split into smaller problems of size 2 or larger.

> For three days i.e. `high=low+2` — let `mid` be the index between `low` and `high` (`mid` = `low`+1). Report the one of (`low`,`mid`), (`low`,`high`), (`mid`,`high`) that has the maximum profit.

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

The steps of the development process don't need to be done in order, so, since we're thinking about correctness as we develop the algorithm, it is convenient to write down those steps here.

For two days, the only legal buy/sell pair (different dates, buy before sell) is (`low`,`high`) so that is the correct answer for maximizing profit.

For three days, there are only three legal buy/sell pairs — all three are considered, and the correct answer is the one that maximizes profit.

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.

This is where the approaches come in — only process input was applicable. As identified above, that gives us the framework of split the input in half, have the friends solve each half (giving two buy/sell pairs), and figure out the overall buy/sell pair from that:

```
split A[low..high] in half:  mid ← (low+high)/2
(buy1,sell1) ← maxprofit(A,low,mid)
(buy2,sell2) ← maxprofit(A,mid+1,high)
return ??  as the buy/sell pair that maximizes profit
```

So, given the buy/sell pairs from each half, what's the overall answer? Is it just the better of the two? No, because there are three cases: the max buy/sell could be entirely within the first half of the array, it could be entirely with the second half of the array, or the buy day could be in the first half and the sell day in the second half. The friends have covered the first two possibilities but we still need to handle the third.

How to find the buy/sell pair that maximizes profit with the buy day in the first half and the sell in the second half? The brute force option would be to compare every first-half buy day with every second-half sell day — but that's $n/2 \times n/2 = \Theta(n^2)$ pairs, which doesn't seem good for trying to beat the $\Theta(n^2)$ brute force for the whole problem.

Observe: profit is maximized by buying low and selling high, so what about buying on the min-price day in the first half and selling on the max-price day in the second half? That's a legal buy/sell pair...

```
split A[low..high] in half:  mid ← (low+high)/2
(buy1,sell1) ← maxprofit(A,low,mid)
(buy2,sell2) ← maxprofit(A,mid+1,high)
buy3 ← day with lowest price in A[low..mid] (inclusive)
sell3 ← day with highest price in A[mid+1..high] (inclusive)
report the one of (buy1,sell1), (buy2,sell2), (buy3,sell3) that has the maximum profit
```

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.

There are three cases: the max buy/sell could be entirely within the first half of the array, it could be entirely with the second half of the array, or the buy day could be in the first half and the sell day in the second half. (A buy day in the second half and a sell day in the first half is not a legal solution.)

(`buy1`, `sell1`) and (`buy2`, `sell2`) cover the first two cases.

Profit is maximized by buying low and selling high, so buying on the min-price day in the first half and selling on the max-price day in the second half is the maximum profit in case 3, and that is a legal buy/sell pair because the buy date is before the sell date.

Because these are the only cases, the best overall buy/sell pair is the best of these three.

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.** `maxprofit(A,0,n-1)`
>
> **Setup.** Nothing else is needed.
>
> **Wrapup.** Nothing else is needed — the buy/sell pair is the answer.

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.

> There isn't any setup or wrapup, and the initial subproblem `maxprofit(A,0,n-1` includes all of the original elements so that's the buy/sell pair that maximizes profit.

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

> Small inputs (like $n = 0$ and $n = 1$) aren't legal inputs, so no potential special cases there.
>
> There aren't any other special cases.

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

> `maxprofit(A)` — Determine the best time to have bought and sold 1000 shares of stock.
>
> Input: array of stock prices over an n-day period
>
> Output: a pair of buy and sell dates that maximizes profit
>
> ```
> return maxprofit(A,0,n-1)
> ```
>
> `maxprofit(A,low,high)` — Determine the best time to have bought and sold 1000 shares of stock within the interval `low`..`high` (inclusive).
>
> Input: array of stock prices over an n-day period and indexes `low` and `high` (inclusive) defining the span of days
>
> Output: a pair of buy and sell dates that maximizes profit within the interval `low`..`high` (inclusive)
>
> ```
> if high = low+1
>    report (low,high)
> else if high = low+2
>    mid ← low+1
>    report the one of (low,mid), (low,high), (mid,high) that has the maximum
> profit
> else
>    split A[low..high] in half:  mid ← (low+high)/2
>    (buy1,sell1) ← maxprofit(A,low,mid)
>    (buy2,sell2) ← maxprofit(A,mid+1,high)
>    buy3 ← day with lowest price in A[low..mid] (inclusive)
>    sell3 ← day with highest price in A[mid+1..high] (inclusive)
>    report the one of (buy1,sell1), (buy2,sell2), (buy3,sell3) that has the
> maximum profit
> ```

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. Since `high` $\geq$ `low` $+ 3$, `mid` $>$ `low` and `mid` $<$ `high` so there is at least one element in each half and thus both subproblems are smaller.

Reaching the end. For $n = 4$, cutting in half means two problems of size 2. For $n = 5$, cutting in half means problems of size 2 and size 3. For any $n > 5$, the subproblems won't be any smaller than they are for $n = 5$, so it's not possible to skip the identified base cases.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

We only have simple steps or well-known tasks (such as "find the min or max element in an array"), so nothing needs to be further defined.

Time and space. Assess the running time and space requirements of the algorithm given the implementation identified.

The base cases take O(1) time each. For the main case (n elements), consider each step:

- Dividing the array is O(1) (to compute (low+high)/2).
- Two problems of size $n/2$ are generated.
- Finding the minimum element in half the array and the maximum element in the other half is O(n).
- Computing the profit for three pairs and computing the max of three values is O(1).

This leads to the recurrence relation $T(n) = 2T(n/2) + n$. With T(1) = 1, this yields $T(n) = \Theta(n \log n)$.

Space requirements are O(1) for a few temporary variables.

Is this good? It's better than the $O(n^2)$ brute force algorithm... But can we do better? It doesn't seem plausible to be able to do better than $O(n)$, since it seems like we'd have to at least look at every element once, but is $O(n)$ possible?

The work in the algorithm given above comes in finding the min and max elements in the two halves of the array... but the friends are already working with the two halves. Could we have them do some more work?

`maxprofit(A,low,high)` — find the buy/sell dates that maximize the profit over an interval (`low`,`high`) of the days along with the min and max stock prices in that interval.

Input: array `A` of stock prices and indexes `low` and `high` (inclusive) defining the span of days

Output: the buy/sell date pair in `A[low..high]` which maximizes the profit and the minimum and maximum stock prices in `A[low..high]`

```
split A[low..high] in half:  mid ← (low+high)/2
(buy1,sell1,min1,max1) ← maxprofit(A,low,mid)
(buy2,sell2,min2,max2) ← maxprofit(A,mid+1,high)
report the one of (buy1,sell1), (buy2,sell2), (min1,max2) that has the maximum
profit, along with min(min1,min2) as the minimum element and max(max1,max2)
as the maximum element
```

Then we have $T(n) = 2T(n/2) + O(1) = O(n)$. Success!

(To formally complete this, the base cases also need to be updated to show how the min and max elements are computed, and the correctness argument needs to be updated to explain why the min and max elements for the whole interval are computed correctly.)