5.2 Stock Spans

Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

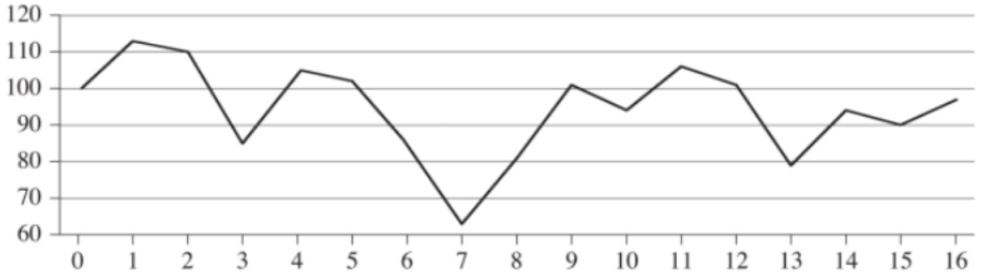
- *Specifications.*

  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?

  Task: find the best time to have bought and sold 1000 shares of a stock (buy and sell once, on different days; $buy < sell$)
  Input: array S of n daily stock prices
  Output: buy day, sell day

- *Examples.*

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



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

  Given the above stock prices, the best time to have bought and sold is to have bought on day 7 and sold on day 11 (profit = 106-63 = 43 per share).

Identify avenues of attack.

Determine the algorithmic approach(es) to try.

- *Targets.*

  Divide-and-conquer algorithms often seek to improve on a polynomial-time iterative brute-force running time. If there aren't other targets, identify the brute force algorithm and its running time.

  The task is to find the best pair of buy and sell days, so the brute force approach would be to compute the profit for every legal pair of buy and sell days ($buy < sell$) and choose the pair with the max profit. This $\Theta(n^2)$.

- *Patterns.*

> Consider the divide-and-conquer patterns.

> There are two main patterns: easy split and easy merge. The goal in this step is to think about how this task might fit into the patterns — we're not trying to come up with an algorithm, just to identify what pieces need to be sorted out to complete an algorithm in a particular vein.
>
> Easy split is potentially applicable when the input is a collection of elements — split the elements into two groups (first half, second half). Easy merge is potentially applicable when the output is a collection of elements — have one friend produce the first part of the solution and the other friend produce the second part.

Easy split: The input is an array of elements, so the easy split is to split the array in half. Then have friends find the best (buy,sell) pair in each half, with the final step being to determine the overall best (buy,sell) pair.

Easy merge is not applicable.

> Easy merge is not applicable because the result is a single thing (a pair of days); it's not possible to have one friend find the buy day and the other find the sell day because these things are not independent of each other due to the requirement that $buy < sell$ and because these are not the same tasks as required by the divide-and-conquer paradigm.

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*

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

  The size is the number of elements in S i.e. the number of daily stock prices.

- *Generalize / define subproblems.*

  > Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.

  > The friends always have the same task as we do — if we are finding the best (buy,sell) pair, they are finding the best (buy,sell) pair. But it is a generalized version of the original task — if the original task is to find a (buy,sell) pair for the whole array, a generalized version is to find one for some portion of the array. This is generalized because if you can find the best (buy,sell) pair in a portion of the array, you can also find the best (buy,sell) pair in the whole array — but if you can only handle the whole array, there's no way to only handle a portion.

  Subproblem task: find the best time to have bought and sold 1000 shares of a stock within a range $low..high$ (inclusive) of the array (buy and sell once, on different days; $low \leq buy < sell \leq high$)
  Subproblem input: array S of n daily stock prices, values $low \leq high$
  Subproblem output: buy day, sell day

- *Base case(s).*

  > Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.

  > The smallest meaningful size of problem would be $n = 2$ because buy and sell must be on different days.

  For $n = 2$ ($low = high - 1$), return $(low, high)$ — the only legal (buy,sell) pair with two days.

- *Main case.*

> Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.

> The easy split pattern gives the outline of the main case: split the array S into the first half and the second half, have the friends solve each half, and then (somehow) figure out how to determine the overall (buy,sell) pair given the results from the friends.
>
> But how to determine the overall (buy,sell) pair? Ultimately specific tactics depend on the particular problem. In this case, consider the brute force algorithm — check all of the legal (buy,sell) pairs to find the max profit combination. Friend 1's answer is the best of the pairs where both buy and sell days are in the first half and friend 2's answer is the best of the pairs where both buy and sell days are in the second half, so we don't need to re-check those. What is missing are those pairs where the buy day is in the first half and the sell day is in the second half. (The other way around is not legal because you must buy before selling.)

```
// split the input
mid ← (low + high)/2

// recursively solve the two parts
(buy1, sell1) ← stocks(S,low,mid)
(buy2, sell2) ← stocks(S,mid+1,high)

// return the answer
let (buy3, sell3) be the best (buy,sell) pair where low ≤ buy3 ≤ mid and
  mid + 1 ≤ sell3 ≤ high
return the best of (buy1, sell1), (buy2, sell2), and (buy3, sell3)
```

> The algorithm should be specified in enough detail to be able to establish correctness, but not more detail than that. The process of finding $(buy3, sell3)$ is not fully specified but the end result is understood and that's all that is needed to continue on with the algorithm — what is necessary for correct functioning is that $(buy3, sell3)$ is the best of the first half buy, second half sell pairs, not the process by which that pair was found.

- *Top level.*

  > The top level puts the context around the recursion.

  - *Initial subproblem.*

    > Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    ```
    stocks(S,0,n-1)
    ```

  - *Setup.*

    > Whatever must happen before the initial subproblem is solved.

    Nothing to do — find the best (buy,sell) pair in S as given.

  - *Wrapup.*

    > Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

    Nothing to do — the best (buy,sell) pair is S is the desired result.

- *Special cases.*

> Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

> Typical special cases to consider for collections are empty collections and collections with a single element. Also consider if particular values could cause problems.

$n < 2$. Having an empty array or an array with only one element is illegal. This can be discounted at the top level as illegal input, but it can also arise when $n = 3$ and the input is split in half, resulting in problems of size 1 and size 2.

Duplicate elements. This could result in a non-unique solution. However, it then doesn't matter which pair is chosen because the profit is the same either way. No special handling is needed.

Only decreasing values so no (buy,sell) pair results in a profit. Taking the max price difference between buy and sell days also finds the smallest loss, so no special handling needed.

- *Algorithm.*

> Assemble the algorithm from the previous steps and state it.
>
> There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

```
algorithm stocks(S)
 Task: find the best time to have bought and sold 1000 shares of a stock (buy and
  sell once, on different days; buy < sell)
 Input: array S of n daily stock prices
 Output: buy day, sell day

 stocks(S,0,n-1)
```

```
algorithm stocks(S,low,high)

 Task: find the best time to have bought and sold 1000 shares of a stock within a
  range low..high (inclusive) of the array (buy and sell once, on different days;
  low ≤ buy < sell ≤ high)
 Input: array S of n daily stock prices, values low ≤ high
 Output: buy day, sell day

 if low == high-1          // n=2
   return (low, high)

 else if low == high-2     // n=3
   return the best of (low, low+1), (low+1, high), and (low, high)

 else
    // split the input
    mid ← (low + high)/2

    // recursively solve the two parts
    (buy1, sell1) ← stocks(S,low,mid)
    (buy2, sell2) ← stocks(S,mid+1,high)

    // return the answer
    let (buy3, sell3) be the best (buy,sell) pair where low ≤ buy3 ≤ mid and
     mid+1 ≤ sell3 ≤ high
    return the best of (buy1, sell1), (buy2, sell2), and (buy3, sell3)
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*
  | Show that the recursion — and thus the algorithm — always terminates.

  - *Making progress.*
    | Explain why what each of your friends get is a smaller instance of the problem.

    The problem is only split when $n \geq 4$. For $n = 4$, $high = low + 3$ and so $mid = \lfloor (low + low + 3)/2 \rfloor = low + 1$. Since $low + 1 > low$ and $low + 2 < high = low + 3$, both subproblems are smaller than n.
  - *The end is reached.*
    | Explain why a base case is always reached.

    For $n \geq 4$, the problem is split into subproblems of size $\lfloor n/2 \rfloor \geq 2$ and $\lceil n/2 \rceil \geq 2$. As a result, base cases $n = 2$ and $n = 3$ are sufficient.

- *Correctness.*
  | Show that the algorithm is correct.

  In all cases, the solution is correct because the answer returned is the max of all legal (buy,sell) pairs (where $low \leq buy < sell \leq high$).

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

    For $n = 2$ there is only one legal (buy,sell) pair. For $n = 3$ there are only three possible legal (buy,sell) pairs and they are all checked.
  - *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.

    For $n \geq 4$, friend 1's answer is the best of the pairs where both buy and sell days are in the first half and friend 2's answer is the best of the pairs where both buy and sell days are in the second half. so we don't need to re-check those. What is missing are those pairs where the buy day is in the first half and the sell day is in the second half and those are explicitly checked.
  - *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.

    `stocks(S,0,n-1)` covers the whole array so the best (buy,sell) pair in $S[0..n-1]$ is the best (buy,sell) pair overall.

Determine efficiency.

| Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  | Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

  All of the (buy,sell) pairs where $low \leq buy3 \leq mid$ and $mid + 1 \leq sell3 \leq high$ can be enumerated with nested loops.

- *Time and space.*

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

> For a recursive algorithm, write a recurrence relation for the running time. Use the table discussed in class to find a big-Oh for each recurrence relation.

The only step that isn't $\Theta(1)$ in the main case is finding the best (buy,sell) pair where $low \leq buy3 \leq mid$ and $mid + 1 \leq sell3 \leq high$. This can be done with nested loops where each loop repeats $n/2$ times, resulting in a $\Theta(n^2)$ runtime for that step.

Then $T(n) = 2T(n/2) + O(n^2) = \Theta(n^2)$.

- *Room for improvement.*

> Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement: Can you leverage the work the friends are doing and have them hand back more in order to reduce your computation? Do you need to pass all of the elements off to friends or can some be eliminated?

$\Theta(n^2)$ is the same as the brute force algorithm.

To do better means decreasing the number and/or size of the subproblems or improving the work done outside of the recursion itself. Since all of the days need to be considered, it doesn't seem likely that it would be possible to decrease the number and/or size of the subproblems, so we focus on the additional work done.

To maximize profit, we want to buy as low as possible and sell as high as possible. Picking the buy day as the day with the overall lowest price and the sell day as the day with the overall highest price yields the max profit unless it is an illegal pair — but with the buy date always in the first half and the sell date always in the second half, it is impossible to have an illegal pairing. Thus

> `let` $(buy3, sell3)$ `be the best (buy,sell) pair where` $low \leq buy3 \leq mid$ `and` $mid + 1 \leq sell3 \leq high$

can be replaced by

> `let` $buy3$ `be the day` $low \leq buy3 \leq mid$ `with the lowest price`
> `let` $sell3$ `be the day` $mid + 1 \leq buy3 \leq mid$ `with the highest price`

This now takes $\Theta(n)$ time, resulting in the recurrence relation $T(n) = 2T(n/2) + O(n) = O(n \log n)$. This is better than the brute force algorithm, but is it possible to do even better?

"Even better" likely means $\Theta(1)$ to find the best (buy,sell) pair spanning the two halves. $\Theta(1)$ means a value is known without having to compute it. What if the friends also returned the days with the min and max prices?

```
// split the input
mid ← (low + high)/2

// recursively solve the two parts
(buy1, sell1, min1, max1) ← stocks(S,low,mid)
(buy2, sell2, min2, max2) ← stocks(S,mid+1,high)

// return the answer
(buy3, sell3) ← (min1, max2)    // best (buy,sell) pair spanning the two halves
return the best of (buy1, sell1), (buy2, sell2), and (buy3, sell3) along with
  min(min1, min2) and max(max1, max2)
```

Now $T(n) = 2T(n/2) + \Theta(1) = O(n)$.

> To complete the development of the algorithm, update the statement of the complete algorithm with this version of the main case and include the explanation of why buying on the lowest day in the first half and selling on the highest day in the second half is the best (buy,sell) pair spanning the two halves in the main case correctness argument.