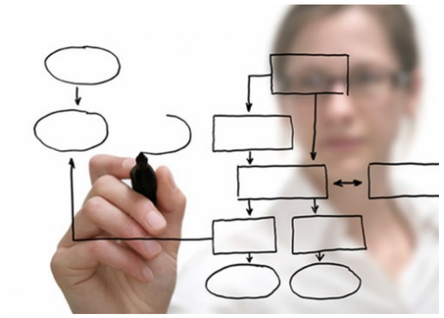
CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT



Distributed Data Management

11: Introduction to Divide & Conquer Algorithms

In Previous Lectures…

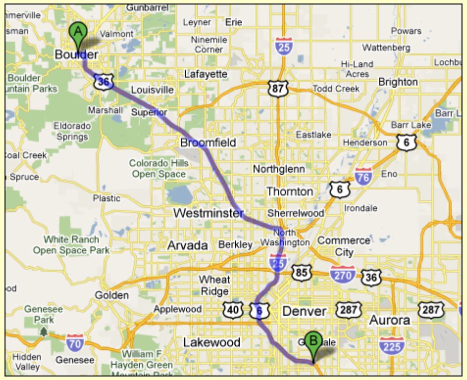
❑ Computing-driven Society loop:

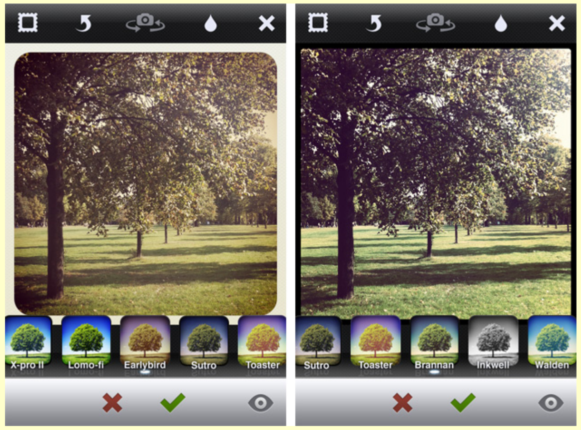
i. Need to solve **problems**.
ii. Problems that require complex strategies (**algorithms**) to be addressed.
iii. Algorithms that are implemented in programming languages via **programs**.
iv. Programs that are executed (run) in **computers**.

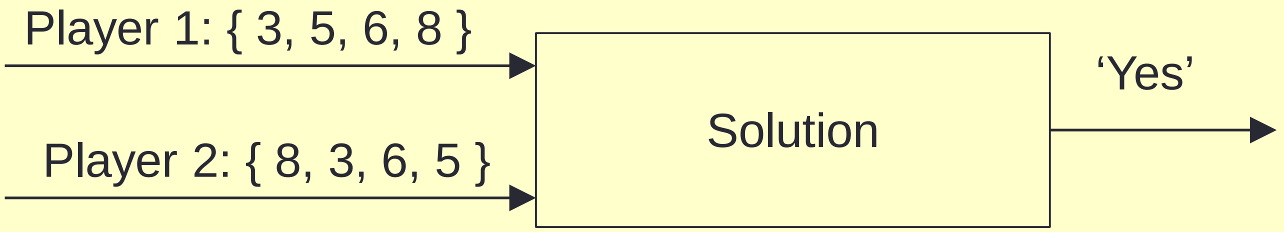
❑ Try to think of Computer Science (in general) in terms of this loop.

In Previous Lectures…

- Examples of problems to be solved:







In Previous Lectures…

Big Data Revolution

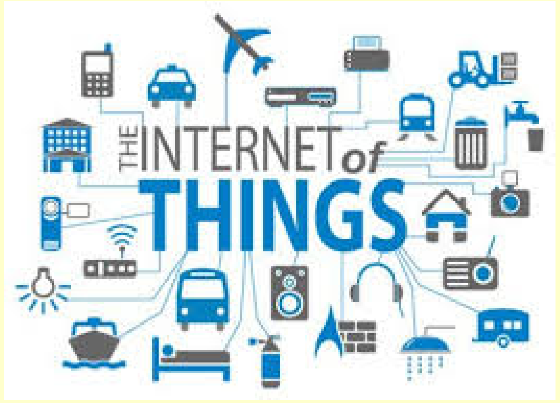
Problem to be solved:
Gather data and analyse it, so as to get some insights / hidden knowledge from it.

- ❑ New problem? **No.**
- ❑ Thus, why is this problem driving a technological revolution?
  - o Develop a more efficient hard drive, processor or data base so as to store and analyse data → *Technological Evolution*.
  - o Modify the hard drive, processor and data base architectures so as to store and analyse data → ***Technological Revolution***.

In Previous Lectures…

Big Data Revolution Ingredients

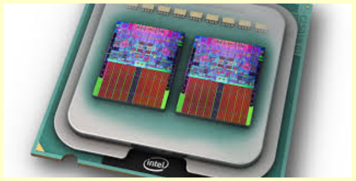
- ❑ On the one hand…data scalability!



In Previous Lectures…

<u>Big Data Revolution Ingredients</u>

- ❑ On the other hand…processing scalability!



In Previous Lectures…

Our two main definitions:

- ❑ Big Data refers to a huge volume of data that cannot be stored and processed with a traditional approach within a given time frame.

- ❑ Analytics for large data sets focus on providing efficient analysis.
  - It is all about turning the data into high-quality information, providing deeper insight about the business to enable better decisions.

Outline

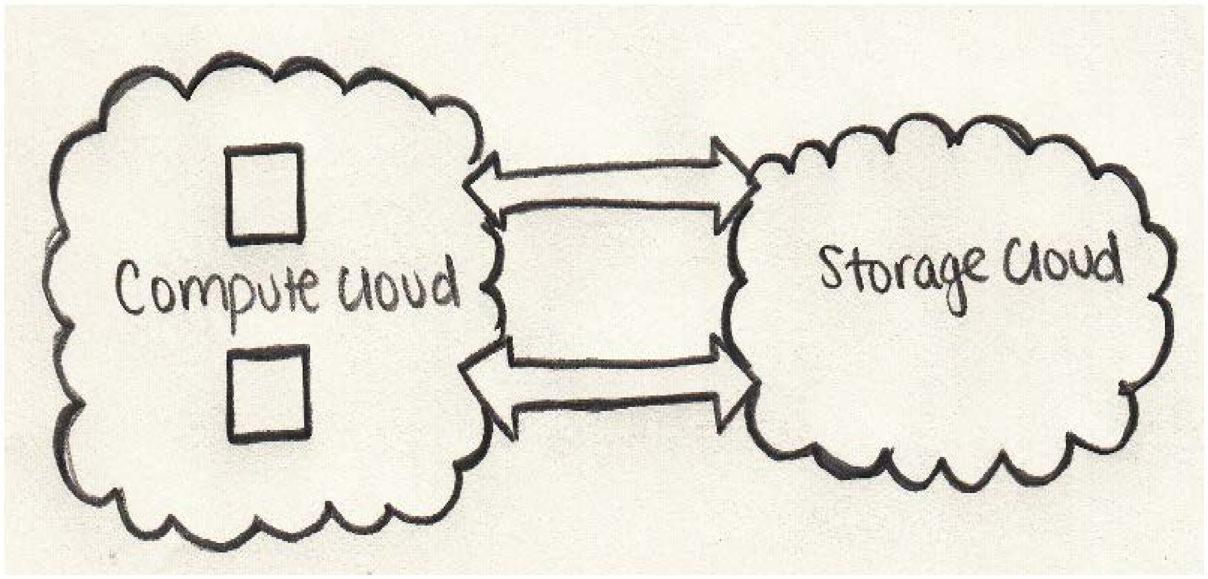
1. Big Data Analytics.
2. Divide & Conquer Algorithms Schema.
3. Recursion: A Possible Instantiation of the Schema.
4. Concurrency: A Possible Parallelisation of the Map.

Outline

1. Big Data Analytics.
2. Divide & Conquer Algorithms Schema.
3. Recursion: A Possible Instantiation of the Schema.
4. Concurrency: A Possible Parallelisation of the Map.

Big Data Analytics

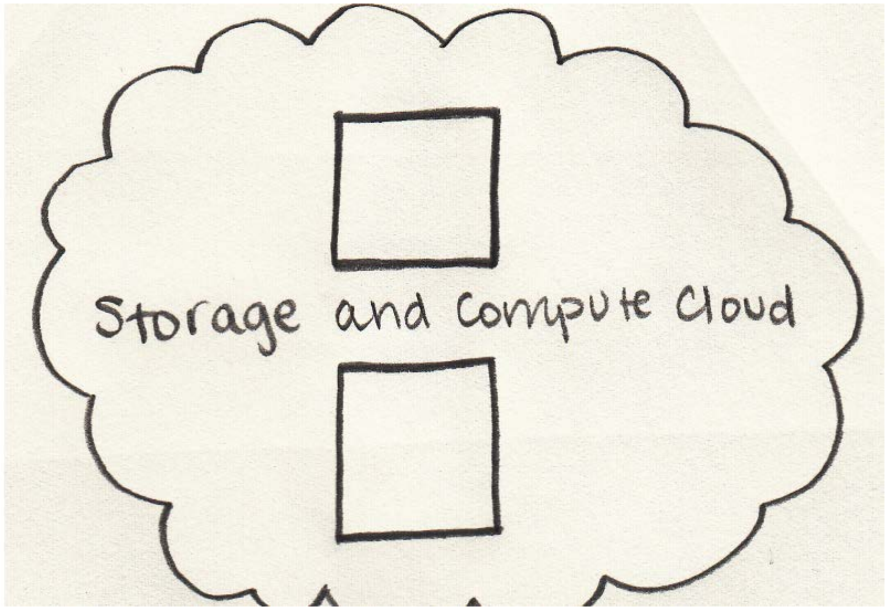
- ❑ The traditional data processing model:
  - o Have data stored in a 'storage cluster‘.
  - o Copy it over to a 'compute cluster' for processing.
  - o Write the results back to the ‘storage cluster’…does not work anymore!
- ❑ Amount of data to be copied is an unaffordable bottleneck!



Big Data Analytics

New processing model:

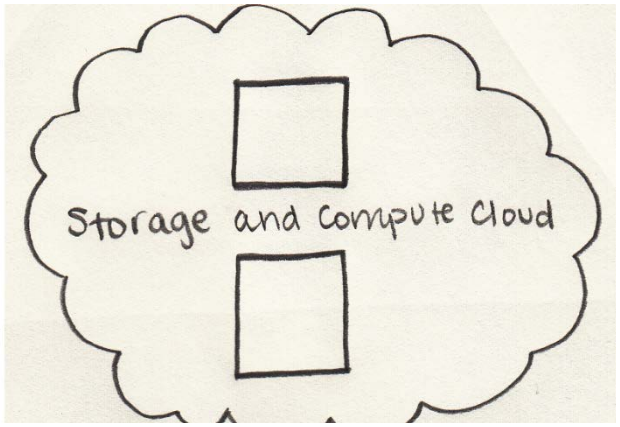
- ❑ If data cannot go to the processor…bring the processor to the data!



Big Data Analytics

Core Concept:

- Distribute the data as it is initially stored in the system.
    - Computation happens where the data is stored, wherever possible.
    - No data transfer over the network is required for initial processing.
    - Data is replicated multiple times on the system for increased availability and reliability



Big Data Analytics

Processing Model:

- ❑ But if the data is split among different nodes in the cluster, and each piece of data is processed on the node it is hosted, then…

<u>…our processing model has suddenly become a distributed one!</u>

- ❑ Let’s think again of our example of Zara and the Facebook database.

Big Data Analytics

- Data distributed on 21 DCs (shards).
- Facebook (in our example) uses MongoDB collection split among the 21 shards based on the shard key “country”.
- Metadata (conf. server) determines the shard hosting each document.

<u>Metadata:</u>

[alb --- cro) → S1

[cro --- lit) → S2

…

[tai --- zam) → S21

S1	**S2**	…	**S21**
user1	user7		user3
user32	user9		user4
…	…		…

Big Data Analytics

Scenarios:

1. Query by shard key:
   - Need only access the nodes (shards) containing the associated documents.
2. Query not based on shard key:
   - Need to access all nodes (shards).

Metadata:

[alb --- cro) → S1

[cro --- lit) → S2

…

[tai --- zam) → S21

S1	**S2**	…	**S21**
user1	user7		user3
user32	user9		user4
…	…		…

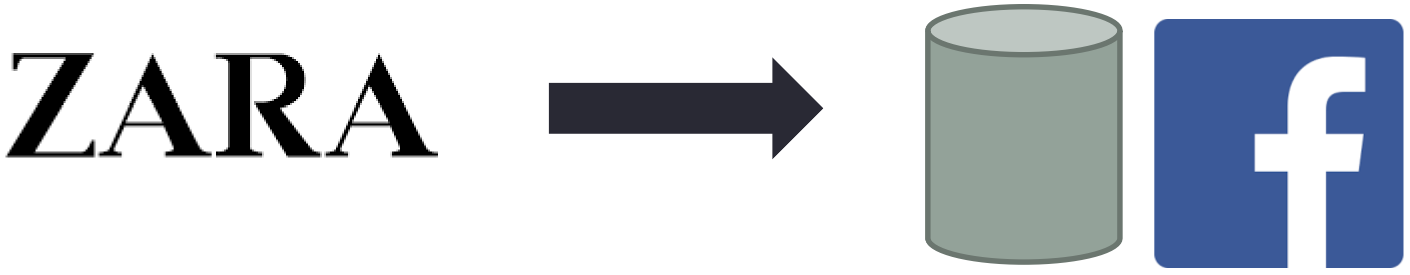
Big Data Analytics

But if we pay attention…

<u>**What do we have here?**</u>

- ❑ A <u>single pool</u> of documents with the 500 million users of Facebook.

- ❑ A company, Zara, that wants to do some data analytics on it to make good decisions for the company…
  - o What is the ratio of population per country interested in shopping?
  - o What are the 5 countries with the biggest ratio?

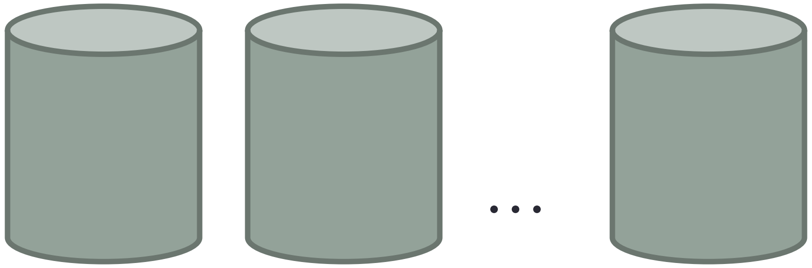


Big Data Analytics

But if we pay attention…

<u>What do we have here?</u>

- ❑ The pool is so big that it cannot be stored and processed into one single node (machine). Instead, data is distributed across *n* nodes.



Big Data Analytics

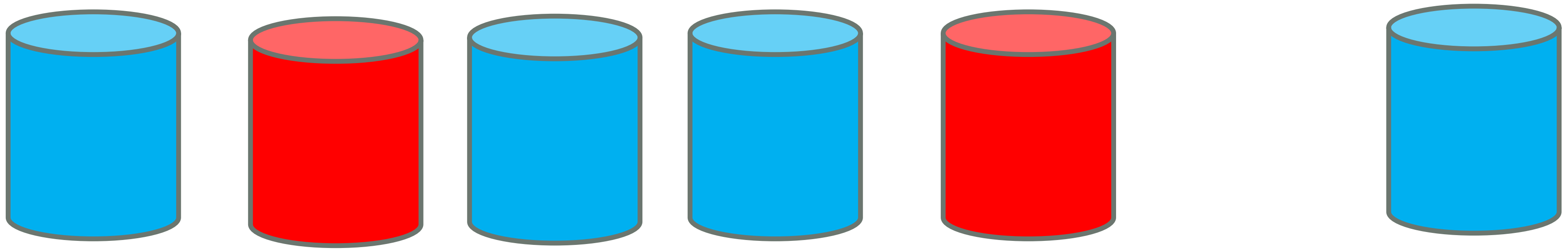
ZARA

- To query the collection, Zara has to:

1. Determine the <u>subset of nodes</u> containing the relevant info it is looking for. This includes determining the piece of info contained on each node.

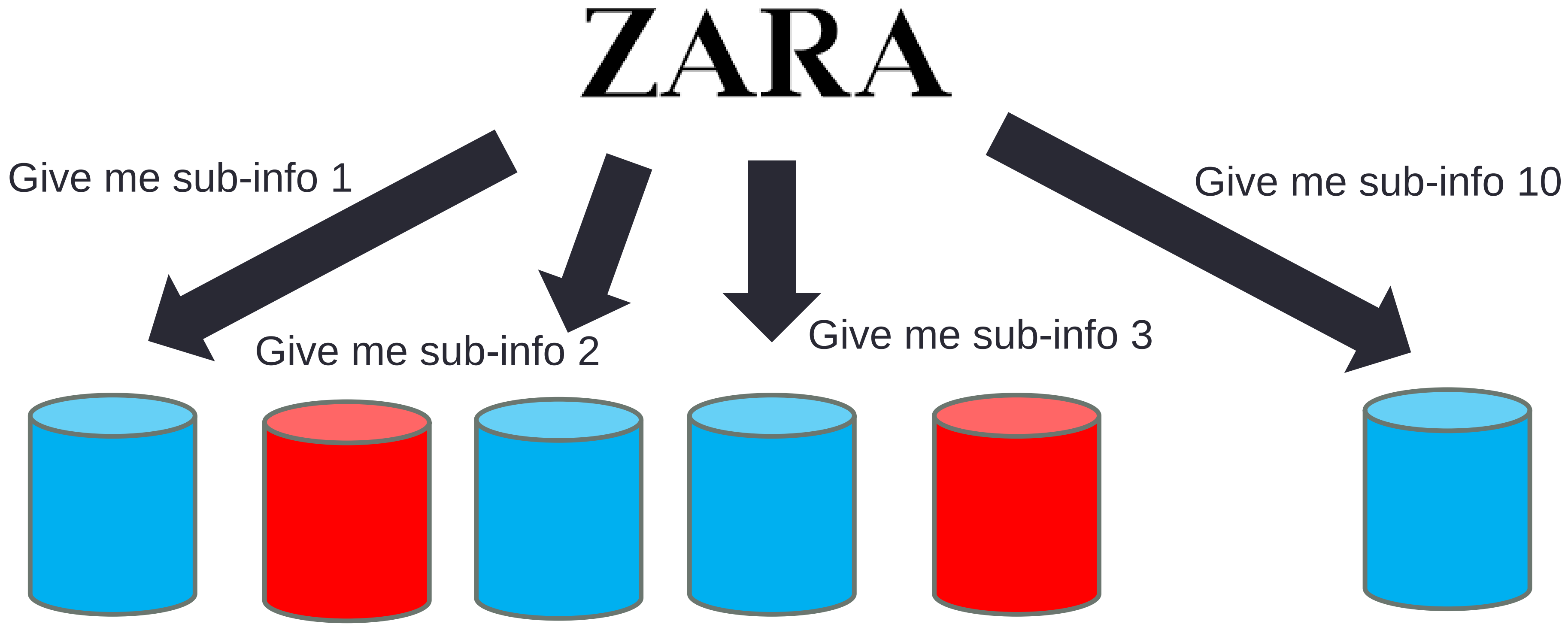
Roughly speaking, Zara says:

Ok, I need to access to this info. The info is split across10 nodes (out of the 21), as this particular subset of the info is in node 1, this other in node 3, …, and this one in node 21.



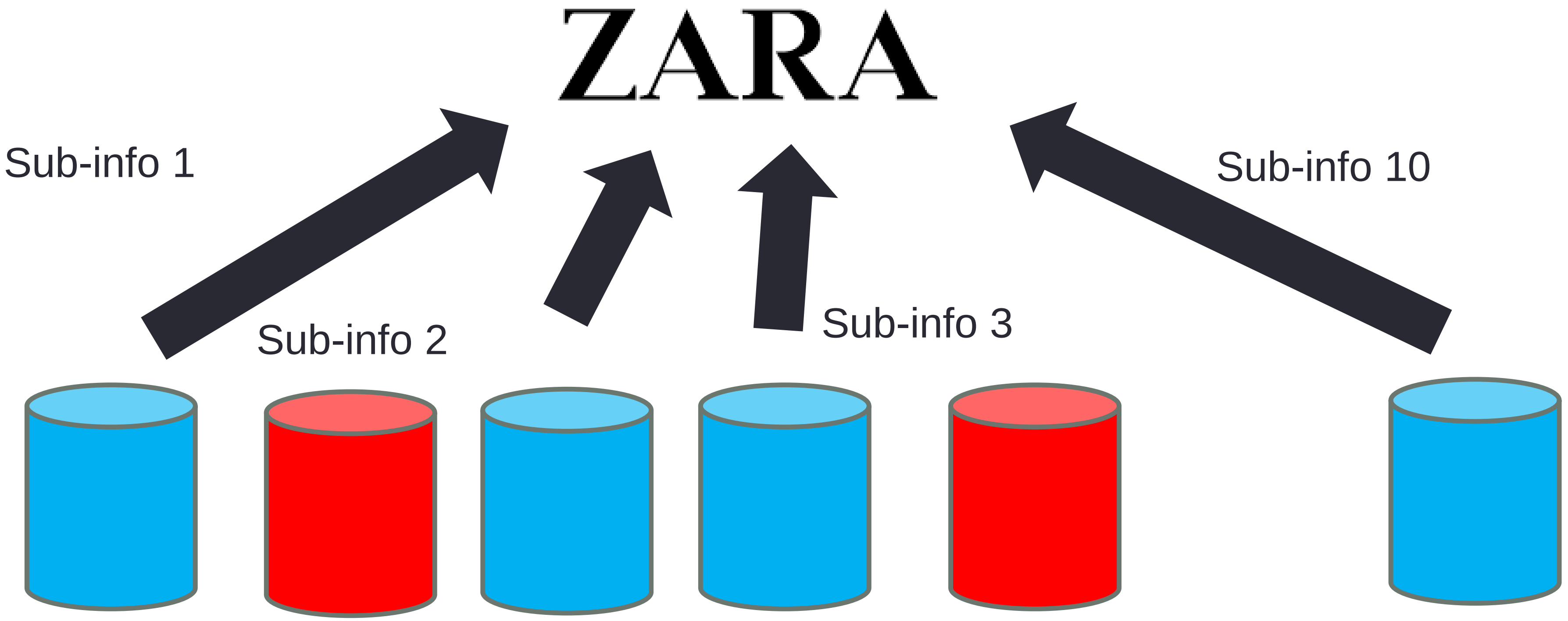
Big Data Analytics

- To query the collection, Zara has to:
  2. Query each node for its associated subset of info.



Big Data Analytics

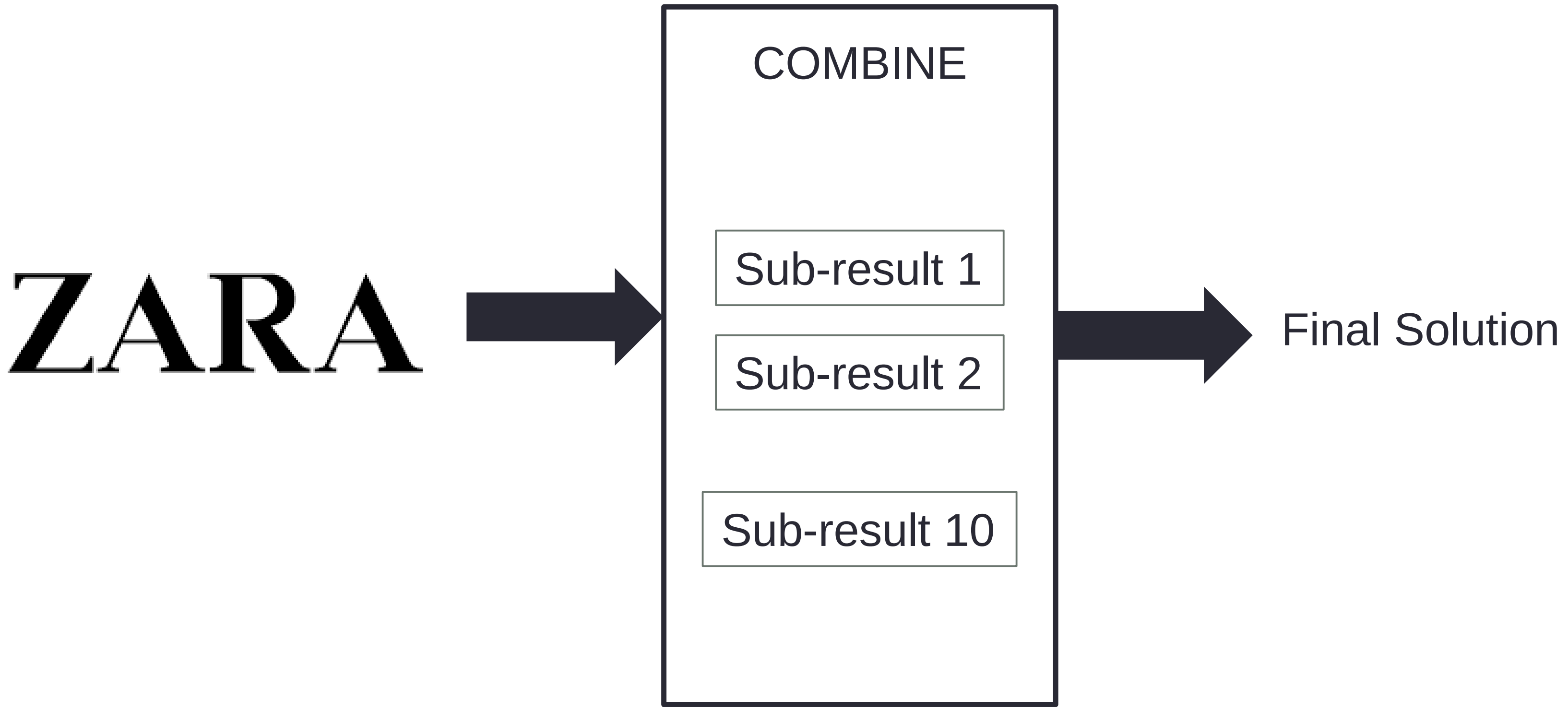
- To query the collection, Zara has to:
  3. Wait for the nodes to process the queries and return their sub-info (or, in other words, the sub-results of the info Zara is looking for).



Big Data Analytics

- To query the collection, Zara has to:

4. Finally combine the sub-results to achieve the total result it is looking for.



Big Data Analytics

- ❑ Let’s enumerate the stages of the process again:
  1. Divide the original problem into smaller sub-problems…
  2. …Solve the sub-problems…
  3. …Combine the sub-problems solutions into the original problem solution.

- ❑ In data analytics terminology, this process is known as **map-reduce**.
- ❑ But if we abstract enough, this process is nothing but an instantiation of the more general schema used by **Divide & Conquer** algorithms.

Outline

1. Big Data Analytics.
2. Divide & Conquer Algorithms Schema.
3. Recursion: A Possible Instantiation of the Schema.
4. Concurrency: A Possible Parallelisation of the Map.

Divide & Conquer Algorithms Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
        sol: Sol;
        If small(P) then
                sol = solve(P);
        else{
            <subP1, subP2, …, subPn> = divide(P);
            <solP1, solP2, …, solPn> =
                map(divide&conquer, <subP1, subP2, …, subPn>);
            sol = reduce(<solP1, solP2, …, solPn>)
        }
}
{ sol is the solution to the problem P }
```

Divide & Conquer Algorithms Schema

❑ Let’s reason about the different actors taking place here:

1. divide&conquer
2. small
3. solve
4. divide
5. map
6. reduce

Divide & Conquer Algorithms Schema

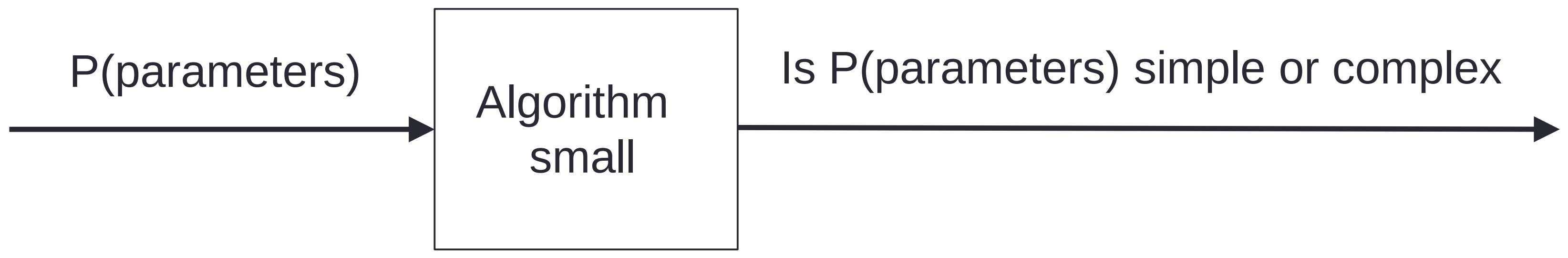
divide&conquer

- ❑ Here we are not interested in a single problem P_0, but in a set of problems $\{P_0, P_1, \ldots, P_k\}$ having a common structure.
- ❑ Thus, our idea is not to present a concrete algorithm A_0, but a concrete algorithm schema S, presenting the generic steps to be followed by any algorithm solving one of these problems $\{P_0, P_1, \ldots, P_k\}$.

Divide & Conquer Algorithms Schema

small

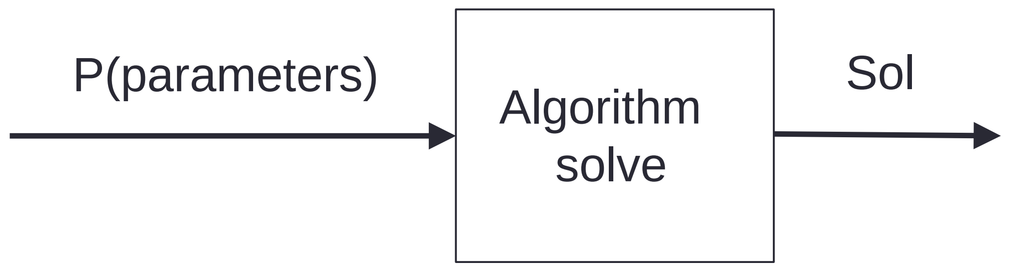
- small is a generic algorithm.
  - It receives as an input a problem P with whatever parameters P(parameters).
  - It produces as an output a Boolean result, stating whether the problem to be solved, P(parameters), is simple or complex to solve.



Divide & Conquer Algorithms Schema

solve

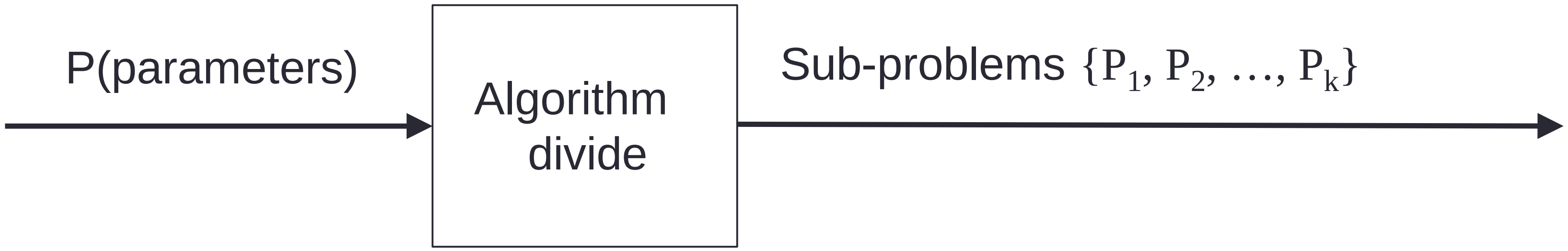
- solve is a generic algorithm.
  - It receives as an input a simple problem P with whatever parameters P(parameters).
  - It produces as an output a solution for the problem.



Divide & Conquer Algorithms Schema

divide

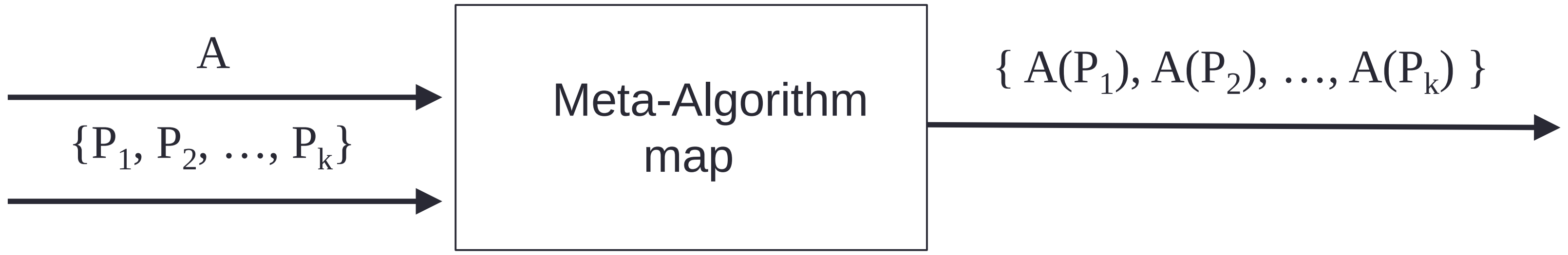
- divide is a generic algorithm.
  - It receives as an input a complex problem P with whatever parameters P(parameters).
  - It produces as an output a split of the problem P into multiple smaller sub-problems $\{P_1, P_2, \ldots, P_k\}$.



Divide & Conquer Algorithms Schema

map

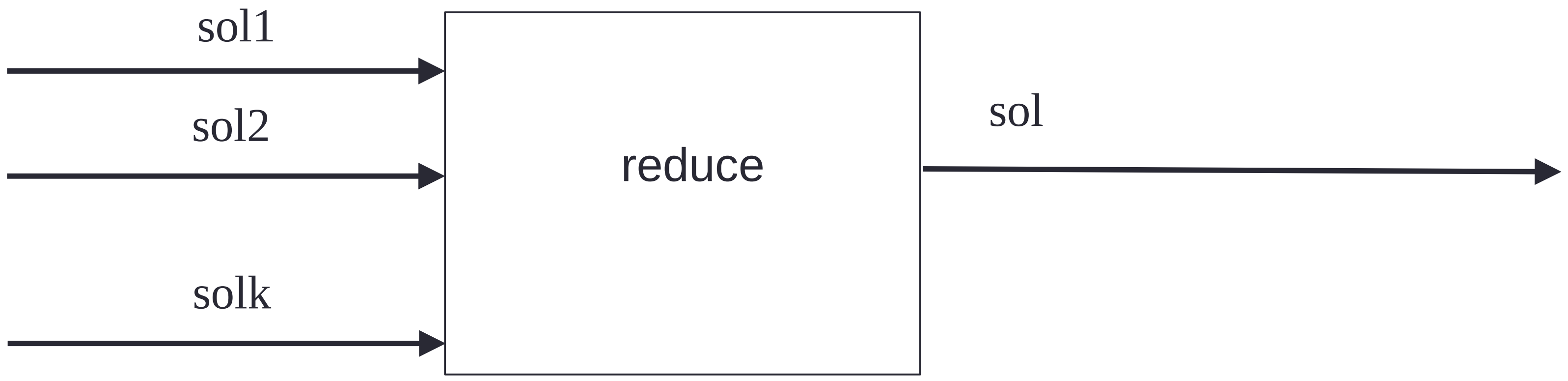
- map is a generic meta-algorithm.
  - It receives as an input a concrete algorithm A and a list of problems $\{P_1, P_2, \ldots, P_k\}$ each of them with its own concrete parameters.
  - It produces as an output the application of algorithm A to each of the sub-problems: $\{ A(P_1), A(P_2), \ldots, A(P_k) \}$


A
$\{P_1, P_2, \ldots, P_k\}$
Meta-Algorithm map
$\{ A(P_1), A(P_2), \ldots, A(P_k) \}$

Divide & Conquer Algorithms Schema

reduce

- reduce is a generic algorithm.
  - It receives as an input the solution to a set of problems $\{sol_1, sol_2, \ldots, sol_k\}$.
  - It produces as an output a combined single solution : sol



Outline

1. Big Data Analytics.
2. Divide & Conquer Algorithms Schema.
3. Recursion: A Possible Instantiation of the Schema.
4. Concurrency: A Possible Parallelisation of the Map.

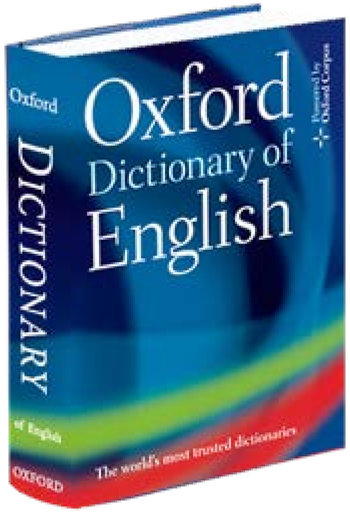
Recursion: A Possible Instantiation of the Schema

- ❑ Recursion: Programming technique in which a method calls itself.
- ❑ Recursion main idea: Problem P is solved by reducing it to a smaller problem P' of the same form.
  - o So we can say that recursion solves a problem by solving a smaller problem of the same type.

- ❑ Recursion is a surprisingly effective technique for solving some very complex programming problems.

Recursion: A Possible Instantiation of the Schema

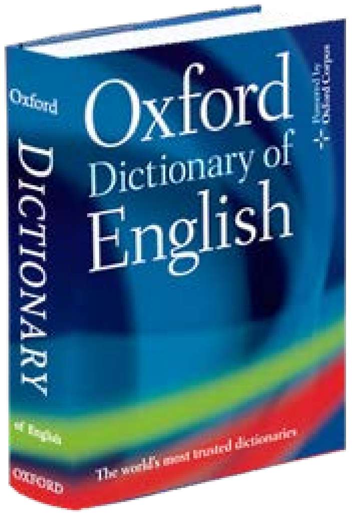
- Consider the problem of looking up a word in a dictionary.
  - Suppose you wanted to look up the word “zone”
  - Sequential search would start at the beginning of the dictionary and look at every word in order until you found the search term.


Oxford
DICTIONARY
Oxford Dictionary of English
Powered by Oxford Corpus
The world's most trusted dictionaries
OXFORD

Recursion: A Possible Instantiation of the Schema

- Alternative: Use binary search.
  - Open the dictionary at a middle point.
  - Glancing at the page, determine which "half" of the dictionary contains the desired word, and look for it on this half.

- What would the basic pseudocode for this look like?



Recursion: A Possible Instantiation of the Schema

```
1. if (the dictionary contains only one page)
    1.1. Scan the page for the word
2. else {
    2.1. Open the dictionary to a point near the middle
    2.2. Determine which half of the dictionary contains the word

   2.3. if (the word is in the first half of the dictionary)
        2.3.1. Search the first half of the dictionary for the word
   2.4. else
        2.4.1. Search the second half of the dictionary for the word
}
```

Recursion: A Possible Instantiation of the Schema

On the one hand…

- ❑ When you divide the dictionary in halves and pick the appropriate half, you know how to search for the word on it:
  - o <u>You can use exactly the same strategy that you employed to search the original dictionary</u>.

- ❑ Notice each time you do this the complexity of the problem is reduced (if the dictionary has x pages, new sub-problem just contains x / 2 pages).

- ❑ This case is called the <u>recursive case</u>.

Recursion: A Possible Instantiation of the Schema

One of the actions of the method is to call itself; that is, the method *search* calls the method *search*. This action is what makes the solution recursive.
The solution strategy is to split *theDictionary* in halves, determine which half contains *aWord*, and apply the same strategy to the appropriate half.

```
search(in Dictionary:Dictionary, in aWord:String) {
     if (theDictionary is one page in size)
        Scan the page for aWord
     else {
        Open theDictionary to a point near the middle
        Determine which half of theDictionary contains aWord

         if (aWord is in the first half of theDictionary)
           search(first half of theDictionary, aWord)
         else
           search(second half of theDictionary, aWord)
     }
 }
```

Recursion: A Possible Instantiation of the Schema

Each call to the method *search* made from within the method *search* passes a dictionary that is one-half the size of the previous dictionary.
That is, at each successive call to search (theDictionary, aWord), the size of *theDictionary* is cut in half

```
search(in Dictionary:Dictionary, in aWord: String) {
      if (theDictionary is one page in size)
          Scan the page for aWord
      else {
          Open theDictionary to a point near the middle
          Determine which half of theDictionary contains aWord

           if (aWord is in the first half of theDictionary)
             search(first half of theDictionary, aWord)
           else
            search(second half of theDictionary, aWord)
      }
  }
```

Recursion: A Possible Instantiation of the Schema

On the other hand…

- ❑ Note that there is a special case different from all others:
  - o After you have divided the dictionary so many times that you are left with only a single page, the halving ceases.
  - o At this point, the problem is sufficiently small that you can solve it directly by scanning the single page that remains for the word.

- ❑ This special case is called the <u>base case</u>.

Recursion: A Possible Instantiation of the Schema

There is one search problem that you handle differently from all of the others. When *theDictionary* contains only a single page, you use another technique: You scan the page directly.
Searching a one-page dictionary is the base case of the search problem.

```
search(in Dictionary:Dictionary, in aWord: String) {
       if (theDictionary is one page in size)
           Scan the page for aWord
       else {
           Open theDictionary to a point near the middle
           Determine which half of theDictionary contains aWord

            if (aWord is in the first half of theDictionary)
              search(first half of theDictionary, aWord)
            else
             search(second half of theDictionary, aWord)
       }
  }
```

Recursion: A Possible Instantiation of the Schema

Key Concept:

- ❑ There are four key questions you should ask yourself to solve the problem P using a recursive approach (i.e. a recursive algorithm).

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will it reach the base case?

Recursion: A Possible Instantiation of the Schema

- Let's suppose we have the following problem:

  Given a myList<Integer>, compute the sum of all its elements.

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
      ??
{ Post-condition: s is the sum of all elements of l }
```

- Examples:

sumList([3,2,9,7,1]) → 22   sumList([4,2]) → 6 sumList([ ]) → 0

- How can we implement `sumList` iteratively? And recursively?

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.   s = 0
        2.   for i in 0 … length(l)-1
                 s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ The variable s represents the invariant of the loop:

“s represents the sum of all the elements of l that we have considered so far”

Recursion: A Possible Instantiation of the Schema

- <u>Iterative approach:</u>

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.   s = 0
        2.   for i in 0 … length(l)-1
                 s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

- Given myList<Integer> l = [3, 2, 9, 7, 1].

Before start the loop s = 0 → So far we have considered no elements at all, so their sum should be 0.

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.  s = 0
        2.  for i in 0 … length(l)-1
                s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ Given myList<Integer> l = [3, 2, 9, 7, 1].

After the first iteration s = 3 → So far we have considered only the first element, so the total sum of the elements considered is 3.

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.  s = 0
        2.  for i in 0 … length(l)-1
                s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ Given myList<Integer> l = [3, 2, 9, 7, 1].

After the second iteration s = 5.

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.   s = 0
        2.   for i in 0 … length(l)-1
                 s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ Given myList<Integer> l = [3, 2, 9, 7, 1].

After the third iteration s = 14.

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.   s = 0
        2.   for i in 0 … length(l)-1
                  s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ Given myList<Integer> l = [3, 2, 9, 7, 1].

After the fourth iteration s = 21.

Recursion: A Possible Instantiation of the Schema

❑ Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.   s = 0
        2.   for i in 0 … length(l)-1
                 s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

❑ Given myList<Integer> l = [3, 2, 9, 7, 1].

After the fifth iteration s = 22.

Recursion: A Possible Instantiation of the Schema

- Iterative approach:

```
{ Pre-condition: }
Algorithm sumList(l : myList<Integer>) returns s: int
        1.  s = 0
        2.  for i in length(l)-1 … 0
                s = s + l[i]
{ Post-condition: s is the sum of all elements of l }
```

- Interestingly, the problem can also be solved by traversing the elements of l in decreasing order.
- Given myList<Integer> l = [3, 2, 9, 7, 1] it still returns s = 22.

Recursion: A Possible Instantiation of the Schema

- Recursive approach:

{ Pre-condition: }

Algorithm sumList(l : myList<Integer>) returns s: int

??

{ Post-condition: s is the sum of all elements of l }

- Let’s start by reasoning on our 4 key questions!

Recursion: A Possible Instantiation of the Schema

❑ Recursive approach:

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?

❑ Given a list = [elem1, elem2, elem3, elem4, elem5]

Let's consider it as: [elem1, elem2, elem3, elem4] + [elem5]

❑ Sum up all the elements of the list [elem1, elem2, elem3, elem4] is a problem of the same datatype as sum up all the elements of the list [elem1, elem2, elem3, elem4, elem5].

❑ But, it is a smaller sub-problem, as it contains less elements.

Recursion: A Possible Instantiation of the Schema

- Recursive approach:

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?

- Given a list = [elem1, elem2, elem3, elem4, elem5]

  Let's consider it as: [elem1, elem2, elem3, elem4] + [elem5]
- Thus, if "*somebody does by me*" the job of summing up [elem1, elem2, elem3, elem4] (with, let's say, solution y)…
  …then, I can solve the original problem of summing up [elem1, elem2, elem3, elem4, elem5] with solution y + elem5
- The "somebody does by me" is the recursive call to sumList.

Recursion: A Possible Instantiation of the Schema

- Recursive approach:

2. How does each recursive call diminish the size of the problem?

- By reducing the size of the list in one element.

3. What instance of the problem can serve as the base case?

- The one in which the myList received as input contains 0 elements.
- In this case, we know (without having anybody else to do some job for me) that the result is 0.

4. As the problem size diminishes, will it reach the base case?

- Yes, by reducing the size of myList in one element with each recursive call, we will eventually reach a myList with no elements.

Recursion: A Possible Instantiation of the Schema

❑ Recursive approach:

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0    Base Case
            1.1  s = 0
        2. else    Recursive Case
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. l.append(value)
            2.5. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: A Possible Instantiation of the Schema

- ❑ Recursive approach:
- ❑ Let’s trace the execution for solving: sumList([3, 2, 9])

- ❑ Person 1 is asked to do sumList([3, 2, 9]).

  She/he takes note of the last element → value = 9.

  Asks person 2 to do sumList([3, 2])

  Waits for her/his result (e.g. y) and returns y + 9 as final result.

Recursion: A Possible Instantiation of the Schema

- Recursive approach:

- Let’s trace the execution for solving: sumList([3, 2, 9])

- Person 2 is asked to do sumList([3, 2]).

  She/he takes note of last element → value = 2.

  Asks person 3 to do the sumList([3])

  Waits for her/his result (e.g., z) and returns z + 2 as final result.

Recursion: A Possible Instantiation of the Schema

- ❑ Recursive approach:

- ❑ Let’s trace the execution for solving: sumList([3, 2, 9])

- ❑ Person 3 is asked to do sumList([3]).

  She/he takes note of last element → value = 3.

  Asks person 4 to do the sumList([ ])

  Waits for her/his result (e.g. t) and returns t + 3 as final result.

Recursion: A Possible Instantiation of the Schema

❑ <u>Recursive approach:</u>

❑ Let’s trace the execution for solving: sumList([3, 2, 9])

❑ Person 4 is asked to do sumList([ ]).

She/he knows the solution of the problem: 0.
To come up with this solution, she/he does not need to ask somebody else to do part of the job on her/his behalf.
So, she/he returns 0 as a final result without the help of anybody else.

Recursion: A Possible Instantiation of the Schema

❑ Recursive approach:

❑ Let's trace the execution for solving: sumList([3, 2, 9])

❑ Person3 receives the result from person 4 → t = 0

Thus, she/he returns 0 + 3 = 3 as final result.

❑ Person2 receives the result from person 3 → z = 3

Thus, she/he returns 3 + 2 = 5 as final result.

❑ Person1 receives the result from person 2 → y = 5

Thus, she/he returns 5 + 9 = 14 as final result.

❑ The control comes back to the original person triggering the problem in first place (e.g., main method) with the result of 14.

Recursion: A Possible Instantiation of the Schema

❑ The file **1.recursion.py** (included in this week's folder in Blackboard) presents the recursive implementation of the function sumList.

Final Conclusion:

Let's see how our concrete algorithm **sumList** instantiates the **divide&conquer** algorithm schema.

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
          sol: Sol;
          If small(P) then
                    sol = solve(P);
          else{
             <subP1, subP2, …, subPn> = divide(P);
             <solP1, solP2, …, solPn> =
                     map(divide&conquer, <subP1, subP2, …, subPn>);
             sol = reduce(<solP1, solP2, …, solPn>)
          }
}
{ Post-condition List }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0
            1.1  s = 0
        2. else
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
        sol: Sol;
        If small(P) then
                sol = solve(P);
        else{
           <subP1, subP2, …, subPn> = divide(P);
          <solP1, solP2, …, solPn> =
                  map(divide&conquer, <subP1, subP2, …, subPn>);
          sol = reduce(<solP1, solP2, …, solPn>)
        }
}
{ Post-condition List }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0
            1.1  s = 0
        2. else
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
        sol: Sol;
        If small(P) then
                sol = solve(P);
        else{
           <subP1, subP2, …, subPn> = divide(P);
          <solP1, solP2, …, solPn> =
                map(divide&conquer, <subP1, subP2, …, subPn>);
          sol = reduce(<solP1, solP2, …, solPn>)
        }
}
{ Post-condition List }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0
            1.1  s = 0
        2. else
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
        sol: Sol;
        If small(P) then
                sol = solve(P);
        else{
           <subP1, subP2, …, subPn> = divide(P);
          <solP1, solP2, …, solPn> =
                 map(divide&conquer, <subP1, subP2, …, subPn>);
          sol = reduce(<solP1, solP2, …, solPn>)
        }
}
{ Post-condition List }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0
            1.1  s = 0
        2. else
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: P is a problem with whatever parameters }
Algorithm schema divide&conquer(P: Problem) returns sol: Sol {
        sol: Sol;
        If small(P) then
                sol = solve(P);
        else{
            <subP1, subP2, …, subPn> = divide(P);
            <solP1, solP2, …, solPn> =
                map(divide&conquer, <subP1, subP2, …, subPn>);
            sol = reduce(<solP1, solP2, …, solPn>)
        }
}
{ Post-condition List }
```

Recursion: A Possible Instantiation of the Schema

```
{ Pre-condition: }
Algorithm sumList(l : list<Integer>) returns s: int
        1. if length(l) == 0
            1.1  s = 0
        2. else
            2.1. value = l[ length(l) - 1]
            2.2. remove(l[ length(l) - 1])
            2.3. y = sumList(l)
            2.4. s = y + value
{ Post-condition: s is the sum of all elements of l }
```

Recursion: Another Example

- The Fibonacci sequence is a series of numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Many examples of the pattern followed by the sequence can be found in nature: https://www.youtube.com/watch?v=nt2OlMAJj6o

- Whereas the first two numbers of the series are fixed to 0 and 1, each new element (the n-est number of the series) is computed as the sum of the previous two elements of the sequence (say n-1 and n-2 numbers of the series). This leads to a neat recursive definition:

$$\text{fib}(n) = \begin{cases} n \text{ if } (n == 0) \text{ or } (n == 1) \\ \text{fib}(n-1) + \text{fib}(n-2) \text{ if } n > 1 \end{cases}$$

Recursion: Another Example

- Recursive approach:

```
{ Pre-condition: n ≥ 0}
Algorithm fib(n : int) returns s: int
        1. if ((n == 0) || (n == 1)
            1.1. s = n;
        2. else
            2.1. s = fib(n-1) + fib(n-2);
{ Post-condition: s is the n-est element of the Fibonacci sequence }
```

Recursion: Another Example

❑ Recursive approach:

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?

- ❑ Given the problem of obtaining the n-est term of the sequence… …We are reducing it to the problems of obtaining the n-1 and n-2 terms!
- ❑ Obtaining the terms (n-1) and (n-2) of the sequence are problems of the same datatype as the original one (obtaining the term n)… …but these problems are smaller in the sense that the terms (n-1) and (n-2) arise earlier in the sequence than the term n, and thus obtaining them requires less computation.

Recursion: Another Example

❑ Recursive approach:

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?

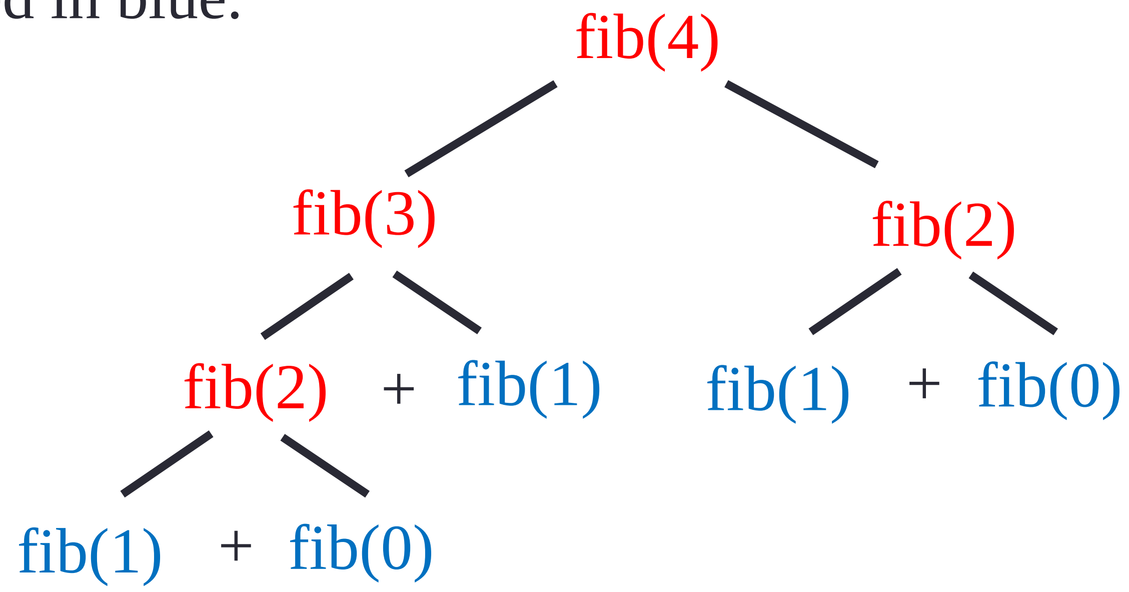
- ❑ Obviously, the delegation to compute both the terms (n-1) and (n-2) are the recursive calls.
- ❑ So, as we can see, each call to our problem leads to 2 recursive calls (rather than the single recursive call of the sumList example).

Recursion: Another Example

- Recursive approach:

1. How to define the problem P in terms of a smaller sub-problem P' of the same type?

- Example of the trace to compute fib(4): Whereas fib calls leading to the recursive case are marked in red, fib calls leading to the base case are marked in blue.



Recursion: Another Example

- Recursive approach:

2. How does each recursive call diminish the size of the problem?

- By reducing in one and two units the index of the element on the series we are looking at.

3. What instance of the problem can serve as the base case?

- The one in which we are looking at the first or second element of the series.
- In this case, we know that these terms are 0 and 1, respectively.

4. As the problem size diminishes, will it reach the base case?

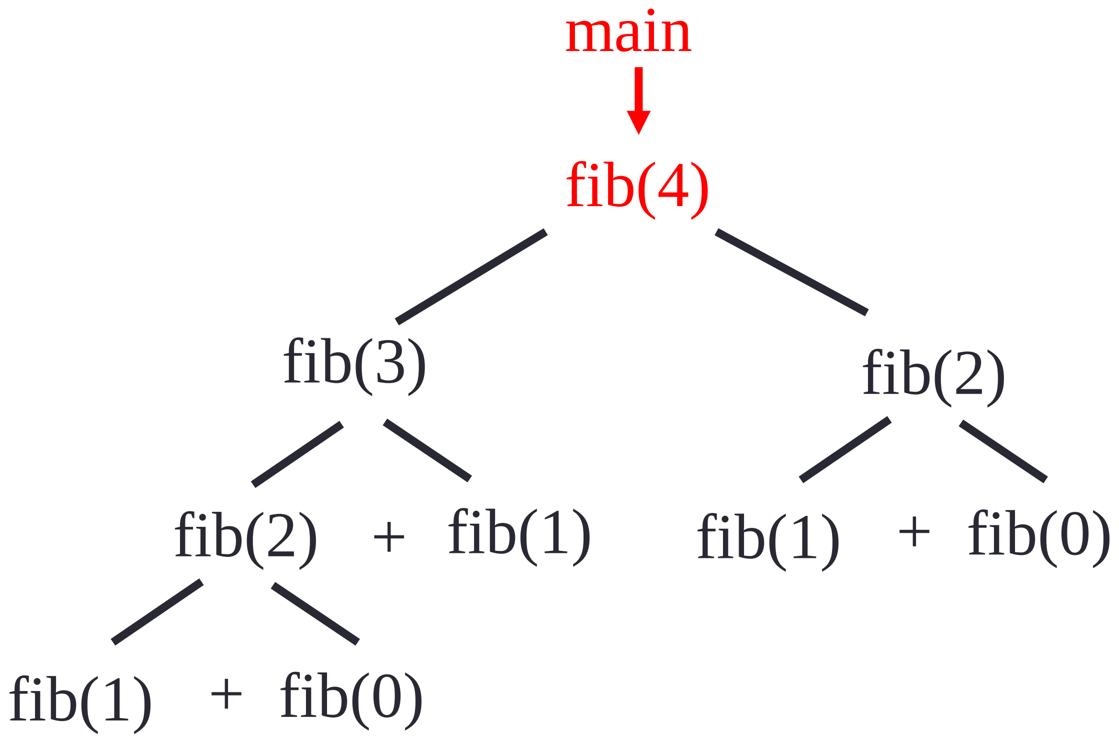
- Yes, by reducing in one and two units the index of the element, eventually we will be asked for the first or second elements of the series.

Recursion: Another Example

- Recursive approach:

- Let’s see the trace of fib(4) in more detail:

1. We call to fib(4) from the main.

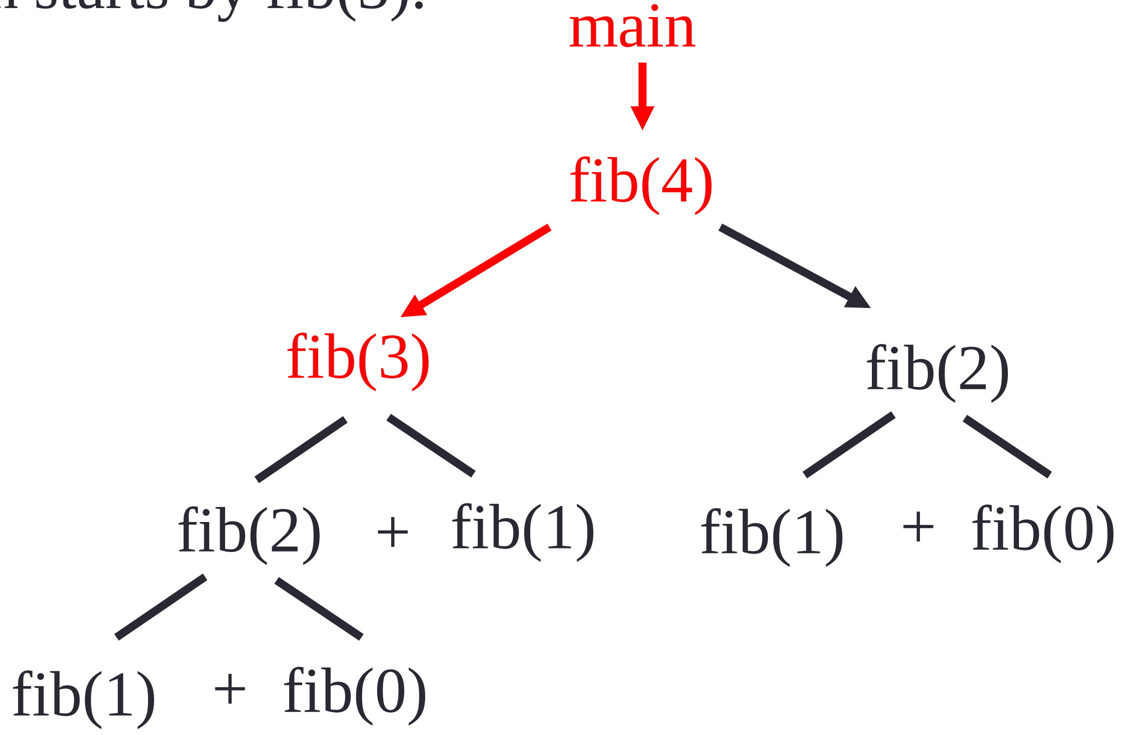


Recursion: Another Example

❑ Recursive approach:

❑ Let’s see the trace of fib(4) in more detail:

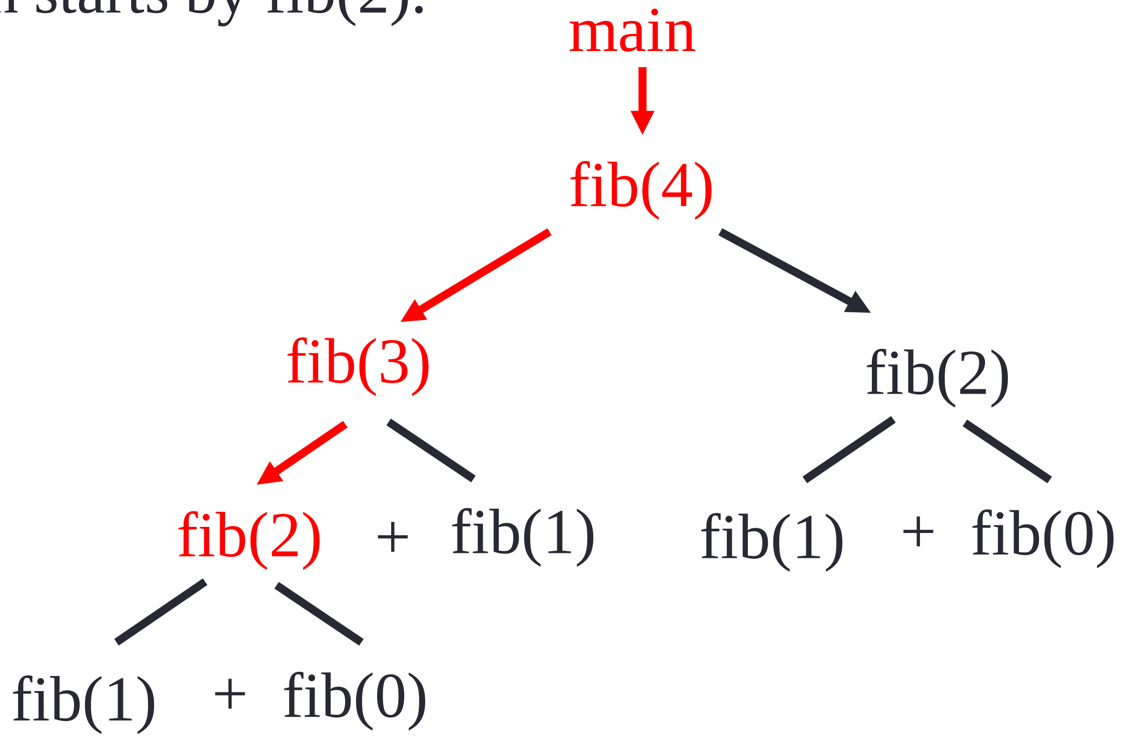
2. fib(4) makes two recursive calls to fib(3) and fib(2). The computation starts by fib(3).



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

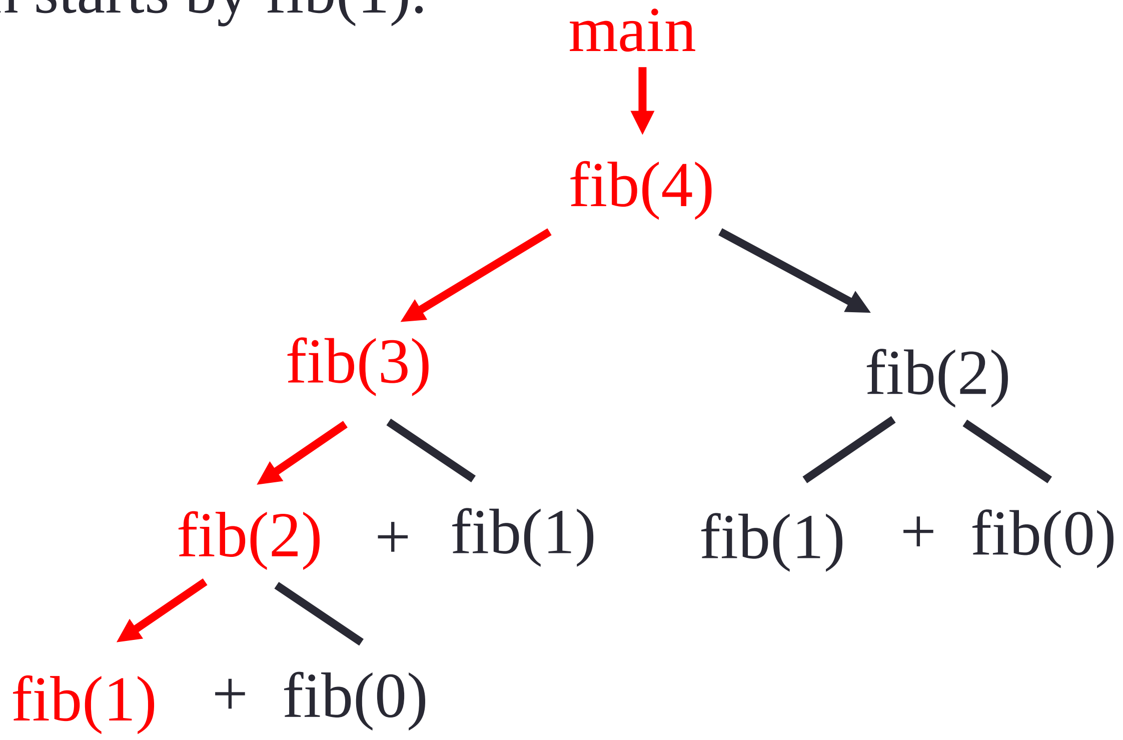
3. fib(3) makes two recursive calls to fib(2) and fib(1). The computation starts by fib(2).



Recursion: Another Example

- Recursive approach:
- Let's see the trace of fib(4) in more detail:

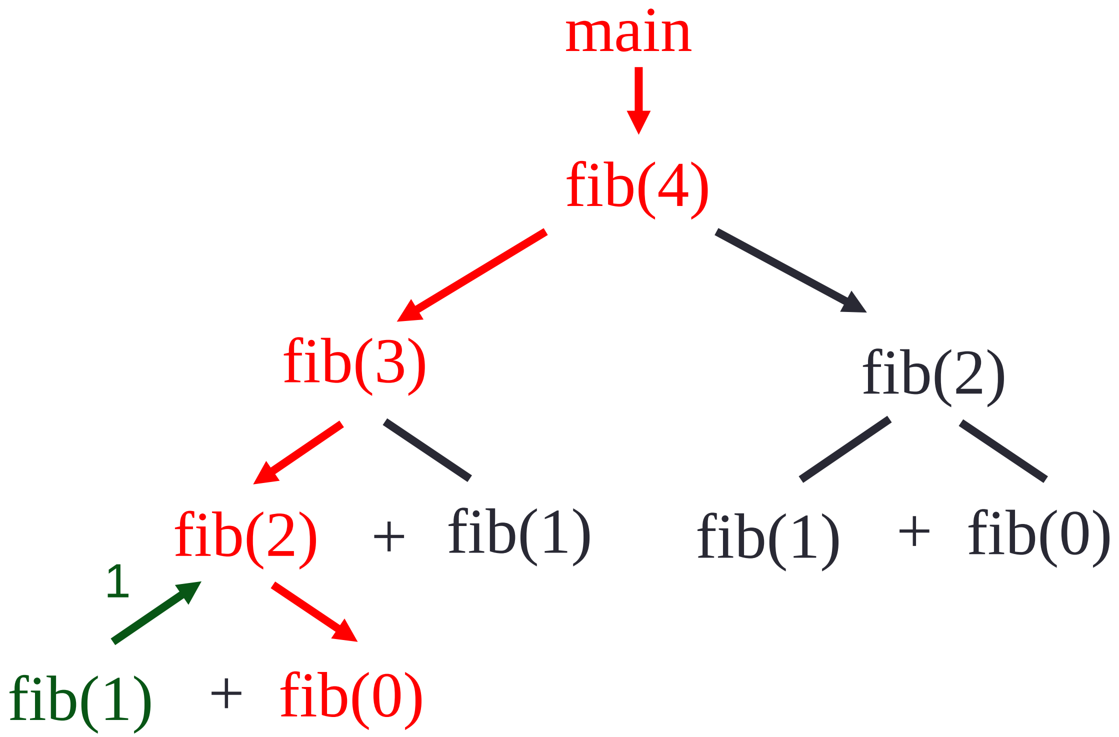
4. fib(2) makes two recursive calls to fib(1) and fib(0). The computation starts by fib(1).



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

5. fib(1) returns 1. The computation then continues with fib(0).

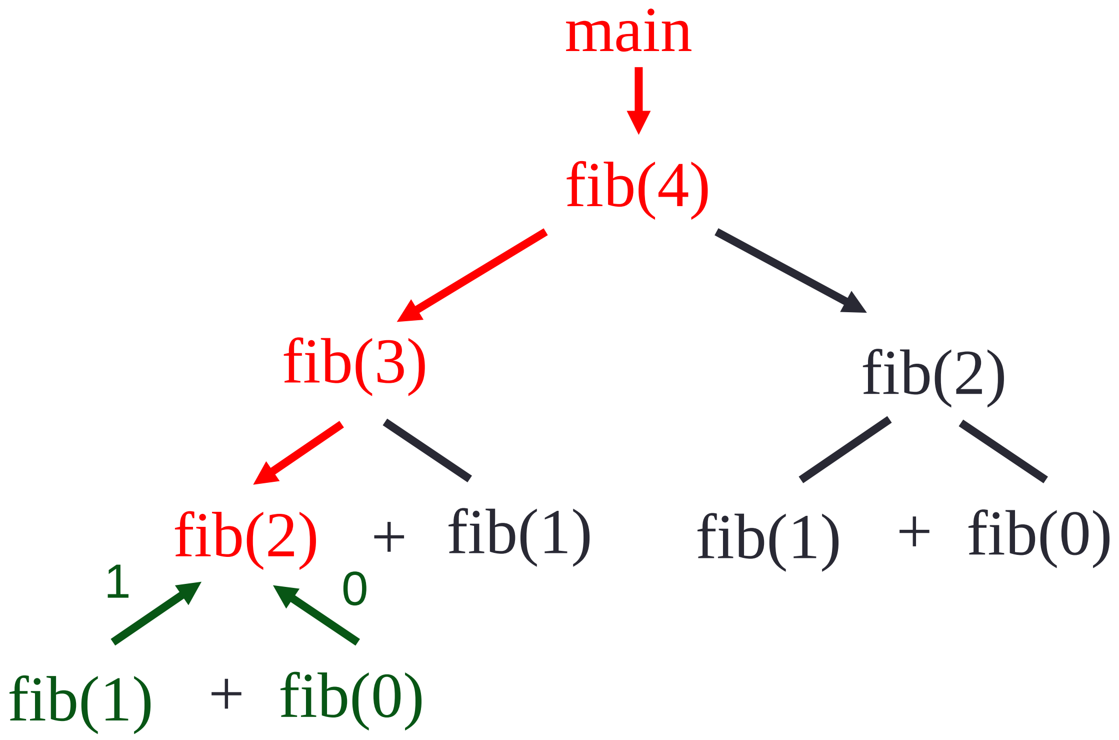


Recursion: Another Example

❑ <u>Recursive approach:</u>

❑ Let’s see the trace of fib(4) in more detail:

6. fib(0) returns 0.

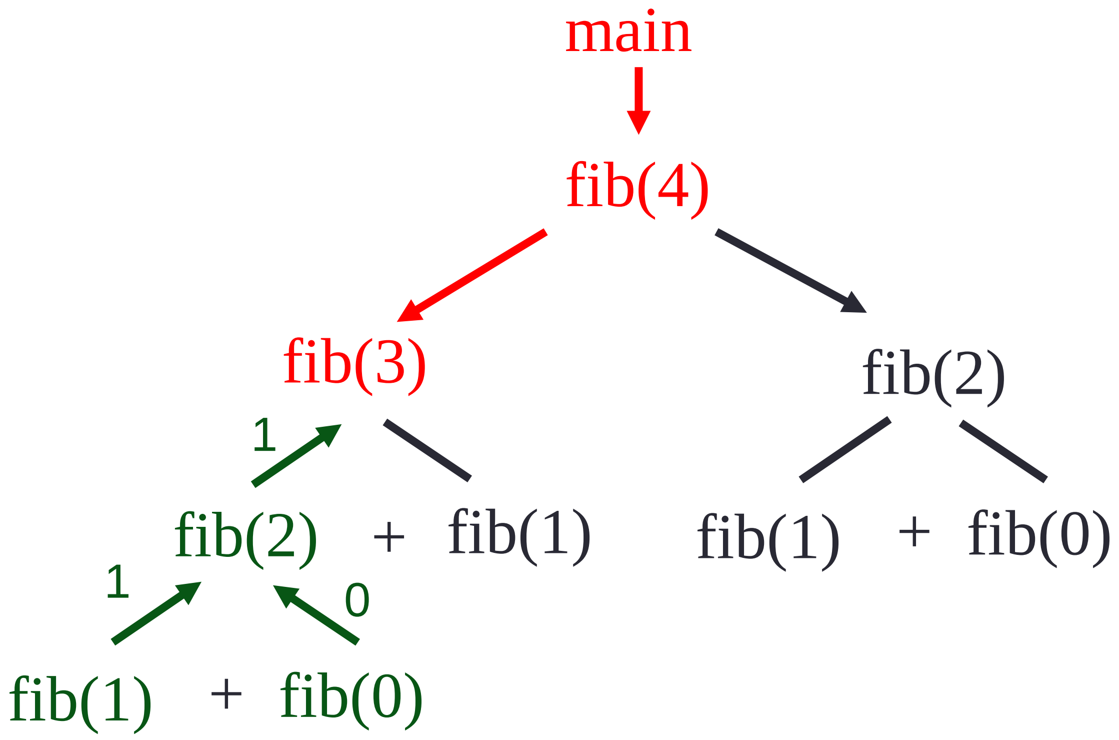


Recursion: Another Example

❑ Recursive approach:

❑ Let’s see the trace of fib(4) in more detail:

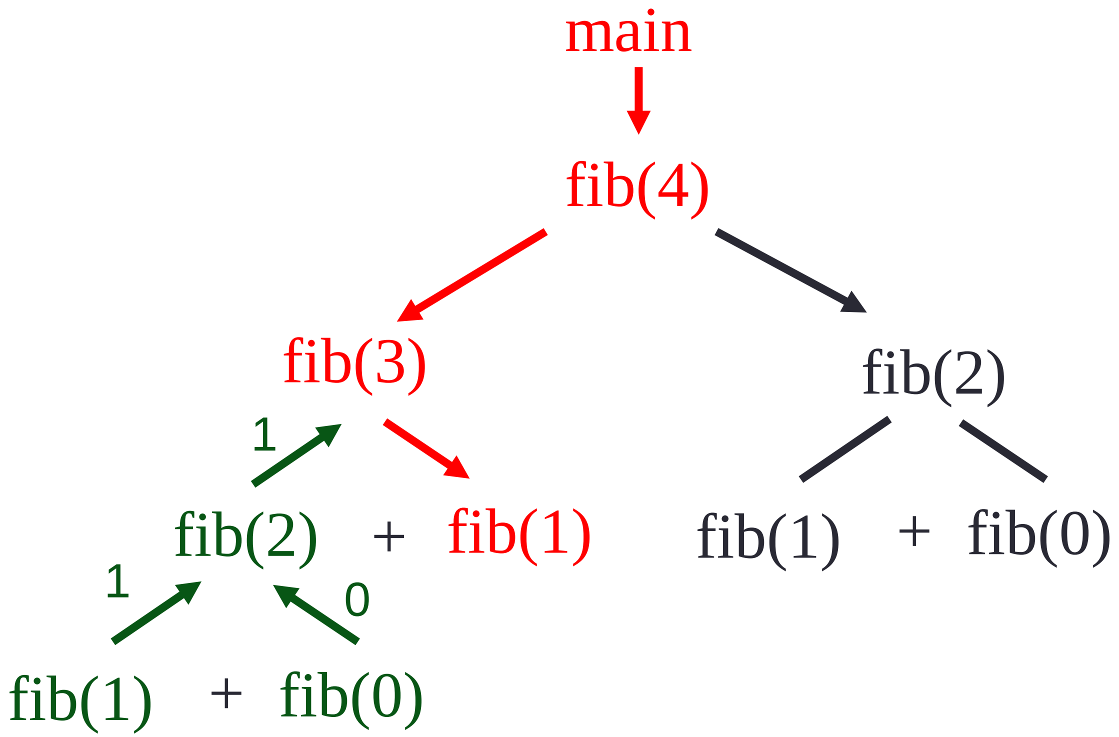
7. fib(2) can compute now its result as 1 + 0 = 1.



Recursion: Another Example

- Recursive approach:
- Let's see the trace of fib(4) in more detail:

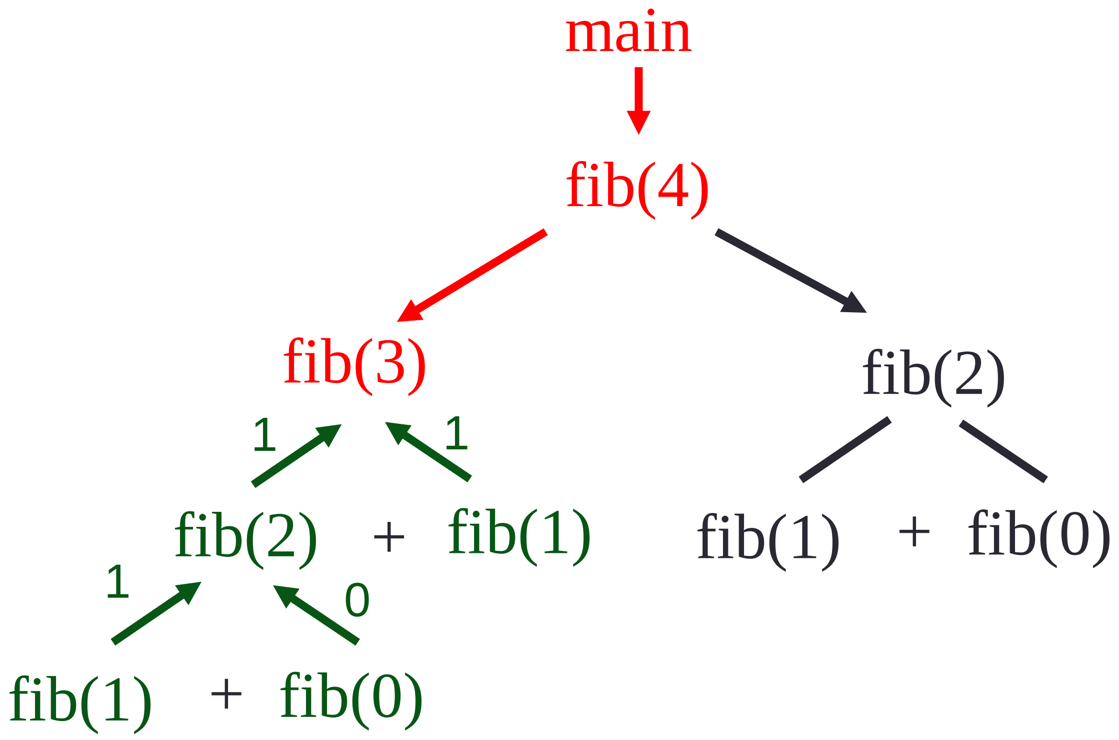
8. fib(3) continues now by computing fib(1).



Recursion: Another Example

- <u>Recursive approach:</u>
- Let’s see the trace of fib(4) in more detail:

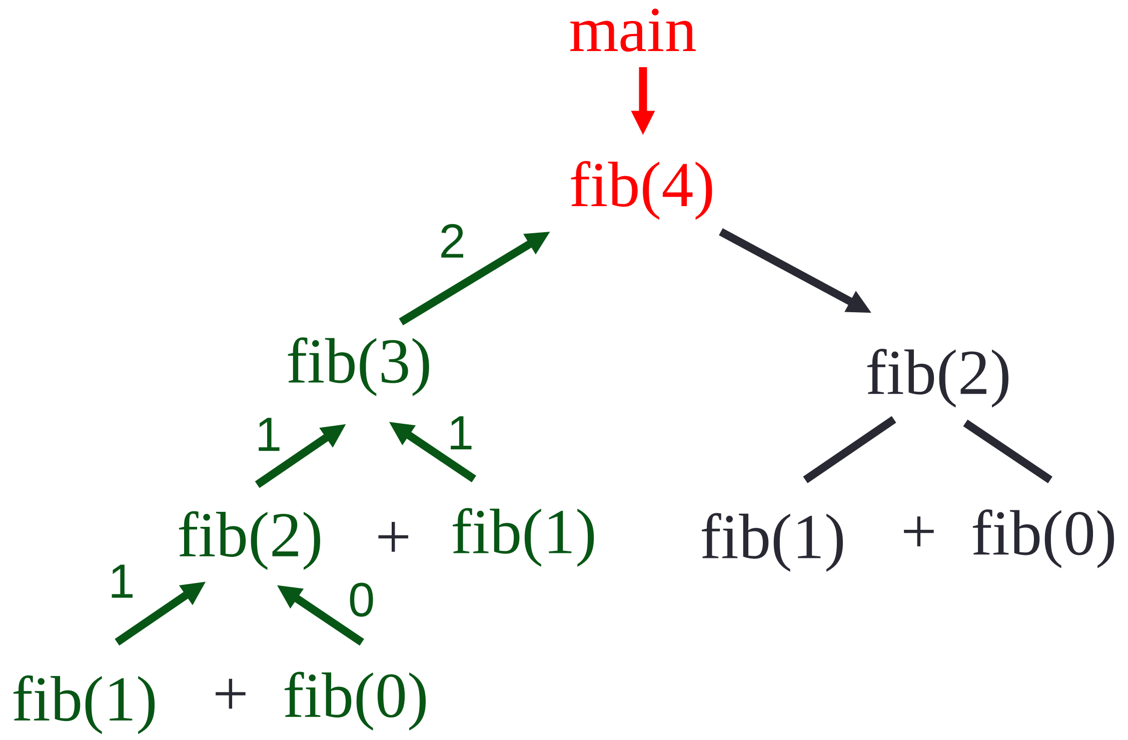
   9. fib(1) returns 1.



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

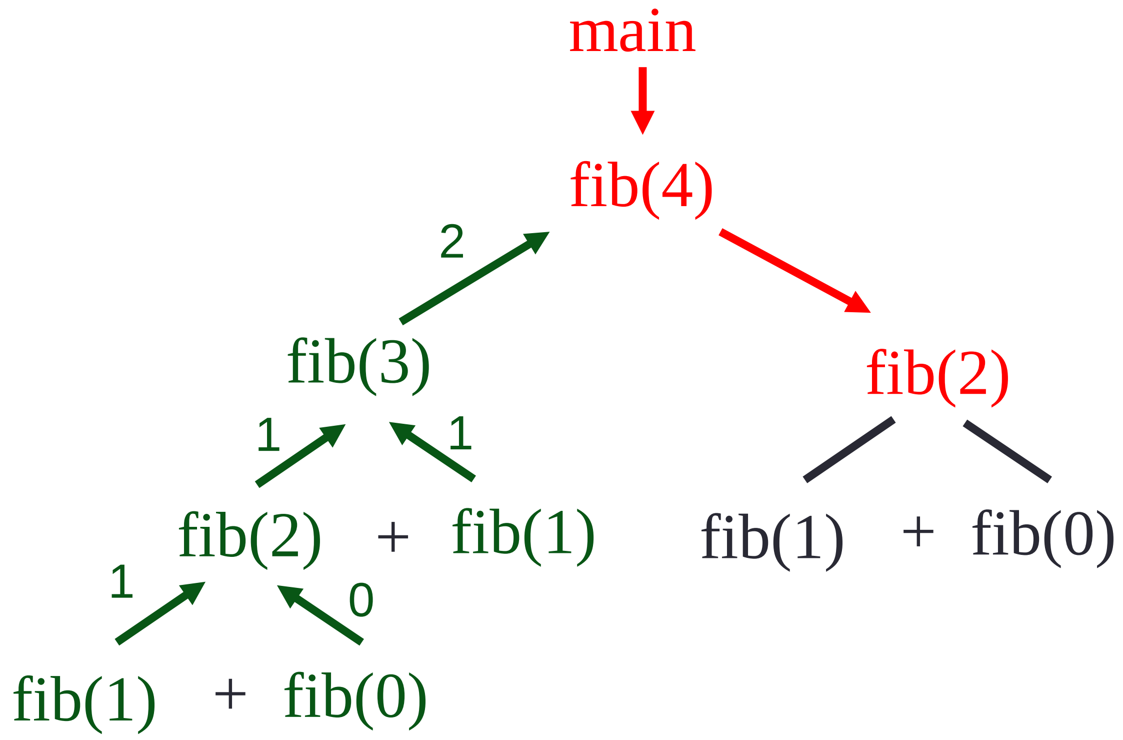
10. fib(3) can compute now its result as 1 + 1 = 2.



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

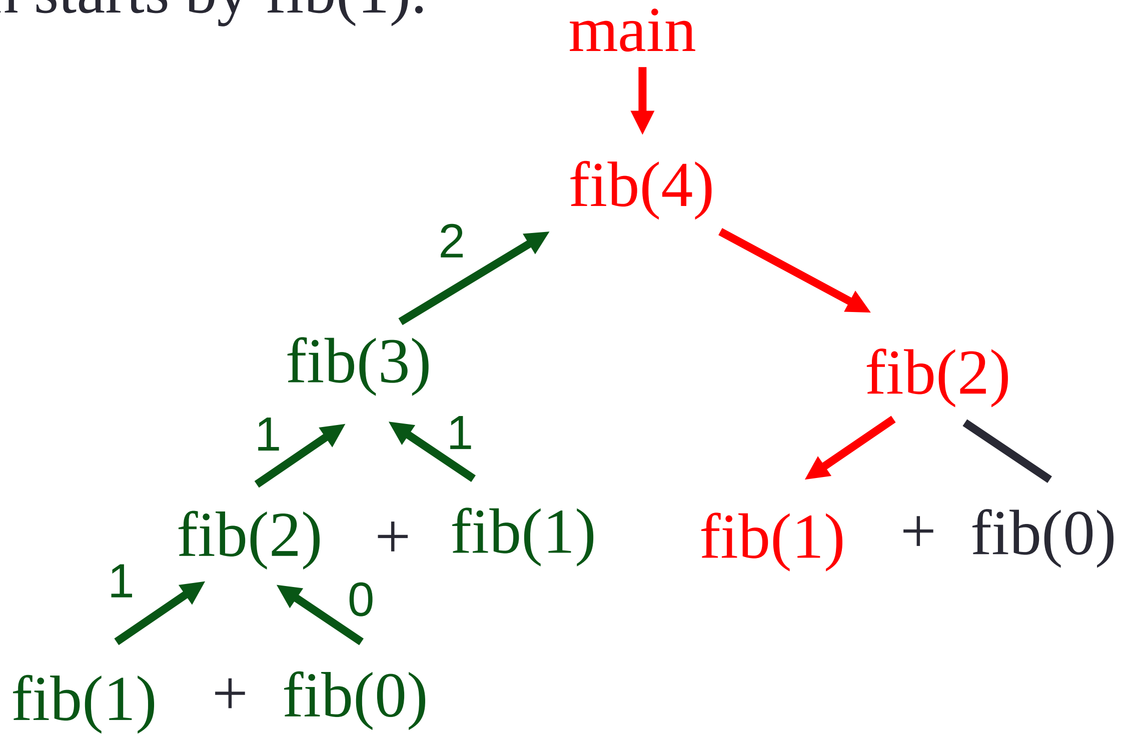
11. fib(4) continues now by computing fib(2).



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

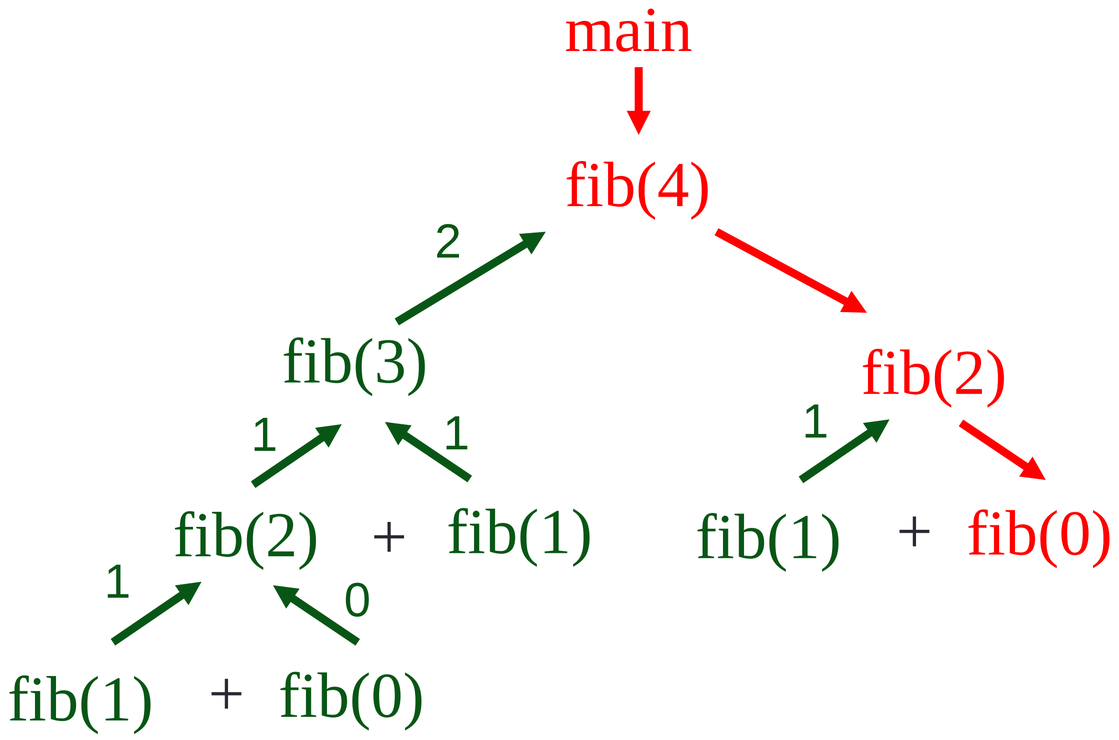
12. fib(2) makes two recursive calls to fib(1) and fib(0). The computation starts by fib(1).



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

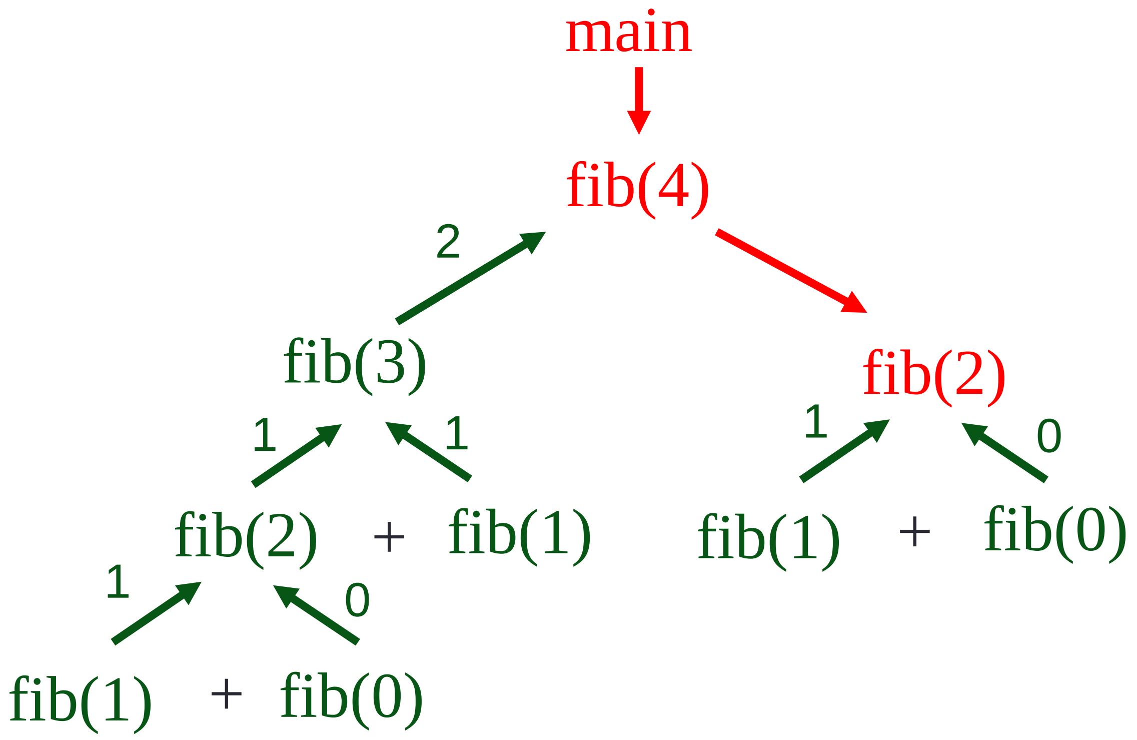
13. fib(1) returns 1. The computation then continues with fib(0).



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

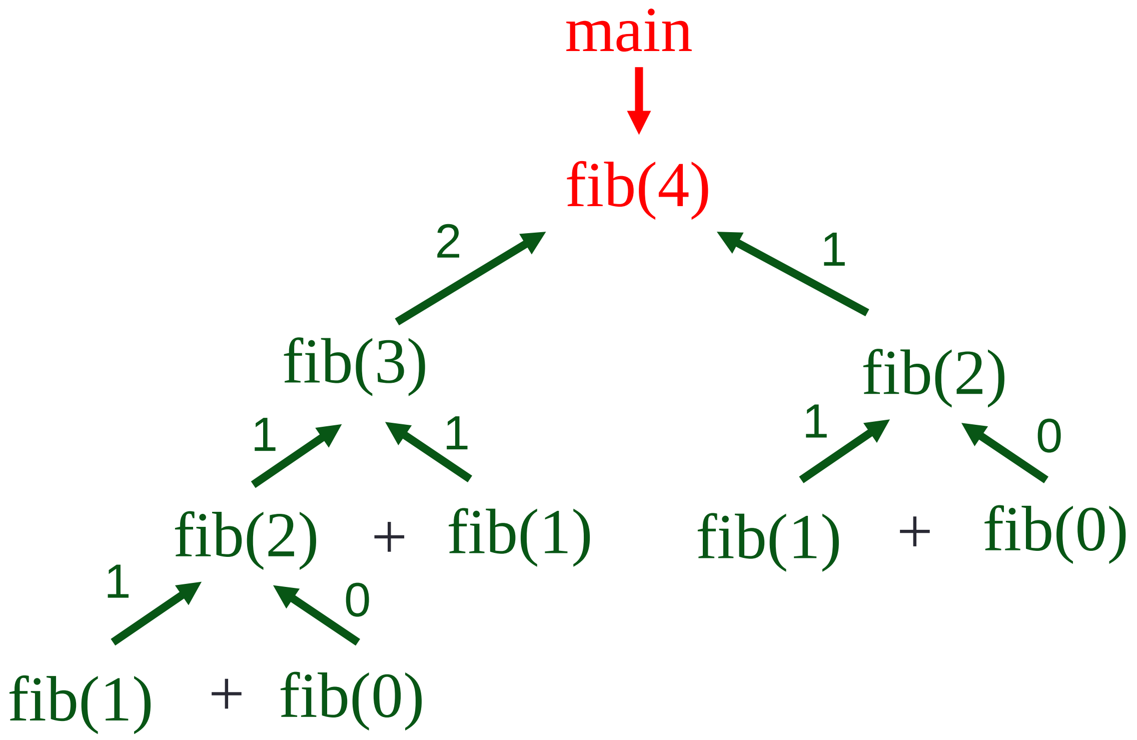
14. fib(0) returns 0.



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

15. fib(2) can compute now its result as 1 + 0 = 1.

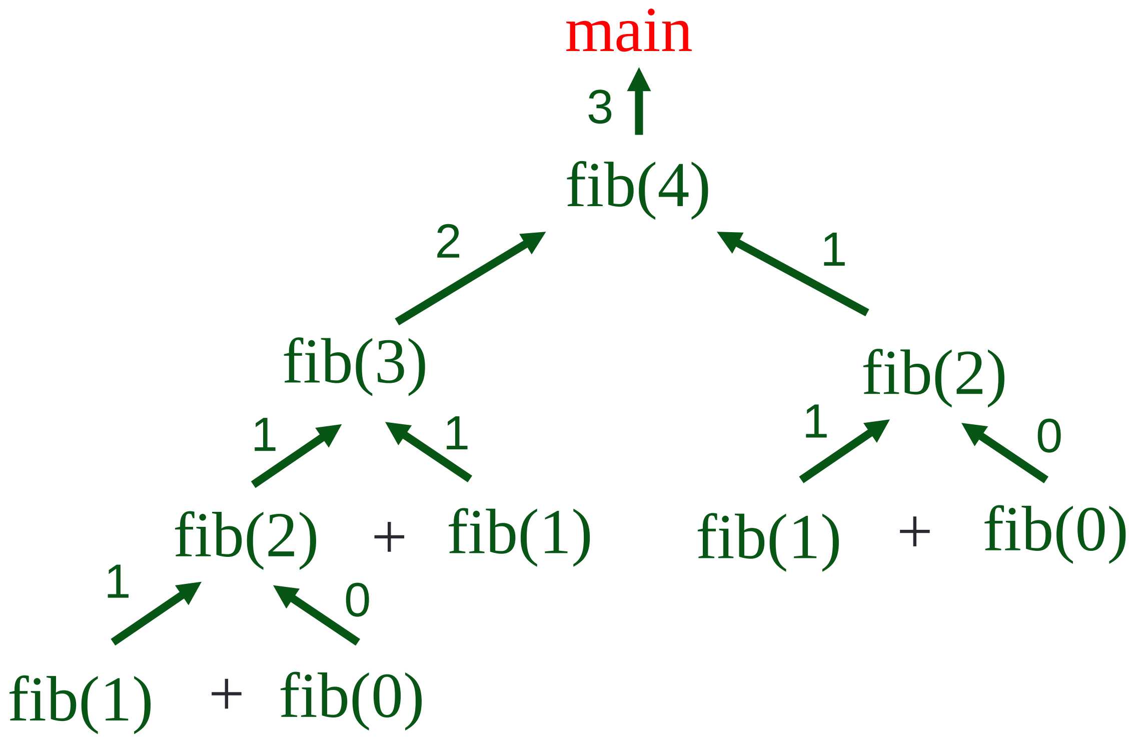


Recursion: Another Example

❑ Recursive approach:

❑ Let’s see the trace of fib(4) in more detail:

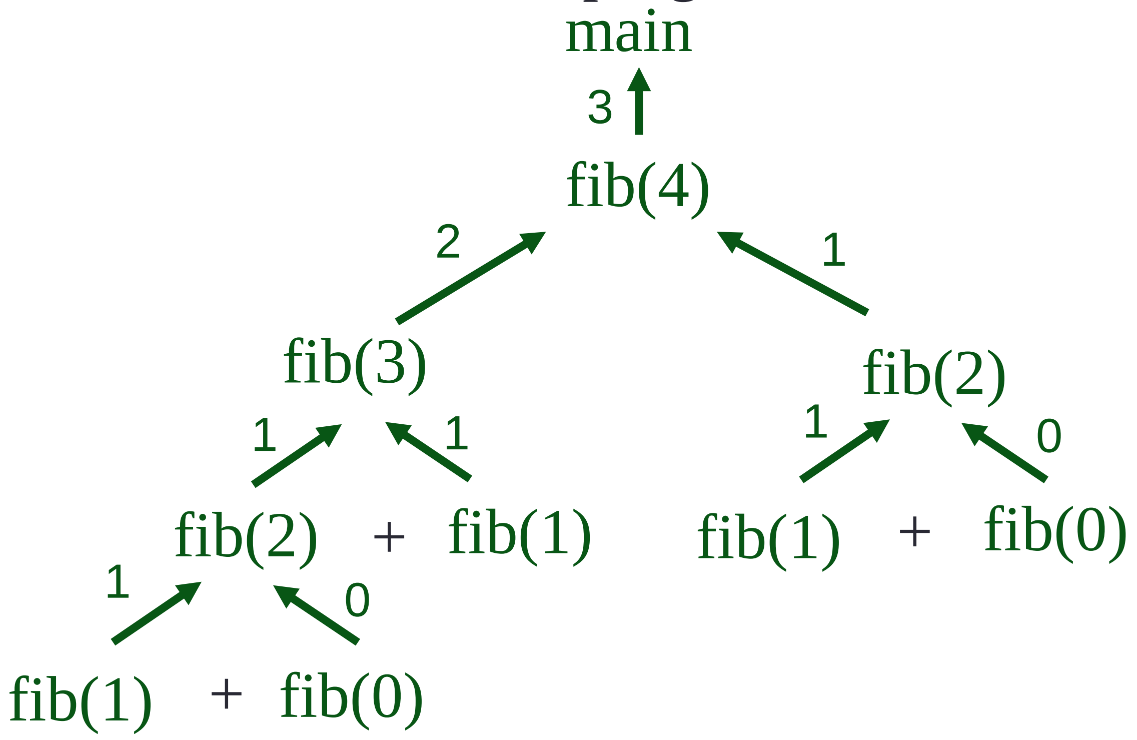
16. fib(4) can compute now its result as 2 + 1 = 3.



Recursion: Another Example

- Recursive approach:
- Let’s see the trace of fib(4) in more detail:

17. The main receives the solution of fib(4), which is 3, and continue with the execution of the program.

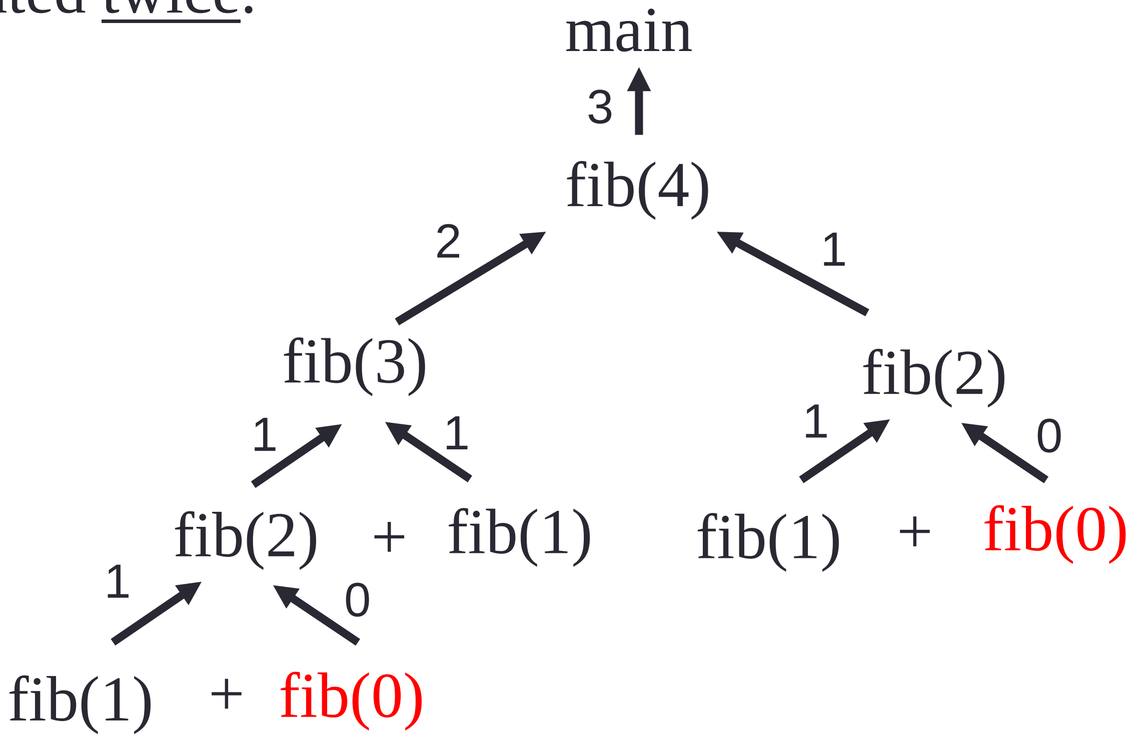


Recursion: Another Example

<u>Recursive Approach Efficiency:</u>

A fundamental drawback of recursive solutions is that they can end up computing the same values over an over again. In our example:

❑ fib(0) is computed <u>twice</u>.

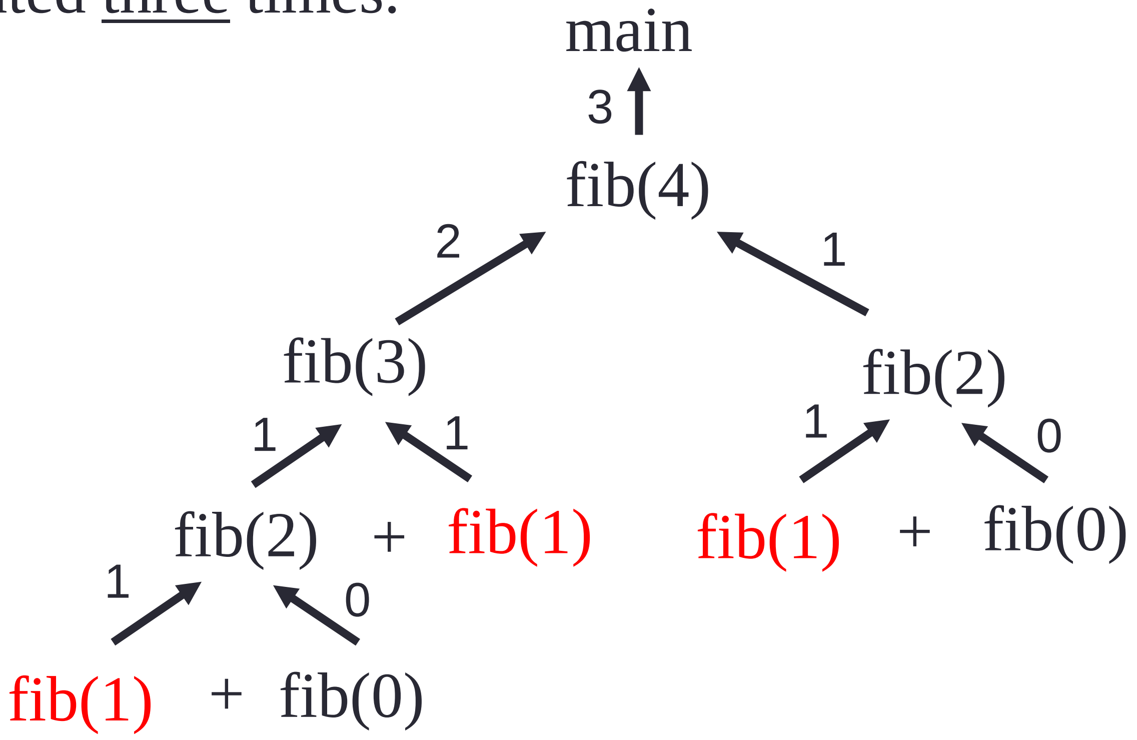


Recursion: Another Example

<u>Recursive Approach Efficiency:</u>

A fundamental drawback of recursive solutions is that they can end up computing the same values over an over again. In our example:

- ❑ fib(1) is computed <u>three</u> times.



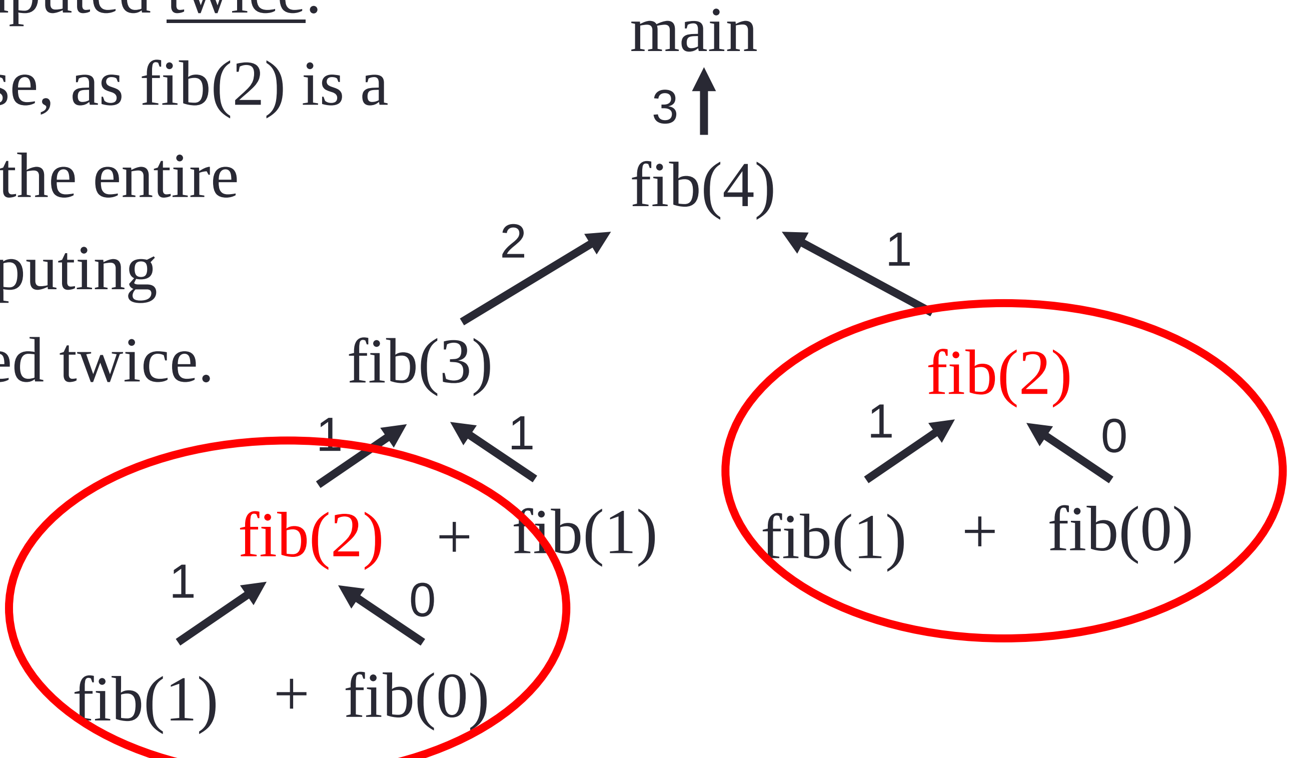
Recursion: Another Example

<u>Recursive Approach Efficiency:</u>

A fundamental drawback of recursive solutions is that they can end up computing the same values over an over again. In our example:

- ❑ fib(2) is computed <u>twice</u>.

And, even worse, as fib(2) is a recursive case, the entire process of computing fib(2) is repeated twice.

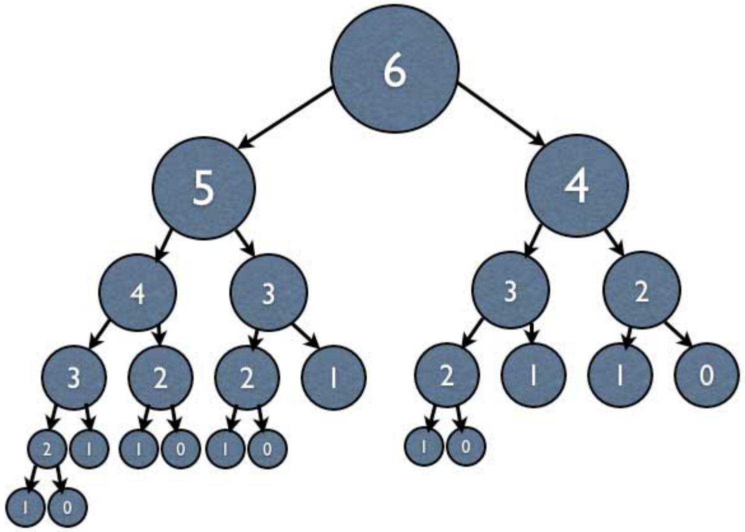


Recursion: Another Example

<u>Recursive Approach Efficiency:</u>

A fundamental drawback of recursive solutions is that they can end up computing the same values over an over again. In our example:

- ❑ Imagine the computation of fib(6).
  - o The entire process of compute fib(2) will be repeated 5 times.
  - o The entire process fib(3) 3 times.
  - o The entire process fib(4) twice
- ❑ So, this is very inefficient!
  - o Imagine a large enough fib(n), with computations repeated thousands of times!



Outline

1. Big Data Analytics.
2. Divide & Conquer Algorithms Schema.
3. Recursion: A Possible Instantiation of the Schema.
4. Concurrency: A Possible Parallelisation of the Map.

Concurrency: A Possible Parallelisation of the Map

As **divide(P)** splits the original problem into independent sub-problems <P1, P2, …, Pk>, then:

map(divide&conquer(<P1, P2, …, Pk>) , can run the processes

{ divide&conquer(P1),

divide&conquer(P2),

…

divide&conquer(Pk)

either in sequential mode or in parallel!

Concurrency: A Possible Parallelisation of the Map

- The file **2.concurrency.py** (included in the code examples of the lecture) presents how to run processes in parallel in Python.
  - It uses the Python package multiprocessing to setup a container for processes → pool (i.e. a thread pool)
  - We define a concrete algorithm dummy_algorithm (expecting three input parameters) to run some dummy code.
  - We create 3 different sets of input parameters:
    - Parameters 1 → (5, 5, 1000000)
    - Parameters 2 → (5, 3, 1000000)
    - Parameters 3 → (5, 7, 1000000)

Concurrency: A Possible Parallelisation of the Map

- The file **2.concurrency.py** (included in the code examples of the lecture) presents how to run processes in parallel in Python.
  - Finally, we use the function **map** of the package multiprocessing so as to run in parallel the 3 processes, i.e.:

    dummy_algorithm(parameters1) → process1

    dummy_algorithm(parameters2) → process2

    dummy_algorithm(parameters3) → process3

  - We gather the results from the three processes in a list *res*.

Concurrency: A Possible Parallelisation of the Map

- Thus, if we open the Windows task manager and run the Python program:
  - We can see that, initially a single thread for Python is created.
  - When the execution of the program reaches the call to the map function → 3 new threads are created.
  - The 3 threads are thus running the three processes (dummy_algorithms with each concrete parameters) in parallel.

Concurrency: A Possible Parallelisation of the Map

- Thus, if we open the Windows task manager and run the Python program:
  - As long as each execution of the algorithm finishes, the thread disappears.
  - However, the proper map function does not finish until <u>all</u> the parallel executions of dummy_algorithm finish.
  - At that moment, only the original Python thread for the main program remains in the Windows task manager, and the execution of the original Python program continues with the next instruction (the one following the map operation).

Thank you for your attention!