Principles of Computer Science II

Divide and Conquer Algorithms

Marco Zecchini

Sapienza University of Rome

Lecture 6

Divide and Conquer Algorithms

A divide-and-conquer algorithm proceeds in two distinct phases:

1. a divide phase in which the algorithm splits a problem instance into smaller problem instances and solves them;
2. a conquer phase in which it stitches/merge the solutions to the smaller problems into a solution to the bigger one.

Why do we need it?

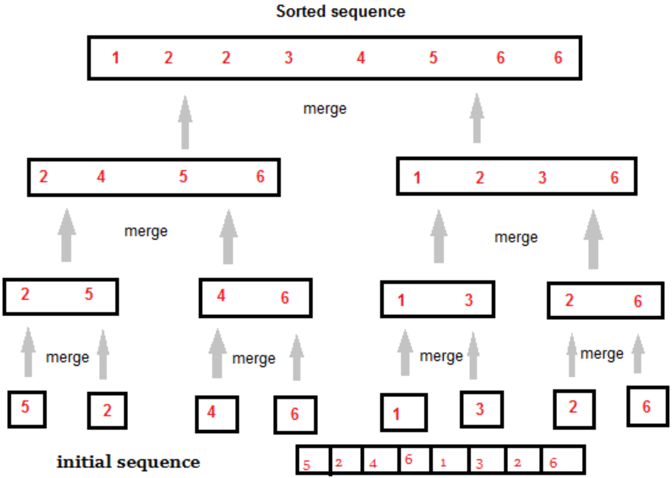
This strategy often works when a solution to a large problem can be built from the solutions of smaller problem instances.

Merge Sort Algorithm

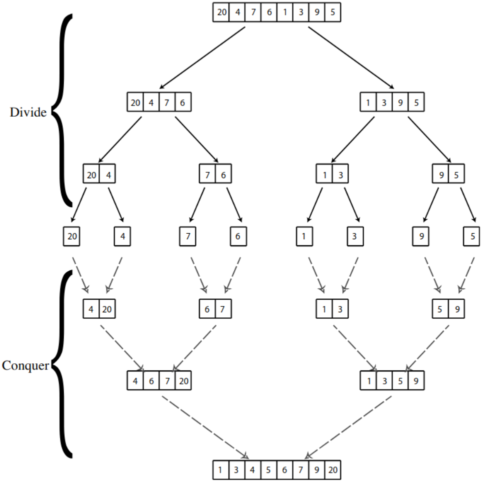
In Merge Sort, an unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

- Divide and Conquer algorithm
- Performance always same for Worst, Average, Best case

Merge Sort: Example



Merge Sort: Divide and Conquer algorithm



Merge Sort Code

```
a = [25, 52, 37, 63, 14, 17, 8, 6]

def mergesort(list):
    if len(list) == 1:
        return list

    left = list[0: len(list) // 2]
    right = list[len(list) // 2:]

    left = mergesort(left)
    right = mergesort(right)

    return merge(left, right)
```

Merge Sort Code

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))

    while len(left) > 0:
        result.append(left.pop(0))

    while len(right) > 0:
        result.append(right.pop(0))

    return result

print("Before: ", a)
r = mergesort(a)
print("After: ", r)
```

How good is Merge Sort?

- How many comparisons are required until the list is sorted?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n comparisons
  - In total $n \log n$ comparisons

Searching algorithms

Do we know an algorithm/technique to find an element in a list?

Searching algorithms

Do we know an algorithm/technique to find an element in a list?
Which is its time complexity?

Divide and conquer for search problem - Binary Search

- Binary search is an efficient algorithm for finding an element in a sorted list.
- It requires the array to be sorted.
- The time complexity is $O(\log n)$.

How it Works

1. Compare the target element with the middle element of the array.
2. If the target is equal to the middle element, the element is found.
3. If the target is smaller, search in the left half; if it's larger, search in the right half.
4. Repeat until the element is found or the array is exhausted.

Binary Search: recursive approach

```
def binarySearch(arr, low, high, x):
    # Check base case
    if high >= low:
        mid = low + (high - low) // 2
        # If element is present at the middle itself
        if arr[mid] == x:
            return mid
        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, low, mid-1, x)
        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, high, x)
    # Element is not present in the array
    else:
        return -1
```

Example: Recursive Binary Search

- Array: $\{1, 2, 4, 5, 7, 9, 10, 15, 20, 25, 30, 35, 40, 50\}$
- Target: 15

1. Call: `binarySearch(arr, 0, 13, 15)` $\Rightarrow$ `mid = 6`, `arr[mid] = 10` $\Rightarrow 15 > 10 \Rightarrow$ recursive call on right half
2. Call: `binarySearch(arr, 7, 13, 15)` $\Rightarrow$ `mid = 10`, `arr[mid] = 30` $\Rightarrow 15 < 30 \Rightarrow$ recursive call on left half
3. Call: `binarySearch(arr, 7, 9, 15)` $\Rightarrow$ `mid = 8`, `arr[mid] = 20` $\Rightarrow 15 < 20 \Rightarrow$ recursive call on left half
4. Call: `binarySearch(arr, 7, 7, 15)` $\Rightarrow$ `mid = 7`, `arr[mid] = 15` $\Rightarrow$ target found

Target found at index 7

Binary Search: iterative approach

```
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
         # Check if x is present at mid
        if arr[mid] == x:
            return mid
        # If x is greater, ignore left half
        elif arr[mid] >= x:
            left = mid + 1
        # If x is smaller, ignore right half
        else:
            right = mid - 1
    return -1
```

Example

- Array: $\{1, 2, 4, 5, 7, 9, 10, 15, 20, 25, 30, 35, 40, 50\}$
- Target: 15

1. `left` $= 0$, `right` $= 13$
2. `mid` $= 6$ (`array[mid]` $= 10$), since $15 > 10$, search in the right half
3. Update `left` to `mid + 1 = 7`
4. `mid` $= 10$ (`array[mid]` $= 30$), since $15 < 30$, search in the left half
5. Update `right` to `mid - 1 = 9`
6. `mid` $= 8$ (`array[mid]` $= 20$), since $15 < 20$, search in the left half
7. Update `right` to `mid - 1 = 7`
8. `mid` $= 7$ (`array[mid]` $= 15$), **target found at index 7**

Questions

Space complexity of the two algorithms? Which is better?

Questions

Space complexity of the two algorithms? Which is better?

Considering the recursive call stack then the auxiliary space for the recursive approach is $O(\log N)$.

The iterative approach space complexity is $O(1)$

Questions

Can we do this algorithm on an unsorted list?

Questions

Can we do this algorithm on an unsorted list?

NO

Why Divide and Conquer performs better on large datasets

- **Smaller subproblems:** Each recursive call divides the problem into smaller pieces, quickly reducing the input size. For example, Binary Search halves the search space at each step.

Why Divide and Conquer performs better on large datasets

- **Smaller subproblems:** Each recursive call divides the problem into smaller pieces, quickly reducing the input size. For example, Binary Search halves the search space at each step.
- **Adaptability to large data:** Large datasets are often hierarchical or structured; divide and conquer naturally fits this structure.

Why Divide and Conquer performs better on large datasets

- **Smaller subproblems:** Each recursive call divides the problem into smaller pieces, quickly reducing the input size. For example, Binary Search halves the search space at each step.
- **Adaptability to large data:** Large datasets are often hierarchical or structured; divide and conquer naturally fits this structure.
- **Parallelism:** Independent subproblems can be processed simultaneously on multiple processors/machines, making divide and conquer ideal for modern parallel architectures.

Problem: Lots of data

- Example: Homo sapiens high coverage assembly GRCh37
  - 27478 contigs.
  - contig length total 3.2 Gbase.
  - chromosome length total 3.1 Gbase.
  - Multiple TBs of data for human genome.
- One computer can read 30-35MB/sec from hard disc
  - $\sim$ 10 months to read the data
- $\sim$ 100 hard drives just to store the data in compressed format
- Even more to do something with the data.

Spread the work over many machines

Spread the work over many machines

- Good news: same problem with 1000 machines: ≤ 1 hour

Spread the work over many machines

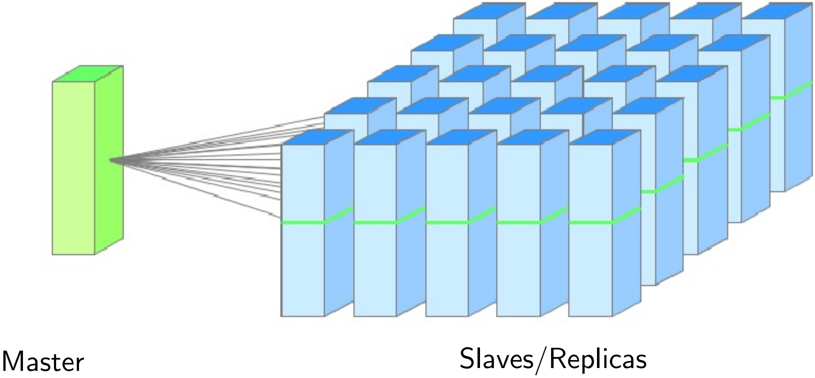
- Good news: same problem with 1000 machines: $\leq$ 1 hour
- Bad news: concurrency
  - communication and coordination
  - recovering from machine failure
  - status reporting
  - debugging
  - optimization

Spread the work over many machines

- Good news: same problem with 1000 machines: $\leq$ 1 hour
- Bad news: concurrency
    - communication and coordination
    - recovering from machine failure
    - status reporting
    - debugging
    - optimization
- Bad news 2: repeat for every problem you want to solve

Computing Clusters

- Many racks of computers
- Thousands of machines per cluster
- Limited bandwidth between racks



Computing Environment (e.g., AWS, Google Cloud Service)

- Each machine has 2-4 CPUs
  - Typically quad-core
  - Future machines will have more cores
- 1-6 locally-attached disks
  - $\sim$ 10TB of disk
- Overall performance more important than peak performance of single machines
- Reliability
  - In 1 server environment, it may stay up for three years (1000 days)
  - If you have 10000 servers, expect to lose 10 each day
- Ultra reliable hardware still fails
  - We need to keep in mind cost of each machine

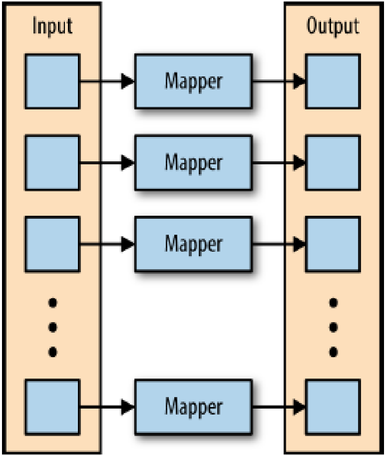
Map Reduce Computing Paradigm

- A simple programming model
  - Applies to large-scale computing problems
- Hides difficulties of concurrency
  - automatic parallelization
  - load balancing
  - network and disk transfer optimization
  - handling of machine failures
  - robustness
  - improvements to core libraries benefit all users of library

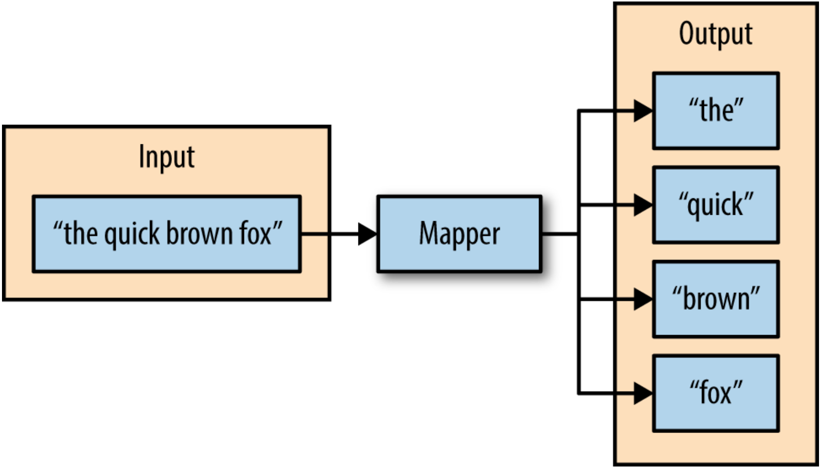
A typical problem

- Read a lot of data
- Map: extract something important from each record
- Shuffle and sort
- Reduce: aggregate, summarize, filter or transform
- Write the results

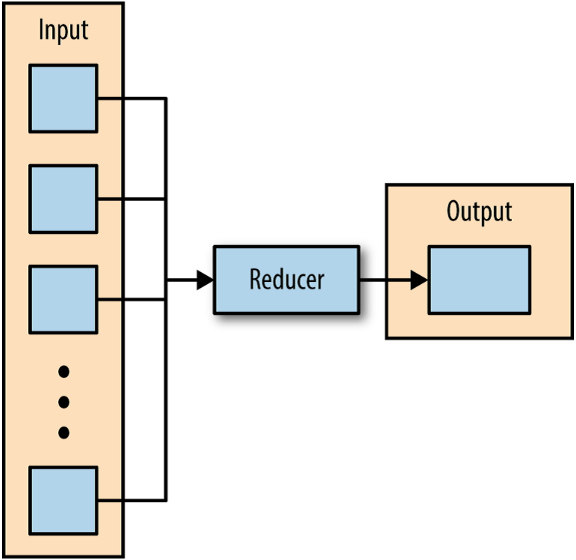
How map works



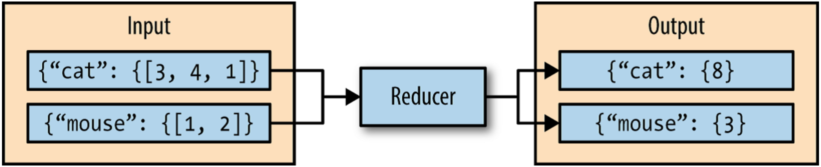
Map Reduce Computing Paradigm



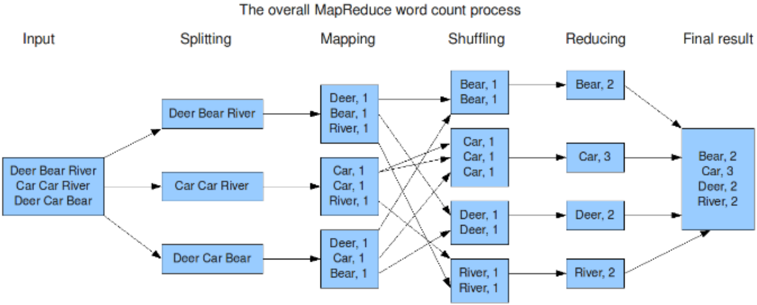
Map Reduce Computing Paradigm



How reduce works



Word Count example



In more details

- Programmer specifies two primary methods:
  - map($k, v,$ **script**) $\rightarrow$ $< k', v' >^*$
    - Takes a key-value pair and outputs a set of key-value pairs arranged according to **script**
    - $*$ denotes a set of pairs
    - There is one Map call for every (k, v) pair
  - reduce($k', < v' >^*,$ **script**$'$) $\rightarrow$ $< k', v' >^*$
    - All values v' with same key k' are reduced together with **script**$'$ and processed in v' order
    - There is one Reduce function call per unique key k'
- All v' with same k' are reduced together with **script**$'$, in order.

An example: Frequencies in DNA sequence

A typical exercise for a new engineer in his/her first week:

- Input files with one document per record
- Specify a map function that takes a key/value pair
  - key = document URL
  - value = document contents
- Output of map function is (potentially many) key/value pairs.
- In this case, output:
  (word, 1) once per word in the document

"document 1", "CTGGGCTAA"
converted to
(C, 1), (T, 1), (G, 1), (G, 1), (G, 1), (C, 1), . . .

An example: Frequencies in DNA sequence

- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The reduce function combines the values for a key
- In this example:

key = "A"	key = "G"	key = "C"	key = "T"
values = 1, 1	values = 1, 1, 1	values = 1, 1	values = 1, 1
summarize	summarize	summarize	summarize
2	3	2	2

- Output of reduce paired with key and saved

(A, 3), (G, 3), (C, 2), (T, 2)

An example: Frequencies in DNA sequence

```
s = 'CTGGGCTAA'
seq = list(s)  #['C', 'T', 'G', 'G', 'G', 'C', 'T', 'A', 'A
    ']
sc.map(lambda symbol: (symbol, 1))\
  .reduce(add)\
  .collect()
```

Output:

```
[('A', 2), ('C', 2), ('G', 3), ('T', 2)]
```

Fault tolerance: handled via re-execution

In large scale computation on multiple nodes, there is a master that orchestrate the entire computation and workers that executes what the master tell them to do.

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- On master failure:
  - Restart execution

Let us see map reduce in Python

Open this Jupyter Notebook and let us see how to use MapReduce (there are two exercises at the end): `https://drive.google.com/file/d/1Cf3UWGZPiOG9iXvIXpsh2jIlAmu6WlF view?usp=drive_link`