Lecture 3: Counting at Scale: MapReduce
Modeling Social Data, Spring 2017
Columbia University

Aarshay Jain

February 3, 2017

Part1: Guest Lecture

We would cover the following topics in today's class:

- Computational Tractability: how to measure running time
- Asymptotic order of growth: how to analyze and compare running times
- Common running times: typical data structures and algorithms

1 Computational Tractability

The running time is measured in terms of input size - n. The typical run times are:

- 2^n (exponential)
  - generally brute force algorithms have exponential running times
  - this running time is not ideal as problems become practically unsolvable for even small running times
- $cn^d,\ c, d > 0$ (polynomial)
  - polynomial time algorithms are considered to be theoretical efficient and theoreticians stop here
  - but practically, the constants c,d matter. for example, if we compare $20n^100$ and $n^{1+0.02log(n)}$, the later is prefered though is exponential because the former polynomial runtime has very high coefficients.
- Order of growth: $n < nlog(n) < n^2 < n^3 < 1.5^n < 2^n < n!$
  - there are some sub-linear algorithms as well which are better than linear but are mostly approximate algorithms
  - the ideal range for an algo is below n^2 as anything beyond that becomes difficult to handle as data size increases

Different types of analysis of run-time can be performed as following:

- Worst Case Analysis: this is most commonly used and compares algorithms on asymptotic performance
- Average Case Analysis: here we analyze how the algo performs on average. we need some sense of the number of times a particular operation is expected to be run to be able to get the average
- Amortized Analysis: here the time taken by a sequence of operations is analyzed
- These 3 can be compared using the example of push and popall into a stack. A stack is a LIFO (last in first out) data structure.

- Worst case: push - $O(1)$ — popall - $O(n)$
- Average Case: if popall is a rarely called, then average runtime is $O(1)$
- Amortized: a popall operation will take only as much time as the number of push operations before it. So a sequence of push and popall will take $O(1)$ amortized time. This can be understood as every push operation investing 1$ in the bank on each call and popall using that invested money to perform the operation. So the amortized cost is constant.

2 Asymptotic Order of Growth

Three types of notations can be used for asymptotic analysis. These are:

2.1 Big-Oh (O) - Upper Bounds

This specifies the upper bound on the running time of an algorithm. The runtime T(n) can be written as:

$$T(n) \ = \ O(f(n)) \ if \exists \ c > 0, n_0 \geq 0$$

$$s.t. \ T(n) \leq c.f(n), \ \forall \ n > n_0$$

Example: $T(n) = 32n^2 + 17n + 1 \ ==> \ T(n) = O(n^2)$ This can be verified by taking any c greater than 32 say c=50 and n_0=1

Note that theoretically $T(n) \subset O(f(n))$ but using $T(n) = O(f(n))$ is accepted in the computer science community.

2.2 Big-Omega (Ω) - Lower Bounds

This specifies the lower bound on the running time of an algorithm. The runtime T(n) can be written as:

$$T(n) \ = \ \Omega(f(n)) \ if \exists \ c > 0, n_0 \geq 0$$

$$s.t. \ T(n) \geq c.f(n), \ \forall \ n > n_0$$

Example: $T(n) = 32n^2 + 17n + 1 \ ==> \ T(n) = \Omega(n) = \Omega(n^2)$

2.3 Big-Theta (Θ) - Tight Bounds

This specifies a tight bound on the running time of an algorithm, i.e. an order of polynomial which can be both an upper and lower bound with different constants. The runtime T(n) can be written as:

$$T(n) \ = \ \Theta(f(n)) \ if \exists \ c_1, c_2 > 0, n_0 \geq 0$$

$$s.t. \ c_1.f(n) \leq T(n) \leq c_2.f(n), \ \forall \ n > n_0$$

Example: $T(n) = 32n^2 + 17n + 1 \ ==> \ T(n) = \Theta(n^2)$ as n^2 is both an upper as well as a lower bound

2.4 Rules of Thumb

The following rules can be typically applied when determining the running time:

1. $T(n) = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d \ ==> \ T(n) = \Theta(n^d)$
2. $O(log_a n) = O(log_b n), \ a, b > 0$ [base of log doesn't matter because base change involes a constant factor]
3. $n^d = O(r^n), \ \forall \ r > 1, d > 0$ [any polynomial runtime will be dominated by exponential asymptotically]

3 Common Running Times

Now lets understand some common running times.

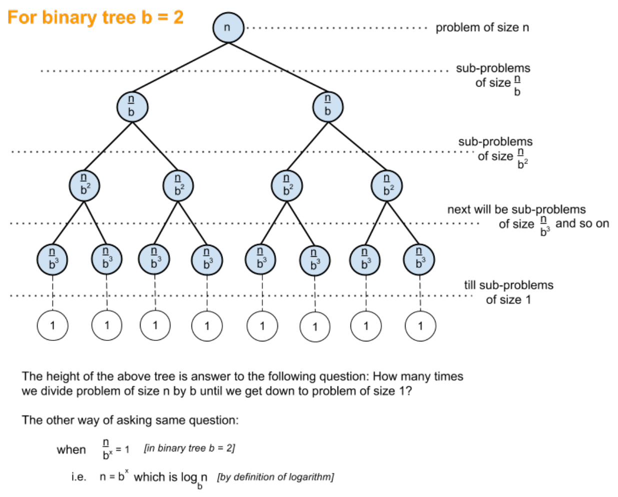
3.1 $O(n)$ - linear

The algo iterates over all the entries of the data a constant number of times. Eg: max, sum, min, etc.

Note: This assumes that constant time is spent at every iteration. But note that any number K requires $log_2(K)$ bits to be represented so it takes $log_2(K)$ time at each index. However, modern computers perform all these tasks in parallel so we can ignore this effect.

3.2 $O(nlogn)$

This is generally the runtime of divide and conquer algorithms like sorting. Figure 1 illustrates the concept well.



For binary tree b = 2
problem of size n
sub-problems of size n/b
sub-problems of size n/b^2
next will be sub-problems of size n/b^3 and so on
till sub-problems of size 1
The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to problem of size 1?
The other way of asking same question:
when n/b^x = 1 [in binary tree b = 2]
i.e. n = b^x which is log_b n [by definition of logarithm]

Figure 1: Working of divide and conquer algorithms [2]when problem gets divided into 2 sub-problems at each step

Key observations:

- each problem gets divided into 2 sub-problems at each step
- at any particular step (horizontal layer), equal amount, i.e. $O(n)$ time is being spent
- there are $O(log_2(n))$ such layers
- so total time is $O(nlogn)$ [note: base of log doesn't matter in big-oh notations]

Note: it has been proven that any comparison-based sorting algorithm can't do better than $O(nlogn)$.

3.3 $O(n^2)$ - quadratic

In this case, all of the n observations are iterated over a some linear function of n. So total time is quadratic. Eg: bubble sort, quick sort

3.4 $O(n^3)$ - cubic

Example: counting the number of triangles in graphs or number of triplets in a node.

Note: matrix multiplication of two nxn matrices is cubic in n but the input is not really n but n^2. So its debatable whether these are cubic runtime algorithms or not.

[2]https://i.stack.imgur.com/spHFh.png

3.5 $O(logn)$

These are highly efficient algorithms which run without even scanning the full array once. Example: binary search on sorted arrays, binary tree search on balanced trees

3.6 Hash Tables

They allow searching in near constant time. Figure 2 describes the working of hash tables.

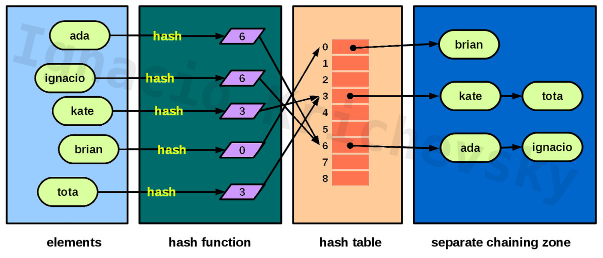


Figure 2: Working of hash tables with chaining [3]

It works as following:

- a hash function h converts an input element into an index of the hash table where it is stored
- the function h randomizes the placement of items so that each element has equal probability of being at any index in the hash table

An idea hash function is the one which gives a unique index to each element. If all the elements are known beforehand, then its possible to define such a hash function. But if the inputs are unknown, then there might be collisions in the hash table, i.e. multiple elements being hashed to the same index.

A collision occurs when $\exists k1, k2\ s.t.\ h(k1) = h(k2)\ but\ k1 \neq k2$. When this happens, a chain is formed at the index of collision as shown in Figure 2. The expected number of collisions depend on the number of items (n) to be hashed and the size of the hash table (t).

$$p(collision) = \frac{1}{n}$$
$$E(\#collisions) = \binom{n}{2}\frac{1}{n} = O(n)$$

Thus, the expected number of collisions per bin is $O(1)$. Typically, the size of the hash table is kept slightly bigger than n.

3.7 Joining tables

Problem: Given 2 tables A and B with a common columns say ID, we need to merge the rows with same IDs together. There are various ways to solve the problem:

- For every ID in table A, search in entire table B for matching ID and merge. This takes $O(n^2)$ time
- Create an index on the ID column for table B. Now for every element in table A, we can search for an element in table B in $O(logn)$ time. So the total time is $O(nlogn)$.

[3] http://krichevsky.com.ar/root/programming/data%20structures%20&%20algorithms/hashTableWithSeparateChaining.png

- Create a hash function which maps IDs into a hash table. Push IDs of table A into the hash table. Then for every ID in table B, push it to the hash table and combine the data with that of table A in the same index. This gives a $O(n)$ time algorithm for joining.
- If both the ID columns of table A and table B can be sorted, then you can simply walk through the IDs in linear time and merge.

Part2 - Data Manipulation in R using tidyverse

In this section, I'll try to augment the content of slides with additional information discussed in class. The notes are referenced by slide number of the updated slides on GitHub repository:

- Slide #5:
  - factor variables are coded as numeric in the background and appears as labels to the user. we can do str(variable_name) to know the real structure.
- Slide #6:
  - dataframes are different from matrices in the same way as lists are from vectors. matrices accept data of single type but different columns of dataframes can have different types of data.
- Slide #10:
  - In this slide, N: #rows, K: #columns
  - for-loops are constructs in R but we should try to avoid them as much as possibe. These functions give faster ways of acheiving the same objective as for-loops. If you're using a for-loop, you're probably missing a trick.
  - All functions, except group_by, are not inplace, i.e. they won't affect the original dataframe but return a new dataframe which can be stored different or overwritten on the old dataframe.
- Slide #11:
  - Note that here "start_station_name" is not a variable in the R environment but the name of a column.
  - filter would first try to parse this as a column in the dataframe trips and if not found then it will look into the global environment. So you've to be careful if you have variable names which are same as column names.
  - the same concept applies to other functinos as well.

Finally, some notes on the "wierdness" of R over general programming languages:

- it uses "$<-$" as an assignment operator and not "$=$". Even "$=$" works but there is some difference in the background.
- there are some datasets which are lurking in the background but you don't see them in your environment unless you explicitly save them in a variable. For example "head(iris)" will give you the top 6 rows of the iris dataset which might not even by in your environment. "iris $<-$ iris" will save it in your environment as well.
- "$" can be used to index columns but try to avoid it
- comparison operators "&" and "&&" are different. Former works vectorized, i.e. element by element while later works considering every operand as a single element.
- %>% are R's version of pipes we saw in unix