Lecture 3: Computational complexity
Modeling Social Data, Spring 2017
Columbia University

February 3, 2017

Notes from aj2713

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.

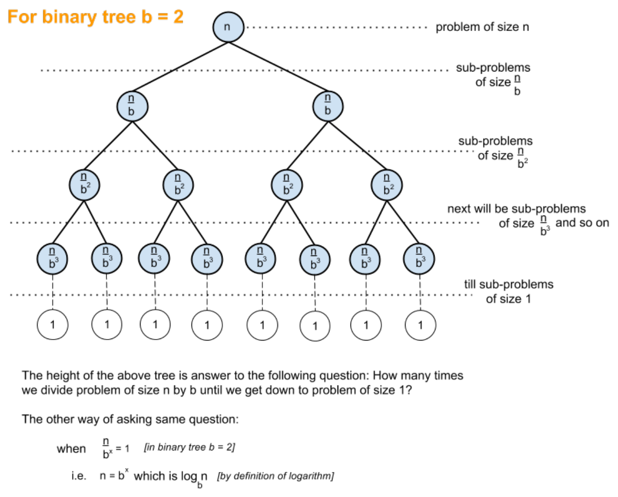

For binary tree b = 2
problem of size n
sub-problems of size n/b
sub-problems of size n/b²
next will be sub-problems of size n/b³ and so on
till sub-problems of size 1
The height of the above tree is answer to the following question: How many times we divide problem of size n by b until we get down to 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

3.2 $O(nlogn)$

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

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.

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.

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

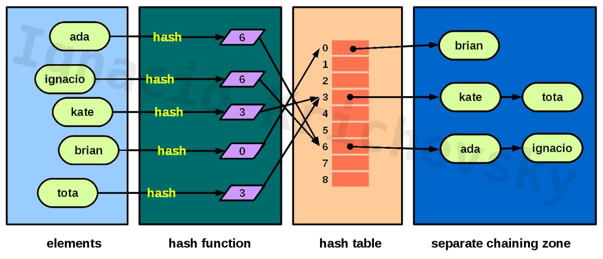


Figure 2: Working of hash tables with chaining [3]

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 $O(nlogn)$.
- 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.

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

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

Notes from ksz2109

1 Introduction

In determining the efficiency of an algorithm, we will want to discuss **worst case**, **average case**, and **amortized** running times, in terms of input n.

- Worst case running time is the longest an algorithm would take to run on input n.
- Average case is the amount of time an algorithm would take to run on some average input n. This requires identifying what an average input would be, e.g. what an average string might look like.
- Amortized is the amount of time an algorithm would take to run on input n making no assumptions about what average input looks like.

For example, a stack data structure has `push`, `pop`, and `pop_all` operations.

- We would say `pop` runs in $O(1)$ time, since we only need to perform one operation on the stack to remove one item.
- We would say `pop_all` runs in $O(n)$ time for a stack of size n, since we need to call `pop` once for each of n elements in the stack.

1.1 Upper Bounds

The running time $T(n)$ of an algorithm on some input of size n is $O(f(n))$ if $\exists c > 0, n_0 \geq 0$ such that $\forall n > n_0,\ T(n) \leq c_1 f(n)$.

1.2 Lower Bounds

The running time $T(n)$ of an algorithm on some input of size n is $\Omega(f(n))$ if $\exists c > 0, n_0 \geq 0$ such that $\forall n > n_0,\ T(n) \geq c_1 f(n)$.

1.3 Examples of Common Running Times

- $O(n)$ a.k.a. linear time. Example of this would be to compute the maximum number in an array of numbers.
- $O(n \log n)$. Example of this would be mergesort, average case performance of quicksort, and other divide-and-conquer algorithms.
- $O(n^2)$ a.k.a. quadratic time. Example of this would be bubble sort and worst case performance of quicksort.
- $O(n^3)$ a.k.a. cubic time. Example of this would be naive matrix multiplication.

1.4 Misc

A rule of thumb is that for every $r > 1$, $d > 0$, $n^d = O(r^n)$.

2 Hash Tables

This data structure relies on a **hash function** that maps the input to an integer between 1 and n, where n is the size of the hash table. Here are some things to know about hash tables

- Most hash tables are implemented using separate chaining, in which each element of an array contains a linked list of elements. Items are retrieved and added to the hash table by first hashing the input to determine which entry in the array it belongs to, then is appended to the linked list in that entry (if adding) or that linked list is searched (if retrieving).

- Because in most practical applications hash functions do not run in $O(1)$ time, and because of separate chaining, there is no guarantee about accessing elements in constant time.
- The hash function, if designed well, does a good job at spacing out elements in the hash table such that the load factor (average length of each linked list in the array) is minimal.
- The hash function, if designed well, has a probability of collision at $1/n$, where n is the length of the array. As a result, for n insertions, the expected number of collisions is $n * 1/n = O(1)$.

3 Join

The join operation takes as input two tables that has some data (e.g. a column) in common, and the output is one table that combines information from the two tables. For example, we can have a table where each row has a user ID and a phone number, and another table where each row has a user ID and an address. A join between these two tables on user ID would yield a new table where each row contains both the address and phone number of the corresponding user.

The following are some approaches to computing a join operation:

- For each row in A, find the corresponding row in B. This runs in $O(nm)$, where n is the number of rows in A, and m the number of rows in B.
- Create an index (e.g. using a B-tree) that spans the rows of table B. Then whenever we want to find the corresponding row in table B, it takes only $O(\log m)$ time. However, our total run time is $O(n \log m)$.
- Use a hash function on each row in both table A and B so the rows with the same ID get inserted into the same bin. Then, for each bin, perform the necessary join operations. This runs in $O(n + m)$ time, since we only go through each element in both tables twice: once to put them in the corresponding bin, and once to combine it with the other element in the bin.

4 Intro to R

The rest of the class was spent on slides and demonstrations on RStudio.

Notes from ts2838

1 Introduction

This is an overview of Sid's lecture for Modeling Social Data. These notes will cover three main topics:

1. Computational Tractability
2. Asymptotic Order of Growth (Big O)
3. Common Running Times
   - Data Structures
   - Algorithms

In general, when we are analyzing algorithms, we want to think about the worst-case, average-case, and amortized run time of them.

2 Definitions

Let us start with the concept of Big O notation and define the upper bound. $T(n)$, the running time, is O($t(n)$) if there exists a $c >$ 0, $n_0 \geq$ 0 such that $T(n) \leq c \cdot t(n)$ for all $n > n_0$. Here is an example. Suppose $T(n) = 32n^2 + 17 \cdot n + 1$. Therefore, $T(n) = O(n^2)$ for $c = 50, n_0 = 1$.

The lower bound is the following. $T(n)$ is $\Omega(t(n))$ if there exists a $c >$ 0, $n_0 \geq$ 0 such that $T(n) \geq c \cdot t(n)$ for all $n > n_0$. Using the same $T(n)$ as defined in the upper bound example, we get $T(n) = \Omega(n^2)$ for $c = 31, n_0 \geq 1$.

The tight bound is defined as follows. $c_1 \cdot t(n) \leq T(n) \leq c_2 \cdot t(n)$ for all n $> n_0 = 1$ such that $T(n) = \Theta(T(n))$. Using the same T(n) as defined in the upper bound example, we get $T(n) = \Theta(n^2)$.

3 Rules of Thumb

1. If $T(n) = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d$, then T(n) = $\Theta(n^d)$.
2. $O(\log_a n) = O(\log_a n)$. Doesn't matter what base is.
3. For every $r > 1$ and $d > 0$, $n^d = O(r^n)$. Exponential function will always be dominated by a polynomial. This is true for big n, not necessarily for small n.

4 Examples

1. $O(n)$. An example of algorithm that runs linear time is computing the max of n numbers. The algorithm would set initialize the maximum at 0 and then iterate through the set, and compare each value i to the maximum. If the value i is greater than the max value, the max value would become i. We assume computer computation is $O(1)$. In reality, the real amount of time depends on the bits sent. Because the set has length n, there are $n \cdot O(1) = O(n)$ comparisons and as such runs in linear time
2. Divide and conquer algorithms such as sorting have $O(n \cdot log(n))$ time. There are $log(n)$ levels since the amount doubles every time, and on each level it takes $O(n)$ time to sort. $\Omega(n \cdot logn)$ is much harder to determine, as it is much more difficult to come up with a hypothetical lower bound as one needs to show there is no possible computational model that is higher than the proposed lower bound.

3. Bubblesort has quadratic time, $O(n^2)$. This is because their are $n \cdot O(n) = O(n^2)$, where there are n items in the array and a linear scan of the array takes $O(n)$ (from example 1).

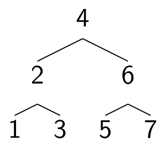
4. Quicksort has an expected run time of $O(n \cdot log(n))$ if the picks are pretty good, which they are on average. The worst case time for quick sort $O(n^2)$.

5. $O(n^3)$ often comes up in graphs. Specifically, if you are trying to count triangles in a graph, it would take cubic time. You would think that matrix multiplication is $O(n^3)$, but in actuality the fastest running time is something like $O(n^{2.37})$.

6. $O(log(n))$ is even better than linear time. Each iterations halves the amount of array, so it reduces in size exponentially, and as such takes $O(log(n))$.

5 Binary Trees

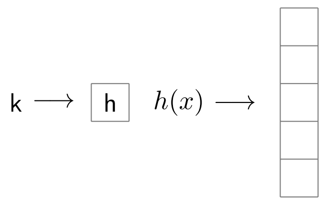
Trees are useful because they allow you to find things quickly.



As we can see, it much quicker to find a number in a tree then it is linearly since linear search has n time which is bad.

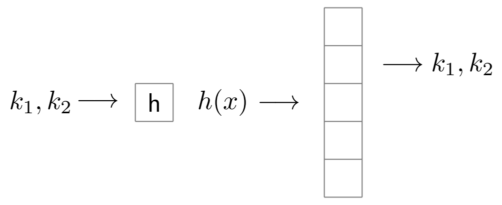
6 Hash Tables

A hash table is a data structure that maps keys to values. Here is an illustration:



The hash is h, and the function $h(x)$ defines which bin to put each item. This is done so as to minimize the length of each bucket in the hash. A hash table has $O(1)$ access, which is a big increase over using a tree for searching.

Now let's think about the chance that 2 items k_1, k_2 collide in the same bin, represented by:

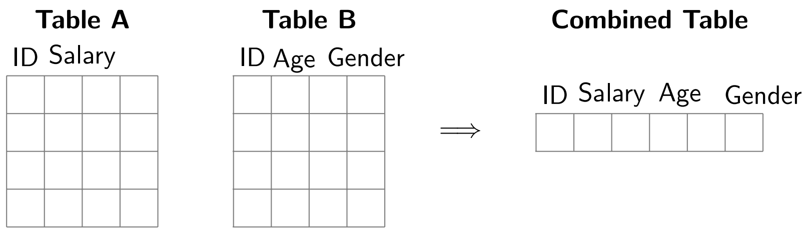


Here, k_1, k_2 is a linked list for the second from the top bin. This function is deterministic, so for the same input, the function will spit out the same output. Now, how long will the linked list get? On average,there should be a constant since the hash function should spread this out. The probability of a collision is $P(colllision) = \frac{1}{n}$. The expected number of collisions therefore is $\binom{n}{c} \cdot \frac{1}{n} = \frac{O(n)}{n} = O(n) \cdot$ items per bin.

What size hash table should we want? We only want the hash table to be slightly bigger than n, as that will have a pretty good distribution. This is an application of the balls and bins problem. The running time of this is $\Theta(\frac{lg(n)}{lg(lg(n))})$.

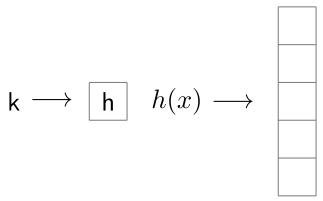
7 Application to Joining Data

We will now take all that we have learned and apply it to gain a better understanding of the running time associated with joining data. Suppose we have 2 tables and want to combine them:



The tables are of size $n \times k$. The first column of each table corresponds to the ID, the others correspond to different features. Table A and Table B have different features; for example, Table A might have data on Salary while Table B might have data on age and gender (as shown above). The question is, how do you combine these tables into one table which has all the features corresponding to the same ID?

If you naively search both columns in a linear scan, the running time is $O(n^2)$. But how could we do better? We could use a binary tree to create an index. It would take log time to find index through tree, and for n rows, the running time would be $O(n \cdot log(n))$ which is an improvement. If we want to move from log to constant time, we'll have to use hashing:



Here, our k will be the rows and our buckets will be rows of the combined table. How long will it take to make this table? Each operation takes constant time, so this is linear and as such runs in $O(n)$ time. You put rows from Table B into the hash, and where there is a collision, the ID is both in Table A and B. This works because the hash table is deterministic, and the features that correspond to the same ID for each table will be sent by the hash function to the same bucket. It wouldn't work if the hash function places rows in a table randomly.

8 Conclusion

This was a high level overview of data structures and algorithms. For some parting wisdom, Sid mentioned that if you connect the person who discovered the algorithm to the algorithm itself it makes studying them that much more exciting. After Sid's lecture, Hoffman introduced us to the R programming language.

Notes from yz3012

NOTES

1.Computational Tractability

- The optimal value could be calculated in a reasonable amount of time.

2.Asymptotic order of growth — Big O notation

- It is used to understand approximation when variables go to infinite.
- In computer science, big O notation is usually used to evaluate algorithms on their running time and space as the input size grows.

3.Common running times

- It depends on data structure and algorithms.

4.Running time in size n input

- brute force algorithm : $2^n \Rightarrow$ exponential
- cn^d $(c > 0\ d > 0) \Rightarrow$ polynomial
- It is theoretical step here, but does not perform well in practice purpose.
  Constant matter: $\underbrace{20n^{100}}_{polynomial}$ vs $\underbrace{n^{1+0.02lgn}}_{notpolynominal, butstillbetter}$ $\Rightarrow$ So notice that constant cannot be ignored!
- Order of running time: $n < nlogn < n^2 < n^3 < 1.5n^3 < 2^n < n!$

5.Runtime analysis

1. Worst-case Complexity analysis(Our focus):
   Worst-case Complexity analysis considers the maximal complexity of the algorithm over all possible inputs.
2. Average-case Complexity:
   Average-case Complexity considers the average complexity over all possible inputs. (Making an assumption of the statical distribution of the inputs $\Rightarrow$ expected time, uniform distribution) It may be a more accurate measure of an algorithm's performance in practice.
3. Amortized Analysis:
   Amortized Analysis considers the running time for a sequence of operations rather than worst-case analysis focusing on each operation.

   - Eg. In stack, we push in for O(1), pop all for O(n) in worst case.
   For worst-case analysis, its running time should be O(n);
   For average-case analysis, calculate the expectation of it;
   For amortized analysis, O(1).

6.Asymptotic Analysis

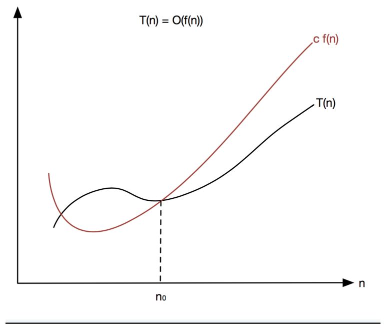


Figure 3: Big-) notation

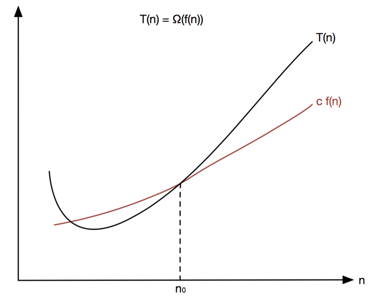


Figure 4: Big-Omega notation

1. - Upper Bound(Big-O notation):
   For a function T(n)(running time of y), f(n) is an upper bound if for "big enough n", $\exists c > 0$, $T(n) \leq cf(n)$. Here f dominates T.(Figure1)
   Eg. $T(n) = 32n^2 + 17n + 1 \Rightarrow T(n) = O(n^2)$, choose $c = 50, n = 1$

2. - Lower Bound(Big-Ω notation):
   For a function T(n)(running time of y), f(n) is a lower bound if for "big enough n", $\exists c > 0$, $T(n) \leq cf(n)$. Here f dominates g. (Figure2)
   Eg. For the same example $\Rightarrow T(n) = \Omega(n)$; $T(n) = \Omega(n^2)$, choose $c = 31, n \geq 1$

3. -Tight Bound(Big-Θ notation):
   If f(n) is both upper bound and lower bound of T(n) [with different c], we say f(n) is a tight bound for T(n). That is, $\exists c_1 > 0, c_2 > 0$, $c_1 f(n) \leq T(n) \leq c_2 f(n)$.(Figure3)
   Eg. For the same example $\Rightarrow T(n) = \Theta(n^2)$

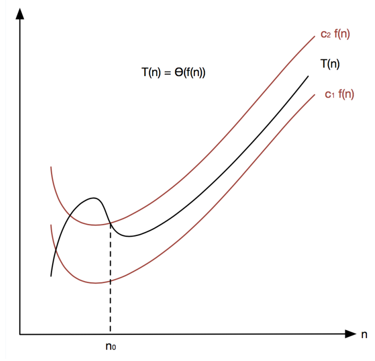


Figure 5: Big-Theta notation

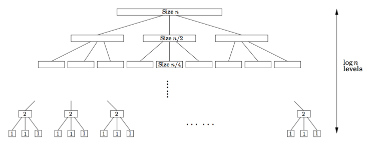


Figure 6: Divide and Conquer

7. Rules of Thumb

1. $T(n) = a_0 + a_1 n + \cdots + a_d n^d \Rightarrow T(n) = O(n^d)$
2. $O(log_a^n) = O(log_b^n)$, $a, b > 0$
3. For every $r > 1, d > 0 \Rightarrow n^d = O(r^n)$

8. Examples

1. O(n) - Linear
   Assumption: We can do constant time work for each number for i in the set $((O(1)))$
   Notion: Modern computers have parallel computation ability. for example, all log could be computed at once.
2. $O(nlog_2 n)$ - Divide and Conquer algorithms (figure4)
   Divide the problem into a number of subproblems that are smaller instances of the same problem.
   Conquer the subproblems by solving them recursively. Combine the solutions to the subproblems into the solution for the original problem. Eg.Sorting: each level: $log_2 n$, n levels $\Rightarrow n(log_2 n + 1) \Rightarrow O(nlog_2 n)$
3. $O(n^2)$ - Quadratic
   Eg. Bubble sort, quick sort $\Rightarrow$ Worst:$O(n^2)$; Average: $O(nlogn)$
4. $O(n^3)$ - Cubic time
   Eg. Count the number of triangles in graphs
5. $O(logn)$
   Eg. Searching sorted array; Binary Tree Search(Balance Tree)

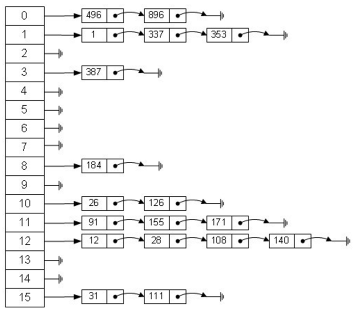


Figure 7: Hash Table.

9.Hash table

- It is a data structure used to implement an associative array, a structure that can map keys to values. It adopts hash function to compute an index into an array of buckets or slots with values. (Figure5)
- Hash collisions are unavoidable in practice because hashing a random subset of a large set of possible keys often occurs.
- Eg. Query problem:
  - Sorting $\Rightarrow O(nlogn)$
  - Hash Table $\Rightarrow O(n)$