**ASYMPTOTIC NOTATIONS**

The Process of Algorithm Development

* Design ---> for 321 course
  + divide&conquer, greedy, dynamic programming
* Validation ---> for 321 course
  + check whether algorithm is correct
* Analysis
  + determine the properties (space & time) of algorithm
* Implementation
* Testing
  + check whether implementation works for all possible cases

Analysis investigates “What are the properties of the algorithm in terms of time and space?”.

Properties of an algorithm:

* Used memory space
  + Number of bits S(n)
  + Number of elements
* Running time
  + Wall-clock time in seconds
  + Number of operations
  + Number of most important operation T(n)
    - basic operation

They are investigated as a function T(n) of a parameter n indicating problem’s size.

n : input T(n) : running time

The properties can be calculated:

* Emprically 🡪 after implementation
* Theoretically 🡪 before implementation

If there are many algorithm ideas, it is better to evaluate them without implementation.

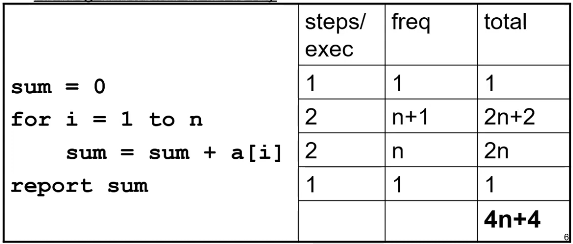
* Exact calculation is not easy

Analysis of Algorithm

**Table Method**

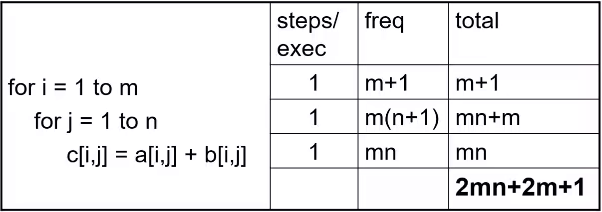
Used to calculate complexity of an algorithm.

Example: adding elements of an array



T(n) = 4n + 4 (number of operations) line2 🡪 increment i and check if it is less than n

Example: matrix addition (assume a, b, c are mxn matrices)



Constant terms are not so important. We assumed steps are 1.

T(n, m) = 2mn + 2m + 1

Analysis investigates:

* What are the properties of the algorithm in terms of time and space?
* How good is the algorithm? Whether it satisfies our needs or not?
* How it compares with others?
  + not always exact so we use asymptotical notations
* Is it the best that can be done? 🡪 not our concern for this course

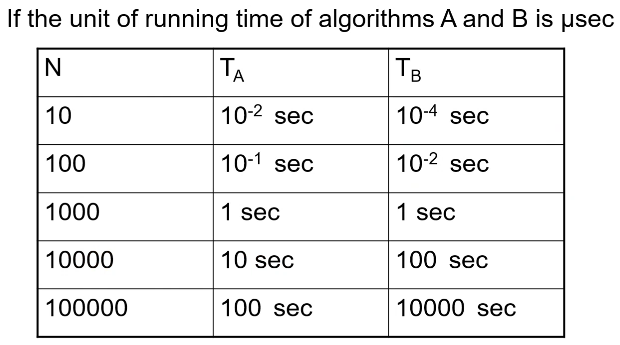
Comparison of algorithms:

Assume the running times of 2 algorithms are calculated:

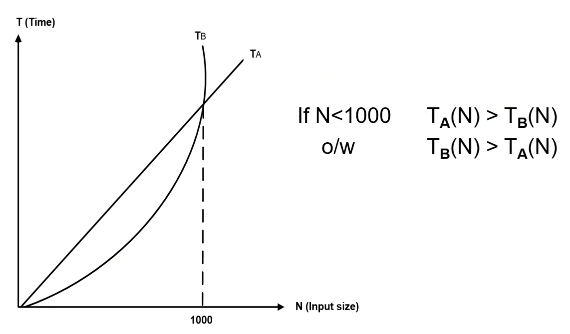
For input size N:

Running time of algorithm A = T(N) = 1000N

Running time of algorithm B = T(N) = N^2



Speed depends on the input size. It is not important for small inputs but for bigger inputs, difference is obvious.

We compare their relative growth. We are gonna do this with asymptotic notations.

Growth rate of quadratic function is faster.

Is it always possible to calculate exact value in seconds? 🡪 NO

Run-time is affected by

* compiler
* OS usually affects the constants & lower order terms, order of the function stays
* computer same
* algorithm 🡪 use asymptotic notation, affects order of the function

We said constant terms are not so important because run-time is affected by too many factors.

We are comparing algorithms, not the architecture etc. To be able to compare the algorithms, we assume a theoretical machine. In that machine all operations take constant time, machine has infinite memory, and we count the number of steps for that machine.

To be able to concentrate on the algorithm only, not considering the complexity of all compiler, OS, etc. we use asymptotic notations. Low order terms and constant multipliers have no meaning.

We compare the relative growth. All asymptotic notations consider relative growth.

Is it always possible to have definite results? 🡪 NO

The running times of algorithms can change because of the platform, the properties of the computer, etc.

We use asymptotic notations (O, Ω, θ, o) to express running time and memory space used:

* compare relative growth
  + No constants
  + No lower order terms
* compare only algorithms - larger problems

For input size N

Running time of Alg. A = T(N) = 1000N = O(N)

Running time of Alg. B = T(N) = 7(N^2) + N = O(N^2)

Big O Notation (O)

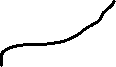
Provides an “upper bound” for the function f

T(N) = O(f(N)) if there are positive constants c and n0 such that: { f(N) is another function }

cf(N)

T(N)

 T(N) <= cf(N) for large N



* T(N) grows no faster than f(N)



* growth rate of T(N) is less than or equal to growth rate of f(N) for large N



* f(N) is an upper bound on T(N)
  + not fully correct

You can choose any value to the right as n0.

n0 always integer, you can only have exact amount of input.

Analysis of Algorithm A

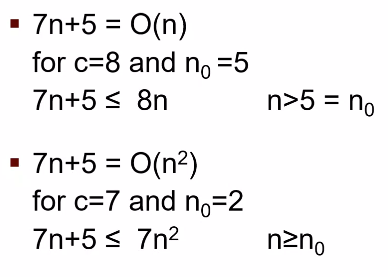
T(N) = 1000N = O(N)

1000N <= cN 

if c = 2000 and n0 = 1 for all N

T(N) = 1000N = O(N) is right

EXAMPLES

We are finding an upeer bound so both of them can happen.

n should grow at the same rate as 7n+5

n^2 should grow faster than 7n+5 or equally.

Both holds to be true in big O notation.

If you showed it for some c and n0, then you are done. It is not necessary to be correct for all the values of c and n0. Just one is enough.

7(n^2) + 3n = O(n) 🡪 NOT CORRECT. Quadratic functions grow faster than linear functions.

7(n^2) + 3n <= cn ----> 7(n^2) <= (c-3)n ----> 7n <= c-3 ----> n <= (c-3)/7 (can’t be true always, n always grows and c is constant)

Advantages of O Notation

While comparing 2 algorithms based on their running times

* Constants can be ignored
  + Units are not important
    - O(7(n^2)) = O(n^2)
* Lower order terms are ignored
  + Compare relative growth only
    - O(n^3 + 7(n^2) + 3) = O(n^3)

A 🡪 T(N) = 1000N = O(N)

B 🡪 T(N) = N^2 = O(N^2)

A is asymptotically faster than B.

We said asymptotically because for small values, B is faster.

Omega Notation (Ω)

T(N) = Ω(f(N)) if there are positive constants c and n0 such that;

T(N) >= cf(N) when N >= n0

* T(N) grows no slower than f(N)
* growth rate of T(N) is greater than or equal to growth rate of f(N) for large N
* f(N) is a lower bound on T(N)
  + not fully correct

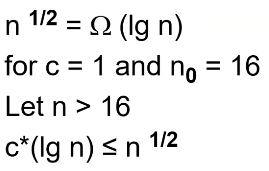
T(N)

cf(N)

Small values are not important for us.



Example:





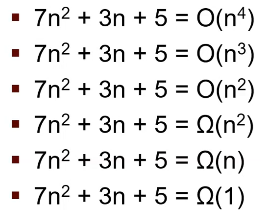
Theorem:

f(N) = O(g(n)) ⬄ g(n) = Ω(f(N))

Proof:

f(N) <= c1g(n) ⬄ g(n) >= c2f(N) divide the left side with c1

(1/c1)f(N) <= g(n) ⬄ g(n) >= c2f(N) if we choose c2 as 1/c1 then theorem is right



---> faster than 7n^2 + 3n + 5

---> faster than func.

---> same rate

all of them are correct

---> same rate

---> slower than func.

---> slower than func.

What they tell us are different.

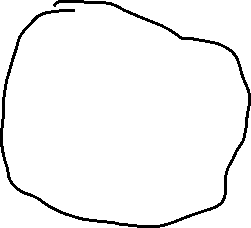
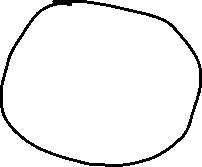
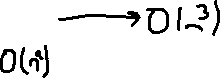
O(n^2) gives more information than O(n^3) and O(n^4).

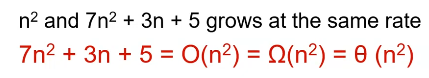
O(n^2) says function grows faster or equal way with n^2.

Ω(n^2) gives more information than Ω(n) and Ω(1).

Ω(n^2) says function grows slower or equal way with n^2.

If both blue sentences are true, growth rate should be equal.





Theta notation gives a lot more information.

🡪 Running time cannot be smaller than constant, it cannot be negative.

Theta Notation (θ)

T(N) = θ(h(N)) ⬄ T(N) = O(h(N)) and T(N) = Ω(h(N))

c1g(n) <= f(n) <= c2g(n) for all n >= n0 🡪 g(n) : thetanın içi , f(n) : genel fonksiyon

* T(N) grows as fast as h(N)
* growth rate of T(N) and h(N) are equal for large N
* h(N) is a tight bound on T(N)
  + not fully correct

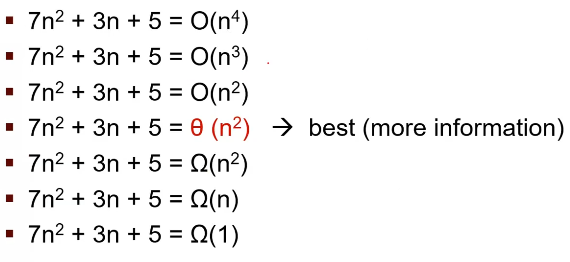
c1h(N)

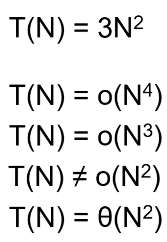
T(N)

Small values are not important for us.



c2h(N)



Little o Notation (o)

T(N) = o(p(N)) if T(N) = O(p(N)) and T(N) != θ(p(N))

* p(N) grows strictly faster (not equal) than T(N)
* growth rate of T(N) is less than the growth rate of p(N) for large N
* p(N) is an upperbound on T(N) but not tight
  + not fully correct

**SOME RULES**

RULE1:

* if T1(N) = O(f(N)) and T2(N) = O(g(N)) then
  + T1(N) + T2(N) = max( O(f(N)), O(g(N)) )
  + T1(N) \* T2(N) = O(f(N) \* g(N))

[ T1(N) = O(N^2) + T2(N) = O(N) ] = O(N^2 + N) = O(N^2) {we don’t write low order term}

[ T1(N) = O(N^2) \* T2(N) = O(N) ] = O(N^3)

We use summation to sum 2 seperate algorithms of the program.

We use multiplication if algorithm (T1(N)) is repeated several times for example in a loop and loop iterates T2(N) times.

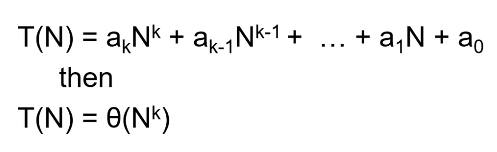
Rule 1 is also true for both θ and Ω notations.

[ T1(N) = O(N^2) \* T2(N) = θ(N) ] = O(N^3)

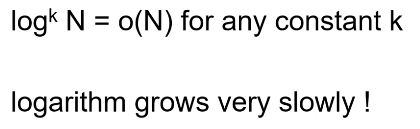
not tight tight result must be not tight

RULE 2:

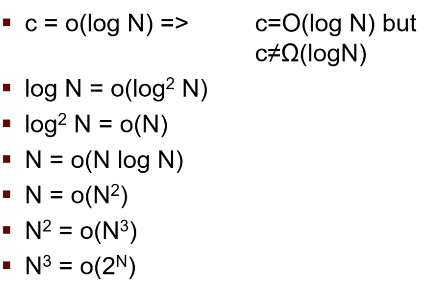
* if T(N) is a polynomial of degree k



RULE 3:

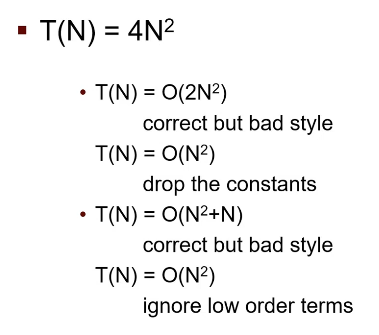


SOME COMMON FUNCTIONS

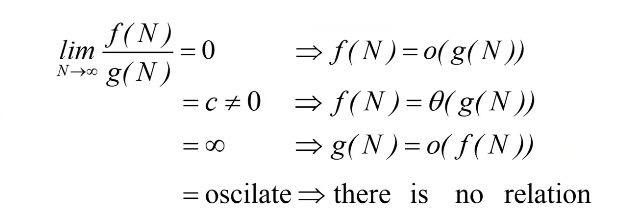
Logarithmic functions grow faster than constant functions.

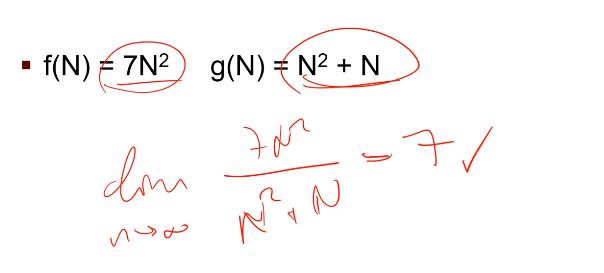
…

Example:



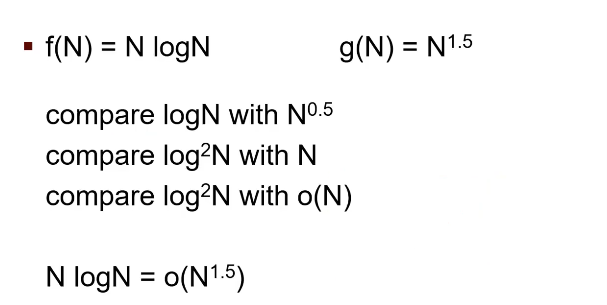
Another Way to Compute Growth Rates:



Example:

Growth rates are same.

Example:





GENERAL RULES TO ANALYSIS WITH ASYMPTOTIC NOTATIONS

RULE 1: FOR LOOPS

The running time of a for loop is at most the running time of the statements in the for loop times the number of iterations.

Diagram, text

Description automatically generatedLoopun içine bak, T(N)’li running time bul.

Loopa bak, kaç kere çalıştığını (T(N)) bul.

İkisini çarp.

In this example, each iteration has print and increment statements. Both takes constant time. So inside of the loop takes constant times 🡪 θ(1)

Number of time the loop is executed is n. So 🡪 θ(n)

Mutiply these two and we get 🡪 T(n) = θ(n)

RULE 2: NESTED LOOPS

Analyze nested loops inside out.

Text, letter

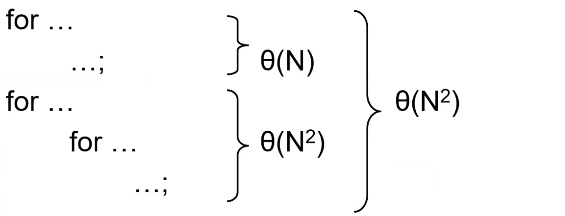
Description automatically generated



T(n) = θ(r\*q)

RULE 3: CONSEQUTIVE STATEMENTS

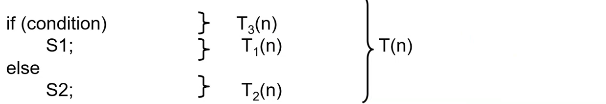
Add the running times.



T(N) = θ(N) + θ(N^2) = θ(N^2)

Defining variable and returning sth. is constant.

RULE 4: IF / ELSE



Sometimes if condition is true 🡪 T1(n) + T3(n)

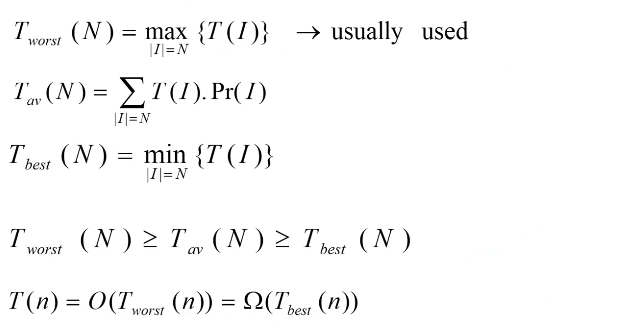
Sometimes if condition is false 🡪 T2(n) + T3(n)

Running time is not fixed.

Running time is never more than running time of the test (T3(n)) plus larger of the running times of S1 and S2. May overestimate but never underestimates.



T(n) = O(T3(n) + max(T1(n), T2(n))) 🡪 worst case analysis



Pr : probability

For the if/else example former page:

* Tw(n) = T3(n) + max(T1(n), T2(n))
* Tb(n) = T3(n) + min(T1(n), T2(n))
* Tav(n) = p(T)T1(n) + p(F)T2(n) + T3(n)
  + p(T) 🡪 p(condition = True)
  + p(F) 🡪 p(condition = False)

probability of false is (1 - probability of true)

If T1(n) = o(T2(n)), means T2(n) grows faster, so max is T2(n) and min is T1(n).

Then,

* T(n) = O(T3(n) + T2(n)) = Ω(T3(n) + T1(n))

If T1(n) = θ(n^2) , T2(n) = θ(n) , T3(n) = θ(n) , then;

* Tw(n) = T3(n) + max(T1(n), T2(n)) = θ(n^2)
* Tb(n) = T3(n) + min(T1(n), T2(n)) = θ(n)

p(T) = 1/n p(F) = (n-1)/n olsun

(1/n)\*θ(n^2) + ((n-1)/n)\*θ(n) + θ(n)

θ(n) + θ(n) + θ(n) = θ(n)

* if p(T) = p(F) = ½
  + Tav(n) = p(T)T1(n) + p(F)T2(n) + T3(n) = θ(n^2)
* T(n) = O(n^2) = θ(n)

RECURSIVE CALLS

Algorithm for computing factorial

Graphical user interface, text, application

Description automatically generated



Text

Description automatically generated with low confidence

Text, letter

Description automatically generated



BOOK SLIDES FOR SAME TOPIC

**ALGORITHM EFFICIENCY AND BIG-O**

Getting a precise measure of the performance of an algorithm is difficult.

Big-O notation expresses the performance of an algorithm as a function of the number of items to be processed.

This permits algorithms to be compared for efficiency.

For more than a certain number of data items, some problems cannot be solved by any computer.

Linear Growth Rate

If processing time increases in proportion to the number of inputs n, the algorithm grows at a linear rate

Text

Description automatically generated



Problem büyüklüğü 🡪 arrayin eleman sayısı (n diyelim) + target 🡪 n + 1

Constants çok önemli değil o yüzden n diyebilirsin.

If statement’ı ister true ister false olsun, hep θ(1) zaman alır.

for döngüsü her zaman n kere dönmüyor, bazen return edip çıkıyor.

* T2best(n) = θ(1)

linear (worst)

T2 yeşilde bir yerlerde

* T2worst(n) = θ(n)
* T2(N) = O(n) = Ω(1)

constant (best)



T(n) = T1(n) \* T2(n) + θ(1) { θ(1) returnden geldi}

T(n) = θ(1) \* O(n) + θ(1) = O(n) + θ(1) = O(n) 🡪 worst case’den dolayı

T(n) = Ω(1) 🡪 best case’den dolayı (söylemenin mantığı yok, bütün algoritmalar Ω(1)’dir)

Tb(n) = θ(1) \* θ(1) + θ(1) = θ(1)

Tw(n) = θ(1) \* θ(n) + θ(1) = θ(n)

Eğer target arrayde varsa, her bir elementte olma ihtimali 1/n’dir. 1. elementteyse döngü 1 kere, 2. elementteyse döngü 2 kere, … , n. elementteyse döngü n kere döner.

döngünün ortalama dönme sayısı

(1/n)\*1 + (1/n)\*2 + (1/n)\*3 + (1/n)\*n = (1/n)(1+2+…+n) = (1/n)(n\*(n+1)/2) = (n+1)/2 🡪



probability



for’ün dönme sayısı

Tav(n) = θ(n) 🡪 (n+1)/2 θ(n)’dir. Ortalama değer olduğu için θ oldu. O(n) de diyebilirdin. Yanlış olmazdı ama eksik olurdu.

🡪 Çalışma süresi negative olamayacağından bütün algoritmalar en genel haliyle Ω(1)’dir.

🡪 θ(1) = θ(2) = θ(1000) = …

n x m Growth Rate

Processing time can be dependent on 2 different inputs.

Text

Description automatically generated with medium confidence



T(n+m) ya da T(n, m) diyebilirsin. Genelde T(n, m) deriz.

T(n, m) = T1(n)\*T2(m) + θ(1)

search metodu 🡪 Tw(m) = θ(m) {linear} , Tb(m) = θ(1) {constant}

T1w(m) = θ(m) - T1b(m) = θ(1)

Tb(n, m) = θ(1) \* θ(1) + θ(1) = θ(1)

Tw(n, m) = θ(n) \* θ(m) + θ(1) = θ(n\*m)

T(n, m) = O(n \* m)

condition’ın best case’i



for’un best case’i



returnden

The for loop will execute x.length times

But it will call search, which will execute y.length times

The total execution time is proportional to (x.length \* y.length)

The growth rate has an order of n x m or O(n\*m)

Quadratic Growth Rate

If processing time is proportional to the square of the number of inputs n, the algorithm grows at a quadratic rate.

Graphical user interface, text

Description automatically generated



T(n)’i bulacağız. 🡪 T(n) = T1(n) \* T2(n)

O(1) ile θ(1) farklı değildir.

T1b(n) = θ(1) 🡪 iç for en az 2 kere çalışabilir, θ(2) de diyebilirsin o da θ(1) ile aynı

T1w(n) = θ(n) 🡪 T1’in en kötü halde artış hızı kesin n’dir

T2b(n) = θ(1)

T2w(n) = θ(n)

Tb(n) = θ(1)

Tw(n) = θ(n^2)

T(n) = O(n^2) 🡪 T’nin artış hızı n^2’ye eşit veya daha küçüktür

The for loop with i as index will execute x.length times.

The for loop with j as index will execute x.length times.

The total number of times the inner loop will execute is (x.length)^2

The growth rate has an order of n^2 or O(n^2)

The O() in the previous examples can be thought of as an abbreviation of “order of magnitude” is small or equal

A simple way to determine the big-O notation of an algorithm is to look at the loops and to see whether the loops are nested

Assuming a loop body consists only of simple statements,

* a single loop is O(n)

Hoca çok sevmiyor.

* a pair of nested loops is O(n^2)
* a nested pair of loops inside another is O(n^3)
* and so on …

You must also examine the number of times a loop is executed

for(i = 1; i < x.length; i \*= 2) {

// Do something with x[i]

}

The loop body will execute k-1 times, with i having the following values:

1, 2, 4, 8, 16, …, 2^k until 2^k is greater than x.length

Since 2^(k-1) = x.length < 2^k and  is k, we know that k-1 =  < k

Thus we say the loop is O(logn) (in analyzing algorithms, we use logarithms to the base 2)

Logarithmic functions grow slowly as the number of data items n increases

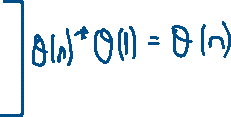
------------------------------------------------------------------------------------------------------------------------------------------

Graphical user interface, application

Description automatically generated with medium confidence

S(n) = {# of simple statement executed} = n^2 + 5n + 25 -----> θ(n^2)

T(n) = θ(n^2)



Shape

Description automatically generatedfor (int i = 0; i < n; i++){



for (int j = i; j < n; j++) {

*Simple statement*



}

}

n + (n-1) + (n-2) + … + 2 + 1 = (n(n+1)) / 2 -----> 2 döngünün dönme sayısı ----> θ(n^2)

A picture containing text

Description automatically generated

T(n) = 3(n-1) + 3(n-2) + … + 3

Factoring out the 3:

3(n-1 + n-2 + n-3 + … + 1)

1 + 2 + … + n-1 = (n\*(n-1)) / 2

T(n) = 3 ((n\*(n-1)) / 2) = 1.5n^2 - 1.5n

When n=0, the polynomial has the value 0 for values of n > 1, 1.5n^2 > 1.5n^2-1.5n

Therefore T(n) is O(n^2) when n0 is 1 and c is 1.5

Chart, line chart

Description automatically generated

Symbols Used in Quantifying Performance

Text

Description automatically generated with medium confidence

Common Growth Rates

Table

Description automatically generatedBüyüme hızı aşağı indikçe artar.

Chart

Description automatically generated

Effects of Different Growth Rates

Table

Description automatically generated

Algorithms with Exponential and Factorial Growth Rates

They have an effective practical limit on the size of the problem they can be used to solve

With an O(2^n) algorithm, if 100 inputs takes an hour then,

* 101 inputs will take 2 hours
* 105 inputs will take 32 hours
* 114 inputs will take 16384 hours (almost 2 years)

Encryption algorithms take advantage of this characteristic

Some cryptographic algorithms can be broken in O(2^n) time, where n is the number of bits in the key

A key length of 40 is considered breakable by a modern computer, but a key length of 100 bits will take a billion billion (10^18) times longer than a key length of 40

Performance of KWArrayList

The set and get methods execute in constant time: O(1)

Inserting or removing general elements is linear time: O(N)

Adding at the end is (usually) constant time:

* With our reallocation technique the average (amortized) is O(1)
* The worst case is O(n) because of reallocation

constructor 🡪 θ(1)

add()

* Sonda boş yer varsa: θ(1)
* Boş yer yoksa kopyalama sebebiyle: θ(n)
* Tamortized(n) = θ(1)
  + Reallocation ile 2 katına çıkarttığımız ve 2 katına çıkarttıktan sonra tekrar 2 katına çıkartmak için yapılan addlere paylaştırdığımız zaman amortized running time’ı oluştururuz.

add with index

* Tb(n) = θ(1) (en sona eklemek ve sonda yer olması)
* Tw(n) = θ(n) (en başa eklemek, tüm array shift edilir)
* T(n) = O(n)

set and get 🡪 θ(1)

remove

* Tw(n) = θ(n) ----> ilk elemanı silmek
* Tb(n) = θ(1) ----> son elemanı silmek
* T(n) = O(n)

reallocate

k elemana gelince reallocate yaptım, k zaman harcadım.

2k elemana gelince reallocate yaptım, 2k zaman harcadım.

4k elemana gelince reallocate yaptım, 4k zaman harcadım.

Harcadığım k zamanı, k’dan 2k’ya kadar geçen zamana; 2k zamanı, 2k’dan 4k’ya kadar geçen zamana… paylaştırdığımız zaman her biri constant time olmuş oluyor ve aslında AMORTIZED CONSTANT TIME diyebiliriz.



Genelde theta notasyonunu kullanacağız çünkü en fazla detayı o veriyor.

Theta cinsinden analiz yapamazsak worst case için O, best case için omega notasyonuna geçeceğiz.

Sadece bir yerlerde karşılaşırsan diye “o” aklında bulunsun.