# Асимптотска нотација

Асимптотската нотација претставува едноставна карактеризација на ефикасноста на алгоримот и овозможува споредба помеѓу перформансите на алтернативните алгоритми. Иако понекогаш може прецизно да се утврди рантајмот на алгоритмот, дополнителна прецизност не е неопходна. Големо „О“( Big O notation), заедно со Big Theta и Big Omega се најчесто употребувани асимптотски нoтaции( или во литература уште познати како Бахман-Ландау нотации).

### Big Theta

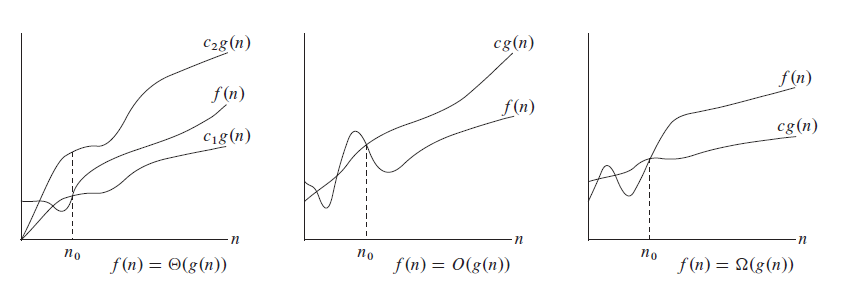
Да земеме дека имаме функција , со ги обележуваме множеството од функции:

### Big O

Да земеме дека имаме функција , со ги обележуваме множеството од функции:

### Big Omega

Да земеме дека имаме функција , со ги обележуваме множеството од функции:



## Најдобар, најлош и просечен случај(best, worst and average case)

Најдобар, најлош и просечен случај за даден алгоритам ја покажуваат употребата на ресурсот во респективните случаи. Најчесто ресурсот е време, но може да биде меморија или нешто друго.

Најдобриот случај го опишува алгоритмот со однесување под оптимални услови. На пример, најдобар случај за линеарно пребарување би било кога саканиот елемент се наоѓа на првата позиција во листата. Употребата и развојот на алгоритмите генерално не се заснова врз најдобриот случај бидејќи влезот што го дава ретко се јавува во праксата.

Најлошиот случај е особено важен во ситуации каде што времето е критично(одредени машини, авиони итн.). Со случајот со линеарно пребарување тоа би значело бараниот елемент да биде последниот елемент во низата.

# Временска комплексност

Временска комплексност го квантифицира времето кое е потребно за еден алгоритам да изврши во однос на бројот на елементите на влезот.[[1]](#footnote-1) За обележување се користи асимптотската нотација.

|  |  |  |
| --- | --- | --- |
| Име |  | Пример |
| константно време |  | Одредување дали бројот е парен или непарен |
| инверзна Акерман функција |  | Операции во дисјунктен сет |
| логаритамска |  | бинарно пребарување |
| линеарна |  | наоѓање на најмал елемент во низа |
| линеарно-аритметичка |  | сортирање на елементи во алгоритам со споредување |
| квадратна | ) | Bubble sort, Insertion sort |
| кубна | ) | множење на две матрици |
| експоненцијална |  | множење на повеќе матрици |
| факторијална |  | проблем со трговскиот патник |

# Просторна комплексност

Поимот просторна комплексност(space complexity) често се употребува онаму каде што треба да се употреби дополнителен простор(auxiliary space).

Auxiliary space e дополнителниот простор кој го употребува алгоритмот.

Space complexity е целосниот простор што го зазема алгоритмот во однос на влезната големина. Просторната комплексност го вклучува влезот и дополнителниот простор.

На пример, за споредување на стандардните алгоритми за сортирање, auxiliary space е подобар критериум отколку просторната комплексност. Merge sort има auxiliary space, insertion sort , a просторната комплексност е за двата алгоритми.[[2]](#footnote-2)

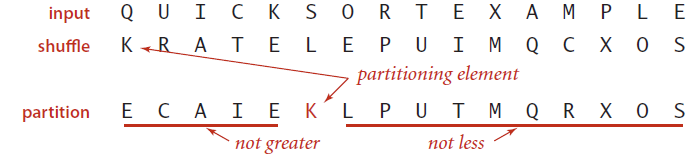
# Quicksort

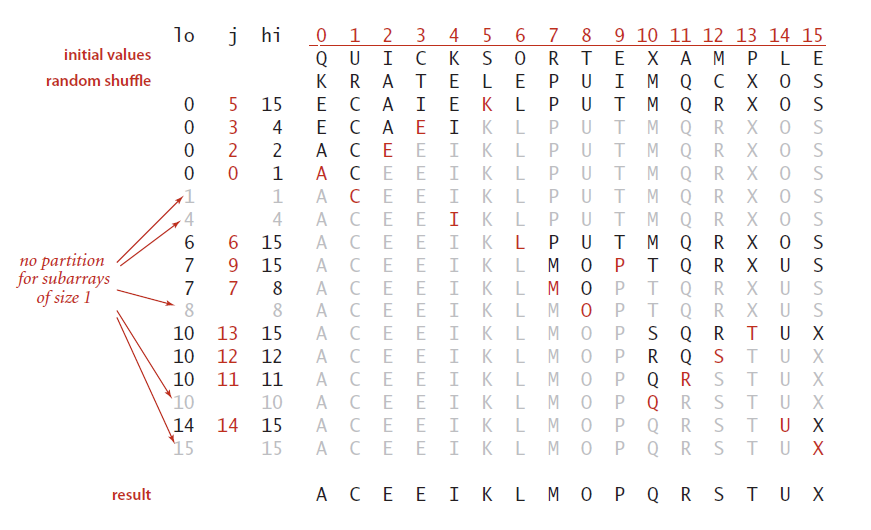
Quicksort се смета за еден од најважните 10 алгоритми во 20-тиот век.[[3]](#footnote-3) Релативно лесната имплементација и average case performance од O()) се заслужни за неговата популарност. Quicksort е раздели-па-владеј алгоритам. Се заснова на делење на низата на две поднизи, и потоа независно нивно сортирање. Во основа има три чекори:

1. Одбирање на елемент од низата која сакаме да ја сортираме, ко се нарекува pivot
2. Partitioning кој се заснова на:

* Елементот a[j] се наоѓа на својата конечна позиција, за некој j
* Ниту еден елемент од a[lo] до a[j-1] е поголем од a[j]
* Ниту еден елемент од a[j+1] до a[hi] е помал од a[j]

1. Рекурзивна применување на првите 2 чекори.





Во нашата имплементација тоа е одбегнато и има , бидејќи замена настапува и кога елементите се еднакви на пивотот. Затоа:

**B A A B A B B B C C C**

Постои и пожелно решение, а тоа е сите елементи еднакви на пивотот да бидат подредени, односно:

**A A A B B B B B C C C**

## Класичен Quicksort

**public** **static** **void** quicksort(Comparable [] array, **int** low, **int** high)

{

**if**(high <= low)

**return**;

**int** j = *partitioning*(array, low, high);

*quicksort*(array, low, j - 1);

*quicksort*(array, j + 1, high);

}

**private** **static** **int** partitioning(Comparable[] array, **int** low, **int** high)

{

**int** i = low, j = high + 1;

**while**(**true**)

{

**while**(*lessOrEquals*(array[++i], array[low]))

{

**if**(i == high)

**break**;

}

**while**(*lessOrEquals*(array[low], array[--j]))

{

**if**(j == low)

**break**;

}

**if**(i >= j)

**break**;

*exch*(array, i, j);

}

*exch*(array, low, j);

**return** j;

}

**public** **static** **boolean** less(Comparable<Comparable> v, Comparable w)

{

**return** (v.compareTo(w) < 0);

}

**public** **static** **boolean** lessOrEquals(Comparable<Comparable> v, Comparable w)

{

**return** (v.compareTo(w) <= 0);

}

**public** **static** **void** exch(Comparable[] a, **int** i, **int** j)

{

Comparable swap = a[i];

a[i] = a[j];

a[j] = swap;

}

Проблемот со оваа класична имплементации на Quicksort е што за низа со еден ист елемент, има . Тоа е така бидејќи елементите што се еднакви на пивотот се ставаат на една иста страна:

**B A A B A B B B C C C**

## Quicksort – Sedgewick’s partitioning

Класичната имплементација, била дел од скоро сите библиотеки на познатите програмски јазици и покрај гореспоменатите недостатоци. Разликата помеѓу класичниот начин на поделба и оној на Sedgewick е минимална, односно настанува промена на елементите и кога елементите се еднакви на пивотот.

**private** **static** **int** partitioning(Comparable[] array, **int** low, **int** high)

{

**int** i = low, j = high + 1;

**while**(**true**)

{

**while**(**less**(array[++i], array[low]))

{

**if**(i == high)

**break**;

}

**while**(***less***(array[low], array[--j]))

{

**if**(j == low)

**break**;

}

**if**(i >= j)

**break**;

*exch*(array, i, j);

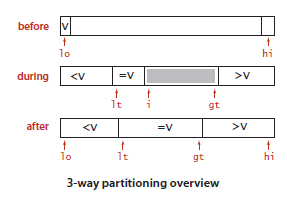
}

*exch*(array, low, j);

**return** j;

}

## Dijkstra’s partitioning



Една од основните идеи како да се подобри quicksort е поделба на низата на три дела, дел каде што сите елементи се помали од пивотот, дел каде што сите елементи се еднакви на пивотот, и дел каде што сите елементи се поголеми од пивотот. Имплементацијата на овој е покомплексна и различни можни имплементации биле сугерирани за да исполни тој услов. Еден од попознатите методи е оној на Дијкстра, исто така наречен и Холандско знаме.

Решението на Дијкстра се состои од воведување на неколку нови покажувачи. lt поинтер така што a[lo .. lт-1] е помало од v, поинтер gt така што a[gt+1, hi] е поголем од v и поинтер I така што a[lt.. I – 1] се еднакви на v и a[i..gt], чии елементи се уште не се обработени. Почнувајќи од i еднакво на lo, продолжуваме со a[i] употребувајќи споредување на три можни вредности(помало, еднакво или поголемо).

Всушност постојат три случаи:

* a[i] помало или еднакво на v: размени ги a[lt] со a[i] и зголеми ги lt и i
* a[i] поголемо од v: размена на a[i] со a[gt] и намали го gt
* a[i] еднаков на v: зголеми го i

**public** **static** **void** sort(Comparable[] array, **int** lo, **int** hi)

{

**if**(hi <= lo)

**return**;

**int** lt = lo;

**int** i = lo;

**int** gt = hi;

Comparable v = array[lo];

**while**(i <= gt)

{

**int** cmp = array[i].compareTo(v);

**if**(cmp < 0)

{

*exch*(array, lt, i);

lt++;

i++;

}

**else** **if**(cmp > 0)

{

*exch*(array,gt, i);

gt--;

}

**else**

{

i++;

}

}

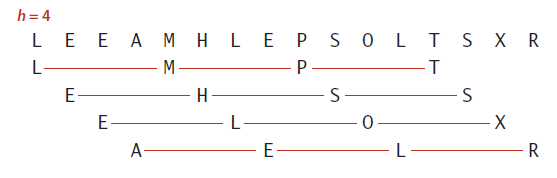
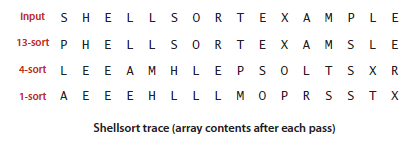
*sort*(array, lo, lt - 1);

*sort*(array, gt + 1, hi);

}

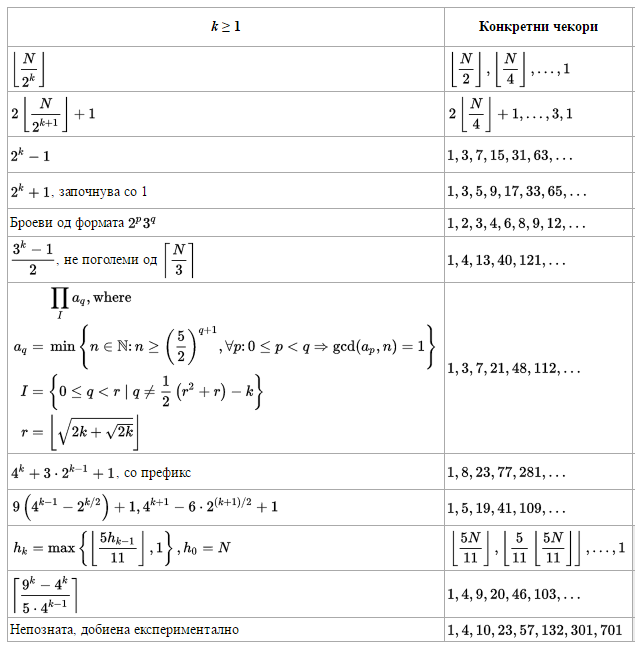
Иако овој алгоритам беше измислен кратко по класичниот quicksort, не беше често употребуван поради големиот број на размени во однос на стандардниот алгоритам, бидејќи во практичната употреба бројот на елементи што се повторуваат во низата често не е голем. Овие ограничувања ги надминува Bently-McIlroy алгоритмот .

# Shellsort

Shellsort е измислен во 1959 година од Donald Shell и е еден од најстарите алгоритми за сортирање. Функционира на начин сличен на insertion sort, меѓутоа покажува подобрени перформанси. Insertion sort е бавен за големи неподредени низи, бидејќи вклучува замена на соседните елементи, така што елементот во низата може да ја промени својата позиција само за едно место со една замена. Така, ако елементот што се наоѓа на крајот на низата треба да се најде на почетокот на низата, ќе бидат потребни N – 1 замени за елементот да се најде на правилната позиција. Shellsort е своевидно проширување на insertion sort, чии подобрени перформанси се базираат на способноста да изврши замена на елементи кој се наоѓаат далеку едни од други, создавајќи делумно сортирани низи кои потоа се сортираат со insertion sort.

Главната идеја е да се подреди низата така што земајќи го секој h-ти елементи(почнувајќи од било каде) создава сортирана подниза. Кажано на поинаков начин, h-сортирана низа е мноштво од h-сортирани поднизи. Со тоа што се с h-сортира за некои големи вредности на h, можеме да ги придвижиме елементите поблиски до нивната вистинска положба, правејќи го g-сортирањето, за g < h, полесно. Целиот алгоритам завршува со h со вредност 1, односно insertion sort.

Чекорот кој треба да се одбере, односно секвенцата, е еден од попознатите проблеми во сортирањето, бидејќи можеби постои секвенца чија ефикасност за големи низи ја надминува ефикасност на останатите алгоритми за сортирање. Значи, проблемот за оптимална секвенца останува отворено прашање. Одредени историски суштествени, како и моментални најдобро познатите математички и емпириски добиени секвенци се дадени подoлу:



Shellsort е алгоритам кои се уште има употреба во индустријата, особено во системите каде што меморијата е ограничен. Затоа, негова имплементацијата е дел од кернелот на Линукс, како и на одредени имплементации на qsort функцијата од стандардната C библиотека.

**public** **static** **void** shellSort(Comparable [] array) {

**if**(array.length == 0 || array.length == 1)

**return**;

//presmetka na brojot na cekori

**int** number\_of\_gaps = 1;

**int** number = 1;

List<Integer> gaps = **new** ArrayList<Integer>();

gaps.add(0, 1);

**while**(**true**) {

number = (**int**) (Math.*pow*(2, number\_of\_gaps) + 1);

**if**(number < array.length) {

number\_of\_gaps++;

gaps.add(number);

}

**else** {

**break**;

}

}

**for**(**int** i = gaps.size() - 1; i >= 0; i--) {

**for**(**int** j = gaps.get(i); j < array.length; j++) {

**int** current = j;

**while**( ((current - gaps.get(i)) >= 0) &&

*less*(array[current], array[current - gaps.get(i)])) {

*exch*(array, current, current - gaps.get(i));

current -= gaps.get(i);

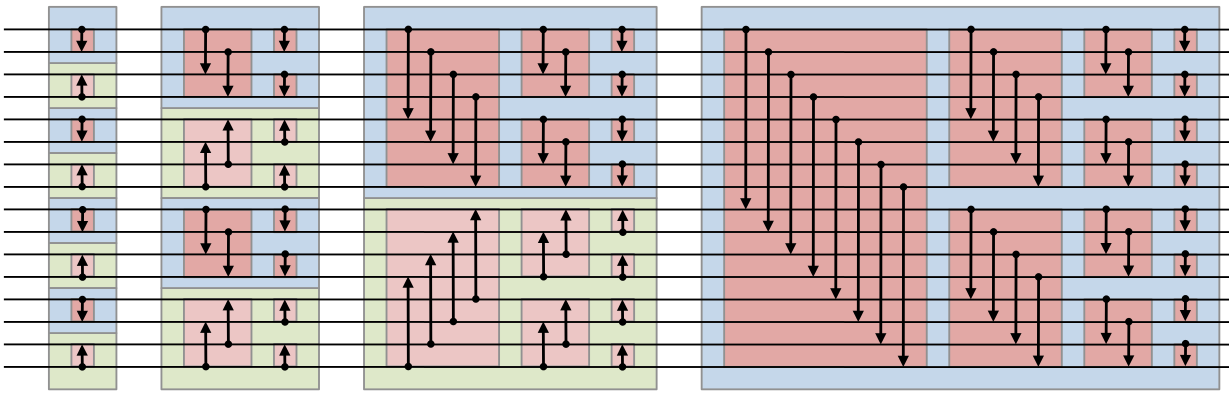
}

}

}

# Bitonic sorter

Bitonic sorter е паралелен алгоритам за сортирање. Во својата основа ја има таканаречената bitonic секвенца така што: , за



16 броеви кој се внесени на влез на левата страна, поминуваат низ 16 жици, и излегуваат од десната страна. Низата која е внесена е секогаш сортирана така што бројот што се наоѓа најдолу е најголем.

Во приказот стрелките претставуваат споредувач. Броевите кој се наоѓаат на почеток и на крајот од стрелката се споредуваат, така што врвот на стрелката секогаш покажува на поголемиот број. Ако тоа не е случај, односно ако стрелката со врвот покажува кон помал елемент, настанува промена на позицијата на елементите. Бојата на елементите нема вистинско значење, тука се поставани единствено за појасно објаснување на алгоритмот.

Секој црвена кутија има иста структура – секој влез на горната половина се споредува со елементот кој одговара од долната половина, така што сите стрелки покажуваат надолу(темно црвен) или нагоре(светло црвен).

1. [*Sipser, Michael*](https://en.wikipedia.org/wiki/Michael_Sipser)*(2006). Introduction to the Theory of Computation, стр. 226* [↑](#footnote-ref-1)
2. http://www.geeksforgeeks.org/g-fact-86/ [↑](#footnote-ref-2)
3. <http://www.uta.edu/faculty/rcli/TopTen/topten.pdf> - јануари, 2017 [↑](#footnote-ref-3)