Algorithms Complexity

Fabrizio Ferrandi

Contacts

Luca Ezio Pozzoni

lucaezio.pozzoni@polimi.it

Algorithm as a recipe

Ingredients:

- spaghetti:450g (1 pound)
- bacon: 225g (¼ pound)
- egg yolks: 5
- Pecorino or Parmigiano-Reggiano cheese, grated: 360ml (1½ cups)
- olive oil, extra-virgin: 3-4 tablespoons
- pepper, freshly ground: ½ tablespoon
- Salt

Utensils:

- large pot
- large skillet
- bowl
- measuring cups and spoons
- fork

Procedure

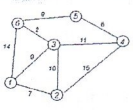
1. Dice the bacon into small pieces (1 inch [2.5cm] will do)
2. The Pot: Bring a big pot of water to a boil and add salt when it begins to simmer
3. The Pot: Cook the spaghetti until it is al dente and drain it, reserving ½ cup (118 ml) of water
4. The Skillet: As spaghetti is cooking, heat the olive oil in a large skillet over a medium-high heat. When the oil is hot, add the pancetta and cook for about 10 minutes over a low flame until the pancetta has rendered most of its fat but is still chewy and barely browned
5. The Bowl: In a bowl, slowly whisk about ½ cup of the pasta water into the egg yolks, using a fork. Add the Parmesan cheese and pepper. Mix with a fork.
6. The Skillet: Transfer the spaghetti immediately to the skillet with the pancetta. Toss it and turn off the heat. Add the egg mixture to the skillet with the pasta and toss all the ingredients to coat the pasta. Taste the pasta and add salt and black pepper, if necessary.

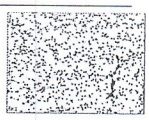
Algorithm Complexity

- How long it takes to prepare *carbonara* for 2 persons?
  - How long for 10?
    - How long for 100?

Time + spatial complexity

Shortest path computation





Is there any algorithm computing the shortest path between two vertices in a graph? How long does it take?

Analysis of algorithms

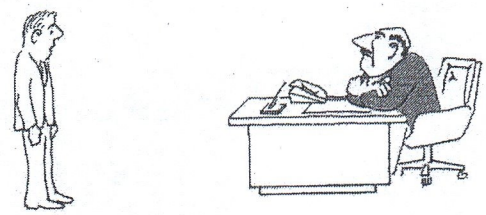
The theoretical study of computer-program performance and resource usage

What's usually more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability

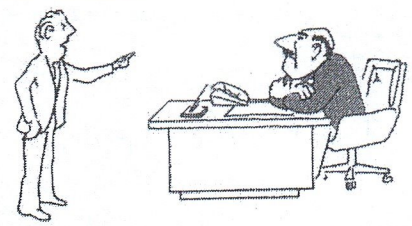
Why study algorithms and performance?

- Algorithms analysis help us to understand *scalability*
- Performance often draws the line between what is feasible and what is impossible
- Algorithmic mathematics provides a *language* for talking about program behavior
- Performance is the *currency* of computing
- The lessons of program performance generalize to other computing resources
- Speed is fun!



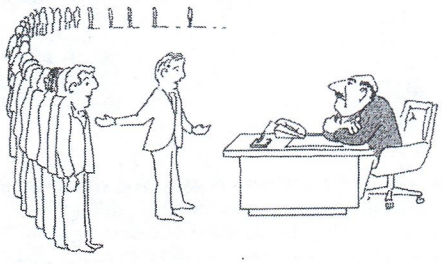
"I can't find an efficient algorithm, I guess I'm just too dumb."

Comic from Computers and *Intractability*, Garey and Johnson



"I can't find an efficient algorithm, because no such algorithm is possible!"

Comic from Computers and *Intractability*, Garey and Johnson



"I can't find an efficient algorithm, but neither can all these famous people."

Comic from Computers and *Intractability*, Garey and Johnson

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

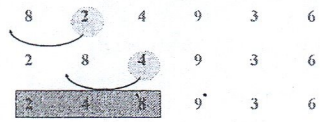
2 8 4 9 3 6

Example of insertion sort

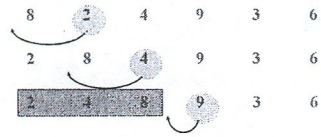
8 2 4 9 3 6

2 8 4 9 3 6

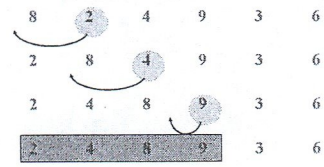
Example of insertion sort



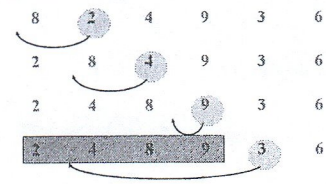
Example of insertion sort



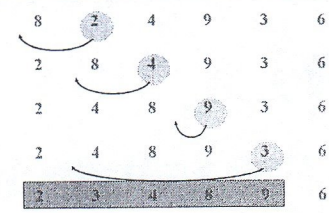
Example of insertion sort



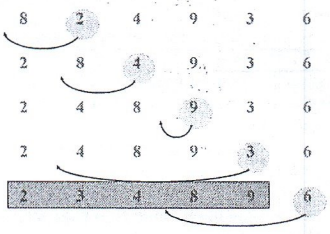
Example of insertion sort



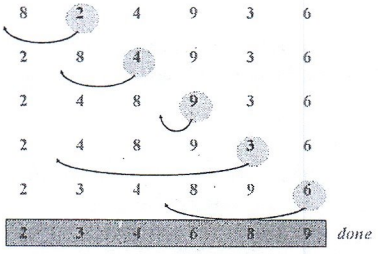
Example of insertion sort



Example of insertion sort

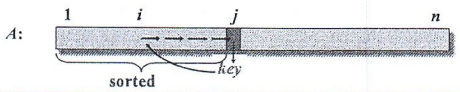


Example of insertion sort



Insertion sort

INSERTION-SORT(A, n) ▷ $A[1..n]$
for $j \leftarrow 2$ to n do
  $key \leftarrow A[j]$
  $i \leftarrow j-1$
  while $i > 0$ and $A[i] > key$ do
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i-1$
  $A[i+1] = key$



Insertion sort C++ implementation

```
void insertion_sort(std::vector<int> &A)
{
    int i;    C
    int j;    C
    int key;  C
    for(j=1; j<A.size(); j++)
    {
       key = A[j];   C
       i = j-1;      C
       while(i>=0 && A[i]>key)
       {
          A[i+1] = A[i];   C
          i = i-1;         C
       }
       A[i+1] = key;       C
    }
} // http://ideone.com/siE9V
```

A
B

We have two cycles: A, B.
How many times we'll perform A?
A.size() times, which is n.
How many times we'll go through the B loop?
It depends because we have to analyze
$i \geq 0$ and $A[i] > key$. In the worst case
scenario we'll run it $1 + 2 + \dots + (n-1)$, which
is of the magnitude of n.
⟹ Worst case: $\Theta(n^2)$.

C = constant time for that operation

Running time

- The running time depends on the input: an already sorted sequence is easier to sort
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee

Kinds of analyses

Worst-case: (usually)
- $T(n)$ = maximum time of algorithm on any input of size n

Average-case: (sometimes)
- $T(n)$ = expected time of algorithm over all inputs of size n
- Need assumption of statistical distribution of inputs

Best-case: (bogus)
- Cheat with a slow algorithm that works fast on *some* input

Machine-independent time

What is insertion sort's worst-case time?
- It depends on the speed of our computer

BIG IDEA:
- Ignore machine-dependent constants
- Look at *growth* of $T(n)$ as $n \to \infty$

"Asymptotic Analysis"

Θ-notation

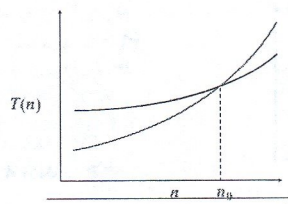
Math:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Engineering:
- Drop low-order terms; ignore leading constants
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm



- We shouldn't ignore asymptotically slower algorithms, however
- Real-world design situations often call for a careful balancing of engineering objectives
- Asymptotic analysis is a useful tool to help to structure our thinking

Insertion sort analysis

Worst case: Input reverse sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \quad \text{[arithmetic series]}$$

Average case: All permutations equally likely

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is insertion sort a fast sorting algorithm?
- Moderately so, for small n
- Not at all, for large n

Merge sort

MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$
3. "*Merge*" the 2 sorted lists

Key subroutine: **MERGE**

Merge Sort in C++

```
void merge_sort(std::vector<int> &A)
{
        std::vector<int> A1;
        std::vector<int> A2;

        if (A.size()==1) return;

        for(int i=0; i<A.size()/2; i++)
                A1.push_back(A[i]);

        for(int i=A.size()/2; i<A.size(); i++)
                A2.push_back(A[i]);

        merge_sort(A1);
        merge_sort(A2);
        A = merge(A1,A2);
}
```

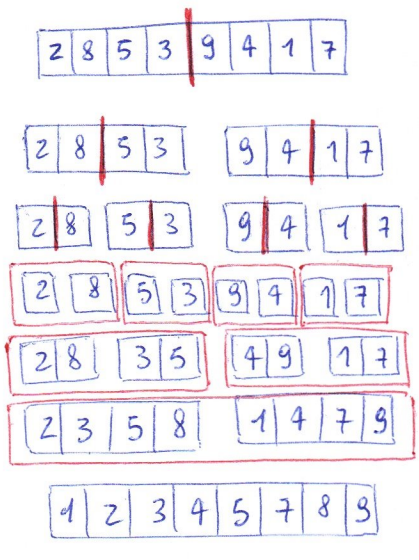
] $n/2$ times

] $n/2$ times

This takes two already sorted vectors and it merges them sorting them (let's consider this function done)

Merging two sorted arrays

20 12
13 11
7 9
2 1



Merging two sorted arrays

20 12
13 11
7 9
2 1

1

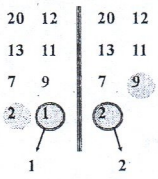
Merging two sorted arrays

20 12
13 11
7 9
2 1

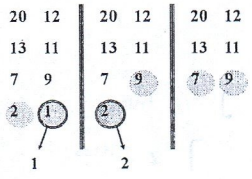
1

20 12
13 11
7 9
2

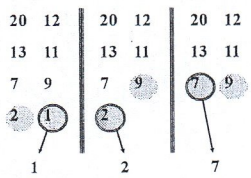
Merging two sorted arrays



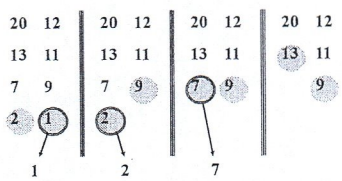
Merging two sorted arrays



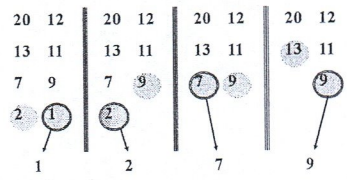
Merging two sorted arrays



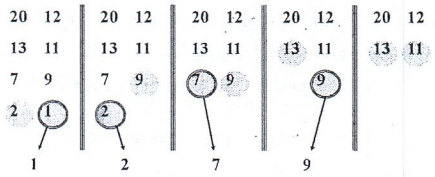
Merging two sorted arrays



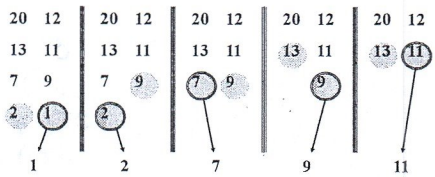
Merging two sorted arrays



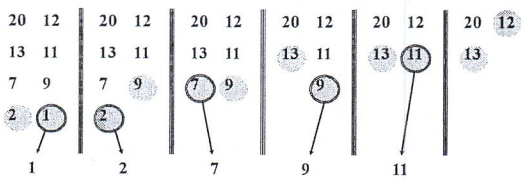
Merging two sorted arrays



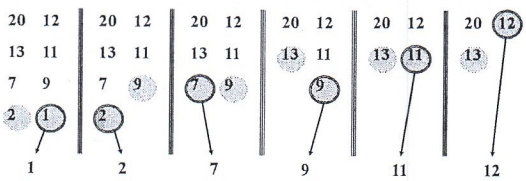
Merging two sorted arrays



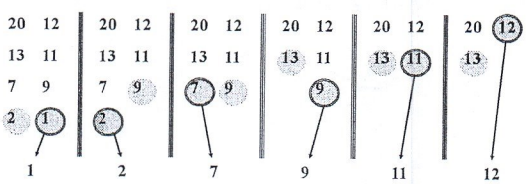
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time)

At each step we only compare the two smallest elements.

Analyzing merge sort

$T(n)$
$\Theta(1)$
$2T(n/2)$
$\Theta(n)$

Abuse

MERGE-SORT $A[1 \,..\, n]$

1. **If** $n = 1$, **done**
2. **Recursively sort** $A[\,1 \,..\, \lceil n/2 \rceil\,]$ **and** $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$
3. ***"Merge"* the 2 sorted lists**

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$

c
$\Theta(n)$ for loops
$2T(n/2)$ call
$\Theta(n)$ merge

complexity (whole) = 2 · complexity (half) + merging the halfs

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant

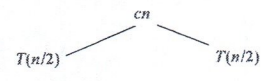
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant

$T(n)$

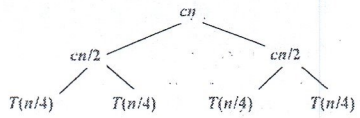
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



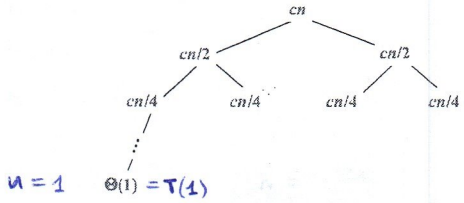
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant

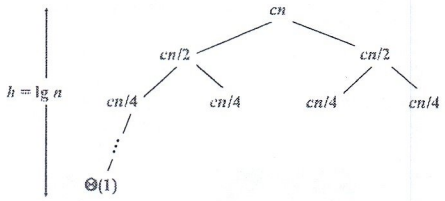


n = 1 $\Theta(1) = T(1)$

How many layers do we have? log(n)
How many elements in each layer?
The last layer will have n leaves of size 1.

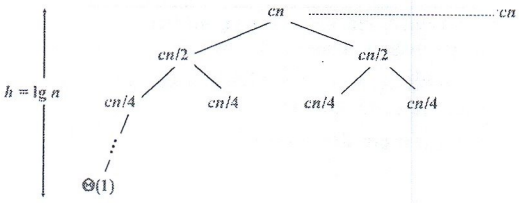
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



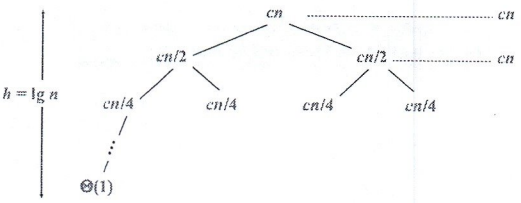
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



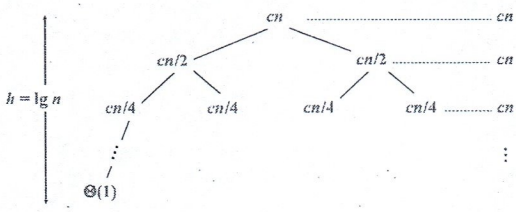
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



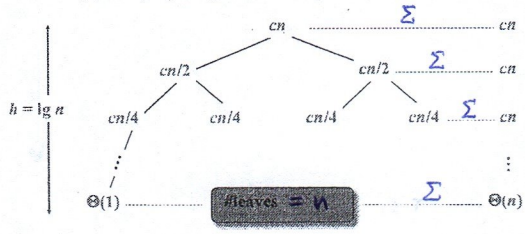
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant

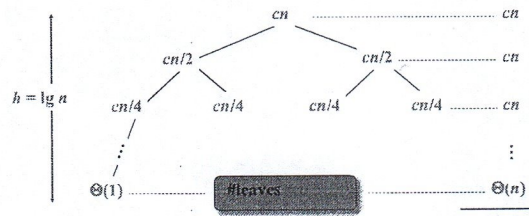


Complexity:
Σ cn = Θ(n)
Σ cn = Θ(n)
Σ cn = Θ(n)
Σ Θ(n) = Σ Θ(1) = Θ(1) · n = Θ(n)

We have to perform a computation which complexity is Θ(n) for each layer, for log(n) layers.
Final complexity:
Θ(n log(n))

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Total = $\Theta(n \lg n)$

Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$
- Therefore, merge sort asymptotically beats insertion sort in the worst case
- In practice, merge sort beats insertion sort for $n > 30$ or so
- Go test it out for yourself!

Asymptotic notation

O-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

Asymptotic notation

O-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

EXAMPLE: $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

Asymptotic notation

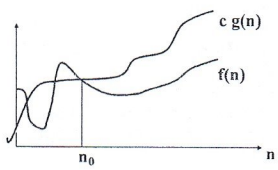
O-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

EXAMPLE: $2n^2 = O(n^3)$ $(c = 1, n_0 = 2)$

functions, not values

Asymptotic notation



Examples:
$2n^2 = O(n^3)$
$n^2+1000n = O(n^2)$
$n^{1.99999} = O(n^2)$
$n^{2.1}/\lg \lg n = O(n^3)$

Set definition of O-notation

$O(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

EXAMPLE: $2n^2 \in O(n^3)$

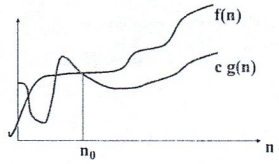
Ω-notation (lower bounds)

O-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$

$\Omega(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

EXAMPLE: $\sqrt{n} = \Omega(\lg n)$ $(c = 1, n_0 = 16)$

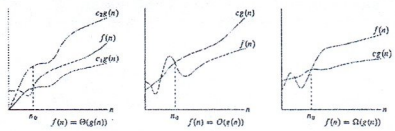
Ω-notation (lower bounds)



Examples:
$\text{sqrt}(n) = \Omega(\lg n)$
$n^2+1000n = \Omega(n^2)$
$n^2 \lg \lg n = \Omega(n^2)$

Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$



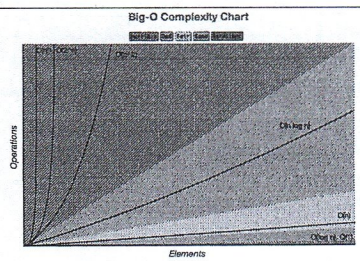
T. Cormen, C. R. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms

e.g. $n^2 + 1000n \in O(n^2)$
$n^2 + 1000n \in \Omega(n^2)$
$\Rightarrow n^2 + 1000n \in \Theta(n^2)$

O vs. Θ

- In the literature, we sometimes find O-notation informally describing asymptotically tight bounds, that is, what we have defined using Θ-notation
- In algorithms analysis, when we write f(n)= O(g(n)), we are merely claiming that some constant multiple of g(n) is an asymptotic upper bound on f (n), with no claim about how tight an upper bound it is
- Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm overall structure

Resources



http://bigocheatsheet.com

Resources

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	N/A	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
kD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

http://bigocheatsheet.com

Resources

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com