Sorting and Searching

Bruno MARTIN,
University of Nice - Sophia Antipolis
mailto:Bruno.Martin@unice.fr
http://deptinfo.unice.fr/~bmartin/mathmods.html

Sorting Time Complexity

Main **performance parameter**: **time complexity**

Differents criteria are used to evaluate the **time complexity** of an internal sorting algorithm:

- The **number** of **steps** required
- The **number** of **comparisons** between keys. Comparisons can be **expensive** when keys are **long character strings**
- The **number** of time a record is **moved**. Only keys are compared, but entire records are moved

Sorting Problem

input: sequence of N numbers $\langle a_1, a_2, ..., a_N \rangle$
output: permutation $(a'_1, a'_2, ..., a'_N, \leq)$ of the input

The input sequence is usually an array of N elements

Internal or External Sort ?

- If the input fits into memory: **internal sort**
- Sorting sets from tape or disk: **external sorting**
- Internal Sorts access to any records
- External Sorts only access records by blocks

we focus on internal sorts

Some Ideas about Time Complexity of Sorting

Simple algorithms

- like **Bubble Sort**, **Insertion Sort**, **Selection Sort**, ...
- usual **time complexity**: $O(N^2)$
- useful only for sorting **shorts lists** of records (< 500)

Famous algorithms

- **QuickSort**
  - **time complexity**: $O(N \log N)$ in the **average** case
  - **time complexity**: $O(N^2)$ in the **worst** and **best** case

Notation

We keep our focus on algorithms and think of them as sorting arrays of **N records** in ascending order of their **key** ($<$)
The algorithm of array sorting uses **key comparisons** ($<$) and **record movements** (swap)
The procedure **swap!(i,j)** is an exchange operation : $a[i] \leftrightarrow a[j]$

```
class Array
    def swap!(a,b)
         self[a], self[b] = self[b], self[a]
    self
    end
end

[1,2,3,4].swap!(2,3)  # = [1,2,4,3]
```


Implementation

```
class Array
    def bubble!
      for i in 1..self.length-1
        1.upto(self.length - i) { |j|
               self.swap!(j-1,j) if self[j-1] > self[j] }
      end
      self
    end
end
```


Bubble Sort

Description: It keeps passing through the array $[a_0, ..., a_{N-1}]$, exchanging each pair of adjacent elements (a_{j-1}, a_j) which are out of order $(a_{j-1} > a_j)$
Why does it works ?

- during the **first pass**, the **largest element** is exchanged with each of the elements to its right, and gets into position a_{N-1}
- After the **second pass** the **second largest** gets into position a_{N-2}, ...
- after step k, the sub-array $[a_{N-k}, ..., a_{N-1}]$ is ordered, we need to continue on the interval $[\![0, N-k-1]\!]$
- when no more exchanges are required: the array is sorted

Bubble Sort Average Time Complexity in Number of Comparisons

The Average number of Comparisons is $N(N-1)/2$
We count the number of comparisons needed by the algorithm:

- At the first step, we need $N-1$ comparisons to put the **largest** element at position $N-1$
- At the second step we only need $N-2$ comparisons : we avoid comparing elements with the last one

$$\vdots$$

-
- Summing up: $(N-1)+(N-2)+...+1 = N(N-1)/2$

Bubble Sort Time Complexity in Best and Worst Case

We count the number of comparisons needed by the algorithm:
Best Case: Bubble Sort on an already sorted array:

- It does like for the average case $N(N-1)/2$ **comparisons**
- During the iterations on the array: 0 **exchange**

Worst Case: array already sorted in reverse order :

- It does like for the average case $N(N-1)/2$ **comparisons**
- It does a exchange each time it does a comparison $N(N-1)/2$ **exchanges**

Selection Sort

Find the **smallest** element in the array and **exchange it** with the element in the **first** position, then find the **second smallest** element and exchange it with the element in the **second** position, continue until the entire array is sorted
Why does it works ?

- After the i^{th} step, the array between $0, \dots, i-1$ is ordered
- You are sure that the next "minimum" $a[min]$ will be larger than $a[0], \dots, a[i-1]$

Notice: A brute-force approach but, since each item is moved at most once, Selection Sort is a **method of choice when exchanging record is expensive** (large records with small keys)

Bubble Sort Average Number of Exchanges

The Average number of exchanges is $N(N-1)/4$ in a list L of N items

- Consider L randomly ordered and $\overline{L}$ its exact reverse
- Apply a bubble sort separately to both L and $\overline{L}$
- i and j are out of order in exactly one of L and $\overline{L}$, there is a swap in either L or $\overline{L}$
- the property applies to any two items in either L or $\overline{L}$ for every pair of items
- Since there are $N(N-1)/2$ distinct pairs, sorting both L and $\overline{L}$ requires $N(N-1)/2$ exchanges
- On average, $N(N-1)/4$ swaps are required for a list of size N

Ruby implementation

```
def selsort!
  return self if self.size <=1 # already sorted
  for i in 0..self.length-2 # while there are elements to sort
    min = i                 # variable for the min
    for j in i+1..self.length-1 # check every item in the array
            min = j if self[j] < self[min] # is it smaller?
    end
    self.swap!(i,min) if i != min   # exchange leftmost and min
  end
  self
end
```

Selection Sort Average Time Complexity

The average number of **comparisons** is $N(N-1)/2$ and of **exchanges** is $(N-1)/2$

- The **first step** requires $N-1$ comparisons to find the **min**
- The **second step** requires $N-2$ comparisons to find the **second min**
- The **last step** requires **1** comparison to find the min
- We do $(N-1)+(N-2)+\ldots+1 = N(N-1)/2$ comparisons
- We need **less than** $N-1$ **exchanges**. One for **each element** except when you try to exchange one element with **itself**

Insertion Sort

You consider the elements one at a time. You insert each in its proper place among those already sorted
Notice:

- After placing the element a_i, the elements $[a_0, \ldots, a_i]$ are sorted
- To place the element a_{i+1}, you iterate down the sorted array (from a_i *to* a_0) shifting one place to the right the current element if it is greater than a_{i+1}
- When the current element is smaller than a_{i+1} you have after it a free place to insert a_{i+1}

Selection Sort Time Complexity in Best and Worst Case

Best Case: Selection Sort on an already sorted array:

- It still iterates on the array to find the minimum and it does $N(N-1)/2$ **comparisons**
- doesn't exchange during the iterations: 0 **exchange**

Worst Case: Selection Sort on an array sorted in reverse order:

- It does like for the best and average case $N(N-1)/2$ **comparisons**
- It does its maximum number of exchanges $N-1$ **exchanges**

Insertion Sort Average Time Complexity

The Average number of Comparisons is $(N(N+3)/4)-1$

- The number of comparisons to insert an element in the sorted set of its predecessors is equal to the number of exchanges it causes plus one because we also compare it with the first element smaller than itself
- For the permutation α corresponding to the array to sort, the total number of comparisons is the total number of exchanges plus $N-1$
- $N(N-1)/4$=average number of exchanges in a permutation[1]
- The average number is $N-1+N(N-1)/4 = (N(N+3)/4)-1$

[1] admitted, cf. http://mathworld.wolfram.com/RandomPermutation.html

Insertion Sort Time Complexity in Best and Worst Case

Best Case: Insertion Sort on an already sorted array:

- does $N-1$ iterations on the array, at each iteration it does 1 comparison
- It doesn't exchange during the $N-1$ iterations on the array: 0 **exchange**

Worst Case: Insertion sort on an array sorted in the reverse order:

- At step i a_i is the minimum of the sorted part of the array
- algo compares a_i i times (with $a_{i-1}, ..., a_0$) $N(N-1)/2$ **comparisons**
- It does its maximum number of shifts $N(N-1)/2$ **"exchanges"**

Comparisons between Elementary Sorts

Consider the operation of sorting an "almost sorted" array:

- Insertion Sort becomes useful because its time complexity depends quite heavily on the order present in the array
- For each element you count the number of elements to its left which are greater
- This is the distance the elements have to move when inserted into the array
- In an almost sorted array the distance is small
- When records are large in comparison to the keys, **Selection Sort** is **linear** in exchanges

Comparisons between Elementary Sorts in the Best Case

Consider the operation of sorting an already sorted array:

- Bubble Sort can be **linear** : it iterates one time on the array using $N-1$ comparisons and 0 exchange and stops
- Insertion sort is **linear** : each element is immediately determined to be in its proper place in the array
- Selection Sort is **quadratic** : it keeps searching the minimum element

Quick Sort

Invented by C.A.R. Hoare in 1960, easy to implement, a good general purpose internal sort

It is a **divide-and-conquer** algorithm :

- take at random an element in the array, say v
- divide the array into two partitions :
  - One contains elements smaller than v
  - The other contains elements greater than v
- put the elements $\leq v$ at the begining of the array (say, index between 1 and $m-1$) and the elements $\geq v$ at the end of the array (index between $m+1$ and N) then you have found the place to put v between the two partitions (at position m)
- recursively call QuickSort on $([a_0, ..., a_{m-1}]$ and $[a_{m+1}, ..., a_{N-1}])$
- stop when the partition is reduced to a single element

Implementation with ruby features

It uses the ideas of the quicksort

```
def qsort
    return self if empty?
    select { |x| x < first }.qsort
     + select { |x| x==first}
     + select { |x| x > first }.qsort
end
```

How can we replace the `select` operator from ruby?

Algorithm of the Partition of the Array

Scans (index i) from the left until you find an elt $\geq v$ ($a[i] \geq v$)
Scans (index j) from the right until you find an elt $\leq v$ ($a[j] \leq v$)
Both elements are obviously out of place: swap $a[i]$ and $a[j]$
Continue until the scan pointers cross ($j \leq i$)
Exchange v ($a[right]$) with the element $a[i]$

```
until j<=i do
  i+=1 until self[i]>=v #scans for i:self[i]>=v
  j-=1 until self[j]<=v #scans for j:self[j]<=v
  if i<=j
    self.swap!(i,j)      #exchange both elements
    i+=1; j-=1           #modify indexes:clean recursion
  end
end
```


Algorithm of Quick Sort

For example, the random element can be the leftmost or the rightmost element, we choose the rightmost.
"Our" QuickSort runs on an array $[a_{left}, ..., a_{right}]$:

```
def quick!(left,right)
  if left < right
    m = self.partition(left,right)
    self.quick!(left, m-1)
    self.quick!(m+1, right)
  end
end
```


The big picture

```
def qsort!
  def lqsort(left,right)      #sort from left to right
    if left<right
      v,i,j=self[right],left,right
      until j<=i do
        i+=1 until self[i]>=v #scans for i:self[i]>=v
        j-=1 until self[j]<=v #scans for j:self[j]<=v
        if i<=j
          self.swap!(i,j)     #exchange both elements
          i+=1; j-=1          #modify indexes:clean recursion
        end
      end
      self.lqsort(left,j)     #sort left part
      self.lqsort(i,right)    #sort right part
    end
  end
  self.lqsort(0,self.length-1)
  self
end
```

Quick Sort

We test that neither i nor j cross the array bounds *left* and *right*
Because $v = self[right]$ you are sure that the loop on i stops at least when $i = right$
But if $v = self[right]$ happens to be the smallest element between *left* and *right*, the loop on j might pass the left end of the array
To avoid the tests, you can choose another solution

- Take three elements in the array: the leftmost, the rightmost and the middle one
- Sort them
- Put the smallest at the leftmost position, the greatest at the rightmost position and the middle one as v

C_N : average number of comparisons for sorting N elements:
$C_N = \ N+1 \ + \ \frac{1}{N}\sum_{k=1}^{N}(C_{k-1}+C_{N-k})$

- $N+1$ comparisons during the two inner whiles $N-1+2$ (2 when i and j cross)
- Plus the average number of comparisons on the two sub-arrays $((C_0+C_{N-1})+(C_1+C_{N-2})+...+(C_{N-1}+C_0))/N$

By symmetry : $C_N = \ N+1 \ + \ \frac{2}{N}\sum_{k=1}^{N} C_{k-1}$
substract $NC_N-(N-1)C_{N-1}$

$$NC_N = (N+1)C_{N-1}+2N$$

divide both side by $N(N+1)$ to obtain the recurrence :

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N}+\frac{2}{N+1} = \frac{C_{N-2}}{N-1}+\frac{2}{N}+\frac{2}{N+1} = ... = \frac{C_2}{3}+2\sum_{k=4}^{N+1}\frac{1}{k}$$

Approximation : $\frac{C_N}{N+1} \approx 2\sum_{k=1}^{N}\frac{1}{k} \approx 2\int_1^N \frac{1}{x}dx \approx 2 \ ln \ N$

$$C_N \approx 2NlnN \approx 2Nln(2)Log(N) \approx 1.38NLogN$$

Quick Sort on Average-Case Partitioning

Average performance of Quick Sort is about $1.38N\log N$:
very efficient algorithm with a very small constant
Quick Sort is a divide-and-conquer algorithm which splits the problem in two recursive calls and "combines" the results
Divide-and-conquer is a good method every time you can split your problem in smaller pieces and combine the results to obtain the global solution
But divide-and-conquer leads to an efficient algorithm only when the problem is divided without overlap

Quick Sort on worst-case partitioning

Quick Sort is very inefficient on already sorted sets: $O(N^2)$

- Suppose $a[0],...,a[N-1]$ sorted without equal elements
- At the first call $v = a[N-1]$
  - The `while` on i continues until $i = N-1$ and stops because $a[N-1] = v$: the sort does N comparisons
  - The `while` on j stops on $j = N-2$ because $a[N-2] < v$: 1 comparison
  - We exchange $a[N-1]$ with itself : 1 exchange
  - We call QuickSort on $a[0],...,a[N-2]$ and on $a[N-2],...,a[N-1]$ which imediately stops
- So $(N+1)+N+(N-1)+...+2 = N(N+3)/2$
- QuickSort is in $O(N^2)$ on sorted sets

Intuition for the performance of quick sort

Quicksort running time depends on whether the partitioning is balanced

The worst-case partitioning occurs when the partitioning produces one region with 1 element and one with $N-1$ elements: $O(N^2)$

The best-case partitioning occurs when the partitioning produces two regions with $N/2$ elements ($C_N = N + 2C_{N/2}$): $O(N \log N)$

```
  worst-case          ^            best-case               N   ^
      N               |                N                       |
     / \              |              /   \                 N   |
    1  N-1            |            N/2    N/2                 log
        / \           N           / \      / \             N   |
       1  N-2         |        N/4  N/4  N/4  N/4              |
           / \        |                .                       |
          1  N-3      |                .                   N   |
              ...     |         1 1    ...    1 1              v
              / \     |
             1   1    v
```


Representing the decision tree model

Set to sort: $\{a1, a2, a3\}$ the corresponding decision tree is :

```
                      a1 > a2
                     /        \
              a2 > a3          a1 > a3
              /    \              /  \
  (a1,a2,a3)  a1 > a3  (a2,a1,a3)  a2 > a3
                 / \                  /  \
     (a1,a3,a2)  (a3,a1,a2)  (a3,a2,a1)  (a3,a2,a1)
```

The **decision tree** to sort N elements has $N!$ leaves (all **possible permutations**)

A binary tree with $N!$ leaves has a height order of $\log(N!)$ which is approximately $N \log N$ (Stirling)

$N \log N$ is a lower bound for sorting

Lower Bound for Sorting

Is it possible to **sort an array** of **N** elements in less than $N \log N$ **operations** ?

If you use element comparisons: it is impossible

You need to **model** your **computation problem**:

You express each sort by a **decision tree** where **each internal node** represents the **comparison between two elements**

The **left child** correspond to the **negative answer** and the **right child** to the **positive one**

Each leaf represents a given permutation

Overview

Introduction to Searching

Searching: **fundamental operation** in many tasks: retrieving a particular information among a large amount of stored data

The stored data can be viewed as a **set**
Information divided into **records** with field **key** used for searching

Goal of Searching: find the records whose key matches a given searched key

Dictionaries and **symbol tables** are two examples of data structures needed for searching

Sequential Searching in an Array is $O(N)$

Sequential Searching in an **array** uses

- $N+1$ **comparisons** for an **unsuccessful search** in the best, average and worst case
- $(N+1)/2$ **comparisons** for a **successful search** on the average[2]
  - Suppose that the records have the same probability to be found
  - We do 1 comparison with the first one,
  - $\vdots$
  - N to find the last one
  - on the average: $(1+2+...+N)/N = N(N+1)/2N$

[2] average=mean= $\frac{\text{sum of all the entries}}{\text{number of entries}}$

Operations of Searching

The **time complexity** often depends on the **structure** given to the **set of records** (eg lists, sets, arrays, trees,...)

So, when programming a **Searching** algorithm on a structure, one often needs to provide operations like **Insertion**, **Deletion** and sometimes **Sorting** the set of records

In any case, the **time complexity** of the searching algorithm might be sensitive to operations like comparison of keys, insertion of one record in the set, shift of records, exchange of records, ...

Sequential Searching in a Sorted List is in $O(N)$

Sequential searching in a **sorted list** approximately uses $N/2$ for both a **successful** and an **unsuccessful** search

- The (average) complexity of the **successful** search in sorted lists equals the successful search on array in the average case
- For unsuccessful:
  - The search can be ended by each of the elements of the list
  - We do 1 comparison if the searched key is less than the first element,..., $N+1$ comparison if the key is greater than the last one (the sentinel)
  - $(1+...+(N+1))/N = (N+1)(N+2)/2N$

An Elementary Searching Algorithm : the Binary Search

When the set of records gets large and the records are **ordered** to reduce the searching time, use a **divide-and-conquer** strategy:

- Divide the set into two parts
- Determine in which part the key might belong to
- Repeat the search on this part of the set

Performance of Binary Search

Binary Search uses approximately $\log N$ comparisons for both **(un)successful** search in **best, average** and **worst** case

Maximal number of comparisons when the search is **unsuccessful**

Application to numerical analysis

For finding an approximate of the zeroes of a cont. function by the

Theorem (Intermediate value theorem)

If the function $f(x) = y$ is continuous on $[a, b]$ and u is a number st $f(a) < u < f(b)$, then there is a $c \in [a, b]$ s.t. $f(c) = u$.

if one can evaluate the sign of $f((a + b)/2)$;
Let f be strictly increasing on $[a, b]$ with $f(a) < 0 < f(b)$
The binary search allows to find y st $f(y) = 0$:

1. start with the pair (a, b)
2. evaluate $v = f((a + b)/2)$
3. if $v < 0$ replace a by v otherwise replace b by v
4. iterate on the new pair until the diff. between the values is less than an arbitrary given precision

Performance of Binary Search

Proof 1 :

- Consider the tree of the recursive calls of the Search
- At each call the array is split into two halves
- The tree is a full binary tree
- The number of comparisons equals the tree height : $\log_2 N$

Proof 2 :

- The number of comparisons at step N equals the number of comparisons in one subarray plus 1 because you compare with the root
- Solve the recurrence
  $C_N = C_{N/2} + 1, \text{ for } N \geq 2 \text{ with } C_1 = 0 \ \rightarrow \log N$
  $C_N = C_{N/2} + 1 \ N = 2^n \ C_{2^n} = C_{2^{n-1}} + 1 \ldots C_{2^n} = n = \log N$

Order of magnitude

Searching on the average case :

- A successful sequential search in a set of 10000 elements takes 5000 comparisons
- A successful binary search in the same set takes 14 comparisons

BUT
Inserting an element :

- In an array takes 1 operation
- In a sorted array takes N operations : to find the place and shift right the other elements

Elementary Searching Algorithm: Interpolation Searching

Dictionary search: if the word begins by **B** you look near the beginning and if the word begins by **T** you turn a lot of pages.

Suppose you search the key k, in the binary search you cut the array in the middle

$$middle = left + \frac{1}{2}(right - left)$$

In the **interpolation** you takes the values of the keys into account by replacing 1/2 by a better progression

$$position = left + \frac{k - A[left].key}{A[right].key - A[left].key}(right - left)$$

Performance of the Interpolation Search

The interpolation search uses approximately $\log(\log N)$ comparisons for both **(un)successful** search in the array

But Interpolation search heavily depends on the fact that the keys are well distributed over the interval

The method **requires some computation**; for small sets the $\log N$ of binary search is close to $\log(\log N)$

So interpolation search should be used for **large sets** in applications where **comparisons** are particularly **expensive** or for **external** methods where access costs are high