

Lecture 8-2

Algorithms



Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...



Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

Typical run-time functions

Sorting

The sorting task:

- A prevalent computational task:
 - Sort web pages by relevance / pageRank
 - Sort your contact list
 - Sorting before searching
- Input: an array of values, e.g. (5, 2, 10, 8, 5)
- Output: a sorted array, e.g. (2, 5, 5, 8, 10)
- The values can be any object whose data type has an order

For simplicity, in this lecture we will sort integers;
All the algorithms we show can be easily adapted
to sort values of any ordered data type



Selection sort

```
// Sort an array of length N using selection sort
for j = 0 .. N-1
  min = j
  for k = j+1 .. N
    if (a[k] < a[min])
      min = k
  // Switch
  if (min != j)
    temp = a[min]
    a[min] = a[j]
    a[j] = temp
```

Unsorted:	13	10	7	3	15	1	4	11
Step 0:	1	10	7	3	15	13	4	11
Step 1:	1	3	7	10	15	13	4	11
Step 2:	1	3	4	10	15	13	7	11

Etc.

Selection sort : run-time analysis

```
// Sort an array of length N using selection sort
for j = 0 .. N-1
  min = j
  for k = j+1 .. N
    if (a[k] < a[min])
      min = k
  // Switch
  if (min != j)
    temp = a[min]
    a[min] = a[j]
    a[j] = temp
```

Let N be the size of the array

- o In step 0, we scan N - 1 numbers
- o In step 1, we scan N - 2 numbers
- o In step 2, we scan N - 3 numbers
- o In step j, we scan N - 1 - j numbers

Thus, the total number of steps is

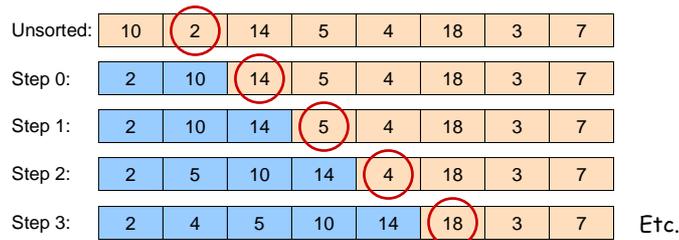
$$(N - 1) + (N - 2) + \dots + 1 = \frac{1}{2} N (N - 1) = \frac{1}{2} N^2 - \frac{1}{2} N$$

Algorithm's run-time: order of N^2

How good is N^2 ?

If you have to sort 100,000 numbers, it will take 10,000,000,000 steps.

Insertion sort



- In step j, entries $a[0] \dots a[j-1]$ are sorted; Insert item $a[j]$ (also called "key") in its correct location.
- "Inserting" the key can be done in two ways:
 - Sequential search: requires pair-wise switches until the key arrives to its right place
 - Binary search: find the correct location of the key within $a[0 \dots j-1]$, shift the necessary array entries to the right, then insert the key.

Insertion sort: algorithm and run-time analysis

```
// Sorts an array of length N using insertion sort
for k = 1 .. N-1 {
  x = A[k]
  j = k - 1
  // shift the array to the right and
  // insert x to its right place
  while (A[j] > x and j>=0) {
    A[j+1] = A[j]
    j--
  }
  A[j+1] = x
}
```

Worst-case analysis:

- In step 1, we make 1 shift
- In step 2, we make 2 shifts
- In step 3, we make 3 shifts
- In step j, we make j shifts

The total number of steps is $1 + 2 + 3 + \dots + N = \frac{1}{2} N(N+1)$

Algorithm's run-time: order of N^2 .

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

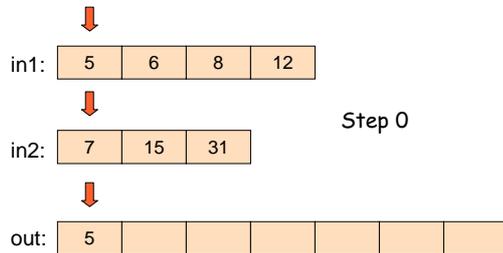
Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

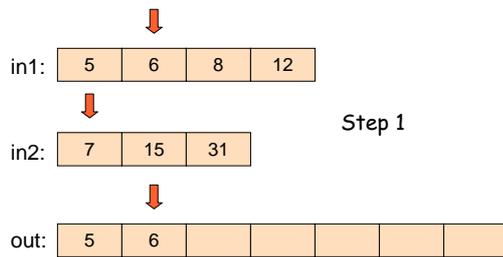
Typical run-time functions



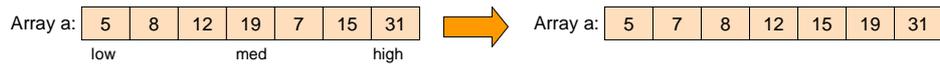
Merging



```
// Merges two sorted arrays in1, in2
// into a sorted array named out
out = construct an array of size
      in1.length + in2.length
i1 = i2 = i = 0
while there is more work to do:
  if (in1[i1] < in2[i2])
    out[i++] = in1[i1++]
  else
    out[i++] = in2[i2++]
```

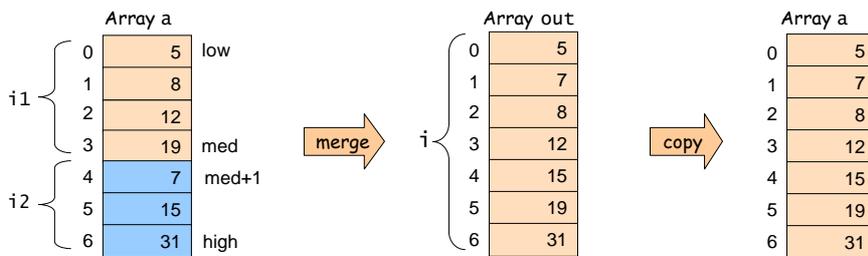


In-place merging



In-place merging:

```
merge(a, low, med, high):
  out = construct an array of size high-low+1
  i1=low..med, i2=(med+1)..high, i=low..high
  while there is more work to do:
    if (a[i1] < a[i2])
      out[i++] = a[i1++]
    else
      out[i++] = a[i2++]
  copy out onto a
```



Merge sort

Array a:

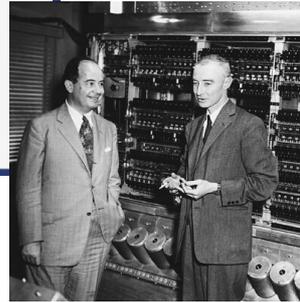
31	12	19	8	5	15	7
----	----	----	---	---	----	---

 Array a:

5	7	8	12	15	19	31
---	---	---	----	----	----	----

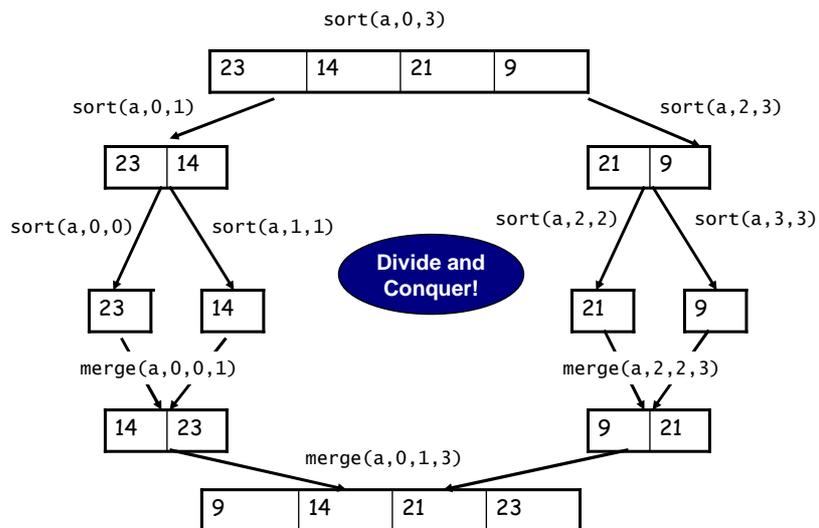
```
// Sorts an array using the MergeSort algorithm
mergeSort(a):
  sort(A,0,N-1) // Sort A[0] to A[N-1]

// Sorts an array between given indexes low and high
sort(a, low, high):
  if (low >= high ) return
  med = (low + high) / 2;
  sort(a, low, med);
  sort(a, med + 1, high);
  merge(a, low, med, high);
```

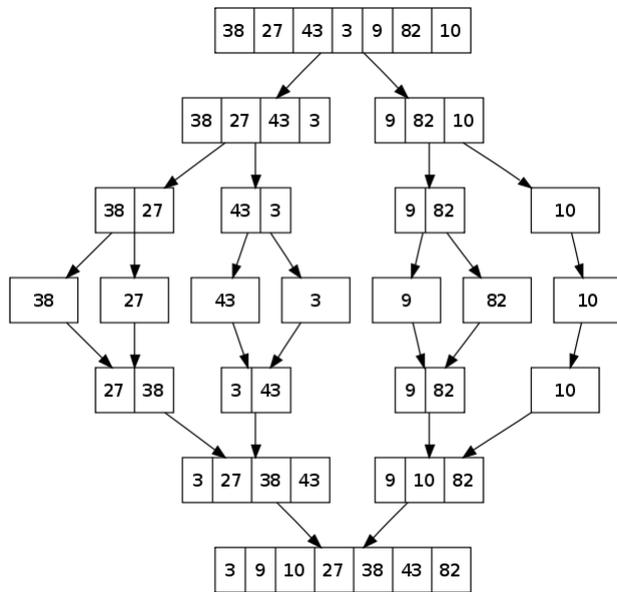


Invented by John Von Neumann, 1945

Merge sort: sample run



Merge sort: run-time analysis



Algorithm's run-time:
Order of $n \log_2 n$

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

➡ Typical run-time functions

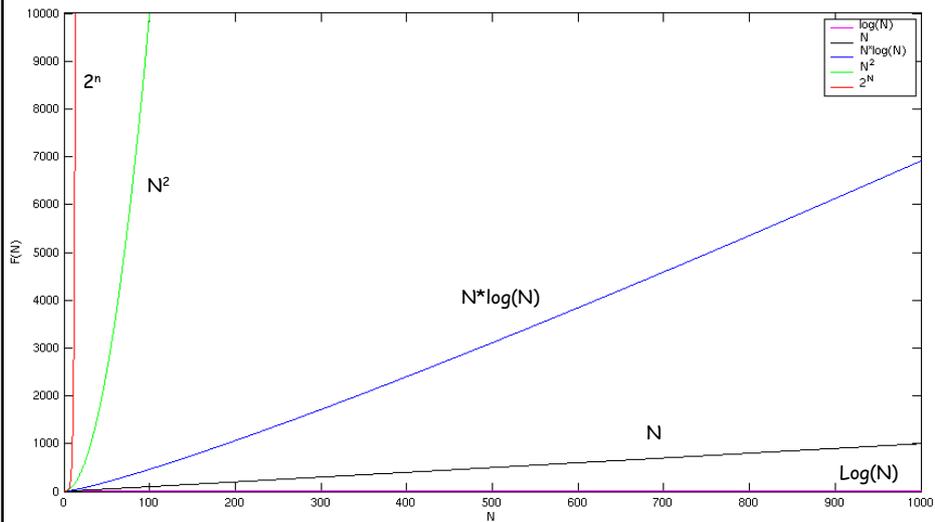
Running time analysis: summary of results thus far

When asked to analyze the performance of a given algorithm, we try to describe its run-time using a nicely stated functions of N , the input's size

Examples:

- Sequential search: order of N
- Binary search: order of $\log_2 N$
- Selection sort: order of N^2
- Insertion sort: order of N^2
- Merge sort: order of $N \log_2 N$
- Exponential run-time: order of 2^N

Functions that typically come up in run-time analysis



Counting primes

```
// Counts how many primes exist up to N
numPrimes = 0
for i = 2 .. N {
  isPrime = true;
  for j = 2 to i
    if j|i
      isPrime = false;
  if isPrime
    numPrimes++;
}
```

(Version 1, to be improved later)

Counting primes: Java implementation

```
public class CountPrimes {
  final static int N = 10000;
  public static void main (String args[]) {
    int numPrimes = 0;
    boolean isPrime;
    long startTime = System.currentTimeMillis();
    for (int i = 2; i < (N + 1); i++) {
      isPrime = true;
      for (int j = 2; j < i; j++)
        if (i % j == 0)
          isPrime = false;
      if (isPrime)
        numPrimes++;
    }
    System.out.println("There are " + numPrimes + " primes below " + N);
    long endTime = System.currentTimeMillis();
    System.out.println("Run-time = " + (endTime - startTime));
  }
}
```

Counting primes: run-time analysis

```
// Counts how many primes exist up to N
numPrimes = 0
for i = 2 .. N {
  isPrime = true;
  for j = 2 to i
    if j|i
      isPrime = false;
  if isPrime
    numPrimes++;
}
```

Run-time analysis

- Number of iterations is $1 + 2 + 3 + \dots + N = \frac{1}{2}(N^2 + N)$
- Running-time is order of N^2

Empirical analysis: (Using the Java program)

Input (N)	10K	20K	30K	40K
NumPrimes	1229	2262	3245	4203
Run-time	181	725	1605	2837
Input growth rate	1	2	3	4
Run-Time growth rate	1	4	9	16

A quadratic growth function

Can we do better ?

Counting primes: version 2

```
// Counts how many primes exist up to N
numPrimes = 0
for i = 2 .. N {
  isPrime = true;
  for j = 2 to sqrt(i)
    if j|i
      isPrime = false;
  if isPrime
    numPrimes++;
}
```

Run-time analysis:

- Number of iterations is $\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{N} < N\sqrt{N}$
- Running time is at most order of $N\sqrt{N}$
- That's an example of establishing an *upper-bound* on the running time.

Empirical analysis:

Input (N)	10K	20K	30K	40K
Run-time (ms) without sqrt	780	3000	6800	12000
Run-time (ms) with sqrt	10	28	50	79

Significant improvement compared to the quadratic running-time