Introduction to Programming (in C++)

Sorting

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC
Sorting

• Let \texttt{elem} be a type with a \(\leq\) operation, which is a total order

• A \texttt{vector\<\texttt{elem}\>} \(v\) is (increasingly) sorted if for all \(i\) with \(0 \leq i < v\text{.size()}-1\), \(v[i] \leq v[i+1]\)

• Equivalently:
  
  \[
  \text{if } i < j \text{ then } v[i] \leq v[j]
  \]

• A fundamental, very common problem: \texttt{sort v}
  
  Order the elements in \(v\) and leave the result in \(v\)
Another common task: sort \( v[a..b] \)
We will look at four sorting algorithms:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort

Let us consider a vector \( v \) of \( n \) elems (\( n = v\text{.size}() \))

- Insertion, Selection and Bubble Sort make a number of operations on \( \text{elems} \) proportional to \( n^2 \)

- Merge Sort is proportional to \( n \cdot \log_2 n \): faster except for very small vectors
Selection Sort

• Observation: in the sorted vector, v[0] is the smallest element in v

• The second smallest element in v must go to v[1]...

• ... and so on

• At the i-th iteration, select the i-th smallest element and place it in v[i]
Selection Sort

From http://en.wikipedia.org/wiki/Selection_sort
• Selection sort keeps this invariant:

```
-7 -3 0 1 4 9 ? ? ? ? ? ?
```

- this is sorted and contains the \( i-1 \) smallest elements
- this may not be sorted… but all elements here are larger than or equal to the elements in the sorted part
// Pre: --
// Post: v is now increasingly sorted

void selection_sort(vector<elem>& v) {
    int last = v.size() - 1;
    for (int i = 0; i < last; ++i) {
        int k = pos_min(v, i, last);
        swap(v[k], v[i]);
    }
}

// Invariant: v[0..i-1] is sorted and
// if a < i <= b then v[a] <= v[b]

Note: when i=v.size()-1, v[i] is necessarily the largest element. Nothing to do.
Selection Sort

// Pre: 0 <= left <= right < v.size()
// Returns pos such that left <= pos <= right
// and v[pos] is smallest in v[left..right]

int pos_min(const vector<elem>& v, int left, int right) {
    int pos = left;
    for (int i = left + 1; i <= right; ++i) {
        if (v[i] < v[pos]) pos = i;
    }
    return pos;
}
Selection Sort

• At the i-th iteration, Selection Sort makes
  – up to v.size()-1-i comparisons among elems
  – 1 swap (=3 elem assignments) per iteration

• The total number of comparisons for a vector of size n is:
  \[(n-1)+(n-2)+\ldots+1 = n(n-1)/2 \approx n^2/2\]

• The total number of assignments is 3(n-1).
**Insertion Sort**

- Let us use induction:
  - If we know how to sort arrays of size \( n-1 \),
  - do we know how to sort arrays of size \( n \)?
Insertion Sort

- Insert $x = v[n-1]$ in the right place in $v[0..n-1]$

- Two ways:
  - Find the right place, then shift the elements
  - Shift the elements to the right until one $\leq x$ is found
• Insertion sort keeps this invariant:

\[-7 \ -3 \ 0 \ 1 \ 4 \ 9 \ \text{?} \ \text{?} \ \text{?} \ \text{?} \ \text{?} \ \text{?} \ \text{?} \]

This is sorted

This may not be sorted and we have no idea of what may be here
Insertion Sort

From http://en.wikipedia.org/wiki/Insertion_sort
// Pre: --
// Post: v is now increasingly sorted
void insertion_sort(vector<elem>& v) {
    for (int i = 1; i < v.size(); ++i) {
        elem x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > x) {
            v[j] = v[j - 1];
            --j;
        }
        v[j] = x;
    }
}

// Invariant: v[0..i-1] is sorted in ascending order
Insertion Sort

• At the i-th iteration, Insertion Sort makes up to i comparisons and up to i+2 assignments of type elem

• The total number of comparisons for a vector of size n is, at most:

\[ 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2} \approx \frac{n^2}{2} \]

• At the most, \( \frac{n^2}{2} \) assignments

• But about \( \frac{n^2}{4} \) in typical cases
Selection Sort vs. Insertion Sort
void insertion_sort(vector<elem>& v) {
  for (int i = 1; i < v.size(); ++i) {
    elem x = v[i];
    int j = i;
    while (j > 0 and v[j - 1] > x) {
      v[j] = v[j - 1];
      --j;
    }
    v[j] = x;
  }
}

• How about: while (v[j - 1] > x and j > 0) ?

• Consider the case for j = 0 → evaluation of v[-1] (error !)

• How are complex conditions really evaluated?
Evaluation of complex conditions

• Many languages (C, C++, Java, PHP, Python) use the **short-circuit evaluation** (also called **minimal** or **lazy** evaluation) for Boolean operators.

• For the evaluation of the Boolean expression

\[ \text{expr1 \ op \ expr2} \]

\(\text{expr2}\) is only evaluated if \(\text{expr1}\) does not suffice to determine the value of the expression.

• Example: \((j > 0 \ \text{and} \ v[j-1] > x)\)

\(v[j-1]\) is only evaluated when \(j>0\)
Evaluation of complex conditions

- In the following examples:
  
  \[ n \neq 0 \text{ and } \frac{\text{sum}}{n} > \text{avg} \]
  
  \[ n == 0 \text{ or } \frac{\text{sum}}{n} > \text{avg} \]

  \( \frac{\text{sum}}{n} \) will never execute a division by zero.

- Not all languages have short-circuit evaluation. Some of them have \textit{eager evaluation} (all the operands are evaluated) and some of them have both.

- The previous examples could potentially generate a runtime error (division by zero) when eager evaluation is used.

- Tip: short-circuit evaluation helps us to write more efficient programs, but cannot be used in all programming languages.
Bubble Sort

• A simple idea: traverse the vector many times, swapping adjacent elements when they are in the wrong order.

• The algorithm terminates when no changes occur in one of the traversals.
The largest element is well-positioned after the first iteration.

The second largest element is well-positioned after the second iteration.

The vector is sorted when no changes occur during one of the iterations.
Bubble Sort

From http://en.wikipedia.org/wiki/Bubble_sort
void bubble_sort(vector<elem>& v) {
    bool sorted = false;
    int last = v.size() - 1;
    while (!sorted) {
        // Stop when no changes
        sorted = true;
        for (int i = 0; i < last; ++i) {
            if (v[i] > v[i + 1]) {
                swap(v[i], v[i + 1]);
                sorted = false;
            }
        }
        // The largest element falls to the bottom
        --last;
    }
}

Observation: at each pass of the algorithm, all elements after the last swap are sorted.
void bubble_sort(vector<elem>& v) {
    int last = v.size() - 1;
    while (last > 0) {
        int last_swap = 0; // Last swap at each iteration
        for (int i = 0; i < last; ++i) {
            if (v[i] > v[i + 1]) {
                swap(v[i], v[i + 1]);
                last_swap = i;
            }
        }
        last = last_swap; // Skip the sorted tail
    }
}
Bubble Sort

• Worst-case analysis:
  – The first pass makes n-1 swaps
  – The second pass makes n-2 swaps
  – ...
  – The last pass makes 1 swap

• The worst number of swaps:
  
  \[1 + 2 + ... + (n-1) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}\]

• It may be efficient for nearly-sorted vectors.

• In general, bubble sort is one of the least efficient algorithms. It is not practical when the vector is large.
Merge Sort

• Recall our induction for Insertion Sort:
  – suppose we can sort vectors of size $n-1$,  
  – can we now sort vectors of size $n$?

• What about the following:
  – suppose we can sort vectors of size $n/2$,      
  – can we now sort vectors of size $n$?
Merge Sort

9 -7 0 1 -3 4 3 8 -6 8 6 2

Induction!

-7 -3 0 1 4 9 3 8 -6 8 6 2

Induction!

-7 -3 0 1 4 9 -6 2 3 6 8 8

Induction!

How do we do this?

-7 -6 -3 0 1 2 3 4 6 8 8 9
Merge Sort

From http://en.wikipedia.org/wiki/Merge_sort
Merge Sort

• We have seen almost what we need!

// Pre: A and B are sorted in ascending order
// Returns the sorted fusion of A and B

`vector<elem> merge(const vector<elem>& A, const vector<elem>& B);`

• Now, \( v[0..n/2-1] \) and \( v[n/2..n-1] \) are sorted in ascending order.

• Merge them into an auxiliary vector of size \( n \), then copy back to \( v \).
Merge Sort

9 -7 0 1 4 -3 3 8

9 -7 0 1

4 -3 3 8

Merge Sort

Split

Merge Sort

-7 0 1 9

Merge

-3 3 4 8

-7 -3 0 1 3 4 8 9
Merge Sort

// Pre: $0 \leq \text{left} \leq \text{right} < v.\text{size}()$
// Post: $v[\text{left}..\text{right}]$ has been sorted increasingly

```cpp
void merge_sort(vector<elem>& v, int left, int right) {
    if (left < right) {
        int m = (left + right)/2;
        merge_sort(v, left, m);
        merge_sort(v, m + 1, right);
        merge(v, left, m, right);
    }
}
```
Merge Sort – merge procedure

// Pre:  0 <= left <= mid < right < v.size(), and
//       v[left..mid], v[mid+1..right] are both sorted increasingly
// Post: v[left..right] is now sorted

void merge(vector<elem>& v, int left, int mid, int right) {
    int n = right - left + 1;
    vector<elem> aux(n);
    int i = left;
    int j = mid + 1;
    int k = 0;
    while (i <= mid and j <= right) {
        if (v[i] <= v[j]) { aux[k] = v[i]; ++i; }
        else { aux[k] = v[j]; ++j; }
        ++k;
    }
    while (i <= mid) { aux[k] = v[i]; ++k; ++i; }
    while (j <= right) { aux[k] = v[j]; ++k; ++j; }

    for (k = 0; k < n; ++k) v[left+k] = aux[k];
}
Merge Sort

: merge_sort
: merge
• How many elem comparisons does Merge Sort do?
  – Say v.size() is n, a power of 2
  – merge(v,L,M,R) makes k comparisons if k=R-L+1
  – We call merge \( \frac{n}{2^i} \) times with R-L=2^i
  – The total number of comparisons is

\[
\sum_{i=1}^{\log_2 n} \frac{n}{2^i} \cdot 2^i = n \cdot \log_2 n
\]

The total number of elem assignments is \( 2n \cdot \log_2 n \)
## Comparison of sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td><img src="Image" alt="Diagram" /></td>
</tr>
<tr>
<td>Insertion</td>
<td><img src="Image" alt="Diagram" /></td>
</tr>
<tr>
<td>Bubble</td>
<td><img src="Image" alt="Diagram" /></td>
</tr>
<tr>
<td>Merge</td>
<td><img src="Image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Comparison of sorting algorithms

• Approximate number of comparisons:

<table>
<thead>
<tr>
<th>n = v.size()</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion, Selection and Bubble Sort ($\approx n^2/2$)</td>
<td>50</td>
<td>5,000</td>
<td>500,000</td>
<td>50,000,000</td>
<td>5,000,000,000</td>
</tr>
<tr>
<td>Merge Sort ($\approx n \cdot \log_2 n$)</td>
<td>67</td>
<td>1,350</td>
<td>20,000</td>
<td>266,000</td>
<td>3,322,000</td>
</tr>
</tbody>
</table>

• Note: it is known that every general sorting algorithm must do at least $n \cdot \log_2 n$ comparisons.
Comparison of sorting algorithms

Execution time (µs)

For small vectors

Vector size

Insertion Sort
Selection Sort
Bubble Sort
Merge Sort
Comparison of sorting algorithms

**Execution time (ms)**

- **Insertion Sort**
- **Selection Sort**
- **Bubble Sort**
- **Merge Sort**

**For medium vectors**

- **Vector size**
  - 100
  - 200
  - 300
  - 400
  - 500
  - 600
  - 700
  - 800
  - 900
  - 1000

**Introduction to Programming**

© Dept. CS, UPC
Comparison of sorting algorithms

Execution time (secs)

For large vectors

- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge Sort

Vector size

Introduction to Programming

© Dept. CS, UPC
Other sorting algorithms

• There are many other sorting algorithms.

• The most efficient algorithm for general sorting is \textit{quick sort} (C.A.R. Hoare).
  – The worst case is proportional to $n^2$
  – The average case is proportional to $n \cdot \log_2 n$, but it usually runs faster than all the other algorithms
  – It does not use any auxiliary vectors

• Quick sort will not be covered in this course.
A sorting procedure is available in the C++ library

It probably uses a quicksort algorithm

To use it, include:

```
#include <algorithm>
```

To increasingly sort a vector v (of int’s, double’s, string’s, etc.), call:

```
sort(v.begin(), v.end());
```
Sorting with the C++ library

- To sort with a different comparison criteria, call
  
  ```cpp
  sort(v.begin(), v.end(), comp);
  ```

- For example, to sort int’s **decreasingly**, define:
  
  ```cpp
  bool comp(int a, int b) {
      return a > b;
  }
  ```

- To sort people by age, then by name:
  
  ```cpp
  bool comp(const Person& a, const Person& b) {
      if (a.age == b.age) return a.name < b.name;
      else return a.age < b.age;
  }
  ```
Sorting is not always a good idea...

- **Example:** to find the min value of a vector

```cpp
min = v[0];
for (int i=1; i < v.size(); ++i)
  if (v[i] < min) min = v[i];
```

```
sort(v);
min = v[0];
```

- **Efficiency analysis:**
  - **Option (1):** $n$ iterations (visit all elements).
  - **Option (2):** $2n \cdot \log_2 n$ moves with a good sorting algorithm (e.g., merge sort)