Simple sorting algorithms and their complexity

- Bubble sort
- Selection sort
- Insertion sort

Complexity of bubble sort

- For an array of size \(n\), in the worst case:
  1st passage through the inner loop: \(n-1\) comparisons and \(n-1\) swaps
  
  ...  
  
  \((n-1)\)st passage through the inner loop: one comparison and one swap.

- All together: \(cn((n-1) + (n-2) + ... + 1)\), where \(c\) is the time required to do one comparison, one swap, check the inner loop condition and increment \(j\).

  We also spend constant time \(k\) declaring \(i,j,\text{temp}\) and initialising \(i\). Outer loop is executed \(n-1\) times, suppose the cost of checking the loop condition and decrementing \(i\) is \(c_i\).

Complexity of bubble sort

- Need to find \(n_0\) and \(K\), such that for all \(n \geq n_0\):
  \[
  \frac{1}{2} c \left( \frac{n^2-n}{n^2} \right) + \frac{1}{2} c(n-1) + k \leq K \cdot n^2
  \]

  \[
  \frac{1}{2} c \frac{n^2}{n^2} - \frac{1}{2} c n + c_n - c_i + k \leq \frac{1}{2} c n^2 + c_i n + k \leq \frac{c n^2}{n^2} + c_i n^2 + k n^2 \quad \text{for} \quad n \geq 1
  \]

  Take \(K = c + c_i + k\) and \(n_0 = 1\).

Complexity of bubble sort

- \(n^2(n-1)/2 + k + c_i(n-1)\)

  \((n-1) + (n-2) + ... + 1 = n(n-1)/2\)

  So our function equals

  \[c n^2(n-1)/2 + k + c_i(n-1) = \frac{1}{2} c (n^2-n) + c(n-1) + k\]

  Complexity \(O(n^2)\).

Bubble sort

```c
void bubbleSort(int arr[]){
    int i;
    int j;
    int temp;
    for(i = arr.length-1; i > 0; i--){
        for(j = 0; j < i; j++){
            if(arr[j] > arr[j+1]){  // swap adjacent elements, if in the wrong order
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        } // end inner loop
    } // end outer loop
} // end bubble sort
```

Bubble sort of lists

- Bubble sort is just as efficient (or rather inefficient) on linked lists.
- We can easily bubble sort even a singly linked list.
- Assume we have a class `Node` with fields:
  element of type `E` and `next` of type `Node`.
- (Strictly speaking getter/setter methods would be better, but this is just for the sake of brevity...)
- The `List` class just has `head` field.
Bubble sort of a linked list

Node border = null; // first node in the sorted part
while (border != head) {
    Node current = head; // start from the first node
    while (current.next != border) {
        if (current.element > current.next.element) {
            E element v = current.element;
            current.element = current.next.element;
            current.next.element = v;
        }
        current = current.next;
    }
    border = current; // the sorted part increases by one
}

Complexity of bubble sort on lists

• Same complexity as for arrays O(n^2):
  • First time we iterate until we see a null (swapping elements)
  • Second time we iterate until we see the last node;
  • … each time the border is one link closer to the head of the list
  • until border == head.

Selection sort

void selectionSort(int arr[]){
    int i, j, temp, pos_greatest;
    for( i = arr.length-1; i > 0; i--){
        pos_greatest = 0;
        for(j = 0; j <= i; j++){
            if( arr[j] > arr[pos_greatest])
                pos_greatest = j;
        } //end inner for loop
        temp = arr[i];
        arr[i] = arr[pos_greatest];
        arr[pos_greatest] = temp;
    } //end outer for loop
} //end selection sort

Complexity of selection sort

• Same number of iterations
• Same number of comparisons in the worst case
• fewer swaps (one for each outer loop = n-1)
• also O(n^2)

Selection sort on linked lists

• Implementation similar to bubble sort; also O(n^2)
• Instead of pos_greatest, have a variable Node largest which keeps the reference of the node with the largest element we have seen so far
• Swap elements once in every iteration through the unsorted part of the list:
  E element v = current.element;
  current.element =largest.element;
  largest.element= v;

Selection sort on linked lists

Insertion sort

void insertionSort(int arr[]){
    int i,j,temp;
    for(j=1; j < arr.length-1; j++){
        temp = arr[j];
        i = j; // range 0 to j-1 is sorted
        while(i > 0 && arr[i-1] >= temp){
            arr[i] = arr[i-1];
            i--;
        }
        arr[i] = temp;
    } // end outer for loop
} // end insertion sort

Insertion sort

Find a place to insert temp in the sorted range; as you are looking, shift elements in the sorted range to the right
Complexity of insertion sort

- In the worst case, has to make \( \frac{n(n-1)}{2} \) comparisons and shifts to the right
- also \( O(n^2) \) worst case complexity
- best case: array already sorted, no shifts.

Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don’t need to shift other nodes to make space for it (but need to find the place…):

```
10 ——> 12 ——> 23 ——> 5 ——> 14
```

Implementation of insertion sort on linked lists

```
border = head; // last node in the sorted part
while (border.next != null) {
    DNode toinsert = border.next;
    // do we need to move it?
    if (toinsert.element >= border.element) {
        border = border.next;
    }
else { // toinsert.element < border.element
    // remove toinsert from the list
    toinsert.prev.next = toinsert.next;
    toinsert.next.prev = toinsert.prev;
```

```
Implementation of insertion sort on linked lists

// find a place between head and border to insert toinsert, in order
if (head.element > toinsert.element) {
    toinsert.next = head;
    toinsert.prev = null;
    head.prev = toinsert;
    head = toinsert;
}

Implementation of insertion sort on linked lists

// find a place between head and border to insert toinsert, in order
DNode current = head.next;
while (current.element < toinsert.element) {
    current = current.next; // this is O(n) in worst case
} // now current.element >= toinsert.element
    toinsert.next = current;
    toinsert.prev = current.prev;
    current.prev.next = toinsert;
    current.prev = toinsert; // and close all brackets

Implementation of insertion sort on linked lists

• This is O(n²): in each iteration through the outer loop, we move border one step to the right
• In the inner loop, we iterate through the sorted portion, looking for a place to insert
• In the worst case (list sorted in reverse order) will have to do 1+2+…+n-1 comparisons in the sorted part.
• It would have been better (if we planned to use insertion sort for sorting almost sorted lists) to search for the insertion point not from the head of the list going towards the border, but from the border going towards the head, if the list is almost sorted, this would result in a shorter iteration.