Sorting

DS 2018/2019

Content

The sorting problem

- Case 1:
  - Input: n, $(v_0, \ldots, v_{n-1})$
  - Output: $(w_0, \ldots, w_{n-1})$ such that $(w_0, \ldots, w_{n-1})$ is a permutation of $(v_0, \ldots, v_{n-1})$ and $w_0 \leq \ldots \leq w_{n-1}$
- Case 2:
  - Input: n, $(R_0, \ldots, R_{n-1})$ with the keys $k_0, \ldots, k_{n-1}$
  - Output: $(R'_0, \ldots, R'_{n-1})$ such that $(R'_0, \ldots, R'_{n-1})$ is a permutation of $(R_0, \ldots, R_{n-1})$ and $R'_0.k_0 \leq \ldots \leq R'_{n-1}.k_{n-1}$
- Data structure
  Array $a[0..n-1]$
  $a[0] = v_0, \ldots, a[n-1] = v_{n-1}$

Content

Bubble-sort

- Basic principle:
  - (i, j) with $i < j$ is an inversion if $a[i] > a[j]$
  - While there is an inversion $(i, i+1)$ interchange $a[i]$ with $a[i+1]$

- Algorithm:

Procedure *bubbleSort(a, n)*
begin
    $last \leftarrow n - 1$
    while $(last > 0)$ **do**
        $n1 \leftarrow last - 1$; $last \leftarrow 0$
        for $i \leftarrow 0$ **to** $n1$ **do**
            if $(a[i] > a[i+1])$ **then**
                swap$(a[i], a[i+1])$
                $last \leftarrow i$
end

Bubble sort - example

3 **7** 2 1 4 (n1 = 3)
3 **7** **2** 1 4
3 **2** **7** 1 4
3 2 **7** **1** 4
3 2 **1** **7** 4
3 2 1 **7** **4**
3 2 1 **4** **7**
3 2 1 4 **7**

3 **2** 1 4 7 (n1 = 2)
2 **3** 1 4 7
2 **3** **1** 4 7
2 **1** **3** 4 7
2 1 **3** **4** 7
2 1 **3** **4** 7

2 **1** 3 4 7 (n1 = 0)
1 **2** 3 4 7
1 **2** 3 4 7

Bubble sort

- Analysis
  - Worst-case performance
    $a[0] > a[1] > ... > a[n-1]$
    Time for searching: $O(n-1+n-2+...+1) = O(n^2)$
    $T_{bubbleSort}(n) = O(n^2)$
  - Best-case performance: $O(n)$

Content

Insertion sort

- Basic principle:
  suppose $a[0..i-1]$ sorted
  insert $a[i]$ such that $a[0..i]$ becomes sorted

- Algorithm (search sequentially the position of $a[i]$):

Procedure *insertSort(a, n)*
begin
    for $i \leftarrow 1$ **to** $n-1$ **do**
        $j \leftarrow i$–1 // $a[0..i-1]$ sorted
        $temp \leftarrow a[i]$ // search the place of temp
        while $((j \geq 0)$ *and* $(a[j] > temp))$ **do**
            $a[j+1] \leftarrow a[j]$
            $j \leftarrow j$–1
        if $(a[j+1]! = temp)$ **then**
            $a[j+1] \leftarrow temp$
end

Insertion sort

- Example

  3 **7** 2 1

  3 7 **2** 1

  2 3 7 **1**

  1 2 3 7
- Analysis
  - searching the position i in $a[0..j-1]$ needs $O(j-1)$ steps
  - worst-case scenario $a[0] > a[1] > ... > a[n-1]$

    Search time: $O(1+2+...+n-1) = O(n^2)$

    $T_{insertSort}(n) = O(n^2)$
  - best-case scenario: $O(n)$

Content

Selection sort

- Apply the following scheme:
  - the current step: select an element and bring it on the final position in the sorted table;
  - repeat the current step until all elements reach the final positions.
- After the type of the selection of an element:
  - Naive selection: choose elements in the order they are initially (from $n-1$ to 0 or from 0 to $n-1$)
  - Systematic selection: use max-heap

Selection sort (naive selection)

- In the order $n-1, n-2, ..., 1, 0$, namely:
  $(\forall i) 0 \leq i < n \implies a[i] = max\{a[0], ..., a[i]\}$

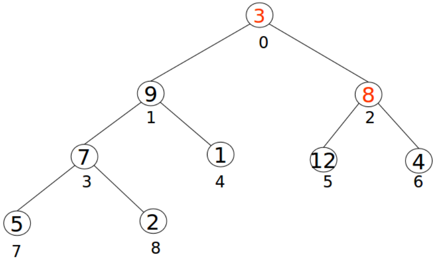
  Procedure *naivSort(a, n)*
  begin
      for $i \leftarrow n-1$ **downto** *1* **do**
          $imax \leftarrow i$
          for $j \leftarrow i-1$ **downto** *0* **do**
              if $(a[j] > a[imax])$ **then**
                  $imax \leftarrow j$
          if $(i! = imax)$ **then**
              swap($a[i]$, $a[imax]$)
  end
- The time complexity for all cases is $O(n^2)$

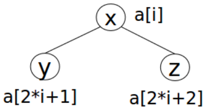
Heap sort (sorting by systematic selection)

Phase I

- organize the table like a max-heap: $(\forall k) 1 \leq k \leq n-1 \implies a[k] \leq a[(k-1)/2]$;
- initially the table satisfies the max-heap property starting with the position $n/2$;
- insert in max-heap the elements from the positions $n/2-1, n/2-2, \cdots, 1, 0$.

3	9	8	7	1	12	4	5	2
0	1	2	3	4	5	6	7	8

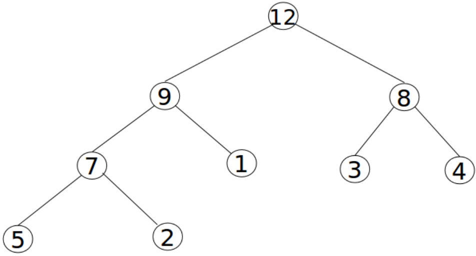




Heap sort (sorting by systematic selection)

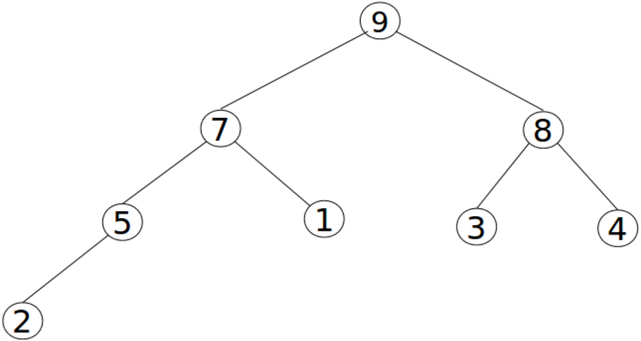
Phase II

- select the maximum element and bring it on his place by interchanging it with the last element;
- decrease n by 1 and then redo the max-heap;
- repeat the above steps until all elements reach their place.



Heap sort (sorting by systematic selection)

9	7	8	5	1	3	4	2	12
0	1	2	3	4	5	6	7	8



The operation of insertion in heap

```
Procedure insertTth(a, n, t)
begin
    j ← t
    heap ← false
    while ((2 * j + 1 < n) and not heap) do
        k ← 2 * j + 1
        if ((k < n − 1) and (a[k] < a[k + 1])) then
            k ← k + 1
        if (a[j] < a[k]) then
            swap(a[j], a[k])
            j ← k
        else
            heap ← true
end
```

Heap sort

Procedure *heapSort(a, n)*
begin
    // build the max-heap
    for $t \leftarrow (n-1)/2$ **downto** *0* **do**
        insertTth(a, n, t)
    // remove
    $r \leftarrow n - 1$
    while *(*$r > 0$*)* **do**
        swap($a[0]$, $a[r]$)
        insertTth(a, r, 0)
        $r \leftarrow r - 1$
end

Heap sort - Example

10	17	5	23	7	(n = 5)
10	**17**	$\underline{5}$	$\underline{23}$	$\underline{7}$	
10	$\underline{23}$	$\underline{5}$	$\underline{17}$	$\underline{7}$	
23	10	5	17	7	
$\underline{23}$	$\underline{17}$	$\underline{5}$	$\underline{10}$	$\underline{7}$	(max-heap n)

Heap sort - Example

$\underline{23}$	$\underline{17}$	$\underline{5}$	$\underline{10}$	$\underline{7}$	(max-heap n)
$\underline{7}$	$\underline{17}$	$\underline{5}$	$\underline{10}$	23	
$\underline{17}$	$\underline{10}$	$\underline{5}$	$\underline{7}$	23	(max-heap n-1)
$\underline{7}$	$\underline{10}$	$\underline{5}$	17	23	
$\underline{10}$	$\underline{7}$	$\underline{5}$	17	23	(max-heap n-2)
$\underline{5}$	$\underline{7}$	$\underline{10}$	17	23	
$\underline{7}$	$\underline{5}$	$\underline{10}$	17	23	(max-heap n-3)
$\underline{5}$	7	10	17	23	
$\underline{5}$	7	10	17	23	(max-heap n-4)
5	7	10	17	23	

Heap sort - complexity

- building the heap (suppose $n = 2^k - 1$)
  $\sum_{i=0}^{k-1} 2(k-i-1)2^i = 2^{k+1} - 2(k+1)$
- remove from heap and redo the heap
  $\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$
- complexity of the sorting algorithm
  $T_{heapSort}(n) = 2nlogn - 2n = O(nlogn)$

Content

Divide-et-impera paradigm

- $P(n)$: problem of dimension n
- base:
  - if $n \leq n_0$ then solve P by elementary methods
- divide-et-impera:
  - **divide** P in a problems $P_1(n_1), \ldots, P_a(n_a)$ cu $n_i \leq n/b, b > 1$
  - **solve** $P_1(n_1), \ldots, P_a(n_a)$ in the same way and obtain the solutions $S_1, \ldots, S_a$
  - **assemble** $S_1, \ldots, S_a$ to obtain the solution S to the problem P

Paradigma divide-et-impera: algorithm

Procedure *DivideEtImpera(P, n, S)*
begin
    if $(n \leq n_0)$ **then**
        find S by elementary methods
    else
        Divide P in $P_1, ..., P_a$
        DivideEtImpera(P_1, n_1, S_1)
        ...
        DivideEtImpera(P_a, n_a, S_a)
        Assemble$(S_1, ..., S_a, S)$
end

Merge sort

- generalization: $a[p..q]$
- base: $p \geq q$
- divide-et-impera
  - divide: $m = [(p + q)/2]$
  - subproblems: $a[p..m]$, $a[m + 1..q]$
  - assembling: interclass the sorted subsequences $a[p..m]$ și $a[m + 1..q]$
    - initially store the result of interclassing in *temp*
    - copy from $temp[0..q - p + 1]$ in $a[p..q]$
- complexity:
  - time: $T(n) = O(n\ log\ n)$
  - additional space: $O(n)$

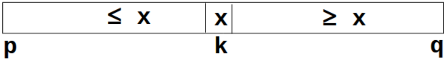
Interclassing two sorted sequences

- the problem:
  - given $a[0] \leq a[1] \leq \cdots \leq a[m-1]$, $b[0] \leq b[1] \leq \cdots \leq b[n-1]$, construct $c[0] \leq c[1] \leq \cdots \leq c[m+n-1]$ such that $(\forall k)((\exists i)c[k] = a[i]) \vee (\exists j)c[k] = b[j])$ and for $k! = p$, $c[k]$ and $c[p]$ come from different elements
- the solution
  - initially: $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
  - the current step:
    - if $a[i] \leq b[j]$ then $c[k] \leftarrow a[i]$, $i \leftarrow i+1$
    - if $a[i] > b[j]$ then $c[k] \leftarrow b[j]$, $j \leftarrow j+1$
    - $k \leftarrow k+1$
  - termination criterion: $i > m-1$ or $j > n-1$
  - if it is the case, copy in c the elements from the unfinished table

Content

Quick sort

- generalization: $a[p..q]$
- base: $p \geq q$
- divide-et-impera
  - divide: find k between p and q *by interchanges* such that after finding k we have:
    - $p \leq i \leq k \implies a[i] \leq a[k]$
    - $k < j \leq q \implies a[k] \leq a[j]$



  - subproblems: $a[p..k-1]$, $a[k+1..q]$
  - assembling: none

Quick sort: partitioning

- initially:
  - $x \leftarrow a[p]$ (may choose x arbitrarily from $a[p..q]$)
  - $i \leftarrow p + 1; j \leftarrow q$
- the current step:
  - if $a[i] \leq x$ then $i \leftarrow i + 1$
  - if $a[j] \geq x$ then $j \leftarrow j - 1$
  - if $a[i] > x > a[j]$ si $i < j$ then

    $swap(a[i], a[j])$

    $i \leftarrow i + 1$

    $j \leftarrow j - 1$
- termination:
  - the condition $i > j$
  - operations

    $k \leftarrow i - 1$

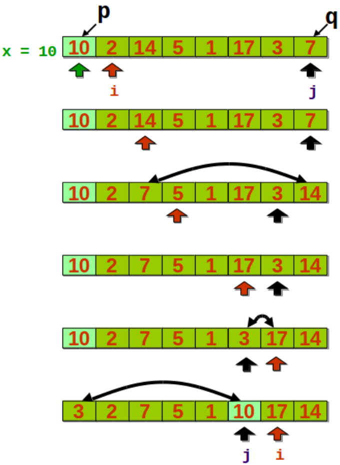
    $swap(a[p], a[k])$

Quick sort: partitioning - example

Procedure *partitioning(a, p, q, k)*
begin

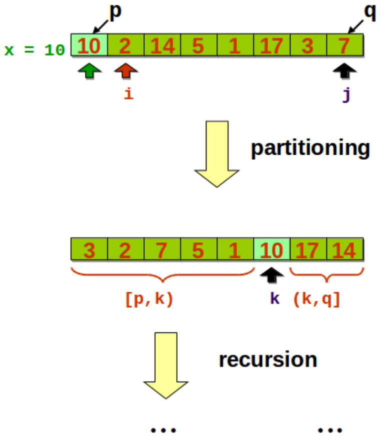
$x \leftarrow a[p]$
$i \leftarrow p + 1$
$j \leftarrow q$
while $(i <= j)$ **do**
  if $(a[i] <= x)$ **then**
    $i \leftarrow i + 1$
  if $(a[j] >= x)$ **then**
    $j \leftarrow j - 1$
  if $(i < j)$ *and* $(a[i] > x)$ *and* $(x > a[j])$
  then
    $swap(a[i], a[j])$
    $i \leftarrow i + 1$
    $j \leftarrow j - 1$
$k \leftarrow i - 1$
$a[p] \leftarrow a[k]$
$a[k] \leftarrow x$

end

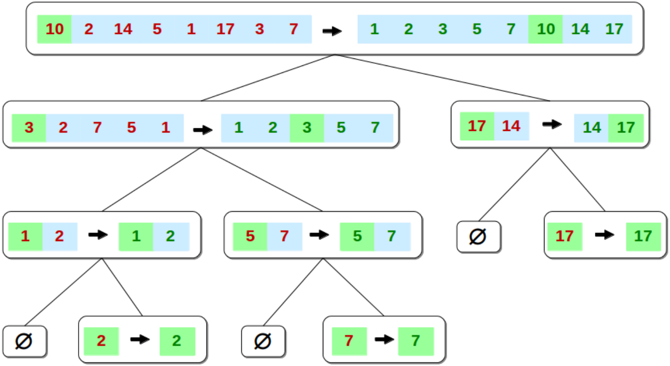


Quick sort: recursion - example

Procedure *quickSort(a, p, q)*
begin
    while $(p < q)$ **do**
        $partitioning(a, p, q, k)$
        $quickSort(a, p, k-1)$
        $quickSort(a, k+1, q)$
end

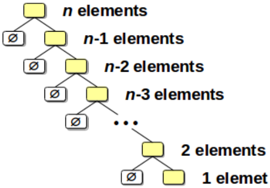


Quick sort: the recursion tree



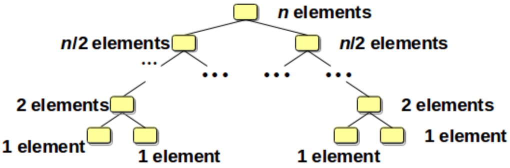
Quick sort - complexity

- The choice of the pivot influences the efficiency of the algorithm
- Worst-case scenario: the pivot is the smallest (the biggest) value. The time is proportional with $n + n - 1 + ... + 1$.
- $T_{quickSort}(n) = O(n^2)$
- The recursion tree:



Quick sort - complexity

- A "good" pivot divides the table in two subtables of comparable dimensions
- The height of the recursion tree is $O(log\ n)$
- The average complexity is $O(n\ log\ n)$



Content

Counting sort

- Assumption: $a[i] \in \{1, 2, \ldots, k\}$
- Find the position of each element in the sorted table by counting how many elements are lower than him

```
Procedure countingSort(a, b, n, k)
begin
    for i ← 1 to k do
        c[i] ← 0
    for j ← 0 to n − 1 do
        c[a[j]] ← c[a[j]] + 1
    for i ← 2 to k do
        c[i] ← c[i] + c[i − 1]
    for j ← n − 1 downto 0 do
        b[c[a[j]] − 1] ← a[j]
        c[a[j]] ← c[a[j]] − 1
end
```

Complexity: $O(k + n)$

Counting sort – example ($k = 6$)

	0	1	2	3	4	5	6	7
a	3	6	4	1	3	4	1	4

lines 5-6

	1	2	3	4	5	6
c	2	0	2	3	0	1

lines 7-8

	1	2	3	4	5	6
c	2	2	4	7	7	8

	0	1	2	3	4	5	6	7
b							4	

	1	2	3	4	5	6
c	2	2	4	6	7	8

lines 9-11, $j = 7$

	0	1	2	3	4	5	6	7
b		1					4	

	1	2	3	4	5	6
c	1	2	4	6	7	8

lines 9-11, $j = 6$

	0	1	2	3	4	5	6	7
b		1				4	4	

	1	2	3	4	5	6
c	1	2	4	5	7	8

lines 9-11, $j = 5$

---

sorted table:

	0	1	2	3	4	5	6	7
b	1	1	3	3	4	4	4	6

Content

Distribution sort

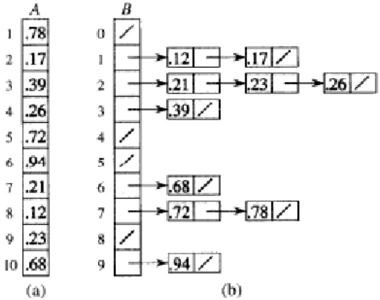
- Assumption: the elements $a[i]$ are uniformly distributed over the interval $[0, 1)$
- Principle:
  - divide the interval $[0, 1)$ in n subintervals of equal sizes, indexed from 0 to $n - 1$;
  - distribute the elements $a[i]$ in the corresponding interval: $\lfloor n \cdot a[i] \rfloor$;
  - sort each package using another method;
  - combine the n packages in a sorted list.

- Algorithm:

Procedure *bucketSort(a, n)*
begin
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $insert(B[\lfloor n \cdot a[i] \rfloor], a[i])$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        sort the list $B[i]$
        concatenate in order the lists $B[0], B[1], \cdots, B[n-1]$
    end

The average complexity: $O(n)$

Distribution sort – example



(Cormen T.H. et al., Introducere în algoritmi)

Sorting - complexity

Algorithm	Case		
	best-case	average	worst-case
bubbleSort	n	n^2	n^2
insertSort	n	n^2	n^2
naivSort	n^2	n^2	n^2
heapSort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$
mergeSort	$n\ log\ n$	$n\ log\ n$	$n\ log\ n$
quickSort	$n\ log\ n$	$n\ log\ n$	n^2
countingSort	$-$	$n + k$	$n + k$
bucketSort	$-$	n	$-$

When a sorting algorithm is favourite?

- A sorting method is *stable* if it keeps unchanged the relative order of elements with identical keys
- Recommendations
  - *Quick sort*: when you don't need a stable method and the average performance is more important than the worst one; $O(n\ log\ n)$ - the average time complexity, $O(log\ n)$ auxiliary space
  - *Merge sort*: when a stable method is required; time complexity $O(n\ log\ n)$; drawback: $O(n)$ auxiliary space, a larger constant than QuickSort
  - *Heap sort*: when you don't need a stable method and you are interested more in the worst case performance than the average one; time $O(n\ log\ n)$, space $O(1)$
  - *Insert sort*: when n is small

When a sorting algorithm is favourite?

- Under some limited conditions, it is possible a sorting in $O(n)$
  - *Counting sort*: the values are in an interval
  - *Bucket sort*: the values are approximately uniformly distributed