Unit 2
Sorting Algorithms

2.1 Local sorting algorithms

InsertSort

- The idea of InsertSort consists of having the i-1 first elements of the array sorted at the beginning of iteration i.

1	2		i-1	**i**	i+1		N
σ(1)	σ(2)	….	σ(i-1)	**σ(i)**	σ(i+1)	…..	σ(N)

sorted

element to be inserted

- During iteration **I**, element σ(i) is inserted in in the right place (in a position between index 1 and i) so that the first i elements of the array are sorted among them.

Element to be inserted

1	3	6	7	2	5	4

Sorted elements

Insert location

1	3	3	6	7	5	4

Inserted element

1	2	3	6	7	5	4

Sorted elements

InsertSort

```
InsertSort(array T, ind F, ind L)
   for i= F+1 to L;
        A=T[i];
        j=i-1;
        while (j ≥ F && T[j]>A);
                T[j+1]=T[j];
                j--;
        T[j+1]=A;
```

Basic operation (KC)

- Observations:
  - The work of the innermost loop depends on the input
  - The work on an input σ is:

$$n_{IS}(\sigma) = \sum_{i=2}^{N} n_{IS}(\sigma, i)$$

  - Furthermore: $1 \le n_{IS}(\sigma,i) \le i-1$

InsertSort: worst and best cases

- Since $1 \le n_{IS}(\sigma,i) \le i-1$, then $\forall\sigma\in\Sigma_N$:

$$\sum_{i=2}^{N} 1 \le \sum_{i=2}^{N} n_{IS}(\sigma,i) \le \sum_{i=2}^{N}(i-1) \Rightarrow N-1 \le n_{IS}(\sigma) \le \frac{N(N-1)}{2}$$

- Worst case
  - Step 1: Considering the above, $\forall\sigma\in\Sigma_N$, $n_{IS}(\sigma) \le N(N-1)/2$
  - Step 2: $\boldsymbol{n_{IS}([N,N-1,N-2,\ldots\ldots,1]) = N(N-1)/2}$

  $\Rightarrow W_{IS}(N)=N(N-1)/2$

- Best case
  - Step 1: Considering the above, $\forall\sigma\in\Sigma_N$, $n_{IS}(\sigma) \ge N-1$
  - Step 2: $\boldsymbol{n_{IS}([1,2,3,\ldots\ldots,N]) = N-1}$

  $\Rightarrow B_{IS}(N)=N-1$

InsertSort: average case I

- Using the definition of average case:

$$A_{IS}(N) = \sum_{\sigma \in \Sigma_N} p(\sigma) n_{IS}(\sigma) = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_{IS}(\sigma) = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \sum_{i=2}^{N} n_{IS}(\sigma, i) =$$

$$= \sum_{i=2}^{N} \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_{IS}(\sigma, i) = \sum_{i=2}^{N} \boxed{A_{IS}(N, i)}$$

Average number of operations of IS in iteration **i**

- Status of the array in iteration i:

1	2		i-1	**i**	i+1		N
σ(1)	σ(2)	….	σ(i-1)	**σ(i)**	σ(i+1)	…..	σ(N)

Sorted elements

Element to be inserted

InsertSort: average case II

- Observation: when IS considers σ(i), this element can end up in positions

  i, i-1, i-2, …, j, …, 2, 1

  requiring

  1, 2, 3, …, i-j+1, …, i-1 e i-1

  key comparisons (KCs), respectively

Final position	"Lost" KCs (σ(**i**)<σ(j))	"Won" KCs (σ(**i**)>σ(j))	Total KCs
i	0	1 (σ(**i**)>σ(i-1))	1=i–i+1
i-1	1 (σ(**i**)<σ(i-1))	1 (σ(**i**)>σ(i-2))	2=i–(i-1)+1
i-2	2 (σ(**i**)<σ(i-1), σ(**i**)<σ(i-2))	1 (σ(**i**)>σ(i-3))	3=i–(i-2)+1
…j…	i-j	1 (σ(**i**)>σ(j-1)	i–j+1
3	i-3 (σ(**i**)<σ(i-1),…, σ(**i**)<σ(3))	1 (σ(**i**)>σ(2))	i-2=i–3+1
2	i-2 (σ(**i**)<σ(i-1),…, σ(**i**)<σ(2))	1 (σ(**i**)>σ(1))	i-1=i–2+1
1	i-1 (σ(**i**)<σ(i-1),…, σ(**i**)<σ(1))	0	i-1

Escuela Politécnica Superior

InsertSort: average case III

- Thus, $n_{IS}(i \rightarrow j) = i-j+1$ when $1<j\leq i$ and $n_{IS}(i \rightarrow 1) = i-1$, where $n_{IS}(i \rightarrow j)$ is the number of KCs needed to insert element $\sigma(i)$ in the j-th position.

- An alternative expression for the average case in iteration i

$$A_{IS}(N,i) = \sum_{j=1}^{i} p(j) n_{IS}(i \rightarrow j)$$

- Question: what is the probability that $\sigma(i)$ ends up in position j?

InsertSort: average case IV

- If all σ are equiprobable, it is reasonable to assume that P(σ(i) ends up in j) is also equiprobable.
- That is, P(σ(i) ends up in j) = 1/i for all j between 1 and i.
- Hence we can deduce that the mean work ***A_{IS} (N, i) of IS on the i-th input of an array of N elements is i/2 + O(1)***
- And thus ***$A_{IS}(N) = N^2/4 + O(N)$***
- In more detail…

InsertSort: average case V

- Recalling that $A_{IS}(N,i) = \sum_{j=1}^{i} p(j) n_{IS}(i \rightarrow j)$ where p(j) is the probability that element σ(i) ends up in position j.
- We assume **equiprobability** so that p(j) = 1/i (j=1,2,…i) , and thus we have:

$$A_{IS}(N,i) = \frac{1}{i}\sum_{j=1}^{i} n_{IS}(i \rightarrow j) = \frac{1}{i}\left[\left(\sum_{j=1}^{i-1}(i-j)\right) + (i-1)\right] = \frac{i-1}{2} + \frac{i-1}{i}$$

- Since $A_{IS}(N) = \sum_{i=2}^{N} A_{IS}(N,i)$, substituting the above yields:

$$A_{IS}(N) = \sum_{i=2}^{N}\left(\frac{i-1}{2} + \frac{i-1}{i}\right) = \frac{1}{2}\sum_{i=1}^{N-1} i + \sum_{i=1}^{N-1}\frac{i-1}{i} = \frac{N^2}{4} + O(N)$$

Summarizing InsertSort

- We know that

  $$W_{IS}(N) = N^2/2 + O(N)$$

  $$A_{IS}(N) = N^2/4 + O(N)$$

- Conclusion: IS is a little bit better than SS and BS, but not too much in the average case and the same in the worst case.
- **Question: Have we reached a limit in efficacy for sorting?**

Abstract runtime lower bounds

- How much can we improve a key-comparison based sorting algorithm?
- Obviously, we know that $n_A(\sigma) \geq N$, and thus $n_A(\sigma) = \Omega(N)$.
- But, is there a universal $f(N)$ such that $n_A(\sigma) \geq f(N)$ for any A?
- Is there any algorithm that reaches that lower bound?
- If it exists, how is this algorithm and in what conditions reaches the lower bound?
- Tool: a measure of disorder in an array.

How to measure disorder in an array?

- The operations that an algorithm performs depend on the order of the array.
- How to measure the order or disorder in an array?
- **Definition:**
  - We say that two indexes $i<j$ form an **inversion** if $\sigma(i) > \sigma(j)$
- Example: $\sigma=(3\ 2\ 1\ 5\ 4)$

  1 2 3 4 5
  - Inversion in $\sigma=((1,2),(1,3),(2,3),(4,5))$ => 4 inversions.
  - $Inv(\sigma)=4$
- In practice, we say that "3 forms an inversion with 2" instead that "the indexes 1 and 2 form an inversion".
- That is, **$\sigma(i)$ forms an inversion with $\sigma(j)$ if $\sigma(i)>\sigma(j)$ but $i<j$**

How to measure disorder in an array?

- Observations:
  1. inv([1,2,3,…..,N-1,N])=0
  2. inv([5,4,3,2,1])=10
  3. inv([N,N-1,N-2,….,2,1])=(N-1)+….+2+1=$N^2/2-N/2$

Obs: There cannot be any permutation **with more inversions** than σ= [N,N-1,N-2,….,2,1] since

$$inv([\sigma(1), \sigma(2),\ldots, \sigma(N)])= \underset{\substack{\wedge\!| \\ N-1}}{inv(\sigma(1))}+\underset{\substack{\wedge\!| \\ N-2}}{inv(\sigma(2))}+ \ldots+\underset{\substack{\wedge\!| \\ 1}}{inv(\sigma(N-1))}+inv(\sigma(N))\leq$$

$$\leq(N-1)+(N-2)+\ldots+1=N^2/2-N/2$$

Lower bounds for local sorting algorithms

- **Definition**: A sorting algorithm that uses key comparisons (KC) is **local** if **for each KC** that the algorithm performs it **fixes at most one inversion**.
- InsertSort, BubbleSort and (*morally*) SelectSort are local.
- **Obs:** If A is a local algorithm, the minimum number of KC that A performs is the number of inversions in the array σ to be sorted, i.e., $n_A(\sigma) \geq inv(\sigma)$.
- **Worst case:** ***If A is local, $W_A(N) \geq N^2/2-N/2$***

$$W_A(N) \geq n_A([N,N-1,N-2,\ldots,2,1]) \geq inv([N,N-1,N-2,\ldots,2,1]) = N^2/2-N/2$$

- **Thus:** IS, BS y SS **are optimal** in the **worst case** among local sorting algorithms.

Lower bound for the average case

- **Definition:** If $\sigma \in \sum_N$ we define its transpose, σ^t, as $\sigma^t(i) = \sigma(N-i+1)$.
- Example $\sigma=[3,2,1,5,4]$ then $\sigma^t=[4,5,1,2,3]$
- Observations:
  - $(\sigma^t)^t = \sigma$
  - inv([3,2,1,5,4])+inv([4,5,1,2,3])=4+6=10=(5*4)/2
  - inv([5,4,3,2,1])+inv([1,2,3,4,5])=10+0=10=(5*4)/2
- **Proposition**: If $\sigma \in \sum_N$

  $$inv(\sigma)+ inv(\sigma^t)=N(N-1)/2$$

Demo: If $1 \le i < j \le N$, either (i,j) is in inversion in σ or (N-j+1, N-i+1) is in inversion in σ^t => each pair (i,j) adds 1 to inv(σ)+ inv(σ^t) and there are N(N-1)/2 such pairs.

Lower bound for the average case

- ***If A is local $A_A(N) \geq N^2/4+O(N)$***

$$A_A(N)=\frac{1}{N!}\sum_{\sigma\in\Sigma_N} n_A(\sigma) \overset{\text{A local}}{\geq} \frac{1}{N!}\sum_{\sigma\in\Sigma_N} inv(\sigma)=\frac{1}{N!}\sum_{\sigma,\sigma^t}\left(inv(\sigma)+inv(\sigma^t)\right)=$$

$$=\frac{1}{N!}\frac{N(N-1)}{2}\sum_{\sigma,\sigma^t}1=\frac{1}{N!}\frac{N(N-1)}{2}\frac{N!}{2}=\frac{N^2}{4}+O(N)$$

- InsertSort is **optimal** for the average case among local sorting algorithms.

Local sorting algorithms are rather inneficcient.

In this section we have learnt…

- How InsertSort works, and the estimation of its best, worst and average cases.
- The concept of local sorting algorithms, and their lower bounds for the worst and average cases.

Tools and techniques to work on…

- InsertSort evolution on tables.
- Local algorithms' evolution on input arrays.
- Best, worst and average cases of InsertSort and similar algorithms and variants.
- Identification and counting of permutation inversions.
- Lower bound for the runtime of local algorithms on specific permutations.
- Exercises to solver (at least !!!): those recommended in sections 3, 4 y 5.

2.2 Recursive Sorting Algorithms

Divide and Conquer Methods (D&C)

- The idea of divide and conquer sorting methods is the following:
  - Divide the array/table T in two subarrays T_1 y T_2
  - Sort T_1 and T_2 recursively.
  - Combine the already sorted T_1 and T_2 in an also sorted T.
- General pseudocode of a D&C algorithm

```
D&CSort(table T)
  if dim(T)≤Mindim:
     directSort(T);
  else :
     Divide(T,T1,T2);
     D&CSort(T1);
     D&CSort(T2);
     Combine(T,T1,T2);
```

- A first option could consist in implementing a simple **Divide** function and a complex **Combine** routine.
- Result: **MergeSort.**

MergeSort

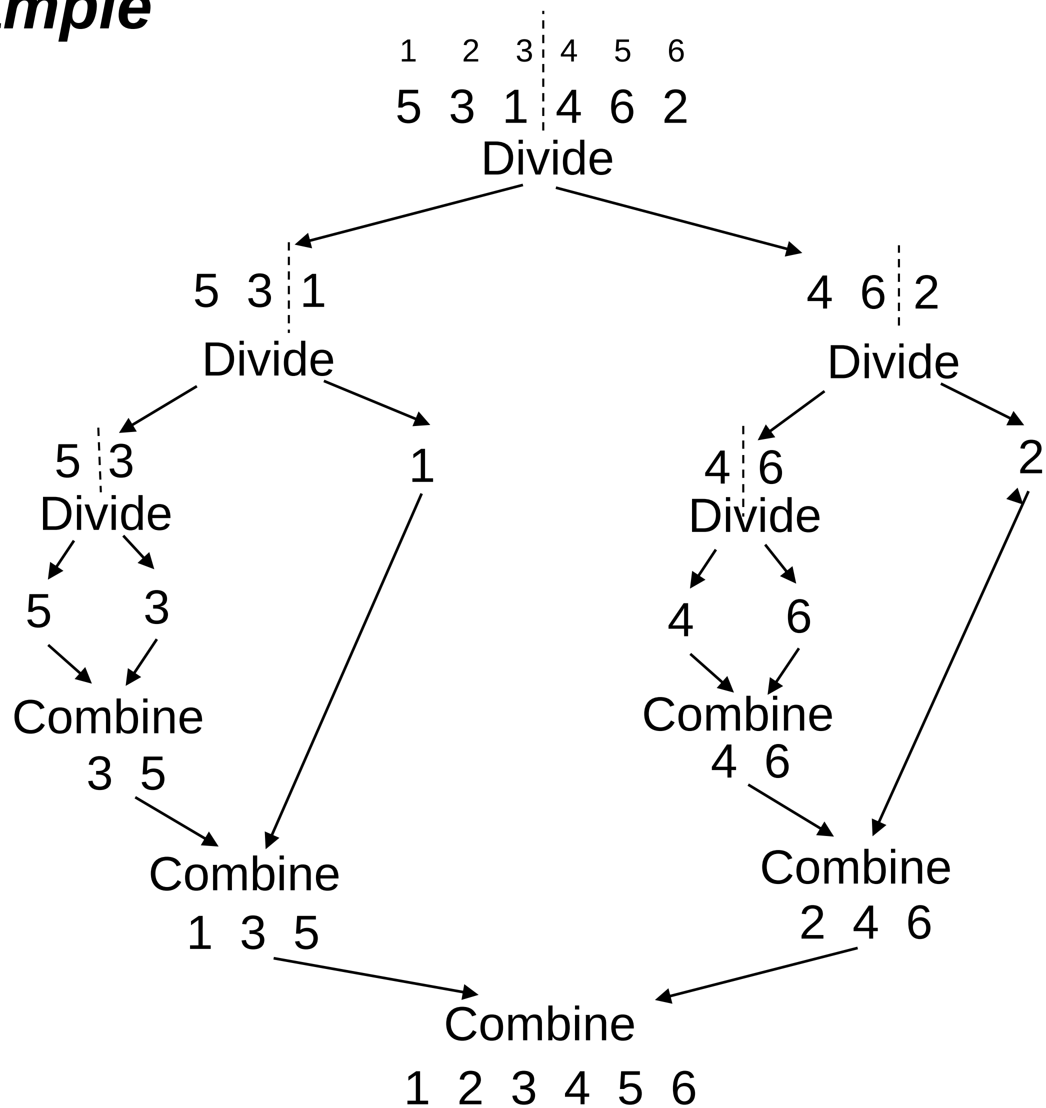
```
status MergeSort(array T, ind F, ind L)
  if F>L:
     return ERROR;
  if F==L: //array with only one element
     return OK;
  else:
     M=⌊(F+L)/2⌋;          // "divide"
     MergeSort(T,F,M);
     MergeSort(T,M+1,L);
     return Combine(T,F,M,L)
```

This requires dynamic memory

This is a first version, it needs to be polished before the implementation

MergeSort

- ***Example***



MergeSort: Combine

```
status Combine(array T,ind F, ind M, ind L)
      T'=AuxTable(F,L);
      If T'==NULL: return Error;
      i=F;j=M+1;k=F;
      while i≤M and j≤L:
          if T[i]<T[j]: T'[k]=T[i];i++;
          else: T'[k]=T[j];j++;
          k++;
      if i>M:     // copy the rest of the right subtable
          while j≤L:
              T'[k]=T[j];j++;k++;
      else if: j>L: // copy the rest of the left subtable
          while i≤M:
              T'[k]=T[i];i++;k++;
      Copy(T',T,P,L);
      Free(T');
      return T;
```

Auxiliary table with indexes F to L

Basic operation in Combine and also in MS

Copy T' on T between indexes F and L

Combine: Example

	1	2	3	4	5	6			1	2	3	4	5	6
	i			j					k					
T	1	3	5	2	4	6		T’						

T[1]<T[4]
i++,k++

	1	2	3	4	5	6			1	2	3	4	5	6
		i		j						k				
T	1	3	5	2	4	6		T’	1					

T[3]>T[4]
j++,k++

	1	2	3	4	5	6			1	2	3	4	5	6
		i			j						k			
T	1	3	5	2	4	6		T’	1	2				

T[2]<T[5]
i++,k++

	1	2	3	4	5	6			1	2	3	4	5	6
			i		j							k		
T	1	3	5	2	4	6		T’	1	2	3			

T[3]>T[5]
j++,k++

	1	2	3	4	5	6			1	2	3	4	5	6
			i			j								k
T	1	3	5	2	4	6		T’	1	2	3	4	5	

T[3]<T[6]
i++,k++
i>M

	1	2	3	4	5	6			1	2	3	4	5	6
				i		j								k
T	1	3	5	2	4	6		T’	1	2	3	4	5	6

Finally, we copy T’ on T and free T’

MergeSort: Runtime

- Observations
  1. BO: in Combine T[i]<T[j]
  2. $n_{MS}(\sigma)= n_{MS}(\sigma_l)+ n_{MS}(\sigma_r)+ n_{Combine}(\sigma, \sigma_l , \sigma_r)$
  3. $size(\sigma_l)=\lceil N/2 \rceil; size(\sigma_r)=\lfloor N/2 \rfloor$
  4. $\lfloor N/2 \rfloor \leq n_{Combine}(\sigma, \sigma_l , \sigma_r)\leq N-1$
- **With this observations we have:**

$$W_{MS}(N) \leq W_{MS}(\lceil N/2 \rceil) + W_{MS}(\lfloor N/2 \rfloor) + N-1;$$

$$W_{MS}(1)=0.$$

- First example of **recurrent inequality**

Recursive case: $T(N) \leq T(N_1)+T(N_2)+ \ldots..+T(N_k) +f(N)$, with $N_k<N$

Base case: $T(1)=X$ (X constant).

MergeSort: abstract runtime, worst case

- How to solve a recurrent inequality?
  - **Step 1**: We obtain a solution for a special case, e.g., a case that is particularly easy to calculate.
  - For example, for MS, we can consider $N=2^k$
    - ***In MergeSort with $N=2^k$ we have the following recurrent inequality:***

      $$\boxed{W_{MS}(N) \leq 2W_{MS}(N/2)+N-1 \text{ and } W_{MS}(1)=0}$$

    - ***By repeated substitution we have:***

      $W_{MS}(N) \leq N\lg(N)+O(N)$.
  - **Step 2**: We demonstrate that the expression obtained in Step 1 is valid for all N by **induction**.

Note: It is important to follow the steps that lead to the solution during the class or in the course notes.

MergeSort: Runtime, best and average cases

- With a similar reasoning we have:
  $B_{MS}(N) \geq B_{MS}(\lceil N/2 \rceil) + B_{MS}(\lfloor N/2 \rfloor) + \lfloor N/2 \rfloor$ and
  $B_{MS}(1)=0$
- **Considering again $N=2^k$ we obtain the recurrent inequality: *$B_{MS}(N) \geq 2B_{MS}(N/2)+N/2$ and $B_{MS}(1)=0$***
- ***Solving the above inequality: $B_{MS}(N) \geq (1/2)N\lg(N)$***
- To estimate the average case we can observe that:

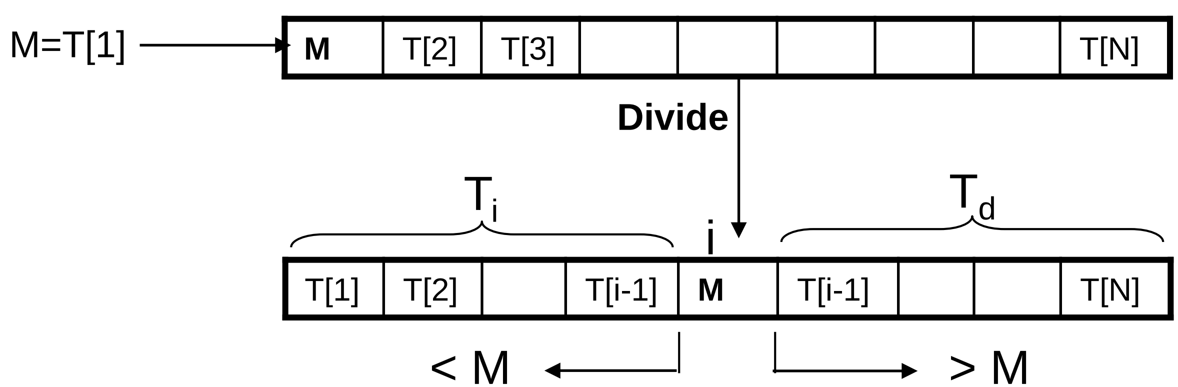
$(1/2)N\lg(N) \leq B_{MS}(N) \leq A_{MS}(N) \leq W_{MS}(N) \leq N\lg(N)+O(N)$, so that: $\boxed{A_{MS}(N)=\Theta(N\lg(N))}$

- The runtime is good, but the algorithm needs **dynamic memory** and also has **extra costs because of the recursive approach**.

QuickSort

- In Quicksort (QS), we use a **Divide** function that deals with the sorting and makes the function **Combine** unnecessary**.**
- The idea of **Divide** in QS consists in choosing and element M=T[m] in the array to be sorted (pivot).
- After Divide, **elements in the array are sorted with respect to M**=>no need for Combine.

Example:



QuickSort: pseudocodes

```
status QS(array T, ind F, ind L)
  si F>L:
    return ERROR;
  if F==L:
    return OK;
  else:
    M=Divide(T,F,L);
    if F<M-1:
      QS(T,F,M-1);
    if M+1 < L:
      QS(T,M+1,L);
  return OK;
```

```
ind Divide(array T, ind F, ind L)
  M=Mid(T,F,L);        <---- Pivot
  k=T[M];
  swap(T[F],T[M]);
  M=F;
  for i=F+1 to L:
    if T[i]<k:
      M++;
      swap(T[i],T[M]);
  swap(T[F],T[M]);
  return M;
```

QuickSort: pivot selection

- There are several options to choose the pivot:
  - The first element in the array (return F).
  - The last element (return L).
  - The position in the middle of the table (return (P+U)/2).
  - A random position between the first and the last element in the array (return rand(P,U)).
- **It is not guaranteed** that the value of the pivot is approximately the middle value of the array.

Example

	1	2	3	4	5	6	
	4	5	3	1	6	2	M=1,k=4 swap(T[1],T[1])
i=2	**4**	5	3	1	6	2	T[2]>k
i=3	**4**	5	3	1	6	2	T[3]<k, M=2 swap(T[2],T[3])
i=4	**4**	3	5	1	6	2	T[4]<k, M=3 swap(T[3],T[4])
i=5	**4**	3	1	5	6	2	T[5]>k
i=6	**4**	3	1	5	6	2	T[6]<k, M=4 swap(T[4],T[6])
	4	3	1	2	6	5	swap(T[1],T[4])
	2	3	1	**4**	6	5	return 4
		<				>	

QS: Abstract runtime in the worst case

- Observations
  1. BO: in Divide "if T[i]<k"
  2. $n_{QS}(\sigma)= n_{QS}(\sigma_l)+ n_{QS}(\sigma_r)+ n_{Divide}(\sigma)$
  3. $n_{Divide}(\sigma)$=N-1 (If Mid returns P, $n_{Mid}(\sigma)=0$)
- Thus, if σ_l has k elements, σ_r has N-1-k elements and then

  $n_{QS}(\sigma) \leq N-1+ W(k) + W(N-1-k)$

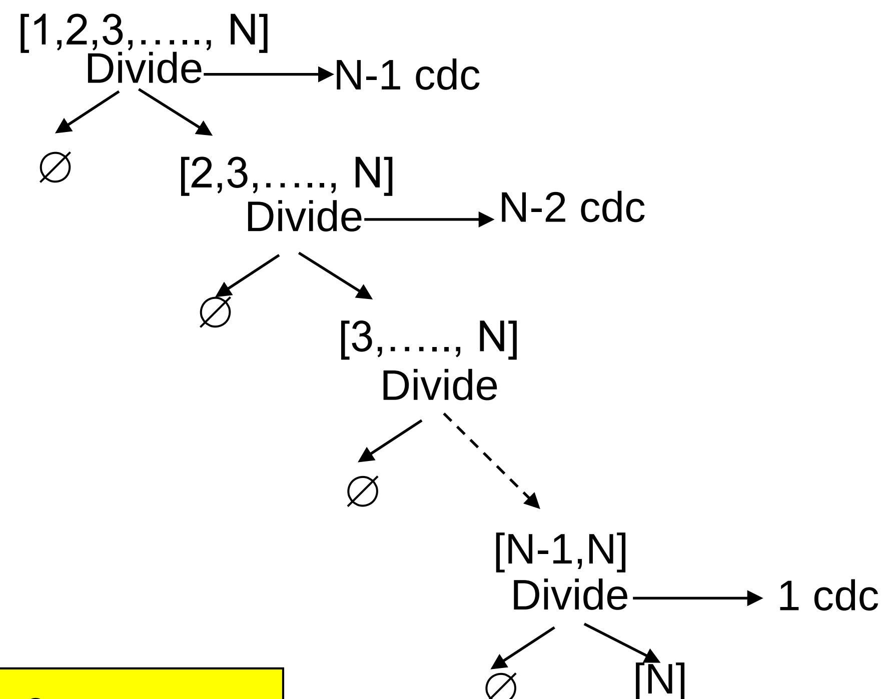
  $\leq N-1+ \max_{k = 1,\ldots, N-1} \{ W(k) + W(N-1-k) \}$
- So,

  $$W(N) \leq N-1+ \max_{k = 1, \ldots, N-1} \{ W(k) + W(N-1-k) \}$$
- And we can demonstrate by induction that

$$W(N) \leq N^2/2-N/2$$

QS: Abstract runtime in the worst case

- And also $W_{QS}(N) \geq N^2/2-N/2$.
  Considering T=[1,2,3,….., N]



- Thus, we have:
  $n_{QS}([1,2,3,\ldots,N])=$
  $(N-1)+(N-2)+\ldots..+1=$
  $N^2/2-N/2$

Then $\boxed{W_{QS}(N) = N^2/2-N/2}$

QS: Average case

- Again we have $n_{QS}(\sigma)= n_{QS}(\sigma_l)+ n_{QS}(\sigma_r)+ N-1$.
- We approximate $n_{QS}(\sigma_l)\cong A_{QS}(i-1)$ and $n_{QS}(\sigma_r)\cong A_{QS}(N-i)$

  Then we have $n_{QS}(\sigma) \cong A_{QS}(i-1) + A_{QS}(N-i) + N-1$.
- We obtain the following approximate recurrent expression:

$$A_{QS}(N) = (N-1) + \frac{1}{N}\sum_{i=1}^{N}[A_{QS}(i-1) + A_{QS}(N-i)]$$

$$A(1) = 0$$

- It can be demonstrated that

$$\boxed{A_{QS}(N) = 2N\log(N) + O(N)}$$

Note: It is important to follow the steps that lead to the solution during the class or in the course notes.

In this section we have learnt…

- Quick and MergeSort divide & conquer algorithms.
- Their runtime equations in the worst and in the average cases.
- How to solve such equations.
- How to write the abstract runtime equations of recursive algorithms.
- How to estimate the solution of recurrent equations:
  - Estimating a solution for a special case by repeated substitution.
  - Obtaining a solution for all cases by induction.

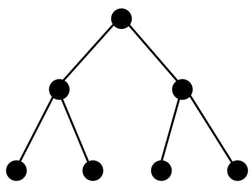
Tools and techniques to work on…

- Operation of MergeSort and QuickSort.
- Worst, average and best cases of MS, QS and associated variants.
- Estimation of growth function in recurrent inequalities.
- Writing abstract runtime equations of recurrent algorithms and solving them.
- Problems to solve (at least !!!): Those recommended in sections 6, 7 y 8.

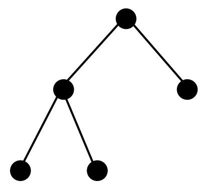
2.3 HeapSort

HeapSort

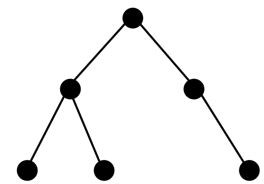
- **Definition:** A heap is an almost complete binary (i.e., it has only room at the rightmost elements in the last level).



Complete binary tree



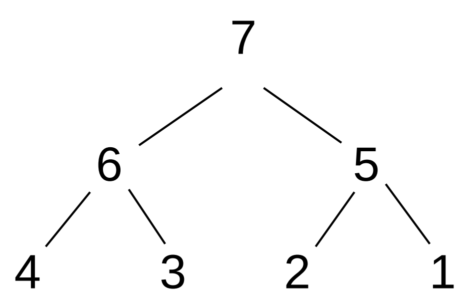
Almost complete binary tree (heap)



Binary tree (non heap)

- **Definition:** A Max-heap is a heap such that ∀ subtree T' of T: info(T')>info(T'_l), info(T'_r)

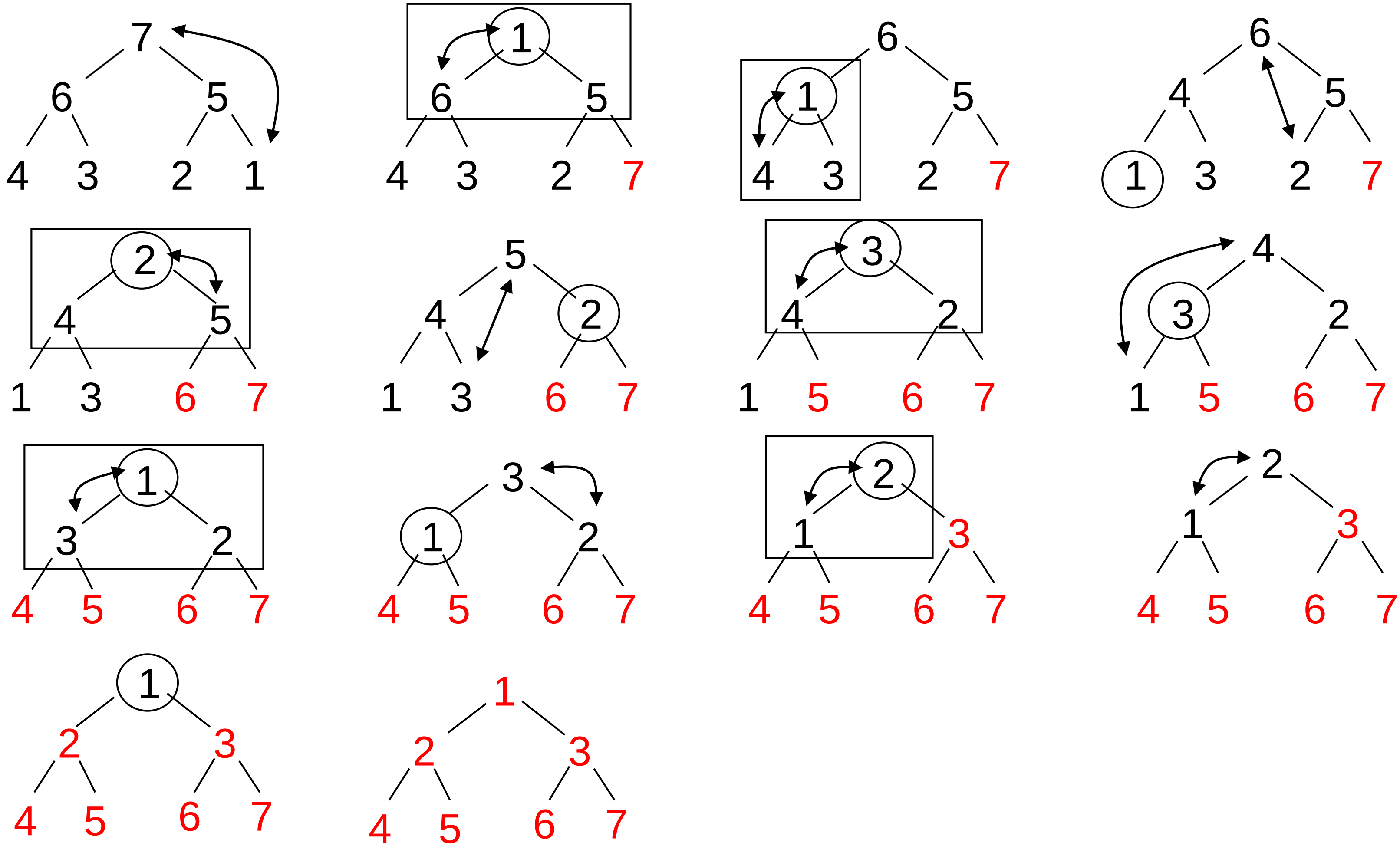
Example:



- **Observation:** A max-heap is easy to sort.

Sorting a max-heap.

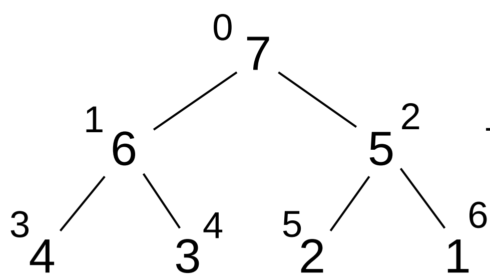
1. Swap the root node with the last node in the heap (lowest rightmost node).
2. Heapify the new root node to keep the max-heap condition.



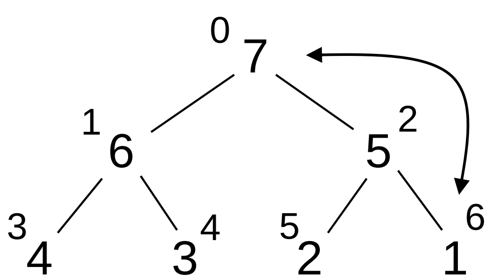
HeapSort: sorting in arrays

- Observations:
  1. If we go over a maxheap from top-down and left-to-right we have a sorted array.
  2. The nodes of a maxheap can be placed in an array so that the sorting method is in-place.

Example:



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



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

swap(T[0],T[6])

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

HeapSort: Maxheap array representation

- Parent → Left child and Parent → Right child

P	C_l	C_r
0	1	2
1	3	4
2	5	6
…	…	…

→

Parent	Left child	Right child
j	2j+1	2j+2

- Child → Parent

C	P
1	0
2	0
3	1
4	1
5	2
6	2
…	…

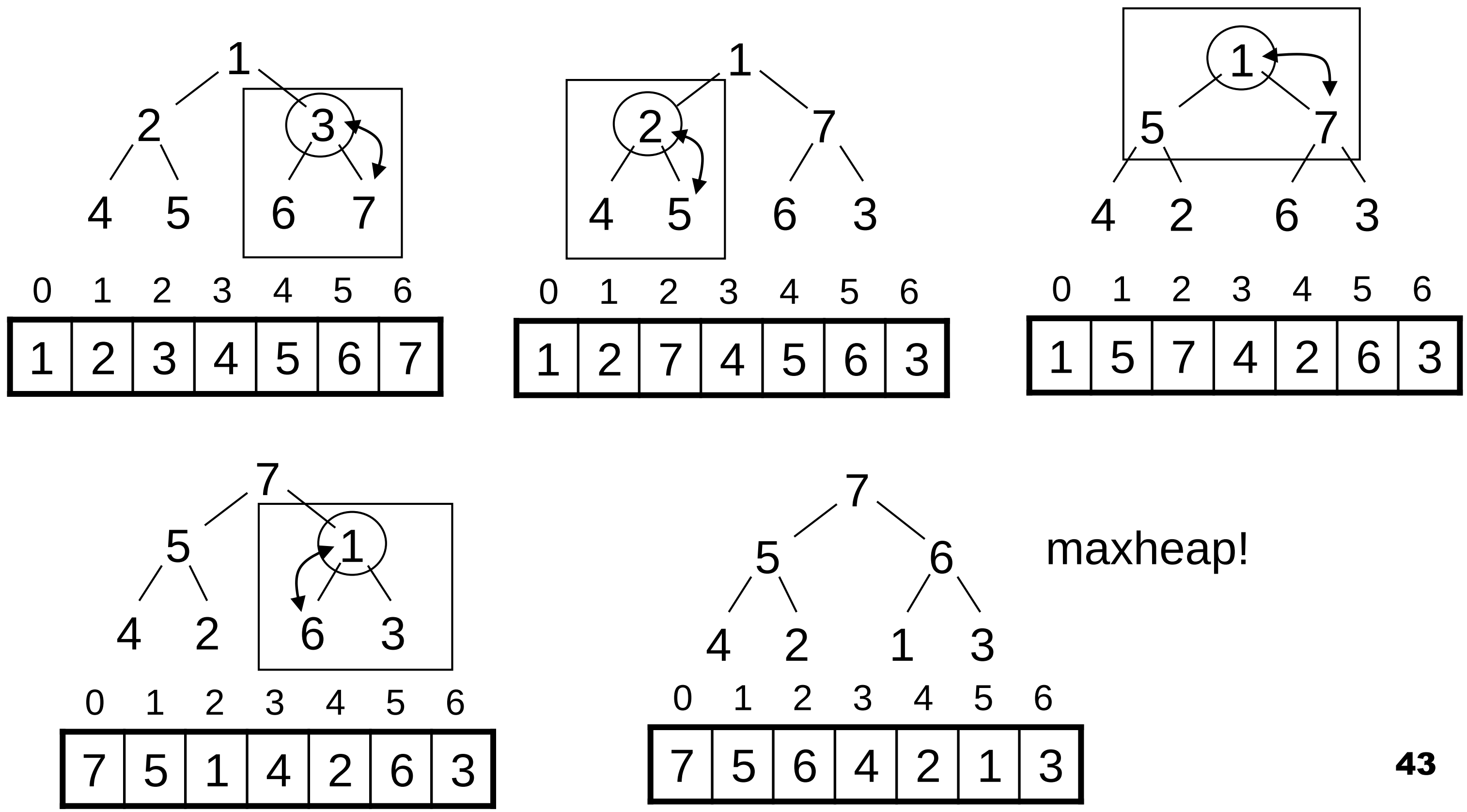
→

Child	Parent
j	$\lfloor (j-1)/2 \rfloor$

HeapSort: Creating the max heap

- The discussed process allows sorting a maxheap.
- How can we create a maxheap from a given array?
  - We keep the maxheap condition in all internal nodes (nodes that have at least one children) from right to left and from bottom to top.

Example:



HeapSort: Pseudocode

HeapSort

```
HeapSort(array T, int N)
   CreateMaxHeap(T,N);
   SortMaxHeap(T,N);
```

CreateMaxHeap

```
CreateMaxHeap(array T, int N)
   if N==1:
      return ;
   for i = ⌊N/2⌋ -1 to 0 :
      heapify(T,N,i);
```

SortMaxHeap

```
SortMaxHeap(array T, int N)
   for i=N-1 to 1 :
      swap(T[0],T[i]);
      heapify(T,i,0);
```

heapify

```
heapify(array T, int N, ind i)
  while  ( 2*i+2 ≤ N) :
     ind=max(T, N, i, 2*i+1, 2*i+2);
     if (ind ≠ i) :
       swap( T[i], T[ind] ) ;
        i = ind ;
     else :
        return;
```

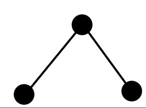
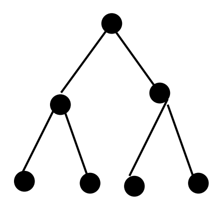
where routine

max(T, N, i, 2*i+1,2*i+2)

Returns the index of the element in array T which contains the larger value in i, 2*i+1,2*i+2 (parent, left and right children)

Height of maxheaps

- **Observation:** If T is a heap with N nodes, Height(T)= $\lfloor \log(N) \rfloor$

# of nodes N	Heap example	Height
1		0
2, 3		1
4, 5, 6, 7		2
…		…

HeapSort: abstract runtime

- Observations:
  - $n_{HeapSort}(T)=n_{CreateHeap}(T)+n_{SortHeap}(T)$
  - The maximum number of key comparisons that CreateHeap and SortHeap perform on a node is Height(T).
  - Height(T)= $\lfloor \log(N) \rfloor$ since T is almost complete.
- $n_{CreateHeap}(T) \leq N \lfloor \log(N) \rfloor$ y $n_{SortHeap}(T) \leq N \lfloor \log(N) \rfloor$
- $W_{HS}(N)=O(N\log(N))$
- $n_{CreateHeap}([1,2,\ldots,N])=N\log(N)$

$$\left.\begin{array}{l} W_{HS}(N)=O(N\log(N)) \\ n_{CreateHeap}([1,2,\ldots,N])=N\log(N) \end{array}\right\} W_{HS}(N)=\Theta(N\log(N))$$

- This method is non-recursive.
- It is the most efficient sorting method so far in our analysis!

In this section we have learnt…

- The concept of Maxheap and how to build it.
- The HeapSort algorithm and its abstract runtime.

Tools and techniques to work on

- Maxheap construction
- Application of the HeapSort algorithm
- Problems to solve (at least !!!): those recommended in section 9.

2.4 Decision trees for sorting algorithms

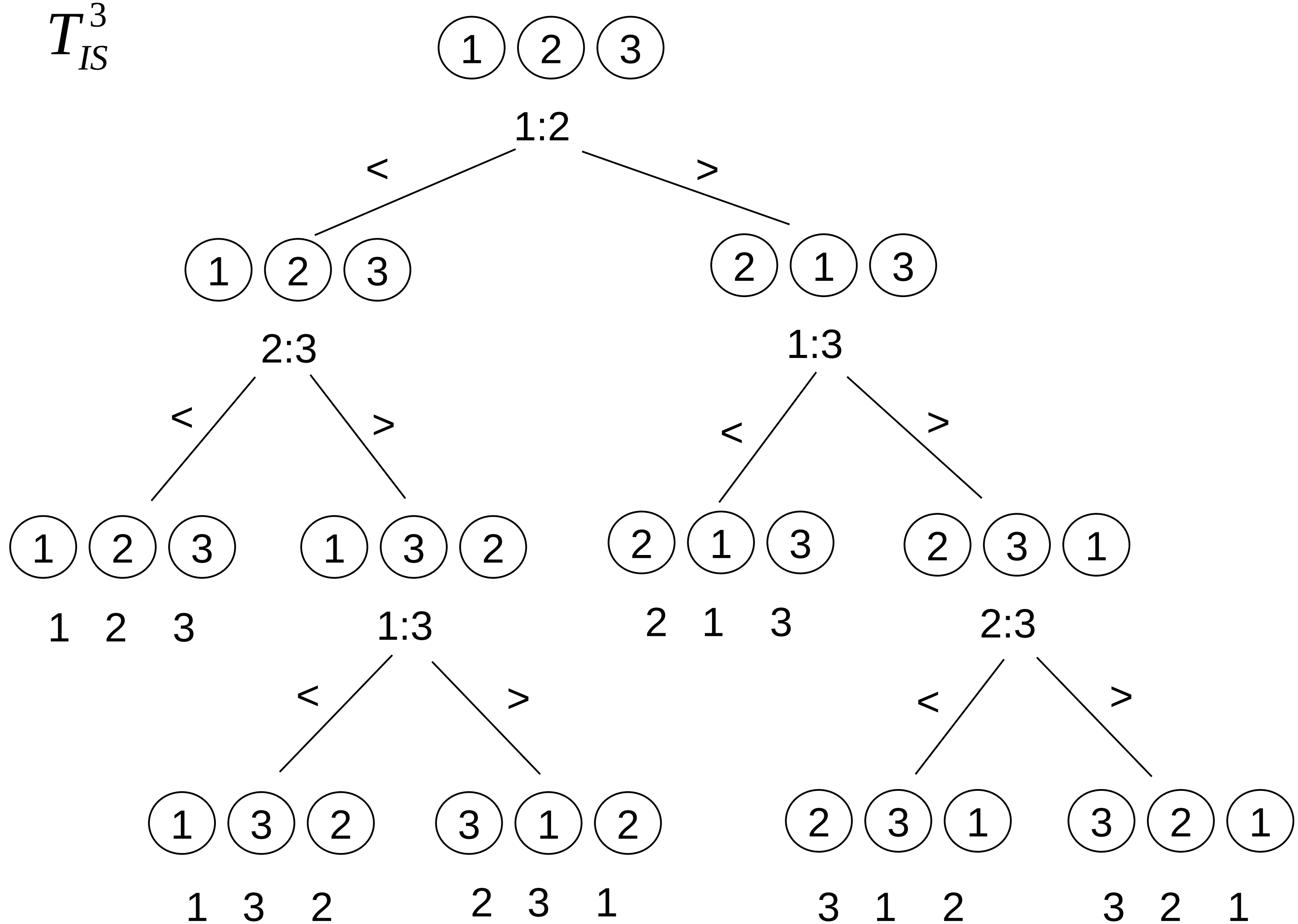
Lower bounds for sorting algorithms that use key comparisons

- So far the best sorting algorithm that uses key comparisons is HeapSort.
- No sorting algorithm can have an abstract runtime better than $\Theta(N)$.

- **Question:** Is there any sorting algorithm whose abstract runtime is better than $\Theta(N\log(N))$?
- **Answer: NO** , if it works with key comparisons.
- Tool to proof this: **decision trees.**

Decision tree. Example: InsertSort

T_{IS}^{3}

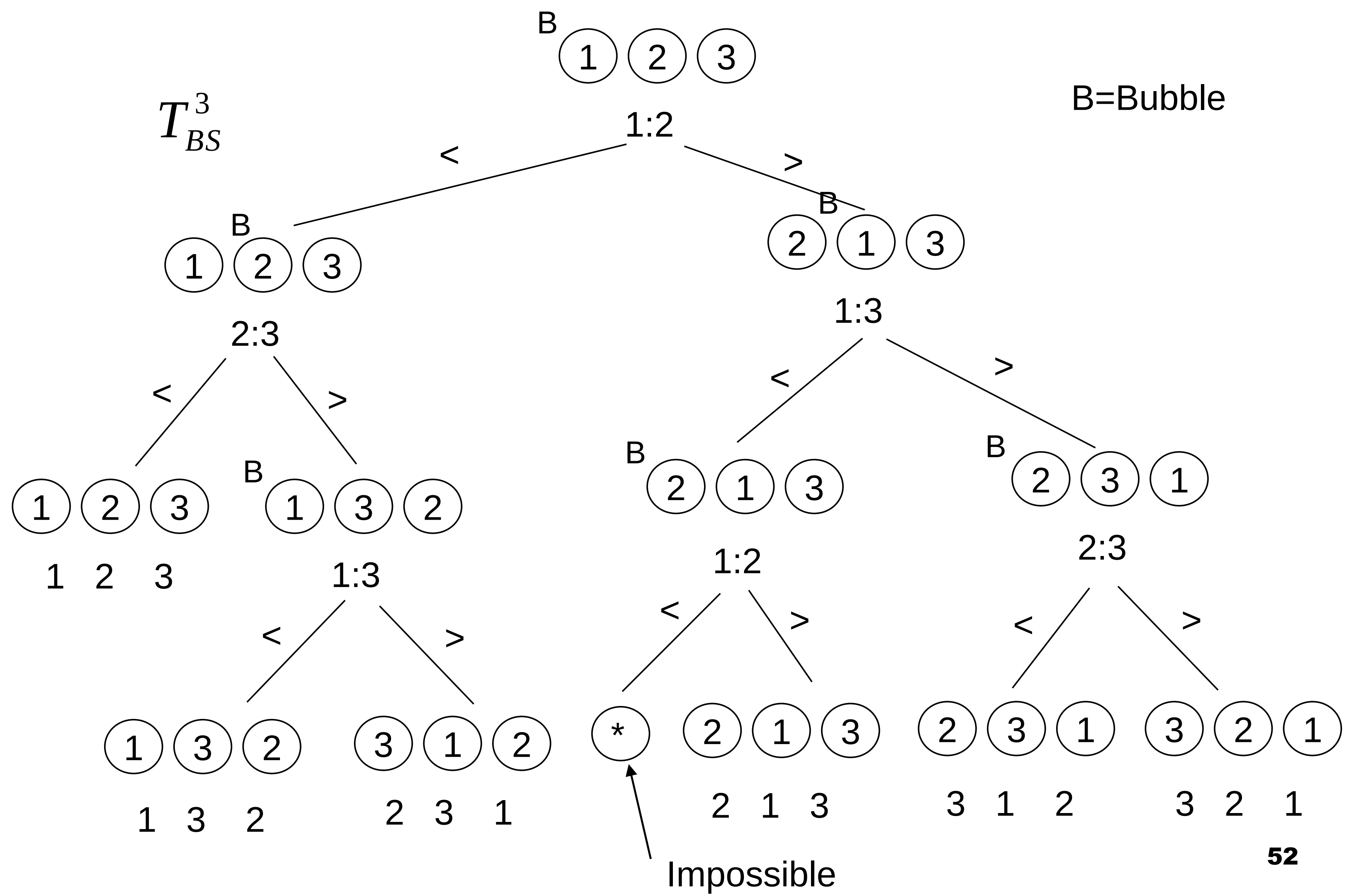


Decision tree: definition

- If **A** is a comparison based sorting algorithm and **N** is the size of the array, its **decision tree** $\mathbf{T_A^N}$ can be built over $\sum_N$ with the following 4 rules:

1. The tree contains nodes in the form $\boxed{\mathbf{i:j}}$ (i<j) which indicate the key comparison between the elements **initially** in positions **i** and **j**.
2. The left subtree of $\boxed{\mathbf{i:j}}$ in $\mathbf{T_A^N}$ contains the key comparisons that A performs if **i** < **j**.
3. The right subtree of $\boxed{\mathbf{i:j}}$ in $\mathbf{T_A^N}$ contains the key comparisons that A performs if **i** > **j**.
4. Each input $\sigma \in \sum_N$ is associate with a unique leaf L_σ in $\mathbf{T_A^N}$ and the **nodes between the root and the leaf** H_σ represent the **successive key comparisons** that algorithm **A** performs to sort permutation σ.

Decision tree. Example: BubbleSort



Recalling: tree height and depth

- **Height of a node** in a tree is the # of edges on the longest path from the node to a leaf.
- **Depth of a node** in a tree is the # of edges from the node to the root node.

- **Height of a tree** is height of root node
- **Depth of a tree is** depth of deepest node

- **Height and depth of a tree are the same**, typically the use of height is more common.

Decision tree: observations

1. The number of leaves in $\mathbf{T_A^N}$ is $N! = |\Sigma_N|$.
2. $n_A(\sigma)$= # of key comparisons= depth of the leaf L_σ in $\mathbf{T_A^N}$

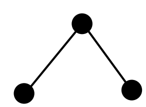
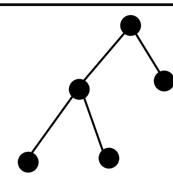
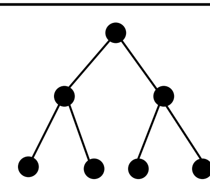
$$\boxed{n_A(\sigma) = depth_{T_A^N}(L_\sigma)}$$

3. Thus,

$$W_A(N) = \max_{\sigma \in \Sigma_N} n_A(\sigma) = \max_{\sigma \in \Sigma_N} depth_{T_A^N}(L_\sigma)$$

Lower bound in the worst case I

- What is the **minimum height** of a binary tree with L leaves?

# of leaves L	$BT^{Minimum}(L)$	Height
1		0
2		1
3		2
4		2
….	….	….

Lower bound in the worst case II

- It seems that the minimum height of a binary tree with L leaves is $\lceil \log(L) \rceil$
- For the worst case we have:

$$W_A(N) = \max_{\sigma \in \Sigma_N} depth_{T_A^N}(L_\sigma) \geq \min height \text{ with N! leaves}$$

$$= \lceil \lg(N!) \rceil$$

- Since we know that lg(N!)=Θ(Nlog(N)) = Θ(Nlg(N)), then W_A(N)=Ω(Nlg(N)).
- HeapSort and MergeSort are **optimal for the worst case**.

Lower bound for the average case I

- From the definition of average case:

$$A_A(N) = \frac{1}{N!}\sum_{\sigma\in\Sigma_N} n_A(\sigma) = \frac{1}{N!}\sum_{L\in T_A^N} depth_{T_A^N}(L)$$

- Then, $A_A(N)$≥minimum average depth(N!) (AD_{min}) where

  $AD_{min}(k)$ = *min {aver. depth(T): T BT with k leaves}*

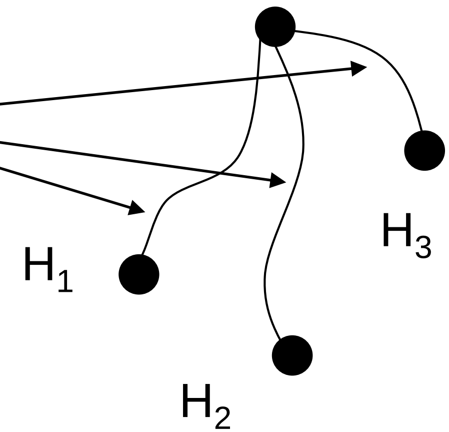
- And

$$\underset{\text{with } k \text{ leaves}}{aver.\,depth(T)} = \frac{1}{k}\sum_{L\in leaves \text{ in T}} \mathrm{depth(L)} = \frac{1}{k} EPL(T)$$

- EPL: **External path length** (leaf path length)

Lower bound in the average case II

Sum of the lengths of all paths from the root to a leaf=EPL (external path length)



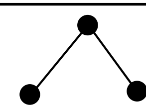
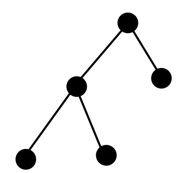
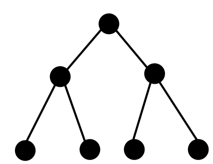
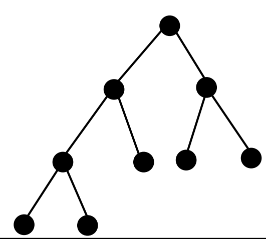
- Thus,

$$A_A(N) \geq \frac{1}{N!} EPL_{\min}(N!) \text{ with}$$

$$EPL_{\min}(k) = \min\left\{EPL(T):\ \text{T with k leaves}\right\}$$

Lower bound for the average case III

- Estimating $EPL_{min}(k)$

k	T Optimal	$EPL_{min}(k)$
1		0
2		2(1+1)
3		5(2+2+1)
4		8(2+2+2+2)
5		12(3+3+2+2+2)
….	….	….

Lower bound in the average case IV

- **It can be shown (see class notes) that**
  $EPL_{min}(k)=k\lceil \lg(k) \rceil+k-2^{\lceil \lg(k) \rceil}$
- Since

$$A_A(N) \geq \frac{1}{N!} EPL_{\min}(N!) =$$

$$\frac{1}{N!}\left(N!\lceil \lg(N!) \rceil + N! - 2^{\lceil \lg(N!) \rceil}\right) = \lceil \lg(N!) \rceil + 1 - \frac{2^{\lceil \lg(N!) \rceil}}{N!} =$$

$$\lceil \lg(N!) \rceil = \Omega(N \lg(N))$$

- Thus, $\boxed{A_A(N) = \Omega(N \lg(N))}$
- MS, QS and HS are **optimal** in the **average case.**

In this section we have learnt…

- The concept of **decision tree** for a sorting algorithm that uses key comparisons.
- To **build** a decision tree for a sorting algorithm based on key comparisons.
- The **lower bounds** for sorting algorithms based on key comparisons.
- **How these lower bounds** are estimated from the use of decision trees.

Tools and techniques to work on

- The construction of decision trees for arrays of three elements.
- The construction of partial decision trees for tables of four elements.
- Problems to solve (at least !!!): those recommended for section 10.