Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

CAIDAS WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Sorting

Prof. Dr. Goran Glavaš

6.11.2023

Content

- Sorting
- Merge Sort
- Quick Sort

Sorting problem

- How do we measure time complexity?
  - In terms of **number of elementary operations executed**
  - How does that number depend on the input? What is the size of the problem?
  - What about the operations that do not depend on the size of the input?

- Let us go back to the sorting problem...

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Why Sorting?

- Sorting is considered to be the **most fundamental problem** in the study of algorithms
  - Some applications are basically directly expressible as sorting problems
    - E.g., Banks are legally obliged to issue checks in sorted order
      Companies must issue invoices in some order
  - Many algorithms use sorting as a component, i.e., a subroutine
  - There's a **wide variety of sorting algorithms**: they use techniques and data structures used in more complex algorithms too
    - Good starting point for „algorithmic thinking"
  - We can prove a nontrivial lower-bound complexity for sorting, and also know that the best sorting algorithms reach this bound asymptotically
    - This can be used to prove lower-bound complexity for more complex problems

Keys and Records

comparison central elementary operation in all sorting algorithms

- All examples will sort **numbers**
  - How do we sort items of other data types?
  - We just need to define a comparison operator for other primitive types
    - E.g., strings can be converted into integers. **Q**: how?

  - We typically sort more complex items („**records**”), with **key** being the numeric field of the record based on which we sort
    - The rest of the record is just moved together with the key

Lower-bound complexity

- A **lower bound** for a problem is the worst-case running time of the best (most efficient) possible algorithm that solves the problem

- Lower-bound for **sorting**?

- So far, we've seen only one sorting algorithm: **Insert**(ion) **sort**
  - Insert sort has the quadratic complexity, it's running time is in $\mathbf{O(n^2)}$
  - A sorting algorithm with lower/better worst-case running time?
  - A sorting algorithm of linear complexity: in $\mathbf{O(n)}$?

Insert sort

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, ..., a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq ... \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)  # L is a list of numbers
  for i = 1 to L.length - 1 # 0-indexing, first element is at index 0, last at len-1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

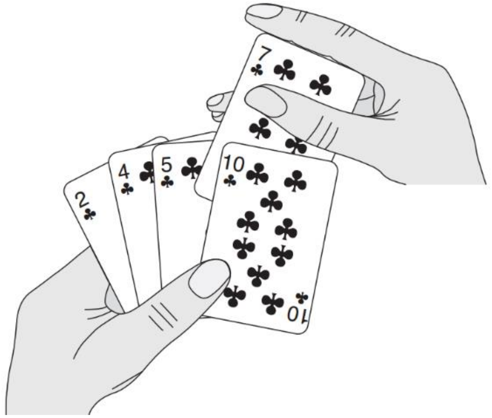


Image from *Cormen et al.*

Insert sort: **running time**

Algorithm: insert(ion) sort

```
insert_sort(L)
  for i = 1 to L.length - 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key # ∑_{i=1}^{n-1} c4 * ti
      L[j+1] = L[j] # ∑_{i=1}^{n-1} c5 * (ti-1)
      j = j - 1  # ∑_{i=1}^{n-1} c6 * (ti - 1)
    L[j+1] = key # (n-1) * c7
```

- Total running time **T(n)**

T(n) = $(n\text{-}1) * (c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * t_i + (c_5 + c_6) * (t_i - 1)$$

- What is the **worst possible** scenario (largest possible running time)?
  - If the input L is inversely sorted (from largest to smallest value)
  - $t_i = i$ for each i
  - $\sum_{i=1}^{n-1} c_4 * ti = (1 + 2 + \ldots + (n\text{-}1)) * c_4 = \frac{(n-1)*n}{2} * c_4$
  - $\sum_{i=1}^{n-1} c_5 * (t_i - 1) = (0 + 1 + \ldots + (n\text{-}2)) * c_5 = \frac{(n-2)*(n-1)}{2} * c_5$
  - $\sum_{i=1}^{n-1} c_6 * (ti - 1) = (0 + 1 + \ldots + (n\text{-}2)) * c_6 = \frac{(n-2)*(n-1)}{2} * c_6$

Insert sort: **running time**

Algorithm: insert(ion) sort

```
insert_sort(L)
  for i = 1 to L.length - 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key # Σ_{i=1}^{n-1} c4 * t_i
      L[j+1] = L[j] # Σ_{i=1}^{n-1} c5 * (t_i-1)
      j = j - 1  # Σ_{i=1}^{n-1} c6 * (t_i - 1)
    L[j+1] = key # (n-1) * c7
```

- Total running time **T(n)**

T(n) = $(n-1) * (c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * ti + (c_5 + c_6) * (t_i - 1)$$

- What is the **worst possible** scenario (largest possible running time)?
  - If the input L is inversely sorted (from largest to smallest value)
  - $t_i = i$ for each i
  - $T(n) = (n\text{-}1) * (c_1 + c_2 + c_3 + c_7) + \frac{(n-1)*n}{2} * c_4 + \frac{(n-2)*(n-1)}{2} * (c_5 + c_6)$
  - $T(n) = a*n^2 + b*n + c$
  - This is a **quadratic function of n** → **$O(n^2)$**

Rates of growth and complexity

- Growth rates for some common complexity functions
  - Θ(1) (constant)
  - Θ(log n) (logarithmic)
  - Θ(n) (linear)
  - Θ(n log n) (loglinear)
  - Θ(n^2) (quadratic complexity)
  - Θ(n^3) (cubic complexity)
    - … Θ(n^k) for k ≥ 0 (polynomial)
  - Θ(2^n) (exponential)
  - Θ(n!) (factorial)

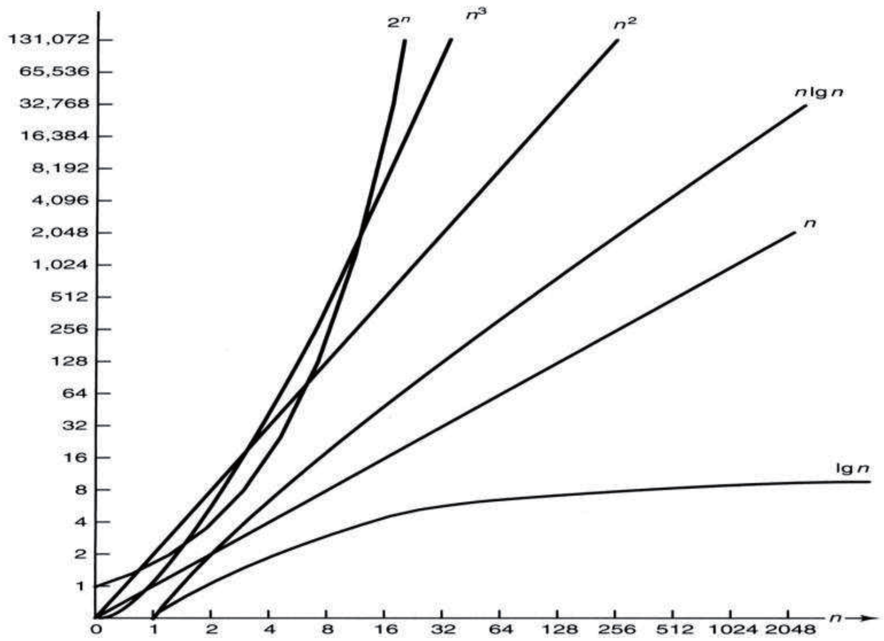


Image from https://tinyurl.com/46c3cssy

Sorting algorithms

- We will not only consider time complexity, but also **space complexity**
  - Space is normally not an issue, but to emphasize **space-time trade-off**
- **In-place** sorting
  - Algorithm that only needs to store **a constant number of elements** from the input array outside of that array
  - Is **insert**(ion) **sort** an in-place sorting algorithm?
    - How many elements are stored outside of the input array at any given time?
- When sorting **very large arrays**, „in-place” sorting becomes important

Sorting and algorithm design techniques

- When building algorithms, we often resort to some common **algorithm design techniques**

- **Insert sort**: sorting based on **incremental** approach
  - Having sorted the subarray `L[0:i-1]`
  - We proceed to insert the i-th element into the correct place
  - This yields the correct sorting for the subarray `L[0:i]`

```
insert_sort(L)
 for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

Sorting and algorithm design techniques

- When building algorithms, we often resort to some common **algorithm design techniques**

- Sorting based on **divide-and-conquer** approach (recursion!)
- **Divide-and-conquer:**
  - **DIVIDE**: divide the problem into a number of subproblems that are instances of the same problem
  - **CONQUER**: solve the subproblems
    - if the size of the subproblem is small enough, solve it the straightforward way
    - If the size of the subproblem is still large, DIVIDE it further
  - **COMBINE**: create the solution to the problem by combining the solutions to the subproblems

Content

- Sorting
- Merge Sort
- Quick Sort

Merge Sort

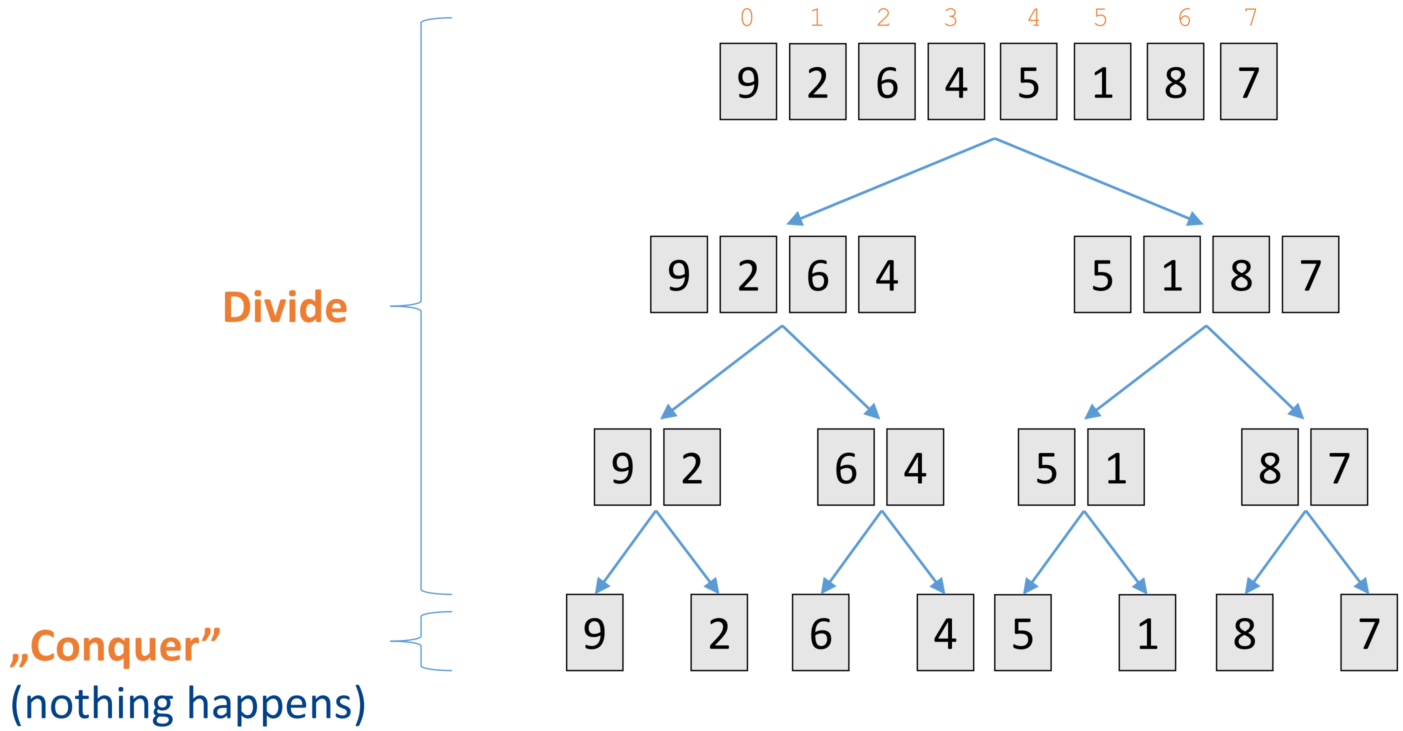
Merge Sort implements the „divide-and-conquer" algorithm design

- **DIVIDE**: divide the n-element input array to be sorted into two subarrays of length n/2 each
- **CONQUER**: sort each of the subarrays recursively (the recursion hits the „bottom" when the subarray to be sorted is of length 1)
- **COMBINE**: Merge the sorted subarrays to produce the sorted array
    - Key is the **merge function** here, otherwise merge sort is a simple recursion

Merge Sort: illustration

Divide until reaching single-element subarrays

Conquer: trivial – „sort one-element arrays" (no real sorting)

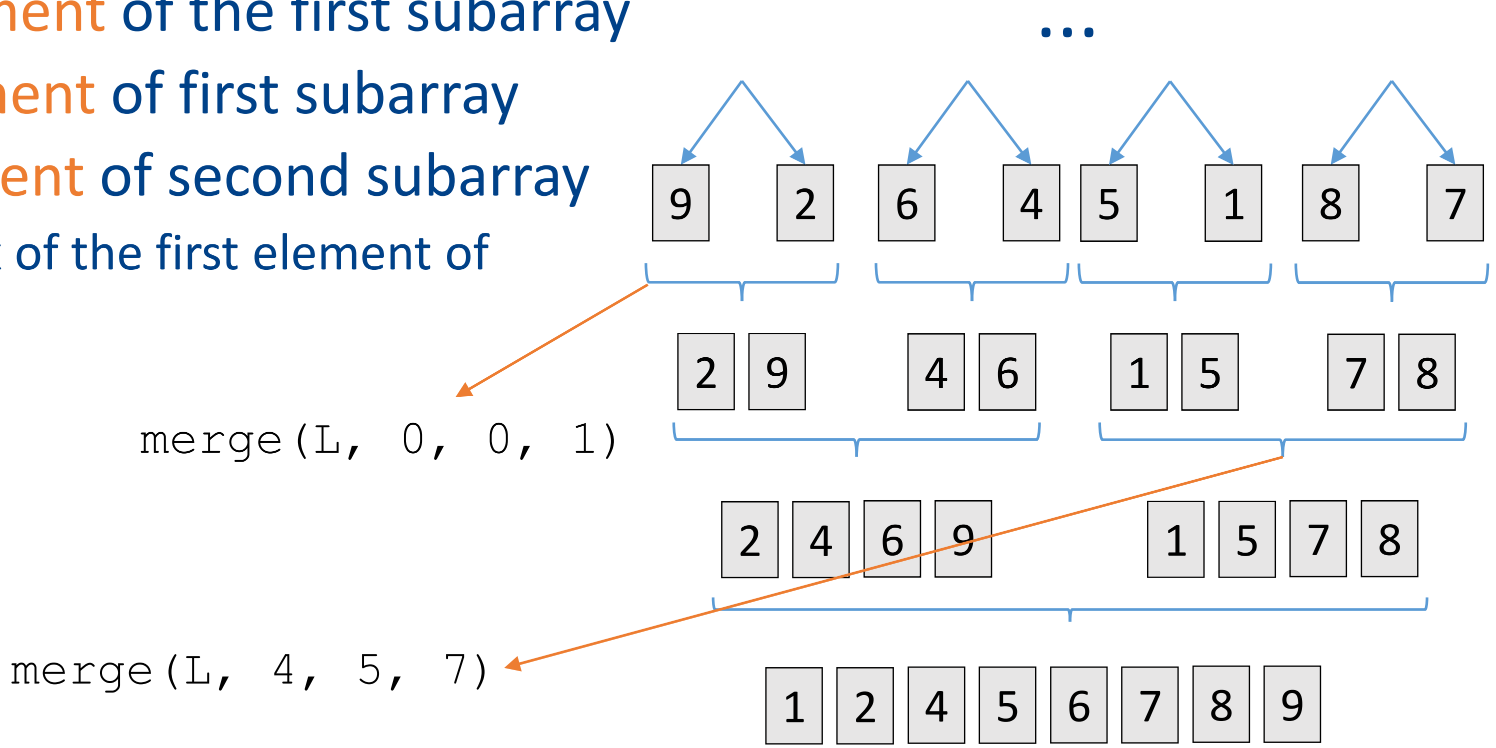


Merge Sort: illustration

Combine: merge two **sorted** subarrays into a sorted array

We need to define the critical `merge(A, p, q, r)` function

- A: the input array
- p: index of first element of the first subarray
- q: index of last element of first subarray
- r: index of last element of second subarray
  - **Q:** what's the index of the first element of second subarray?



Merge Sort: `merge` function

```
merge(A, p, q, r)
    n_left = q - p + 1 # number of elements in the left subarray
    n_right = r - q # number of of elements in the right subarray
    L = array[n_left] # create the left subarray
    R = array[n_right] # create the right subarray
    # copy the elements from the original array into subarrays
    for i = 0 to n_left - 1:
      L[i] = A[p + i]
    for j = 0 to n_right - 1:
      R[j] = A[q + 1 + j]
    # the real „merging” starts now
    ind_l = 0
    ind_r = 0
    for k = p to r
      if ind_r > n_right - 1 or L[ind_l] ≤ R[ind_r]
        A[k] = L[ind_l]
        ind_l = ind_l + 1
      else
        A[k] = R[ind_r]
        ind_r = ind_r + 1
```

Merge sort: `merge` function

- What is the running time of the **`merge`** function?
- What is the „input size" n?
  - Length of (sub)array under consideration: r – p + 1
  - Consists of two subarrays
- If we ignore the constant runtime costs, we get

n/2 + n/2 + n = 2n = **O(n)**

```
merge(A, p, q, r)
   n_left = q - p + 1
   n_right = r – q
   L = array[n_left]
   R = array[n_right]
   for i = 0 to n_left - 1: # runtime = n/2
     L[i] = A[p + i]
   for j = 0 to n_right – 1: # runtime = n/2
     R[j] = A[q + 1 + j]
   ind_l = 0
   ind_r = 0
   for k = p to r # runtime = n
     if ind_r > n_right – 1 or L[ind_l] ≤ R[ind_r]
       A[k] = L[ind_l]
       ind_l = ind_l + 1
     else
       A[k] = R[ind_r]
       ind_r = ind_r + 1
```

Merge sort

- Now that we have defined the merge function, let's see the whole **recursive merge sort** algorithm

```
merge_sort(A, p, r)
  n = r - p + 1
  if n % 2 == 1 # odd number of elements
    q = p + n//2 # a//b is integer division, 7//2 = 3
  else # even number of elements
    q = p + n/2 - 1

  merge_sort(A, p, q)
  merge_sort(A, q + 1, r)
  merge(A, p, q, r)
```

Merge sort: **runtime**

- Runtime of the `merge` function is $2n = O(n)$
- Merge-sort on 1-element array
  - Constant time (nothing actually), **O(1)**
- When $n > 1$
  - **DIVIDE**: just computes the middle of the subarray, constant time →
    - D(n) = **O(1)**
  - **CONQUER**: recursively sort two subproblems of size n/2
    - C(n) = **2 * T(n/2)**
  - **COMBINE** (merge): runtime of the `merge` function
    - M(n) = **O(n)**

```
merge_sort(A, p, r)
   n = r - p + 1
   if n % 2 == 1
     q = p + n//2
   else
     q = p + n/2 - 1

   merge_sort(A, p, q)
   merge_sort(A, q + 1, r)
   merge(A, p, q, r)
```

Merge sort: **runtime**

DIVIDE: D(n) = **O(1)**

CONQUER: C(n) = **2 * T(n/2)**

COMBINE (merge): M(n) = **O(n)**

- Summing D(n) + M(n) gives O(n) + O(1) = **O(n)**

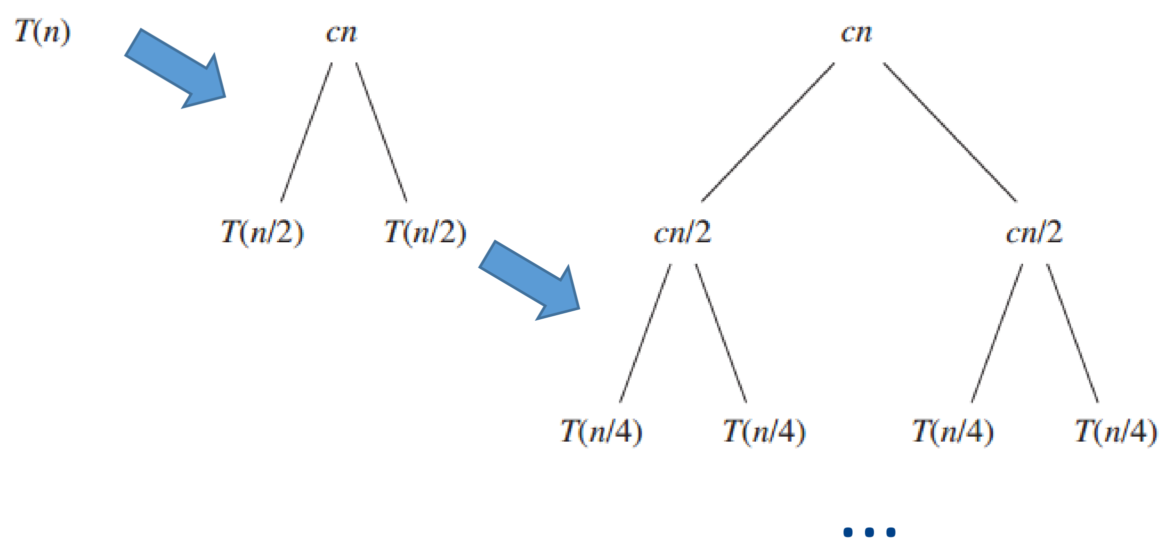
- So, T(n) for **merge sort** is
  → **O(1)**, **if** n = 1
  → **2*T(n/2) + O(n)**, **if** n > 1   (recursively defined runtime)

- Or, removing the O notation, introducing the constants, T(n) =
  → c, if n = 1
  → 2*T(n/2) + c*n, **if** n > 1

Merge sort: **runtime**

- So, T(n) is
  - → c, if n = 1
  - → 2*T(n/2) + c*n, **if** n > 1

- Recursive runtime computation
  - T(n/2) = 2*T(n/4) + c*n/2
  - T(n/4) = 2* T(n/8) + c*n/4
  - …
  - T(n = 1) = c



(Adapted) Image from Cormen et al.

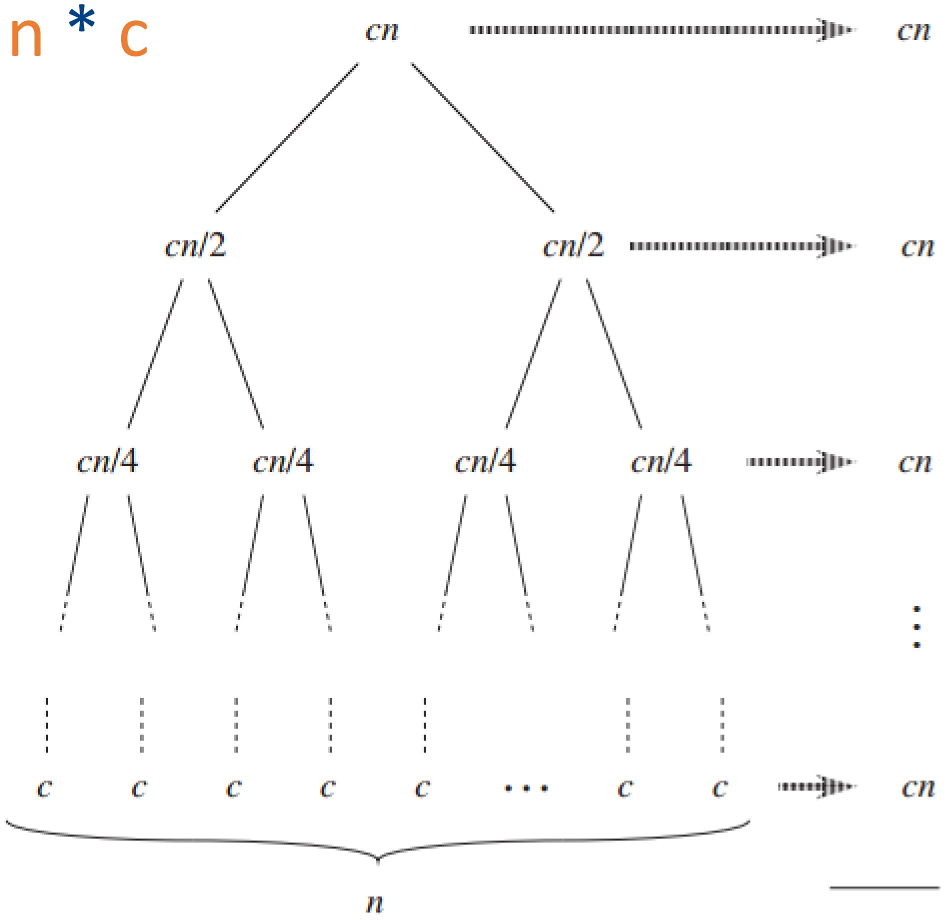
Merge sort: **runtime**

- $T(n) = c*n + 2*c*n/2 + 4*c*n/4 + ... + n * c$

  $= c*n + c*n + c*n + ... + c*n$

  How many times do we have $c*n$?

- Depth of the tree = $\log_2 n$
- $T(n) = c*n*\log_2 n =$ **$O(n \log n)$**



Merge sort: space complexity

- **Q**: Is merge sort an „in place” sorting algorithm?
- How much **additional** memory does it need besides A?
  - Is that additional memory of constant size or depends on n?
- In `merge` function, we copy all elements into subarrays `L` and `R`
  - `L+R` have n elements
  - So total memory occupation is 2n
- Not in place sorting
  - A problem only in case of extremely large arrays

```
merge(A, p, q, r)
  n_left = q - p + 1
  n_right = r - q
  L = array[n_left]
  R = array[n_right]
  for i = 0 to n_left - 1: # runtime - n/2
    L[i] = A[p + i]
  for j = 0 to n_right - 1: # runtime - n/2
    R[j] = A[q + 1 + j]
  ind_l = 0
  ind_r = 0
  for k = p to r # runtime - n
    if ind_r > n_right - 1 or L[ind_l] ≤ R[ind_r]
      A[k] = L[ind_l]
      ind_l = ind_l + 1
    else
      A[k] = R[ind_r]
      ind_r = ind_r + 1
```

Content

- Sorting
- Merge Sort
- Quick Sort

Quick Sort

Quick sort is another „divide-and-conquer” sorting algorithm

- Unlike merge sort, sorts the array **in place**

- **DIVIDE: central part of the algorithm**
  - Partition the array **A[p, r]** into two subarrays **A[p, q-1]** and **A[q+1, r]**, such that all elements of **A[p, q-1]** are smaller than **A[q]** and all elements of **A[q+1, r]** are larger than **A[q]**
  - After sorting **A[p, q-1]** and **A[q+1, r]** (recursively) the whole array is **sorted**

```
quick_sort(A, p, r)
    q = partition(A, p, r)
    quick_sort(A, p, q - 1)
    quick_sort(A, q + 1, r)
```

Quick sort: **partition**

```
partition(A, p, r)
    pivot = A[r]
    s = p - 1 # index of the last element smaller (or same) than pivot
    for i = p to r - 1:
      if A[i] ≤ pivot
        s = s + 1
        exchange(A[i], A[s])
    exchange(A[s+1], A[r])
    return s + 1
```

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

```
pivot = A[7] = 4
s = 0 - 1 = -1

#for loop, 1. iteration
A[0] = 9 ≤ pivot = 4 → False

#for loop, 2. iteration
A[1] = 2 ≤ pivot = 4 → True
s = s + 1 = 0
exchange A[1], A[0] (2 and 9)
```

2	9	6	7	5	1	8	4

Quick sort: **partition**

```
partition(A, p, r)
    pivot = A[r]
    s = p - 1 # index of the last element smaller (or same) than pivot
    for i = p to r - 1:
        if A[i] ≤ pivot
            s = s + 1
            exchange(A[i], A[s])
    exchange(A[s+1], A[r])
    return s + 1
```

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

```
#for loop, 3. iteration
A[2] = 6 ≤ pivot = 4 → False

#for loop, 4. iteration
A[3] = 7 ≤ pivot = 4 → False

#for loop, 5. iteration
A[4] = 5 ≤ pivot = 4 → False

...
```

Quick sort: **partition**

```
partition(A, p, r)
    pivot = A[r]
    s = p - 1 # index of the last element smaller (or same) than pivot
    for i = p to r - 1:
      if A[i] ≤ pivot
        s = s + 1
        exchange(A[i], A[s])
    exchange(A[s+1], A[r])
    return s + 1
```

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

```
#for loop, 6. iteration
A[5] = 1 ≤ pivot = 4 → True

s = s + 1 = 1
exchange A[1], A[5] (1 and 9)
```

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

Quick sort: partition

```
partition(A, p, r)
   pivot = A[r]
   s = p - 1 # index of the last element smaller (or same) than pivot
   for i = p to r - 1:
     if A[i] ≤ pivot
       s = s + 1
       exchange(A[i], A[s])
   exchange(A[s+1], A[r])
   return s + 1
```

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

```
#for loop, 7. iteration
A[6] = 8 ≤ pivot = 4 → False
# for loop over, s = 1

exchange A[7](pivot), A[s+1 = 2]
(6 and 4)
```

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

```
return 2 (s+1)
quick_sort([2,1])
quick_sort([7, 5, 9, 8, 6])
```

Quick sort: **running time**

- The running time of the quick sort depends on whether the partitioning is (mostly) **balanced** or **unbalanced**

- If the partitioning is **balanced**, quick sort will have the running time **of a merge sort** (but with in place sorting!)
  - On average, partitioning will be balanced! **Q**: why?
  - So average runtime of quick sort is **O(n*log n)**!
  - Not just that, the constants in running time are lower for quick sort

- **Worst case scenario**
  - Running time of quick sort will be $O(n^2)$.
  - **Q**: why?

Quick sort: worst **running time**

- If `A[i]` ≤ `pivot` is never fulfilled
- So the partitions will be **[]** and **A[1…r]**

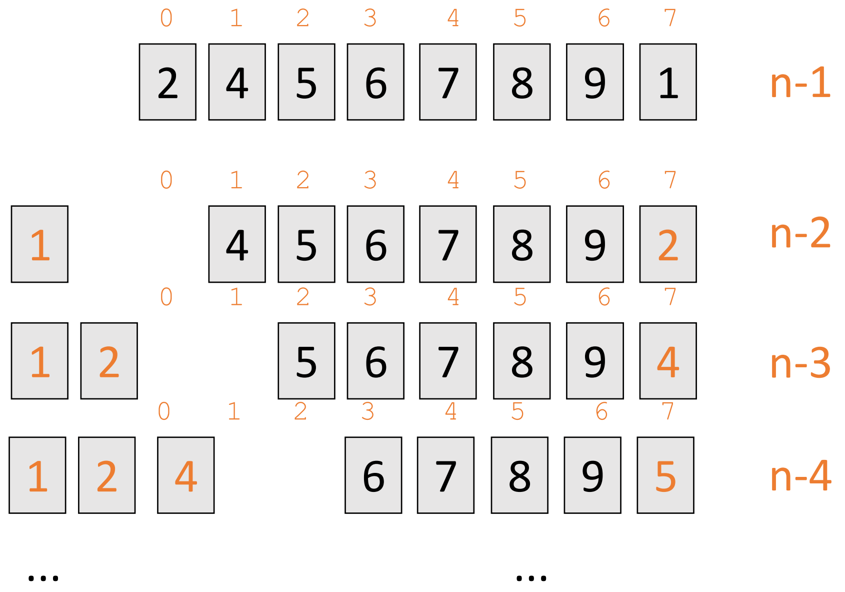
- **Q**: Can you think of a worst case example for quick sort?

- T(n) = (n-1) + (n-2) + … + 2 + 1

  = (n - 1) * n / 2

  = **O(n²)**

```
partition(A, p, r)
    pivot = A[r]
    s = p - 1
    for i = p to r - 1:
      if A[i] ≤ pivot
        s = s + 1
        exchange(A[i], A[s])
    exchange(A[s+1], A[r])
    return s + 1
```



Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟