Algorithms

Divide et impera

Emanuele Rodolà
rodola@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

2nd semester a.y. 2019/2020 · March 12, 2020

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems
   - Each subproblem is similar to the original problem

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size
2. Solve each subproblem recursively

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size
2. Solve each subproblem recursively
   - Each subproblem will be split into sub-subproblems, and so on...

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size
2. Solve each subproblem recursively
   - Each subproblem will be split into sub-subproblems, and so on...
3. Combine the solutions

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems **(DIVIDE)**
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size
2. Solve each subproblem recursively **(CONQUER)**
   - Each subproblem will be split into sub-subproblems, and so on...
3. Combine the solutions **(COMBINE)**

Breaking the problem

Overall idea:

1. Break the main problem into many subproblems **(DIVIDE)**
   - Each subproblem is similar to the original problem
   - Each subproblem is smaller in size
2. Solve each subproblem recursively **(CONQUER)**
   - Each subproblem will be split into sub-subproblems, and so on...
3. Combine the solutions **(COMBINE)**

This general approach is called divide et impera or also divide and conquer.

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \ldots, a'_n)$ such that:

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

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \dots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \dots, a'_n)$ such that:

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

Divide-et-impera recipe:

1. Divide the n-element sequence into two $\frac{n}{2}$-element sequences

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \ldots, a'_n)$ such that:

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

Divide-et-impera recipe:

1. Divide the n-element sequence into two $\frac{n}{2}$-element sequences
2. Sort each subsequence using mergesort

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \ldots, a'_n)$ such that:

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

Divide-et-impera recipe:

1. Divide the n-element sequence into two $\frac{n}{2}$-element sequences
2. Sort each subsequence using mergesort
3. Merge the two sorted subsequences

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \ldots, a'_n)$ such that:

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

Divide-et-impera recipe:

1. Divide the n-element sequence into two $\frac{n}{2}$-element sequences
2. Sort each subsequence using mergesort
3. Merge the two sorted subsequences

At some point, the generated subsequences will have length **1**
$\Rightarrow$ no more splitting needed!

We call this the base case.

Merge sort

Again, we consider the sorting problem as a toy example.

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A reordered sequence $(a'_1, a'_2, \ldots, a'_n)$ such that:

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

Divide-et-impera recipe:

1. Divide the n-element sequence into two $\frac{n}{2}$-element sequences
2. Sort each subsequence using mergesort
3. Merge the two sorted subsequences $\leftarrow$**key step**

At some point, the generated subsequences will have length **1**
$\Rightarrow$ no more splitting needed!

We call this the base case.

We translate the original problem into a different and easier subproblem of smaller size.

Divide-et-impera

We translate the original problem into a different and easier subproblem of smaller size.

Sorting problem $\Rightarrow$ Merging problem

Complete algorithm

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Complete algorithm

$\text{Merge-Sort}(A, p, r)$

if $p < r$
  then $q \leftarrow \lfloor (p + r)/2 \rfloor$
    $\text{Merge-Sort}(A, p, q)$
    $\text{Merge-Sort}(A, q + 1, r)$
    $\text{Merge}(A, p, q, r)$

- The procedure calls itself recursively

Complete algorithm

MERGE-SORT(A, p, r)

if $p < r$
  then $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

- The procedure calls itself recursively
- Start: call merge sort on $(A, 1, length[A])$

Complete algorithm

```
MERGE-SORT(A, p, r)
if p < r
    then q ← ⌊(p + r)/2⌋
         MERGE-SORT(A, p, q)
         MERGE-SORT(A, q + 1, r)
         MERGE(A, p, q, r)
```

- The procedure calls itself recursively
- Start: call merge sort on $(A, 1, length[A])$
- Keep splitting until $p < r$ is false...

Complete algorithm

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

- The procedure calls itself recursively
- Start: call merge sort on $(A, 1, length[A])$
- Keep splitting until $p < r$ is false...
- ...which is the base case: the two halves have length 1

Complete algorithm

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

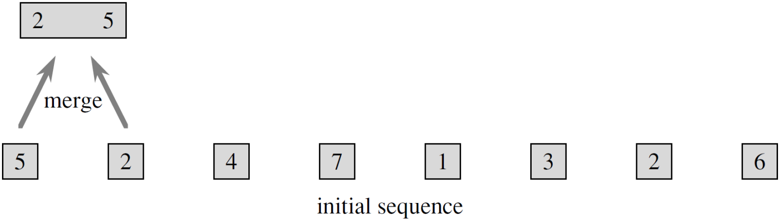
- The procedure calls itself recursively
- Start: call merge sort on $(A, 1, length[A])$
- Keep splitting until $p < r$ is false...
- ...which is the base case: the two halves have length 1
- If $length[A]$ is a power of 2, we always split in equal halves

Example

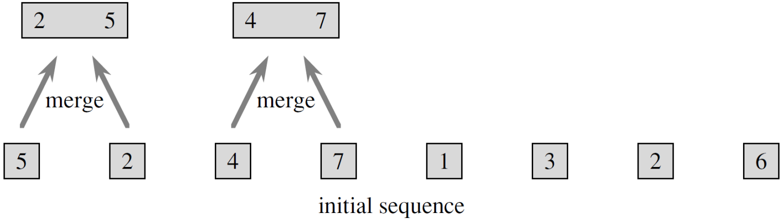
$\boxed{5}\quad\boxed{2}\quad\boxed{4}\quad\boxed{7}\quad\boxed{1}\quad\boxed{3}\quad\boxed{2}\quad\boxed{6}$

initial sequence

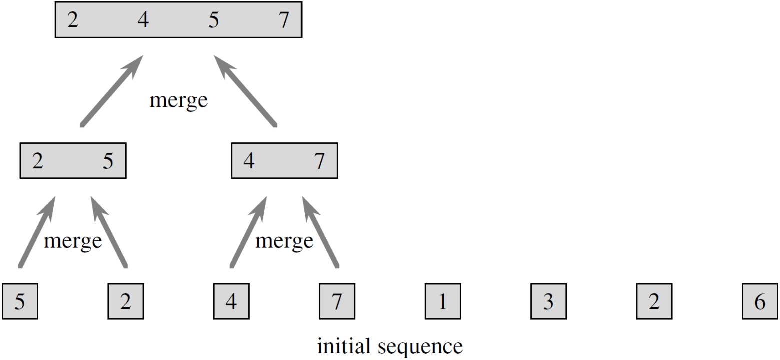
Example



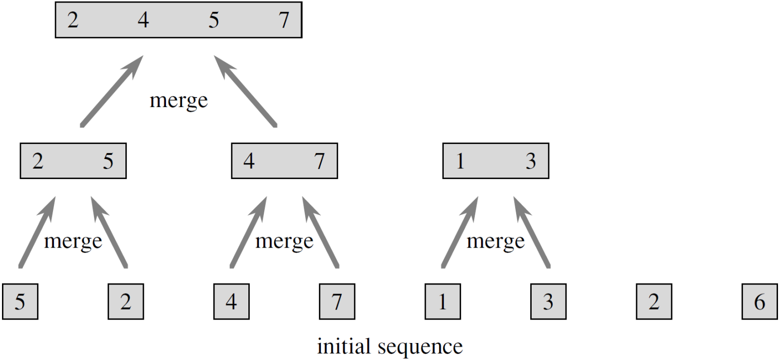
Example



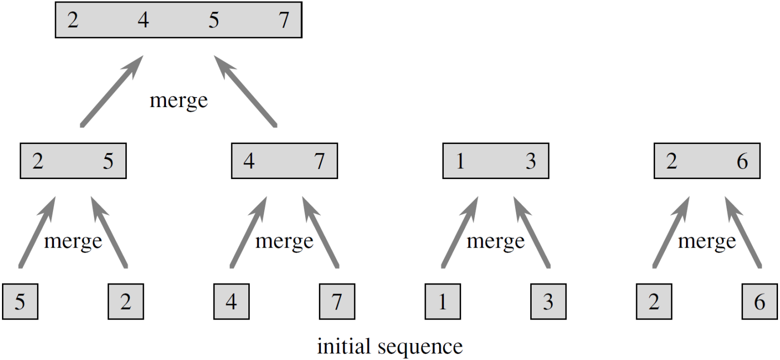
Example



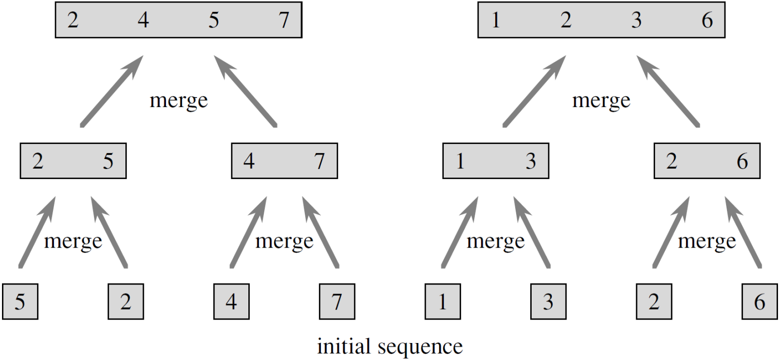
Example


2 4 5 7
merge
2 5
4 7
1 3
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Example

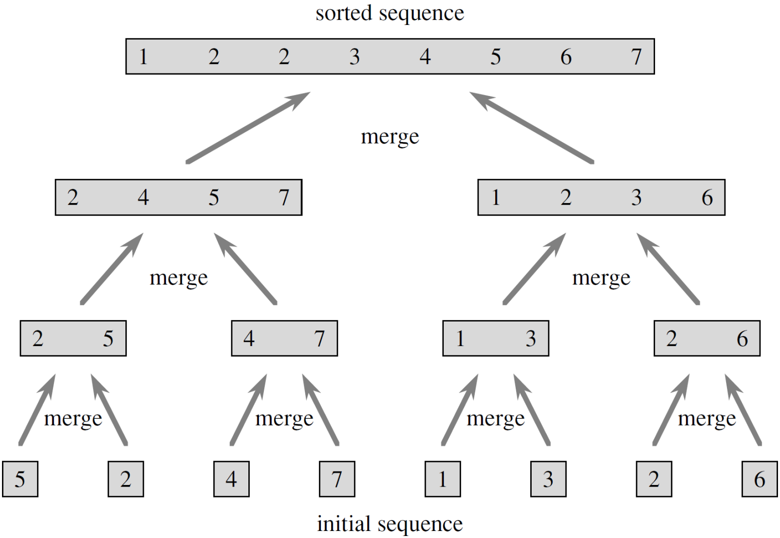

2 4 5 7
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Example



2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Example


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Merge operation

Splitting is easy: just separate sequence A into two subsequences:

$$A[p \dots q] \qquad A[q+1 \dots r]$$

Merge operation

Splitting is easy: just separate sequence A into two subsequences:

$$A[p \ldots q] \qquad A[q+1 \ldots r]$$

Assume that the two subsequences are sorted.

We want to merge them and form a sorted subsequence

$$A[p \ldots r]$$

Merge operation

Splitting is easy: just separate sequence A into two subsequences:

$$A[p \dots q] \qquad A[q+1 \dots r]$$

Assume that the two subsequences are sorted.

We want to merge them and form a sorted subsequence

$$A[p \dots r]$$

Main idea:

1. Compare only the first element of the two sequences

Merge operation

Splitting is easy: just separate sequence A into two subsequences:

$$A[p \dots q] \qquad A[q+1 \dots r]$$

Assume that the two subsequences are sorted.

We want to merge them and form a sorted subsequence

$$A[p \dots r]$$

Main idea:

1. Compare only the first element of the two sequences
2. Take out the smaller one, and put it in the output sequence

Merge operation

Splitting is easy: just separate sequence A into two subsequences:

$$A[p \dots q] \qquad A[q+1 \dots r]$$

Assume that the two subsequences are sorted.

We want to merge them and form a sorted subsequence

$$A[p \dots r]$$

Main idea:

1. Compare only the first element of the two sequences
2. Take out the smaller one, and put it in the output sequence
3. Go back to step (1) until one of the two sequences is empty

MERGE(A, p, q, r)

```
n_1 ← q − p + 1
n_2 ← r − q
```

MERGE(A, p, q, r)

```
n_1 ← q − p + 1
n_2 ← r − q
create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]
```

MERGE(A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
for $i \leftarrow 1$ **to** n_1
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
    do $R[j] \leftarrow A[q + j]$

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
```

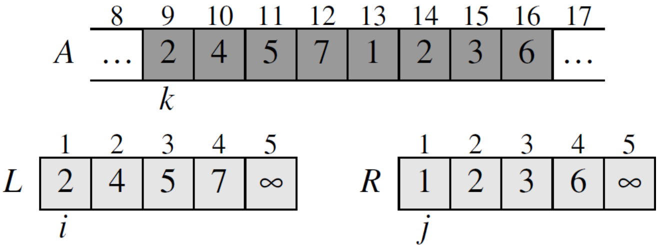
MERGE(A, p, q, r)

```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
      do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

Merge example

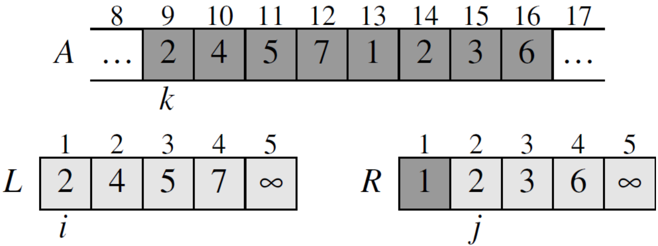
	8	9	10	11	12	13	14	15	16	17
A	…	2	4	5	7	1	2	3	6	…
		k								

Merge example



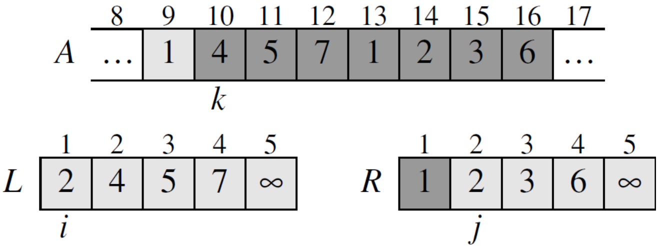
The light gray numbers are always those of the original sequence

Merge example



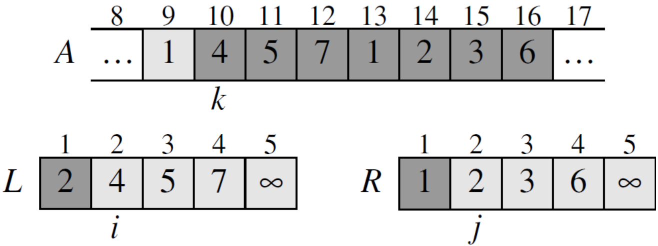
The light gray numbers are always those of the original sequence

Merge example



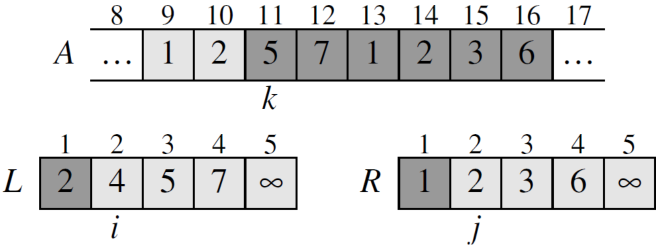
The light gray numbers are always those of the original sequence

Merge example



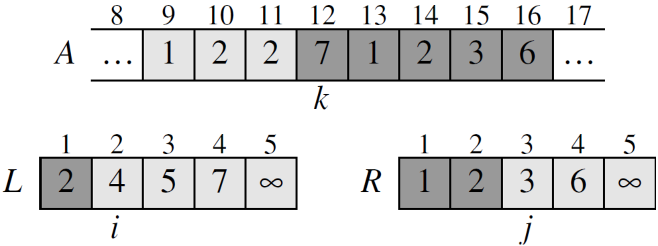
The light gray numbers are always those of the original sequence

Merge example



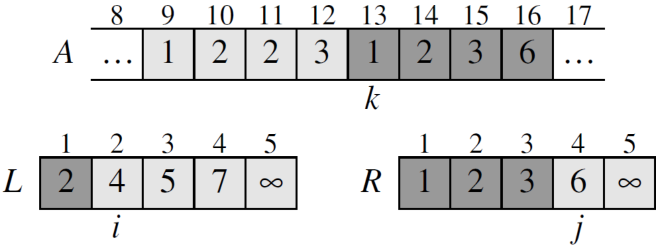
The light gray numbers are always those of the original sequence

Merge example



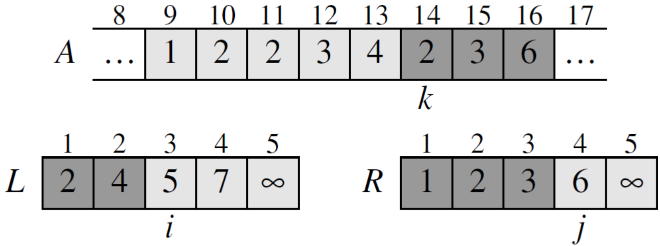
The light gray numbers are always those of the original sequence

Merge example



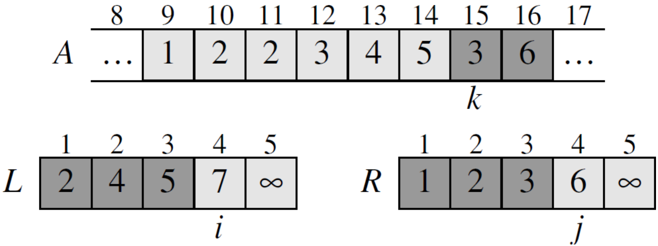
The light gray numbers are always those of the original sequence

Merge example



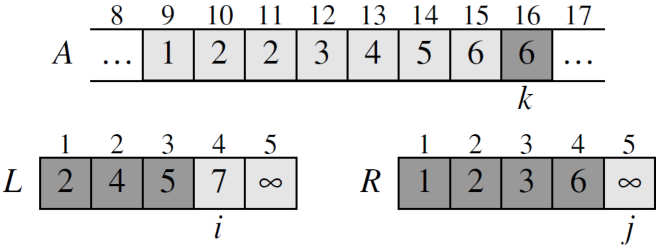
The light gray numbers are always those of the original sequence

Merge example



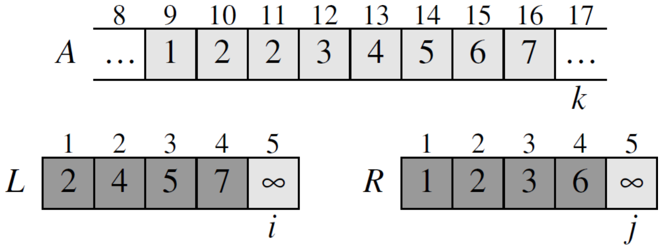
The light gray numbers are always those of the original sequence

Merge example



The light gray numbers are always those of the original sequence

Merge example



The light gray numbers are always those of the original sequence

Order of growth (merge operation only)

```
MERGE(A, p, q, r)
            n1 ← q − p + 1
constant    n2 ← r − q
            create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
            for i ← 1 to n1
                 do L[i] ← A[p + i − 1]
            for j ← 1 to n2
                 do R[j] ← A[q + j]
            L[n1 + 1] ← ∞
constant    R[n2 + 1] ← ∞
            i ← 1
            j ← 1
            for k ← p to r
                 do if L[i] ≤ R[j]
                       then A[k] ← L[i]
                            i ← i + 1
                       else A[k] ← R[j]
                            j ← j + 1
```

Order of growth (merge operation only)

```
               MERGE(A, p, q, r)
                   n1 ← q − p + 1
constant           n2 ← r − q
                   create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
linear in n1       for i ← 1 to n1
                        do L[i] ← A[p + i − 1]
linear in n2       for j ← 1 to n2
                        do R[j] ← A[q + j]
                   L[n1 + 1] ← ∞
constant           R[n2 + 1] ← ∞
                   i ← 1
                   j ← 1
                   for k ← p to r
                        do if L[i] ≤ R[j]
                              then A[k] ← L[i]
                                   i ← i + 1
                              else A[k] ← R[j]
                                   j ← j + 1
```

Order of growth (merge operation only)

MERGE(A, p, q, r)

constant

```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
```

linear in n

```
for i ← 1 to n1
        do L[i] ← A[p + i − 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
```

constant

```
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
```

```
for k ← p to r
        do if L[i] ≤ R[j]
              then A[k] ← L[i]
                   i ← i + 1
              else A[k] ← R[j]
                   j ← j + 1
```

Order of growth (merge operation only)

```
MERGE(A, p, q, r)

constant
    n_1 ← q − p + 1
    n_2 ← r − q
    create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]

linear in n
    for i ← 1 to n_1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n_2
        do R[j] ← A[q + j]

constant
    L[n_1 + 1] ← ∞
    R[n_2 + 1] ← ∞
    i ← 1
    j ← 1

linear in n
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

Order of growth (merge operation only)

MERGE(A, p, q, r)

$\Theta(1)$

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$

$\Theta(n)$

for $i \leftarrow 1$ **to** n_1
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
    do $R[j] \leftarrow A[q + j]$

$\Theta(1)$

$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$

$\Theta(n)$

for $k \leftarrow p$ **to** r
    do if $L[i] \le R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$

Order of growth (merge operation only)

$\Theta(n)$

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
      do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

Order of growth (complete merge sort algorithm)

$$T(n) =$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ & \\ & \end{cases}$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ \underbrace{D(n)}_{\text{split}} + \qquad\qquad\qquad + \underbrace{C(n)}_{\text{merge}} & n > c \end{cases}$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ \underbrace{D(n)}_{\text{split}} + \underbrace{2T(\frac{n}{2})}_{\substack{\text{cost on the}\\ \text{two halves}}} + \underbrace{C(n)}_{\text{merge}} & n > c \end{cases}$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Conquer: $2T(\frac{n}{2})$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Conquer: $2T(\frac{n}{2})$

Combine: $\Theta(n)$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Conquer: $2T(\frac{n}{2})$

Combine: $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Conquer: $2T(\frac{n}{2})$

Combine: $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

We are not done yet!

Order of growth (complete merge sort algorithm)

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ \underbrace{D(n)}_{\textbf{divide}} + \underbrace{2T(\frac{n}{2})}_{\textbf{conquer}} + \underbrace{C(n)}_{\textbf{combine}} & n > c \end{cases}$$

Divide: $\Theta(1)$

Conquer: $2T(\frac{n}{2})$

Combine: $\Theta(n)$

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\frac{n}{2}) + cn & n > 1 \end{cases}$$

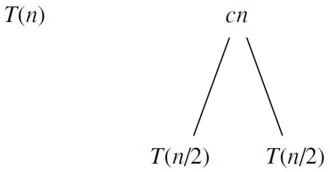
We are not done yet!

Recursion tree

$T(n)$

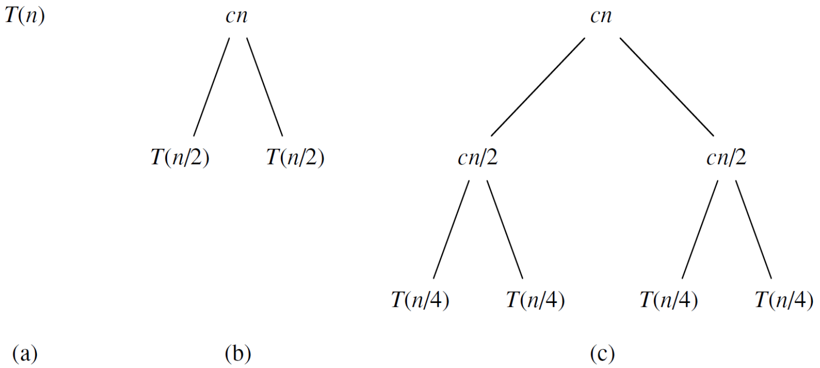
(a)

Recursion tree

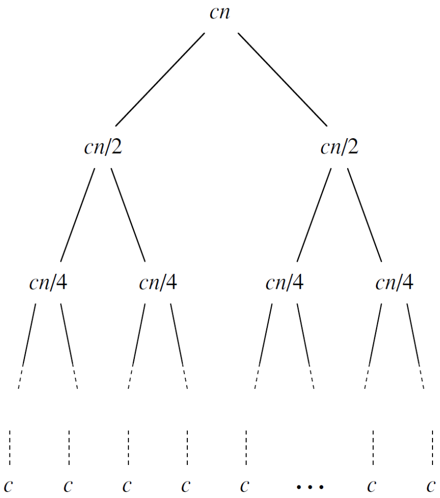


(a) (b)

Recursion tree

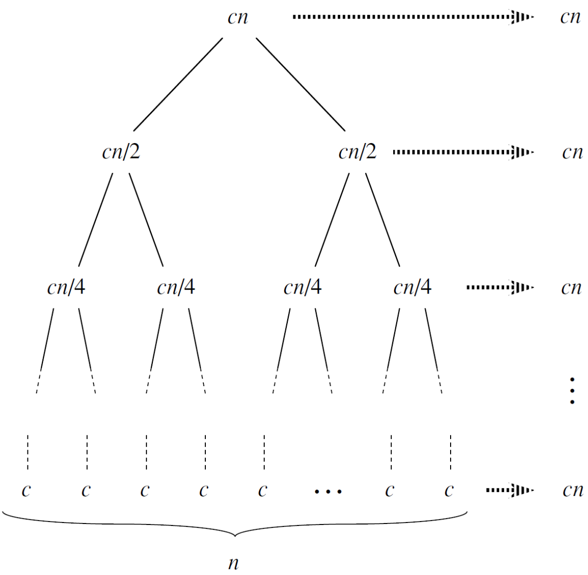


Recursion tree



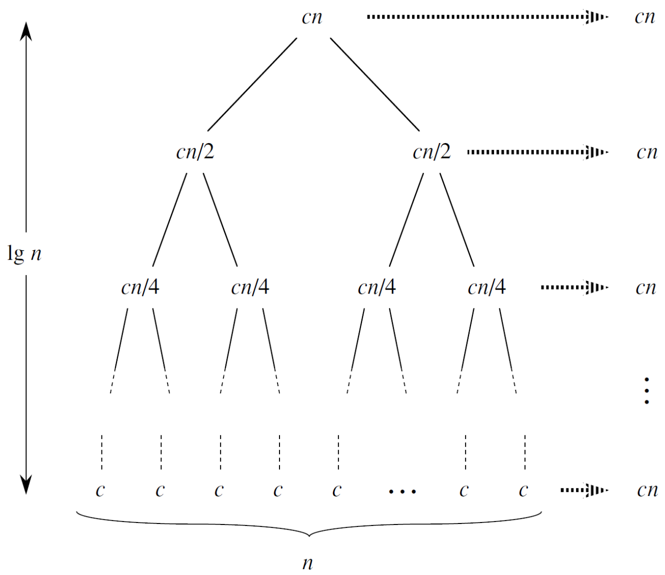
(d)

Recursion tree



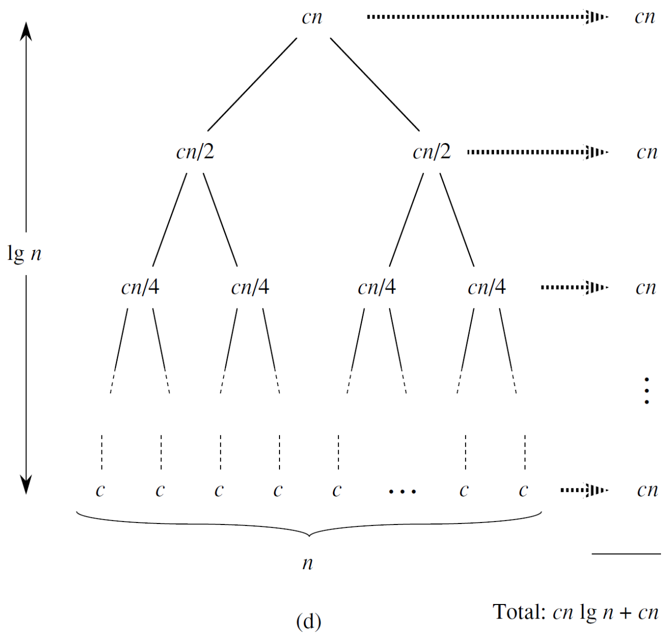
(d)

Recursion tree



(d)

Recursion tree



Order of growth (complete merge sort algorithm)

$$T(n) = cn \lg n + cn$$

Order of growth (complete merge sort algorithm)

$$T(n) = \underbrace{cn \lg n}_{\text{dominant}} + cn$$

Order of growth (complete merge sort algorithm)

$$T(n) = \underbrace{cn \lg n}_{\text{dominant}} + cn$$

Therefore, merge sort grows like:

$$\Theta(n \lg n)$$

Order of growth (complete merge sort algorithm)

$$T(n) = \underbrace{cn \lg n}_{\text{dominant}} + cn$$

Therefore, merge sort grows like:

$$\Theta(n \lg n)$$

We will see that this is true also if n is not a power of 2.

For now, we observe that dividing seems to be beneficial!

Exercises

Write pseudocode for binary search. The task is the same as the one for linear search:

Given a sequence of numbers $A = (a_1, \ldots, a_n)$ and a number v, find an index i such that $v = A[i]$. Return a special number if v can not be found in the sequence.

However, the algorithm is different:

1. Assume that the sequence is sorted
2. Check at the midpoint, and eliminate half of the sequence
3. Keep halving, either iteratively or recursively
4. Check that the worst-case running time is $\Theta(\lg n)$. Is this more or less efficient than linear search?

Suggested reading

Chapters 2.3.1 and 2.3.2 of:

"Introduction to Algorithms – 2nd Ed.", Cormen et al.