DIT181: Data Structures and Algorithms

Lecture 4: Divide and conquer, Quicksort

Michał Pałka, Gül Calikli
Email: michal.palka@chalmers.se, calikli@chalmers.se

This lecture

- Divide-and-conquer
- Big-O for recurrences
- Quicksort

Divide-and-conquer

Multiplication

$$
\begin{array}{r}
325 \\
\times\ 712 \\
\hline
650 \\
325 \\
+2275 \\
\hline
23140
\end{array}
$$

Works in any base

Multiplication: divide and conquer

- We want to compute the multiplication XY of two numbers represented in base B
- We split the digits of X and Y into

$$X = X_1 B^m + X_0$$
$$\mathrm{Y} = Y_1 B^m + Y_0$$

- We compute

$$XY = X_1 Y_1 B^{2m} + (X_1 Y_0 + X_0 Y_1) B^m + X_0 Y_0$$

- This requires performing multiplications for smaller numbers, which are performed in the same way
- When we reach single digits, we multiply them normally
- This the same pattern of multiplications as in the naïve method

Multiplication: faster

- We want to compute the multiplication XY of two numbers represented in base B
- We split the digits of X and Y into

X_0

Y_0

One multiplication

Two multiplications

- We compute

$$Z_0 = X_0 Y_0$$
$$Z_2 = X_1 Y_1$$
$$Z_1 = (X_1 + X_0)(Y_1 + Y_0) - Z_2 - Z_0 = X_1 Y_0 + X_0 Y_1$$
$$XY = Z_2 B^{2m} + Z_1 B^m + Z_0$$

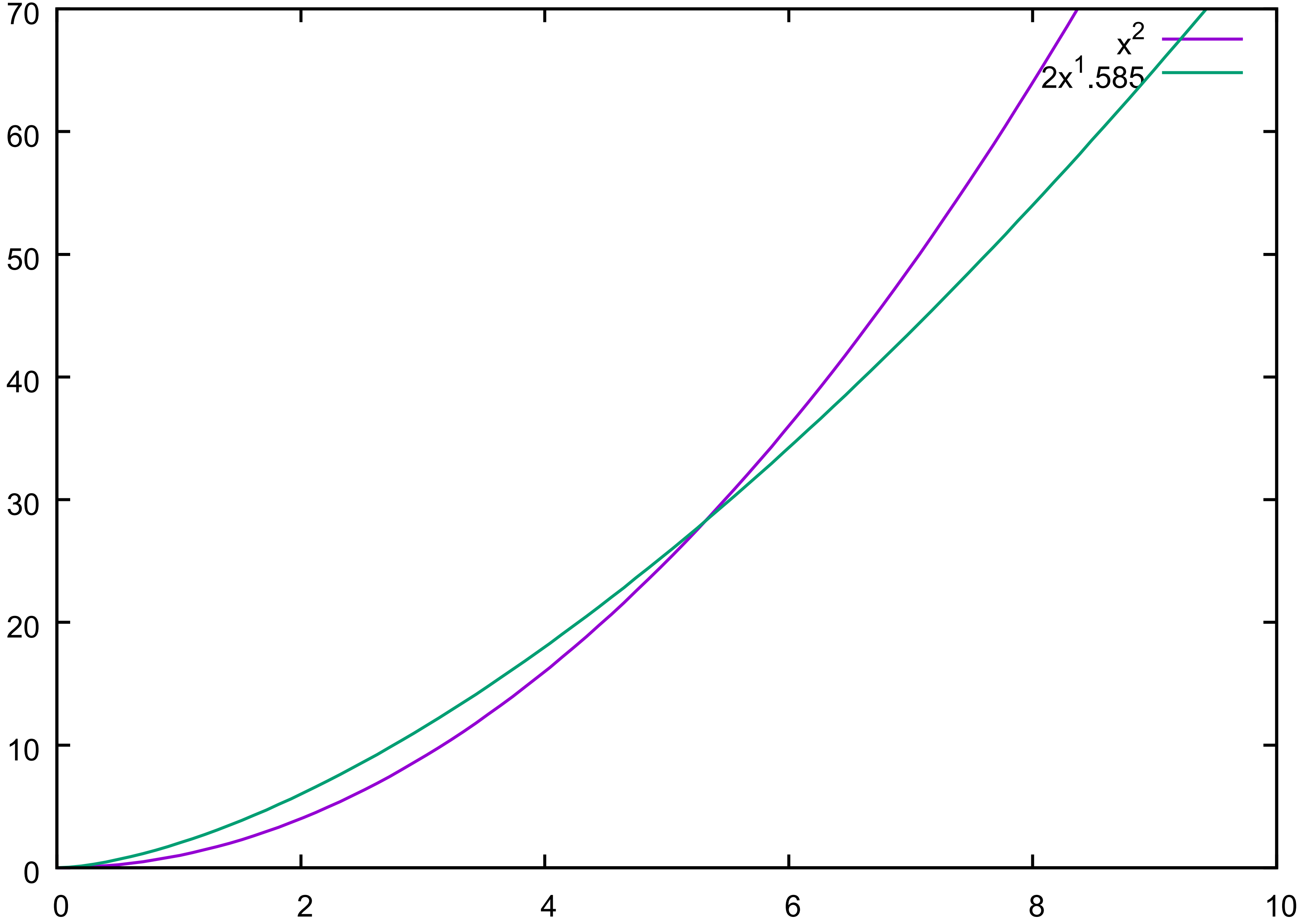
- As a result, we perform 3 recursive multiplications instead of 4 (Karatsuba algorithm)

Multiplication: complexity

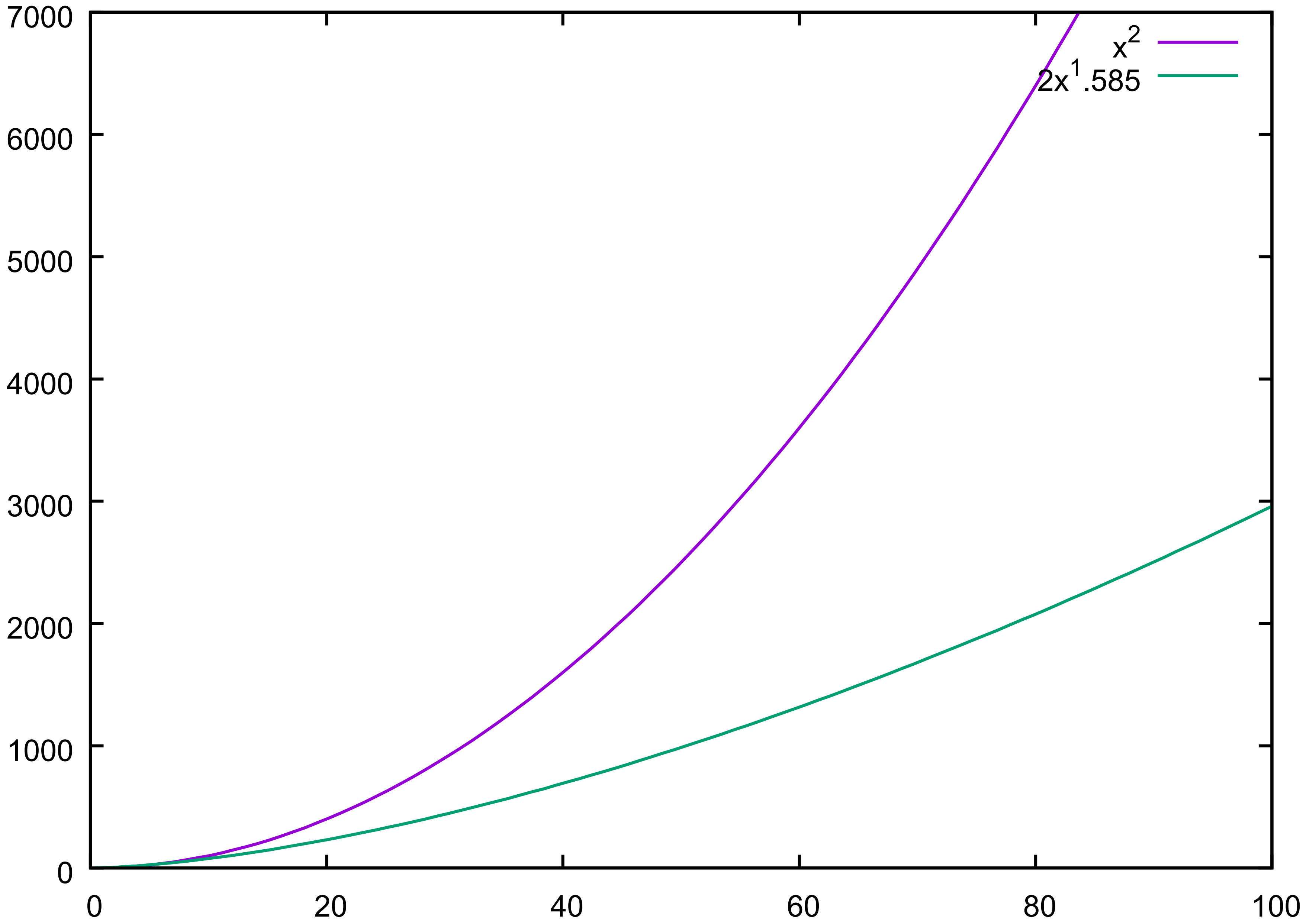
- The complexity of divide-and-conquer algorithms can be given as recurrences
- $T_1(n) = 4T_1(n/2) + \Theta(n)$ (slow version)
- $T_2(n) = 3T_2(n/2) + \Theta(n)$ (fast version)

Does this make a difference?

Slow versus fast multiplication



Slow versus fast multiplication



Multiplication: complexity

- $T_1(n) = 4T_1(n/2) + \Theta(n)$ (slow version)
  $T_1(n) \in \Theta(n^2)$
  $$\begin{aligned} T_1(n) &= 4T_1(n/2) + \Theta(n) \\ &= 16T_1(n/4) + 4\Theta(n/2) + \Theta(n) \\ &= 64T_1(n/8) + 16\Theta(n/4) + 4\Theta(n/2) + \Theta(n) \ldots \end{aligned}$$
- $T_2(n) = 3T_2(n/2) + \Theta(n)$ (fast version)
  $\mathrm{T}_2(n) \in \Theta(n^{1.585})$ $(\lg 3 \approx 1.585)$
- In practice: the 'fast' algorithm is faster for large inputs; it still pays off to use the 'slow' one for small inputs.

Maximum subarray

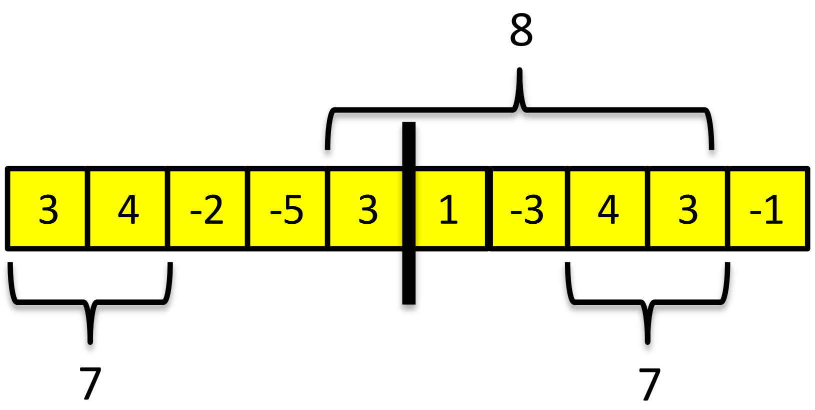
- Given an array of integers, find the contiguous subarray that has the largest sum

$$\boxed{3}\ \boxed{4}\ \boxed{-2}\ \boxed{-5}\ \underbrace{\boxed{3}\ \boxed{1}\ \boxed{-3}\ \boxed{4}\ \boxed{3}}_{8}\ \boxed{-1}$$

- Naïve algorithm: $\Theta(n^2)$

Maximum subarray: divide and conquer

- Divide the array into two halves, and solve the problem for both halves



- The resulting maximum subarray is either among the results from the subproblems, or is a subarray that crosses the border
- The cost of find the border-crossing maximum subarray is $\Theta(n)$

Maximum subarray: complexity

- $T_1(n) = 2T_1(n/2) + \Theta(n)$
  $T_1(n) \in \Theta(n \lg n)$
  $$\begin{aligned} T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &= 4T_1(n/4) + 2\Theta(n/2) + \Theta(n) \\ &= 8T_1(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \ldots \end{aligned}$$

Divide-and-conquer algorithms

- We have seen two divide-and-conquer algorithms
- In each, the problem is first **decomposed into subproblems**
- Then, the **subproblems are solved**
- Finally, the **results of solving the subproblems are combined**
- D-a-c algorithms may be easier to explain and implement
- D-a-c allows us to come up with optimisations
- Typically d-a-c algorithms are implemented using recursive functions
- D-a-c allows for parallelisation

Maximum subarray: implementation

Solving subproblems

```
public static int maxInterval (int[] arr
                                int lo, i   hi) {
  if (hi - lo == 1) return array[lo];
  int mid = lo + (hi - lo) / 2;

  int loRes = maxInterval (array, lo, mid);
  int hiRes = maxInterval (array, mid, hi);

  int maxBorder;
  // Find the maximum subarray that crosses the border
  // ...

  return Math.max(loRes, Math.max(hiRes, maxBorder));
}
```

Combining results

Maximum subarray: implementation

Solving subproblems

```
public static int maxInterval (int[] arr
                               int lo, i   hi) {
  if (hi - lo == 1) return array[lo];
  int mid = lo + (hi - lo) / 2;

  int loRes = maxInterval (array, lo, mid);
  int hiRes = maxInterval (array, mid, hi);

  int maxBorder;
  // Find the maximum subarray that crosses the border
  // ...

  return Math.max(loRes, Math.max(hiRes, maxBorder));
}
```

Combining results

Mergesort

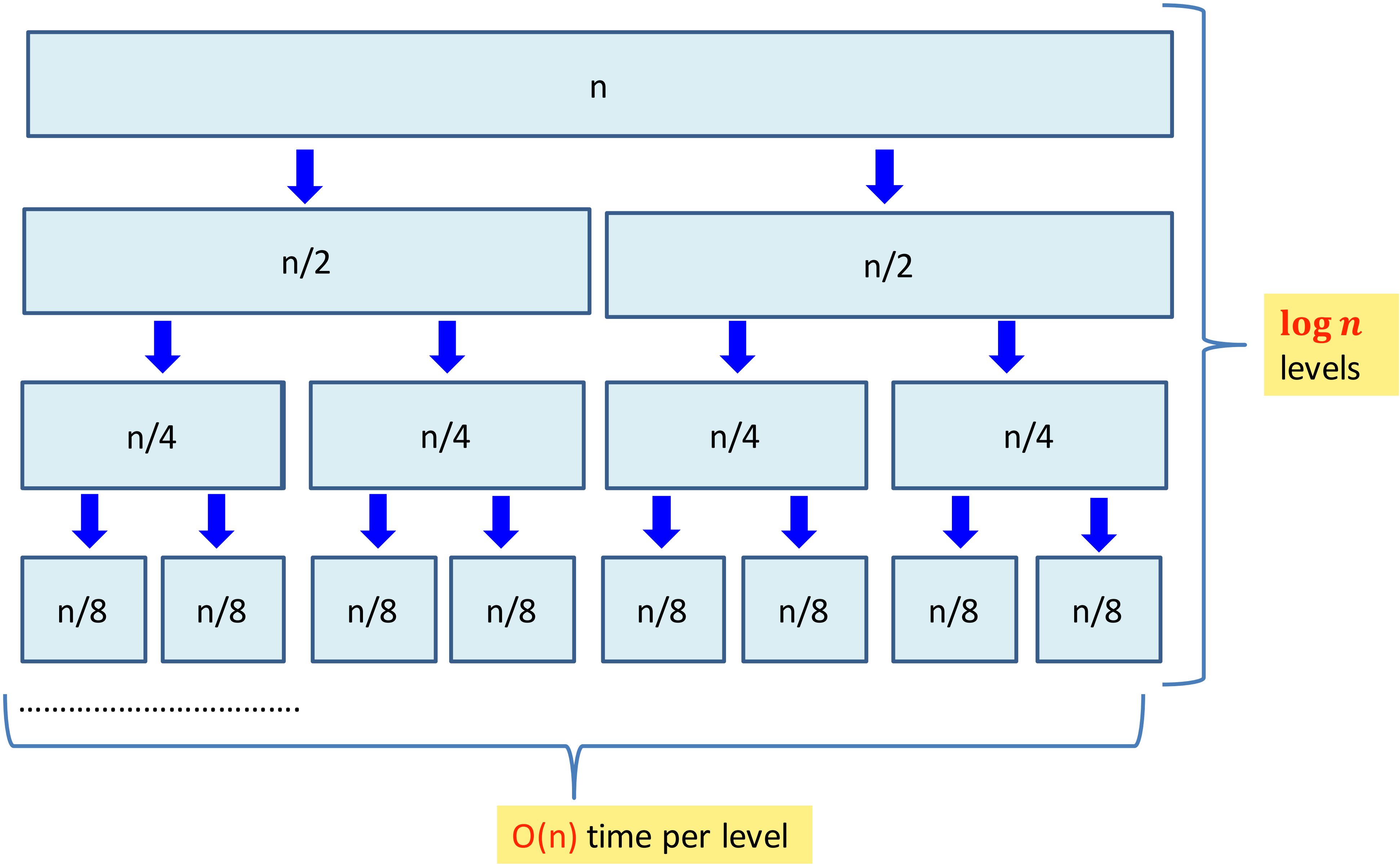
Mergesort

- Split the array into two subarrays
- Sort them recursively
- Merge the results ($\Theta(n)$)
- Complexity:

$$T(n) = 2T\left({}^{n}/_{2}\right) + \Theta(n)$$

Looks familiar

Mergesort (Complexity)



Solving recurrences

Solving recurrences

- Recurrences describing divide-and-conquer algorithm complexity often look as follows
$$T(n) = 2T\left({}^{n}/_{2}\right) + \Theta(n)$$
- To solve a recurrence (get a closed-form complexity), we may expand the recurrence to get an idea of the kind of a solution
$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 4T(n/4) + 2\Theta(n/2) + \Theta(n) \\ &= 8T(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \ \ldots \end{aligned}$$
- Once we make a guess e.g. $T(n) \in \Theta(n \lg n)$, we can try to prove it.

Solving recurrences, cont.

$$T(n) = 2T\left({}^{n}/_{2}\right) + \Theta(n)$$

- So we hypothesise that $T(n) \in \Theta(n \lg n)$
- We will first show that $T(n) \in O(n \lg n)$
- $T(n) \leq 2T({}^{n}/_{2}) + dn$ for some constant d
- We will perform a proof by induction
- From the induction hypothesis, $T({}^{n}/_{2}) \leq c\,{}^{n}/_{2} \lg {}^{n}/_{2}$ for some constant c
- We have $T(n) \leq 2c\,{}^{n}/_{2} \lg {}^{n}/_{2} + dn = cn\,(\lg n - 1) + dn = cn \lg n - cn + dn$
- Since we can increase constant c, we can ensure that $c > d$, and thus $T(n) \leq cn \lg n$

Solving recurrences, cont.

$$T(n) = 2T\left({}^{n}/_{2}\right) + \Theta(n)$$

- So we hypothesise that $T(n) \in \Theta(n \lg n)$
- After showing that $T(n) \in O(n \lg n)$, we can show that $n \lg n \in O(T(n))$
- $T(n) \geq 2T({}^{n}/_{2}) + dn$ for some constant d
- This part is usually easier
- From the induction hypothesis, $T({}^{n}/_{2}) \geq c\, {}^{n}/_{2} \lg {}^{n}/_{2}$ for some constant c
- We have $T(n) \geq 2c\, {}^{n}/_{2} \lg {}^{n}/_{2} + dn = cn\,(\lg n - 1) + dn = cn \lg n - cn + dn$
- Since we can decrease constant c, we can ensure that $c < d$, and thus $T(n) \geq cn \lg n$

Some pitfalls

- Instead of $T(n) = 2T(n/_2) + \Theta(n)$ it is really
$$T(n) = T\left(\lfloor n/_2 \rfloor\right) + T\left(\lceil n/_2 \rceil\right) + \Theta(n)$$
In most cases it is not a problem (the computations work in the same way)
- The O and Θ definitions talk about all n above n_0
But since we are interested in asymptotic results, we only care about what happens above certain numbers

Solving recurrences, cont.

- Expand recurrence
- Make a guess
- Try to prove it

The master method

- But… if the recurrence has he form

$$T(n) = aT\left({}^{n}/_{b}\right) + f(n)$$

  then there is a ‘cookbook’ solution

The master method, cont.

- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n) = aT({}^{n}/_{b}) + f(n)$, where we interpret ${}^{n}/_{b}$ to mean either $\lceil {}^{n}/_{b} \rceil$ or $\lfloor {}^{n}/_{b} \rfloor$. Then $T(n)$ has the following asymptotic bounds:
- If $f(n) \in O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$ then $T(n) \in \Theta(n^{\log_b a})$
- If $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} \lg n)$
- If $n^{\log_b a+\epsilon} \in O(f(n))$ for some constant $\epsilon > 0$, and if $af({}^{n}/_{b}) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n then $T(n) \in \Theta(f(n))$
- See ch. 4, *Introduction to Algorithms*, T. H. Cormen et. al., or ch. 5 from the textbook

Quicksort

Partition

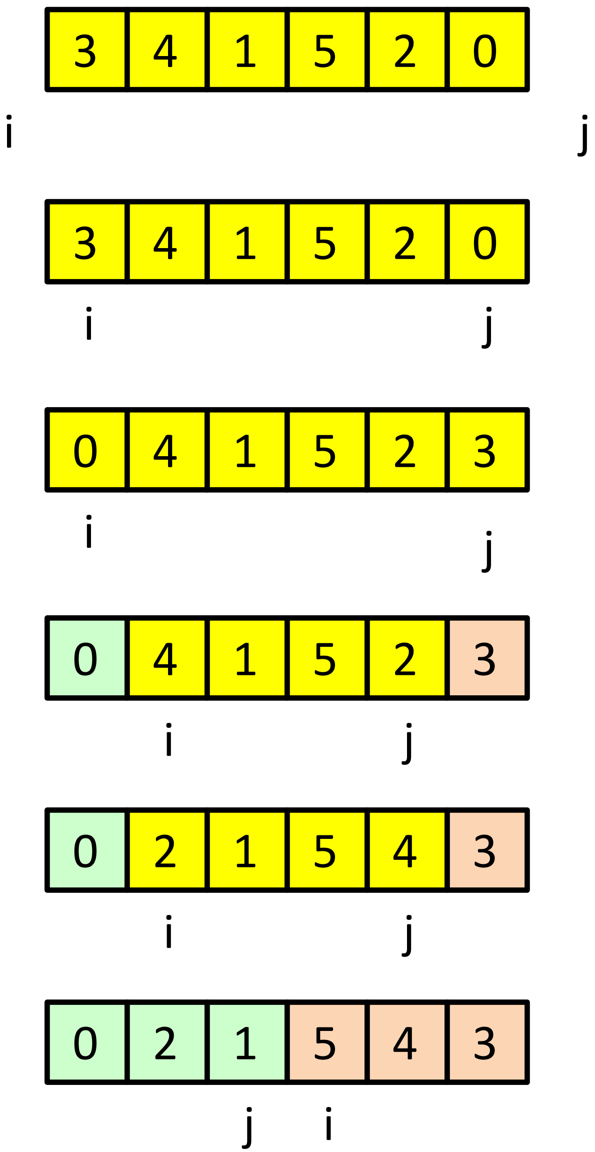
- Recall the code for partitioning an array

```
private static int partition (int[] array,
                                int lo, int hi) {
    if (hi - lo <= 1) return 0;

    int pivot = array[lo];
    int i = -1;
    int j = hi;

    while(true) {
      do { ++i; } while (array[i] < pivot);
      do { --j; } while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Example execution




```java
private static int partition
(int[] array, int lo, int hi) {
    if (hi - lo <= 1) return 0;

    int pivot = array[lo];
    int i = -1;
    int j = hi;

    while(true) {
      do { ++i; }
      while (array[i] < pivot);
      do { --j; }
      while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Quicksort

- Here is the code of quicksort proper

```
private static void quicksort(int[] array,
                                int lo, int hi) {
  if (hi - lo <= 1) return;
  int p = partition(array, lo, hi);
  quicksort(array, lo, p);
  quicksort(array, p, hi);
}
```

- We partition the array, and then perform quicksort recursively for the resulting subarrays

Quicksort: complexity

- Worst case: the pivot element is always the smallest one
  $T(n) = T(n-1) + \Theta(n)$, $T(n) \in \Theta(n^2)$
- Average case: we need an assumption on the distribution of all possible arrays that we will get as inputs
- If the elements of the array are randomly placed (have random positions), then we can expect the partition to be approximately balanced, for example in 1:9 ratio

$$T(n) = T\left({}^{9}/_{10}\, n\right) + T\left({}^{1}/_{10}\, n\right) + \Theta(n)$$

Quicksort: complexity

- $T(n) = T(^9/_{10}\, n) + T(^1/_{10}\, n) + \Theta(n)$
  $= T\left(^{81}/_{100}\, n\right) + T\left(^9/_{100}\, n\right) + \Theta\left(^9/_{10}\, n\right) + T(^9/_{100}\, n)$
  $+ \mathrm{T}(^1/_{100}\, n) + \Theta(^1/_{10}\, n) + \Theta(n)$
- Each level of the expansion will be at most $\Theta(n)$, and the will be at most $\Theta(\lg n)$ levels
- This argument makes a ‘reasonable’, but false assumption
- For a rigorous argument, see ch. 7, *Introduction to Algorithms*, T. H. Cormen et. al.

What could possibly go wrong?

- Worst-case complexity: $\Theta(n^2)$
- Will this terminate at all in the worst case?
- Are there no off-by-one errors in the code?

```
private static void quicksort(int[] array,
                              int lo, int hi) {
  if (hi - lo <= 1) return;
  int p = partition(array, lo, hi);
  quicksort(array, lo, p);
  quicksort(array, p, hi);
}
```

Pre-conditions and post-conditions

- **Pre-condition** is a condition that concerns inputs to a method, and/or the state of the object and/or the environment before the invocation
- **Post-condition** is a condition that concerns the result of a method and/or the state of the object and/or the environment after the invocation
- We can state that a particular pre-condition is required before an invocation of a method
- We can state that a particular post-condition holds after an invocation of a method
- Pre- and post-conditions can also concern lines of code/blocks

Pre-conditions and post-conditions: example

- Pre-condition 1: $lo \geq 0, hi \geq 0, lo \leq hi$
- Post-condition 1: given pre-condition 1, $\mathrm{lo} \leq res \leq hi$
- Post-condition 2: …

```
 private static int mid(int lo, int hi) {
  return lo + (hi - lo) / 2;
}
```

What post-conditions do we need?

```
private static void quicksort(int[] array,
                              int lo, int hi) {
  if (hi - lo <= 1) return;
  int p = partition(array, lo, hi);
  quicksort(array, lo, p);
  quicksort(array, p, hi);
}
```

- We need to be sure that $lo < p < hi$, as otherwise the procedure will loop (infinite recursion)
- We need to be sure that after partition all elements of the array before index p are not greater than all elements starting from $p + 1$ on

How to show the first post-condition?

- We want to show that when the return statement is executed (`return j + 1`), then $lo < j + 1 < hi$ holds
- But the method contains loops
- We will need **invariants**

```
private static int partition
(int[] array, int lo, int hi) {
    if (hi - lo <= 1) return 0;

    int pivot = array[lo];
    int i = -1;
    int j = hi;

    while(true) {
      do { ++i; }
      while (array[i] < pivot);
      do { --j; }
      while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Invariants

- Invariant is a condition that is always true in a given method/block of code, possibly assuming pre-condition
- Exampe: pre-condition: $array.length > 0$
- Invariant: if cur and I are defined, then $cur = maximum(array[0], \ldots, array[i-1])$

```
private static int maximum (int[] array) {
    if (array.length == 0)
      throw new IllegalArgumentException
                 ("maximum requires a non-empty array");
    int cur = array[0];
    for (int i = 1; i < array.length; ++i) {
      cur = Math.max(cur, array[i]);
    }
    return cur;
 }
```

How to show the first post-condition?

- We want to show that when the return statement is executed (`return j + 1`), then $lo < j + 1 < hi$ holds
- Invariant: $array[k] \leq pivot \; \forall k < i$, $array[k] \geq pivot \; \forall k > j$

```
private static int partition
(int[] array, int lo, int hi) {
    if (hi - lo <= 1) return 0;

    int pivot = array[lo];
    int i = -1;
    int j = hi;

    while(true) {
      do { ++i; }
      while (array[i] < pivot);
      do { --j; }
      while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

How to show the first post-condition?

- We want to show that when the return statement is executed (`return j + 1`), then $lo < j+1 < hi$ holds
- Invariant:

```
private static int partition
(int[] array, int lo, int hi) {
    if (hi - lo <= 1) return 0;

    int pivot = array[lo];
    int i = -1;
    int j = hi;

    while(true) {
      do { ++i; }
      while (array[i] < pivot);
      do { --j; }
      while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Conclusion

- **Divide-and-conquer** algorithms offer an effective way of solving certain problems
- The complexity of divide-and-conquer algorithms can be often estimated using recurrences
- Quicksort is a fast sorting algorithm, but its worst-case performance is $\Theta(n^2)$, and its analysis is pretty complex