Algorithms: COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au
office: 504 (CSE building K 17)
Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
University of New South Wales Sydney

2. DIVIDE-AND-CONQUER

An old puzzle

Problem

We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Hint

You can reduce the search space by a third in one weighing!

An old puzzle

Problem

We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Hint

You can reduce the search space by a third in one weighing!

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

Divide and Conquer

- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: merge sort.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Divide and Conquer

- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: merge sort.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Divide and Conquer

- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: merge sort.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Divide and Conquer

- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: merge sort.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Counting the number of inversions

- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).
- How should we measure the degree of similarity of two users A and B?
- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.
- For the i^{th} movie on $B's$ list we can now look at the position (i.e., index) of that movie on $A's$ list, denoted by $a(i)$.

Counting the number of inversions

- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).

- How should we measure the degree of similarity of two users A and B?

- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.

- For the i^{th} movie on $B's$ list we can now look at the position (i.e., index) of that movie on $A's$ list, denoted by $a(i)$.

Counting the number of inversions

- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).
- How should we measure the degree of similarity of two users A and B?
- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.
- For the i^{th} movie on $B's$ list we can now look at the position (i.e., index) of that movie on $A's$ list, denoted by $a(i)$.

Counting the number of inversions

- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).
- How should we measure the degree of similarity of two users A and B?
- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.
- For the i^{th} movie on $B's$ list we can now look at the position (i.e., index) of that movie on $A's$ list, denoted by $a(i)$.

Counting the number of inversions

- A good measure of how different these two users are, is the total number of *inversions*, i.e., total number of pairs of movies i, j such that movie i precedes movie j on $B's$ list but movie j is higher up on $A's$ list than the movie i.

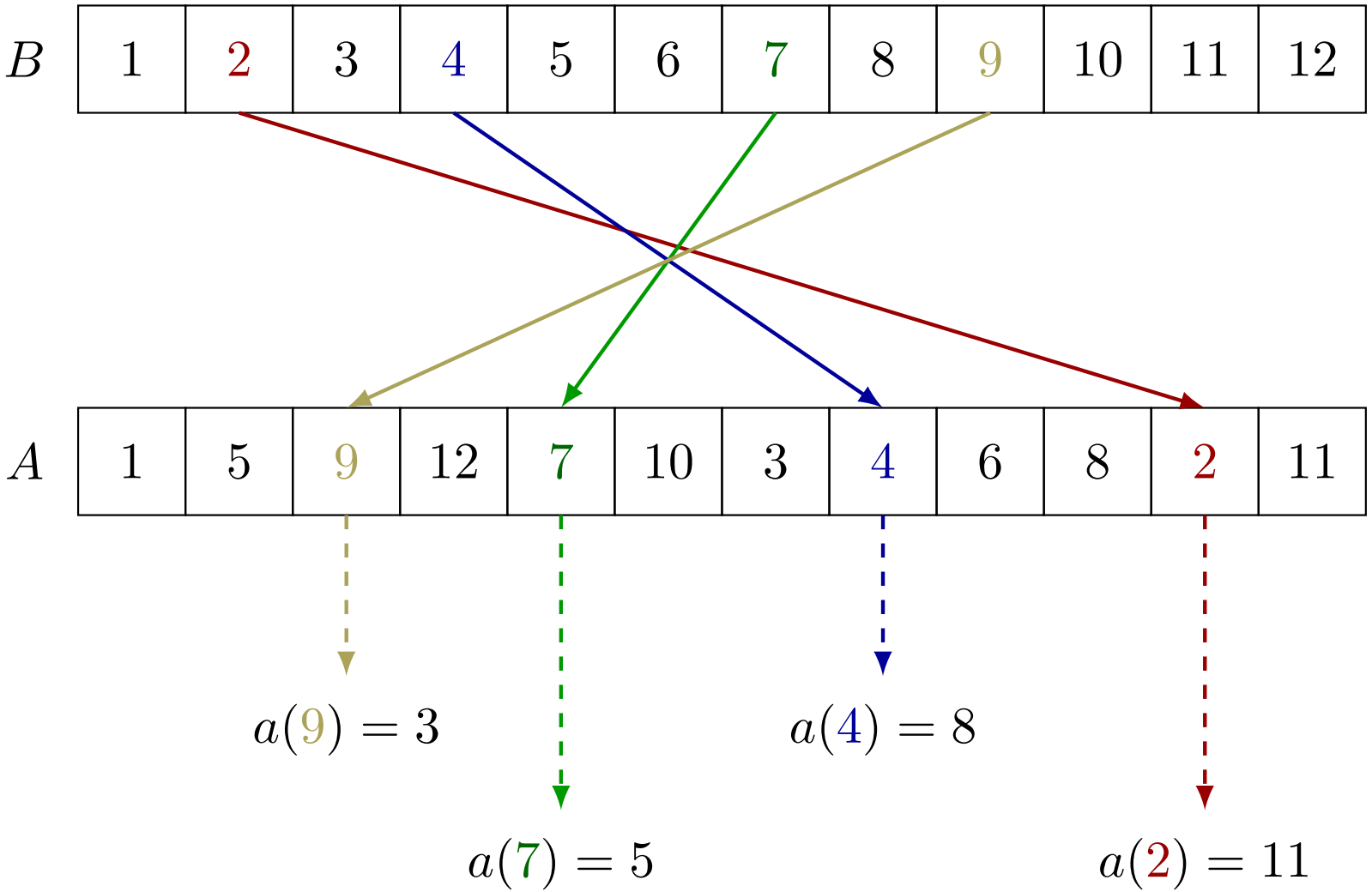
- In other words, we count the number of pairs of movies i, j such that $i < j$ (movie i precedes movie j on $B's$ list) but $a(i) > a(j)$ (movie i is in the position $a(i)$ on $A's$ list which is after the position $a(j)$ of movie j on $A's$ list.

Counting the number of inversions

- A good measure of how different these two users are, is the total number of *inversions*, i.e., total number of pairs of movies i, j such that movie i precedes movie j on $B's$ list but movie j is higher up on $A's$ list than the movie i.

- In other words, we count the number of pairs of movies i, j such that $i < j$ (movie i precedes movie j on $B's$ list) but $a(i) > a(j)$ (movie i is in the position $a(i)$ on $A's$ list which is after the position $a(j)$ of movie j on $A's$ list.

Counting the number of inversions



Counting the number of inversions

- For example 1 and 2 do not form an inversion because $a(1) < a(2)$ ($a(1) = 1$ and $a(2) = 11$ because $a(1)$ is on the first and $a(2)$ is on the 11^{th} place in A);
- However, for example 4 and 7 do form an inversion because $a(7) < a(4)$ ($a(7) = 5$ because seven is on the fifth place in A and $a(4) = 8$)

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of movies on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.
- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a DIVIDE-AND-CONQUER strategy.
- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.
- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of movies on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.

- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a DIVIDE-AND-CONQUER strategy.

- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.

- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of movies on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.

- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a DIVIDE-AND-CONQUER strategy.

- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.

- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of movies on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.

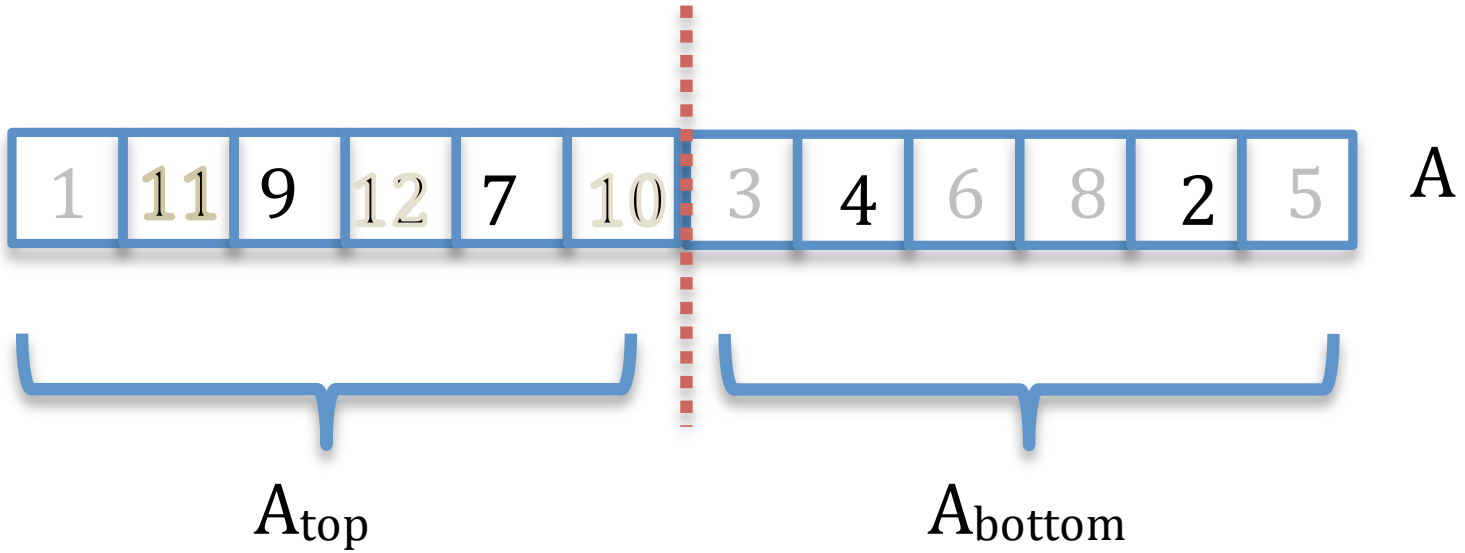
- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a DIVIDE-AND-CONQUER strategy.

- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.

- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

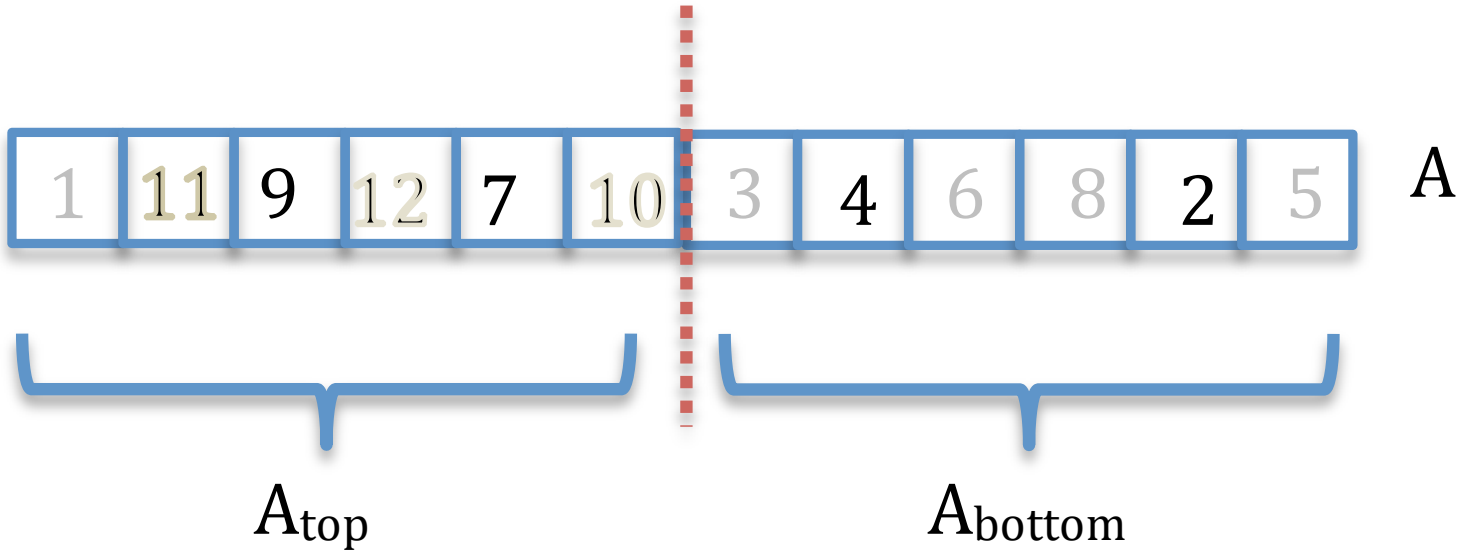
Counting the number of inversions

- We split the array A into two (approximately) equal parts $A_{top} = A[1 \ldots \lfloor n/2 \rfloor]$ and $A_{bottom} = A[\lfloor n/2 \rfloor + 1 \ldots n]$.
- Note that the total number of inversions in array A is equal to the sum of the number of inversions $I(A_{top})$ in A_{top} (such as 9 and 7) plus the number of inversions $I(A_{bottom})$ in A_{bottom} (such as 4 and 2) plus the number of inversions $I(A_{top}, A_{bottom})$ across the two halves (such as 7 and 4).



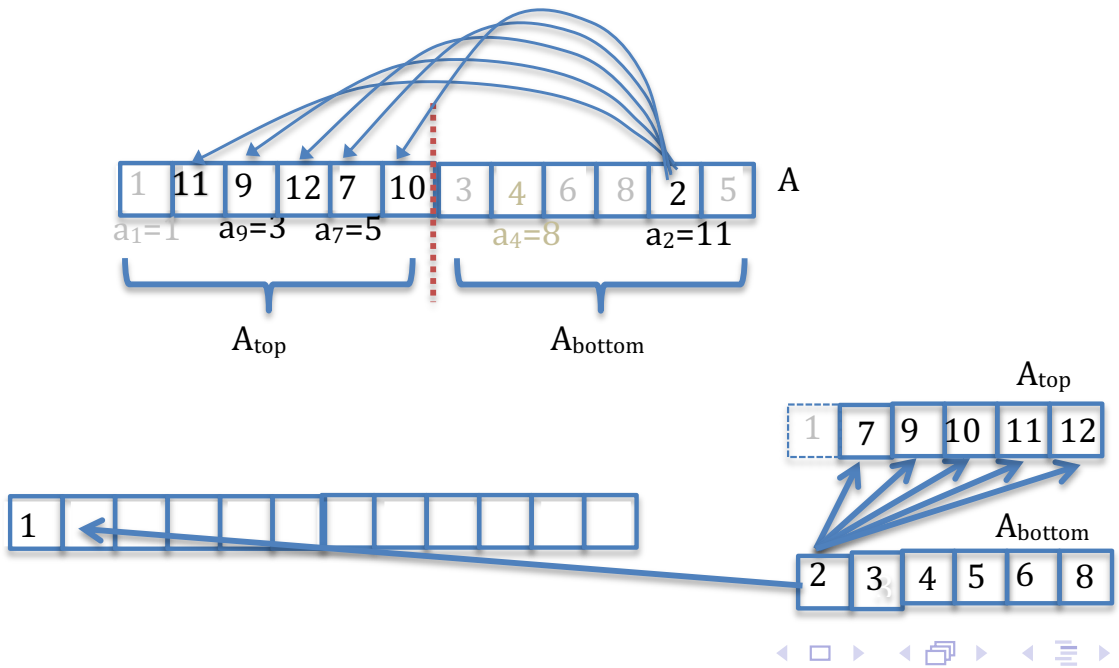
Counting the number of inversions

- We split the array A into two (approximately) equal parts $A_{top} = A[1 \ldots \lfloor n/2 \rfloor]$ and $A_{bottom} = A[\lfloor n/2 \rfloor + 1 \ldots n]$.
- Note that the total number of inversions in array A is equal to the sum of the number of inversions $I(A_{top})$ in A_{top} (such as 9 and 7) plus the number of inversions $I(A_{bottom})$ in A_{bottom} (such as 4 and 2) plus the number of inversions $I(A_{top}, A_{bottom})$ across the two halves (such as 7 and 4).



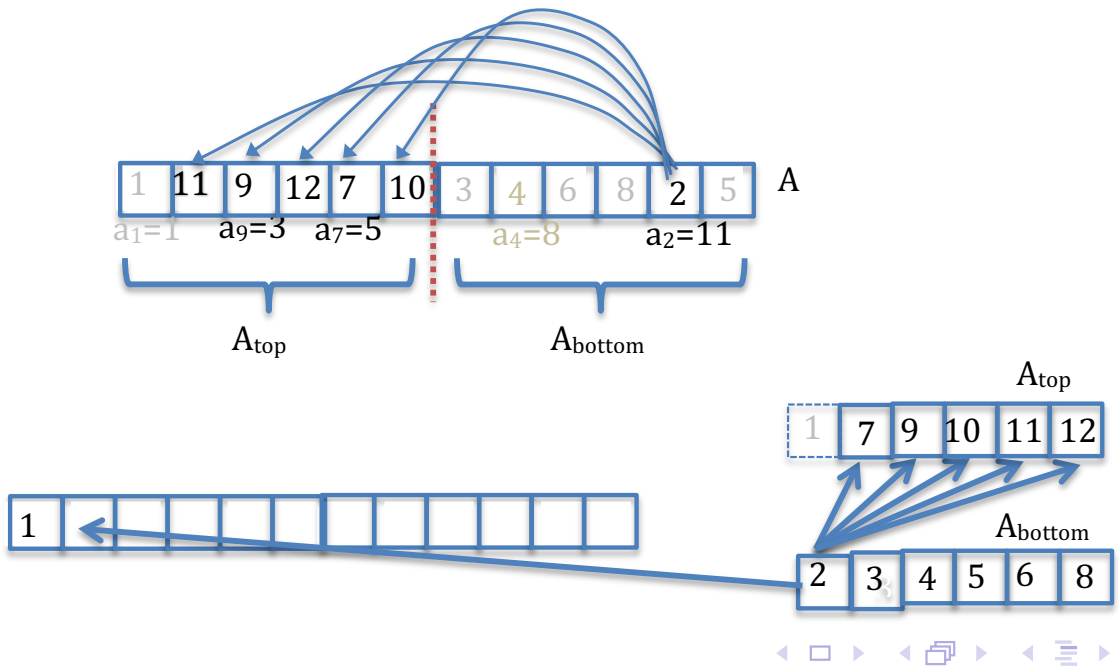
Counting the number of inversions

- We now recursively sort arrays A_{top} and A_{bottom} also obtaining in the process the number of inversions $I(A_{top})$ in the sub-array A_{top} and the number of inversions $I(A_{bottom})$ in the sub-array A_{bottom}.
- We now merge the two sorted arrays A_{top} and A_{bottom} while counting the number of inversions $I(A_{top}, A_{bottom})$ which are across the two sub-arrays.
- When the next smallest element among all elements in both arrays is an element in A_{bottom}, such an element clearly is in an inversion with all the remaining elements in A_{top} and we add the total number of elements remaining in A_{top} to the current value of the number of inversions across A_{top} and A_{bottom}.



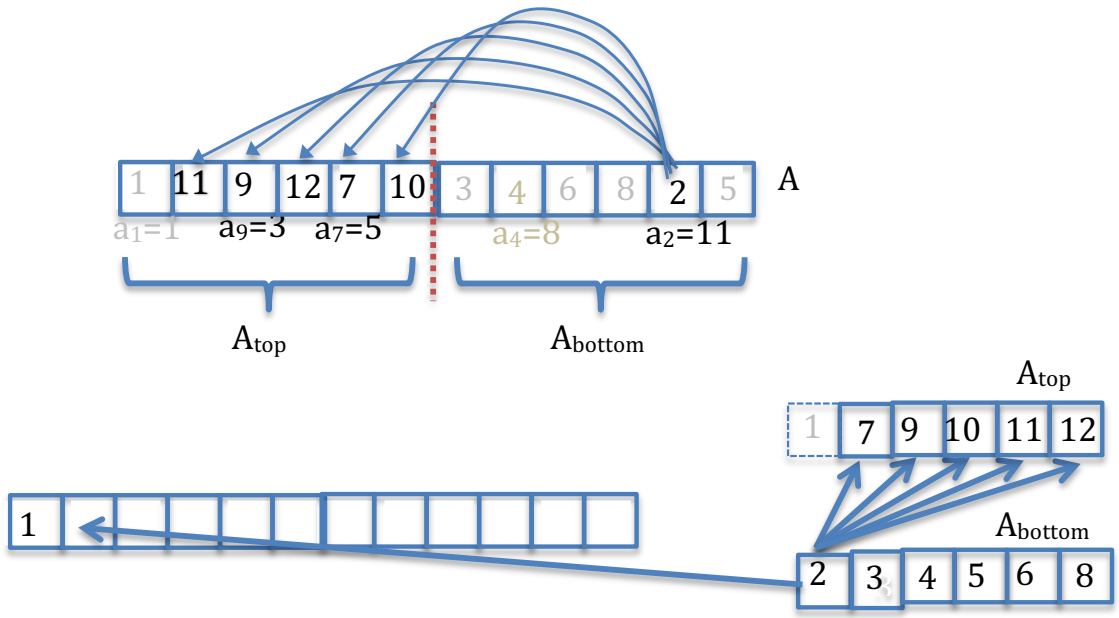
Counting the number of inversions

- We now recursively sort arrays A_{top} and A_{bottom} also obtaining in the process the number of inversions $I(A_{top})$ in the sub-array A_{top} and the number of inversions $I(A_{bottom})$ in the sub-array A_{bottom}.
- We now merge the two sorted arrays A_{top} and A_{bottom} while counting the number of inversions $I(A_{top}, A_{bottom})$ which are across the two sub-arrays.
- When the next smallest element among all elements in both arrays is an element in A_{bottom}, such an element clearly is in an inversion with all the remaining elements in A_{top} and we add the total number of elements remaining in A_{top} to the current value of the number of inversions across A_{top} and A_{bottom}.



Counting the number of inversions

- We now recursively sort arrays A_{top} and A_{bottom} also obtaining in the process the number of inversions $I(A_{top})$ in the sub-array A_{top} and the number of inversions $I(A_{bottom})$ in the sub-array A_{bottom}.
- We now merge the two sorted arrays A_{top} and A_{bottom} while counting the number of inversions $I(A_{top}, A_{bottom})$ which are across the two sub-arrays.
- When the next smallest element among all elements in both arrays is an element in A_{bottom}, such an element clearly is in an inversion with all the remaining elements in A_{top} and we add the total number of elements remaining in A_{top} to the current value of the number of inversions across A_{top} and A_{bottom}.



Counting the number of inversions

- Whenever the next smallest element among all elements in both arrays is an element in A_{top}, such an element clearly is not involved in any inversions across the two arrays (such as 1, for example).
- After the merging operation is completed, we obtain the total number of inversions $I(A_{top}, A_{bottom})$ across A_{top} and A_{bottom}.
- The total number of inversions $I(A)$ in array A is finally obtained as:

$$I(A) = I(A_{top}) + I(A_{bottom}) + I(A_{top}, A_{bottom})$$

- **Next:** we study applications of divide and conquer to arithmetic of very large integers.

Counting the number of inversions

- Whenever the next smallest element among all elements in both arrays is an element in A_{top}, such an element clearly is not involved in any inversions across the two arrays (such as 1, for example).
- After the merging operation is completed, we obtain the total number of inversions $I(A_{top}, A_{bottom})$ across A_{top} and A_{bottom}.
- The total number of inversions $I(A)$ in array A is finally obtained as:

$$I(A) = I(A_{top}) + I(A_{bottom}) + I(A_{top}, A_{bottom})$$

- **Next:** we study applications of divide and conquer to arithmetic of very large integers.

Counting the number of inversions

- Whenever the next smallest element among all elements in both arrays is an element in A_{top}, such an element clearly is not involved in any inversions across the two arrays (such as 1, for example).

- After the merging operation is completed, we obtain the total number of inversions $I(A_{top}, A_{bottom})$ across A_{top} and A_{bottom}.

- The total number of inversions $I(A)$ in array A is finally obtained as:

$$I(A) = I(A_{top}) + I(A_{bottom}) + I(A_{top}, A_{bottom})$$

- **Next:** we study applications of divide and conquer to arithmetic of very large integers.

Counting the number of inversions

- Whenever the next smallest element among all elements in both arrays is an element in A_{top}, such an element clearly is not involved in any inversions across the two arrays (such as 1, for example).

- After the merging operation is completed, we obtain the total number of inversions $I(A_{top}, A_{bottom})$ across A_{top} and A_{bottom}.

- The total number of inversions $I(A)$ in array A is finally obtained as:

$$I(A) = I(A_{top}) + I(A_{bottom}) + I(A_{top}, A_{bottom})$$

- **Next:** we study applications of divide and conquer to arithmetic of very large integers.

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X     first integer
+  X X X X X     second integer
  -----------
 X X X X X X     result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we add two numbers?

```
 C C C C C      carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
---------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two X's can be multiplied in $O(1)$. time (each X could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
---------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$. time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
---------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$. time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
--------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two X's can be multiplied in $O(1)$. time (each X could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
--------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$. time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Basics revisited: how do we multiply two numbers?

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
--------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$. time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_0,\ B_0$ - the least significant bits; $A_1,\ B_1$ the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \qquad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \qquad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \qquad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

$$\underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \quad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \qquad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

$$\underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \quad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- A_0, B_0 - the least significant bits; A_1, B_1 the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \qquad (1)$$

What we mean is that the product AB can be calculated recursively by the following program:

```
1:  function MULT(A, B)
2:      if |A| = |B| = 1 then return AB
3:      else
4:          A_1 ← MoreSignificantPart(A);
5:          A_0 ← LessSignificantPart(A);
6:          B_1 ← MoreSignificantPart(B);
7:          B_0 ← LessSignificantPart(B);
8:          X ← MULT(A_0, B_0);
9:          Y ← MULT(A_0, B_1);
10:         Z ← MULT(A_1, B_0);
11:         W ← MULT(A_1, B_1);
12:         return W 2^n + (Y + Z) 2^{n/2} + X
13:     end if
14: end function
```

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n \tag{2}$$

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{2}$$

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{2}$$

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n \tag{2}$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$\begin{aligned} T(n) &= 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn \\ &= n^2(c+1) - 2cn + cn = n^2(c+1) - cn \end{aligned}$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$T(n) = 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn$$
$$= n^2(c+1) - 2cn + cn = n^2(c+1) - cn$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$\begin{aligned} T(n) &= 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn \\ &= n^2(c+1) - 2cn + cn = n^2(c+1) - cn \end{aligned}$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$T(n) = 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn$$
$$= n^2(c+1) - 2cn + cn = n^2(c+1) - cn$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$T(n) = 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn$$

$$= n^2(c+1) - 2cn + cn = n^2(c+1) - cn$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \tag{3}$$

then

$$T(n) = n^2(c+1) - cn$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\frac{n}{2}$$

and prove that it is also true for n:

$$\begin{aligned} T(n) &= 4T\left(\tfrac{n}{2}\right) + cn = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}c\right) + cn \\ &= n^2(c+1) - 2cn + cn = n^2(c+1) - cn \end{aligned}$$

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n$ then

$$T(n) = n^2(c+1) - c\,n = \Theta(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + A_0 B_1) 2^{\frac{n}{2}} + A_0 B_0$$

$$= A_1 B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0) 2^{\frac{n}{2}} + A_0 B_0$$

- So we have saved one multiplication at each recursion round!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

$$\underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

$$\underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

- Thus, the algorithm will look like this:

1: **function** MULT(A, B)
2: **if** $|A| = |B| = 1$ **then return** AB
3: **else**
4: $A_1 \leftarrow \text{MoreSignificantPart}(A)$;
5: $A_0 \leftarrow \text{LessSignificantPart}(A)$;
6: $B_1 \leftarrow \text{MoreSignificantPart}(B)$;
7: $B_0 \leftarrow \text{LessSignificantPart}(B)$;
8: $U \leftarrow A_0 + A_1$;
9: $V \leftarrow B_0 + B_1$;
10: $X \leftarrow \text{MULT}(\text{A}_0, \text{B}_0)$;
11: $W \leftarrow \text{MULT}(\text{A}_1, \text{B}_1)$;
12: $Y \leftarrow \text{MULT}(\text{U}, \text{V})$;
13: **return** $W\,2^n + (Y - X - W)\,2^{n/2} + X$
14: **end if**
15: **end function**

- How fast is this algorithm?

- Thus, the algorithm will look like this:

1: **function** MULT(A, B)
2: **if** $|A| = |B| = 1$ **then return** AB
3: **else**
4: $A_1 \leftarrow \text{MoreSignificantPart}(A)$;
5: $A_0 \leftarrow \text{LessSignificantPart}(A)$;
6: $B_1 \leftarrow \text{MoreSignificantPart}(B)$;
7: $B_0 \leftarrow \text{LessSignificantPart}(B)$;
8: $U \leftarrow A_0 + A_1$;
9: $V \leftarrow B_0 + B_1$;
10: $X \leftarrow \text{MULT}(\text{A}_0, \text{B}_0)$;
11: $W \leftarrow \text{MULT}(\text{A}_1, \text{B}_1)$;
12: $Y \leftarrow \text{MULT}(\text{U}, \text{V})$;
13: **return** $W\,2^n + (Y - X - W)\,2^{n/2} + X$
14: **end if**
15: **end function**

- How fast is this algorithm?

How many multiplications does this take? (addition is in linear time!)

We need A_1B_1, A_0B_0 and $(A_1 + A_0)(B_1 + B_0)$; thus

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

How many multiplications does this take? (addition is in linear time!)
We need A_1B_1, A_0B_0 and $(A_1 + A_0)(B_1 + B_0)$; thus

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

How many multiplications does this take? (addition is in linear time!)
We need A_1B_1, A_0B_0 and $(A_1 + A_0)(B_1 + B_0)$; thus

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)}_{\cdots} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)}_{} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)}_{} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)}_{\cdots} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)}_{} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)}_{} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}_{\cdots}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

$$\dots$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \dots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)}_{\cdots} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}_{\cdots}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)}_{\cdots} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}_{\cdots}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)}_{\cdots} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}_{\cdots}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)}_{\cdots} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}_{\cdots}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)}_{\cdots} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}_{\cdots}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)}_{\cdots} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}_{\cdots}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

So we get $T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)}_{\cdots} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}_{\cdots}\right) + c\,n$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)}_{\cdots} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}_{\cdots}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)}_{\cdots} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}_{\cdots}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \ldots$$

The Karatsuba trick

$$T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\,\underbrace{\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right)} + c\,n = 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\tfrac{3n}{2} + c\,n$$

$$= 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\tfrac{3n}{2} + c\,n = 3^3 T\left(\tfrac{n}{2^3}\right) + c\tfrac{3^2 n}{2^2} + c\tfrac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^4 T\left(\tfrac{n}{2^4}\right) + c\,n\left(\tfrac{3^3}{2^3} + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\tfrac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + c\,n\left(\left(\tfrac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + c\,n\,\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = 3^{\log_2 n} T(1) + 2c\,n\left(\left(\tfrac{3}{2}\right)^{\log_2 n} - 1\right)$$

The Karatsuba trick

$$T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\,\underbrace{\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right)} + c\,n = 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\tfrac{3n}{2} + c\,n$$

$$= 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\tfrac{3n}{2} + c\,n = 3^3 T\left(\tfrac{n}{2^3}\right) + c\tfrac{3^2 n}{2^2} + c\tfrac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^4 T\left(\tfrac{n}{2^4}\right) + c\,n\left(\tfrac{3^3}{2^3} + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\tfrac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + c\,n\left(\left(\tfrac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + c\,n\,\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = 3^{\log_2 n} T(1) + 2c\,n\left(\left(\tfrac{3}{2}\right)^{\log_2 n} - 1\right)$$

The Karatsuba trick

$$T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\,\underbrace{\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right)} + c\,n = 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\tfrac{3n}{2} + c\,n$$

$$= 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\tfrac{3n}{2} + c\,n = 3^3 T\left(\tfrac{n}{2^3}\right) + c\tfrac{3^2 n}{2^2} + c\tfrac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^4 T\left(\tfrac{n}{2^4}\right) + c\,n\left(\tfrac{3^3}{2^3} + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\tfrac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + c\,n\left(\left(\tfrac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + c\,n\,\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = 3^{\log_2 n} T(1) + 2c\,n\left(\left(\tfrac{3}{2}\right)^{\log_2 n} - 1\right)$$

The Karatsuba trick

$$T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\,\underbrace{\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right)} + c\,n = 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\tfrac{3n}{2} + c\,n$$

$$= 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\tfrac{3n}{2} + c\,n = 3^3 T\left(\tfrac{n}{2^3}\right) + c\tfrac{3^2 n}{2^2} + c\tfrac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^4 T\left(\tfrac{n}{2^4}\right) + c\,n\left(\tfrac{3^3}{2^3} + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\tfrac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + c\,n\left(\left(\tfrac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + c\,n\,\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = 3^{\log_2 n} T(1) + 2c\,n\left(\left(\tfrac{3}{2}\right)^{\log_2 n} - 1\right)$$

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$T(n) \approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right)$$
$$= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn$$
$$= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$T(n) \approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right)$$
$$= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn$$
$$= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$T(n) \approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right)$$

$$= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn$$

$$= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$T(n) \approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right)$$

$$= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn$$

$$= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$\begin{aligned} T(n) &\approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right) \\ &= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn \\ &= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2 \end{aligned}$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2cn \left(\left(\frac{3}{2} \right)^{\log_2 n} - 1 \right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$\begin{aligned} T(n) &\approx n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 \frac{3}{2}} - 1 \right) = n^{\log_2 3} T(1) + 2cn \left(n^{\log_2 3 - 1} - 1 \right) \\ &= n^{\log_2 3} T(1) + 2cn^{\log_2 3} - 2cn \\ &= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2 \end{aligned}$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

Some Important Questions:

1. Can we multiply large integers faster than $O\left(n^{\log_2 3}\right)$??
2. Can we avoid messy computations like:

$$T(n) = 3T\left(\frac{n}{2}\right) + cn = 3\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn = 3^2 T\left(\frac{n}{2^2}\right) + c\frac{3n}{2} + cn$$

$$= 3^2\left(3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{3n}{2} + cn = 3^3 T\left(\frac{n}{2^3}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + cn$$

$$= 3^3 T\left(\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) =$$

$$= 3^3\left(3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) =$$

$$= 3^4 T\left(\frac{n}{2^4}\right) + cn\left(\frac{3^3}{2^3} + \frac{3^2}{2^2} + \frac{3}{2} + 1\right) =$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\frac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + cn\left(\left(\frac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \frac{3^2}{2^2} + \frac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + cn\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1}$$

$$= 3^{\log_2 n} T(1) + 2cn\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)$$

Some Important Questions:

1. Can we multiply large integers faster than $O\left(n^{\log_2 3}\right)$??
2. Can we avoid messy computations like:

$$
\begin{aligned}
T(n) &= 3T\left(\frac{n}{2}\right) + cn = 3\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn = 3^2 T\left(\frac{n}{2^2}\right) + c\frac{3n}{2} + cn \\
&= 3^2\left(3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{3n}{2} + cn = 3^3 T\left(\frac{n}{2^3}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + cn \\
&= 3^3 T\left(\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&= 3^3\left(3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&= 3^4 T\left(\frac{n}{2^4}\right) + cn\left(\frac{3^3}{2^3} + \frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&\cdots \\
&= 3^{\lfloor \log_2 n \rfloor} T\left(\frac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + cn\left(\left(\frac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \frac{3^2}{2^2} + \frac{3}{2} + 1\right) \\
&\approx 3^{\log_2 n} T(1) + cn\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} \\
&= 3^{\log_2 n} T(1) + 2cn\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)
\end{aligned}
$$

Recurrences

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.

MERGE-SORT(A, p, r) *sorting A[p..r]*

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
3. Merge-Sort(A, p, q)
4. Merge-Sort$(A, q+1, r)$
5. Merge(A, p, q, r)

- Since Merge(A, p, q, r) runs in linear time, the runtime $T(n)$ of Merge-Sort(A, p, r) satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Recurrences

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.

MERGE-SORT(A, p, r) *sorting A[p..r]*

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
3. Merge-Sort(A, p, q)
4. Merge-Sort$(A, q+1, r)$
5. Merge(A, p, q, r)

- Since Merge(A, p, q, r) runs in linear time, the runtime $T(n)$ of Merge-Sort(A, p, r) satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Recurrences

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.

MERGE-SORT(A, p, r) *sorting A[p..r]*

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
3. Merge-Sort(A, p, q)
4. Merge-Sort$(A, q+1, r)$
5. Merge(A, p, q, r)

- Since Merge(A, p, q, r) runs in linear time, the runtime $T(n)$ of Merge-Sort(A, p, r) satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = a\,T\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = a\,T\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number;
- Assume that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b;
  - the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is if $f(n)$,
- then the time complexity of such algorithm satisfies

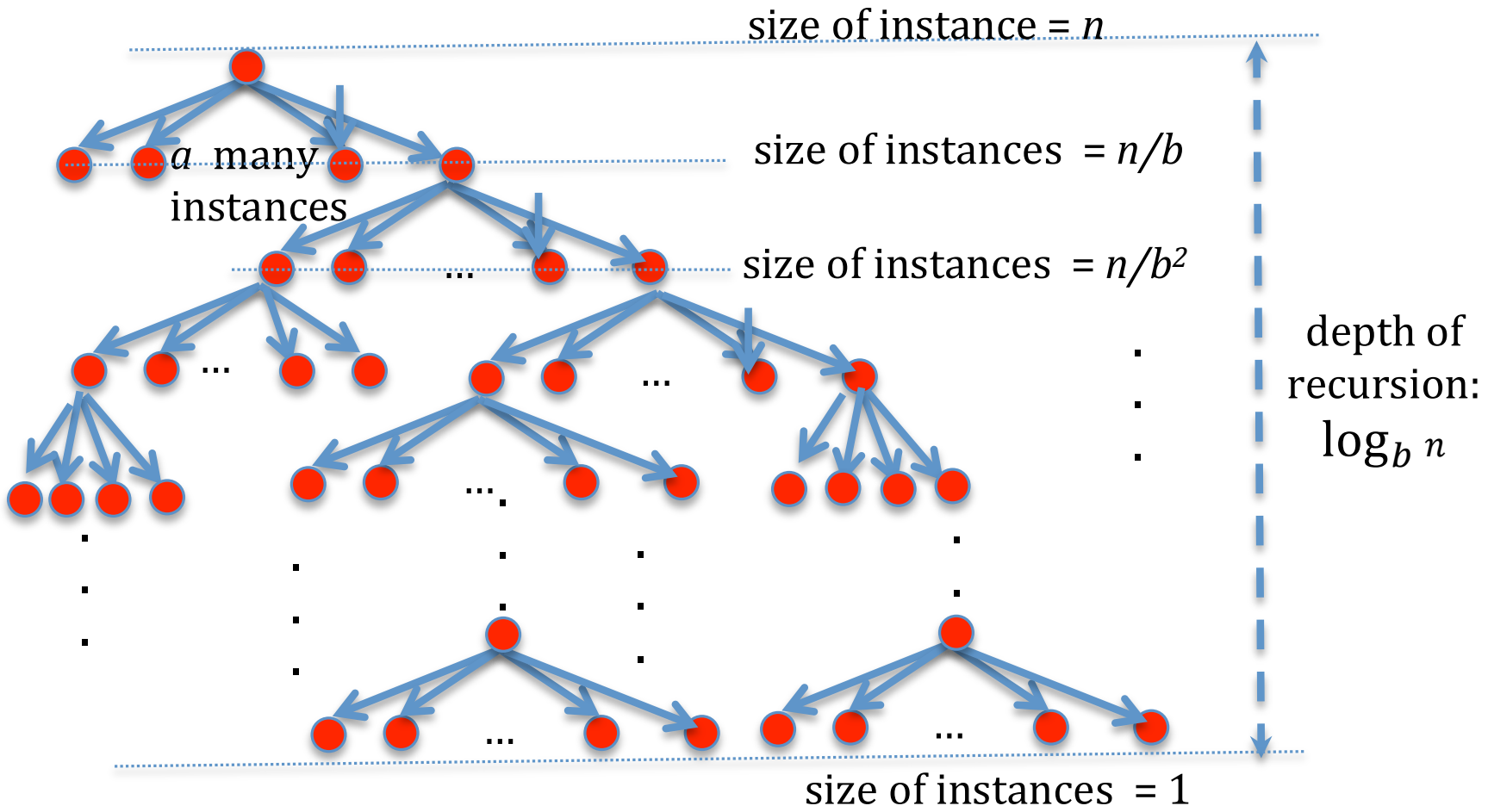
$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- **Note:** we should be writing

$$T(n) = a\,T\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that ignoring the integer parts and additive constants is OK when it comes to obtaining the asymptotics.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$



- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

- Some recurrences can be solved explicitly, but this tends to be tricky.

- Fortunately, to estimate efficiency of an algorithm we **do not** need the exact solution of a recurrence

- We only need to find:
  1. the **growth rate** of the solution i.e., its asymptotic behaviour;
  2. the (approximate) **sizes of the constants** involved (more about that later)

- This is what the **Master Theorem** provides (when it is applicable).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

   (But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

   (But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

   (But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

(But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

(But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

(But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

(But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem:

Let:

- $a \geq 1$ be an integer and and $b > 1$ a real;
- $f(n) > 0$ be a non-decreasing function;
- $T(n)$ be the solution of the recurrence $T(n) = a\,T(n/b) + f(n)$;

Then:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$;
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** for some $c < 1$ and some n_0,

$$a\,f\,(n/b) \leq c\,f(n)$$

   holds for all $n > n_0$, then $T(n) = \Theta(f(n))$;
4. If none of these conditions hold, the Master Theorem is NOT applicable.

   (But often the proof of the Master Theorem can be tweaked to obtain the asymptotic of the solution $T(n)$ in such a case when the Master Theorem does not apply; an example is $T(n) = 2T(n/2) + n \log n$).

Master Theorem - a remark

- Note that for any $b > 1$,

$$\log_b n = \log_b 2 \log_2 n;$$

- Since $b > 1$ is constant (does not depend on n), we have for $c = \log_b 2 > 0$

$$\log_b n = c \log_2 n;$$

$$\log_2 n = \frac{1}{c} \log_b n;$$

- Thus, $\log_b n = \Theta(\log_2 n)$ and also $\log_2 n = \Theta(\log_b n)$.

- So whenever we have $f(n) = \Theta(g(n) \log n)$ we do not have to specify what base the log is - all bases produce equivalent asymptotic estimates (but we do have to specify b in expressions such as $n^{\log_b a}$).

Master Theorem - a remark

- Note that for any $b > 1$,

$$\log_b n = \log_b 2 \log_2 n;$$

- Since $b > 1$ is constant (does not depend on n), we have for $c = \log_b 2 > 0$

$$\log_b n = c \log_2 n;$$

$$\log_2 n = \frac{1}{c} \log_b n;$$

- Thus, $\log_b n = \Theta(\log_2 n)$ and also $\log_2 n = \Theta(\log_b n)$.

- So whenever we have $f(n) = \Theta(g(n) \log n)$ we do not have to specify what base the log is - all bases produce equivalent asymptotic estimates (but we do have to specify b in expressions such as $n^{\log_b a}$).

Master Theorem - a remark

- Note that for any $b > 1$,

$$\log_b n = \log_b 2 \log_2 n;$$

- Since $b > 1$ is constant (does not depend on n), we have for $c = \log_b 2 > 0$

$$\log_b n = c \log_2 n;$$

$$\log_2 n = \frac{1}{c} \log_b n;$$

- Thus, $\log_b n = \Theta(\log_2 n)$ and also $\log_2 n = \Theta(\log_b n)$.

- So whenever we have $f(n) = \Theta(g(n) \log n)$ we do not have to specify what base the log is - all bases produce equivalent asymptotic estimates (but we do have to specify b in expressions such as $n^{\log_b a}$).

Master Theorem - a remark

- Note that for any $b > 1$,

$$\log_b n = \log_b 2 \log_2 n;$$

- Since $b > 1$ is constant (does not depend on n), we have for $c = \log_b 2 > 0$

$$\log_b n = c \log_2 n;$$
$$\log_2 n = \frac{1}{c} \log_b n;$$

- Thus, $\log_b n = \Theta(\log_2 n)$ and also $\log_2 n = \Theta(\log_b n)$.

- So whenever we have $f(n) = \Theta(g(n) \log n)$ we do not have to specify what base the log is - all bases produce equivalent asymptotic estimates (but we do have to specify b in expressions such as $n^{\log_b a}$).

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3 - \varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let $T(n) = 4\,T(n/2) + n$, as in our failed divide and conquer attempt to speed up multiplication.

  then $n^{\log_b a} = n^{\log_2 4} = n^2$;

  thus $f(n) = n = O(n^{2-\varepsilon})$ for any $\varepsilon < 1$.

  Condition of case 1 is satisfied; thus, $T(n) = \Theta(n^2)$.

- Let $T(n) = 3T(n/2) + n$ as in the Karatsuba algorithm

  then $n^{\log_b a} = n^{\log_2 3}$ and $f(n) = n = O(n^{\log_2 3-\varepsilon})$, for any $\varepsilon < \log_2 3 - 1$
  Again, condition of case 1 is satisfied; thus, $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem - Examples

- Let k be a constant and $T(n) = 2T(n/2) + k\,n$, as in the Merge Sort algorithm.

  then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$;

  thus $f(n) = k\,n = \Theta(n) = \Theta(n^{\log_2 2})$.

  Thus, condition of case 2 is satisfied; and so,

  $$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n).$$

Master Theorem - Examples

- Let k be a constant and $T(n) = 2T(n/2) + k\,n$, as in the Merge Sort algorithm.

  then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$;

  thus $f(n) = k\,n = \Theta(n) = \Theta(n^{\log_2 2})$.

  Thus, condition of case 2 is satisfied; and so,

$$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n).$$

Master Theorem - Examples

- Let k be a constant and $T(n) = 2T(n/2) + k\,n$, as in the Merge Sort algorithm.

  then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$;

  thus $f(n) = k\,n = \Theta(n) = \Theta(n^{\log_2 2})$.

  Thus, condition of case 2 is satisfied; and so,

$$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n).$$

Master Theorem - Examples

- Let k be a constant and $T(n) = 2T(n/2) + k\,n$, as in the Merge Sort algorithm.

  then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$;

  thus $f(n) = k\,n = \Theta(n) = \Theta(n^{\log_2 2})$.

  Thus, condition of case 2 is satisfied; and so,

$$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n).$$

Master Theorem - Examples

- Let k be a constant and $T(n) = 2T(n/2) + k\,n$, as in the Merge Sort algorithm.

  then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$;

  thus $f(n) = k\,n = \Theta(n) = \Theta(n^{\log_2 2})$.

  Thus, condition of case 2 is satisfied; and so,

$$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n).$$

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\, n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n \log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n \log_2 n = \Omega(n)$.
  - However, $f(n) = n \log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\, n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n \log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n \log_2 n = \Omega(n)$.
  - However, $f(n) = n \log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Examples

- Let $T(n) = 3\,T(n/4) + n$;
  - then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$;
  - thus $f(n) = n = \Omega(n^{0.8+\varepsilon})$ for any $\varepsilon < 0.2$.
  - Also, $af(n/b) = 3f(n/4) = 3/4\,n < cn = cf(n)$ for $c = .9 < 1$.
  - Thus, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.
- Let $T(n) = 2\,T(n/2) + n\log_2 n$;
  - then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
  - Thus, $f(n) = n\log_2 n = \Omega(n)$.
  - However, $f(n) = n\log_2 n \neq \Omega(n^{1+\varepsilon})$, no matter how small $\varepsilon > 0$.
  - This is because for every $\varepsilon > 0$, and every $c > 0$, no matter how small, $\log_2 n < c \cdot n^{\varepsilon}$ for all sufficiently large n.
  - **Homework:** Prove this.
    Hint: Use *de L'Hôpital's Rule* to show that $\log n/n^{\varepsilon} \to 0$.
  - Thus, in this case the Master Theorem does **not** apply!

Master Theorem - Proof: (optional material)

Since

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \tag{4}$$

implies (by applying it to n/b in place of n)

$$T\left(\frac{n}{b}\right) = a\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right) \tag{5}$$

and (by applying (1) to n/b^2 in place of n)

$$T\left(\frac{n}{b^2}\right) = a\,T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right) \tag{6}$$

and so on . . ., we get

$$\overbrace{T(n) = a\,\underbrace{T\left(\frac{n}{b}\right)}_{(2L)} + f(n)}^{(1)} = a\Big(\underbrace{a\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)}_{(2R)}\Big) + f(n)$$

$$= a^2\,\underbrace{T\left(\frac{n}{b^2}\right)}_{(3L)} + a\,f\left(\tfrac{n}{b}\right) + f(n) = a^2\Big(\underbrace{a\,T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)}_{(3R)}\Big) + a\,f\left(\tfrac{n}{b}\right) + f(n)$$

$$= a^3\,\underbrace{T\left(\frac{n}{b^3}\right)}_{} + a^2 f\left(\tfrac{n}{b^2}\right) + a\,f\left(\tfrac{n}{b}\right) + f(n) = \ldots$$

Master Theorem Proof: (optional material)

Continuing in this way $\log_b n - 1$ many times we get ...

$$T(n) = a^3 \underbrace{T\left(\frac{n}{b^3}\right)} + a^2 f\left(\frac{n}{b^2}\right) + a\, f\left(\frac{n}{b}\right) + f(n) =$$

$$= \ldots$$

$$= a^{\lfloor \log_b n \rfloor} T\left(\frac{n}{b^{\lfloor \log_b n \rfloor}}\right) + a^{\lfloor \log_b n \rfloor - 1} f\left(\frac{n}{b^{\lfloor \log_b n \rfloor - 1}}\right) + \ldots$$
$$+ a^3 f\left(\frac{n}{b^3}\right) + a^2 f\left(\frac{n}{b^2}\right) + a\, f\left(\frac{n}{b}\right) + f(n)$$

$$\approx a^{\log_b n} T\left(\frac{n}{b^{\log_b n}}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

We now use $a^{\log_b n} = n^{\log_b a}$:

$$T(n) \approx n^{\log_b a} T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) \tag{7}$$

Note that so far we did not use any assumptions on $f(n)$ ...

$$T(n) \approx n^{\log_b a} T(1) + \underbrace{\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)}$$

Case 1: $f(m) = O(m^{\log_b a - \varepsilon})$

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i O\left(\frac{n}{b^i}\right)^{\log_b a - \varepsilon}$$

$$= O\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \varepsilon}\right) = O\left(n^{\log_b a - \varepsilon} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a^i}{(b^i)^{\log_b a - \varepsilon}}\right)\right)$$

$$= O\left(n^{\log_b a - \varepsilon} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a - \varepsilon}}\right)^i\right) = O\left(n^{\log_b a - \varepsilon} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a} b^{-\varepsilon}}\right)^i\right)$$

$$= O\left(n^{\log_b a - \varepsilon} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a\, b^{\varepsilon}}{a}\right)^i\right) = O\left(n^{\log_b a - \varepsilon} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} (b^{\varepsilon})^i\right)$$

$$= O\left(n^{\log_b a - \varepsilon} \frac{(b^{\varepsilon})^{\lfloor \log_b n \rfloor} - 1}{b^{\varepsilon} - 1}\right); \qquad \text{we are using } \sum_{i=0}^{m-1} q^i = \frac{q^m - 1}{q - 1}$$

Master Theorem Proof: (optional material)

Case 1 - continued:

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = O\left(n^{\log_b a - \varepsilon} \frac{(b^\varepsilon)^{\lfloor \log_b n \rfloor} - 1}{b^\varepsilon - 1}\right)$$

$$= O\left(n^{\log_b a - \varepsilon} \frac{\left(b^{\lfloor \log_b n \rfloor}\right)^\varepsilon - 1}{b^\varepsilon - 1}\right)$$

$$= O\left(n^{\log_b a - \varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right)$$

$$= O\left(\frac{n^{\log_b a} - n^{\log_b a - \varepsilon}}{b^\varepsilon - 1}\right)$$

$$= O\left(n^{\log_b a}\right)$$

Since we had: $T(n) \approx n^{\log_b a} T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$ we get:

$$T(n) \approx n^{\log_b a} T(1) + O\left(n^{\log_b a}\right)$$

$$= \Theta\left(n^{\log_b a}\right)$$

Master Theorem Proof: (optional material)

Case 2: $f(m) = \Theta(m^{\log_b a})$

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \Theta\left(\frac{n}{b^i}\right)^{\log_b a}$$

$$= \Theta\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}\right)$$

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a^i}{(b^i)^{\log_b a}}\right)\right)$$

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a}}\right)^i\right)$$

$$= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} 1\right)$$

$$= \Theta\left(n^{\log_b a} \lfloor \log_b n \rfloor\right)$$

Master Theorem Proof: (optional material)

Case 2 (continued):

Thus,

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = \Theta\left(n^{\log_b a} \log_b n\right) = \Theta\left(n^{\log_b a} \log_2 n\right)$$

because $\log_b n = \log_2 n \cdot \log_b 2 = \Theta(\log_2 n)$. Since we had (1):

$$T(n) \approx n^{\log_b a} T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

we get:

$$\begin{aligned} T(n) &\approx n^{\log_b a} T(1) + \Theta\left(n^{\log_b a} \log_2 n\right) \\ &= \Theta\left(n^{\log_b a} \log_2 n\right) \end{aligned}$$

Master Theorem Proof: (optional material)

Case 3: $f(m) = \Omega(m^{\log_b a+\varepsilon})$ and $a\,f(n/b) \le c\,f(n)$ for some $0 < c < 1$.

We get by substitution:

$$f(n/b) \le \frac{c}{a} f(n)$$

$$f(n/b^2) \le \frac{c}{a} f(n/b)$$

$$f(n/b^3) \le \frac{c}{a} f(n/b^2)$$

$$\dots$$

$$f(n/b^i) \le \frac{c}{a} f(n/b^{i-1})$$

By chaining these inequalities we get

$$f(n/b^2) \le \frac{c}{a} \underbrace{f(n/b)}_{} \le \frac{c}{a} \cdot \underbrace{\frac{c}{a} f(n)}_{} = \frac{c^2}{a^2} f(n)$$

$$f(n/b^3) \le \frac{c}{a} \underbrace{f(n/b^2)}_{} \le \frac{c}{a} \cdot \underbrace{\frac{c^2}{a^2} f(n)}_{} = \frac{c^3}{a^3} f(n)$$

$$\dots$$

$$f(n/b^i) \le \frac{c}{a} \underbrace{f(n/b^{i-1})}_{} \le \frac{c}{a} \cdot \underbrace{\frac{c^{i-1}}{a^{i-1}} f(n)}_{} = \frac{c^i}{a^i} f(n)$$

Master Theorem Proof: (optional material)

Case 3 (continued):

We got $$f(n/b^i) \leq \frac{c^i}{a^i} f(n)$$

Thus,

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) \leq \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \frac{c^i}{a^i} f(n) < f(n) \sum_{i=0}^{\infty} c^i = \frac{f(n)}{1-c}$$

Since we had (1):

$$T(n) \approx n^{\log_b a} T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

and since $f(n) = \Omega(n^{\log_b a + \varepsilon})$ we get:

$$T(n) < n^{\log_b a} T(1) + O\left(f(n)\right) = O\left(f(n)\right)$$

but we also have

$$T(n) = aT(n/b) + f(n) > f(n)$$

thus,

$$T(n) = \Theta\left(f(n)\right)$$

Master Theorem Proof: (optional material)

Exercise 1: Show that condition

$$f(n) = \Omega(n^{\log_b a+\varepsilon})$$

follows from the condition

$$a\, f(n/b) \leq c\, f(n) \quad \text{for some } 0 < c < 1.$$

Example: Let us estimate the asymptotic growth rate of $T(n)$ which satisfies

$$T(n) = 2\, T(n/2) + n \log n$$

Note: we have seen that the Master Theorem does **NOT** apply, but the technique used in its proof still works! So let us just unwind the recurrence and sum up the logarithmic overheads.

$$
\begin{aligned}
T(n) &= 2\underbrace{T\left(\frac{n}{2}\right)} + n\log n \\
&= 2\left(\overbrace{2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\log\frac{n}{2}}\right) + n\log n \\
&= 2^2\underbrace{T\left(\frac{n}{2^2}\right)} + n\log\tfrac{n}{2} + n\log n \\
&= 2^2\left(\overbrace{2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\log\frac{n}{2^2}}\right) + n\log\tfrac{n}{2} + n\log n \\
&= 2^3\underbrace{T\left(\frac{n}{2^3}\right)} + n\log\tfrac{n}{2^2} + n\log\tfrac{n}{2} + n\log n \\
&\dots \\
&= 2^{\log n}T\left(\tfrac{n}{2^{\log n}}\right) + n\log\tfrac{n}{2^{\log n-1}} + \ldots + n\log\tfrac{n}{2^2} + n\log\tfrac{n}{2} + n\log n \\
&= nT(1) + n(\log n \times \log n - \log 2^{\log n-1} - \ldots - \log 2^2 - \log 2 \\
&= nT(1) + n((\log n)^2 - (\log n - 1) - \ldots - 3 - 2 - 1) \\
&= nT(1) + n((\log n)^2 - \log n(\log n - 1)/2 \\
&= nT(1) + n((\log n)^2/2 + \log n/2) \\
&= \Theta\left(n(\log n)^2\right).
\end{aligned}
$$

Deterministic Linear Time Algorithm for Order Statistic

- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

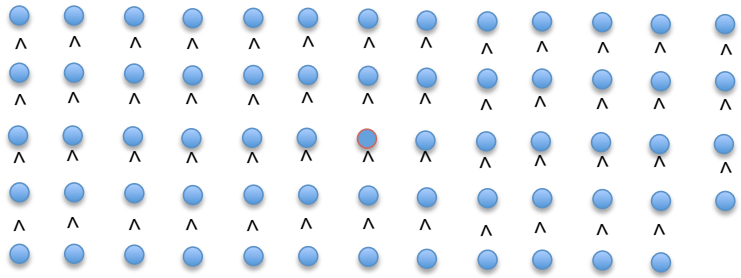
- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

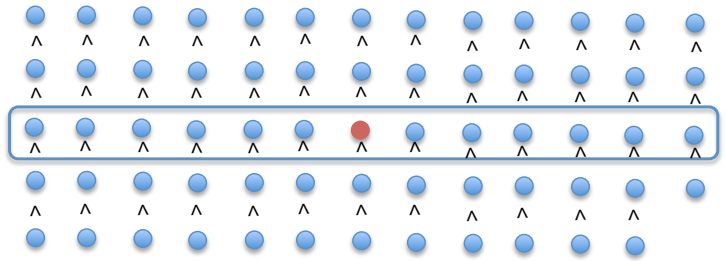
- Order statistic is a generalisation of a median.
- Median of a set S of n numbers is an element $x \in S$ such that the number of elements of S smaller than x and the number of elements larger than x is either equal (if the number of elements in S is odd) or differ by one (if the number of elements in S is even)
- So if the set S was in a sorted array, the median would be in the middle of the array.
- Order statistic is the following generalisation: *given a set S of n numbers and an integer k such that $1 \leq k \leq n$ find the k^{th} element in size.*
- Clearly, this problem can be solved in time $n \log n$ by sorting S and picking the k^{th} element.
- However, the problem can be solved in linear time by an algorithm from 1973 by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ron Rivest, and Robert Tarjan, which we now present.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) :
  - Split the numbers in groups of five (the last group might contain less than 5 elements);
  - Order each group by brute force in an increasing order.



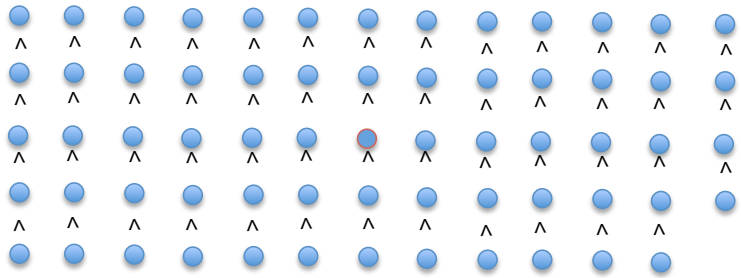
  - Take the collection of all $\lfloor \frac{n}{5} \rfloor$ middle elements of each group (i.e., the medians of each group of five).



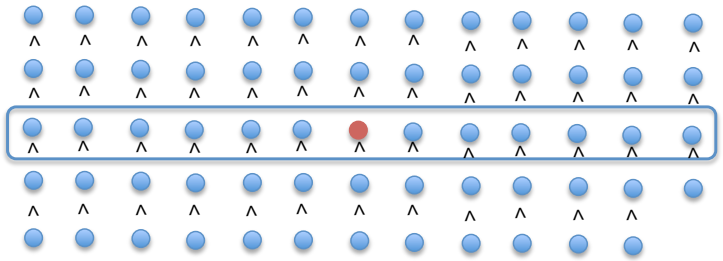
  - Apply recursively SELECT algorithm to find the median p of this collection.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) :
  - Split the numbers in groups of five (the last group might contain less than 5 elements);
  - Order each group by brute force in an increasing order.



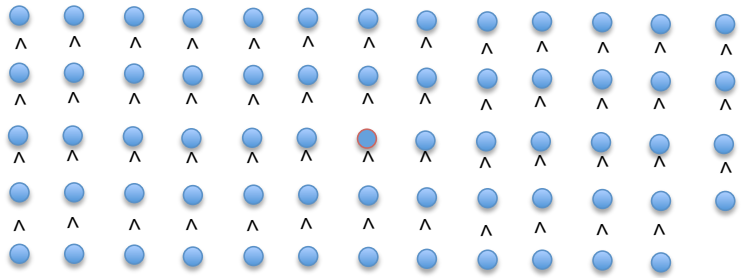
  - Take the collection of all $\lfloor \frac{n}{5} \rfloor$ middle elements of each group (i.e., the medians of each group of five).



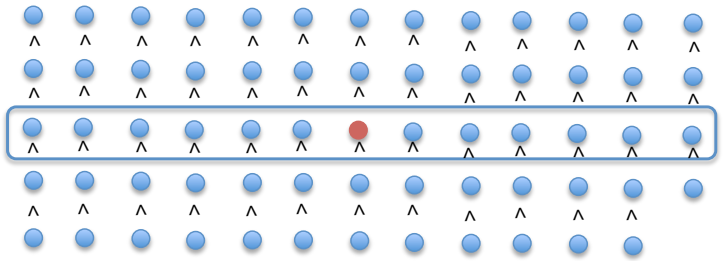
  - Apply recursively SELECT algorithm to find the median p of this collection.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) :
  - Split the numbers in groups of five (the last group might contain less than 5 elements);
  - Order each group by brute force in an increasing order.



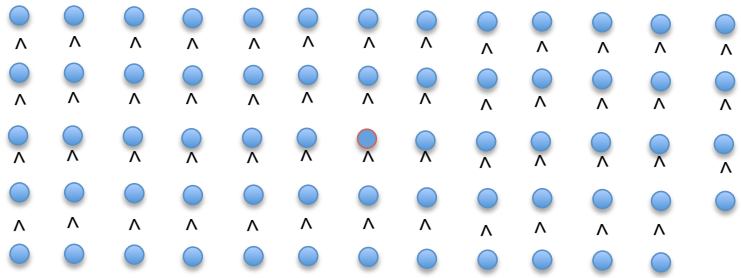
  - Take the collection of all $\lfloor \frac{n}{5} \rfloor$ middle elements of each group (i.e., the medians of each group of five).



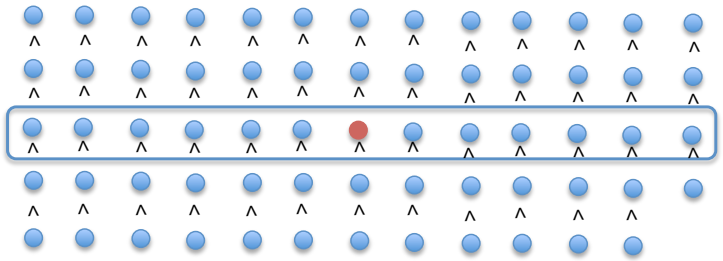
  - Apply recursively SELECT algorithm to find the median p of this collection.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) :
  - Split the numbers in groups of five (the last group might contain less than 5 elements);
  - Order each group by brute force in an increasing order.



  - Take the collection of all $\lfloor \frac{n}{5} \rfloor$ middle elements of each group (i.e., the medians of each group of five).



  - Apply recursively SELECT algorithm to find the median p of this collection.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select(n, i) continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

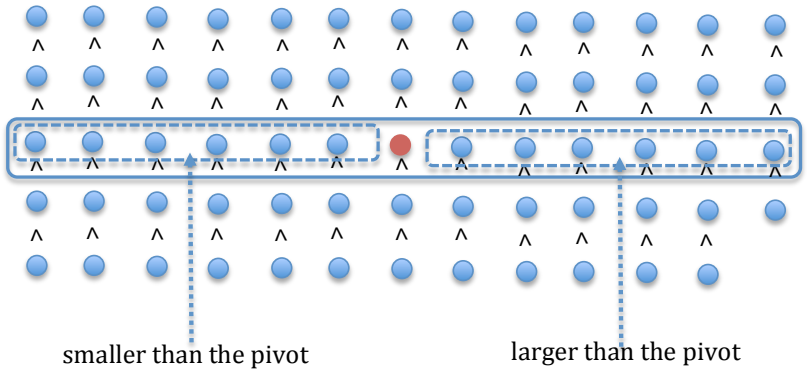
- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

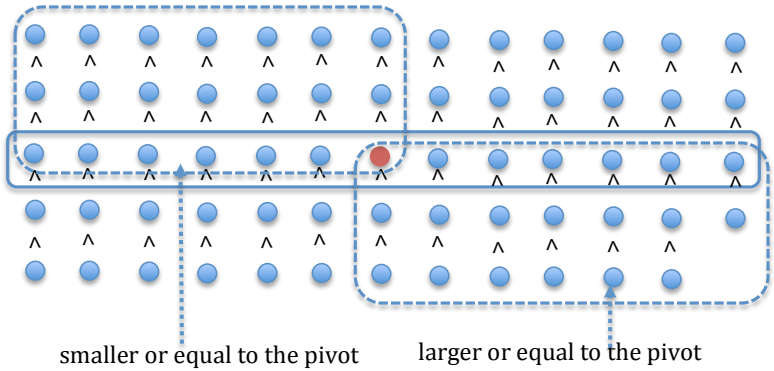
Deterministic Linear Time Algorithm for Order Statistic

- **Algorithm Select**(n, i) **continued:**
  - partition all elements using p as a pivot;
  - Let k be the number of elements in the subset of all elements smaller than the pivot p.
  - **if** $i = k$ **then return** p
  - **else if** $i < k$ **then** recursively SELECT the i^{th} smallest element of the set of elements smaller than the pivot.
  - **else** recursively SELECT the $(i - k)^{th}$ smallest element of the set of elements larger than the pivot.

Deterministic Linear Time Algorithm for Order Statistic

- What have we accomplished by such a choice of the pivot?

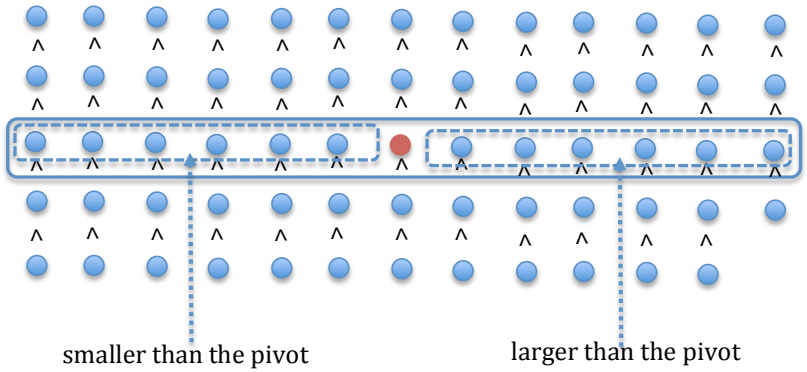


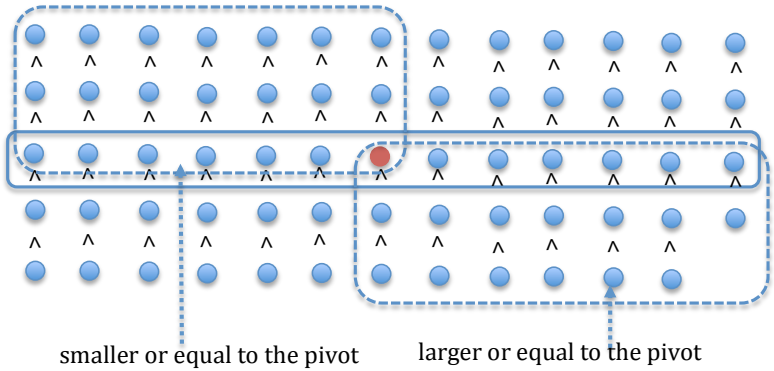


- Note that at least $\lfloor (n/5)/2 \rfloor = \lfloor n/10 \rfloor$ group medians are smaller or equal to the pivot; and at least that many larger than the pivot.
- But this implies that at least $\lfloor 3n/10 \rfloor$ of the total number of elements are smaller than the pivot, and that many elements larger than the pivot.
- (caveat: we are assuming all elements are distinct; otherwise we have to slightly tweak the algorithm to split all elements equal to the pivot evenly between the two groups.)

Deterministic Linear Time Algorithm for Order Statistic

- What have we accomplished by such a choice of the pivot?

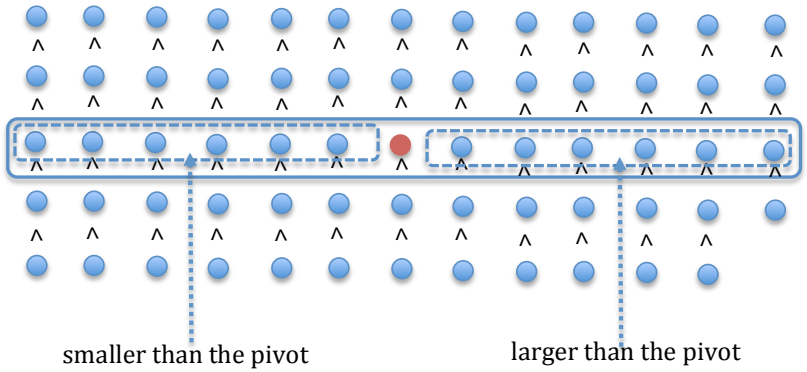


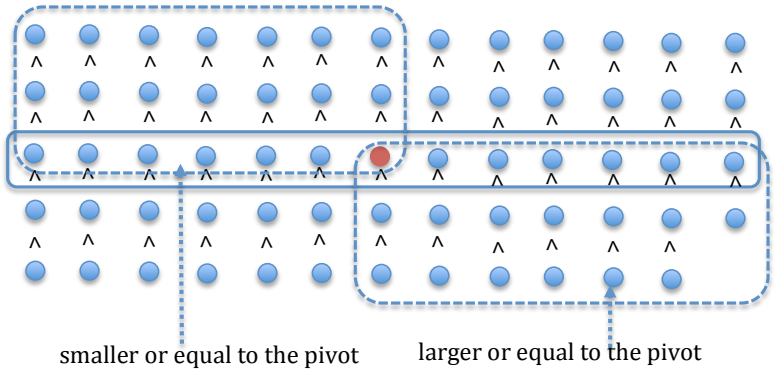


- Note that at least $\lfloor (n/5)/2 \rfloor = \lfloor n/10 \rfloor$ group medians are smaller or equal to the pivot; and at least that many larger than the pivot.
- But this implies that at least $\lfloor 3n/10 \rfloor$ of the total number of elements are smaller than the pivot, and that many elements larger than the pivot.
- (caveat: we are assuming all elements are distinct; otherwise we have to slightly tweak the algorithm to split all elements equal to the pivot evenly between the two groups.)

Deterministic Linear Time Algorithm for Order Statistic

- What have we accomplished by such a choice of the pivot?

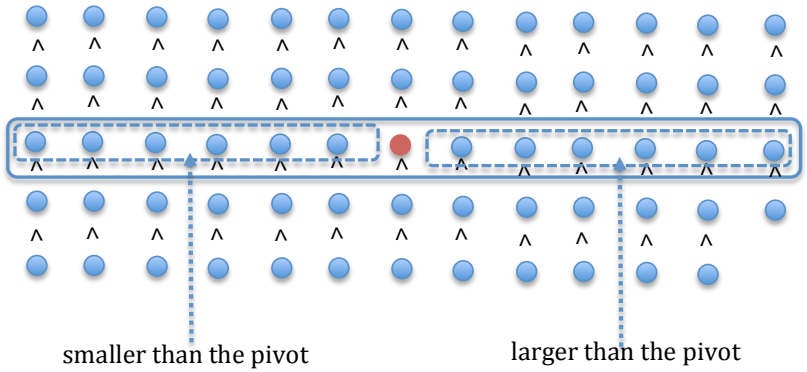


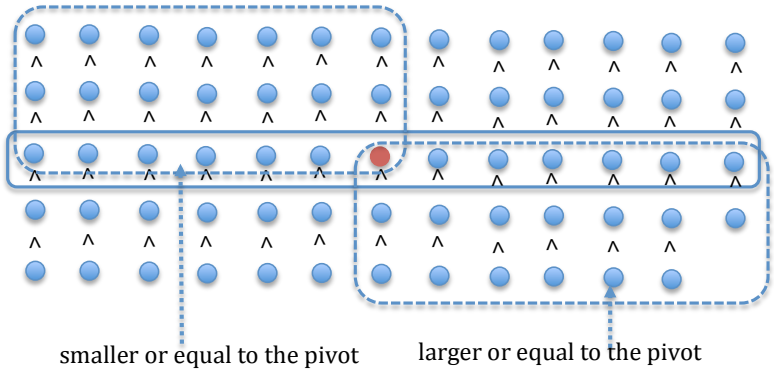


- Note that at least $\lfloor (n/5)/2 \rfloor = \lfloor n/10 \rfloor$ group medians are smaller or equal to the pivot; and at least that many larger than the pivot.
- But this implies that at least $\lfloor 3n/10 \rfloor$ of the total number of elements are smaller than the pivot, and that many elements larger than the pivot.
- (caveat: we are assuming all elements are distinct; otherwise we have to slightly tweak the algorithm to split all elements equal to the pivot evenly between the two groups.)

Deterministic Linear Time Algorithm for Order Statistic

- What have we accomplished by such a choice of the pivot?

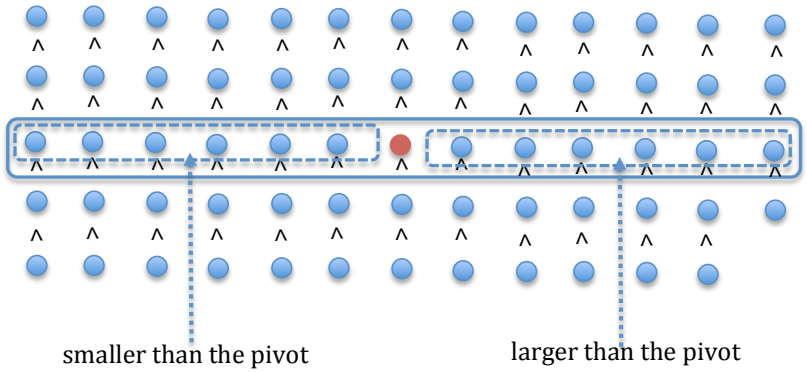


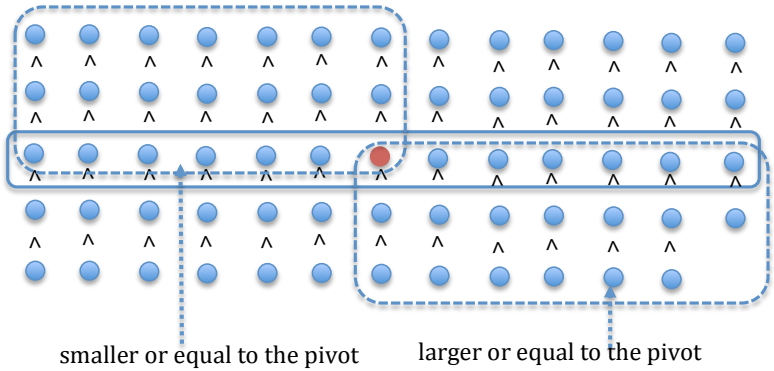


- Note that at least $\lfloor (n/5)/2 \rfloor = \lfloor n/10 \rfloor$ group medians are smaller or equal to the pivot; and at least that many larger than the pivot.
- But this implies that at least $\lfloor 3n/10 \rfloor$ of the total number of elements are smaller than the pivot, and that many elements larger than the pivot.
- (caveat: we are assuming all elements are distinct; otherwise we have to slightly tweak the algorithm to split all elements equal to the pivot evenly between the two groups.)

Deterministic Linear Time Algorithm for Order Statistic

- What have we accomplished by such a choice of the pivot?





- Note that at least $\lfloor (n/5)/2 \rfloor = \lfloor n/10 \rfloor$ group medians are smaller or equal to the pivot; and at least that many larger than the pivot.
- But this implies that at least $\lfloor 3n/10 \rfloor$ of the total number of elements are smaller than the pivot, and that many elements larger than the pivot.
- (caveat: we are assuming all elements are distinct; otherwise we have to slightly tweak the algorithm to split all elements equal to the pivot evenly between the two groups.)

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$

$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$
$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

Deterministic Linear Time Algorithm for Order Statistic

- What is the run time of our algorithm?

$$T(n) \leq T(n/5) + T(7n/10) + Cn.$$

- Let us show that $T(n) < 11Cn$ for all n. Assume that this is true for all $k < n$ and let us prove it is true for n as well.
  - Note: this is a proof using the following form of induction: $\varphi(0)\ \&\ (\forall n)((\forall k < n)\varphi(k) \to \varphi(n)) \to (\forall n)\varphi(n)$.
- Thus, assume $T(n/5) < 11C \cdot n/5$ and $T(7n/10) < 11C \cdot 7n/10$; then

$$T(n) \leq T(n/5) + T(7n/10) + Cn < 11C \cdot \frac{n}{5} + 11C \cdot \frac{7n}{10} + Cn$$
$$= 109\frac{Cn}{10} < 11C \cdot n$$

which proves out statement that $T(n) < 11C \cdot n$.

PUZZLE!

On a circular highway there are n petrol stations, unevenly spaced, each containing a different quantity of petrol. It is known that the total quantity of petrol on all stations is enough to go around the highway once, and that the tank of your car can hold enough fuel to make a trip around the highway. Prove that there always exists a station among all of the stations on the highway, such that if you take it as a starting point and take the fuel from that station, you can continue to make a complete round trip around the highway, never emptying your tank before reaching the next station to refuel.

Hint: Try proving it by induction. Find a way for reducing the case with $n + 1$ petrol stations to the case with only n petrol stations by choosing suitably a petrol station to remove, pouring its petrol to the preceding petrol station.

That's All, Folks!!