Announcements

- No homework this week
- Exam 1 on Friday

Exam 1

- In class on Friday
- Random assigned seating
- 3Qs in 45 minutes
- 6 one-sided pages of notes
- No textbook or electronic aids
- On Chapters 3 & 4
- Review video online

What constitutes a proof for this class?

- Needs to be a proof:
  - Establish result from known premises
  - Probably consists of English sentences
  - Might be quite short depending on problem
- How much detail to give?
  - Hard to spell out exactly
  - Rule of thumb: enough to convince the grader that you *could* fill in further details if pressed

Last Time

- Divide and Conquer
- Karatsuba Multiplication
- Master Theorem

Divide and Conquer

This is the first of our three major algorithmic techniques.

1. Break problem into pieces
2. Solve pieces recursively
3. Recombine pieces to get answer

Karatsuba Multiplication

```
KaratsubaMult(N,M)
  If N+M<99, Return Product(N,M)
  Let X be a power of 2⌊log(N+M)/2⌋
  Write N = AX + B, M = CX + D
  P1 ← KaratsubaMult(A,C)
  P2 ← KaratsubaMult(B,D)
  P3 ← KaratsubaMult(A+B,C+D)
  Return P1X^2 + [P3-P1-P2]X + P2
```

Runtime Recurrence

Karatsuba multiplication on inputs of size n spends O(n) time, and then makes three recursive calls to problems of (approximately) half the size.

If T(n) is the runtime for n-bit inputs, we have the recursion:

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ 3T(n/2 + O(1)) + O(n) & \text{otherwise} \end{cases}$$

Generalization

We will often get runtime recurrences with D&C looking something like this:

$T(n) = O(1)$ for $n = O(1)$

$T(n) = a\,T(n/b + O(1)) + O(n^d)$ otherwise.

Master Theorem

Theorem: Let T(n) be given by the recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ aT(n/b + O(1)) + O(n^d) & \text{otherwise} \end{cases}$$

Then we have that

$$T(n) = \begin{cases} O(n^{\log_b(a)}) & \text{if } a > b^d \\ O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \end{cases}$$

Karatsuba Runtime

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ 3T(n/2 + O(1)) + O(n) & \text{otherwise} \end{cases}$$

- a = 3, b = 2, d = 1
- a > b^d

Runtime: $O(n^{\log_2(3)}) = O(n^{1.585\ldots})$.

Better than easy $O(n^2)$ algorithm!

Question: Runtimes

Suppose that a divide and conquer algorithm needs to solve 4 recursive subproblems of half the size and do $O(n^2)$ additional work. What is the runtime?

A) $O(n)$
B) $O(n \log(n))$
C) $O(n^2)$
D) $O(n^2 \log(n))$
E) $O(n^3)$

Question: Runtimes

Suppose that a divide and conquer algorithm needs to solve 4 recursive subproblems of half the size and do $O(n^2)$ additional work. What is the runtime?

A) $O(n)$
B) $O(n \log(n))$
C) $O(n^2)$
D) $O(n^2 \log(n))$
E) $O(n^3)$

$a = 4$, $b = 2$, $d = 2$.
$a = b^d$.
So runtime is $O(n^d \log(n))$

Today

- Strassen’s algorithm
- Mergesort
- Order statistics

Note

In divide and conquer, it is important that the recursive subcalls are a constant *fraction* of the size of the original.

Note

In divide and conquer, it is important that the recursive subcalls are a constant *fraction* of the size of the original.

For example, if we have

$$T(n) = 2T(n-1)$$

then $T(n) = O(2^n)$.

Matrix Multiplication

Problem: Multiply two nxn matrices.

Matrix Multiplication

Problem: Multiply two nxn matrices.

Recall: If AB=C then

$$C_{ij} = \Sigma_k A_{ik}B_{kj}$$

Matrix Multiplication

Problem: Multiply two nxn matrices.

Recall: If AB=C then

$$C_{ij} = \Sigma_k A_{ik}B_{kj}$$

Naïve algorithm computes this sum of n terms, for each of n^2 entries. Runtime = $O(n^3)$.

Matrix Multiplication

Problem: Multiply two nxn matrices.

Recall: If AB=C then

$$C_{ij} = \Sigma_k A_{ik}B_{kj}$$

Naïve algorithm computes this sum of n terms, for each of n^2 entries. Runtime = $O(n^3)$.

Can we do better?

Block Matrix Multiplication

If you divide the matrix into blocks, you can get the product of the full matrix in terms of products of the blocks.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Here, A,B,C,… are (n/2)x(n/2) matrices.

Easy D&C

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Compute 8 products of (n/2)x(n/2) matrices, and do some addition to get answer.

Easy D&C

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Compute 8 products of (n/2)x(n/2) matrices, and do some addition to get answer.

$T(n) = 8T(n/2)+O(n^2)$.

Easy D&C

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}.$$

Compute 8 products of (n/2)x(n/2) matrices, and do some addition to get answer.

$T(n) = 8T(n/2)+O(n^2)$.

Master Theorem: a = 8, b = 2, d = 2.
$a > b^d$ so $O(n^C)$ with $C = \log_2 8 = 3$. $O(n^3)$.

Strassen's Algorithm

Instead compute:

- $M_1=(A+D)(E+H)$
- $M_2=(C+D)E$
- $M_3=A(F-H)$
- $M_4=D(G-E)$
- $M_5=(A+B)H$
- $M_6=(C-A)(E+F)$
- $M_7=(B-D)(G+H)$

Entries of product matrix are:

- $M_1+M_4-M_5+M_7$
- M_3+M_5
- M_2+M_4
- $M_1-M_2+M_3+M_6$

How do you figure this out? Don't ask me. Magic, probably.

Strassen's Algorithm Analysis

You need to compute 7 recursive calls of half the size plus $O(n^2)$ extra work.

Strassen's Algorithm Analysis

You need to compute 7 recursive calls of half the size plus $O(n^2)$ extra work.

Master Theorem: $a = 7$, $b = 2$, $d = 2$.

$a > b^d$. $O(n^C)$ with $C = \log_2 7 \approx 2.807\ldots$

Strassen's Algorithm Analysis

You need to compute 7 recursive calls of half the size plus $O(n^2)$ extra work.

Master Theorem: $a = 7$, $b = 2$, $d = 2$.

$a > b^d$. $O(n^C)$ with $C = \log_2 7 \approx 2.807\ldots$

Improvement for very large matrices.

Strassen's Algorithm Analysis

You need to compute 7 recursive calls of half the size plus $O(n^2)$ extra work.

Master Theorem: $a = 7$, $b = 2$, $d = 2$.

$a > b^d$. $O(n^C)$ with $C = \log_2 7 \approx 2.807\ldots$

Improvement for very large matrices.

Best known algorithm: $O(n^{2.376\ldots})$ very impractical.

Note on Proving Correctness

There's a general procedure for proving correctness of a D&C algorithm:

Note on Proving Correctness

There’s a general procedure for proving correctness of a D&C algorithm:

<u>**Use Induction:**</u> Prove correctness by induction on problem size.

Note on Proving Correctness

There's a general procedure for proving correctness of a D&C algorithm:

Use Induction: Prove correctness by induction on problem size.

Base Case: Your base case will be the non-recursive case of your algorithm (which your algorithm does need to have).

Note on Proving Correctness

There's a general procedure for proving correctness of a D&C algorithm:

Use Induction: Prove correctness by induction on problem size.

Base Case: Your base case will be the non-recursive case of your algorithm (which your algorithm does need to have).

Inductive Step: Assuming that the (smaller) recursive calls are correct, show that algorithm works.

Sorting

Problem: Given a list of n numbers, return those numbers in ascending order.

Sorting

Problem: Given a list of n numbers, return those numbers in ascending order.

Example:

Input: {0, 5, 2, 7, 4, 6, 3, 1}

Output: {0, 1, 2, 3, 4, 5, 6, 7}

Divide and Conquer

How do we make a divide and conquer algorithm?

Divide and Conquer

How do we make a divide and conquer algorithm?

1. Split into Subproblems
2. Recursively Solve
3. Combine Answers

Divide and Conquer

How do we make a divide and conquer algorithm?

1. Split into Subproblems - Split list in two
2. Recursively Solve
3. Combine Answers

Divide and Conquer

How do we make a divide and conquer algorithm?

1. Split into Subproblems
2. Recursively Solve
3. Combine Answers

- Split list in two
- Sort each sublist

Divide and Conquer

How do we make a divide and conquer algorithm?

1. Split into Subproblems
2. Recursively Solve
3. Combine Answers

- Split list in two
- Sort each sublist
- ???

Merge

Problem: Given two sorted lists, combine them into a single sorted list.

Merge

Problem: Given two sorted lists, combine them into a single sorted list.

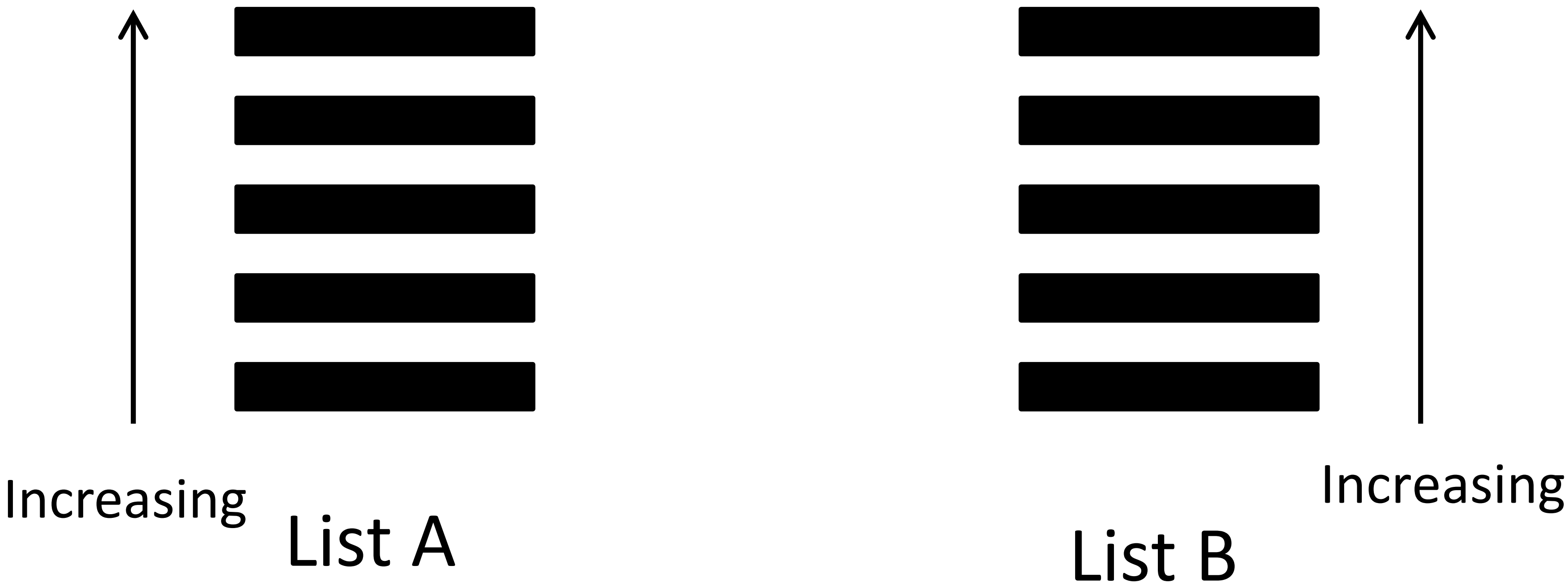
We want something that takes advantage of the individual lists being sorted.

Merge Idea

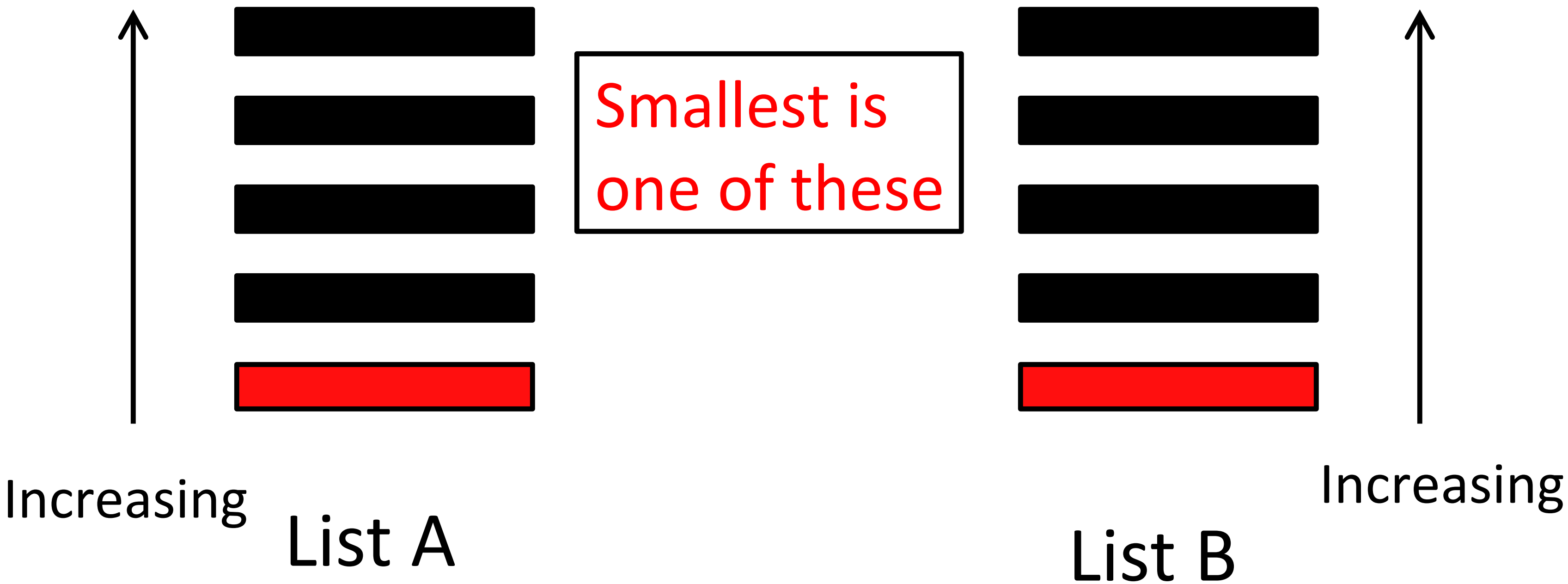
List A

List B

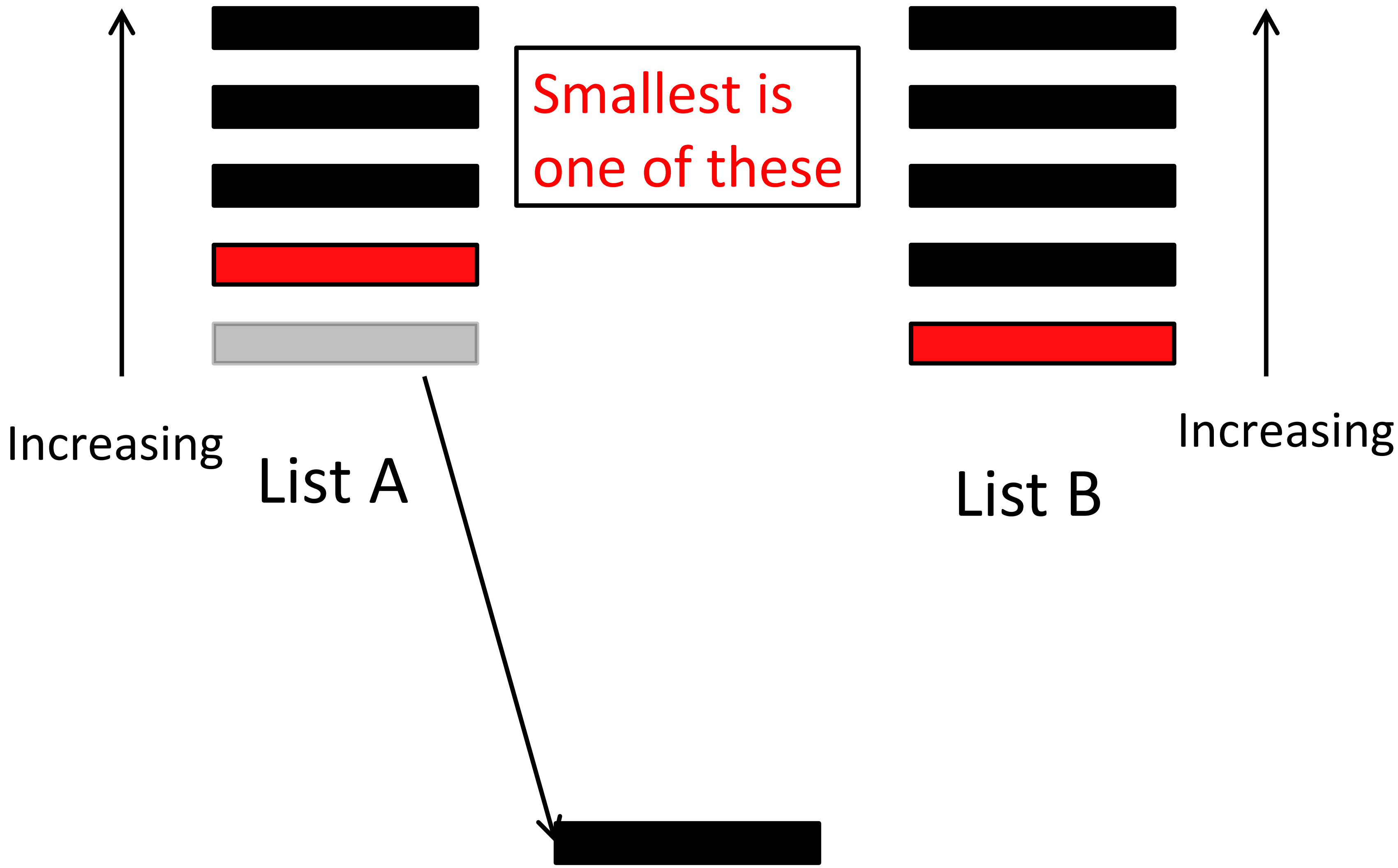
Merge Idea



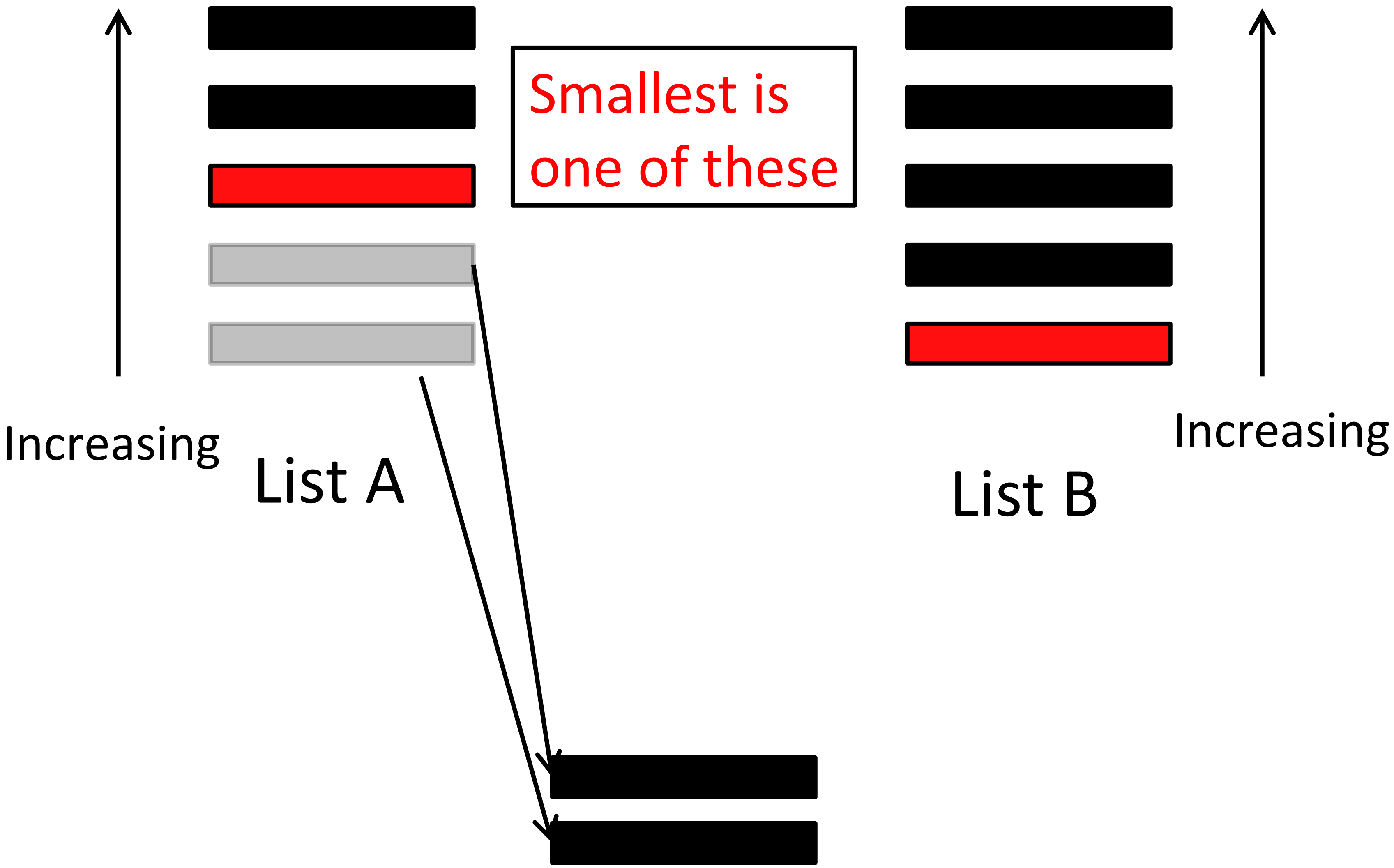
Merge Idea


Smallest is one of these
Increasing
List A
Increasing
List B

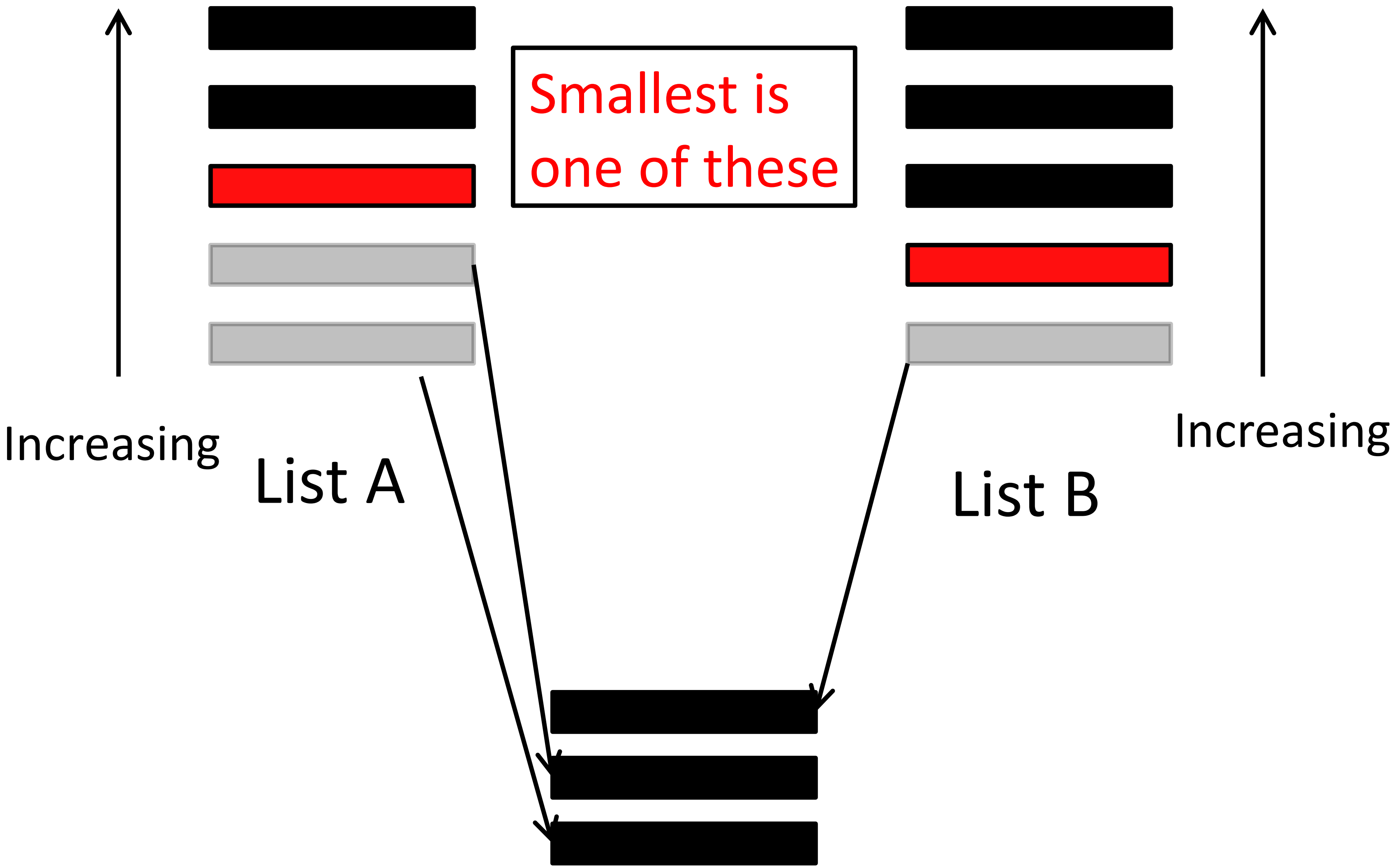
Merge Idea


Smallest is
one of these
Increasing
List A
Increasing
List B

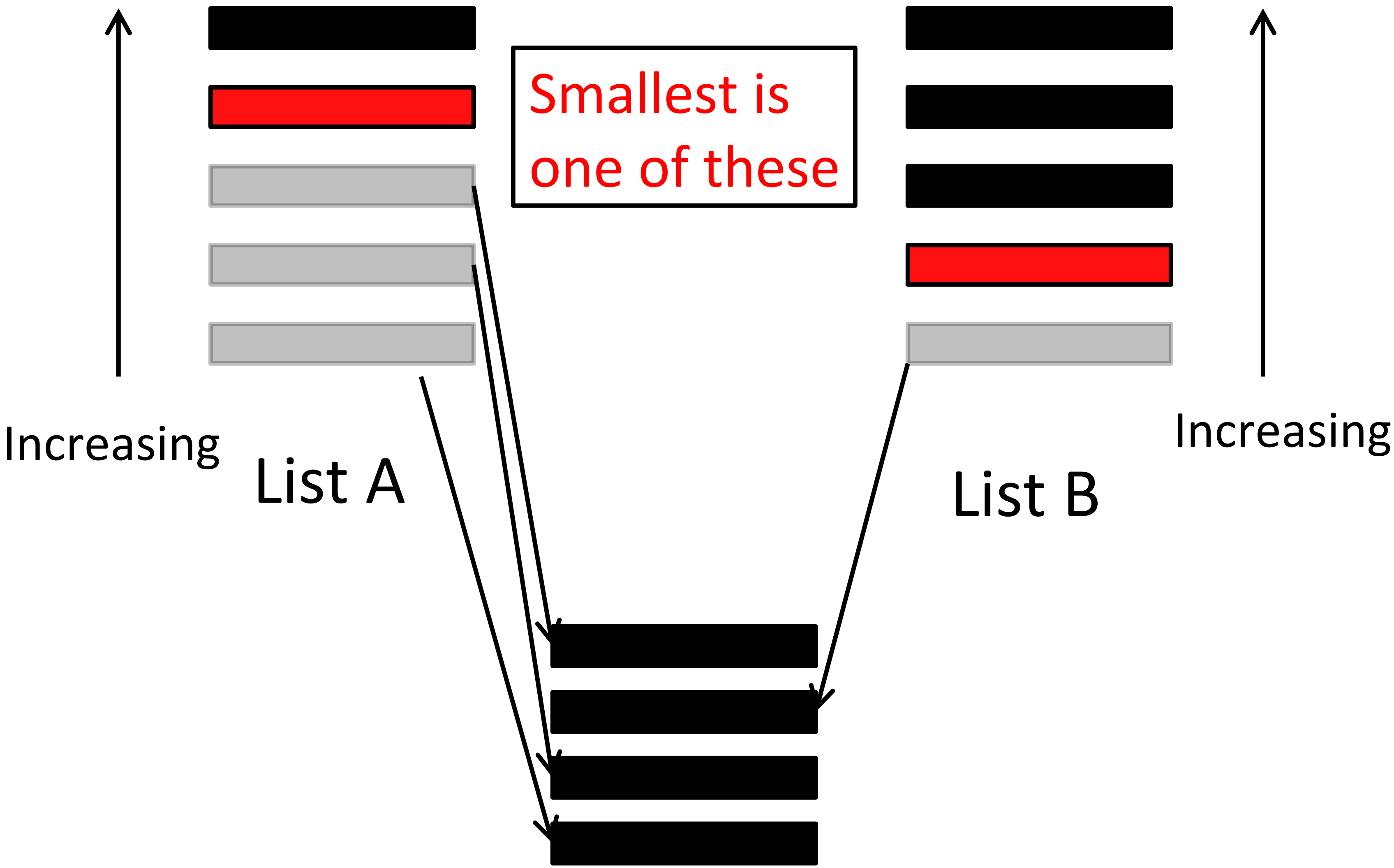
Merge Idea


Smallest is
one of these
Increasing
List A
Increasing
List B

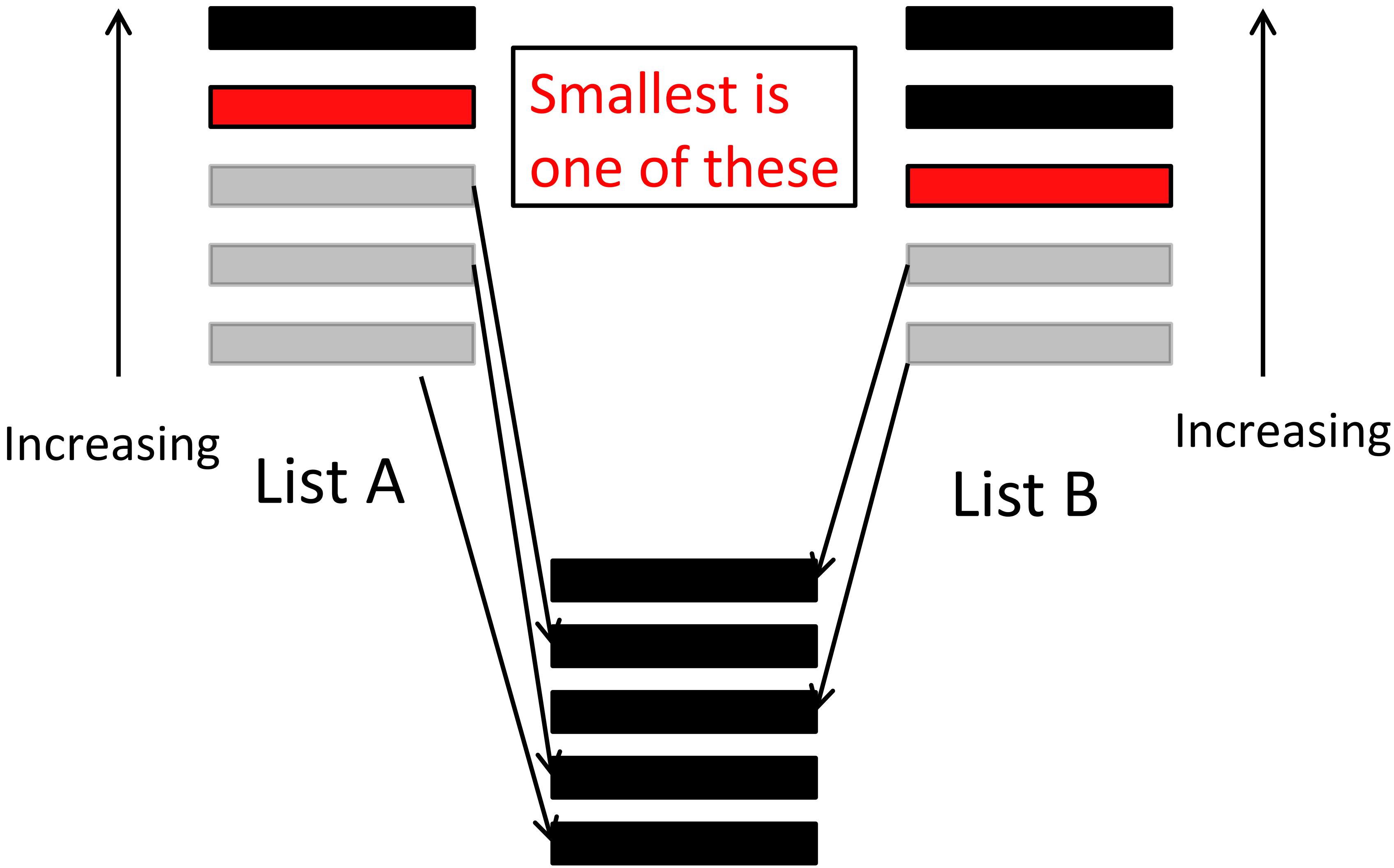
Merge Idea


Smallest is one of these
Increasing
List A
Increasing
List B

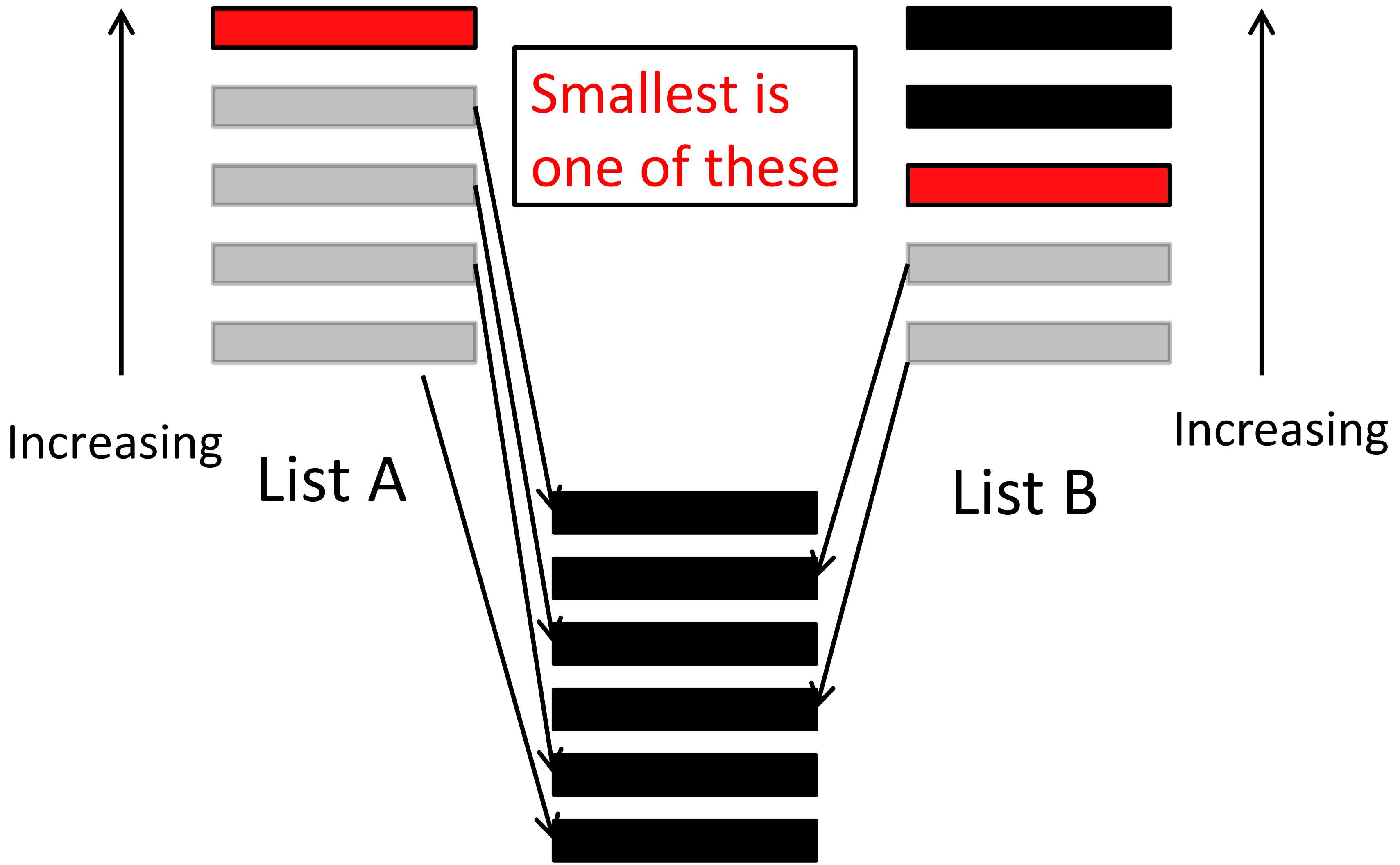
Merge Idea


Smallest is one of these
Increasing
List A
Increasing
List B

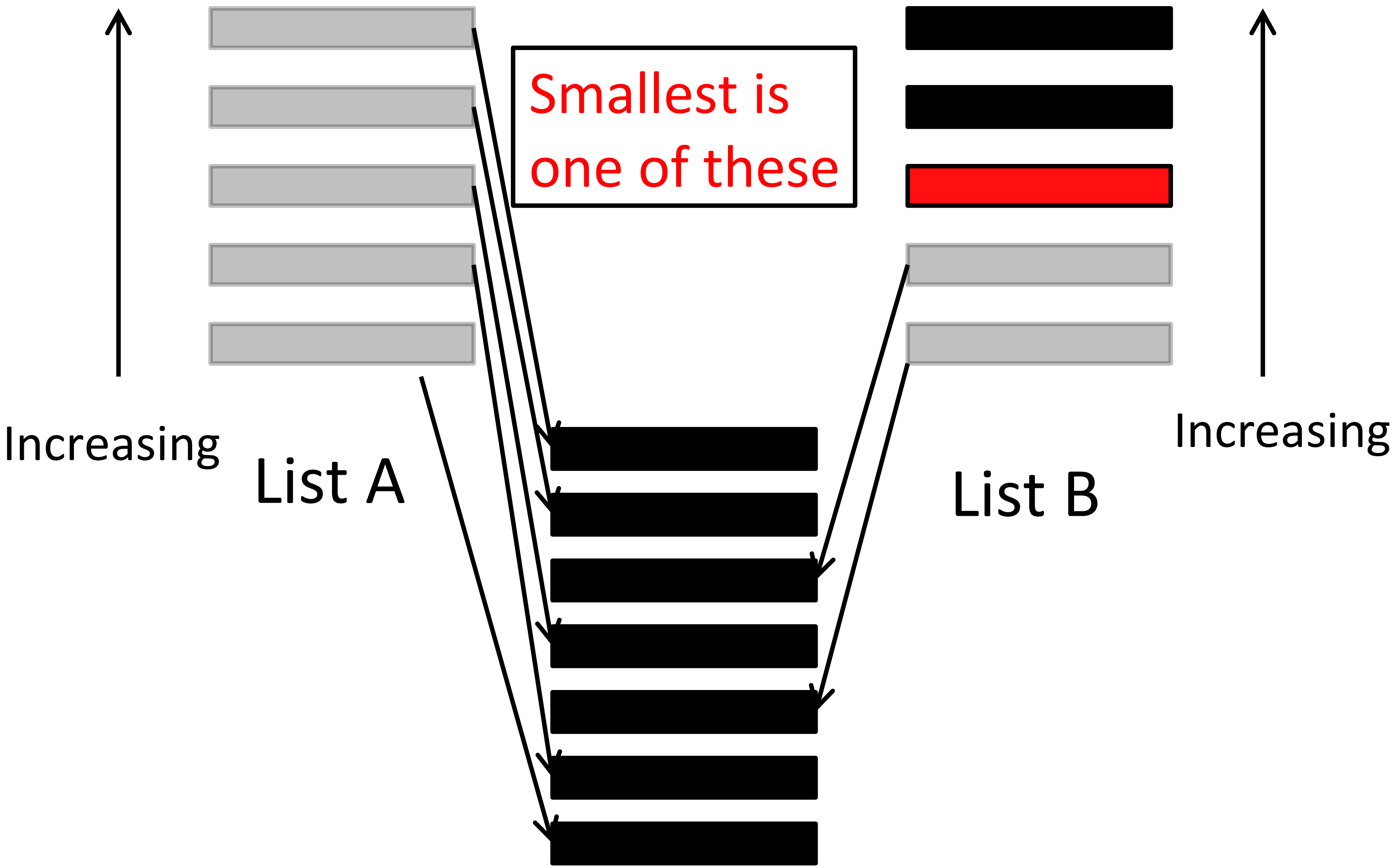
Merge Idea


Smallest is
one of these
Increasing
List A
Increasing
List B

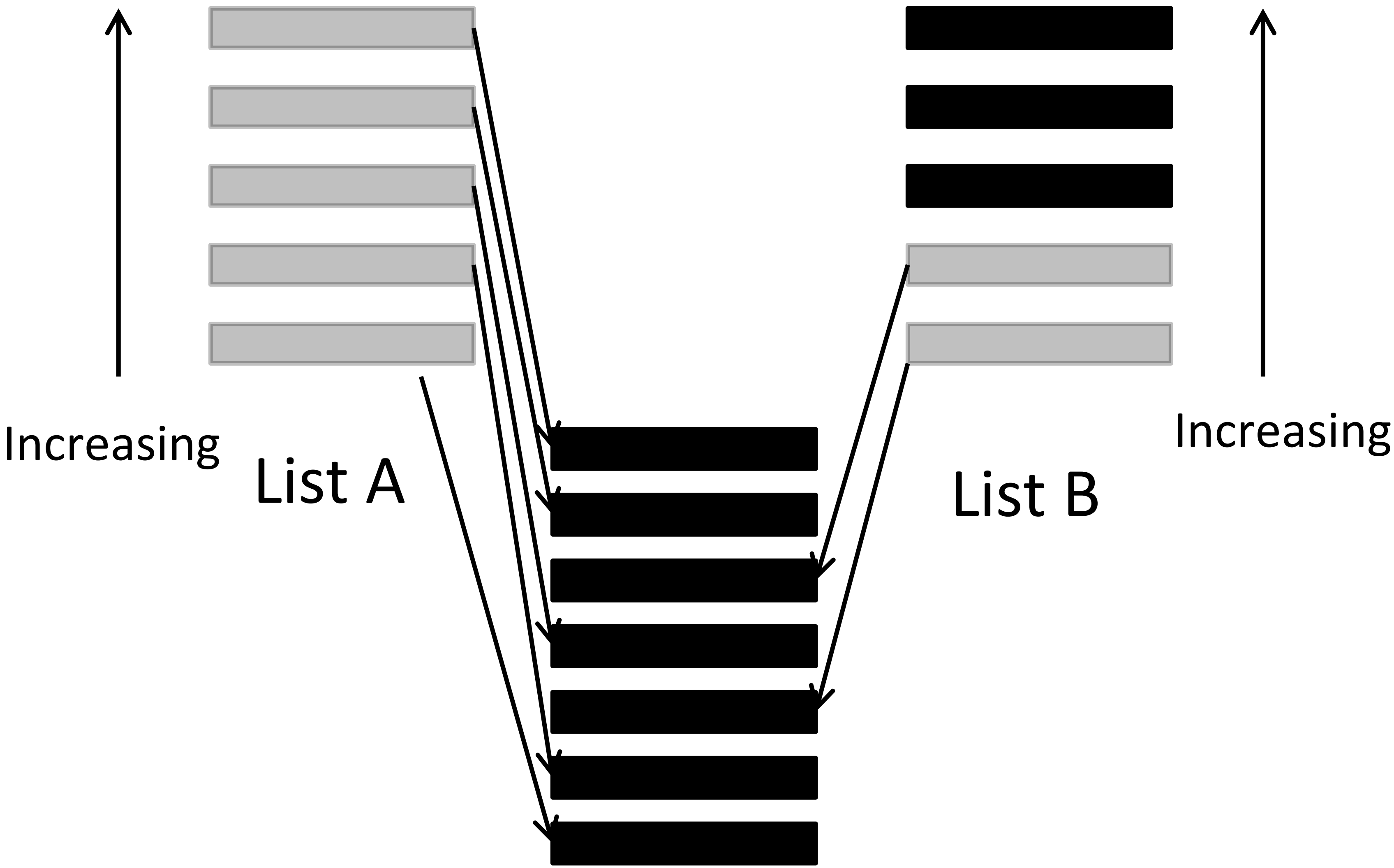
Merge Idea


Smallest is
one of these
Increasing
List A
Increasing
List B

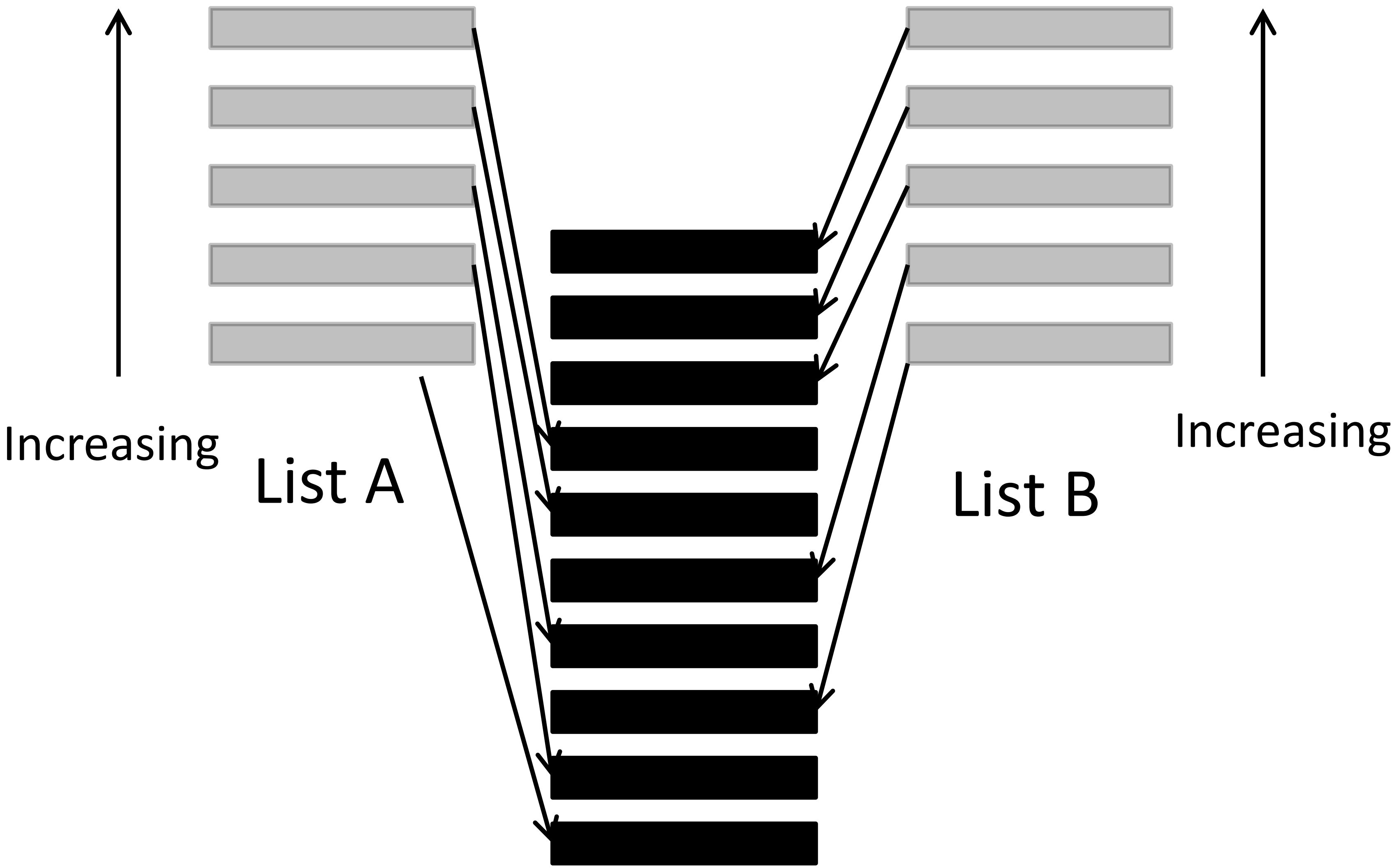
Merge Idea


Smallest is one of these
Increasing
List A
Increasing
List B

Merge Idea



Merge Idea



Merge

```
Merge(A,B)
  C ← List of length Len(A)+Len(B)
  a ← 1, b ← 1
  For c = 1 to Len(C)
    If (b > Len(B))
      C[c] ← A[a], a++
    Else if (a > Len(A))
      C[c] ← B[b], b++
    Else if A[a] < B[b]
      C[c] ← A[a], a++
    Else
      C[c] ← B[b], b++
  Return C
```

Merge

```
Merge(A,B)
  C ← List of length Len(A)+Len(B)
  a ← 1, b ← 1
  For c = 1 to Len(C)
    If (b > Len(B))
      C[c] ← A[a], a++
    Else if (a > Len(A))
      C[c] ← B[b], b++
    Else if A[a] < B[b]
      C[c] ← A[a], a++
    Else
      C[c] ← B[b], b++
  Return C
```

Runtime: $O(|A|+|B|)$

MergeSort

```
MergeSort(L)
   If Len(L) = 1          \\ Base Case
      Return L
   Split L into equal L1 and L2
   A ← MergeSort(L1)
   B ← MergeSort(L2)
   Return Merge(A,B)
```

MergeSort

```
MergeSort(L)
  If Len(L) = 1          \\ Base Case
    Return L
  Split L into equal L1 and L2
  A ← MergeSort(L1)
  B ← MergeSort(L2)
  Return Merge(A,B)
```

O(n)

O(n)

MergeSort

```
MergeSort(L)
  If Len(L) = 1          \\ Base Case
    Return L
  Split L into equal L₁ and L₂
  A ← MergeSort(L₁)
  B ← MergeSort(L₂)
  Return Merge(A,B)
```

O(n)

2T(n/2)

O(n)

Question: Runtime

What is the runtime of `MergeSort`?

A) $O(n)$
B) $O(n \log(n))$
C) $O(n^2)$
D) $O(n^{\log(3)})$
E) $O(n^2 \log(n))$

Question: Runtime

What is the runtime of `MergeSort`?

A) $O(n)$
B) $O(n \log(n))$
C) $O(n^2)$
D) $O(n^{\log(3)})$
E) $O(n^2 \log(n))$

Master Theorem:

$a = 2$, $b = 2$, $d = 1$

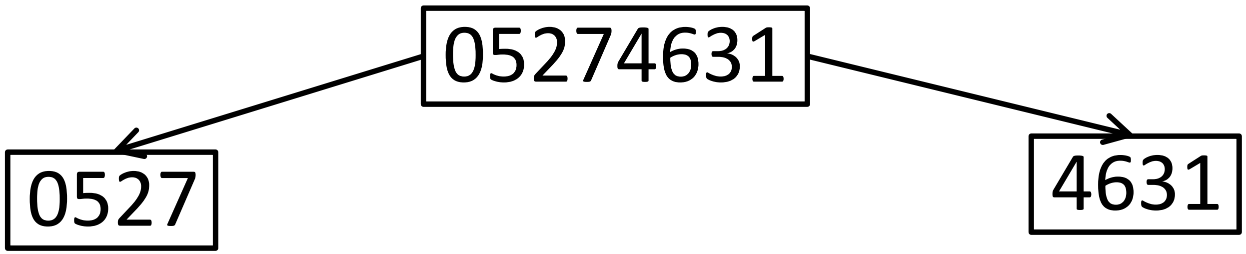
$a = b^d$

$O(n^d \log(n)) = O(n \log(n))$

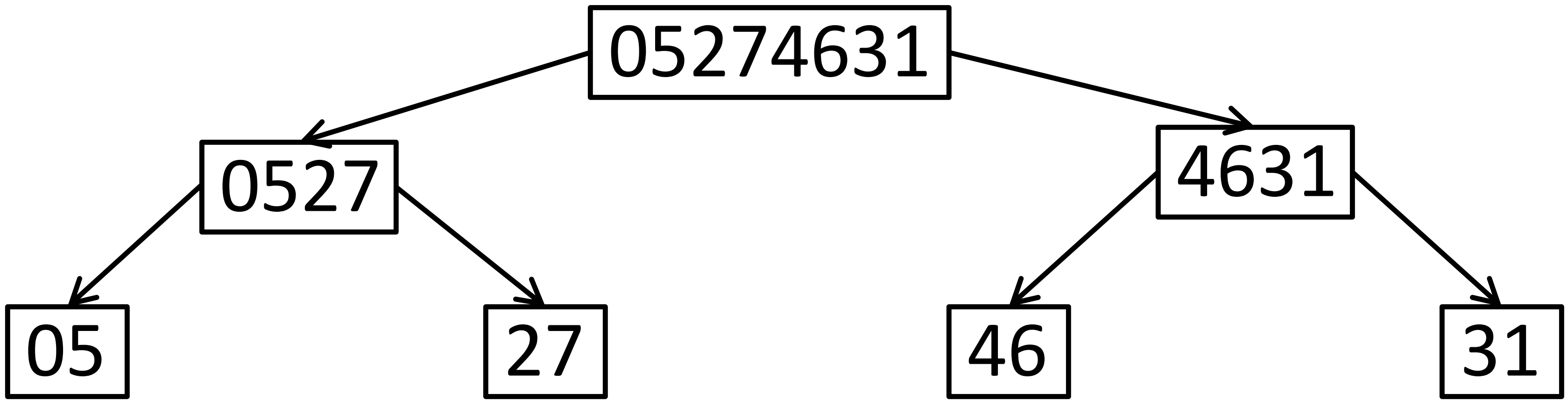
Example

$\boxed{05274631}$

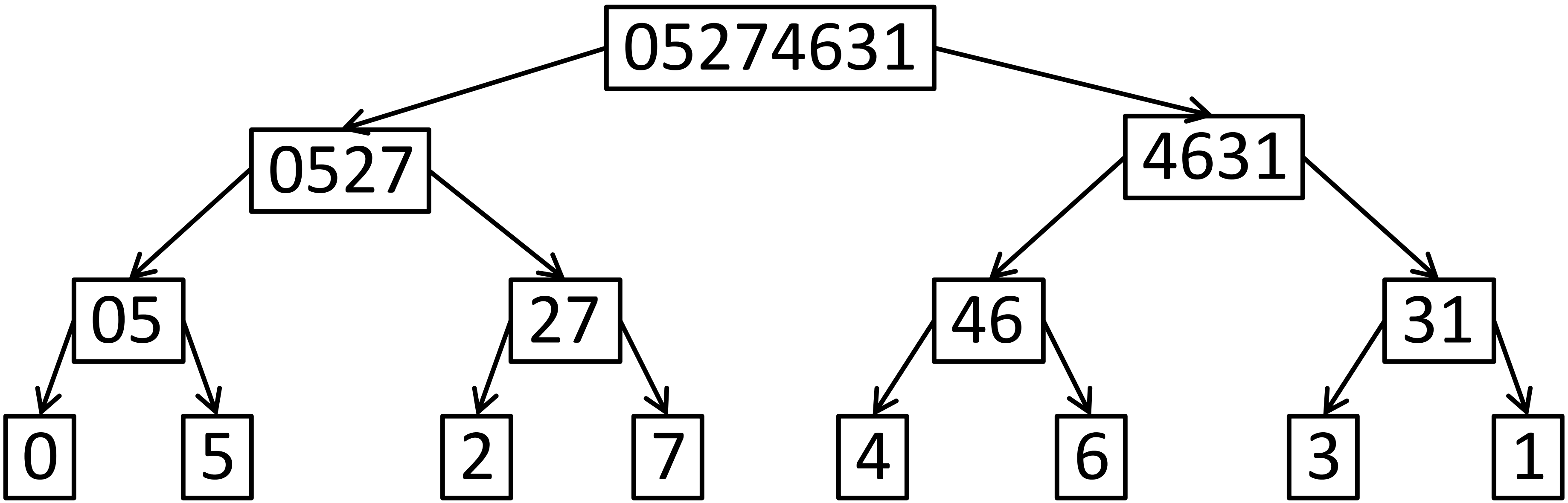
Example



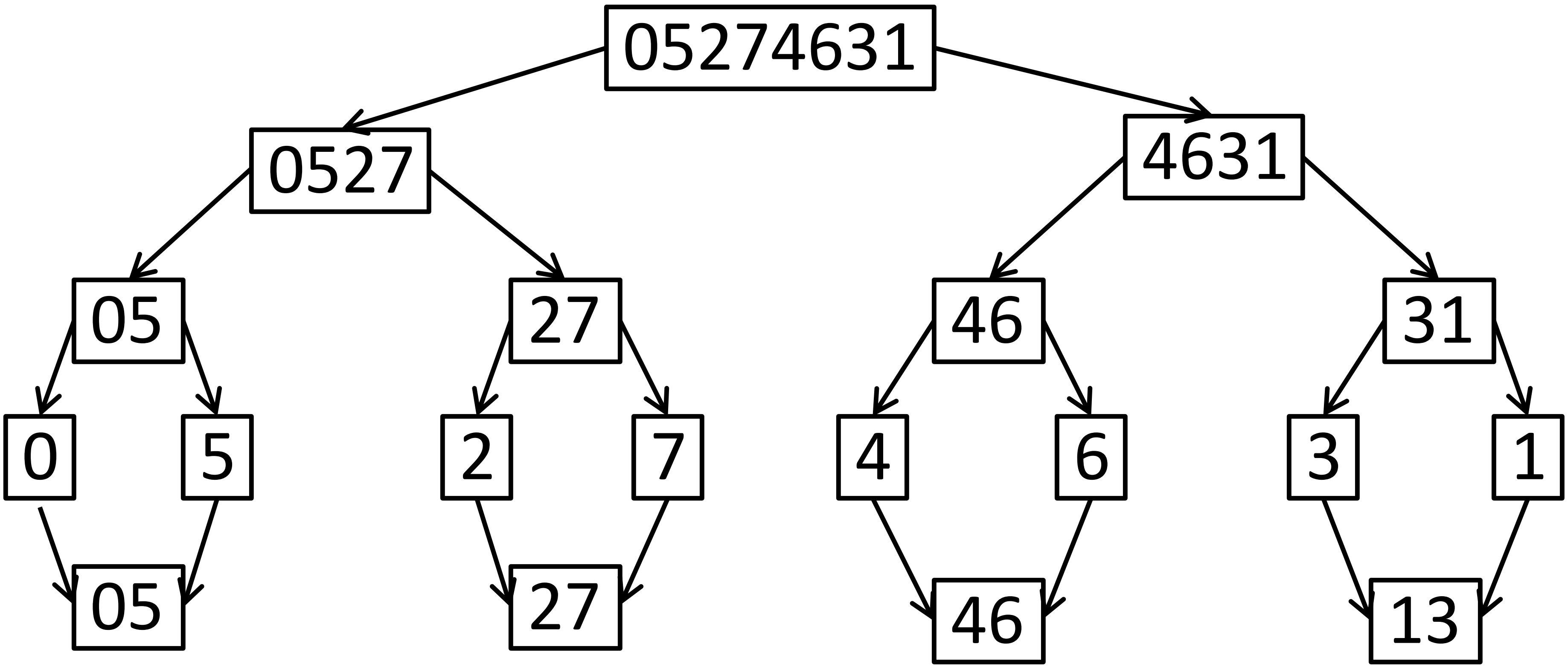
Example



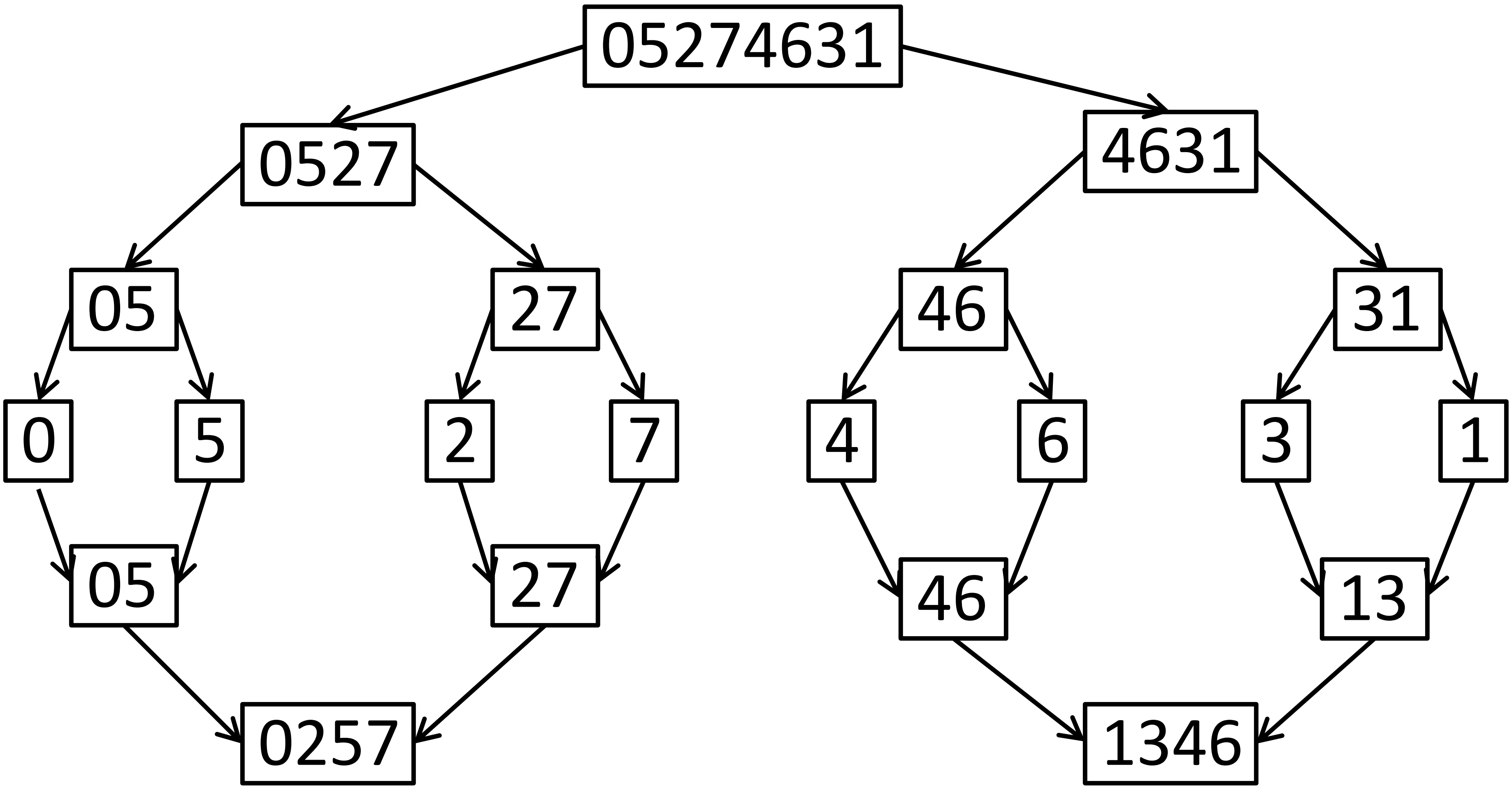
Example



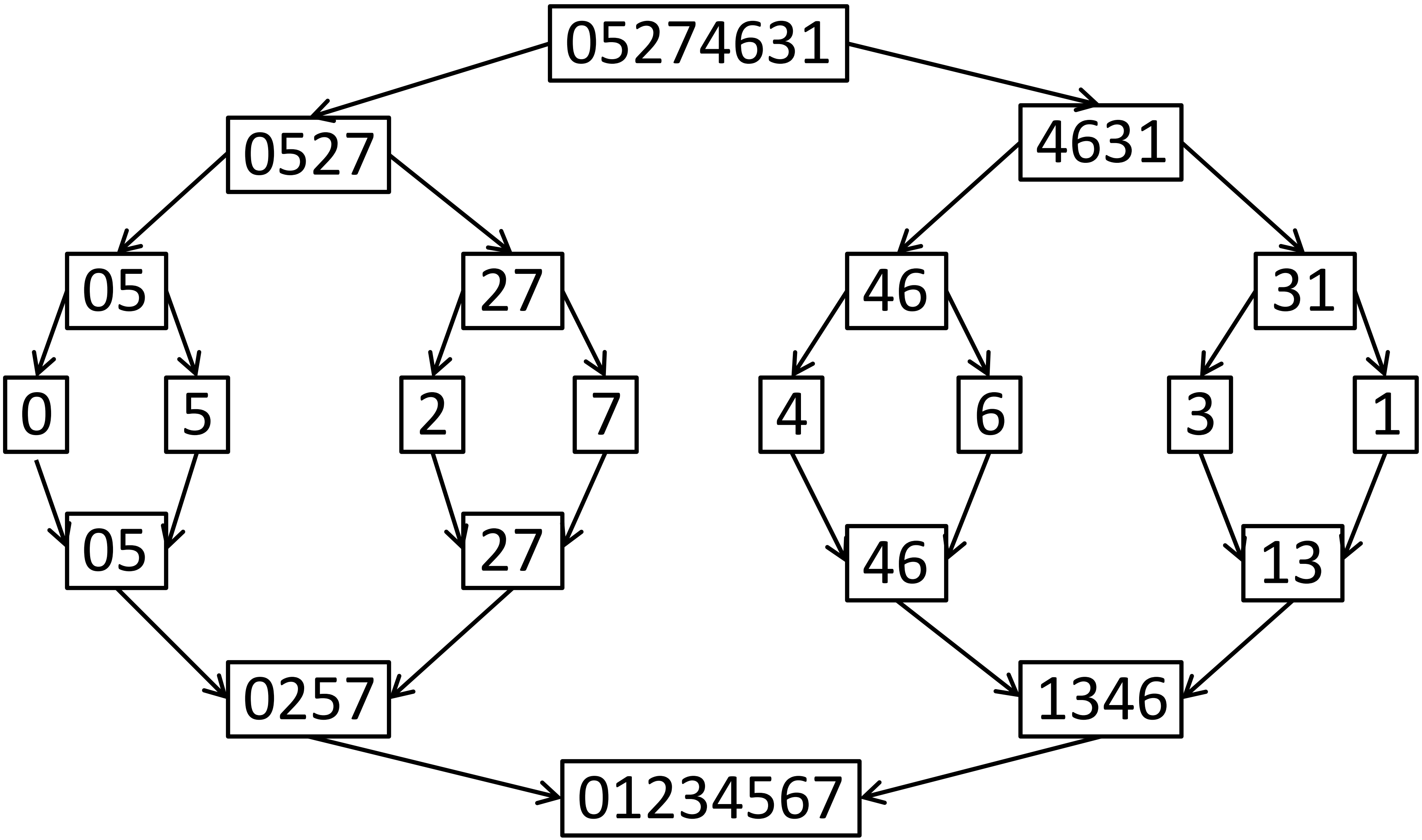
Example



Example



Example


05274631
0527
4631
05
27
46
31
0
5
2
7
4
6
3
1
05
27
46
13
0257
1346
01234567

Optimaility

O(n log(n)) is *optimal* for comparison based sorting algorithms.

Another Sorting Algorithm

- Repeatedly take the smallest remaining element and put it in the list.

Another Sorting Algorithm

- Repeatedly take the smallest remaining element and put it in the list.
- Naïve implementation takes $O(n^2)$ time.

Another Sorting Algorithm

- Repeatedly take the smallest remaining element and put it in the list.
- Naïve implementation takes $O(n^2)$ time.
- Use priority queues.

Heap Sort

```
HeapSort(L)
  Priority Queue Q
  For x in L
    Q.Insert(x)
  List C
  While(Q not empty)
    C ← C.Append(Q.DeleteMin())
  Return C
```

Heap Sort

```
HeapSort(L)
  Priority Queue Q
  For x in L
    Q.Insert(x)
  List C
  While(Q not empty)
    C ← C.Append(Q.DeleteMin())
  Return C
```

O(n) Inserts

Heap Sort

```
HeapSort(L)
  Priority Queue Q
  For x in L
    Q.Insert(x)
  List C
  While(Q not empty)
    C ← C.Append(Q.DeleteMin())
  Return C
```

O(n) Inserts

O(n) DeleteMins

Heap Sort

Runtime O(n log(n))

```
HeapSort(L)
  Priority Queue Q
  For x in L
    Q.Insert(x)
  List C
  While(Q not empty)
    C ← C.Append(Q.DeleteMin())
  Return C
```

O(n) Inserts

O(n) DeleteMins

Order Statistics

Suppose that we just want to find the median element of a list, or the largest, or the 10^{th} smallest.

Order Statistics

Suppose that we just want to find the median element of a list, or the largest, or the 10^{th} smallest.

Problem: Given a list L of numbers and an integer k, find the kth largest element of L.

Order Statistics

Suppose that we just want to find the median element of a list, or the largest, or the 10^{th} smallest.

Problem: Given a list L of numbers and an integer k, find the kth largest element of L.

Naïve Algorithm: Sort L and return kth largest. O(n log(n)) time.