# CS151 Intro to Data Structures

Hashmaps & Sorting

# Announcements

HW06 due next Wednesday 11/29

Lab08 due next Wednesday too

No lab this week

HW07 due 12/05

Need to leave office hours around 3:15 today

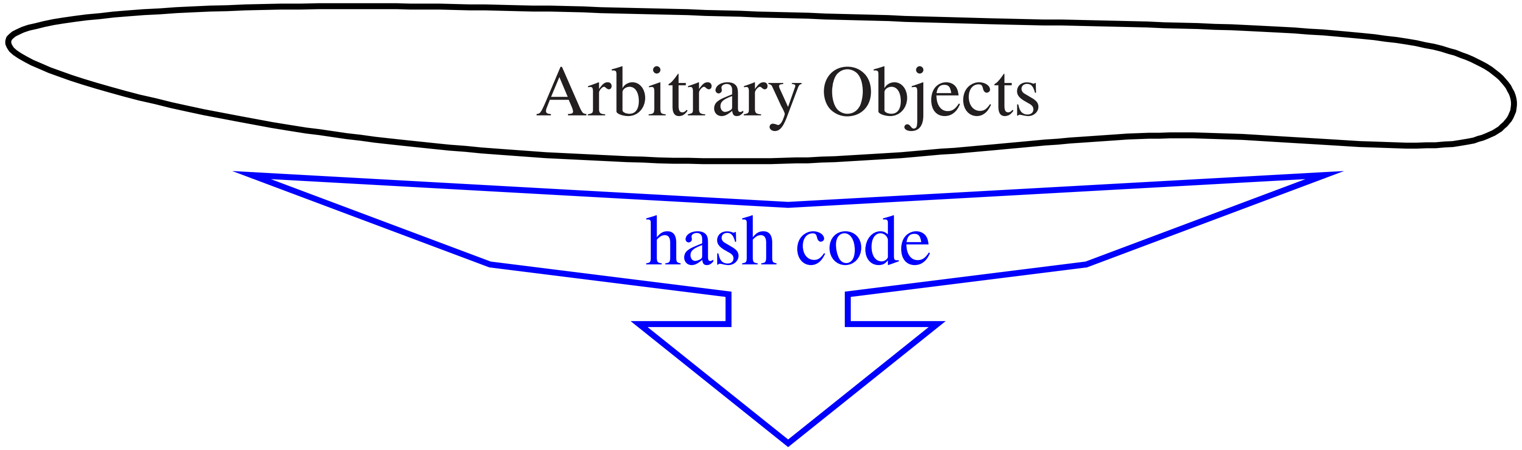
# Outline

**Review**

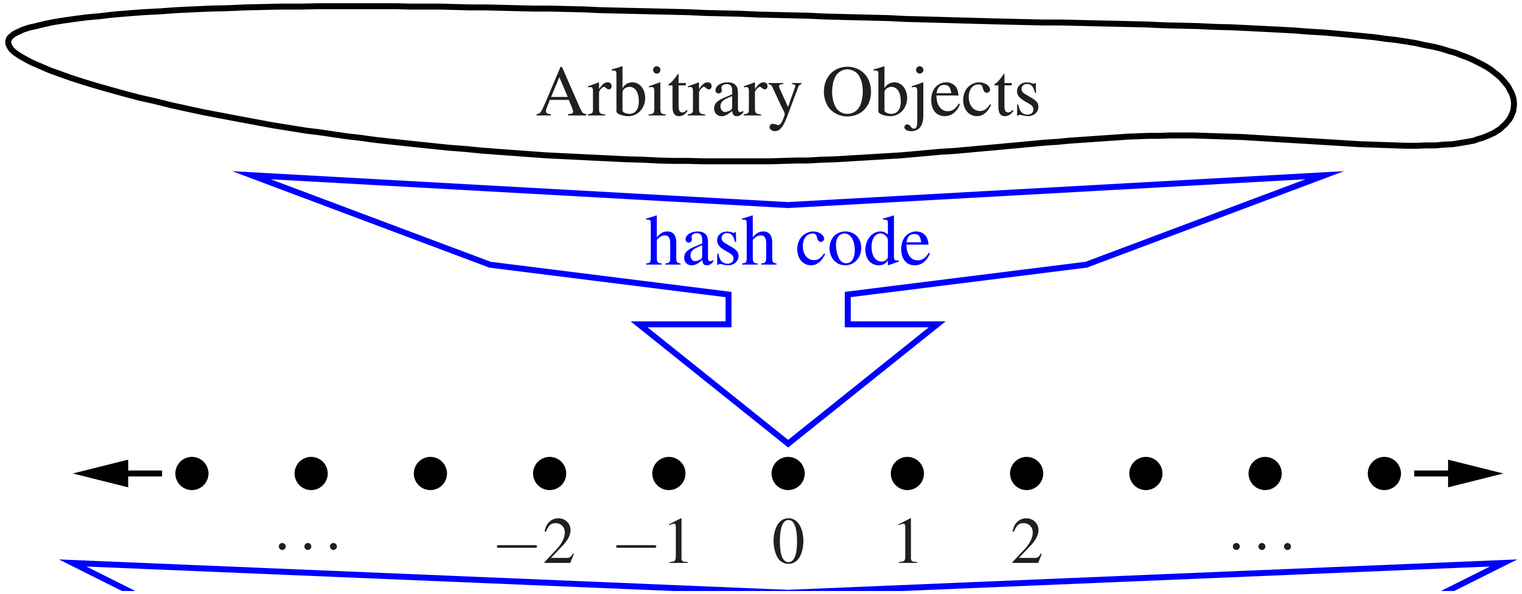
MergeSort

# Hash Function Illustration

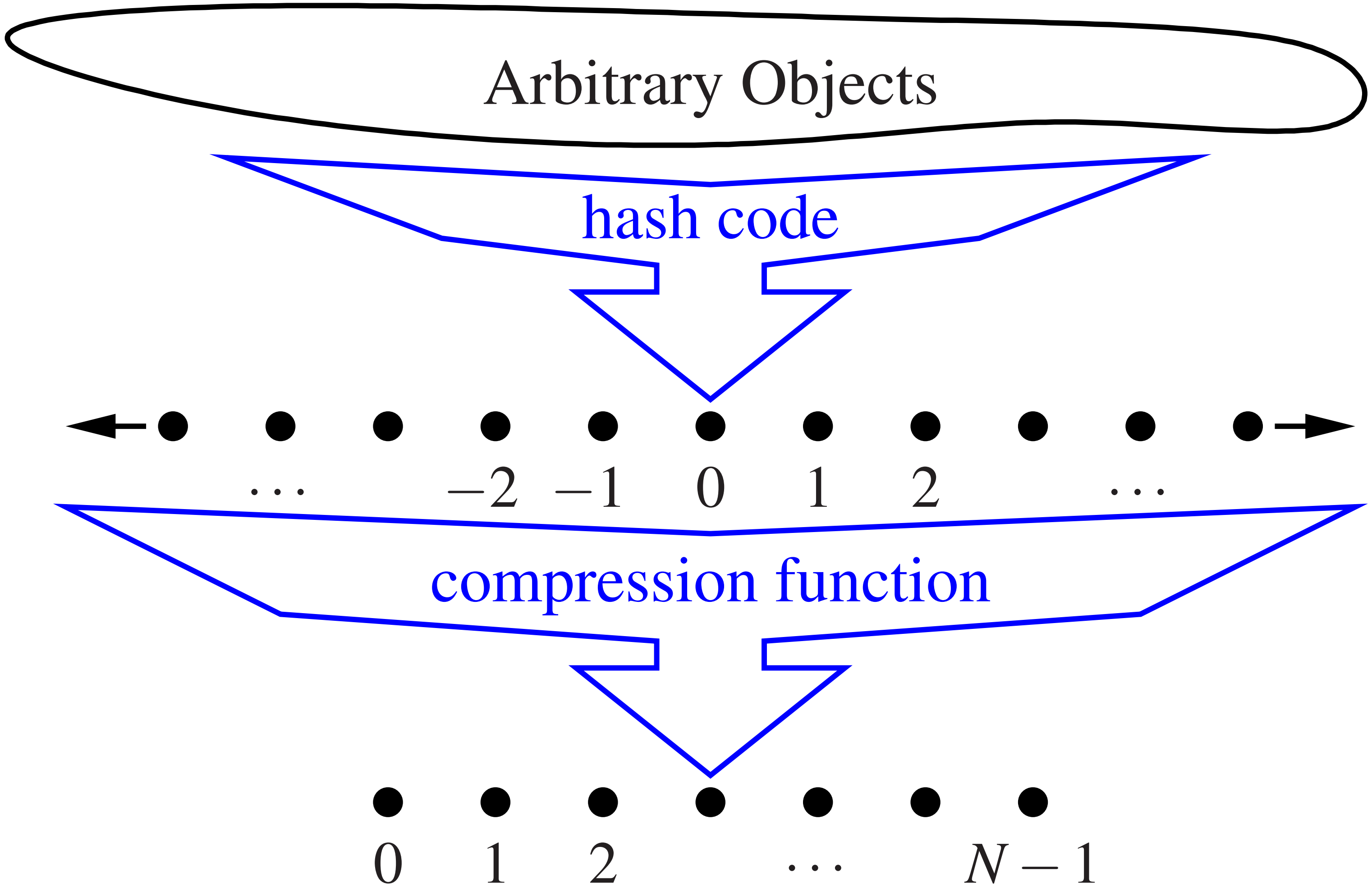
# Hash Function Illustration



# Hash Function Illustration



# Hash Function Illustration



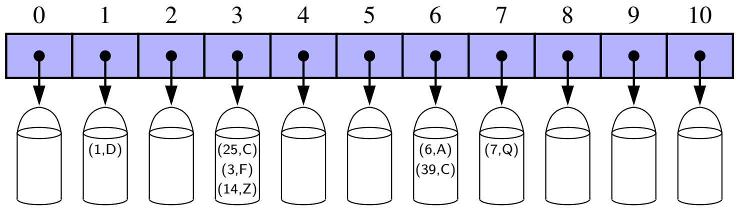
# Collision Handling

Collison: keys mapped to same hash value

A hash function does not guarantee one-to-one mapping

no hash function does

Separate chaining: Each index holds a collection of entries




Open addressing

Linear/Quadratic probing

Double hasing

# Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list
- Efficient probing requires soft/lazy deletions – tombstoning, why?
- May require graveyard defragmenting

# Performance Analysis

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- Expected time of all operations is $O(1)$ provided $\alpha$ is not close to 100%

# Probing Tradeoffs

- Linear probing – best cache performance but most sensitive to clustering
- Double hashing – poor cache performance but exhibits virtually no clustering
- Quadratic – inbetween

- As load factor approaches 100%, number of probes rises dramatically
- Even with good hash functions, keep load factor 80% or below (50% is typical)
- Other open addressing methods besides probing

# Performance of Hashtable

| | Hash Expected | Hash Worst |
|---|---|---|
| search | | |
| insert | | |
| remove | | |
| min/max | | |

| | Unsorted array | Sorted array | Unsorted list | Sorted list | BST balanced | Hash Expected |
|---|---|---|---|---|---|---|
| search | $O(n)^*$ | $O(logn)$ | $O(n)$ | $O(n)$ | $O(logn)$ | $O(1)$ |
| insert | $O(1)^*$ | $O(n)$ | $O(1)$ | $O(n)$ | $O(logn)$ | $O(1)$ |
| remove | $O(1)^*$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(logn)$ | $O(1)$ |
| min/max | $O(n)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(logn)$ | $O(n)$ |

# Performance of Hashtable

| | Hash Expected | Hash Worst |
|---|---|---|
| search | $O(1)$ | $O(n)$ |
| insert | $O(1)$ | $O(n)$ |
| remove | $O(1)$ | $O(n)$ |
| min/max | $O(n)$ | $O(n)$ |

| | Unsorted array | Sorted array | Unsorted list | Sorted list | BST balanced | Hash Expected |
|---|---|---|---|---|---|---|
| search | $O(n)^*$ | $O(logn)$ | $O(n)$ | $O(n)$ | $O(logn)$ | $O(1)$ |
| insert | $O(1)^*$ | $O(n)$ | $O(1)$ | $O(n)$ | $O(logn)$ | $O(1)$ |
| remove | $O(1)^*$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(logn)$ | $O(1)$ |
| min/max | $O(n)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(logn)$ | $O(n)$ |

# Hashtable vs Array

- A hashtable is an unsorted array with a fast search – $O(1)$ expected
- An array is more memory efficient, but slower for searching (without key-index pairing)
- If your data has natural indexing (a way to assign/associate an ID/unique integer to each entry), then you are better off using an array. You have a hash function with 1-to-1 mapping and guaranteed no collisions

# Hashtable Size

Should be a prime

twice the size of max number of keys

- or 1.3 times if $n$ is very large
- 1/1.333 = 75% load factor

Keep track of load factor and expand (rehash) the hash table when necessary

# Outline

Review

**MergeSort**

# Divide-and-Conquer

*Divide* – the problem (input) into smaller pieces

*Conquer* – solve each piece individually, usually recursively

*Combine* – the piecewise solutions into a global solution

Usually involves recursion

Analysis usually involves solving recurrence relations

# Merge Sort

Sort a sequence of numbers $A$, $|A| = n$

Base: $|A| = 1$, then it's already sorted

General

- divide: split $A$ into two halves, each of size $\frac{n}{2}$ ($\left\lfloor\frac{n}{2}\right\rfloor$ and $\left\lceil\frac{n}{2}\right\rceil$)
- conquer: sort each half (by calling mergeSort recursively)
- combine: merge the two sorted halves into a single sorted list

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 8 | 4 | 1 | 7 | 2 | 5 | 3 |

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 8 | 4 | 1 | 7 | 2 | 5 | 3 |

| | | | |
|---|---|---|---|
| 6 | 8 | 4 | 1 |

| | | | |
|---|---|---|---|
| 7 | 2 | 5 | 3 |

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 8 | 4 | 1 | 7 | 2 | 5 | 3 |

| | | | |
|---|---|---|---|
| 6 | 8 | 4 | 1 |

| | | | |
|---|---|---|---|
| 7 | 2 | 5 | 3 |

| | |
|---|---|
| 6 | 8 |

| | |
|---|---|
| 4 | 1 |

| | |
|---|---|
| 7 | 2 |

| | |
|---|---|
| 5 | 3 |

# Example

6 8 4 1 7 2 5 3

6 8 4 1     7 2 5 3

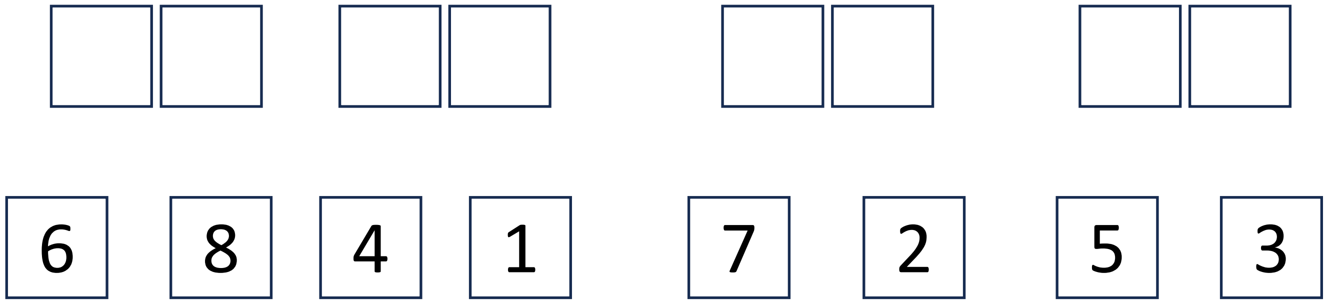
6 8    4 1    7 2    5 3

6   8   4   1   7   2   5   3

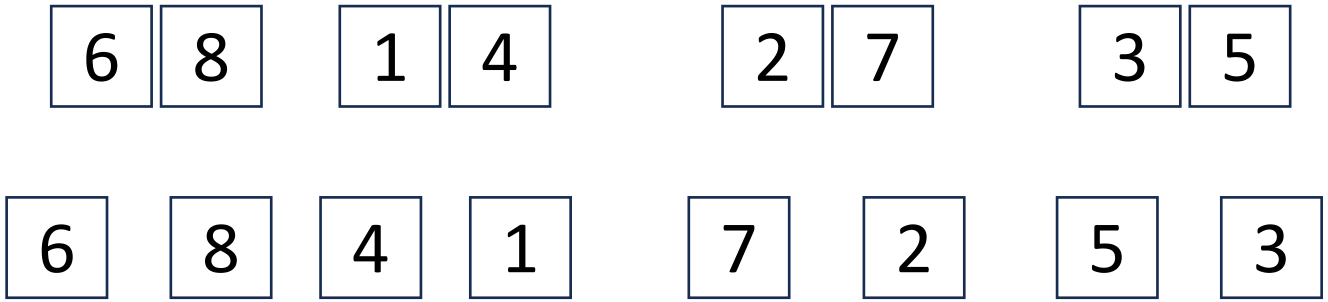
# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 8 | 4 | 1 | 7 | 2 | 5 | 3 |

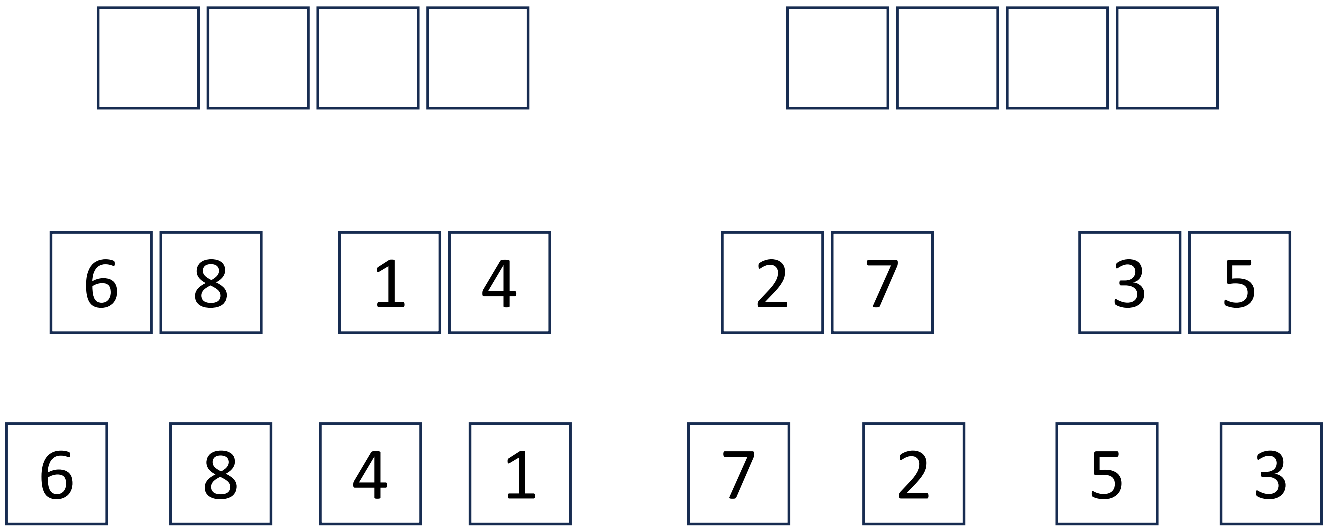
# Example



# Example



# Example



# Example

1 4 6 8    2 3 5 7

6 8    1 4    2 7    3 5

6    8    4    1    7    2    5    3

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

1 4 6 8    2 3 5 7

6 8    1 4    2 7    3 5

6    8    4    1    7    2    5    3

# Example

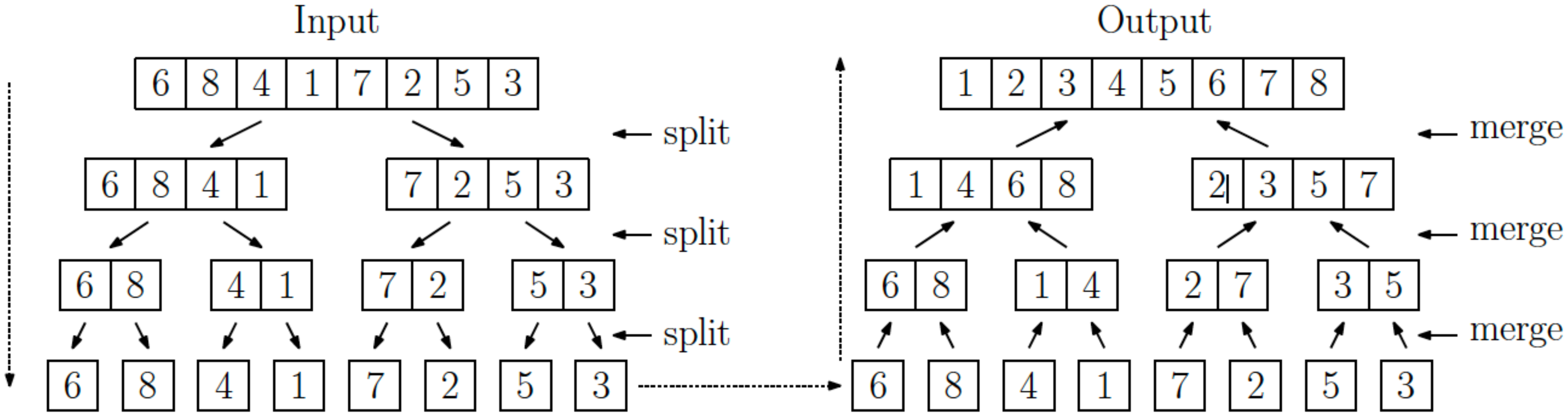
1 2 3 4 5 6 7 8

1 4 6 8      2 3 5 7

6 8    1 4    2 7    3 5

6   8   4   1   7   2   5   3

# Example - summary



# Merge Sort

Sort a sequence of numbers $A$, $|A| = n$

Base: $|A| = 1$, then it's already sorted

General

- divide: split $A$ into two halves, each of size $\frac{n}{2}$ ($\left\lfloor\frac{n}{2}\right\rfloor$ and $\left\lceil\frac{n}{2}\right\rceil$)
- conquer: sort each half (by calling mergeSort recursively)
- combine: merge the two sorted halves into a single sorted list

# Algorithm

```
mergeSort(S):
  if …
    …
  else
    s1 = …
    s2 = …
    …
    S = …
```

# Algorithm

```
mergeSort(S):
  if S.size() <= 1
    return
  else
    s1 = S[0,n/2]
    s2 = S[n/2+1,n-1]
    mergeSort(s1)
    mergeSort(s2)
    S = merge(s1, s2)
```

# The Merge

The key is the merging process

How does one merge two sorted lists?

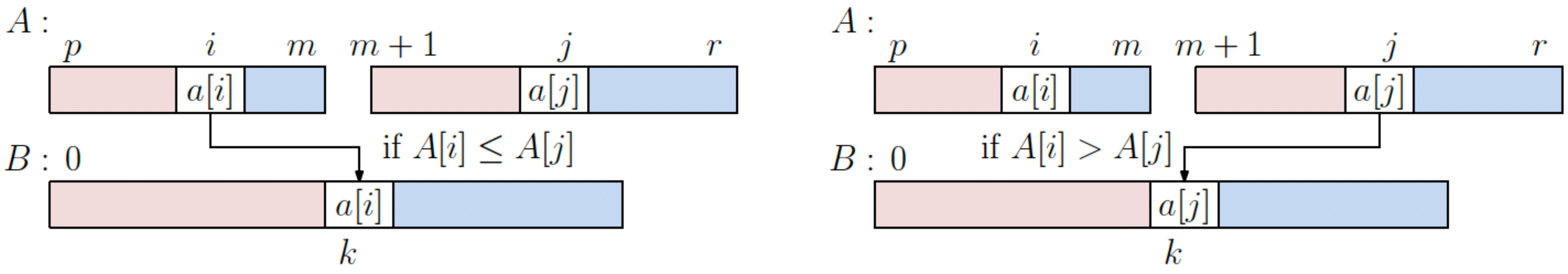
- Each element in $A \cup B$ is considered once
- $O(n)$

```
Algorithm merge(A, B)
  Input sorted A and B
  Output sorted A ∪ B
  S = empty sequence
  while(!A.isEmpty() and
        !B.isEmpty())
     if A.first() < B.first()
      S.addLast(A.removeFirst())
     else
      S.addLast(B.removeFirst())
   while (!A.isEmpty())
     S.addLast(A.removeFirst())
   while (!B.isEmpty())
     S.addLast(B.removeFirst())
   return S
```

# In-place Merge

What if we don't want to use new lists?

How does one merge two sorted lists `A[p,…,m]` and `A[m+1,…,r]`?




Use a temp array `B` and maintain two indices `i` and `j`, one for each subarray

# Array mergeSort

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]









}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]
  i=p; j=m+1; k=0









}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]
  i=p; j=m+1; k=0
  while(i<=m and j<=r){




  }



}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]
  i=p; j=m+1; k=0
  while(i<=m and j<=r){
    if (A[i]<=A[j])

    else

  }

}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]
  i=p; j=m+1; k=0
  while(i<=m and j<=r){
    if (A[i]<=A[j])
      B[k++] = A[i++]
    else
      B[k++] = A[j++]
  }



}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Array mergeSort

```
merge(A, p, m, r){
  new B[0, r-p]
  i=p; j=m+1; k=0
  while(i<=m and j<=r){
    if (A[i]<=A[j])
      B[k++] = A[i++]
    else
      B[k++] = A[j++]
  }
  while(i<=m) B[k++]=A[i++]
  while(j<=r) B[k++]=A[j++]
  copy B back to A[p, r]
}
```

```
mergeSort(A, p, r){
  if (p<r) {
    m = (p+r)/2
    mergeSort(A, p, m)
    mergeSort(A, m+1, r)
    merge(A, p, m, r)
  }
}
```

# Analysis

`merge`:

$O(r - p + 1) \to O(n)$

Let $T(n)$ denote the worse case running time of `mergeSort` on an input of size $n$

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

# Solving the Recurrence

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

# Solving the Recurrence

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

# Solving the Recurrence

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

# Solving the Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n$
- $= 2\left(2\left(2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right) + \left(\frac{n}{2}\right)\right) + n$

# Solving the Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n$
- $= 2\left(2\left(2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right) + \left(\frac{n}{2}\right)\right) + n = 2^3T\left(\frac{n}{2^3}\right) + 3n$

# Solving the Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n$
- $= 2\left(2\left(2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right) + \left(\frac{n}{2}\right)\right) + n = 2^3T\left(\frac{n}{2^3}\right) + 3n$
- $= \cdots$

# Solving the Recurrence

- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n$
- $= 2\left(2\left(2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right) + \left(\frac{n}{2}\right)\right) + n = 2^3T\left(\frac{n}{2^3}\right) + 3n$
- $= \cdots$
- $= 2^kT\left(\frac{n}{2^k}\right) + kn$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

- $T(n) = 2^{logn} \cdot T(1) + logn \times n$
$$= n + nlogn$$
- $= O(n \log n)$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

$T(n)$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

$T(n) = 2^{logn} \cdot T(1) + logn \times n$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

$$T(n) = 2^{logn} \cdot T(1) + logn \times n$$
$$= n + nlogn$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

What is the value of $k$?

- $\frac{n}{2^k} = 1$
- $n = 2^k$
- $k = logn$

$$T(n) = 2^{logn} \cdot T(1) + logn \times n$$
$$= n + nlogn$$
$$= O(n \log n)$$

# Summary of Sorting Algorithms

| Algorithm | Time | Notes |
|---|---|---|
| selection-sort | $O(n^2)$ | ▪ slow<br>▪ in-place<br>▪ for small data sets (< 1K) |
| insertion-sort | $O(n^2)$ | ▪ slow<br>▪ in-place<br>▪ for small data sets (< 1K) |
| heap-sort | $O(nlogn)$ | ▪ fast<br>▪ in-place<br>▪ for large data sets (1K — 1M) |
| merge-sort | $O(nlogn)$ | ▪ fast<br>▪ sequential data access<br>▪ for huge data sets (> 1M) |