Sorting

Each record contains a field called the **key**.
Linear order: comparison.

The Sorting Problem

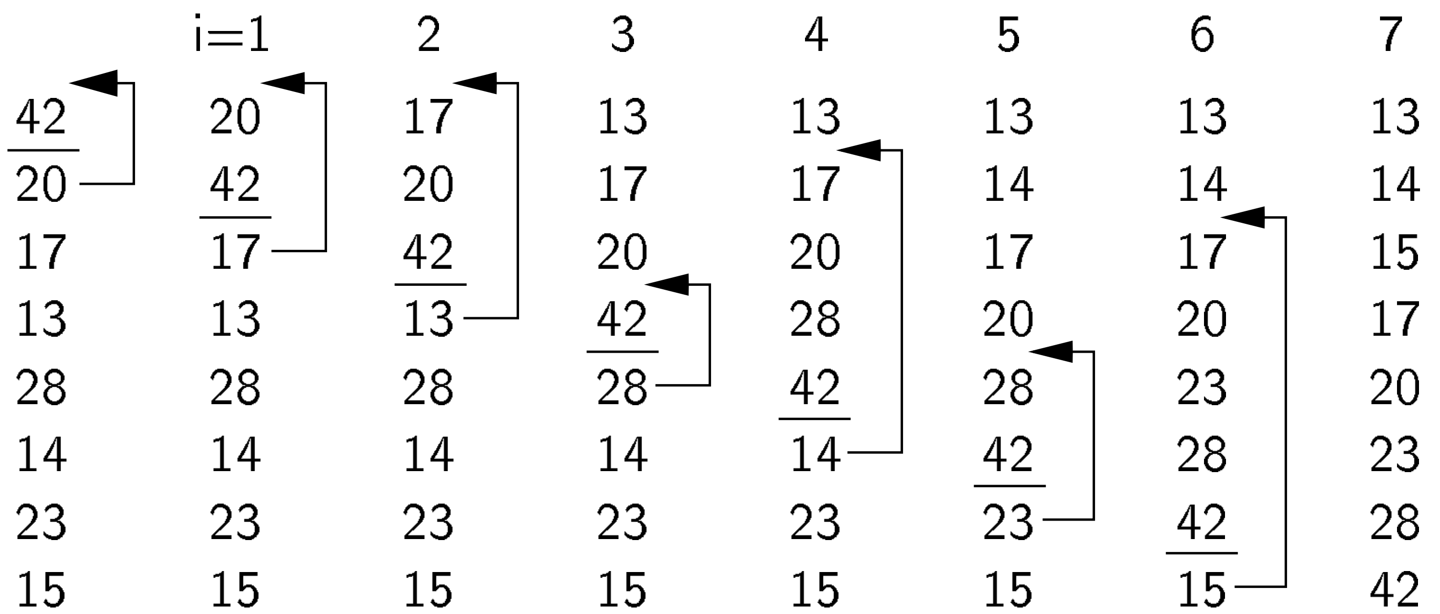
Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

Insertion Sort

```
static void inssort(Elem[] array) {  // Insertion Sort
  for (int i=1; i<array.length; i++) // Insertrecord
    for (int j=i; (j>0) &&
            (array[j].key()<array[j-1].key()); j--)
      DSutil.swap(array, j, j-1);
}
```



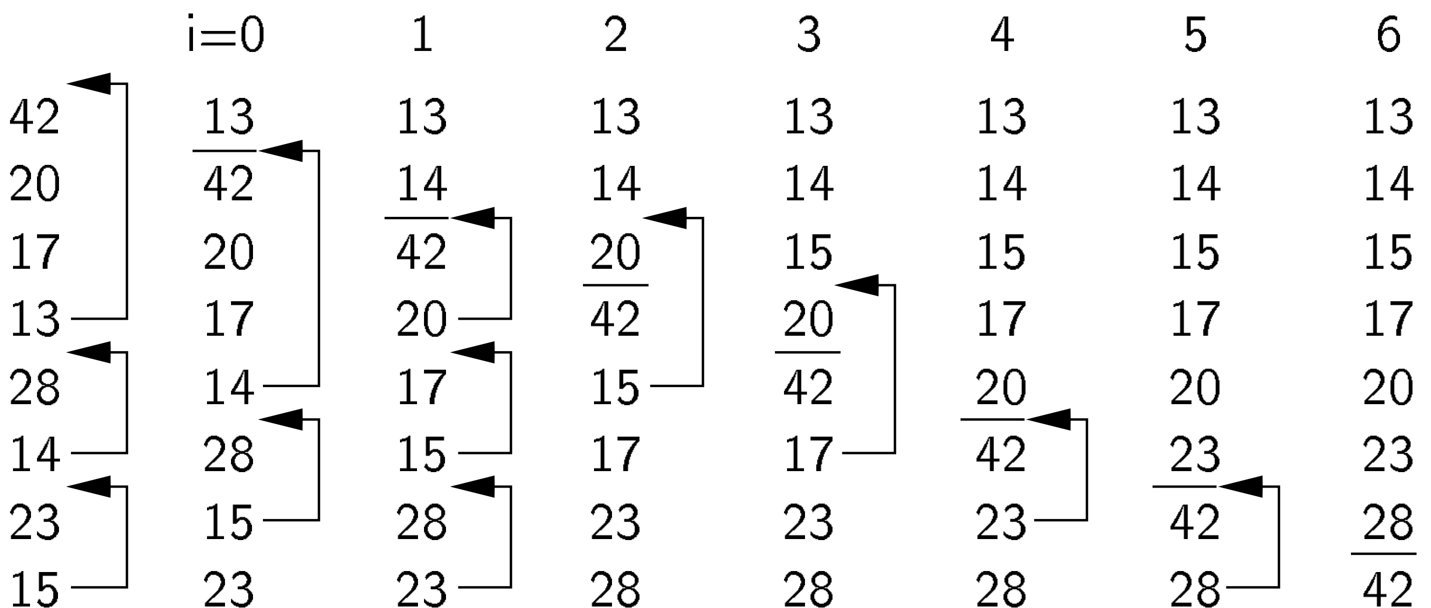
Best Case:

Worst Case:

Average Case:

Bubble Sort

```
static void bubsort(Elem[] array) {    // Bubble Sort
  for (int i=0; i<array.length-1; i++) // Bubble up
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
}
```



Best Case:

Worst Case:

Average Case:

Selection Sort

```
static void selsort(Elem[] array) {   // Selection Sort
  for (int i=0; i<array.length-1; i++) { // Select i'th
    int lowindex = i;                 // Remember its index
    for (int j=array.length-1; j>i; j--) // Find least
      if (array[j].key() < array[lowindex].key())
        lowindex = j;                 // Put it in place
    DSutil.swap(array, i, lowindex);
  }
}
```

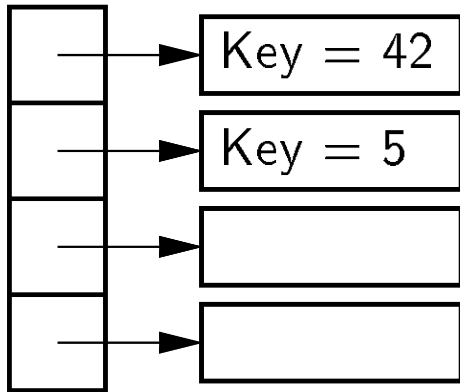
	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Best Case:

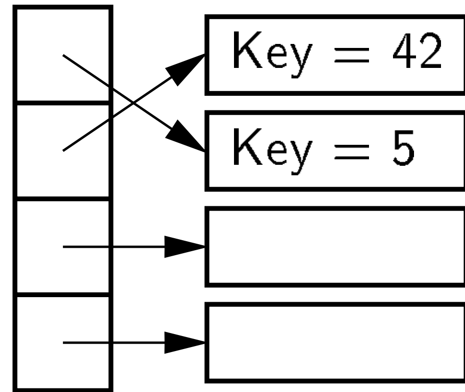
Worst Case:

Average Case:

Pointer Swapping



(a)



(b)

Exchange Sorting

Summary

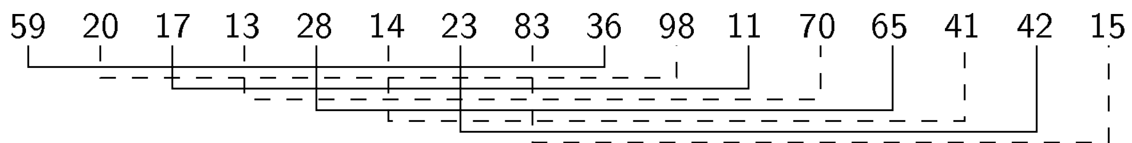
	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

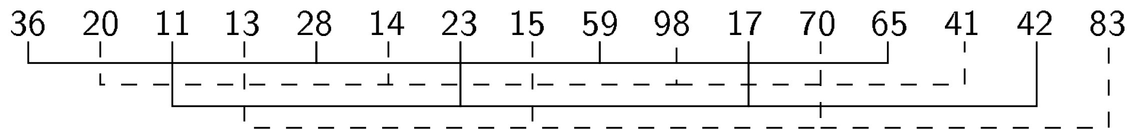
All of these sorts rely on **exchanges** of adjacent records.

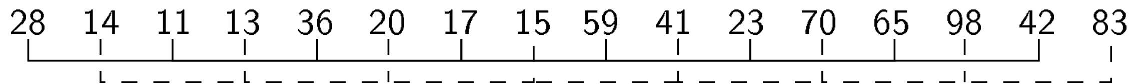
What is the average number of exchanges required?

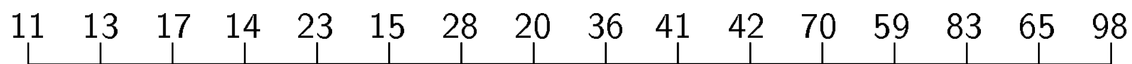
Shellsort

```
static void shellsort(Elem[] array) { // Shellsort
  for (int i=array.length/2; i>2; i/=2)
    for (int j=0; j<i; j++)           // Sort sublists
      inssort2(array, j, i);
  inssort2(array, 0, 1); // Could call regular inssort
}
// Modified version of Ins Sort for varying increments
static void inssort2(Elem[] A, int start, int incr) {
  for (int i=start+incr; i<A.length; i+=incr)
    for (int j=i; (j>=incr)&&
                (A[j].key()<A[j-incr].key()); j-=incr)
      DSutil.swap(A, j, j-incr);
}
```









11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

$O(n^{1.5})$

Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```
static void qsort(Elem[] array, int i, int j) {
  int pivotindex = findpivot(array, i, j); // Pick piv
  DSutil.swap(array, pivotindex, j); // Stick at end
  // k will be the first position in the right subarray
  int k = partition(array, i-1, j, array[j].key());
  DSutil.swap(array, k, j);    // Put pivot in place
  if ((k-i) > 1) qsort(array, i, k-1);  // Sort left
  if ((j-k) > 1) qsort(array, k+1, j);  // Sort right
}

static int findpivot(Elem[] array, int i, int j)
  { return (i+j)/2; }
```

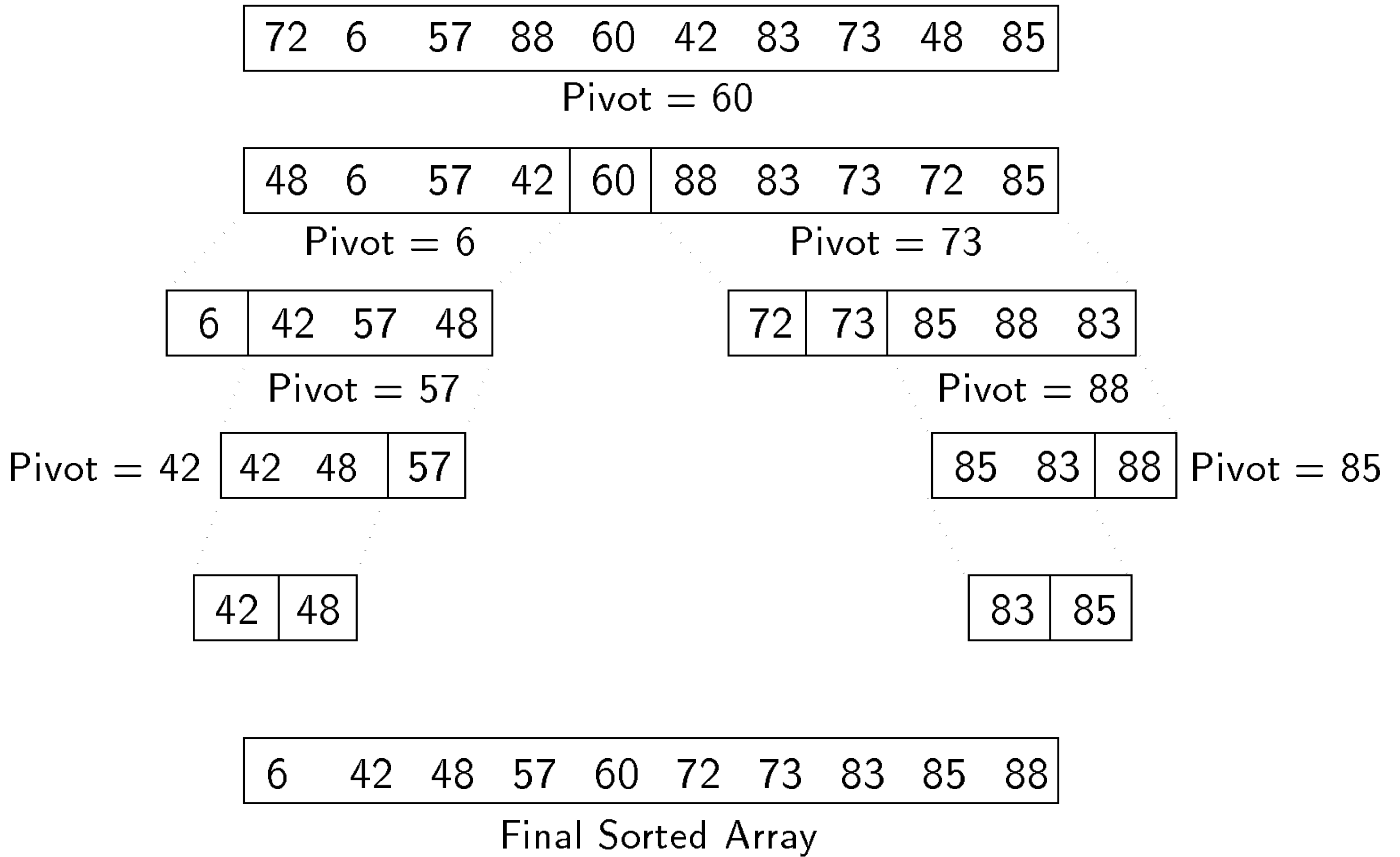
Quicksort Partition

```
static int partition(Elem[] array, int l, int r,
                     int pivot) {
  do {          // Move bounds inward until they meet
    while (array[++l].key() < pivot); // Move it right
    while ((r!=0) && (array[--r].key()>pivot));
    DSutil.swap(array, l, r); // Swap out-of-place vals
  } while (l < r);            // Stop when they cross
  DSutil.swap(array, l, r);   // Reverse wasted swap
  return l; }// Return first pos in right partition
```

```
Initial        72  6  57  88  85  42  83  73  48  60
            l                                     r

Pass 1         72  6  57  88  85  42  83  73  48  60
               l                              r

Swap 1         48  6  57  88  85  42  83  73  72  60
               l                              r

Pass 2         48  6  57  88  85  42  83  73  72  60
                          l       r

Swap 2         48  6  57  42  85  88  83  73  72  60
                          l       r

Pass 3         48  6  57  42  85  88  83  73  72  60
                          r   l

Swap 3         48  6  57  85  42  88  83  73  72  60
                          r   l

Reverse Swap   48  6  57  42 |85  88  83  73  72  60
                          r   l
```

The cost for Partition is $\Theta(n)$.

Quicksort Example



Final Sorted Array

Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:

$$\begin{aligned} T(n) &= n + 1 + \frac{1}{n-1}\sum_{k=1}^{n-1}(T(k) + T(n-k)) \\ &= \Theta(n \log n) \end{aligned}$$

Optimizations for Quicksort:

- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```

36 20 17 13 28 14 23 15

| 20 36 | 13 17 | 14 28 | 15 23 |

| 13 17 20 36 | 14 15 23 28 |

| 13 14 15 17 20 23 28 36 |

Mergesort Implementation

Mergesort is tricky to implement.

```
static void mergesort(Elem[] array, Elem[] temp,
                      int l, int r) {
  int mid = (l+r)/2;                 // Select midpoint
  if (l == r) return;                // One element list
  mergesort(array, temp, l, mid);    // Ssort first half
  mergesort(array, temp, mid+1, r); // Sort second half
  for (int i=l; i<=r; i++)           // Copy subarray
    temp[i] = array[i];
  // Do the merge operation back to array
  int i1 = l; int i2 = mid + 1;
  for (int curr=l; curr<=r; curr++) {
    if (i1 == mid+1)  // Left sublist exhausted
      array[curr] = temp[i2++];
    else if (i2 > r)  // Right sublist exhausted
      array[curr] = temp[i1++];
    else if (temp[i1].key() < temp[i2].key())
      array[curr] = temp[i1++]; // Get smaller val
    else array[curr] = temp[i2++];
  }
}
```

Mergesort cost:

Mergesort is good for sorting linked lists.

Optimized Mergesort

```
static void mergesort(Elem[] array, Elem[] temp,
                      int l, int r) {
  int i, j, k, mid = (l+r)/2;  // Select the midpoint
  if (l == r) return;          // List has one element
  if ((mid-l) >= THRESHOLD)
    mergesort(array, temp, l, mid);
  else inssort(array, l, mid-l+1);
  if ((r-mid) > THRESHOLD)
    mergesort(array, temp, mid+1, r);
  else inssort(array, mid+1, r-mid);
  // Do the merge operation.  Copy 2 halves to temp.
  for (i=l; i<=mid; i++) temp[i] = array[i];
  for (j=1; j<=r-mid; j++) temp[r-j+1] = array[j+mid];
  // Merge sublists back to array
  int a = temp[l].key(); int b = temp[r].key();
  for (i=l,j=r,k=l; k<=r; k++)
    if (a < b)
      { array[k] = temp[i++]; a = temp[i].key(); }
    else { array[k] = temp[j--]; b = temp[j].key(); }
}
```

Heapsort

Heapsort uses a max-heap.

```
static void heapsort(Elem[] array) {  // Heapsort
  MaxHeap H = new MaxHeap(array, array.length,
                          array.length);
  for (int i=0; i<array.length; i++)  // Now sort
    H.removemax();   // Put max value at end of heap
}
```

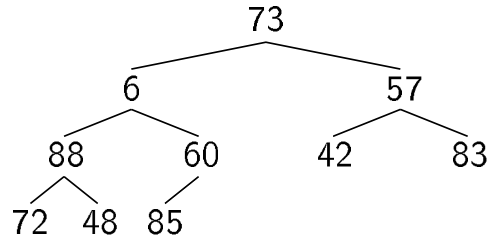
Cost of Heapsort:

Cost of finding k largest elements:

Heapsort Example

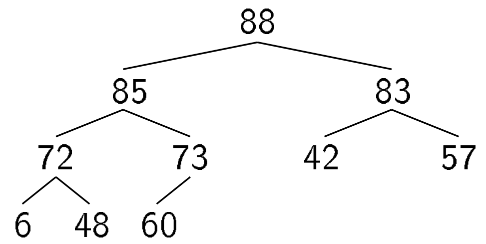
Original Numbers

73 6 57 88 60 42 83 72 48 85



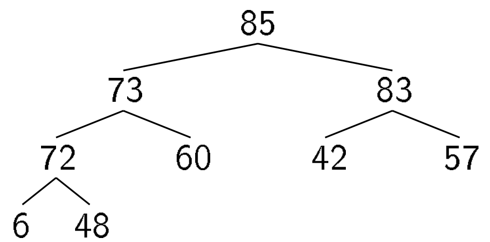
Build Heap

88 85 83 72 73 42 57 6 48 60



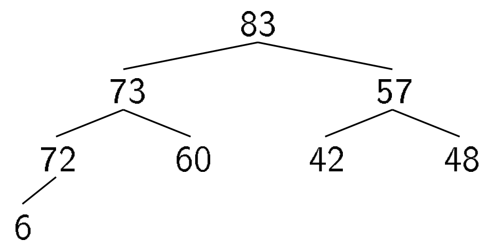
Remove 88

85 73 83 72 60 42 57 6 48 | 88



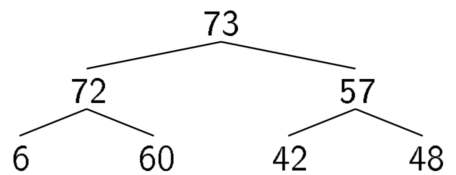
Remove 85

83 73 57 72 60 42 48 6 | 85 88



Remove 83

73 72 57 6 60 42 48 | 83 85 88



Binsort

A simple, efficient sort:

```
for (i=0; i<n; i++)
  B[A[i].key()] = A[i];
```

Ways to generalize:

- Make each bin the head of a list.

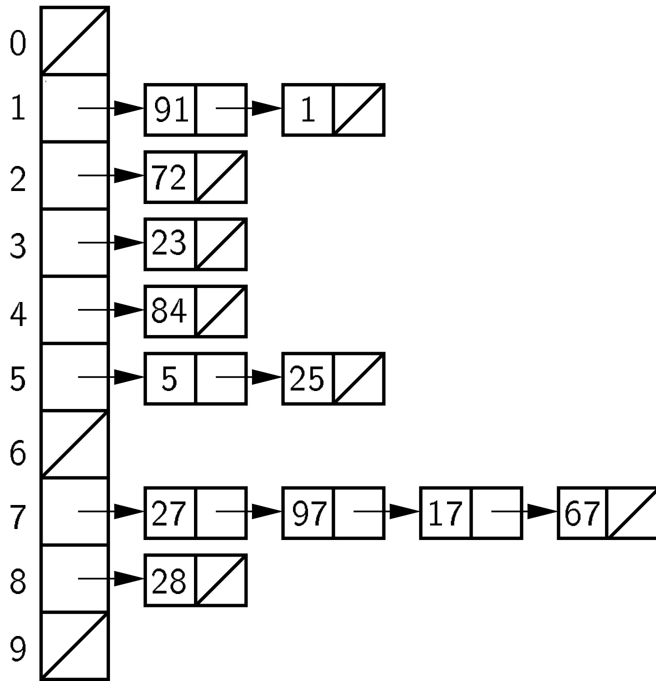
- Allow more keys than records.

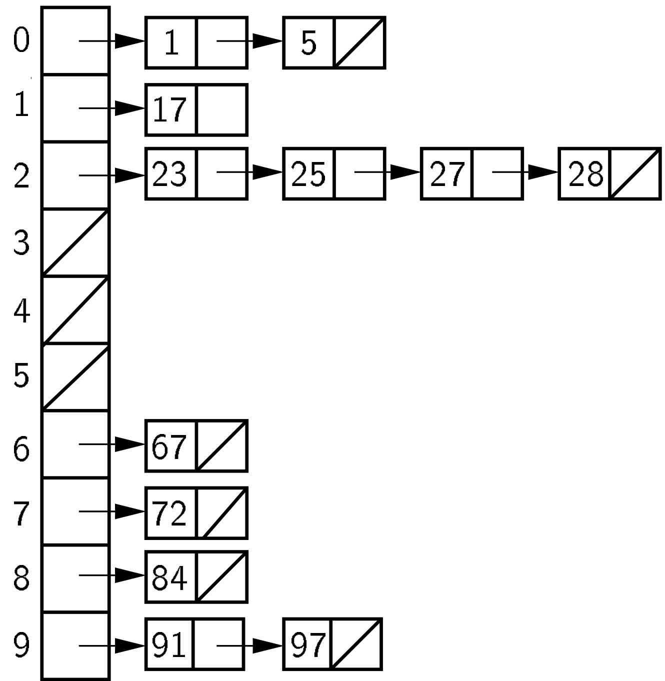
```
void binsort(ELEM *A, int n) {
  list B[MaxKeyValue];
  for (i=0; i<n; i++) B[A[i].key()].append(A[i]);
  for (i=0; i<MaxKeyValue; i++)
    for (B[i].first(); B[i].isInList(); B[i].next())
      output(B[i].currValue());
}
```

Cost:

Radix Sort

Initial List: 27 91 1 97 17 23 84 28 72 5 67 25





Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Cost of Radix Sort

```
static void radix(Elem[] A, Elem[] B,
                  int k, int r, int[] count) {
  // Count[i] stores number of records in bin[i]
  int i, j, rtok;

  for (i=0, rtok=1; i<k; i++, rtok*=r) {// For k digits
    for (j=0; j<r; j++) count[j] = 0;   // Initialize

    // Count number of recs for each bin on this pass
    for (j=0; j<A.length; j++)
      count[(A[j].key()/rtok)%r]++;

    // Index B: count[j] is index for last slot of j.
    for (j=1; j<r; j++) count[j] = count[j-1]+count[j];

    // Put recs in bins, work from bottom of each bin.
    // Since bins fill from bottom, j counts downwards
    for (j=A.length-1; j>=0; j--)
      B[--count[(A[j].key()/rtok)%r]] = A[j];

    for (j=0; j<A.length; j++) A[j] = B[j]; // Copy
  }
}
```

Cost: $\Theta(nk + rk)$.

How do n, k and r relate?

Radix Sort Example

Initial Input: Array A

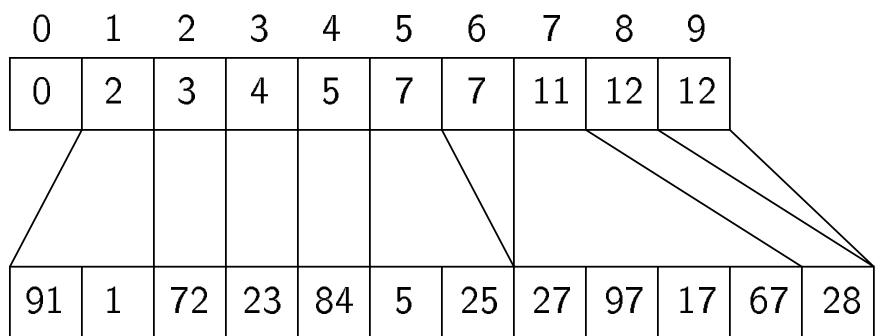
27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count.
`rtok` = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

End of Pass 1: Array A.

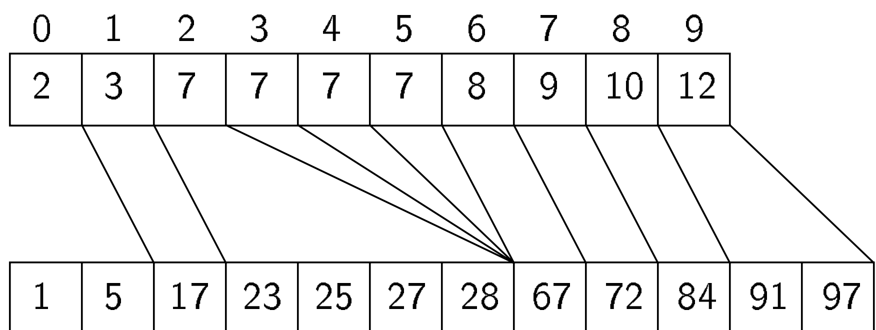


Second pass values for Count.
`rtok` = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

End of Pass 2: Array A.



Empirical Comparison

Algorithm	10	100	1000	10,000
Insert. Sort	.10	9.5	957.9	98,086
Bubble Sort	.13	14.3	1470.3	157,230
Select. Sort	.11	9.9	1018.9	104,897
Shellsort	.09	2.5	45.6	829
Quicksort	.15	1.8	23.6	291
Quicksort/O	.10	1.6	20.9	274
Mergesort	.12	2.4	36.8	505
Mergesort/O	.08	1.8	28.0	390
Heapsort	–	50.0	60.0	880
Radix Sort/1	.87	8.6	89.5	939
Radix Sort/4	.23	2.3	22.5	236
Radix Sort/8	.19	1.2	11.5	115

Algorithm	10	100	1000	10,000
Insert. Sort	.66	65.9	6423	661,711
Bubble Sort	.90	85.5	8447	1,068,268
Select. Sort	.73	67.4	6678	668,056
Shellsort	.62	18.5	321	5,593
Quicksort	.92	12.7	169	1,836
Quicksort/O	.65	10.7	141	1,781
Mergesort	.76	16.8	234	3,231
Mergesort/O	.53	11.8	189	2,649
Heapsort	–	41.0	565	7,973
Radix Sort/1	7.40	67.4	679	6,895
Radix Sort/4	2.10	18.7	160	1,678
Radix Sort/8	4.10	11.5	97	808

Sorting Lower Bound

Want to prove a lower bound for *all possible* sorting algorithms.

Sorting is O($n \log n$).

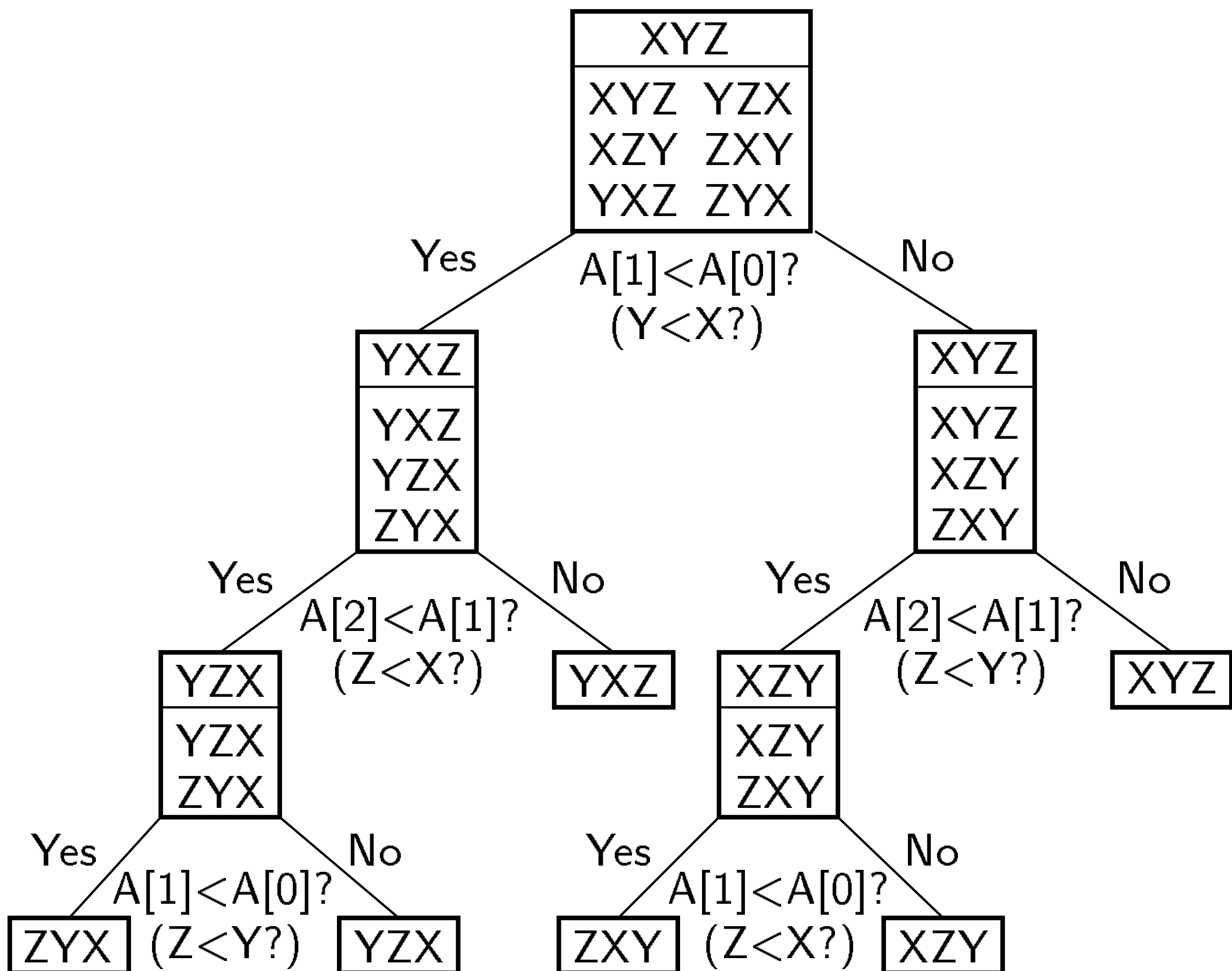
Sorting I/O takes $\Omega(n)$ time.

Will now prove $\Omega(n \log n)$ lower bound.

Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves.
- The tree must be $\Omega(n \log n)$ levels deep.

Decision Trees



There are $n!$ permutations, and at least 1 node for each permutation.

A tree with n nodes has at least $\log n$ levels.

Where is the worst case in the decision tree?

$\log n! = \Omega(n \log n)$.