Data Structures
and Algorithms
Course’s slides: Sorting Algorithms
www.mif.vu.lt/~algis
Card players all know how to sort …

First card is already sorted
With all the rest

1. Scan back from the end until you find the first card larger than the new one,
2. Move all the lower ones up one slot
3. Insert it
Sorting terminology and notation

- Records are compared to one another by means of a comparator class.

- The key method of the comparator class is `prior`, which returns true when its first argument should appear prior to its second argument in the sorted list.

- For every record type there is a `swap` function that can interchange the contents of two records in the array.

- Given a set of records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$, the Sorting Problem is to arrange the records into any order $s$ such that records $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n}$.
Sorting terminology and notation

**Time analysis:** some algorithms are much more efficient than others.

For sorting algorithms, we’ll focus on two types of operations: **comparisons and moves (swaps)**

To express the time complexity of an algorithm, we’ll express the number of operations performed as a function of $n$

- $C(n) = \text{number of comparisons, } M(n) = \text{number of moves}$

Examples: $C(n) = n^2 + 3 \ n, \ M(n) = 2 \ n^2 - 1$
Sorting terminology and notation

When \( n \) is large, expressions of \( n \) are dominated by their “largest” term – the term that grows fastest as a function of \( n \).

In characterizing the time complexity of an algorithm, we’ll focus on the largest term in its operation-count expression.

• for selection sort, \( C(n) = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2} \)

In addition, we’ll typically ignore the coefficient of the largest term (e.g., \( \frac{n^2}{2} \rightarrow n^2 \)).
Mathematical definition of Big-O notation:

\[ f(n) = O(g(n)) \] if there exist positive constants \( c \) and \( n_0 \), such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

Example: \( f(n) = \frac{n^2}{2} - \frac{n}{2} \) is \( O(n^2) \), because
\[ \frac{n^2}{2} - \frac{n}{2} \leq n^2 \] for all \( n \geq 0 \).
## Sorting terminology and notation

### Common classes of algorithms:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example expressions</th>
<th>Big-O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant time</td>
<td>1, 7, 10</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Logarithmic time</td>
<td>$3 \log_{10} n$, $\log_2 n + 5$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Linear time</td>
<td>$5n$, $10n - 2 \log_2 n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>n logn time</td>
<td>$4n \log_2 n$, $n \log_2 n + n$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quadratic time</td>
<td>$2n^2 + 3n$, $n^2 - 1$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Exponential time</td>
<td>$2^n$, $5e^n + 2n^2$</td>
<td>$O(c^n)$</td>
</tr>
</tbody>
</table>

For large inputs, efficiency matters more than CPU speed. e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine.
**Sorting terminology and notation**

*big-theta* notation (Θ) is used to specify a tight bound:

\[ f(n) = \Theta(g(n)) \]

if there exist constants \( c_1, c_2, \) and \( n_0 \) such that

\[ c_1 g(n) \leq f(n) \leq c_2 g(n) \]

for all \( n > n_0 \)

*Example:* \( f(n) = n^2/2 - n/2 \) is \( \Theta(n^2) \), because

\[ (1/4) * n^2 \leq n^2/2 - n/2 \leq n^2 \]

for all \( n \geq 2 \)

---

**Big-Theta Notation**

- In theoretical computer science, *big-theta* notation is used to specify a tight bound.

- \( f(n) = \Theta(g(n)) \) if there exist constants \( c_1, c_2, \) and \( n_0 \) such that

\[ c_1 g(n) \leq f(n) \leq c_2 g(n) \]

for all \( n > n_0 \)

- Example: \( f(n) = n^2/2 - n/2 \) is \( \Theta(n^2) \), because

\[ (1/4) * n^2 \leq n^2/2 - n/2 \leq n^2 \]

for all \( n \geq 2 \)

---

**Time Analysis of Selection Sort**

- **Comparisons:** we showed that

\[ C(n) = n^2/2 - n/2 \]

- **Selection sort** performs \( O(n^2) \) comparisons

- **Moves:** after each of the \( n-1 \) passes to find the smallest remaining element, the algorithm performs a swap to put the element in place.

- \( n-1 \) swaps, \( 3 \) moves per swap

- \( M(n) = 3(n-1) = 3n - 3 \)

- **Selection sort** performs \( O(n) \) moves.
Selection sorting works according to the prescript:

- first find the smallest element in the array and exchange it with the element in the first position,
- then find the second smallest element and exchange it with the element in the second position, etc.
### Sorting: Selection Sort

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>20</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>17</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>28</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>17</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

Figure 7.3: An example of Selection Sort. Each column shows the array after the iteration with the indicated value of \( i \) in the outer for loop. Numbers above the line in each column have been sorted and are in their final positions.

Figure 7.4: An example of swapping pointers to records. (a) A series of four records. The record with key value 42 comes before the record with key value 5. (b) The four records after the top two pointers have been swapped. Now the record with key value 5 comes before the record with key value 42.

In Selection Sort, the number of comparisons is \( \Theta(n^2) \), but the number of swaps is much less than that required by bubble sort. Selection Sort is particularly advantageous when the cost to do a swap is high, for example, when the elements are long strings or other large records. Selection Sort is more efficient than Bubble Sort (by a constant factor) in most other situations as well.

There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move. This technique is illustrated by Figure 7.4. Additional space is needed to store the pointers, but the return is a faster swap operation.
Sorting: Selection Sort

The following program is a full implementation of this process:

```pascal
procedure selection; var i, j, min, t: integer;
begin
  for i := 1 to N -1 do
    begin
      min := i;
      for j := i+1 to N do
        if a[j] < a[min] then min := j; t := a[min]; a[min] := a[i]; a[i] := t
    end;
end;
```

**Complexity:** $O(n^2)$ - uses about $N^2/2$ comparisons and $N$ exchanges
Sorting: Selection Sort

To sort $n$ elements, selection sort performs $n - 1$ passes:

- on 1\textsuperscript{st} pass, it performs $n - 1$ comparisons to find indexSmallest,
- on 2\textsuperscript{nd} pass, it performs $n - 2$ comparisons .......
- on the $(n-1)$\textsuperscript{st} pass, it performs 1 comparison

adding up the comparisons for each pass, we get:

$$C(n) = 1 + 2 + ... + (n - 2) + (n - 1) = \frac{(n - 1)n}{2}$$
Sorting: Selection Sort

**Comparisons:** $C(n) = \frac{n^2}{2} - \frac{n}{2}$, so selection sort performs $O(n^2)$ comparisons.

**Moves:** after each of the $n-1$ passes to find the smallest remaining element, the algorithm performs a swap to put the element in place: $n-1$ swaps, $3$ moves per swap:

$M(n) = 3(n-1) = 3n - 3$,

selection sort performs $O(n)$ moves.
Sorting – Insertion sort

We start with insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Figure 2.1: Sorting a hand of cards using insertion sort.

Our pseudocode for insertion sort is presented as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1 \ldots n]$ containing a sequence of length $n$ that is to be sorted. (In the code, the number $n$ of elements in $A$ is denoted by $\text{length}[A].$) The input numbers are sorted in place: the numbers are rearranged within the array $A$, with at most a constant number of them stored outside the array at any time. The input array $A$ contains the sorted output sequence when INSERTION-SORT is finished.

```
INSERTION-SORT(A)
1 for j <-> 2 to length[A]
2 do key <-> A[j]
3 Insert A[j] into the sorted sequence A[1 ... j - 1].
4 i <-> j - 1
5 while i > 0 and A[i] > key
6 do A[i + 1] <-> A[i]
7 i <-> i - 1
8 A[i + 1] <-> key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = 5, 2, 4, 6, 1, 3$. The index $j$ indicates the “current card” being inserted into the hand. At the beginning of each iteration of the “outer” for loop, which is indexed by $j$, the subarray consisting of elements $A[1 \ldots j - 1]$ constitutes the currently sorted hand, and elements $A[j + 1 \ldots n]$ correspond to the pile of cards still on the table. In fact, elements $A[1 \ldots j - 1]$ are the elements originally in positions 1 through $j - 1$, but now in sorted order. We state these properties of $A[1 \ldots j - 1]$ formally as a loop invariant:
Sorting – Insertion sort

- Insertion sort:
  - Input: A sequence of \( n \) numbers \( a_1, a_2, \ldots, a_n \).
  - Output: A permutation (reordering) of the input sequence such that the numbers that we wish to sort are also known as the keys.

- Insertion sort is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand.
### Sorting – Insertion sort

<table>
<thead>
<tr>
<th>i=1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>17</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>42</td>
<td>20</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>42</td>
<td>20</td>
<td>20</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>42</td>
<td>28</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>42</td>
<td>28</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>42</td>
<td>28</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
Sorting – Insertion sort

Insertion sort:

procedure insertion;

var i,j,v:integer;

begin

for i:=2 to N do 

begin

v:=a[i]; j:=i;

while a[j-1]>v do 

begin

a[j]:=a[j-1]; j:=j-1 

end;

a[j]:=v

end
Insertion sort

Complexity:

For each card: Scan - $O(n)$, Shift up - $O(n)$, Insert - $O(1)$, Total - $O(n)$

First card requires $O(1)$, second $O(2)$, … For $n$ card - $O(n^2)$

- selection sort uses about $N^2/2$ comparisons and $N$ exchanges;
- insertion sort uses $N^2/4$ comparisons and $N^2/8$ exchanges on the average, twice as many in the worst case;
- bubble sort uses $N^2/2$ comparisons and $N^2/2$ exchanges on the average and in the worst case
Shellsort or Shell's method, is an in-place comparison sort, it is a generalization of insertion sort. The method starts by sorting elements far apart from each other and progressively reducing the gap between them.

The list of elements considering every $h$th element gives a sorted list. Such a list is said to be $h$-sorted.

If the file is then $k$-sorted for some smaller integer $k$, then the file remains $h$-sorted. Following this idea for a decreasing sequence of $h$ values ending in 1 is guaranteed to leave a sorted list in the end.
Sorting – Shellsort

An example run of Shellsort with gaps 7, 3 and 1:

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

3 7 9 0 5 1 6
3 7 9 0 5 1 6
8 4 2 0 6 1 5 -> 7 4 4 0 6 1 6
7 3 4 9 8 2
7 3 4 9 8 2
The new data: 3 3 2 0 5 1 5 7 4 4 0 6 1 6 8 7 9 9 8 2

are devided into columns by 3 elements
Sorting – Shellsort

Different sequences of decreasing increments can be used.

The version uses values: $2^k - 1$ for some $k$: 63, 31, 15, 7, 3, 1

get to the next lower increment using integer division: $\text{incr} = \text{incr}/2$

Should avoid numbers that are multiples of each other.
Otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes.

Example of a bad sequence: 64, 32, 16, 8, 4, 2, 1

What happens if the largest values are all in odd positions?
Sorting – Shellsort

Difficult to analyze precisely, typically use experiments. With a bad interval sequence, it’s $O(n^2)$ in the worst case. With a good interval sequence, it’s better than $O(n^2)$. At least $O(n^{1.5})$ in the average and worst case. Some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$.

Significantly better than insertion or selection for large $n$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n^{1.5}$</th>
<th>$n^{1.25}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>31.6</td>
<td>17.8</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>1000</td>
<td>316</td>
</tr>
<tr>
<td>10,000</td>
<td>100,000,000</td>
<td>1,000,000</td>
<td>100,000</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$10^{12}$</td>
<td>$10^9$</td>
<td>3.16 x $10^7$</td>
</tr>
</tbody>
</table>
Sorting – Shellsort

Algorithm:

\[ h = 1 \]
while \( h < n \), \( h = 3h + 1 \)
while \( h > 0 \), 
\[ h = h / 3 \]
for \( k = 1 : h \), insertion sort \( a[ k : h : n] \)

→ invariant: each \( h \)-sub-array is sorted

end

- Not stable
- \( O(1) \) extra space
- \( O(n^{3/2}) \) time, depend on sequence \( h \) (neither tight upper bounds nor the best increment sequence are known)
- Adaptive: \( O(n \cdot \lg(n)) \) time when nearly sorted
Perform a sequence of passes through the array. On each pass: proceed from left to right, swapping adjacent elements if they are out of order.

Larger elements “bubble up” to the end of the array.

At the end of the $k^{th}$ pass, the $k$ rightmost elements are in their final positions, so we don’t need to consider them in subsequent passes.

Repeat from the first to $n-1$. Stop when you have only one element to check.
### Sorting – Bubble sort

#### 7.2.2 Bubble Sort

Our next sort is called **Bubble Sort**. Bubble Sort is often taught to novice programmers in introductory computer science courses. This is unfortunate, because Bubble Sort has no redeeming features whatsoever. It is a relatively slow sort, it is no easier to understand than Insertion Sort, it does not correspond to any intuitive counterpart in “everyday” use, and it has a poor best-case running time. However, Bubble Sort serves as the basis for a better sort that will be presented in Section 7.2.3.

Bubble Sort consists of a simple double for loop. The first iteration of the inner for loop moves through the record array from bottom to top, comparing adjacent keys. If the lower-indexed key’s value is greater than its higher-indexed neighbor, then the two values are swapped. Once the smallest value is encountered, this process will cause it to “bubble” up to the top of the array. The second pass through the array repeats this process. However, because we know that the smallest value reached the top of the array on the first pass, there is no need to compare the top two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. Figure 7.2 illustrates Bubble Sort. A Java implementation is as follows:

```java
static <E extends Comparable<? super E>> void Sort(E[] A) {
    for (int i=0; i<A.length-1; i++) // Bubble up i'th record
        for (int j=A.length-1; j>i; j--)
            if ((A[j].compareTo(A[j-1]) < 0))
                DSutil.swap(A, j, j-1);
}
```

#### Figure 7.2

An illustration of Bubble Sort. Each column shows the array after the iteration with the indicated value of i in the outer for loop. Values above the line in each column have been sorted. Arrows indicate the swaps that take place during a given iteration.
#define swap (a,b)   { int t; t=a; a=b; b=t; }

void bubble ( int a[], int n ) {
    int i, j;
    for (i=0; i<n; i++) { /* n passes thru the array */
        /* From start to the end of unsorted part */
        for (j=1; j<(n-i); j++) {
            /* If adjacent items out of order, swap */
            if ( a[j-1]>a[j] ) swap (a[j-1],a[j]);
        }
    }
}
Sorting - Bubble Sort

\[ \sum_{i=1}^{n} i = \Theta(n^2). \]

Bubble Sort’s running time is roughly the same in the best, average, and worst cases.

The number of swaps required depends on how often a value is less than the one immediately preceding it in the array.

We can expect this to occur for about half the comparisons in the average case, leading to \( \Theta(n^2) \) for the expected number of swaps.
A very special situation for which there is a simple sorting algorithm: "sort a file of $N$ records whose keys are distinct integers between 1 and $N$" or

sort a file of $N$ records whose keys are integers between 0 and $M – 1$.

If $M$ is not too large, an algorithm called distribution counting can be used to solve this problem. The idea is to count the number of keys with each value and then use the counts to move the records into position on a second pass through the file.

This method will work very well for the type of files postulated.
The Divide and Conquer Approach

The most well known algorithm design strategy:

1. **Divide** the problem into two or more smaller subproblems.

2. **Conquer** the subproblems by solving them recursively.

3. **Combine** the solutions to the subproblems into the solutions for the original problem.
Sorting – Quicksort

- Efficient sorting algorithm
  - Discovered by C.A.R. Hoare
  - Example of “Divide and Conquer” algorithm
- Two phases
  - Partition phase
    - Divides the work into half
  - Sort phase
    - Conquers the halves!
Sorting – Quicksort

Partition

- Choose a pivot
- Find the position for the pivot so that
  - all elements to the left are less
  - all elements to the right are greater

< pivot  pivot  > pivot
Sorting – Quicksort

- Conquer
- Apply the same algorithm to each half

\[ < \text{pivot} \]

\[ < p' \quad p' \quad > p' \]

\[ \text{pivot} \]

\[ < p'' \quad p'' \quad > p'' \]

It works by partitioning a file into two parts, then sorting the parts independently. The exact position of the partition depends on the file, and the algorithm has the following recursive structure:

If the partition method can be made more precise, (for example, choosing right-most element as a partition element), and the recursive call of this procedure may be eliminated, then the implementation of the algorithm looks like this:

```plaintext
procedure quicksort(l, r: integer);

var
    v, t, i, j: integer;

begin
    if r > l then
        begin
            v := a[r]; i := l - 1; j := r;
            repeat
                i := i + 1
            until a[i] >= v;
            repeat
                j := j - 1
            until a[j] < v;
            t := a[i]; a[i] := a[j]; a[j] := t;
            until j <= i;
            a[j] := a[i]; a[i] := a[r]; a[r] := t;
            quicksort(l, i - 1);
            quicksort(i + 1, r)
        end;
end;
```

An example of a partitioning of a larger file (choosing the right-most element):

The crux of the method is the partition procedure, which must rearrange the array to make the following conditions hold:

- The element \( a[i] \) is in its final place in the array for some \( i \);
- All the elements left to the \( a[i] \) are less than or equal to it;
- All the elements right to the \( a[i] \) are greater than or equal to it.

The first improvement: Removing Recursion

The recursion may be removed by using an explicit pushdown stack, which is containing "work to be done" in the form of subfiles to be sorted. Any time we needed a subfile to process, we pop the stack. When we partition, we create two subfiles to be processed which can be pushed on the stack. This leads to the following nonrecursive implementation:

```plaintext
procedure quicksort;

var
    t, i, 1, r: integer

begin
    l := 1; r := N; stackinit;
    push (l); push (r);
```

An example of a partitioning of a larger file (choosing the right-most element):
Quicksort – Partition

```c
int partition( int *a, int low, int high ) {
    int left, right;
    int pivot_item;
    pivot_item = a[low];
    pivot = left = low;
    right = high;
    while ( left < right ) {
        /* Move left while item < pivot */
        while( a[left] <= pivot_item ) left++;
        /* Move right while item > pivot */
        while( a[right] >= pivot_item ) right--;
        if ( left < right ) SWAP(a,left,right);
    }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```
Quicksort – Analysis

- **Partition**
  - Check every item once $O(n)$
- **Conquer**
  - Divide data in half $O(\log_2 n)$
- **Total**
  - Product $O(n \log n)$
- Quicksort is *generally* faster
  - Fewer comparisons
Quicksort – The truth!

- What happens if we use quicksort on data that’s already sorted (or nearly sorted)

- We’d certainly expect it to perform well!
Quicksort – The truth!

- Sorted data

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

- pivot

- < pivot

- > pivot
Quicksort – The truth!

- Sorted data
- Each partition produces
  - a problem of size 0
  - and one of size \( n-1 \)!
- Number of partitions?
Quicksort – The truth!

- Sorted data
- Each partition produces
  - a problem of size 0
  - and one of size $n-1$!
- Number of partitions?
  - $n$ each needing time $O(n)$
  - Total $n O(n)$
    or $O(n^2)$
Quicksort – The truth!

- Quicksort’s $O(n \log n)$ behaviour
  - Depends on the partitions being nearly equal
  - there are $O(\log n)$ of them
  - On average, this will nearly be the case

  _and quicksort is generally $O(n \log n)$_

- Can we do anything to ensure $O(n \log n)$ time?
  - In general, no
  - But we can improve our chances!!
Quicksort – Choice of the pivot

- Any pivot will work …
- Choose a different pivot …

- so that the partitions are equal
- *then* we will see $O(n \log n)$ time
Quicksort – Median-of-3 pivot

✦ Take 3 positions and choose the median
✦ say … First, middle, last
✦ median is 5
✦ perfect division of sorted data every time!
✦ $O(n \log n)$ time
✦ Since sorted (or nearly sorted) data is common, median-of-3 is a good strategy
✦ especially if you think your data may be sorted!
Quicksort – Random pivot

- Choose a pivot randomly
  - Different position for every partition
    - *On average*, sorted data is divided evenly
    - $O(n \log n)$ time

- Key requirement
  - Pivot choice must take $O(1)$ time
Mergesort

- The *merge sort* algorithm is closely following the divide-and-conquer paradigm. Intuitively, it operates as follows:
  - **Divide:** Divide the *n-element* sequence to be sorted into two subsequences of \((n/2)\) elements each.
  - **Conquer:** Sort the two subsequences recursively using *mergesort*.
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer.
  - Suppose we have two sorted arrays \(a[1...N]\), \(b[1...M]\), and we need to merge them into one. The mergesort algorithm is based on a *merge* procedure, which can be presented as follows:
Mergesort
Mergesort

procedure merge (a, b);

var a,b,c: array [1..M + N] of integer;

begin i:=1; j:=1; a[M+1]:=maxint; b[N+1]:=maxint;

for k:=1 to M+N do

if a[i]<b[j] then

begin c[k]:=a[i]; i:=i+1 end else begin c[k]:=b[j]; j:=j+1 end;

end;
Mergesort

To perform the *mergesort*, the algorithm for arrays can be as follows:

```pascal
procedure mergesort(l, r: integer); var i, j, k, m: integer;
begin
    if r-l > 0 then
        begin
            m := (r + l) div 2;
            mergesort(l, m);
            mergesort(m + 1, r);
            for i := m downto l do b[i] := a[i];
            for j := m + 1 to r do b[r + m + 1 - j] := a[j];
            for k := l to r do
                if b[i] < b[j] then
                    begin
                        a[k] := b[i]; i := i + 1
                    end else
                    begin
                        a[k] := b[j]; j := j - 1
                    end;
        end;
end;
```
Mergesort

Property 1: Mergesort requires about $N \lg N$ comparisons to sort any file of $N$ elements.

Property 2: Mergesort is stable.

Property 3: Mergesort is insensible to the initial order of its input.
Quicksort – comparing to insert

- Quicksort is generally faster
- Fewer comparisons and exchanges
- Some empirical data

<table>
<thead>
<tr>
<th>n</th>
<th>Quick Comp</th>
<th>Exch</th>
<th>Heap Comp</th>
<th>Exch</th>
<th>Insert Comp</th>
<th>Exch</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>712</td>
<td>148</td>
<td>2842</td>
<td>581</td>
<td>2595</td>
<td>899</td>
</tr>
<tr>
<td>200</td>
<td>1682</td>
<td>328</td>
<td>9736</td>
<td>9736</td>
<td>10307</td>
<td>3503</td>
</tr>
<tr>
<td>500</td>
<td>5102</td>
<td>919</td>
<td>53113</td>
<td>4042</td>
<td>62746</td>
<td>21083</td>
</tr>
</tbody>
</table>
Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>algorithm</th>
<th>best case</th>
<th>avg case</th>
<th>worst case</th>
<th>extra memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>algorithm</th>
<th>best case</th>
<th>avg case</th>
<th>worst case</th>
<th>extra memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
External Sorting and Basic Algorithm

Problem: If a list is too large to fit in main memory, the time required to access a data value on a disk or tape dominates any efficiency analysis.

1 disk access ≡ Several million machine instructions

Solution: Develop external sorting algorithms that minimize disk accesses
A Typical Disk Drive
Disk Access

Disk Access Time =

Seek Time (moving disk head to correct track)

+ Rotational Delay (rotating disk to correct block in track)

+ Transfer Time (time to transfer block of data to main memory)

Assume unsorted data is on disk at start

Let M = maximum number of records that can be stored & sorted in internal memory at one time
Basic External Sorting Algorithm

Algorithm

Repeat:
1. Read M records into main memory & sort internally.
2. Write this sorted sub-list onto disk, until all data are processed into runs

Repeat:
1. Merge two runs into one sorted run twice as long
2. Write this single run back onto disk, until all runs processed into runs twice as long

Merge runs again as often as needed until only one large run: the sorted list
Basic External Sorting

Unsorted Data on Disk

Assume $M = 3$ (M would actually be much larger, of course.) First step is to read 3 data items at a time into main memory, sort them and write them back to disk as runs of length 3.
Basic External Sorting

Next step is to merge the runs of length 3 into runs of length 6.

11 81 94 17 28 99 15
12 35 96 41 58 75
11 12 35 81 94 96
17 28 41 58 75 99
15
Basic External Sorting

Next step is to merge the runs of length 6 into runs of length 12.

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>35</th>
<th>81</th>
<th>94</th>
<th>96</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>28</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

11 12 17 28 35 41 58 75 81 94 96 99

15
Basic External Sorting

Next step is to merge the runs of length 12 into runs of length 24. Here we have less than 24, so we’re finished.
Streaming Data through RAM

- An important method for sorting & other DB operations
- Simple case:
  - Compute $f(x)$ for each record, write out the result
  - Read a page from INPUT to Input Buffer
  - Write $f(x)$ for each item to Output Buffer
  - When Input Buffer is consumed, read another page
  - When Output Buffer fills, write it to OUTPUT
- Reads and Writes are not coordinated
  - E.g., if $f()$ is Compress(), you read many pages per write.
  - E.g., if $f()$ is DeCompress(), you write many pages per read.
2-Way Sort: Requires 3 Buffers

- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 1, 2, 3, …, etc.:
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.
Two-Way External Merge Sort

Each pass we read + write each page in file.

N pages in the file => the number of passes

\[ = \lceil \log_2 N \rceil + 1 \]

So total cost is:

\[ 2^N \left( \lceil \log_2 N \rceil + 1 \right) \]

Idea: Divide and conquer: sort subfiles and merge
General External Merge Sort

To sort a file with $N$ pages using $B$ buffer pages:

- **Pass 0**: use $B$ buffer pages. Produce $\lceil N / B \rceil$ sorted runs of $B$ pages each.
- **Pass 1, 2, …, etc.**: merge $B-1$ runs.
Cost of External Merge Sort

- Number of passes: \(1 + \lceil \log_{B-1} \left\lfloor \frac{N}{B} \right\rfloor \rceil\)
- Cost = \(2N \times (\text{# of passes})\)

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: \(\left\lfloor \frac{108}{5} \right\rfloor = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\left\lfloor \frac{22}{4} \right\rfloor = 6\) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: \(\lceil \log_4 22 \rceil = 3 \ldots + 1 \Rightarrow 4 \text{ passes} \checkmark\)
# Number of Passes of External Sort

(I/O cost is 2N times number of passes)

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
I/O for External Merge Sort

- Do I/O a page at a time
  - Not one I/O per record
  - In fact, read a *block* (chunk) of pages sequentially!
- Suggests we should make each buffer (input/output) be a *block* of pages.
  - But this will reduce fan-in during merge passes!
  - In practice, most files still sorted in 2-3 passes.
Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block`.

- Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.
To begin, we'll trace through the various steps of the simplest sort-merge procedure for a small example. Suppose that we have records with the keys

ASORTINGANDMERGINGEXAMPLE

on an input tape; these are to be sorted and put onto an output tape. Using a "tape" simply means that we're restricted to reading the records sequentially: the second record can't be read until the first is read, and so on. Assume further that we have only enough room for three records in our computer memory but that we have plenty of tapes available.
Balanced multiway merging

- The first step is to read in the file three records at a time, sort them to make three-record blocks, and output the sorted blocks. Thus, first we read in A S 0 and output the block A 0 S, next we read in R T I and output the block I R T, and so forth. Now, in order for these blocks to be merged together, they must be on different tapes. If we want to do a three-way merge, then we would use three tapes.

<table>
<thead>
<tr>
<th>Tape 1</th>
<th>Tape 2</th>
<th>Tape 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOS</td>
<td>DMN</td>
<td>AEX</td>
</tr>
<tr>
<td>IRT</td>
<td>EGR</td>
<td>LMP</td>
</tr>
<tr>
<td>AGN</td>
<td>GIN</td>
<td>E</td>
</tr>
</tbody>
</table>

Now we're ready to merge the sorted blocks of size three. We read the first record off each input tape (there's just enough room in the memory) and output the one with the smallest key. Then the next record from the same tape as the record just output is read in and, again, the record in memory with the smallest key is output. When the end of a three-word block in the input is encountered, that tape is ignored until the blocks from the other two tapes have been processed and nine records have been output. Then the process is repeated to merge the second three-word block on each tape into a nine-word block (which is output on a different tape, to get ready for the next merge). By continuing in this way, we get three long blocks configured as shown in figure next:

Now one more three-way merge completes the sort. If we had a much longer file with many blocks of size 9 on each tape, then we would finish the second pass with blocks of size 27 on tapes 1, 2, and 3, then a third pass would produce blocks of size 81 on tapes 4, 5, and 6, and so forth. We need six tapes to sort an arbitrarily large file: three for the input and three for the output of each three-way merge. (Actually, we could get by with just four tapes: the output could be put on just one tape, and then the blocks from that tape distributed to the three input tapes in between merging passes.)
Balanced multiway merging

- We read the first record off each input tape (there's just enough room in the memory) and output the one with the smallest key. Then the next record from the same tape as the record just output is read in and, again, the record in memory with the smallest key is output. When the end of a three-word block in the input is encountered, that tape is ignored until the blocks from the other two tapes have been processed and nine records have been output. Then the process is repeated to merge the second three-word block on each tape into a nine-word block (which is output on a different tape, to get ready for the next merge).

| Tape 1 | □ |
| Tape 2 | □ |
| Tape 3 | □ |
| Tape 4 | □ AAGINORST |
| Tape 5 | □ DEGGIMNNR |
| Tape 6 | □ AEELMPX |
Now one more three-way merge completes the sort. If we had a much longer file with many blocks of size 9 on each tape, then we would finish the second pass with blocks of size 27 on tapes 1, 2, and 3, then a third pass would produce blocks of size 81 on tapes 4, 5, and 6, and so forth. We need six tapes to sort an arbitrarily large file: three for the input and three for the output of each three-way merge.

Actually, we could get by with just four tapes: the output could be put on just one tape, and then the blocks from that tape distributed to the three input tapes in between merging passes.

This method is called the balanced multiway merge: it is a reasonable algorithm for external sorting and a good starting point for the implementation of an external sort.
Replacement selection

- It turns out that the details of the implementation can be developed in an elegant and efficient way using priority queues. First, we'll see that priority queues provide a natural way to implement a multiway merge. More important, it turns out that we can use priority queues for the initial sorting pass in such a way that they can produce sorted blocks much longer than could fit into internal memory.

- The basic operation needed to do P-way merging is repeatedly to output the smallest of the smallest elements not yet output from each of the P blocks to be merged. That smallest element should be replaced with the next element from the block from which it came.

- The *replace* operation on a priority queue of size P is exactly what is needed.
Replacement selection

- The process of merging A0S with IRT and AGN (the first merge from our example above), using a heap of size three in the merging process is shown:
In summary, the replacement selection technique can be used for both the "sort" and the "merge" steps of a balanced multiway merge.

Property: A file of $N$ records can be sorted using an internal memory capable of holding $M$ records and $(P + 1)$ tapes in about $1 + \log P \left(\frac{N}{2M}\right)$ passes.

We first use replacement selection with a priority queue of size $M$ to produce initial runs of size about $2M$ (in a random situation) or more (if the file is partially ordered), then use replacement selection with a priority queue of size $P$ for about $\log P \left(\frac{N}{2M}\right)$ (or fewer) merge passes.
One problem with balanced multiway merging for tape sorting is that it requires either an excessive number of tape units or excessive copying. For P-way merging either we must use 2P tapes (P for input and P for output) or we must copy almost all of the file from a single output tape to P input tapes between merging passes, which effectively doubles the number of passes to be about $2 \log P \left( \frac{N}{2M} \right)$. Several clever tape-sorting algorithms have been invented which eliminate virtually all of this copying by changing the way in which the small sorted blocks are merged together. The most prominent of these methods is called polyphase merging.

The basic idea behind polyphase merging is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (leaving one empty) and then to apply a "merge-until-empty" strategy, at which point one of the output tapes and the input tape switch roles.
Polyphase merging

- For example, suppose that we have just three tapes, and we start out with the initial configuration of sorted blocks on the tapes shown at the top of figure:
Polyphase merging

This merge-until-empty strategy can be extended to work for an arbitrary number of tapes. Table below shows how six tapes might be used to sort 497 initial runs:

<table>
<thead>
<tr>
<th>Tape 1</th>
<th>61</th>
<th>0</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>3</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tape 2</td>
<td>0</td>
<td>61</td>
<td>30</td>
<td>14</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tape 3</td>
<td>120</td>
<td>59</td>
<td>28</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tape 4</td>
<td>116</td>
<td>55</td>
<td>24</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tape 5</td>
<td>108</td>
<td>47</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tape 6</td>
<td>92</td>
<td>31</td>
<td>0</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Polyphase merging

- The main difficulty in implementing a polyphase merge is to determine how to distribute the initial runs. It is not difficult to see how to build the table above by working backwards:
  - take the largest number in each column, make it zero, and add it to each of the other numbers to get the previous column.
  - This corresponds to defining the highest-order merge for the previous column which could give the present column. This technique works for any number of tapes (at least three):
    - the numbers which arise are "generalized Fibonacci numbers" which have many interesting properties.
External sorting is important

External merge sort minimizes disk I/O cost:

- Pass 0: Produces sorted runs of size $B$ (# buffer pages). Later passes: merge runs.
- # of runs merged at a time depends on $B$, and block size.
- Larger block size means less I/O cost per page.
- Larger block size means smaller # runs merged.
- In practice, # of runs rarely more than 2 or 3.
Choice of internal sort algorithm may matter:

- Quicksort: Quick!
- Heap/tournament sort: slower (2x), longer runs

The best sorts are wildly fast:

- Despite 40+ years of research, still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.