Paper folding myth
• no paper can be folded more than 7 times
• WHY?

Repeated Minimum: Code
for i := 1 to n-1 do
  for j := i+1 to n do
    if L[i] > L[j] then
      Temp := L[i];
      L[i] := L[j];
      L[j] := Temp;
    endif
  endfor
endfor

Repeated Minimum: Analysis
Doing it the dumb way:
\[ \sum_{i=1}^{n-1} (n - i) \]
The smart way: I do one comparison when i=n-1, two when i=n-2, …, n-1 when i=1.
\[ \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \in \Theta(n^2) \]

Sorting
Practice with Analysis

Repeated Minimum
• Search the list for the minimum element.
• Place the minimum element in the first position.
• Repeat for other n-1 keys.
• Use current position to hold current minimum to avoid large-scale movement of keys.

Bubble Sort
• Search for adjacent pairs that are out of order.
• Switch the out-of-order keys.
• Repeat this n-1 times.
• After the first iteration, the last key is guaranteed to be the largest.
• If no switches are done in an iteration, we can stop.
**Bubble Sort: Code**

```plaintext
for i := 1 to n-1 do  
  Switch := False;
  for j := 1 to n-i do  
    if L[j] > L[j+1] then
      Temp = L[j];
      L[j] := L[j+1];
      L[j+1] := Temp;
      Switch := True;
    endif
  endfor
if Not Switch then break;
endfor
```

**Bubble Sort Analysis**

Being smart right from the beginning:

\[ \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2) \]

**Insertion Sort I**

- The list is assumed to be broken into a sorted portion and an unsorted portion
- Keys will be inserted from the unsorted portion into the sorted portion.

**Insertion Sort II**

- For each new key, search backward through sorted keys
- Move keys until proper position is found
- Place key in proper position

**Insertion Sort: Code**

```plaintext
for i := 2 to n do  
  x := L[i];
  j := i-1;
  while j>=0 and x < L[j] do
    L[j+1] := L[j];
    j := j-1;
  endwhile
  L[j+1] := x;
endfor
```

**Insertion Sort: Analysis**

- Worst Case: Keys are in reverse order
- Do i-1 comparisons for each new key, where i runs from 2 to n.
- Total Comparisons: \(1 + 2 + 3 + \ldots + n-1\)

\[ \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2) \]
Insertion Sort: Average I
- Assume: When a key is moved by the While loop, all positions are equally likely.
- There are i positions (i is loop variable of for loop) (Probability of each: 1/i.)
- One comparison is needed to leave the key in its present position.
- Two comparisons are needed to move key over one position.

Insertion Sort Average II
- In general: k comparisons are required to move the key over k-1 positions.
- Exception: Both first and second positions require i-1 comparisons.

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>i-2</th>
<th>i-1</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i+1</td>
<td>i+1</td>
<td>i+2</td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Comparisons necessary to place key in this position.

Insertion Sort Average III
Average Comparisons to place one key
\[ \sum_{j=1}^{i-1} j + \frac{1}{i} (i-1) \]
Solving
\[ = \frac{1}{i} \sum_{j=1}^{i-1} j + \left(1 - \frac{1}{i} \right) = \frac{1}{i} \frac{i(i-1)}{2} + \frac{2}{i} - 1 = \frac{i+1}{2} - \frac{1}{i} \]

Optimality Analysis I
- To discover an optimal algorithm we need to find an upper and lower asymptotic bound for a problem.
- An algorithm gives us an upper bound.
- The worst case for sorting cannot exceed \( \Theta(n^2) \) because we have Insertion Sort that runs that fast.
- Lower bounds require mathematical arguments.

Optimality Analysis II
- Making mathematical arguments usually involves assumptions about how the problem will be solved.
- Invalidating the assumptions invalidates the lower bound.
- Sorting an array of numbers requires at least \( \Theta(n) \) time, because it would take that much time to rearrange a list that was rotated one element out of position.
Rotating One Element

Assumptions:
- Keys must be moved one at a time
- All key movements take the same amount of time
- The amount of time needed to move one key is not dependent on \( n \).

Maximum Inversions

- The total number of pairs is:
  \[
  \left( \frac{n}{2} \right) = \frac{n(n-1)}{2}
  \]
- This is the maximum number of inversions in any list.
- Exchanging adjacent pairs of keys removes at most one inversion.

Other Assumptions

- The only operation used for sorting the list is swapping two keys.
- Only adjacent keys can be swapped.
- This is true for Insertion Sort and Bubble Sort.
- Is it true for Repeated Minimum? What about if we search the remainder of the list in reverse order?

Swapping Adjacent Pairs

- The only inversion that could be removed is the (possible) one between the red and green keys.
- The relative position of the red and blue areas has not changed.
- No inversions between the red key and the blue area have been removed.
- The same is true for the red key and the orange area. The same analysis can be done for the green key.

Inversions

- Suppose we are given a list of elements \( L \), of size \( n \).
- Let \( i \) and \( j \) be chosen so \( 1 \leq i < j \leq n\).
- If \( L[i] > L[j] \) then the pair \((i, j)\) is an inversion.

Lower Bound Argument

- A sorted list has no inversions.
- A reverse-order list has the maximum number of inversions, \( \Theta(n^2) \) inversions.
- A sorting algorithm must exchange \( \Theta(n^2) \) adjacent pairs to sort a list.
- A sort algorithm that operates by exchanging adjacent pairs of keys must have a time bound of at least \( \Theta(n^2) \).
Lower Bound For Average I

- There are \( n! \) ways to rearrange a list of \( n \) elements.
- Recall that a rearrangement is called a *permutation*.
- If we reverse a rearranged list, every pair that used to be an inversion will no longer be an inversion.
- By the same token, all non-inversions become inversions.

Lower Bound For Average II

- There are \( \frac{n(n-1)}{2} \) inversions in a permutation and its reverse.
- Assuming that all \( n! \) permutations are equally likely, there are \( \frac{n(n-1)}{4} \) inversions in a permutation, on the average.
- The average performance of a “swap-adjacent-pairs” sorting algorithm will be \( \Theta(n^2) \).

Quicksort II

- Big is defined as “bigger than the pivot point”
- Little is defined as “smaller than the pivot point”
- The pivot point is chosen “at random”
- Since the list is assumed to be in random order, the first element of the list is chosen as the pivot point

Quicksort Split: Code

```plaintext
Split(First, Last)
    SplitPoint := 1;
    for i := 2 to n do
        if L[i] < L[1] then
            SplitPoint := SplitPoint + 1;
            Exchange(L[SplitPoint], L[i]);
        endif
    endfor
    Exchange(L[SplitPoint], L[1]);
    return SplitPoint;
End Split
```

Quick Sort I

- Split List into “Big” and “Little” keys
- Put the Little keys first, Big keys second
- Recursively sort the Big and Little keys

Quicksort III

- Pivot point may not be the exact median
- Finding the precise median is *hard*
- If we “get lucky”, the following recurrence applies (\( n/2 \) is approximate)

\[
Q(n) = 2Q(n/2) + n - 1 \in \Theta(n \log n)
\]
### Quicksort IV

- If the keys are in order, “Big” portion will have n-1 keys, “Small” portion will be empty.
- N-1 comparisons are done for first key
- N-2 comparisons for second key, etc.
- Result: \( \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2) \)

### QS Avg: Recurrence

\[ A(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (A(i-1) + A(n-i)) \]

\[ \sum_{i=1}^{n} (A(i-1) + A(n-i)) = (A(0) + A(n-1)) + (A(1) + A(n-2)) + \ldots + (A((n-1)-1) + A(n-(n-1)) + (A(n-1) + A(n-n)) \]

### QS Avg: Formulation

- A(0) = 0
- If the pivot appears at position i, 1\(\leq i \leq n\) then A(i-1) comparisons are done on the left hand list and A(n-i) are done on the right hand list.
- n-1 comparisons are needed to split the list

### QS Avg: Solving Recurr.

**Guess:** \( A(n) \leq an \lg n + b \quad a>0, b>0 \)

\[ A(n) = \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \]

\[ \leq \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} (ai \lg i + b) \]

\[ = \Theta(n) + \frac{2a}{n} \sum_{i=1}^{n-1} i \lg i + \frac{2b}{n} (n-1) \]

### QS Avg: Recurrence II

\[ \sum_{i=1}^{n} (A(i-1) + A(n-i)) = 2A(0) + 2 \sum_{i=1}^{n-1} A(i) \]

\[ A(n) = \Theta(n) + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \]
QS Avg: Continuing

By Integration:

\[
\sum_{i=1}^{n} i \log i \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2
\]

The Merge Algorithm

Assume we are merging lists A and B into list C.

\[
\begin{align*}
A_x &:= 1; B_x := 1; C_x := 1; \\
\text{while } A_x \leq n \text{ and } B_x \leq n \\
\text{do if } A[A_x] < B[B_x] \\
& \quad \text{then } C[C_x] := A[A_x]; \\
& \quad \text{Ax} := Ax + 1; \\
& \quad \text{else} \\
& \quad \text{C}[C_x] := B[B_x]; \\
& \quad B_x := Bx + 1; \\
& \quad \text{endif} \\
& \quad C_x := Cx + 1; \\
& \text{endwhile}
\end{align*}
\]

QS Avg: Finally

\[
\begin{align*}
A(n) &\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \frac{2b}{n} (n - 1) + \Theta(n) \\
&\leq an \log n - \frac{a}{4} n + 2b + \Theta(n) \\
&= an \log n - \frac{a}{4} n + 2b + \Theta(n) \\
&\leq an \log n + b
\end{align*}
\]

Merge Sort: Analysis

- Sorting requires no comparisons
- Merging requires \(n-1\) comparisons in the worst case, where \(n\) is the total size of both lists (\(n\) key movements are required in all cases)
- Recurrence relation:

\[
W(n) = 2 \frac{W(n / 2)}{n} + n - 1 \in \Theta(n \log n)
\]

Merge Sort

- If List has only one Element, do nothing
- Otherwise, Split List in Half
- Recursively Sort Both Lists
- Merge Sorted Lists

Merge Sort: Space

- Merging cannot be done in place
- In the simplest case, a separate list of size \(n\) is required for merging
- It is possible to reduce the size of the extra space, but it will still be \(\Theta(n)\)
Heapsort: Heaps
- Geometrically, a heap is an “almost complete” binary tree.
- Vertices must be added one level at a time from right to left.
- Leaves must be on the lowest or second lowest level.
- All vertices, except one must have either zero or two children.

Heapsort: Heaps II
- If there is a vertex with only one child, it must be a left child, and the child must be the rightmost vertex on the lowest level.
- For a given number of vertices, there is only one legal structure.

Heapsort: Heap Values
- Each vertex in a heap contains a value.
- If a vertex has children, the value in the vertex must be larger than the value in either child.
- Example:

Heapsort: Heap Properties
- The largest value is in the root.
- Any subtree of a heap is itself a heap.
- A heap can be stored in an array by indexing the vertices thus:
- The left child of vertex \( v \) has index \( 2v \) and the right child has index \( 2v+1 \).

Heapsort: FixHeap
- The FixHeap routine is applied to a heap that is geometrically correct, and has the correct key relationship everywhere except the root.
- FixHeap is applied first at the root and then iteratively to one child.
Heapsort FixHeap Code

```
FixHeap(StartVertex)
v := StartVertex;
while 2*v ≤ n do
    LargestChild := 2*v;
    if 2*v < n then
        if L[2*v] < L[2*v+1] then
            LargestChild := 2*v+1;
        endif
    endif
    if L[v] < L[LargestChild] then
        Exchange(L[v],L[LargestChild]);
        v := LargestChild
    else
        v := n;
    endif
endwhile
end FixHeap
```

- `n` is the size of the heap
- Worst case run time is $\Theta(lg n)$

Heapsort: Creating a Heap

- An arbitrary list can be turned into a heap by calling FixHeap on each non-leaf in reverse order.
- If `n` is the size of the heap, the non-leaf with the highest index has index $n/2$.
- Creating a heap is obviously $O(n \lg n)$.
- A more careful analysis would show a true time bound of $\Theta(n)$

Heap Sort: Sorting

- Turn List into a Heap
- Swap head of list with last key in heap
- Reduce heap size by one
- Call FixHeap on the root
- Repeat for all keys until list is sorted
Heap Sort: Analysis

- Creating the heap takes $\Theta(n)$ time.
- The sort portion is obviously $O(n \log n)$.
- A more careful analysis would show an exact time bound of $\Theta(n \log n)$.
- Average and worst case are the same.
- The algorithm runs in place.

Lower Bound Assumptions II

- Assume that all keys are distinct, since all sort algorithms must handle this case.
- Because there are no “tricks” that work, the only information we can get from a key comparison is:
  - Which key is larger.

A Better Lower Bound

- The $\Theta(n^2)$ time bound does not apply to Quicksort, Mergesort, and Heapsort.
- A better assumption is that keys can be moved an arbitrary distance.
- However, we can still assume that the number of key-to-key comparisons is proportional to the run time of the algorithm.

Lower Bound Assumptions III

- The choice of which key is larger is the only point at which two “runs” of an algorithm can exhibit divergent behavior.
- Divergent behavior includes, rearranging the keys in two different ways.

Lower Bound Assumptions

- Algorithms sort by performing key comparisons.
- The contents of the list is arbitrary, so tricks based on the value of a key won’t work.
- The only basis for making a decision in the algorithm is by analyzing the result of a comparison.

Lower Bound Analysis

- We can analyze the behavior of a particular algorithm on an arbitrary list by using a tree.
Lower Bound Analysis

- In the tree we put the indices of the elements being compared.
- Key rearrangements are assumed, but not explicitly shown.
- Although a comparison is an opportunity for divergent behavior, the algorithm does not need to take advantage of this opportunity.

The leaf nodes

- In the leaf nodes, we put a summary of all the key rearrangements that have been done along the path from root to leaf.

1->2  
2->3  
3->1  
2->3  
3->2  
1->2  
2->1  

The Leaf Nodes II

- Each Leaf node represents a permutation of the list.
- Since there are n! initial configurations, and one final configuration, there must be n! ways to reconfigure the input.
- There must be at least n! leaf nodes.

Lower Bound: More Analysis

- Since we are working on a lower bound, in any tree, we must find the longest path from root to leaf. This is the worst case.
- The most efficient algorithm would minimize the length of the longest path.
- This happens when the tree is as close as possible to a complete binary tree.

Lower Bound: Final

- A Binary Tree with k leaves must have height at least lg k.
- The height of the tree is the length of the longest path from root to leaf.
- A binary tree with n! leaves must have height at least lg n!

Lower Bound: Algebra

\[
\sum_{i=1}^{n!} lg i = \sum_{i=2}^{n} \frac{1}{ln 2} \sum_{i=2}^{n} ln i \\
\int_{1}^{lg x} dx \leq \sum_{i=2}^{n} lg i \leq \int_{1}^{lg x} dx \\
\int \ln x \ dx = x \ln x - x \\
n \ln n = \sum_{i=2}^{n!} lg i \leq (n+1) \ln(n+1) - n - 2 \ln 2 + 2 \\
\Theta(n \ln n) \leq \sum_{i=2}^{n!} lg i \leq \Theta(n \ln n) \\
lg n! \in \Theta(n \ lg n)\]
Lower Bound Average Case

- Cannot be worse than worst case \( \Theta(n \lg n) \)
- Can it be better?
- To find average case, add up the lengths of all paths in the decision tree, and divide by the number of leaves.

\[ \Theta(\text{average case}) \]

Sorting - Better than \( O(n \log n) \)?

- If all we know about the keys is an ordering rule
  - No!
- However,
  - If we can compute an address from the key (in constant time) then
    - Integer algorithms can provide better performance

Lower Bound Avg. II

- Because all non-leaves have two children, compressing the tree to make it more balanced will reduce the total sum of all path lengths.

\[
\begin{align*}
\text{Switch X and C} \\
\text{C} & \quad \text{C} \\
\text{X} & \quad \text{A} \\
\text{A} & \quad \text{B} \\
\text{B} & \quad \text{C}
\end{align*}
\]

Path from root to C increases by 1, Path from root to A&B decreases by 1, Net reduction of 1 in the total.

Sorting - Bin Sort

- Assume
  - All the keys lie in a small, fixed range
    - eg
      - integers 0-99
      - characters 'A'-'Z', '0'-'9'
    - There is at most one item with each value of the key
- Bin sort
  - Allocate a bin for each value of the key
    - Usually an entry in an array
  - For each item,
    - Extract the key
    - Compute its bin number
    - Place it in the bin
  - Finished!

Lower Bound Avg. III

- Algorithms with balanced decision trees perform better, on the average than algorithms with unbalanced trees.
- In a balanced tree with as few leaves as possible, there will be \( n! \) leaves and the path lengths will all be of length \( \lg n! \).
- The average will be \( \lg n! \), which is \( \Theta(n \lg n) \)

Sorting - Bin Sort: Analysis

- All the keys lie in a small, fixed range
  - There are \( m \) possible key values
  - There is at most one item with each value of the key
- Bin sort
  - Allocate a bin for each value of the key \( O(m) \)
    - Usually an entry in an array
  - For each item, \( n \) times
    - Extract the key \( O(1) \)
    - Compute its bin number \( O(1) \)
    - Place it in the bin \( O(1) \times n \leftrightarrow O(n) \)
  - Finished! \( O(n) + O(m) = O(n+m) = O(n) \) if \( n >> m \)
Sorting - Bin Sort: Caveat

- Key Range
  - All the keys lie in a small, fixed range
    » There are \( m \) possible key values
    » If this condition is not met, e.g. \( m \gg n \), then bin sort is \( O(m) \)
- Example
  - Key is a 32-bit integer, \( m = 2^{32} \)
  - Clearly, this isn’t a good way to sort a few thousand integers
  - Also, we may not have enough space for bins!
- Bin sort trades space for speed
  - There’s no free lunch!

Bucket Sort

Each element of the array is put in one of the \( N \) “buckets”

- There is at most one item with each value of the key
- Bin sort
  - Allocate a bin for each value of the key \( O(m) \)
    » Usually an entry in an array
    » Array of list heads
  - For each item, \( n \) times
    » Extract the key \( O(1) \)
    » Compute it’s bin number \( O(1) \)
    » Add it to a list \( O(1) \)
    » Join the lists \( O(m) \)
  - Finished! \( O(n) + O(m) = O(n+m) = O(n) \) if \( n \gg m \)

Bucket Sort

Now, pull the elements from the buckets into the array

At last, the sorted array (sorted in a stable way):

Bucket Sort

- Bucket sort
  - Assumption: the keys are in the range \([0, N]\)
  - Basic idea:
    1. Create \( N \) linked lists (buckets) to divide interval \([0,N]\) into subintervals of size 1
    2. Add each input element to appropriate bucket
    3. Concatenate the buckets
  - Expected total time is \( O(n + N) \), with \( n = \) size of original sequence
    » if \( N \) is \( O(n) \) → sorting algorithm in \( O(n) \)!

Does it Work for Real Numbers?

- What if keys are not integers?
  - Assumption: input is \( n \) reals from \([0, 1]\)
  - Basic idea:
    » Create \( N \) linked lists (buckets) to divide interval \([0,1]\) into subintervals of size \( 1/N \)
    » Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution \( \rightarrow O(1) \) bucket size
    » Therefore the expected total time is \( O(n) \)
  - Distribution of keys in buckets similar with …. ?
Non-Comparison Sort – Counting Sort

- Assumption: n input numbers are integers in the range \([0, k]\), \(k = O(n)\).
- Idea:
  - Determine the number of elements less than \(x\), for each input \(x\).
  - Place \(x\) directly in its position.

Counting Sort – pseudocode

\[
\text{Counting-Sort}(A, B, k) \\
\begin{align*}
\text{for } i &\leftarrow 0 \text{ to } k \\
\text{do } C[i] &\leftarrow 0 \\
\text{for } j &\leftarrow 1 \text{ to } \text{length}[A] \\
\text{do } C[A[j]] &\leftarrow C[A[j]] + 1 \\
\text{// C[i] contains number of elements equal to } i. \\
\text{for } i &\leftarrow 1 \text{ to } k \\
\text{do } C[i] &\leftarrow C[i] - 1 \\
\text{// C[i] contains number of elements } \leq i. \\
\text{for } j &\leftarrow \text{length}[A] \text{ downto } 1 \\
\text{do } B[C[A[j]]] &\leftarrow A[j] \\
\text{C[A[j]]} &\leftarrow C[A[j]] - 1 \\
\end{align*}
\]

Total cost is \(\Theta(k+n)\), suppose \(k = O(n)\), then total cost is \(\Theta(n)\).

So, it beats the \(\Omega(n \log n)\) lower bound!

Counting Sort – analysis

1. for \(i = 0\) to \(k\) \(\Theta(1)\)
2. do \(C[i] = 0\) \(\Theta(1)\)
3. for \(i = 1\) to \(\text{length}[A]\) \(\Theta(n)\)
4. do \(C[A[j]] = C[A[j]] + 1\) \(\Theta(1)\) \(\Theta(1)\) \(\Theta(n)\)
5. // \(C[i]\) contains number of elements equal to \(i\). \(\Theta(0)\)
6. for \(i = 1\) to \(k\) \(\Theta(k)\)
7. do \(C[i] = C[i] + C[i-1]\) \(\Theta(1)\) \(\Theta(n)\) \(\Theta(n)\)
8. // \(C[i]\) contains number of elements \(\leq i\). \(\Theta(0)\)
9. for \(j = \text{length}[A] \text{ downto } 1\) \(\Theta(n)\)
10. do \(B[C[A[j]]] = A[j]\) \(\Theta(1)\) \(\Theta(1)\) \(\Theta(n)\)
11. \(C[A[j]] = C[A[j]] - 1\) \(\Theta(1)\) \(\Theta(n)\) \(\Theta(n)\)

Stable sort

- Preserves order of elements with the same key.
- Counting sort is stable.

Crucial question: can counting sort be used to sort large integers efficiently?

Radix sort

\[
\text{Radix-Sort}(A, d) \\
\begin{align*}
\text{for } i &\leftarrow 1 \text{ to } d \\
\text{do } \text{use a stable sort to sort } A \text{ on digit } i \\
\end{align*}
\]

Analysis:

Given \(n d\)-digit numbers where each digit takes on up to \(k\) values, Radix-Sort sorts these numbers correctly in \(\Theta(d(n+k))\) time.
Introduction to Algorithms

Chapter 9: Median and Order Statistics.

Selection

- General Selection Problem:
  - select the i-th smallest element from a set of n distinct numbers
  - that element is larger than exactly i - 1 other elements
- The selection problem can be solved in $O(n \log n)$ time
  - Sort the numbers using an $O(n \log n)$-time algorithm, such as merge sort
  - Then return the i-th element in the sorted array

Medians and Order Statistics

- The i-th order statistic of a set of n elements is the i-th smallest element.
- The minimum of a set of elements:
  - The first order statistic $i = 1$
- The maximum of a set of elements:
  - The n-th order statistic $i = n$
- The median is the “halfway point” of the set
  - $i = \lceil (n+1)/2 \rceil$ is unique when n is odd
  - $i = \lfloor (n+1)/2 \rfloor = n/2$ (lower median) and $\lceil (n+1)/2 \rceil = n/2 + 1$ (upper median), when n is even

Finding Minimum or Maximum

- MINIMUM(A, n)
  - $\text{min} \leftarrow A[1]$
  - for $i \leftarrow 2$ to $n$
    - if $\text{min} > A[i]$
      - then $\text{min} \leftarrow A[i]$
  - return $\text{min}$
- How many comparisons are needed?
  - $n - 1$: each element, except the minimum, must be compared to a smaller element at least once
  - The same number of comparisons are needed to find the maximum
  - The algorithm is optimal with respect to the number of comparisons performed

Simultaneous Min, Max

- Find min and max independently
  - Use $n - 1$ comparisons for each $\Rightarrow$ total of $2n - 2$
- At most $3n/2$ comparisons are needed
  - Process elements in pairs
  - Maintain the minimum and maximum of elements seen so far
  - Don’t compare each element to the minimum and maximum separately
  - Compare the elements of a pair to each other
  - Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far
  - This leads to only 3 comparisons for every 2 elements
Analysis of Simultaneous Min, Max

- Setting up initial values:
  - \( n \) is odd: set both min and max to the first element
  - \( n \) is even: compare the first two elements, assign the smallest one to min and the largest one to max

- Total number of comparisons:
  - \( n \) is odd: we do \( \frac{3(n-1)}{2} \) comparisons
  - \( n \) is even: we do 1 initial comparison + \( \frac{3(n-2)}{2} \) more comparisons = \( \frac{3n}{2} - 2 \) comparisons

Example: Simultaneous Min, Max

- \( n = 5 \) (odd), array \( A = [2, 7, 1, 3, 4] \)
  1. Set \( \text{min} = \text{max} = 2 \)
  2. Compare elements in pairs:
     - \( 1 < 7 \) compare \( 1 \) with \( \text{min} \) and \( 7 \) with \( \text{max} \)
       \( \Rightarrow \text{min} = 1, \text{max} = 7 \)
     - \( 3 < 4 \) compare \( 3 \) with \( \text{min} \) and \( 4 \) with \( \text{max} \)
       \( \Rightarrow \text{min} = 1, \text{max} = 7 \)
  We performed: \( \frac{3(n-1)}{2} = 6 \) comparisons

Example: Simultaneous Min, Max

- \( n = 6 \) (even), array \( A = [2, 5, 3, 7, 1, 4] \)
  1. Compare \( 2 \) with \( 5 \): \( 2 < 5 \)
  2. Set \( \text{min} = 2, \text{max} = 5 \)
  3. Compare elements in pairs:
     - \( 3 < 7 \) compare \( 3 \) with \( \text{min} \) and \( 7 \) with \( \text{max} \)
       \( \Rightarrow \text{min} = 3, \text{max} = 7 \)
     - \( 1 < 4 \) compare \( 1 \) with \( \text{min} \) and \( 4 \) with \( \text{max} \)
       \( \Rightarrow \text{min} = 1, \text{max} = 7 \)
  We performed: \( \frac{3n}{2} - 2 = 7 \) comparisons

Medians and Order Statistics

- CLRS Chapter 9

What Are Order Statistics?

The \( k \)-th order statistic is the \( k \)-th smallest element of an array.

What are Order Statistics?

Selecting \( i \)-th ranked item from a collection.
- First:
  \[
  \frac{i}{n} \leq \frac{n}{2} \leq \frac{i}{n}
  \]
- Last:
  \[
  \frac{i-1}{n} \leq \frac{n}{2} \leq \frac{i}{n}
  \]
- Median(s):
  \[
  i = \frac{n}{2}
  \]
Order Statistics Overview

- Assume collection is unordered, otherwise trivial.
  
  \[ \text{find } \text{ith order stat } = A[i] \]

- Can sort first \( \Theta(n \lg n) \), but can do better \( \Theta(n) \).
- I can find max and min in \( \Theta(n) \) time (obvious)

Order Statistics Overview

How can we modify Quicksort to obtain expected-case \( \Theta(n) \)?

- Pivot, partition, but recur only on one set of data. No join.

Randomized Selection

- Analyzing \text{RandomizedSelect()}:
  - Worst case: partition always 0:n-1
    \[ T(n) = T(n-1) + O(n) \]
    \[ = O(n^2) \]
    » No better than sorting!
  - “Best” case: suppose a 9:1 partition
    \[ T(n) = T(9n/10) + O(n) \]
    \[ = O(n) \] (Master Theorem, case 3)
    » Better than sorting!
  - Average case: \( O(n) \) remember from quicksort

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Guarantee a good partitioning element
  - Guarantee worst-case linear time selection
- Warning: Non-obvious & unintuitive

Using the Pivot Idea

- \text{Randomized-Select}(A[p..r],i) looking for \( \text{ith } \text{a.s.} \)
  if \( p = r \)
    return \( A[p] \)
  q <- Randomized-Partition(A,p,r)
  k <- q-p+1 \( \text{the size of the left partition} \)
  if \( i = k \)
    return \( A[q] \)
  else if \( i < k \)
    then the answer is in the front
    return Randomized-Select(A,p,q-1)
  else \( i > k \)
    then the answer is in the last partition
    return Randomized-Select(A,q+1,r)

Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide \( n \) elements into groups of 5
  2. Find median of each group (How? How long?)
  3. Use Select() recursively to find median \( x \) of the \( \lfloor n/5 \rfloor \) medians
  4. Partition the \( n \) elements around \( x \). Let \( k = \text{rank}(x) \)
  5. if \( i = k \) then return \( x \)
    if \( i < k \) then use Select() recursively to find \( i \)th smallest element in first partition
    else \( i > k \) use Select() recursively to find \( (i-k) \)th smallest element in last partition
Order Statistics: Algorithm

Select(A,n,i):
Divide input into groups of size 5.
/* Partition on median-of-medians */
medians = array of each group's median.
pivot = Select(medians, , )
Left Array L and Right Array G = partition(A, pivot)
/* Find i
 element in L, pivot, or G */
k = |L| + 1
If i=k, return pivot
If i<k, return Select(L, k-1, i)
If i>k, return Select(G, n-k, i-k)

Order Statistics: Analysis

T(n) = T(floor[n/5]) + T(max(k-1, n-k)) + O(n)

How to simplify?

Order Statistics: Analysis 1

One group of 5 elements.
Order Statistics: Analysis 1

At most
\[
\frac{n}{5} \cdot 2 + \left( \frac{n}{5} \right)^2 \leq \frac{7n}{10} + 6
\]

Order Statistics

Why groups of 5?

Sum of two recurrence sizes must be < 1. Grouping by 5 is smallest size that works.

Order Statistics: Analysis

\[ T(n) = T\left(\left\lfloor \frac{n}{5} \right\rfloor \right) + T\left(\left\lfloor \frac{7n}{10} \right\rfloor + 6 \right) + O(n) \]

A very unusual recurrence. How to solve?

Order Statistics: Analysis

Substitution: Prove \( T(n) \leq c \times n \).

\[
T(n) \leq c \times \left\lfloor \frac{n}{5} \right\rfloor + c \times \left( \frac{7n}{10} + 6 \right) + d \times n
\]

\[
\leq c \times \left( \frac{n}{5} + 1 \right) + c \times \left( \frac{7n}{10} + 6 \right) + d \times n
\]

Overestimate ceiling

\[
= \frac{9}{10} c \times n + 7c + d \times n
\]

Algebra

\[
= c \times n - (c \times n/10 - 7c - d \times n)
\]

Algebra

\[
\leq c \times n
\]

when choose \( c,d \) such that \( 0 \leq c \times n/10 - 7c - d \times n \)

General Selection Problem

• Select the \( i \)-th order statistic (\( i \)-th smallest element) form a set of \( n \) distinct numbers

• Idea:
  - Partition the input array similarly with the approach used for QuickSort (use RANDOMIZED-PARTITION)
  - Recurse on one side of the partition to look for the \( i \)-th element depending on where \( i \) is with respect to the pivot

• Selection of the \( i \)-th smallest element of the array \( A \) can be done in \( \Theta(n) \) time

Randomized Select

\[
\text{RANDOMIZED-SELECT}(A, p, r, i)
\]

if \( p = r \) then return \( A[p] \)

\[
q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r) \text{ in this partition pivot}
\]

\[
i < k \Rightarrow \text{search in this partition}
\]

\[
i > k \Rightarrow \text{search in this partition}
\]

\[
i = k \Rightarrow \text{pivot value is the answer}
\]

\[
t = q + 1
\]

\[
\text{else if } i < k \text{ then return } \text{RANDOMIZED-SELECT}(A, p, q-1, i)
\]

\[
\text{else return } \text{RANDOMIZED-SELECT}(A, q+1, r, i-k)
\]

Try: \( A = \{1,4,2,6,8,5\} \)
**Analysis of Running Time**

- **Worst case running time:** $\Theta(n^2)$
  - If we always partition around the largest/smallest remaining element
  - Partition takes $\Theta(n)$ time
  - $T(n) = O(1)$ (choose the pivot) + $\Theta(n)$ (partition) + $T(n-1)$
    $$1 + n + T(n-1) = \Theta(n^2)$$

**Example**

- Find the $-11$th smallest element in array: $A = \{12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 18, 20, 33, 16, 33, 21, 30, 3, 47\}$

**Example (cont.)**

- Divide the array into groups of 5 elements

  | 12 | 4  | 43 | 2  | 20 | 30 |
  | 34 | 17 | 82 | 19 | 33 | 3  |
  | 0  | 32 | 25 | 12 | 16 | 47 |
  | 3  | 3  | 27 | 5  | 33 | n  |

- 1. Divide the input array around \( x \)
- 2. Sort the groups and find their medians
- 3. Find the median of the medians

**A Better Selection Algorithm**

- Can perform Selection in $O(n)$ Worst Case
- Idea: guarantee a good split on partitioning
  - Running time is influenced by how “balanced” are the resulting partitions
- Use a modified version of PARTITION
  - Takes as input the element around which to partition

**Example (cont.)**

- First partition:
  - Pivot: 17 (position of the pivot is \( q = 11 \))
- Second partition:
  - First partition: $\{12, 0, 3, 4, 3, 2, 12, 5, 16, 3\}$
  - Pivot: 17
  - 6th smallest element in array:

**Selection in $O(n)$ Worst Case**

- Divide the \( n \) elements into groups of 5 \( \lceil n/5 \rceil \) groups
- Find the median of each of the \( \lceil n/5 \rceil \) groups
- Use insertion sort, then pick the median
- Use SELECT recursively to find the median \( x \) of the \( \lceil n/5 \rceil \) medians
- Partition the input array around \( x \), using the modified version of PARTITION
  - There are \( k-1 \) elements on the low side of the partition and \( n-k \) on the high side
- If \( i = k \) then return \( x \). Otherwise, use SELECT recursively:
  - Find the \( i-th \) smallest element on the low side if \( i \leq k \)
  - Find the \( (i-k) \)-th smallest element on the high side if \( i > k \)
Sorting - Generalised Bin Sort

- Radix sort - Bin sort in phases
  - Phase 1 - Sort by least significant digit
    - Example
      - Phase 1 - Sort by least significant digit
        - Phase 2 - Sort by most significant digit
          - Add to end of anything in the bin already!
Radix sort - Bin sort in phases
- Phase 1 - Sort by least significant digit
- Phase 2 - Sort by most significant digit

Note that the 0 bin had to be quite large!

Radix Sort
- Start with least significant digit
- Separate keys into groups based on value of current digit
- Make sure not to disturb original order of keys
- Combine separate groups in ascending order
- Repeat, scanning digits in reverse order

Radix Sort: Example

Sorting - Generalised Bin Sort
- Radix sort - Bin sort in phases
  - Phase 1 - Sort by least significant digit
  - Phase 2 - Sort by most significant digit

Sorting - Generalised Bin Sort
- Radix sort - Analysis
  - Phase 1 - Sort by least significant digit
    - Create m bins \( O(m) \)
    - Allocate \( n \) items \( O(n) \)
  - Phase 2
    - Create m bins \( O(m) \)
    - Allocate \( n \) items \( O(n) \)
  - Final
    - Link m bins \( O(m) \)
    - All steps in sequence, so add
    - Total \( O(3m+2n) \rightarrow O(m+n) \) for \( m \ll n \)
**Radix Sort: Analysis**

- Each digit requires $n$ comparisons
- The algorithm is $\Theta(n)$
- The preceding lower bound analysis does not apply, because Radix Sort does not compare keys.
- Radix Sort is sometimes known as bucket sort. (Any distinction between the two is unimportant)
- Alg. was used by operators of card sorters.