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.

Repeated Minimum: Code

Fixed n-1 iterations

Fixed n-i iterations

```
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.

$$\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}\in\Theta(n^2)$$

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

```
for i := 1 to n-1 do          ← Worst case n-1 iterations
    Switch := False;
    for j := 1 to n-i do      ← Fixed n-i iterations
        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:

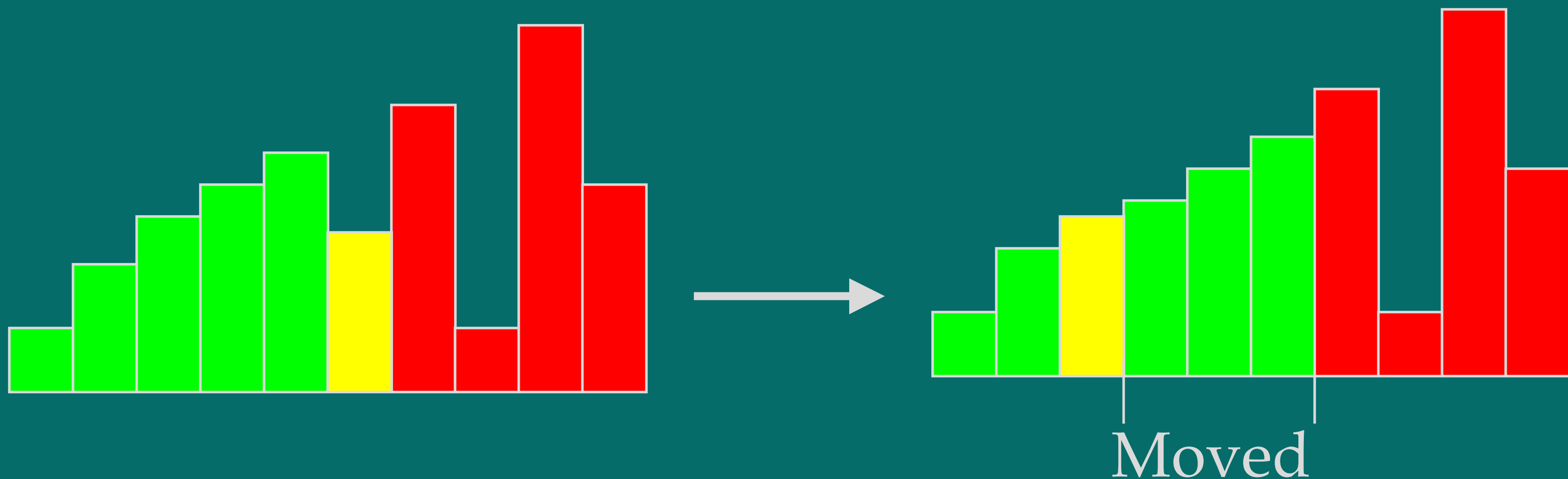
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

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

Fixed n-1 iterations

Worst case i-1 comparisons

```
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+ … + n-1

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

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.

Position

1	2	3	...	i-2	i-1	i	
			...				
i-1	i-1	i-2	...	3	2	1	

Comparisons necessary to place key in this position.

Insertion Sort Average III

Average Comparisons to place one key

$$\sum_{j=1}^{i-1} \frac{1}{i} j + \frac{1}{i}(i-1)$$

Solving

$$= \frac{1}{i}\sum_{i=1}^{i-1} j + \left(1 - \frac{1}{i}\right) = \frac{1}{i}\frac{i(i-1)}{2} + \frac{2}{2} - \frac{1}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Insertion Sort Average IV

For All Keys:

$$A(n) = \sum_{i=2}^{n}\left(\frac{i+1}{2} - \frac{1}{i}\right) = \frac{1}{2}\sum_{i=2}^{n} i + \frac{1}{2}\sum_{i=2}^{n} 1 - \sum_{i=2}^{n}\frac{1}{i}$$

$$= \frac{n(n+1)}{4} - \frac{1}{2} + \frac{2n}{4} - \frac{1}{2} - \sum_{i=2}^{n}\frac{1}{i}$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=2}^{n}\frac{1}{i} \in \Theta\left(n^2\right)$$

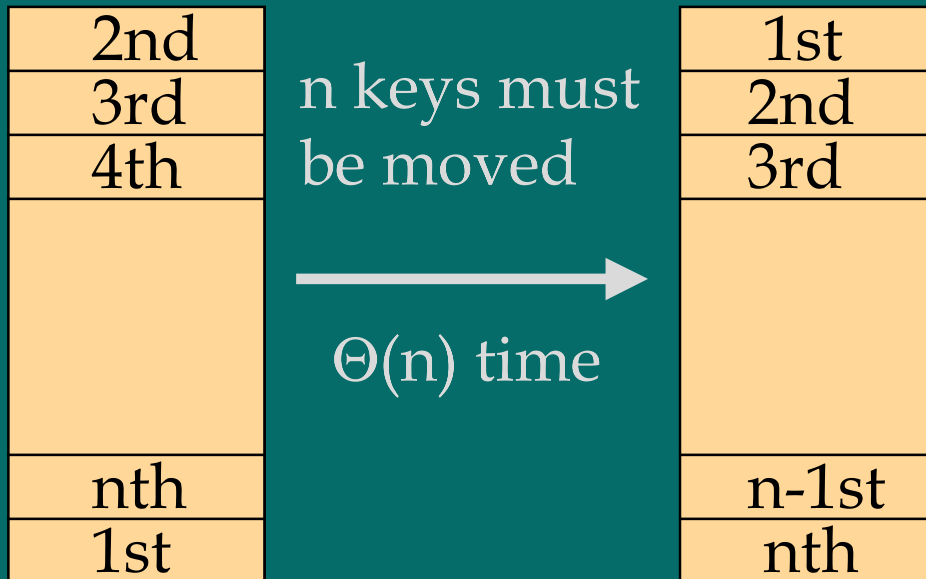
Optimality Analysis I

- 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

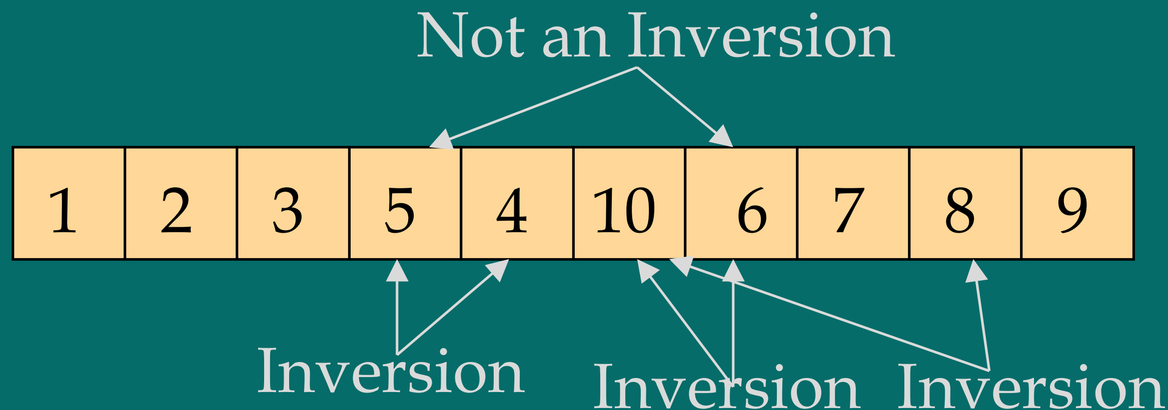
The amount of time needed to move one key is not dependent on n.

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?

Inversions

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



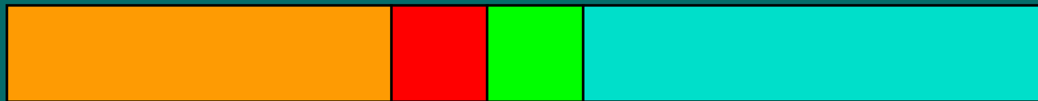
Maximum Inversions

- The total number of pairs is:

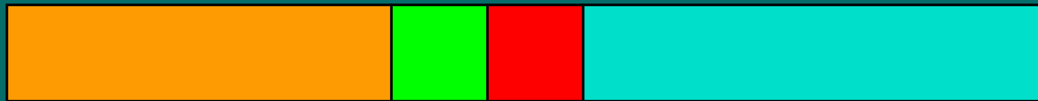
$$\binom{n}{2} = \frac{n(n-1)}{2}$$

- This is the maximum number of inversions in any list.
- Exchanging adjacent pairs of keys removes at most one inversion.

Swapping Adjacent Pairs



Swap Red and Green



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.

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

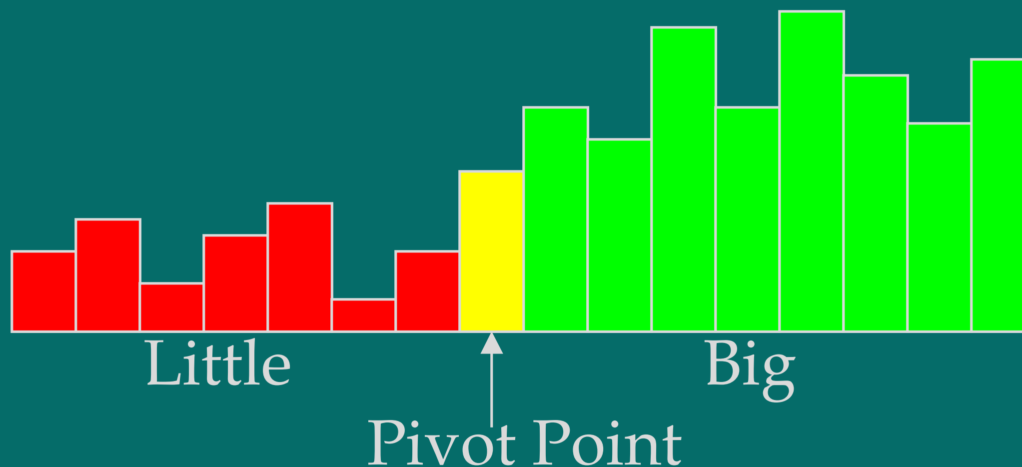
- There are n! ways to rearrange a list of n elements.
- Recall that a rearrangement is called a *permutation*.
- If we reverse a rearranged list, every pair that used to be an inversion will no longer be an inversion.
- By the same token, all non-inversions become inversions.

Lower Bound For Average II

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

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 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

```
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
```

Points to last element in "Small" section.

Fixed n-1 iterations

Make "Small" section bigger and move key into it.

Else the "Big" section gets bigger.

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)

$$Q(n) = 2Q(n/2) + n - 1 \in \Theta(n \lg n)$$

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:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

QS Avg. Case: Assumptions

- **Average will be taken over *Location of Pivot***
- **All Pivot Positions are equally likely**
- **Pivot positions in each call are independent of one another**

QS Avg: Formulation

- $A(0) = 0$
- If the pivot appears at position i, $1 \le i \le n$ 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: Recurrence

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big)$$

$$\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big)$$

$$= \big(A(0) + A(n-1)\big) + \big(A(1) + A(n-2)\big) +$$

$$\ldots + \big(A((n-1)-1) + A(n-(n-1)\big) +$$

$$\big(A(n-1) + A(n-n)\big)$$

QS Avg: Recurrence II

$$\sum_{i=1}^{n} \left(A(i-1) + A(n-i) \right) =$$

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

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

QS Avg: Solving Recurr.

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

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

$$\le \Theta(n) + \frac{2}{n}\sum_{i=1}^{n-1}(ai\lg i + b)$$

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

QS Avg: Continuing

By Integration:

$$\sum_{i=1}^{n-1} i \lg i \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

QS Avg: Finally

$$A(n) \le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n)$$

$$\le an\lg n - \frac{a}{4}n + 2b + \Theta(n)$$

$$= an\lg n - \frac{a}{4}n + 2b + \Theta(n)$$

$$\le an\lg n + b$$

Merge Sort

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

The Merge Algorithm

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

```
Ax := 1; Bx := 1; Cx := 1;
while Ax ≤ n and Bx ≤ n do
   if A[Ax] < B[Bx] then
     C[Cx] := A[Ax];
     Ax := Ax + 1;
   else
     C[Cx] := B[Bx];
     Bx := Bx + 1;
   endif
   Cx := Cx + 1;
endwhile
```

```
while Ax ≤ n do
   C[Cx] := A[Ax];
   Ax := Ax + 1;
   Cx := Cx + 1;
endwhile
while Bx ≤ n do
   C[Cx] := B[Bx];
   Bx := Bx + 1;
   Cx := Cx + 1;
endwhile
```

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:

$$W(n) = 2\,W(n/2) + n - 1 \in \Theta(n \lg n)$$

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 $\Theta(n)$

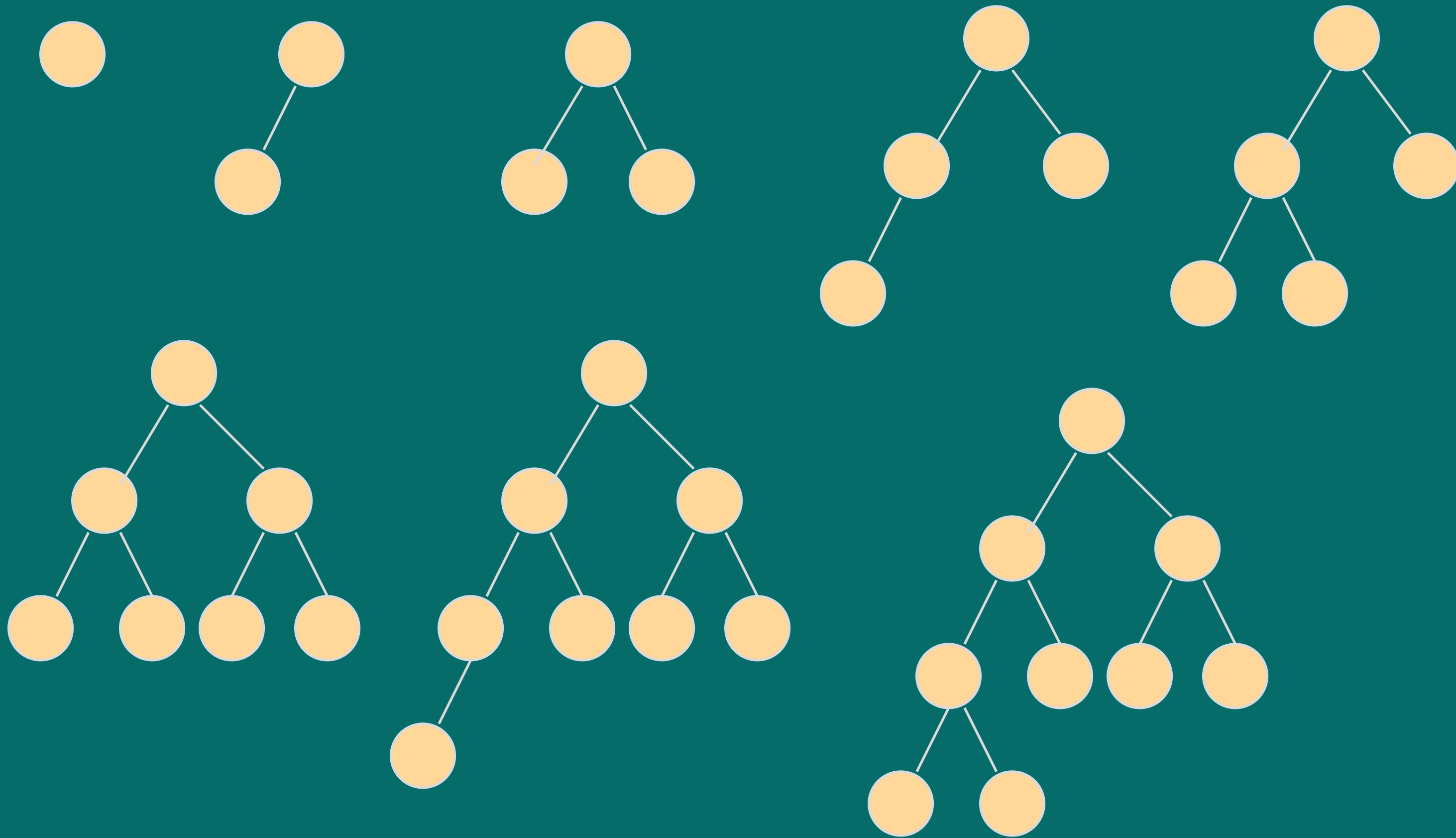
Heapsort: Heaps

- 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

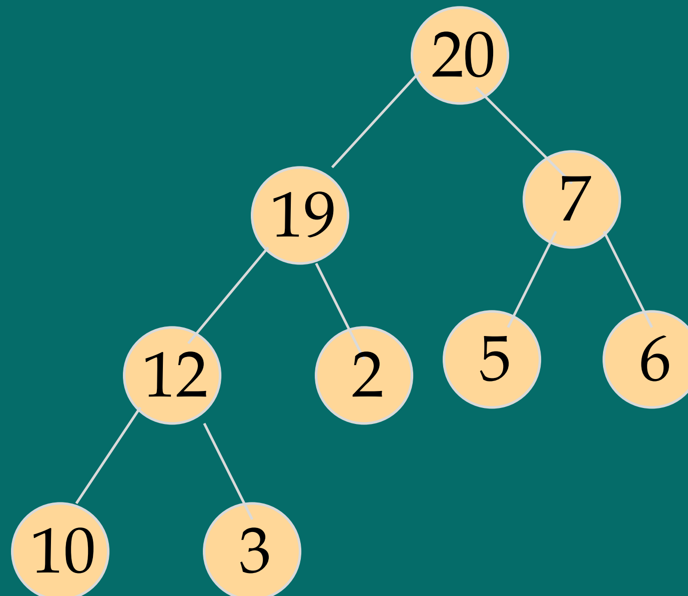
- If there is a vertex with only one child, it must be a left child, and the child must be the rightmost vertex on the lowest level.
- For a given number of vertices, there is only one legal structure

Heapsort: Heap examples



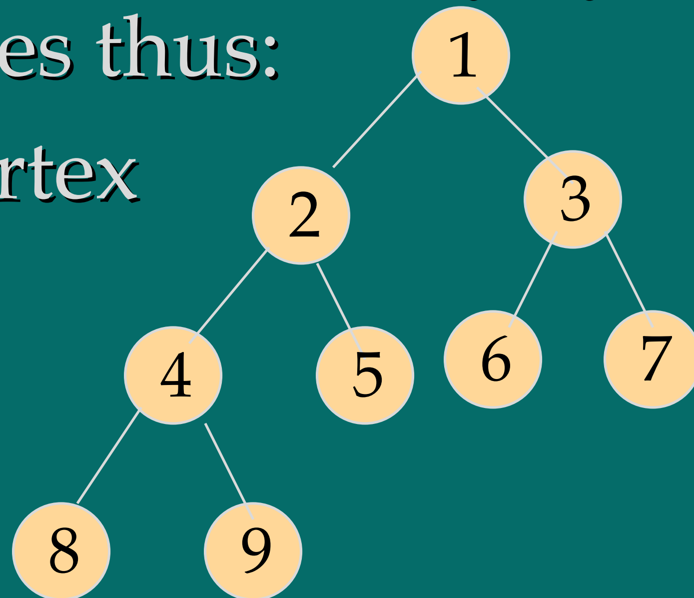
Heapsort: Heap Values

- 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 $\Theta(\lg n)$

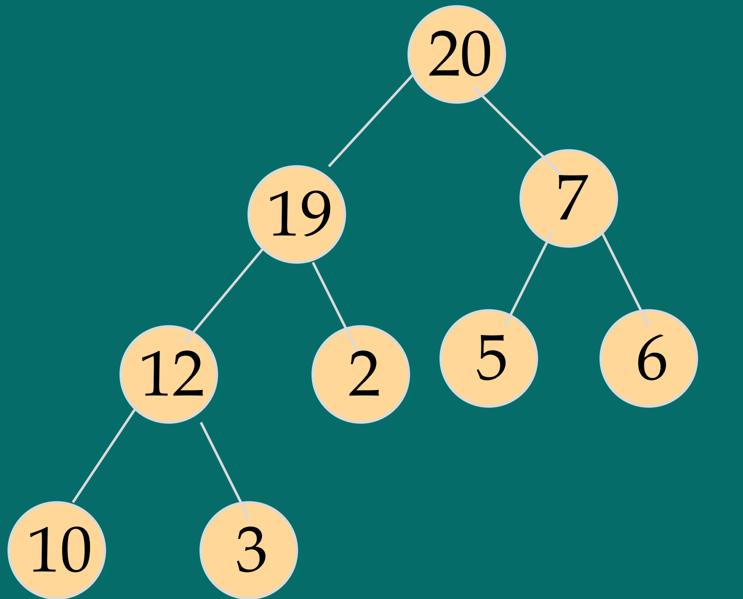
Heapsort: Creating a Heap

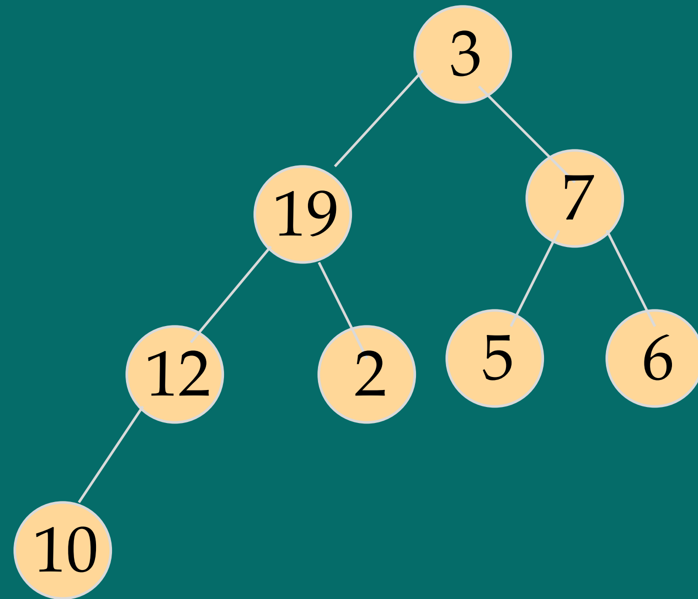
- An arbitrary list can be turned into a heap by calling FixHeap on each non-leaf in reverse order.
- If n is the size of the heap, the non-leaf with the highest index has index n/2.
- Creating a heap is obviously O(n lg n).
- A more careful analysis would show a true time bound of $\Theta(n)$

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

Sorting Example I

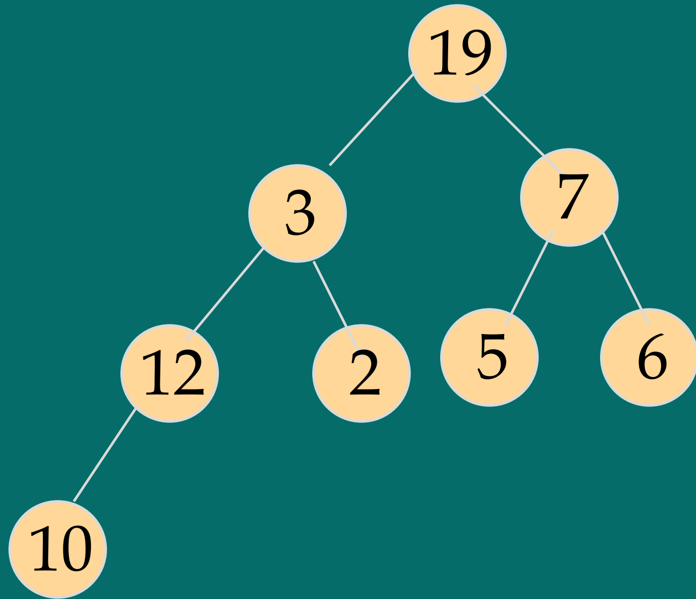


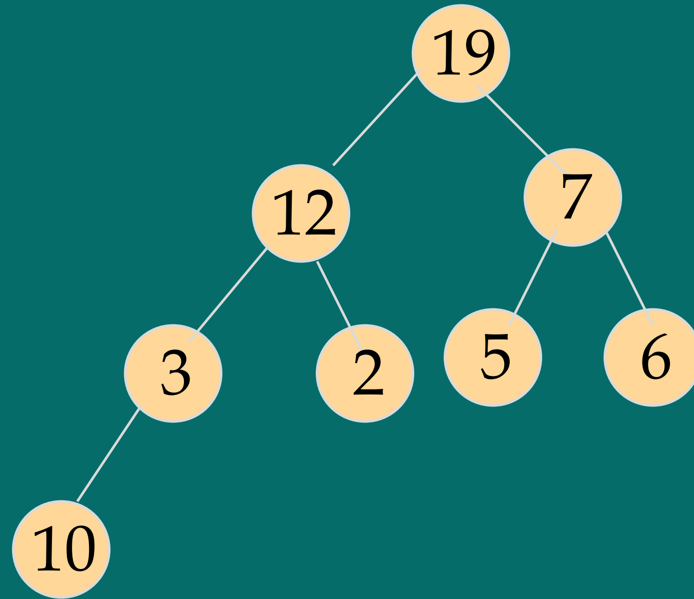


20	19	7	12	2	5	6	10	3

3	19	7	12	2	5	6	10	20

Sorting Example II

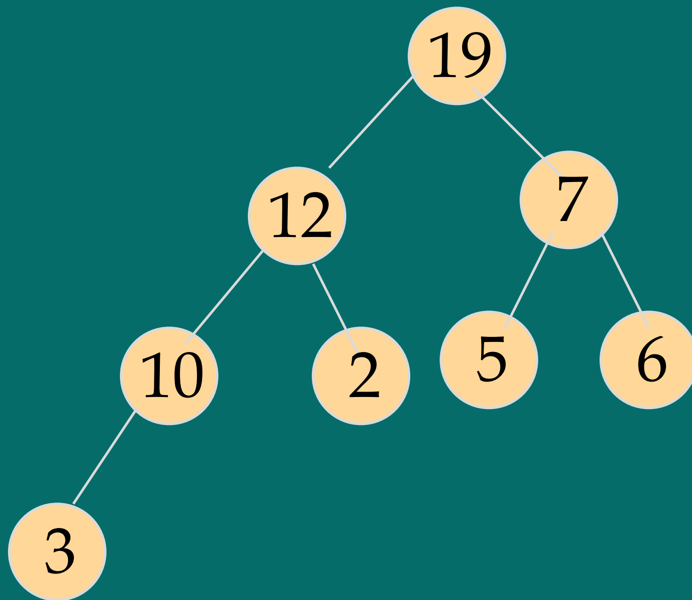




19	3	7	12	2	5	6	10	20

19	12	7	3	2	5	6	10	20

Sorting Example III



19	12	7	10	2	5	6	3	20

Ready to swap 3 and 19.

Heap Sort: Analysis

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

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

- 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 Assumptions II

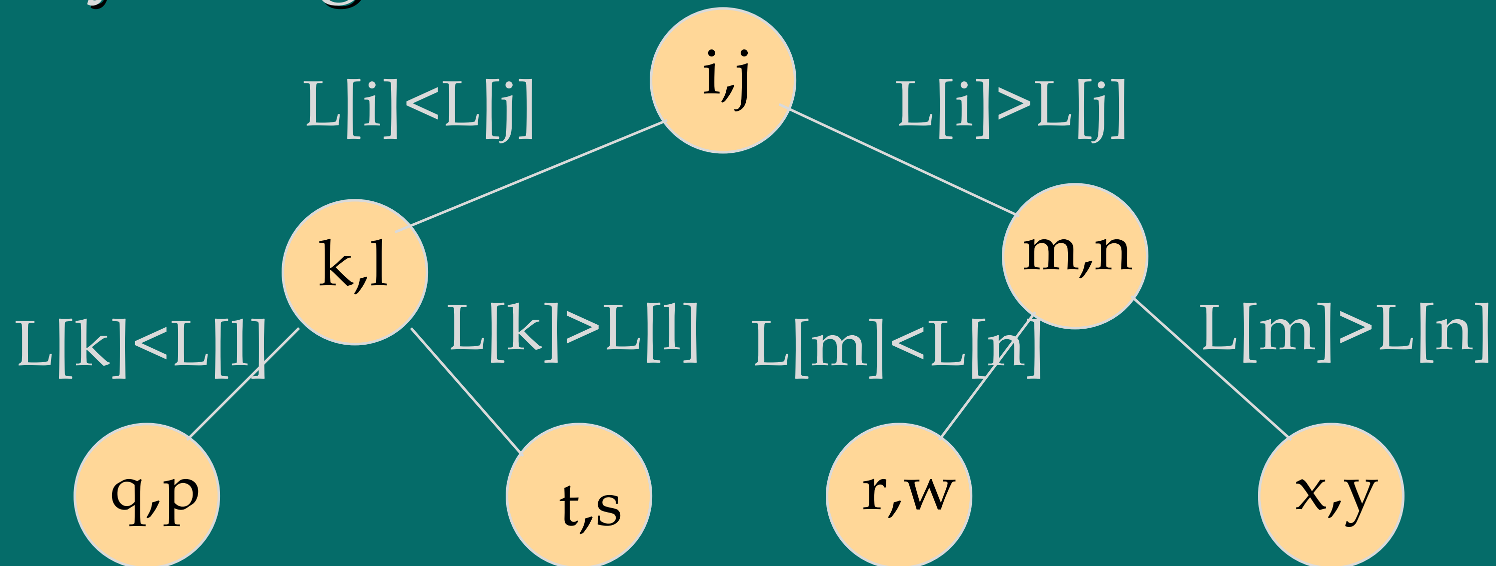
- Assume that all keys are distinct, since all sort algorithms must handle this case.
- Because there are no "tricks" that work, the only information we can get from a key comparison is:
- Which key is larger

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 Analysis

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

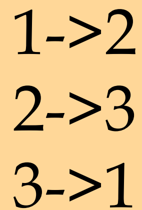


Lower Bound Analysis

- 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.



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

$$\lg n! = \sum_{i=2}^{n} \lg i = \frac{1}{\ln 2} \sum_{i=2}^{n} \ln i$$

$$\int_{1}^{n} \lg x \, dx \le \sum_{i=2}^{n} \lg i \le \int_{2}^{n+1} \lg x \, dx \qquad \int \ln x \, dx = x \ln x - x$$

$$n \ln n - n + 1 \le \sum_{i=2}^{n} \lg i \le (n+1) \ln(n+1) - n - 1 - 2 \ln 2 + 2$$

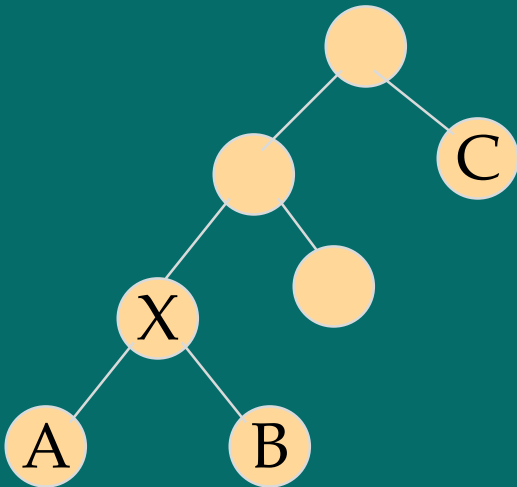
$$\Theta(n \ln n) \le \sum_{i=2}^{n} \lg i \le \Theta(n \ln n) \qquad \lg n! \in \Theta(n \lg n)$$

Lower Bound Average Case

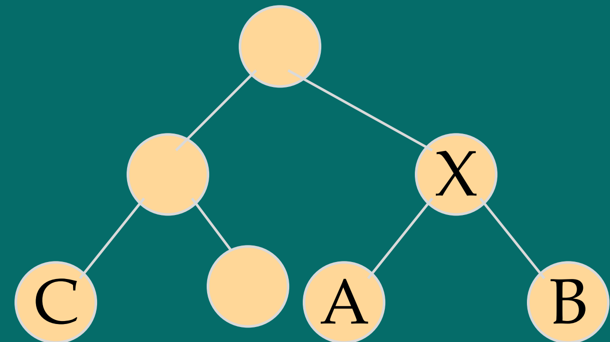
- Cannot be worse than worst case $\Theta(n \lg n)$
- Can it be better?
- To find average case, add up the lengths of all paths in the decision tree, and divide by the number of leaves.

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.



Switch X and C



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

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 $\Theta(n \lg n)$

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

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.

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.

Repeated Minimum: Code

Fixed n-1 iterations

Fixed n-i iterations

```
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.

$$\sum_{k=1}^{n-1}k = \frac{n(n-1)}{2} \in \Theta(n^2)$$

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

```
for i := 1 to n-1 do          ← Worst case n-1 iterations
    Switch := False;
    for j := 1 to n-i do      ← Fixed n-i iterations
        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
```

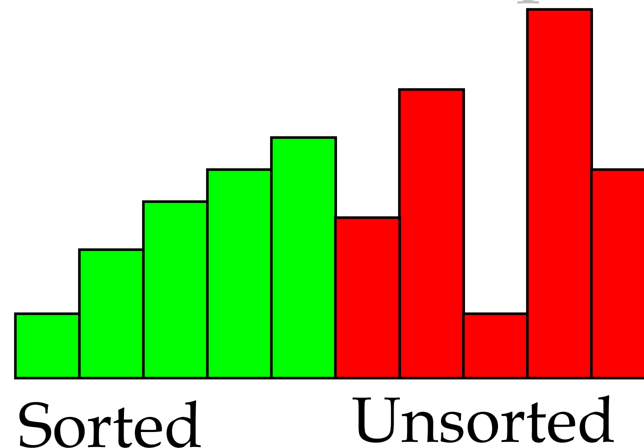
Bubble Sort Analysis

Being smart right from the beginning:

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

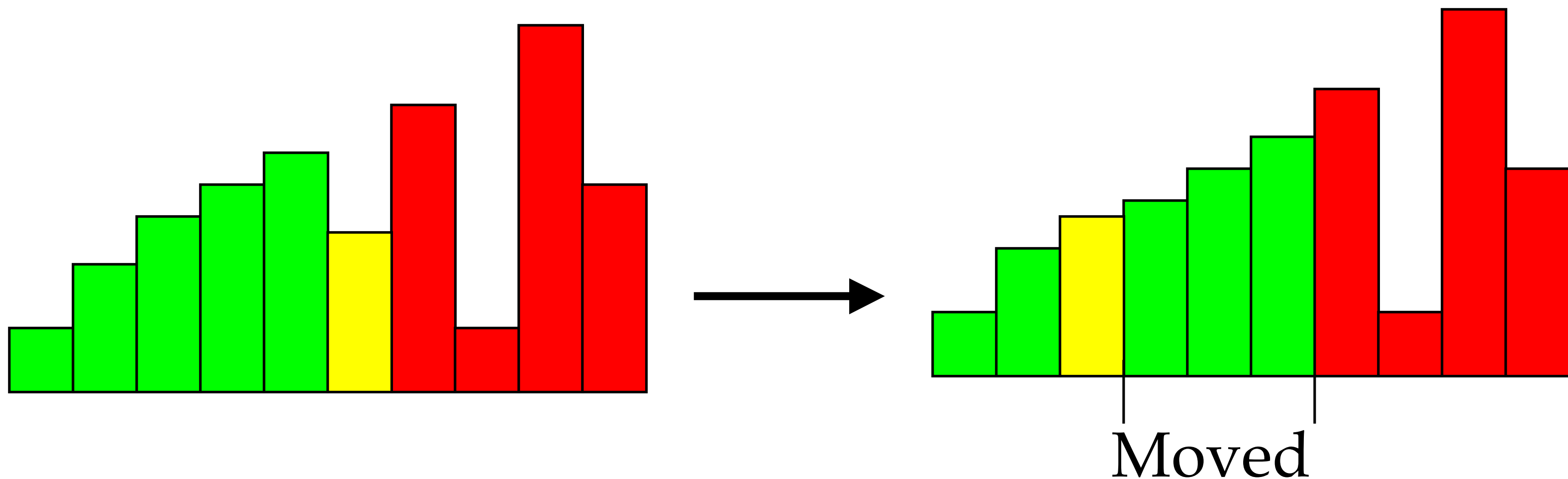
Insertion Sort I

- 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

```
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+ … + n-1

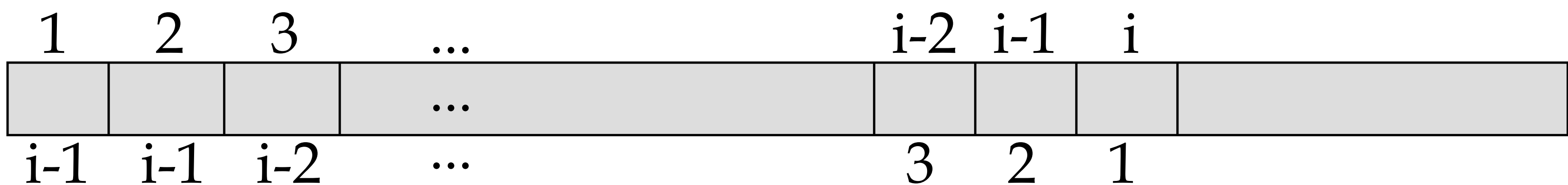
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

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.



Comparisons necessary to place key in this position.

Insertion Sort Average III

Average Comparisons to place one key

$$\sum_{j=1}^{i-1} \frac{1}{i} j + \frac{1}{i}(i-1)$$

Solving

$$= \frac{1}{i} \sum_{i=1}^{i-1} j + \left(1 - \frac{1}{i}\right) = \frac{1}{i} \frac{i(i-1)}{2} + \frac{2}{2} - \frac{1}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Insertion Sort Average IV

For All Keys:

$$A(n) = \sum_{i=2}^{n}\left(\frac{i+1}{2} - \frac{1}{i}\right) = \frac{1}{2}\sum_{i=2}^{n} i + \frac{1}{2}\sum_{i=2}^{n} 1 - \sum_{i=2}^{n}\frac{1}{i}$$

$$= \frac{n(n+1)}{4} - \frac{1}{2} + \frac{2n}{4} - \frac{1}{2} - \sum_{i=2}^{n}\frac{1}{i}$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=2}^{n}\frac{1}{i} \in \Theta\left(n^2\right)$$

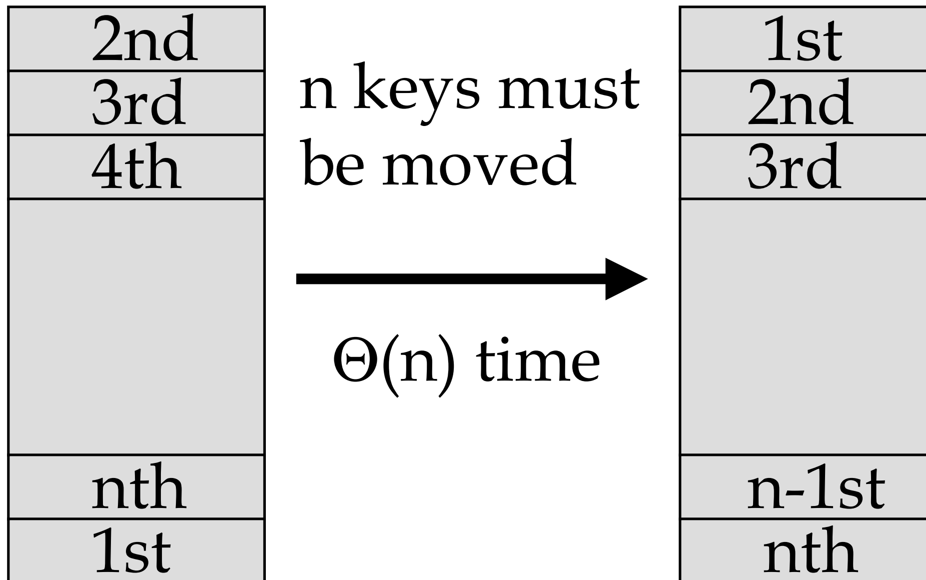
Optimality Analysis I

- 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

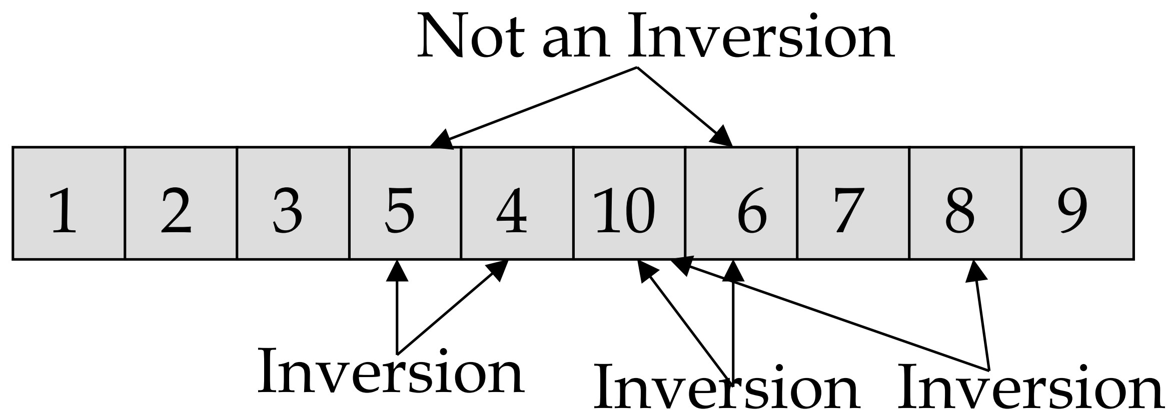
The amount of time needed to move one key is not dependent on n.

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?

Inversions

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



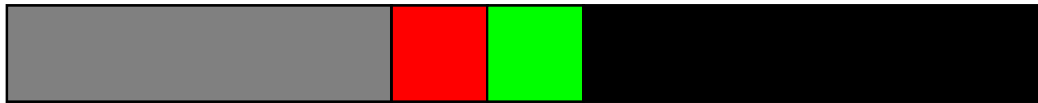
Maximum Inversions

- The total number of pairs is:

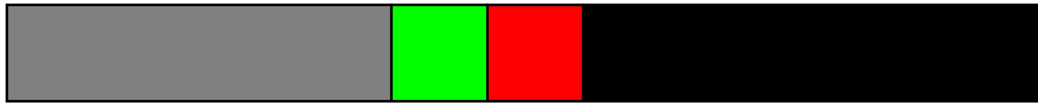
$$\binom{n}{2} = \frac{n(n-1)}{2}$$

- This is the maximum number of inversions in any list.
- Exchanging adjacent pairs of keys removes at most one inversion.

Swapping Adjacent Pairs



Swap Red and Green



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.

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

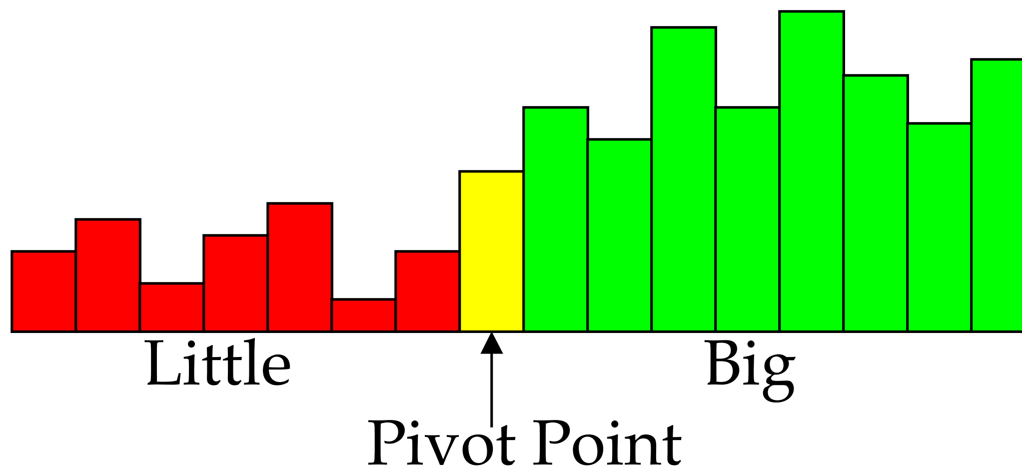
- There are n! ways to rearrange a list of n elements.
- Recall that a rearrangement is called a *permutation*.
- If we reverse a rearranged list, every pair that used to be an inversion will no longer be an inversion.
- By the same token, all non-inversions become inversions.

Lower Bound For Average II

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

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 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

```
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
```

Points to last element in "Small" section. (SplitPoint := 1)

Fixed n-1 iterations (for i := 2 to n do)

Make "Small" section bigger and move key into it. (SplitPoint := SplitPoint + 1)

Else the "Big" section gets bigger. (endif)

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)

$$Q(n) = 2Q(n/2) + n - 1 \in \Theta(n \lg n)$$

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:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta\left(n^2\right)$$

QS Avg. Case: Assumptions

- **Average will be taken over *Location of Pivot***
- **All Pivot Positions are equally likely**
- **Pivot positions in each call are independent of one another**

QS Avg: Formulation

- **A(0) = 0**
- **If the pivot appears at position i, 1≤i≤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: Recurrence

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big)$$

$$\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big)$$

$$= \big(A(0) + A(n-1)\big) + \big(A(1) + A(n-2)\big) +$$

$$\ldots + (A((n-1)-1) + A(n-(n-1)) +$$

$$\big(A(n-1) + A(n-n)\big)$$

QS Avg: Recurrence II

$$\sum_{i=1}^{n}\left(A(i-1)+A(n-i)\right)=$$

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

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

QS Avg: Solving Recurr.

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

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

$$\le \Theta(n) + \frac{2}{n}\sum_{i=1}^{n-1}(ai \lg i + b)$$

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

QS Avg: Continuing

By Integration:

$$\sum_{i=1}^{n-1} i \lg i \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

QS Avg: Finally

$$A(n) \le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n)$$

$$\le an\lg n - \frac{a}{4}n + 2b + \Theta(n)$$

$$= an\lg n - \frac{a}{4}n + 2b + \Theta(n)$$

$$\le an\lg n + b$$

Merge Sort

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

The Merge Algorithm

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

```
Ax := 1; Bx := 1; Cx := 1;
while Ax ≤ n and Bx ≤ n do
   if A[Ax] < B[Bx] then
     C[Cx] := A[Ax];
     Ax := Ax + 1;
   else
     C[Cx] := B[Bx];
     Bx := Bx + 1;
   endif
   Cx := Cx + 1;
endwhile
```

```
while Ax ≤ n do
   C[Cx] := A[Ax];
   Ax := Ax + 1;
   Cx := Cx + 1;
endwhile
while Bx ≤ n do
   C[Cx] := B[Bx];
   Bx := Bx + 1;
   Cx := Cx + 1;
endwhile
```

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:

$$\mathrm{W}(n) = 2\,\mathrm{W}(n/2) + n - 1 \in \Theta(n \lg n)$$

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 $\Theta(n)$

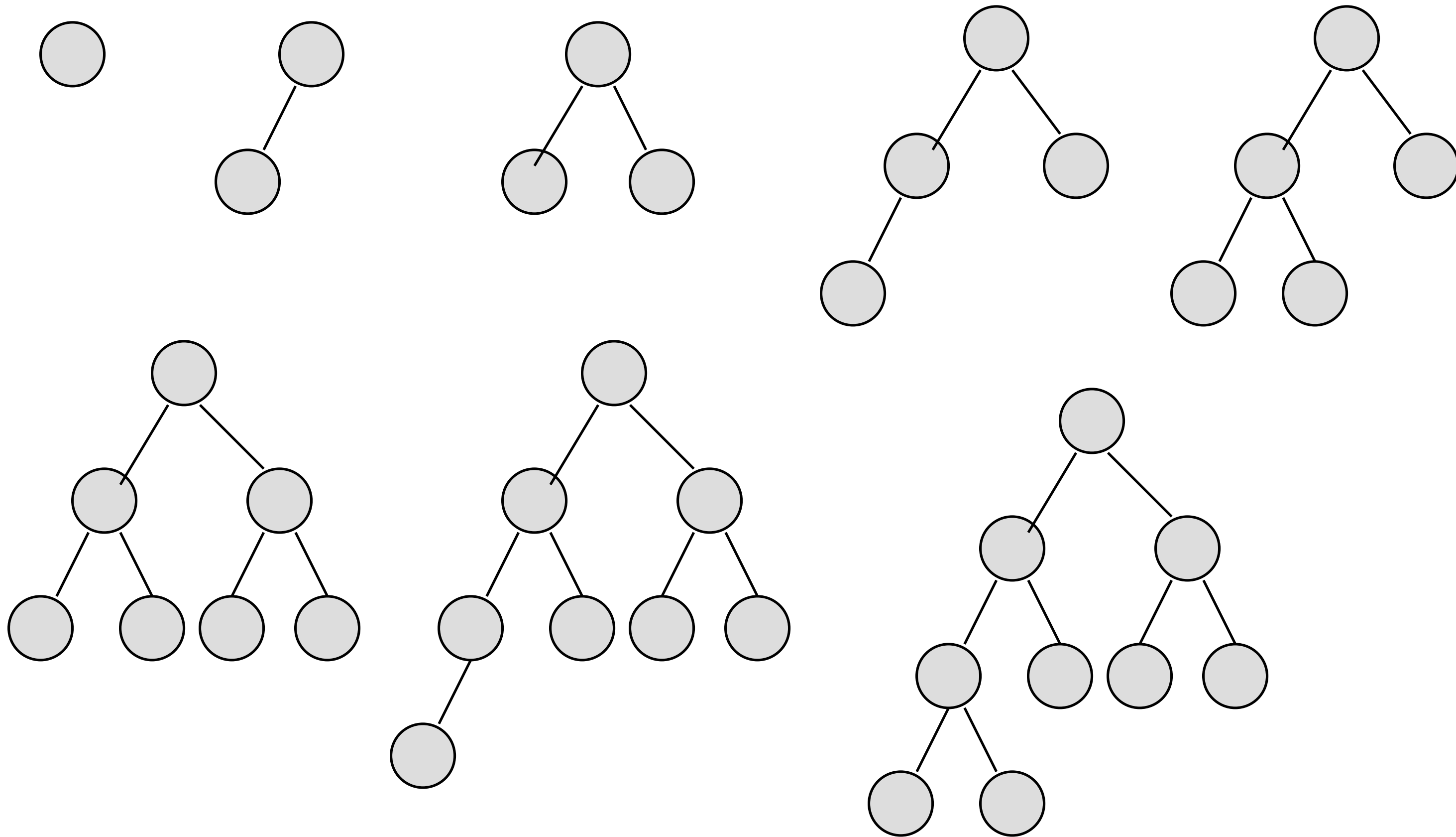
Heapsort: Heaps

- 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

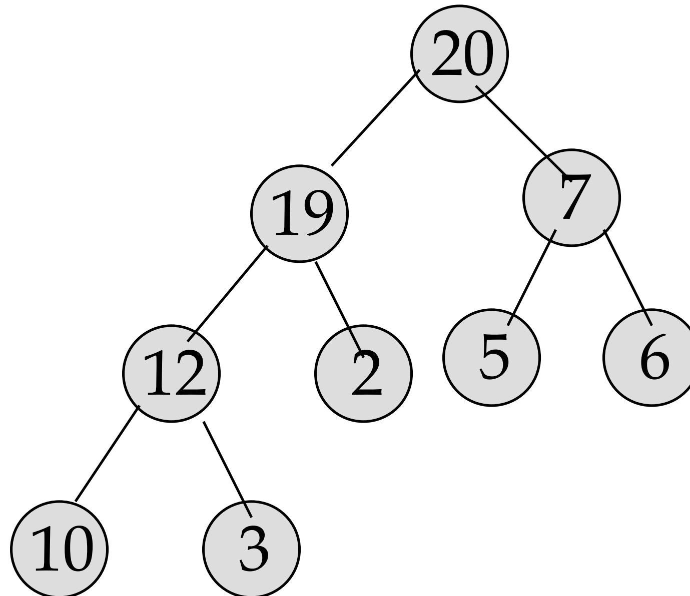
- If there is a vertex with only one child, it must be a left child, and the child must be the rightmost vertex on the lowest level.
- For a given number of vertices, there is only one legal structure

Heapsort: Heap examples



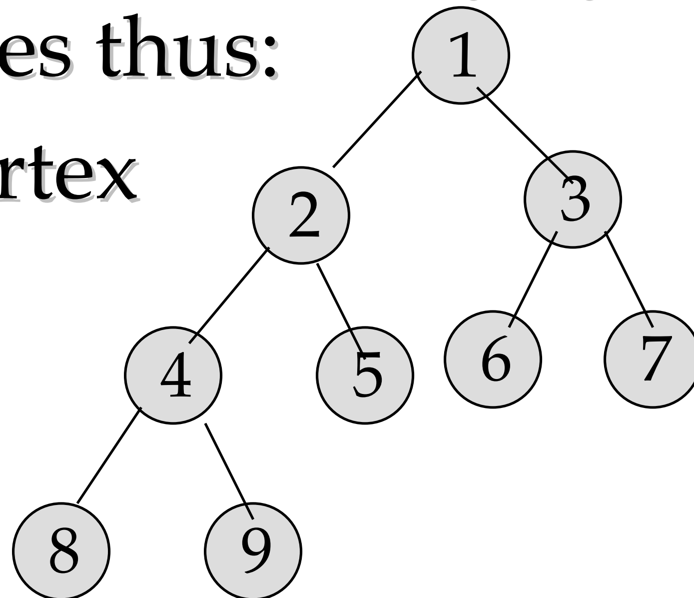
Heapsort: Heap Values

- 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 $\Theta(\lg n)$

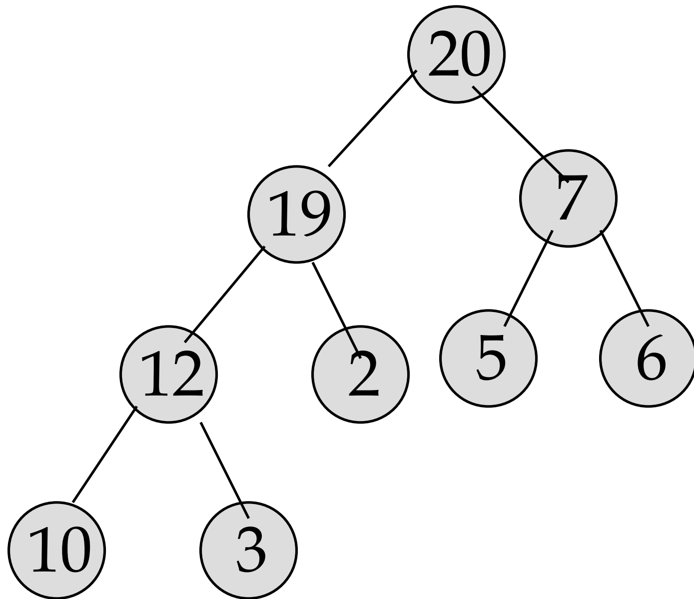
Heapsort: Creating a Heap

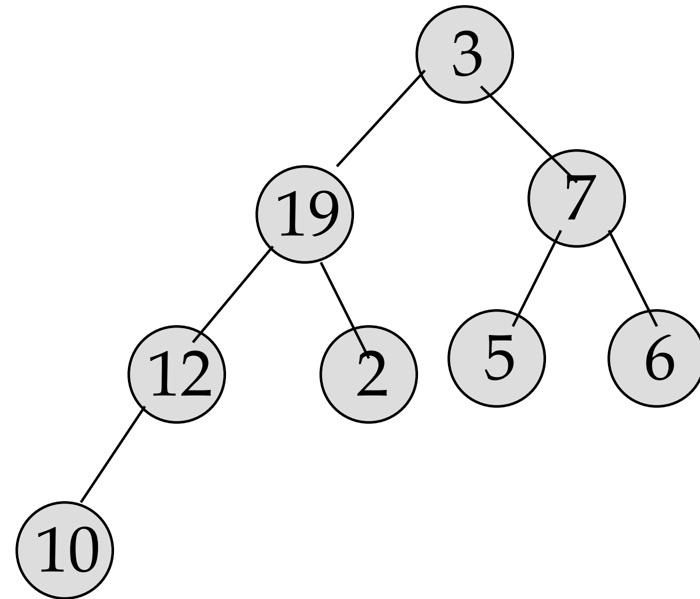
- An arbitrary list can be turned into a heap by calling FixHeap on each non-leaf in reverse order.
- If n is the size of the heap, the non-leaf with the highest index has index n/2.
- Creating a heap is obviously $O(n \lg n)$.
- A more careful analysis would show a true time bound of $\Theta(n)$

Heap Sort: Sorting

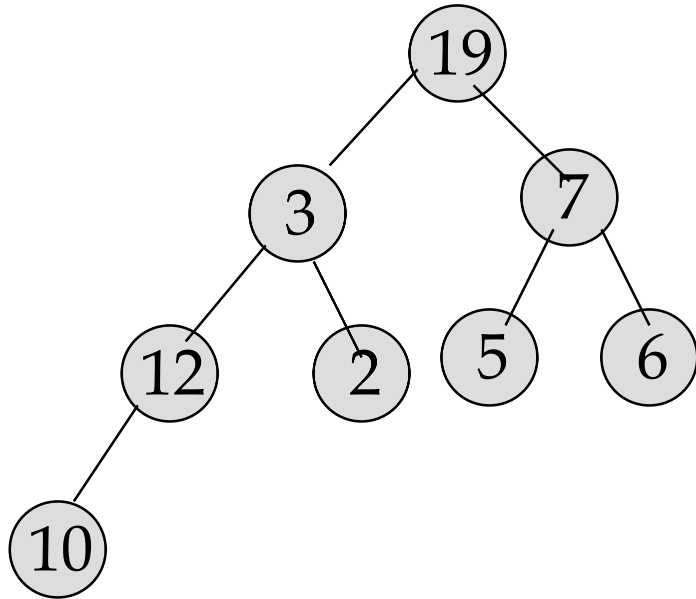
- Turn List into a Heap
- Swap head of list with last key in heap
- Reduce heap size by one
- Call FixHeap on the root
- Repeat for all keys until list is sorted

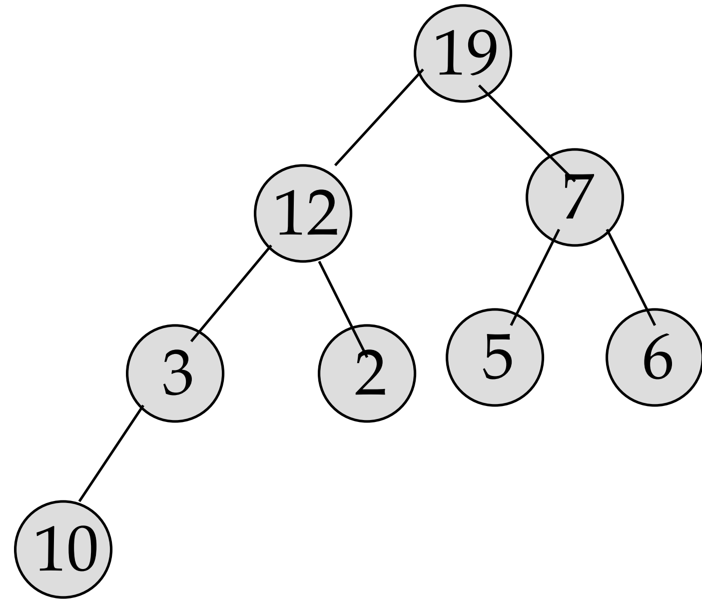
Sorting Example I





Sorting Example II

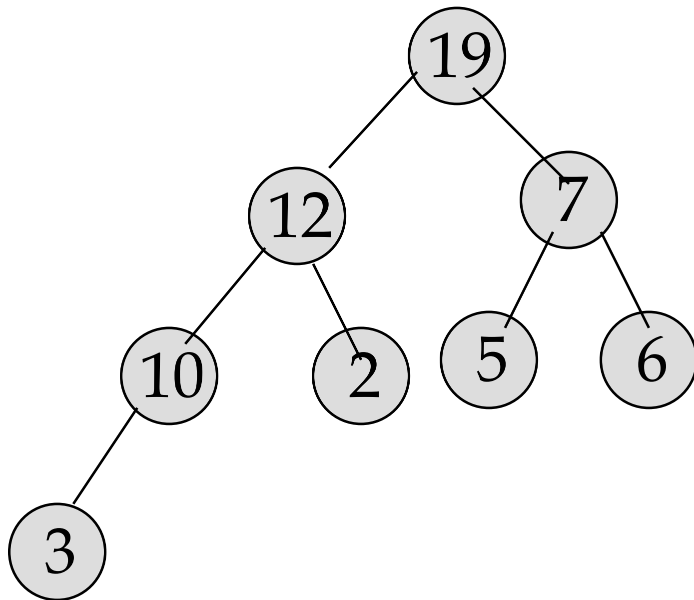




19	3	7	12	2	5	6	10	20

19	12	7	3	2	5	6	10	20

Sorting Example III



Ready to swap 3 and 19.

19	12	7	10	2	5	6	3	20

Heap Sort: Analysis

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

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

- 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 Assumptions II

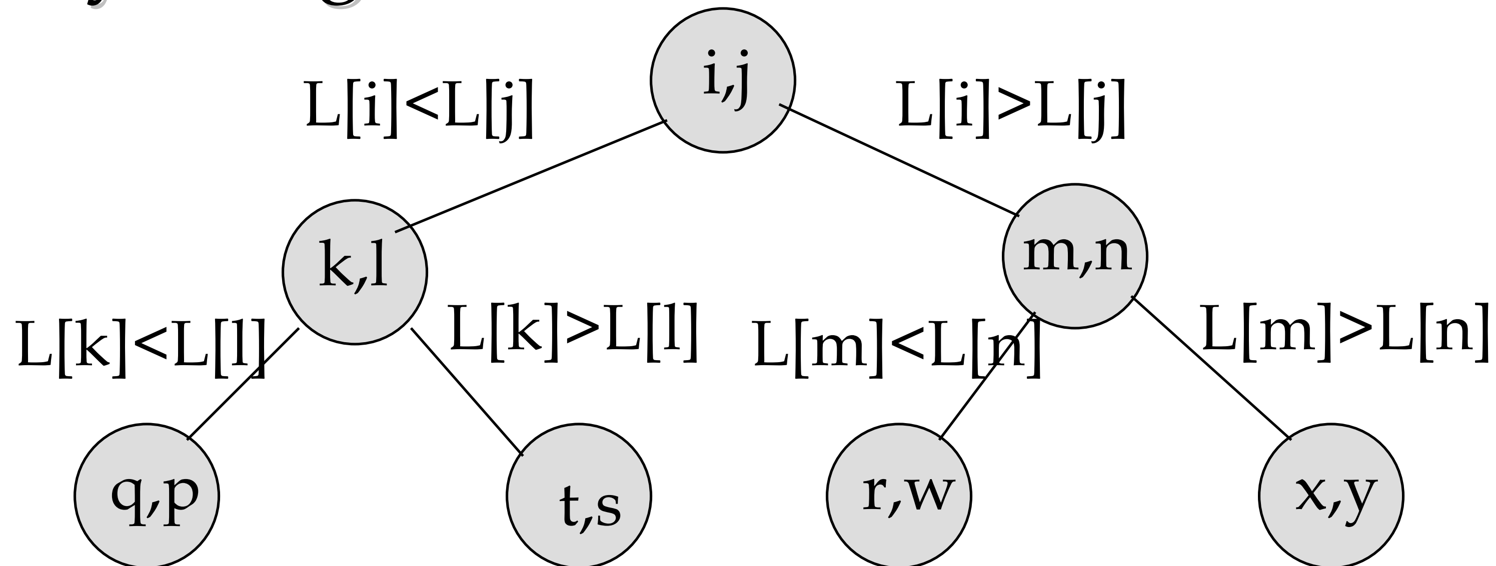
- Assume that all keys are distinct, since all sort algorithms must handle this case.
- Because there are no "tricks" that work, the only information we can get from a key comparison is:
- Which key is larger

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 Analysis

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

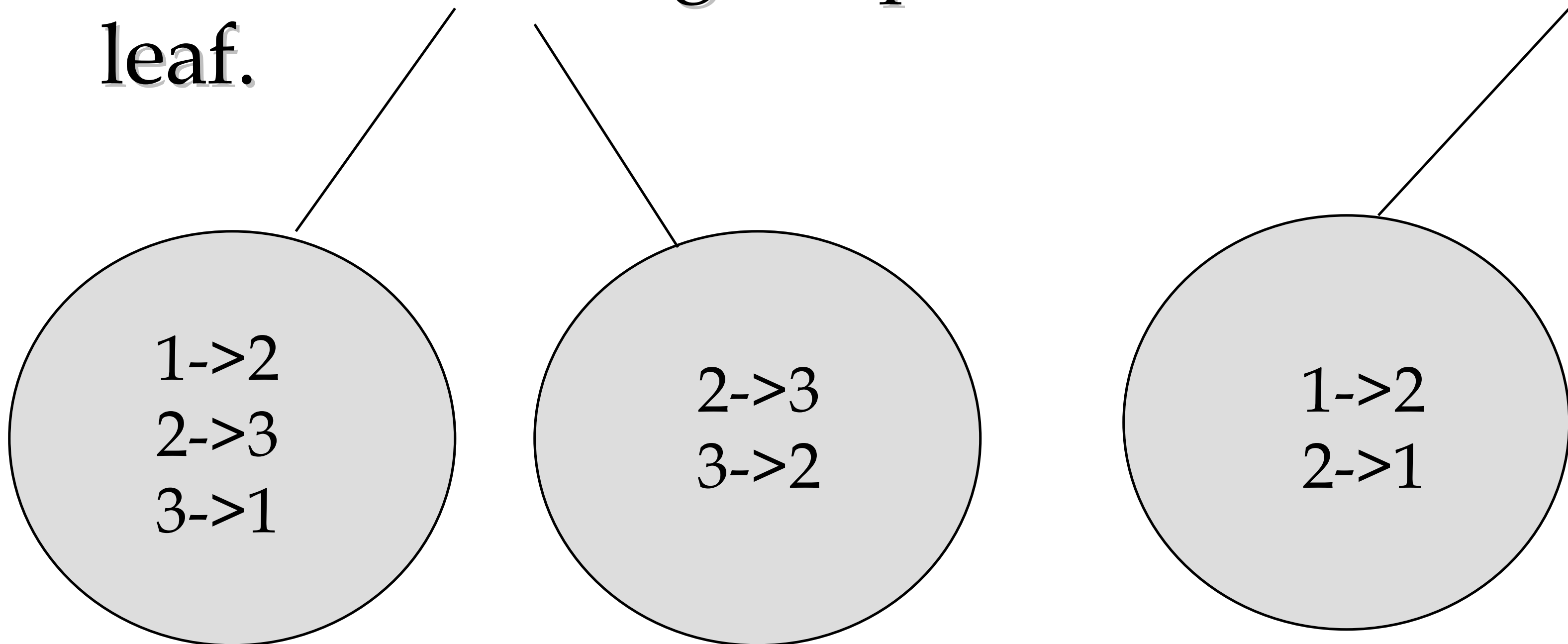


Lower Bound Analysis

- 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.



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

$$\lg n! = \sum_{i=2}^{n} \lg i = \frac{1}{\ln 2} \sum_{i=2}^{n} \ln i$$

$$\int_{1}^{n} \lg x \, dx \le \sum_{i=2}^{n} \lg i \le \int_{2}^{n+1} \lg x \, dx \qquad \int \ln x \, dx = x \ln x - x$$

$$n \ln n - n + 1 \le \sum_{i=2}^{n} \lg i \le (n+1) \ln(n+1) - n - 1 - 2 \ln 2 + 2$$

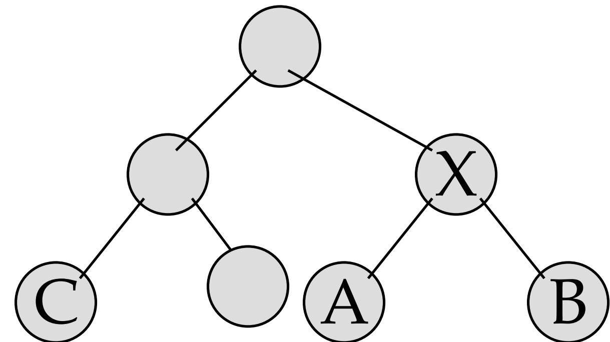
$$\Theta(n \ln n) \le \sum_{i=2}^{n} \lg i \le \Theta(n \ln n) \qquad \lg n! \in \Theta(n \lg n)$$

Lower Bound Average Case

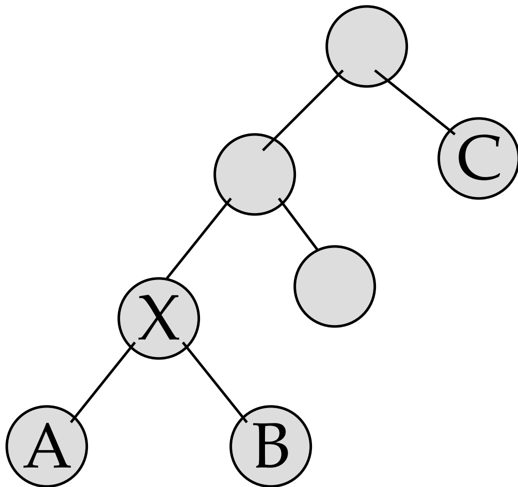
- Cannot be worse than worst case $\Theta(n \lg n)$
- Can it be better?
- To find average case, add up the lengths of all paths in the decision tree, and divide by the number of leaves.

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.



Switch X and C



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

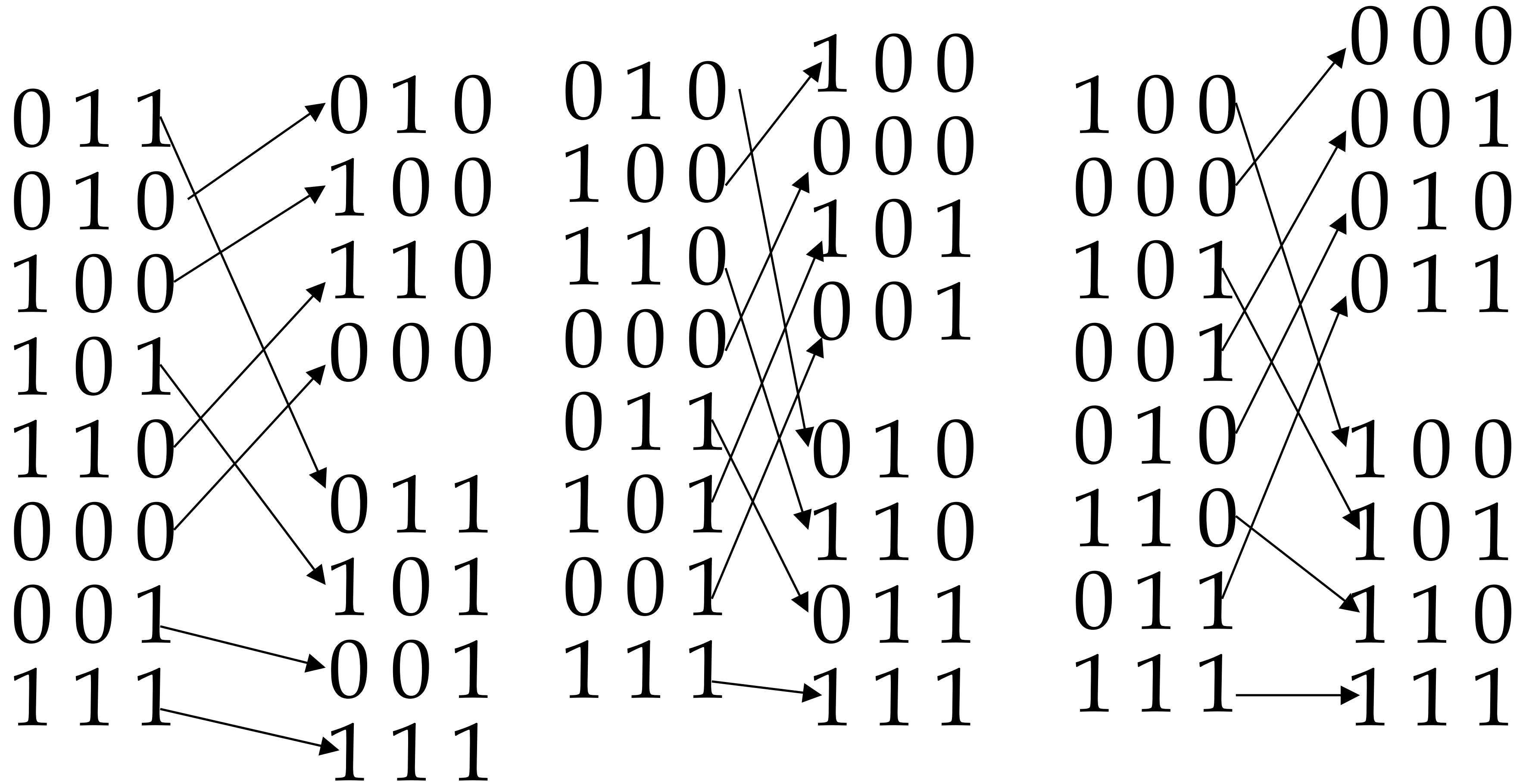
Lower Bound Avg. III

- 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 $\Theta(n \lg n)$

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



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.