Sorting

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2016

Sorting Algorithms

Sorting Algorithms

- We will study four sorting algorithms:

Sorting Algorithms

- We will study four sorting algorithms:
  - Insertion Sort

Sorting Algorithms

- We will study four sorting algorithms:
  - Insertion Sort
  - Quick Sort

Sorting Algorithms

- We will study four sorting algorithms:
  - Insertion Sort
  - Quick Sort
  - Heap Sort

Sorting Algorithms

- We will study four sorting algorithms:
  - Insertion Sort
  - Quick Sort
  - Heap Sort
  - Merge Sort

Sorting Algorithms

- We will study four sorting algorithms:
  - Insertion Sort
  - Quick Sort
  - Heap Sort
  - Merge Sort
- Each is useful in a different way.

Insertion Sort

Insertion Sort

- Insertion Sort

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

  `3 1 4 1 5 9 2 6`

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

  `3 1 4 1 5 9 2 6`

- Let's say we have sorted the first 6 elements and are inserting the 2.

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

  `3 1 4 1 5 9 2 6`

- Let's say we have sorted the first 6 elements and are inserting the 2.
- I put a dash in to indicate that we are ignoring the 6.

  `1 1 3 4 5 9 2 | 6`

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

  `3 1 4 1 5 9 2 6`
- Let's say we have sorted the first 6 elements and are inserting the 2.
- I put a dash in to indicate that we are ignoring the 6.

  `1 1 3 4 5 9 2 | 6`
- Take the 2 out and put it here: 2

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

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

- Let's say we have sorted the first 6 elements and are inserting the 2.
- I put a dash in to indicate that we are ignoring the 6.

```
1 1 3 4 5 9 2 | 6
```

- Take the 2 out and put it here: 2
- Copy elements forward until you get to where the 2 goes:

```
1 1 3 4 5 9 9 | 6
1 1 3 4 5 5 9 | 6
1 1 3 4 4 5 9 | 6
1 1 3 3 4 5 9 | 6
```

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

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

- Let's say we have sorted the first 6 elements and are inserting the 2.
- I put a dash in to indicate that we are ignoring the 6.

```
1 1 3 4 5 9 2 | 6
```

- Take the 2 out and put it here: 2
- Copy elements forward until you get to where the 2 goes:

```
1 1 3 4 5 9 9 | 6
1 1 3 4 5 5 9 | 6
1 1 3 4 4 5 9 | 6
1 1 3 3 4 5 9 | 6
```

- Now put the 2 back in:

```
1 1 2 3 4 5 9 | 6
```

Insertion Sort

- Insertion Sort
  - Inserts each element, one at a time, into a sorted array.
  - To insert an element, you move elements forward until you get to where it goes.
  - The trick is you do it multiple times.
  - The elements you are inserting are already in the array.
- Starting list:

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

- Let's say we have sorted the first 6 elements and are inserting the 2.
- I put a dash in to indicate that we are ignoring the 6.

```
1 1 3 4 5 9 2 | 6
```

- Take the 2 out and put it here: 2
- Copy elements forward until you get to where the 2 goes:

```
1 1 3 4 5 9 9 | 6
1 1 3 4 5 5 9 | 6
1 1 3 4 4 5 9 | 6
1 1 3 3 4 5 9 | 6
```

- Now put the 2 back in:

```
1 1 2 3 4 5 9 | 6
```

- We are ready to insert the 6.

Insertion Sort Properties

Insertion Sort Properties

- Good:

Insertion Sort Properties

- Good:
  - Easy to implement.

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if n is very small

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if *n* is very small
  - Fast on large *n* if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
- Bad:

Insertion Sort Properties

- Good:
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
- Bad:
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted

STABLE Sorting

STABLE Sorting

- What's the deal with STABLE?

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".
- But if the sort is STABLE, then it won't swap two pdf files from the first sort.

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".
- But if the sort is STABLE, then it won't swap two pdf files from the first sort.
- So the pdf files will be presented to me in order of increasing date, making it

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".
- But if the sort is STABLE, then it won't swap two pdf files from the first sort.
- So the pdf files will be presented to me in order of increasing date, making it
- easy to look for ones created in October.

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".
- But if the sort is STABLE, then it won't swap two pdf files from the first sort.
- So the pdf files will be presented to me in order of increasing date, making it
- easy to look for ones created in October.
- Is that how it happens in Finder on my Mac?

STABLE Sorting

- What's the deal with STABLE?
- Suppose I have a directory (folder) on my computer with a lot of files of all different sorts.
- I am looking for a pdf file created somewhere around last October.
  - Sort by date.
  - Sort by type (dmg, doc, pdf, txt, etc.)
- As far as the second sort is concerned, all pdf files are "equal".
- But if the sort is STABLE, then it won't swap two pdf files from the first sort.
- So the pdf files will be presented to me in order of increasing date, making it
- easy to look for ones created in October.
- Is that how it happens in Finder on my Mac?
- No!

Quick Sort

Quick Sort

- Quick Sort is recursive.

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.
  - Put it together.

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.
  - Put it together.
- Input:

  `3 1 4 1 5 9 2 6`

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.
  - Put it together.
- Input:

  `3 1 4 1 5 9 2 6`
- Pick 3 and partition the others

  `3`

  `1 1 2`

  `4 5 9 6`

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.
  - Put it together.
- Input:

  3 1 4 1 5 9 2 6

- Pick 3 and partition the others

  3
  1 1 2
  4 5 9 6

- Sort the other two groups recursively:

  3
  1 1 2
  4 5 6 9

Quick Sort

- Quick Sort is recursive.
  - Pick an pivot element, say the first one.
  - Compare all the other elements to it and separate into those <= and those >.
  - Sort those two groups recursively.
  - Put it together.
- Input:

  `3 1 4 1 5 9 2 6`
- Pick 3 and partition the others

  `3`

  `1 1 2`

  `4 5 9 6`
- Sort the other two groups recursively:

  `3`

  `1 1 2`

  `4 5 6 9`
- Put it together

  `1 1 2 3 4 5 6 9`

Quick Sort Running Time

Quick Sort Running Time

- Running time?

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.

Quick Sort Running Time

- Running time?
  - It takes n (actually $n-1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.
  - If the splits are even, then there are about $\log_2 n$ levels.

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.
  - If the splits are even, then there are about $\log_2 n$ levels.
- So $\mathrm{O}(n \log n)$.

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.
  - If the splits are even, then there are about $\log_2 n$ levels.
- So $\mathrm{O}(n \log n)$.
- But if the splits are very uneven, it could be $\mathrm{O}(n^2)$ again.

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.
  - If the splits are even, then there are about $\log_2 n$ levels.
- So $\mathrm{O}(n \log n)$.
- But if the splits are very uneven, it could be $\mathrm{O}(n^2)$ again.
- What is the worst possible input?

Quick Sort Running Time

- Running time?
  - It takes n (actually $n - 1$) comparisons to split.
  - Each level of the recursion uses less than n comparisons.
  - If the splits are even, then there are about $\log_2 n$ levels.
- So $\mathrm{O}(n \log n)$.
- But if the splits are very uneven, it could be $\mathrm{O}(n^2)$ again.
- What is the worst possible input?
- Why?

Partitioning in Place

Partitioning in Place

- Can Quick Sort be done without a second array?

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:
  - Everything to the left of i should be <=3.

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:
  - Everything to the left of i should be <=3.
  - Everything to the right of j should be >3.

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:
  - Everything to the left of i should be <=3.
  - Everything to the right of j should be >3.
- It is safe to increment i and decrement j:

```
3 1 4 1 5 9 2 6
    i       j
```

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:
  - Everything to the left of i should be <=3.
  - Everything to the right of j should be >3.
- It is safe to increment i and decrement j:

```
3 1 4 1 5 9 2 6
    i       j
```

- This is bad, we cannot increment i nor decrement j. What to do?

Partitioning in Place

- Can Quick Sort be done without a second array?
- Yes.

```
3 1 4 1 5 9 2 6
  i           j
```

- Invariants:
  - Everything to the left of i should be <=3.
  - Everything to the right of j should be >3.
- It is safe to increment i and decrement j:

```
3 1 4 1 5 9 2 6
    i       j
```

- This is bad, we cannot increment i nor decrement j. What to do?
- Swap!

```
3 1 2 1 5 9 4 6
    i       j

3 1 2 1 5 9 4 6
      i   j
```

Partitioning continued

Partitioning continued

- Eventually they pass each other!

```
3 1 2 1 5 9 4 6
        j
        i

3 1 2 1 5 9 4 6
      j i
```

Partitioning continued

- Eventually they pass each other!

```
3 1 2 1 5 9 4 6
        j
        i

3 1 2 1 5 9 4 6
      j i
```

- Now we swap [0] and [j]

```
1 1 2 3 5 9 4 6
      j i
```

Partitioning continued

- Eventually they pass each other!

```
3 1 2 1 5 9 4 6
        j
        i


3 1 2 1 5 9 4 6
      j i
```

- Now we swap [0] and [j]

```
1 1 2 3 5 9 4 6
      j i
```

- Recursively sort 0 to j-1 and i to size-1

```
1 1 2 3 4 5 6 9
      j i
```

Partitioning continued

- Eventually they pass each other!

```
3 1 2 1 5 9 4 6
        j
        i

3 1 2 1 5 9 4 6
      j i
```

- Now we swap [0] and [j]

```
1 1 2 3 5 9 4 6
      j i
```

- Recursively sort 0 to j-1 and i to size-1

```
1 1 2 3 4 5 6 9
      j i
```

- Done!

Quick Sort Properties

Quick Sort Properties

- Good:

Quick Sort Properties

- Good:
  - $\mathrm{O}(n \log n)$ on average

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:
  - $O(n^2)$ if input is sorted.

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:
  - $O(n^2)$ if input is sorted.
  - If you do it IN PLACE then it won't be STABLE

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:
  - $O(n^2)$ if input is sorted.
  - If you do it IN PLACE then it won't be STABLE
- Easy to fix $O(n^2)$ case:

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:
  - $O(n^2)$ if input is sorted.
  - If you do it IN PLACE then it won't be STABLE
- Easy to fix $O(n^2)$ case:
  - just swap first element with random element

Quick Sort Properties

- Good:
  - $O(n \log n)$ on average
  - Fastest in practice
- Bad:
  - $O(n^2)$ if input is sorted.
  - If you do it IN PLACE then it won't be STABLE
- Easy to fix $O(n^2)$ case:
  - just swap first element with random element
  - or just the middle element

Heap Sort

Heap Sort

- Heap Sort uses the heap idea.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $\mathrm{O}(n \log n)$ sorting algorithm. Guaranteed!

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $\mathrm{O}(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $O(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $O(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $\mathrm{O}(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.
  2. Each poll puts the polled element into the spot just vacated.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $O(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $O(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.
  2. Each poll puts the polled element into the spot just vacated.
- To heapify in place, work from the bottom.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $O(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $O(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.
  2. Each poll puts the polled element into the spot just vacated.
- To heapify in place, work from the bottom.
- Remember: even thought I am writing it like a tree, it is still just an array.

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

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $\mathrm{O}(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $\mathrm{O}(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.
  2. Each poll puts the polled element into the spot just vacated.
- To heapify in place, work from the bottom.
- Remember: even thought I am writing it like a tree, it is still just an array.

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

- The 6 has no kids, and neither do 2, 9, nor 5.

Heap Sort

- Heap Sort uses the heap idea.
  - We already know that we can insert and remove from a heap in $O(\log n)$ time.
  - So insert n elements and remove them, and they will be sorted.
  - Instant $O(n \log n)$ sorting algorithm. Guaranteed!
- There are two additional tricks.
  1. We can heapify the contents of an array in place.
  2. Each poll puts the polled element into the spot just vacated.
- To heapify in place, work from the bottom.
- Remember: even thought I am writing it like a tree, it is still just an array.

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

- The 6 has no kids, and neither do 2, 9, nor 5.
- 1 has 6 as a kid, which is o.k.

Heapifying in Place continued

Heapifying in Place continued

- 4 has 9 and 2, not good.

Heapifying in Place continued

- 4 has 9 and 2, not good.
- Swap 4 and 2.

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

Heapifying in Place continued

- 4 has 9 and 2, not good.
- Swap 4 and 2.

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

- 1 is o.k. (kids are 1 and 5). 3 is not. Swap with 1:

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

Heapifying in Place continued

- 4 has 9 and 2, not good.
- Swap 4 and 2.

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

- 1 is o.k. (kids are 1 and 5). 3 is not. Swap with 1:

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

- Still not good, swap with 1 again:

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

Heapifying in Place continued

- 4 has 9 and 2, not good.
- Swap 4 and 2.

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

- 1 is o.k. (kids are 1 and 5). 3 is not. Swap with 1:

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

- Still not good, swap with 1 again:

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

- Now it is a heap.

Heap Sort Polling Phase

Heap Sort Polling Phase

- Now, let's remove the root and put it in the last element.

Heap Sort Polling Phase

- Now, let’s remove the root and put it in the last element.
- We were going to put the 6 at the root for the removal process anyway

Heap Sort Polling Phase

- Now, let's remove the root and put it in the last element.
- We were going to put the 6 at the root for the removal process anyway
- So swap them:

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

Heap Sort Polling Phase

- Now, let's remove the root and put it in the last element.
- We were going to put the 6 at the root for the removal process anyway
- So swap them:

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

- Now swap down the 6, but ignore the 1 at the bottom. (Decrement size.)

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

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

Polling Phase continued

Polling Phase continued

- Swap the 1 and the last element, which is the 4 now, and ignore that 1 thereafter (decrement size):

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

Polling Phase continued

- Swap the 1 and the last element, which is the 4 now, and ignore that 1 thereafter (decrement size):

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

- Fix the 4:

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

Polling Phase continued

- Swap the 1 and the last element, which is the 4 now, and ignore that 1 thereafter (decrement size):

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

- Fix the 4:

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

- Can you continue?

Polling Phase continued

- Swap the 1 and the last element, which is the 4 now, and ignore that 1 thereafter (decrement size):

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

- Fix the 4:

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

- Can you continue?
- The result is the array sorted in reverse order.

Polling Phase continued

- Swap the 1 and the last element, which is the 4 now, and ignore that 1 thereafter (decrement size):

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

- Fix the 4:

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

- Can you continue?
- The result is the array sorted in reverse order.
- But if you can do that, you can do it right!

Heap Sort Properties

Heap Sort Properties

- Good:

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
- Bad:

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
- Bad:
  - not stable

Heap Sort Properties

- Good:
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
- Bad:
  - not stable
  - apparently slower than quick sort in practice

Merge Sort

Merge Sort

- Merge Sort is a little like quick sort but backwards.

Merge Sort

- Merge Sort is a little like quick sort but backwards.
- Just split the array in two:

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

Merge Sort

- Merge Sort is a little like quick sort but backwards.
- Just split the array in two:

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

- Sort each recursively:

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

Merging

Merging

- Now merge them. You only have to look at the front of each list:

```
1 3 4
2 5 6 9
1

3 4
2 5 6 9
1 1

3 4
5 6 9
1 1 2

4
5 6 9
1 1 2 3


5 6 9
1 1 2 3 4
```

Merging continued

Merging continued

- Since the first list is empty, we can just copy the rest of the second list:

  `1 1 2 3 4 5 6 9`

Merge Sort Properties

Merge Sort Properties

- Good:

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists

Merge Sort Properties

- Good:
  - $\mathrm{O}(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:

Merge Sort Properties

- Good:
  - $\mathrm{O}(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:
  - Notice that when we merge two lists together,

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:
  - Notice that when we merge two lists together,
  - we only access and/or remove the head of each (half) list

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:
  - Notice that when we merge two lists together,
  - we only access and/or remove the head of each (half) list
  - and add at the tail of the merged list.

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:
  - Notice that when we merge two lists together,
  - we only access and/or remove the head of each (half) list
  - and add at the tail of the merged list.
  - These are $O(1)$ operations for a linked list.

Merge Sort Properties

- Good:
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
- Bad:
  - Very hard to do in place
- Regarding linked lists:
  - Notice that when we merge two lists together,
  - we only access and/or remove the head of each (half) list
  - and add at the tail of the merged list.
  - These are $O(1)$ operations for a linked list.
- So the running time is the same.

External Sorting

External Sorting

- Here is how to do it on the hard disk.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.
  - Read in 4 and 1 and write out 14.

External Sorting

- Here is how to do it on the hard disk.
    - This is sometimes called "out of core" computing.
    - "Core" is what they used to call RAM.
    - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
    - 3452
    - 1196
- Open both files and read elements from the files, merging them and writing them out.
    - Read in 3 and 1 and write out 13.
    - Read in 4 and 1 and write out 14.
    - Read in 5 and 9 and write out 59.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.
  - Read in 4 and 1 and write out 14.
  - Read in 5 and 9 and write out 59.
  - Read in 2 and 6 and write out 26.

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.
  - Read in 4 and 1 and write out 14.
  - Read in 5 and 9 and write out 59.
  - Read in 2 and 6 and write out 26.
- “Deal out” to different files:

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.
  - Read in 4 and 1 and write out 14.
  - Read in 5 and 9 and write out 59.
  - Read in 2 and 6 and write out 26.
- “Deal out” to different files:
  - 1359

External Sorting

- Here is how to do it on the hard disk.
  - This is sometimes called "out of core" computing.
  - "Core" is what they used to call RAM.
  - This is a “Big Data” technique.
- First, let’s assume we have FOUR hard drives connected to the computer. (Not so unusual.)
- “Deal out” the elements of the file to two different files on different hard disks.
  - 3452
  - 1196
- Open both files and read elements from the files, merging them and writing them out.
  - Read in 3 and 1 and write out 13.
  - Read in 4 and 1 and write out 14.
  - Read in 5 and 9 and write out 59.
  - Read in 2 and 6 and write out 26.
- “Deal out” to different files:
  - 1359
  - 1426

External Sorting continued

External Sorting continued

- Next we will merge groups of two.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.
  - The 4 is smaller so write it out.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.
  - The 4 is smaller so write it out.
  - Write out the 5 and read in the 6.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.
  - The 4 is smaller so write it out.
  - Write out the 5 and read in the 6.
  - Write out the 6 and read in the 9.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.
  - The 4 is smaller so write it out.
  - Write out the 5 and read in the 6.
  - Write out the 6 and read in the 9.
  - Write out the 9.

External Sorting continued

- Next we will merge groups of two.
  - Read in 1 and 1.
  - The first 1 wins the tie so write it out and read in 3.
  - The second 1 is smaller so write it out and read in 4.
  - The 3 is smaller so write it out, but don't read any more
  - because that group of two is done.
  - The 4 is all we have left, so write it out.
- So now we have 1134 in one file.
  - Read in the 5 and 2.
  - The 2 is smaller so write it out and read in the 6.
  - The 5 is smaller so write it out and read in 9.
  - The 6 is smaller so write it out.
  - Write out the 9.
  - We have 2569.
- Now we need to merge 1134 and 2569 into a single file.
  - Read in the 1 and 2.
  - The 1 is smaller so write it out and read in 1.
  - The 1 is smaller so write it out and read in 3.
  - The 2 is smaller so write it out and read in 5.
  - The 3 is smaller so write it out and read in 4.
  - The 4 is smaller so write it out.
  - Write out the 5 and read in the 6.
  - Write out the 6 and read in the 9.
  - Write out the 9.
- Result: 11234569.

External Sorting Running Time

External Sorting Running Time

- Time?

External Sorting Running Time

- Time?
  - We read through each file sequentially, which is very fast.

External Sorting Running Time

- Time?
  - We read through each file sequentially, which is very fast.
  - Just put the read-head in the right place and spin the disk.

External Sorting Running Time

- Time?
  - We read through each file sequentially, which is very fast.
  - Just put the read-head in the right place and spin the disk.
  - We have to do $\log_2 n$ rounds (why?).

External Sorting Running Time

- Time?
  - We read through each file sequentially, which is very fast.
  - Just put the read-head in the right place and spin the disk.
  - We have to do $\log_2 n$ rounds (why?).
- So $O(n \log n)$.

Summary

Summary

- Insertion Sort

Summary

- Insertion Sort
  - Easy to implement.

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don’t randomize somehow.

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice
  - $\mathrm{O}(n^2)$ if input is sorted and you don’t randomize somehow.
  - If you do it IN PLACE then it won’t be STABLE
- Heap Sort
  - Guaranteed $\mathrm{O}(n \log n)$

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don’t randomize somehow.
  - If you do it IN PLACE then it won’t be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.
  - IN PLACE

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.
  - IN PLACE
  - not stable

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice
  - $\mathrm{O}(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is “almost sorted”
  - STABLE: doesn’t flip elements if it doesn’t have to
  - IN PLACE: doesn’t require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don’t randomize somehow.
  - If you do it IN PLACE then it won’t be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice
  - $\mathrm{O}(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort
  - $\mathrm{O}(n \log n)$ guaranteed

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice
  - $\mathrm{O}(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort
  - $\mathrm{O}(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $O(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $O(n \log n)$ on average
  - Fastest in practice
  - $O(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $O(n \log n)$
  - Heapifying is $O(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort
  - $O(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks

Summary

- Insertion Sort
  - Easy to implement.
  - Fast if n is very small
  - Fast on large n if input is "almost sorted"
  - STABLE: doesn't flip elements if it doesn't have to
  - IN PLACE: doesn't require a second array
  - $\mathrm{O}(n^2)$ running time in general, so slow on large n when input is not nearly sorted
- Quick Sort
  - $\mathrm{O}(n \log n)$ on average
  - Fastest in practice
  - $\mathrm{O}(n^2)$ if input is sorted and you don't randomize somehow.
  - If you do it IN PLACE then it won't be STABLE
- Heap Sort
  - Guaranteed $\mathrm{O}(n \log n)$
  - Heapifying is $\mathrm{O}(n)$, actually.
  - IN PLACE
  - not stable
  - apparently slower than quick sort in practice
- Merge Sort
  - $\mathrm{O}(n \log n)$ guaranteed
  - STABLE if you break ties correctly
  - Works great for sorting linked lists
  - Works great for sorting files on hard disks
  - Very hard to do in place