Data Structures and Algorithm Analysis (CSC317)

Introduction: sorting as example

Sorting

- We’re looking at sorting as an example of developing an algorithm and analyzing run time

Sorting as example: Insertion sort

Animation example:

http://cs.armstrong.edu/liang/animation/web/InsertionSortNew.html

Insertion sort: analysis of run time

- We've slightly simplified notation from book – don't care if repeated 2 or 3 constant times…
- We only care about limiting step relative to input size n – ignore constant and lower order terms (always true?)
- We often care about worst case scenario
- Average case often roughly as bad as worst case

Insertion sort: summary analysis of run time

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1]
   by comparing and swapping into correct position

WORST case total:

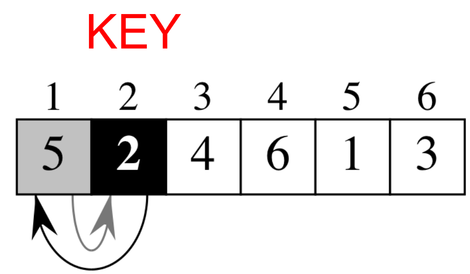
$$T(n) = c_1 n + c_2 n + c_3\left(\frac{n^2}{2} - \frac{n}{2}\right)$$

We'll usually ignore constants and lower order terms for the form
$an^2 + bn + c$ (why?)

grows like n^2

Insertion sort: full pseudo code

- Insertion sort pseudo code

(a) 


```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Sorting as example

- Insertion sort “grows like” n^2
- Can we do better??

Sorting as example

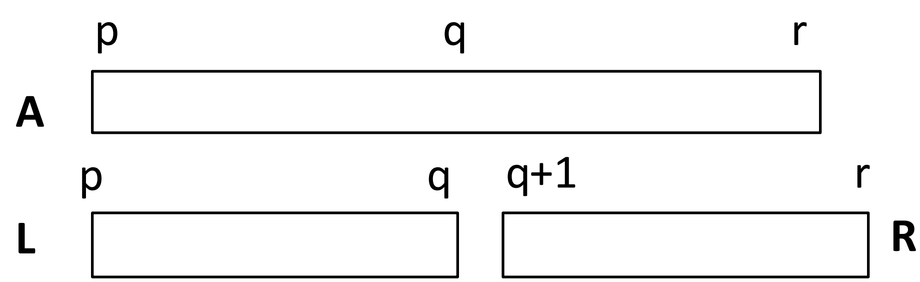
- Insertion sort "grows like" n^2

- Can we do better??

Yes, cheaper by halves…
http://www.cs.miami.edu/~burt/learning/Csc517.101/workbook/cheaperbyhalf.html

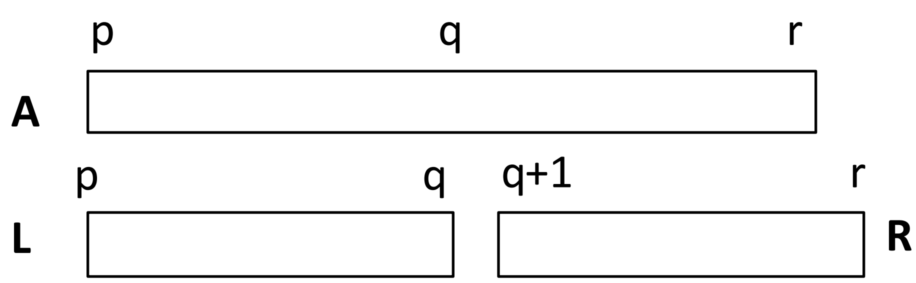
(show on the board with example)

Sorting as example: Towards Merge sort




```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Sorting as example: Towards Merge sort



Animation example of merge:

http://cs.armstrong.edu/liang/animation/web/MergeSortNew.html

Sorting as example: Towards Merge sort

- Splitting array in half might pose an easier problem…

Costs:

- Divide Left and Right: constant c_1
- Sort Left: with Insertion Sort $c_2\left(\frac{n}{2}\right)^2$
- Sort Right: with Insertion Sort $c_2\left(\frac{n}{2}\right)^2$
- Merge $c_3 n$
- Total: $c_1 + 2c_2\left(\frac{n}{2}\right)^2 + c_3 n$

  (Have we gained from Insertion sort? cn^2)

Another example: find min

Input: Array A
Output: Find minimum value in array A

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Another example: find min

- Splitting array in half might pose an easier problem… (does it always?)

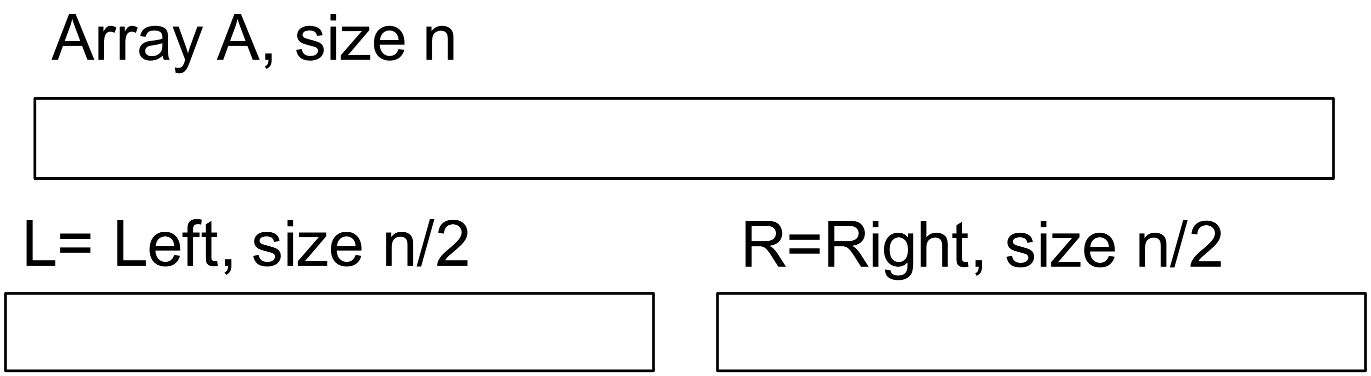
Costs:

- Divide Left and Right: constant c_1
- Sort Left: with Insertion Sort $c_2\left(\frac{n}{2}\right)$
- Sort Right: with Insertion Sort $c_2\left(\frac{n}{2}\right)$
- Merge $c_3 n$
- Total: $c_1 + 2c_2\left(\frac{n}{2}\right) + c_3 n$

  (Have we gained from find min on full array? cn)

Sorting as example: BACK TO Merge sort

- Splitting array in half might pose an easier problem…



If we can split once and make the problem easier,
we can continue to do so….

(on the board)

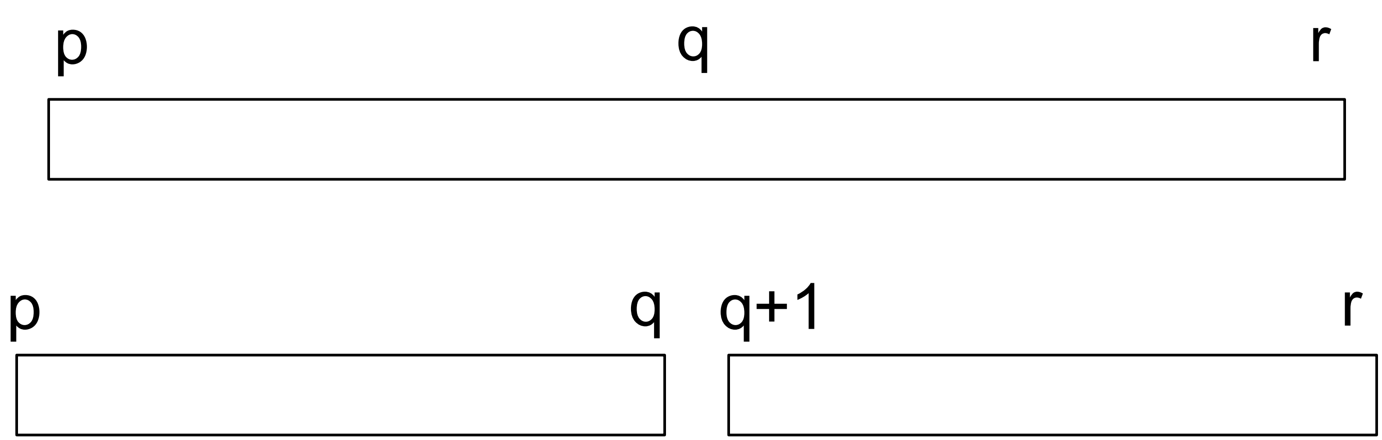
Merge sort: high level pseudo code

Merge-Sort

```
If array larger than size 1
    Divide array into Left and Right arrays // divide
    Merge-Sort(Left array)  // conquer left; recursive call
    Merge-Sort(Right array) // conquer right; recursive call
    Merge sorted Left and Right arrays // combine
```

Merge sort: pseudo code

```
MERGE-SORT(A, p, r)
if p < r
        q = ⌊(p + r)/2⌋               // divide
        MERGE-SORT(A, p, q)           // conquer left
        MERGE-SORT(A, q + 1, r)       // conquer right
        MERGE(A, p, q, r)             // combine
```



Merge sort: run time analysis

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋            // divide
    MERGE-SORT(A, p, q)        // conquer left
    MERGE-SORT(A, q + 1, r)    // conquer right
    MERGE(A, p, q, r)          // combine
```

Question:

At what step do we have most work?

Merge sort: run time analysis

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋             // divide
    MERGE-SORT(A, p, q)          // conquer left
    MERGE-SORT(A, q + 1, r)      // conquer right
    MERGE(A, p, q, r)            // combine
```

Question:
At what step do we have most work?

In the Merge (combine) step; the rest is just splitting arrays…

Merge sort: run time analysis

Total work:

Divide: constant

Combine: cn

Conquer: recursively solve two subproblems, each size n/2

Merge sort: run time analysis

Total work:

Divide: constant
Combine: cn
Conquer: recursively solve two subproblems, each size n/2

We'll write out the recursion as follows:

$$T(n) = 2T(\frac{n}{2}) + cn$$

recursion combine

Merge sort: run time analysis

Total work:

Divide: constant
Combine: cn
Conquer: recursively solve two subproblems, each size n/2

We'll write out the recursion as follows:

$$T(n) = 2T(\frac{n}{2}) + cn$$

recursion combine

Total: grows like $n\log_2(n)$

Good deal!
Compare to insertion sort

Merge sort: run time analysis

$$T(n) = 2T(\frac{n}{2}) + cn$$

recursion combine

Total: grows like $n\log_2(n)$

Good deal!
Compare to insertion sort

How did we get this?
Easy way to see is recursion tree

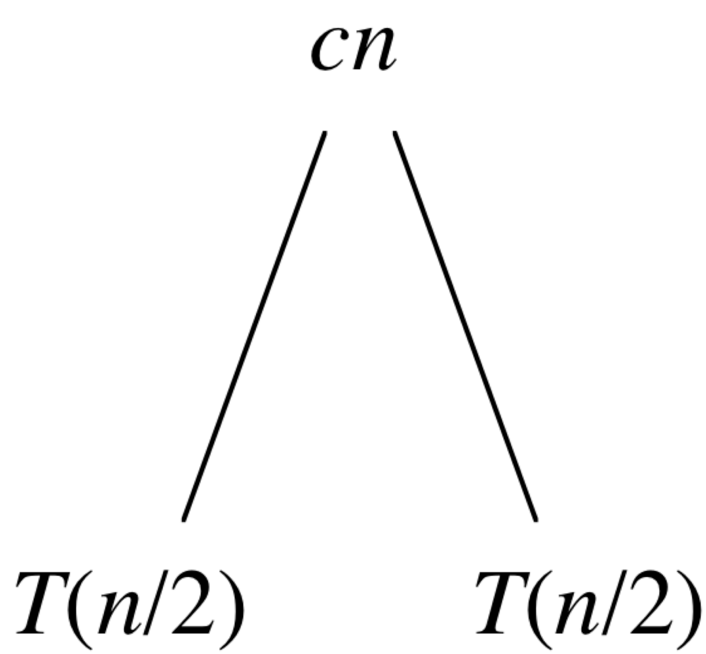
(On the board…)

Merge sort: recursion tree

$$T(n) = 2T(\frac{n}{2}) + cn$$

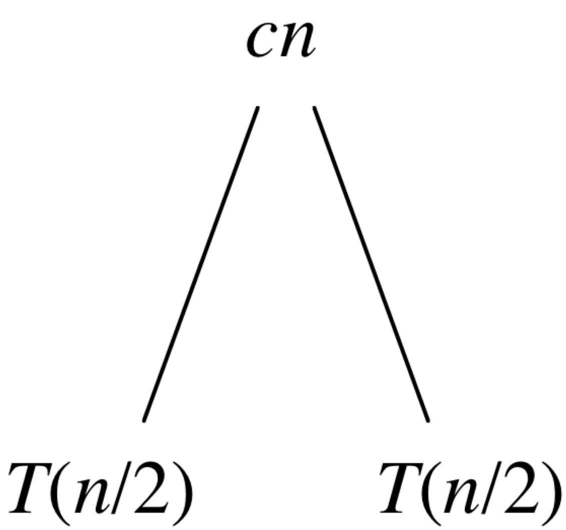
recursion combine

$T(n)$



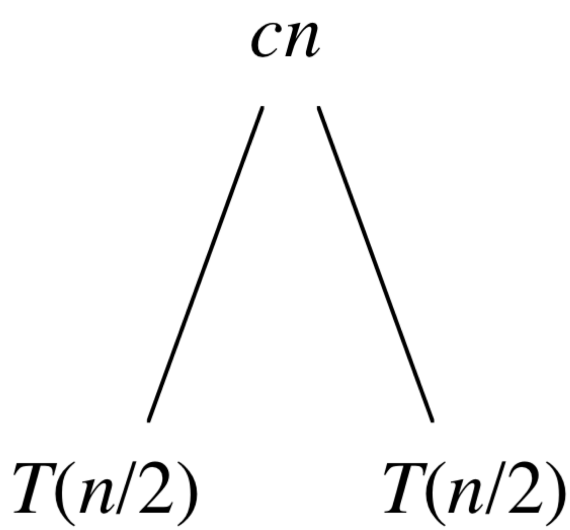
Merge sort: recursion tree

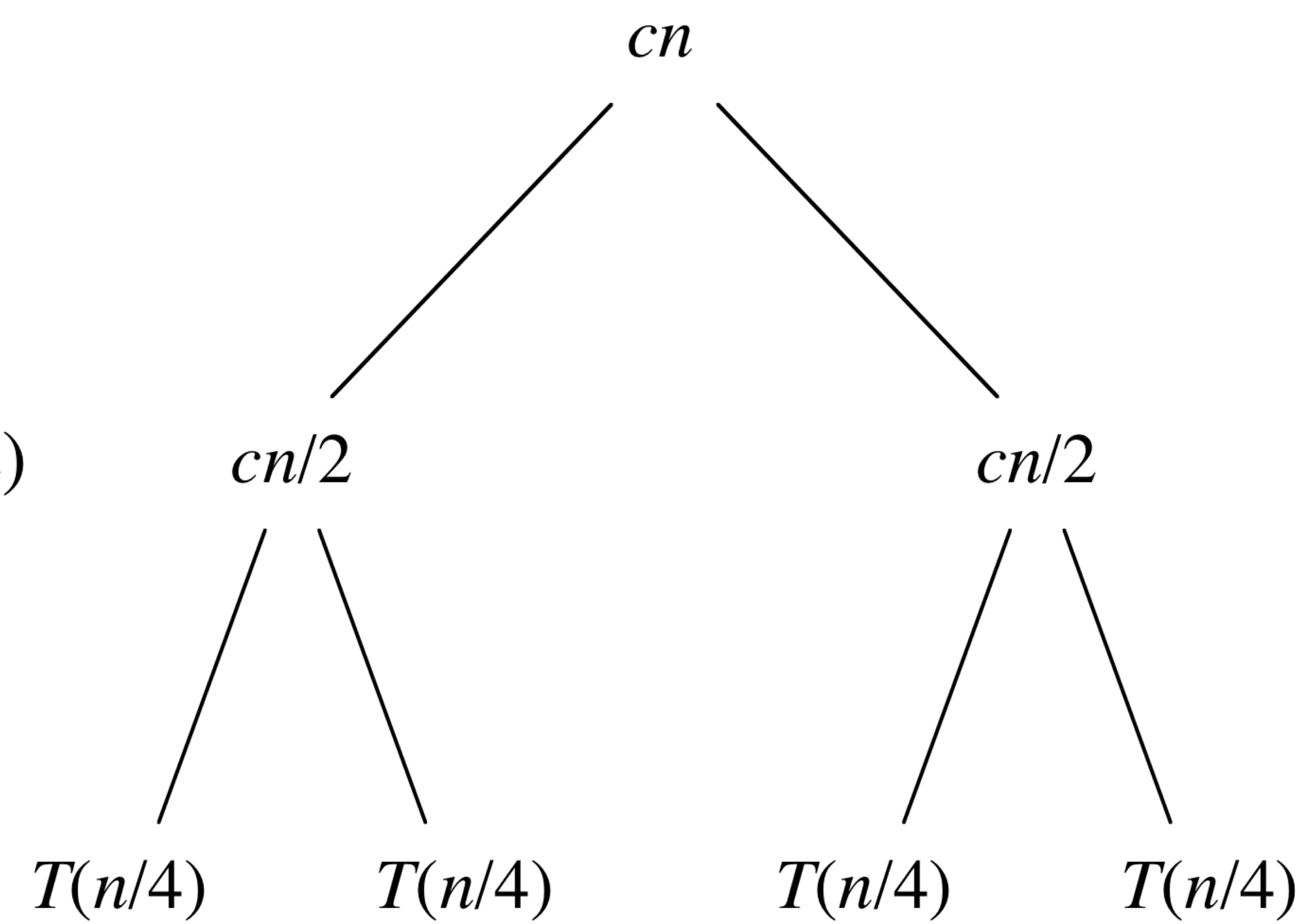
Let's expand each T(n/2): $T(n/2) = 2T(\frac{n/2}{2}) + c(n/2)$



Merge sort: recursion tree

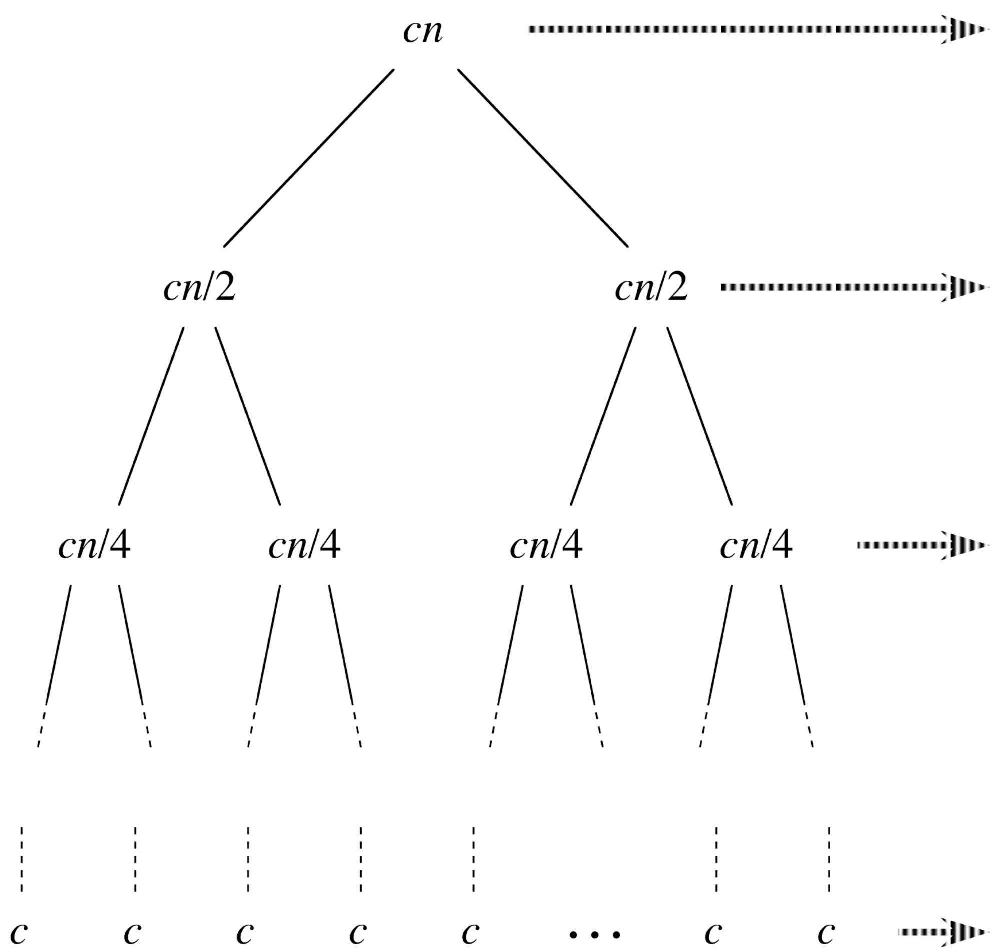
Let's expand each T(n/2): $T(n/2) = 2T(\frac{n/2}{2}) + c(n/2)$





Merge sort: recursion tree

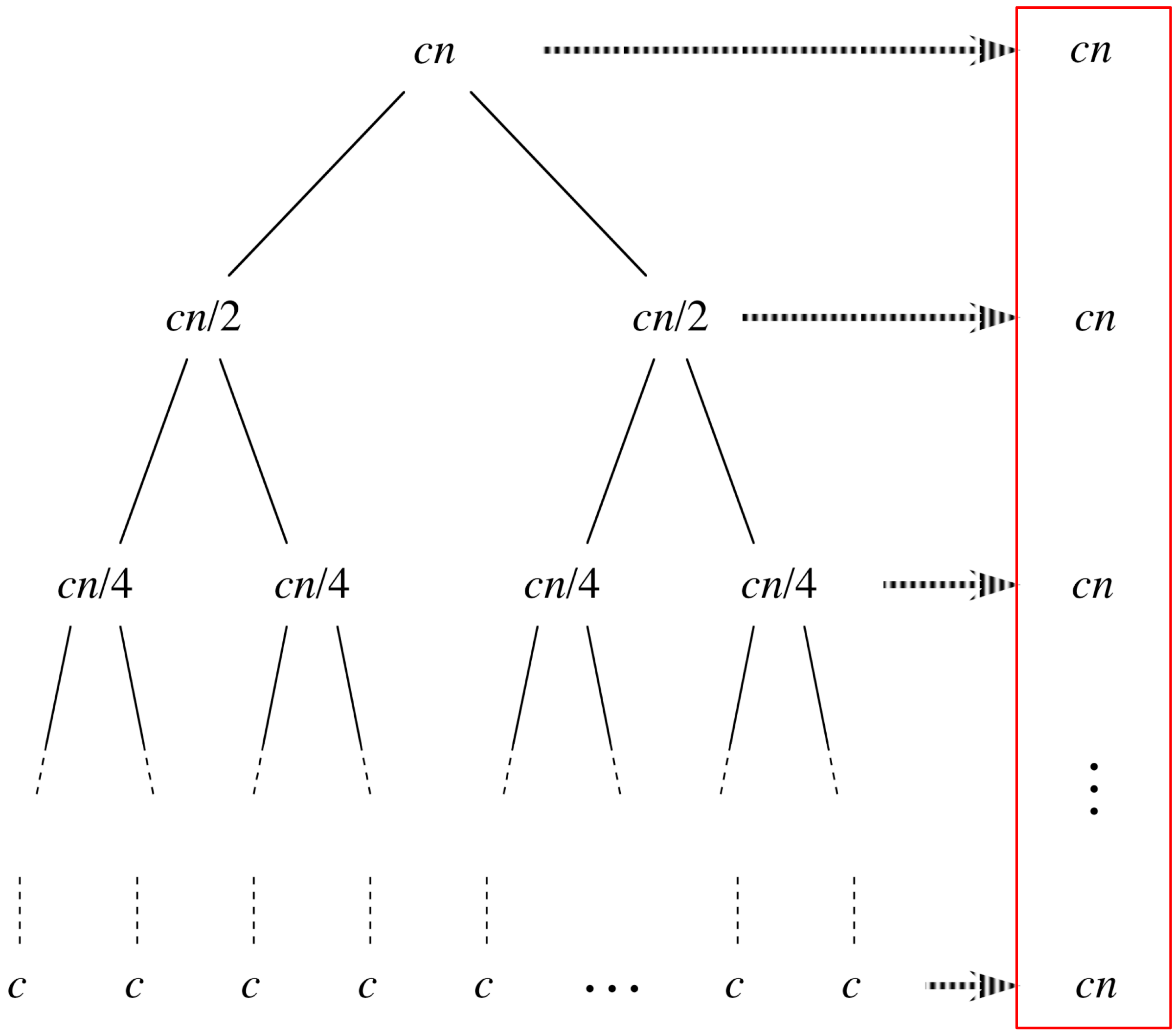
Let's keep expanding…



Each row adds to
How much work?

Merge sort: recursion tree

Let's keep expanding…



Cost per level stays the same!

So what is the Total cost?

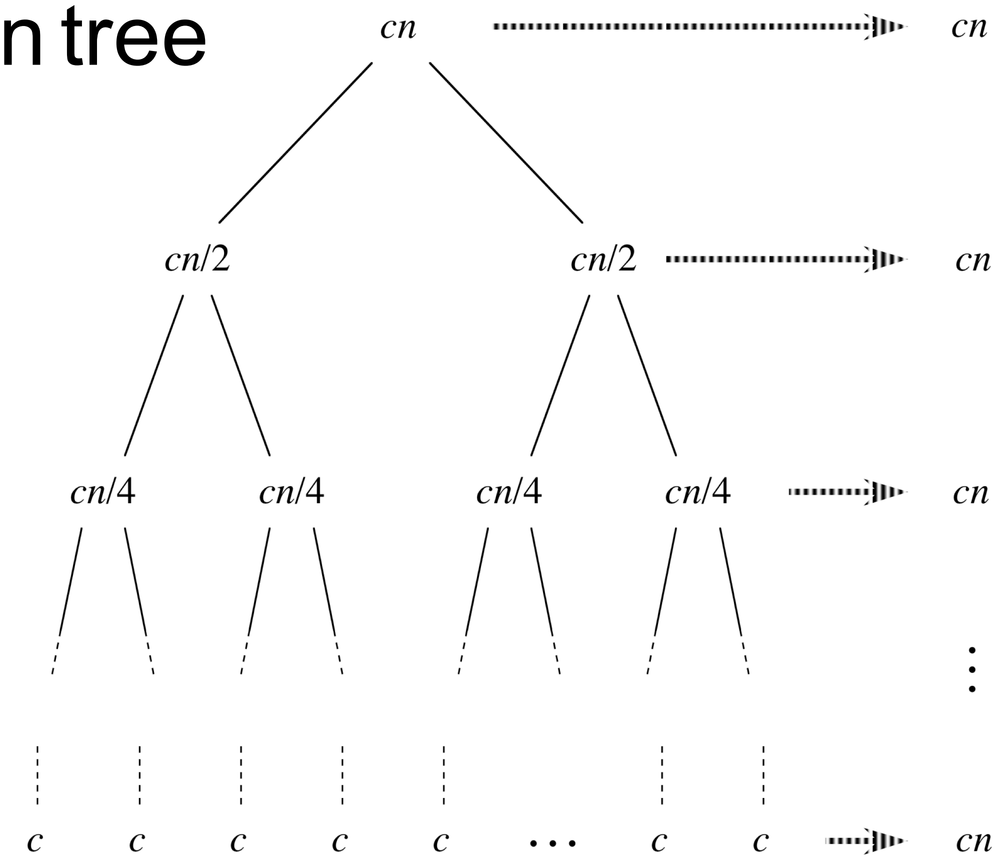
Merge sort: recursion tree

Level 1: $\frac{n}{2} = \frac{n}{2^1}$

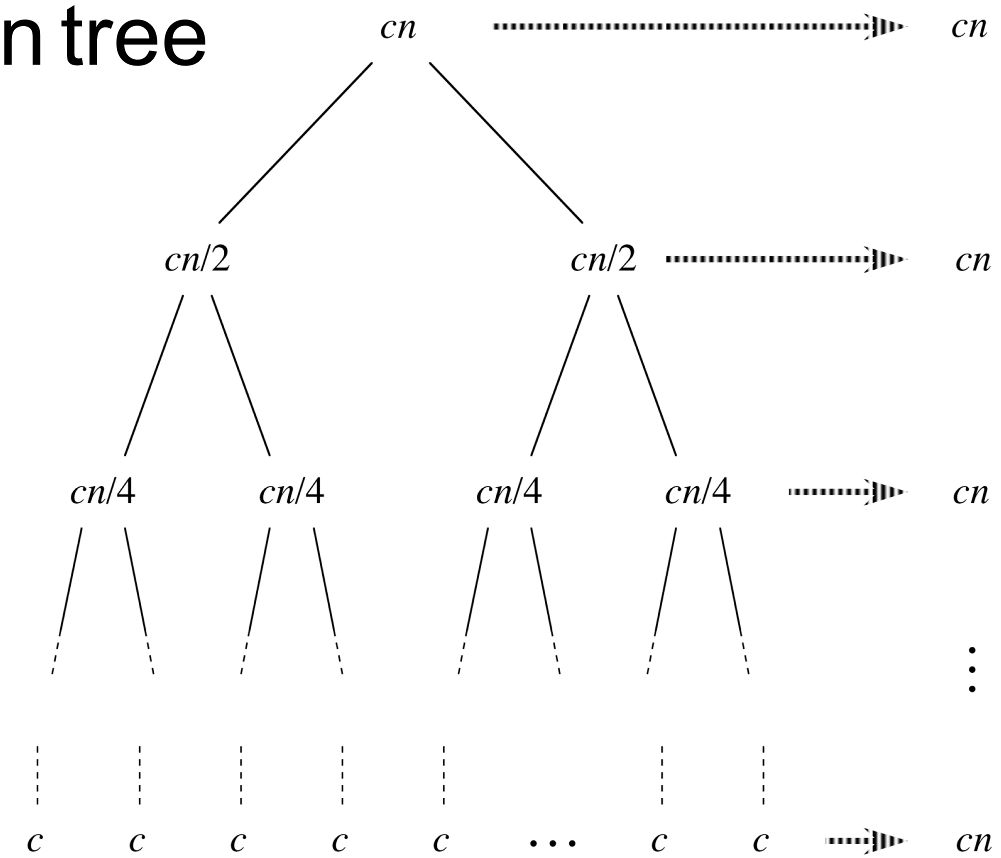
Level 2: $\frac{n}{4} = \frac{n}{2^2}$

Do you see a pattern??

Level k? $\frac{n}{2^k}$



Merge sort: recursion tree

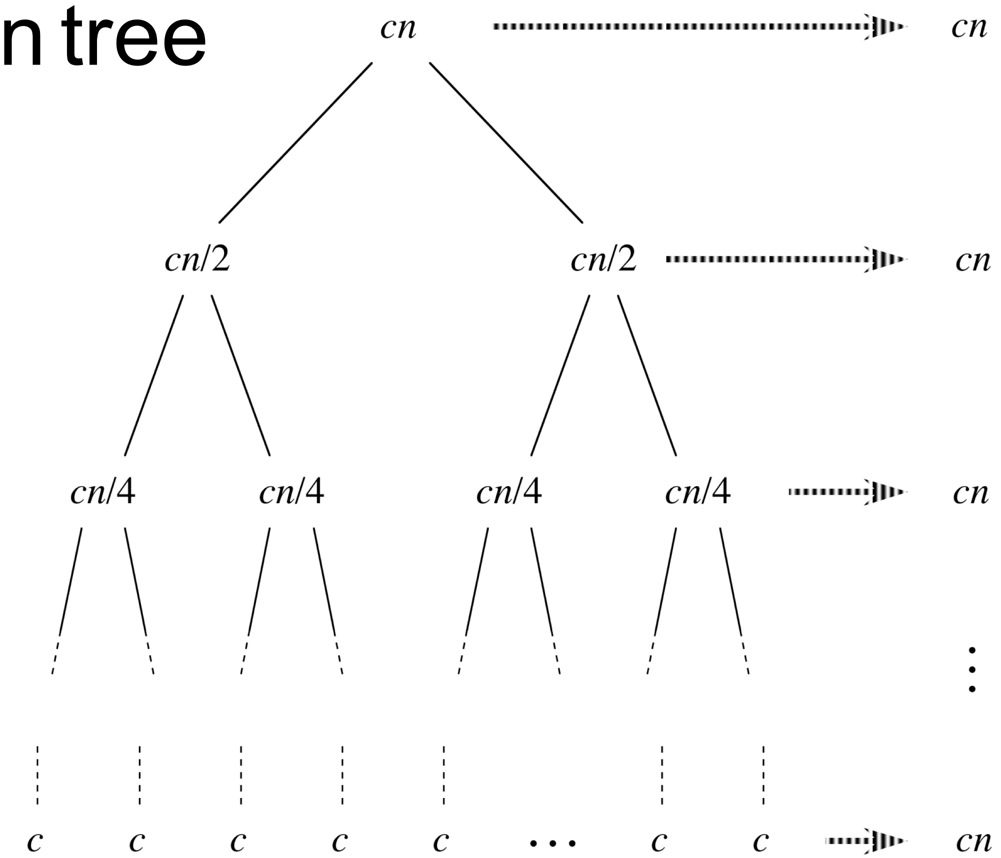


$$1 = \frac{n}{2^k}$$

Level of leaf nodes, we know that n=1; need to find level k.

Why do we care? Level k will give us height of the tree

Merge sort: recursion tree

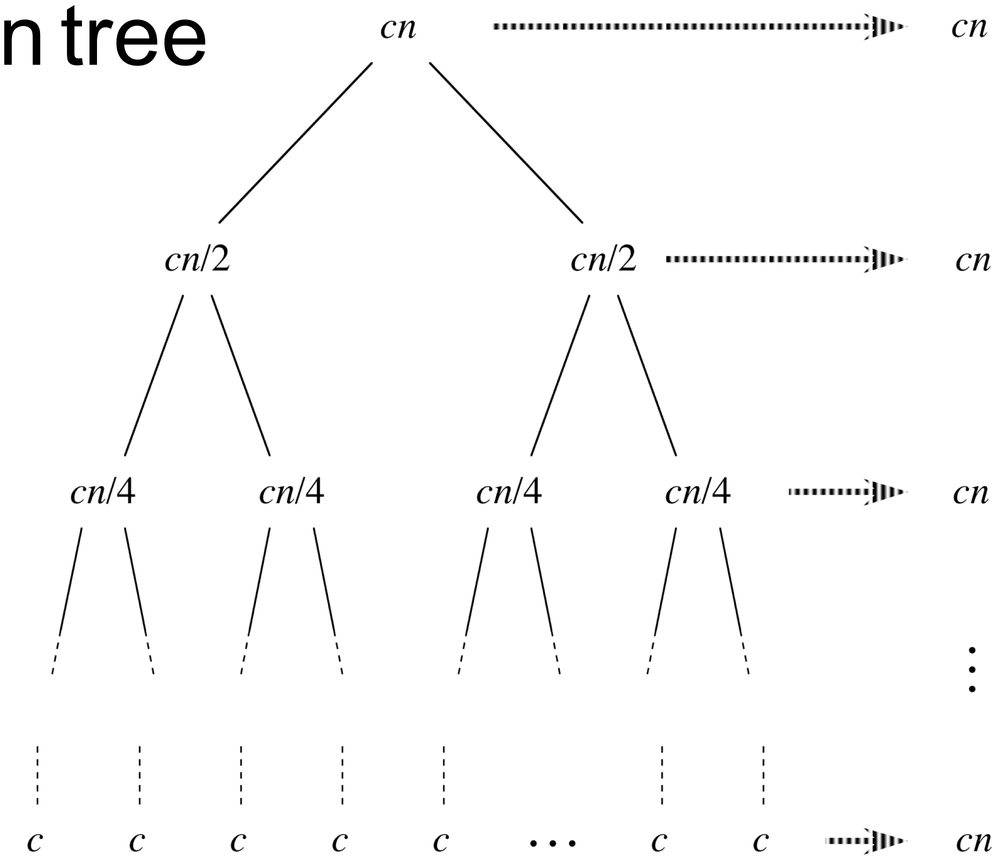


$$1 = \frac{n}{2^k}$$

$$2^k = n$$

$$k = \log_2(n)$$

Merge sort: recursion tree



Number of levels: $k = \log_2(n)$

Work at each level: cn

Merge sort: run time analysis

Number of levels: $\log_2(n)$

Work at each level: cn

Total work: $cn\log_2(n)$

As usual, we'll ignore constants. Grows as $n\log_2(n)$

What happens for the best case in Merge sort?

Merge sort: run time analysis

Summary:

Insertion sort: grows as n^2

Merge sort: grows as $n \log_2(n)$

Merge sort: run time analysis

Summary:

Insertion sort: grows as n^2

Merge sort: grows as $n \log_2(n)$

Is Merge sort always faster than Insertion sort?

Any disadvantages relative to insertion sort?

Some sort animations

Sorting as example: Insertion sort

Animation example:

http://cs.armstrong.edu/liang/animation/web/InsertionSortNew.html

Merge two lists

Animation example:

http://cs.armstrong.edu/liang/animation/web/MergeSortNew.html

Correctness & loop invariants

Correctness and loop invariants

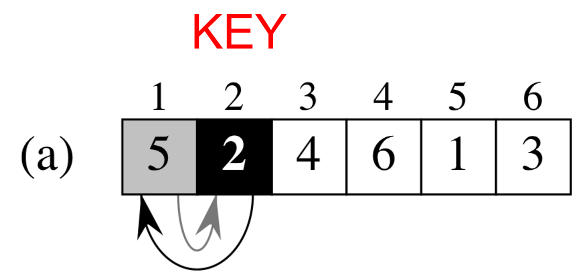
- How do we know that an algorithm is correct, i.e., always gives the right answer?
- We use loop invariants

Loop invariants

- Invariant = something that does not change
- Loop invariant = a property about the algorithm that does not change at every iteration before the loop
- This is usually the property we would like to prove is correct about the algorithm!
- The essence is intuitive, but we would like to state mathematically

Loop invariant example: insertion sort

- Insertion sort pseudo code

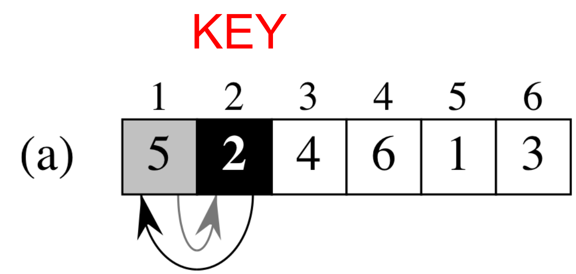



```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

What invariant property would make this algorithm correct?

Loop invariant example: insertion sort

- Insertion sort pseudo code

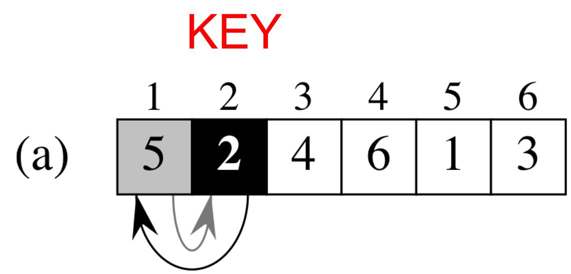


1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

What invariant property would make this algorithm correct?

Loop invariant example: insertion sort

- Insertion sort pseudo code



1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

What invariant property would make this algorithm correct?
That before each iteration of the for loop, the elements thus far are sorted. We would like to state this more formally

Loop invariant example: insertion sort

- Insertion sort pseudo code

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

- **Insertion sort loop invariant:** at the start of each iteration of the for loop, A[1..j-1] consists of elements originally in A[1..j-1], but in sorted order

Loop invariants

Proving involves 3 steps:

1. Initialization: Algorithm is true prior to first iteration of the loop (base case)

2. Maintenance: If it is true before an iteration of the loop it remains true before the next iteration (like an induction step)

3. When the loop terminates, the invariant gives a useful property that shows the algorithm is correct

Loop invariants example: insertion sort

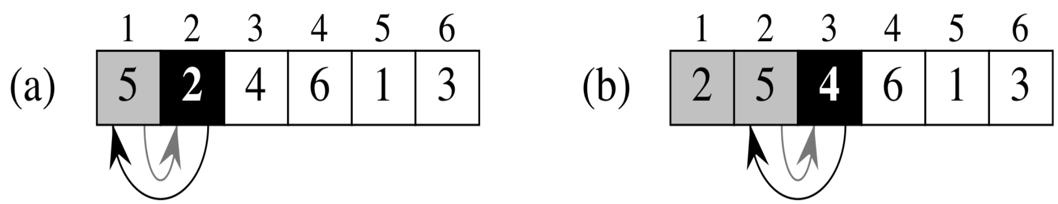
1. Initialization: Algorithm is true prior to first iteration of the loop (base case)

   When j=2, A[1] is just one element, which is the original element in A[1], and must be already sorted. So A[1..j-1] = A[1] which is already sorted

j=2 → 1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

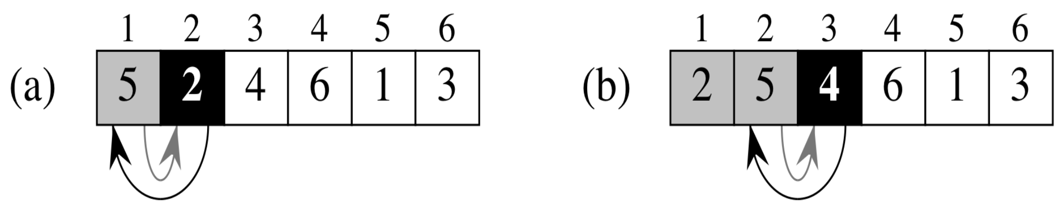
Loop invariants example: insertion sort

2. Maintenance: If it is true before an iteration of the loop it remains true before the next iteration (like an induction step).



Loop invariants example: insertion sort

2. Maintenance: If it is true before an iteration of the loop it remains true before the next iteration (like an induction step)

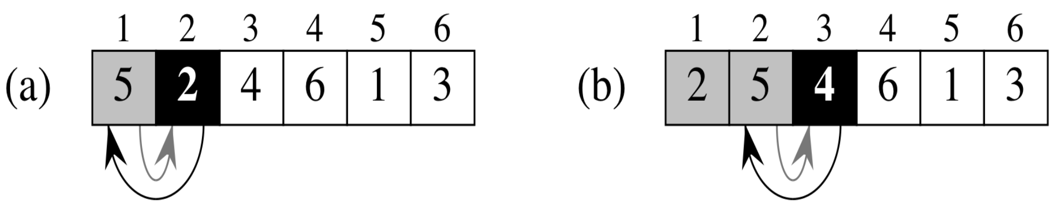


Assume: True here for j-1, that A[1..j-1] sorted

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

Loop invariants example: insertion sort

2. Maintenance: If for j-1 it is true that A[1..j-1] in sorted order before start of the for loop, then for j we will have A[1..j] in sorted order before start of next for loop iteration

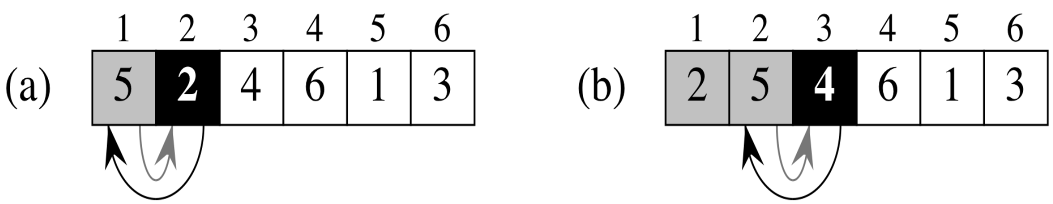


Assume: True here for j-1, that A[1..j-1] sorted

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

Loop invariants example: insertion sort

2. Maintenance: If for j-1 it is true that A[1..j-1] in sorted order before start of the for loop, then for j we will have A[1..j] in sorted order before start of next for loop iteration



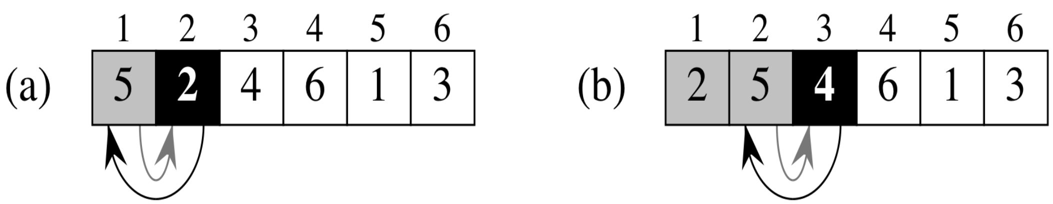
True here for j-1

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

Might not be true in loop. But make sure here that it remains sorted after, by pairwise swaps

Loop invariants example: insertion sort

2. Maintenance: If for j-1 it is true that A[1..j-1] in sorted order before start of the for loop, then for j we will have A[1..j] in sorted order before start of next for loop iteration



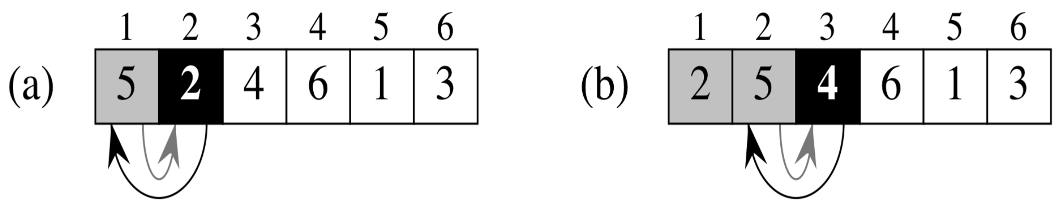
So will remain true for j that A[1..j] in sorted order

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

We make sure here that it remains sorted, by pairwise swaps

Loop invariants example: insertion sort

2. Maintenance: If it is true before an iteration of the loop it remains true before the next iteration (like an induction step)



If A[1..j-1] sorted before iteration of loop, then for key=A[j], we pairwise swap it into correct position; so now A[1..j] is also sorted

Loop invariants example: insertion sort

3. When the loop terminates, the invariant gives a useful property that shows the algorithm is correct

For loop to terminate j=n+1; for this to happen, A[1..j] must be in sorted order, which is A[1..n] or the entire array.

j=n+1 →

1. For j = 2 to n
2. Key = A[j]
3. Insert Key into sorted array A[1 .. j-1] by comparing and swapping into correct position

Loop invariants example: find min

Input: Array A
Output: Find minimum value in array A

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

What invariant would make this algorithm correct?

Loop invariants example: find min

Input: Array A
Output: Find minimum value in array A

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
        if min > A[i]
                min = A[i]
return min
```

What invariant would make this algorithm correct? That at each iteration of the loop, we have identified the smallest element thus far. Stated more formally…

Loop invariants example: find min

Input: Array A
Output: Find minimum value in array A

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Loop invariant: At the start of each iteration of the for loop, min is the smallest element in A[1 .. i-1]

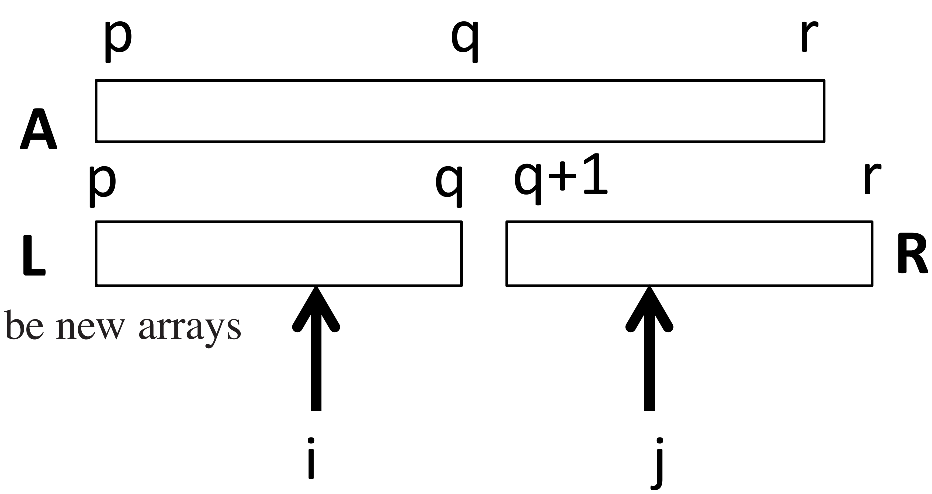
Loop invariants example: find min

Animation example (Burt Rosenberg)
http://www.cs.miami.edu/~burt/learning/Csc517.101/workbook/findmin.html

Input: Array A
Output: Find minimum value in array A

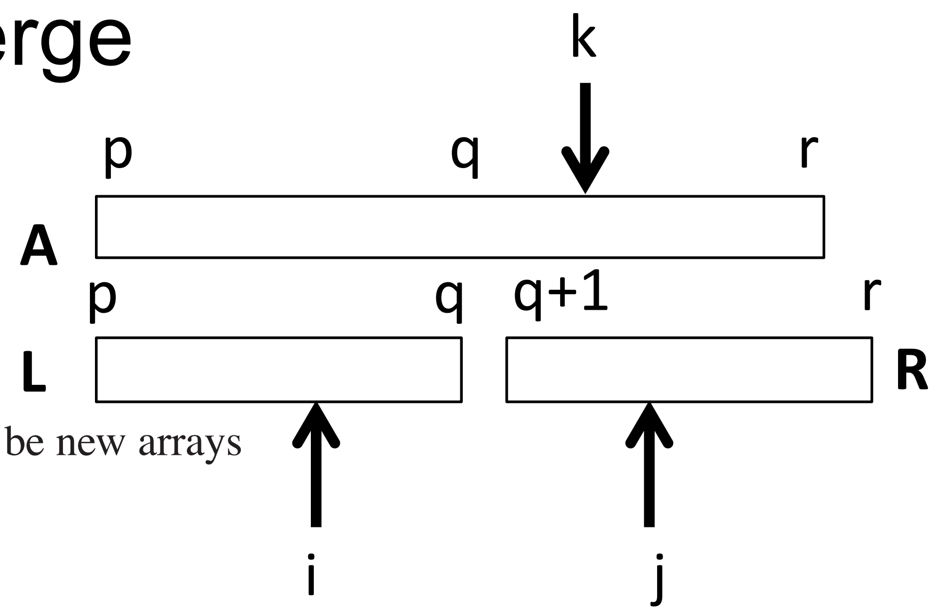
```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Loop invariant of Merge




```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Loop invariant of Merge



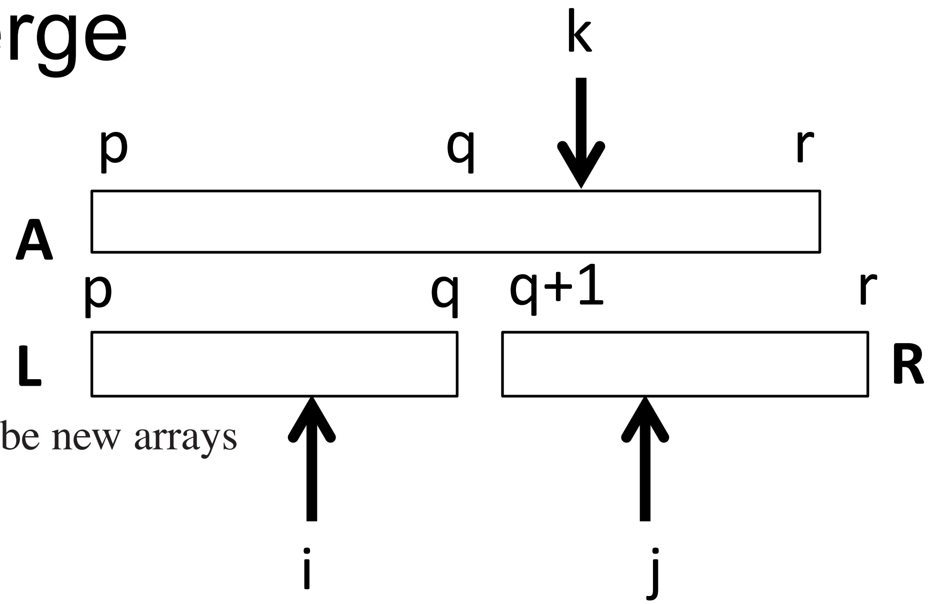

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Loop invariant: At the start of each iteration of the for loop, A[p.. k-1] contains the k-p smallest elements, in sorted order.

Loop invariant of Merge

MERGE(A, p, q, r)

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



Loop invariant: At the start of each iteration of the for loop, A[p.. k-1] contains the k-p smallest elements, in sorted order. Also, L[i] and R[j] are smallest elements of their arrays not yet copied back into A