Quicksort and Randomized Algs

David Kauchak
cs302
Spring 2012

Administrative

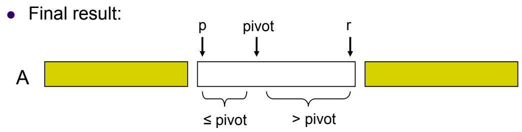
- Homework 3
- Homework 4 out today

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

- What does it do?

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

- A[r] is called the ***pivot***
- Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot
- Operates in place
- Final result:



… 5 7 1 2 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i

… 5 7 1 2 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

PARTITION(A, p, r)

```
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 7 1 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 7 2 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 2 7 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 2 7 8 4 3 6 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

i j

… 5 1 2 7 8 4 3 6 …

p r

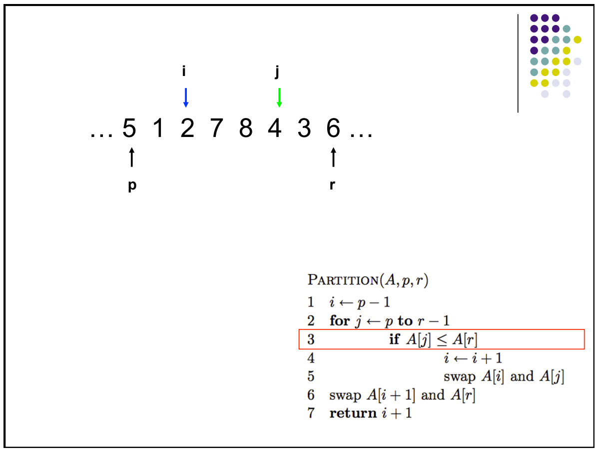
What’s happening?

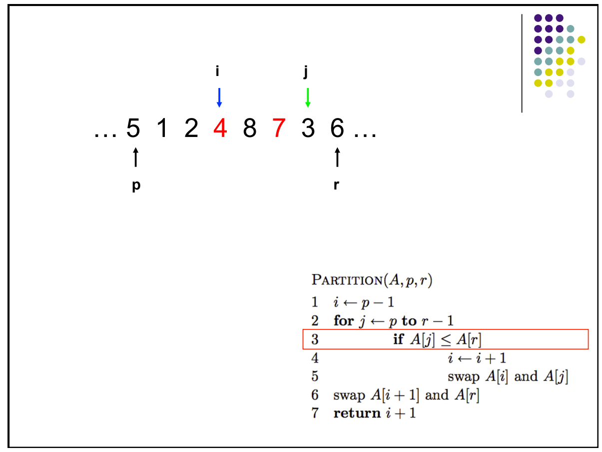
i j

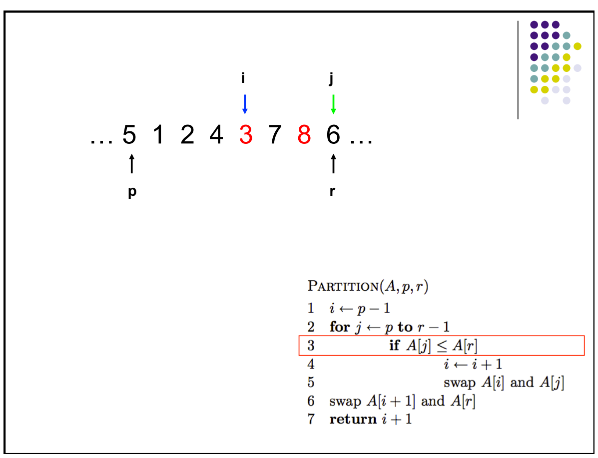
… 5 1 2 7 8 4 3 6 …

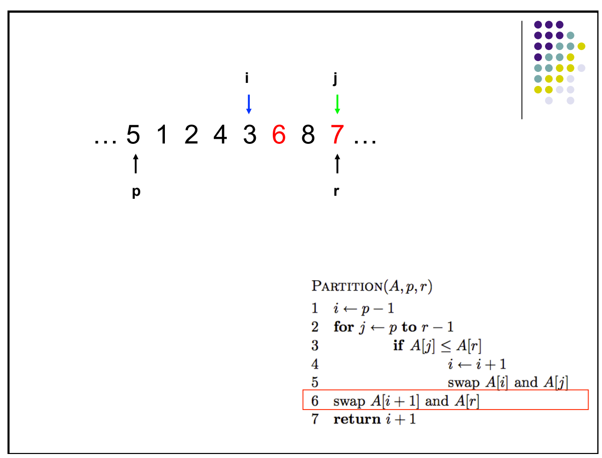
p r

≤ pivot > pivot unprocessed


i
j
… 5 1 2 7 8 4 3 6 …
p
r
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
if A[j] ≤ A[r]
i ← i + 1
swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1


i
j
… 5 1 2 4 8 7 3 6 …
p
r
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
if A[j] ≤ A[r]
i ← i + 1
swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1


i
j
… 5 1 2 4 3 7 8 6 …
p
r
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
if A[j] ≤ A[r]
i ← i + 1
swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1


i
j
… 5 1 2 4 3 6 8 7 …
p
r
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
if A[j] ≤ A[r]
i ← i + 1
swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1

i j

… 5 1 2 4 3 $\boxed{6}$ 8 7 …

p r

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Is Partition correct?

- Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot?
- Loop Invariant:

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Is Partition correct?

- Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot?
- Loop Invariant: A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Proof by induction

- Loop Invariant: A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]
- Base case: A[p…i] and A[i+1…j-1] are empty
- Assume it holds for *j* -1
- two cases:
  - A[j] > A[r]
    - A[p…i] remains unchanged
    - A[i+1…j] contains one additional element, A[j] which is > A[r]

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Proof by induction

- Loop Invariant: A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]
- 2nd case:
  - A[j] ≤ A[r]
    - *i* is incremented
    - A[i] swapped with A[j] – A[p…i] constains one additional element which is ≤ A[r]
    - A[i+1…j-1] will contain the same elements, except the last element will be the old first element

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Partition running time?

- **Θ(n)**

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Quicksort

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
i ← p − 1
for j ← p to r − 1
        if A[j] ≤ A[r]
                i ← i + 1
                swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

8	5	1	3	6	2	7	4

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

8 5 1 3 6 2 7 4

```
QUICKSORT(A, p, r)
  if p < r
      q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

1 3 2 4 6 8 7 5

```
QUICKSORT(A, p, r)
  if p < r
      q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

1 3 2 4 6 8 7 5

```
QUICKSORT(A, p, r)
  if p < r
      q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

1 3 2 4 6 8 7 5

```
QUICKSORT(A, p, r)
  if p < r
      q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q − 1)
      QUICKSORT(A, q + 1, r)
```

1 $\boxed{2}$ 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

$\boxed{1}$ 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
if p < r
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

1 2 3 4 5 8 7 6

What happens here?

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

1 2 3 4 5 8 7 6

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

1 2 3 4 5 8 7 6

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

1 2 3 4 5 6 7 8

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

1 2 3 4 5 6 7 8

```
QUICKSORT(A, p, r)
if p < r
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Some observations

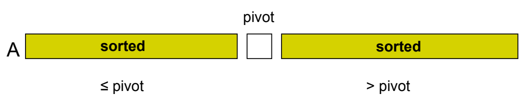
- Divide and conquer: different than MergeSort – do the work before recursing
- How many times is/can an element selected for as a pivot?
- What happens after an element is selected as a pivot?

1 3 2 4 6 8 7 5

Is Quicksort correct?

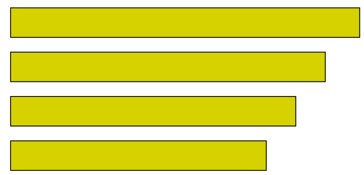
Is Quicksort correct?

- Assuming Partition is correct
- Proof by induction
  - Base case: Quicksort works on a list of 1 element
  - Inductive case:
    - Assume Quicksort sorts arrays for arrays of smaller < n elements, show that it works to sort n elements
    - If partition works correctly then we have:
    - and, by our inductive assumption, we have:



Running time of Quicksort?

- Worst case?
- Each call to Partition splits the array into an empty array and n-1 array



Quicksort: Worse case running time

$$T(n) = T(n-1) + \Theta(n)$$

Which is? $\Theta(n^2)$

- When does this happen?
  - sorted
  - reverse sorted
  - near sorted/reverse sorted

Quicksort best case?

- Each call to Partition splits the array into two equal parts

$$T(n) = 2T(n/2) + \Theta(n)$$

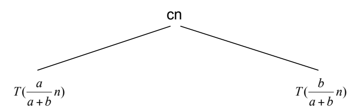
- O(n log n)
- When does this happen?
  - random data?

Quicksort Average case?

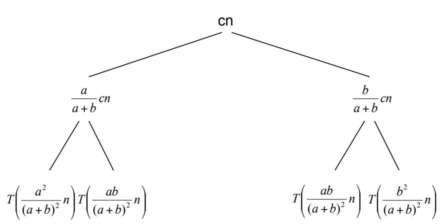
- How close to "even" splits do they need to be to maintain an O(n log n) running time?
- Say the Partition procedure always splits the array into some constant ratio b-to-a, e.g. 9-to-1
- What is the recurrence?

$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

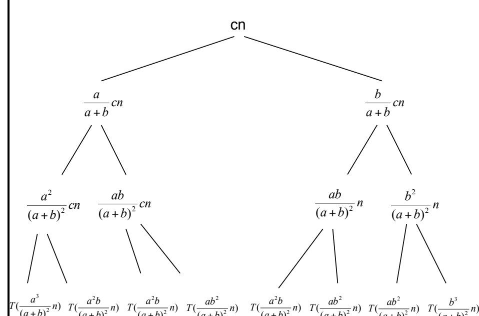
$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

Level 0: cn

Level 1: $= cn\left(\frac{a}{a+b}\right) + cn\left(\frac{b}{a+b}\right) = cn$

Level 2: $= cn\left(\frac{a^2}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{b^2}{(a+b)^2}\right)$

$$= cn\left(\frac{a^2+2ab+b^2}{(a+b)^2}\right) = cn\left(\frac{(a+b)^2}{(a+b)^2}\right) = cn$$

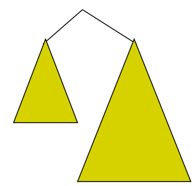
Level 3: $= cn\left(\frac{(a+b)^2 a + (a+b)^2 b}{(a+b)^3}\right)$

$$= cn\left(\frac{(a+b)(a+b)^2}{(a+b)^3}\right) = cn$$

Level d: $= cn\left(\frac{(a+b)^d}{(a+b)^d}\right) = cn$

What is the depth of the tree?

- Leaves will have different heights
- Want to pick the deepest leave
- Assume a < b



What is the depth of the tree?

- Assume a < b

$$\left(\frac{b}{a+b}\right)^d n = 1$$

$$\ldots$$

$$d = \log_{\frac{a+b}{b}} n$$

Cost of the tree

- Cost of each level $\le$ cn

?

Cost of the tree

- Cost of each level ≤ cn
- Times the maximum depth

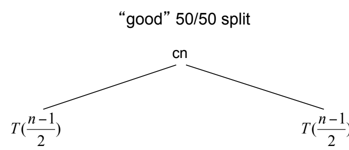
$$O(n \log_{\frac{a+b}{b}} n)$$

- Why not?

$$\Theta(n \log_{\frac{a+b}{b}} n)$$

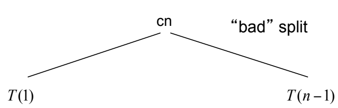
Quicksort average case: take 2

- What would happen if half the time Partition produced a “bad” split and the other half “good”?

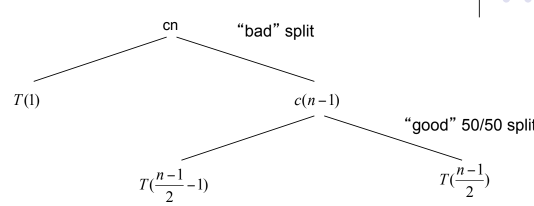


$$T(n) = 2T(\frac{n-1}{2}) + \Theta(n)$$

Quicksort average case: take 2

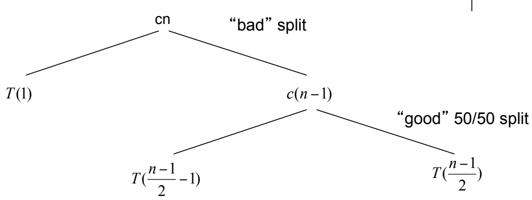


Quicksort average case: take 2



$$T(n) = \underbrace{T(1) + T(\frac{n-1}{2}-1) + T(\frac{n-1}{2})}_{\text{recursion cost}} + \underbrace{\Theta(n) + \Theta(n-1)}_{\text{partition cost}}$$

Quicksort average case: take 2



$$T(n) = T(\frac{n-1}{2}-1) + T(\frac{n-1}{2}) + \Theta(n)$$

We absorb the "bad" partition. In general, we can absorb any constant number of "bad" partitions

How can we avoid the worst case?

- Inject randomness into the data

```
RANDOMIZED-PARTITION(A, p, r)
  i ← RANDOM(p, r)
  swap A[r] and A[i]
  return PARTITION(A, p, r)
```

What is the running time of randomized Quicksort?

- Worst case?

$O(n^2)$

- Still could get very unlucky and pick "bad" partitions at every step

randomized Quicksort: expected running time

- How many calls are made to Partition for an input of size *n*? *n*
- What is the cost of a call to Partition?
- Cost is proportional to the number of iterations of the **for** loop

```
PARTITION(A, p, r)
  i ← p − 1
  for j ← p to r − 1
        if A[j] ≤ A[r]
              i ← i + 1
              swap A[i] and A[j]
  swap A[i + 1] and A[r]
  return i + 1
```

the total number of comparisons will give us a bound on the running time

Counting the number of comparisons

- Let z_i of $z_1, z_2, \ldots, z_n$ be the *i* th smallest element
- Let Z_{ij} be the set of elements $Z_{ij} = z_i, z_{i+1}, \ldots, z_j$

A = [3, 9, 7, 2]

$z_1 = 2$

$z_2 = 3$ $\quad Z_{24} =$

$z_3 = 7$

$z_4 = 9$

Counting the number of comparisons

- Let z_i of $z_1, z_2, \ldots, z_n$ be the *i* th smallest element
- Let Z_{ij} be the set of elements $Z_{ij} = z_i, z_{i+1}, \ldots, z_j$

A = [3, 9, 7, 2]

$z_1 = 2$

$z_2 = 3$ $\quad Z_{24} = [3, 7, 9]$

$z_3 = 7$

$z_4 = 9$

Counting comparisons

$$\text{Let } X_{ij} = I\{z_i \text{ is compared to } z_j\} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$$

(indicator random variable)

- How many times can z_i be compared to z_j?
- At most once. Why?

Total number of comparisons $\quad X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Counting comparisons: average running time

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$ expectation of sums is the sum of expectations

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p\{z_i \text{ is compared to } z_j\}$$

remember,

$$X_{ij} = I\{z_i \text{ is compared to } z_j\} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$$

$p\{z_i \text{ is compared to } z_j\}$?

- The pivot element separates the set of numbers into two sets (those less than the pivot and those larger). Elements from one set will never be compared to elements of the other set
- If a pivot x is chosen $z_i < x < z_j$ then z_i and z_j how many times will z_i and z_j be compared?
- What is the only time that z_i and z_j will be compared?
- In Z_{ij}, when will z_i and z_j will be compared?

$p\{z_i \text{ is compared to } z_j\}$?

$$p\{z_i \text{ is compared to } z_j\} = p\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$$

p(a,b) = p(a)+p(b) for independent events

$$= p\{z_i \text{ is first pivot chosen from } Z_{ij}\} + p\{z_j \text{ is first pivot chosen from } Z_{ij}\}$$

pivot is chosen randomly over *j-i+1* elements

$$= \frac{1}{j-1+1} + \frac{1}{j-1+1}$$

$$= \frac{2}{j-1+1}$$

E[X] ?

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k+1}$$

Let $k = j\text{-}i$

$$< \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$\boxed{\sum_{k=1}^{n} 2/k = \ln n + O(1) = O(\log n)}$$

$$= O(n \log n)$$

Memory usage?

- Quicksort only uses O(1) additional memory
- How does randomized Quicksort compare to Mergesort?

Merge-Sort: Another view

```
MERGE-SORT2(A, p, r)
if p < r
        q ← ⌊(p+r)/2⌋
        MERGE-SORT2(A, p, q)
        MERGE-SORT2(A, q + 1, r)
        MERGE2(A, p, q, r)
```

Merge-Sort: Another view

```
MERGE-SORT2(A, p, r)
if p < r
        q ← ⌊(p+r)/2⌋
        MERGE-SORT2(A, p, q)
        MERGE-SORT2(A, q + 1, r)
        MERGE2(A, p, q, r)
```

difference?

```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q + 1.. length[A]]
        copy A[1..q] to L
        copy A[q + 1.. length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```

Merge-Sort: Another view

```
MERGE2(A, p, q, r)
n1 ← q − p + 1      ▷ length of the left array
n2 ← r − q      ▷ length of the right array
create arrays L[1..n1 + 1] and R[1...n2 + 1]
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
```

Merge-Sort2

- Running time?

Merge-Sort2

- Running time?

- Same as MergeSort except the cost to divide the arrays is constant, i.e. *D(n) = c*
- Still results in:

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Memory?

```
MERGE-SORT2(A, p, r)
if p < r
        q ← ⌊(p+r)/2⌋
        MERGE-SORT2(A, p, q)
        MERGE-SORT2(A, q + 1, r)
        MERGE2(A, p, q, r)
```

```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q + 1.. length[A]]
        copy A[1..q] to L
        copy A[q + 1.. length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```

Memory?

- MergeSort

$$S(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2S(n/2) + cn & \text{otherwise} \end{cases}$$


```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q + 1.. length[A]]
        copy A[1..q] to L
        copy A[q + 1.. length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```


- MergeSort2

$$S(n) = \begin{cases} c & \text{if } n \text{ is small} \\ cn & \text{otherwise} \end{cases}$$


```
MERGE-SORT2(A, p, r)
if p < r
        q ← ⌊(p+r)/2⌋
        MERGE-SORT2(A, p, q)
        MERGE-SORT2(A, q + 1, r)
        MERGE2(A, p, q, r)
```


Memory?

- MergeSort2


```
MERGE-SORT2(A, p, r)
if p < r
        q ← ⌊(p+r)/2⌋
        MERGE-SORT2(A, p, q)
        MERGE-SORT2(A, q + 1, r)
        MERGE2(A, p, q, r)
```


$$S(n) = \begin{cases} c & \text{if } n \text{ is small} \\ cn & \text{otherwise} \end{cases}$$

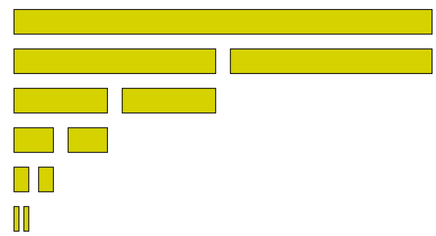
Memory?

- MergeSort

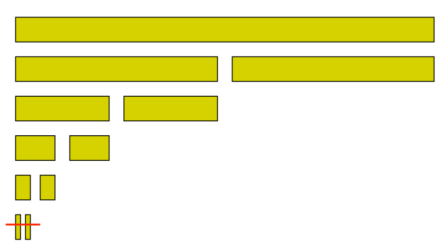
```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q+1..length[A]]
        copy A[1..q] to L
        copy A[q+1..length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```

$$S(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2S(n/2) + cn & \text{otherwise} \end{cases}$$?

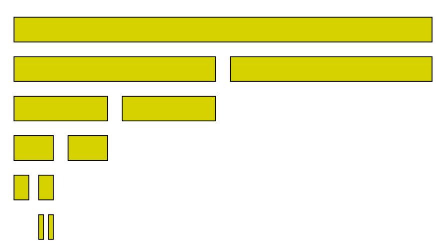
Memory can be reused



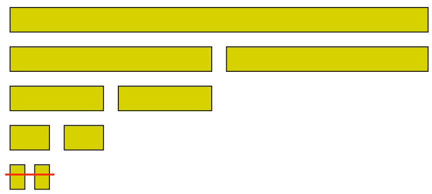
Memory can be reused



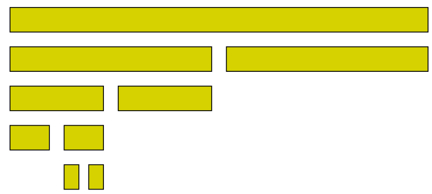
Memory can be reused



Memory can be reused



Memory can be reused



$$s(n) = n + n + n/2 + n/4 + n/8 + \ldots$$

Memory?

- Both MergeSort and MergeSort2 are O(n) memory
- In general, we' re interested in maximum memory used
  - MergeSort ~3n
  - MergeSort2 ~2n
- We may also be interested in average memory usage
  - MergeSort > MergeSort2

MergeSort: Another view

- How difficult are the two versions to implement?

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

```
MERGE2(A, p, q, r)
n1 ← q − p + 1      ▷ length of the left array
n2 ← r − q      ▷ length of the right array
create arrays L[1..n1 + 1] and R[1...n2 + 1]
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
```