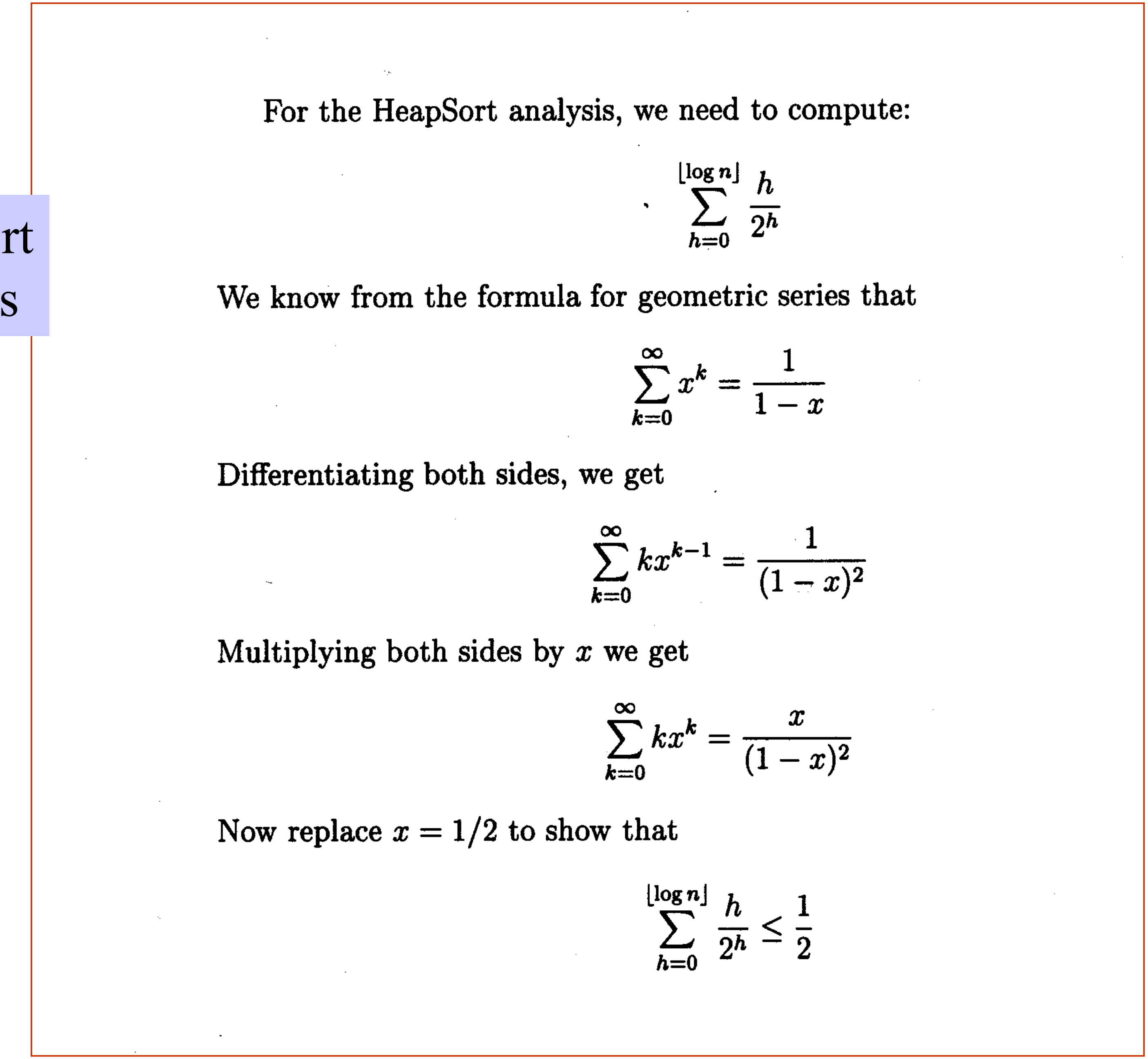
HeapSort Analysis



For the HeapSort analysis, we need to compute:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

We know from the formula for geometric series that

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Differentiating both sides, we get

$$\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides by x we get

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Now replace $x = 1/2$ to show that

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2}$$

Animation Demos

http://www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html

http://cg.scs.carleton.ca/~morin/misc/sortalg/

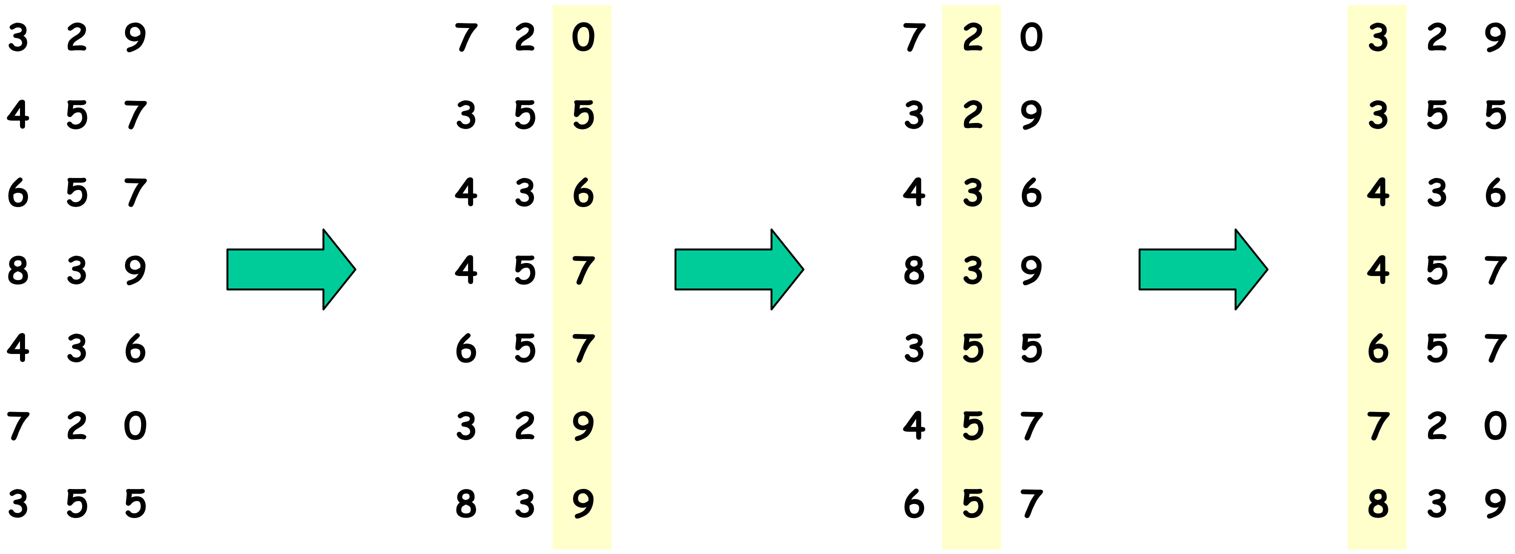
Bucket Sort

- N values in the range [a..a+m-1]
- For e.g., sort a list of 50 scores in the range [0..9].
- Algorithm
  - Make m buckets [a..a+m-1]
  - As you read elements throw into appropriate bucket
  - Output contents of buckets [0..m] in that order
- Time O(N+m)

Stable Sort

- A sort is stable if equal elements appear in the same order in both the input and the output.
- Which sorts are stable? Homework!

Radix Sort



3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Algorithm

```
for i = 1 to d do
        sort array A on digit i using a stable sort algorithm
```

Time Complexity: O((n+k)d)

Counting Sort

Initial Array

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

Counts

0	1	2	3	4	5
2	0	2	3	0	1

Cumulative Counts

0	1	2	3	4	5
2	2	4	7	7	8

Order Statistics

- Maximum, Minimum   n-1 comparisons

7	3	1	9	4	8	2	5	0	6

- MinMax
  - 2(n-1) comparisons
  - 3n/2 comparisons
- Max and 2ndMax
  - (n-1) + (n-2) comparisons
  - ???

k-Selection; Median

- Select the k-th smallest item in the list
- Naïve Solution
  - Sort;
  - pick the k-th smallest item in sorted list.

    O(n log n) time complexity
- Randomized solution: Average case O(n)
- Improved Solution: worst case O(n)

```
QuickSort(A, p, r)
    if (p < r) then
        q = Partition(A, p, r)
        QuickSort(A, p, q)
        QuickSort(A, 1+1, r)

Partition(A, p, r)
  x = A[p]
  i = p-1
  j = r+1
  while TRUE do
      repeat
          j- -
      until (A[j] <= x)
       repeat
          i++
      until (A[i] >= x)
      if (i < j) SWAP(A[i], A[j])
      else return j
```

Partition Procedure Revisited

- The Partition code can be rewritten so that it accepts another parameter, namely, the pivot value. Let's call this new variation as PivotPartition.
- This change does not affect its time complexity.
- RandomizedPartition as used in RandomizedSelect picks the pivot uniformly at random from among the elements in the list to be partitioned.

Randomized Selection

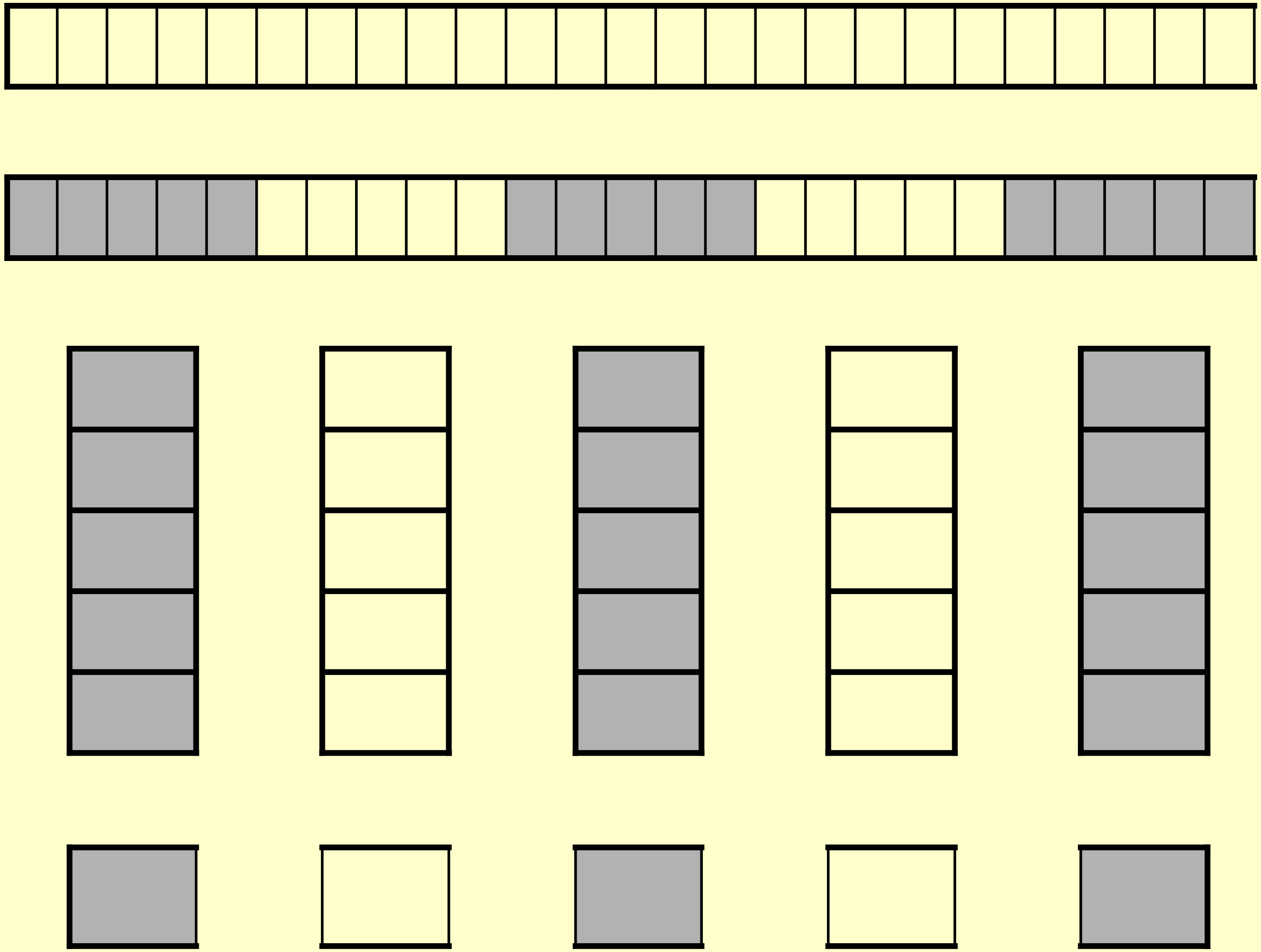
```
RandomizedSelect(A, p, r, i)
    if (p = r) then
        return A[p]
    q = RandomizedPartition(A, p, r)
    k = q – p + 1
    if (i <= k)
        return RandomizedSelect(A, p, q, i)
    else
        return RandomizedSelect(A, q+1, r, i-k)
```

Randomized Selection: Rewritten

```
RandomizedSelect(A, p, r, i)
    if (p = r) then
        return A[p]
    Pivot = A[random(p,r)]
    q = PivotPartition(A, p, r, Pivot)
    k = q – p + 1
    if (i <= k)
        return RandomizedSelect(A, p, q, i)
    else
        return RandomizedSelect(A, q+1, r, i-k)
```

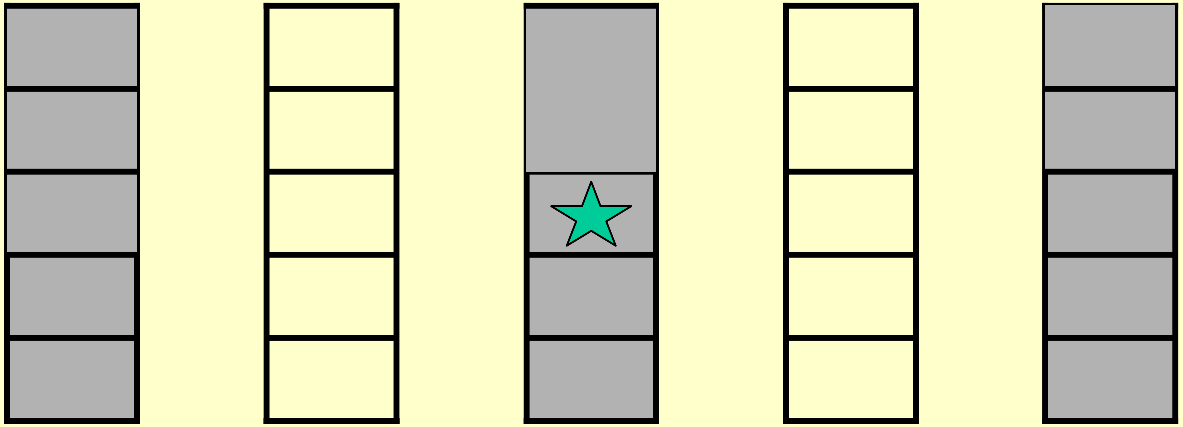
k-Selection & Median: Improved Algorithm

- Start with initial array



k-Selection & Median: Improved Algorithm(Cont'd)

- Use median of medians as pivot



- $T(n) < O(n) + T(n/5) + T(3n/4)$

Improved Selection

```
ImprovedSelect(A, p, r, i)
    if (p = r) then
        return A[p]
    else N = r – p + 1
    Partition A[p..r] into subsets of 5 elements and collect all
     the medians of the subsets in B[1..(N/5)].
    Pivot = ImprovedSelect (B, 1, $\lceil N/5 \rceil$, $\lceil N/10 \rceil$ )
    q = PivotPartition (A, p, r, Pivot)
    k = q – p + 1
    if (i <= k)
        return ImprovedSelect(A, p, q, i)
    else
        return ImprovedSelect(A, q+1, r, i-k)
```

Binary Search Trees

```
• TreeSearch(x, k)                    // pg 257
  // Search for key k in tree rooted at x
      if ((x = NIL) or (k = key[x]))
          return x
      if (k < key[x])
          return TreeSearch(left[x], k)
      else
          return TreeSearch(right[x], k)
```

Binary Search Trees

```
• TreeInsert (T,z)                      // pg 261
  // Insert node z in tree T
        y = NIL
        x = root[T]                     // y follows x down the tree
                                        // when x is NIL, y points to a leaf
        while (x ≠ NIL) do
              y = x
              if (key[z] < key[x])
                    x = left[x]
                    x = right[x]
        p[z] = y
        if (y == NIL)
              root[T] = z
        else if (key[z] < key[y])
              left[y] = z
        else right[y] = z
```

Binary Search Trees

- TreeDelete(T,z)

```
// delete node z in tree T
        if (left[z] == NIL) or (right[z] == NIL) then
                y = z
        else    y = TreeSuccessor(z)           // y has at most 1 child
        if (left[y] ≠ NIL) then
                x = left[y]
        else    x = right[y]                   // x points to a child of y
        if (x ≠ NIL)  then
                p[x] = p[y]
        if (p[y] == NIL) then
                root[T] = x
        else    if (y == left[p[y]]) then
                        left[p[y]] = x
                else    right[p[y]] = x
        if (y ≠ z) then
                key[z] = key[y]
                copy y's data into z
        return y
```

Red-Black Trees

- RB-Insert (T,z) // pg 261

```
// Insert node z in tree T
y = NIL
x = root[T]
while (x ≠ NIL) do
        y = x
        if (key[z] < key[x])
                x = left[x]
                x = right[x]
p[z] = y
if (y == NIL)
        root[T] = z
else if (key[z] < key[y])
        left[y] = z
else right[y] = z
// new stuff
left[z] = NIL[T]
right[z] = NIL[T]
color[z] = RED
RB-Insert-Fixup (T,z)
```

```
RB-Insert-Fixup (T,z)
while (color[p[z]] == RED) do
        if (p[z] = left[p[p[z]]]) then
            y = right[p[p[z]]]
            if (color[y] == RED) then        // C-1
                color[p[z]] = BLACK
                color[y] = BLACK
                z = p[p[z]]
            else   if (z == right[p[z]]) then // C-2
                    z = p[z]
                    LeftRotate(T,z)
                color[p[z]] = BLACK          // C-3
                color[p[p[z]]] = RED
                RightRotate(T,p[p[z]])
        else
            // Symmetric code: “right” ↔ “left”
            . . .
color[root[T]] = BLACK
```

Rotations

```
• LeftRotate(T,x)    // pg 278
  // right child of x becomes x’s parent.
  // Subtrees need to be readjusted.
  y = right[x]
  right[x] = left[y]     // y’s left subtree becomes x’s right
  p[left[y]] = x
  p[y] = p[x]
  if (p[x] == NIL[T]) then
        root[T] = y
  else if (x == left[p[x]]) then
        left[p[x]] = y
  else right[p[x]] = y
  left[y] = x
  p[x] = y
```