CS583 Lecture 03

Jana Kosecka

Heapsort, Quicksort

some materials here are based on E. Demaine , D. Luebke slides

Previously

- Solving recurrences
- Substitution method
- Iteration methods
- Recursion tree
- Masters's theorem

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$, and regularity condition
$\Rightarrow T(n) = \Theta(f(n))$.

Loop Invariants

```
1: for j ← 2 to n do
2:     Temp ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > Temp do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← Temp
9: end for
```

- Insertion Sort
- At any point we are looking at element j
- There is an invariant that is being maintained at each iteration of the loop
- Loop invariant: At the beginning of each iteration, elements in A[1, … j-1] are sorted
- At each iteration you add one element and increase the length of sorted elements

Loop Invariants

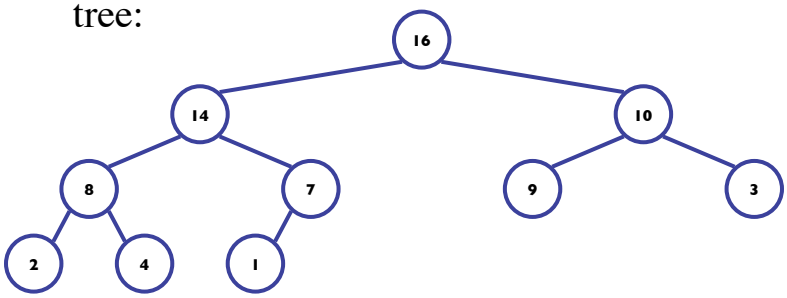
- Useful for showing correctness of programs
- Step 1: Show that the loop invariant is true at initialization
- Step 2: Maintenance: if it is true before iteration of the loop, it is true after the iteration of the loop
- Step 3: Termination: When the loop terminates, the invariant gives useful property showing that the algorithm is correct

Sorting Continued

- So far we've talked about two algorithms to sort an array of numbers
  - What is the advantage of merge sort?
  - What is the advantage of insertion sort?
- Next on the agenda: *Heapsort*
  - Combines advantages of both previous algorithms

Heaps

- A *heap* can be seen as a complete binary tree:

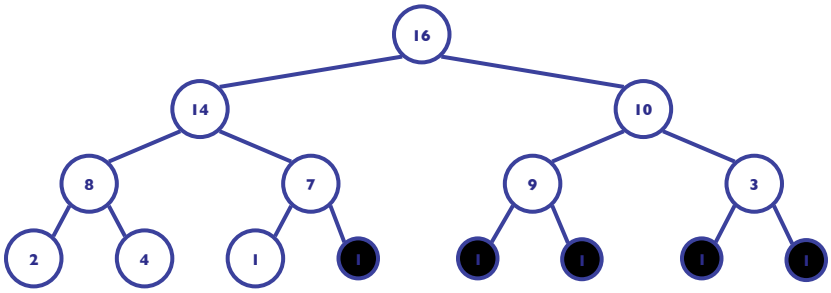


What makes a binary tree complete?

Is the example above complete?

Heaps

- A *heap* can be seen as a complete binary tree:

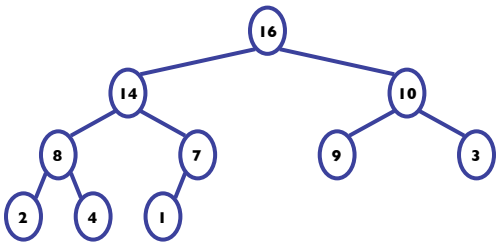


The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

Heaps

- In practice, heaps are usually implemented as arrays:

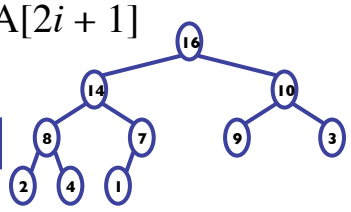
A =	16	14	10	8	7	9	3	2	4	1	



Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 | |



Referencing Heap Elements

- So…

```
Parent(i) { return ⌊i/2⌋; }

Left(i) { return 2*i; }

right(i) { return 2*i + 1; }
```

- An aside: *How would you implement this most efficiently?*

The Heap Property

- Heaps also satisfy the *heap property*:

  $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - *Where is the largest element in a heap stored?*
- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root

Heap Height

- *What is the height of an n-element heap? Why?*
- This is nice: basic heap operations take at most time proportional to the height of the heap

Heap Height

- *What is the height of an n-element heap? Why?* $\Theta(\lg n)$
- This is nice: basic heap operations take at most time proportional to the height of the heap $O(\lg n)$
- Max- heap for sorting
- Min-heap for priority cues

Heap Height

- Heapsort procedures
- **`Heapify`**
- **`Build-heap`**
- **`Heapsort`**

Heap Operations: Heapify()

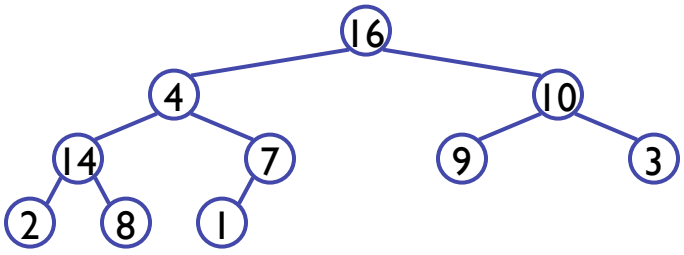
- **`Heapify()`:** maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r?*

Example

Heap Operations: Heapify()

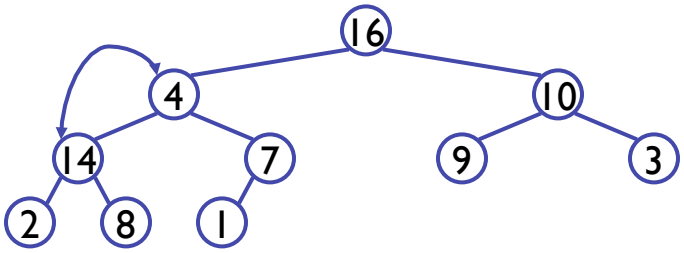
```
Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
     largest = l;
   else
     largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
     largest = r;
   if (largest != i)
     Swap(A, i, largest);
     Heapify(A, largest);
}
```

Heapify(A,2) Example

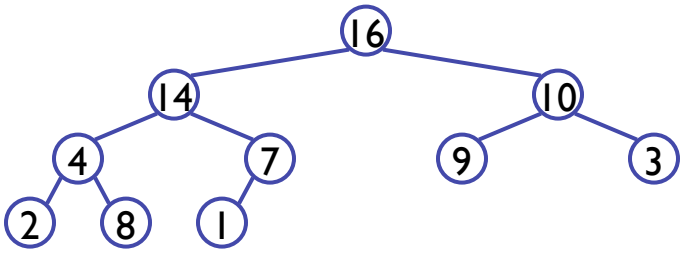


A =	16	4	10	14	7	9	3	2	8	1

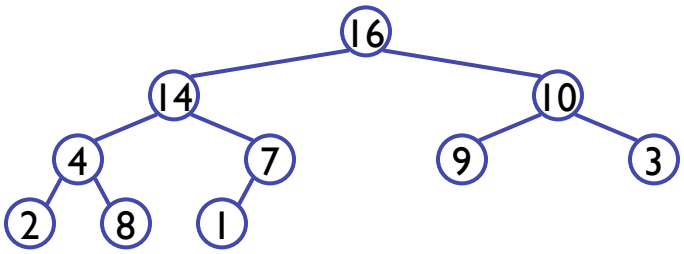
Heapify(A,2) Example



Heapify(A,2) Example

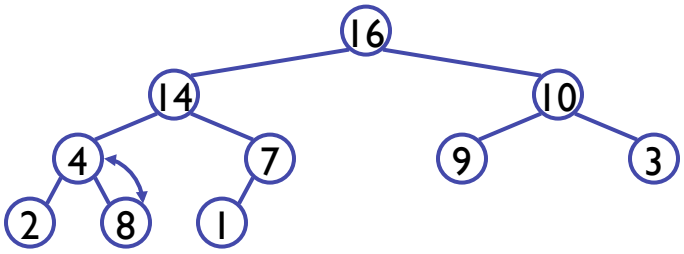


Heapify(A,2) Example



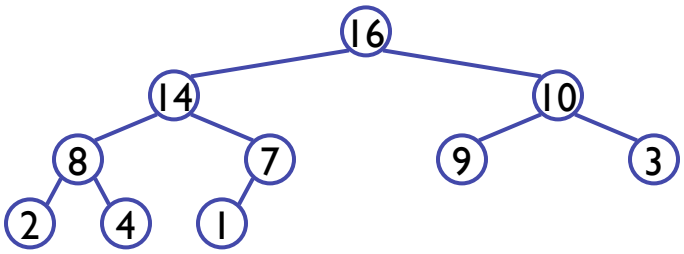
A =	16	14	10	4	7	9	3	2	8	1

Heapify(A,4) Example



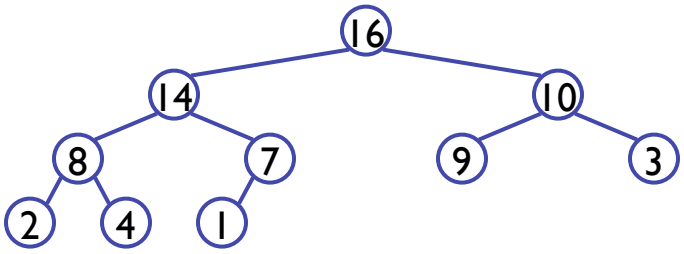
A =	16	14	10	4	7	9	3	2	8	1

Heapify(A,4) Example



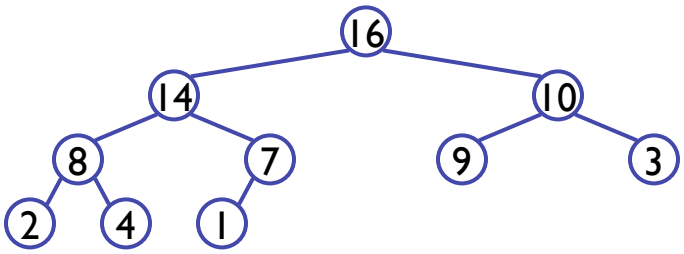
A =	16	14	10	8	7	9	3	2	4	1

Heapify(A,4) Example



A =	16	14	10	8	7	9	3	2	4	1

Heapify(A,9) Example



A =	16	14	10	8	7	9	3	2	4	1

Analyzing Heapify(): Informal

- Aside from the recursive call, what is the running time of **`Heapify()`** ?
- How many times can **`Heapify()`** recursively call itself?
- What is the worst-case running time of **`Heapify()`** on a heap of size n?

Analyzing Heapify(): Formal

- Fixing up relationships between i, l, and r takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **Heapify()** is given by $T(n) \leq T(2n/3) + \Theta(1)$

Analyzing Heapify(): Formal

- So we have

  $T(n) \leq T(2n/3) + \Theta(1)$

- By case 2 of the Master Theorem,

  $T(n) = O(\lg n)$

- Thus, **Heapify()** takes logarithmic time

Heap Operations: BuildHeap()

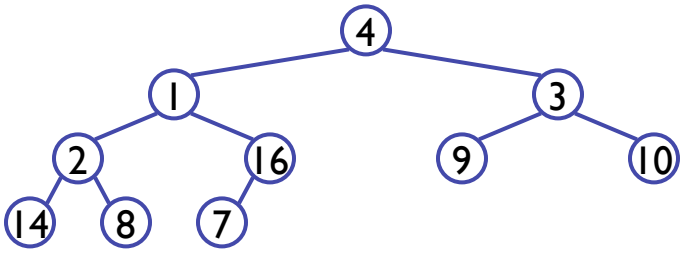
- We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[⌊n/2⌋ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **Heapify()** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

BuildHeap()

```
// given an unsorted array A, make A a
   heap
BuildHeap(A)
{
   heap_size(A) = length(A);
   for (i = ⌊length[A]/2⌋  downto 1)
     Heapify(A, i);
}
```

BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Analyzing BuildHeap()

- Each call to **Heapify()** takes O(lg n) time
- There are O(n) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(n lg n)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O(n)
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **`Heapify()`** a subtree takes O(h) time where h is the height of the subtree
  - h = O(lg m), m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- CLR 7.3 uses this fact to prove that **`BuildHeap()`** takes O(n) time

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size (A)]

Heapsort

```
Heapsort(A)
{
    BuildHeap(A);
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);
        heap_size(A) -= 1;
        Heapify(A, 1);
    }
}
```

Analyzing Heapsort

- The call to **`BuildHeap()`** takes O(n) time
- Each of the n - 1 calls to **`Heapify()`** takes O(lg n) time
- Thus the total time taken by **`HeapSort()`**
  = O(n) + (n - 1) O(lg n)
  = O(n) + O(n lg n)
  = O(n lg n)

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set S of elements, each with an associated value or *key*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - *What might a priority queue be useful for?*

Priority Queue Operations

- **`Insert(S, x)`** inserts the element x into set S
- **`Maximum(S)`** returns the element of S with the maximum key
- **`ExtractMax(S)`** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*

Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts O(n^2) in the worst case
- *So why would people use it instead of merge sort?*

Review: Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often (but more on this later…)

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

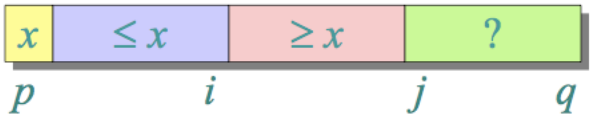
Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this?*

Partition procedure

```
PARTITION(A, p, q)  ▷ A[p . . q]
    x ← A[p]        ▷ pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
              then  i ← i + 1
                    exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

Invariant:



Partition Example

```
PARTITION(A, p, q)  ▷ A[p . . q]
    x ← A[p]        ▷ pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
              then  i ← i + 1
                    exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

```
A = {6,10,13,5,8,3,2,11}
```

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Analyzing Quicksort

- In the worst case, input sorted or reverse sorted. One side of partion has no elemets

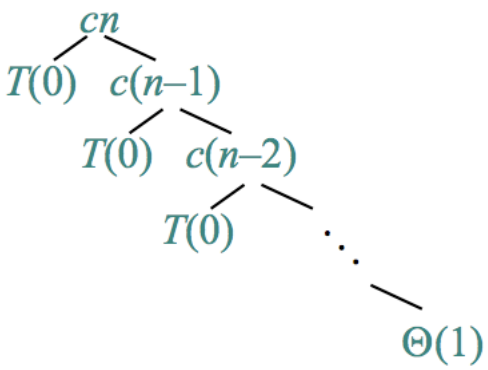
  $T(1) = \Theta(1)$

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

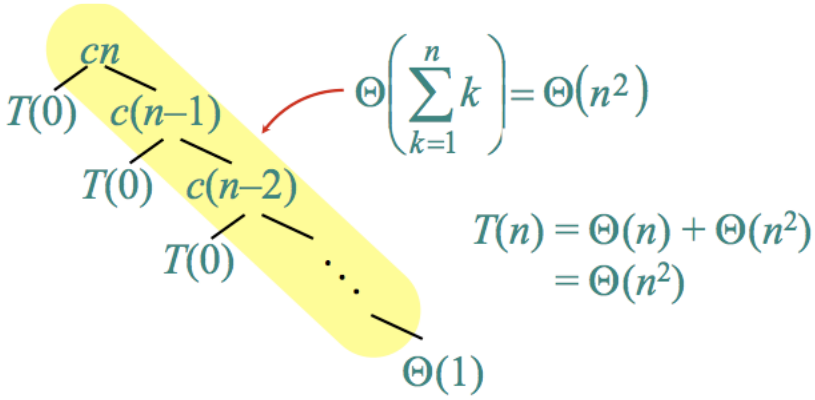
- Works out to (via aritmetic series)

  $T(n) = \Theta(n^2)$

Worst case recursion tree



Worst case recursion tree



Analyzing Quicksort

- In the best case:

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

- What does this work out to?

  $T(n) = \Theta(n \lg n)$

Analyzing Quicksort

- What is we split

  $T(n) = T(9n/10) + T(n/10) + cn$

Improving Quicksort

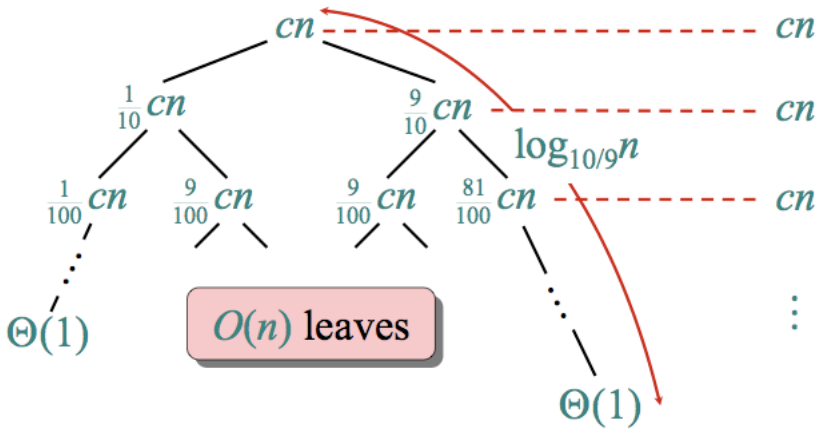
- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Analyzing Quicksort: Average Case

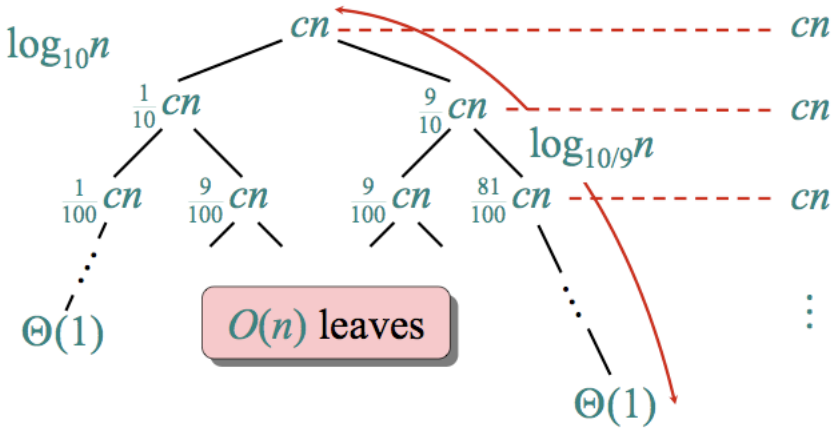
- Assuming random input, average-case running time is much closer to O(n lg n) than $O(n^2)$
- First, a more intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus: $T(n) = T(9n/10) + T(n/10) + n$
  - *How deep will the recursion go?* (draw it)

Use n instead of O(n) for convenience (how?)

Using Recurrence tree



Using Recurrence tree



$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n/2 : n/2) and worst-case (n-1 : 1)
  - *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n/2 : n/2) and worst-case (n-1 : 1)
- *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*
  - We end up with three subarrays, size 1, (n-1)/2, (n-1)/2

    $T(n) = 2(T(n/2 - 1) + \Theta(n/2)) + \Theta(n) = \Theta(n \lg n)$
  - No worse than if we had good-split the root node!

Analyzing Quicksort: Average Case

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- How can we be more rigorous?

Analyzing Quicksort: Average Case

- Idean partition around random element
- Running time is independent of inout order
- No assumptions made about input distribution
- No specific case gives worst case behavior
- Worst case is determined only by the output of random number generator
- Idea: let T(n) = random variable for running time of quicksort

Analyzing Quicksort: Average Case

- For simplicity, assume:
  - All inputs distinct (no repeats)
  - Slightly different **`partition()`** procedure
    - partition around a random element, which is not included in subarrays
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) equally likely
- *What is the probability of a particular split happening?*
- Answer: 1/n,

Analysis

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)$$

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$$E[X_k] = \Pr\{X_k = 1\} = 1/n$$

Analysis

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = 1/n$.

Analysis

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = 1/n$.

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Analysis

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

Prove: $E[T(n)] \le an \lg n$ for constant $a > 0$.

- Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Use substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

Analysis

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

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

$$= an \lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

$$\le an \lg n$$

If a is large enough such that $an/4$ dominates $\Theta(n)$

Quicksort summary

- Great general purpose sorting algorithm
- Typically twice as fast as merge sort
- Can benefit from code tuning

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

- *What is each term under the summation for?*
- *What is the Θ(n) term for?*

Alternative Analysis

Without formal expectations E[.]

Analyzing Quicksort: Average Case

- So…

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

Write it on the board

  - Note: this is just like the book's recurrence (p166), except that the summation starts with k=0
  - We'll take care of that in a second

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the substitution method
  - Guess the answer
    - *What's the answer?*
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \le an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \le an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - *What value?*
  - Prove that it follows for n

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$ The recurrence to be solved

$$\le \frac{2}{n}\sum_{k=0}^{n-1} (ak \lg k + b) + \Theta(n)$$ Plug in inductive hypothesis

$$\le \frac{2}{n}\left[b + \sum_{k=1}^{n-1} (ak \lg k + b)\right] + \Theta(n)$$ Expand out the k=0 case

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \frac{2b}{n} + \Theta(n)$$ 2b/n is just a constant, so fold it into Θ(n)

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n)$$ Note: leaving the same recurrence as the book

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n)$$ The recurrence to be solved

$$= \frac{2}{n}\sum_{k=1}^{n-1} ak \lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n)$$ Distribute the summation

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n)$$ Evaluate the summation: b+b+…+b = b (n-1)

$$\le \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$ Since n-1<n, 2b(n-1)/n < 2b

This summation gets its own set of slides later

Analyzing Quicksort: Average Case

$$T(n) \le \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$ The recurrence to be solved

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$ We'll prove this later

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

$$= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$ Remember, our goal is to get T(n) ≤ an lg n + b

$$\le an \lg n + b$$ Pick a large enough that an/4 dominates Θ(n)+b

Analyzing Quicksort: Average Case

- So T(n) ≤ an lg n + b for certain a and b
  - Thus the induction holds
  - Thus T(n) = O(n lg n)
  - Thus quicksort runs in O(n lg n) time on average
- Oh yeah, the summation…

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

Split the summation for a tighter bound

$$\leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

The lg k in the second term is bounded by lg n

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move the lg n outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$\leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The lg k in the first term is bounded by lg n/2

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k (\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

lg n/2 = lg n – 1

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move (lg n - 1) outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribute the (lg n - 1)

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summations overlap in range; combine them

$$= \lg n \left(\frac{(n-1)(n)}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The Gaussian series

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \left(\frac{(n-1)(n)}{2} \right) \lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summation bound so far

$$\le \frac{1}{2} \left[n(n-1) \right] \lg n - \sum_{k=1}^{n/2-1} k$$

Rearrange first term, place upper bound on second

$$\le \frac{1}{2} \left[n(n-1) \right] \lg n - \frac{1}{2} \left(\frac{n}{2} \right) \left(\frac{n}{2} - 1 \right)$$

Gaussian series

$$\le \frac{1}{2} \left(n^2 \lg n - n \lg n \right) - \frac{1}{8} n^2 + \frac{n}{4}$$

Multiply it all out

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \text{ when } n \ge 2$$

Done!!!