

The hiring problem

HIRE-ASSISTANT(n)

```
1  best = 0           // candidate 0 is a least-qualified dummy candidate
2  for  $i = 1$  to  $n$ 
3      interview candidate  $i$ 
4      if candidate  $i$  is better than candidate best
5          best =  $i$ 
6          hire candidate  $i$ 
```

Interviewing has a low cost, say c_i , whereas hiring is expensive, costing c_h .

- Total cost $O(c_i n + c_h m)$
 - worst case hiring cost $O(c_h n)$



Randomized algorithms

- We call an algorithm *randomized* if its behavior is determined not only by its input but also by values produced by a *random-number generator*.
 - Need a random-number generator $\text{RANDOM}(a,b)$
 - returns an integer between a and b
- *Pseudorandom-number generator*
 - a deterministic algorithm returning numbers that “look” statistically random.



Indicator random variables

- the *indicator random variable* $I\{A\}$.

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases}$$

- Example:** fair coin $X_H = I\{H\}$
$$= \begin{cases} 1 & \text{if } H \text{ occurs ,} \\ 0 & \text{if } T \text{ occurs .} \end{cases}$$

$$\begin{aligned} E[X_H] &= E[I\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$



Lemma 5.1

Given a sample space S and an event A in the sample space S , let $X_A = I\{A\}$. Then $E[X_A] = \Pr\{A\}$.

$$E[X] = \sum_{x=1}^n x \Pr\{X = x\} ,$$

$$\begin{aligned} X_i &= I\{\text{candidate } i \text{ is hired}\} & E[X_i] &= \Pr\{\text{candidate } i \text{ is hired}\} \\ &= \begin{cases} 1 & \text{if candidate } i \text{ is hired,} \\ 0 & \text{if candidate } i \text{ is not hired,} \end{cases} & &= 1/i . \end{aligned}$$

$$\begin{aligned} X &= X_1 + X_2 + \cdots + X_n . & E[X] &= E\left[\sum_{i=1}^n X_i\right] && \text{(by equation (5.2))} \\ & & &= \sum_{i=1}^n E[X_i] && \text{(by linearity of expectation)} \\ & & &= \sum_{i=1}^n 1/i && \text{(by equation (5.3))} \\ & & &= \ln n + O(1) && \text{(by equation (A.7)) .} \end{aligned}$$

$$O(c_h n) \longrightarrow O(c_h \ln n).$$



Randomized algorithms

RANDOMIZED-HIRE-ASSISTANT(n)

```
1  randomly permute the list of candidates
2   $best = 0$            // candidate 0 is a least-qualified dummy candidate
3  for  $i = 1$  to  $n$ 
4      interview candidate  $i$ 
5      if candidate  $i$  is better than candidate  $best$ 
6           $best = i$ 
7          hire candidate  $i$ 
```

Lemma 5.3

The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.



Randomly permuting arrays

PERMUTE-BY-SORTING(A)

```
1   $n = A.length$ 
2  let  $P[1..n]$  be a new array
3  for  $i = 1$  to  $n$ 
4       $P[i] = \text{RANDOM}(1, n^3)$ 
5  sort  $A$ , using  $P$  as sort keys
```

Lemma 5.4

Procedure PERMUTE-BY-SORTING produces a uniform random permutation of the input, assuming that all priorities are distinct.

RANDOMIZE-IN-PLACE(A)

```
1   $n = A.length$ 
2  for  $i = 1$  to  $n$ 
3      swap  $A[i]$  with  $A[\text{RANDOM}(i, n)]$ 
```

Lemma 5.5

Procedure RANDOMIZE-IN-PLACE computes a uniform random permutation.



Quicksort



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



Divide-and-conquer

1. *Divide*: Partition the array into two subarrays around a pivot x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



2. *Conquer*: Recursively sort the two subarrays.
3. *Combine*: Trivial (because in place).

Key: Linear-time partitioning procedure.



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 \leq all values in second
 - Returns the index of the “pivot” element separating the two subarrays
- *How do you suppose we implement this?*



Partition In Words

- Partition(A, p, r):
 - Select an element to act as the “pivot” (*which?*)
 - Grow two regions, A[p..i] and A[j..r]
 - All elements in A[p..i] \leq pivot
 - All elements in A[j..r] \geq pivot
 - Increment i until A[i] \geq pivot
 - Decrement j until A[j] \leq pivot
 - Swap A[i] and A[j]
 - Repeat until i \geq j
 - Return j

*Note: slightly different from
book's partition()*



Partition Code

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

Illustrate on

A = {5, 3, 2, 6, 4, 1, 3, 7};

*What is the running time of
partition()?*



Partition Code

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

partition() runs in $O(n)$ time



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:

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

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

- Works out to

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



Analyzing Quicksort

- In the best case:

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

- What does this work out to?

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




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



*Use n instead of $O(n)$
for convenience (how?)*
 - *How deep will the recursion go?* (draw it)



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, $(n-1)/2$, $(n-1)/2$
 - Combined cost of splits = $n + n - 1 = 2n - 1 = O(n)$
 - 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

- 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: $1/n$



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,

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

- *What is each term under the summation for?*
- *What is the $\Theta(n)$ term for?*



Analyzing Quicksort: Average Case

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

$$= \frac{2}{n} \sum_{k=0}^{n-1} T(k) + \Theta(n) \quad \leftarrow \text{Write it on the board}$$



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 dreaded 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
 - *What's the inductive hypothesis?*
 - 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
 - $T(n) \leq an \lg n + b$ 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
 - $T(n) \leq an \lg n + b$ 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

- 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
 - $T(n) \leq an \lg n + b$ for some constants a and b
 - Substitute it in for some value $< n$
 - The value k in the recurrence
 - 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
 - $T(n) \leq an \lg n + b$ for some constants a and b
 - Substitute it in for some value $< n$
 - The value k in the recurrence
 - Prove that it follows for n
 - Grind through it...



Analyzing Quicksort: Average Case

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

The recurrence to be solved

$$\leq \frac{2}{n} \sum_{k=0}^{n-1} (ak \lg k + b) + \Theta(n)$$

Plug in inductive hypothesis

$$\leq \frac{2}{n} \left[b + \sum_{k=1}^{n-1} (ak \lg k + b) \right] + \Theta(n)$$

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 $\Theta(n)$*

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

*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:
 $b+b+\dots+b = b(n-1)$*

$$\leq \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$

Since $n-1 < n$, $2b(n-1)/n < 2b$

This summation gets its own set of slides later



Analyzing Quicksort: Average Case

$$T(n) \leq \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$

The recurrence to be solved

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

We'll prove this later

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

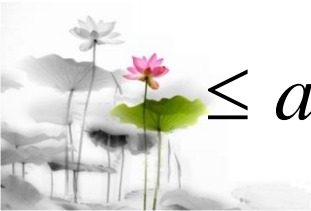
Distribute the $(2a/n)$ term

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

Remember, our goal is to get $T(n) \leq an \lg n + b$

$$\leq an \lg n + b$$

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



Analyzing Quicksort: Average Case

- So $T(n) \leq an \lg n + b$ 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 (pewh!)
- 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

$$\sum_{k=1}^{n-1} k \lg k \leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \quad \textit{The summation bound so far}$$

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

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \quad \textit{The summations overlap in range; combine them}$$

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



Tightly Bounding The Key Summation

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

The summation bound so far

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

Rearrange first term, place upper bound on second

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

X Guassian series

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

Multiply it all out



Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} (n^2 \lg n - n \lg n) - \frac{1}{8} n^2 + \frac{n}{4}$$
$$\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \text{ when } n \geq 2$$

Done!!!



Quicksort in practice

- Quicksort is great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from *code tuning*.
- Quicksort behaves well even with caching and virtual memory.



