4-2 Find the missing integer from 0 to n using $O(n)$ "is bit[j] in A[i]" queries.

Note - there are a total of $n \lg n$ bits, so we are not allowed to *read* the entire input!

Also note, the problem is asking us to minimize the number of bits we read. We can spend as much time as we want doing other things provided we don't look at extra bits.

How can we find the last bit of the missing integer?

Ask all the n integers what their last bit is and see whether 0 or 1 is the bit which occurs less often than it is supposed to. That is the last bit of the missing integer!

How can we determine the second-to-last bit?

Ask the $\approx n/2$ numbers which ended with the correct last bit! By analyzing the bit patterns of the numbers from 0 to n which end with this bit.

By recurring on the remaining candidate numbers, we get the answer in $T(n) = T(n/2) + n = O(n)$, by the Master Theorem.

Quicksort

Although mergesort is $O(n \lg n)$, it is quite inconvenient for implementation with arrays, since we need space to merge.

In practice, the fastest sorting algorithm is Quicksort, which uses *partitioning* as its main idea.

Example: Pivot about 10.

17 12 6 19 23 8 5 10 – before

6 8 5 10 23 19 12 17 – after

Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.

Note that the pivot element ends up in the correct place in the total order!

Partitioning the elements

Once we have selected a pivot element, we can partition the array in one linear scan, by maintaining three sections of the array: $<$ pivot, $>$ pivot, and unexplored.

Example: pivot about 10

— 17 12 6 19 23 8 5 — 10

— 5 12 6 19 23 8 — 17

5 — 12 6 19 23 8 — 17

5 — 8 6 19 23 — 12 17

5 8 — 6 19 23 — 12 17

5 8 6 — 19 23 — 12 17

5 8 6 — 23 — 19 12 17

5 8 6 ——23 19 12 17

5 8 6 10 19 12 17 23

As we scan from left to right, we move the left bound to the right when the element is less than the pivot, otherwise we swap it with the *rightmost unexplored* element and move the right bound one step closer to the left.

Since the partitioning step consists of at most n swaps, takes time linear in the number of keys. But what does it buy us?

1. The pivot element ends up in the position it retains in the final sorted order.

2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

Thus we can sort the elements to the left of the pivot and the right of the pivot independently!

This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem.

Quicksort Animations

Pseudocode

```
Sort(A)
      Quicksort(A,1,n)


Quicksort(A, low, high)
      if (low < high)
            pivot-location = Partition(A,low,high)
            Quicksort(A,low, pivot-location - 1)
            Quicksort(A, pivot-location+1, high)


Partition(A,low,high)
      pivot = A[low]
      leftwall = low
      for i = low+1 to high
            if (A[i] < pivot) then
                  leftwall = leftwall+1
                  swap(A[i],A[leftwall])
      swap(A[low],A[leftwall])
```

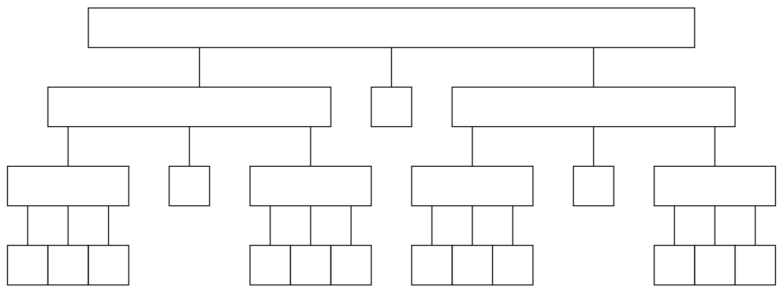
Best Case for Quicksort

Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take?

The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.

The partition step on each subproblem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is $O(n)$.

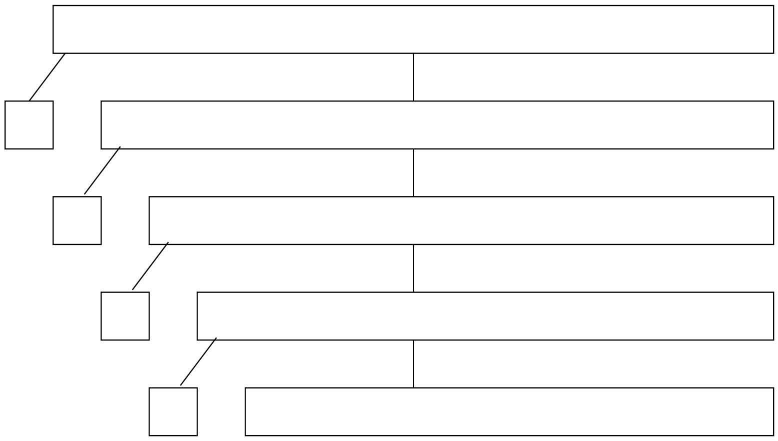
The recursion tree for the best case looks like this:



The total partitioning on each level is $O(n)$, and it take $\lg n$ levels of perfect partitions to get to single element subproblems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is $O(n \lg n)$.

Worst Case for Quicksort

Suppose instead our pivot element splits the array as unequally as possible. Thus instead of $n/2$ elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.



Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.

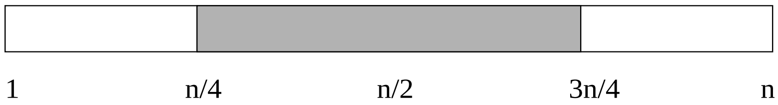
Thus the worst case time for Quicksort is worse than Heapsort or Mergesort.

To justify its name, Quicksort had better be good in the average case. Showing this requires some fairly intricate analysis.

The divide and conquer principle applies to real life. If you will break a job into pieces, it is best to make the pieces of equal size!

Intuition: The Average Case for Quicksort

Suppose we pick the pivot element at random in an array of n keys.



Half the time, the pivot element will be from the center half of the sorted array.

Whenever the pivot element is from positions $n/4$ to $3n/4$, the larger remaining subarray contains at most $3n/4$ elements.

If we assume that the pivot element is always in this range, what is the maximum number of partitions we need to get from n elements down to 1 element?

$$(3/4)^l \cdot n = 1 \longrightarrow n = (4/3)^l$$

$$\lg n = l \cdot \lg(4/3)$$

Therefore $l = \lg(4/3) \cdot \lg(n) < 2 \lg n$ good partitions suffice.

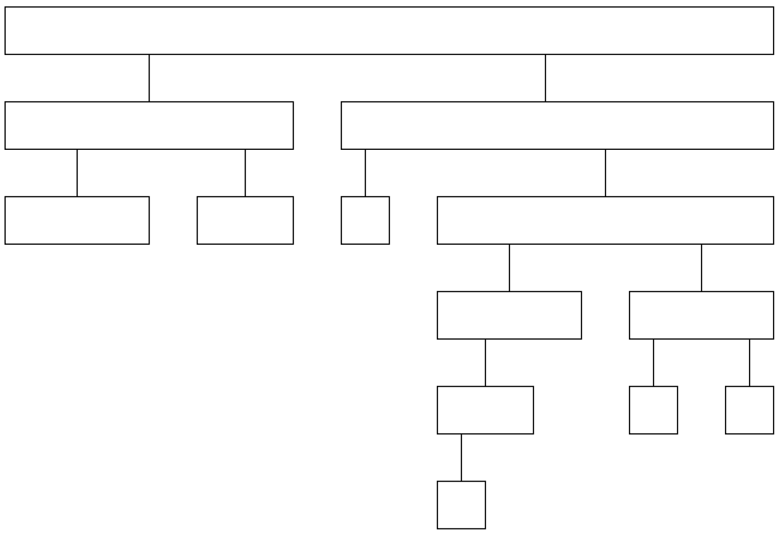
What have we shown?

At most $2 \lg n$ levels of *decent partitions* suffices to sort an array of n elements.

But how often when we pick an arbitrary element as pivot will it generate a decent partition?

Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, we get one half the time on average.

If we need $2 \lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4 \lg n$ levels.



Since $O(n)$ work is done partitioning on each level, the average time is $O(n \lg n)$.

More careful analysis shows that the expected number of comparisons is $\approx 1.38 n \lg n$.

Average-Case Analysis of Quicksort

To do a precise average-case analysis of quicksort, we formulate a recurrence given the exact expected time $T(n)$:

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}(T(p-1) + T(n-p)) + n - 1$$

Each possible pivot p is selected with equal probability. The number of comparisons needed to do the partition is $n-1$.

We will need one useful fact about the Harmonic numbers H_n, namely

$$H_n = \sum_{i=1}^{n} 1/i \approx \ln n$$

It is important to understand (1) where the recurrence relation comes from and (2) how the log comes out from the summation. The rest is just messy algebra.

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}(T(p-1) + T(n-p)) + n - 1$$

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

$$nT(n) = 2\sum_{p=1}^{n} T(p-1) + n(n-1) \quad \text{multiply by n}$$

$$(n-1)T(n-1) = 2\sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2) \quad \text{apply to n-1}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

rearranging the terms give us:

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

substituting $a_n = A(n)/(n+1)$ gives

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$

$$a_n \approx 2\sum_{i=1}^{n} \frac{1}{(i+1)} \approx 2 \ln n$$

We are really interested in $A(n)$, so

$$A(n) = (n+1)a_n \approx 2(n+1)\ln n \approx 1.38 n \lg n$$

What *is* the Worst Case?

The worst case for Quicksort depends upon how we select our partition or pivot element. If we always select either the first or last element of the subarray, the worst-case occurs when the input is already sorted!

A B D F H J K

B D F H J K

D F H J K

F H J K

H J K

J K

K

Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.

To eliminate this problem, pick a better pivot:

1. Use the middle element of the subarray as pivot.

2. Use a *random* element of the array as the pivot.

3. Perhaps best of all, take the median of three elements (first, last, middle) as the pivot. Why should we use median instead of the mean?

Whichever of these three rules we use, the worst case remains $O(n^2)$. However, because the worst case is no longer a natural order it is much more difficult to occur.

Is Quicksort really faster than Heapsort?

Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.

But how can we compare two $\Theta(n \lg n)$ algorithms to see which is faster? Using the RAM model and the big Oh notation, we can't!

When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort. The primary reason is that the operations in the innermost loop are simpler. The best way to see this is to implement both and experiment with different inputs.

Since the difference between the two programs will be limited to a multiplicative constant factor, the details of how you program each algorithm will make a big difference.

If you don't want to believe me when I say Quicksort is faster, I won't argue with you. It is a question whose solution lies outside the tools we are using.

Randomization

Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.

If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.

But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at *random*.

Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:

> "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time."

Where before, all we could say is:

> "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."

Since the time bound how does not depend upon your input distribution, this means that unless we are *extremely* unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.

Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

The worst-case is still there, but we almost certainly won't see it.