Random Algorithms

Randomize the order of the input array:

(1) Either prior to calling insertion sort,
(2) or during insertion sort

// We can avoid best or worst case by randomizing array ourselves

Algorithm 1 int[] insertionSort(int $A[1 \ldots n]$)

```
//Insert code to randomize order of array here
j = 2;
while j <= n do
    key = A[j];
    i = j − 1;
    while (i > 0) and (A[i] > key) do
        A[i + 1] = A[i];
        i − −;
    end while
    A[i + 1] = key;
    j + +;
end while
return A;
```

// Input array now based on randomized order instead of initial array

- disadvantages
  - cost time to randomize
  - Can't take advantage of best case scenario (array in sorted order)

This makes the runtime depend on a probabilistic experiment (depending on the sequence of numbers obtained from random number generator).

Since runtime is independent of input order, no assumptions need to be made about input distribution. In addition, No one specific input elicits worst-case behavior (the worst case is determined only by the output of a random number generator).

Runtime is a random variable (maps sequence of random numbers to runtimes)

Expected runtime = expected value of runtime

Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts "in place" (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- We are going to perform an expected runtime analysis on randomized quicksort

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

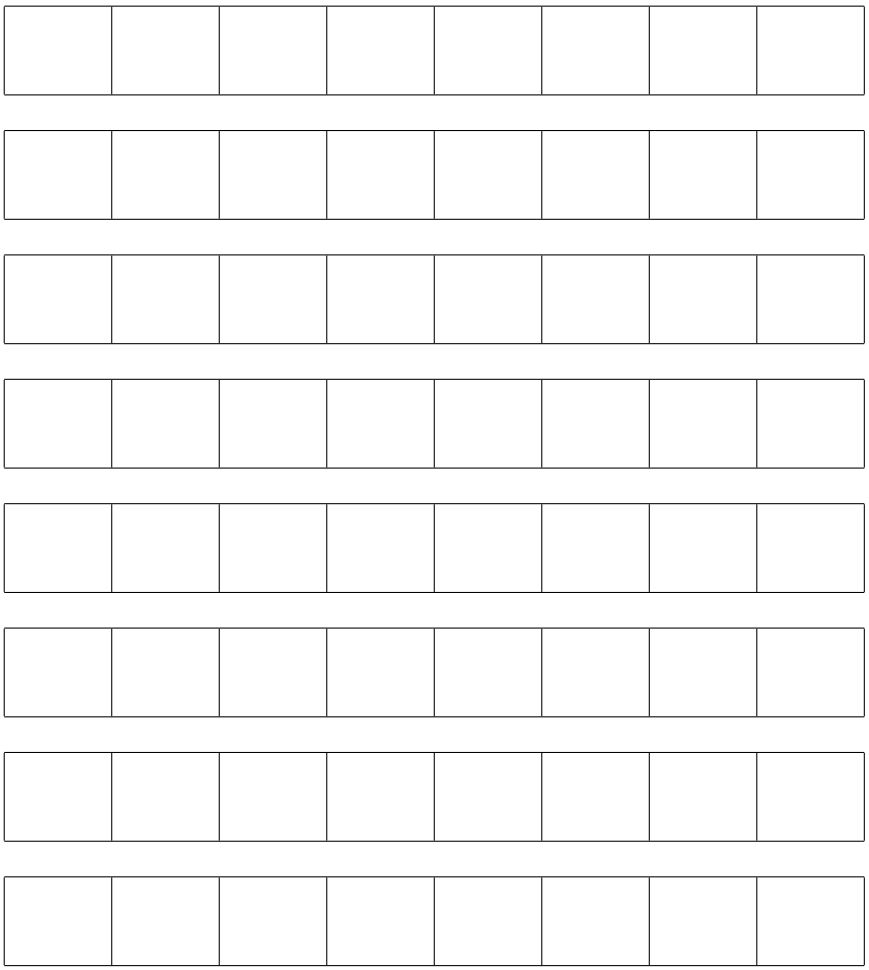
(2) **Conquer:** Recursively sort the two subarrays.
(3) **Combine:** Trivial.

Key idea: Linear-time partitioning subroutine.

Algorithm 2 int partition(int $A[1 \ldots n]$, int p, int q)

```
x = A[p];
i = p;
j = p + 1;
while j <= q do
    key = A[j];
    if A[j] ≤ x then
        i = i + 1;
        exchange A[i] and A[j]
    end if
    j + +;
end while
exchange A[i] and A[p] (Puts pivot into correct sorted position)
return i;
```

Example of partitioning:



T(n)

T(??)

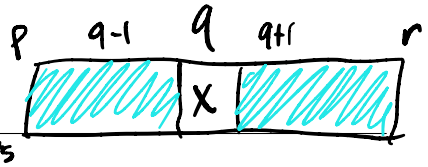
T(??)

Algorithm 3 void quicksort(int $A[1 \ldots n]$, int p, int r)

if $p < r$ **then**
  q =partition(A,p,r);
  quicksort(A,p,q-1);
  quicksort(A,q+1,r);
end if

Θ(n) = n is size of Array

Θ(r-p)
└ # elements = (r-p)



Initial call: quicksort(A,1,n);

* //pivot you choose dictates what T(??) will be

Analysis of quicksort

Assume all input elements are distinct. (In practice, there are better partitioning algorithms for when duplicate input elements may exist.)

Let $T(n)$ denote the worst-case runtime of quicksort on an array of n elements.

What is the worst case for partitioning our array into the two subarrays?

//T(0) because we already sorted entire array

$$T(n) = T(n-1) + T(0) + n$$
$$= T(n-1) + \Theta(1) + n$$

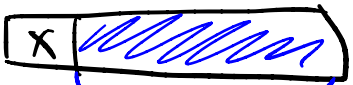
cost to partition

This occurs when when the input array is either sorted or reverse sorted.

=>

(n-1)

or



(n-1)

=> T(n) = T(n-1) + n

↳ This is where recursion is done thus, T(n-1) recursive calls

Worst case: $T(n) = T(n-1) + n$

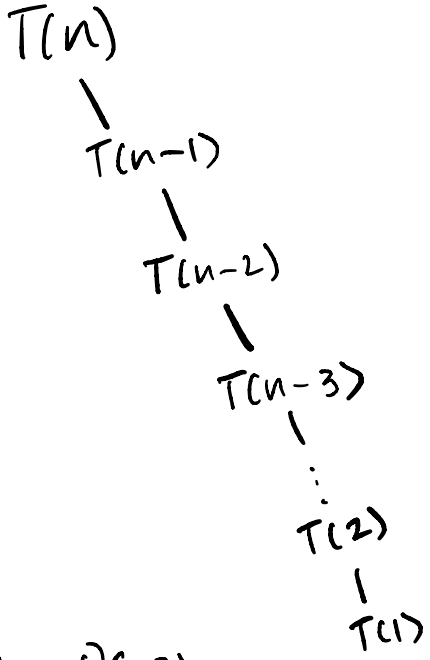
Input sorted or reverse sorted. Partition around min or max element. In this case, one side of the partition always has no elements.

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

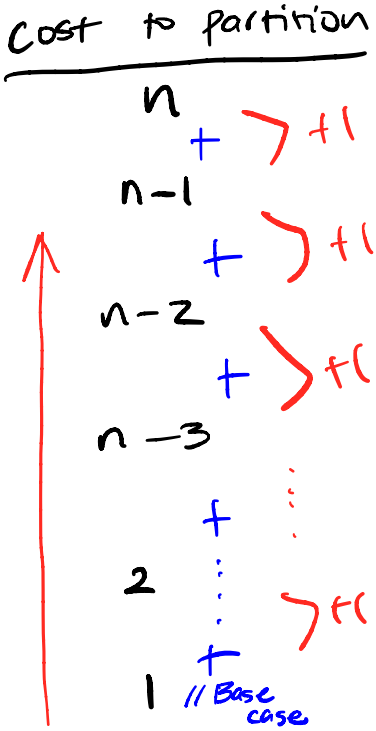
//what does this solve to?
=> do recursion tree to see

Worst case recursion tree:

//LHS has T(0) on e/a node but that is not useful



//trivially sorted

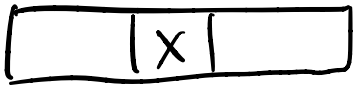


$$\Rightarrow \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

//Found this by noticing we are iterating 1 element at a time
=> Arithmetic Series

What is the best case for partitioning our array into the two subarrays?

This balances cost of recursion



$\Rightarrow T(n) = 2T(n/2) + n$ // same recurrence of merge sort $= \theta(n \log n)$

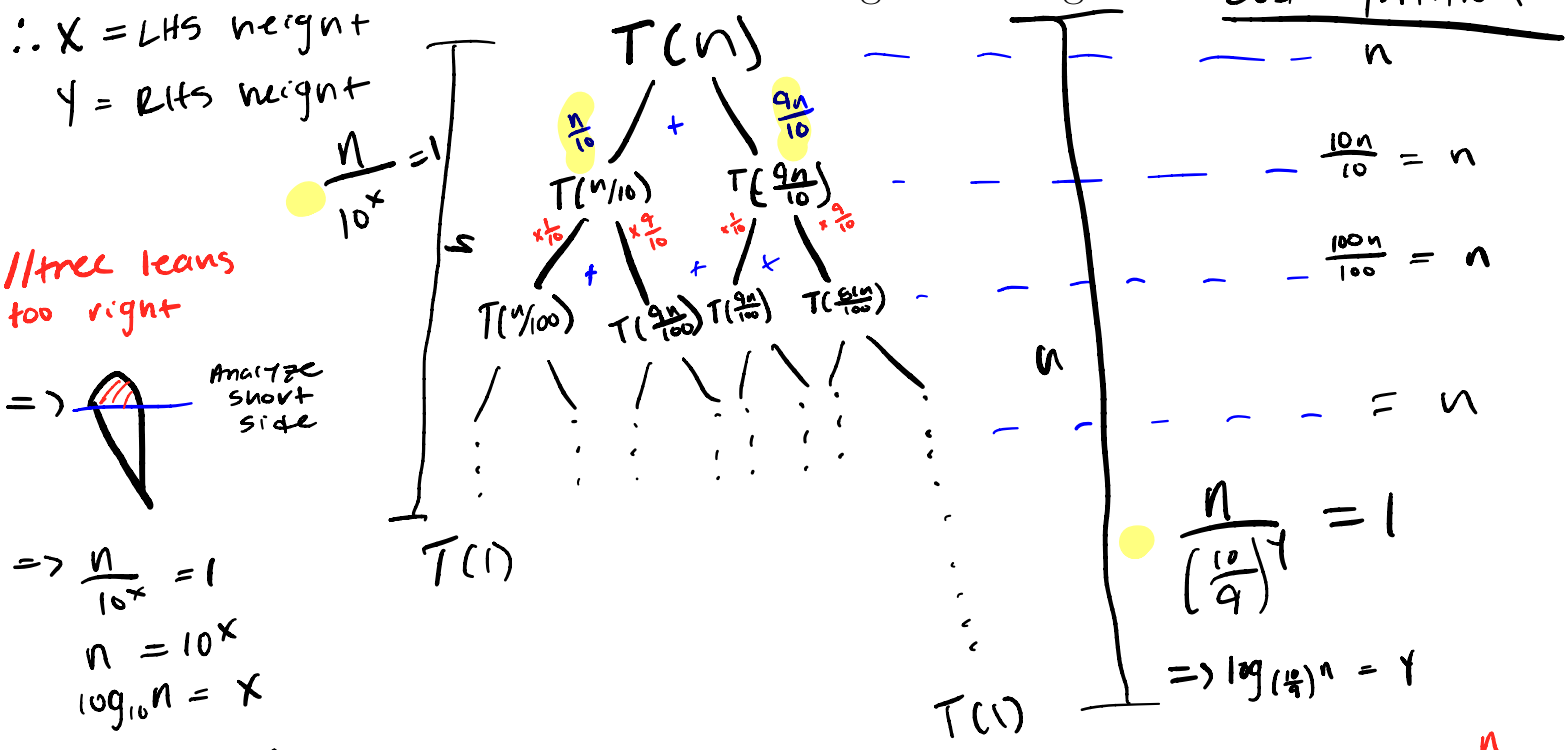
What is the run time for an "almost-best" case? Specifically, what if the split is always $\frac{1}{10} : \frac{9}{10}$?

$\Rightarrow T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

What is the solution to this recurrence?

Find upper & lower bound to get a $T(n)$

Use the recursion tree method to generate a guess.



∴ X = LHS height

Y = RHS height

// tree leans too right

$\Rightarrow \frac{n}{10^x} = 1$

$n = 10^x$

$\log_{10} n = x$

$\Rightarrow h = \log_{10} n$

$\Rightarrow \log_{(10/9)} n = y$

$n \cdot \log_{10} n \leq T(n) \leq n \cdot \log_{(10/9)} n$ // $n \cdot \frac{9}{10} = \frac{n}{(10/9)}$

lower upper

// (cost per level) · (tree height)

$\Rightarrow$ logs only differ by constant factor

$\Rightarrow \theta(n \log n)$ // Trivial true

Best case runtime: $\theta(n \log n)$ //common (Average)

Worst case runtime: $\theta(n^2)$ // rare

Given this worst case runtime we might assume that quicksort is worse than merge sort. But, as it turns out, this is not the case at all.

Specifically, its average runtime $T_{avg}(n) \in O(n \log n)$.

In addition, we can create a randomized quicksort which will have an expected runtime in $O(n \log n)$.

Randomized quicksort

Key idea: Randomly choose an element from the array to use as the pivot in our partition function. // Sorted array (or reverse) no longer our worst case

The runtime is independent of the input order. It depends only on the sequence s of random numbers No specific input elicits the worst-case behavior. Instead, The worst case is determined only by the sequence s of random numbers.

Let $T(n)$ denote the runtime of randomized quicksort on an input of size n. Let X be a RV representing the number of comparisons done by quicksort on an input size of n.

Clearly $T(n) \in O(n + X)$ — determines runtime of algorithm

read array

$\Rightarrow T(n) \in O(X)$

So Let's find $E(X)$ (i.e. expected number of comparisons done by quicksort).

Firstly let $\{z_1, z_2, z_3, \ldots, z_n\}$ be the n elements in the array after being sorted by quicksort.

We define the RV X_{ij} as follows:

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

Thus the total number of comparisons, X, can be defined as X_{ij} summed up for all i and j pairs ($i \neq j$). In quicksort, values are only compared when one of them is the pivot. Consequently, each comparison is done at most once.

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

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

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij})$$

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

Computing the probability z_i is compared to z_j is bit tricky. Let's instead consider when z_i and z_j are NOT compared. If any pivot is chosen in the range $\{z_i, z_{i+1}, z_{i+2}, \ldots z_j\}$ // sorted order other than z_i or z_j then z_i and z_j will never be compared (since they're on different sides of the pivot). Thus:

↳ pick an element between z_i & z_j then they are never compared

$$
\begin{aligned}
&P(z_i \text{ is compared to } z_j) \\
&= P(z_i \text{ or } z_j \text{ is the first pivot chosen in the range } z_i, \ldots, z_j) \\
&= P(z_i \text{ is the first pivot chosen in the range } z_i, \ldots, z_j) + \text{or} \\
&\quad P(z_j \text{ is the first pivot chosen in the range } z_i, \ldots, z_j) \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \qquad \text{// Found using => high - low + 1} \\
&= \frac{2}{j-i+1} = E(X_{ij})
\end{aligned}
$$

We can then plug this probability in for $E(X_{ij})$

$$
\begin{aligned}
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \qquad \text{// want to start at 1, not } j=i+1 \text{ (subtract } i \text{ } j \text{ } j=i+1 \Rightarrow k=j-i) \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \qquad \text{// using sum rules (nth harmonic \#)} \\
&= \sum_{i=1}^{n-1} O(\log n) \qquad = \log n + \log n + \ldots + \log n \\
&= O(n \log n) \qquad\qquad n-1 \text{ times} = n
\end{aligned}
$$

looking for upper bound

Thus, the expected number of comparisons done by quicksort is $O(n \log n)$

$\Rightarrow T(n) \in \Theta(n \log n)$ expected value

Alternate Proof

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ T(2) + T(n-3) + \Theta(n) & \text{if } 2 : n-3 \text{ split} \\ \vdots & \vdots \\ T(n-2) + T(1) + \Theta(n) & \text{if } n-2 : 1 \text{ split} \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

$$\begin{aligned}
&E(T(n)) \\
&= E\left(\sum_{k=0}^{n-1}(1/n)(T(k) + T(n-k-1) + \Theta(n))\right) \\
&= E\left((1/n)\sum_{k=0}^{n-1}(T(k) + T(n-k-1) + \Theta(n))\right) \\
&= (1/n)\sum_{k=0}^{n-1} E(T(k) + T(n-k-1) + \Theta(n)) \\
&= (1/n)\sum_{k=0}^{n-1}(E(T(k)) + E(T(n-k-1)) + \Theta(n)) \\
&= (1/n)\sum_{k=0}^{n-1} E(T(k)) + (1/n)\sum_{k=0}^{n-1} E(T(n-k-1)) + (1/n)\sum_{k=0}^{n-1}\Theta(n) \\
&= (1/n)\sum_{k=0}^{n-1} E(T(k)) + (1/n)\sum_{k=0}^{n-1} E(T(k)) + (1/n)\sum_{k=0}^{n-1}\Theta(n) \\
&= (2/n)\sum_{k=0}^{n-1} E(T(k)) + (1/n)\sum_{k=0}^{n-1}\Theta(n) \\
&= (2/n)\sum_{k=0}^{n-1} E(T(k)) + (1/n)(n)\Theta(n) \\
&= (2/n)\sum_{k=0}^{n-1} E(T(k)) + \Theta(n)
\end{aligned}$$

Guess that $E(T(k)) = ck \log k$

$E(T(n, s))$
$= (2/n) \sum_{k=0}^{n-1} E(T(k)) + \Theta(n)$ (From above)
$\leq (2/n) \sum_{k=0}^{n-1} ck \log k + \Theta(n)$
. (From fact that $\sum_{k=0}^{n-1} k \log k \leq (1/2)n^2 \log n - (1/8)n^2$)
$\leq (2/n)c((1/2)n^2 \log n - (1/8)n^2) + \Theta(n)$
$= cn \log n - c(1/4)n + \Theta(n)$
$= cn \log n - cn/4 + \Theta(n)$
$= cn \log n - cn/4 + d_n n$ (where d_n is a constant)
$= cn \log n + n(d_n - c/4)$ (choose $c/4 \geq d_n$)
$\leq cn \log n + 0$

$\Rightarrow E(T(n)) \in \Theta(n \log n)$