Unit 7: QuickSort, Randomness, Lower bound

Agenda:

- QuickSort : Worst, Best and Average case
- Random QuickSort
- Lower bound for sorting

Reading:

- CLRS: 170-193

Quicksort: Another sorting meets divide-and-conquer

- The ideas:
  - Pick one key ($pivot$), compare it to all others.
  - Rearrange A to be: [elements $< pivot$, $pivot$, elements $> pivot$]
  - Recursively sort subarrays before and after the $pivot$.
- Pseudocode:

```
procedure Quicksort(A, p, r)
    ** sorts the subarray A[p, ..., r]
    if (p < r) then
        q ← Partition(A, p, r)
            ** Partition returns q such that (i) A[q] = pivot,
            **(ii) All elements < pivot appear in A[p, ..., q − 1]
            **(iii) All elements > pivot appear in A[q + 1, ..., r]
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

- How will you prove `QuickSort`'s correctness?
- By induction on $n = \#\text{ elements in } A = r - p + 1$
  - Your base case needs to be both $n = 1$ and $n = 0$. (Why?)
  - Induction step is easy if we know `Partition()` is correct

Partition(A, p, r):

- procedure Partition(A, p, r)

```
    ** last element, A[r], is the pivot key picked of the partition
    pivot ← A[r]
    i ← p − 1   ** i is the location of the last element known to be ≤ pivot
    for (j from p to r − 1) do
        if (A[j] ≤ pivot) then
            i ← i + 1
            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```

- Example: $A[1..8] = \{3, 1, 7, 6, 4, 8, 2, 5\}$, $p = 1$, $r = 8$, $pivot = A[8] = 5$

3	1	7	6	4	8	2	5	$i = 0, j = 1$
3	1	7	6	4	8	2	5	$i = 1, j = 2$
3	1	7	6	4	8	2	5	$i = 2, j = 3$
3	1	7	6	4	8	2	5	$i = 2, j = 4$
3	1	7	6	4	8	2	5	$i = 2, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 5$
3	1	4	6	7	8	2	5	$i = 3, j = 6$
3	1	4	6	7	8	2	5	$i = 3, j = 7$
3	1	4	2	7	8	6	5	$i = 4, j = 7$
3	1	4	2	5	8	6	7	$i = 4, j = 8$ (for-loop ended)

`Partition`(A, p, r):

- The *pivot* happens to be $A[r]$.

- Works by traversing the keys of the array from p to $r - 1$.
- j indicates the current element we are considering.
- i is the index of the last known element which is $\leq pivot$.

- The invariant:
  - $i < j$
  - $A[p..i]$ contains keys $\leq pivot$
  - $A[(i+1)..(j-1)]$ contains keys $> pivot$
  - $A[j..(r-1)]$ contains keys yet to be compared to $pivot$
  - $A[r]$ is the $pivot$
- Ideas:
  - $A[j]$ is the current key
  - If $A[j] > pivot$ — no need to change i or exchange keys as the invariant is maintained
  - If $A[j] \leq pivot$, exchange $A[j]$ with the first larger-then-pivot element ($A[i+1]$) and increment i to maintain the fact that $A[p, ..., j]$ is built from two consecutive subarrays of elements $\leq pivot$ and elements $> pivot$
  - At the end, exchange $A[r] \leftrightarrow A[i+1]$ such that:
    - $A[p..i]$ contains keys $\leq pivot$
    - $A[i+1]$ contains $pivot$
    - $A[(i+2)..r]$ contains keys $> pivot$

Quicksort correctness:

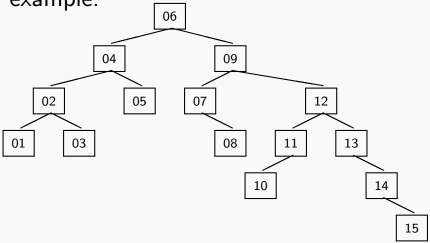
- Proof by induction. The hardest part: the correctness of `Partition()`.
- `Partition` correctness:
  `Partition` returns s such that $A[p,...,s]$ contain elements $\leq A[s]$ and $A[s+1,...,r]$ contains elements $> A[s]$.
  - Loop invariant:
    - $A[p..i]$ contains keys $\leq A[r](\stackrel{\text{def}}{=} pivot)$
    - $A[(i+1)..(j-1)]$ contains keys $> A[r]$
    - $i < j$
  - Proof of LI:
    1. Initialization: $i = p-1$, $j = p$. Both arrays are empty and $i < j$.
    2. Maintenance:
       If $A[j] > pivot$: we don't increment i (so $i < j+1$), so first claim holds trivially, and any $y \in A[(i+1),...,(j-1),j]$ satisfies $y > pivot$.
       If $A[j] \leq pivot$: we exchange $A[i+1] \leftrightarrow A[j]$, so now $A[p,...,i,(i+1)]$ contain elements $\leq pivot$; and now $A[(i+2),...,j]$ contains the same keys that *were* in places $(i+1),...,(j-1)$ as the start of the iteration. As we increment both i and j, $i+1 < j+1$.
    3. Termination: Clearly the for-loop ends as each operation inside it takes constant time and j never decreases. At the end, $j = r$ thus $A[p,..i]$ and $A[(i+1),...(r-1)]$ satisfy the requires and $i \leq r-1$.
  - LI $\Rightarrow$ remainder of proof:
    First, $p-1 \leq i < j \leq r$ as only gets incremented but the LI is maintained, so replacing $A[i+1] \leftrightarrow A[r]$ is fine ($i+1$ is an index between p and r).
    By replacing $A[i+1] \leftrightarrow A[r]$ we now make sure that $A[p...(i+1)]$ all contains elements $\leq pivot$ and $A[i+2,...,r]$ contains elements $> pivot$. As `Partition` returns $i+1$ then it points exactly to the new location of $pivot$.
- Runtime $= \Theta(p-r)$ as each iteration in the for-loop takes constant time.

QuickSort Analysis

- Why we study QuickSort and its analysis:
  - very efficient, in use
  - divide-and-conquer
  - huge literature
  - a model for analysis of algorithms
  - a terrific example of the usefulness of *randomization*
- Observations:
  - (Again) key comparison is the dominant operation
  - Counting KC
    — *only* need to know (at each call) the rank of the split key

QuickSort recursion tree:

- root = pivot, left and right children = trees for the first and second recursive calls
- An example:



- More observations:
  - In the resulting recursion tree, at each node
    (all keys in left subtree) $\leq$ (key in this node) $<$ (all keys in right subtree)
  - 1-1 correspondence:
    quicksort recursion tree $\longleftrightarrow$ binary search tree

QuickSort Running Time

- Like before - dominated by #KC.
- The pivot is compared with every other key: $(n-1)$ KC
- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n > 2 \end{cases}$$

  where $0 \leq n_1 \leq n-1$
- This raises the question: how can we estimate what n_1 is going to be?
- There is no single answer.
- Instead, we will try different thought-experiments.

QuickSort WC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \le 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n > 2 \end{cases}$$

- Notice that when both subarrays are non-empty, then #KC in next level is

$$(n_1 - 1) + (n - 1 - n_1 - 1) = (n - 3)$$

  But when one subarray is empty then #KC in next level is $(n-2)$.
- WC recurrence:

$$T(n) = T(0) + T(n-1) + (n-1) = T(n-1) + (n-1),$$

- Solving the recurrence — Master Theorem doesn't apply

$$\begin{array}{rcl} T(n) & = & T(n-1) + (n-1) = T(n-2) + (n-2) + (n-1) \\ & = & \ldots \\ & = & T(1) + 1 + 2 + \ldots + (n-1) \\ & = & \frac{(n-1)n}{2} \end{array}$$

  So, $T(n) \in \Theta(n^2)$
- Therefore, quicksort is *bad* in terms of WC running time !
- What is a worst-case instance for QuickSort?

QuickSort WC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n-1-n_1) + (n-1), & n > 2 \end{cases}$$

- Let's try an almost-WC situation.
- At every step, we find a pivot for which $n_1 = n - 2$.
- WC recurrence:

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

- Solving the recurrence —

$$\begin{aligned} T(n) &= T(n-2) + (n-1) = T(n-4) + (n-3) + (n-1) \\ &= \ldots \\ &= T(1) + 1 + 3 + \ldots + (n-3) + (n-1) \end{aligned}$$

- Clearly $T(n) \leq \frac{n}{2}(n-1) \in O(n^2)$, and also $T(n) \geq \frac{n}{4} \cdot \frac{n}{2} \in \Omega(n^2)$. So, $T(n) \in \Theta(n^2)$.
- QuickSort has bad running time when the pivot is close to the endpoints.

QuickSort BC running time:

- Recurrence:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n_1) + T(n - 1 - n_1) + (n - 1), & n > 2 \end{cases}$$

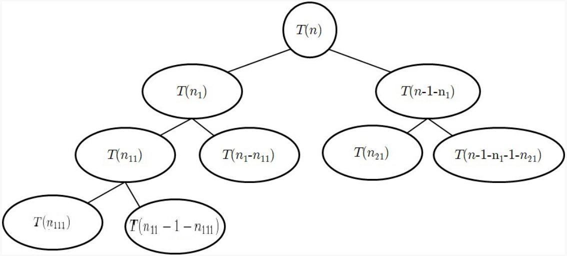
- Notice that when both subarrays are non-empty, we are saving 1 KC ...
- Best case: each partition is a bipartition !!!
  Saving as many KC as possible every level ...
  The recursion tree is as short as possible ...
- Recurrence:

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

- Solving the recurrence — Master Theorem $T(n) = 2T(\frac{n}{2}) + (n - 1)$ solves to $\Theta(n \log(n))$.

OPTIONAL: Visualize QuickSort BC running time (cont'd):

- Let us draw the recursion tree of QuickSort, assuming the pivot is never the largest or the smallest element (so there are two recursive calls, one on size n_1 subarray and one on size $n - 1 - n_1$ subarray).



- Question: In the recursion tree, what is the number of KC at each level?

OPTIONAL: Visualize QuickSort BC running time (cont'd):

- Let us draw the recursion tree of QuickSort, assuming the pivot is never the largest of the smallest element (so there are two recursive calls, one on size n_1 subarray and one on size $n-1-n_1$ subarray).
- Question: In the recursion tree, what is the number of KC at each level?
- Answer:
  - one node (root) at the top level and we make $n-1$ KC
  - 2 nodes at the 2nd level, and regardless of the value of n_1 in the two nodes together we do $(n_1-1)+(n-1-n_1-1)=n-3$ KC
  - 4 nodes at the 3rd level, and regardless of the value of n_{11} and n_{21} in all 4 nodes to total #KC we do is $(n_{11}-1)+(n_1-1-n_{11}-1)+(n_{21}-1)+(n-1-n_1-n_{21}-1)=n-7$
  - ...
  - at ith level — if we have all 2^{k-i} nodes — we do $n-(1+2+3+..+2^{i-1})=n-2^i+1$ KC
- Therefore, the best to minimize the #KC is to minimize the number of layers in the tree. Best case: $\lceil \log(n) \rceil$ levels — complete binary tree
- So, we make $\sum_{i=1}^{\log(n)+1}(n-2^i+1)$ KC, and

$$\sum_{i=1}^{\log(n)+1}(n-2^i+1)=(n+1)(\log(n)+1)-(4n-2)\in\Theta(n\log n)$$

- Exercise: prove this bound for $n=2^k-1$. (Induction)

QuickSort BC running time:

- Best case: each partition is a bipartition !!!
  Saving as many KC as possible every level ...
  The recursion tree is as short as possible ...
- Recurrence:
  $$T(n) = 2 \times T(\frac{n-1}{2}) + (n-1),$$
- Solving the recurrence — $T(n) \in \Theta(n \log n)$
- Question:
  - What is the best case array for the case of $\{1, 2, ..., 7\}$?
  - $A = [1, 3, 2, 6, 5, 7, 4]$. What about $\{1, 2, ...n = 2^k - 1\}$?

QuickSort Almost-BC running time:

- Let's assume the at each round we get an approximated bi-partition Namely, each split is $\frac{3}{4}n$ and $\frac{1}{4}n$.
- Recurrence:

$$T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + (n-1), \qquad \text{with } T(0) = T(1) = 0$$

- Then:

$$\begin{aligned}
T(n) &= n - 1 + T(\tfrac{3}{4}n) + T(\tfrac{1}{4}n) \\
&= \big(n - 1 + \tfrac{3n}{4} - 1 + \tfrac{n}{4} - 1\big) \\
&\quad + T(\tfrac{3}{4}^2 n) + T(\tfrac{3}{4}\cdot\tfrac{1}{4}n) + T(\tfrac{1}{4}\cdot\tfrac{3}{4}n) + T(\tfrac{1}{4}^2 n) \\
&= (2n-3) + T(\tfrac{3}{4}^2 n) + 2T(\tfrac{3}{4}\cdot\tfrac{1}{4}n) + T(\tfrac{1}{4}^2 n) \\
&= \Big(2n - 3 + (\tfrac{3}{4}^2 n - 1) + 2(\tfrac{3}{4}\cdot\tfrac{1}{4} - 1) + (\tfrac{1}{4}^2 n - 1)\Big) \\
&\quad\ + T(\tfrac{3}{4}^3 n) + T(\tfrac{3}{4}^2\cdot\tfrac{1}{4}n) + 2T(\tfrac{3}{4}^2\cdot\tfrac{1}{4}n) + 2T(\tfrac{3}{4}\cdot\tfrac{1}{4}^2 n) + T(\tfrac{3}{4}\cdot\tfrac{1}{4}^2 n) \\
&= (3n-7) + T(\tfrac{3}{4}^3 n) + 3T(\tfrac{3}{4}^2\cdot\tfrac{1}{4}n) + 3T(\tfrac{3}{4}\cdot\tfrac{1}{4}^2 n) + T(\tfrac{1}{4}^3 n) \\
&\ \vdots \\
&= (kn - (2^k - 1)) + \sum_{j=0}^{k} \binom{k}{j} \cdot T(\tfrac{3}{4}^j \cdot \tfrac{1}{4}^{k-j}) \cdot n)
\end{aligned}$$

QuickSort Almost-BC running time:

- Let's assume we have an approximated bi-partition
  Namely, each split is $\frac{3}{4}n$ and $\frac{1}{4}n$.
- Recurrence:

$$T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + (n-1), \qquad \text{with } T(0) = T(1) = 0$$

- Then:

$$T(n) = \ldots = (kn - (2^k - 1)) + \sum_{j=0}^{k} \binom{k}{j} \cdot T(\frac{3}{4}^j \cdot \frac{1}{4}^{k-j}) \cdot n)$$

- Since not all branches of the recurrence end at the same layer, arguing an exact solution isn't simple
- But the shortest root→leaf path is $\log_4 n$ and longest is $\log_{4/3}(n)$.
  - At $k = \log_4(n)$ we already have $T(n) \geq n\log_4(n) - \sqrt{n} = \Omega(n\log(n))$
  - At $k = \log_{4/3}(n)$ the summation is at most $k \cdot n = O(n\log(n))$.
- Exercise: Prove that $T(n) \leq 100n\log(n)$ for $n \geq 4$.
- Hence, if we have a "substantial" partition, we get $T(n) \in \Theta(n\log(n))$.

QuickSort Average Case running time:

- The recurrence for running time is:
$$T(n) = \begin{cases} 0, & \text{if } n = 0, 1 \\ T(n_1) + T(n - 1 - n_1) + (n - 1), & \text{if } n \geq 2 \end{cases}$$
- Average case: "What is the probability for the left subarray to have size n_1?"
- Average case: always ask "average over what input distribution?"
The moment we look at average case, we make a huge assumption about the data. If it doesn't hold, our analysis doesn't apply.
And that is why we normally prefer WC.
  - Here, we assume each possible input equiprobable, i.e. *uniform distribution*.
  - Here, each of the ____ possible inputs equiprobable
  - Key observation: equiprobable inputs imply for each key, rank among keys so far is equiprobable
So, n_1 can be $0, 1, 2, \ldots, n-2, n-1$, with the same probability $\frac{1}{n}$

Solving $T(n)$:

- As $\Pr[n_1 = i] = \frac{1}{n}$ for every i we get

$$\begin{aligned}
T(0) &= 0 \\
T(1) &= 0 \\
T(n) &= (n-1) + \tfrac{1}{n}\left(T(0) + T(n-1)\right) \\
&\qquad + \tfrac{1}{n}\left(T(1) + T(n-2)\right) \\
&\qquad\quad + \ldots \\
&\qquad\qquad + \tfrac{1}{n}\left(T(n-2) + T(1)\right) \\
&\qquad\qquad\quad + \tfrac{1}{n}\left(T(n-1) + T(0)\right) \\
&= (n-1) + \tfrac{2}{n}\sum_{i=0}^{n-1} T(i)
\end{aligned}$$

- Master Theorem does NOT apply here.
- But you can guess and check that $T(n) \leq 2(n+1)[H(n+1) - 1]$

  (The harmonic number $H(n) = \sum_{i=1}^{n} \frac{1}{i} \leq \ln(n) + 1$)

OPTIONAL: Solving $T(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$:

- Therefore,
  - $n \cdot T(n) = 2\sum_{i=0}^{n-1} T(i) + n(n-1)$
  - $(n-1) \cdot T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)(n-2)$
- Subtract the two terms:

$$n \cdot T(n) - (n-1) \cdot T(n-1) = 2T(n-1) + 2(n-1)$$

- Rearrange it:

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

- And with some arithmetics:

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} = \frac{T(n-1)}{n} + \frac{2n}{n(n+1)} - \frac{2}{n(n+1)} \\ &= \frac{T(n-1)}{n} + \frac{2}{n+1} - 2\left(\frac{1}{n} - \frac{1}{n+1}\right) \\ &= \frac{T(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}\end{aligned}$$

OPTIONAL: Solving $T(n)$ (cont'd):

- which gives you (iterated substitution)

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n} = \ldots = \sum_{i=1}^{n} \frac{2}{i+1} + \left(\frac{2}{n+1} - 2\right)$$

  Recall that $\sum_{i=1}^{n} \frac{1}{i} = H(n) = \ln n + \gamma$ — the Harmonic number where $\gamma \approx 0.577\cdots$

- So, from

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

  we have

$$\begin{aligned} T(n) &= 2(n+1)H(n+1) - (4n+2) \\ &\approx 2(n+1)(\ln(n+1) + \gamma) - (4n+2) \\ &\in \Theta(n \log n) \end{aligned}$$

- Conclusion: Running time of QuickSort on AC *on the uniform distribution* is $\Theta(n \log n)$.

QuickSort Improvement and space requirement:

- QuickSort is considered an in-place sorting algorithm:
  - extra space required at each recursive call is only constant.
  - whereas in MergeSort, at each recursive call up to $\Theta(n)$ extra space is required.
- To improve the algorithm we use a **random** pivot

Difference between a Randomized Algorithm and AC Analysis

- AC analysis means we make an assumption on the input
  - No guarantee that the assumption holds
  - Input is chosen once: on avg we have might have a good running time, but once input is given our running time is determined.
- A randomized algorithm works for *any* input (WC analysis)
  - Randomness in the coins we toss (not in the input) so we control the distribution of the coin toss
  - We can always start the algorithm anew if it takes too long; or run it multiple times in parallel

Randomized QuickSort

- We invoke `RandomPartition` rather than `Partition`
- Pseudocode

```
procedure RandomPartition(A, p, r)
    i ← uniformly chosen random integer in {p, ..., r}
    exchange A[i] ↔ A[r]
    return Partition(A, p, r)
```

- Q: How do we analyze the #KC of the Random QuickSort?
  Depends on which pivot we pick, we can compare any two elements.
  And of course, there is a chance we pick the worst-pivot (last element) in every iteration...
- A: We analyze the *expected* WC #KC
  (As always, proportional to Expected WC running time)

Probability 101

- A *random variable* takes values in some range according to a *probability distribution*.
  - E.g. $\Pr[X = 0] = 0.5$, $\Pr[X = 3] = 0.19$,
    $\Pr[X = -11] = 0.22$, $\Pr[X = 2\pi] = 0.09$

  Probability is non-negative and sums to 1.
- The *expectation* of a random variable is a weighted average of the outcome according to the probability distribution: $\mathrm{E}[X] = \sum_x x \cdot \Pr[X = x]$
  - In our example: $\mathrm{E}[X] = 0 \times 0.5 + 3 \times 0.19 + (-11) \times 0.22 + 2\pi \times 0.09$
- *Bernoulli* random variables: random variable that takes values in $\{0, 1\}$ indicating whether some event happened or didn't happen.
  - E.g., X is a Bernoulli random variable indicating whether my coin toss came up Heads. That is, $X = 1$ iff the coin-toss was "Heads"
  - And $\mathrm{E}[X] = 1 \times \Pr[heads] + 0 \times \Pr[tails] = \Pr[heads]$. So for a fair coin: $\mathrm{E}[X] = 0.5$.
  - This is an example of the following phenomena: when X is a Bernoulli r.v. indicating whether some event did happen or did not happen, if this event happens with probability p then

$$\mathrm{E}[X] = 1 \times \Pr[\text{event happened}] + 0 \times \Pr[\text{event didn't happened}] = p$$

Probability 101 (Cont'd)

- Expectation has a beautiful property — Expectation is *linear*.
  - For **any** two random variables X and Y, define the random variable $Z = X + Y$. Then we have that $\mathrm{E}[Z] = \mathrm{E}[X] + \mathrm{E}[Y]$.
  - Q: I toss a coin $1,000,000$ times, what is the expected number of heads?
  - A: Let X_j denote the Bernoulli random variable indicating whether $toss_j = heads$. Define $Z = \sum_{j=1}^{1,000,000} X_j$. Then

$$\mathrm{E}[Z] = \mathrm{E}[\sum_{j=1}^{1,000,000} X_j] \overset{\substack{\text{linearity}\\\text{of}\\\text{expectation}}}{=} \sum_{j=1}^{1,000,000} \mathrm{E}[X_j] = 1,000,000 \times \Pr[coin = heads]$$

- Motivation for studying expectation:
  - As the name implies, tossing the coin many times means we expect to see about $\mathrm{E}[Z]$ 'Heads'. In our example: it becomes *really* unlikely, for a fair coin, to see $< 495,000$ heads...
  - For our randomized algorithm: if we feel the algorithm makes far more than $\mathrm{E}[\#\mathrm{KC}]$ comparisons — abort it and start anew
  - Or have multiple *independent* instances running in parallel and use whichever halts first.

Probability 101 (Cont'd)

- Given two random variables X and Y, the *conditional probability* of X given $Y=y$ is defined as: $\Pr[X=x|\ Y=y] = \frac{\Pr[X=x \text{ and } Y=y]}{\Pr[Y=y]}$.
  - E.g., given that I have tossed a fair die and got a number ≥ 3, what is the probability that the outcome of the die is 6?

$$\Pr[\text{die}=6|\ \text{die}\geq 3] = \frac{\Pr[\text{die}=6 \text{ and } \text{die}\geq 3]}{\Pr[\text{die}\geq 3]} = \frac{\Pr[\text{die}=6]}{\Pr[\text{die}\geq 3]} = \frac{1/6}{4/6} = \frac{1}{4}$$

- Two random variables X and Y are called *independent* if for any outcomes x and y we have $\Pr[X=x|\ Y=y] = \Pr[X=x]$
  - Intuitively, Y doesn't effect X and vice-verse
  - Suppose I I roll a die twice, X is the result of the first roll and Y is the result of the second roll — then X & Y are independence.
  - But X and $Z \stackrel{\text{def}}{=} X+Y$ are *not* independent
- If X and Y are independent random variables, then

$$\mathrm{E}[X\cdot Y] = \mathrm{E}[X]\cdot \mathrm{E}[Y]$$

Back to Randomized QuickSort

- Our goal: bound *expected* $\#$KC
- Technique $\#1$: Recurrence relation for the expectation.
  Let $T(n)$ denote the *expected $\#$KC in an array of size n*.
  This is the technique you should be proficient in.
- Technique $\#2$: Sum of expected Bernoulli random variables.
  Denote $X_{i,j}$ as the Bernoulli random variable indicating whether Random QuickSort compared a_i with a_j.
  (a_t is the t-th largest elements in the sorted array.)
  It follows that $\mathrm{E}[\#\mathrm{KC}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}]$.
  This is for you to see how elegant can a random-algorithm's analysis be.

Randomized QuickSort Runtime Analysis #1

- Our goal: bound *expected* #KC
- Technique #1: Recurrence relation for the expectation.
  Let $T(n)$ denote the *expected #KC in an array of size n*.
- Like before, if we know `Partition` put n_1 keys in one side of the pivot and $n - n_1 - 1$ on the other side, then
  $T(n) = T(n_1) + T(n-1-n_1) + (n-1)$
  Key point: We're using the same function (#KC on instances of size $n_1, n-1-n_1$) **because of independence!**
- Moreover, $n_1 = k$ means that the pivot is the $(k+1)$-largest element in A.
- Since the pivot it chosen uniformly at random, then

$$\Pr[n_1 = k] = \Pr[\text{pivot} = (k+1)\text{-largest element}] = \tfrac{1}{n}$$

- Thus

$$T(n) \overset{\substack{\text{Definition}\\\text{of}\\\text{Expectation}}}{=} \sum_{k=0}^{n-1} \Big(T(k) + T(n-1-k) + (n-1)\Big) \cdot \Pr[n_1 = k]$$

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

- Now solve to find a close-form solution for $T(n)$.
  We already solved this recurrence relation: $T(n) \in \Theta(n\log(n))$

Randomized QuickSort Runtime Analysis #2

- Our goal: bound *expected* #KC
- Technique #2: Denote $X_{i,j}$ as the Bernoulli random variable indicating whether Random QuickSort compared a_i with a_j.
  It follows that $\mathrm{E}[\#\mathrm{KC}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}]$.
- So let's analyze $\mathrm{E}[X_{i,j}]$. Fix some i, j s.t. $i < j$.
- At each iteration of random QuickSort, the pivot is chosen randomly between $\{a_p, a_{p+1}, ..., a_r\}$.
  - If we pick pivot$= a_i$ or pivot$= a_j$ — we compare all keys to the pivot so $X_{i,j} = 1$.
  - If we pick $a_i <$pivot$< a_j$ — we place a_i and a_j in two separate partitions and *never* compare them, so $X_{i,j} = 0$.
  - If we pick $a_p \leq$pivot$< a_i$ or $a_j <$pivot$\leq a_r$ — we don't know the value of $X_{i,j}$ and need to wait to the next recursive call.
    - However, with each level of the recursion the number of elements decreases, so at some point we will pick a pivot between a_i and a_j.
- Hence:

$$\begin{aligned}
\mathrm{E}[X_{i,j}] = \sum_t & \Pr[\text{pivot} = a_i \text{ or pivot} = a_j | a_i \leq \textit{pivot} \leq a_j] \\
& \cdot \Pr[\text{in round } t \text{ we picked } a_i \leq \textit{pivot} \leq a_j] \\
= & \Pr[\text{pivot} = a_i \text{ or pivot} = a_j | a_i \leq \textit{pivot} \leq a_j] \cdot \sum_t \Pr[\text{in round } t ...] \\
= & \Pr[\text{pivot} = a_i \text{ or pivot} = a_j | a_i \leq \textit{pivot} \leq a_j]
\end{aligned}$$

Randomized QuickSort Runtime Analysis #2 (cont'd)

- What is the probability of pivot$= a_k$ *given* that we pick a pivot$\in \{a_i, ..., a_j\}$? (a_k is in this set)

$$\begin{aligned} \Pr[\text{pivot} = a_k | \text{ pivot} \in \{a_i, ..., a_j\}] &= \tfrac{\Pr[\text{pivot}=a_k \text{ and pivot}\in\{a_i,...,a_j\}]}{\Pr[\text{pivot}\in\{a_i,...,a_j\}]} \\ &= \tfrac{\Pr[\text{pivot}=a_k]}{\Pr[\text{pivot}\in\{a_i,...,a_j\}]} \\ &= \tfrac{1/(r-p+1)}{(j-i+1)/(r-p+1)} = \frac{1}{j-i+1} \end{aligned}$$

- This makes sense: pivot chosen uniformly at random — so given that the pivot is between a_i and a_j, it is equiprobable that the pivot would be any key a_k among the $j-i+1$ keys between a_i and a_j.
- Conclusion:
  $\mathrm{E}[X_{i,j} = 1] = \Pr[\text{pivot} = a_i \text{ or pivot} = a_j | a_i \le pivot \le a_j] = \frac{2}{j-i+1}$.
- Hence,

$$\mathrm{E}[\#\mathrm{KC}] = \sum_{i=1}^{n-1}\sum_{j>i}^{n} \mathrm{E}[X_{i,j}] \le \sum_{i=1}^{n-1} 2\left(\tfrac{1}{1} + \tfrac{1}{2} + ... + \tfrac{1}{n-i+1}\right) = 2\sum_{i=1}^{n-1} H(n-i+1)$$

- So we have $\mathrm{E}[\#\mathrm{KC}] \le 2\sum_{i=1}^{n} H(n) \le 2n \cdot H(n) \in O(n\log(n))$;
- and also $\mathrm{E}[\#\mathrm{KC}] \ge 2\sum_{i=1}^{n/2} H(n/2) \ge 2\frac{n}{2}(\ln(n) - 1) \in \Omega(n\log(n))$.
- Conclusion: runtime of Random QuickSort $\in \Theta(n\log(n))$. ■

Sorting Algorithms So Far: Running Time Comparison

Alg.	BC	WC	AC
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$
Sort via AVL/RB-Tree	$\Theta(n \log n)$[1]	$\Theta(n \log n)$	$*$
QuickSort	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
Random QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$*$

What it "$*$"?

[1] We haven't formally shown this, but in a BST the average height of a node is $\Omega(\log(n))$. So inserting n elements into a BST has to cost $\Omega(n \log(n))$.

Sorting lower bound:

- So far: we looked at BC runtime — for lower bounds purposes.
  - They serve as lower bounds, for specific algorithms.
  - E.g., "Even in the best case, my algorithm makes _ KC."
  - So this is a lower bound of the form

$$\exists \text{ algoritm } A \text{ s.t. } \forall \text{ input } I, \quad \text{runtime}(A(I)) > ....$$

- We now give a lower bound for the problem of sorting.
  - A lower bound on *any* algorithm for sorting - even those not invented yet.
  - This is a lower bound of the form

$$\forall \text{ algoritm } A \; \exists \text{ input } I, \quad \text{runtime}(A(I)) > ....$$

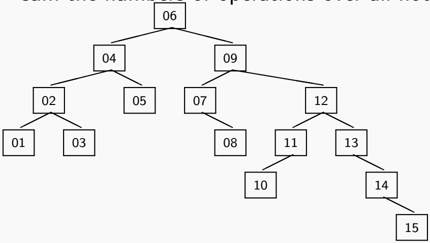
- Q: Can we derive a lower bound of the form

$$\forall \text{ algoritm } A \text{ and } \forall \text{ input } I, \quad \text{runtime}(A(I)) > ....?$$

- A: Not a very informative bound, since for every input I_0 we can always "massage" any algorithm into an algorithm that first checks for I_0.
  `If` (input$= I_0$) `return` solution(I_0) `else ...` (run original algorithm)

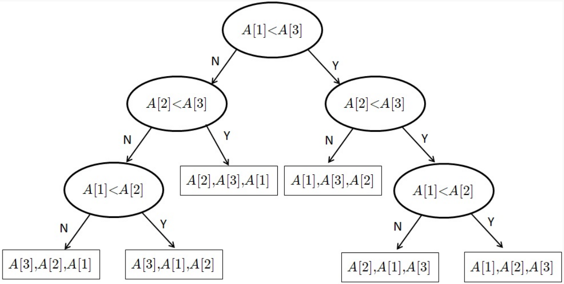
Two useful trees in algorithm analysis:

- Recursion tree
  - node $\longleftrightarrow$ recursive call
  - describes algorithm execution for one particular input by showing all calls made
  - one algorithm execution $\longleftrightarrow$ all nodes (a tree)
  - useful in analysis:
    sum the numbers of operations over all nodes



Two useful trees in algorithm analysis:

- Recursion tree
- Decision tree
  - node $\longleftrightarrow$ algorithm decision
  - describes algorithm execution for all possible inputs by showing all possible algorithm decisions
  - one algorithm execution $\longleftrightarrow$ one root-to-leaf path
  - useful in analysis:
    sum the numbers of operations over nodes on one path



Sorting lower bound:

- Consider comparison-based sorting algorithms. These algorithms map to decision trees (nodes have exactly 2 children).
- Binary tree facts:
  - Suppose there are t leaves and k levels. Then,
  - $t \leq 2^{k-1}$
  - So, $\lg t \leq (k-1)$
  - Equivalently, $k \geq 1 + \lg t$
    — binary tree with t leaves has *at least* $(1 + \lg t)$ levels
- Comparison-based sorting algorithm facts:
  - Look at its *Decision Tree*. It's a binary tree.
  - It should contain every possible output: every permutation of the positions $\{1, 2, \ldots, n\}$.
  - So, it contains at least $n!$ leaves ...
  - Equivalently, it has at least $1 + \lg(n!)$ levels.
  - A longest root-to-leaf path of length at least $\lg(n!)$.
  - So in the worst case, the algorithm makes at least $\lg(n!)$ KC, and $\lg(n!) \in \Theta(n \log n)$

Sorting lower bound:

- Same argument — only in contradiction.
  - Suppose that there exists an algorithm making $< \lg(n!)$ comparisons for *any* instance.
  - Based on the algorithm's comparisons and the sequence of Yes/No answers it gets, the algorithm outputs a permutation of $\{1, 2, .., n\}$.
  - The output is determined by the algorithm's queries and which one of the $< n!$ sequences of Yes/No answers it saw.
  - But there are $n!$ possible permutations.
  - So – at least two permutations result in the same Yes/No sequence.
  - These two permutations look the same for the algorithm, which means the algorithm returns the same output on both.
  - This means the algorithm errs on at least one of these permutations.
- Such a lower bound is called an *information theoretic* lower bound.
  - There are x possible outputs, each query-operation gives me one binary bit of information, so I must make $\log_2(x)$ queries that "encode the output."
- Information theoretic lower bound hold also for randomized algorithms.
  - The analysis is of the same spirit, but a tad hairier as we show $\Pr[\text{making few queries}]$ is small.
- We conclude that MergeSort, HeapSort, BalancedBST-Tree Sort and Random QuickSort are all asymptotically optimal (comparison-based) sorting algorithms.

Additional lower bounds:

- We can use a similar argument to prove a linear-time lower-bound for most problems
  - ASOC some sub-linear algorithm for this problems exists. Observe that such an algorithm cannot even **read** the entire input.
  - So find two different inputs that map to two different outputs, whose difference lies solely on the part of the input our algorithm does not read.
- Example: `FindMin()` must take $\Omega(n)$-time.
  - ASOC some algorithm ALG runs in sub-linear time. Then this algorithm cannot even read the entire input. Fix ALG and let i be the index that it disregards. (We assume that ALG is deterministic.)
  - Let A and B be two arrays which are identical on all cells but $A[i]$ and $B[i]$. In A we have $A[i]$ is the smallest element and in B we have that $B[i]$ is the largest element.
  - As ALG ignores the ith cell, both inputs look the same to ALG so it must produce the same output for the `FindMin()` problem.
  - Hence ALG is wrong either on A or on B.
- If ALG is a random algorithm, we argue that there has to be an index i that ALG ignores with probability $\geq 1/2$ (and the proof continues as before).
  - Otherwise, for any i we have $\Pr[ALG \text{ reads index } i] \geq 1/2$
  - Thus, because of linearity of expectation (let X_i be the Bernoulli r.v. indicating whether ALG read the ith entry) we have $\mathrm{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{E}[X_i] \geq \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$.
    and so ALG's expected runtime has to be at least $\frac{n}{2}$ — linear in n.

Sorting lower bound:

- Corollary 1: There does not exists a data-structure that implements both `Insert()` and `Successor()` in time $o(\log(n))$.
- Proof: Suppose such a data-structure existed. We will use it to sort A in time $o(n\log(n))$.
  - Insert all elements to this data-structure
    By assumption, takes $n \cdot o(\log(n)) = o(n\log(n))$ time.
  - Find $x \leftarrow$ min-element in A (naïvely, in $O(n) = o(n\log(n))$ times)
  - Starting with x: (1) print(x), (2) set $x \leftarrow$`Successor(`x`)`
    By assumptions, takes $o(n\log(n))$-time
- Overall runtime is in $o(n\log(n))$ — contradiction to sorting-lower bound.
- Corollary 2: In any Priority-Queue that we can build in time $o(n\log(n))$, `ExtractMax()` must take $\Omega(\log(n))$ time.
- Proof – HW. (Similar proof: ASOC that such a Priority-Queue does exist, use it to sort in $o(n\log(n))$-time.)

Lessons Learned:

1. Recursion
2. Correctness analysis.
   - Induction for recursions
   - Loop-invariant for loops
3. Runtime analysis.
   - Identify a "key" step
   - WC, BC, AC
   - Asymptotic analysis
4. Divide and Conquer — a powerful paradigm for algorithmic design
   - Sometimes you can reduce the number of recursive calls using clever tricks
5. Data structures (arrays, lists, heaps & priority-queues, BST)
6. Randomness — very useful tool
   - Allows us to get improved worst-case guarantees, for any input the expected-runtime is smaller then deterministic runtime.
7. Recursion trees (to get a sense of runtime), Decision trees (to get a sense of algorithm's execution over all possible inputs).
8. Lower bounds — power of negative thinking