Sorting and order statistics

Cormen et al. Section II, Chapters 7, 8, 9

Sorting

- Given an input array A (e.g. a `list` in Python) which contains distinct integers, output an array with the same elements but in increasing order
- We allow ourselves to mutate (modify) the input, so the goal is to manipulate A such that it contains the same elements but when we're done we have $A_1 < A_2 < \cdots < A_n$
- In math indices starting from 1 are common
- In most programming languages we have 0-based indexing: `A[0] < A[1] < … < A[n-1]`
- We use n for the length of the array

Quicksort (Hoare 1961)

- If A has length 0 or 1 there is nothing to be done.
- Choose an element of A to be the **pivot** p (uniformly at random)
- **Partition** A so that for some index q we have
  - $A_1, \ldots, A_{q-1}$ contains the elements $< p$ (in any order)
  - $A_q = p$
  - $A_{q+1}, \ldots, A_n$ contains the elements $> p$ (in any order)
- **Recursively** quicksort $A_1, \ldots, A_{q-1}$ and $A_{q+1}, \ldots, A_n$ separately

Simple Python code

```
import random

def quicksort(arr):
    if len(arr) <= 1: return arr
    pivot = random.choice(arr)                  # Choose the pivot uniformly (optional)
    left = [x for x in arr if x < pivot]        # Elements smaller than pivot
    middle = [x for x in arr if x == pivot]     # Elements equal to pivot
    right = [x for x in arr if x > pivot]       # Elements larger than pivot
    return quicksort(left) + middle + quicksort(right)
```

In-place Python code

```
def partition(A, lo, hi):
      """Partition the subarray of A indexed by range(lo, hi)"""”
      i = randrange(lo, hi)
      A[i], A[hi-1] = A[hi-1], A[i]
      i = lo
      for j in range(lo, hi-1):
            if A[j] <= A[hi-1]:
                  A[i], A[j] = A[j], A[i]
                  i = i + 1
      A[i], A[hi-1] = A[hi-1], A[i]
      return i
```

Examples

```
A = [5,4,1,2,0,3]
```

```
def partition(A, lo, hi):
      # omit randomization of pivot…
      i = lo
      for j in range(lo, hi-1):
            if A[j] <= A[hi-1]:
                  A[i], A[j] = A[j], A[i]
                  i = i + 1
      A[i], A[hi-1] = A[hi-1], A[i]
      return i
```

More in-place Python code

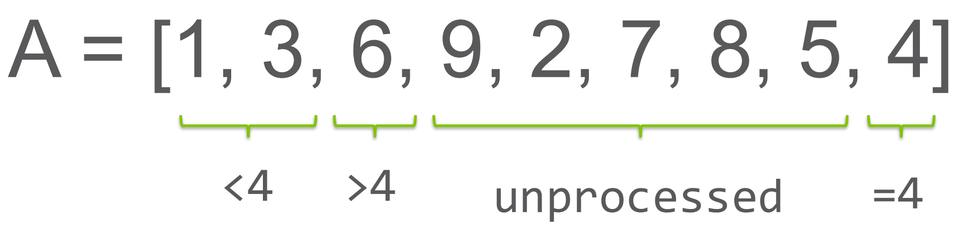
```
def quicksort(A, lo=None, hi=None):
      """Quicksort the subarray of A indexed by range(lo, hi)"""
      if lo is None or hi is None: lo = 0; hi = len(A)       # Allows us to call quicksort(A)
      if lo < hi - 1:                                        # What does this line do?
            q = partition(A, lo, hi)
            quicksort(A, lo, q)
            quicksort(A, q+1, hi)
```

Which implementation is faster?

- The naïve implementation iterates over the array three times!
- The naïve implementation does $3n$ comparisons when partitioning an array of length n
- The in-place implementation does $n - 1$ comparisons when partitioning an array of length n
- Surely fewer is faster…
- Not if the other operations are expensive!
- In Python list access is slow
- The code `A[i], A[j] = A[j], A[i]` is readable but awful, it allocates temporary tuples
- You may find the naïve implementation runs **faster**, and this is because the list comprehensions are language functions that bypass Python-level inefficiencies
- See this colab for a few benchmarks, including one where we compile to C to put the two implementations on more level footing (an optimization to avoid intermediate tuples comes for free and is very important)

Proving correctness 1

- It's possible to prove that `partition(A, lo, hi)` does what it's supposed to with a **loop invariant**
- The idea is that as the loop over `j` progresses, we have the following:
  - For `k` such that `lo <= k <= i-1` we have `A[k] < A[hi-1]`
  - For `k` such that `i <= k <= j-1` we have `A[k] > A[hi-1]`
- This means that as we process the array, we control 4 regions of `A`:



We skip the details, try it if you're interested!

Proving correctness 2

- It's easier to prove the correctness of `quicksort` assuming that `partition` is correct
- This is by **induction** on the length of the array
- The base cases are clear; there is nothing to do if the array has length 0 or 1
- For the inductive step, we assume the length is ≥ 2 and that quicksort correctly sorts all shorter arrays
- Since partition is correct, we put the pivot in the right place, and the two recursive calls are on strictly shorter arrays
- By induction they get sorted correctly
- Then the final answer must be correct
- Filling in the details is up to you…

```
def quicksort(A, lo, hi):
    if lo < hi - 1:
        q = partition(A, lo, hi)
        quicksort(A, lo, q)
        quicksort(A, q+1, hi)
```

Time complexity

- One nice way to count the time complexity of quicksort is according to the number of **comparisons**
- In the `partition` subroutine we have a loop with some conditions, but each iteration of the loop involves a comparison of two array elements
- We can convince ourselves that up to big-O equivalence the number of elementary operations the code performs is the same as the number of comparisons (see Lemma 7.1 in Cormen et al)
- The number of comparisons we do depends on the input we're processing, so there isn't an easy answer

Examples

[1, 3, 2]
[1, 2, 3]

[1] [3]

[1, 2, 3]
[1, 2, 3]

[1, 2]
[1, 2]

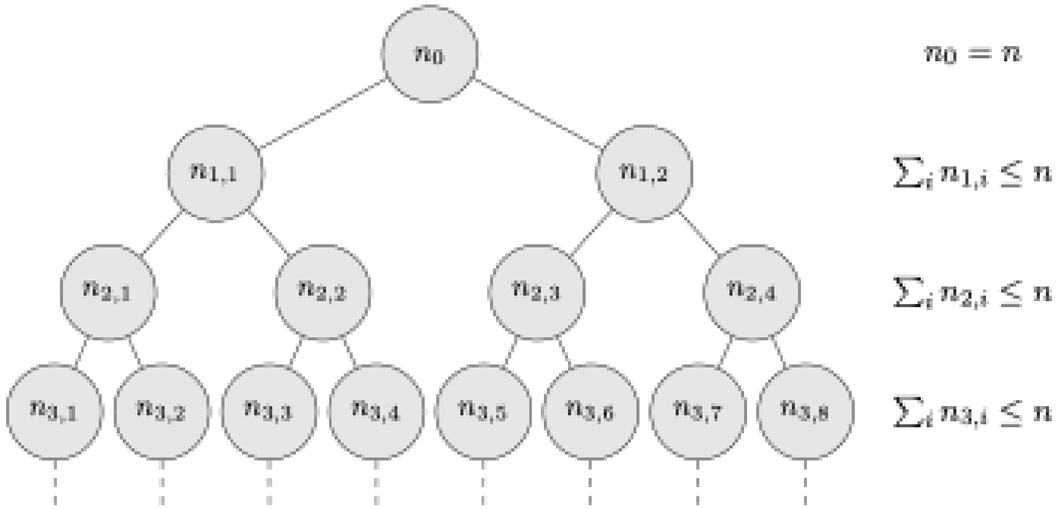
[1]

```
def partition(A, lo, hi):
    i = lo
    for j in range(lo, hi-1):
        if A[j] <= A[hi-1]:
            A[i], A[j] = A[j], A[i]
            i = i + 1
    A[i], A[hi-1] = A[hi-1], A[i]
    return i
```

```
def quicksort(A, lo, hi):
    if lo < hi - 1:
        q = partition(A, lo, hi)
        quicksort(A, lo, q)
        quicksort(A, q+1, hi)
```

Recursion trees

- Each node of the tree is a call to a recursive function, labelled with the size of the input
- The leaves are the base cases of the recursion
- Start with $n_0 = n$ elements, and note that the sum of the labels in a level must be at most n
- Technically, we lose an element for each recursive call as we exclude the pivot from the recursive calls



Analyzing recursion trees

- When we call quicksort on arrays of length 0 or 1, we do 0 comparisons.
- When we call quicksort on arrays of length $n \geq 2$, we do $n - 1$ comparisons and then recursively sort two arrays whose length sums to $n - 1$.
- The recursion tree stores all this information
- An upper bound on the number of comparisons is the sum of all the labels in the tree
- What different sums can we get based on the shape of the tree?

Examples

Best and worst case

- If the tree is somewhat balanced, then we perform $O(n \log n)$ comparisons
- If the tree is very unbalanced, then we perform $\Omega(n^2)$ comparisons
- The clever idea with quicksort is that when the **pivot** is chosen uniformly at random, we are highly likely to get a somewhat balanced tree
- We will analyze the **expected number of comparisons** when we sort an array of n distinct elements where the pivot is chosen uniformly at random at every step

```
from random import randrange
def partition(A, lo, hi):
    i = randrange(lo, hi)
    A[i], A[hi-1] = A[hi-1], A[i]
    i = lo
    for j in range(lo, hi-1):
        if A[j] <= A[hi-1]:
            A[i], A[j] = A[j], A[i]
            i = i + 1
    A[i], A[hi-1] = A[hi-1], A[i]
    return i
```

Linearity of expectation

- Let A be the array to be sorted containing elements 1, 2, …, n
- Let X_{ij} be the indicator random variable for the event that we compare i and j
- Because the algorithm never compares two elements twice, the total number of comparisons is

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

This is because when we call `quicksort(A, lo, hi)` we compare all elements indexed by `range(lo, hi-1)` to the pivot at index `hi-1` and then recursive calls are on ranges that do not contain the pivot

By linearity of expectation, $\mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}X_{ij}$ so it makes sense to compute $\mathbb{E}X_{ij} = \mathbb{P}(i \text{ is compared to } j)$

Probability of comparison

- We compute $\mathbb{P}(i \text{ is compared to } j)$, and without loss of generality we assume that $i < j$
- Consider the elements from i to j inclusive: $A_{ij} = \{i, i+1, \ldots, j\}$
- Quicksort has two important properties
  1. Two elements are compared **if and only if** one of them is chosen as a pivot while the other is still in the **same** recursive subproblem
  2. Once an element is a pivot in one call to quicksort, it is never compared to anything in future calls to quicksort
- Therefore, for i and j to get compared, one of them must be the **first element** out of A_{ij} which gets chosen as a pivot
  - If some p gets chosen as a pivot out of A_{ij} first, then $i < p < j$ so i and j go into different subproblems
  - If i is chosen first out of A_{ij} then it is compared to j
  - If j is chosen first out of A_{ij} then it is compared to i
- Since pivots are chosen uniformly at random whenever they are chosen, each element of A_{ij} is equally likely to be a pivot first. Then $\mathbb{P}(i \text{ is compared to } j) = \frac{2}{j-i+1}$ because the length of A_{ij} is $j - i + 1$

Nasty math

$$\mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Nasty math

$$\mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{2}{k}$$

- Appendix A of Cormen et al. contains a sketch proof of the bound $\sum_{k=2}^{n} \frac{1}{k} \leq \log n$, so

$$\mathbb{E}X \leq \sum_{i=1}^{n-1} 2 \log n \leq 2n \log n$$

- It's more annoying, but one can also show a lower bound $\mathbb{E}X \geq 2n \log n - 4n$ using e.g. $\sum_{k=1}^{n} \frac{1}{k} \geq \log(n+1)$

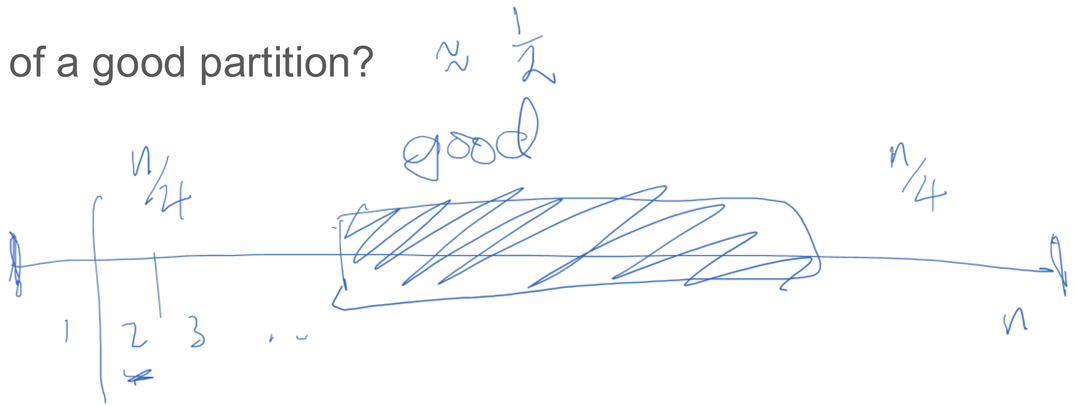
- **Again**, we benefit from understanding linearity of expectation and the consequences of algorithms that make choices uniformly at random

Different analyses

- Exercise 7-3 of Cormen et al gives a different analysis
- Understanding the expected number of comparisons is great, but it doesn't tell the entire story
- Some random variables have sensible expectation, but on very rare occasions they are enormous
- We prove that with probability at least $1 - \frac{1}{n}$ the run of quicksort involves at most $21n \log n$ comparisons
- This is a "with high probability" result that gives more information: we have a specific understanding of how rare a "bad" run of quicksort must be

Recursion tree analysis

- We already discussed the bound that if the recursion tree has depth d then the call to quicksort involves at most dn comparisons
- So what is the probability that we get a depth $d = O(\log n)$?
- We say a partition of n elements is “good” is both parts of the partition have $\leq 3n/4$ elements
- What is the probability of a good partition?



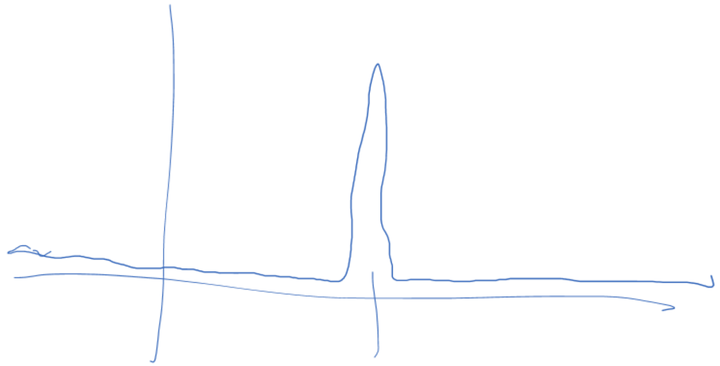
Depth analysis

- We start with n counters $c_1, \ldots, c_n$ all equal to n
- As the algorithm progresses, we let c_i be the size of the recursive subproblem containing element i
- Each c_i starts out at n
- If i is a pivot, we set c_i to 0
- If i is not a pivot, but a **good** pivot is chosen, then c_i goes down by a factor $\frac{3}{4}$
- If i is not a pivot, but a **bad** pivot is chosen, then c_i goes down by at least 1
- We must be done when all counters are at most 1

Good pivot analysis

- The events that i is a pivot or that we have bad pivot choice can only decrease c_i
- The important effect is that a good pivot choice makes c_i go down by a factor ¾
- We know that $c_i \leq \left(\frac{3}{4}\right)^{g_i} n$ when i has been involved in g_i good pivot choices
- If $g_i \geq \log_{4/3} n$ then we must have $c_i \leq 1$ and this branch of the recursion tree terminates
- Since we want $\log_{4/3} n$ good pivot choices, and each choice is good independently with probability $\frac{1}{2}$, **on average** we only need to make $2 \log_{4/3} n$ recursive calls before the branch terminates
- Using fancy math called the **Chernoff bound**, we can show that with probability at most n^{-2} the branch involving i has depth at most $6 \log_{4/3} n \leq 21 \log n$

Chernoff bound



- Suppose that we try $T = c \log_{4/3} n$ pivots
- Each one is good with probability $\frac{1}{2}$, and these are independent events
- Let G be the number of good pivots, and note that $\mathbb{E}G = \frac{c}{2}\log_{4/3} n$
- The Chernoff bound says that for $\delta \in (0,1)$, in this setting

$$\mathbb{P}(G < (1-\delta)\mathbb{E}G) \leq \exp\left(-\frac{\delta^2}{2}\mathbb{E}G\right)$$

If we choose parameters carefully, this implies the bound we need

We set $1 - \delta = 2/c$ and find that $c = 6$ does makes the probability $\leq n^{-2}$ as required

Union bound

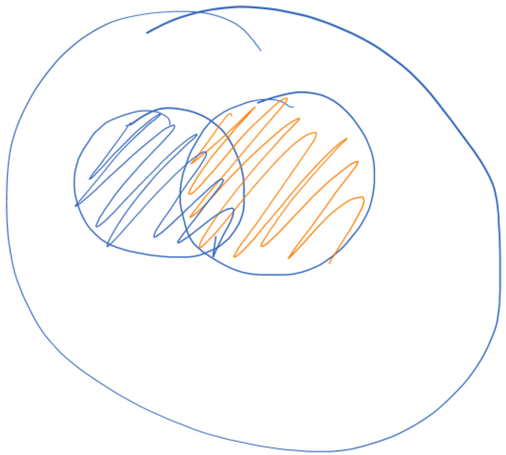
$(1-p)^n \geq 1 - np$

- Let B_i be the **bad** event that the branch involving i has depth more than $21 \log n$
- Then $\mathbb{P}(B_i) \leq n^{-2}$
- If none of the B_i events occur then we know the run of quicksort involved at most $21n \log n$ comparisons
- What is the probability that none of the B_i occur? $\geq 1 - \frac{1}{n}$
- The event that some B_i occurs is $B_1 \cup B_2 \cup \cdots \cup B_n$

$\mathbb{P}(B_1 \cup B_2 \cup \ldots \cup B_n) \leq \sum_{i=1}^{n} \mathbb{P}(B_i) \leq \frac{1}{n}$

$\mathbb{P}(B_1 \cup B_2) = \mathbb{P}(B_1) + \mathbb{P}(B_2) - \mathbb{P}(B_1 \cap B_2)$

$\leq \mathbb{P}(B_1) + \mathbb{P}(B_2)$



$|B_1 \cup B_2|$

$= |B_1| + |B_2|$

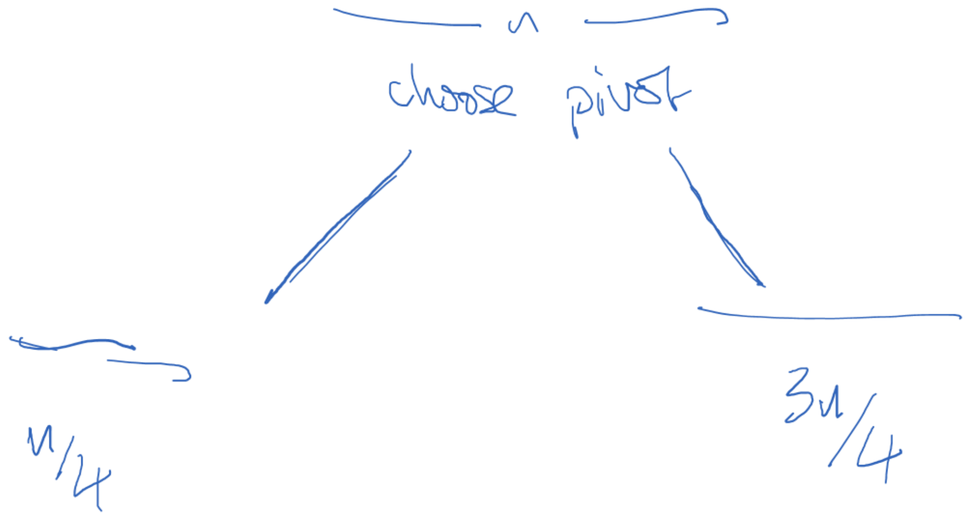
$- |B_1 \cap B_2|$

Order statistics

- The **median** of a list of numbers is the one in the middle
- If the list is of odd length $n = 2k + 1$ then we take the element at position k (counting from 1)
- If the list is of even length $n = 2k$ then we'll decide to take the **lower median:** the element at index k

- Clearly, we can find the median of a **sorted** array in constant time
- But can we find the median **faster** than we can sort?
- Try to modify quicksort to do less work if all you want is the median…

Examples



Quickselect

```
def quickselect(A, k, lo=None, hi=None):
      if lo is None or hi is None: lo = 0; hi = len(A)
      if 1 + lo == hi:
            return A[lo]
      q = partition(A, lo, hi)
      if k == q:        return A[k]
      elif k < q:       return quickselect(A, k, lo, q)
      else:             return quickselect(A, k, q+1, hi)
```

Theorem

The expected running time of quickselect is at most $4n$ when the input has length n

- The proof we give is slightly delicate
- It is possible to extend the analysis we gave for quicksort, but $\mathbb{E}X_{ij}$ is much harder to compute than for quicksort.
- In quicksort, we only “discard” an element when it is chosen as a pivot
- In quickselect, more elements can be discarded since we only recurse into one of the two subarrays either side of the pivot
- We give a more direct upper bound

Expected running time

- Let $T(n)$ be the **expected** number of comparisons of quickselect on arrays of length n
- It's hard to study $T(n)$ directly but we can easily give a strict upper bound
- We assume that we never terminate early because the kth element was the pivot
- We assume that we always get forced to recurse into the larger subarray
- In practice, sometimes we get a fewer comparisons. But under these assumptions we get an upper bound on $T(n)$
- We know that $T(1) = 0$
- We have $T(n) \leq n - 1 + \frac{1}{n}\sum_{i=0}^{n-1} T(\max(i, n-i-1))$
- The parity of n is annoying to handle, but we get an upper bound by assuming that each size from $\left\lfloor \frac{n}{2} \right\rfloor$ to $n-1$ appears twice in the combined sums
- Drawing a picture helps…

Examples

Expected running time

- We want to solve the recurrence $T(1) = 0$ and $T(n) = n - 1 + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i)$
- Let's prove by strong induction on n that $T(n) \leq 4n$
- This clearly holds for $n = 1$
- By induction, when $n \geq 2$ we have $T(n) \leq n - 1 + \frac{8}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} i$
- More annoying math: $\sum_{i=k}^{n-1} i = \sum_{i=1}^{n-1} i - \sum_{i=1}^{k-1} i = \frac{n(n-1)}{2} - \frac{k(k-1)}{2}$ and we have $k = \left\lfloor \frac{n}{2} \right\rfloor \geq \frac{n-1}{2}$
- Then $\sum_{i=\lfloor n/2 \rfloor}^{n-1} i \leq 3n^2/8$
- So $T(n) \leq 4n$ as required