Randomized QuickSort continued!

Randomized QuickSort

\[ \text{pivot} = \text{random} (lo, hi) \]

\[ \text{swap pivot with } lo \]
Randomized Quick sort:

What is the average running time?

Running time: # comparisons between elements

- comparisons happen inside partition
- two elements are compared if one of them is the pivot
  \[ A[t+t_i] \leq A[v] \quad \text{or} \quad A[v] \leq A[t+t_j] \]
- consider the \( k \)-th call to partition and compute the probability that elements \( i \) and \( j \) are compared.
  (indicator variable \( X_{ij}^k = 1 \))
If compared \( 0 \) otherwise)

average of comparisons

\[ E[X_{ij}] = \sum_{k} \text{prob} (X_{ij} = 1) \]

key idea is that if \( A[i,j] \) and \( A[j,i] \) are compared once then they are not compared again because one of them will be the pivot and after that partition, \( \sum_{k} X_{ij} \) is not needed.

Let’s say that the
cell to partition where the components happen is the $i^{th}$ cell.

Pivot selected randomly

\[
\text{prob (i or j are selected)} = \frac{1}{j-i+1} + \frac{1}{j-i+1}
\]

\[
i = 0, \quad j = 9
\]

\[
E[X_{ij}] = \frac{2}{j-i+1}
\]

average total

\[
E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
\]
\[ \frac{h(n-1)}{2} \text{ pair of } i, j \]

\[ = \frac{2}{2} + \frac{2}{3} + \frac{2}{2} + \frac{2}{2} + \frac{2}{2} + \frac{2}{2} + \ldots + \frac{2}{n-1} \]

\[ + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{2}{n-3} \]

\[ = \frac{2}{2} \binom{n-2+1}{1} + \frac{2}{3} \binom{n-3+1}{1} + \frac{2}{4} \binom{n-4+1}{1} + \frac{2}{n} \binom{n-n+1}{1} \]

\[ = \sum_{k=2}^{n} \frac{2}{k} (n-k+1) \]
\[
= \sum_{k=2}^{n} \left( \frac{2n}{k} - 2 + \frac{2}{k} \right)
\]
\[
= 2n \left( \sum_{k=2}^{n} \frac{1}{k} \right) - 2(n-1) + \sum_{k=2}^{n} \frac{1}{k}
\]

\[
\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}
\]

is \(\Theta(\log n)\)

harmonic sum
how can we bound this?
Assume that \(n = 2^k\)

\[
\frac{1}{2} + \frac{1}{4}
\]

\[
\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}
\]

\[
\frac{1}{2} + \frac{1}{2}
\]

\[
\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}
\]

\[
\frac{1}{2^k + \ldots + \frac{1}{2^k}} \leq \frac{1}{2^{k-1}} + \ldots + \frac{1}{2^1}
\]
\[ K \left( \frac{1}{2} \right) \leq \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \leq K \cdot 1 \]

\[ k \text{ was } \log n \]

\[ \frac{1}{2} \log n \leq \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \leq \log n \]

\[ 2 \log n - 2(n-1) + 2 \log n \]

is \( \Theta(n \log n) \)

average of total

number of comparisons

expectation

\[ \text{quick sort: } \min E[T(n)] \quad \text{mergesort: } \min (\omega(2T(n))) \]
Stacks and Queues are data structures (ADT: abstract data type)

**Stack**
- `push`
- `pop`

**Queue**
- `enqueue`
- `dequeue`

**LIFO**
- last in, first out

**FIFO**
- first in, first out

Diagram:

```
client
  `API` interface
    `runtime` implementation
```
this year only
linked list implementation
(no array implementation
no amortized analysis)
linked list for stack

push (str):
oldfirst = first
first = new Node()
first.content = str
first.next = oldfirst

pop():
old = first
content = first.next
return old

Θ(1)
linked list for queue

two pointers: first and last

enqueue (string)
  old last = last
  last = new Node()
  last. content = string
  last. next = null

dequeue()
  same for first

\( \Theta(1) \)
Matching parentheses

Input: \( [\{[ ]\}] \) = \( X \)

Output: True, False

S empty stack

For \( i = 0 \ldots n-1 \)

If \( X[i] \) is opening

S. push(\( X[i] \) )

Else if \( X[i] \) is closing

If \( S \). is Empty

Return False

If \( S \). pop() does \( \text{not match} \) \( X[i] \)

Return False

If \( S \). empty return true

Else return false
Computing the span of an array element
Example Solution:

```
def foo(S):
    total = S[1]
    for i in range(2, len(S)):
        total += S[i] - S[i-1]
    return total
```

Given \( X[1:n] \), define \( S[i] \) as the number of preceding elements \( X[i] \) that are less than or equal to \( X[i] \). The 'X' axis should be from 0 to 10 (Greek letter 'hi').
Keep a stack of the indices visible when you look back!

pop indices until we find an \( j \) such \( i \) larger!

we need some memory for the length of the visibility!
input X
stack A contains indices

"lexicographic" new array $S$

for $i = 0 \ldots n-1$
  while A not empty AND $X[A\text{.top()]} \leq X[i]$
    $A\text{.pop()}$
  if A is empty
    $S[i] = i+1$
  else
    $S[i] = i - A\text{.top()}$
    $A\text{.push}(i)$
return $S \sim$ container

$\Theta(n)$ because while never looks all the way back!