- Randomized algorithms
- Review basics from ``Think like the pros"

Recall

```
QuickSort(low, high)  {
  if (high-low ≤ 1) return;
  partition(low, high) and return split;
  QuickSort(low, split);
  QuickSort(split+1, high);
}
```

Partition rearranges the input array a[low..high] into two (possibly empty) sub-arrays a[low.. split] and a[split+1.. high]

each element in a[low.. split] is ≤ a[split],

each element in a[split.. high] is ≥ a[split].

Recall

```
QuickSort(low, high)  {
  if (high-low ≤ 1) return;
  partition(low, high) and return split,
  QuickSort(low, split);
  QuickSort(split+1, high);
  }
```

The choice of split determines the running time of Quick sort. If the partitioning is balanced, Quick sort is as fast as Merge sort, if the partitioning is unbalanced, Quick sort is as slow as Bubble sort.

Recall

Quick sort(low, high)

```
if (high-low ≤ 1) return;
pivot = a[high-1];
split = low;
for (i=low; i<high-1; i++)
   if (a[i] <pivot) {
      swap a[i] and a[split];
      split++;
   }
swap a[high-1] and a[split];
QuickSort(low, split);
QuickSort(split+1, high);
Return;
```

Partition w.r.t. last element

Recall

Analysis of running time

T(n) = worst-case number of comparisons in Quick sort on an arrays of length n.

- Choosing pivot deterministically:

  the worst case happens when one sub-array is empty and the other is of size n-1, in this case :

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

  $= O(n^2).$

- Choosing pivot randomly we can guarantee

  $T(n) = O(n \log n)$ with high probability

Randomized-Quick sort:

```
R-QuickSort(low, high) {
  if (high-low ≤ 1) return;
  R-partition(low, high) and return split,
  R-QuickSort(low, split-1);
  R-QuickSort(split+1, high);
}

R-partition(low, high)
   i:= random(low, high);
   exchange (a[i],A[low]);
   partition(low,high);
```

We bound the total time spent by Partition

```
Partition(low, high)
pivot = a[high-1];
   split = low;
   for (i=low; i<high-1; i++)
   ★if (a[i] <pivot) {
         swap a[i] and a[split];
         split++;
      }
  swap a[high-1] and a[split];
```

We shall bound X, the number of times the ★ line is executed during entire execution of R-quicksort.

When does the algorithm compare two elements?

When does the algorithm compare to elements?

- Rename array A as $z_1, z_2, \ldots z_n$, with z_i being the ith smallest element
- Define $Z_{ij}:=\{z_i, z_{i+1}, \ldots z_j\}$.

When does the algorithm compare to elements?

- Rename array A as $z_1, z_2, \ldots z_n$, with z_i being the ith smallest element
- Define $Z_{ij} := \{z_i, z_{i+1}, \ldots z_j\}$.
- Note: each pair of elements z_i, z_j is compared at most once.

  Elements are compared with the pivot, after a particular call to Partition that pivot is never used again.

When does the algorithm compare to elements?

- Rename array A as $z_1, z_2, \ldots z_n$, with z_i being the ith smallest element
- Define $Z_{ij} := \{z_i, z_{i+1}, \ldots z_j\}$.
- Note: each pair of elements z_i, z_j is compared at most once. Elements are compared with the pivot, after a particular call to Partition that pivot is never used again.

- Define indicator random variable $X_{ij} := 1 \{ z_i \text{ is compared to } z_j \}$, $X_{ij} := 0 \{ z_i \text{ is not compared to } z_j \}$

When does the algorithm compare to elements?

- Rename array A as $z_1, z_2, \dots z_n$, with z_i being the ith smallest element
- Define $Z_{ij} := \{z_i, z_{i+1}, \dots z_j\}$.
- Note: each pair of elements z_i, z_j is compared at most once. Elements are compared with the pivot, after a particular call to Partition that pivot is never used again.

- Define indicator random variable $X_{ij} := 1 \{ z_i \text{ is compared to } z_j \}$, $X_{ij} := 0 \{ z_i \text{ is not compared to } z_j \}$

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

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

Taking expectation of both sides and the using linearity of E =>

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

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\,[X_{ij}]$$

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

Pr {z_i is compared to z_j} =?

When two elements z_i and z_j are compared?

$\Pr\{z_i \text{ is compared to } z_j\} = ?$

When two elements z_i and z_j are compared? It's useful to think when they are not compared!

Pr {z_i is compared to z_j}=?

When two elements z_i and z_j are compared? It's useful to think when they are not compared!

If some element y, $z_i < y < z_j$ is chosen as pivot, we know that z_i and z_j can not be compared.

Why?

$\Pr\{z_i$ is compared to $z_j\}=?$

When two elements z_i and z_j are compared? It's useful to think when they are not compared!

If some element y, $z_i < y < z_j$ is chosen as pivot, we know that z_i and z_j can not be compared.

Because list of numbers will be partitioned and z_i and z_j will be in two different parts.

Pr {z_i is compared to z_j}=?

When two elements z_i and z_j are compared? It's useful to think when they are not compared!

If some element y, $z_i < y < z_j$ is chosen as pivot, we know that z_i and z_j can not be compared.

Because list of numbers will be partitioned and z_i and z_j will be in two different parts.

Therefore z_i and z_j are compared if the first element chosen as pivot from Z_{ij} is either z_i or z_j.

$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}] \\
&= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}] \\
&\quad + \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]
\end{aligned}
$$

$$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$$

$$= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}]$$

$$+ \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]$$

$$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1) .$$

$$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$$

$$= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}]$$

$$+ \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]$$

$$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1) \,.$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1) \,.$$

$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$

$= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}]$

$+ \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]$

$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1)$.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/(k+1)$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} 2/k$$

$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$

$= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}]$

$+ \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]$

$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1)$.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/(k+1)$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} 2/k = \sum_{i=1}^{n-1} O(\log n) = O(n \log n).$$

$\Pr\{z_i \text{ is compared to } z_j\} = \Pr[z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}]$

$= \Pr[z_j \text{ is first pivot chosen from } Z_{ij}]$

$+ \Pr[z_i \text{ is first pivot chosen from } Z_{ij}]$

$= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1)$.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/(k+1)$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} 2/k = \sum_{i=1}^{n-1} O(\log n) = O(n \log n).$$

Expected running time of Randomized-QuickSort is O(n log n).

An application of Markov's inequality

Let T be the running time of Randomized Quick sort.

We just proved $E[T] \leq c\, n \log n$, for some constant c.

Hence, $\Pr[\, T > 100\, c\, n \log n] < ?$

An application of Markov's inequality

Let T be the running time of Randomized Quick sort.

We just proved $E[T] \le c\, n \log n$, for some constant c.

Hence, $\Pr[\, T > 100\, c\, n \log n] < 1/100$

Markov's inequality useful to translate bounds on the expectation in bounds of the form: “It is unlikely the algorithm will take too long.”

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	Search time	Extra space
$f(x) = x$ $t = 2^n$, open addressing	?	?

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	O(1)	2^u
Any deterministic function	?	?

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	? expected	?

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	n/t expected $\forall\, x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$	$2^u \log(t)$
Now what? We ``derandomize'' random functions		

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	n/t expected $\forall\, x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$	$2^u \log(t)$
Pseudorandom function A.k.a. hash function	n/t expected Idea: Just need $\forall\, x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$	$O(u)$

Construction of hash function: Let t be prime. Write u-bit elements in base t.

$x = x_1\ x_2 \ldots x_m$ for $m = u/\log(t)$

Hash function specified by an element $a = a_1\ a_2 \ldots a_m$

$f_a(x) := \sum_{i \leq m} a_i\ x_i$ modulo

Claim: $\forall\ x \neq x'$, $\Pr_a [f_a(x) = f_a(x')] = 1/t$

Different constructions of hash function: u-bit keys to r-bit hashes

Classic solution: pick a prime $p>2^u$, and a random a in [p], and

$h_a(x) := ((ax) \bmod p) \bmod 2^r$

Problem: mod p is slow, even with Mersenne primes ($p=2^i-1$)

Alternative: let b be a random odd u-bit number and

$h_b(x) = ((bx) \bmod 2^u) \text{ div } 2^{u-r}$

= bits from u-r to u of integer product bx

Faster in practice. In C, think x unsigned integer of u=32 bits

$h_b(x) = (b*x) >> (u-r)$

Static search:

Given n elements, want a hash function that gives no collisions.

Probabilistic method: Just hash to $[t] = n^2$ elements

$\Pr[\ \exists\ x \neq y : \text{hash}(x) = \text{hash}(y)\ ]$

$\leq n^2/2 \Pr[\text{hash}(0) = \text{hash}(1)]$ (union bound)

$\leq n^2 / (2\,t) = 1/2$

$\Rightarrow \exists$ hash : $\forall\ x \neq y$, $\text{hash}(x) \neq \text{hash}(y)$ (probabilistic method)

Can you have no collisions with $[t] = O(n)$?

Static search:

Given n elements, want a hash function that gives no collisions.

Two-level hashing:

- First hash to $t = O(n)$ elements,
- then hash again using the previous method. That is, if i-th cell in first level has c_i elements, hash to c_i^2 cells at the second level.

Expected total size $\leq E[\sum_{i \leq t} c_i^2]$

Note $\sum_{i \leq t} c_i^2 =$

Θ(expected number of colliding pairs in first level) =

O(???)

Static search:

Given n elements, want a hash function that gives no collisions.

Two-level hashing:

- First hash to $t = O(n)$ elements,
- then hash again using the previous method. That is, if i-th cell in first level has c_i elements, hash to c_i^2 cells at the second level.

Expected total size $\leq E[\ \sum_{i \leq t} c_i^2\ ]$

Note $\sum_{i \leq t} c_i^2 =$

Θ(expected number of colliding pairs in first level) =

$O(n^2 / t) =$

$O(n)$