Algorithm design techniques: randomization

1. Introduction to randomization
2. Quicksort
3. Linear search

1. Introduction to randomization

Categories of design techniques

Category I: techniques to handle entire computation problem
- decrease and conquer (incremental)
- divide and conquer
- dynamic programming
- tranform and conquer

Category II: techniques to be used in conjunction with other techniques
- randomization

Deterministic algorithm: no randomization

Randomized algorithm: randomization intentionally added

Even when the input data are fixed, a randomized algorithm will not behave the same way each time it is run.

Q: Why use a randomized algorithm?

A 1: It improves the algorithm's behaviour.

A 2: A deterministic algorithm may have a worst case that significantly degrades the algorithm behaviour. A randomized algorithm may avoid the worst case.

Successful applications of randomized algorithms:

- searching and sorting: quicksort and hash tables

- optimization: simulated annealing, genetic algorithms

Randomization: pluses and minuses

+ shown to work in practice

+ worst case should be no more probable than any other case

- adding randomization also takes time

- algorithm becomes more complicated

- algorithm analysis requires probability theory

- need a random number generator

2. Quicksort

Pseudocode for standard QUICKSORT

```
1   QUICKSORT(A, L, R)
2   input number array A, L is index of leftmost element to be
3   handled, R is index of rightmost element to be handled
4   /* We sort the subarray A[L..R] from smallest to largest using the
5   quicksort algorithm. */
6   if L < R then
7     k = PARTITION(A, L, R)
8     QUICKSORT(A, L, k-1)
9     QUICKSORT(A, k+1, R)
10  end
```

```
1   PARTITION(A, L, R)
2   input number array A, L is index of leftmost element to be
3   handled, R is index of rightmost element to be handled
4   /* We partition subarray A[L..R] using A[R] as the pivot, which we
5   denote as α. Let the final location of pivot α be k, L ≤ k ≤ R.
6   After execution elements A[L..(k−1)] are less than or equal to α
7   and elements A[(k+1)..R] are greater than α. The procedure
8   returns location k. */
9   α = A[R], cut = L − 1
10  for j = L to R − 1
11    if A[j] ≤ α then
12      cut = cut + 1, swap elements A[cut] and A[j]
13    end
14  end
15  k = cut + 1, swap elements A[k] and A[R]
16  return k
```

PARTITION(*A*, 1, *n*)

starting array:

$A[1]$	$A[2]$	$A[3]$	$\ldots$	$\alpha = A[n]$

after partitioning when pivot a is at its correct location:

$A[1]$	$A[2]$	$\ldots$	$A[k-1]$	$A[k] = \alpha$	$A[k+1]$	$\ldots$	$A[n]$

Worst case input

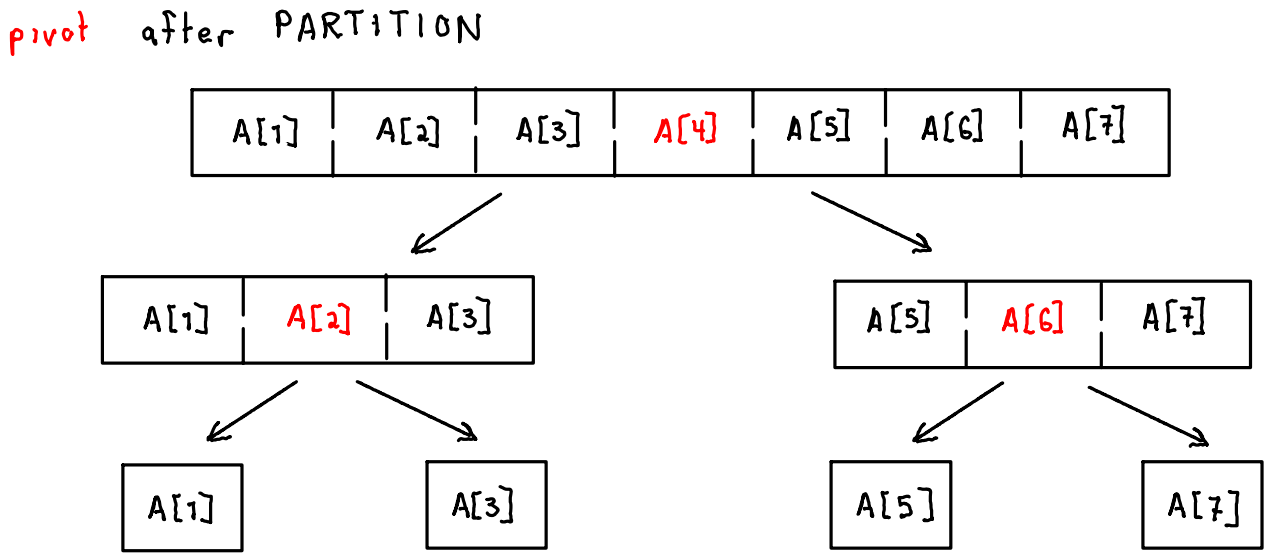
Q: What is the most number of times we will need to call QUICKSORT for array *A*[1..*n*]?

Consider runnig QUICKSORT when *n* = 7 for two situations:

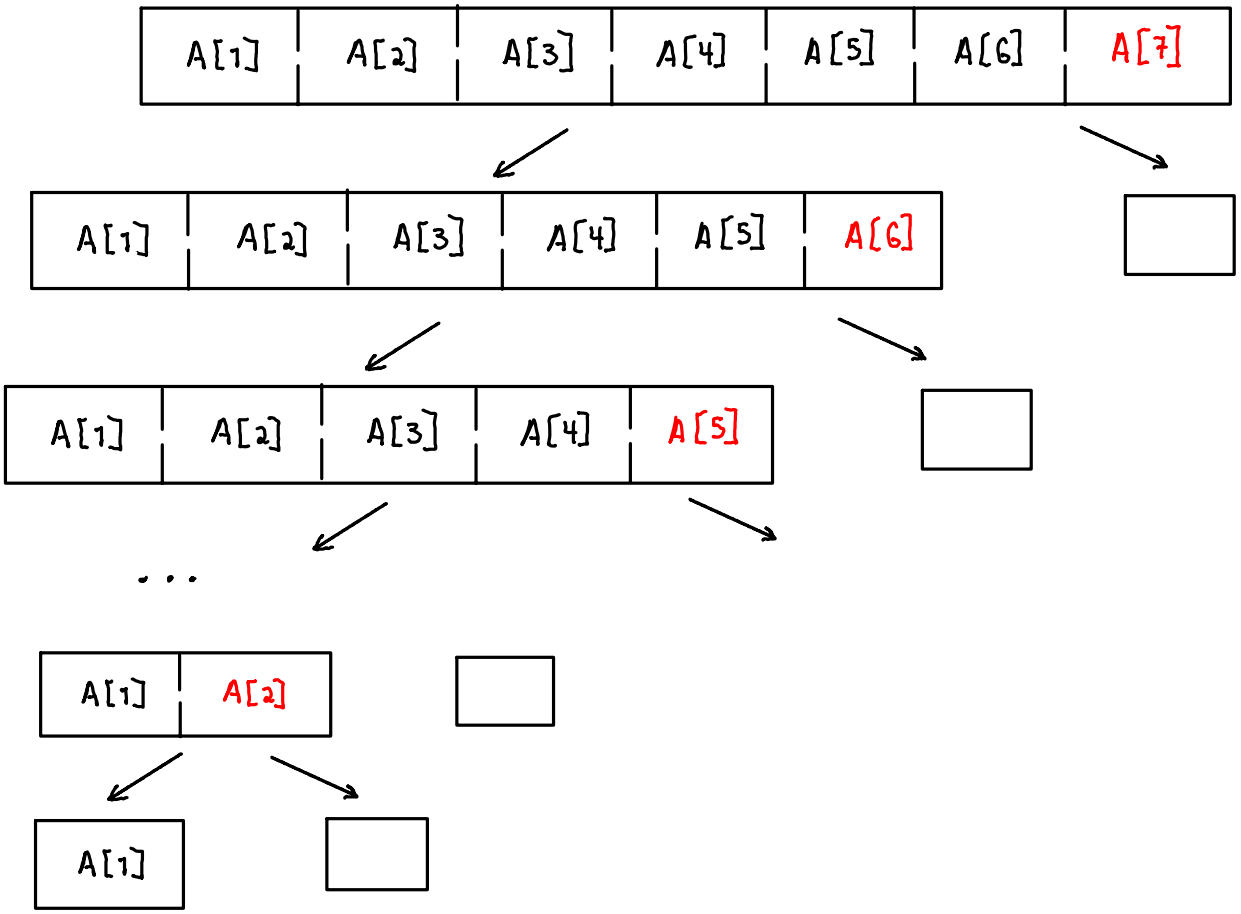
(i) PARTITION always results in two subarrays that are as close as possible to being same size

(ii) PARTITION always results in two subarrays that are as far as possible from being same size

(i) PARTITION always results in two subarrays that are as close as possible to being same size



(ii) PARTITION always results in two subarrays that are as far as possible from being same size



Q: What is the most number of times we will need to call QUICKSORT for input array $A[1..n]$?

A: $2n-1$

Q: Under what conditions do we need to call QUICKSORT $2n-1$ times for input array $A[1..n]$?

A: When $k = L$ or $k = R$ for <u>every</u> call to k = PARTITION(A, L, R).

Let $m(L,R)$ be the number of iterations in PARTITION(A, L, R): $m(L,R) = R - L$

Consider QUICKSORT($A, 1, n$) when $k = R$ for <u>every</u> call to k = PARTITION(A, L, R).

recursion level	calls	k	$m(L,R)$
1	k = PARTITION(A, 1, n), QUICKSORT(A, 1, n-1), QUICKSORT(A, n+1, n)	n	n-1
2	k = PARTITION(A, 1, n-1), QUICKSORT(A, 1, n-2), QUICKSORT(A, n, n-1)	n-1	n-2
3	k = PARTITION(A, 1, n-2), QUICKSORT(A, 1, n-3), QUICKSORT(A, n-1, n-2)	n-2	n-3
. . .			
n-1	k = PARTITION(A, 1, 2), QUICKSORT(A, 1, 1), QUICKSORT(A, 3, 2)	2	1

$f(n) = \sum$

$$f(n) = 1 + 2 + \ldots + n-2 + n-1 = \frac{n^2 - n}{2}$$

Upper bound on running time of QUICKSORT: $O(n^2)$

Standard QUICKSORT's running time is $O(n^2)$ for following input arrays:

[1, 2, 3, ... n] [n, n-1, n-2, ... 1] [n/2, ... 2, n-1, 1, n] (n even)

Randomization

Assume procedure RANDOM(a, b) exists.

```
1   RANDOM(a, b)
2   input integers a and b
3   output a randomly generated integer x such that a ≤ x ≤ b
```

Assume RANDOM(a, b) generates each value between a and b with equal probability.

I Strategy for avoiding worst case: random choice of pivot

Use procedure RANDOMIZED-PARTITION and RANDOMIZED-QUICKSORT.

```
1   RANDOMIZED-PARTITION(A, L, R)
2   input number array A, L is index of leftmost element to be
3   handled, R is index of rightmost element to be handled
4   m =RANDOM(L, R)
5   swap elements A[R] and A[m]
6   α = A[R], cut = L − 1
7   for j = L to R − 1
8     if A[j] ≤ α then
9       cut = cut + 1, swap elements A[cut] and A[j]
10    end
11  end
12  k = cut + 1, swap elements A[k] and A[R]
13  return k
```

```
1  RANDOMIZED-QUICKSORT(A, L, R)
2  input number array A, L is index of leftmost element to be
3  handled, R is index of rightmost element to be handled
4  if L < R then
5    k = RANDOMIZED-PARTITION(A, L, R)
6    RANDOMIZED-QUICKSORT(A, L, k-1)
7    RANDOMIZED-QUICKSORT(A, k+1, R)
8  end
```

II Strategy for avoiding worst case: random permutation of starting array

Random permutation also called shuffling.

Run SHUFFLE(*A*, 1, *n*) before calling standard QUICKSORT(*A*, 1, *n*).

```
1  SHUFFLE(A, a, b)
2  input number array A[1..n]
3  ouput a permuted version of A, where elements A[a..b] are shuffled
4  for i from a to b
5     j =RANDOM(i, b)
6     swap elements A[i] and A[j]
7  end
```

3. Linear search

Pseudocode for searching number array *L* for *x*.

```
1     SEARCH(L,x)
2     input: int array L[0..(n-1)], int x output: int i
3     /* We search array L for integer x.  If x occurs in L, we return
4     the first index where it occurs, otherwise we return -1. */
5     for i from 0 to L.length-1
6       if L[i] == x then
7         return i
8       end
9     end
10    return -1
```

Running times:

- lower bound: **for**-loop is executed only once: $\Omega(1)$

- upper bound: **for**-loop is executed n times: $O(n)$

Someone tries to 'improve' SEARCH with randomization.

```
1     RANDOMIZED-SEARCH(L,x)
2     input: int array L[0..(n-1)], int x output: int i
3     /* We search array L for integer x.  If x occurs in L, we return
4     the first index where it occurs, otherwise we return -1. */
5     SHUFFLE(L, 0, L.length-1)
6     for i from 0 to L.length-1
7       if L[i] == x then
8         return i
9       end
10    end
11    return -1
```

For array $L[0 .. n-1]$ running time of SHUFFLE is $\Theta(n)$.

So running time of RANDOMIZED-SEARCH is $\Omega(n)$.

Hence: running time of RANDOMIZED-SEARCH is worse than running time of SEARCH.

Conclusion: use randomization wisely