1) Dulika Gamage 101263208

2) As we saw in class, a good pivot is defined like this for 3/4 phases:

bad p	good p	bad p
$m/4$	$m/2$	$m/4$

m = length of sequence

$(3/4)^{i+1} \cdot n < m \leq (3/4)^i \cdot n$

$Pr(p \text{ is good}) = 1/2$

I will use this definition of a good pivot for 1/2 phases.

bad p	good p	bad p
$m/4$	$m/2$	$m/4$

m = length of sequence

$(1/2)^{i+1} \cdot n < m \leq (1/2)^i \cdot n$

$Pr(p \text{ is good}) = 1/2$

if pivot p is good:

Next call to RSelect: length of sequence $\leq m - m/4$

$= 3/4 \cdot m \leq (1/2)^{i+1} \cdot n$

↳ but we can't say this leaves phase i because now it's 1/2, to guarantee we leave phase i we would need to do at least 3 good pivots $(3/4)^3 \approx 0.422$

↳ $(3/4)^2 \approx 0.563 > 1/2$ (we can't do this)

$= (3/4)^3 m \leq (1/2)^{i+1} \cdot n$

so we leave phase i to phase $\geq i+1$

x_i = # calls to leave phase i.

$E(x_i) \leq 3\left(1/(1/2)\right) = 6$ → we need 3 good pivots to leave and the Pr(good pivot) = 1/2

time $T = \sum_{i \geq 0} x_i \cdot c\left(\frac{1}{2}\right)^i \cdot n$

$E(T) = \sum_{i \geq 0} E(x_i) \cdot c\,(1/2)^i \cdot n$

$\leq 6cn \sum_{i=0}^{\infty} (1/2)^i$

$= 6cn \cdot \frac{1}{1 - 1/2}$ → $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

$= 6cn \cdot 2$

$= 12cn$

$\therefore = \boxed{O(n)}$

3.1)

The comparisons:

1. A[1] and A[2] (*)
2. A[i] < smallest (**)
3. A[i] < secondsmallest (***) – may do this one

always does these 2

$x_i \begin{cases} 1 & \text{if } (***) \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$

& since we know (*) always occurs & (**) always occurs $(i \geq 3)$ we include these as +1.

$$X = 1 + \sum_{i=3}^{n} \left[1 + P(x_i = 1) \right]$$

$$E(X) = 1 + \sum_{i=3}^{n} \left[1 + \frac{i-1}{i} \right]$$

since (***) executes if A[i] > smallest

↳ probability that this happens is $\frac{i-1}{i}$

$$= 1 + \sum_{i=3}^{n} \left[1 + \frac{i}{i} - \frac{1}{i} \right]$$

$$= 1 + \sum_{i=3}^{n} \left[2 - \frac{1}{i} \right]$$

$$= 1 + \sum_{i=3}^{n} 2 - \sum_{i=3}^{n} \frac{1}{i}$$

$$= 1 + 2 \sum_{i=3}^{n} 1 - \sum_{i=3}^{n} \frac{1}{i}$$

⇒ we observe that this is harmonic sum & we know $H_n = \Theta(\log n)$

↳ (textbook)

$$= 1 + 2(n-2) - (H_n - H_2)$$

$$= 1 + 2(n-2) - (H_n - 1 + \tfrac{1}{2})$$

$$= 1 + 2(n-2) - H_n - \tfrac{1}{2}$$

$$= 1 + 2n - 4 - H_n - \tfrac{1}{2}$$

$$= 2n - \tfrac{7}{2} - H_n$$

$$= 2n - \Theta(\log n)$$

$\therefore E(X) = 2n - \Theta(\log n)$

3.2) Similar to 3.1, we know (*) & (**) always occur. We can denote these as +1 again (since guaranteed).

We also know, similarly to 3.1, that (***) happening will have the same probability as (***) happening in 3.1.

This is because as long as A[i] > smallest the comparison at (***) will happen.

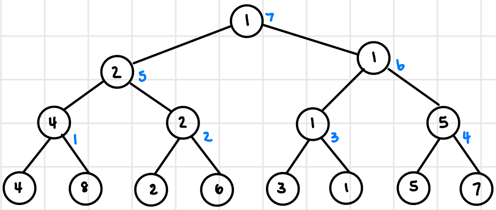
↳ probability of $\frac{i-1}{i}$

∴ the equation set up is the exact same as 3.1 $X = 1 + \sum_{i=3}^{n}\left[1 + \frac{i-1}{i}\right]$

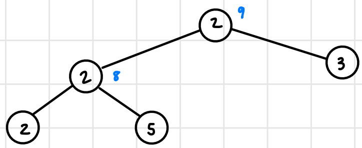
and the expected amount of comparisons is $E(Y) = 2n - \theta(\log n)$

3.3) To visualize, we can draw the tournament as a tree: #comparisons

ex. $n = 2^3 = 8$ [4, 8, 2, 6, 3, 1, 5, 7]



to find second smallest :



$8 + \log 8 - 2 = \boxed{9}$

Algorithm:

- for each round in the tournament we compare a pair of players (their APT ranking) and whoever is better (smaller APT number) advances & the other is eliminated.

↳ we repeat this in each round until there's one player left. This player is the winner & has the smallest APT rank in the array.

- To get the winner (the smallest num) we do $n-1$ comparisons.

↳ in the first round we do $n/2$ comparisons, in the second we do $n/4$, then $n/8$ ... etc until we reach $n/n = 1$

↳ $n/2 + n/4 + \ldots + 1 = n-1 \rightarrow$ ex $8/2 + 8/4 + 8/8 = 7 = 8-1$

- After we find the smallest, we need to find the second smallest.

- we know that the second smallest would have had to lose to the smallest at some point.

↳ because no one else is smaller, so no one else can eliminate the 2^{nd} smallest

- we search all of the players who the smallest ranking played.

- we know this is equal to the height of the tree since smallest started on the bottom and ended on the top.

- we also know the height of a balanced binary tree is equal to $\log n$ (as seen in class) & we know the tree is always balanced because $n = 2^k$

- if we say $\log n = m$ then m is the number of players who played the smallest in the tourn. & by similar logic as calculating the smallest, we can calc. the second smallest by doing $m-1$ comparisons which is $\log n - 1$.

- if we add the comparisons to find the smallest: $n-1$ + ⇒ we get: $n-1 + \log n - 1 = \boxed{n + \log n - 2}$

& the comparisons to find the second smallest: $\log n - 1$

3.4) Visualization #comparisons

ex. $n = 8$ $[4, 8, 2, 6, 3, 1, 5, 7]$

1 $(4, 8) \rightarrow$ smaller: 4, larger: 8

2 $(2, 6) \rightarrow$ smaller: 2, larger: 6

3 $(3, 1) \rightarrow$ smaller: 1, larger: 3

4 $(5, 7) \rightarrow$ smaller: 5, larger: 7

smaller set: 4, 2, 1, 5 4 = global min

larger set: 8, 6, 3, 7 8 = global max

5 $(4, 2) \rightarrow 2$ 6 $(2, 1) \rightarrow 1$ 7 $(1, 5) \rightarrow 1$

8 $(8, 6) \rightarrow 8$ 9 $(8, 3) \rightarrow 8$ 10 $(8, 7) \rightarrow 8$

$$\frac{8(3)}{2} - 2 = \boxed{10}$$

Algorithm:

- we take the array, A, and we compare each adjacent pair. From this we get the smaller & larger nums from each pair
  - ↳ we know all nums will have a pair because n is even
  - ↳ by doing this, we do $n/2$ comparisons
- after we seperate into the small & large set, we take the first number in each set and we compare against the other nums to find the global max & global min.
  - ↳ by doing this we do $n/2 - 1$ comparisons for each set (min & max) (-1 because we don't compare the current global min/max with itself.
- if we add the comparisons together:

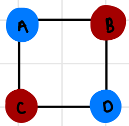
initial split comparisons: $n/2$ +

smaller set comparisons: $n/2 - 1$ +

larger set comparisons: $n/2 - 1$

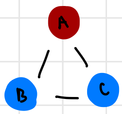
$$\Rightarrow n/2 + n/2 - 1 + n/2 - 1 = \boxed{\frac{3n}{2} - 2}$$

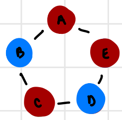
4) The smallest integer k would be 3. This is because this graph, G, has exactly 1 cycle with an odd number of vertices. Lets say, for example, there is a cycle with an even amount of vertices:



In this case, where the vertices are even, we can have minimum 2 different colours without violating the rule.

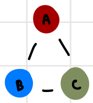
But try with an odd amount & 2 colours:

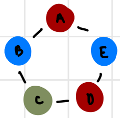




No matter how you try with 2 colours, it will always violate the rule.

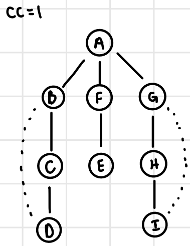
Try with 3 colours:

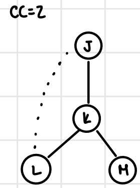




In this case, it doesn't violate the rule. ∴ The minimum is 3

5) 5.1) A, B, C, D, E, F, G, H, I, J, K, L, M



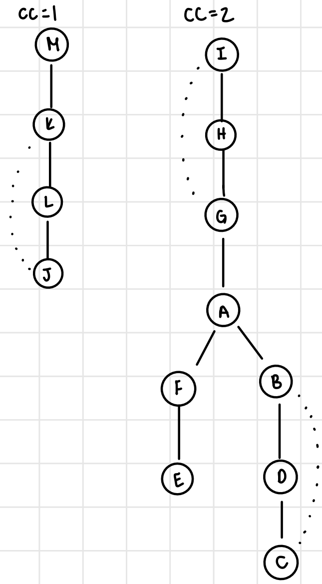


Start at A:

- CC=1
  - Explore(A) — visited(A) = true, ccnumber(A) = 1
  - ↳ Explore(B) — visited(B) = true, ccnumber(B) = 1
  - ↳ Explore(C) — visited(C) = true, ccnumber(C) = 1
  - ↳ Explore(D) — visited(D) = true, ccnumber(D) = 1
  - (back track to A)
  - ↳ Explore(F) — visited(F) = true, ccnumber(F) = 1
  - ↳ Explore(E) — visited(E) = true, ccnumber(E) = 1
  - (back track to A)
  - ↳ Explore(G) — visited(G) = true, ccnumber(G) = 1
  - ↳ Explore(H) — visited(H) = true, ccnumber(H) = 1
  - ↳ Explore(I) — visited(I) = true, ccnumber(I) = 1
  - (back track to A)
- go to next vertex thats false: J, inc. cc = 2
- Explore(J) - visited(J) = true, ccnumber(J) = 2
  - ↳ Explore(K) - visited(K) = true, ccnumber(K) = 2
  - ↳ Explore(L) - visited(L) = true, ccnumber(L) = 2
  - (back track to K)
  - ↳ Explore(M) - visited(M) = true, ccnumber(M) = 2
  - (back track to J)
- all vertices have been visited

5) 5.2)

M, L, K, J, I, H, G, F, E, D, C, B, A



Start at M:

- cc = 1

- Explore(M) - visited(M) = true, ccnumber(M) = 1

↳ Explore(K) - visited(K) = true, ccnumber(K) = 1

↳ Explore(L) - visited(L) = true, ccnumber(L) = 1

↳ Explore(J) - visited(J) = true, ccnumber(J) = 1

(back track to J)

- go next vertex thats false: I, inc. cc = 2

- Explore(I) - visited(I) = true, ccnumber(I) = 2

↳ Explore(H) - visited(H) = true, ccnumber(H) = 2

↳ Explore(G) - visited(G) = true, ccnumber(G) = 2

↳ Explore(A) - visited(A) = true, ccnumber(A) = 2

↳ Explore(F) - visited(F) = true, ccnumber(F) = 2

↳ Explore(E) - visited(E) = true, ccnumber(E) = 2

(back track to A)

↳ Explore(B) - visited(B) = true, ccnumber(B) = 2

↳ Explore(D) - visited(D) = true, ccnumber(D) = 2

↳ Explore(C) - visited(C) = true, ccnumber(C) = 2

(back track to I)

- all vertices have been visited