CS 217 – Algorithm Design and Analysis

Shanghai Jiaotong University, Fall 2019

Yujie Lu Jiabao Ji Tong Chen

Handed out on Monday, 2020-03-23
First submission and questions due on Friday, 2020-03-27
You will receive feedback from the TA.
Final submission due on Friday, 2020-04-04

2 Sorting Algorithms

Exercise 1 Given an array A of n items (numbers), we can find the maximum with $n-1$ comparisons (this is simple). Show that this is optimal: that is, any algorithm that does $n-2$ or fewer comparisons will fail to find the maximum on some inputs.

Proof: First, consider taking $n-2$ comparisons. If we list the $n-2$ comparisons like below, and we get the maximal M from them.

$$a_{i_1} \leq a_{j_1}, a_{i_2} \leq a_{j_2}, \ldots a_{i_{n-2}} \leq a_{j_{n-2}}$$

For smaller array $a_{i_1}, a_{i_2}, ...a_{i_{n-2}}$, there are at most $n-2$ different element from the original array. Choose 2 in the others as a_x, a_y, at least one of them is not maximal, suppose a_x is not maximal, then if we change a_x to $a_x = M+1$, the new array still suffices the $n-2$ comparisons, but the maximal is not M anymore.

Next, if we take $m < n-2$ comparisons, the proof is similar to the case above. The smaller array has at most m different elements, and at least $n-m$ element are not in the smaller part, we can still choose one from them not maximal, and change it to maximal + 1, leading to the parodox. $\square$

Exercise 2 Let A be an array of size n, where n is even. Describe how to find both the minimum and the maximum with at most $\frac{3}{2}n - 2$ comparisons. Make sure your solution is *simple*, in describe it in a clear and succinct way!

Proof:

```
def ComputeMaxMin(A):
    # A is an array with n = 2m elements
    n = len(A)
    m = n // 2
    MinCandidate = []
    MaxCandidate = []
    for i in range(m):
        x = A[2 * i]
        y = A[2 * i + 1]
        if (x < y):
            MinCandidate.append(x)
            MaxCandidate.append(y)
        else:
            MinCandidate.append(y)
            MaxCandidate.append(x)

    Min = sys.maxint
    Max = -sys.maxint
    for i in range(m):
        if MinCandidate[i] < Min:
            Min = MinCandidate[i]
        if MaxCandidate[i] > Max:
            Max = MaxCandidate[i]
    return Min, Max
```

Based on the algorithm above, we can get the minimal element and maximal element in the n items with exactly $\frac{3}{2}n - 2$ comparisons.

Since the array has $n = 2m$ elements, we can first compare any consecutive two elements in the array for m times to divide the original array into two subarray as $MaxCandidate$, $MinCandidate$. We know for sure that maximal element is in $MaxCandidate$ and minimal element is in $MinCandidate$.

Using the result in $Exer.1$, we need $m - 1$ times of comparisons to get minimal element form $MinCandidate$ and another $m - 1$ comparisons for maximal. In all, we do $m + m - 1 + m - 1 = 3m - 2 = \frac{3}{2}n - 2$ comparisons. $\square$

Exercise 3 Given an array A of size $n = 2^k$, find the second largest element element with at most $n + \log_2(n)$ comparisons. Again, your solution should be *simple*, and you should explain it in a clear and succinct way!

Proof:

Exer.3:

```
class Node:
    def __init__(self, weight, father, biggerSon, smallerSon
                                    ):
        self.weight = weight
        self.father = father
        self.biggerSon = biggerSon
        self.smallerSon = smallerSon

def ComputeSecondMax(A):
        # A should be an array with 2^k elements
        comparisons = 0
        n = len(A)
        k = math.floor(math.log(n, 2))
        MaxCandidate = []
        for x in A:
            MaxCandidate.append(Node(x, None, None, None))
        for _ in range(k):
            tmpMaxCandidate = []
            for j in range(len(MaxCandidate) // 2):
                x = MaxCandidate[2 * j]
                y = MaxCandidate[2 * j + 1]
                if (x.weight < y.weight):
                    comparisons += 1
                    newNode = Node(y.weight, None, y, x)
                    tmpMaxCandidate.append(newNode)
                    x.father = newNode
                    y.father = newNode
                else:
                    comparisons += 1
                    newNode = Node(x.weight, None, x, y)
                    tmpMaxCandidate.append(newNode)
                    x.father = newNode
                    y.father = newNode
            MaxCandidate = tmpMaxCandidate

        res = 0
        curNode = MaxCandidate[0]
        for _ in range(k):
```

```
            if curNode.smallerSon.weight > res :
                res = curNode.smallerSon.weight
                comparisons += 1
            comparisons += 1
            curNode = curNode.biggerSon
        return comparisons, res
```

The algorithm is similar to the one in *Exer*.2, we can divide the original array for k times, each time, we get the bigger element from consecutive 2 elements in the last iteration. The array size is $2^k, 2^{k-1} ... 2^1$ And it takes $2^i - 1$ comparisons for 2^i size array. But to get the second max element we still need to do something else. After k iterations of comparing consecutive

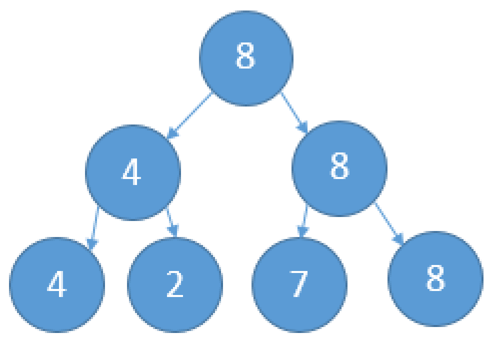


Figure 1: Comparing Tree in Exer.3

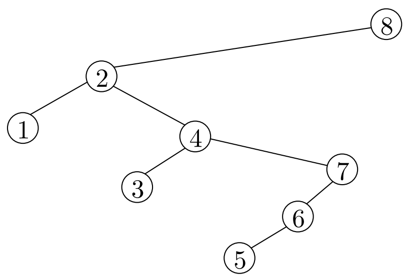
elements, we constructed the comparing tree above. Obviously, the maximal element floats up from bottom to top, and the candidates for second maximal element are those who once compared to the maximal element in the tree, 4 and 7 in above specific tree, for example.

So we need to compare another $log_2(n) - 1$ times to get the maximal element in the candidates since maximal element needs to be compared for $log_2(n)$ times to float to top.

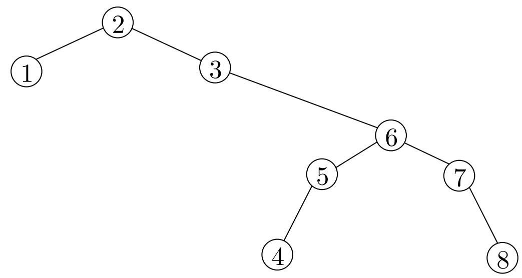
In all we do $1 + 2 + 2^2 + ... + 2^{k-1} + log_2(n) = n + log_2(n)$ comparisons.

□

Recall the quicksort tree defined in the lecture.



quicksort tree of $[8, 2, 4, 1, 7, 6, 5, 3]$



quicksort tree of $[2, 3, 6, 1, 7, 5, 8, 4]$

Proof:

```
def ComputeSecondMax(A):
    # A should be an array with 2^k elements
    n = len(A)
    k = math.floor(math.log(n, 2))
    MaxCandidate = A
    for _ in range(k):
        tmpMaxCandidate = []
        for j in range(len(MaxCandidate) // 2):
            x = MaxCandidate[2 * j]
            y = MaxCandidate[2 * j + 1]
            if (x < y):
                tmpMaxCandidate.append(y)
            else:
                tmpMaxCandidate.append(x)
        MaxCandidate = tmpMaxCandidate
    return MaxCandidate[0]
```

The algorithm is similar to the one in $Exer.2$, we can divide the original array for k times, each time, we get the bigger element from consecutive 2 elements in the last iteration. The array size is $2^k, 2^{k-1}...2^1$ And it takes $2^i - 1$ comparisons for 2^i size array. In all we do $1 + 2 + 2^2 + ... + 2^{k-1} = 2^k = n$ comparisons.

FixMe: $\square$

We denote a specific list (ordering) by π and the tree by $T(\pi)$. $A_{i,j}$ is an indicator variable which is 1 if i is an ancestor of j in the tree $T(\pi)$, and 0 otherwise. In the lecture, we have derived:

$$\mathbb{E}[A_{i,j}] = \frac{1}{|i-j|+1}$$

$$\text{total number of comparisons } = \sum_{i \neq j} A_{i,j} \ .$$

Exercise 4 Determine the expected number of comparisons made by quicksort. Your final formula must be *closed*, meaning it must not contain $\mathbb{E}$, $\prod$, or $\sum$. It may, however, contain $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, the n^{th} Harmonic number. **Remark.** This gets a bit tricky, and you will need some summation wizardry towards the end.

Proof: From the sum equation in the class, we can derive the result as follows.

$$
\begin{aligned}
\mathbb{E} &= \sum_{i \neq j} \frac{1}{|i-j|+1} \\
&= 2 \sum_{1 \leq i < j \leq n} \frac{1}{j-i+1} \\
&= 2(\frac{1}{1+1}(n-1) + \frac{1}{2+1}(n-2) + ... + \frac{1}{n-1+1}(n-(n-1))) \\
&= 2(\frac{1}{2}n + \frac{n}{3} + ... + \frac{n}{n} - (\frac{1}{2} + \frac{2}{3} + ... + \frac{n-1}{n})) \\
&= 2(n(\frac{1}{2} + ... + \frac{1}{n}) - (1 - \frac{1}{2}) - (1 - \frac{1}{3}) - ... - (1 - \frac{1}{n})) \\
&= 2(n(H_n - 1) - (n-1) + (H_n - 1)) \\
&= 2((n+1)H_n - 2n) \\
&= 2nH_n + 2H_n - 4n
\end{aligned}
$$

$\square$

2.1 Quickselect

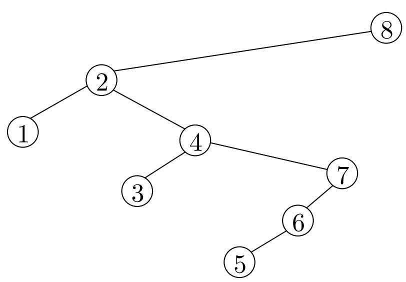
Remember the recursive algorithm QUICKSELECT from the lecture. I write it below in pseudocode. In analogy to quicksort we define QuickSelect deterministically and assume that the input array is random, or has been randomly shuffled before QuickSelect is called. We assume that A consists of distinct elements and $1 \leq k \leq |A|$.

Let C be the number of comparison made by QUICKSELECT. In the lecture we proved that $\mathbb{E}[C] \leq O(n)$ when we run it on a random input.

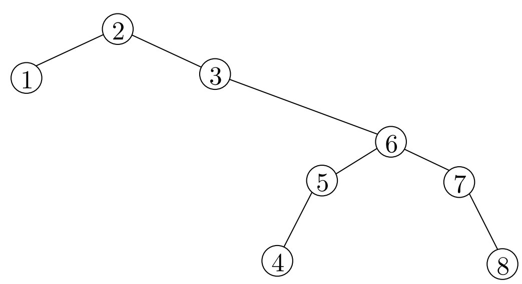
Exercise 5 Explain how QUICKSELECT can be viewed as a "partial execution" of quicksort with the random pivot selection rule. Draw an example quicksort tree and show which part of this tree is visited by QuickSelect.

Algorithm 1 Select the k^{th} smallest element from a list A

procedure QUICKSELECT(X, k)
    if $|X| = 1$ **then**
        return $X[1]$
    else:
        $p := X[1]$
        $Y := [x \in X \mid x < p]$
        $Z := [x \in X \mid x > p]$
        if $|B| = k - 1$ **then**
            return p
        else if $|Y| \geq k$ **then**
            return QUICKSELECT(Y, k)
        else
            Return QUICKSELECT$(Z, k - |Y| - 1)$
        end if
    end if
end procedure



quicksort tree of $[8, 2, 4, 1, 7, 6, 5, 3]$



quicksort tree of $[2, 3, 6, 1, 7, 5, 8, 4]$

Proof: For example, if quicksort-tree is left-side tree and k=5, then the visit order of quickselect algorithm is [8, 2, 4, 7, 6, 5]. This is a chain from the root to one of the nodes. Everytime, if the pivot fits the rank we want to find, the result is it. Otherwise, we will recusive the interval fitted the rank what represents the left or right child in quicksort-tree. So, now we get the conclusion: quickselect can be viewed as a "a partial execution" of quicksort with random pivot selection rule. □

Let $B_{i,j,k}$ be an indicator variable which is 1 if i is a common ancestor of j and k in the quicksort tree. That is, if both j and k appear in the subtree of $T(\pi)$ rooted at i.

Exercise 6 What is $\mathbb{E}[B_{i,j,k}]$? Give a succinct formula for this.

Proof: Similarly, we know that only when i ranks the first in $[i,j,k]$, can i be the ancestors of both j,k. The notation $[i,j,k]$ means

$$[i,j,k] = \{\min(i,j,k), \ldots, \max(i,j,k) - 1, \max(i,j,k)\}.$$

Hence $\mathbb{E}[B_{i,j,k}] = \frac{1}{(\max(i,j,k) - \min(i,j,k)) + 1}$ □

Exercise 7 Let $C(\pi,k)$ be the number of comparisons made by QuickSelect when given π as input. Design a formula for $C(\pi,k)$ with the help of the indicator variables $A_{i,j}$ and $B_{i,j,k}$ (analogous to the formula $\sum_{i \neq j} A_{i,j}$ for the number of comparisons made by quicksort).

Proof: Observe that only when i is j's ancestor, and k is also j's sibling, woll the algorithm compare i,j. Thus it's easy to find that

$$C(\pi,k) = \sum_{i \neq j,k} B_{i,j,k}.$$

□

Exercise 8 Suppose we use QuickSelect to find the minimum of the array. On expectation, how many comparisons will it make? Give an answer that is exact up to additive terms of order $o(n)$. You can use the fact that $H_k := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln(n) + o(1)$.

Proof: Since $k = 1$, we have

$$\begin{aligned}
\mathbb{E}\left[C\left(\pi,k\right)\right] &= \sum_{i=2}^{n}\sum_{j\neq i}\frac{1}{\max\left(i,j\right)} \\
&= \sum_{i=2}^{n}\frac{i-1}{i} + \sum_{i=2}^{n}\sum_{j>i}\frac{1}{j} \\
&= n - H_n + \left(n-1\right)H_n - \sum_{i=2}^{n}H_i \\
&= \left(n-2\right)H_n + n + 1 - \sum_{i=1}^{n}H_i \\
&= \left(n-2\right)H_n + n + 1 - \left(\left(n+1\right)H_n - n\right) \\
&= 2n + 1 - 3H_n = O\left(n\right) - O\left(\log\left(n\right)\right) \\
&= O\left(n\right).
\end{aligned}$$

$\square$

Exercise 9 Derive a formula for $\mathbb{E}_\pi[C(\pi,k)]$, up to additive terms of order $o(n)$. You might want to introduce $\kappa := k/n$.

Proof: We need to compute

$$\begin{aligned}
\mathbb{E}\left[C\left(\pi,k\right)\right] &= \sum_{i=1,i\neq k}^{n}\sum_{j\neq i}\frac{1}{\max\left(i,j,k\right)-\min\left(i,j,k\right)+1} \\
&= \sum_{i<k}\sum_{j=1}^{n}B_{i,j,k} + \sum_{i>k}\sum_{j=1}^{n}B_{i,j,k} - \sum_{i\neq k,j=i}B_{i,j,k} \\
&= S_1 + S_2 - S_3.
\end{aligned}$$

We'll compute the three part one by one

$$S_3 = \sum_{i\neq k}\frac{1}{|k-i|+1} = H_k + H_{n-k+1} - 2.$$

And use the formula $\sum_{i=1}^{n} H_i = (n+1) H_n - n$ we have

$$\begin{aligned}
S_2 &= \sum_{j<k} \sum_{i>k} \frac{1}{i-j+1} + \sum_{i>k} \sum_{k\le j\le i} \frac{1}{i-k+1} + \sum_{j>i} \sum_{i>k} \frac{1}{j-k+1} \\
&= n-k+\sum_{i>k} \left(H_i + H_{n-k+1} - 2H_{i-k+1}\right) \\
&= 2n-2k+4+(k-n-3) H_{n-k+1} + (n+1) H_n - (k+1) H_k
\end{aligned}$$

.

The formula above is not trivial, it's not hard to compute but require some patience. Using the same trick we have

$$\begin{aligned}
S_1 &= \sum_{j<i} \sum_{i<k} \frac{1}{k-j+1} + \sum_{i<k} \sum_{i\le j\le k} \frac{1}{k-i+1} + \sum_{j>k} \sum_{i<k} \frac{1}{j-i+1} \\
&= k-1+\sum_{i<k} \left(H_k + H_{n-i+1} - 2H_{k-i+1}\right) \\
&= 2k+2+(n+1) H_n - (k+3) H_k - (n-k+2) H_{n-k+1}
\end{aligned}$$

.

let $\kappa = \frac{k}{n}$ and sum them up:

$$\begin{aligned}
S_1 + S_2 - S_3 &= 2n+8+2(n+1) H_n - (2k+5) H_k - (2n-2k+6) H_{n-k+1} \\
&\approx 2n + O(1) + 2\left(nH_n - kH_k - (n-k) H_{n-k+1}\right) \\
&\approx 2n + O(1) + 2n \left(\log n - \frac{k}{n}\log k - \log(n-k+1) + \frac{k}{n}\log(n-k+1)\right) \\
&= 2n + O(1) + 2n\left(\log\left(\frac{1}{1-\kappa+\frac{1}{n}}\right) - \kappa\log\left(\frac{\kappa}{1-\kappa+\frac{1}{n}}\right)\right) \\
&= 2n + O(1) + 2\lambda n.
\end{aligned}$$

Now we only need to consider the size of λ, since $\kappa \in (0,1]$, and $\frac{1}{n} \to 0$ when n is large we have

$$\begin{aligned}
\lambda &= -(1-\kappa)\log(1-\kappa) - \kappa\log\kappa \\
&= -\mathcal{L}(\kappa, 1-\kappa).
\end{aligned}$$

The form of $\mathcal{L}$ is exactly the ML thing which is called **cross entropy loss**. And we can compute its range by **Jensen's Inequality**: consider $f(x) =$

$x \log x$, and $f''(x) = \frac{1}{x} > 0$ which means $f(x)$ is convex. Let $a+b = 1, a, b > 0$, by Jensen Inequality we have

$$f(a) + f(b) \geq 2f\left(\frac{a+b}{2}\right) = -\log 2.$$

plus

$$f(a) + f(b) < \lim_{x \to 1} (f(x) + f(1-x)) = 0.$$

This indicates that $\lambda \in (0, \log 2)$. Which reveals that if $\frac{k}{n} = \frac{1}{2}$, we have:

$$\mathbb{E}[C(\pi, k)] = n(2 + \log 2).$$

In a word, the running time is $O(n)$. Plus the **exact coeffient** is

$$C = 2 + 2\lambda = 2 - 2(1-\kappa)\log(1-\kappa) - 2\kappa \log \kappa.$$

which may varies from 2 to $2 + 2\log 2$, while k get closer to $\frac{n}{2}$, the constant will be closer to $2 + 2\log 2$. $\square$