Selection Problems

Medians and Order Statistics
Order Statistics

- The \( i^{th} \) order statistic of a set of \( n \) numbers is the \( i^{th} \) smallest element in sorted sequence:

  \[
  A = \begin{array}{cccccccc}
  4 & 1 & 3 & 2 & 16 & 9 & 10 & 14 & 8 & 7 \\
  \end{array}
  \]

- **Minimum** or first order statistic: 1
- **Maximum** or \( n^{th} \) order statistic: 16
- **Median** or \( (n/2)^{th} \) order statistic: 7 or 8
  (both are medians, happens when \( n \) is even!)

The Selection problem:

- **Input**: An array \( A \) of distinct numbers of size \( n \), and a number \( i \).
- **Output**: The element \( x \) in \( A \) that is larger than exactly \( i-1 \) other elements in \( A \).

- Finding **maximum** and **minimum** can be easily solved in linear time \( (O(n)) \).
  (it’s actually \( \Theta(n) \) ).
Trivial Solution:

- Sort the array $A$, and return the entry in $i^{th}$ position:
  - Sorting $A$ takes $O(n \log n)$.
  - The $i^{th}$ entry can be returned in constant time.
- Worst case running time: $O(n \log n)$
- Can we do better?
  - Comparing to maximum and minimum, the general $i$ is taking a long time.

A Randomized Selection Algorithm (idea):

- Think about the properties of $\text{Partition}(\ )$ algorithm:

  \[
  \begin{array}{c|c|c}
  \leq x & x & > x \\
  \hline
  p & q & r
  \end{array}
  \]

- If $i=q$, then we have $x$ as the $i^{th}$ order statistic.
  (what if this not the case?)
A Randomized Selection Algorithm (idea):

- If \( i < q \), then we have look for the \( i^{th} \) order statistic among first \( p-q+1 \) elements:

![Diagram of partition with elements \( p \leq x \leq x \leq x \leq r \)]

- We can call \textbf{Partition( )}, with parameters \((A, p, q)\)

---

A Randomized Selection Algorithm (idea):

- If \( i > q \), then we have look for \( i^{th} \) order statistic among elements between \( q \) and \( r \):

![Diagram of partition with elements \( p \leq x \leq x \geq x \geq r \)]

- We can call \textbf{Partition( )}, with parameters \((A, q, r)\)
The Algorithm:

Randomized-Select(A,p,r,i)
  if p=r then
    Return A[p]
  q=Randomized-Partition(A,p,r)
  k=q-p+1
  if i<= k then
    Randomized-Select(A,p,q,i)
  else
    Randomized-Select(A,q,r,i-k)

Running time:

- The recurrence:
  - Lucky: \[ T(n) = T(9n/10) + \Theta(n) = \Theta(n) \]
    Using master theorem:
    \[ n^{-\log_{10}1} = n^0 = 1 \]
  - Unlucky: \[ T(n) = T(n-1) + \Theta(n) = \Theta(n^2) \]
    Worst than sorting!
Average Case:

- Assume \textbf{Partition( )} Algorithm breaks $A$ to two pieces with sizes $k$ and $n-k-1$,

\[
T(n) = \frac{1}{n} \sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)
\]

\[
\leq \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + \Theta(n)
\]

- Assume $T(n) \leq cn$ for some $c$.

\[
T(n) = \frac{2}{n} \sum_{k=n/2}^{n-1} \frac{ck}{n} + \Theta(n)
\]

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

\[
= \frac{2c}{n} \left( \frac{n(n-1)}{2} - \frac{n^2}{4} \right) + \Theta(n)
\]

\[
= c(n-1) - \frac{c}{2} \left( \frac{n}{2} - 1 \right) + \Theta(n)
\]

\[
= cn - \frac{cn}{4} + c - \Theta(n)
\]

\[
\leq cn
\]
Worst-case Linear-Time O.S.

Select(A,p,q,i) Algorithm:
1. Divide A to n/5 groups of size 5.
2. Find the median of each group of 5 by brute force, and store them in a set A' of size n/5.
3. Use Select(A',1,n/5,n/10) to find the median x of n/5 medians.
4. Partition the n elements around x. Let k = q-p+1 (rank of x).
5. if i=k then
   return x
   if i<k then Select(A,p,q,i)
   else Select(A,q,r,i-k)

Analysis

- At least half of A' is less than x, which is at least n/10 elements of A'.
- Thus 3n/10 elements are smaller than x.
- If n>=50 then 3n/10>=n/4, so n/4 elements are smaller than x, and we know where they are!
- The components of recurrence for T(n):
  T(n/5): to find median of n/5 medians,
  T(3n/4): the complexity of step 5.
  Θ(n): The time for Partition().
  T(n)=T(n/5)+T(3n/4)+Θ(n)
Analysis (cont’d.)

- **Claim**: \( T(n) = cn. \)

\[
T(n) = cn / 5 + 3cn / 4 + \Theta(n) \\
\leq 19cn / 20 + O(n) \\
= cn - (cn / 20 - O(n)) \\
\leq cn, \text{ for large enough } c.
\]

Simplified Master Theorem:

- Assume that \( T(1) = d, \) and for \( n > 1: \)
  \[
  T(n) = aT(n / b) + cn.
  \]
- If \( a < b, \) Then \( T(n) = O(n); \)
- If \( a = b, \) Then \( T(n) = O(n \log n); \)
- If \( a > b, \) Then \( T(n) = O(n^{\log_b a}); \)

E.g. \( T(n) = 4T(n / 2) + cn \) gives \( T(n) = O(n^{\log_b 4}) = O(n^2) \)
Matrix Multiplication:

- Given two \( n \times n \) matrices \( A \) and \( B \), we are interested in computing the \( n \times n \) matrix \( C = AB \).
- Consider the case of two \( 2 \times 2 \) matrices: 
  \[
  A = \begin{pmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
  \end{pmatrix}
  \quad \text{and} \quad
  B = \begin{pmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
  \end{pmatrix}
  \]
  We have:
  \[
  C = AB = \begin{pmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
  \end{pmatrix}
  \]
  
  \[
  \begin{align*}
  c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\
  c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\
  c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\
  c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22}
  \end{align*}
  \]
  which need 8 multiplications and 4 additions.

Matrix Multiplication (cont’d.):

- For arbitrary \( n \times n \) matrices \( A \) and \( B \), the entry \( (i, j) \) of \( C = AB \), \( c_{i,j} \) can be computed as follows:
  \[
  c_{i,j} = (a_{i,1} \cdots a_{i,n}) \times (b_{1,j} \cdots b_{n,j})
  \]
  \[
  = a_{i,1}b_{1,j} + \cdots + a_{i,n}b_{n,j}
  \]
  \[
  = \sum_{k=1}^{n} a_{i,k}b_{k,j} \quad \text{(needs} \; O(n^3) \; \text{multiplications).}
  \]
Can we use fewer multiplications?

- Back to case \( n=2 \):
  \[
  \begin{bmatrix}
  a & b \\
  c & d
  \end{bmatrix} \times \begin{bmatrix}
  e & f \\
  g & h
  \end{bmatrix} = \begin{bmatrix}
  s_1 + s_2 - s_3 + s_6 & s_4 + s_5 \\
  s_6 + s_7 & s_2 - s_3 + s_5 - s_7
  \end{bmatrix}
  \]

\[
\begin{align*}
  s_1 &= (b - d)(g + h) \\
  s_2 &= (a + d)(e + h) \\
  s_3 &= (a - c)(e + f) \\
  s_4 &= h(a + b) & \text{, needs 7 multiplications and 18 additions.} \\
  s_5 &= a(f - h) \\
  s_6 &= d(g - e) \\
  s_7 &= e(c + d)
\end{align*}
\]

In General:

- Assume \( n \) is a power of 2, apply 2x2 algorithm recursively on a pair \( n \times n \) matrices by breaking each of them up to four square sub-matrices of size \((n/2) \times (n/2)\):

\[
\begin{bmatrix}
  A & B \\
  C & D
  \end{bmatrix} \times \begin{bmatrix}
  E & F \\
  G & H
  \end{bmatrix} = \begin{bmatrix}
  S_1 + S_2 - S_3 + S_6 & S_4 + S_5 \\
  S_6 + S_7 & S_2 - S_3 + S_5 - S_7
  \end{bmatrix}
  \]

\[
\begin{align*}
  S_1 &= (B - D)(G + H) \\
  S_2 &= (A + D)(E + H) \\
  S_3 &= (A - C)(E + F) \\
  S_4 &= H(A + B) \\
  S_5 &= A(F - H) \\
  S_6 &= D(G - E) \\
  S_7 &= E(C + D)
\end{align*}
\]
Recurrence:

- Everything is the same as 2x2 algorithm, except we are multiplying \((n/2) \times (n/2)\) matrices instead of scalars:
  \[
  T(n) = 7T\left(\frac{n}{2}\right) + dn^2
  \]
- With solution:
  \[
  T(n) = \left(1 + \frac{4}{3}d\right)n^{\log_2 7} + O(n^2)
  = O(n^{\log_2 7})
  = O(n^{2.81})
  \]

Today’s Lecture

- Binary Search Trees
- Balanced Search Trees
The Structure

- Each node $x$ in a binary search tree (BST) contains:
  - $key[x]$ - The value stored at $x$.
  - $left[x]$ - Pointer to left child of $x$.
  - $right[x]$ - Pointer to right child of $x$.
  - $p[x]$ - Pointer to parent of $x$.

BST - Property

- Keys in BST satisfy the following properties:
  - Let $x$ be a node in a BST:
  - If $y$ is in the left subtree of $x$ then:
    
    $key[y] \leq key[x]$
  
  - If $y$ is in the right subtree of $x$ then:
    
    $key[y] > key[x]$
Example:

- Two valid BST’s for the keys: 2,3,5,5,7,8.

```
       5
     /   \
   3     7
  / \
 2   5
```

```
  2
 /\ 
3   7
 /   \
5     8
```

In-Order Tree walk

- Can print keys in BST with in-order tree walk.
- Key of each node printed between keys in left and those in right subtrees.
- Prints elements in monotonically increasing order.
- Running time?
In-Order Traversal

Inorder-Tree-Walk(x)
1: If x!=NIL then
2: Inorder-Tree-Walk(left[x])
3: Print(key[x])
4: Inorder-Tree-Walk(right[x])

What is the recurrence for T(n)?
What is the running time?

In-Order Traversal

- In-Order traversal can be thought of as a projection of BST nodes on an interval.
- At most 2^d nodes at level d=0,1,2,...
Other Tree Walks

Preorder-Tree-Walk\( (x) \)
1: If \( x \neq NIL \) then
2: Print(\( key[x] \))
3: Preorder-Tree-Walk(\( left[x] \))
4: Preorder-Tree-Walk(\( right[x] \))

Postorder-Tree-Walk\( (x) \)
1: If \( x \neq NIL \) then
2: Postorder-Tree-Walk(\( left[x] \))
3: Postorder-Tree-Walk(\( right[x] \))
4: Print(\( key[x] \))

Searching in BST:

- To find element with key \( k \) in tree \( T \):
  - Compare \( k \) with \( key[root[T]] \)
  - If \( k < key[root[T]] \) search for \( k \) in \( left[root[T]] \)
  - Otherwise, search for \( k \) in \( right[root[T]] \)

Search\( (T,k) \)
1: \( x=\text{root}[T] \)
2: If \( x=\text{NIL} \) then return(“not found”)
3: If \( k=\text{key}[x] \) then return(“found the key”)
4: If \( k < \text{key}[x] \) then Search\( (left[x],k) \)
5: else Search\( (right[x],k) \)
Examples:

- Search($T, 11$)

- Search($T, 6$)

Analysis of Search

- Running time of height $h$ is $\square$
- After insertion of $n$ keys, worst case running time of search is $\square$
**BST Insertion**

- Basic idea: similar to search.
- **BST-Insert:**
  - Take an element \( z \) (whose right and left children are NIL) and insert it into \( T \).
  - Find a place where \( z \) belongs, using code similar to that of Search.
  - Add \( z \) there.

---

**Insert Key**

BST-Insert(\( T, z \))

1: \( y = \text{NIL} \)
2: \( x = \text{root}[T] \)
3: While \( x \neq \text{NIL} \) do
4: \( y = x; \)
5: if \( \text{key}[z] < \text{key}[x] \) then
6: \( x = \text{left}[x] \)
7: else \( x = \text{right}[x] \)
8: \( p[z] = y \)
9: if \( y = \text{NIL} \) the \( \text{root}[T] = z \)
10: else if \( \text{key}[z] < \text{key}[y] \) then \( \text{left}[y] = z \)
11: else \( \text{right}[y] = z \)
Locating the Minimum

BST-Minimum($T$)
1: $x = \text{root}[T]$
2: While $\text{left}[x]! = \text{NIL}$ do
3: \hspace{0.5cm} $x = \text{left}[x]$
4: return $x$

Application: Sorting

- Can use BST-Insert and Inorder-Tree-Walk to sort list of $n$ numbers

BST-Sort
1: $\text{root}[T] = \text{NIL}$
2: for $i = 1$ to $n$ do
3: \hspace{0.5cm} BST-Insert($T, A[i]$)
4: Inorder-Tree-Walk($T$)

Sort Input: 5, 10, 3, 5, 7, 5, 4, 8

Inorder Walk: 3, 4, 5, 5, 7, 8, 10
Analysis:

- The running time depends on the height of the tree (the Insert time).
- The average case analysis is like quick sort (which element will sit in the root).
- Therefore the expected running time is $O(n \log n)$.
- Average BST height is $O(\log n)$.

Successor

Given $x$, find node with smallest key greater than $key[x]$. Here are two cases depending on right subtree of $x$.

- Successor Case 1:
  - The right subtree of $x$ is not empty. Successor is leftmost node in right subtree. That is, we must return BST-Minimum(right[$x$])
Successor

- **Successor Case 2**: The right subtree of $x$ is empty. Successor is lowest ancestor of $x$. Observe that, “Successor” is defined as the element encountered by *inorder* traversal.

```
BST-Successor(x)
1: If right[x]! = NIL then
2:   return BST-Minimum(right[x])
3: y = p[x]
4: While (y != NIL) and (x = right[y])
5:   x = y
6:   y = p[y]
7: return y
```

**Running time?**

Deletion

- **Delete a node $x$ from tree $T$**:
  - Case 1: $x$ has no children.
Deletion:

- Case 2: \( x \) has one child (call it \( y \)). Make \( p[x] \) to replace \( y \) instead of \( x \) as its child, and make \( p[x] \) to be \( p[y] \).

```
```

Deletion:

- Case 3: \( x \) has two children:
  - Find its successor (or predecessor) \( y \).
  - Remove \( y \). (Note \( y \) has at most one child, why?)
  - Replace \( x \) by \( y \).

```

```
Delete Procedure

BSFT-Delete(\(T, z\))
1: If \((\text{left}[z] = \text{NIL}) \text{ or } (\text{right}[z] = \text{NIL})\) then
2: \(y = z\)
3: else \(y = \text{BST-Successor}(z)\)
4: If \(\text{left}[y] = \text{NIL}\) then
5: \(x = \text{left}[y]\)
6: else \(x = \text{right}[y]\)
7: If \(x \neq \text{NIL}\) then \(p[x] = p[y]\)
8: If \(p[y] = \text{NIL}\) then \(\text{root}[T] = x\)
9: else if \(y = \text{left}[p[y]]\) then \(\text{left}[p[y]] = x\)
10: else \(\text{right}[p[y]] = x\)
11: if \(y \neq z\) then \(\text{key}[z] = \text{key}[y]\)
12: return \(y\)