Analysis of Algorithms

Divide and Conquer Strategy

Hikmat Farhat

April 7, 2020

Binary Search

- The simplest example of divide-and-conquer strategy is probably binary search.
- Given a **sorted** array A of n elements and a value x, return **true** if x is an element of A.
- The key in this problem is that A is **sorted**.
- We follow a divide-and-conquer strategy by considering the "middle" element m of A, and considering the half of A to the left of m, L, and the other half to the right of m, R.
- Since A is sorted then all elements of L are less or equal than m and all elements of R are greater or equal than m.

Binary Search Code

```
bool binarySearch(int *A,int l,int r,int x){
    int m=(l+r)/2;

    if(x==A[m]) return true;
    if(x>A[m])
        return binarySearch(A,m+1,r,x);
    else
        return binarySearch(A,l,m-1,x);
}
```

- The complexity of the above code obeys the recurrence

$$T(n) = T(\frac{n}{2}) + \Theta(1)$$

- The solution of the recurrence (see Master theorem later) is $T(n) = \Theta(\log n)$.

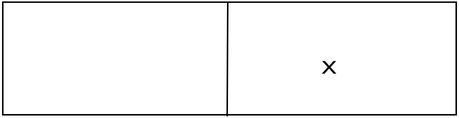
Counting Inversions

- Given an array A, a pair of elements $(A[i], A[j])$ is said to be an inversion iff $i < j$ and $A[i] > A[j]$.
- The simplest way to count the number of inversions in an array is using a double loop

```
count=0;
for(int  i=1;i<n;i++){
    for(int  j=0;j<i-1;j++){
       if(A[i]<A[j])count++;
    }
}
```

- Obviously the above algorithm is $\Theta(n^2)$.
- We will use divide-and-conquer to count the inversions in $\Theta(n \log n)$.

Modified Merge Sort



- If we divide an array in two then the total number of inversions is the sum of three parts
  1. Inversions in the left part
  2. Inversions in the right part
  3. Inversions of elements of the right part relative to elements on the left part
- The first two part are just recursive calls.
- The third part can be computed using the merge procedure of merge sort
- Note that in the above figure the inversions due to element x relative to elements in the left part are the same whether the parts are sorted or not.

- The basic idea for the counting algorithm is to modify the merge procedure of merge sort to allow us to count the inversion.
- Given two sorted arrays L and R, we merge them the same way as it was done using merge sort
- Let i and j be the indices of elements of L and R respectively where initially $i = j = 0$ we merge L and R into an array C indexed by k.
- If $L[i] < R[j]$ then $L[i]$ is copied to $C[k]$ and $i = i + 1$ and $k = k + 1$.
- If $R[j] < L[i]$ then $R[j]$ is copied to $C[k]$ and $j = j + 1$ and $k = k + 1$. Also in this case all the remaining elements of L are larger than $R[j]$ which means that the number of inversions is incremented by the number of elements remaining in L.

Example

- $L = \{\underset{\overset{\uparrow}{i}}{2}, 7, 12\}$ and $R = \{\underset{\overset{\uparrow}{j}}{4}, 8, 15\}$.
- First copy 2 to C to obtain $L = \{2, \underset{\overset{\uparrow}{i}}{7}, 12\}$ and $R = \{\underset{\overset{\uparrow}{j}}{4}, 8, 15\}$.
- Copy 4 to C and increment the number of inversions by 2 because there are two elements remaining in L, namely 7 and 12. to obtain
- $L = \{2, \underset{\overset{\uparrow}{i}}{7}, 12\}$ and $R = \{4, \underset{\overset{\uparrow}{j}}{8}, 15\}$.
- Copy 7 to C to obtain $L = \{2, 7, \underset{\overset{\uparrow}{i}}{12}\}$ and $R = \{4, \underset{\overset{\uparrow}{j}}{8}, 15\}$.
- Copy 8 to C and increment the number of inversions by 1 because there is one element remaining in L, namely 12.
- The total number of inversions is 3.

Master Theorem (special case)

- A generalization of the previous cases is done using a **simplified** version of the Master theorem

$$T(n) = aT(n/b) + \Theta(n^d)$$

$$
\begin{aligned}
T(n) &= aT(n/b) + cn^d \\
&= a\left[aT(n/b^2) + c(n/b)^d\right] + cn^d \\
&= a^2 T(n/b^2) + cn^d(a/b^d) + cn^d \\
&= a^2\left[aT(n/b^3) + c(n/b^2)^d\right] + cn^d(a/b^d) + cn^d \\
&= a^3 T(n/b^3) + cn^d(a/b^d)^2 + cn^d(a/b^d) + cn^d \\
&= a^i T(n/b^i) + cn^d \sum_{l=0}^{i-1}(a/b^d)^l
\end{aligned}
$$

The above reaches $T(1)$ when $b^k = n$ for some k. We get

$$T(n) = a^k T(1) + cn^d \sum_{l=0}^{k-1} (a/b^d)^l$$

There are three cases

1. $a = b^d$
2. $a < b^d$
3. $a > b^d$

case 1: $a = b^d$

If $a = b^d$ (i.e $\frac{a}{b^d} = 1$) then we get

$$T(n) = a^k T(1) + cn^d \cdot k$$

Since $k = \log_b n$ then

$$\begin{aligned} T(n) &= a^{\log_b n} T(1) + cn^d \log_b n \\ &= n^{\log_b a} T(1) + cn^d \log_b n \\ &= n^d T(1) + cn^d \log_b n \\ &= \Theta(n^d \log n) \end{aligned}$$

case 2: $a < b^d$

$$T(n) = a^k T(1) + cn^d \sum_{l=0}^{k-1} (a/b^d)^l$$

$$= a^k T(1) + cn^d \frac{(a/b^d)^k - 1}{(a/b^d) - 1}$$

for large n, i.e. $n \to \infty$ then $k = \log_b n \to \infty$ and since $a < b^d$ then $a/b^d \to 0$ Therefore

$$T(n) = n^{\log_b a} T(1) + cn^d$$

but $a < b^d \Rightarrow \log_b a < d$ and finally

$$T(n) = \Theta(n^d)$$

case 3: $a > b^d$

In this case we can write

$$
\begin{aligned}
T(n) &= a^k T(1) + cn^d \frac{(a/b^d)^k - 1}{(a/b^d) - 1} \\
&= n^{\log_b a} T(1) + gn^d (a/b^d)^k \\
&= n^{\log_b a} T(1) + gn^d (a/b^d)^{log_b n} \\
&= n^{\log_b a} T(1) + gn^d n^{log_b (a/b^d)} \\
&= n^{\log_b a} T(1) + gn^d n^{(-d + log_b a)} \\
&= \Theta(n^{\log_b a})
\end{aligned}
$$

Maximum Subarray Sum

- Given an array A of n elements we ask for the maximum value of

$$\sum_{k=i}^{j} A_k$$

- For example if A is -2,11,-4,13,-5,-2 then the answer is $20 = \sum_{k=2}^{4}$

Brute Force

- Compute the sum of all subarrays of an array A of size n and return the largest.
- A subarray starts at index i and ends at index j where $0 \leq i < n$ and $0 \leq j < n$.
- Therefore for **each possible** i and j compute the sum of $A[i] \ldots A[j]$.

```
int maxSubarray(int *A, int n){
  int sum=0, max=A[0];

  for(int i=0;i<n;i++){
      for(j=i;j<n;j++){
           sum=0;
          for(int k=i;k<=j;k++)
               sum+=A[k];
         if(max<sum)max=sum;
      }
  }
return max;
}
```

Complexity

- To determine the complexity of the brute force approach we can see that there are 3 nested loop therefore the complexity of the problem depends on how many times line 14 is executed
- The number of executions is

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1 = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} j - i + 1$$

- To evaluate the first sum let $m = j - i + 1$ then

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

- Finally, we get

$$\sum_{i=0}^{n-1}(n-i)(n-i+1)/2 = \frac{n^3+3n^2+2n}{6}$$
$$= \Theta(n^3)$$

Divide and Conquer

- general technique that divides a problem in 2 or more parts (divide) and patch the subproblems together (conquer).
- In this context if we divide an array in two subarrays. We have 3 possibilities:
  1. max is entirely in the first half
  2. max is entirely in the second half
  3. max spans both halves.
- Therefore the solution is max(left,right,both)

Both halves

- If the sum spans both halves it means it includes the last element of the first half and the first element of the second half
- This means that the we are looking for the sum of
  1. Max subsequence in first half that includes the last element
  2. Max subsequence in the second half that includes the first element

$$
\begin{aligned}
S_3 &= \max_{\substack{0\le i<n/2 \\ n/2\le j<n}} \sum_{k=i}^{j} A[k] \\
&= \max_{\substack{0\le i<n/2 \\ n/2\le j<n}} \left[\sum_{k=i}^{n/2-1} A[k] + \sum_{k=n/2}^{j} A[k] \right] \\
&= \max_{0\le i<n/2} \sum_{k=i}^{n/2-1} A[k] + \max_{n/2\le j<n} \sum_{k=n/2}^{j} A[k]
\end{aligned}
$$

Computing max that spans both halves

```
computeBoth (A,left,right)
```

$sum_1 \leftarrow sum_2 \leftarrow 0$
$center \leftarrow (left + right)/2$
for $i = center$ **to** $left$ **do**
    $sum_1 \leftarrow sum_1 + A[i]$
    if $sum_1 > max_1$ **then**
        $max_1 \leftarrow sum_1$
for $j = center + 1$ **to** $right$ **do**
    $sum_2 \leftarrow sum_2 + A[j]$
    if $sum_2 > max_2$ **then**
        $max_2 \leftarrow sum_2$
return $max_1 + max_2$

Recursive Algorithm

`maxSubarray`$(A, left, right)$

if $left = right$ **then**
  return $A[left]$
$center \leftarrow (left + right)/2$
$S_1 \leftarrow \texttt{maxSubarray}(A, left, center)$
$S_2 \leftarrow \texttt{maxSubarray}(A, center + 1, right)$
$S_3 \leftarrow computeBoth(A, left, right)$
return $\max(S_1, S_2, S_3)$

Complexity

- Given an array of size n the cost of the call to *maxSubarray* is divided into two computations
  1. The work of *computeBoth* which is $\Theta(n)$.
  2. **Two** recursive calls on the problem with **half the size**
  3. Therefore the total cost can be written as

$$T(n) = 2T(n/2) + \Theta(n)$$

- Using the Master theorem we get $T(n) = \Theta(n \log n)$

Medians

- The *median*,m of a sequence of n numbers is defined such that half of values (more precisely $\lfloor n/2 \rfloor$) of the sequence are bigger than m. For example for the sequence $48, 5, 10, 25, 42$ the median is 25.
- Obviously the median of n numbers can be computed by sorting the sequence in $\Theta(n \log n)$ steps then selecting the value at position $\lfloor n/2 \rfloor$
- Can we do better?
- It turns out that yes, by solving the general problem of selecting the k^{th} smallest element of an array of n elements.

Strategy

- We use a divide and conquer strategy as follows:
  1. Given an array A of n elements, select *randomly* a value m from A.
  2. Partition A into three arrays: L that contains all the elements smaller than m in no particular order, E all the elements that are equal to m and R an array containing all the elements bigger than m.
  3. Now we have three cases:
     1. if $k \leq\mid L \mid$ then the k^{th} element is in L and we call the algorithm recursively on the, smaller array, L
     2. if $\mid L \mid< k \leq\mid L \mid + \mid E \mid$ then the k^{th} element is in E and therefore it is equal to m.
     3. if $m >\mid L \mid + \mid E \mid$ then the k^{th} element is in R and we call the algorithm recursively on the, smaller array, R

$\texttt{Select}(A, \mathit{left}, \mathit{right}, k)$

if $\mathit{left} = \mathit{right}$ **then**
  return $A[\mathit{left}]$
$m \leftarrow \mathit{random}(\mathit{left}, \mathit{right})$
$\mathit{val} \leftarrow A[m]$
$\texttt{Partition}(A, L, E, R, m)$
if $k \leq\mid L \mid$ **then**
  return $\texttt{Select}(A, \mathit{left}, \mathit{left}+ \mid L \mid -1, k)$
else if $\mid L \mid < k \leq\mid E \mid + \mid L \mid$ **then**
  return val
else
  return $\texttt{Select}(A, \mathit{left}+ \mid E \mid + \mid L \mid, \mathit{right}, k- \mid E \mid - \mid L \mid)$

The partition algorithm

- The partition algorithm is a simple extension of the partition algorithm used for quicksort.
- In the select algorithm we had the arrays in the order L, E, R.
- for convenience and similar to the partition in quicksort the partitions will look like the figure below

Partitioning Algorithm

- Assuming that the pivot is already in place in $a[r]$.

PARTITION(a,p,r)

$i \leftarrow p - 1$
$k \leftarrow p$
$j \leftarrow r$
$pivot \leftarrow a[r]$
while $k < j$ **do**
    if $a[k] > pivot$ **then**
        $k \leftarrow k + 1$
    else if $a[k] < pivot$ **then**
        $i \leftarrow i + 1$
        swap($a[i], a[k]$)
        $k \leftarrow k + 1$
    else
        $j \leftarrow j - 1$
        swap($a[j], a[k]$)
return i, j

- The partition algorithm is used by the select algorithm as follows:
  - the array L is $a[p] \dots a[i]$.
  - the array E is $a[j] \dots a[r]$.
  - the array R is $a[i+1] \dots a[j-1]$.
  - In code for the select algorithm we assumed that the order of the subarrays is L followed by E followed by R.
  - Using the partitioning we modified select is

```
Select(A, left, right, k)
if left==right then
    return A[left]
m ← random(left, right)
val ← A[m]
Partition(A, i, j, m)
if k ≤ i − left + 1 then
    return Select(A, left, i, k)
else if k ≤ i − s + e − j + 2 then
    return val
else
    return Select(A, i + 1, j − 1, k − (right − left + i − j + 2))
```

- the complexity of the k selection problem depends on both the recursive part and the partition part.
- for an array of n items the partition part is clearly $\Theta(n)$.
- The recursive part depends on the pivot. Suppose the pivot is the i^{th} element then the subproblems are of size i (i.e. from 0 to $i-1$) and $n-i-1$ (i.e from $i+1$ and $n-1$)
- In the k selection problem, unlike quicksort, the recursion is called on only one subproblem.
- the worst case behavior occurs when the algorithm repeatedly selects the largest or the smallest element as the pivot.
- in this case the subproblem size is $n-1$ and the algorithm obeys the recurrence

$$T(n) = T(n-1) + \Theta(n)$$

- whose solution is $T(n) = \Theta(n^2)$

Average case complexity

- The average case complexity is much better than the worst case
- we start by assuming that any index can be equally likely selected as the pivot
- Since the algorithm selects only one subproblem we can bound the complexity by selecting the largest subproblem. Let X_i be a random variable

$$T(n) = T(max(i, n-i-1)) + \Theta(n)$$

- averaging over all possible values of i we get

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{i=0}^{n-1} T(max(i, n-i-1)) + \Theta(n) \\ &= \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i) + \Theta(n) \end{aligned}$$

we use the substitution method to prove that the average case complexity is $O(n)$. To show that $T(n) = O(n)$ we need to find $c > 0$ and n_0 such that $T(n) \leq cn$ for all $n \geq n_0$.

- Now assume that $T(k) \leq c \cdot k$ then the recurrence becomes

$$T(n) \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} c \cdot k + a \cdot n$$

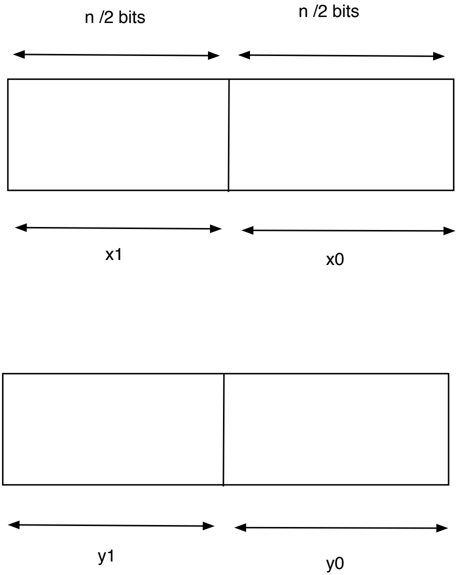
- Keeping mind that $\lfloor n/2 \rfloor \geq n/2 - 1$

$$
\begin{aligned}
T(n) &\leq \frac{2 \cdot c}{n} \left(\sum_{k=0}^{n-1} k - \sum_{k=0}^{\lfloor n/2 \rfloor - 1} \right) k + a \cdot n \\
&\leq \frac{2 \cdot c}{n} \left[n(n-1)/2 - \lfloor n/2 \rfloor (\lfloor n/2 \rfloor - 1)/2 \right] + a \cdot n \\
&\leq \frac{2 \cdot c}{n} \left[n(n-1)/2 - (n/2 - 1)(n/2 - 2)/2 \right] + a \cdot n \\
&\leq \frac{c}{n} \left[n^2 - n - n^2/4 + 3n/2 - 2 \right] + a \cdot n \\
&\leq \frac{c}{n} (3n^2/4 + n/2 - 2) + a \cdot n \\
&\leq c \cdot n - \left(\frac{c \cdot n}{4} - \frac{c}{2} - a \cdot n \right)
\end{aligned}
$$

- choose $c > 4a$ and $n_0 = \frac{2c}{c - 4a}$

Multiplying two numbers

- Given 2 n-bit numbers the "traditional" multiplication takes $\Theta(n^2)$ since there are n^2 2-bit multiplications and $\Theta(n)$ additions of $n-bit$ numbers (for a total of $\Theta(n^2)$.
- In this section we give a divide-and-conquer algorithm to compute the product of two n-bit numbers.
- The basic ideas is that an $n-bit$ x can be divided into the most significant $n/2$ bits and least significant $n/2$ bit. Two numbers x and y can be written as $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.


n /2 bits
n /2 bits
x1
x0
y1
y0

- Therefore $x \cdot y$ can be written as

$$
\begin{aligned}
&(x1 \cdot 2^{n/2} + x0) \cdot (y1 \cdot 2^{n/2} + y0) = \\
&x1 \cdot y1 \cdot 2^n + \\
&(x1 \cdot y0 + x0 \cdot y1) \cdot 2^{n/2} + x0 \cdot y0
\end{aligned}
$$

- We have reduced the multiplication of n-bit numbers to that of $n/2$-bit numbers and multiplication by 2^n and $2^{n/2}$.
- Multiplication by 2^n is equivalent with n-bit left shift and it can be done in $\Theta(n)$.
- Therefore the recurrence can be written as

$$
T(n) = 4T(n/2) + \Theta(n)
$$

- Using the master theorem : $a = 4, b = 2, d = 1$ The solution is $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$!!!

- We can get a better performance by noticing the following

$$(x1 + x0) \cdot (y1 + y0) = x1 \cdot y1 + x0 \cdot y0 + (x1 \cdot y0 + x0 \cdot y1)$$

- Rearranging terms we get

$$(x1 \cdot y0 + x0 \cdot y1) = (x1 + x0) \cdot (y1 + y0) - x1 \cdot y1 - x0 \cdot y0$$

- Since $x1 \cdot y1$ and $x0 \cdot y0$ are already computed then we need one extra multiplication instead of two. The recurrence becomes

$$T(n) = 3T(n/2) + \Theta(n)$$

- Thus $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58})$

Divide-and-Conquer algorithm

```
int multiply(int x,int y,int n){
   int x1=x>>n/2;
   int y1=y>>n/2;
   int mask=(1<<n/2)-1;
   int x0=x & mask;
   int y0=y &mask;
   int x1y1=multiply(x1,y1,n/2);
   int x0y0=multiply(x0,y0,n/2);
   int sum=x1y1+x0y0-multiply((x0+x1),(y0+y1),n/2);
   x1y1=x1y1<<n;
   sum=sum<<n/2;
   return x1y1+sum+x0y0;
}
```

Tower of Hanoi

- Let $move(n, start, end, aux)$ be the function that moves n bricks from peg **start** to peg **end** using peg **aux** as auxiliary.
- Suppose that we can move $n-1$ bricks from the **start** peg and put them in **aux** then all we have to do is move the remaining brick from **start** to **end** **then** transfer the $n-1$ from **aux** to **end**
- we can write

```
move(n,start,end,aux){
  if(n==1)cout<<"("<<start<<","<<end<<")"<<endl;
  else {
      move(n-1,start,aux,end);
      move(1,start,end,aux);
      move(n-1,aux,end,start);
  }
```

Complexity

- The solution to the Tower of Hanoi obeys the following recurrence relation

$$
\begin{aligned}
T(n) &= 2T(n-1) + \Theta(1) \\
&= 2T(n-1) + c \\
&= 2\left[2T(n-2) + c\right] + c \\
&= 2^2 T(n-2) + 2c + c \\
&= 2^2 \left[2T(n-3) + c\right] + 2c + c \\
&= 2^3 T(n-3) + 2^2 c + 2^1 c + 2^0 c \\
&\cdots\cdots \\
&= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i c \\
&= 2^k T(n-k) + (2^k - 1)c
\end{aligned}
$$

- The recursion stops when $k = n - 1$ and we get

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1)c = \Theta(2^n)$$